TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

UNIVERSIDAD DE ALMERIA

ESCUELA SUPERIOR DE INGENIERÍA

Despliegue de aplicaciones escalables con Kubernetes

Curso 2019/2020

Alumno/a:

Antonio López García

Director/es:

Manuel Torres Gil

UNIVERSIDAD DE ALMERÍA

Resumen/Abstract

Las arquitecturas de aplicaciones basadas en microservicios son cada vez más utilizadas por las empresas por la multitud de mejoras que presentan respecto a las arquitecturas monolíticas tradicionales. Para llevar a cabo esta arquitectura, surge la tecnología Docker la cual propone una nueva técnica de virtualización de entornos y en el desarrollo de diferentes aplicaciones y proyectos software utilizando contenedores, y Kubernetes como orquestador de estos contenedores que permite la autoescalabilidad de estas aplicaciones implementadas en estos contenedores.

En este trabajo, se lleva a cabo el desarrollo de una aplicación web basada en microservicios utilizando contenedores Docker y Kubernetes a la hora de desplegar la aplicación para hacer que esta sea autoescalable.

Application architectures based on microservices are increasingly used by companies due to the multitude of improvements they present compared to traditional monolithic architectures. To carry out this architecture, Docker technology arises which proposes a new technique for virtualization of environments and in the development of different applications and software projects using containers, and Kubernetes as an orchestrator of these containers that allows the self-scalability of these applications implemented in these containers.

In this project, the development of a web application based on microservices is carried out using Docker and Kubernetes containers when deploying the application to make it self-scalable.

GRADO EN INGENIERÍA INFORMÁTICA, 2019/2020

Trabajo Fin de Grado

Despliegue de aplicaciones escalables con Kubernetes.

Alumno/a: Antonio López García
Director: Manuel Torres Gil

Grado en Ingeniería Informática
Escuela Superior de Ingeniería
Universidad de Almería

Curso 2019/2020

CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

ÍNDICE FRAGMENTOS DE CÓDIGO

1. INTRODUCCIÓN

En este primer punto, se explicará la motivación principal por la que se decidió la elaboración y desarrollo de este este proyecto, así como los objetivos principales que se pretenden alcanzar. También se comentará la planificación que se ha planteado para la elaboración de este trabajo así como la estructura que presentará esta memoria.

1.1. MOTIVACIÓN

En estos últimos años, cada vez más empresas (tales como Netflix, Amazon, Ebay y más) se decantan por una arquitectura basada en microservicios, debido a las ventajas que presenta frente a la arquitectura monolítica tradicional como son la capacidad de ofrecer cada uno de los componentes de forma aislada, lo que permite una mayor facilidad a la hora de mantener y responder la aplicación frente a distintos niveles de escalabilidad; u ofrecer la posibilidad de replicar aquellas instancias de microservicios que sean necesarios evitando así tener que replicar toda la aplicación en su conjunto como ocurre en la arquitectura monolítica, entre muchas de las ventajas que esta metodología implica y lo que hace que hoy en dia se esté convirtiendo en un estándar.

Los microservicios están muy ligados al concepto contenedor, una unidad estándar de software que empaqueta el código junto a todas sus dependencias para que el servicio o aplicación en cuestión se ejecute de forma rápida y fiable de un entorno informático a otro de lo que es responsable Docker, la tecnología que permite el uso de estos contenedores.

Gracias a estos contenedores, se abre una nueva posibilidad en el desarrollo y despliegue de servicios y aplicaciones, ya que suponen una manera de virtualización mucho más rápida y eficiente que otros métodos de virtualización, como las máquinas virtuales creadas mediante Hipervisores que instalaban el sistema operativo en el que se quería trabajar y estos trabajaban de forma aislada frente al sistema operativo del equipo propio.

Además del despliegue de contenedores, hace falta una orquestación de estos debido a posibles problemas que pudieran tener estos servicios o aplicaciones como en un momento dado, obtener un gran tráfico de entrada y por ello, tener que escalar la aplicación o servicio a más servidores. De esto y más es de lo que se encarga un orquestador de contenedores, siendo este conocido como Kubernetes y que en el despliegue de una aplicación basada en contenedores, es fundamental su dominio y su uso.

La motivación de este Trabajo de Fin de Grado, por tanto, es ser capaz de desarrollar una aplicación web mediante el uso de contenedores, basandonos en la arquitectura de los microservicios creando distintos servicios que se comuniquen entre sí y poder llevar a cabo su despliegue mediante Kubernetes, para permitir que esta aplicación sea escalable, además de el resto de ventajas que implica el uso de una plataforma como esta.

La motivación final del proyecto, que es la elaboración del Complemento del Trabajo de Fin de Grado, es poder llevar a cabo tareas de monitorización de los microservicios mediante herramientas como Istio creando un Service Mesh lo cual aporta una infraestructura extra a toda la red de servicios para poder llevar a cabo cambios de versiones entre aplicaciones, acceso a versiones según distintos criterios como usuarios de la aplicación, o direcciones ip, entre otras muchas más posibilidades.

1.2. OBJETIVOS

El objetivo principal de este Trabajo de Fin de Grado es el desarrollo y despliegue de una aplicación web basada en una arquitectura de microservicios utilizando los contenedores Docker en lugar de otros métodos de virtualización como máquinas virtuales y su posterior orquestación mediante Kubernetes.

Esta aplicación web debe estar compuesta por un backend y un front-end, teniendo lo primero una base de datos que almacene toda la información principal para su correcto funcionamiento y una API REST que interactúe con la base de datos creada y que permita interactuar con esta. El front-end deberá conectarse con la API para que un usuario pueda llevar a cabo distintas acciones en la aplicación web. Además se llevará a cabo una segunda versión de la aplicación que permita observar más ventajas de la orquestación de contenedores que lleva a cabo Kubernetes.

Para cumplir el objetivo principal, habrá que llevar a cabo distintas tareas:

- Estudio de la tecnología Docker para trabajar con los contenedores.
- Selección de un sistema gestor de bases de datos y elaborar la base de datos que tendrá nuestra aplicación.
- Seleccionar la tecnología con la que desarrollar la API REST.
- Construccion de la API REST para que lleve a cabo tareas CRUD.
- Selección de tecnologías para desarrollar el front-end.
- Estudio de Docker-Compose para el despliegue de la aplicación contenedorizada.
- Realizar el despliegue de los distintos contenedores mediante Docker-Compose.
- Creación de imágenes Docker que permitan un ciclo de desarrollo basado en versiones.
- Estudio de Kubernetes para su posterior uso en el despliegue de la aplicación y su orquestación.
- Creación de archivos YAML que permiten el despliegue de la aplicación mediante la plataforma Kubernetes.
- Selección de herramientas open-source de despliegue de aplicaciones en contenedores que mejor se adapte a las necesidades de este proyecto.
- Despliegue de la aplicación en la herramienta elegida.
- Pruebas de escalado de la aplicación y monitorización de la misma, así como pruebas de despliegues de versiones de la aplicación desplegada.

Con la realización de estas tareas, se debería lograr cumplir con el objetivo principal de este proyecto y disponer así, de una aplicación que tendría como estructura la observada en la figura 1.

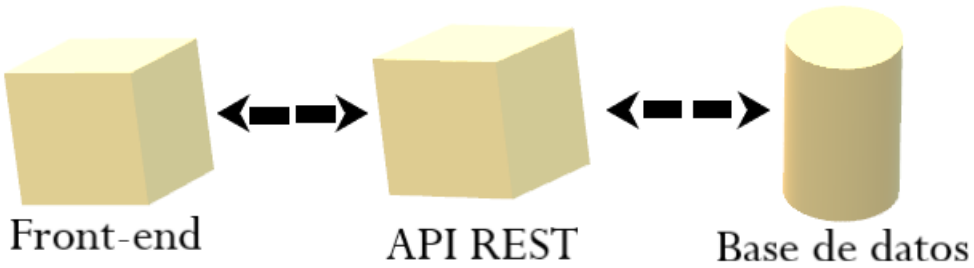

Figura 1: Estructura básica de la aplicación web

1.3. PLANIFICACIÓN

Para llevar a cabo este proyecto, se han planificado una serie de etapas para que el desarrollo de este Trabajo de Fin de Grado se llevara a cabo con éxito. Estas etapas agrupan las tareas descritas en el punto 1.2 y son las etapas que han sido marcadas para concretar reuniones donde se comprobara que se han llevado a cabo los objetivos correspondientes a cada etapa, se analizaría la siguiente etapa a llevar a cabo y su realización.

Estas etapas son las siguientes:

I. Estudio de Docker para conocer el funcionamiento de sus contenedores, como tratarlos mediante los comandos que nos otorga esta tecnología, así como el uso de volúmenes y la creacion de imágenes Docker.
II. Creacion de la base de datos que almacenará toda la información con la que trabajará la aplicación web.
III. Estudio de las API REST e implementación de la que usará nuestra aplicación. Para ello, habrá que llevar a cabo una selección de las tecnologías a utilizar para este desarrollo. Se desarrollarán dos versiones para las pruebas de despliegues progresivos.
IV. Selección de las tecnologías a utilizar para el desarrollo del front-end y su correspondiente implementación. Se desarrollarán dos versiones para hacer pruebas de despliegues progresivos.
V. Estudio de Kubernetes para conocer sus caracteristicas, como llevar a cabo despliegues de aplicaciones basadas en contenedores, los distintos objetos que esta plataforma presenta, etc.
VI. Selección de plataforma en la que llevará a cabo el despliegue de la aplicación y su posterior despliegue, realizando las configuraciones necesarias en la plataforma elegida.
VII. Despliegue de la aplicación, así como de los elementos necesarios para que la aplicación sea escalable y la realización de pruebas de autoescalado de la misma.
VIII. Elaboración de este documento.

En la figura 2 se aprecia de una forma más visual esta planificacion donde además, se indica la duración en horas requeridas para cada una de las etapas descritas, las cuales suman un total de las 300 horas que se requieren para la elaboración de este Trabajo de Fin de Grado.

Figura 2: Cronograma Trabajo Fin de Grado

Al inicio de cada etapa, se llevó a cabo una reunión donde se trataron los aspectos necesarios a tratar para poder desarrollar esa etapa y a medida que se llevaba a cabo, se realizaban seguimientos de forma que al finalizar la etapa, se llevaba a cabo comprobaciones

que aseguraran el correcto funcionamiento y la posibilidad de seguir avanzando en el desarrollo del proyecto y así marcar el objetivo de la siguiente etapa.

Este enfoque a la hora de trabajar en el proyecto es característico de las metodologías *Agile*, las cuales se refieren a un grupo de metodologías aplicadas en la creación de software que basa su desarrollo en un ciclo iterativo, en el que las necesidades y las soluciones evolucionan a través de la colaboración entre los diferentes miembros involucrados en un proyecto.

1.4. ESTRUCTURA DE LA MEMORIA

El resto de este documento se organiza en los siguientes puntos.

En el capitulo 2 se describen las tecnologías y herramientas que se han utilizado para completar el objetivo principal de este Trabajo de Fin de Grado.

En el capitulo 3 se describe todo el desarrollo del proyecto, las implementaciones de todo el software creado así como imágenes con fragmentos de código y figuras para hacer más entendible las implementaciones, las ventajas que aporta la tecnología de los contenedores y cómo crear una aplicación basada en estos paquetes software y cómo desplegar una aplicación mediante plataformas de despliegue de contenedores, además de otorgar la funcionabilidad de autoescalabilidad a la aplicación y mostrar la aplicación en sí.

En el capítulo 4 se comentan las conclusiones del trabajo realizado, así como posibles mejoras al trabajo realizado y novedosas tecnologias que continúan impulsando el uso de las tecnologías aplicadas.

En el capítulo 5 se muestra toda la bibliografía utilizada en el desarrollo de este Trabajo de Fin de Grado.

2. HERRAMIENTAS Y TECNOLOGÍA

Una vez se ha estudiado el objetivo principal, las distintas tareas que hay que realizar y las etapas programadas para llevar a cabo este proyecto, toca seleccionar las distintas tecnologías con las que se llevará a cabo este Trabajo de Fin de Grado.

2.1. DOCKER

Esta tecnología es la base de este proyecto. Lanzado el 20 de Marzo de 2013, es un proyecto de código abierto que automatiza el despliegue de aplicaciones dentro de contenedores de software, proporcionando así una capa de abstracción y automatización de virtualización de aplicaciones en múltiples sistemas operativos. Como ya se comentó en el apartado 1.1, los contenedores son unidades estándar de software que empaqueta el código junto a las dependencias necesarias que el desarrollo necesite para que pueda ser ejecutado. Esto último permite que estos contenedores se puedan ejecutar en cualquier entorno, cualquier sistema operativo y cualquier servidor.

Esto supone una nueva forma de virtualización que presenta ventajas frente a otros tipos de virtualización como las máquinas virtuales creadas mediante Hipervisores (plataformas que permiten aplicar diversas técnicas de control de virtualización para utilizar, al mismo tiempo, diferentes sistemas operativos en una misma computadora) debido a que Docker ejecuta los contenedores sobre el mismo sistema operativo anfitrión de forma aislada, sin necesidad de un sistema operativo propio ya que comparten el mismo kernel, lo que los hace mucho más ligeros como se ve en la figura 3. Mientras que un contenedor puede ocupar unas cuantas decenas de megas, una máquina virtual, al tener que emular todo un sistema operativo, puede ocupar varios gigas de memoria.

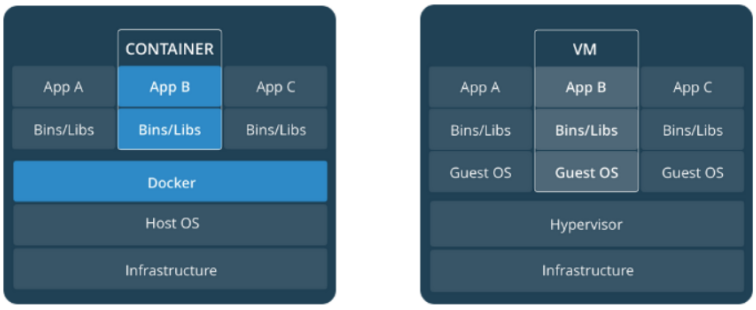

Figura 3: Diferencia entre contenedores y máquinas virtuales

No solo supone una mejora en cuanto al espacio y recursos consumidos por estos contenedores, sino que presentan la posibilidad de llevar a cabo una programación con arquitectura basada en microservicios, y cada uno de estos, poder replicarlos hasta satisfacer las necesidades de la aplicación o servicio a desarrollar, que a diferencia de esta situación en máquinas virtuales, hay que replicar toda la máquina virtual y con ello, no solo volvemos a la problemática del consumo de recursos, sino de velocidad de despliegue.

En resumen, las grandes virtudes que presentan estos contenedores como método de virtualización son su ligereza (ya que como se ha dicho antes, comparten el kernel del equipo host), son intercambiables (permiten despliegue de actualizaciones en caliente), son portables (se pueden desplegar y ejecutar en cualquier entorno) y son escalables (permite un aumento y distribución automática de réplicas de contenedores).

Sin embargo, Docker no es la única que utiliza esta tecnología de contenedores, ya que existen los LXC Linux Containers, la cual es una tecnología de virtualización a nivel de sistema operativo que permite la creación y ejecución de múltiples entornos virtuales Linux aislados en un solo host de control. Estos contenedores se pueden usar para aplicaciones específicas de sandbox o para emular un entorno completamente nuevo. La ventaja que presenta Docker es que realmente es una extensión de las capacidades de LXC ya que utiliza una API de alto nivel que proporciona una virtualización ligera para ejecutar procesos de forma aislada. Sus contenedores están basados en LXC, pero el contenedor de Docker no incluye un sistema operativo separado, sino que se basa en la propia funcionalidad del sistema operativo proporcionada por la infraestructura subyacente. Así es como Docker actúa como un motor de contenedor portátil, empaquetando aplicaciones y todas sus dependencias en un contenedor virtual que puede ejecutarse en cualquier servidor.

Además de Docker, existen herramientas con las que consigue desplegar y gestionar contenedores, como es *Containerd*. Esta herramienta se trata de un *Daemon* (programa o servicio no interactivo, es decir, que se ejecuta en segundo plano y que no es controlado directamente por el usuario) que gestiona el ciclo de vida completo del contenedor del sistema host, desde la transferencia y el almacenamiento de imágenes hasta la ejecución y supervisión del contenedor, y demás. Ayuda a abstraer las llamadas al sistema o la funcionalidad específica del sistema operativo para ejecutar los contenedores Linux, Windows o cualquier otro sistema operativo. Proporciona una capa cliente sobre la que cualquier otra plataforma como Docker o Kubernetes (orquestador de contenedores) puede construir.

Docker despliega los contenedores utilizando imágenes Docker. Una imagen es una plantilla con las instrucciones de creación de un contenedor Docker. Estas imágenes, por tanto, contienen el código en sí, las librerías que necesita, variables de entorno, archivos de configuración y los *runtime* que necesite el contenedor.

Estas imágenes se almacenan en registros de imágenes y existen varios, algunos públicos como los proporcionados por Google (Google Container Registry) y Amazon (Amazon Elastic Container Registry) y otros privados como Harbor. Docker proporciona también su propio registro de imágenes denominado Docker Hub, el cual es un repositorio público en la nube que contiene multitud de imágenes, de carácter gratuito, que se pueden descargar y sirven de plantillas evitando así el tener que llevar a cabo una implementación de una imagen Docker. Estas imágenes una vez descargadas, se almacenan en nuestro equipo y son gestionadas por Docker para cuando sean necesarias, desplegar los contenedores que necesiten estas imágenes mediante comandos que Docker nos permite utilizar en la terminal de nuestro equipo. Este es el funcionamiento básico de Docker, el cual se aprecia mejor en la figura 4.

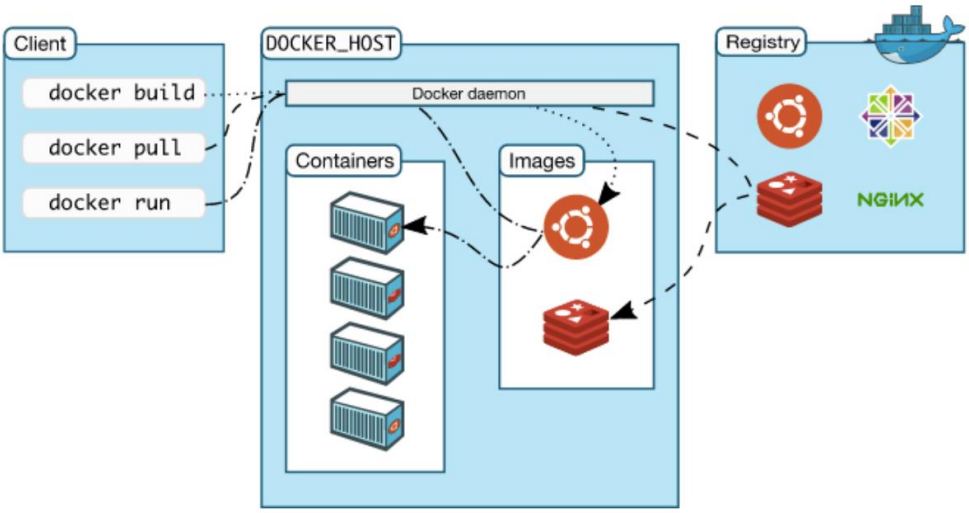

Figura 4: Funcionamiento básico de Docker

Docker nos proporciona numerosos comandos para tratar los contenedores, siendo algunos de estos, los mostrados en la tabla 2.1 y que son básicos para trabajar con Docker.

Comando	Acción
docker run <image>	Crea un contenedor a partir de una imagen.
docker run -d -p "puerto1:puerto2"	Crea un contenedor en modo *deattached* accesible desde nuestro puerto1 al puerto2 del contenedor.
docker stop|start <name or id>	Detiene|Continúa un contenedor.
docker ps -a	Listado de contenedores (con -a muestra también los parados).
docker stop <name or id>	Detiene el contenedor indicado.
docker rm <name or id>	Elimina el contenedor indicado.
docker exec -it <name or id> bash	Abre un terminal en un contenedor.

Tabla 1: Comandos básicos Docker

Los contenedores son volátiles, y por tanto la eliminación de estos supone la pérdida de información que estos contienen. Por ello, Docker permite utilizar volúmenes como solución para el almacenamiento de datos fuera de los contenedores como los denominados *Bind Mounts* cuya finalidad es montar un archivo o directorio del equipo host en un contenedor.

Esto es básico en el desarrollo de aplicaciones o servicios contenedorizadas, para realizar pruebas de desarrollo levantando contenedores temporalmente y comprobar así el funcionamiento de las implementaciones. Para llevar a cabo estas uniones de directorios, basta con utilizar el parámetro *-v* a la hora de levantar un contenedor, e indicar en primer lugar la ruta donde se encuentra el archivo o directorio que deseamos que esté en el contenedor y en

segundo lugar, el directorio del contenedor donde se desea que esté la información de la primera ruta teniendo la forma *-v rutaequipolocal:rutacontenedor* [1,2,3,4]. Más información de la aplicación en https://docs.docker.com/.

2.1.1. DOCKER-COMPOSE

Docker-Compose es una herramienta para definir y ejecutar aplicaciones Docker con múltiples contenedores. Con Compose, se utiliza un archivo Compose (.yml) para configurar los distintos servicios de la aplicación.

Un ejemplo de este tipo de archivos es el mostrado en el fragmento de código 1, en el que se despliega una aplicación con dos contenedores, cada uno de ellos desplegando un servicio distinto. Uno de los dos servicios hace referencia a una base de datos utilizando una imagen mysql, así como un *bind mount* (con la opción `volumes`) en el que enlaza el directorio del contenedor cuya finalidad es inicializar una base de datos con el directorio en el que se encuentra el archivo de inicialización y también enlaza el directorio del contenedor en el que se almacenan los datos de la base de datos con el directorio donde se creará la base de datos en el equipo local. El otro servicio hace referencia a una aplicación PHP que posee un *bind mount* que enlaza el directorio en el que se almacena toda la implementación de la aplicación con el directorio que utiliza el contenedor para desplegar una aplicación web PHP.

```yaml
version: '2'
services:
  mysql:
    container_name: mysql
    restart: always
    image: mysql:5.7
    environment:
      MYSQL_ROOT_PASSWORD: 'secret'
    ports:
      - "3306:3306"
    volumes:
      - ./data:/var/lib/mysql
      - ./init.sql:/docker-entrypoint-initdb.d/init.sql
  php:
    container_name: php
    restart: always
    image: ualmtorres/phalcon-apache-ubuntu
    ports:
      - "80:80"
    volumes:
      - ./html:/var/www/html
```

Fragmento código 1: Ejemplo archivo Compose

Además, como se aprecia en el fragmento de código, se establece la unión de puertos entre el equipo local y el contenedor en la sección `ports`.

Después, con un solo comando, el cual es

```
docker-compose up -d
```

se despliega la aplicación contenedorizada (el uso de -d permite que se ejecute en segundo plano) y este comando debe utilizarse en el directorio donde se encuentra el archivo Compose, que crea e inicia todos los servicios configurados y con el comando

```
docker-compose down
```

se da de baja la aplicación. Para mas información de esta herramienta en https://docs.docker.com/compose/.

2.2. KUBERNETES

Plataforma de código abierto para el despliegue, escalado y gestión de aplicaciones contenedorizadas. Lanzado en 2014 por Google, propone una solución a la orquestación de contenedores debido a las carencias que presenta Docker a la hora de tratar con aplicaciones que poseen numerosos contenedores y servicios y que además tengan un gran tráfico de entrada. Permite por tanto llevar actividades como *dónde debe ir un contenedor*, *cómo se monitorizan esos contenedores*, *cómo se reinician cuando tengan un problema*, *necesito escalar de forma automática de 1 a 20 réplicas en función de la carga*, y una larga lista de más acciones posibles, además de que ofrece una abstracción que le permite desplegar las aplicaciones en un cluster sin pensar en las máquinas que lo soportan.

Sin embargo, Docker también posee un orquestador de contenedores denominado Docker Swarm, el cual no es más que una herramienta que se utiliza para agrupar y programar contenedores de Docker. Con esta herramienta, los desarrolladores y administraciones pueden establecer y administrar un clúster de nodos Docker en un solo sistema virtual. De manera similar a Kubernetes, Docker Swarm puede implementar en nodos y administrar la disponibilidad de esos nodos, llamando al principal nodo administrador. Dentro del *Swarm* los nodos administradores se comunican con los nodos trabajadores.

Las principales diferencias entre ambos residen en la capacidad de autoescalado que presenta Kubernetes y que no posee Docker Swarm, mientras que por otro lado, comenzar por Kubernetes presenta una curva de aprendizaje más costosa que Docker Swarm. Además, Docker Swarm está limitado a las capacidades de la API de Docker, mientras que Kubernetes permite superar las limitaciones de Docker y su API y otra gran diferencia entre estas tecnologías es la experiencia con las implementaciones de producción a escala, siendo en Kubernetes mucho mayores que en Docker Swarm.

Un cluster de Kubernetes está formado por dos tipos de recursos (máquinas físicas o virtuales) como se aprecia en la figura 5:

- El Master que coordina el cluster y se encarga de coordinar todas las actividades de este como organizar las aplicaciones, mantener el estado deseado de las mismas, el escalado, las actualizaciones de las mismas, etc. También recoge información de los nodos worker que veremos a continuación y de los Pods(unidades mínimas de despliegue en Kubernetes que contienen al menos un contenedor).
- Los nodos Workers que son los encargados de ejecutar las aplicaciones. Cada nodo tiene un agente llamado Kubelet que gestiona el nodo y mantiene la comunicación con el Master. También posee herramientas para trabajar con contenedores como por ejemplo Docker.

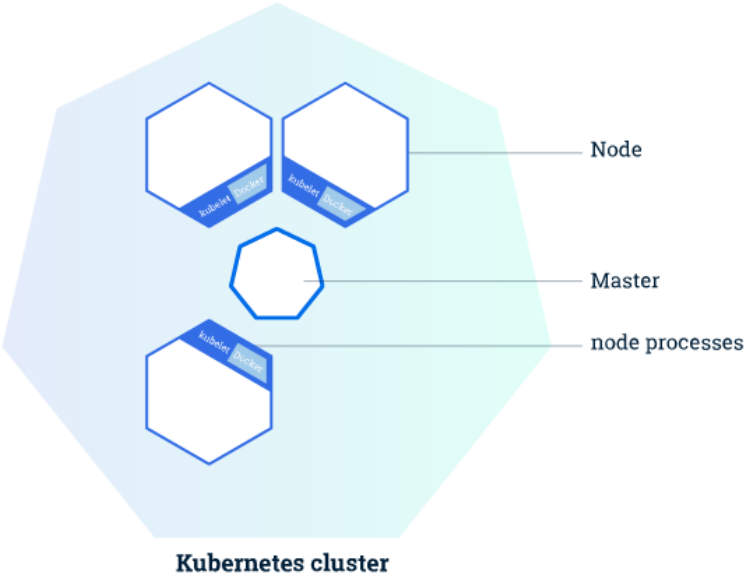

Figura 5: Nodos de Kubernetes

Kubernetes presenta diferentes objetos básicos y abstracciones de nivel superior llamados *Controladores* con los que lleva a cabo los distintos despliegues de aplicaciones basadas en contenedores en los distintos nodos. Algunos de estos elementos más importantes son:

- Deployment. Una configuración de Deployment pide a Kubernetes que cree y actualice las instancias de una aplicación. Tras crear el Deployment, el Master organiza las instancias de la aplicación en los nodos disponibles del cluster. Una vez creadas las instancias de la aplicación, el *controlador* de Deployment de Kubernetes monitoriza continuamente las instancias. Si un nodo en el que se encuentra una instancia cae o es eliminado, el *controlador* de Deployment de Kubernetes sustituye la instancia por otra instancia en otro nodo disponible del cluster.
  Esta funcionalidad de "autocuración" de las instancias de las aplicaciones supone un cambio radical en la gestión de las aplicaciones, ya que antes de los orquestadores no existía esta característica de recuperación de fallos a través de la creación de nuevas instancias que reemplacen a las defectuosas.
  Al crear un Deployment se especifica la imagen del contenedor (imagen Docker por ejemplo) que usará la aplicación y el número de réplicas que se quieren mantener en ejecución gracias a los ReplicaSet.
- Pods. La creación de un Deployment supone por parte del *controlador* la creación de Pods para ejecutar las distintas instancias de la aplicación. Un Pod es la unidad atómica de Kubernetes y se trata de una abstracción que representa un grupo de uno o más contenedores de una aplicación y algunos recursos compartidos de esos contenedores (por ejemplo, volúmenes, redes, etc.) como se aprecia en la figura 6. Los contenedores de un Pod comparten una IP y un espacio de puertos, y siempre van juntos y se despliegan juntos en un nodo. Son efímeros por lo que pierden su información al eliminarse.

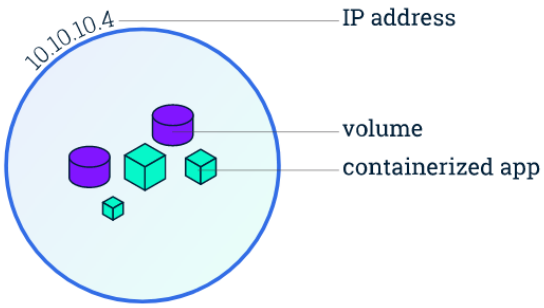

Figura 6: Pod

- ReplicaSets. Este *controlador* trata de mantener un conjunto estable de réplicas de Pods ejecutándose en todo momento. Así, se usa en numerosas ocasiones para garantizar la disponibilidad de un número específico de Pods idénticos.
  Un ReplicaSet se define con campos, incluyendo un selector que indica cómo identificar a los Pods que puede adquirir a través de su etiqueta, un número de réplicas indicando cuántos Pods debería gestionar, y una plantilla pod especificando los datos de los nuevos Pods que debería crear para conseguir el número de réplicas esperado. Un ReplicaSet alcanza entonces su propósito mediante la creación y eliminación de los Pods que vea necesario para alcanzar el número esperado. Cuando necesita crear nuevos Pods, utiliza la plantilla pod.
- Services. Los servicios de Kubernetes son una abstracción que definen un conjunto lógico de Pods y una política de acceso a ellos estableciendo un nombre para acceder a ellos. Esto permite un acoplamiento débil entre pods dependientes. El acceso puede ser interno o externo al cluster. De esta forma, las aplicaciones solo usan los nombres de los servicios y no las IPs de los pods, ya que estas nunca son fijas debido a que, por un lado, los pods se crean y se destruyen para mantener un número de réplicas deseado y, por otro lado, un pod puede ser sustituido por otro ante problemas que surjan y este nuevo tendrá una IP diferente. En función del ámbito de la exposición del servicio existen distintos tipos de servicio los cuales son:
  - ClusterIP: El servicio recibe una IP interna a nivel de cluster y hace que el servicio sólo sea accesible a nivel interno.
  - NodePort: Expone el servicio fuera del cluster concatenando la IP del nodo en el que está el pod y un número de puerto entre 30000 y 32767.
  - LoadBalancer: Crea en cloud, si es posible, un balanceador externo con una IP externa asignada.
  - ExternalName: Expone el servicio usando un nombre arbitrario.

  Los servicios agrupan a sus pods usando etiquetas y selectores como ocurre con los ReplicaSet y su unión con los Pods que deben gestionar. Los servicios usan selectores y los pods son creados con etiquetas. Su emparejamiento por valores coincidentes es lo que agrupa los pods de un servicio, al igual que las agrupaciones de distintos Pods con sus Deployments y ReplicaSets correspondientes como se aprecia en la figura 7, donde se observa cómo hay distintos pods en los dos nodos superiores que son controlados y gestionados por el Deployment B y un pod en el nodo inferior que es controlado por el Deployment A. Esto se sabe ya que la etiqueta de los pods superiores es “B”, el cual es el selector del Deployment B que

se encuentra en el nodo Master. Además también se aprecia cómo se agrupan pods de distintos nodos en un mismo servicio, como ocurre en los dos nodos superiores, lo cual se lleva a cabo de la misma forma siendo el selector de los servicios en este caso la misma letra que el nombre de los Deployments.

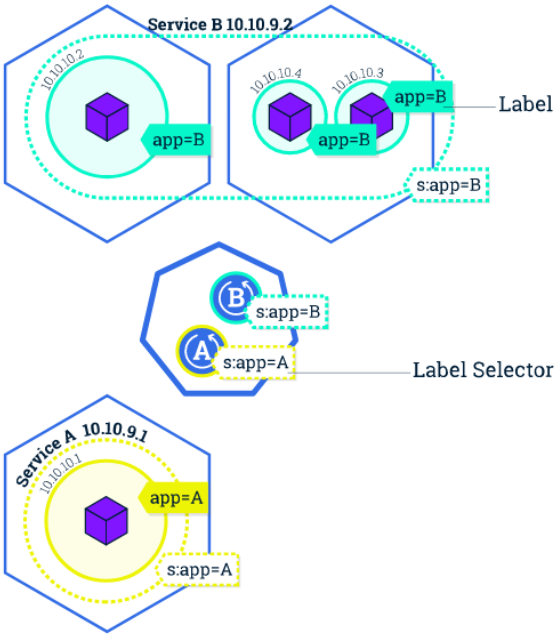

Figura 7: Etiquetas y Selectores Kubernetes

- Volúmenes. Se trata almacenamiento externo que se monta en un directorio, que son accesibles a los contenedores de un Pod y que persisten a los reinicios de los pods. El medio que se use para el almacenamiento y cómo se comporte ante una eliminación del Pod depende del tipo de volumen que se use, ya que hay de almacenamiento local, almacenamiento en el sistema de archivos de los nodos de Kubernetes, NFS y almacenamiento cloud entre otros.
- ConfigMaps. Permiten almacenar datos no sensibles en forma de pares clave-valor para que puedan usarse posteriormente en despliegues parametrizados y hacerlos más portables. Se usa para datos no sensibles ya que se guardan tal cual, sin estar codificados como ocurre en el caso de los Secrets. Un ejemplo de creación de un ConfigMap se encuentra en el fragmento de código 2.

```
1  apiVersion: v1
2  kind: ConfigMap
3  metadata:
4    name: myconfigmap
5  data:
6    nombre: Antonio
7    apellidos: Lopez
```

Fragmento código 2: Ejemplo ConfigMap

- Secrets. Objetos utilizados para almacenar información sensible como contraseñas, claves ssh y demás elementos de valor. Pese a no cifrar los datos que contienen

estos objetos, sí que codifica en base64. En el fragmento de código 3 se muestra un ejemplo de creación de objeto Secret.

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: mysqlpassword
type: Opaque
data:
  password: cGFzc3dvcmQ=
```

Fragmento código 3: Ejemplo Secret

Todos estos elementos descritos y más se almacenan en la base de datos propia de Kubernetes denominada *etcd* en la cual además se almacena información de los propios clústeres creados. Además, estos elementos se pueden crear sobre la marcha utilizando comandos, pero la manera más optima es implementarlos mediante archivos YAML, los cuales permiten una automatización en el despliegue de aplicaciones especificando los distintos elementos que necesita la aplicación. Los fragmentos de código 1, 2 y 3 son ejemplos de distintos archivos YAML en los que se lleva a cabo el despliegue de un Deployment, un ConfigMap y un Secret respectivamente.

Para la interacción con un cluster local o remoto se utiliza *kubectl*, un CLI que nos permite realizar tareas habituales como despliegues, escalar un cluster u obtener información sobre los servicios en ejecución y para ello se utiliza comandos, siendo algunos de los más importantes los mostrados en la tabla 2 [5,10]. Más información en https://kubernetes.io/es/docs/home/.

Comando	Acción
kubectl get pods	Lista todos los pods desplegados.
kubectl get all	Lista todos los objetos desplegados.
kubectl apply -f <name YAML file>	Despliega los elementos configurados en un archivo YAML
kubectl logs <pod>	Muestra los logs de un contenedor en un pod.
kubectl exec <pod><command>	Ejecuta un comando en un contenedor de un pod

Tabla 2: Comandos básicos Kubectl

2.3. PHALCON

Framework de código abierto para PHP, escrito como una extensión de C. Está optimizado para tener un alto rendimiento cuya arquitectura permite al framework estar siempre en memoria, ofreciendo su funcionalidad siempre que sea necesario, sin accesos costosos y lecturas de archivos que utilizan los frameworks PHP tradicionales. Se basa en el patrón modelo-vista-controlador (MVC) aunque permite realizar pequeños programas utilizando su librería Micro la cual permite con menos código e implementaciones, desarrollar en nuestro caso una API con distintos métodos HTTP, como es el caso del fragmento de código X en el que se le asigna a una ruta, una simple función que muestra un título <h1> basado en un parámetro.

```php
<?php

use Phalcon\Mvc\Micro;

$app = new Micro();

$app->get(
    '/orders/display/{name}',
    function ($name) {
        echo "<h1>This is order: {$name}!</h1>";
    }
);

$app->handle();

?>
```

Fragmento código 4: Ejemplo asociación ruta a función en Phalcon

Una de las principales características de este framework en la implementación de API REST reside en el trato de las distintas tablas de la base de datos a la que se conecta estos softwares, ya que mediante una clase denominada *Model*, las gestiona como si fueran objetos. Esto facilita las peticiones sql que las implementaciones en este software realizan y que han permitido con relativamente poco código, el desarrollo de los distintos endpoints de la API. Mas información en https://docs.phalcon.io/3.4/es-es/introduction.

2.4. POSTMAN

Esta herramienta open source se ha utilizado para realizar operaciones básicas sobre la base de datos MySQL, así como para realizar pruebas sobre la API construida durante el proyecto. Para más información y descargar la herramienta en https://www.postman.com/.

2.5. MYSQL

Sistema de gestión de bases de datos relacional considerada como la base de datos de código abierto más popular del mundo y una de las más populares junto a Oracle y Microsoft SQL Server. Posee de varias interfaces que permiten a aplicaciones escritas en distintos lenguajes de programación, acceder a las bases de datos de MySQL como son Python, PHP, Java, etc. Fue el elegido para desplegar la base de datos de la aplicación por su conocimiento previo de las asignaturas estudiadas en el grado. Más información en https://dev.mysql.com/doc/.

2.6. BOOTSTRAP

Biblioteca multiplataforma o conjunto de herramientas de código abierto para el diseño de sitios y aplicaciones web. Contiene plantillas de diseño con topografía, formularios, botones, cuadros, menús de navegación y otros elementos basados en HTML y CSS, así como extensiones de JavaScript adicionales. A diferencia de muchos frameworks web, Bootstrap solo se ocupa del desarrollo Front-end y gracias a la sencilles que presenta para implementar estos desarrollos, fue la elección a la hora de implementar el front-end de la aplicación. Más información en https://getbootstrap.com/docs/4.1/getting-started/introduction/.

2.7. JQUERY

Biblioteca multiplataforma de JavaScript que permite simplificar la manera de interactuar con los documentos HTML, manejar eventos, desarrollar animaciones y agregar interacción

con la técnica AJAX a páginas web(esto en especifico es lo utilizado para que nuestra aplicación web consuma la API desarrollada en Phalcon). Al tratarse de software libre y de código abierto ofrece una serie de funcionalidades basadas en JavaScript que de otra manera requerirían de mucho más código, por lo que se ahorra en tiempo y espacio y fue el motivo fundamental del uso de esta biblioteca en la conexión entre front-end y la API REST de la aplicación. Más información en https://api.jquery.com/.

2.8. MINIKUBE

Implementación ligera de Kubernetes que crea una máquina virtual localmente y despliega un cluster sencillo formado por un solo nodo. Es una gran herramienta para el desarrollo de aplicaciones Kubernetes que permite características habituales como *LoadBalancer, NodePort, Ingress, dashboard…*

Para instalar y utilizar Minikube, debe descargarse un archivo desde la página oficial de Kubernetes y añadirlo al PATH del equipo local. Además se debe tener instalado Kubectl, la interfaz de comandos mencionada en la sección 2.2 y manera estándar de comunicación con un clúster tanto local como remoto. Para instalar Kubectl, basta con descargar un ejecutable disponible en la página oficial de Kubernetes y añadirlo al PATH del equipo local como con el archivo ejecutable de minikube.

Una vez estos dos elementos están a disposición del desarrollador, basta con iniciar minikube con el comando

```
minikube start
```

para que se inicialice la máquina virtual con el cluster de Kubernetes de un solo nodo. Para desplegar los distintos elementos que componen la aplicación desarrollada por un usuario, desde el escritorio donde se encuentran los archivos YAML con las implementaciones de los elementos configurados, con el siguiente comando se despliega uno a uno estos elementos

```
kubectl apply -f "nombre del archivo a desplegar"
```

y con los dos siguiente comandos, se detiene y se elimina el cluster respectivamente

```
minikube stop
```

```
minikube Delete
```

Esta herramienta se utilizó para llevar a cabo despliegues en el equipo de forma local y comprobar posibles errores y fallos de la aplicación implementada antes de llevar a cabo el despliegue de la aplicación en Rancher y OpenStack. Más información en https://kubernetes.io/docs/setup/learning-environment/minikube/.

2.9. OPENSTACK

Proyecto de computación en la nube que proporciona infraestructura como servicio. Controla grandes grupos de recursos informáticos, de almacenamiento y de redes, todos administrados a través de una API. Mas allá de la funcionalidad estándar de infraestructura como servicio, los componentes adicionales brindan orquestación, administración de fallas y administración de servicios que garantizan una alta disponibilidad a los usuarios que utilicen esta plataforma.

Se utilizó para desplegar la instancia en la que se instaló Rancher y que esta plaforma a su vez, se retroalimentara con OpenStack para crear las instancias (maquinas virtuales) que necesitara oportunas para la creación del cluster de Kubernetes. Más información en https://docs.openstack.org/install-guide/overview.html.

2.9.1. INSTALACIÓN MÁQUINA VIRTUAL OPENSTACK

Para crear la instancia en la que se desplegó Rancher, desde la opción Instancias, al seleccionar "Lanzar instancia", aparece una ventana con numerosas configuraciones de la instancia que se va a lanzar. De todas estas configuraciones, se estableció en la sección *Detalles* el nombre de la instancia el cual fue "Rancher", en la sección *Origen* la imagen que se instaló en la máquina virtual que fue la popular distribucion Ubuntu en su versión 16.04 LTS, en *Sabor*, las especificaciones de recursos que esta máquina virtual tendría, para la que se eligio el "sabor" *small*, que dota a la máquina virtual de 2 GB de RAM y 20 GB de disco duro como se aprecia en la figura 8.

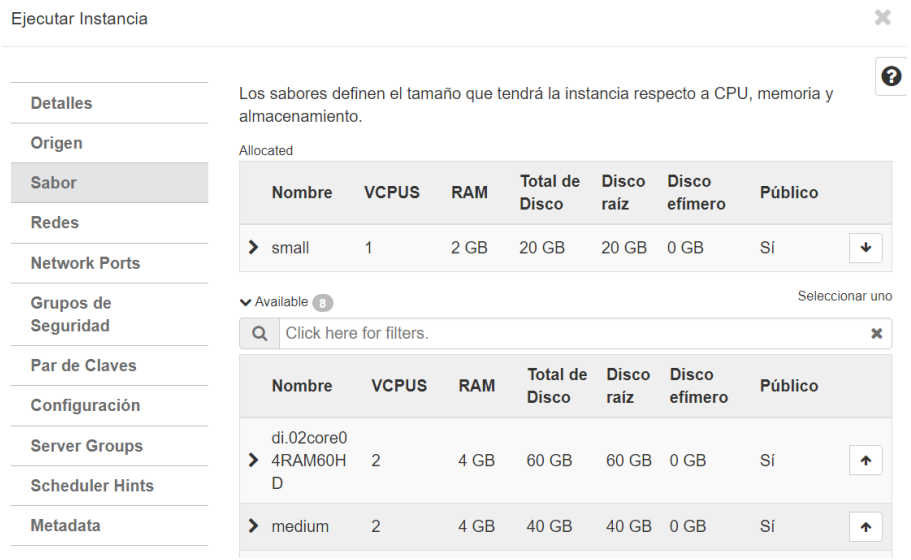

Figura 8: Instalación máquina virtual en OpenStack

En *Par de Claves* las claves se seleccionaron las claves que utilizo para llevar a cabo conexiones ssh a instancias creadas en esta plataforma, en *Grupos de Seguridad* se seleccionó las configuraciones de puertos que adoptan las máquinas virtuales que se crean y en mi caso fue la configuracion "Default" con todos los puertos abiertos que pese a no ser una buena práctica, se llevó a cabo para no tener en el futuro problemas de conexión ni otros posibles y por último en la sección *Configuración* la cual sirve para escribir scripts que se ejecutan al crearse la máquina y permiten la instalación en esta de diverso software, se utilizó un script que se basó en instalar Docker en la máquina virtual, debido a que esta tecnología es fundamental para la instalación de Rancher.

2.10. RANCHER

Software para equipos que adoptan contenedores. Aborda los desafíos operativos y de seguridad de administrar múltiples clústeres de Kubernetes en cualquier infraestructura, al tiempo que brinda herramientas integradas para ejecutar cargas de trabajo en contenedores. No solo implementa clústeres de Kubernetes sino que también los une con autenticación, control de acceso y observabilidad centralizados.

Su principal competencia es OKD (OpenShift Kubernetes Distribution) la distribución de Kubernetes integrada en Red Hat OpenShift, la cual está optimizada para el desarrollo continuo de aplicaciones, además de que agrega herramientas centradas en las operacioens y el desarrollador además de Kubernetes para permitir el desarrollo rápido de aplicaciones, la implementación y escalados sencillos, y el mantenimiento del ciclo de vida a largo plazo para equipos pequeños y grandes.

Se decidió utilizar Rancher en vez de OKD, debido a varios factores como la facilidad de instalación que presenta, ya que el primero en cuestión de horas se puede tener en marcha un despliegue de toda una aplicación mientras que en el segundo se habla de días. También por el apoyo y soporte que proporciona, ya que mientras que Rancher proporciona soporte de la plataforma y de las herramientas del ecosistema Kubernetes, OKD solo presta soporte de la propia plataforma. El ultimo de los motivos fueron las actualizacioens, siendo en Rancher progresivas y consistentes mientras que OKD presenta numerosos fallos en sus actualizaciones, restablecimientos de versiones previas y demás.

Para el uso de Rancher, se decidió instalar esta tecnología con la ayuda de OpenStack, ya que la idea consistió en utilizar OpenStack como proveedor de infraestructura para que Rancher indique a OpenStack las instancias que debe crear para desplegar los distintos nodos de los clústeres que se desplegarían, así como posibles volúmenes y los distintos elementos que Rancher necesitara para desplegar con efectividad en este proyecto la aplicación basada en contenedores. Más información en su página oficial https://rancher.com/docs/.

2.10.1. INSTALACIÓN Y CONFIGURACIÓN DE RANCHER

Para llevar a cabo la instalación de Rancher, tras crear la máquina virtual comentada en la sección 2.9.1 en la que se instaló Docker, tras comprobar que se ha llevado a cabo la correcta instalación del software comprobando la versión instalada, se utilizó el siguiente comando el cual se encuentra en la propia página web de Kubernetes

```
docker run -d --restart=unless-stopped -p 80:80 -p 443:443 -v
/home/ubuntu/rancher data:/var/lib/rancher rancher/rancher:v2.3.0
```

que su finalidad es levantar un contenedor Docker con la imagen rancher 2.3.0 (la elección de esta versión se debe a que las versiones posteriores, presentan muchas dificultades y trabas a la hora de conectar Rancher con un proyecto OpenStack y las versiones anteriores no lo permitían por lo que se esta versión era la necesaria para llevar a cabo el desarrollo del proyecto) además de utilizar ciertos volúmenes que son necesarios en el despliegue de Rancher.

Figura 9: Despliegue de Rancher en máquina virtual

Una vez desplegado el contenedor con la imagen Rancher, con acceder al puerto 80 de la máquina virtual utilizando la dirección ip de la misma en el buscador web del equipo local, aparece una pantalla de registro de Rancher donde pide un usuario para el usuario *admin* y tras realizar el registro, se accede a un menú que da por finalizado el despliegue de Rancher en la máquina virtual.

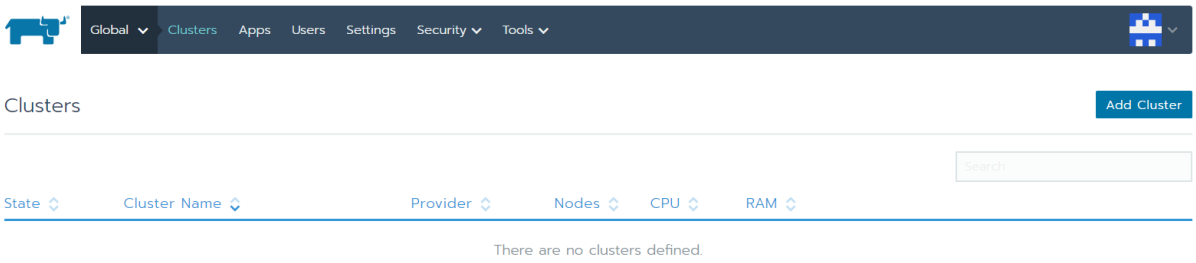

Figura 10: Menú Rancher

Tras esto, falta realizar las distintas configuraciones para que a la hora de crear un cluster de Kubernetes, Rancher acceda, en el caso de este Trabajo de Fin de Grado, a mi proyecto propio de OpenStack en el que se encuentra la máquina virtual que levanta Rancher.

Rancher, de forma predeterminada, ofrece proveedores de Kubernetes como Google Container Engine, Amazon EKS y Azure Kubernetes Service, debido a que estas plataformas ya tienen sus propias configuraciones para desplegar instancias que permiten el despliegue de Kubernetes en estas instancias. Sin embargo, en el caso de OpenStack, al tratarse de un proyecto open source, no dispone de unas plantillas que permitan crear los clústeres de Kubernetes de una forma ya predeterminada y con los distintos valores como zonas de disponibilidad, nombres de las imágenes para crear los nodos y demás datos ya establecidos.

Para lograr desplegar clústeres en OpenStack, hay que activar los conocidos como *Node Drivers* de OpenStack y tras ello, crear plantillas que indicarán a Rancher tanto el proyecto al que deben acceder como las características de las máquinas virtuales que se desplegarán.

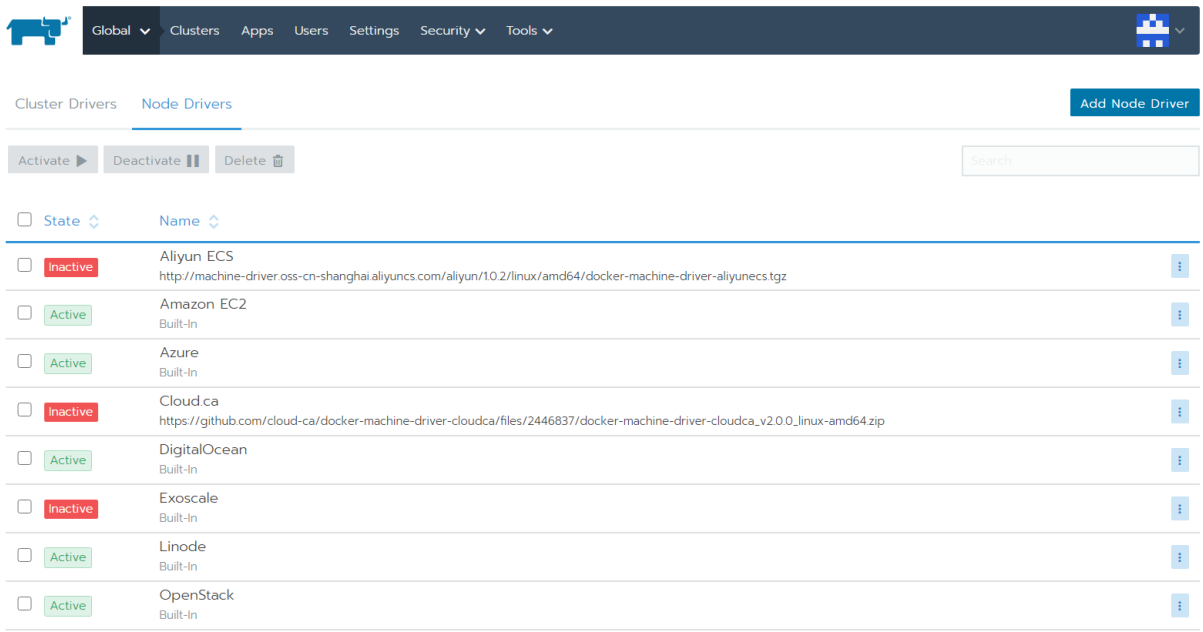

Figura 11: Activación Node Drivers OpenStack

Para la creación de plantillas con las que crear los distintos nodos, en la pestaña de usuario, en la opción "Node Templates" hay que acceder a la función "Add Templates". Tras esto, aparece una ventana de configuración con numerosas opciones a configurar y una lista de proveedores en los que crear la plantilla, de la que hay que seleccionar en este caso el proveedor OpenStack.

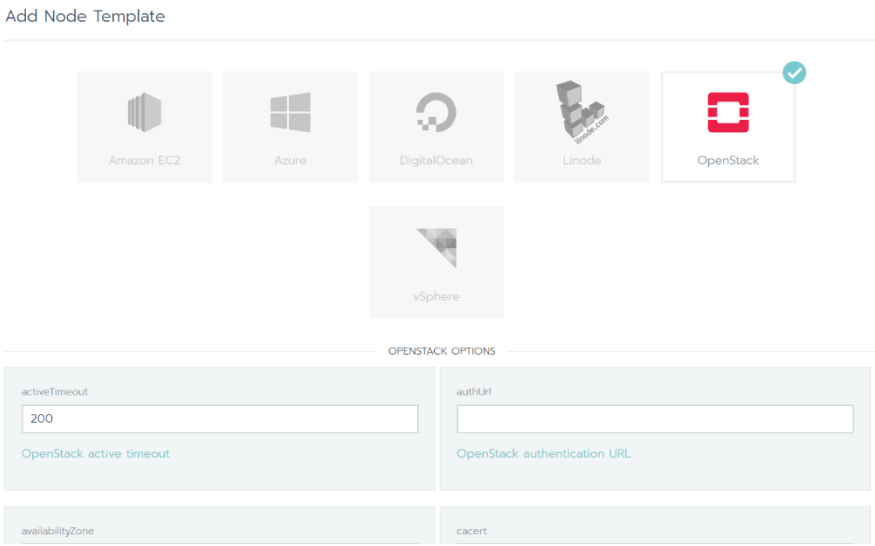

Figura 12: Creación de Node Template

Los distintos parámetros a rellenar para que Rancher cree los distintos nodos de un clúster de Kubernetes en un proyecto OpenStack son los siguientes:

- activeTimeout: Se deja el valor predeterminado de 200.
- authURL: Endpoint público que proporciona la autenticación. Este valor debe ser http://openstack.di.ual.es:5000/v3 (solo accesible desde la VPN de la UAL).

- availabilityZone: Zona de disponibilidad donde se crean las instancias para los nodos Kubernetes que se creen con esta plantilla. La única zona en nuestro OpenStack es “nova”.
- domainName: Nombre del dominio al que pertenece el usuario. El valor que se debe introducir es “default”.
- endPointType: Tipo de endpoint que usaremos el cual debe ser “publicURL”.
- flavorName: Nombre del sabor con el que se crearán los nodos. Esto es a gusto del usuario, pero en mi caso utilicé el sabor “large”.
- floatingipPool: Nombre de la red externa que proporciona las IP flotantes a las instancias creadas. El valor que hay que introducir es “ext-net”.
- imageName: Nombre de la imagen que utilizará para crear las instancias. En este proyecto, la imagen utilizada es Ubuntu 16.04 LTS.
- ipVersion: Se deja al valor predeterminado 4.
- keyparName: Nombre del archivo de clave pública que se utilizará en las instancias que cree y por tanto debe estar disponible en el proyecto OpenStack en el que se van a crear las instancias. En mi caso el nombre es “claveslinux”.
- netName: Nombre de la red a la que se conectarán las instancias que se creen. En mi caso es “alg804-net”.
- password: Contraseña del usuario de OpenStack para poder dar acceso a Rancher y que así cree las instancias.
- privateFileKey: En este caso, hay que insertar todo el contenido del archivo que contiene la clave privada que se utiliza para acceder a las instancias.
- region: El valor a introducir aquí es RegionOne.
- secGroups: Lista de grupos de seguridad del proyecto OpenStack que en mi caso basta con el grupo de seguridad “default”.
- sshPort: Puerto de acceso a las instancias. Se deja el valor predeterminado 22.
- sshUser: Nombre de usuario de la instancia que va a crear. Como la imagen utilizada es Ubuntu, el usuario también es ubuntu.
- tenantName: Nombre del proyecto en el que se crearán las instancias. En este caso es “alg804” pero esto depende de cada usuario.
- userName: Nombre de usuario del proyecto OpenStack. También como en tenantName es “alg804” en este proyecto.

Tras rellenar todos estos parámetros, basta con asignarle un nombre a esta plantilla, que en mi caso utilicé el nombre de la imagen utilizada seguido del sabor elegido con el que se crearían las distintas instancias. Una vez llevado a cabo todos los pasos descritos, se puede crear un cluster de Kubernetes que en el caso de este Trabajo de Fin de Grado, Rancher (desde la máquina virtual creada en el proyecto OpenStack) mediante las plantillas creadas, accede a mi proyecto personal de OpenStack y en él, crea los nodos correspondientes que en este caso son máquinas virtuales con la imagen Ubuntu 16.04 LTS y un sabor “small”.

Para crear el cluster de Kubernetes en Rancher, en la pestaña Clusters del menú, basta con pulsar la opción Add Cluster, donde aparece una serie de configuraciones de las cuales una es el nombre que se le otorga al cluster, y a continuación de este, se configuran los distintos nodos que componen el clúster creado. En estas opciones se le establece un nombre al nodo correspondiente, así como la plantilla que indica cómo crear la instancia que creará el nodo, el número de instancias que habrá y las funciones que llevará a cabo el nodo en cuestión. Hay

tres opciones, *etcd* que hace las labores de base de datos del clúster, *Control Plane* que indica que el nodo creado tomará las funciones del nodo Master de un cluster de Kubernetes y la opción *Worker* que indica al nodo creado que llevará las funciones de los nodos Worker descritas en la sección 2.2. En el caso del cluster creado en este proyecto está en la siguiente figura.

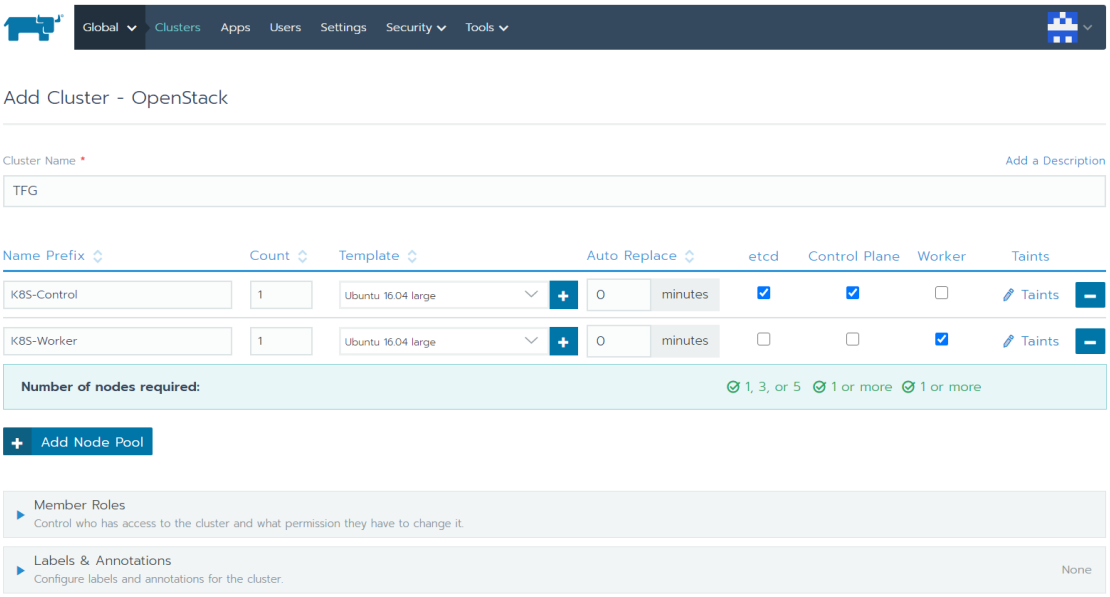

Figura 13: Creación cluster Kubernetes en Rancher

Es recomendado que en la creación del nodo Master, este tenga las funciones de *Control Plane* y las de *etcd* y dejar al resto de nodos los cuales hacen la función del nodo Worker, la opción que posee el mismo nombre.

Tras confirmar los distintos nodos que se van a crear y tras unos minutos en los que Rancher le indica a OpenStack la creación de los distintas máquinas que debe crear, así como el aprovisionamiento de las distintas tecnologías que necesita para llevar a cabo cada nodo sus distintas funcionalidades, se habrá creado el cluster mostrándose en la pantalla principal del mismo un dashboard con distinta información como los recursos tanto de CPU como de Memoria utilizados por los distintos nodos en su conjunto así como los Pods utilizados.

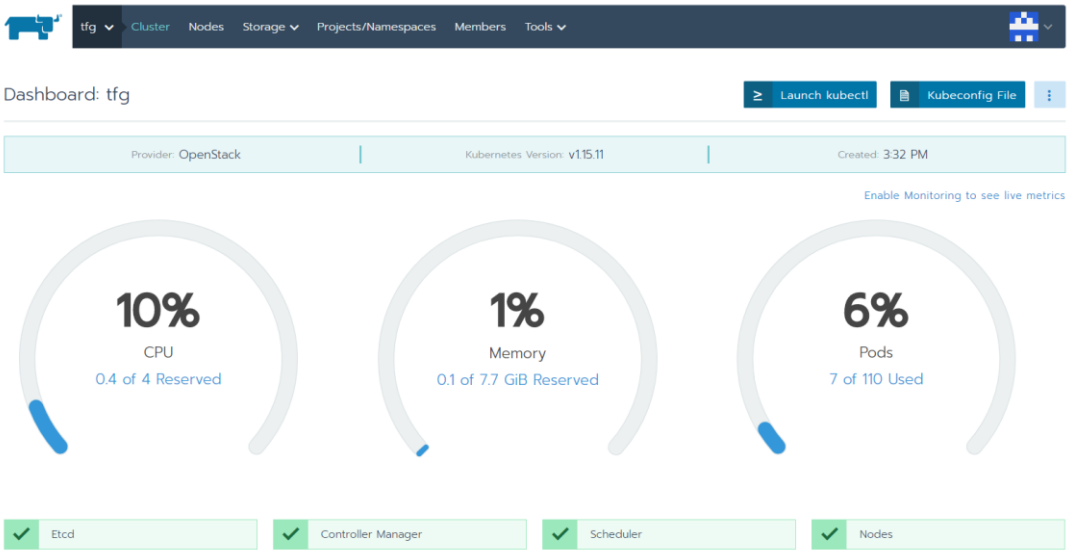

Figura 14: Dashboard de cluster creado

Como se comentó anteriormente, Kubectl permite llevar a cabo tareas en un cluster tanto local como remoto. Para que esta herramienta instalada en un equipo local permita al desarrollador comunicarse con el cluster creado utilizando los distintos comandos descritos en la tabla 2, hay que cargar los "contextos" del cluster en el archivo de configuración de kubectl. Estos contextos son una serie de datos del cluster que permiten que los comandos utilizados en la herramienta hagan referencia al cluster objetivo en concreto. Para conocer los "contextos de un cluster, en la pestaña Kubeconfig File de la figura 14, aparece toda esta información relacionada con el cluster para su conexión con Kubectl.

Esta información la cual se encuentra en la figura 14, se debe colocar en el archivo config de kubectl. Se sustituyen los datos existentes en el archivo config (los cuales hacen refencia a cualquier cluster creado en la herramienta minikube explicada en la sección 2.8) y tras ello, con el comando

```
Kubectl config use-context "nombre del contexto"
```

se aplica el nuevo contexto configurado. Para comprobarlo, con utilizar el comando

```
kubectl config get-contexts
```

se puede comprobar que la herramienta, está conectada con el cluster creado y que en este Trabajo de Fin de Grado se conectó al cluster denominado "TFG" como se ha indicado anteriormente.

```
users:
- name: "tfg"
  user:
    token: "kubeconfig-user-2wk67.c-bww57:nvmnsmrsfmnzb8bfbfvgpv9jqdh8fxf5kx7mpm5qklbbfd8kwhkfls"

contexts:
- name: "tfg"
  context:
    user: "tfg"
    cluster: "tfg"
- name: "tfg-k8s-control1"
  context:
    user: "tfg"
    cluster: "tfg-k8s-control1"

current-context: "tfg"
```

Figura 15: Contexto de cluster creado

Además, para que el cluster pueda proveerse de distintos volúmenes y elementos ingress de OpenStack, hay que llevar a cabo una configuración más del clúster. Para ello, en las configuraciones del propio cluster, accediendo al archivo YAML que contiene toda la información de este, en la línea 22 de la figura 16 hay que rellenar la opción "cloud_provider" con la siguiente información:

```
cloud_provider:
  name: "openstack"
  openstackCloudProvider:
    block_storage:
      ignore-volume-az: true
      trust-device-path: false
    global:
      auth-url: "http://192.168.64.12:5000/v3/"
      domain-name: "default"
      tenant-id: "id del proyecto OpenStack"
      username: "usuario del proyecto OpenStack"
      password: "contraseña del proyecto OpenStack"
    metadata:
      request-timeout: 0
    load_balancer:
      create-monitor: false
      floating-network-id: "id de la red del proyecto"
      manage-security-groups: false
      monitor-max-retries: 0
      subnet-id: "id de la subred del proyecto"
      use-octavia: false
```

Los distintos valores como tenant-id, userName, password, floating-network-id y subnet-id fueron completados con datos correspondientes a mi proyecto de OpenStack para el desarrollo de este Trabajo de Fin de Grado.

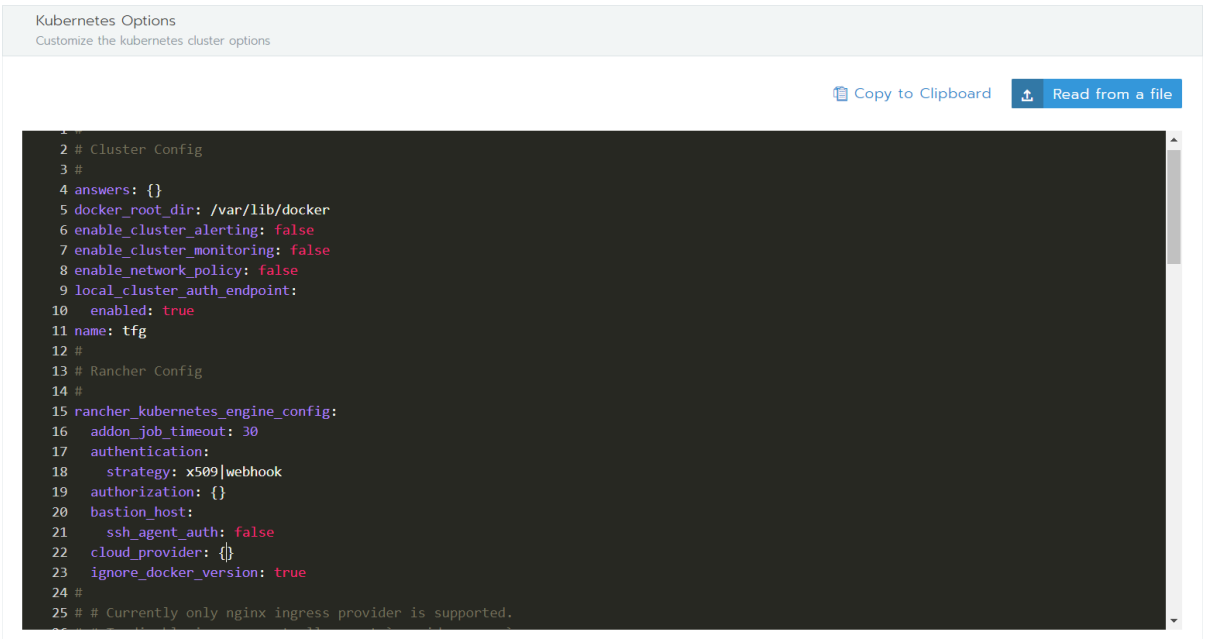

Figura 16: Configuración de volúmenes e ingress del cluster

Tras realizar todas estas configuraciones, se dispone de un cluster que en cualquier momento puede desplegar aplicaciones basadas en contenedores y que puede abastecerse de distintos objetos como volúmenes e ingress [11].

2.11. VISUAL STUDIO

Editor de textos gratuito proporcionado por Microsoft utilizado tanto en el desarrollo del código tanto del front-end como de la API REST, como en la configuración de imágenes Dockerfile, archivos Compose y YAML, y archivo de inicialización de base de datos. Más información en https://code.visualstudio.com/.

2.12. RESUMEN DEL CAPÍTULO

En este segundo capítulo, se ha llevado a cabo una descripción de las tecnologías que se han utilizado y de aspectos importantes de estas (como es el caso de Docker y Kubernetes, la base de este Trabajo de Fin de Grado) y las distintas herramientas que se han utilizado.

Docker y Kubernetes como las tecnologías básicas en las que se fundamenta el desarrollo de este proyecto, que permiten el despliegue de aplicaciones basadas en contenedores y que permitirán la escalabilidad de la aplicación desarrollada.

MySQL como gestor de la base de datos que tendrá la aplicación web, así como Phalcon como framework con el que se desarrolla la API REST que interactúa con la base de datos implementada y tanto Bootstrap como JQuery para aspectos del front-end de la aplicación web como su diseño y la conexión con la API REST respectivamente, todo ello usando Visual Studio como editor de textos.

Para el despliegue de la aplicación, se utilizó OpenStack para levantar la instancia que contiene Rancher y que esta plataforma a su vez se configuró para poder crear el cluster de Kubernetes en OpenStack y desplegar la aplicación en OpenStack. Se ha explicado en este capítulo también, como OpenStack ha servido como proveedor de infraestructura para

Rancher y cómo se ha configurado esta última plataforma para que pudiera crear las instancias necesarias del cluster utilizando máquinas virtuales de OpenStack.

3. DESARROLLO DEL PROYECTO

En este capítulo se expondrán las principales tareas de las que ha constado el proyecto, explicando detalladamente los pasos que se han seguido para completarlas.

3.1. PRESENTACIÓN, INVESTIGACIÓN Y FORMACIÓN PARA EL PROBLEMA

Al comienzo de este proyecto, la primera tarea que se planteó fue el estudio de Docker. Las ventajas del uso de contenedores frente a la virtualización llevada a cabo mediante Hipervisores, apreciar las diferencias de trabajar con contenedores y con máquinas virtuales instaladas en el equipo, la importancia de las imágenes Docker, y el resto de las características comentadas en la sección 2.1. Además, se llevó a cabo la instalación de Docker para realizar pequeñas pruebas y profundizar en los conocimientos de esta tecnología.

Una vez obtenidos los conocimientos sobre los contenedores y la forma de trabajar en Docker, lo primero fue estructurar la aplicación web, planificar los requisitos que debería de cumplir y buscar una idea que se plasmara en una aplicación web que cumpliera los requisitos que se habían propuesto. La propuesta del tutor fue una aplicación web que constara de un back-end y un front-end, en distintos contenedores y que se conectaran entre sí, para llevar a cabo un desarrollo con una arquitectura basada en contenedores Docker. Se decidió que el back-end estaría compuesto por una base de datos y una API REST que pudiera realizar los 4 tipos básicos de peticiones HTTP los cuales son Get, Post, Put y Delete. Como toda aplicación con front-end y back-end, el primer elemento debía conectarse a este segundo otorgando a cualquier usuario que accediera a la aplicación, una web que le permitiera llevar a cabo de forma visual, los distintos tipos de peticiones que permite la API.

Se propusieron diversos temas en los que apoyarse para la temática de la aplicación web, y se decantó por llevar una temática relacionada con el cine. Se propuso la idea de que, en un caso hipotético, la Universidad de Almería abría un cine para el entretenimiento de sus estudiantes y necesita una aplicación web que le permitiese a los usuario poder visualizar las películas que poseen en cartelera, así como sus entradas, poder sacar entradas para visualizar esas películas, modificar estas entradas que obtienen los usuarios y, por último, poder eliminar estas entradas. Así, se dispondría de una aplicación que permite llevar a cabo las 4 peticiones comentadas anteriormente y además tener una aplicación basada en microservicios teniendo dos distintos, el back-end y el front-end, además, de concretar que se debía implementar una segunda versión de esta aplicación que tendría su finalidad y uso en el Complemento del Trabajo de Fin de Grado.

3.2. CONSTRUCCIÓN DEL BACK-END DATABASE

El desarrollo de la aplicación comienza con al elaboración del back-end, principalmente con la base de datos que almacena toda la información que la aplicación de cine utiliza. La investigación de los distintos sistemas gestores de bases de datos dio como resultado la elección de MySQL para la elaboración de esta base de datos.

Como la elección fue MySQL, lo primero que se ha de llevar a cabo es la instalación de los distintos programas que se necesitan para llevar a cabo la creación de la base de datos. Tales instalaciones son el propio MySQL Workbench, MySQL Server y XAMP para poder desarrollar la base de datos.

Aquí entra la primera gran ventaja del uso de contenedores. Todo esto que se acaba de comentar, no solo requiere mucho espacio en nuestro equipo, ya que hay que contar además con la instalación de otro sistema operativo como puede ser Linux con cualquiera de sus versiones Ubuntu, sino que en caso de que por cualquier circunstancia, hubiera que llevar a cabo una replica de esta parte de la aplicación, habría que llevar a cabo copias en nuestro equipo de todo esto que hay que instalar para el desarrollo, despliegue y funcionamiento de este elemento de la aplicación.

Con Docker, no hace falta toda esta inversión de tiempo gracias a los contenedores y a las imágenes Docker. Basta con buscar en el repositorio Docker Hub la palabra MySQL, con la que se obtienen numerosos resultados de imágenes de este sistema gestor de bases de datos y por tanto, pueden ser usados por cualquier persona (que en este proyecto se utilizó la versión 5.7).

Para utilizar una de estas imágenes, basta con (una vez iniciado Docker en un equipo local) escribir en una terminal el comando:

```
docker run -d -p 3306:3306 mysql:5.7
```

Gracias este comando, docker descargará la imagen mysql:5.7 (ya que al ser la primera vez, el repositorio local de imágenes Docker no posee esta imagen) y levantará un contenedor asignandole su puerto 3306 al puerto 3306 de nuestro equipo y este contenedor solo contiene las dependencias necesarias para poder desplegar un servidor mysql y con ello, la base de datos que nuestra aplicación necesita. Con un simple comando desde una terminal, y en cuestión de segundos, se ha desplegado en nuestro equipo un contenedor MySQL con el que llevar a cabo la implementación de la base de datos de la aplicación web de este proyecto.

Como se aprecia, el poder de desarrollo que presenta Docker no se había observado hasta antes, permitiendo en cuestión de segundos, crear desde cero en este caso un servidor MySQL con únicamente las dependencias que necesita este servidor, con lo que se ahorra en tiempo de despliegue, así como en recursos que consume este servidor, los cuales son mucho menores que si se hubiera instalado un sistema operativo completo y se hubieran instalado todas las librerías y software que se necesita para levantar un servidor de estas características.

Sin embargo, existen discrepancias sobre si el uso de contenedores a la hora de llevar a cabo despliegues de bases de datos utilizando esta tecnología es lo más adecuado para sistemas de produccion. Esto está fuera del alcance de este Trabajo de Fin de Grado pero igualmente se utilizó MySQL en un contenedor sólo por cuestiones de desarrollo. Además, si no se utiliza MySQL en un contenedor, basta con cambiar la url de conexión.

3.2.1. DIAGRAMA ENTIDAD-RELACION Y DIAGRAMA RELACIONAL

En el caso de la base de datos de nuestra aplicación web, lo primero que hay que concretar a la hora de crear una base de datos para el back-end de una aplicación, es enumerar los distintos elementos o entidades que esta base de datos tendrá, y como estos elementos se relacionan entre sí, es decir, estructurar la base de datos. Para ello, la elaboración de un diagrama entidad-relación es básica para poder crear más adelante las distintas tablas que conforman la base de datos. Como se comentó en la sección anterior, la aplicación web está basada en un cine, por lo que esta base de datos debe contener distintos elementos como las películas que tiene el cine, así como las distintas funciones que tiene cada película, las entradas

que permiten ir a las distintas funciones, las salas donde se emiten las películas y se ponen en marcha las funciones, y los distintos asientos que hay en una sala.

De esta manera se acaba de enumerar los distintos elementos que conforman la base de datos de este proyecto y su diagrama entidad-relación se aprecia mejor en la figura 17, donde además de llevar a cabo las relaciones entre las distintas entidades, están los distintos atributos de algunas de estas entidades, siendo en el caso de las películas valores como un id, nombre, sinopsis y duración, en las funciones su id, horario y fecha de reproducción, en las entradas su id al igual que en las salas y en el caso de los asientos su propio id, la fila y el número de la butaca.

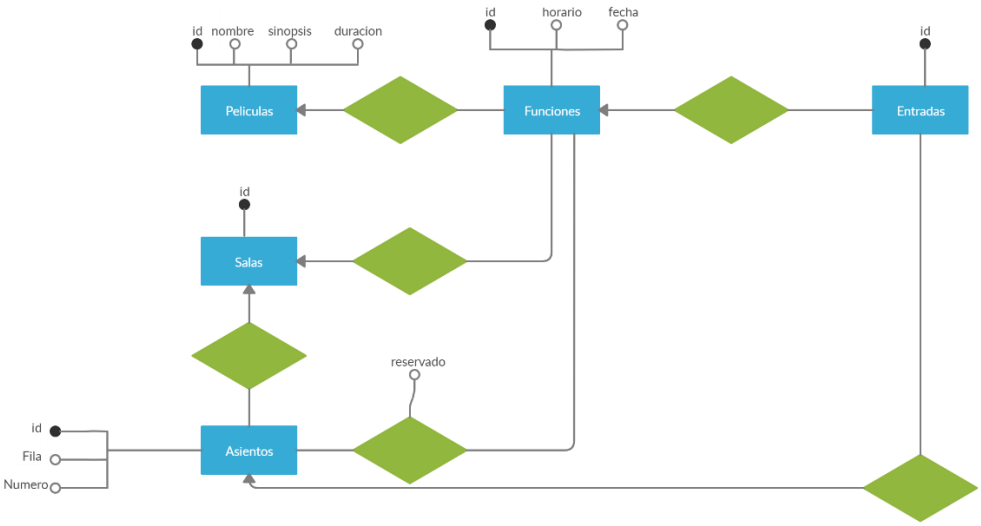

Figura 17: Diagrama entidad-relación

Además se pueden observar las distintas relaciones entre las relaciones, siendo la mayoría de estas relaciones 1:M como es el caso de las películas con las funciones, en las que la primera entidad implica muchas de la segunda, y así ocurre entre las funciones y las entradas, las salas con las funciones, las salas con los asientos y los asientos con las entradas.

Estas relaciones deben ser plasmadas en un diagrama relacional, la cual es la plantilla para la futura implementación de la base de datos. Este diagrama es una simple traducción de las entidades y sus relaciones, a las tablas relacionales de una base de datos con sus respectivas claves primarias y secundarias conodidas como *primary key* y *foreign key* respectivamente.

Para ello, basta con crear una tabla por cada entidad indicada en el diagrama de la figura 7 y añadirle como atributos, cada uno de los que aparecen en el mismo diagrama, estableciendo como *primary key* el atributo id de cada entidad en su correspondiente tabla. Para las relaciones 1:M se debe trasladar la clave primaria de la entidad 1 como *foreign key* a la tabla que actúa como entidad M en la relación como ocurre entre las películas y las funciones, donde una película tiene muchas funciones y una función solo pertenece a una película. Esto se realiza en el resto de las relaciones 1:M que se han establecido como son las relaciones Salas-Funciones, Salas-Asientos, Función-Entradas y Asiento-Entrada. Además, está la relación M:N entre Asiento y Función, ya que un asiento está en varias funciones y cada función tiene varios asientos. Aquí entra en juego la relación que tiene como atributo reservado y que se aprecia en la figura 7 y para trasladar esta relación hay que crear una tabla la cual se denominó Reservas y

sus atributos deben ser las *primary key* de las entidades relacionadas (en este caso tanto Asientos como Funciones).

El resultado de todas estas conversiones para obtener el diagrama relacional se puede observar en la figura 18 donde se ven todas las entidades convertidas en tablas y las distintas relaciones entre estas.

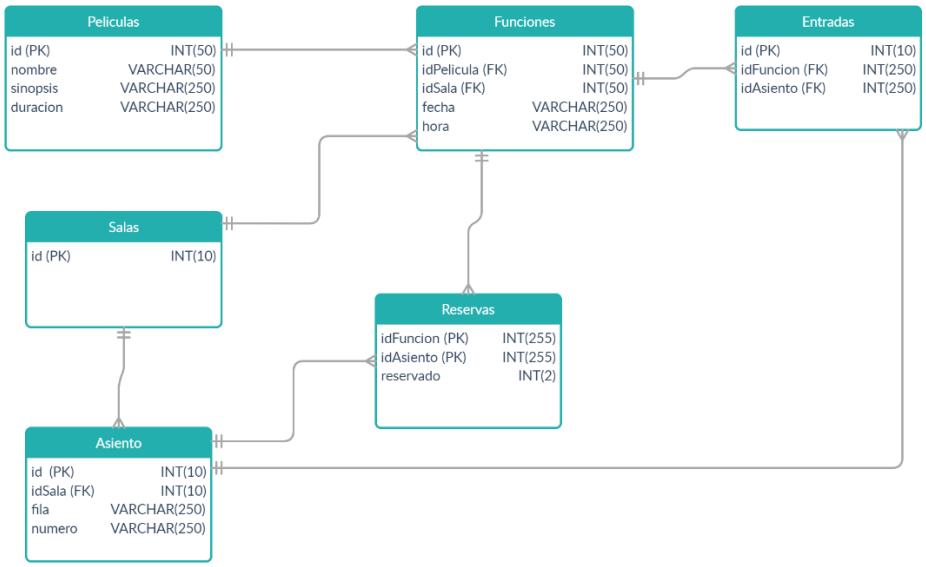

Figura 18: Diagrama relacional

3.2.2. CREACIÓN DE BASE DE DATOS MySQL

El objetivo de este primer desarrollo, es lograr el despliegue de un contenedor MySQL que se inicializa con la creación de la base de datos que se estructuró cuando se llevaron a cabo los diseños de los diagramas entidad-relación y relacional. Para ello, la mejor solucion es preparar un script mysql que cree la base de datos, las distintas tablas y los valores que estas tablas almacenan y anclarlo como un volumen al contenedor en cuestión para que siempre que se inicialice, tenga a su disposición la base de datos creada y nunca se pierda la información.

Para que este despliegue fuera exitoso, se debía de crear la base de datos mediante la sintaxis:

```
CREATE DATABASE "NOMBRE BASE DE DATOS"
```

El nombre utilizado en este proyecto para la base de datos fue "cine". Otro de los elementos a crear son las distintas tablas mediante la sintaxis:

```
CREATE TABLE "NOMBRE TABLA" (ATRIBUTO 1, ATRIBUTO 2…)
```

Indicando en cada atributo sus características como el tipo de atributo que es, así como si pueden o no ser NULL sus valores y demás. Además, en la creación de las tablas, se asigna ya la *primary key* y las distintas *foreign keys* que pueden tener estas con el uso de las dos siguientes líneas de códigos respectivamente:

```
PRIMARY KEY (ATRIBUTO)
```

FOREIGN KEY (ATRIBUTO) REFERENCES "TABLA DE LA QUE RECIBE LA CLAVE"(ATRIBUTO DE LA TABLA REFERENCIADA)

Con estas líneas de código sql, podemos crear la base de datos que se estructuró en el diagrama relacional de la figura 8, y crear así tanto la base de datos *cine* como las tablas *Películas, Funciones, Salas, Entradas, Asientos, Reservas* con sus respectivas *primary key* y *foreign key* y sus demás atributos como se muestra en el fragmento de código 5.

Como se comentó, además de la estructura de las tablas y sus relaciones, también deben contener datos algunas de las entidades que se han implementado, como son las películas que los usuarios pueden ver, las funciones que tienen cada una de estas películas, las salas de las que dispone el cine y los asientos que tienen cada una de las salas. Todos estos valores se introducen mediante la siguiente expresión:

INSERT INTO "TABLA" VALUES (VALOR 1, VALOR 2…)

```sql
DROP DATABASE IF EXISTS cine;
CREATE DATABASE cine;
USE cine;

CREATE TABLE peliculas(
id      INT(50) NOT NULL,
nombre      VARCHAR(50) NOT NULL,
sinopsis    VARCHAR(500)NOT NULL,
duracion    VARCHAR(250)NOT NULL,
PRIMARY KEY(id));

CREATE TABLE salas(
id      INT(10) NOT NULL,
PRIMARY KEY (id));

CREATE TABLE asientos(
id      INT(250) NOT NULL,
fila        VARCHAR(250) NOT NULL,
asiento     VARCHAR(250) NOT NULL,
idSala      INT(10) NOT NULL,
PRIMARY KEY(id),
FOREIGN KEY(idSala) REFERENCES salas(id));

CREATE TABLE funciones(
id      INT(50) NOT NULL,
idPelicula  INT(50) NOT NULL,
idSala      INT(50) NOT NULL,
fecha       VARCHAR(50) NOT NULL,
hora        VARCHAR(50) NOT NULL,
PRIMARY KEY(id),
FOREIGN KEY(idPelicula) REFERENCES peliculas(id),
FOREIGN KEY(idSala) REFERENCES salas(id));

CREATE TABLE reservas(
idFuncion   INT(255) NOT NULL,
idAsiento   INT(255) NOT NULL,
reservado   INT(2));

CREATE TABLE entradas(
id      INT(255) primary key auto_increment NOT NULL ,
idFuncion   INT(255),
idAsiento   INT(255),
FOREIGN KEY(idFuncion) REFERENCES funciones(id),
FOREIGN KEY(idAsiento) REFERENCES asientos(id));
```

Fragmento código 5: Script inicialización base de datos

Un detalle que comentar del script del fragmento de código 5 es el id de la tabla Entradas, el cual tiene la propiedad auto_increment pensando en el futuro usuario que utilice la

aplicación, que permite que, tras la creación de una entrada, automáticamente se asigne el id de forma incremental de uno en uno.

Para desplegar la base de datos en un contenedor, en una terminal, desde el directorio donde se encuentra el archivo de inicialización, bastaba con usar el comando

```
docker run -d-p 3306:3306 -v ./datos:/var/lib/mysql
./init.sql:/docker-entrypoint-initdb.d/init.sql
```

En los volúmenes, se utiliza un *bind mount* que une un directorio denominado "datos" donde se almacenará toda la información con el directorio donde se almacena la información en el contenedor docker mysql y el segundo volumen es el que incluye el archivo de inicialización creado con el directorio que utilizan los contenedores docker para inicializar bases de datos cuando estos tipos de contenedores se despliegan.

Si se desea comprobar la correcta inicialización de la base de datos, basta con comprobar que el contenedor se desplegó con el comando

```
docker ps
```

así como abrir una terminal dentro del contenedor con

```
docker exec -it "nombre/id contenedor" mysql
```

y una vez en él, pedirle que muestre las distintas bases de datos y las tablas que contiene esta. En la figura 19 se observa la comprobación de la base de datos y las distintas tablas que esta posee de la aplicación realizada.

```
PS C:\Users\lopez\onedrive\escritorio\proyecto> docker exec -it bdcine mysql -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.7.29 MySQL Community Server (GPL)

Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| cine               |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
5 rows in set (0.14 sec)

mysql> use cine;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> show tables;
+----------------+
| Tables_in_cine |
+----------------+
| asientos       |
| entradas       |
| funciones      |
| peliculas      |
| reservas       |
| salas          |
+----------------+
6 rows in set (0.00 sec)

mysql>
```

Figura 19: Comprobación funcionamiento base de datos

La implementación de esta base de datos supone el primer acercamiento a esta nueva forma de desarrollar e implementar microservicios basándose en contenedores Docker. Como se ha explicado, de una manera rápida y sencilla, se puede desarrollar una pequeña base de datos y comprobar su funcionamiento sin la necesidad de instalar máquinas virtuales e instalar distintos programas que se necesiten, ya que con desplegar un contenedor que contenga las dependencias y librerías necesarias que en este caso son de MySQL utilizando una imagen Docker que se encuentre en el repositorio Docker Hub (la cual crea el contenedor con estas dependencias necesarias), y elaborar un script de inicialización que se vincula al contenedor mediante un volumen o *bind mount* llevamos a cabo todo el desarrollo que sin los contenedores, ocupa muchos más recursos de los utilizados y tiempo invertido.

3.3. CONSTRUCCIÓN DE LA API Y DOCKER-COMPOSE

Una vez completada la implementación de la base de datos, uno de los tres servicios que componen la aplicación web, tocaba el desarrollo de la API REST, el segundo de estos servicios. Surge un problema en este instante, el cual es que Docker como tal, no plantea ninguna forma de comunicación entre los contenedores.

De esto se encarga Docker Compose, que permite la comunicación entre contenedores utilizando los nombres de estos mismos para establecer las comunicaciones entre ellos, y surge aquí, el despliegue de la aplicación basada en múltiples contenedores. Se podría elaborar un único contenedor que tuviera tanto la base de datos como la API REST, pero en el momento de despliegue de la aplicación, si se necesitara escalar el servicio de la API REST, también se escalaría la base de datos, cosa que no interesa. Esto es solo otra ventaja más que aporta el uso de contenedores y arquitectura basada en microservicios.

Con Docker Compose se definen los distintos servicios que posee la aplicación que esta herramienta va a desplegar. Para ello, se configuró el archivo *docker-compose.yml* en el cual, se definieron los dos primeros servicios, tanto la base de datos, como la API REST y que se puede observar en el fragmento de código 6.

```yaml
version: '2'
services:
  bdcine:
    container_name: bdcine
    restart: always
    image: mysql:5.7
    environment:
      MYSQL_ROOT_PASSWORD: 'root'
    ports:
      - "3306:3306"
    volumes:
      - ./datos:/var/lib/mysql
      - ./init.sql:/docker-entrypoint-initdb.d/init.sql
  api-cine:
    container_name: api-cine
    restart: always
    image: ualmtorres/phalcon-apache-ubuntu
    ports:
      - "81:80"
    volumes:
      - ./api:/var/www/html
```

Fragmento código 6: Docker Compose base de datos y API REST

En el archivo Compose, tras establecer la versión 2 del archivo (la cual le indica a Docker Compose que se trata de un archivo YML) en el apartado `services` indicamos los dos servicios que de momento tendrá la aplicación, los cuales se han denominado para este proyecto tanto "bdcine" como "api-cine" para la base de datos y la API REST respectivamente.

Tras la definición de los servicios, hace falta configurar los contenedores que se desplegarán para cada servicio. En el caso de la base de datos, es trasladar el comando de ejecución que se utilizó en el apartado 3.2.2 a este documento. Para ello, le damos un nombre al contenedor con la sección `container_name` que servirá para comunicarse con otros contenedores pues será su nombre de host (en el proyecto recibió el nombre de "bdcine"). Con la opción `restart` establecida en el valor "always" indicamos que en caso de que falle el contenedor y se elimine, se restablezca. Para indicar que el contenedor se debe desplegar utilizando una imagen determinada, se establece el nombre de esta imagen Docker en la sección `image` que para desplegar un contenedor MySQL se estableció la misma imagen que cuando se construyó la base de datos (mysql:5.7). A los contenedores se les puede añadir variables mediante `environment` y en esta sección, establecer las variables que el desarrollador necesite. En este proyecto, se estableció la contraseña del usuario *root* mediante la variable MYSQL_ROOT_PASSWORD. Los puertos se asignan mediante la sección `ports` indicando los puertos que se enlazarán y por último en `volumes` establecemos los diferentes volúmenes o *bind Mounts* (que en esta aplicación son los mismos que los descritos en el comando de despliegue del contenedor mysql descrito en el apartado 3.2.2).

Para el servicio de la API REST, se llevaron a cabo las mismas configuraciones que las implementadas en el contenedor de la base de datos. Se le estableció el nombre "api-cine", se le indicó que utilizara la imagen *ualmtorres/phalcon-apache-ubuntu* la cual es una imagen aportada por el tutor que crea un contenedor con las dependencias necesarias para desplegar software desarrollado para el framework Phalcon como son dependencias de Ubuntu y apache. Esta imagen resultó de gran ayuda debido a problemas de instalación que surgían a la hora de crear una imagen que tuviera estas dependencias. Se unió el puerto 81 del equipo local con el puerto 80 del contenedor (ya que queríamos comprobar su funcionamiento y más tarde será necesario para la conexión entre el front-end y este servicio) y en los volúmenes se estableció un *bind mount* entre el directorio /var/www/html y el directorio /api el equipo local, en el cual se almacenó todo el desarrollo de la API REST y que se encuentra en el directorio donde se almacena todo el desarrollo de la aplicación.

Una vez establecidos los distintos servicios, el desarrollador puede implementar la API REST teniendo en cuenta que debe llevar a cabo las distintas configuraciones como conexión a base de datos pensando en el contenedor que contiene la base de datos llamada "bdcine".

3.3.1. CONFIGURACIÓN DE LA API REST

Como se comentó en la sección 3.1.1 la API REST debe llevar a cabo 4 tipos distintos de métodos http, los cuales son GET, POST, PUT y DELETE. El uso de la librería Micro hace que la implementación se agilice al permitir el desarrollo de pequeñas aplicaciones con menos código del utilizado habitualmente en este tipo de aplicaciones.

Para llevar a cabo esta API REST se ha seguido el ejemplo otorgado por la propia Phalcon para aprender así las ventajas descritas antes y que se encuentra en su propia página web a modo de tutorial [7].

Lo primero que se llevó a cabo fue dentro del directorio “api” el archivo “.htaccess” el cual se trata de un archivo de configuración que permite el funcionamiento y redireccionamiento de las peticiones http a nuestra API REST.

A continuación, se creó un directorio denominado “Models” donde se creó un archivo por cada tabla que teníamos en la base de datos. Esta carpeta y sus archivos son importantes en la implantación de la API ya que cuando se conecte a la base de datos y lleve a cabo las consultas para llevar a cabo los distintos métodos http, el acceso a las distintas tablas las hará gracias a los modelos que sirven además para acceder a las tablas de una manera orientada a objetos y más ventajas como la posibilidad de añadir validaciones para incluir datos en las tablas correspondientes.

Importante comentar que cada uno de estos archivos que se encuentran en el directorio Models debe heredar la clase `Phalcon\Mvc\Model` y tener un namespace para cuando se implementan las instrucciones sql de los métodos que tiene la API REST.

Tras esto, se creó el archivo index.php en el directorio *api* donde se implementó la API REST con sus endpoints correspondientes. Pero antes de implementar las distintas rutas, se configuró la API, primeramente, incluyendo todas las librerías necesarias para configurar la API. Estas son `Event`, `Manager`, `Loader`, `Micro`, `FactoryDefault`, `Response`, `Mysql` y `Plugin`. De todas estas, Plugin, que como su nombre indica es un plugin que se debio añadir a nuestra API para no tener problemas en el intercambio de recursos de origen cruzado (o más conocido en ingles como CORS policy) que permite que se puedan solicitar recursos desde un dominio fuera del dominio que se solicitó el primer recurso. Sobre todo lo aplicaremos para cuando la API tenga que llevar a cabo una petición tipo PUT o DELETE que en ocasiones el buscador las interpretó como OPTIONS y no llevaba a cabo la petición. Para ello creamos la clase CORSPlugin que hereda de la clase Plugin y creamos la condición de que si el método es OPTIONS que permita que se lleve a cabo los otros tipos de método.

La librería `FactoryDefault` crea objetos con su mismo nombre y sirven para configurar distintos aspectos de la aplicación, como la conexión a la base de datos que hace la API REST. Para ello, basta con crear un objeto de esta clase y establecer los distintos parámetros de conexión con la base de datos, teniendo en cuenta las configuraciones tanto del contenedor que despliega la base de datos como la propia base de datos que hay en el contenedor. Así, en estas configuraciones de la API se establecen tanto el host al que se debe conectar (para ello debemos utilizar el nombre del contenedor), así como el usuario, contraseña y el nombre de la base de datos a la que se va a conectar la API. Usar las contraseñas como tal y demás valores en el código es una mala práctica pero se llevó a cabo para conocer cómo llevar a cabo la conexión en este framework. Toda esta conexión se aprecia de forma visual en el fragmento de código 7.

Esta forma tan sencilla de conectar ambos servicios es solo otro ejemplo más de las ventajas que supone el desarrollo basado en contenedores, ya que si hubieramos llevado a cabo esta implantación mediante máquinas virtuales, deberíamos haber desplegado dos máquinas virtuales, una con la base de datos con su respectivo software y otra con un sistema operativo Linux, más concretamente Ubuntu, con la instalación de Apache y el añadido del framework Phalcon. Además, la propia conexión entre máquinas virtuales habría resultado mucho más compleja, ya que se deberían haber configurado conexiónes entre máquinas virtuales distintas lo que implica más pasos que simplemente configurar un archivo Compose y en la API REST, indicar el nombre del contenedor como tal.

```php
$di = new FactoryDefault();

$di->set(
    'db',
    function () {
        return new PdoMysql(
            [
                'host'     => 'bdcine',
                'username' => 'root',
                'password' => 'root',
                'dbname'   => 'cine',
            ]
        );
    }
);
```

Fragmento código 7: Configuración conexión API REST con base de datos

Hasta aquí, la API REST está configurada para conectarse con el contenedor que despliega el servicio de la base de datos, además de evitar futuros problemas de intercambio de origen cruzado de datos que se solucionó con la implementación del Plugin. A partir de aquí, solo queda implementar los distintos métodos de la API REST con sus respectivos endpoints.

3.3.2. MÉTODOS DE LA API

El siguiente y último paso en la implementación de la API REST consistió en el desarrollo de los distintos métodos que interactúan con la base de datos[6,7]. Los métodos de la API tienen su url y argumentos y antes de pasar a la implementación de estos, en la tabla 3 se muestra un resumen de todos los endpoints creados.

url	Tipo petición	Argumentos	Descripción
/api/películas	GET	Ninguno	Obtiene todas las películas y sus características como título, sinopsis y duración
/api/entradas	GET	Ninguno	Obtiene todas las entradas con su información (id de la función e id del asiento)
/api/entradas	POST	Ninguno	Introduce una entrada en la tabla Entradas de la base de datos
/api/entradas/{id}	PUT	Id de la entrada a modificar	Modifica una entrada creada la cual se indica con el parámetro id.
/api/entradas/{id}	DELETE	Id de la entrada a modificar	Elimina la entrada que se indica con el id de la que se desea eliminar.

Tabla 3: Endpoints API REST

3.3.2.1. GET

Como ya se comentó en secciones anteriores, con este método obtenemos información de la base de datos que se le muestra al usuario. En el caso de la aplicación web del cine de la universidad, se necesitaron dos funciones GET. Como se explicaron brevemente en la tabla 3, el primer método sirve para mostrarle al usuario las distintas películas que se ofertan en el cine y el segundo método, muestra las entradas de las que dispone actualmente.

Para llevar a cabo la primera función, lo primero fue crear el método get asignandole una ruta. Como el objetivo del método consistía en mostrar las películas, la ruta que se definió fue "/api/peliculas". Se siguió con la creación de la función que se encargaba de recuperar las películas que se pueden visionar en el cine. Para ello, se creó una variable $phql la cual nos sirve dialecto a MySQL para la comunicación con la base de datos y se escribió en ella una consulta donde pedimos que nos devuelva todos los valores de la tabla peliculas, pero gracias al uso de los modelos, para referenciarnos a la tabla peliculas debemos utilizar el namespaces que habíamos añadido en los distintos archivos del directorio models, seguido del nombre de la tabla de la que vamos a obtener la información y por tanto la consulta quedaría tal que así:

```
SELECT * FROM MyApp\Model\Peliculas;
```

En una variable denominada $peliculas se guardó el resultado de la ejecución de la consulta con la función `executeQuery($phql)` de la funcionalidad de ModelsManager de la aplicación (esto lo utilizaremos en todas las funciones pues es la forma de ejecutar las distintas consultas) y en otra variable que se denominó $data que se trata de un array, guardó en él cada película gracias a un foreach que recorre cada película de todas las obtenidas en la variable $películas. Por último, tendremos que mostrarlas en formato JSON con la función `json_encode($data)`. El código de esta implementación está en el fragmento de código 8.

```php
$app->get(
    '/api/peliculas',
    function () use ($app) {
        $phql = 'SELECT * FROM MyApp\Model\Peliculas';

        $peliculas = $app->modelsManager->executeQuery($phql);

        $data = [];

        foreach ($peliculas as $pelicula) {
            $data[] = [
                'id'    => $pelicula->id,
                'nombre' => $pelicula->nombre,
                'sinopsis' => $pelicula->sinopsis,
                'duracion' => $pelicula->duracion,
            ];
        }

        echo json_encode($data);
    }
);
```

Fragmento código 8: GET obtención de películas

Para la función que devuelve las entradas que posee un usuario se llevaron a cabo los mismos pasos, con la diferencia de que en la consulta sql se cambió para que devolviera todas las columnas de la tabla entradas por lo que la parte FROM debe ser MyApp\Model\Entradas quedando la consulta tal que así:

```
SELECT * FROM MyApp\Model\Entradas;
```

Además, la variable donde se guardan los resultados de las entradas, en vez de llamarse $peliculas se llamó $entradas. Además se cambió la ruta que para este método pasó a ser "/api/entradas". Usar el código SQL como se está llevando a cabo en los métodos es susceptible de recibir ataques de inyección de código SQL, pero se ha llevado a cabo de esta debido a la falta de conocimiento para llevar a cabo esta acción de diferente forma.

3.3.2.2. POST

El objetivo de estos métodos consiste en introducir valores a una o varias tablas de la base de datos a la que se conecta la API REST. En este proyecto, este método sirve para que el usuario pueda introducir los datos de una entrada que quiera obtener.

En la implementación de este metodo lo primero fue asignarle la ruta la cual es la misma que para el último método get explicado ("/api/entradas"). En este método, como el objetivo no es mostrar datos sino introducir, la consulta SQL debe tener la sintaxis del tipo `INSERT INTO` como las utilizadas en la creación de la base de datos en la sección 3.2.2.

Como el objetivo era añadir una entrada, se decidió en que el usuario pudiera escribir el id de la función a la que desea ir y el id del asiento que quiere ocupar. Estos valores se toma en formato JSON, y en la consulta se deben establecer estos valores por parámetros lo que hace que la consulta tenga la siguiente forma que se puede apreciar además en el fragmento de código 9:

```
INSERT INTO "TABLA" (COLUMNAS) VALUES (:VALOR COLUMNA 1:, :VALOR COLUMNA 2: …);
```

```php
$app->post(
    '/api/entradas',
    function () use ($app) {
        $entrada = $app->request->getJsonRawBody();
        $phql  = 'INSERT INTO MyApp\Model\Entradas '
               . '(idFuncion, idAsiento) '
               . 'VALUES '
               . '(:idFuncion:, :idAsiento:)'
        ;

        $status = $app
            ->modelsManager
            ->executeQuery(
                $phql,
                [
                    'idFuncion' => $entrada->idFuncion,
                    'idAsiento' => $entrada->idAsiento,
                ]
            )
        ;
```

Fragmento código 9: POST introducir entradas

Además, se decidió a crear una respuesta para este método. Para ello, se utilizó la clase `Response` y se creó con variable que se llamó igual que la clase. Esta respuesta es de dos tipos, una en la que se envia la entrada que se ha creado como muestra de que ha sido un éxito el metodo (estableciendo un estado 201) y en caso contrario en el que no saliera bien, se establece el estado número 409 el cual hace referencia a que hay algún conflicto, creandole un mensaje con todos los errores que sonn los que han provocado el error en la petición. Toda esta implementación es aprecia mejor en el fragmento de código 10.

```
140         $response = new Response();
141
142         if ($status->success() === true) {
143             $response->setStatusCode(201, 'Created');
144
145             $entrada->id = $status->getModel()->id;
146
147             $response->setJsonContent(
148                 [
149                  $entrada
150                 ]
151             );
152         } else {
153             $response->setStatusCode(409, 'Conflict');
154
155             $errors = [];
156             foreach ($status->getMessages() as $message) {
157                 $errors[] = $message->getMessage();
158             }
159
160             $response->setJsonContent(
161                 [
162                     'status'   => 'ERROR',
163                     'messages' => $errors,
164                 ]
165             );
166         }
167
168         return $response;
169     }
170 );
```

Fragmento código 10: Respuesta método POST

3.3.2.3. PUT

En este método, el objetivo consiste en modificar una entrada por los motivos que el usuario de la aplicación web considere oportuno, por lo que el primer paso consiste en determinar la ruta, que es la misma que se usó en todas las rutas relacionadas con las entradas "/api/entradas/{id:[0-9]+}". Como se aprecia, hay un añadido a la ruta y esto surge debido a la necesidad de indicar qué entrada es la que se va a modificar por lo que se establece el parámetro del id y para ello se añade esa expresión.

Lo siguiente fue la adición de la consulta SQL que en este caso es del tipo UPDATE. Para ello en la consulta debemos decir que actualice en la tabla entradas, la entrada que tenga la id

que le indiquemos en la ruta, con los nuevos valores que deseemos por lo que estos valores se pasarán por parámetros. Por ello, la consulta tiene la siguiente forma:

```
UPDATE MyApp\Model\Entradas
SET idFuncion = :idFuncion: , idAsiento = :idAsiento:
WHERE id = :id:;
```

Esta consulta se ejecuta con la función `executeQuery` que ya hemos utilizado en los métodos anteriores y como se implementó en el método POST lo cual se puede apreciar en el fragmento de código 11, se creó una respuesta que indica si la modificación se ha llevado a cabo con éxito o si ha fracasado.

```
172  $app->put(
173      '/api/entradas/{id:[0-9]+}',
174      function ($id) use ($app) {
175          $entrada = $app->request->getJsonRawBody();
176          $phql  = 'UPDATE MyApp\Model\Entradas '
177                 . 'SET idFuncion = :idFuncion:, idAsiento = :idAsiento:'
178                 . 'WHERE id = :id:';
179
180          $status = $app
181              ->modelsManager
182              ->executeQuery(
183                  $phql,
184                  [
185                      'id'    => $id,
186                      'idFuncion' => $entrada->idFuncion,
187                      'idAsiento' => $entrada->idAsiento,
188                  ]
189              );
```

Fragmento código 11: PUT modificar entrada

La estructura de esta respuesta es identica a la del método anterior, con dos condiciones tanto de si se lleva a cabo de forma efectiva como si falla en alguna parte y no se lleva a cabo. Esta respuesta tiene la misma estructura y funcionamiento que la descrita en el fragmento de código 10.

3.3.2.4. DELETE

Para el último método, el objetivo es permitir eliminar una entrada elige por su id. Por este hecho, la ruta de este método debe tener la misma estructura que la del método creado justo antes, es decir "/api/entradas/{id:[0-9]+}". Además, como lo que se desea es eliminar un registro de la tabla entradas, la consulta SQL debe ser de la forma DELETE y como se acaba de comentar, será la entrada que indiquemos por su id, por lo que la consulta SQL debe ser:

```
DELETE
FROM MyApp\Model\Entradas
WHERE id = :id:;
```

Como hemos ha implementado en todos los casos anteriores, se ejecuta la consulta con la función `executeQuery` y como se ha desarrollado en los dos últimos métodos, se ha implementado también una respuesta que indica si la ejecución del método ha sido exitosa o errónea. Como en los casos anteriores, envía una respuesta afirmativa si ha sido un éxito la

eliminación y en caso contrario, un mensaje de error con los distintos errores que han surgido en el proceso de eliminacion de la entrada. El código del metodo sin las respuestas se encuentra en el fragmento de código 12.

```
220  $app->delete(
221      '/api/entradas/{id:[0-9]+}',
222      function ($id) use ($app) {
223          $phql = 'DELETE '
224                . 'FROM MyApp\Model\Entradas '
225                . 'WHERE id = :id:';
226
227          $status = $app
228              ->modelsManager
229              ->executeQuery(
230                  $phql,
231                  [
232                      'id' => $id,
233                  ]
234              )
235          ;
```

Fragmento código 12: DELETE eliminar entrada

3.3.2.5. NOT FOUND Y CIERRE DE LA API

Para terminar con la implementación de la API también se añadió un método el cual salta cuando la ruta que se escribe en el buscador no coincide con ninguna de las rutas implementadas en la API REST. Para este metodo simplemente devuelve el mensaje "Nada por aquí. Vuelve a otra página...", además establece el estado 404 "NOT FOUND" tan conocido cuando no funciona algunas páginas web o tienen algún problema. Además, ninguno de los métodos desarrollados funcionará si no se establece en la aplicación la función `handle()` gracias a la cual la aplicación pone en marcha sus distintos métodos. El código de todo es el mostrado en el fragmento de código 13.

```
265  $app->notFound(
266      function () use ($app) {
267          $app->response->setStatusCode(404, 'Not Found');
268          $app->response->sendHeaders();
269
270          $message = 'Nothing to see here. Move along....';
271          $app->response->setContent($message);
272          $app->response->send();
273      }
274  );
```

Fragmento código 13: NOT FOUND

3.3.3. SEGUNDA VERSIÓN API REST

Antes de pasar a la implementación del tercer servicio, el front-end que consume esta API, se implementó también la segunda versión de la API. Para ello, en un nuevo directorio denominado "proyecto2" se almacenó esta segunda versión de la aplicación para tener separadas ambas versiones (la primera versión de la aplicación la almacenamos en el directorio

"proyecto"). La diferencia está en el contenido del archivo index.php, donde se modifican unos métodos y se crean otros nuevos. Todos los endpoints de esta segunda versión son los que aparecen en la tabla 4.

url	Tipo petición	Argumentos	Descripción
/api/peliculas	GET	Ninguno	Muestra todas las funciones de todas las películas.
/api/peliculas/fechas	GET	Ninguno	Muestra todas las fechas almacenadas en la base de datos.
/api/peliculas/search/{fecha}	GET	Fecha de la función.	Muestra las películas según la función y con más información que en la versión anterior
/api/asientos/{id}	GET	Id de la función	Muestra los asientos de una función.
/api/entradas	GET	Ninguno	Muestra las entradas con más información que en la versión anterior.
/api/entradas	POST	Ninguno	Introduce entradas en la base de datos.
/api/entradas/{id}	PUT	Id de la entrada a modificar.	Modifica una entrada existente de la base de datos.
/api/entradas/{id}	DELETE	Id de la entrada a eliminar.	Elimina una entrada existente de la base de datos.

Tabla 4: Endpoints API REST segunda versión

En esta segunda versión, el objetivo fue implementar nuevas funcionalidades como la posibilidad de mostrar los asientos de una función, así como las películas según la fecha en la que se quieran visionar y las propias fechas de las películas, además de las entradas pero con más información. Estas nuevas funcionalidades están enfocadas a la segunda versión del front-end y facilitar con estos cambios la interacción de la aplicación web de cara al usuario que la utilice.

El resto de métodos POST, PUT y DELETE son exactamente iguales entre las dos versiones. Sabiendo todo esto, el primer GET que se implementó fue aquel que obtiene todas las fechas en las que se emiten las distintas funciones de las películas. Para ello, se asigno la ruta "/api/peliculas/fechas" y se implementó una consulta SQL en la que se selecciona solamente la fecha de las distintas funciones. Como se repiten los valores de las fechas, solo necesitamos un valor de cada fecha, siendo la consulta:

```
SELECT DISTINCT fecha FROM MyApp\Model\Funciones
```

Ejecutamos la consulta y los valores los guardamos en una variable llamada `$fechas` que como con las películas y entradas, guardamos en otra variable llamada $data cada fecha obtenida y lo mostramos en formato JSON con la función `json_enconde()` como se observa en el fragmento de código 14.

```php
//obtener las fechas
$app->get(
    '/api/peliculas/fechas',
    function() use($app){
        $phql = 'SELECT DISTINCT fecha FROM MyApp\Model\Funciones';

        $fechas = $app->modelsManager->executeQuery($phql);

        $data = [];

        foreach($fechas as $fecha){
            $data[] = [
                'fecha' => $fecha->fecha,
            ];
        }

        echo json_encode($data);
    }
);
```

Fragmento código 14: GET obtención fechas

Otro de los nuevo métodos es el de mostrar las películas según la fecha que se elija. Para ello, se modificó el método GET que elaboramos anteriormente sobre la obtención de películas haciendo que se obtengan según la fecha por lo que la ruta de este método se decidió que fuese "/api/peliculas/search/{fecha}". En este caso, debemos hacer JOINs en las consultas para llevar a cabo este método.

Un `JOIN` es el motivo por el que en la base de datos, las tablas tenían claves ajenas y es la forma con la que se combinan registros de una o varias tablas. Para esta consulta, debemos hacer un `JOIN` entre las tablas peliculas y funciones pero solo nos deben devolver aquellos registros en los que coincidan el id de la película. Hay varios tipos de `JOIN` pero en nuestro caso utilizamos `INNER JOIN` el cual es la sentencia `JOIN` por defecto. Además la consulta debe tener en `WHERE` la condición de que la fecha coincida con la que escribimos en la ruta y por tanto se usa com parámetro.

Además en el `SELECT`, en vez de obtener todas las columnas como llevabamos a cabo anteriormente, en esta ocasión solo nos interesa el nombre de la película, su sinopsis, duración, el id de la función, el id de la sala donde se proyecta, la fecha y la hora a la que se emite.

Una vez se ejecute la consulta pasandole por parámetro la fecha que debe coincidir para saber que películas se emiten en tal día, debemos guardar esas películas en una variable que llamaremos `$peliculas` y en otra `$data` guardaremos cada pelicula para que con la función `json_encode()` nos las devuelva en formato JSON. Todo este método se aprecia de forma visual en el fragmento de código 15.

```php
//obtener peliculas por fechas
$app->get(
    '/api/peliculas/search/{fecha}',
    function ($fecha) use ($app) {
        $phql = 'SELECT peliculas.nombre, peliculas.sinopsis, peliculas.duracion,
                        funciones.id, funciones.idSala, funciones.fecha, funciones.hora
                FROM MyApp\Model\Peliculas as peliculas
                INNER JOIN MyApp\Model\Funciones as funciones on peliculas.id = funciones.idPelicula
                WHERE fecha LIKE :fecha:';

        $peliculas = $app->modelsManager->executeQuery(
            $phql,
            [
                'fecha' => $fecha
            ]
        );

        $data = [];

        foreach ($peliculas as $pelicula) {
            $data[] = [
                'nombre' => $pelicula->nombre,
                'sinopsis' => $pelicula->sinopsis,
                'duracion' => $pelicula->duracion,
                'idFuncion' => $pelicula->id,
                'idSala' => $pelicula->idSala,
                'fecha' => $pelicula->fecha,
                'hora' => $pelicula->hora
            ];
        }

        echo json_encode($data);
    }
);
```

Fragmento código 15: GET obtención películas según fechas

Otro de los nuevos método que surgió en esta nueva versión, es la posibilidad de mostrar los asientos según la función que se elija. Para este nuevo GET se estableció la ruta "/api/asientos/{id:[0-9]+}" cuyo parámetro que aparece hace referencia al id de la función elegida. En este caso, la sentencia SQL siguiendo el diagrama relacional que creamos, debe contener varios `JOIN` que deben unir las tablas funciones, salas y asientos, donde la condición de estas uniones es la coincidencia de los atributos identificación (*id*) entre estas tablas usando sus claves ajenas correspondientes. Además en el `SELECT` solo queremos que nos devuelva el id del asiento, su fila y su asiento y la consulta debe devolver los asientos de la función que se le indica por lo que se debe añadir una condición `WHERE` en la que se indica que devuelva todos los asientos según el id de la función. Con todo esto, la consulta debería tener la siguiente forma:

```
SELECT asientos.id, asientos.fila, asientos.asiento
FROM  MyApp\Model\Funciones as funciones
INNER JOIN MyApp\Model\Salas as salas on funciones.idSala
 = salas.id
INNER JOIN MyApp\Model\Asientos as asientos on salas.id=a
sientos.idSala
WHERE funciones.id LIKE :id: ;
```

Tras la ejecución de la consulta, la función como en todos los casos anteriores, almacena el resultado de los asientos que devuelve en una variable, y en otra, a través de un foreach almacenamos cada asiento para mostrarla en formato json con la función `json_encode()`. El código de este método está en el fragmento de código 16.

```php
$app->get(
    '/api/asientos/{id:[0-9]+}',
    function ($id) use ($app) {
        $phql = 'SELECT asientos.id, asientos.fila, asientos.asiento
                 FROM  MyApp\Model\Funciones as funciones
                 INNER JOIN MyApp\Model\Salas as salas on funciones.idSala = salas.id
                 INNER JOIN MyApp\Model\Asientos as asientos on salas.id=asientos.idSala
                 WHERE funciones.id LIKE :id:';

        $asientos = $app->modelsManager->executeQuery(
            $phql,
            [
                'id' => $id
            ]
        );

        $data = [];

        foreach ($asientos as $asiento) {
            $data[] = [
                'id' => $asiento->id,
                'fila' => $asiento->fila,
                'asiento' => $asiento->asiento,
            ];
        }

        echo json_encode($data);
    }
);
```

Fragmento código 16: GET asientos según función

Por último, esta nueva versión de la API presenta una modificación del GET que obtenía las entradas, mostrando ahora más información sobre estas como información del asiento que se ha elegido, así como más información de la función mostrando características como el nombre de la película que se va a visionar, la fecha y la hora. Por ello, en esta modificación, se añadieron varios `JOIN` que unen las tablas entradas, funciones, películas y asientos coincidiendo el atributo id de las claves ajenas y primarias correspondientes a cada tabla. La consulta presenta por tanto la siguiente forma:

```
SELECT entradas.id, entradas.idFuncion, entradas.idAsiento,
funciones.fecha,funciones.hora, peliculas.nombre ,asientos.fila,
asientos.asiento
FROM MyApp\Model\Entradas as entradas
INNER JOIN MyApp\Model\Funciones as funciones ON entradas.idFuncion=
funciones.id
INNER JOIN MyApp\Model\Peliculas as peliculas ON
funciones.idPelicula=peliculas.id
INNER JOIN MyApp\Model\Asientos as asientos ON
entradas.idAsiento=asientos.id
```

Tras esto, el resto del método sigue igual que antes simplemente que se añadió en la variable $data, donde guardabamos cada entrada que nos devolvía la consulta; más información, concretamente toda la que seleccionamos en el `SELECT`. El código de este método modificado está plasmado en el fragmento de código 17.

```php
//obtener todas las entradas
$app->get(
    '/api/entradas',
    function () use ($app) {
        $phql = 'SELECT entradas.id, entradas.idFuncion, entradas.idAsiento,
                        funciones.fecha, funciones.hora, peliculas.nombre ,asientos.fila, asientos.asiento
                FROM MyApp\Model\Entradas as entradas
                INNER JOIN MyApp\Model\Funciones as funciones ON entradas.idFuncion=funciones.id
                INNER JOIN MyApp\Model\Peliculas as peliculas ON funciones.idPelicula=peliculas.id
                INNER JOIN MyApp\Model\Asientos as asientos ON entradas.idAsiento=asientos.id';

        $entradas = $app->modelsManager->executeQuery($phql);

        $data = [];

        foreach($entradas as $entrada){
            $data[] = [
                'id' => $entrada->id,
                'idFuncion' => $entrada->idFuncion,
                'idAsiento' => $entrada->idAsiento,
                'nombre' => $entrada->nombre,
                'fecha' => $entrada->fecha,
                'hora' => $entrada->hora,
                'fila' => $entrada->fila,
                'asiento'=> $entrada->asiento,
            ];
        }

        echo json_encode($data);
    }
);
```

Fragmento código 17: GET obtención entradas segunda versión

3.3.4. DESPLIEGUE Y FUNCIONAMIENTO DE SERVICIOS

En el desarrollo de microservicios, desplegar la base de datos creada y esta API REST para comprobar su funcionamiento sin contenedores, utilizando virtualizaciones como máquinas virtuales (lo cual es contra lo que se está observando las diferencias en realizar el desarrollo mediante contenedores) supondría ahora la necesidad de desplegar dos de estas máquinas, una para la base de datos y otra para la API REST.

Esto supone principalmente, un coste de recursos demasiado elevado, ya que se llevaría a cabo la ejecución de dos máquinas virtuales para ejecutar simplemente un archivo script de inicialización de base de datos MySQL y en otra máquina virtual, un archivo index.php, un archivo de configuración y un par de archivos más (los Models que hacían referencia a las tablas) que ocupan una decena de líneas de código. Un gasto de recursos demasiado elevado, sin contar también con la unión de ambas máquinas virtuales para que se conecten ambos servicios, lo cual puede suponer diferentes errores de conexión y demás.

Con el uso de los contenedores, y Docker Compose, todo este proceso de despliegue se resume en ejecutar el archivo Compose que se implementó y comentó en la sección 3.3 donde se indicaban los distintos servicios de esta aplicación web, tanto la base de datos como la API REST, y se configuraban cómo se debían desplegar los contenedores (eligiendo las imágenes Docker con las que crearse los contenedores, puertos de conexión entre nuestro equipo y los de los contenedores y volúmenes donde se encuentran los archivos necesarios para el funcionamiento de los servicios).

Disponiendo de las implementaciones propias de cada servicio siendo estas el script de inicialización de la base de datos y el directorio *api* donde hemos guardado la implementación de la API REST descrita en estas últimas secciones, basta con ejecutar el archivo Compose para

que, en cuestión de segundos, se levanten los contenedores con la configuración establecida y se pueda comprobar el funcionamiento de los servicios desarrollados.

Las diferencias entre estos dos tipos de virtualización, por tanto, son más que evidentes, y se empieza a ver las grandes virtudes que presenta esta tecnología de contenedores software que se comentaban en la sección 2.1 en la que se destacaba el poco uso de recursos de estos contenedores debido a la reducción de software a las dependencias únicas y necesarias para el funcionamiento de las implementaciones realizadas, así como los ahorros de tiempo en la puesta en marcha de estas aplicaciones y demás ventajas a las que tanto se están haciendo referencia.

Usando la aplicación web que se desarrolló en este proyecto, para ver el despliegue de estos servicios fue necesario usar una terminal, viajar al directorio donde se encuentra el archivo docker-compose.yml y el resto de los archivos como el script de inicialización y el directorio *api* que contiene todos los elementos y usar el comando

```
docker-compose up -d.
```

Tras unos segundos, con el uso del comando

```
docker ps
```

se puede comprobar como los dos contenedores que levantan cada uno su servicio, se encuentran en marcha como se ve en la figura 20.

```
PS C:\Users\lopez\onedrive\escritorio\proyecto> docker-compose up -d
Creating network "proyecto_default" with the default driver
Creating bdcine   ... done
Creating api-cine ... done
PS C:\Users\lopez\onedrive\escritorio\proyecto> docker ps
CONTAINER ID        IMAGE                           COMMAND                  CREATED             STATUS                         PORTS                               NAMES
67473906cc6d        mysql:5.7                       "docker-entrypoint.s…"   3 seconds ago       Up 1 second                    0.0.0.0:3306->3306/tcp, 33060/tcp   bdcine
9010a0c37366        ualmtorres/phalcon-apache-ubuntu   "/bin/sh -c 'apachec…"   3 seconds ago       Up 1 second (health: starting)   443/tcp, 0.0.0.0:81->80/tcp         api-cine
```

Figura 20: Despliegue servicios base de datos y API REST

Pero no basta con comprobar que se han desplegado los contenedores. Como todo desarrollo software, se ha de comprobar el correcto funcionamiento de lo implementado. Para comprobar el funcionamiento de la API REST, se llevaron a cabo peticiones mediante Postman. Un ejemplo de la comprobación del funcionamiento se encuentra en la figura 21.

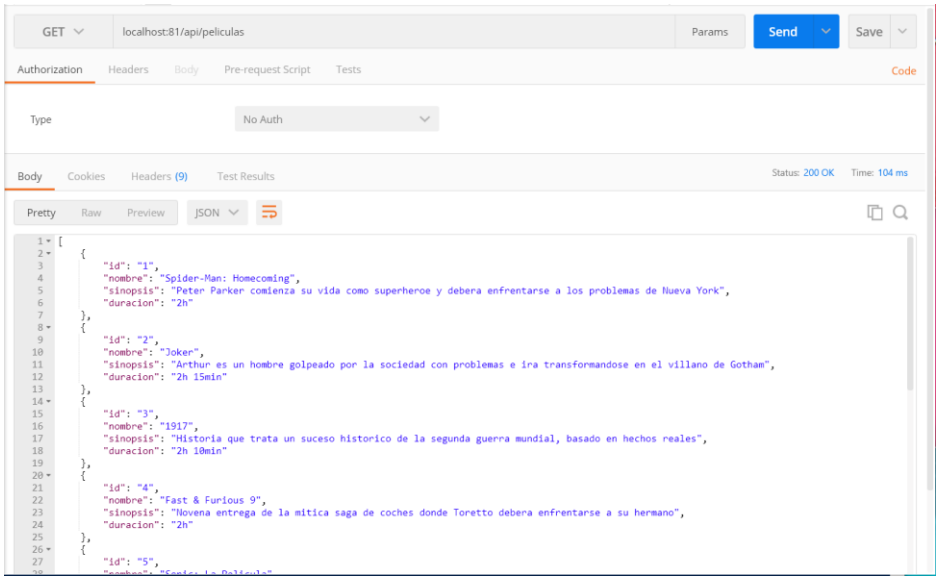

Figura 21: Comprobación funcionamiento API REST

Un detalle importante que comentar es la url utilizada para comprobar que funciona la API REST. Como se explicó en la elaboración del archivo Compose, se conectaron los puertos 81 del equipo local con el puerto 80 del contenedor que levanta la API. Por este motivo, debemos enviar peticiones al puerto 80 de nuestro equipo seguido con la ruta establecida para el método que se desea llevar a cabo. Este dato es importante para llevar a cabo la conexión entre el front-end y el back-end de la aplicación web de este proyecto.

3.4. DESARROLLO DEL FRONT-END

El último servicio que se implementó, el cual no es otro que el front-end que sirve como interfaz a los usuarios para que puedan llevar a cabo los distintos métodos implementados en la API REST, se llevó a cabo mediante Bootstrap y JQuery creando con el primero tanto la estructura de la web como su diseño, y con el segundo la conexión entre los dos servicios gracias a las funciones AJAX, gracias a las cuales se llevan a cabo peticiones a las API REST.

Si pensamos en un desarrollo sin contenedores, habría que implementar en una máquina virtual, un servidor ya sea XAMP o APACHE, para se pudiera desplegar todo el front-end, además de implementar las distintas páginas.

Sin embargo, gracias a este desarrollo basado en contenedores, bastó con implementar el front-end en nuestro equipo y tras esto, bastó con añadir el servicio al archivo Compose creado en la sección 3.3 y desplegar toda la aplicación contenedorizada con Docker Compose.

La implementación de este servicio se basó en llevar a cabo una web con 4 páginas distintas pensando en las distintas funcionalidades que se habían desarrollado en el servicio de la API REST, siendo una de estas la página principal donde se muestran las películas que el cine oferta, otra página donde el usuario puede ver sus entradas y eliminarlas, una tercera página que le permita al usuario obtener una entrada (realizando el método POST descrito en la implementación de la API REST) y una cuarta página que le permita modificar las entradas que posee.

Todas estas páginas, las librerías pertenecientes a bootstrap y un archivo javascript denominado main.js que sirve para la conexión entre front-end y back-end, se almacenaron en el directorio */front/bootstrap/* del directorio *proyecto* (donde estamos guardando todo el desarrollo de la aplicación web y donde se encuentra el archivo docker-compose.yml) lo que deja claro cual será la dirección del volumen que utilizará el contenedor que levante este servicio.

3.4.1. ESTRUCTURA DEL FRONT-END, HTML Y BOOTSTRAP

La primera tarea en la implementación del front-end consistió en estructurar todas las páginas que conforman este servicio. Como se ha comentado en la sección 3.4, este front-end dispone de 4 páginas en las que en cada una de ellas se llevan a cabo distintas peticiones de la API REST[8].

En las páginas como tal, se usaron las etiquetas básicas `<html>` `<head>` `<body>` y `<footer>` para estructurar cada una de las páginas, usando `<head>` donde se almacenan metadatos, `<body>` donde se encuentran los menús, formularios y demás elementos con los que interactúa el usuario y `<footer>` que indica el final de la página con, en este caso, información de contacto de la Universidad de Almería.

Un elemento que se encuentra en todas las páginas es el menú de navegación, cuya tarea es dirigir al usuario en el resto de las páginas de la aplicación. La cual se ha implementado como se muestra en el fragmento de código 18 y en el que se utilizó etiquetas para menús de navegación como `<nav>`, `<button>` para distintos botones y `<a>` para crear los enlaces al resto de páginas. Gracias a Bootstrap, se consigue que tanto este menú, como el resto de las páginas web tengan un responsive design, es decir, un diseño adaptado a todo tipo de dispositivos tanto para pantallas de sobremesa como smartphones y tablets.

```
16    <body>
17      <!-- menú superior-->
18      <div class="container-fluid text-center">
19        <nav class="navbar navbar-expand-lg navbar-light container"
20             style="background-color: ghostwhite">
21          <a class="navbar-brand" href="#"><img src="imagenes/escudo.gif"
22            width="50" height="50" class="d-inline-block align-center mr-2" alt="">
23            Cine UAL</a>
24          <button class="navbar-toggler" type="button" data-toggle="collapse"
25           data-target="#navbarNavAltMarkup" aria-controls="navbarNavAltMarkup"
26           aria-expanded="false" aria-label="Toggle navigation">
27            <span class="navbar-toggler-icon"></span>
28          </button>
29          <div class="collapse navbar-collapse" id="navbarNavAltMarkup">
30            <div class="navbar-nav ml-auto">
31              <a class="nav-item nav-link" href="index.html">Inicio</a>
32              <a class="nav-item nav-link" href="misentradas.html">Mis entradas</a>
33              <a class="nav-item nav-link" href="buyentradas.html">Conseguir Entradas</a>
34              <a class="nav-item nav-link" href="modentradas.html">Modificar Entradas</a>
35            </div>
36          </div>
37        </nav>
38      </div>
39    <!-- fin menu superior-->
```

Fragmento código 18: Menú de navegación html

Otro elemento que se repite en el resto de las páginas es el footer, que como se comentó, incluye información de la universidad. Para ello, usando la etiqueta `<footer>` y dentro de esta, los distintos datos que hacen referencia a la universidad. Además, otra parte importante es la carga del archivo JavaScript y las bibliotecas de Bootstrap para lo que se utilizó la etiqueta `<script>`. Todo esto se puede observar en el fragmento de código 19.

```
<!--footer-->
<footer class="container-fluid text-white py-3 text-center container" style="background-color:midnightblue">
<p>
  Universidad de Almeria    Carretera Sacramento s/n 04120 La Cañada de San Urbano    Almeria
</p>
</footer>
<!--fin footer-->
<!-- Optional JavaScript -->
<!-- jQuery first, then Popper.js, then Bootstrap JS -->
<script src="https://code.jquery.com/jquery-3.4.1.slim.min.js"
integrity="sha384-J6qa4849blE2+poT4WnyKhv5vZF5SrPo0iEjwBvKU7imGFAV0wwj1yYfoRSJoZ+n" crossorigin="anonymous"></script>
<script src="https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist/umd/popper.min.js"
integrity="sha384-Q6E9RHvbIyZFJoft+2mJbHaEWldlvI9IOYy5n3zV9zzTtmI3UksdQRVvoxMfooAo" crossorigin="anonymous"></script>
<script src="js/bootstrap.min.js"></script>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.4.1/jquery.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/mustache.js/3.1.0/mustache.min.js"></script>
<script src="main.js"></script>
</body>
</html>
```

Fragmento código 19: Footer del front-end

Una vez estructurada las páginas del front-end, faltó pensar en cómo se quería implementar la interacción del usuario con la aplicación. Por ello, en la página principal del front-end se decidió en implementar una lista con todas las películas que devuelve la API REST, por lo que se decidió crear una lista mediante la etiqueta `<ul>` como se ve en el fragmento de código 20.

```
<!--main-->
<div class="container mt-5 mb-5">
  <div class="h2">DISFRUTA DE TUS PELÍCULAS EN LA UNIVERSIDAD</div>
</div>
<div class="container mt-3 mb-3">
  <p>Ya puedes disfrutar de las mejores películas en tu universidad y aquí tienes una selección de las peliculas que puedes ver.</p>
  <div class="container-fluid mt-3">
    <ul class="list-group list-group-flush" id="resultado">
    </ul>
  </div>
</div>
<!--fin main-->
```

Fragmento código 20: Lista películas html

En la página que muestra las entradas que posee un usuario, se decidió en implementar otra lista también, por lo que se volvió a utilizar la etiqueta `<ul>` en esta página y que tiene la misma estructura que la mostrada en el fragmento de código 20.

Para las páginas donde un usuario obtiene una entrada y la modifica, lo más lógico fue la implementación de formularios que sirvieran para poder realizar las peticiones POST y PUT respectivamente. Para estos formularios, dentro de sus etiquetas correspondientes se utilizó la etiqueta `<form>` la cual es la necesaria para alcanzar el objetivo de estas páginas como se muestra en el fragmento de código 21.

```
<!--main-->
<div class="container mt-5 mb-5">
  <div class="h2 mb-3">Conseguir entradas</div>
  <p>Aquí podras conseguir tus entradas, para ello, escribe la función y el asiento que quieras. Una vez escritas, pulsa Conseguir entrada.</p>

  <!--formulario meter entradas-->
  <div class="container my-4">
    <form>
        <div class="form-group">
          <label >Id Funcion</label>
          <input type="text" class="form-control" id="id1"  placeholder="Escriba aquí el id de la funcion">
        </div>
        <div class="form-group">
          <label >Id Asiento</label>
          <input type="text" class="form-control" id="id2" placeholder="Escriba aquí el id del asiento">
        </div>
        <button type="submit" class="btn btn-primary" id="meterentrada">Conseguir Entradas</button>
    </form>
  </div>

<!--fin main-->
```

Fragmento código 21: Formulario conseguir entradas html

Tanto en las listas implementadas, como los inputs de los formularios, es vital el uso de los *id* que nos servirán a continuación para tanto mostrar datos en las listas, como introducir los datos con los que se llevarán a cabo las peticiones POST y PUT.

3.4.2. JQUERY Y CONEXIÓN ENTRE FRONT-END Y BACK-END

JQuery permite, a través de la función `ajax()`, llevar a cabo peticiones a rutas que poseen las API REST. En esta implementación reside la conexión entre front-end y back-end ya que aquí, el primer servicio se encargará de contactar con la API REST a través de las peticiones utilizando las rutas que se definieron en las secciones correspondientes a la implementación de los distintos métodos.

Estas funciones `ajax()` funcionan estableciendo una serie de parámetros los cuales permiten llevar a cabo las distintas peticiones. Estos parámetros son:

- type: Indica el tipo de petición que se va a realizar (GET, POST, PUT o DELETE).
- url: Ruta a la que se llevará a cabo la petición.
- dataType: Formato en el que se enviarán y devolverán los datos tanto los que se envían a la petición como los que recibe.
- success: Función que se lleva a cabo si la petición es un éxito
- data: Datos que se envían en métodos de tipo POST y PUT.

Para estas funciones, existen muchos más parámetros con los que hacer muchas más tareas, pero para nuestra aplicación web, se decidió utilizar estos 4 ya que no había ningún motivo para utilizar más parámetros.

Las conexiones entre front-end y back-end se realizan en los elementos que se han comentado en la sección 3.4.1 los cuales son las listas tanto de películas como de entradas, y los formularios para conseguir una entrada y modificar una de las que ya se poseen.

Para la conexión entre estos dos servicios, en principio con poner en la url la dirección tipo "http://nombrecontenedorAPI/rutaAPI" debería funcionar. Sin embargo, al ejecutarse JavaScript desde el buscador donde se utiliza, como las peticiones se van a llevar a cabo desde el buscador de un usuario concreto (que en mi caso para llevar a cabo distintas pruebas y demás es el mío el de mi equipo local) las rutas deben llamar desde el exterior al puerto donde se encuentra el contenedor que contiene la API REST (que en este proyecto y en este instante es el puerto 81 de el equipo local) con la forma http://localhost:81/rutaAPI .

Para la lista de películas, el objetivo de esta conexión se basa en la devolución por parte de la API REST de un listado de películas, y que cada una de estas películas se introduzca en la lista. Para ello, se establece en `type` la petición tipo GET, en `url` la dirección "http://localhost:81/api/peliculas" debido a los motivos que se comentaron hace un instante, en `dataType` "json" ya que es el formato en el que trabaja la API REST y en `success()` se implementó una función que introduce un elemento html mediante la etiqueta `<li>` que hace referencia a cada película de la lista. Esta etiqueta es la que almacena de cada película su nombre, duración y sinopsis. Toda esta implementación se aprecia en el fragmento de código 22.

```
    var $peliculas = $('#resultado');
    //obtener todas las peliculas
    $.ajax({
        type: 'GET',
        url: 'http://localhost:81/api/peliculas',
        dataType: "json",
        success: function(peliculas){
            $.each(peliculas, function(i, pelicula){
                $peliculas.append(`<li class="list-group-item">${pelicula.nombre}<br>
                                   <b>Sinopsis:</b> ${pelicula.sinopsis}<br>
                                   <b>Duracion:</b> ${pelicula.duracion}</li>`)
            });
        }
    });

});
```

Fragmento código 22: AJAX obtención de películas

Esta es la estructura utilizada también en la lista de entradas, donde se utiliza la ruta "http://localhost:81/api/entradas" estableciendo en `type` el método GET, en `dataType` el formato "json" y en `success()`, la función introduce elementos `<li>` siendo cada uno de estos, las entradas que se han obtenido. Aquí entra también un botón con la palabra "Eliminar" que permite la eliminación de la entrada en la que se encuentre este botón y así llevar a cabo la petición tipo DELETE. Esta implementación de la muestra de entradas es la mostrada en el fragmento de código 23.

```
    //obtener todas las entradas
    $.ajax({
        type: 'GET',
        url: 'http://localhost:81/api/entradas',
        dataType: "json",
        success: function(entradas){
            $.each(entradas, function(i, item){
                añadirMisEntrada(item);
            });
        }
    });
```

Fragmento código 23: AJAX mostrar entradas

Para hacer más eficiente este código, y teniendo en cuenta que hay que añadir entradas también a la base de datos mediante el método POST y modificar entradas mediante el método PUT y esos cambios se deben ver reflejados en la lista de entradas, se decidió crear una función

denominada "`añadirMisEntrada(entrada)`" que introduce una entrada la cual se pasa por parámetro en la lista gracias a la etiqueta `<li>` y su botón correspondiente para poder eliminarla.

Para la eliminación de entradas, en la función `AJAX()` se indicó que se trataba de una petición tipo DELETE, y en la url se utilizó como parámetro para la ruta (si recordamos la ruta de la petición DELETE tenía como parámetro la id de la entrada a eliminar) el id de la entrada que se encuentra en el elemento de la lista donde se pulsa el botón Eliminar. El éxito de la petición se refleja con la eliminación de la entrada con la función `remove()`.

```
71      //eliminar entrada
72      $entradas.delegate("button", 'click', function(){
73
74          var url = `http://localhost:81/api/entradas/${$(this).attr('id')}`;
75          console.log(url);
76          var $li = $(this).closest('li');
77          $.ajax({
78              type: 'DELETE',
79              url: url,
80              success: function(){
81                  $li.fadeOut(300, function() {
82                      $(this).remove();
83                  });
84              }
85          });
86      });
```

Fragmento código 24: AJAX eliminar entrada

Para los formularios, en el que tiene como objetivo permitir a un usuario conseguir una entrada, se guardaron los valores introducidos en los distintos `<input>` en distintas variables, que sirven para crear la entrada en formato "json" y que se envía en la petición, tanto para crear la entrada e introducirla en la base de datos, como para seleccionar una entrada escribiendo su id y estableciendo sus nuevos datos en el formulario de modificación de entradas.

Para el primer formulario el código correspondiente se encuentra en el fragmento de código 21, en el que como se ha descrito, con los valores de los `<input>` se crea una entrada en formato json mediante la función `JSON.stringify(entrada)` siendo el parámetro entrada los valores de los `<input>` y como éxito de llevar a cabo la petición, que esta entrada se introduzca también en la lista de entradas mediante la función `añadirMisEntrada(entrada)`.

```
    //conseguir entrada
    $('#meterentrada').on('click',function() {

        var ent = {
            idFuncion: $1.val(),
            idAsiento: $2.val(),
        };
        var jsonenvio = JSON.stringify(ent);
        console.log(jsonenvio);
        $.ajax({
            type: 'POST',
            url: 'http://localhost:81/api/entradas',
            data: jsonenvio,
            success: function(item){
                console.log(item);
                añadirMisEntrada(item);
            }
        });
    });
```

Fragmento código 25: AJAX introducir entrada

Para el segundo formulario, tiene una estructura idéntica a la descrita en el método anterior, con el añadido del valor *id* de la entrada que se desea modificar y que se añade en la ruta como pide la API REST.

```
    //modificar entrada
    $('#modentrada').on('click',function() {

        var idmodificar = $mod1.val();

        var modent = {
            idFuncion: $mod2.val(),
            idAsiento: $mod3.val(),
        };

        var url= `http://localhost:81/api/entradas/` + idmodificar;

        var modentenviar = JSON.stringify(modent);

        $.ajax({
            type: 'PUT',
            url: url,
            data: modentenviar,
            success: function(modent){
                añadirMisEntrada(modent);
            }
        });
    });
```

Fragmento código 26: AJAX modificar entrada

3.4.3. SEGUNDA VERSIÓN FRONT-END

Al igual que con la API REST, el Front-End tiene una segunda versión que se conecta a la segunda versión de la API, la cual ofrece funcionalidades nuevas descritas e implementadas en la sección 3.3.3. por lo que las nuevas páginas de este front-end deben aprovechar esas nuevas características.

Una de ellas es la página principal, donde en ella se pasa de ver solamente las películas que tiene el cine disponible, a un nuevo modo de ver las películas el cual consiste en la selección de una fecha en la que el cliente desea ver la película y una vez seleccionada, aparece una lista de películas con sus funciones que estarán disponibles en esa fecha. Además, estas películas

aparecen con información adicional como la hora a la que se reproducirá la película, la sala donde se reproduce, etc.

El cambio introducido a nivel estructural fue la adición de una lista desplegable que muestre todas las fechas con las que trabaja la aplicación. Para ello, se creó la lista desplegable de fechas mediante la etiqueta `<select>` y se añadió un botón que, al pulsarlo, busque las películas en esa fecha y aparezca la lista con las peliculas encontradas.

Para llevar a cabo esta funcionalidad, se creó un nuevo archivo JavaScript denominado main2.js donde se crearon dos funciones. La primera de estas dos devuelve todas las fechas disponibles que contiene la base de datos, y la otra añade las películas encontradas por esa fecha a la lista por lo que se usaron las rutas de los métodos explicados en la sección 3.3.3. Con todo ello, la implementación de estas funciones se puede observar en los fragmentos de código 27 y 28 respectivamente.

```
6     $fechas = $('#seleccionarfecha');
7     $.ajax({
8         type:'GET',
9         url:'http://localhost:81/api/peliculas/fechas',
10        dataType: "json",
11        success: function(resultado){
12            $.each(resultado, function(i,fecha){
13                $fechas.append('<option>'+fecha.fecha+'</option>');
14            });
15        }
16    });
```

Fragmento código 27: AJAX mostrar fechas

Del fragmento de código 24, comentar que, para mostrar más información de las películas, basta con añadir esos datos a la función `añadirPeliculas(película)` la cual añade las películas a la lista, y para obtener el parámetro de la fecha, basta con guardar el valor de la fecha de la lista desplegable seleccionada para poder completar la ruta de la segunda versión de la API REST que llamaba al método que devolvía películas según la fecha.

Además, antes de llamar a la función que añade las películas, se usa la función `empty()` para borrar la lista cada vez que se realiza una búsqueda y que, de esta forma, tras buscar tres veces, la lista no contenga las películas de las 2 fechas anteriores mostradas.

```
    //mostrar funciones cuando selecciono fecha
    var $peliculas = $('#resultado');

    function añadirPeliculas(pelicula){
        $peliculas.append(`<li class="list-group-item"><b>Nombre:</b> ${pelicula.nombre} <br>
        <b>Sinopsis:</b> ${pelicula.sinopsis} <br>
        <b>Duracion:</b> ${pelicula.duracion} <br>
        <b>Id Funcion:</b> ${pelicula.idFuncion} <br>
        <b>Id Sala:</b> ${pelicula.idSala} <br>
        <b>Fecha:</b> ${pelicula.fecha} <br>
        <b>Hora:</b> ${pelicula.hora}`);
    }

    $('#buscarfunciones').on('click', function(){

        var fechaseleccionada = $fechas.val();
        console.log(fechaseleccionada);
        url = `http://localhost:81/api/peliculas/search/`+ fechaseleccionada;
        console.log(url);

        $.ajax({
            type:'GET',
            url: url,
            dataType: "json",
            success: function(solucion){
                $peliculas.empty();
                $.each(solucion, function(i,pelicula){
                    añadirPeliculas(pelicula);
                });
            }
        });
    });

});
```

Fragmento código 28: AJAX mostrar películas por fecha

La siguiente página que sufre cambios en esta segunda versión, es la página que muestra la lista de entradas, la cual, se comentó en la segunda versión de la API, muestra más información sobre las entradas que el usuario tiene.

En esta nueva funcionalidad, la estructura de la página no cambia, ya que el único cambio reside en que ahora la conexión ha de realizarse al método de la segunda versión de la API REST que muestra entradas con más información.

La función `ajax()`, por tanto, es igual que al implementado en la versión anterior, con el único añadido de más información reflejada en las etiquetas `<li>` de la lista que muestra las entradas al usuario que utiliza la aplicación web. Esta función se encuentra en el fragmento de código 25.

```
//entradas
$(function(){

    //variable para lista mis entradas
    var $entradas = $('#resultado1');

    //Como añado entradas a la lista de Mis Entradas
    function añadirMisEntrada(entrada){
        $entradas.append(`<li class="list-group-item ">
        <b>Id: </b> ${entrada.id} <br>
        <b>Pelicula:</b> ${entrada.nombre} <br>
        <b>Fecha:</b> ${entrada.fecha} <br>
        <b>Hora:</b> ${entrada.hora} <br>
        <b>Fila:</b> ${entrada.fila} <br>
        <b>Asiento:</b> ${entrada.asiento} <br>
        <button type="button" class="btn btn-primary" id="${entrada.id}">Eliminar</button>`);
    }
    //obtener todas las entradas
    $.ajax({
        type: 'GET',
        url: 'http://localhost:81/api/entradas',
        dataType: "json",
        success: function(entradas){
            $.each(entradas, function(i, item){
                añadirMisEntrada(item);
            });
        }
    });
```

Fragmento código 29: AJAX mostrar entradas segunda versión

Las últimas modificaciones importantes se encuentran en los formularios donde un usuario puede conseguir una entrada y modificar una de las que ya posee. La forma de llevar a cabo estas acciones en la primera versión del front-end pasaba por rellenar estos datos mediante etiquetas `<input>`, lo cual pensando en la facilidad del usuario a la hora de utilizar la aplicación web, resulta muy complejo conocer el id de cierta entrada, así como los id de las funciones a las que se quiere ir o la función del asiento.

Por ello, en el formulario basado en la obtención de entradas y que lleva a cabo el método POST, se cambiaron los `<input>` por listas desplegables mediante la etiqueta `<select>` como las utilizadas en la selección de fechas para buscar las películas que se proyectan en esos días, además del uso de botones que marquen el envío de información y llevar así un uso mucho más intuitivo.

Esto provoca cambios en las funciones del archivo JavaScript ya que ahora hay una lógica más elaborada que en la versión anterior, ya que ahora debemos mostrar en una lista desplegable las distintas funciones, que en otra lista aparezcan los asientos de esa función (con estos dos datos se creaba la entrada) y una vez seleccionados ambos valores de las listas, enviarlos a la base de datos.

Para la lista desplegable de las funciones, se llevó a cabo una función `ajax()` que enviara una petición al método de la segunda versión de la API REST que devuelve todas las funciones de las películas, y tras ello, la adición de cada una de estas funciones a la lista desplegable.

Para la lista de los asientos, se implementó otra función `ajax()` siguieron los mismos pasos cambiando en este caso la ruta por la implementada en la segunda versión de la API REST la cual hace referencia al método que devuelve los asientos según la función. Esta función que se pasa por parámetro es la que se selecciona en la lista de las funciones que se acaba de explicar. Estas dos funciones se muestran en el fragmento de código 30.

```
//COMPRAR ENTRADAS
//Valores en buyEntradas
var $funciones = $('#id1');
var $asientos = $('#seleccionarasiento');
//seleccionar funcion
$.ajax({
    type:'GET',
    url: 'http://localhost:81/api/peliculas',
    dataType: "json",
    success: function(peliculas){
        $.each(peliculas, function(i,funcion){
            $funciones.append(`<option value="${funcion.idFuncion}">
                                    Pelicula:${funcion.nombre}
                                    Fecha:${funcion.fecha}
                                    Hora:${funcion.hora}</option>`)
        });
    }
});
//seleccionar asiento
$('#seleccion').on('click', function(){
    url = 'http://localhost:81/api/asientos/' + $funciones.val();
    console.log(url);

    $.ajax({
        type: 'GET',
        url: url,
        dataType: "json",
        success: function(data){
            $asientos.empty();
            $.each(data, function(i,asiento){
                $asientos.append(`<option value="${asiento.id}">Fila:${asiento.fila}
                                                    Asiento:${asiento.asiento}</option>`);
            });
        }
    });
    document.getElementById('mostrarasientos').style.display="block";
});
```

Fragmento código 30: AJAX listas desplegables de funciones y asientos

Como se ve en la última línea del fragmento de código 26 se añadió una función con la que desaparece la segunda lista desplegable de los asientos, con la función `document.getElementById('elemento a ocultar').style.display="block"`. Esta función la añadimos para que la lista aparezca en la página cuando el usuario entre para que no intente seleccionar un asiento hasta que no haya seleccionado una función y que solo aparezca cuando pulse el botón de Seleccionar un asiento.

Una vez seleccionados los datos que conforman la entrada mediante las listas desplegables, falta enviar estos datos en la petición POST para crear la entrada tanto con la función seleccionada como con el asiento elegido. Para ello, basta con utilizar la misma función `ajax()` que se utilizó en la versión anterior explicada en la sección 3.4.2 ya que la petición sigue siendo igual en ambas versiones, y las diferencias radican en la forma en las que el usuario "introduce" los datos necesarios para crear la entrada que desea obtener.

Todo lo comentado en los cambios realizados en esta página se aplican a la modificación de una entrada que posee un usuario ya que para modificar una entrada se debe escribir el id de la entrada que quería modificar, así como el id de la nueva función y del nuevo asiento. Pero de la misma forma que se ha explicado en la página anterior, se llevó a cabo la misma forma de selección de datos que conforman la nueva entrada mediante listas desplegables con más información y que son más intuitivos para el usuario final.

Para ello, se crearon las listas en el archivo .html como se ha descrito antes mediante la etiqueta `<select>`. Literalmente se utilizan copias de las funciones `ajax()` de la funcionalidad de obtener las listas desplegables tanto de las funciones como de los asientos, ya que el funcionamiento es el mismo a diferencia de que hay que añadir una lista desplegable más en este caso, la de las entradas que posee un usuario y su método `ajax()` que devuelve las entradas, pero esta función es igual que la comentada en la figura de código 25.

Todos estos cambios son los que conforman la segunda versión del front-end con la que hemos llevado a cabo todas las novedades que hemos comentado en puntos anteriores. Con esta versión, el usuario puede navegar por la aplicación web y llevar a cabo las distintas funciones de forma más intuitiva que en la versión anterior.

3.5. DESPLIEGUE DE TODA LA APLICACIÓN Y DOCKER COMPOSE

El desarrollo mediante contenedores, permite que a medida que se implementan ciertos elementos el front-end, como pueden ser las distintas etiquetas html y el diseño de la página, se pueden observar los resultados de manera rápida, desplegando un pequeño contenedor con una imagen ubuntu reducida (por ejemplo la conocida como tutum/apache-php que es una versión muy reducida de ubuntu con lo necesario para desplegar front-end) y asignarle como volumen, el directorio en el que se almacena toda la información, en vez de arrancar una máquina virtual con lo que ello conlleva, y ejecutar todo el software para que podamos visualizar en el buscador las distintas páginas implementadas.

En el caso del desarrollo de este proyecto, para todo lo relacionado con el front-end menos las conexiones explicadas en las secciones 3.4.2 y 3.4.3 se comprobaron de manera rápida los desarrollos desplegando únicamente un contenedor enlazando el puerto 80 del equipo local con el puerto 80 del contenedor, asignando como volumen el directorio /var/www/html del contenedor con el directorio /front/Bootstrap de nuestro equipo donde se almacenan todas las implementaciones explicadas en toda la sección 3.4.

Sin embargo, como ocurre en el caso del despliegue de la API REST, para comprobar el correcto funcionamiento de las funciones AJAX implementadas y comprobar así la conexión entre front-end y back-end, hay que desplegar toda la aplicación en su conjunto, añadiendo al archivo Compose de la sección 3.3 en la que se había definido ya tanto la base de datos como la API REST.

Por ello, se trasladó los despliegues individuales realizados al front-end a este archivo para automatizar mediante la ejecución de este archivo, el despliegue de toda la aplicación web, que con ello se pudo comprobar, además, el correcto funcionamiento de la conexión entre el front-end y el back-end.

Para automatizar el despliegue del servicio front-end, basta con llevar las mismas configuraciones que se llevaron a cabo en la sección 3.3, donde se inicializa el servicio con nombre front, además de configurar el contenedor en el que se despliega asignándole un nombre, así como el volumen del que obtiene todas las implementaciones realizadas en la ruta /var/www/html del contenedor y el puerto por el que se unen. Además, la imagen con la que se levanta el contenedor, como solo necesitamos un servidor apache se optó por la imagen tutum/apache-php que a pesar de estar descontinuada desde hace bastante tiempo, para el desarrollo de este proyecto se utilizó para mostrar las ventajas de uso de recursos y espacio que

aportan los contenedores Docker, y el archivo Compose quedó como se aprecia en el fragmento de código 31.

```yaml
version: '2'
services:
  bdcine:
    container_name: bdcine
    restart: always
    image: mysql:5.7
    environment:
      MYSQL_ROOT_PASSWORD: 'root'
    ports:
      - "3306:3306"
    volumes:
      - ./datos:/var/lib/mysql
      - ./init.sql:/docker-entrypoint-initdb.d/init.sql
  api-cine:
    container_name: api-cine
    restart: always
    image: ualmtorres/phalcon-apache-ubuntu
    ports:
      - "81:80"
    volumes:
      - ./api:/var/www/html
  front:
    container_name: front
    restart: always
    image: tutum/apache-php
    ports:
      - "80:80"
    volumes:
      - ./front/bootstrap:/var/www/html
```

Fragmento código 31: Archivo Compose con toda la aplicación

Este archivo constituye lo que se conoce en el desarrollo basado en contenedores como un "entorno de contenedores" y hace referencia al despliegue de los distintos contenedores que llevan a cabo los distintos servicios de los que consta una aplicación contenedorizada.

Se vuelve a recalcar en las diferencias que este desarrollo presenta frente a las máquinas virtuales, ya que llevar a cabo simplemente este despliegue, supone iniciar tres máquinas virtuales distintas, en las que en cada una de ellas se encuentra cada uno de los servicios descritos. Esto ocuparía muchos gigas de espacio por la iniciación de cada una de estas máquinas virtuales, así como la dificultad en la conexión entre estas, mientras que con los contenedores y Docker Compose, con simplemente ejecutar un comando, se despliega en cuestión de segundos estos tres contenedores, así como la conexión entre estas que se encuentra implementada en cada uno de los servicios como se ha ido desarrollando a lo largo de las secciones.

El despliegue del entorno de contenedores configurado, en el caso de este proyecto, se lleva a cabo mediante el comando

```
docker-compose up -d
```

como ya se llevó a cabo en la sección 3.3.4. Este despliegue se aprecia en la figura 22 donde además, antes de desplegar los contenedores se comprueba que todos los directorios y archivos se encuentran en el directorio “proyecto”.

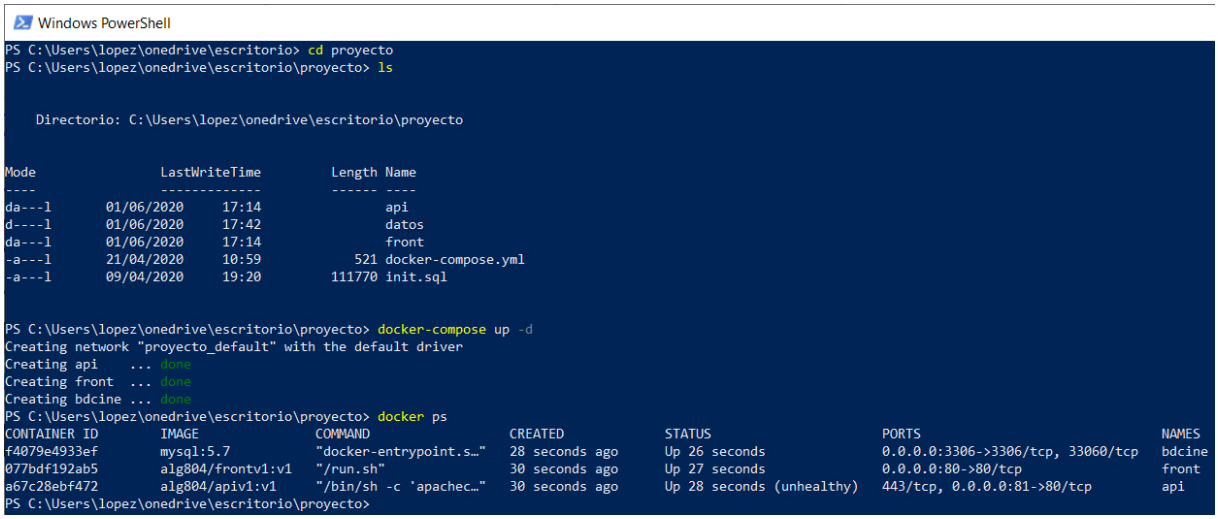

Figura 22: Creación entorno

Para desplegar el entorno de contenedores de la segunda versión de la aplicación, basta con crear otro archivo Compose en el directorio donde se guardan las implementaciones realizadas y se sigue el mismo patrón de configuraciones que las llevadas a cabo en esta sección[9].

3.6. DISTRIBUCIÓN Y EJECUCIÓN DE APLICACIONES Y DOCKERFILE

Como ya se comentó en las primeras secciones de esta memoria, el desarrollo de este proyecto supone la implementación de una aplicación basada en contenedores para seguir una arquitectura basada en microservicios. Además, se comentó que se llevaría a cabo una segunda versión de la aplicación para en el Complemento del Trabajo de Fin de Grado, llevar a cabo pruebas de despliegue de actualizaciones y redireccionamiento de tráfico a estas distintas versiones.

El desarrollo de la aplicación comprende la implementación de varios servicios que trabajan entre sí para que la aplicación funcione correctamente. En un caso de desarrollo real de producción software, una vez hemos llegado a este punto en el que desplegamos los entornos de contenedores y comprobamos el correcto funcionamiento de la aplicación, se puede confirmar su fin de desarrollo.

Sin embargo, poner en marcha esta aplicación, en esta fase en la que se encuentra, supone que cualquier otra persona que desee iniciar nuestra aplicación web desde su equipo, necesito primero, descargarse desde cualquier repositorio como GitHub todo el código referente a las implementaciones o que el propio desarrollador le envíe el código. Después, esa persona debe configurar un archivo Compose, teniendo en cuenta los distintos volúmenes, teniendo que conocer el tipo de imagen Docker que usa (no es lo mismo desplegar una web desde un sistema Linux que desde un sistema Windows) y tras el resto de las configuraciones, desplegar el contenedor.

Una gran virtud de Dockerfile es que la posibilidad de crear nuevas imágenes a partir de imágenes ya creadas. Gracias a esto, cuando se finaliza el desarrollo de un software en concreto, y se comprueba su correcto funcionamiento, se puede crear una imagen Docker que no solo contenga la instalación de las dependencias necesarias para que las distintas implementaciones funcionen, sino que se puede almacenar todo el desarrollo en una imagen que contenga así las dependencias que necesita el código implementado y el propio código, para así, con descargar solo estas imágenes Docker, poder desplegar no solo contenedores sueltos para comprobar ciertos servicios, sino crear de esta forma, entornos de contenedores de forma más rápida y que sirve además para poner estas aplicaciones en producción.

Además, se pueden guardar así distintas versiones de un desarrollo en distintas imágenes docker, lo que permite llevar a cabo un ciclo de desarrollo basado en versiones en las que cada vez que se finaliza la construcción de una versión de cierto software y se comprueba su funcionamiento, se puede comprimir en una imagen docker y subir a un repositorio de imágenes (trabajando con docker, lo más óptimo es subirlas a Docker Hub) para facilitar así su intercambio con el resto de los desarrolladores. Esto último, permite una de las ventajas que se comentaba en la sección 2.1 y que hacía referencia a la portabilidad de los contenedores Docker, ya que cualquier persona que tenga Docker instalado en su equipo, descargándose las imágenes Docker creadas, pueden ejecutarlas en cualquier entorno de desarrollo.

Para crear imágenes docker usando Dockerfile, basta con crear un archivo con el mismo nombre en el directorio donde se encuentran todos los ficheros y directorios con los que se desea crear la imagen, y en este archivo Dockerfile, implementar una serie de instrucciones para que Docker cree la imagen. Algunas de estas instrucciones son:

- FROM: Sirve para utilizar como base una imagen ya oficial que se encuentra en Docker Hub, el repositorio de Docker donde se encuentran todas las imágenes disponibles. Además, inicializa el contenedor con el sistema de archivos que se indique.
- ADD: Añade todos los archivos y directorios que se encuentran en una ruta local en una ruta concreta del contenedor.
- EXPOSE: Esta instrucción indica los puertos por los que los contenedores realizan conexiones, por lo que se debe utilizar los puertos tradicionalmente utilizados para la función que van a desarrollar (por ejemplo, un contenedor que va a funcionar como base de datos MySQL debe exponer el puerto 3306).
- ENV: Establece variables que son utilizadas en la creación de una imagen.
- VOLUME: Crea un punto de montaje que se puede conectar al sistema de archivos del host.

Con estas instrucciones y muchas más, se crean imágenes docker como las que se utilizaron en el desarrollo de los distintos servicios del proyecto (como la imagen mysql:5.7 que se utilizó para la base de datos, así como la imagen tutum/apache-php la cual se trata de una imagen comprimida de Ubuntu como Alpine que ocupa 2 mb).

Un ejemplo del uso de Dockerfile se encuentra en los fragmentos de código 32 y 33, donde se usa Dockerfile para crear dos imágenes que corresponden a la API REST y al front-end. En ellos, se utilizan las instrucciones `FROM`, `ADD` y `EXPOSE` para sobre una imagen ya creada (que supone el sistema de archivos que nuestro código va a utilizar) añadir toda la información de

los distintos directorios en los que se almacenaron las implementaciones explicadas a lo largo de toda esta sección 3.

```
1  FROM ualmtorres/phalcon-apache-ubuntu
2
3  ADD api /var/www/html
4
5  EXPOSE 80
```

Fragmento código 32: Dockerfile API REST

En el caso del fragmento de código 29, se utilizó la imagen ualmtorres/phalcon-apache-ubuntu ya que se trata de una imagen que contiene estas tres últimas tecnologías y que son necesarias para el despliegue de la API REST. Con la instrucción `ADD`, se copió el contenido del directorio *api* en la ruta /var/www/html del contenedor que levanta y, por último, con el comando `EXPOSE`, se expone el puerto 80 ya que se necesita para acceder a él desde la red y poder llevar a cabo las peticiones http.

```
1  FROM tutum/apache-php
2
3  ADD front/bootstrap /var/www/html
4
5  EXPOSE 80
```

Fragmento código 33: Dockerfile front-end

En el caso del fragmento 30 se llevan a cabo los mismos pasos que en el fragmento 29 a diferencia de que el sistema de archivos sobre el que se construye la imagen es tutum/apache-php (no ser recomienda su uso ya que, como se comentó anteriormente esta imagen hace tiempo que está descontinuada) al ser esta una imagen reducida de ubuntu que contiene apache, y la instrucción `ADD` que en este caso debe añadir la información de la ruta front/Bootstrap en el directorio /var/www/html.

Una vez creados los archivos Dockerfile, para crear las imágenes docker, se usa el comando

`docker build -t "nombre que recibirá la imagen"`

Desde el directorio donde se encuentra el archivo Dockerfile, que debe ser el mismo directorio en el que se encuentre la ruta de los directorios de la instrucción `ADD`. Un ejemplo de una imagen docker creada mediante el comando es la que se aprecia en la figura 13 donde se observa cómo Docker ejecuta cada instrucción para crear la imagen Docker en este caso de la implementación de la API REST

```
Build an image from a Dockerfile
PS C:\Users\lopez\onedrive\escritorio\aplicacion\appv1> docker build -t alg804/api:v0 .
Sending build context to Docker daemon  211.9MB
Step 1/3 : FROM ualmtorres/phalcon-apache-ubuntu
 ---> 317e01446c74
Step 2/3 : ADD api /var/www/html
 ---> 2a3e689cf7db
Step 3/3 : EXPOSE 80
 ---> Running in a07246c8bf47
Removing intermediate container a07246c8bf47
 ---> 5f3c3d2a82d6
Successfully built 5f3c3d2a82d6
Successfully tagged alg804/api:v0
```

Figura 23: Construcción imagen Docker mediante Dockerfile

Una vez finalizada la creación de la imagen, esta se almacena de manera local en nuestro equipo. Para subir la imagen a Docker Hub, se necesita el comando

`docker push "nombre imagen"`

como aparece en la figura 24 donde se sube la imagen creada de la API REST al repositorio de Docker.

```
PS C:\Users\lopez\onedrive\escritorio\aplicacion\appv1> docker push alg804/api:v0
The push refers to repository [docker.io/alg804/api]
7f4facaf2213: Pushed
f3ca5198b82b: Pushed
e870c5625e09: Pushed
3dc1e0e22fc7: Pushed
be95a4129194: Pushed
36a61d7e978c: Pushed
38658f9b2a44: Pushed
a0895332dff3: Mounted from ualmtorres/phalcon-apache-ubuntu
27134e094b42: Mounted from ualmtorres/phalcon-apache-ubuntu
2c5aab3e0ee8: Mounted from ualmtorres/phalcon-apache-ubuntu
b791c6684410: Mounted from ualmtorres/phalcon-apache-ubuntu
9acfe225486b: Pushed
```

Figura 24: Subida imagen docker a Docker Hub

También se llevó a cabo en este proyecto la construcción de la imagen que contiene toda la implementación del front-end y se subió al repositorio Docker Hub siguiendo los mismos pasos que los descritos hasta ahora. Los nombres de las imágenes tanto de la API REST como del front-end son alg804/api:vo y alg804/front:vo respectivamente.

Los dos puntos que se colocan al final del nombre de la imagen son los TAG. Los TAG indican la versión de la imagen y es lo que permite tener varias versiones de una misma imagen en un repositorio (en el caso de MySQL, en el desarrollo de este proyecto se optó por la imagen de mysql versión 5.7 pero en el mismo repositorio se observa como tienen imágenes de este sistema gestor de bases de datos hasta las versiones más actuales).

Se decidió entonces, disponer en el repositorio propio de Docker Hub, las dos versiones tanto de la API REST, como del front-end, llamando a estas imágenes alg804/api:v1 y alg804/front:v1 respectivamente. Se siguieron los mismos pasos descritos hasta ahora para la creación de estas imágenes y su respectiva subida al repositorio Docker Hub.

Si se desea comprobar que la creación de las imágenes ha sido exitosa, se puede crear un entorno de contenedores en el que se indique solamente a cada servicio la imagen a utilizar para la creación de los contenedores sean las que se han subido al repositorio y los puertos, para que se puedan comunicar entre sí los servicios. En el caso de este proyecto, para comprobar la creación de las imágenes de la primera versión de la aplicación, se creó el entorno de contenedores mediante un archivo Compose en el que el valor `image` tanto en el servicio de la API REST como del front-end eran alg804/api:v0 y alg804/front:v0 respectivamente, como se aprecia en el fragmento de código 34[9].

```yaml
version: '2'
services:
  bd:
    container_name: bdcine
    restart: always
    image: mysql:5.7
    environment:
      MYSQL_ROOT_PASSWORD: 'root'
    ports:
      - "3306:3306"
    volumes:
      - ./datos:/var/lib/mysql
      - ./init.sql:/docker-entrypoint-initdb.d/init.sql
  api:
    container_name: api
    restart: always
    image: alg804/api:v0
    ports:
      - "81:80"
  front:
    container_name: front
    restart: always
    image: alg804/front:v0
    ports:
      - "80:80"
```

Fragmento código 34: Archivo Compose con imágenes docker creadas

3.7. ORQUESTACIÓN DE CONTENEDORES Y KUBERNETES

Llega el momento de poner en producción la aplicación web, la propuesta es un éxito y la aplicación recibe tanto tráfico de entrada, que no puede sostenerse y comienzan los problemas de rendimiento de la aplicación, así como saturaciones y demás.

Surge entonces la necesidad de poder administrar y escalar los contenedores que se despliegan y que conforman la aplicación web para poder admitir todo el tráfico de demanda. Así aparece en escena Kubernetes, un orquestador que permite llevar a cabo todas estas tareas y más como actividades de monitorización.

Además, si este desarrollo se hubiera llevado a cabo mediante métodos tradicionales, ahora en el despliegue, habría que llevar a cabo réplicas de las máquinas virtuales para que se pudiera atender a todo el tráfico que recibe la aplicación web. Aquí, otro punto a favor en el desarrollo basado en contenedores, y que en este caso permite en cuestión de segundos llevar a cabo más copias, para Kubernetes.

Como ya se explicó en la sección 2.2, un cluster de Kubernetes está formado por nodos Worker que se encargan de desplegar las aplicaciones contenedorizadas y un nodo Master que se encarga de la administración de los primero nodos. Además, presenta diversos objetos básicos y *Controladores* como son los Pods, Service, ConfigMaps, Deployemnts, etc. Todos estos están explicados en la sección 2.2.

Para este Trabajo de Fin de Grado, se llevó a cabo la automatización del despliegue de la aplicación web mediante archivos YAML, que sirven para definir los distintos objetos necesarios para su despliegue con Kubernetes.

3.7.1. DESPLIEGUE DE APLICACIONES MEDIANTE ARCHIVOS YAML

Para automatizar el despliegue en Kubernetes, es fundamental el uso de archivos YAML, que hacen el "sustituto" de los archivos Compose que creaban entornos de contenedores en Docker. Sin embargo, con estos archivos lo que se va a automatizar es el despliegue de los distintos objetos y controladores que necesite la aplicación web desarrollada en este proyecto.

Para desplegar los diferentes contenedores que componen cada uno de los servicios se necesitarán Pods, y para que se puedan administrar, gestionar es necesario la creación de Deployments que atiendan a los distintos Pods. Además, se deben exponer los distintos Pods que se despliegen mediante Services para permitir la comunicación entre los distintos Pods y también se necesitarán ConfigMaps que almacenen el archivo de inicialización de la base de datos.

Una vez se tiene claro los distintos objetos a desplegar para el correcto despliegue de la aplicación, se llevó a cabo la implementación de los distintos elementos. Para hacer pruebas de forma local antes de llevar a cabo el despliegue en plataformas de despliegue de aplicaciones contenedorizadas, se utilizó Minikube. Esta herramienta permite crear de forma ligera una máquina virtual que despliega un cluster sencillo formado por un solo nodo para poder llevar a cabo pruebas.

3.7.1.1. DESPLIEGUE BASE DE DATOS CON ARCHIVO YAML

Para la base de datos, recordando el despliegue con Docker, consistía en levantar un contenedor con una imagen MySQL y tras esto, utilizaba el archivo de inicialización de nuestro equipo gracias al volumen que le permitía al contenedor tener los distintos archivos necesarios. Sin embargo, en Kubernetes los volúmenes no funcionan igual que en Docker y no enlaza la dirección especificada del contenedor con la dirección de nuestro equipo por lo que hay que cambiar esta forma de despliegue.

Para poder llevar entonces este despliegue, Kubernetes nos ofrece unos tipos especiales de contenedores denominados InitContainers cuyo cometido consiste en llevar a cabo la inicialización de otro contenedor de un pod. Así, se utilizó este tipo de contenedor para inicializar un contenedor con una imagen MySQL con la información del script de inicialización creado en la sección 3.2.2. Para la configuración de estos InitContainers, primero se creó un ConfigMap que contiene el archivo de inicialización de la base de datos (nuestro archivo init.sql).

Para crear este objeto, se creó un archivo tipo YAML que se denominó "despliegue-config.yaml" y el contenido de este archivo es el que se aprecia en el fragmento de código 35.

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: initsqlsource
data:
  source:
    https://raw.githubusercontent.com/AntonioLopezGarcia98/proyectoTFG/master/init.sql
```

Fragmento código 35: ConfigMap inicialización base de datos

Para configurar estos archivos YAML para que puedan ser utilizados por Kubernetes, lo primero es aplicar la primera característica `apiVersion` al valor "v1" que debe aplicarse siempre que se quiera desplegar un ConfigMap. Tras establecer que este archivo se utiliza para desplegar este tipo de objetos declarado en la opción `kind`, le establecemos un nombre al objeto el cual he decidido llamar "initsqlsource" en la opción `name` de sus metadatos. Por último, la forma en la que el objeto obtiene el archivo init.sql es mediante una url ya que como hemos comentado antes, Kubernetes no da la opción de enlazar directorios del contenedor y el equipo local como en Docker. Por ello, en un repositorio de la plataforma GitHub subimos el archivo init.sql y la url que nos da acceso a ese archivo es el que utilizamos para que el objeto tenga el archivo en cuestión.

El siguiente paso consistió en configurar el despliegue del contenedor que contiene la imagen MySQL y que toma como archivo de inicialización el que contiene este objeto ConfigMap llamado initsqlsource. Para ello, hubo que crear un Deployment, que esto a su vez implica la creación del Pods que contiene el contenedor que despliega la base de datos. Para el despliegue del Deployment, se utilizó otro archivo YAML denominado "despliegue-bd.yaml". La configuración del Deployment es la que aparece en los fragmentos de código 36 y 37.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: bdcine
  namespace: default
spec:
  selector:
    matchLabels:
      app: bdcine
  template:
    metadata:
      labels:
        name: bdcine
        app: bdcine
    spec:
      containers:
      - name: bdcine
        image: mysql:5.7
        env:
        - name: MYSQL_ROOT_PASSWORD
          value: root
        volumeMounts:
        - name: workdir
          mountPath: /docker-entrypoint-initdb.d
```

Fragmento código 36: Deployment base de datos parte 1

En el fragmento de código 36, se aprecia como tras declarar el tipo de objeto que se va a crear (en la opción `kind`), se estableció el nombre del Deployment en la sección `metadata` y el `namespace` al que pertenece (en nuestro caso "default"). El namespace sirve para diferenciar distintos objetos del mismo tipo y que puedan comunicarse entre sí según el espacio al que pertenezcan.

Dentro de las opciones en `spec` establecemos diferentes especificaciones para el funcionamiento correcto del Deployment como es el `selector` con `matchLabels` en el que se especifican los Pods que debe gestionar el Deployment que se está creando. Estos Pods que debe gestionar son aquellos que tenga la etiqueta bdcine. En `template`, configuramos el Pod que necesitamos que tenga a cargo el Deployment. Para ello, le establecemos en sus metadatos la etiqueta "bdcine" como en el `selector` que acabamos de comentar para que el Deployment se haga cargo de este Pod.

Dentro de `template`, especificamos las características del contenedor que deberá desplegarse en el Pod. Para ello, le otorgamos un nombre al contenedor (bdcine para seguir llamando a todo de la misma manera y no provocar confusiones), así como una imagen (en este caso mysql:5.7 como ya hicimos en el despliegue en docker-compose), la contraseña del usuario root mediante una variable, y se le adjudicó también un volumen que es en el que se almacena toda la información de la base de datos. Se le otorgó un nombre el cual fue "workdir" y la dirección donde debe montarse el volumen en el contenedor.

```
26        initContainers:
27        - name: install
28          image: busybox
29          env:
30          - name: SQLSOURCE
31            valueFrom:
32              configMapKeyRef:
33                name: initsqlsource
34                key: source
35          command:
36          - wget
37          - "-O"
38          - "/work-dir/init.sql"
39          args: ["$(SQLSOURCE)"]
40          volumeMounts:
41          - name: workdir
42            mountPath: "/work-dir"
43        dnsPolicy: Default
44        volumes:
45        - name: workdir
46          emptyDir: {}
```

Fragmento código 37: Deployment base de datos parte 2

En el fragmento de código 37, se configuró el initContainer que utiliza una imagen busybox y se le otorgó una variable denominada SQLSOURCE que contiene el ConfigMap que se había creado anteriormente con el enlace al script que inicializa la base de datos con sus

respectivas tablas y valores. Mediante la opción `command` se establecieron varios comandos necesarios para la ejecución del script y se utilizó como volumen el mismo que hemos creado anteriormente. Por último, creamos el volumen en concreto que está siendo utilizado tanto por el contenedor con la imagen MySQL y el initContainer el cual le tenemos que otorgar el mismo nombre que hemos indicado a estos contenedores que deben enlazarse el cual es "workdir" y al establecer la opción "emptyDir {}" estamos estableciendo el tipo de volumen que hemos desplegado el cual se utiliza cuando varios contenedores necesitan conectarse a él y que su contenido se mantiene aunque los contenedores sean eliminados o destruidos. Sin embargo, estos volúmenes se crean en el nodo donde está el pod, por lo que si el pod se elimina y se decide desplegar en otro nodo, los contenedores que se encuentran en el nuevo Pod no podrán acceder a este volumen.

Tras esto, solo faltaba establecer el Servicio con el que se van a exponer los Pods para que el resto de los servicios (en este proyecto la API REST) pudieran acceder a ellos.

La forma de crear un servicio es estableciendo en `kind` la opción "Service" y tras esto, otorgarle un nombre al servicio y utilizar un selector que busque de entre todos los Deployments que estén desplegados, aquellos que su nombre coincida con el que se ha establecido en `selector`. Además, hay que establecer el tipo de servicio que va a ser y los puertos en el que los Pods servirán su contenido. Esta implementación aparece en el fragmento de código 38.

```
49  apiVersion: v1
50  kind: Service
51  metadata:
52    name: bdcine
53    namespace: default
54  spec:
55    type: ClusterIP
56    ports:
57    - name: mysql
58      port: 3306
59      targetPort: 3306
60    selector:
61      app: bdcine
```

Fragmento código 38: Service de la base de datos

Se utilizó el tipo de servicio ClusterIP para que solo fuese accesible a nivel de cluster (es decir, que no sea accesible desde el exterior) y tras indicarle que debía conectarse mediante el puerto 3306, se indicó que debía enlazarse con el Deployment que tuviera el nombre de "bdcine" que es como se llamó al Deployment creado anteriormente y que debe tener este nombre para que la API llame a este servicio ya que si recordamos, en la conexión que hacia la API a la base de datos era a un host denominado "bdcine" por lo que debe tener este nombre para que llame al servicio correcto.

Con todas estas configuraciones, tenemos creado el despliegue de la base de datos en Kubernetes y lo siguiente a realizar son las configuraciones de los despliegues tanto de la API como del front-end pero ya podemos ir adelantando que tendrá la misma estructura, un

Deployment que gestione los Pods necesarios que contendrán los contenedores con las imágenes correspondientes, y sus servicios respectivos para que los distintos elementos se comunicaran entre sí y para que se pudiera acceder a ellos desde el exterior.

3.7.1.2. DESPLIEGUE API CON ARCHIVO YAML

Para construir el despliegue de la API, se llevó a cabo una estructura similar al despliegue de la base de datos, es decir; se creó un Deployment que gestionaba un Pod con la imagen de la API que habíamos creado, así como un Service que expondría el Pod para que pudiera accederse a él.

Para el Deployment, como hemos llevado a cabo antes, debemos darle un nombre al Deployment que en este caso será "api-cine". En las especificaciones, en el matchLabel del selector para que encuentre los Pods que debe gestionar se indicó que fueran aquellos que tuvieran la etiqueta api-cine también, por lo que en la creación del Pod, este debía tener la etiqueta con el mismo nombre. Además, se indicó la imagen con la que debe crearse el contenedor, la cual era alg804/api:vo (es decir, la imagen que creamos y subimos al repositorio Docker, de ahí la importancia de las imágenes Docker) y se indicó que expondríamos el puerto 8o. Además, se le otorgó unos recursos que más en adelante se explicarán ya que nos servirán para hacer escalable la aplicación, pero estos son por defecto y para desplegar la segunda versión, basta con crear otro archivo idéntico a este con la diferencia de que la imagen usada sea la que contiene la segunda versión. Todo este Deployment aparece en el fragmento de código 39.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api-cine
  namespace: default
spec:
  selector:
    matchLabels:
      app: api-cine
  template:
    metadata:
      labels:
        app: api-cine
    spec:
      containers:
      - name: api-cine
        image: alg804/api:v0
        resources:
          limits:
            memory: "128Mi"
            cpu: "500m"
        ports:
        - containerPort: 80
```

Fragmento código 39: Deployment API REST

Para el Service que se enlazaría con este Deployment, se le asignó el nombre "api-cine". En las especificaciones se indicó los puertos tanto del nodo como del contenedor y en el selector indicamos el Deployment que debe enlazar (que debe ser el Deployment api-cine que acabamos de configurar). Con todo esto, hacemos que la API sea accesible incluso por otros

elementos como el front llamando a la API mediante su nombre api-cine. El código de implementación de este servicio es el que aparece en el fragmento de código 40.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: api-cine
  namespace: default
spec:
  ports:
  - port: 8080
    targetPort: 80
    protocol: TCP
    name: http
  selector:
    app: api-cine
```

Fragmento código 40: Service API REST

3.7.1.3. DESPLIEGUE FRONT-END CON ARCHIVO YAML

Para el despliegue del front-end no varía la estructura que hemos llevado hasta ahora en los dos elementos anteriores de la aplicación. Lo primero será configurar el Deployment otorgándole un nombre al mismo, el cual será "front". Este nombre será el mismo que el que tendrá el Pod que levantará el contenedor con la imagen del front-end creado, por lo que tanto en la propiedad `matchLabel` como en la etiqueta `label` del Pod estará el mismo nombre para como hemos comentado antes, el Pod sea gestionado por el Deployment. Además, este Pod deberá utilizar la imagen que hemos creado que contiene el front-end de la aplicación la cual era "alg804/front:v0" y le otorgamos los recursos predeterminados al Pod , así como establecer como puerto para acceder a él el puerto 80.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: front
  namespace: default
spec:
  selector:
    matchLabels:
      app: front
  template:
    metadata:
      labels:
        name: front
        app: front
    spec:
      containers:
      - name: front
        image: alg804/front:v0
        resources:
          limits:
            memory: "128Mi"
            cpu: "100m"
        ports:
        - containerPort: 80
```

Fragmento código 41: Deployment front-end

Como se ha implementado tanto en la base de datos como en la API, falta el Service que exponga el Pod y que permita acceder a él. El nombre del Service será el mismo que se ha utilizado para el Deployment, y como en los casos anteriores, para que el servicio se una al Deployment y lo exponga, en el selector, debemos establecer que debe buscar los Deployment que se llamen "front". La implementación se muestra en el fragmento de código 42.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: front
  namespace: default
spec:
  ports:
  - port: 80
    targetPort: 80
    protocol: TCP
    name: http
  selector:
    app: front
```

Fragmento código 42: Service front-end

Con esta implementación, ya se disponía de todos los elementos necesarios para desplegar la aplicación utilizando Kubernetes, ya que, para cada elemento, hemos implementado su Pod que se despliega el contenedor con la imagen correspondiente tanto para la API como para el front-end, al igual que hemos implementado la inicialización de la base de datos con el uso de los initContainers. Además, les hemos administrado un Deployment para que gestionen los Pods y que en caso de que alguno de ellos falle, que se eliminen y se creen otros para que en ningún momento se caiga alguno de estos elementos. Además, les hemos establecido mediante un Service, una forma de poder comunicarse entre sí los distintos elementos de la aplicación y de esta forma, el front-end se comunica con la API y esta con la base de datos.

3.7.1.4. INGRESS

Como ya se comentó en secciones anteriores, había un problema en cuestión con la conexión entre el front-end y la API REST del back-end. Debido a la forma de funcionar de JavaScript usando el buscador en el que se cargaba la página, no se podía establecer en los métodos `ajax()` la url llamando al servicio de la API REST utilizando el nombre del contenedor que lo desplegaba.

En el archivo Compose y en la construcción del entorno de contenedores se solucionó este problema estableciendo una conexión entro los distintos puertos de conexión de los contenedores con los puertos del equipo local en el que se desplegaban y por ello en la url de los métodos `ajax()` se llamaba al puerto local.

En Kubernetes, la conexión entre servicios se lleva a cabo con los nombres de los propios servicios, pero esto funciona si se utilizara la aplicación desde el contenedor que se encuentra en el Pod que despliega el front-end.

Para solucionar este inconveniente, se utilizó un elemento de Kubernetes denominado Ingress. Los Ingress son elementos que exponen rutas http y https desde fuera del clúster a los

servicios dentro del clúster. Se configuran para proporcionar a los distintos servicios direcciones accesibles externamente y de forma fija. Con estos elementos, se consiguió que desde nuestro buscador web de forma local, cuando utilicemos una dirección determinada, nos redireccione al servicio que deseamos. Con esto, solucionamos el problema de las direcciones en las funciones AJAX. Por tanto, se creó un Ingress para nuestra API para poder acceder a ella sin problemas tanto desde el front-end como desde el buscador.

Para ello, se establecen algunos parámetros que vienen ya predefinidos como la `apiVersion` y en los metadatos una opción que debemos establecer en "true". Declaramos que este objeto se trata de un ingress en `kind` y le damos un nombre al ingress que en este caso es api-ingress.

En las especificaciones del ingress le aplicamos las reglas necesarias para que enrute el tráfico al servicio y para ello, en la opción host indicamos cómo será la dirección que nos redireccionará al servicio. Para la API, se utilizó el host www.api-cine.es. Por último, declaramos que este Ingress debe enviar todo el tráfico que se dirija al host especificado al servicio api-cine que hemos creado antes que además estaba en el puerto 8080. El código de esta implementación es el siguiente:

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: api-ingress
  annotations:
    nginx.ingress.kubernetes.io/use-regex: "true"
spec:
  rules:
    - host: www.api-cine.es
      http:
        paths:
        - path: /
          backend:
            serviceName: api-cine
            servicePort: 8080
```

Fragmento código 43: Ingress API REST

También se decidió implementar un ingress para el front-end, y así escribiendo en el buscador una ruta, directamente se podría acceder a la aplicación web. Las configuraciones de este ingress deben ser iguales que el que acabamos de implementar a diferencia de que el nombre de este ingress será frontend, el host será www.cineual.es y que el servicio al que debe enrutar el tráfico es al servicio front el cual es el que creamos que su puerto es el 80. El código de este ingress es el que se aprecia en la siguiente imagen:

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: frontend
  annotations:
    nginx.ingress.kubernetes.io/use-regex: "true"
spec:
  rules:
  - host: www.cineual.es
    http:
      paths:
      - path: /
        backend:
          serviceName: front
          servicePort: 80
```

Fragmento código 44: Ingress front-end

Pero con estos elementos no estaba todo resuelto, ya que como se ha explicado, con este elemento, cuando busquemos en el buscador estos hosts, se nos redireccionará a los distintos servicios. Pero por ello, hay que cambiar las urls de los distintos métodos AJAX. Por este motivo, se debieron eliminar las imágenes que se habían creado de los front-end y en los archivos tanto main.js como main2.js en las urls, modificarlos y en vez de que tuvieran la forma "http://localhost:81/rutaAPI" que fuesen de la forma "http://www.api-cine.es/rutaAPI".

Una vez se cambiaron las url en todas las funciones de ambos archivos, se volvieron a crear las imágenes con los mismos pasos que se llevaron a cabo en la sección 3.6 y así, se disponía de todos los elementos necesarios listos para su despliegue en minikube y comprobar su funcionamiento para más tarde desplegarlo en una plataforma de despliegue de contenedores.

3.8. DESPLIEGUE Y PUESTA EN PRODUCCIÓN DE LA APLICACIÓN

El último paso en del despliegue de aplicaciones escalables con Kubernetes, consiste en el propio despliegue de la aplicación mediante las plataformas Rancher y OpenStack. Como se comentó en los apartados 2.8 y 2.9, se utilizaron estas plataformas en el caso de Rancher para desplegar la aplicación contenedorizada y OpenStack se utilizó como proveedor de infraestructura para que Rancher pudiera llevar a cabo el despliegue del cluster que contiene la aplicación implementada.

3.8.1. DESPLIEGUE DE APLICACIÓN

Una vez llevados a cabo toda esta serie de pasos y configuraciones de los apartados 2.9.1 y 2.10.1, se puede llevar a cabo el despliegue y la puesta en producción de la aplicación. Para ello, utilizando el comando

```
Kubectl apply -f  "nombre del archivo YAML"
```

con cada uno de los archivos creados en toda la sección 3.7.1, se crean los distintos Deployments, Pods y Services que conforman toda la aplicación implementada.

Para comprobar que se han desplegado, basta con avanzar hasta la sección *Default* del cluster en la zona superior del menú de la figura 10 y en la pestaña *Workloads* aparecen todos

los Deployments creados, así como en *Service Discovery* se encuentran todos los Servicios desplegados.

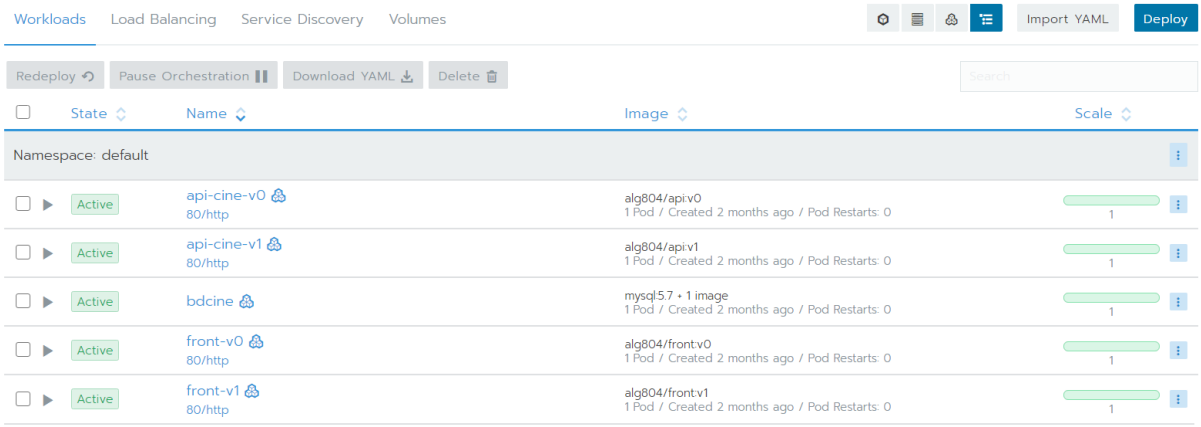

Figura 25: Despliegue aplicación implementada

Decidí llevar a cabo el despliegue de las dos versiones para comprobar el correcto funcionamiento de la aplicación como se ve en la figura 25. Además, de cada Deployment se puede conocer toda su información como el Pod que está controlando el Deployment, así como aumentar el número de Pods y demás si entramos en un Deployment pulsando en él.

Sin embargo, a la hora de acceder al servicio front-end de la aplicación surgió otro problema. Se trataba del ingress creado mediante el archivo YAML, ya que este ingress de forma manual, nos obliga a añadir la dirección IP que nos da Rancher y el DNS que elaboramos nosotros (en este caso www.cineual.es y www.api-cine.es) al archivo "Hosts" de nuestro equipo, por lo que en realidad, el acceso externo a los servicios solo están disponibles en el equipo donde se configure ese fichero. Por ello, en vez de crear nosotros manualmente mediante un fichero YAML el ingress, debemos indicar a Rancher que nos cree un ingress que sea público y accesible para cualquier equipo. Para ello, desplazándonos en el panel principal al cluster en la sección "Default" y ejecutamos la opción "Add Ingress" en "Load Balancing". En esta sección nos aparece una serie de opciones como se observa en la figura 24.

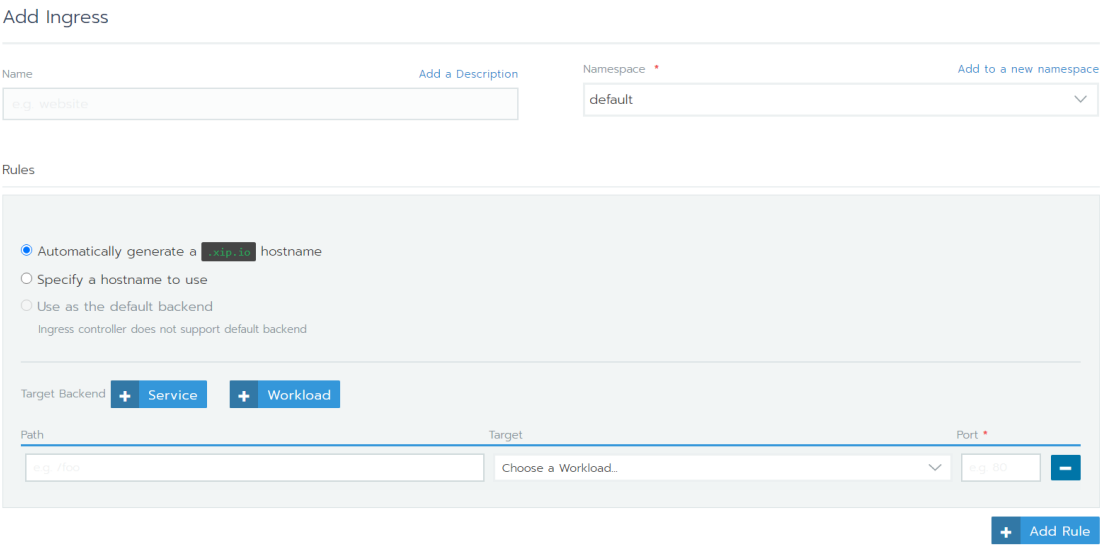

Figura 26: Ingress proporcionado por Rancher

En esta ventana de configuración se creará una dirección pública de la forma "name.default.ipdelnodoworker.xip.io" que permitirá a cualquier equipo acceder a ese servicio introduciendo esta url. Además, se pueden establecer distintos *paths* para distintos servicios estableciéndole aquel que el desarrollador cree oportuno e indicando a qué servicio acceder en la opción "Target" donde nos aparecerá una lista con los servicios que tenemos desplegados en ese instante, por lo que lo primero es desplegar todos los servicios que hemos descrito en los apartados anteriores menos los Ingress que habíamos elaborado manualmente.

Pero como hemos comentado en apartados anteriores de la problemática de la ip para conectar frontend con la API, debemos cambiar los archivos main.js y main2.js, concretamente las urls de los métodos `ajax()` poniendo ahora las que van a recibir los servicios de las dos versiones de la API. Una vez realizados los cambios, desplegamos todos los deployments y servicios creados y una vez ahí, creamos los ingress. Una vez creados (que para el caso del frontend se queda de la forma cineual.default.192.168.66.27.xip.io y en caso de la api es apicine.default.192.2168.66.27.xip.io), simplemente con introducir en el buscador la dirección url otorgada por Rancher, se accede a la aplicación web.

Una vez llevados a cabo estos cambios, ya si se puede acceder a la aplicación implementada accediendo al servicio front-end que se ha implementado en este Trabajo de Fin de Grado.

3.7.2. ESCALABILIDAD DE LA APLICACIÓN

Llegados a este punto, se ha logrado llevar a cabo un despliegue de una aplicación en Kubernetes gracias a Rancher. Sin embargo, en un hipotético caso en el que esta aplicación se desplegara de forma real por parte de la universidad, y obtuviera un tráfico de entrada muy alto, la aplicación colapsaría y sufriría problemas de rendimiento, latencia e incluso fallos de funcionamiento.

Por ello, con Kubernetes se puede aplicar su autoescalabilidad que se comentó como una de las ventajas que lo sitúa por encima de otros orquestadores como Docker Swarm comentado en la sección 2.2 para que en casos como el hipotético descrito anteriormente, la aplicación pueda autoescalarse sin problemas hasta un número de réplicas que el administrador de la aplicación desee.

Para aplicar esta capacidad de autoescalabilidad, Kubernetes ofrece un objeto denominado HPA (Horizontal Pod Autoscaler). Este objeto aporta la capacidad de escalar automáticamente la cantidad de pods según distintas métricas siendo una de las más utilizadas el uso de CPU. A diferencia de los Deployments y los ReplicaSet que mantenían el numero de Pods en el indicado en el despliegue de estos elementos (es decir, si se desplegaba estos elementos indicando que el número de réplicas a implementar era de 10, tanto el Deployment como el ReplicaSet se encargaban de mantener en todo momento 10 réplicas de los pods que gestionaban estos elementos), los HPA permiten aumentar las réplicas de Pods hasta cierto número, indicado en la creación de estos objetos y disminuir hasta el mínimo establecido cuando no se necesiten tantos recursos.

Para obtener estas métricas, el HPA normalmente obtiene estos valores de una serie de APIs agregados en los Pods, como es `metrics.k8s.io`.

En la creación de un HPA se determinan ciertas configuraciones como pueden ser definir un mínimo y máximo de réplicas de un deployment, definir las condiciones del "stress" (condición que indica cuando se aumenta el número de pods, como por ejemplo un porcentaje de uso de la CPU); el tiempo entre consulta de las métricas de uso de los pods, etc.

Todo esto, se configura de forma manual mediante archivos YAML, por lo que se elaboraron otros manifiestos en los que instanciaron distintos HPA para cada elemento de nuestra aplicación y, además, en los archivos YAML en los que instanciábamos los deployments se añadieron los recursos que los pods utilizarían ya que en los Deployments se establecieron los recursos en la sección *resources* y *limits* en la que se asignaban unos valores de uso de memoria y CPU.

Sin embargo, para poder desplegar estos elementos, hay que activar las herramientas de monitorización que se encargan de obtener las métricas necesarias para el correcto funcionamiento de los HPA. Otra ventaja de Rancher, es que desde una pestaña, se puede activar todo el sistema de monitorización del cluster, siendo esta pestaña la denominada "Tools" que se encuentra en la zona superior del menú y en él, se selecciona "Monitoring". Tras activarlo indicando los recursos de CPU y memoria que debería tomar del propio cluster para llevar a cabo estas actividades de monitorización, en el panel del cluster, los gráficos cambian un poco e indican el uso actual del cluster de distintos recursos como la CPU y la memoria, además de activar distintas herramientas como Grafana y Kiali que se estudiarán en el Complemento del Trabajo de Fin de Grado.

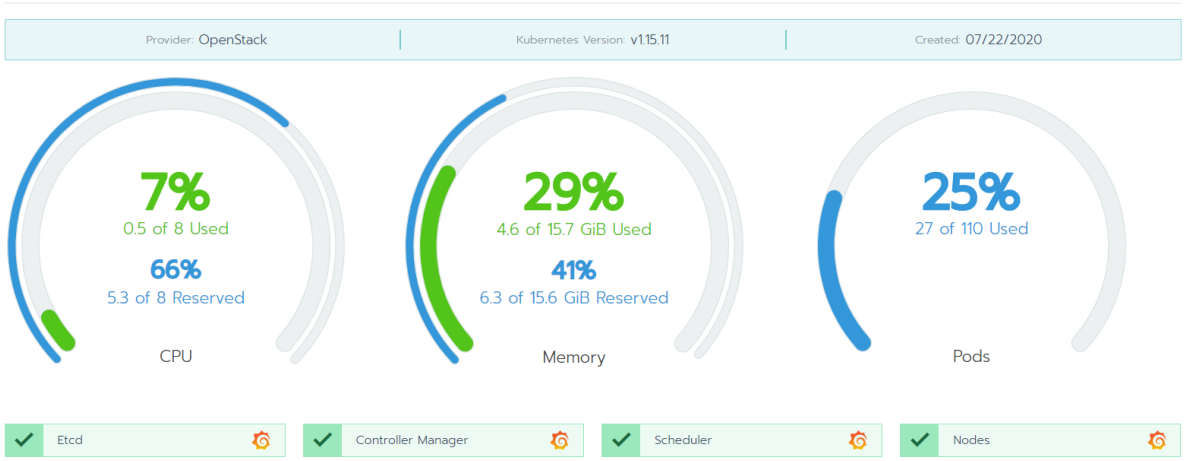

Figura 27: Monitorización activada

Tras activar la monitorización en el cluster, solo falta elaborar los elementos HPA en el que indicamos el porcentaje a partir del cual se deben crear más pods, así como más configuraciones. Para ello, crearemos un archivo YAML denominado "despliegue-hpa.yaml" en el que estableceremos que el objeto es de tipo HorizontalPodAutoscaler, así como un nombre para cada HPA (el que controla el front se denominó frontscaler y el de la API se llamó apiscaler) y una vez establecimos los deployment objetivo a los que tiene que enlazarse cada HPA, se les indicó que el número de réplicas estarán entre 1 y 10 y que estas replicas aumenten cuando se llegue a un 15% del uso de la CPU (el bajo porcentaje es para que se creen con mayor facilidad a la hora de hacer pruebas). El código donde se implementa todo esto se aprecia mejor en la siguiente figura.

```yaml
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: frontscaler
spec:
  scaleTargetRef:
    apiVersion: apps/v1beta1
    kind: Deployment
    name: front
  minReplicas: 1
  maxReplicas: 10
  targetCPUUtilizationPercentage: 15

---

apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: apiscaler
spec:
  scaleTargetRef:
    apiVersion: apps/v1beta1
    kind: Deployment
    name: api-cine
  minReplicas: 1
  maxReplicas: 10
  targetCPUUtilizationPercentage: 15
```

Figura 28: HPA

Con esto, una vez hemos desplegado la aplicación, simplemente desplegamos también este objeto con el comando

```
kubectl apply -f "nombre del archivo"
```

y pasamos a llevar a cabo pruebas para comprobar la escalabilidad de la aplicación [12].

3.7.2.1. PRUEBA ESCALABILIDAD DE LA APLICACIÓN

Una vez desplegados los HPA creados en la sección anterior, llega el momento de comprobar cómo aumentan los pods cuando se alcanza el límite establecido, en este caso un 15% del uso de la CPU.

Tras comprobar que ambos HPA han sido desplegados con el comando

```
kubectl get hpa
```

y comprobar que afectan a los dos deployments referenciados, pasamos a utilizar la herramienta denominada Apache Benchmark, que permite realizar pruebas de carga, otorgando la posibilidad de realizar un número determinado de peticiones totales y peticiones simultáneas.

Con esta herramienta, en este proyecto se realizaron seguimientos observando cómo se creaban más réplicas a medida que se necesitaban más recursos. Por ello, para llevar a cabo las peticiones basta con utilizar el comando

```
ab -n 100000 -c 100 http://cineual.default.192.168.66.27.xip.io/
```

que fue ejecutado en un contenedor con una imagen que contenía apache. Tras ejecutar eso, para poder llevar a cabo el seguimiento, se debía ejecutar el comando

```
kubectl get horizontalpodautoscalers.autoscaling –watch
```

para que muestre en la terminal la escalabilidad de la aplicación.

En este despliegue, se detectaron problemas debido a la imposibilidad de activar la opción Monitoring (la cual es necesaria para poder desplegar elementos como los HPA), por algún motivo que se desconoce y por tanto los HPA no podían llevar a cabo su función.

La única solución que se encontró fue unir ambos nodos, y crear así un cluster cuyo único nodo tiene todas las funciones. Así, este nuevo cluster ("denominado tfgallinone") lleva a cabo el despliegue de todos los elementos que necesita nuestra aplicación y por último los HPA, no sin antes realizar las configuraciones que hemos detallado en los apartados anteriores, pero ahora en este nuevo cluster. Tras esto, se desplegó la aplicación web, así como los HPA correspondientes y se ejecutaron consultas con Apache Benchmark. El funcionamiento de los HPA se aprecia en la figura 29.

```
PS C:\Users\lopez\onedrive\escritorio\despliegueTFG> kubectl get horizontalpodautoscalers.autoscaling --watch
NAME          REFERENCE              TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
apiscaler     Deployment/api-cine    1%/15%     1         10        1          30m
frontscaler   Deployment/front       1%/15%     1         10        1          30m
frontscaler   Deployment/front       211%/15%   1         10        1          31m
frontscaler   Deployment/front       211%/15%   1         10        4          31m
frontscaler   Deployment/front       211%/15%   1         10        8          32m
frontscaler   Deployment/front       211%/15%   1         10        10         32m
frontscaler   Deployment/front       42%/15%    1         10        10         32m
frontscaler   Deployment/front       35%/15%    1         10        10         33m
frontscaler   Deployment/front       1%/15%     1         10        10         34m
PS C:\Users\lopez\onedrive\escritorio\despliegueTFG>
```

Figura 29: HPA en funcionamiento

Como se aprecia en la imagen, a medida que llegan las peticiones y se supera el 15% de uso de la CPU (marcado en la columna TARGETS), empiezan a aumentar el número de réplicas hasta que baja el uso de porcentaje de CPU en las réplicas de los Pods. En este caso, porque se ha decidido detener la observación cuando había 10 réplicas, pero si se vuelve a ejecutar el mismo comando para ver la situación actual de las réplicas, solo hay una réplica ejecutándose, debido a que se han detenido las peticiones y no necesita de tantas réplicas.

Con esto, hemos logrado desarrollar una aplicación web con distintos servicios y se ha conseguido que esta aplicación sea escalable dependiendo de varios criterios que en este caso ha sido según el uso de CPU, pero pueden ser por otros criterios.

Tras esto, ya se dispone de una aplicación escalable mediante el uso de contenedores y la orquestación de estos con Kubernetes. A lo largo de este documento se ha reiterado en numerosas ocasiones las diferencias y lo que supone este tipo de desarrollo frente a otros métodos de virtualización como principalmente las máquinas virtuales. En este último caso haber llevado a cabo esta función de escalabilidad en este tipo de virtualizaciones, habría supuesto simplemente replicar las distintas instancias de las máquinas virtuales que se necesitaran copiar, ya sea las que despliegan el front-end, o la API REST. Además, el uso de otros orquestadores como Docker Swarm habría implicado llevar a cabo manualmente cada réplica, lo que demuestra una de las grandes virtudes de Kubernetes frente al resto de orquestadores de contenedores.

De todas las ventajas que presenta el desarrollo basado en contenedores, se ha hecho especial importancia en el consumo de recursos, así como los tiempos a la hora de implementar y desplegar los desarrollos, ya que suponen una nueva forma tanto de programar como de pensar en las arquitecturas de las aplicaciones. Pero también, esta característica de autoescalado supone un gran avance en las puestas en producción de las aplicaciones basadas en microservicios.

3.9. APLICACIÓN DESARROLLADA

En este último apartado, se mostrará la aplicación en concreto, tanto su primera versión como su segunda versión. Se mostrará cómo lleva a cabo la aplicación web cada una de las solicitudes que se han implementado y comprobar cómo se realizan de forma satisfactoria.

Por ello, lo primero va a ser empezar con la primera versión. La página principal se ve como se aprecia en la siguiente figura:

Figura 30: Página principal de la aplicación web

Como se puede observar en la imagen, en la zona superior tenemos el menú principal con el que se puede acceder al resto de funcionalidades de la aplicación como ver las entradas que tiene el, así como poder eliminar aquellas que no desee denominado “Mis Entradas”, la posibilidad de obtener entradas en la opción “Conseguir Entradas” y poder modificar alguna entrada con la opción “Modificar Entradas”.

Además, ya aquí se puede apreciar una solicitud tipo GET y es la lista de películas que explicamos su implementación en puntos anteriores. Como se ve en la lista, se muestran el título, sinopsis y duración de las películas que posee la aplicación en su base de datos.

Incluso como se comentó en la implementación de la aplicación, gracias a Bootstrap, la aplicación disponía de un “responsive design” lo que le permitía adaptarse a dispositivos móviles donde los textos se adaptan a la resolución de la pantalla, así como la desaparición del menú, apareciendo en sustitución de este un botón que al pulsarlo se despliegan las distintas opciones de navegación como se observa en la figura 31.

Figura 31: Responsive design de la aplicación

Lo siguiente será conseguir una entrada para más tarde verla en la opción "Mis Entradas" y así vemos cómo se lleva a cabo una solicitud de tipo "POST". Para ello, tras navegar a la opción "Conseguir Entradas" donde aparece los formularios implementados en las secciones correspondientes a su implementación el menú principal que aparece en todas las páginas y el formulario con el que creamos una entrada, escribiendo el id de la función a la que queremos asistir, así como el id del asiento en el que queremos sentarnos.

Figura 32: Conseguir entrada

Para ver que se crea la entrada correctamente, creamos una entrada con el id de función 1 y el id de asiento también 1. También vamos a crear otra entrada con el mismo id de función y un id de asiento distinto que va a ser 2. Una vez pulsamos el botón "Conseguir Entradas"

podemos ir a la página “Mis Entradas” y ver como en esta página aparecen las entradas. En esta página también tenemos el menú principal y aparece una lista con las entradas que posee un usuario. La información que aparecen sobre las entradas es el id de la entrada en sí, el id de la función y el id del asiento que tiene asignados esa entrada. Además, a la derecha posee un botón que permite eliminar una entrada.

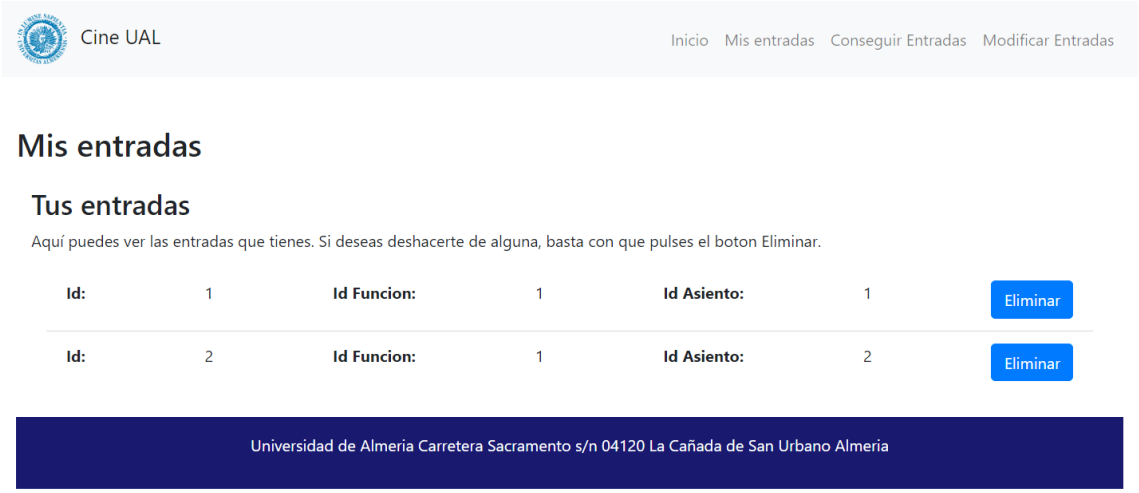

Figura 33: Mis entradas

Podemos aprovechar y llevar a cabo una solicitud tipo “DELETE” eliminando una entrada. Para ello, basta con pulsar en el botón eliminar. Vamos a eliminar la segunda entrada y tras pulsar el botón “Eliminar”, desaparece de la lista de entradas como se aprecia en la siguiente imagen:

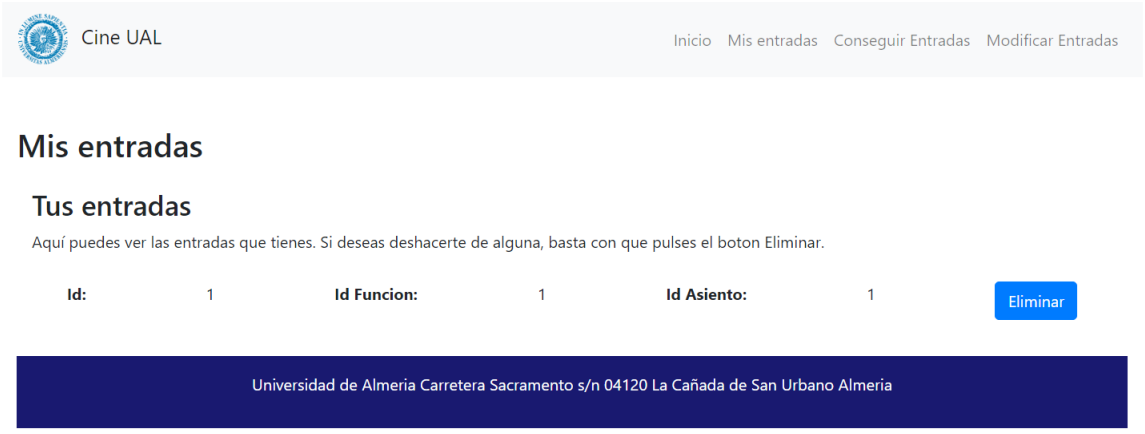

Figura 34: Mis entradas entrada eliminada

Tras probar los tres tipos de solicitudes, falta por probar la de tipo “PUT”. Y para ello, vamos a modificar la entrada que nos queda. Para ello, en “Modificar Entradas”, nos aparece el cuestionario en el que debemos indicar qué entrada queremos modificar y qué valores queremos que tenga ahora en su función y asiento. Para ello, vamos a escribir el id de la entrada que queremos cambiar (el cual es la entrada 1), y vamos a cambiarle la función a la función 2 y el asiento vamos a otorgarle el asiento 3 como se aprecia en la siguiente imagen:

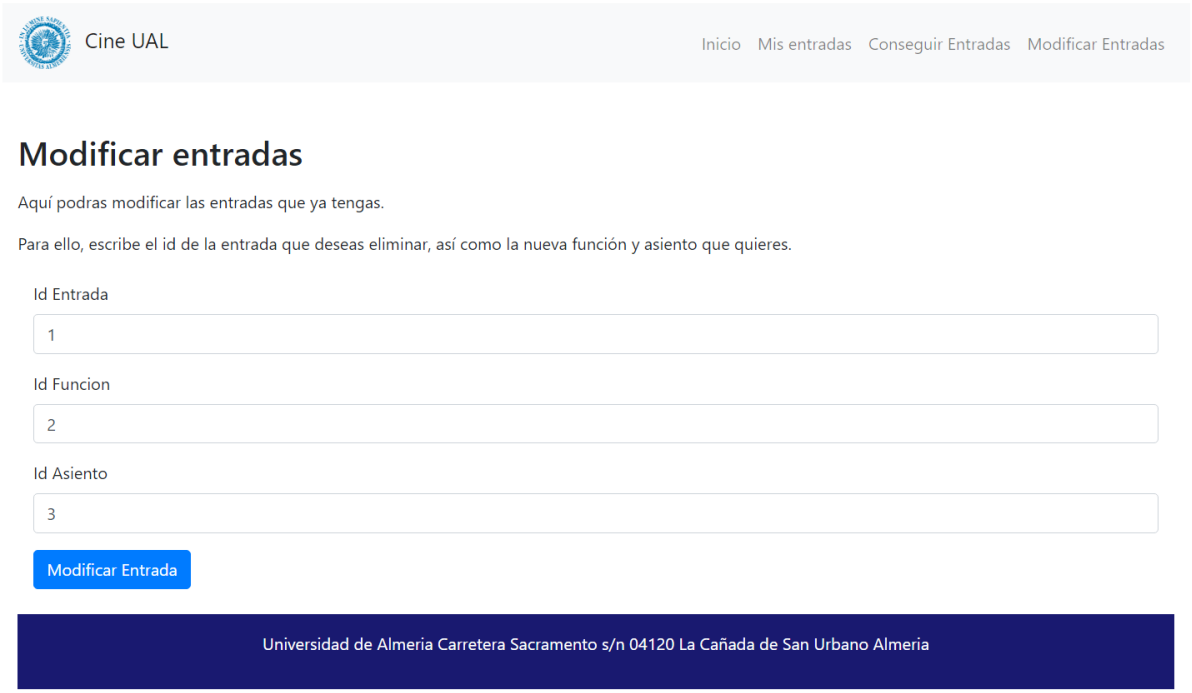

Figura 35: Modificar entrada

Tras pulsar el botón Modificar Entrada, para comprobar que esa entrada se ha modificado con los valores que hemos escrito, nos vamos otra vez a la página "Mis Entradas" donde ahora veremos cómo la entrada sigue siendo la 1 pero la función que tiene es la 2 y el asiento es el 3 como se aprecia en la figura 33.

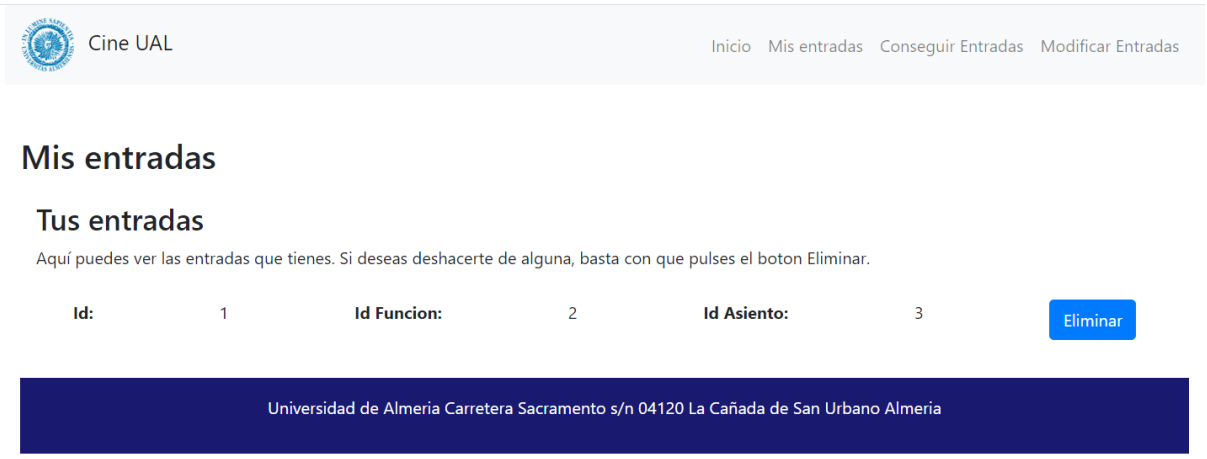

Figura 36: Entrada modificada

Con esto, hemos visto todas las funcionalidades de la página web, pero esto solo ha sido la primera versión que como se ha podido observar, en algunos aspectos es muy poco intuitivo de cara al usuario, como las funcionalidades de crear entradas o modificarlas en las que en vez de poder elegir entre una lista de funciones y asientos es todo escribiendo manualmente los id de las funciones y demás. Por ello, en la segunda versión de la aplicación que vamos a ver a continuación se apreciarán las funcionalidades descritas en la implantación de la segunda versión del front-end.

El uso del id no es el más adecuado en los formularios y se llevó a cabo de esta forma para aprender a cómo realizar los formularios y las peticiones correspondientes a la API. Sin embargo, en la segunda versión como se va a observar, no se usa ya inputs en los que el usuario debe introducir un id, sino que muestra otra información más descriptiva que hace que de cara al usuario, la aplicación sea más intuitiva.

3.10.1. APLICACIÓN DESARROLLADA SEGUNDA VERSIÓN

En la segunda versión, ya en la página principal observamos cambios importantes. Como se comentó en su implementación, esta segunda versión tendría la posibilidad de elegir una fecha y tras seleccionarla, podríamos ver las funciones que se proyectan en esa fecha, así como más información de estas. Tras entrar en la página en el inicio, disponemos de la lista desplegable con numerosas fechas y tras seleccionar una (por ejemplo, el 11/03/2020) y pulsar el botón "Mostrar Funciones", nos aparece las distintas funciones de las películas como se aprecia en la siguiente imagen:

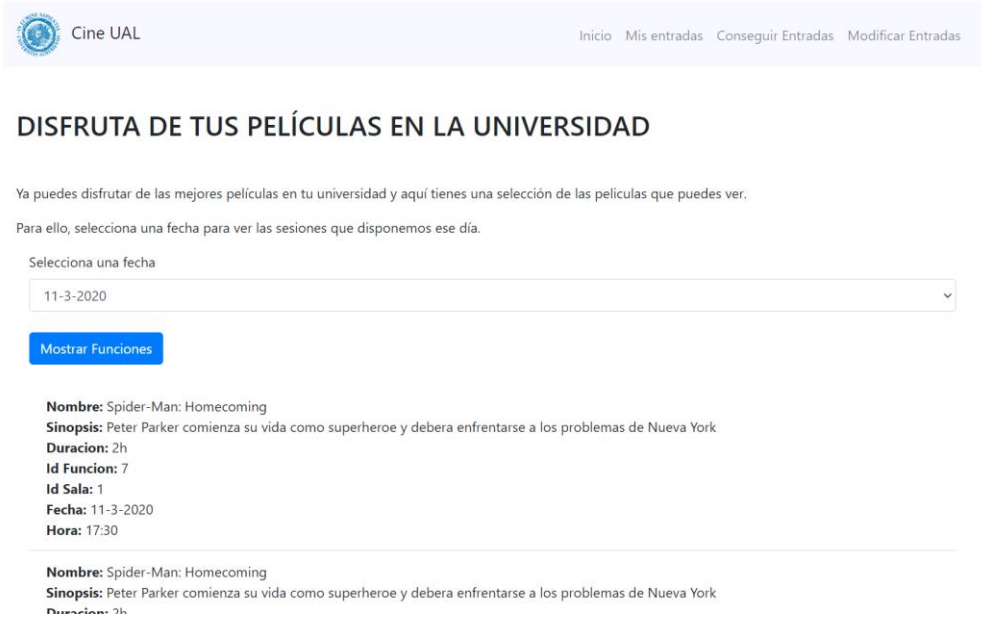

Figura 37: Página principal versión 2

Ahora vamos a obtener una entrada para ver las mejoras respecto a la versión anterior. En esta nueva página, lo primero que vemos que ha cambiado es el formulario, pues ahora tiene una lista desplegable con todas las funciones que el cine puede ofrecer, además de los botones de "Seleccionar Asiento" y "Conseguir entrada. Una vez hemos seleccionado una función de la lista entera, al pulsar el botón "Seleccionar Asiento" aparece otra lista desplegable con los asientos de esa función en concreto (que en este caso vamos a seleccionar la función de la película "Joker" con fecha 6/3/2020 a las 16:30 y el asiento correspondiente a la fila B número 2) y una vez seleccionado todos estos valores, tras pulsar en el botón "Conseguir entrada", ya dispondríamos de una entrada pero con todos estos datos seleccionados.

Figura 38: Conseguir entradas versión 2

Una vez creada la entrada, vamos a ver su correspondiente registro en "Mis Entradas" donde ahora vemos cómo aparece más información de la entrada que acabamos de crear, mostrándonos información extra como el asiento que tenemos, la fecha y hora de la función y la película que vamos a ver, simulando cómo sería de forma más realista esta página web. Esta nueva forma de ver las entradas se aprecia en la figura 36.

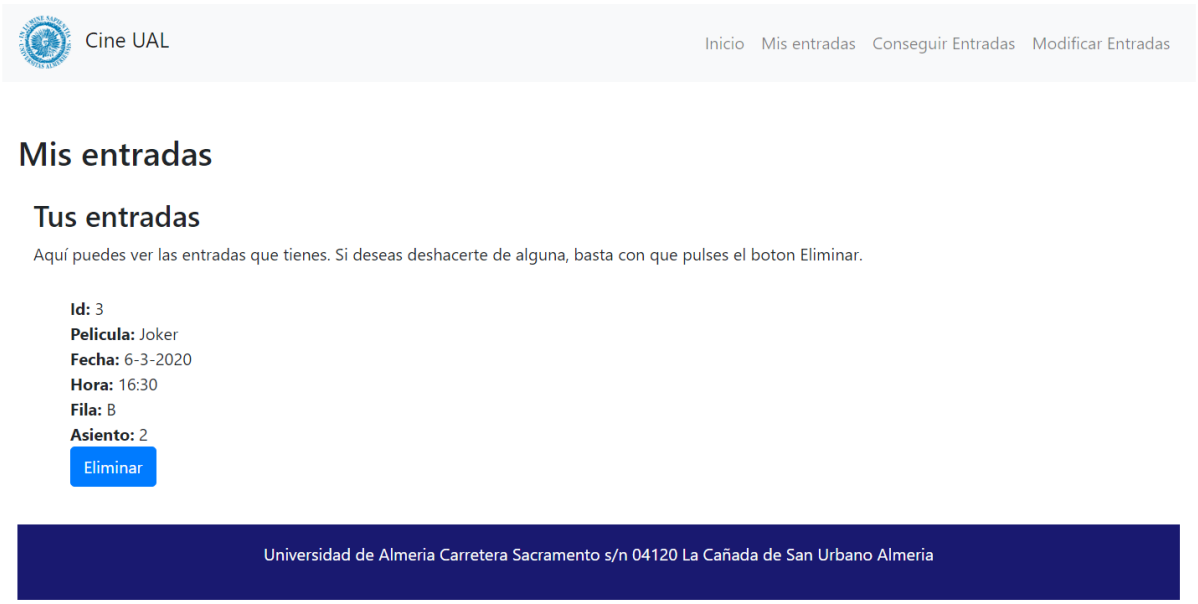

Figura 39: Mis entradas versión 2

Antes de eliminar esta entrada, vamos a modificarla para ver cómo modificar una entrada en esta nueva versión es mucho más intuitivo que en la versión anterior. Para ello, nada más entrar en la página "Modificar Entradas", lo primero que vemos es una lista desplegable con las entradas que tenemos, lo cual ayuda más que tener que recordar el id de la entrada que queríamos modificar. También vemos otra lista desplegable con todas las funciones que hay en

la base de datos, como cuando queríamos obtener una entrada, así se puede elegir mejor la función que ahora se desea tener. Al pulsar el botón "Seleccionar nuevo asiento" aparece otra lista desplegable con los asientos de la función que se ha seleccionado y por último el botón "Modificar Entrada" que tras introducir los nuevos datos y pulsarlo, se llevaran a cabo los cambios. En este caso, vamos a cambiar la única entrada que tenemos (la cual era la de Joker) por la película 1917 a la función del 9/3/2020 a las 17:00 con la butaca que se encuentra en la fila B y número 3 como se puede observar en la siguiente imagen:

Figura 40: Modificar entrada versión 2

Tras pulsar el botón Modificar Entrada, podemos ver los cambios realizados si nos vamos a la sección Mis Entradas donde como se ve en la siguiente imagen, los datos de la entrada que teníamos han cambiado:

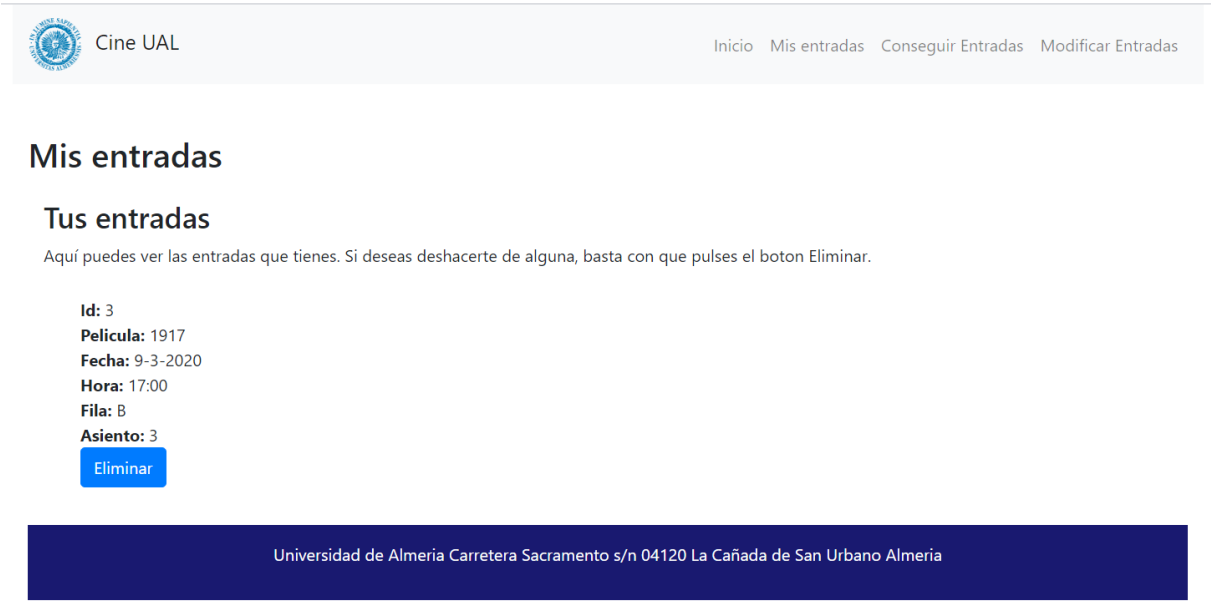

Figura 41: Entrada modificada versión 2

Para eliminar entradas, no ha cambiado respecto a la versión anterior pues basta con pulsar el botón Eliminar de la entrada en cuestión que el usuario por cualquier motivo ya no la desea.

3.10. RESUMEN DEL CAPÍTULO

En este capítulo, se ha llevado a cabo todo el desarrollo de la aplicación escalable.

Se ha llevado a cabo la implementación de cada uno de los servicios de los que conforma la aplicación web, utilizando una metodología de desarrollo basándose en contenedores, apreciando las ventajas que aporta esta tecnología frente a otras virtualizaciones comentadas como las máquinas virtuales creadas por Hipervisores.

Se crearon entornos de contenedores mediante Docker Compose para que los distintos servicios pudieran conectarse entre sí, así como las imágenes Docker mediante Dockerfile que permiten un continuo ciclo de desarrollo basado en versiones y que permiten a su vez su despliegue en orquestadores de contenedores.

Tras exponer la necesidad y motivos de escalabilidad de la aplicación, se llevó a cabo la instalación de Rancher en una máquina virtual de OpenStack, y se vinculó el orquestador al proyecto de OpenStack para autoabastecerse de las instancias que necesite.

Una vez configurada la plataforma de despliegue de aplicaciones basadas en contenedores, se crearon elementos HPA que sirven para autoescalar la aplicación según diversos criterios como el uso de Memoria o CPU de los nodos y se llevaron a cabo pruebas de autoescalado enviando un numero masivo de peticiones simultáneas.

Por último, se mostraron la dos versiones de la aplicación desde el punto de vista del usuario que la consume, llevando a cabo distintas tareas como ver las películas que oferta la aplicación, se obtuvieron entradas, y se modificaron, observando siempre el resultado de cada una de estas acciones desde la lista de entradas que posee el usuario.

4. CONCLUSIONES Y TRABAJO FUTURO

En este último capítulo se comentarán los resultados del trabajo realizado durante este Trabajo de Fin de Grado y se propondrán futuras líneas de desarrollo del mismo.

4.1. CONCLUSIONES

Este Trabajo de Fin de Grado, tenía como objetivo principal el desarrollo y despliegue de una aplicación escalable con Kubernetes, para descubrir una nueva tecnología que lleva pocos años en el mercado, pero cada vez más es usado por empresas multinacionales como Amazon o Netflix, la cual es Docker y sus contenedores.

También se ha logrado observar las ventajas que supone tanto en desarrollo como despliegue en medios virtualizados estas tecnologías (tanto Docker como Kubernetes) frente a modos tradicionales de virtualización como son las máquinas virtuales, de las cuales se obtiene como principales puntos a favor la gran diferencia de recursos y tiempo en el despliegue de las aplicaciones siendo mucho menor en los contenedores Docker utilizados a lo largo del desarrollo de la aplicación.

Como se ha comentado anteriormente, que empresas multinacionales de tan alto nivel estén utilizando estas tecnologías, es un claro indicativo de que el conocimiento del desarrollo basado en contenedores y su orquestación, van a ser fundamentales y cada vez más requeridos por las empresas a los futuros trabajadores que se dediquen a esta profesión.

Por último, el uso de tantas tecnologías distintas aplicadas para la consecución del objetivo principal de este Trabajo de Fin de Grado, han hecho que se trate de un proyecto multidisciplinar, teniendo en cuenta que se ha indagado en distintas áreas como bases de datos, APIs, desarrollo de front-end, virtualización mediante contenedores Docker y orquestación de estos últimos mediante plataformas de gestión y administración como Kubernetes.

4.2. TRABAJO FUTURO

Tras la finalización de este Trabajo de Fin de Grado, este proyecto y en concreto la aplicación podrían mejorarse, como creando nuevos servicios que se acoplen a los ya creados (por ejemplo, adherir un sistema de pago de las entradas).

También, se puede crear un sistema de perfiles de usuarios que les permita identificarse con las credenciales de acceso a los servicios de la Universidad y así cada usuario dispondría de su propia lista de entradas y llevar a cabo las distintas acciones que permite la aplicación.

Asimismo, como se ha llevado a cabo en el proyecto, se ha debido de crear ciertas imágenes docker usando Dockerfile y se han subido al repositorio Docker Hub para disponer en el repositorio de las versiones creadas de la aplicación. Con la tecnología Docker, se podría programar la automatización de la creación y despliegue de imágenes a partir de las actualizaciones del repositorio en lo que se conoce como Integración y Entrega continua.

Por último, el incremento de servicios en una aplicación hace que aumente la complejidad, así como el enrutamiento entre servicios, la tolerancia a fallos, la latencia, descubrimiento de servicios, seguridad, trazabilidad distribuida, y más tareas que hacen que entren en juego los

Service Mesh, infraestructuras de software dedicadas para manejar la comunicación entre microservicios y que será el camino a seguir en el Complemento del Trabajo de Fin de Grado.

5. BIBLIOGRAFÍA

[1] Microservicios: Docker y Kubernetes (Abril 2020) Disponible en: https://www.teldat.com/blog/es/microservicios-containers-dockers-kubernetes-para-sd-wan/

[2] De Docker a Kubernetes: entendiendo qué son los contenedores y por qué es una de las mayores revoluciones de la industria del desarrollo (Abril 2020) Disponible en: https://www.xataka.com/otros/docker-a-kubernetes-entendiendo-que-contenedores-que-mayores-revoluciones-industria-desarrollo#:~:text=Un%20contenedor%20de%20Docker%20puede,ocupar%20varios%20gigas%20de%20memoria.

[3] ¿Qué es Docker? (Abril 2020) Disponible en: https://www.redhat.com/es/topics/containers/what-is-docker

[4] Docker: Docs (Abril 2020) Disponible en: https://docs.docker.com/

[5] Kubernetes: What is Kubernetes (Mayo 2020) Disponible en: https://kubernetes.io/es/docs/concepts/overview/what-is-kubernetes/

[6] Phalcon Docs: Welcome (Abril 2020) Disponible en: https://docs.phalcon.io/4.0/es-es/introduction

[7] Phalcon Docs: Tutorial – Rest (Abril 2020) Disponible en: https://docs.phalcon.io/4.0/es-es/tutorial-rest

[8] Bootstrap Docs: Content – Reboot (Abril 2020) Disponible en: https://getbootstrap.com/docs/4.5/content/reboot/

[9] Desarrollo de Aplicaciones con Docker (Mayo 2020) Disponible en: https://ualmtorres.github.io/usoBasicoDeDocker/

[10] Kubernetes Docs: Concepts (Mayo 2020) Disponible en: https://kubernetes.io/es/docs/concepts/

[11] Uso de OpenStack como proveedor de infraestructura en Rancher (Junio 2020) Disponible en: http://ualmtorres.github.io/howtos/RancherOpenStack/

[12] Kubernetes: Horizontal Pod Autoscaler (Junio 2020) disponible en: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/