GRADO EN INGENIERÍA INFORMÁTICA

TRABAJO DE FINAL DE GRADO

Desarrollo de un prototipo para la generación de arquetipos en Java

---

Autor:

Constantin Alexandru ADIACONITEI

Supervisor:
Rafael GOTERRIS

Tutor académico:
Juan Pablo AIBAR AUSINA

Fecha de lectura: 15 de julio de 2020

Curso académico 2019/2020

Resumen

Este documento describe el desarrollo de un generador de APIs en *Java* desarrollado durante la estancia de prácticas curriculares en la empresa CloudAppi.

El objetivo de este proyecto es agilizar la creación de APIs *rest* en *Java* usando *Spring* de forma que la empresa pueda eliminar las tareas repetitivas cada vez que inicien un nuevo proyecto.

El generador consiste en un sistema que recibe una especificación de una API en formato *OpenAPI* la cual debe especificar el modelo de base de datos que se desea implementar.

El proyecto resultante de usar el generador está implementado en *Spring* y una vez conectado a la base de datos a partir de la cual se creó la especificación de *OpenAPI*, este ya es completamente funcional. Es decir, permite la realización de las operaciones CRUD (*Create, Read, Update y Delete*) definidas en la especificación. Además de servir de plantilla para seguir añadiendo funcionalidades a nuestra API.

Palabras clave

Java, API, Generador, Spring

Keywords

Java, API, Generator, Spring

Índice

Índice 3

Capítulo 1. Introducción 5
1.1. Descripción del proyecto 5
1.2. Estructura de la memoria 6

Capítulo 2. Planificación del proyecto 7
2.1. Metodología 7
2.2. Objetivos y alcance 8
2.3. Recursos 9
2.4. Identificación y gestión de riesgos 9
2.5. Planificación temporal del proyecto 10
2.6. Costes 11
2.7. Seguimiento del proyecto 11

Capítulo 3. Análisis 17
3.1. Arquitectura de los componentes de las APIs generadas 17
3.2. Análisis y definición de requisitos 20

Capítulo 4. Diseño 25
4.1. Diseño del proyecto 25
4.2. Extensiones de OpenAPI 26
4.3. Diseño de un caso de prueba en OpenAPI 28

Capítulo 5. Implementación 31
5.1. Tecnologías 31
5.2. Detalles de la implementación 33
5.2.1 Generadores 33
5.2.2 Estructura de los generadores (Patrón Builder) 35
5.2.3 Detección de campos relacionados en las entidades 35
5.2.4 Generación de validaciones en los parámetros de los endpoints 40

Capítulo 6. Pruebas 43
6.1. Object Mother 43
6.2. Pruebas unitarias 43
6.3. Pruebas unitarias sobre Java Poet 44
6.4. Ejemplo de prueba unitaria 44

Capítulo 7. Conclusiones 45

Bibliografía 47

Capítulo 1.

Introducción

En este documento se describen los detalles del desarrollo del proyecto informático elaborado en la empresa CloudAppi S.L..

1.1. Descripción del proyecto

CloudAppi es una empresa tecnológica situada en el Espaitec, el Parque Científico, Tecnológico y Empresarial de la Universitat Jaume I. También cuenta con oficinas en Madrid y Barcelona, y más recientemente en Perú. Su principal actividad es el desarrollo de sistemas informáticos, principalmente APIs (*Application Program Interface*), para empresas de diferentes sectores como las telecomunicaciones, entidades financieras y administraciones públicas, entre otras.

Como el principal producto de CloudAppi son las APIs, estas suponen una parte muy amplia del coste de desarrollo de sus sistemas. La gran mayoría de las APIs tienen una parte similar la cual se repite en cada desarrollo. Principalmente, las operaciones CRUD (*Create, Read, Update y Delete*) que se realizan mediante estas APIs. Es por esto, que para las APIs desarrolladas en *Java*, han decidido crear un generador que permita agilizar el proceso de creación de APIs, eliminando la parte repetitiva.

Partiendo de una especificación *OpenAPI* (estándar de especificación de APIs) [1] se desea generar una implementación en *Java* usando Spring la cual pueda ejecutar operaciones CRUD contra una base de datos.

1.2. Estructura de la memoria

La memoria contenida en este documento va a estar dividida en varios capítulos. En este capítulo se ha descrito el proyecto y su motivación. En el capítulo 2 se muestra la metodología utilizada así como la planificación y el seguimiento realizados. En el capítulo 3 se expondrá el análisis del sistema desarrollado. En el capítulo 4 se explicará el diseño seguido para el desarrollo de la aplicación. Y por último, en los capítulos 5 y 6 se detallarán la implementación y las pruebas, respectivamente.

Capítulo 2.

Planificación del proyecto

2.1. Metodología

La metodología utilizada durante el desarrollo de este proyecto ha sido *SCRUM*. Se ha elegido esta metodología principalmente por la forma de trabajar de la empresa, pero además es la que mejor se adapta a un proyecto de I+D como el desarrollado. Esto es debido a que la mayoría de los requisitos implementados se han ido decidiendo a medida que el proyecto avanzaba.

SCRUM sigue un modelo de desarrollo basado en *Sprints*, periodos de tiempo en los cuales se trabaja en la implementación de ciertos requisitos y al final del cual se ha obtenido una nueva funcionalidad en el desarrollo global del proyecto.

Para la coordinación de cada *sprint* se ha utilizado el sistema Kanban, proceso basado en la distribución de pequeñas tareas en diferentes bloques según su estado (Pendiente, En proceso, Finalizada).

Al finalizar cada *sprint* se ha revisado el avance conseguido y se han definido los requisitos y tareas del siguiente *sprint*.

Esta metodología ha permitido estructurar el desarrollo de un proyecto el cual era bastante difuso en un principio.

2.2. Objetivos y alcance

El objeto principal de este proyecto es desarrollar un sistema que permita generar código *Java* funcional a partir de una especificación de una API en formato *OpenAPI*. Con este proyecto se pretende agilizar los futuros desarrollos de la empresa. Además se requiere definir una API en formato *OpenAPI* de forma que se pueda probar el generador con un caso real.

El objetivo principal del proyecto se puede desglosar en los siguientes subobjetivos:

- Generar un proyecto *Java* basado en *Maven* y *Spring*.
- Generar el mapeo de entidades de negocio.
- Generar la capa de controladores.
- Generar la capa de repositorios y servicios.
- Generar el mapeo de los datos de entrada y salida de la API generada.

Por otro lado, los objetivos formativos de la estancia en prácticas se pueden desglosar en los siguientes:

- Estudiar el funcionamiento de las APIs *rest* desarrolladas con *Spring*.
- Mejorar las habilidades a la hora de realizar pruebas.
- Aprender a definir una API mediante la especificación *OpenAPI*.
- Conocer las herramientas y tecnologías que se usan en la empresa.
- Aprender las metodologías de trabajo de la empresa.

El **alcance funcional** de este sistema incluye generar los elementos necesarios para obtener una API funcional (entidades, repositorios y controladores) a partir de la especificación, pero no incluye la conexión con una base de datos real. Sin embargo, durante el desarrollo del proyecto se usará una base de datos de prueba previamente generada.

En cuanto al **alcance organizativo**, el desarrollo de este proyecto no involucra ningún otro departamento de la empresa. Pero una vez finalizado el proyecto, este se puede usar de manera interna en la empresa u ofrecer como producto a otras empresas.

2.3. Recursos

2.3.1. Recursos humanos

Se estima que se necesita 1 persona y alrededor de 300 horas para conseguir un prototipo funcional del proyecto descrito.

2.3.2. Recursos de Hardware y Software

En la **Tabla 2.1** podemos ver los recursos de hardware y software los cuales serán detallados más adelante en las **Tablas 5.1, 5.2 y 5.3.**

Hardware	Software	
1 Ordenador	*IDE - IntelliJ*	*Java*
	GitLab	*Java Poet* [2]
	Spring [3]	*Hibernate* [4]
	OpenAPI	*Redmine*
	MapStruct [5]	*OpenApiV3Parser* [6]
	JUnit5	

Tabla 2.1. Recursos de Hardware y Software

2.4. Identificación y gestión de riesgos

El proyecto desarrollado tiene una parte muy amplia de I+D ya que se pretende desarrollar un sistema poco frecuente utilizando una librería casi desconocida (*Java Poet*). A continuación se detallan los riesgos identificados para el desarrollo de este proyecto:

- Desconocimiento por parte del estudiante de las tecnologías a utilizar y dificultad por parte del mismo para entenderlas.
- Impedimentos técnicos a la hora de implementar ciertas funcionalidades complejas relacionadas con la parte de generación de código.

2.5. Planificación temporal del proyecto

Como ya se ha dicho, este proyecto ha sido definido a medida que ha ido avanzando su desarrollo por lo que la planificación inicial del proyecto solo incluía apartados superficiales de su implementación. Además, destacar que la duración de los sprints ha sido de 2 semanas. La **Tabla 2.2** representa la planificación inicial del proyecto:

Referencia	Tarea	Estimación (Horas)
Apigen-1	Generador de Entidades	25
Apigen-2	Generador de Repositorios	25
Apigen-3	Generador de Servicios	25
Apigen-4	Generador de Controladores	50
Apigen-5	Generador de DTO	25
Apigen-6	Generador de Mapeadores	25
Apigen-7	Generador de Validaciones	30
Apigen-8	Crear especificación de la API de prueba	25
Apigen-9	Pruebas Unitarias	50
Total		**280 Horas**

Tabla 2.2. Planificación Inicial

En la **Tabla 2.2** también se puede observar que el total de horas planificadas es de 280. Las 20 restantes para completar las 300 horas de prácticas corresponden a los primeros días de la estancia, los cuales fueron un periodo de adaptación a la empresa .

2.6. Costes

2.6.1. Costes humanos

Para calcular el coste humano, se toma como dato que el sueldo medio de un programador junior es de 1.500€/mes [7], por lo que el sueldo de trabajar durante 3 meses es de 4.500€. A este coste le debemos añadir un 20% de costes indirectos lo cual da un total de 5.400€.

2.6.2. Costes tecnológicos

Tecnología	Coste/€	Comentario
1 Ordenador	100€	La empresa ya cuenta con el equipo así que se tiene en cuenta la siguiente fórmula: **800€ Precio equipo * (3 Meses de uso / 24 Meses de vida útil)**
GitLab	0€	Estas tecnologías son gratuitas
GSuite	25€/usuario/mes	Coste por usuario para utilizar la nube de Google
Redmine	0€	Redmine es gratuito
Servidor para alojar *Redmine*	15€/mes	Servidor donde se aloja *Redmine*, el sistema de gestión de proyectos utilizado

Tabla 2.3. Costes de las tecnologías utilizadas

Teniendo en cuenta la **Tabla 2.3**, el coste final del desarrollo del proyecto es de 220€ en cuanto al gasto en tecnologías.

2.7. Seguimiento del proyecto

A continuación se va a detallar el seguimiento y control del proyecto a lo largo de los diferentes *sprints*. Pero primero un resumen de lo realizado durante la primera quincena de la estancia en prácticas.

Primera quincena de la estancia en prácticas

La primera quincena se dedicó principalmente al estudio del proyecto propuesto para poder realizar la propuesta técnica del mismo. También se estudiaron las tecnologías a utilizar además de prepararse el espacio de trabajo y herramientas necesarias.

Además también se estudió un proyecto anterior de la empresa el cual se ha usado como base para el descrito en el documento actual.

Sprint 1

En el primer *sprint* empieza el desarrollo del proyecto. Como base se toma el código fuente de un proyecto anterior desarrollado por la empresa el cual realiza algunas de las funciones básicas que deseamos, como por ejemplo leer los datos necesarios de las especificaciones *OpenAPI*. En este *sprint* se seleccionó la tarea de la **Tabla 2.4**:

Referencia	Tarea
Apigen-1	Generador de Entidades **Subtareas:** ● Analizar los tipos de atributos que se desean generar ● Analizar la herramienta de extracción de datos de *OpenAPI*. ● Extraer los datos necesarios de la especificación *OpenAPI*. ● Analizar los tipos de relaciones JPA existentes sobre entidades. ● Implementar la detección automática del tipo de relación de cada atributo. ● Generar el código *Java* de las entidades a partir de los datos analizados. ● Pruebas unitarias.

Tabla 2.4. Tareas del *Sprint* 1

La tarea seleccionada para este *sprint*, fue dividida en diferentes subtareas las cuales se completaron con éxito. Sin embargo, se dedicó más tiempo del previsto a la detección del tipo de relaciones ya que existían varios conflictos. Además el desarrollo se acompañó de tests unitarios para verificar el correcto funcionamiento del código implementado.

Cabe destacar que los últimos días de este *sprint* se empezó a realizar la estancia de prácticas en modalidad de teletrabajo debido a la pandemia del Covid-19.

Sprint 2

En el segundo *sprint*, también se toma como base parte del código del citado anterior proyecto de la empresa. En este caso, el código fue modificado en su totalidad ya que no se adaptaba a los estándares actuales requeridos por la empresa. En este *sprint* se seleccionaron las tareas de la **Tabla 2.5**:

Referencia	Tarea
Apigen-2	Generador de Repositorios
Apigen-3	Generador de Servicios

Tabla 2.5. Tareas del *Sprint* 2

Este *sprint* finaliza con ambas tareas completadas. Además ambas recibieron sus correspondientes tests unitarios para verificar su correcta implementación.

Sprint 3

En el tercer *sprint* se empieza con el desarrollo de la capa de controladores y los respectivos recursos que estos necesitan: DTOs y Mapeadores. En la **Tabla 2.6** se muestran las tareas seleccionadas para este *sprint*.

Referencia	Tarea
Apigen-4	Generador de Controladores
Apigen-5	Generador de DTOs
Apigen-6	Generador de Mapeadores

Tabla 2.6. Tareas del *Sprint* 3

Este *sprint* finaliza con ambas tareas completadas. Además ambas recibieron sus correspondientes tests unitarios para verificar su correcta implementación.

Sprint 4

En el cuarto sprint, como ya se tenía una versión bastante avanzada del generador, se decide tomar la tarea relacionada con la creación de una especificación de una API de prueba en *OpenAPI*. Esta tarea se desglosa en varias subtareas como se puede ver en la **Tabla 2.7**:

Referencia	Tarea
Apigen-8	Crear especificación de la API de prueba **Subtareas:** ● Estudio de la documentación de *OpenAPI* ● Creación de un caso de prueba ● Implementación de la especificación del caso de prueba

Tabla 2.7. Tareas del *Sprint* 4

Esta tarea se completó satisfactoriamente, pero debido al desconocimiento por mi parte del estándar de *OpenAPI*, se tuvieron que realizar varios ajustes a la especificación. Gracias a los compañeros de la empresa que ayudaron a identificar los errores, esto se pudo solucionar sin problemas.

Sprint 5

En el quinto y último *sprint* se toma la tarea relacionada con la generación de validaciones en los atributos de las entidades. Durante el desarrollo de esta tarea se descubrieron nuevas necesidades en el generador. Específicamente la necesidad de generar validadores sobre los parámetros que reciben los controladores. Esto se añade como nueva tarea. En la **Tabla 2.8** se pueden ver las tareas realizadas durante este sprint:

Referencia	Tarea
Apigen-7	Generador de Validaciones
Apigen-10	Generador validaciones sobre parámetros de los controladores

Tabla 2.8. Tareas del *Sprint* 5

Cabe destacar que la tarea **Apigen-9** asociada al desarrollo de pruebas, se ha ido desglosando en cada sprint de tal forma que para cada tarea se han realizado sus respectivas pruebas unitarias.

Capítulo 3.

Análisis

Para poder comprender el funcionamiento del sistema desarrollado partimos de un sencillo diagrama (**Imagen 3.1**) que nos ayudará a reconocer las partes implicadas y su función.

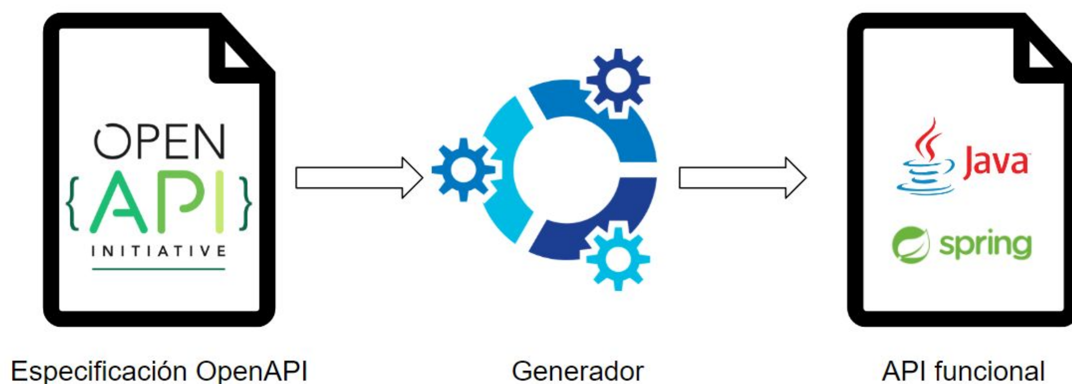

Imagen 3.1. Diagrama del sistema a alto nivel

El funcionamiento de sistema es sencillo, el generador consume un documento YAML (*YAML Ain't Markup Language*) con la especificación de la API siguiendo el estándar de *OpenAPI* y devuelve una API funcional desarrollada en *Java* y *Spring*.

Para poder generar el código de *Java*, se necesitaba cierta información la cual no se puede obtener directamente de la especificación de *OpenAPI*, pero esta permite definir

extensiones propias a la misma [8]. Es por ello que se definieron ciertas extensiones las cuales permiten al generador obtener toda la información que necesita.

3.1. Arquitectura de los componentes de las APIs generadas

Las APIs que se obtienen a partir del generador siguen una estructura muy específica que sigue los estándares de la empresa. A continuación se van a explicar para poder entender más en detalle los nombres y recursos que se nombran.

3.1.1. Entidades

Las entidades en JPA (Java Persistence API) son simplemente POJOs (*Plain Old Java Object*), es decir, objetos que representan información que puede ser guardada en una base de datos [9]. Una entidad representa una tabla guardada en una base de datos y cada instancia de una determinada entidad, representa a su vez una fila en dicha tabla.

Las entidades se representan con un conjunto de anotaciones de JPA: @*Entity*, para declararla como entidad, y @*Table* para declarar el nombre de la tabla de la base de datos donde se guarda la información.

Las entidades se definen en la especificación de *OpenAPI* de entrada del generador.

3.1.2. Repositorios

Los repositorios, son los encargados de la persistencia de las entidades en la base de datos [10].

Los repositorios normalmente utilizan una interfaz de JPA llamada *JPARepository*, la cual brinda las operaciones básicas de guardado y obtención de datos. Pero en este proyecto se utiliza una variante de dicha interfaz llamada *ApigenRepository* la cual implementa ciertas operaciones definidas en los estándares de la empresa. En este caso incluye parámetros configurables de búsqueda como campos a excluir, campos por los que ordenar, filtros de búsqueda, etc.

Se genera un repositorio por cada entidad definida.

3.1.3. Servicios

Los servicios son los encargados de la lógica de negocio y hacen uso de los repositorios para ejecutar las operaciones.

Los servicios generados implementan las operaciones CRUD (*Create, Read, Update, Delete*).

Se genera un servicio por cada entidad definida, todas ellas extendiendo una clase abstracta la cual implementa las operaciones CRUD nombradas.

3.1.4. Recursos/DTO

Los recursos o DTO (*Data Transfer Object*) representan aquellos objetos que se utilizan como entrada o salida de una API *rest*. Normalmente, estos recursos se reciben o envían a una API en formato JSON (*JavaScript Object Notation*).

Se genera un recurso por cada entidad definida.

3.1.5. Mapeadores

Los mapeadores son los encargados de convertir los recursos a objetos y viceversa. Es decir para poder guardar objetos de *Java* en una base de datos necesitamos convertir su información a un tipo que sea legible por la base de datos. El mismo proceso aplica a la inversa.

En este proyecto se ha usado la librería *MapStruct*, la cual permite definir los mapeadores mediante anotaciones JPA.

Se genera un mapeador por cada entidad definida.

3.1.6. Controladores

Los controladores son los encargados de ofrecer la funcionalidad de la API mediante endpoints HTTP (*Hypertext Transfer Protocol*).

Se genera un controlador por cada entidad definida.

Los endpoints de cada controlador se definen en la especificación de *OpenAPI* de entrada del generador.

3.2. Análisis y definición de requisitos

Como este proyecto no involucra ninguna interacción por parte de los usuarios más allá de pasar el fichero necesario para generar una API, no se han definido historias de usuario. En su lugar se han establecido tareas para las cuales se detallan a continuación sus criterios de aceptación y el DoD (*Definition of Done*).

Apigen-1 Generador de Entidades

Criterios de aceptación

El proyecto generado deberá contar con las respectivas entidades de cada objeto definido en la especificación.

Definition of Done

- Las entidades generadas contarán con las anotaciones JPA correspondientes.
- Las entidades generadas contarán con los atributos establecidos y su tipo será válido.
- Los atributos relacionados de las entidades generadas contarán con las anotaciones de relaciones de JPA correspondientes.

Apigen-2 Generador de Repositorios

Criterios de aceptación

El proyecto generado deberá contar con las repositorios para cada entidad que se define en la especificación.

Definition of Done

- El repositorio generado será una interfaz de Java.
- El repositorio generado contará con las anotaciones JPA correspondientes.
- El repositorio generado extenderá la clase de repositorio definida por la empresa.

Apigen-3 Generador de Servicios

Criterios de aceptación

El proyecto generado deberá contar con las repositorios para cada entidad que se define en la especificación.

Definition of Done

- El repositorio generado contará con las anotaciones JPA correspondientes.
- El repositorio generado extenderá la clase de repositorio definida por la empresa.
- El repositorio generado contará con el constructor que referencia al repositorio correspondiente.

Apigen-4 Generador de Controladores

Criterios de aceptación

El proyecto generado deberá contar con los controladores asociados a cada entidad definida en la especificación así como con todos los recursos necesarios (DTO, Mapeador y Respuesta).

Definition of Done

- El controlador generado contará con las anotaciones JPA correspondientes.
- El controlador generado dispondrá de los servicios necesarios y además estarán correctamente instanciados en el mismo.
- El controlador generado contará con todos los endpoints necesarios para realizar operaciones CRUD mediante peticiones HTTP (*POST, GET, PUT, DELETE*).

Apigen-5 Generador de DTO

Criterios de aceptación

El proyecto generado deberá contar con los DTO necesarios para cada entidad definida en la especificación.

Definition of Done

- Los DTO generados contarán con todos los campos de las entidades a las que representan.
- Los campos de los DTO generados contarán con las anotaciones JPA correspondientes.

Apigen-6 Generador de Mapeadores

Criterios de aceptación

El proyecto generado deberá contar con los mapeadores necesarios para cada entidad definida en la especificación.

Definition of Done

- Los DTO generados contarán con todos los campos de las entidades a las que representan.
- Los campos de los DTO generados contarán con las anotaciones JPA correspondientes.

Apigen-7 Generador de Validaciones

Criterios de aceptación

Los campos de las entidades generadas y los atributos de los *endpoints* de los controladores deben contener las validaciones JPA definidas en la especificación.

Definition of Done

- Los campos de las entidades generadas contarán con las anotaciones JPA de validación establecidas en la especificación.
- Los atributos de los *endpoints* de cada controlador deberán contender las anotaciones de validación de acuerdo a lo definido en la especificación.

Apigen-8 Crear especificación de la API de prueba

Criterios de aceptación

La especificación *OpenAPI* de prueba deberá seguir los estándares de *OpenAPI* y también los estándares de la empresa en relación a la definición de APIs.

Definition of Done

- Los controladores de la API tendrán un *endpoint* por cada tipo de operación (*POST, GET, PUT, DELETE*).
- Las entidades contendrán campos simples y campos relacionados.
- Las entidades contendrán campos con validaciones.
- Las respuestas de los endpoints seguirán el estándar de la empresa.

Capítulo 4.

Diseño

4.1. Diseño del proyecto

Para el desarrollo del sistema se ha utilizado un proyecto *Maven* multimodular [11]. Cada módulo encapsula diferentes componentes del generador de tal forma que queden organizados. En la **Tabla 4.1** se exponen los diferentes módulos:

Módulo	Función
archetype-core	Este módulo contiene todos aquellos recursos definidos por la empresa y los cuales el generador necesita para generar la API con los estándares de la empresa.
generator-core	Este módulo contiene todos los componentes necesarios para la generación de la API: la información extraída de la especificación *OpenAPI* encapsulada en objetos *Java* y los generadores de cada fragmento de la API generada.
generator-rest	Este módulo simplemente alberga la capa que media entre las peticiones del usuario que quiere usar el generador y el generador en sí. Es decir, permite la entrada del fichero YAML con la especificación de la API, y devuelve un fichero “.zip” con el proyecto de la API generada.

Tabla 4.1. Módulos Maven del proyecto

4.2. Extensiones de *OpenAPI*

OpenAPI, como ya hemos dicho previamente, es un estándar de especificación de APIs *rest*. Permite definir un fichero YAML o JSON con una estructura específica en el cual se pueden definir los *endpoints* de una API, las respuestas de los *endpoints*, errores, etc.

En el caso de este proyecto, para generar una API completa, se necesita cierta información extra la cual el estándar de *OpenAPI* no reconoce. Por ejemplo, para definir las entidades como objetos de JPA (*Java Persistence API*) se requiere definir en cada campo, su tipo (el cual puede no ser un tipo básico de *Java*, sino por ejemplo un objeto “Dirección”), las columnas de la base de datos a la que corresponde, etc.

Para solventar este problema, *OpenAPI* permite la creación de extensiones en las cuales podemos definir la información que deseemos. Las extensiones en *OpenAPI* se marcan con una “x” en su nombre. En la **Tabla 4.2** se exponen las extensiones definidas:

Extensión	Propósito
x-apigen-models	Permite definir los modelos de todas las entidades que vamos a generar. Aquí se permiten establecer los nombres de los campos, su tipo, su relación, su columna correspondiente en la base de datos, etc.
x-apigen-binding	Extensión que se usa en la definición de cada endpoint para establecer la entidad sobre la cual opera dicho endpoint.
x-apigen-mapping	Extensión que se usa en la definición de *schema* de *OpenAPI* para establecer la entidad a la que representa.
x-apigen-project	Extensión que alberga la información básica necesaria para generar un proyecto *Java*: nombre, descripción, versión, etc.

Tabla 4.2. Extensiones propias de la especificación *OpenAPI*

En la **Imagen 4.1** podemos observar un ejemplo de una entidad definida dentro de la extensión *x-apigen-models*:

```yaml
x-apigen-models:
  Medico:
    read-only: false
    logical-unit: medico
    relational-persistence:
      table: medicos
    attributes:
      - name: id
        type: Long
        relational-persistence:
          column: medico_id
          primary-key: true
        validations:
          - type: NotNull
      - name: dni
        type: String
        relational-persistence:
          column: medico_dni
          primary-key: true
        validations:
          - type: NotEmpty
      - name: nombre
        type: String
        relational-persistence:
          column: nombre
      - name: salario
        type: Double
        relational-persistence:
          column: salario
        validations:
          - type: Positive
      - name: pacientes #@ManyToMany-Owner
        type: Array
        items-type: Paciente
        relational-persistence:
          column: medico_id
          foreign-column: paciente_id
          intermediate-table: consultas
          owner: true
```

Imagen 4.1. Entidad definida en la extensión de *OpenAPI x-apigen-models*

4.3. Diseño de un caso de prueba en *OpenAPI*

Uno de los objetivos de la estancia en prácticas por parte de la empresa era que el estudiante aprendiera a definir especificaciones de APIs usando *OpenAPI*. Además, para probar el generador se necesitaba de una especificación que contemplase todos los posibles casos. Es por eso que a continuación se detalla el caso de prueba diseñado y utilizado con el generador.

El caso de prueba representa un pequeño hospital. Es un caso de prueba en el que se intentan definir la mayor cantidad posible de casos. En la **Imagen 4.2** se pueden ver las entidades en *Java* que se espera que el generador devuelva a partir de la especificación de *OpenAPI* definida.

Médico	
@NotNull Long	id
@NotNull String	dni
String	nombre
@Positive double	salario
@ManyToMany-Owner Set<Paciente>	pacientes

Paciente	
@NotNull Long	id
@NotNull String	dni
String	nombre
@Pattern(regexp="\\d{9}") String	telefono
@Email String	email
@OneToOne-Owner Dirección	direccion
@OneToOne-Owner Cama	cama
@OneToMany Set<Facturas>	facturas
@ManyToMany @Size (min=1) Set<Medico>	medicos

Cama	
@NotNull String	id
@ManyToOne Habitacion	habitacion
@OneToOne Paciente	paciente

Direccion	
@NotNull Long	id
String	Calle
int	numero
String	cp
String	localidad
@OneToOne Paciente	paciente

Factura	
@NotNull Long	id
@ManyToOne Paciente	paciente
@DecimalMin(value=100.0) double	importe
@PastOrPresent LocalDate	fecha
@PastOrPresent LocalDateTime	fechaYHora

Habitacion	
@NotNull String	id
int	planta
@Min(value=1) @Max(value=2) int	numero
@OneToMany Set<Cama>	camas

Imagen 4.2. Estructura deseada en *Java* del caso de prueba

La definición de una especificación de *OpenAPI* es bastante extensa como para mostrarla aquí, por eso me limitaré a decir que se definió de acuerdo a los estándares de *OpenAPI* así como los propios de la empresa. Además se definieron ciertas entidades las cuales se acompañaron de sus respectivos controladores con los *endpoints* necesarios para realizar operaciones CRUD. La **Imagen 4.1**, representa una de las entidades definidas en la especificación de prueba.

Además, de acuerdo a lo definido en el apartado 3.1, en la **Imagen 4.3** se puede observar la estructura de ficheros obtenida en la API generada por el generador, de acuerdo a nuestro caso de prueba.

Imagen 4.3. Estructura de ficheros de la API generada

Capítulo 5.

Implementación

Cabe destacar, como ya se ha dicho previamente, que el proyecto descrito en este documento parte de un proyecto ya existente, por lo que se ha reutilizado parte del código. Sin embargo, las implementaciones descritas en este capítulo han modificado en casi su totalidad el código heredado, debido a que era muy poco mantenible. Además el proyecto del cual parte el actual, no tiene el mismo objetivo, si bien es cierto que ambos generan APIs, el antiguo lo hace a partir de una base de datos y no se puede utilizar sin una. Mientras que el proyecto descrito en este documento no necesita de nada más que una especificación de *OpenAPI*.

5.1. Tecnologías

En este apartado se detallarán las tecnologías (**Tabla 5.1**), herramientas (**Tabla 5.2**) y librerías utilizadas (**Tabla 5.3**).

5.1.1. Tecnologías

Tecnología	Descripción
Java	Lenguaje de programación que se ha utilizado para el desarrollo del proyecto.
Git / GitLab	Sistema de control de versión utilizado mediante la plataforma de *GitLab*.
Spring	Es un *framework* de inversión de control e inyección de dependencias para *Java*. Es la base de la API generada por el generador.
Hibernate	Es una herramienta de mapeo objeto-relacional para *Java*. Principalmente se utiliza para la relación de las entidades generadas y la base de datos, así como las anotaciones de validaciones que permite.
OpenAPI	Estándar de especificación de APIs. Se utiliza para definir las especificaciones que consume nuestro generador.
JUnit5	*Framework* de test para *Java*. Se ha utilizado para realizar las pruebas unitarias del generador.

Tabla 5.1. Tecnologías utilizadas en el desarrollo del proyecto

5.1.2. Herramientas

Herramienta	Descripción
IntelliJ	IDE utilizado durante el desarrollo del proyecto.
Swagger Editor	Editor de texto desarrollado para desarrollar especificaciones de *OpenAPI*. Se ha utilizado para validar la sintaxis del caso de prueba definido.
Visual Studio Code	Editor de texto muy ligero y rápido. Se ha utilizado para gestionar y comprobar los ficheros de las APIs generadas durante el desarrollo.
Redmine	Herramienta de gestión de proyectos utilizada en la empresa durante el desarrollo del proyecto.

Tabla 5.2. Herramientas utilizadas en el desarrollo del proyecto

5.1.3. Librerías

Librería	Descripción
Java Poet	API para la generación de código fuente *Java*. Se ha utilizado para crear todo los ficheros y código fuente necesario para generar APIs.
OpenApiV3Parser	Librería que permite extraer la información a partir de fichero YAML o JSON de *OpenAPI* y convertirla a objetos *Java*.
MapStruct	Librería utilizada para generar la implementación de mapeadores de las entidades.

Tabla 5.3. Librerías utilizadas en el desarrollo del proyecto

5.2. Detalles de la implementación

En este apartado se van a detallar algunos detalles de la implementación los cuales son los puntos clave del proyecto.

5.2.1 Generadores

El objetivo del proyecto es generar una API completa. Para conseguir esto se deben generar varios componentes los cuales tienen funciones muy específicas. Por eso se utilizan pequeños generadores los cuales se encargan de generar cada parte por separado.

En la **Tabla 5.4** se detalla una lista de todos los diferentes generadores que contiene el proyecto.

Generador	Función
AbstractGenerator	Clase abstracta que contiene todas aquellas variables que son comunes entre generadores, además de métodos comunes.
ApplicationGenerator	Genera el fichero *.java* que contiene la inicialización de la aplicación *Spring* que contiene la API generada.
PomGenerator	Genera el fichero *pom.xml* del proyecto generado.
ProjectStructureGenerator	Genera la estructura de ficheros del proyecto generado.
PropertiesGenerator	Genera los ficheros *.properties* utilizados en las aplicaciones *Spring*.
SpringBootBaseGenerator	Genera los ficheros base de la aplicación *Spring* generada.
SpringBootContextTest Generator	Genera un text de carga de contexto en la aplicación *Spring* generada.
EntitiesGenerator	Genera las entidades definidas en la especificación.
RepositoryGenerator	Genera los repositorios para las entidades definidas.
ServiceGenerator	Genera los servicios para las entidades definidas.
ValidationGenerator	Genera las anotaciones de validación (*Hibernate*) en los campos de las entidades definidas.
ControllerGenerator	Genera los controladores definidos en la especificación.
MapperGenerator	Genera los mapeadores necesarios para cada entidad definida. Hace uso de los siguientes subgeneradores: ● *EndpointBuilder*: genera los endpoints necesarios para cada controlador. ● *ParameterBuilder*: genera los parámetros y validaciones de cada endpoint.
ResourceGenerator	Genera los recursos necesarios para cada entidad definida.
ResponseGenerator	Genera los objetos de respuesta para cada entidad definida.

Tabla 5.4. Generadores desglosados

5.2.2 Estructura de los generadores (Patrón *Builder*)

Como hemos visto, se hace uso de una gran cantidad de generadores independientes. Esto hace que el código sea fácil de mantener ya que la lógica de generación de cada componente está encapsulada en un generador independiente. Además para facilitar la implementación y también aumentar la mantenibilidad todos los generadores hacen uso del patrón *Builder* [12]. Esto permite que la lógica de funcionamiento de todos los generadores sea la misma.

Por ejemplo, en el caso del *ControllerGenerator* contamos con varios constructores a su vez independientes entre sí los cuales generan los componentes necesarios. En la **Imagen 5.1** se expone un diagrama para facilitar su comprensión.

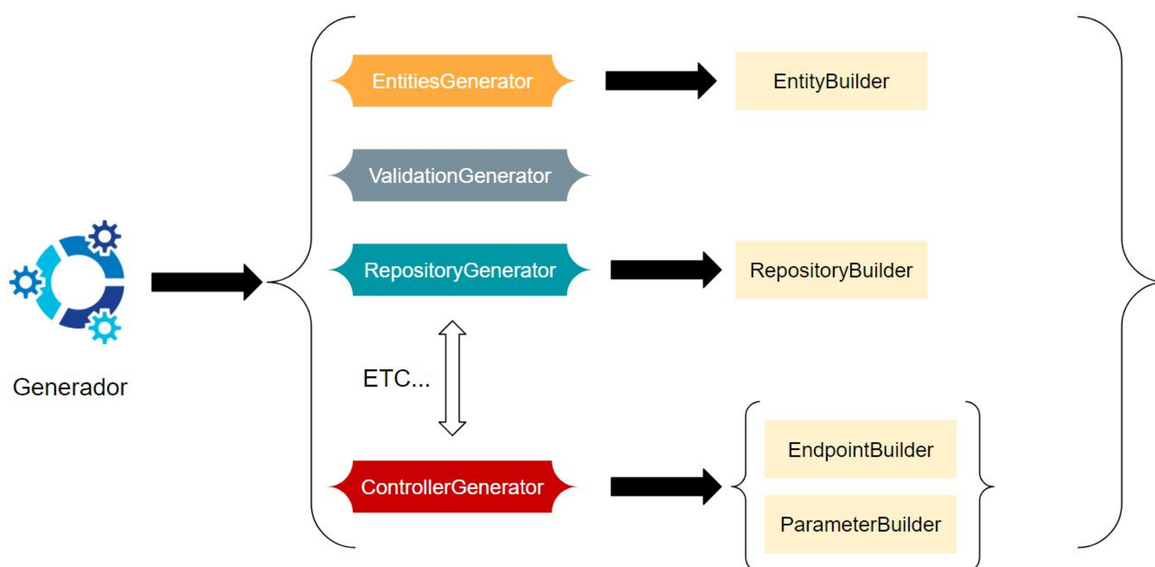

Imagen 5.1. Estructura de los generadores independientes

5.2.3 Detección de campos relacionados en las entidades

En esta sección se va a detallar el proceso que se siguió para detectar los diferentes tipos de relaciones en los campos de una entidad ya que es una de las funciones en las que más se ha tenido que investigar para conseguir desarrollarla.

Primero veamos los tipos de relaciones que pueden tener los campos de las entidades. Tanto en el modelo relacional de las bases de datos como en el modelo orientado a objetos, existen varias tipos de relaciones entre los datos, *Hibernate* las distribuye de las siguiente manera [13]:

- **Relaciones @*OneToOne***: Relaciones uno a uno. Es un tipo de cardinalidad que aplica a la relación entre unas entidades A y B en la cual un elemento A solo puede estar relacionado con un elemento de B, y viceversa.

  En las relaciones @*OneToOne* cabe destacar también que una de las dos partes de la relación actúa como dueña de la misma. Esto implica que en una entidad el campo relacionado contendrá también la anotación @*JoinColumn* mientras que el campo relacionado de la entidad inversa simplemente usa la anotación @*JoinColumn* de la parte dueña.

- Relaciones **@*OneToMany*** y **@*ManyToOne*.** Relaciones uno a muchos y muchos a uno. Es un tipo de cardinalidad que aplica entre unas entidades A y B en la cual un elemento de A referencia múltiples elementos de la entidad B, mientras que varios elementos de la entidad B referencian al mismo campo de la entidad A.

  Al igual que en las relaciones @*OneToOne*, en este caso la relación que actúa como dueña es la @*ManyToOne* y por tanto es quien tendrá la anotación @*JoinColumn* que realiza la conexión con la base de datos, mientras que la parte inversa se aprovecha de ella.

- Relaciones **@*ManyToMany*.** Relación de muchos a muchos. Este tipo de relación implica el uso de una tabla intermedia en la base de datos. Este tipo de cardinalidad aplica a cuando entre dos entidades A y B, varios elementos de A se referencian con múltiples elementos de B y viceversa.

  Al igual que en las relaciones explicadas anteriormente, y de manera muy similar a las relaciones @*OneToOne*, en este tipo de relaciones, una de las partes toma el papel de dueña de la relación e implementa la anotación @*JoinTable* en su campo, mientras que la otra parte hace uso de esto.

Sabiendo esto, se necesita una forma de diferenciar toda esta información a partir de los datos recibidos en la especificación de *OpenAPI*. Para esto, se hace uso de la extensión *x-apigen-models*, descrita anteriormente, donde se describen las entidades junto con sus campos. Aquí es donde en cada campo se establece la información necesaria para

reconocer el tipo de relación y hacía qué otra entidad. En la **Imagen 5.2** se puede ver detalladamente la información recibida para cada tipo de relación.

Campo Extensión	ManyToManyOwner	ManyToMany	ManyToOne	OneToMany	OneToOneOwner	OneToOne
type:	X (Array)	X (Array)	X	X (Array)	X	X
items-type:	X	X	-	X	-	-
column: string	X	X	X	-	X	-
foreign-column: string	X	X	-	X	-	X
intermediate-table: string	X	X	-	-	-	-
owner: boolean	X (true)	X (false)	-	-		

Imagen 5.2. Estructura de la información de las relaciones en la extensión de *OpenAPI*

En la **Imagen 5.2** se puede observar cómo cada tipo de relación recibe una información diferente desde la especificación, lo cual nos permite diferenciar entre ellas.

Una vez solucionado el tipo de relación de cada campo, el problema surge cuando tenemos que averiguar la entidad con la que se relacionan los campos relacionados. Para esto, se hace uso de una clase dedicada exclusivamente al análisis de los campos relacionados de las entidades recibidas en la especificación. Esta clase se llama ***EntityRelationManager*** y desde el generador de entidades se usa para, primero encontrar los campos relacionados, y segundo establecer tanto la entidad a la que referencian, como el tipo de relación con sus respectivas anotaciones.

Para ver esto con un ejemplo más claro, veamos la **Imagen 5.3** y la **Imagen 5.4**, donde a partir de la información recibida de la especificación, somos capaces de generar las anotaciones pertinentes en cada caso.

Extensión de OpenAPI

```yaml
- name: pacientes #@ManyToMany-Owner
  type: Array
  items-type: Paciente
  relational-persistence:
    column: medico_id
    foreign-column: paciente_id
    intermediate-table: consultas
    owner: true
```

Médico

```yaml
- name: medicos #@ManyToMany
  type: Array
  items-type: Medico
  relational-persistence:
    column: paciente_id
    foreign-column: medico_id
    intermediate-table: consultas
    owner: false
```

Paciente

Imagen 5.3. Relaciones definidas en la extensión de *OpenAPI*

API Generada

```
/**
 * Campo pacientes con relación ManyToMany con Paciente */
@ManyToMany
@JoinTable(
        name = "consultas",
        joinColumns = {
                @JoinColumn(name = "medico_id", foreignKey = @ForeignKey(name = "medico_id"))
                },
        inverseJoinColumns = {
                @JoinColumn(name = "paciente_id", foreignKey = @ForeignKey(name = "paciente_id"))
                },
        uniqueConstraints = {
                @UniqueConstraint(columnNames = {"medico_id", "paciente_id"})
                }
)
private Set<Paciente> pacientes;
```

Médico (Entidad)

```
/**
 * Campo Paciente con relación ManyToMany con medicos */
@ManyToMany(
        mappedBy = "pacientes"
)
private Set<Medico> medicos;
```

Paciente (Entidad)

Imagen 5.4. Relaciones generadas en la API

5.2.4 Generación de validaciones en los parámetros de los *endpoints*

Otro de los puntos que necesitó de bastante análisis y creatividad para resolver el problema fue en el caso de la generación de validaciones en los parámetros de los *endpoints* de los controladores.

Para ponernos en contexto, cuando hablamos de *endpoints* hacemos referencia a cada punto el cual devuelve una respuesta en nuestra API [14]. Por ejemplo el que podemos ver en la **Imagen 5.5**:

```
@GetMapping("/{id}")
@ResponseStatus(
        code = HttpStatus.OK
)
public MedicoResponse getMedico(@PathVariable("id") Long id) {
    Medico searchResult = service.search(id);
    MedicoResource result = mapper.toResource(searchResult);
    return new MedicoResponse(result);
}
```

Imagen 5.5. *Endpoint* de obtener por *id*

Como podemos observar, este *endpoint* responde a una petición *GET* en la cual se pasa como parámetro un campo *id*. El endpoint realiza la búsqueda en la base de datos mediante el servicio, mapea el resultado para poder ser devuelto y lo devuelve como respuesta.

Hay casos en los que recibimos parámetros los cuales queremos verificar que cumplan ciertas condiciones, es decir validarlos. Si se cogiera el *endpoint* visto en la **Imagen 5.5** y se quisiera validar que el parámetro *id* recibido sea por ejemplo, mayor que *n*, simplemente se le debería añadir la anotación ***@Min(value = n)***.

Para poder detectar este tipo de validaciones desde el generador es necesario establecerlas en la especificación de *OpenAPI*. Pero esto no se realiza de manera directa, sino que se usa la sintaxis específica de *OpenAPI*. Para poder realizar esta conversión

entre *OpenAPI* y las validaciones de *Hibernate* [15], en la **Imagen 5.6** se puede ver una comparación.

Schema	Annotation
default	@RequestParam(defaultValue = X)
minimum	@Min(X) / @DecimalMin(X)
maximum	@Max(X) / @DecimalMax(X)
minItems maxItems	@Size(min=X, max=X)
exclusiveMinimum	@DecimalMin(value = "0.1", inclusive = false)
exclusiveMaximum	@DecimalMax(value = "0.5", inclusive = false)
minLength maxLength	@Size(min=X, max=X)
pattern	@Pattern(regexp = "X")
nullable	@Nullable

Imagen 5.6. Comparación entre validaciones de parámetros de *OpenAPI* e *Hibernate*

Con la ayuda de la comparación descrita en la **Imagen 5.6** se ha podido implementar de manera sencilla el código necesario para la generación de las validaciones necesarias. En la **Imagen 5.7** se puede observar un ejemplo de la generación exitosa del proceso descrito.

```
public MedicoResponse getMedico(@PathVariable("id") @Min(2) @Max(10) Long id) {...}
```

Imagen 5.7. Generación de validaciones en los parámetros de un *endpoint*

Capítulo 6.

Pruebas

En este capítulo se van a explicar los principios seguidos para la realización de pruebas en el proyecto desarrollado. Las pruebas además, han sido desarrolladas siguiendo las directrices de la empresa.

6.1. *Object Mother*

Object Mother es un patrón de diseño de software el cual nos permite definir una clase encargada de generar objetos con ciertas características específicas de modo que podamos obtenerlos para realizar pruebas [16]. Este patrón se ha utilizado para crear objetos con los cuales realizar pruebas unitarias.

6.2. Pruebas unitarias

Las pruebas unitarias implementadas han seguido el modelo *Cucumber* y la convención *Gherkin* [17]. Dicho modelo consiste en establecer el nombre de los tests de una forma específica, la cual permite entenderlos de manera sencilla. Ese formato es el siguiente:

Given_When_Then

- ***Given*** (Dado): Situación inicial dada
- ***When*** (Cuándo): Acción que se realiza
- ***Then*** (Entonces): Resultado que se espera obtener

6.3. Pruebas unitarias sobre *Java Poet*

Como ya se ha visto, *Java Poet* es una librería la cual proporciona una API para la generación de código *Java.* La principal funcionalidad del sistema desarrollado consiste en generar código con esta librería. Por esto realizar pruebas unitarias sobre el proceso de generación ha sido importante al largo de todo el desarrollo. Para poder realizar tests unitarios sobre *Java Poet* se ha tenido que consultar muy cuidadosamente la documentación del mismo sobre pruebas para entender el funcionamiento correctamente [18]. En esta documentación se puede observar como se recomienda realizar tests sobre fragmentos específicos de código, aunque en ciertas ocasiones se pueden realizar sobre fragmentos más amplios.

Todos los generadores citados en la **Tabla 5.4**, así como sus respectivos *builders* han recibido cobertura de pruebas unitarias.

6.4. Ejemplo de prueba unitaria

Con los principios descritos en los apartados **6.1**, **6.2** y **6.3**, se han implementado pruebas unitarias como las que podemos observar en la **Imagen 6.1**, donde se usa tanto el *Object Mother*, el modelo *Cucumber* y el estándar de pruebas de *Java Poet*.

```
@Test
void givenAResponseWithAttribute_whenGenerateResource_thenFileStructureIsCorrect() {
    EndpointBaseResponse response = EndpointBaseResponseObjectMother.simpleResponseWithSimpleAttribute(
    ResourceBuilder builder = new ResourceBuilder(response, "the.base.package");
    TypeSpec spec = builder.build();
    Assert.assertEquals("The name is wrong", "EntityNameResource", spec.name);
    Assert.assertEquals("The file is not a class", "CLASS", spec.kind.name());
    Assert.assertFalse("The annotations are wrong", spec.annotations.isEmpty());
    Assert.assertEquals("The @Data annotation is wrong", "@lombok.Data",
     spec.annotations.get(0).toString());
    Assert.assertEquals("The @ApigenResource annotation is wrong",
     "@net.cloudappi.apigen.archetypecore.core.resource.ApigenResource",
      spec.annotations.get(1).toString());
    Assert.assertFalse("The fields are wrong", spec.fieldSpecs.isEmpty());
}
```

Imagen 6.1. Ejemplo de prueba unitaria donde se utilizan los principios seguidos

Capítulo 7.

Conclusiones

Después de mi estancia en prácticas y la realización del proyecto descrito en este documento, puedo decir que estoy contento de haberlo realizado. He trabajado en un ambiente profesional, rodeado de compañeros muy amables y profesionales. No ha sido mi primera experiencia laboral, pero sí la primera del sector de la programación.

Dadas las circunstancias vividas debido a la pandemia del Covid-19, las prácticas han tenido que ser realizadas en la modalidad de teletrabajo. Pero esto, no sólo no ha supuesto un deterioro en la calidad de las mismas, sino que he experimentado una nueva forma de trabajo, el teletrabajo. La empresa en la que se han realizado las prácticas tiene muy buena infraestructura para el teletrabajo, y me he sentido apoyado en todo momento por su parte.

En cuanto al proyecto, ha supuesto un verdadero reto ya que en un principio yo carecía de los conocimientos sobre las tecnologías utilizadas. Pero gracias a eso, he adquirido una enorme cantidad de conocimiento sobre APIs y desarrollo *back-end*.

Con el desarrollo de este proyecto considero que he puesto en práctica gran parte de los conocimientos aprendidos durante los estudios del grado.

Por último, el proyecto, una vez finalizada mi estancia en prácticas fue tomado por la empresa y ya están empezando incluso a comercializarlo [19].

Bibliografía

[1] OpenAPI Specification. Documentación oficial. https://swagger.io/specification/ [Consulta 20 de Febrero de 2020].

[2] JavaPoet. Documentación oficial. https://github.com/square/javapoet [Consulta 20 de Febrero de 2020]

[3] Spring. Documentación oficial. https://spring.io/ [Consulta 18 de Febrero de 2020]

[4] Hibernate. Documentación oficial. https://hibernate.org/ [Consulta 25 de Febrero de 2020]

[5] MapStruct. Documentación oficial. https://mapstruct.org/ [Consulta 4 Mayo de 2020]

[6] OpenAPIV3Parser. Documentación oficial. https://github.com/swagger-api/swagger-parser

[Consulta 13 de Marzo de 2020]

[7] Sueldos para Programador Junior.

https://www.glassdoor.es/Sueldos/programador-junior-sueldo-SRCH_KO0,18.htm

[Consulta 26 de Junio de 2020]

[8] OpenAPI Extensions. Documentación oficial.
https://swagger.io/docs/specification/openapi-extensions/ [Consulta 23 de Marzo de 2020]

[9] Defining JPA Entities. https://www.baeldung.com/jpa-entities [Consulta 22 de Febrero de 2020]

[10] JPA Repositories. Documentación de Spring
https://docs.spring.io/spring-data/jpa/docs/1.5.0.RELEASE/reference/html/jpa.repositories.html [Consulta 3 de Marzo de 2020]

[11] Guide to Working with Multiple Modules. Documentación oficial.
https://maven.apache.org/guides/mini/guide-multiple-modules.html [Consulta 19 de Febrero de 2020]

[12] Builder Design Pattern in Java.
https://howtodoinjava.com/design-patterns/creational/builder-pattern-in-java/ [Consulta 3 de Marzo de 2020]

[13] Java persistence with JPA and Hibernate, Part 1: Entities and relationships. https://www.javaworld.com/article/3373652/java-persistence-with-jpa-and-hibernate-part-1-entities-and-relationships.html [Consulta 10 de Marzo de 2020]

[14] API Endpoints - What Are They? Why Do They Matter? https://smartbear.com/learn/performance-monitoring/api-endpoints/ [Consulta 17 de Marzo de 2020]

[15] The Java EE 6 Tutorial. Using Bean Validation. https://docs.oracle.com/javaee/6/tutorial/doc/gircz.html#:~:text=JavaBeans%20Validation%20(Bean%20Validation)%20is,built%20in%20or%20user%20defined. [Consulta 3 de Abril de 2020]

[16] Combining Object Mother and Fluent Builder for the Ultimate Test Data Factory. https://reflectoring.io/objectmother-fluent-builder/ [Consulta 22 de Febrero de 2020]

[17] Gherkin Reference https://cucumber.io/docs/gherkin/reference/ [Consulta 8 de Mayo de 2020]

[18] Java Poet Tests. https://dspot-demo.stamp-project.eu/job/93 [Consulta 15 de Mayo de 2020]

[19] Genera tu API en 5 minutos. CloudAPPI https://cloudappi.net/blog/nuestro-blog-1/post/genera-tu-api-en-5-minutos-96#scrollTop=0