Universidad Zaragoza

Trabajo Fin de Grado

Datos en un contenedor

Data in a container

Autor

Jaime Conchello Bauto

Director

Francisco Javier López Pellicer

ESCUELA DE INGENIERÍA Y ARQUITECTURA

2022

Agradecimientos

Me gustaría comenzar dando las gracias Francisco Javier López Pellicer por haberme ayudado durante todo el proyecto. Sin duda, su esfuerzo y dedicación han sido claves para sacar adelante este TFG.

También quiero agradecer a todos los compañeros y compañeras que he conocido durante el transcurso del grado y con los que he pasado muy buenos momentos.

Por último, y en especial, quiero dar las gracias a mi familia por haberme apoyado siempre que lo necesitaba durante todo estos años.

Resumen

Los métodos actuales para distribuir conjuntos de datos se basan principalmente en la compartición directa de ficheros con la información o en el desarrollo de una solución integradora que ofrezca una API de acceso para cada caso de uso. Una línea no explorada todavía es hacer uso de la funcionalidad ofrecida por la tecnología de contenedores *Docker*. Esta permite empaquetar código junto con todas sus dependencias y desplegarlo en entornos heterogéneos empleando ficheros que describen servicios y con una herramienta de línea de comandos. En este Trabajo Fin de Grado, se desarrollará una herramienta que permita configurar mediante un *script* la construcción y despliegue automático de un contenedor *Docker* que integre y exponga un conjunto de datos a través de una API tipo Web y una API tipo Remote Procedure Call o RPC construida en base de la especificación del conjunto de datos. Además, cada contenedor dispondrá de una página web con información sobre los datos almacenados y la capacidad de probar las interfaces expuestas de forma interactiva.

La solución propuesta simplifica el proceso de distribución de conjuntos de datos. A diferencia de las aproximaciones actuales, es posible compartir un único contenedor que cuente con todo lo necesario para que los distintos tipos de consumidores finales (aplicaciones web, sistemas de información, etc.) consulten la información que deseen a través de las diferentes interfaces expuestas. De este modo, no es necesario que los consumidores procesen manualmente las fuentes de datos, se eliminan los problemas de compatibilidad al trabajar en entornos heterogéneos y se garantiza la integridad de la información almacenada.

Como conclusión, el desarrollo del Trabajo Fin de Grado ha permitido desarrollar una primera aproximación a este modelo de distribución de conjuntos de datos. Supone un paso importante, ya que, se ha podido comprobar que la propuesta tiene potencial y ha permitido sentar la base sobre la que continuar trabajando en futuros trabajos. En el repositorio de *Docker Hub* `https://hub.docker.com/r/776012/diac`, se pueden descargar dos contenedores generados con la herramienta desarrollada en este Trabajo Fin de Grado y que almacenan y exponen datos abiertos de empresas reales.

Índice

1. Introducción

Un contenedor *Docker* es una tecnología que, mediante virtualización a nivel de sistema operativo, es capaz de encapsular código junto a sus dependencias para ser ejecutado en entornos heterogéneos. Las simplificaciones que aporta al proceso de distribución y despliegue de aplicaciones han potenciado su uso durante los últimos años. Pese a ello, todavía no se ha desarrollado una alternativa que permita integrar y distribuir conjuntos de datos de forma automática, siendo los desarrolladores quienes deben crear manualmente sus propias soluciones para cada caso de uso.

El objetivo de este Trabajo Fin de Grado es construir una herramienta mediante la cual un desarrollador pueda vía un script, configurar la construcción y despliegue automático de un contenedor *Docker* que distribuya y despliegue un determinado conjunto de datos exponiendo, por ejemplo, una API tipo Web y una API tipo Remote Procedure Call o RPC como interfaces de acceso en modo lectura.

1.1. Motivación

No se puede ignorar que actualmente se potencia cada vez más el desarrollo y creación de repositorios de recursos abiertos, en los que cada día se producen ingentes cantidades de datos [1]. Pese a ello, todavía no existe un método que permita la distribución de toda esta información de una forma sencilla y eficaz. En la mayoría de las ocasiones, estos datos son publicados o distribuidos directamente en ficheros [2], que difícilmente pueden ser procesados para extraer la información buscada, o es necesario que se desarrolle una solución a medida para ponerlos a disposición de los usuarios. Tampoco se debe olvidar que existen ciertos entornos dónde es imprescindible garantizar la integridad del contenido. Por ejemplo, cuando este haya de ser empleado por un tercero y sea preciso asegurar su reproducibilidad [3], extremo que es mucho más difícil de cumplir en el caso de que se opte por compartir de manera directa un fichero con la información que corresponda. Ya que, al contrario de un contenedor *Docker* [4], formado por diferentes capas de código y dependencias no modificables, un fichero con datos se puede modificar de forma sencilla. En este último caso, sería necesario utilizar funciones hash para asegurar la integridad de la información. De este modo, aunque la compartición directa de ficheros o el desarrollo puntual de una solución distribuidora pueda resultar útil en entornos dónde se gestionen pequeños volúmenes de información. Cuando se traslada a ámbitos en los que estos parámetros escalan y se incluyen relaciones entre los datos, se produce una considerable pérdida de eficiencia.

Todo lo anteriormente comentado ha motivado la realización de este Trabajo Fin de Grado. Se busca transformar la forma en la que se distribuyen conjuntos de datos. En vez de compartir uno o varios ficheros con la información pertinente, se puede poner a disposición del usuario un contenedor que, almacene los datos y ofrezca diferentes interfaces de acceso en modo lectura a los mismos. Antes de generar la imagen del contenedor definitiva, se puede configurar todo aquello que es ejecutado dentro del mismo. De este modo, es posible definir previamente a su distribución todas las consultas que se consideren más relevantes y así ofrecer un acceso estandarizado que además sea capaz de soportar múltiples consultas concurrentemente. Este formato de compartición recibe el nombre de *Data in a container*. Se utiliza este término para representar la herramienta que recibe un *script de configuración* y un conjunto de

datos como entrada y genera como salida un contenedor. El uso de la tecnología *Docker* garantiza que los contenedores puedan ser desplegados en entornos heterogéneos y, gracias a su modo de ejecución aislado del resto del sistema, protege la fuente de datos almacenada frente a posibles intentos de modificación por parte de los usuarios. A diferencia de la aproximación actual, donde las soluciones de integración para distribuir conjuntos de datos deben adaptarse a las dependencias y requisitos de cada entorno donde se ejecuten, trabajar con contenedores permite eliminar ese proceso de adaptación y limitarse a la implementación de la solución. Por último, cabe destacar que *Docker* cuenta con la plataforma *Docker Hub* [5] para publicar imágenes de contenedores y que así los usuarios puedan descargarlas de una forma sencilla y directa.

En la Figura 1, se muestra un esquema en alto nivel de la solución propuesta. Mediante un *script de configuración* se describe el contenido del contenedor *Docker*, que información se almacena, cuál es su modelo de datos, que consultas se definen sobre esos datos y en que interfaces se exponen. Además, enfocado en su distribución, en el *script de configuración* se puede declarar información referente a la licencia, nombre o fuente de los datos almacenados. Los conjuntos de datos a los que se hace referencia en el *script de configuración* deben estar presentes entre los datos que se reciben como entrada. Basándose en la configuración recibida, se carga la información en un sistema de almacenamiento embebido, se crea código a medida que expone las interfaces, y todo ello se empaqueta en el contenedor. Este último expone la información en las interfaces indicadas y ofrece una página de documentación, accesible una vez que se ha ejecutado la imagen del contenedor. La creación de esquemas de datos y la generación de código de forma automática suponen una reducción del trabajo manual, repercutiendo en un aumento de la productividad. Un ejemplo de su aplicación puede darse en procesos de migración, donde es necesario exponer con tecnologías actuales información que hasta el momento se almacenaba en ficheros de texto o en medios legados. Juntando esta herramienta y una plataforma como *Docker Hub*, es posible, por ejemplo, compartir un conjunto de datos con varias entidades relacionadas entre ellas. Bastaría con definir en un script el modelo de datos y las consultas e interfaces expuestas, para que así se generara un contenedor que pudiera ser publicado en *Docker Hub*. En vez de compartir por separado la fuente de datos y que cada usuario tuviera que implementar su propia solución para efectuar consultas, se puede distribuir una imagen del contenedor que contiene todo lo necesario para acceder a los datos de una manera directa y eficaz.

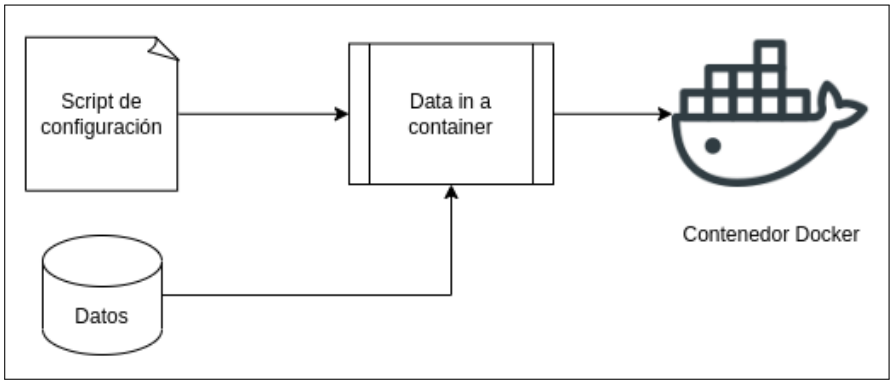


Figura 1: Esquema en alto nivel de la solución propuesta

1.2. Objetivo del Trabajo Fin de Grado

El objetivo de este Trabajo Fin de Grado es proponer un método de compartición de conjuntos de datos que haga uso de la tecnología de contenedores. Para ello, se busca desarrollar una herramienta que, recibiendo como entrada un conjunto de datos y un script donde se defina el modelo de datos del conjunto y las consultas e interfaces expuestas. Construya y despliegue de forma automática un contenedor *Docker* que distribuya y despliegue el conjunto de datos recibido como entrada, exponiendo las consultas definidas en las interfaces de acceso indicadas y ofreciendo la documentación necesaria para que los usuarios puedan consumir los datos almacenados. De la definición de este objetivo se derivan los siguientes desafíos:

- **Desarrollo de un lenguaje de dominio específico (DSL)**: Este DSL debe facilitar definir en el script el modelo de datos de la información almacenada, las consultas aplicadas sobre esta información, las interfaces de acceso expuestas al exterior para ejecutar las consultas e información básica para la distribución del contenedor.
- **Implementación de un intérprete del DSL**: Esta librería debe ser capaz de procesar los scripts en dicho DSL y generar el código necesario, ejecutar los procesos de construcción de programas, procesar los datos para su uso por dichos programas y construir contenedores basándose en esa configuración.
- **Construcción del contenedor**: Utilizando el intérprete y las herramientas de *Docker*, esta herramienta debe ser capaz de construir un contenedor con los requisitos extraídos del procesamiento del script y desplegarlo de forma automática.

1.3. Alcance

El proyecto desarrollado debe encontrarse dentro de los límites de un Trabajo Fin de Grado. En este caso, se trata de una primera aproximación al concepto *Data in a container*. Así, se ha decidido establecer unos requisitos mínimos que debería tener la solución desarrollada para exponer el potencial del concepto y sentar las bases para futuros trabajos (ver Tabla 1). A continuación, se exponen y justifican estos requisitos mínimos.

La solución debe trabajar con *Docker* como tecnología de contenedores. Se ha seleccionado porque es una solución de código abierto, ampliamente utilizada y dispone de la plataforma de distribución *Docker Hub*. No se ha seleccionado otra tecnología

Tecnología	**Requisitos mínimos**
Tecnología de contenedores	*Docker*
Lenguaje de base para el DSL	*Kotlin*
Framework de desarrollo	*Spring Framework*
Modelo de datos soportado	CSV
Interfaces expuestas	REST API, *GraphQL* y *gRPC*

Tabla 1: Resumen de los requisitos mínimos de la solución

de contenedores, ya que no se ha encontrado una alternativa que cuente con la misma funcionalidad y además disponga de un entorno de desarrollo y trabajo como *Docker*. El lenguaje de base para el DSL debe ser *Kotlin*. Su elección se ha basado en la capacidad de crear DSL definidos íntegramente con funcionalidad nativa del lenguaje gracias a la tecnología *Kotlin* DSL [6] y la capacidad para ejecutar ficheros con código *Kotlin* sin compilación previa gracias a la tecnología *Kotlin Scripting* [7]. No se ha encontrado ningún otro lenguaje que ofrezca las mismas características que *Kotlin*. Por otra parte, la solución debe caracterizarse por emplear el *framework* de desarrollo *Spring* [8]. Esto se debe a que se trata de una tecnología madura que permite simplificar aspectos como la configuración o el despliegue, entre otros. Respecto a la herramienta en sí, únicamente se soporta CSV como formato de entrada de los datos almacenados en el contenedor. Esta decisión se debe a que se trata de una primera aproximación, por lo que se ha elegido un formato fácil de procesar y que además es muy empleado para publicar conjuntos de datos en grandes repositorios o por parte de empresas. En lo referente a las interfaces que debe ofrecer el contenedor para ejecutar las consultas, se usa REST API [9], *GraphQL* [10] y *gRPC* [11] debido a que se trata de tecnologías estables, bien documentadas y su uso está extendido en la actualidad. No se han seleccionado otras interfaces, ya que con esta combinación se garantiza que los datos almacenados pueden ser accedidos desde multitud de entornos y se considera un conjunto representativo de las tecnologías utilizadas actualmente. En caso de desconocer estas tecnologías, se puede consultar el glosario, donde se presenta una breve definición de cada una. Relacionado con las interfaces, la solución debe ofrecer la posibilidad de definir consultas que cuenten con una única condición de búsqueda y con la posibilidad de filtrar los resultados devueltos. Se limita a una condición porque se ha determinado que añadir consultas con condiciones más complejas de filtrado queda fuera del alcance este Trabajo Fin de Grado. Además, la herramienta debe generar de forma automática para cada contenedor una página de documentación donde se permita visualizar las entidades y sus atributos, así como probar de forma interactiva las diferentes interfaces expuestas.

Por último, la herramienta debe poderse validar con conjuntos de datos abiertos, que dispongan de esquemas con diferentes tipos de datos y una o más entidades, con relaciones entre las mismas cuando sea posible. Se deben definir consultas que sean expuestas en las tres interfaces descritas anteriormente y el resultado debe poder ser publicado en la plataforma *Docker Hub*.

1.4. Estructura del documento

A continuación, se describe como se ha estructurado la memoria del Trabajo Fin de Grado. En la Sección 2 se explica la fase de análisis del proyecto. La Sección 3 describe la fase de diseño del sistema. Para ello, se comenta la arquitectura del sistema con diferentes diagramas. La Sección 4 explica algunos aspectos del proceso de desarrollo. Seguidamente, la Sección 5 explica como se ha validado el sistema y la Sección 6 expone las conclusiones del Trabajo Fin de Grado. Finalmente, el Apéndice A resume como se ha gestionado el proyecto, el Apéndice B describe la fase de estudio de *Kotlin*, el Apéndice C comenta algunos detalles de implementación de bajo nivel y el Apéndice D y Apéndice E muestran información adicional sobre las pruebas de validación.

2. Análisis

La fase de análisis de todo proyecto es de vital importancia para poder sentar la base del desarrollo y evitar futuras dificultades. En este caso, se ha utilizado para terminar de definir como se debe comportar el sistema, especificar que tecnologías se van a emplear, desarrollar una prueba de concepto, así como estudiar y conocer el funcionamiento de diferentes tecnologías que se utilizan más adelante en el proyecto.

Es importante destacar que, debido a la naturaleza de la herramienta, existen dos partes bien diferencias en el mismo. Por un parte, el contenedor generado, y por otra, el sistema encargado de procesar el script de configuración y construir el contenedor. Por simplicidad, durante esta sección se usará *Data in a container* para referirse a este último.

2.1. Selección de tecnologías

En este apartado se exponen las tecnologías elegidas para el proyecto. En concreto, el lenguaje de programación, el sistema gestor de base de datos y las interfaces de acceso. Para estas últimas, se muestra un ejemplo simple de su funcionamiento.

Tras un análisis de varias posibilidades, se tomó la decisión de desarrollar la implementación íntegramente en el lenguaje de programación *Kotlin* [12]. Al comienzo del proyecto se barajó la posibilidad de utilizar *Python* [13], pero esta quedó descartada, ya que *Kotlin* ofrece una solución interoperable, soporta el extenso abanico de librerías de *Java* [14] y cuenta con una sintaxis sencilla de codificar. Esta decisión también se traslada al código ejecutado dentro del contenedor, donde al igual que en el módulo *Data in a container*, se despliega una aplicación en *Kotlin* con *Spring Boot* [15] que expone los datos almacenados a través de diferentes interfaces.

El siguiente paso fue determinar el sistema gestor de bases de datos que se incluye en el contenedor y donde se almacena la información. En este caso se analizó una única posibilidad, *SQLite* [16]. Se trata de una base de datos embebida, que se ejecuta junto con la aplicación a la que sirve. Su simplicidad, reducido peso, capacidad para ejecutarse sin la necesidad de un proceso servidor adicional y configuración prácticamente nula, la convierten en una gran opción para este sistema. Además, exceptuando algunos aspectos como gestión de permisos, operaciones JOIN de tipo *RIGHT y FULL OUTER* y soporte total de *triggers* y sentencias *alter table*, ofrece un soporte prácticamente total del estándar SQL [17].

Tal y como se ha mencionado anteriormente, las interfaces que ofrece el contenedor para el acceso a datos son REST API [9], *GraphQL* [10] y *gRPC*. De esta forma, tras definir el lenguaje de programación, el sistema de persistencia y las interfaces, se eligieron las dependencias que debía tener el proyecto ejecutado en el contenedor. Como se ha comentado previamente, este es uno de los grandes beneficios de trabajar con *Kotlin*, ya que se pueden utilizar todas las librerías que ofrece el *framework Spring* o que trabajan sobre el mismo (ver Tabla 2).

Para finalizar este apartado de selección de tecnologías, se procede a mostrar a partir de un ejemplo trivial como una misma consulta se expone por REST API, *gRPC* y *GraphQL*. El objetivo es explicar las características y diferencias de cada una de las interfaces seleccionadas. Como ejemplo, se supone que existe un conjunto de datos que almacena información relativa a libros y autores y cuenta con las tuplas mostradas en la Figura 2. Cada autor tiene su clave primaria, nombre, fecha de nacimiento y

Tecnología	Descripción
Spring Data JPA	*Framework* empleado para modelar y acceder a las entidades almacenadas en *SQLite* [18].
Spring Data REST	*Framework* para exponer una vista REST de los datos [19].
GraphQL DGS Spring Boot	*Framework* para exponer una vista *GraphQL* de los datos [20].
gRPC Spring Boot Starter	Framework para ofrecer una vista RPC de los datos sobre una aplicación que utiliza *Spring Boot* [21].

Tabla 2: Librerías utilizadas para la construcción del contenedor.

género literario con el que se identifica. Por otra parte, un libro está definido por su identificador único (ISBN), la clave extranjera de su autor, el título y la descripción de la obra.

Autor

autor_id	*genero_literario*	*fecha_nacimiento*	*nombre*
1	Narrativo	14-03-1958	X
2	Dramático	7-10-1985	Y

Libro

ISBN	*autor_id*	*descripción*	*título*
321	2	[.....]	X1
789	1	[.....]	Y1

Figura 2: Tuplas de ejemplo para el modelo de libros y autores

Es posible definir una consulta que nos devuelva la fecha de nacimiento del autor o autora que escribió un libro con un determinado ISBN. En REST API esta consulta podría exponerse como una operación de tipo GET sobre el recurso `/libros/{ISBN}/autor`. En esta interfaz se devuelve la tupla completa del autor del libro, no se devuelve únicamente el campo `fecha_nacimiento`.

```
Consulta:
GET /libros/789/autor

Resultado:
{
"autor_id": 1,
"genero_literario": "Narrativo",
"fecha_nacimiento": "14-03-1958",
"nombre" : "X"
}
```

Listado 1: Ejemplo de petición REST API

El caso de *GraphQL* es diferente al resto. Es posible exponer una consulta que filtre los libros por el ISBN y especificar en cada ejecución que atributos se

quieren obtener. Para este ejemplo se supone que se expone una consulta denominada `obtenerFechaNacimientoAutor` que recibe como parámetro de entrada el ISBN del libro. Como se aprecia en el Listado 2, se pueden especificar los campos que se desean obtener. *GraphQL* se limita a obtener toda la información relacionada con el libro con ese ISBN y después permite filtrar el resultado.

```
Consulta:
obtenerFechaNacimientoAutor("ISBN" : 789) {
    autor { fecha_nacimiento }
    titulo
}
Resultado:
{
    "autor_id" { "fecha_nacimiento": "14-03-1958" }
    "titulo" : "Y1"
}
```

Listado 2: Ejemplo de petición GraphQL

En la interfaz *gRPC* se implementa la consulta como la invocación a una función de un servicio remoto. El contenido devuelto se encuentra en formato *Protobuf*, por lo que es necesario un cliente capaz de procesar ese tipo de peticiones. Para este caso se supone que existe un servidor *gRPC* que expone una función denominada `obtenerFechaNacimientoAutor`, que recibe como parámetro de entrada el ISBN del libro y devuelve la fecha del autor. Para su invocación bastaría enviar la petición *obtenerFechaNacimientoAutor(789)* al servidor. A diferencia de *GraphQL* no es posible especificar los atributos que se quieren obtener, pero tampoco se devuelve toda la entidad completa como ocurre con REST API.

2.2. Prueba de concepto

Antes de comenzar con el desarrollo del sistema, se realizó una prueba de concepto. El objetivo era desplegar en un contenedor un conjunto de datos, con una única entidad, y exponer consultas sobre el mismo a través de las tres interfaces descritas anteriormente. Cabe destacar que, en esta prueba no se incluye la definición del script o el despliegue parametrizado del contenedor, es decir, todo se lleva a cabo de forma manual para verificar que realmente es posible crear un contenedor con esas características.

Para el desarrollo de la prueba, se seleccionó un fichero CSV que contenía información sobre el catálogo de títulos de Netflix [22]. Todo ello publicado en la plataforma *Kaggle* [23] bajo la licencia *CC0: Public Domain*, que permite copiar, modificar y distribuir el conjunto de datos sin la necesidad de pedir permiso al autor. En la Figura 3 puede observarse el modelo de datos de la entidad con la que se trabaja.

Figura 3: Modelo de datos del catálogo de Netflix

El desarrollo de esta prueba permitió definir los pasos para construir el contenedor, así como la configuración de todas las dependencias necesarias. La estructura se creó con la herramienta *Spring initializr* [24], que permite generar un proyecto de *Spring Boot* de forma sencilla y automática. En la Figura 4 puede observarse el diagrama de paquetes y clases del proyecto desarrollado donde vemos, por una parte, los paquetes `kotlin.com.demo.tfg` que cuentan con las clases necesarias para arrancar y configurar la aplicación, la definición de los controladores encargados de gestionar las consultas en *GraphQL* y *gRPC* y el repositorio donde se especifica el modelo de la entidad, el acceso a datos en el sistema de almacenamiento embebido y el controlador para REST API. Por otra parte, en el paquete `resources`, se encuentra la definición de ciertas variables de configuración y del esquema donde se especifica el servicio expuesto a través de *GraphQL*. Finalmente, el paquete `proto` cuenta con la definición del servicio expuesto en *gRPC*. A continuación, se procede a explicar los diferentes aspectos de este diagrama.

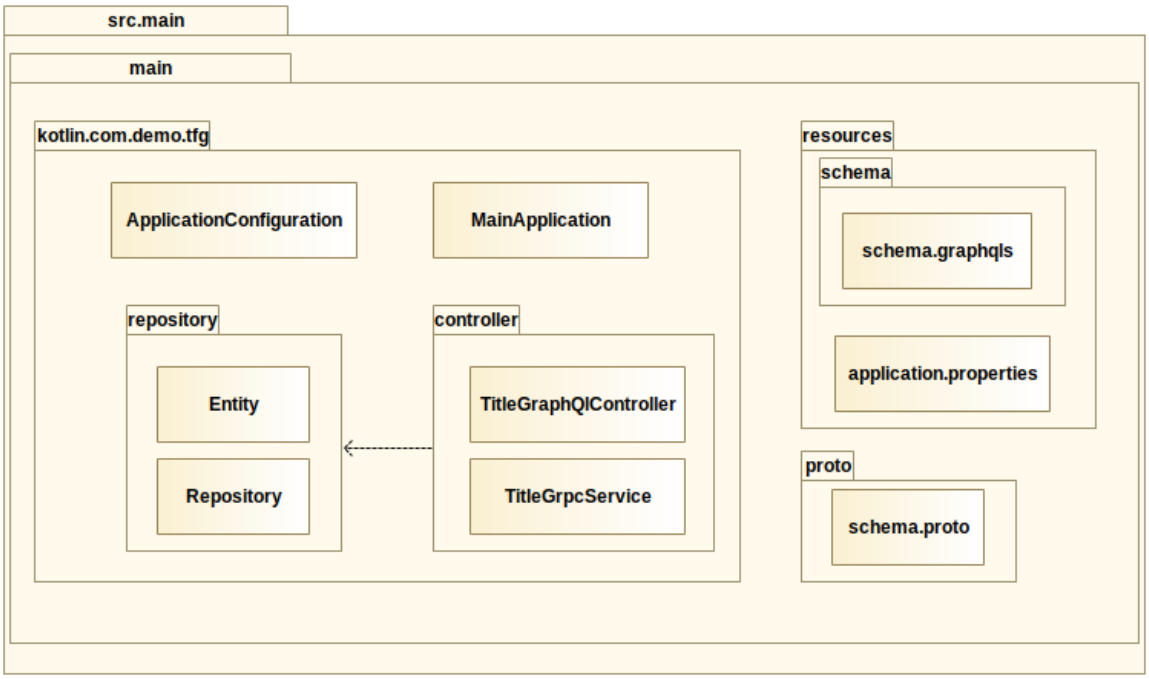


Figura 4: Diagrama de paquetes de la prueba de concepto

El modelado y acceso a los datos se desarrolla en las clases `Entity` y `Repository` respectivamente. La primera relaciona la entidad en la base de datos con una clase de *Kotlin* a través del mapeo objeto-relacional, desarrollado por *Hibernate* [25] y que implementa la API de persistencia de *Java* (JPA). La segunda, implementa el repositorio donde se definen las consultas sobre la entidad. Este, extiende la funcionalidad de una interfaz que por defecto permite contar con las operaciones básicas de *CRUD*, aunque en este caso se ha limitado para exponer únicamente acceso en modo lectura, así como con paginación y ordenación de resultados.

La incorporación de REST API al proyecto es directa al trabajar con la librería *Spring Data REST*, tan solo basta con etiquetar el repositorio creado con la anotación `@RepositoryRestResource` y configurar la ruta donde se expondrá el servicio.

En el caso de *GraphQL*, es necesario definir un esquema que incluya la entidad con la que se trabaja y especificar las consultas que se van a implementar. En la prueba, esto se incluye en `schema.graphqls`. Un fragmento de este fichero puede observarse en el Listado 3, donde se define primero el tipo `Query` con todas las consultas que se exponen en la interfaz y después se especifica la entidad con la que se trabaja, en este caso `TitleEntity`. Además, se debe crear un controlador, `TitleGraphQlController`, que relacione las consultas definidas en el esquema con las implementadas en el repositorio comentado anteriormente. Esto se debe a que *GraphQL* no exige un sistema de almacenamiento concreto, actúa como una capa intermedia entre el repositorio de información y el cliente.

```
type Query {
    titlesByReleaseYear(yearFilter: Int): [TitleEntity]
    titleByShowId(showId : String) : TitleEntity
    allTitles : [TitleEntity]
}
type TitleEntity {
    showId: ID
    releaseYear: Int
    ...
}
```

Listado 3: Fragmento de código de la definición del esquema para GraphQL

Finalmente, para incorporar el soporte de *gRPC*, hay que seguir un proceso similar al de *GraphQL*. Primero, se define el servicio, `schema.proto`, con las operaciones que se van a exponer y los mensajes necesarios para invocarlas. Después, se implementa el controlador para el servidor que recibirá y procesará las peticiones, en este caso `TitleGrpcService`.

```
service TitleService {
  rpc titleByShowId (ById) returns (Title);
  rpc allTitles(ListTitles) returns (stream Title);
  rpc titlesByReleaseYear(ByReleaseYear) returns (stream Title);
}

message ListTitles {}
message ById { string id = 1 ;}
message ByReleaseYear { int32 year = 1; }

message Title {
  string id = 1;
   int32 releaseYear = 2;
  string rating = 3;
  ...
}
```

Listado 4: Fragmento de código de la definición del servicio y mensajes de gRPC en la prueba de concepto

Con el desarrollo de la prueba de concepto, se pudo verificar que era posible construir un contenedor con las características buscadas. Algunos detalles destacables de esta primera aproximación fueron los siguientes:

- En un principio se decidió cargar los datos del CSV al iniciar la aplicación, pero rápidamente se detectó que esto ralentizaba el proceso de arranque. Como solución, se optó por emplear un script SQL que definiera la tabla e importara los datos directamente a la base de datos. De este forma, se generaba un fichero con toda la información, que podía ser accedido directamente por la aplicación, y no era necesario repetir el proceso de carga con cada arranque del sistema.
- El servicio encargado de ofrecer la interfaz *gRPC* se genera gracias al *plugin* *protoc-gen-grpc-java* [26]. Este, tomando como entrada el fichero con la definición del servicio y sus mensajes, `schema.proto`, define las clases y los *stubs* necesarios.

Finalmente, se debe crear un controlador, para cada uno de los servicios definidos, que extienda la clase creada por *protoc-gen-grpc-java* e implemente las operaciones definidas en el servicio. En la Figura 5, se puede apreciar un esquema del funcionamiento de este *plugin*.

- En esta primera versión, la construcción del contenedor se realiza mediante un script de *bash* que carga los datos en un fichero mediante *sqlite3*, un cliente en terminal para acceder a *SQLite*, y junto con el archivo Java (JAR) de la aplicación genera la imagen del contenedor.

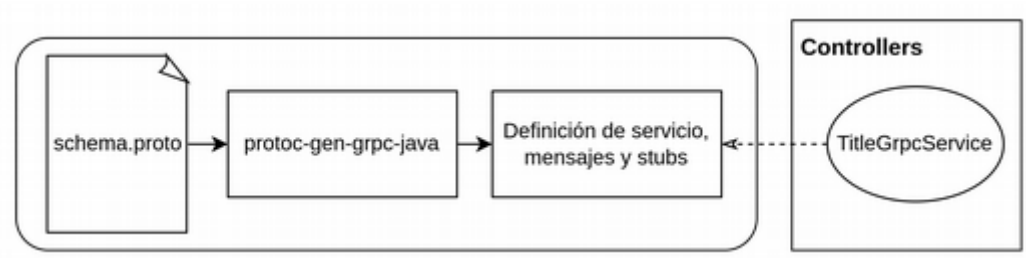


Figura 5: Esquema de la generación del servicio gRPC

En el Apéndice B se comentan pruebas adicionales que se realizaron durante esta fase para analizar la viabilidad de *Kotlin* en el proyecto. En particular, se estudió la ejecución de scripts y la definición de un DSL en el lenguaje.

2.3. Conclusiones de la fase de análisis

La realización de esta fase de análisis permitió confirmar que el conjunto de tecnologías seleccionadas para el desarrollo de la herramienta eran las idóneas. Además, se pudo definir como debía ser el flujo de trabajo seguido para transformar la construcción descrita en el script de configuración en un contenedor con los requisitos necesarios. A su vez, se extrajeron las siguientes conclusiones. Del proceso de estudio de *Kotlin* y el desarrollo de la prueba de concepto, se detectó la necesidad de diseñar una arquitectura modular que facilitara incorporar las diferentes funcionalidades con las que debía contar la herramienta y permitiera en un futuro añadir nuevas características de una forma sencilla. Además, en esta fase se determinó que era necesario que la documentación generada para cada contenedor estuviera centralizada en un único punto. Así, es posible ofrecer a los usuarios que reciben un contenedor creado con esta herramienta, un modo para conocer la estructura de los datos almacenados y probar las diferentes interfaces de acceso de forma estandarizada. De este modo, si es necesario construir otro programa o herramienta que trabaje sobre las interfaces del contenedor, se puede verificar el funcionamiento de las mismas antes de comenzar el desarrollo. Finalmente, desde un punto de vista de gestión del proyecto, esta fase ayudó a definir de manera clara los límites hasta los que resultaba viable llegar con el Trabajo Fin de Grado. En particular, fue muy importante el desarrollo de la prueba de concepto, ya que permitió sentar la base sobre la que poco a poco se añadiría funcionalidad para soportar una construcción paramétrica.

3. Diseño

En esta sección se describe el diseño del sistema. Esta fase debe establecer como se va a alcanzar los objetivos definidos anteriormente. Para ello, se muestra el sistema desde una vista física. En este caso se definen los diagramas de clases y paquetes necesarios para construir la solución.

3.1. Arquitectura del sistema

Por la naturaleza de la herramienta, y sus funcionalidades bien diferenciadas, se ha decidido implementar un proyecto multi-módulo. En la Figura 6 se puede observar el diagrama de paquetes del sistema. Se ha optado por mostrar una vista en alto nivel del mismo para explicar los detalles más importantes en este primer apartado.

- **Módulo core**: Incluye la lógica para, a partir de una configuración, construir la aplicación que se ejecutará en el contenedor.
- **Módulo dsl**: Define el DSL para describir la construcción del contenedor y configura e invoca al módulo `core`. Este DSL es el utilizado en los scripts que procesa el módulo `host`.
- **Módulo script**: Define como se deben procesar y ejecutar los scripts que recibe el módulo `host` y especifica las dependencias del módulo `dsl` que se deben importar por defecto al ejecutar un script.
- **Módulo host**: Actúa como punto de entrada al sistema. Recibe scripts con código *Kotlin* y los compila y ejecuta según la definición del módulo `script`. Durante esta ejecución invoca funcionalidad de los módulos `dsl` y `core`.

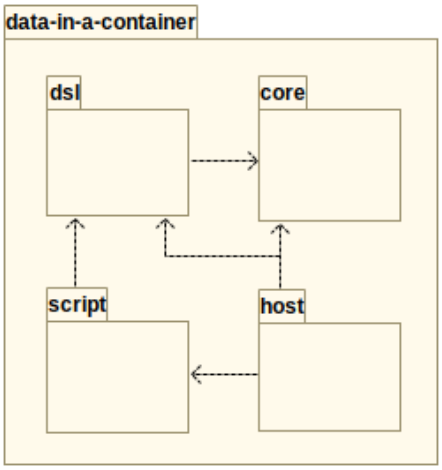


Figura 6: Diagrama de paquetes del sistema

En los siguientes apartados se explica más en detalle los módulos `core` y `dsl`. Se han añadido diagramas para acompañar la descripción de los mismos y así facilitar su comprensión. El diseño de los módulos `script` y `host` no se comentará, ya que es muy sencillo y está basado en la implementación oficial desarrollada por *Kotlin* para el soporte de scripts.

3.2. Diseño de la descripción del contenedor

El módulo `dsl` se comporta como la piedra angular del sistema. De forma similar a la mostrada en el Apéndice B, donde se comenta como especificar un DSL en *Kotlin*, define el lenguaje utilizado para describir como se debe construir el contenedor. Una vez que esta definición se ha completado, se encarga de invocar al módulo `core` con una configuración específica, para que, mediante plantillas, cree un proyecto con las características indicadas en la configuración y listo para ser desplegado en un contenedor.

En la Figura 7 se muestra una vista simplificada del módulo `dsl`. En particular, se ha eliminado de este diagrama el resto del contenido del paquete `entities`, ya que si no resultaba complejo de comprender. Como solución se ha creado un segundo diagrama (Figura 8), donde únicamente se encuentran las clases y paquetes del paquete `entities`. Más adelante, se procede a analizar cada uno de los diferentes aspectos de estas vistas.

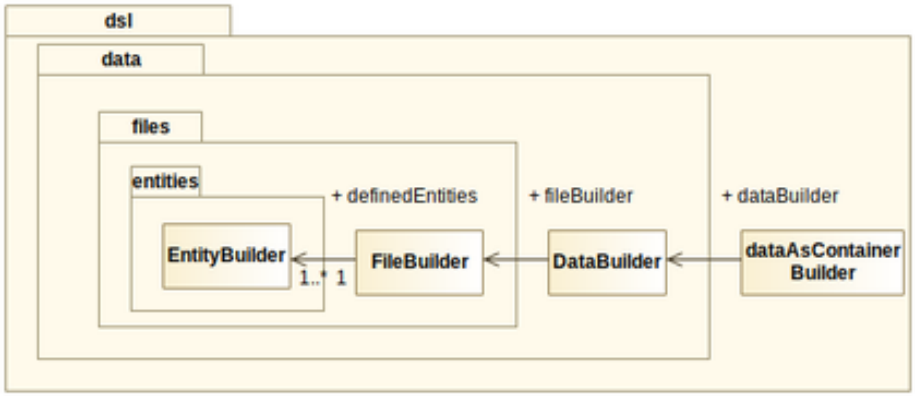


Figura 7: Diagrama de clases y paquetes del módulo DSL simplificado

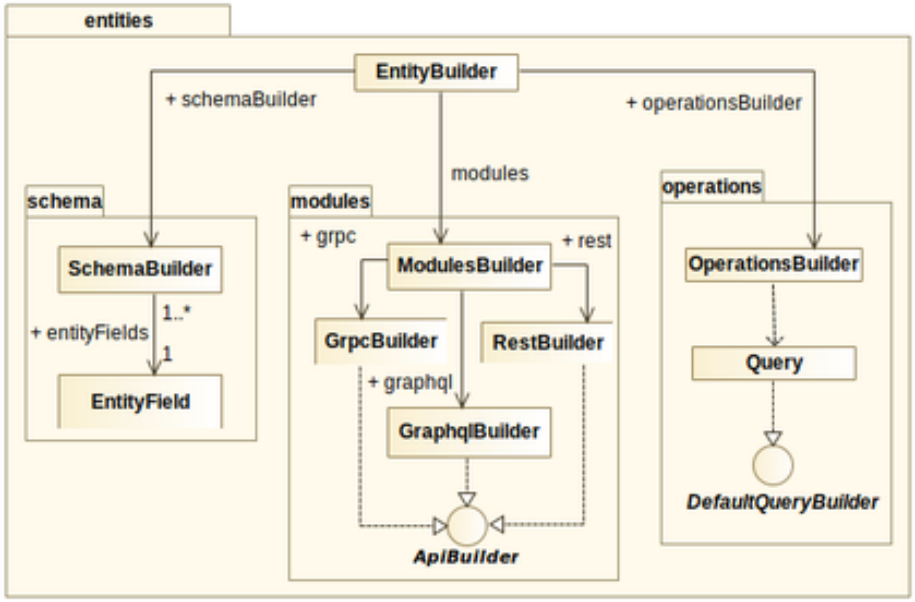


Figura 8: Diagrama de clases y paquetes del paquete `entities`

La arquitectura de este módulo se ha desarrollado siguiendo el siguiente razonamiento: Todos los contenedores descritos con el DSL deben almacenar y exponer un conjunto de datos y contar con una serie de variables para describir mínimamente la información almacenada. Esto se representa en la clase `dataAsContainerBuilder`, que cuenta con los atributos apropiados y la función para describir la información que se guarda.

A su vez, los datos almacenados en el contenedor pueden ser importados al mismo a través de diferentes medios. En este caso, únicamente se trabaja con ficheros CSV, pero dada la arquitectura del sistema, no sería complicado añadir uno nuevo. Dentro de la clase `FileBuilder`, es posible definir múltiples ficheros que serán empleados como fuente de datos. Cada uno de los ficheros definidos se asocia con una entidad, `EntityBuilder`. No es posible definir dos entidades en el mismo fichero. Tal y como ocurre en el modelo relacional, toda entidad cuenta con una serie de atributos, esto se representa en el paquete `schemas`. Adicionalmente, para la construcción del contenedor, es necesario definir las consultas que se van a aplicar sobre los atributos de la entidad, paquete `operations`, y las interfaces en las que se va a exponer la entidad y sus consultas, paquete `modules`. A continuación, se procede a explicar detalladamente los paquetes `schemas`, `modules` y `operations`.

3.2.1. Descripción del modelo de datos

En el siguiente apartado se explica como se ha diseñado el paquete `schema`, encargado de definir el modelo de datos de una entidad, incluyendo atributos y propiedades de los mismos. Como ocurre en el modelo relacional, una entidad puede contar con un número indeterminado de atributos. La clase `EntityField` se utiliza para representar un atributo en la entidad. Cada uno de estos atributos puede ser del tipo entero o cadena de texto. Al tratarse de un prototipo, se ha optado por no incorporar tipos adicionales, pero con el diseño actual se podrían añadir nuevos de forma casi directa en caso de que fuera necesario. Además, se soporta la definición de claves extranjeras y de clave primaria. A modo de ejemplo se muestran como se especificarían los atributos de una entidad perteneciente a un fichero previamente definido.

```
schema {
    <var_name> type <var_type> property isPrimaryKey
    <var_name> type <var_type>
    <var_name> type <var_type> references <table> column <colName>
}
```

Listado 5: Definición del esquema de una entidad con un atributo básico, una clave primaria y una clave extranjera

Como se puede apreciar en el fragmento de código superior, se hace uso de diferentes palabras clave para definir las propiedades de cada uno de los atributos de un esquema. Pese a su apariencia, esas palabras representan invocaciones a funciones de la clase `SchemaBuilder`. En concreto, esta funcionalidad se trata del soporte para la notación de infijo que implementa *Kotlin*. Permite invocar a funciones sin la necesidad de escribir los paréntesis, incluso cuando se pasa un parámetro de entrada. De esta forma, y encadenando invocaciones, se puede conseguir asignar todas las características de un atributo en una misma línea y de una forma muy similar a como se realiza en el estándar SQL. La utilidad de las funciones definidas es la siguiente:

- **type**: Permite determinar el tipo de dato del atributo. Como se ha comentado anteriormente, únicamente se soportan cadenas de texto y enteros.
- **property**: Especifica una propiedad del atributo. En este caso se puede indicar que se trata de la clave primaria, con el parámetro `isPrimaryKey` o de un atributo no nulo, con el parámetro `isNotNull`. La implementación actual no soporta el uso de claves primarias compuestas, esto podría ser un añadido de cara a un posible futuro trabajo.
- **references y column**: Para definir claves extranjeras, es necesario concatenar dos funciones. Primero, se debe utilizar `references` para indicar con que otra entidad está relacionada la actual y `column` para el atributo en concreto que se referencia. Se ha diseñado de tal forma que no es posible indicar primero la columna y luego la tabla. De esta manera se fuerza al usuario a especificar una sintaxis lo más parecido posible al estándar SQL a la hora de definir claves extranjeras. Como detalle de implementación, internamente Spring JPA permite definir una clave extranjera como un objeto del tipo al que apunta. De este modo, se simplifica la navegación entre relaciones y se puede acceder a todos los atributos de la entidad a la que se hace referencia sin necesidad de una segunda consulta.

Finalmente, un aspecto a destacar de este paquete es que es posible definir una entidad sin esquema. Para ello, es obligatorio que la primera fila del fichero contenga la cabecera de las columnas y no sea necesario especificar claves primarias y/o extranjeras. Si esos requisitos se cumplen, se extraen automáticamente los nombres de las columnas y se supone que todos los campos son de tipo cadena de texto no nula.

3.2.2. Descripción de las consultas

En este apartado se explica como se ha diseñado el paquete `operations`, encargado de definir las consultas sobre el conjunto de datos que expondrá el contenedor. Es importante mencionar que las consultas se especifican sobre una entidad, es decir, no se definen a nivel global. De este modo, cada entidad cuenta con sus propias consultas. Las operaciones de consulta propuestas soportan la búsqueda por único atributo. Por ejemplo, suponiendo una entidad como la mostrada en la prueba de concepto donde se almacena información sobre películas, una consulta sería: *Obtener todas las películas publicadas en el año 2021*. No se permite una consulta como: *Obtener todas las películas publicadas en el año 2021 y del director X*. De nuevo, se trata de una funcionalidad que podría ser parte de una segunda fase de desarrollo de este proyecto. Respecto a detalles más específicos del diseño, para cada una de las operaciones de consulta se permiten configurar los siguientes parámetros:

- Nombre de la consulta.
- Atributo por el que se realiza la búsqueda.
- Indicar si se desea que soporte ordenación de resultados y/o paginación.
- Especificar, en caso de ser posible, si los resultados se deben ordenar de forma ascendente o descendente según un atributo de la entidad.
- Limitar el número de resultados devueltos

- Especificar si los resultados devueltos deben ser únicos.
- Indicar las interfaces en las que se expone esta operación.

Respecto al diseño arquitectural del módulo, se ha creado teniendo en cuenta que para cada consulta, puede darse el caso de que sea necesario generar la configuración para tres interfaces distintas, REST API, *GraphQL* y *gRPC*. A continuación, se muestra un ejemplo de como se especifica una operación, sobre una entidad previamente definida, que soporta paginación, devuelve el resultado ordenado ascendentemente y está expuesta en las interfaces gRPC y REST API. Cabe destacar que, en caso de que no se especifique ninguna interfaz al definir la operación, se expone por defecto en las tres. El resto de parámetros no tienen ningún valor por defecto.

```
operations {
    create<Query>(<query_name>) {
        parameters = L[<var_name>, Pageable(), Sort()]
        sorted = asc(<var_name>)
        limit = first(<integer>)
        distinct = <boolean>
        platforms = L[grpc,rest]
    }
}
```

Listado 6: Definición de una operación que soporta paginación y ordenación de resultados y se expone en gRPC y REST API

Como se puede comprobar en el fragmento de código superior, se crea una operación del tipo `Query`. Esta actúa como una plantilla para definir consultas compatibles con la librería *Spring Data JPA*, que es la interfaz utilizada para dar soporte al acceso a datos desde la aplicación.

Pese a que en la implementación actual únicamente se cuenta con un tipo de consulta soportada, se ha optado por un diseño modular que facilite en un futuro añadir nuevos tipos de consultas, por ejemplo, aquellas que tienen asociada una sentencia SQL. Esto se puede ver reflejado en la interfaz `DefaultQueryBuilder` del diagrama de paquetes y clases del módulo (Figura 8), cuya finalidad es definir los atributos y funcionalidad base que deben tener todos los tipos de consultas definidas y que puede ser implementada de diferentes formas.

3.2.3. Especificación de las interfaces de acceso

En el siguiente apartado se describe el diseño desarrollado en el paquete `modules`. Este permite especificar a través de que interfaces se va a exponer cada una de las entidades. En relación con el módulo `operations`, para que una consulta sea accesible desde una interfaz concreta, es necesario que la entidad sobre la que se realiza la consulta haya sido declarada específicamente como accesible a través de esa interfaz. En caso contrario, la operación no se expondrá a través de esa interfaz.

Al igual que con las consultas, se ha aplicado un diseño que facilita la incorporación de nuevas interfaces. En particular, para las que se encuentran incorporadas al proyecto, REST API, GraphQL y gRPC se permiten los siguientes parámetros de configuración:

- **REST API**: Nombre y ruta donde se expone la interfaz.
- **GraphQL**: Nombre del controlador que implementa la vista *GraphQL* de los datos.
- **gRPC**: Nombre del servicio que expone la vista RPC de los datos.

En el fragmento de código inferior, se muestra un ejemplo donde, para una entidad definida previamente, se indica que se va a exponer a través de las tres interfaces y se configura su implementación.

```
modules {
    install(restApi) {
        collectionResourceRel = <collection_name>
        path = <path_name>
    }
    install(graphqlApi) { controllerName = <controller_name> }
    install(grpcApi) { serviceName = <service_name> }
}
```

Listado 7: Definición de las interfaces que exponen una entidad

3.2.4. Descripción de la construcción

El objetivo de este apartado es mostrar un ejemplo donde se describa la estructura de construcción de un contenedor y se haga uso de forma conjunta de toda la funcionalidad comentada en el módulo `dsl`. Cabe destacar que, gracias a la implementación desarrollada por *Kotlin*, la creación del DSL se deriva directamente de la definición de las clases comentada anteriormente. Es por ello por lo que se ha decidido incorporar en esta sección la descripción de construcciones. A continuación, se muestra un fragmento de código con la estructura en concreto.

```
dac {
    name        = <...>
    fullName    = <...>
    desc        = <...>
    license     = <...>
    homepage    = <...>
    packageName = <...>
    mainClass   = <...>
    buildDir    = <...>
    data {
        files {
            <entityName> {
                src = <file_path>
                schema { ... }
                operations { ... }
                modules { ... }
            }
            <otherEntityName> { ... }
        }
    }
}
```

Listado 8: Ejemplo de definición de la construcción de un contenedor

Como se puede observar, la descripción comienza con la invocación a la función `dac`, esta inicializa todos los objetos necesarios para configurar un contenedor y antes de finalizar su ejecución, invoca al módulo `core` con la configuración que ha recibido y procesado. En el cuerpo de la función, se asigna un valor a las siguientes variables:

- **name**: Nombre de la imagen de Docker generada con el proyecto.
- **fullName**: Nombre de la fuente de datos almacenada.
- **desc**: Descripción de la fuente de datos almacenada.
- **license**: Licencia de la fuente de datos almacenada.
- **homepage**: Enlace al sitio donde se ha obtenido la fuente de datos.
- **packageName**: Nombre del paquete donde se crea el proyecto en Kotlin encargado de exponer la fuente de datos.
- **mainClass**: Nombre de la clase principal del proyecto generado.
- **buildDir**: Directorio donde se genera el proyecto

Es obligatorio asignar un valor para los campos `name`, `packageName` y `mainClass`. En el caso de que el campo `buildDir` quede sin valor, se le asigna uno por defecto. El resto de valores no es necesario que tengan un valor asignado siempre. Una vez definidas estas variables, se procede a especificar los datos almacenados en el contenedor. Para ello, se invoca a la función `data`. Seguidamente, y como solo se soporta el uso de ficheros CSV, se llama a la función `files` para indicar cuáles son esos ficheros. Esto se realiza de la siguiente forma, primero, se señala el nombre de la entidad que se generará al procesar el fichero. Después, mediante la variable `src`, se marca la ruta hasta el fichero en concreto y posteriormente se invoca a las funciones `schema`, `operations` y `modules`. Como la estructura de estas últimas ya ha sido descrita anteriormente, se han obviado en este caso. A modo de ejemplo se muestra también como se definiría una segunda entidad, que se configuraría de la misma manera que la anterior.

3.3. Construcción del contenedor

El módulo `core` es el encargado de generar el proyecto ejecutado en el contenedor. Es invocado por el módulo `dsl`, que lo configura según la descripción que ha sido extraída del script. Basándose en esta configuración y en el uso de plantillas, genera la estructura necesaria. En la Figura 9 se muestra el diagrama de clases y paquetes del módulo. Al igual que en secciones anteriores, se ha optado por mostrar una vista simplificada del mismo para facilitar su comprensión y explicación.

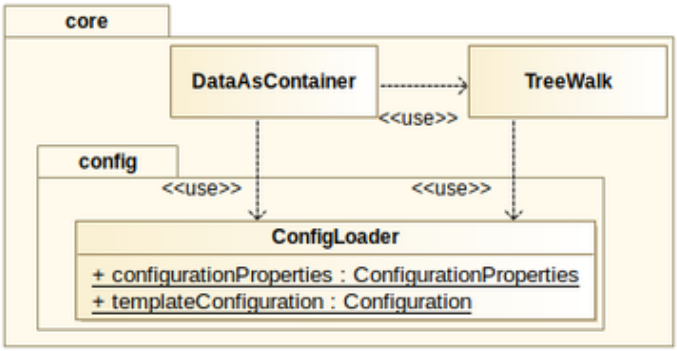


Figura 9: Diagrama de clases y paquetes del módulo core

Se ha definido una clase denominada `DataAsContainer`. Esta se comporta como el punto de entrada al módulo. Recibe la configuración de la construcción del contenedor desde el módulo `dsl` e invoca a la clase `TreeWalk` para que construya el proyecto ejecutado en el contenedor según esa configuración. Además, antes de terminar su ejecución, se encarga de poner en funcionamiento el contenedor generado.

De este modo, la clase `TreeWalk`, bajo las órdenes de la clase `DataAsContainer`, es quien construye realmente el contenedor. Para ello, hace uso de una estructura de ficheros y directorios previamente definida, que cuenta con una serie de plantillas que son completadas según la configuración recibida desde el módulo `dsl`. El paquete `config` expone dos objetos estáticos, uno con todas las constantes utilizadas a lo largo del módulo y otro con la configuración necesaria para utilizar las plantillas. Antes de finalizar con este módulo, cabe destacar que el diseño aplicado soporta múltiples arquitecturas de contenedores. La herramienta debe ser capaz de detectar la arquitectura del sistema donde se está ejecutando y construir un contenedor que sea compatible con la misma. En la Subsección C.1 se explica en detalle la estructura de directorios y ficheros creada, así como, el funcionamiento del motor de plantillas seleccionado para esta tarea.

3.4. Conclusiones de la fase de diseño

El desarrollo de la fase de diseño puso de manifiesto la importancia de todo el trabajo realizado durante el periodo de análisis y permitió confirmar que el desarrollo de un proyecto multi-módulo es la opción más viable. Ha sido posible diseñar con mayor independencia cada uno de los cuatro módulos y el resultado garantiza la mantenibilidad y la facilidad para incorporar nueva funcionalidad en un futuro. Durante esta fase también se ha terminado de consolidar que *Kotlin* es el lenguaje idóneo para este desarrollo, gracias a la simplicidad con la que se pueden definir DSL y ejecutar scripts. Respecto al diseño de los propios módulos, destaca `dsl` porque se ha podido plasmar el razonamiento lógico seguido para idear la solución en el diseño físico y `core` por la separación clara de tareas entre las clases `DataAsContainer` y `TreeWalk`, que favorece la transformación de los datos de configuración recibidos desde el módulo `dsl` en un contenedor *Docker*.

4. Desarrollo

En esta sección se describen algunos aspectos del proceso de desarrollo del sistema. En particular, se comentan detalles de la generación de la página de documentación para cada contenedor, la gestión de las variables de configuración y la creación de un entorno de ejecución. Finalmente, se explican las principales dificultades encontradas durante esta fase. En el Apéndice C se comentan detalles de implementación de la ejecución de scripts y el motor de plantillas.

4.1. Documentación del contenedor

El siguiente apartado comenta la documentación generada en formato HTML para cada construcción. Esta se incluye en el proyecto lanzando en el contenedor y se puede acceder desde cualquier navegador. Para el aspecto visual y la organización de la página, se tomó como referencia el ejemplo propuesto por la agencia del gobierno de los Estados Unidos, General Services Administration para la documentación de API [27].

Los menús de la página de documentación desarrollada son:

- **Overview**: Esta sección actúa como página de inicio. Incluye una breve descripción de la información almacenada en el contenedor. Es aquí donde se muestran las variables configuradas en el script para especificar aspectos como la licencia, nombre de los datos, etc. Además, se muestra información sobre el contenido de las otras dos secciones, API calls y Field reference.

- **API calls**: Esta sección permite explorar las interfaces de acceso a datos de forma interactiva. En el caso de *GraphQL*, se utiliza un entorno de desarrollo denominado *GraphiQL* [28]. Este se despliega en el mismo contenedor y permite descubrir que consultas son accesibles a través de esa interfaz y ejecutar las que se desee. Para probar REST API, se ha empleado la especificación *OpenAPI 3* [29] para describir el servicio y la interfaz gráfica de *Swagger-ui* [30] para generar de modo visual esta documentación y poder enviar peticiones de prueba. Todo ello se ha efectuado mediante anotaciones a través de la librería *springdoc-openapi* [31], que permite automatizar este proceso. Finalmente, para *gRPC* ha sido necesario crear un controlador adicional que permite transformar peticiones HTTP en invocaciones al servicio *gRPC* y devuelve el resultado de las mismas en formato JSON. El motivo de esta decisión se explica más adelante.

- **Field reference**: Esta sección se encarga de mostrar para las diferentes entidades almacenadas en el contenedor, el nombre y el tipo de cada uno de sus atributos.

4.2. Gestión de la configuración

En este apartado se explica como se ha gestionado la definición de todas las constantes empleadas en el módulo `core`. Se ha optado por trabajar con la notación *Human-Optimized Config Object Notation* (HOCON) [32]. Esta permite asignar desde un fichero de configuración externo valores para atributos de objetos de *Kotlin*. La arquitectura desarrollada para dar soporte a este modelo se encuentra en el paquete `config` del diagrama de clases y paquetes mostrado anteriormente (Figura 9). Se ha trabajado con la librería *config4k* [33], que incorpora el soporte para a esta notación en

Kotlin. En particular, se ha empleado para asignar, desde un fichero con la extensión conf, valores para los atributos de la clase `ConfigurationProperties`. Por otro lado, la clase `ConfigLoader` se encarga de cargar esos valores en un objeto estático que se expone al resto de clases del módulo para que accedan a las constantes que necesiten.

4.3. Entorno de ejecución

En este apartado se explica el entorno de ejecución que se ha creado para facilitar el uso de la herramienta. Suponiendo que el proyecto ha sido compilado y se ha generado su ejecutable. Es posible definir un directorio cualquiera, que su a vez cuente con: un fichero de *Kotlin* con el nombre `build.diac.kts` donde se defina la construcción del contenedor con el DSL y un subdirectorio denominado *data* donde se encuentren los ficheros CSV necesarios. De este modo, asumiendo que existe un alias hasta la ruta del ejecutable y se ha generado la estructura comentada, basta con invocar al binario, sin especificar ningún argumento, desde el directorio creado para que se genere dentro del directorio un nuevo subdirectorio denominado *build* con el código fuente del proyecto y se lance el contenedor de forma automática.

4.4. Dificultades encontradas

Configuración del contenedor. Respecto al módulo `dsl`, uno de los obstáculos detectados estuvo relacionado con el paso de toda la información extraída del script al módulo `core`. Como se ha comentado anteriormente, el módulo `dsl` define el lenguaje de descripción de contenedores y, basándose en la información que recibe desde el script, genera una configuración u otra. El problema surge para generar la estructura de datos adecuada para enviar esta configuración al módulo `core` en una forma que facilite su posterior uso en el motor de plantillas. Como solución, se decidió transformar las clases creadas para definir el `dsl` y su configuración en una tabla hash como la que utiliza el motor de plantillas para sustituir las variables. Para una explicación detallada del motor de plantillas se puede consultar el anexo C.1. La conversión se realizó con la librería *Jackson* [34], especializada en ese tipo de transformaciones.

Cliente para el servicio gRPC. Se encontraron también dificultades para enviar peticiones al servidor *gRPC* desde la página de documentación. Durante todo el desarrollo del proyecto, se había trabajado con la herramienta *grpcurl* [35], ya que abstrae la complejidad de la interfaz. El problema surgió cuando se intentó conectar con este servicio a través de una petición HTTP, debido a que no es viable desplegar un cliente *gRPC* junto con el propio contenedor. Ante este problema, se optó por, para cada consulta expuesta a través del servicio *gRPC*, crear su homóloga en la interfaz REST API. La implementación se realiza con la operación de tipo POST para así emular el cuerpo del mensaje en el mismo formato que en las peticiones *gRPC*. De este modo, y con la descripción del servicio que se ofrece en la página de documentación, es posible a través de *Swagger-ui* enviar peticiones POST que se comunican directamente con el servicio *gRPC*. Internamente, las peticiones recibidas en formato JSON se transforman mediante un proceso de *marshalling* a su correspondiente estructura en *Protocol Buffers*. Seguidamente, se invoca al *stub* del servicio con la petición recibida y se espera hasta obtener el resultado para devolverlo en formato JSON. El proceso de *marshalling* se realiza con la librería *protobuf-jackson* [36].

5. Validación

En esta sección se describe como se ha desarrollado la validación del sistema. En particular, se han ejecutado pruebas con conjuntos de datos con una y varías tablas. Las imágenes de los contenedores generados durante esta fase de validación pueden ser descargadas para su uso en el repositorio de Docker Hub `https://hub.docker.com/r/776012/diac`. Solo se han publicado las correspondientes con la arquitectura *amd64*. La decisión de utilizar esta arquitectura se explica en el anexo C.1

5.1. Catálogo de Netflix

En este apartado se explica la prueba realizada con un conjunto de datos con una única tabla. El objetivo de este primer experimento es verificar que el proyecto construido de forma manual en la prueba de concepto se puede replicar únicamente describiendo la construcción en un script. Por ello, se utilizan de nuevo los datos sobre el catálogo de títulos de Netflix. Estos se pueden visualizar en el apartado de desarrollo de la prueba de concepto 2.2. Para la realización de la prueba se ha creado un directorio, *netflix_test*, que contiene el script, `build.diac.kts`, y el subdirectorio *data* con el fichero CSV de los datos. Tanto el script desarrollado para esta prueba, la documentación generada y las consultas expuestas en las tres interfaces, se pueden consultar en el Anexo D. A continuación, se procede a describir el contenido del script de descripción del contenedor.

Inicialmente, se define información básica del contenedor y del conjunto de datos almacenado con las variables específicas. Seguidamente, se determina la entidad que se va a crear para almacenar el contenido. Para ello, se fija el nombre de `title` y se indica la ruta hasta el fichero CSV. Como se trabaja desde el directorio descrito anteriormente, la herramienta se encarga de buscar en *data* para encontrar el fichero. Una vez definida la información básica de la entidad, se comienza a definir el esquema de la misma. En este caso, se trabaja únicamente con atributos del tipo cadena de texto. Es por eso que todos los atributos son del tipo Text. Además, `show_id` actúa como clave primaría de la entidad.

A continuación, se procede a explicar las operaciones creadas:

- **findByReleaseYear**: Consulta que devuelve los diez primeros títulos distintos ordenados ascendentemente por su nombre y que han sido publicados en un año concreto. Soporta paginación y se expone en las interfaces *gRPC* y REST API.

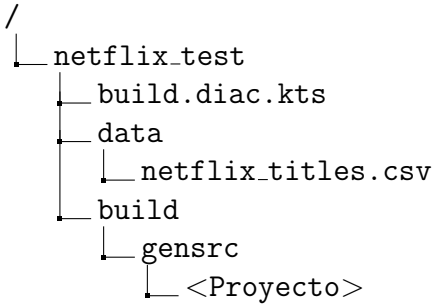


Figura 10: Directorio utilizado para la ejecución de la prueba del catálogo de Netflix

- **findByDirector**: Consulta que devuelve todos los títulos de un director concreto. Al no especificar ninguna plataforma, se expone en las tres interfaces, REST API, *gRPC* y *GraphQL*.
- **findByType**: Consulta que devuelve los veinticinco primeros títulos distintos ordenados descendentemente por el año en él fueron publicados y que son de un tipo concreto (película, serie, etc.). Se expone en las interfaces *GraphQL* y REST API.
- **findByShowId**: Consulta que devuelve un título por su identificador. Expuesta en las tres interfaces.
- Aunque no se especifique en el script, para todas las entidades se genera automáticamente la operación que devuelve todas las entradas de una tabla. En este caso devuelve todos los títulos, sin posibilidad de filtrar la respuesta y se expone en todas las interfaces que se especifiquen en la sección de módulos.

Finalmente, la especificación de los módulos. En esta prueba se instalan las tres interfaces. Como se puede apreciar en el anexo, no se fijan valores para todas las variables que se permiten configurar en cada interfaz. Si esto ocurre, aquellas que han quedado sin designar son asignadas automáticamente según el nombre que se ha especificado para la entidad. Para este contenedor, se ha fijado la ruta *title* para la interfaz REST API. De esta forma se podrá acceder en http://container_ip:8080/title.

En la interfaz *GraphQL* se detalla el nombre para el controlador encargado de gestionar las peticiones recibidas. Por último, para *gRPC* no se especifica ningún valor de configuración, por lo que el nombre del servicio será `TitleService`.

5.2. Alojamientos y reseñas de Airbnb

Este apartado se emplea para comentar la prueba ejecutada con un conjunto de datos formado por dos tablas con una relación entre ellas. Se ha utilizado información publicada por la empresa Airbnb, especializada en el alquiler de alojamientos. En particular, se trabaja con los datos sobre alojamientos de la ciudad de Barcelona y las reseñas de los mismos [37]. Toda la información ha sido publicada bajo la licencia *CC0 1.0 Universal Public Domain Dedication*.

El modelo de datos de la información almacenada cuenta con dos entidades, *listing* y *review*. La primera se corresponde con todas las publicaciones de alojamiento de Barcelona en el último cuarto de año. Para cada una se almacena su nombre, identificador del casero/a, identificador de la zona donde se encuentra, nombre del barrio, tipo de alojamiento, precio de una noche, número mínimo de noches que se debe alquilar y número de reseñas que ha recibido. La segunda representa una reseña emitida para un alojamiento en concreto. En este caso, únicamente se almacena la fecha en la que se realizó y la clave extranjera del alojamiento reseñado. Un alojamiento puede tener un número indeterminado de reseñas. Cabe destacar que, por no sobrecargar el sistema en exceso, ya que el número de reseñas supera las seiscientos mil, se ha eliminado el campo donde se guarda el mensaje de la reseña, puesto que algunas son extremadamente largas y no aportan nada útil al experimento. Respecto al entorno de ejecución, exceptuando el subdirectorio *data*, donde es necesario guardar los ficheros correspondientes con las dos entidades, la estructura del directorio para esta prueba debe ser idéntica a la comentada en la prueba con una única tabla.

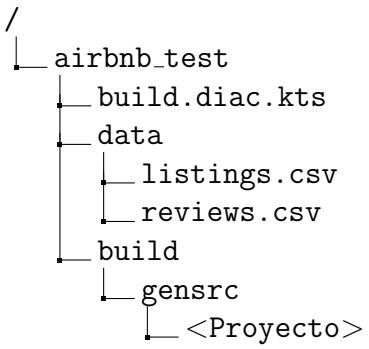


Figura 11: Directorio utilizado para la ejecución de la prueba de alojamientos y reseñas de Airbnb

El modelo de datos de la información, el script desarrollado para esta prueba, la documentación generada y las consultas expuestas en las tres interfaces se pueden consultar en el Anexo E. Respecto al contenido del script `build.diac.kts`, la principal diferencia con la prueba de una única tabla se encuentra en la declaración de las entidades. La declaración de las variables con la información básica del contenedor se realiza igual, aunque cambiando el contenido.

De la misma forma que ocurre en el estándar SQL, cuando se trabaja con tablas con una relación entre ellas. Se debe dejar para el final la declaración de aquella que tiene como atributo la clave extranjera. De este modo, en el script de la prueba, primero se define la entidad `listing` y por último `review`. A continuación se procede a explicar la estructura y consultas de cada una de ellas.

A diferencia de la prueba con los títulos de Netflix, donde se trabaja tan solo com atributos de tipo cadena de texto. `Listing` hace uso del tipo entero para los atributos `listing_id`, `host_id`, `price`, `minimum_nights` y `number_of_reviews`. Además, define una consulta, `findByPrice`, que devuelve todos los alojamientos que tienen un precio determinado y se expone en las tres interfaces, REST API, GraphQL y gRPC.

Por otro lado, `review`, cuenta con la definición de la clave extranjera. En este caso se crea una consulta, expuesta en las tres interfaces, que devuelve una reseña según el identificador indicado. Como se ha comentado en la sección de desarrollo, esta clave extranjera se transforma internamente en un objeto de la clase `Listing`. Así, es posible acceder a toda la información del alojamiento al que hace referencia una reseña sin la necesidad de ejecutar una segunda consulta.

5.3. Conclusiones de la validación

Con el desarrollo de estas dos pruebas se ha verificado que la herramienta cumple con los objetivos marcados al comienzo del Trabajo Fin de Grado. Es capaz de construir y desplegar contenedores *Docker* que exponen consultas sobre modelos de datos simples en las interfaces definidas. Esta fase ha puesto de manifiesto que el sistema creado para ejecutar la herramienta, con la creación de un directorio que cuente con el script de configuración y los datos, reduce la dificultad de uso de la misma y facilita trabajar con ella. Además, ha resultado importante la verificación de que se soporta correctamente el uso de claves extranjeras, ya que su implementación supuso un salto importante de dificultad en el desarrollo de la solución.

6. Conclusiones

6.1. Objetivos alcanzados

Se considera que el desarrollo del Trabajo Fin de Grado ha cumplido con los objetivos establecidos al comienzo del mismo. La herramienta creada, pese a no contar con toda la funcionalidad de la que podría disponer en un entorno de producción, supone una buena base sobre la que continuar trabajando y permite distribuir y desplegar conjuntos de datos con modelos simples en entornos heterogéneos. El DSL desarrollado para definir la construcción del contenedor simplifica en gran parte este proceso y permite que el usuario se limite a configurar el contenedor. La validación realizada demuestra que el concepto tiene un gran potencial y realmente simplifica la distribución y acceso a los datos. Además, ha permitido comprobar que la distribución de los contenedores a través de plataformas como *Docker Hub* es una opción viable y que puede ser explotada en el futuro. El esfuerzo dedicado para alcanzar estos objetivos puede consultarse en el Apéndice A.

6.2. Trabajo futuro

De cara a una posible continuación de la herramienta, se presentan una serie de propuestas de trabajo:

- Incluir soporte para consultas más complejas. Actualmente, solo se pueden realizar búsquedas según el valor de un único atributo.
- Capacidad para inferir automáticamente los tipos de datos de los atributos declarados en un fichero, ya que la implementación actual supone que todos son cadena de texto si no se indica el esquema.
- Posibilidad de definir más propiedades de los atributos de las entidades. Por ejemplo, rangos de valores aceptados o condiciones de mayor complejidad. El objetivo sería replicar en la medida de lo posible la definición de tablas del estándar SQL.
- Soportar la carga de datos desde otros formatos de almacenamiento. Incluyendo la posibilidad de acceder a recursos publicados en un servicio remoto.
- Definir en un mismo contenedor diferentes conjuntos de datos. Aunque en el estado actual se permite trabajar con varias tablas, se trata de información relacionada. Una posible mejora podría permitir almacenar, y exponer, conjuntos de temática distinta.
- Ofrecer una interfaz gráfica para configurar la descripción del contenedor sin la necesidad de codificar directamente el script. De este modo, la herramienta podría ser empleada por un mayor número de usuarios.
- Añadir, si fuera necesario, el soporte para otras interfaces de acceso.

6.3. Reflexiones personales

En lo personal, este Trabajo Fin de Grado me ha permitido poner en práctica muchos de los conocimientos adquiridos durante todo el grado. Además, he podido trabajar con un buen abanico de tecnologías y herramientas muy interesantes. Algunas de las cuales nunca había utilizado, y pienso que su conocimiento puede resultar de gran utilidad en mi futuro profesional. Como cualquier trabajo académico de estas características, han existido dificultades a lo largo del desarrollo. Principalmente relacionadas con la incorporación al proyecto de tecnologías o técnicas desconocidas. Por último, creo que una de las claves detrás de este Trabajo Fin de Grado es la forma en la que se ha organizado su desarrollo. Definiendo tareas muy específicas y de no gran tamaño. Esto redujo la dificultad durante las fases del desarrollo y además ha permitido llevar un buen control de todo el proyecto.

Acrónimos

API Application programming interface.

CSV Comma-separated values.

DSL Domain-specific language (en castellano, Lenguaje específico de dominio).

HOCON Human-Optimized Config Object Notation.

HTTP Hypertext Transfer Protocol.

ISBN International Standard Book Number.

JAR Java Archive.

JPA Java Persistence API.

JSON JavaScript Object Notation.

REST Representational State Transfer.

RPC Remote Procedure Call.

SQL Structured Query Language.

Glosario

Docker Se trata de una plataforma para desplegar aplicaciones en entornos heterogéneos. Fue publicada en el año 2013 bajo la licencia Apache License 2.0. Docker trabaja con contenedores, que se corresponde con software empaquetado junto a todas sus dependencias. Estos contenedores se ejecutan sobre el sistema operativo en un entorno aislado y pueden ser desplegados en diferentes máquinas que soporten Docker [38].

GraphQL Lenguaje de consultas orientado a dar soporte al acceso a datos en aplicaciones clientes-servidor. Desarrollado internamente por Facebook en el año 2012 y publicado en el 2015, cuenta con su propio motor para ejecutar las consultas y soporta introspección de tipos desde el cliente [10].

gRPC Tecnología que permite invocar métodos en sistemas distribuidos como si se tratara de una llamada local [11]. Desarrollado inicialmente por Google y publicado en el año 2015, posibilita que un servidor defina un servicio donde se especifiquen los métodos que se pueden invocar remotamente y los parámetros y tipos devueltos. Estos pueden ser accedidos por clientes gRPC mediante peticiones serializadas con *Protocol Buffers* [39].

Lenguaje específico de dominio Lenguaje creado para resolver un problema específico en un dominio concreto. En este Trabajo Fin de Grado se utiliza para describir como se debe construir el contenedor generado.

REST API Interfaz de un servicio web que responde a las peticiones de los clientes según el estilo REST. Definido este último como un estilo arquitectural para sistemas hipermedia distribuidos [9].

Script Fichero que incluye código en un lenguaje de programación que puede ser ejecutado sin necesidad de compilación previa.

Referencias

[1] J Berends y col. *Reusing open data : a study on companies transforming open data into economic and societal value*. European Union, Publications Office, 2020. DOI: doi/10.2830/876679.

[2] Antonio Bello-García. «Datos abiertos y participación en el gobierno social». En: *Economía industrial* 405 (oct. de 2017), págs. 99-111.

[3] Carl Boettiger. «An Introduction to Docker for Reproducible Research». En: *SIGOPS Oper. Syst. Rev.* 49.1 (ene. de 2015), págs. 71-79. ISSN: 0163-5980. DOI: 10.1145/2723872.2723882.

[4] *Docker*. https://www.docker.com. Accedido: 09-06-2022.

[5] *Docker Hub*. https://hub.docker.com. Accedido: 07-06-2022.

[6] *Type-safe builders*. https://kotlinlang.org/docs/type-safe-builders.html. Accedido: 20-06-2022.

[7] *Get started with Kotlin custom scripting – tutorial*. https://kotlinlang.org/docs/custom-script-deps-tutorial.html. Accedido: 26-05-2022.

[8] *Spring*. https://spring.io. Accedido: 09-06-2022.

[9] Roy T Fielding y Richard N Taylor. «Principled design of the modern web architecture». En: *ACM Transactions on Internet Technology (TOIT)* 2.2 (2002), págs. 115-150.

[10] *GraphQL*. https://spec.graphql.org/October2021/. Accedido: 26-05-2022.

[11] *gRPC*. https://grpc.io. Accedido: 08-06-2022.

[12] *Kotlin Programming Language*. https://kotlinlang.org. Accedido: 06-06-2022.

[13] *Python*. https://www.python.org. Accedido: 06-06-2022.

[14] *Java Oracle*. https://www.java.com. Accedido: 06-06-2022.

[15] *Spring Boot*. https://spring.io/projects/spring-boot. Accedido: 06-06-2022.

[16] *SQLite*. https://www.sqlite.org. Accedido: 06-06-2022.

[17] *SQL Features That SQLite Does Not Implement*. https://www.sqlite.org/omitted.html. Accedido: 26-05-2022.

[18] *Spring Data JPA*. https://spring.io/projects/spring-data-jpa. Accedido: 06-06-2022.

[19] *Spring Data REST*. https://spring.io/projects/spring-data-rest. Accedido: 06-06-2022.

[20] Netflix. *GraphQL for Java with Spring Boot made easy*. Ver. 4.9.24. Mar. de 2021. URL: https://github.com/netflix/dgs-framework.

[21] Michael Zhang. *Spring Boot starter module for gRPC framework*. Ver. 2.31.13. Ene. de 2022. URL: https://github.com/yidongnan/grpc-spring-boot-starte.

[22] *Netflix Movies and TV Shows*. https://www.kaggle.com/datasets/shivamb/netflix-shows. Accedido: 01-06-2022.

[23] *Kaggle*. https://www.kaggle.com. Accedido: 07-06-2022.

[24] *Spring Initializr*. https://start.spring.io. Accedido: 07-06-2022.

[25] *Hibernate*. https://hibernate.org. Accedido: 09-06-2022.

[26] gRPC. *The Java gRPC implementation. HTTP/2 based RPC*. Ver. 1.45.0. Mar. de 2022. URL: https://github.com/grpc/grpc-java.

[27] *Example API documentation*. https://gsa.github.io/api-documentation-template/api-docs/. Accedido: 30-05-2022.

[28] GraphQL. *GraphiQL & the GraphQL LSP Reference Ecosystem for building browser & IDE tools*. Ver. 1.7.2. Mar. de 2022. URL: https://github.com/graphql/graphiql.

[29] *Open API 3*. https://swagger.io/specification/. Accedido: 30-05-2022.

[30] *Swagger UI*. https://swagger.io/tools/swagger-ui/. Accedido: 30-05-2022.

[31] *OpenAPI3 library for Spring-boot*. https://springdoc.org/. Accedido: 30-05-2022.

[32] *HOCON*. github.com/lightbend/config/blob/main/HOCON.md. Accedido: 20-06-2022.

[33] Config4k. *A Kotlin wrapper for Typesafe Config*. Ver. 0.4.2. Feb. de 2020. URL: https://github.com/config4k/config4k.

[34] FasterXML. *Jackson Project*. Ver. 2.3.12. Sep. de 2021. URL: https://github.com/FasterXML/jackson.

[35] Fullstory. *Command-line tool for interacting with gRPC servers*. Ver. 1.8.6. Feb. de 2022. URL: https://github.com/fullstorydev/grpcurl.

[36] Curioswitch. *High performance protobuf JSON marshaler based on Jackson*. Ver. 2.0.0. Ene. de 2022. URL: https://github.com/curioswitch/protobuf-jackson.

[37] *Inside Airbnb: Get the Data*. http://insideairbnb.com/get-the-data/. Accedido: 31-05-2022.

[38] Babak Bashari Rad, Harrison John Bhatti y Mohammad Ahmadi. «An introduction to docker and analysis of its performance». En: *International Journal of Computer Science and Network Security (IJCSNS)* 17.3 (2017), pág. 228.

[39] *Protocol buffers*. https://developers.google.com/protocol-buffers. Accedido: 26-05-2022.

[40] *FreeMarker Java Template Engine*. https://freemarker.apache.org/. Accedido: 29-05-2022.

[41] *Gradle Build Tool*. https://gradle.org/. Accedido: 30-05-2022.

[42] *Curl*. https://curl.se. Accedido: 08-06-2022.

Lista de Tablas

Lista de Figuras

Lista de Código

A. Gestión del proyecto

El siguiente apéndice se utiliza para describir como se ha llevado a cabo la gestión del proyecto. En concreto, se detalla el esfuerzo dedicado y el uso del sistema de control de versiones *Github* para organizar todo el desarrollo del Trabajo Fin de Grado. El control de esfuerzos se ha efectuado a través de una hoja de cálculo. Cada día que se trabajaba en el proyecto se registraba el tiempo invertido. El registro se ha organizado según las fases definidas en el cronograma.

- **Crear prototipo**: Incluye la fase de análisis y la puesta en marcha del entorno de desarrollo.
- **Crear la estructura base del proyecto**: Crear proyecto multi-módulo con la funcionalidad básica para crear un contenedor con muy poca configuración.
- **Implementar construcción y despliegue parametrizado**: Desarrollo total del DSL y soporte para la construcción del contenedor según la descripción del script.
- **Validación del sistema**: Incluye la realización de las dos pruebas descritas en este documento, limpieza de código y refactorizado de la arquitectura.
- **Elaboración de documentación y ejemplos**: Redacción de esta memoria y preparación de ejemplo para la presentación.

Fases del trabajo	**Horas**
Crear prototipo	35
Crear la estructura base del proyecto	49
Implementar construcción y despliegue parametrizado	152
Validación del sistema	24
Redacción de la memoria	42
Esfuerzo total	**302**

Tabla 3: Esfuerzo dedicado a cada fase del Trabajo Fin de Grado

Durante todo el proyecto se ha utilizado la plataforma GitHub para contar con un control de versiones y además para organizar las tareas en *issues* donde se definían objetivos y se mantenía una conversación constante entre el director del proyecto y el desarrollador sobre como continuar con el Trabajo Fin de Grado y posibles mejoras al sistema desarrollado. También se han llevado a cabo reuniones presenciales al comienzo de las fases descritas anteriormente para establecer los límites de la siguiente tarea y fijar hasta donde se quería llegar.

B. Estudio de Kotlin

En este anexo se explican una serie de pruebas realizadas durante la fase de análisis para comprobar si *Kotlin* era el lenguaje indicado para el desarrollo del Trabajo Fin de Grado. Todo este proceso ha permitido familiarizarse más con el lenguaje de programación y ha servido como un aprendizaje previo a comenzar con el desarrollo de la solución definitiva.

B.1. Soporte para scripts

Uno de los aspectos analizado durante la fase de análisis, fue la capacidad para ejecutar código *Kotlin* sin compilación previa, como si se tratara de un script. El objetivo era emplear esta técnica para describir la construcción del contenedor. Se trata de una funcionalidad creada por *JetBrains*, desarrolladora oficial de *Kotlin*, pero que todavía se encuentra en fase experimental. Actualmente, no existe una gran cantidad de documentación al respecto, por lo que durante los inicios de esta fase de análisis aparecieron dificultades asociadas precisamente a esa falta de información. Al igual que con la creación del contenedor de ejemplo, se generó un proyecto de prueba para analizar la viabilidad de esta idea. De este desarrollo se extrajeron las siguientes conclusiones:

- Toda aplicación que ejecuta scripts con código Kotlin debe contar con una clase abstracta que actúa como superclase para todos los scripts del mismo tipo. Se consideran del mismo tipo aquellos que tienen el mismo nombre, por ejemplo `example.kts`. Para ello, es necesario crear una clase anotada con la etiqueta `@KotlinScript` donde se definan aspectos específicos de la compilación de ese tipo de scripts [7]. De este modo, es posible añadir una serie de dependencias por defecto o limitar las rutas desde las que se puede ejecutar el script.
- Además de la definición del script, debe implementarse una clase que sea la encargada de evaluar y ejecutar un fichero con código *Kotlin* tomando como referencia la definición previamente creada.

B.2. Definición de un DSL

Por otra parte, con la especificación de un DSL se busca simplificar la creación del contenedor, de modo que no sea necesario un conocimiento avanzado de *Kotlin* para realizar la misma y se pueda llevar a cabo de una forma lo más similar posible al lenguaje natural.

La implementación del DSL se basa principalmente en el uso de funciones anónimas, también conocidas como expresiones lambda. Se comportan como si se tratará de una función de la cual se pasa directamente el cuerpo de la misma, sin especificar su declaración. Además, *Kotlin* permite que estas se puedan utilizar como parámetro de entrada a otras funciones. En particular, es posible invocar a una función que recibe como parámetro una función lambda sin la necesidad de especificar los paréntesis. En el fragmento de código inferior se muestra un ejemplo de esta característica.

`funcion({expresion-lambda})` → Invocación habitual
`funcion{ expresion-lambda }` → Invocación soportada por Kotlin

Esta funcionalidad toma especial importancia si se hace uso de las denominadas expresiones lambda con receptores. Se trata de funciones anónimas que están asociadas a un objeto concreto, es decir, pueden acceder a las funciones o atributos del mismo en el cuerpo de la función lambda.

En el siguiente fragmento de código, se puede apreciar un breve ejemplo donde se hace uso de esta funcionalidad. En concreto, se define una clase `MyType`, que cuenta únicamente con un atributo de tipo `String` y una función, `doSomething`. Seguidamente, se especifica una función, `test`, que devuelve un objeto de tipo `MyType` y recibe como parámetro de entrada una función anónima. Esta última, denominada `init`, se corresponde con una expresión lambda que no devuelve nada y cuenta con el tipo `MyType` como receptor. En el cuerpo de la expresión lambda `init`, será posible acceder a los atributos y funciones de la instancia del tipo `MyType`.

Finalmente, se muestra un ejemplo de su uso en la función `main`. Tal y como se puede observar, basta con invocar a la función `test` para obtener un objeto del tipo `Mytype` con el valor indicado en el atributo `bar` y que además invoque a la función `doSomething`. Cabe destacar que, para aportar mayor contexto al fragmento de código y de forma opcional, el objeto devuelto por la función `test` se guarda en la variable `type` del tipo `myType`.

```
class MyType {
    lateinit var bar : String

    fun doSomething() { ... }
}

fun test(init: MyType.() -> Unit) : MyType {
    return MyType().apply(init)
}

fun main() {
 var type : MyType = test {
    bar = "prueba"
    doSomething()
  }
}
```

Listado 9: Fragmento de código de ejemplo de funciones lambda en *Kotlin*

Mediante el desarrollo de estas pruebas, tanto el proyecto para construir el contenedor comentando en la sección de análisis, como los ejemplos para ejecutar scripts y diseñar un DSL en *Kotlin*, se pudo verificar que la propuesta era viable y *Kotlin* era el lenguaje apropiado para realizarlo. Esta fase, pese haberla alargado más de lo esperado, sirvió para asentar conocimientos sobre *Kotlin*, ya que no se contaba con la experiencia suficiente y creó la base necesaria para comenzar con la fase de desarrollo.

C. Desarrollo de la solución

En este anexo se explica en detalle algunos aspectos de más bajo nivel relacionados con la implementación de la solución. En particular, se describe el motor de plantillas utilizado y como se lleva a cabo la ejecución de scripts en la herramienta.

C.1. Motor de plantillas

Este apartado describe el uso de un motor de plantillas para generar nuevos contenedores basándose en la descripción especificada en el script. Para ello, se combina un árbol de directorios, que cuenta con la estructura necesaria para un proyecto con esos requisitos, con el empleo de plantillas que son completadas con la configuración recibida. La estructura de directorios y plantillas comentada a lo largo de este apartado se puede observar en la Figura 12.

Como ya se ha comentado anteriormente, durante la fase de análisis se generó un proyecto de ejemplo para analizar si era viable construir una aplicación que expusiera un conjunto de datos de la forma requerida. Este ejemplo resultó muy útil en la fase de desarrollo, ya que permitió extraer la estructura de directorios y ficheros que habría que utilizar en el proyecto cuando este se generara de un modo paramétrico y automático. La principal diferencia con la prueba de concepto, es que en este caso no todos los proyectos tienen porque contar con la estructura completa. Es decir, es posible configurar el script de modo que algunas interfaces de acceso no estén disponibles y, por lo tanto, no sea necesario incorporar al proyecto cierta funcionalidad.

Con esta premisa en mente, se decidió organizar la estructura de directorios y ficheros según la interfaz con la que estaban relacionados. De esta forma, se crearon los siguientes directorios base:

- **common**: Contiene la estructura común a todas las interfaces. Es la base del proyecto y la encargada de arrancar la aplicación.
- **graphql**: Ficheros específicos para la interfaz de acceso *GraphQL*.
- **grpc**: Ficheros específicos para la interfaz de acceso *gRPC*.
- **rest**: Ficheros específicos para la interfaz de acceso REST API.

Antes de comenzar con la explicación en detalle de la estructura de directorios, se procede a explicar el uso de plantillas. Como motor de plantillas se ha utilizado *FreeMarker* [40], se trata de una librería de Java, con soporte para *Kotlin*. Permite completar plantillas al combinarlas con objetos de *Kotlin* donde se asocia un valor para las variables definidas en la plantilla. Los ficheros creados como plantillas deben estar codificados con el lenguaje *FreeMarker Template Language* (FTL) y emplear la extensión *ftlh*. En la Figura 13 se muestra un esquema con el funcionamiento del motor de plantillas. Como se puede apreciar, las variables se representan con el símbolo $ seguido del nombre de la misma. En este caso, `example.kt.ftlh` se corresponde con una plantilla de *FreeMarker* que enmascara a un fichero de *Kotlin*. Este último define una variable, `aux`, a la que se le asigna el valor que tenga en ese momento la variable de la plantilla *$value*. De esa manera, si el motor, a la hora de procesar `example.kt.ftlh`, recibe un objeto de tipo tabla hash donde se asocia un valor concreto para la clave

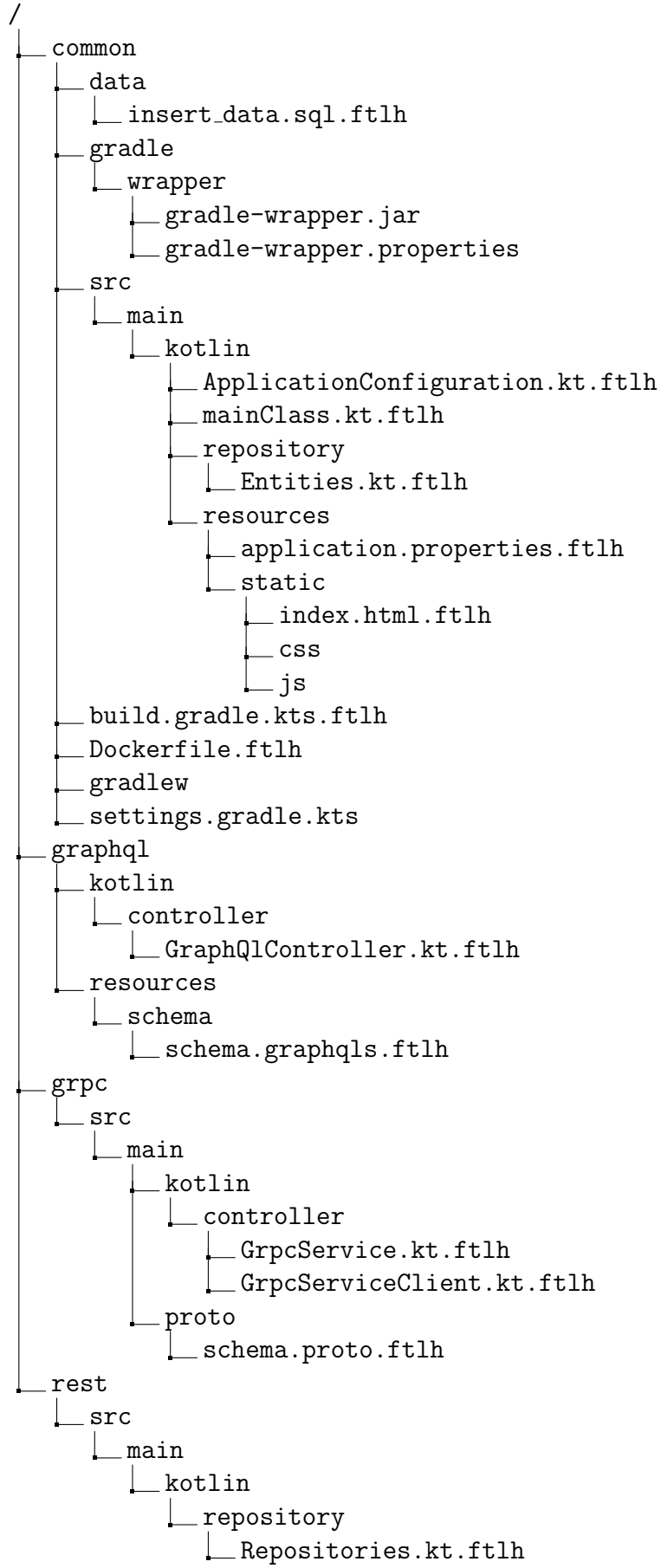


Figura 12: Estructura de directorios para la generación de proyectos

`value`. Sustituirá la variable y generará el fichero `example.kt` en la ruta que se haya indicado en la configuración del mismo.

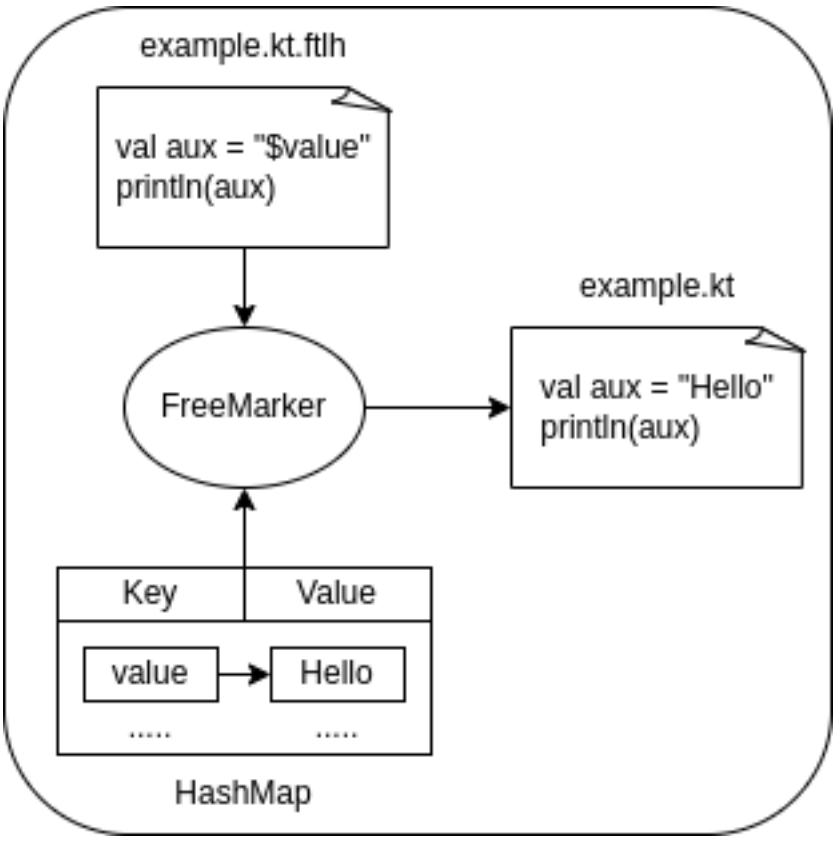


Figura 13: Esquema de funcionamiento de FreeMarker

A continuación, se continúa con la explicación de la estructura de directorios. El directorio *common* contiene:

- **Directorio data**: Incluye la plantilla del script SQL encargado de poblar el fichero de *SQLite* que actuará como base de datos. Además, en este directorio se almacenan los ficheros CSV especificados en la descripción del contenedor.
- **Directorio gradle**: Incluye los ficheros necesarios para incorporar al proyecto la herramienta de automatización *Gradle* [41]. En los proyectos generados se utiliza para definir las dependencias del mismo, así como para crear tareas que permitan de forma automática compilar y desempaquetar los ejecutables, crear y cargar la base de datos y lanzar el contenedor con la imagen generada.
- **build.gradle.kts.ftlh**: Plantilla donde se especifican las dependencias y tareas de gradle del proyecto.
- **DockerFile.ftlh**: Plantilla para crear el documento donde se definen todos los comandos que hay que ejecutar para crear la imagen del contenedor Docker. Todos los contenedores se construyen con la imagen base de Java Runtime Environment 11.0.11_9. Internamente, se configura para que se use una versión compatible con la arquitectura amd64 o arm64-v8a, en función del equipo donde se ejecuta la herramienta. Amd64 se corresponde con la especificación de 64 bits desarrollada

por la compañía AMD del conjunto de instrucciones x86, mientras que arm64 es la especificación de 64 bits de la arquitectura ARM. La elección de estas arquitecturas se debe a que para el desarrollo del Trabajo Fin de Grado se han empleado dos equipos que contaban con una arquitectura distinta y de este modo se podía desplegar de manera directa en ambos sistemas.

- **gradlew**: Script encargado de ejecutar la herramienta *gradle*.
- **settings.gradle.kts**: Se utiliza para definir aspectos de configuración. En este caso fijar el nombre del proyecto.
- **Directorio src**: En esta ruta es donde se almacena el código fuente de la aplicación. La estructura cuenta únicamente con los paquetes `src/main/kotlin`, pero se expande con el nombre de paquete indicado en el script. Es decir, si en la descripción del contenedor se especifica como paquete `com.example.tfg`, la ruta de estos directorios será `src/main/kotlin/com/example/tfg`. El resto de ficheros contenidos en el directorio *src* son los siguientes:
  - **ApplicationConfiguration.kt.ftlh**: Plantilla de la clase donde se configura el controlador necesario para conectar la aplicación con *SQLite*.
  - **mainClass.kt.ftlh**: Plantilla de la clase encargada de arrancar la aplicación.
  - **repository/Entities.kt.ftlh**: Contenida en el directorio repository, se encuentra la plantilla donde se definen las entidades JPA. Estas se corresponden con las definidas en la descripción del contenedor y cada una se asocia con un único fichero CSV.
  - **Directorio resources**: Este directorio almacena, por un lado, la plantilla `application.properties.ftlh`, que se encarga de definir diferentes aspectos de configuración del sistema. Por otra parte, en el subdirectorio static, se encuentra una plantilla de un fichero HTML junto con los ficheros CSS y JavaScript necesarios para crear una página donde, una vez lanzado el contenedor, se pueda consultar documentación sobre las fuentes de datos almacenada en el mismo, así como, probar de forma interactiva las interfaces de acceso. Más adelante, se explica en detalle como se genera todo ello.

El directorio *graphql* contiene las plantillas para definir los controladores y el esquema necesario para dar soporte a la interfaz *GraphQL*. El directorio *grpc* se encarga de lo mismo para la interfaz *gRPC*. En este caso contiene la plantilla para definir el servicio expuesto por el servidor y la plantilla para un servicio cliente. Finalmente, el directorio *rest* contiene la plantilla encargada de definir el repositorio de acceso a datos. Como aspecto destacable, este fichero debe estar presente siempre en el proyecto generado, ya que es ahí donde se definen las consultas que interaccionan con la base de datos directamente. En caso de que la descripción del contenedor indiqué que se debe utilizar REST API, se añadirá a la plantilla la etiqueta correspondiente para añadir esa funcionalidad, como ya se ha comentado en la prueba de concepto. Como se puede apreciar, los directorios y plantillas se han estructurado de la misma forma en la que estarían en el proyecto definitivo, esto se debe a una decisión de implementación que se detalla a continuación.

Tal y como se ha descrito en la sección de diseño. La clase `TreeWalk` del módulo `core` (Figura 9) es quien construye el contenedor según la configuración que le indica la

clase `DataAsContainer`. Para realizar esta tarea, fusiona los directorios necesarios en función de los requisitos del proyecto y rellena las plantillas con los valores adecuados. De esta forma, según las interfaces de acceso que hayan sido especificadas en el script de construcción, se copian unos u otros ficheros y directorios de la estructura base en el proyecto generado. Para acceder estos ficheros, recorre la estructura de directorios copiando y completando plantillas con las variables que se han recibido desde el módulo `dsl`. De la implementación de la clase `TreeWalk`, es de donde aparece la necesidad de declarar las plantillas con todos los directorios donde están contenidas. Esto se debe a que se realiza un recorrido en profundidad del árbol y se extrae la ruta de cada nodo para así copiarlo directamente en la localización del proyecto generado.

C.2. Ejecución de scripts

A continuación, se detallan aspectos de la implementación de los módulos `host` y `script`, comentados en la sección de diseño 3. Al estar muy relacionados entre sí, se ha optado por agruparlos en un único apartado. Cabe destacar que los scripts a los que se hace referencia a lo largo de este documento, se corresponden con ficheros de la extensión *kts* escritos con el DSL desarrollado en este Trabajo Fin de Grado.

Como se ha comentado previamente, el módulo `host` es el encargado de ejecutar los scripts. Para su desarrollo, se ha trabajado con la librería *kotlin.script*, que, en la fecha de desarrollo del proyecto, todavía se encuentra en estado experimental. Por otra parte, el módulo `script` permite especificar aspectos como la extensión de los ficheros que puede ejecutar el módulo `host`, en este caso `build.diac.kts`, y las clases que se van a utilizar para evaluar y compilar el script. Esto último se ha empleado para indicar las dependencias que se importan por defecto del módulo `dsl` al ejecutar el script. De este modo, no es necesario conocer las clases usadas a la hora de describir una construcción y el usuario puede limitarse a codificar la especificación del contenedor.

D. Prueba catálogo de Netflix

En este anexo se muestra todo el contenido relacionado con la prueba ejecutada con el catálogo de *Netflix*. La descripción de la misma puede consultarse en la Subsección 5.1. En el Listado 10 se puede apreciar el contenido del script `build.diac.kts`. Este se utiliza para describir la construcción del contenedor.

```
dac {
    name = "Demo"
    fullName = "Demo application"
    desc = "Contenedor con informacion sobre catalogo de Netflix"
    license = "Apache-2.0"
    homepage = "https://www.kaggle.com/shivamb/netflix-shows"
    packageName = "com.simplificada.tfg"
    mainClass = "SimplificadaTest"
    data {
        files {
          "title" {
           src = "netflix_titles.csv"
            schema {
                "show_id" type text property isPrimaryKey
                "type" type text
                "title" type text
                "director" type text
                "cast" type text
                "country" type text
                "date_added" type text
                "release_year" type text
                "rating" type text
                "duration" type text
                "listed_in" type text
                "description" type text
            }
            operations {
                create<Query>("findByReleaseYear") {
                  parameters = L["releaseYear", Pageable(), Sort()]
                  sorted = asc("title")
                  limit = first(10)
                  distinct = true
                  platforms = L[grpc,rest]
                }
                create<Query>("findByDirector", "director")
                create<Query>("findByType",
                              "type",
                              desc("releaseYear"),
                              first(25),
                              true,
                              L[graphql,rest])
                create<Query>("findByShowId"){parameters="showId"}
            }
            modules {
                install(restApi) { path = "title" }
                install(graphqlApi) { controllerName = "title" }
                install(grpcApi) {}
            } } } } }
```

Listado 10: Script con la descripción del contenedor para la prueba del catálogo de Netflix

En el Listado 11 puede observarse el código generado, de forma automática y según el script del Listado 10, para definir la interfaz *GraphQL* en el contenedor. Se definen las consultas expuestas en la interfaz en el tipo `Query` y posteriormente se define la entidad con la que se trabaja, en este caso, `TitleEntity`.

```
type Query {
  allTitles : [TitleEntity]
  findByDirector(director : String) : [TitleEntity]
  findDistinctFirst25ByTypeOrderByReleaseYearDesc(type : String)
     : [TitleEntity]
  findByShowId(showId : String) : [TitleEntity]
}
type TitleEntity {
    showId : String
    type : String
    title : String
    director : String
    cast : String
    country : String
    dateAdded : String
    releaseYear : String
    rating : String
    duration : String
    listedIn : String
    description : String
}
```

Listado 11: Descripción del esquema de GraphQL para la prueba del catálogo de Netflix

El Listado 12 muestra un ejemplo de como se codificaría cada una de las consultas expuestas en *GraphQL*. Cabe destacar el caso de la consulta `allTitles`, que devuelve todos los títulos del catálogo y ha sido creada de forma automática. Sin necesidad de especificarlo en el script. Esto se expande también a REST API y *gRPC*.

```
{
  allTitles {
    type
  }
  findByDirector(director : "Quentin Tarantino") {
    releaseYear
    cast
  }
  findDistinctFirst25ByTypeOrderByReleaseYearDesc(type:"TV Show"){
    rating
    duration
   }
  findByShowId(showId : "s108") {
    title
    country
    description
  }
}
```

Listado 12: Consultas expuestas a través de GraphQL para la prueba del catálogo de Netflix

En el Listado 13 puede observarse una petición de ejemplo para cada una de las consultas expuestas en REST API. Aunque estas pueden ser ejecutadas de una forma más sencilla desde la página de documentación, gracias a *swagger-ui*, se han decidido mostrar también con la herramienta *curl* [42].

```
curl -X GET http://localhost:8080/title/search/findByDirector\
    ?director=Martin%20Scorsese

curl -X GET http://localhost:8080/title

curl -X GET http://localhost:8080/title/search/findByShowId?showId=s190

curl -X GET http://localhost:8080/title/search/\
    findDistinctFirst10ByReleaseYearOrderByTitleAsc?releaseYear=2020

curl -X GET http://localhost:8080/title/search/\
    findDistinctFirst25ByTypeOrderByReleaseYearDesc?type=Movie

curl -X GET http://localhost:8080/title/s305
```

Listado 13: Ejemplo de peticiones REST API para la prueba del catálogo de Netflix

A continuación, en el Listado 14, se muestra la definición del servicio expuesto en *gRPC*. De nuevo, todo el código mostrado se ha generado de forma automática según la descripción del script. La estructura es muy similar a la del esquema de *GraphQL*. Primero, se definen las consultas y los mensajes necesarios para invocarlas, y después, se especifica la entidad con la que se trabaja. En este caso `Title`.

```
syntax = "proto3";

package proto;

import "google/protobuf/wrappers.proto";
import "google/protobuf/timestamp.proto";
import "google/protobuf/struct.proto";

service TitleService {
 rpc findDistinctFirst10ByReleaseYearOrderByTitleAsc
   (findByReleaseYearRequest) returns (stream Title);
 rpc allTitles(allTitleRequest) returns (stream Title);
 rpc findByDirector(findByDirectorRequest) returns (stream Title);
 rpc findByShowId(findByShowIdRequest) returns (stream Title);
}

message findByReleaseYearRequest { string releaseYear = 1; }
message allTitleRequest {}
message findByDirectorRequest { string director = 1; }
message findByShowIdRequest { string showId = 1; }

message Title {
    string showId = 1;
    string type = 2;
    string title = 3;
    string director = 4;
    string cast = 5;
    string country = 6;
    string dateAdded = 7;
```

```
    string releaseYear = 8;
    string rating = 9;
    string duration = 10;
    string listedIn = 11;
    string description = 12;
}
```

Listado 14: Descripción del servicio gRPC para la prueba del catálogo de Netflix

Para finalizar con *gRPC*, se enseña un ejemplo de las peticiones HTTP que se pueden enviar al contenedor para consular el servicio de *gRPC* en caso de no tener un cliente capaz de trabajar con *Protocol Buffers*. Como ocurre con el caso de REST API, estas peticiones se pueden enviar también desde la página de documentación a través de *swagger-ui*. Por simplicidad, en las peticiones mostradas en el Listado 15, no se muestra el flag *-H 'Content-Type: application/json'*.

```
curl -X POST http://localhost:8080/grpc/findByDirector \
   -d '{"director"  :  "Steven Spielberg"}'
curl -X POST http://localhost:8080/grpc/\
   findDistinctFirst10ByReleaseYearOrderByTitleAsc -d '{"releaseYear" : "2020"}'
curl -X POST http://localhost:8080/grpc/findByShowId \
  -d '{"showId" : "s408"}'
curl -X POST http://localhost:8080/grpc/allTitles
```

Listado 15: Ejemplo de peticiones al adaptador de gRPC para la prueba del catálogo de Netflix

Por último, se muestran capturas de la página de documentación para el contenedor generado durante la prueba. Cabe destacar que, solo se puede acceder a ella una vez que se ha lanzado la imagen del contenedor. La Figura 14 se corresponde con la vista de la sección *Overview* y se comporta como la página de inicio. En la zona de la izquierda se puede apreciar un menú que permite navegar al resto de secciones disponibles en la página.

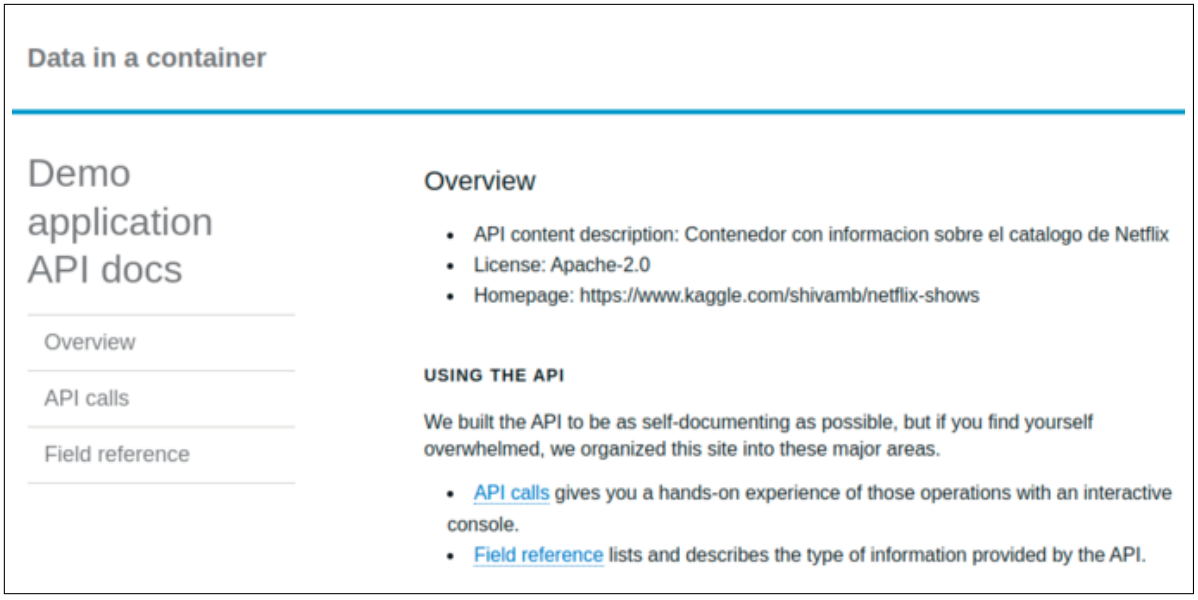


Figura 14: Página de inicio de documentación para la prueba del catálogo de Netflix

El apartado *API calls* permite probar las diferentes interfaces. Expone un enlace para acceder al entorno *GraphiQL* y una serie de desplegables de *swagger-ui* desde donde se puede configurar y ejecutar las consultas a través de REST API y el adaptador para *gRPC-REST*. La Figura 15 enseña una petición de ejemplo desde el entorno *GraphiQL*. En la zona de la izquierda se encuentra la definición de la consulta y en la derecha el resultado de la misma. La zona superior permite, entre otras cosas, ejecutar la consulta y ver un histórico de las consultas ejecutadas. En la Figura 16 se puede apreciar una versión reducida de la entidad. El objetivo es que el usuario disponga de una sección donde pueda consultar la entidad almacenada en el contenedor. Cabe destacar que la tabla se genera con el motor de plantillas según el contenido del script de configuración. La Figura 17, muestra una vista reducida de los desplegables de *swagger-ui*. Estos han sido generados de forma automática gracias a la descripción del servicio en *Open API 3*.

Figura 15: Ejemplo de petición enviada al contenedor desde el entorno GraphiQL para la prueba del catálogo de Netflix

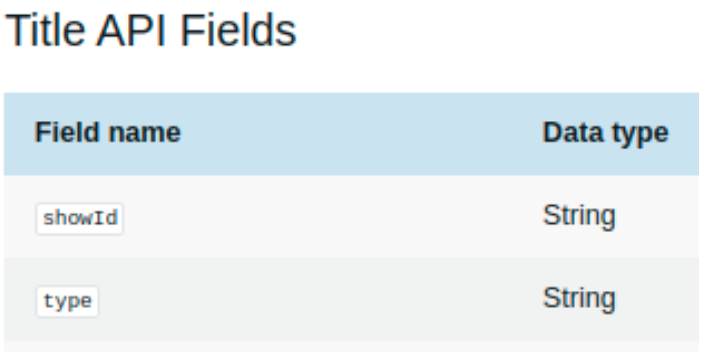


Figura 16: Descripción de los atributos de la entidad para la prueba del catálogo de Netflix generado con el motor de plantillas

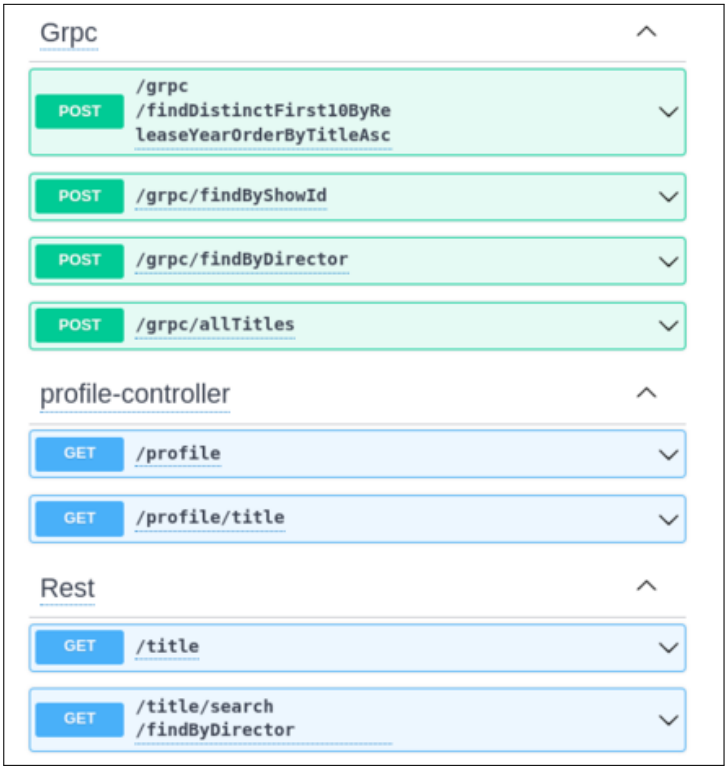


Figura 17: Interfaz de Swagger para REST API y cliente gRPC-REST en la prueba del catálogo de Netflix

E. Prueba publicaciones Airbnb

El siguiente anexo muestra el contenido relacionado con la prueba ejecutada con los datos de Airbnb sobre alojamientos y reseñas en la ciudad de Barcelona. La explicación en detalle de la misma puede consultarse en la sección 5.2. El modelo de datos se muestra en la Figura 18. El Listado 16 muestra el contenido del script `build.diac.kts`. Utilizado para describir la construcción del contenedor.

```
dac {
    name = "airbnb"
    fullName = "Airbnb-bcn"
    desc = "Informacion sobre Airbnb barcelona"
    license = "Apache-2.0"
    homepage = "http://insideairbnb.com/get-the-data/"
    packageName = "com.manyToOne.tfg"
    mainClass = "ManyToOneTest"
    data {
     files {
       "listing" {
          src = "listings.csv"
          schema {
            "listing_id" type integer property isPrimaryKey
            "name" type text
            "host_id" type integer
            "neighbourhood_group" type text
            "neighbourhood" type text
            "room_type" type text
            "price" type integer
            "minimum_nights" type integer
            "number_of_reviews" type integer
          }
          operations { create<Query>("findByPrice", "price")  }
          modules {
            install(restApi) { collectionResourceRel = "listing" }
            install(graphqlApi) {}
            install(grpcApi) { serviceName = "ListingService" }
          }
       }
       "review" {
          src = "reviews.csv"
          schema {
            "review_id" type integer property isPrimaryKey
            "listing_reviewed" type integer references "listing"
                column "listingId"
            "date" type text
          }
          operations {
            create<Query>("findByReviewId", "reviewId")
          }
          modules {
            install(restApi) { path = "review" }
            install(graphqlApi) { controllerName = "review" }
            install(grpcApi) {}
          } } }
    }
}
```

Listado 16: Script con la descripción del contenedor para la prueba de Airbnb

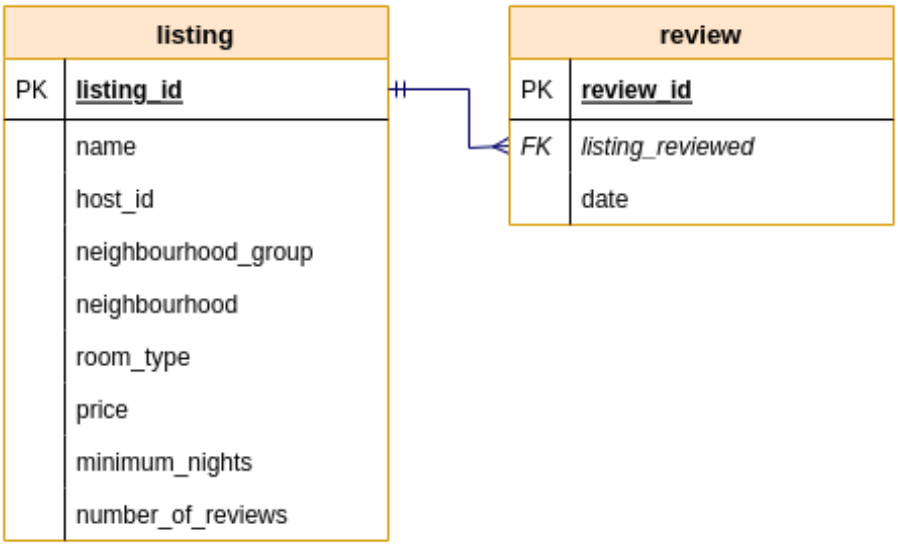


Figura 18: Modelo de datos de la información de Airbnb Barcelona

En el Listado 17 se puede apreciar la descripción del esquema de *GraphQL*. Se encarga de exponer las consultas indicadas a través de la interfaz y de definir las entidades que se almacenan, `ListingEntity` y `ReviewEntity`. En este caso, cabe destacar el atributo `listingReviewed`, que hace referencia a la clave extranjera definida en el script (Listado 16).

```
type Query {
  findByPrice(price : Int) : [ListingEntity]
  allListings : [ListingEntity]
  findByReviewId(reviewId : Int) : [ReviewEntity]
  allReviews : [ReviewEntity]
}

type ListingEntity {
    listingId : Int
    name : String
    hostId : Int
    neighbourhoodGroup : String
    neighbourhood : String
    roomType : String
    price : Int
    minimumNights : Int
    numberOfReviews : Int
}
type ReviewEntity {
    reviewId : Int
    listingReviewed : ListingEntity
    date : String
}
```

Listado 17: Descripción del esquema de GraphQL para la prueba de Airbnb

En el Listado 18 se puede observar un ejemplo de como se definirían las consultas expuestas en *GraphQL*. De nuevo, al igual que en la prueba con el catálogo de Netflix, se han generado automáticamente las consultas `allListings` y `allReviews`.

```
{
  findByPrice(price : 140) {
    neighbourhoodGroup
    roomType
  }

  allListings{
    price
    hostId
  }

  findByReviewId(reviewId: 4) {
    date
    listingReviewed {
      minimumNights
    }
  }

  allReviews{
    date
  }
}
```

Listado 18: Consultas expuestas a través de GraphQL para la prueba de Airbnb

En el Listado 19 se pueden apreciar ejemplos de peticiones para las consultas expuestas en REST API. De nuevo, es posible ejecutarlas también desde la página de documentación a través de *swagger-ui*.

```
curl -X GET http://localhost:8080/ListingServices/search/findByPrice?price=170
curl -X GET http://localhost:8080/ListingServices
curl -X GET http://localhost:8080/ListingServices/582364

curl -X GET http://localhost:8080/review/search/findByReviewId?reviewId=100
curl -X GET http://localhost:8080/review
curl -X GET http://localhost:8080/review/2
curl -X GET http://localhost:8080/review/2/listingReviewed
```

Listado 19: Ejemplo de peticiones REST API para la prueba de Airbnb

Seguidamente, en el Listado 20, se muestra la definición de los servicios expuestos en *gRPC*. Se especifica uno por cada entidad, `Listing` y `Review`. Al igual que ocurría en el esquema de *GraphQL*, el atributo `listingReviewed` de la entidad `Review` permite tratar una clave extranjera como un objeto de la entidad a la que se hace referencia.

```
syntax = "proto3";

package proto;

import "google/protobuf/wrappers.proto";
import "google/protobuf/timestamp.proto";
import "google/protobuf/struct.proto";

service ListingService {
 rpc findByPrice(findByPriceRequest) returns (stream Listing);
 rpc allListings(allListingRequest) returns (stream Listing);
}

message findByPriceRequest { int32 price = 1; }
message allListingRequest {}
message Listing {
 int32 listingId = 1;
 string name = 2;
 int32 hostId = 3;
 string neighbourhoodGroup = 4;
 string neighbourhood = 5;
 string roomType = 6;
 int32 price = 7;
 int32 minimumNights = 8;
 int32 numberOfReviews = 9;
}

service ReviewService {
 rpc findByReviewId(findByReviewIdRequest) returns (stream Review);
 rpc allReviews(allReviewRequest) returns (stream Review);
}

message findByReviewIdRequest { int32 reviewId = 1; }
message allReviewRequest {}
message Review {
 int32 reviewId = 1;
 Listing listingReviewed = 2;
 string date = 3;
}
```

Listado 20: Descripción del servicio gRPC para la prueba de Airbnb

Para finalizar con *gRPC*, se muestra un ejemplo de las peticiones HTTP que se pueden ejecutar para consular el servicio de *gRPC*. Estas peticiones se pueden enviar también desde la página de documentación a través de *swagger-ui*.

```
curl -X POST http://localhost:8080/grpc/findByReviewId -d '{reviewId : 12}'

curl -X POST http://localhost:8080/grpc/findByPrice -d '{price : 55}'

curl -X POST http://localhost:8080/grpc/allReviews

curl -X POST http://localhost:8080/grpc/allListings
```

Listado 21: Ejemplo de peticiones al adaptador de gRPC para la prueba de Airbnb

A continuación, se enseñan capturas de la página de documentación. El diseño de esta es idéntico al mostrado en la prueba de Netflix. La Figura 19 muestra la vista de la página *Overview*, donde la única diferencia a la prueba anterior es el contenido de la descripción del contenedor.

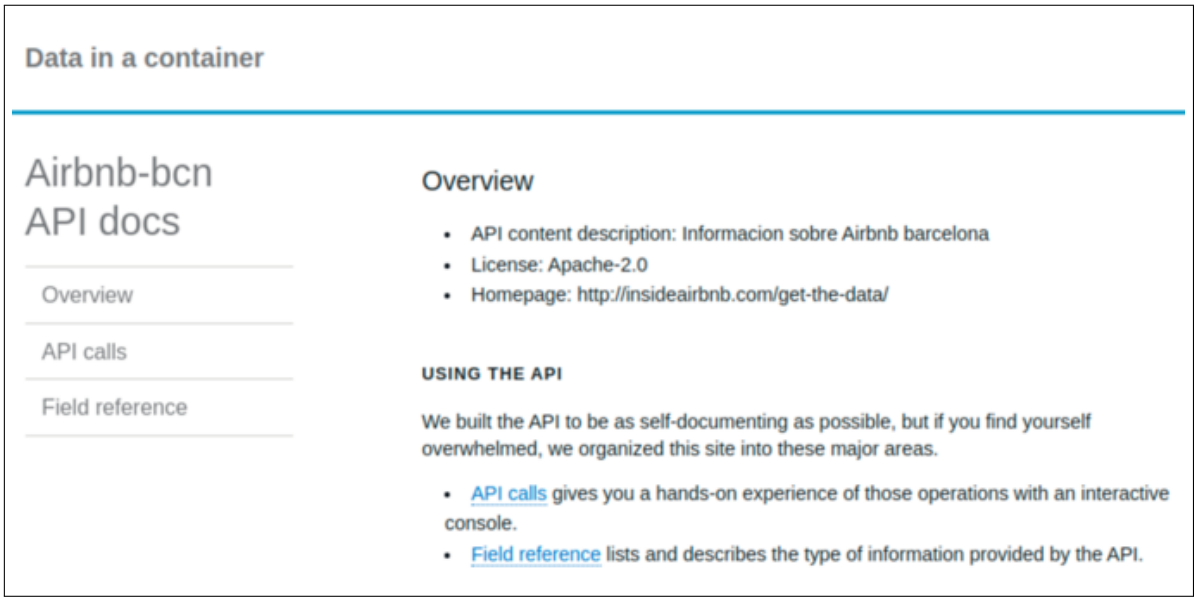


Figura 19: Página de inicio de la documentación para la prueba de Airbnb

La Figura 20 y la Figura 21 muestran los menús desplegables de *swagger-ui*, accesibles en la sección *API calls*. Esta última enseña como se lleva a cabo la configuración y ejecución de una consulta en la interfaz API REST. En este caso se ha decidido no mostrar la ejecución de una consulta de prueba en el entorno *GraphiQL*, ya que no aporta información adicional.

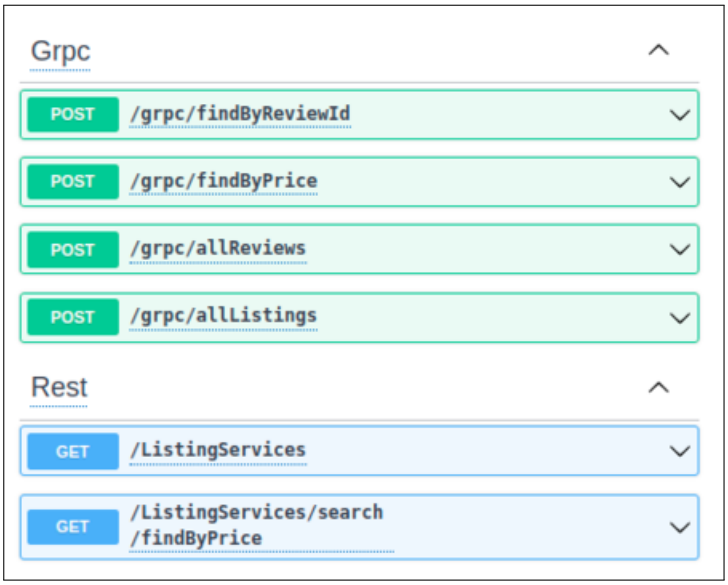


Figura 20: Interfaz de Swagger para REST API y cliente gRPC-REST para la prueba de Airbnb

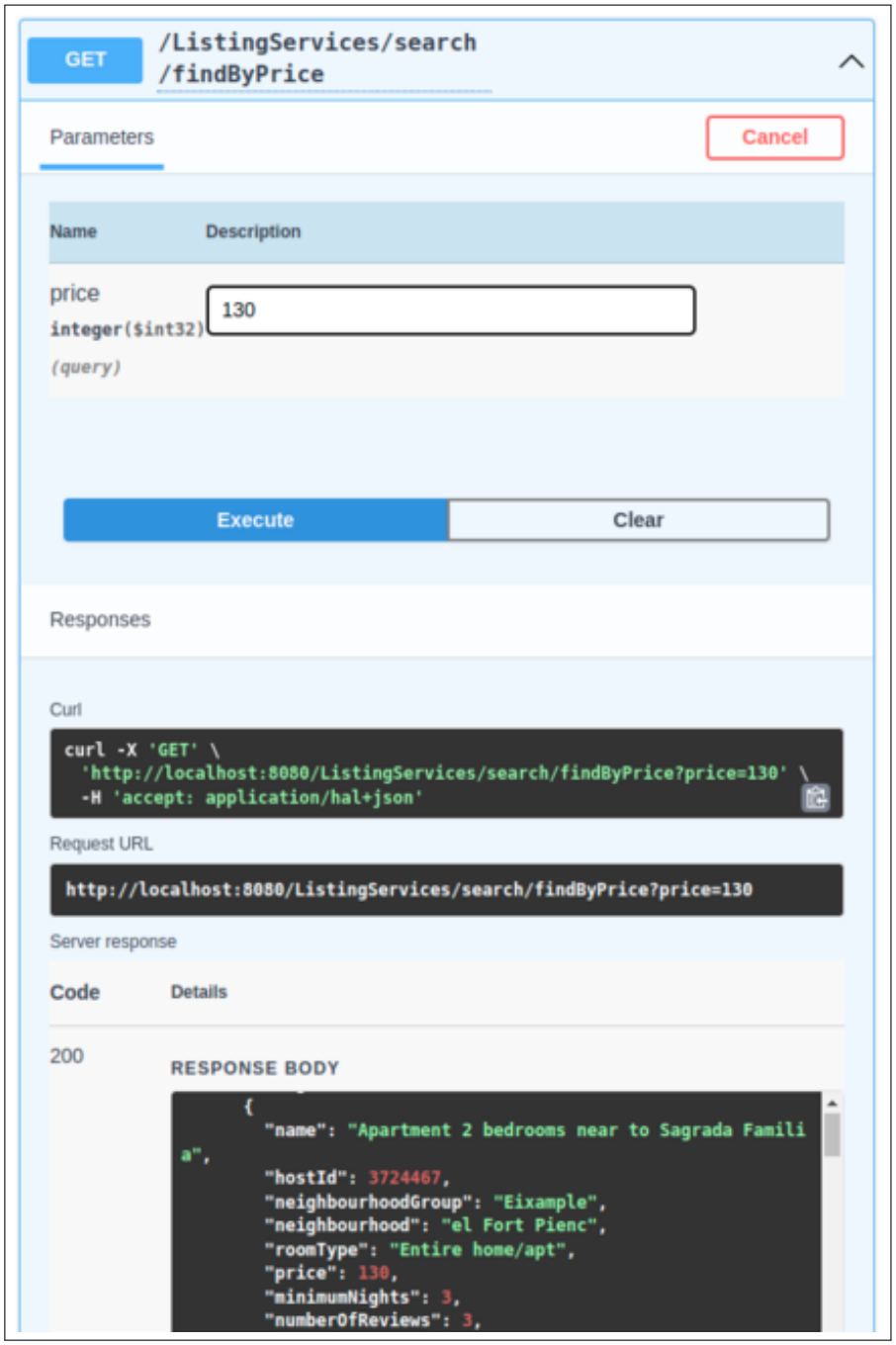
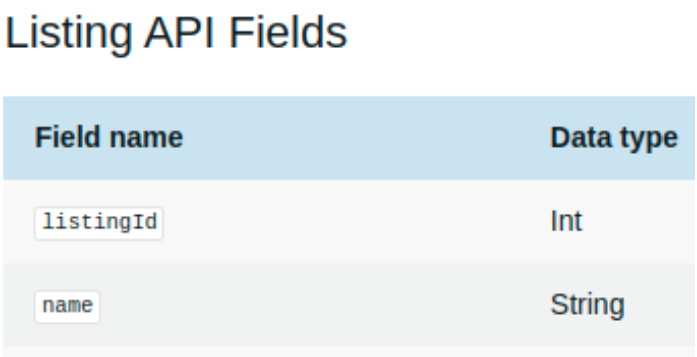


Figura 21: Ejemplo de petición a través de la interfaz de Swagger en la prueba de Airbnb

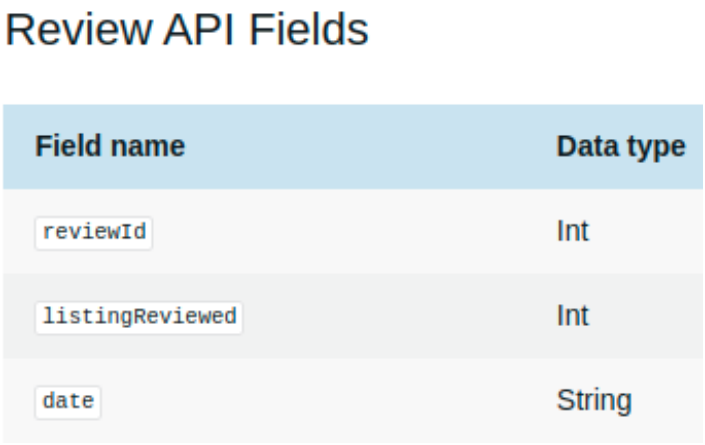
Para concluir, se muestra el contenido de la sección *Field reference*. Esta contiene las tablas de las dos entidades con las que se trabaja, `Listing` y `Review`.



Listing API Fields

Field name	Data type
`listingId`	Int
`name`	String

Figura 22: Descripción de los atributos de la tabla Listing para la prueba de Airbnb



Review API Fields

Field name	Data type
`reviewId`	Int
`listingReviewed`	Int
`date`	String

Figura 23: Descripción de los atributos de la tabla Review para la prueba de Airbnb