



UNIVERSIDAD DE BURGOS  
ESCUELA POLITÉCNICA SUPERIOR  
Grado en Ingeniería Informática



**TFG del Grado en Ingeniería  
Informática**

**título del TFG  
Documentación Técnica**



Presentado por nombre alumno  
en Universidad de Burgos — 18 de septiembre  
de 2023

Tutor: nombre tutor



---

# Índice general

---

<b>Índice general</b>	i
<b>Índice de figuras</b>	iii
<b>Índice de tablas</b>	iv
<b>Apéndice A Plan de Proyecto Software</b>	1
A.1. Introducción . . . . .	1
A.2. Planificación temporal . . . . .	1
A.3. Estudio de viabilidad . . . . .	1
<b>Apéndice B Especificación de Requisitos</b>	3
B.1. Introducción . . . . .	3
B.2. Objetivos generales . . . . .	3
B.3. Catalogo de requisitos . . . . .	3
B.4. Especificación de requisitos . . . . .	3
<b>Apéndice C Especificación de diseño</b>	5
C.1. Introducción . . . . .	5
C.2. Diseño de datos . . . . .	5
C.3. Diseño procedimental . . . . .	5
C.4. Diseño arquitectónico . . . . .	5
<b>Apéndice D Documentación técnica de programación</b>	7
D.1. Introducción . . . . .	7
D.2. Estructura de directorios . . . . .	7
D.3. Manual del programador . . . . .	7

D.4. Compilación, despliegue y ejecución del proyecto . . . . .	15
D.5. Pruebas del sistema . . . . .	21
<b>Apéndice E Documentación de usuario</b>	<b>23</b>
E.1. Introducción . . . . .	23
E.2. Requisitos de usuarios . . . . .	23
E.3. Instalación . . . . .	23
E.4. Manual del usuario . . . . .	23
<b>Bibliografía</b>	<b>25</b>

---

# Índice de figuras

---

D.1. Listado de todas las configuraciones de ejecución disponibles . . . . .	12
D.2. Selección del tipo de configuración . . . . .	13
D.3. Creación de la configuración del debugger remoto . . . . .	13
D.4. Creación de un módulo para un servicio . . . . .	14
D.5. Debug del servicio . . . . .	15
D.6. Página de descarga de Docker Desktop . . . . .	16
D.7. Instalación de Docker Desktop en Mac . . . . .	16
D.8. Ventana principal de Docker Desktop . . . . .	17
D.9. Creación de las imágenes de los contenedores Docker . . . . .	19
D.10. Despliegue de los contenedores Docker . . . . .	20
D.11. Ejecución multi-contenedor de la aplicación . . . . .	20
D.12. Logs de la ejecución de un contenedor en Docker Desktop . . . . .	21

# Índice de tablas

## *Apéndice A*

---

# **Plan de Proyecto Software**

---

### **A.1. Introducción**

### **A.2. Planificación temporal**

### **A.3. Estudio de viabilidad**

Viabilidad económica

Viabilidad legal



## *Apéndice B*

---

# **Especificación de Requisitos**

---

## **B.1. Introducción**

Una muestra de cómo podría ser una tabla de casos de uso:

## **B.2. Objetivos generales**

## **B.3. Catalogo de requisitos**

## **B.4. Especificación de requisitos**

<b>CU-1</b>	<b>Ejemplo de caso de uso</b>
<b>Versión</b>	1.0
<b>Autor</b>	Alumno
<b>Requisitos asociados</b>	RF-xx, RF-xx
<b>Descripción</b>	La descripción del CU
<b>Precondición</b>	Precondiciones (podría haber más de una)
<b>Acciones</b>	<ul style="list-style-type: none"> <li>1. Pasos del CU</li> <li>2. Pasos del CU (añadir tantos como sean necesarios)</li> </ul>
<b>Postcondición</b>	Postcondiciones (podría haber más de una)
<b>Excepciones</b>	Excepciones
<b>Importancia</b>	Alta o Media o Baja...

Tabla B.1: CU-1 Nombre del caso de uso.

*Apéndice C*

---

## **Especificación de diseño**

---

- C.1. Introducción
- C.2. Diseño de datos
- C.3. Diseño procedimental
- C.4. Diseño arquitectónico



## *Apéndice D*

---

# **Documentación técnica de programación**

---

## **D.1. Introducción**

## **D.2. Estructura de directorios**

### **Estructura de los archivos de Docker**

```
NutriMenu
├── docker-compose
│   ├── docker-compose.ci.yml
│   └── docker-compose.dev.yml
└── webapps
    ├── apirest
    │   ├── Dockerfile
    │   ├── run_docker_ci.sh
    │   └── run_docker_dev.sh
    └── frontend
        ├── .dockerignore
        └── Dockerfile
```

## **D.3. Manual del programador**

### **Dockerfiles**

### **API REST**

---

```
FROM eclipse-temurin:17
RUN apt update && apt -y upgrade
RUN apt install -y inotify-tools dos2unix
ENV HOME=/app
RUN mkdir -p $HOME
WORKDIR $HOME
```

---

A la hora de desplegar la parte de *backend* disponemos para ello de un **Dockerfile**, que va a ser el encargado de definir cómo se va a crear la imagen de Docker, y qué parámetros y dependencias va a necesitar. Es el equivalente al proceso al que todos estamos acostumbrados de crear una máquina virtual, instalar el sistema operativo escogido y configurar todas las dependencias y paquetes necesarios, solo que en este caso todo esto podemos ahorrárnoslo (tanto en esfuerzo como en tiempo, ya que el tiempo de despliegue de una imagen de Docker es de apenas un par de minutos en la mayoría de los casos) escribiendo en este archivo la configuración necesaria que necesitamos para nuestro caso de uso.

En este caso me he basado en otra imagen ya existente, **eclipse-temurin** ([Docker Hub](#)), ya que contiene tanto el JDK como el JRE de Java, además de una instalación mínima de **Linux Ubuntu** con todas las dependencias ya instaladas y configuradas. He especificado que quiero usar la etiqueta **17**, puesto que quiero usar Java 17 para este proyecto, ya que es la última versión LTS (*Long Term Support*) disponible en el momento [5]. Existen muchas implementaciones del JDK de Java [1], pero en este caso he usado Eclipse Temurin principalmente por su soporte a arquitecturas *ARMv7* y *ARM64/v8*, lo que permite poder ejecutar los contenedores en plataformas como Apple Silicon, o incluso en mini-computadores como una Raspberry Pi.

Aunque la imagen descarga automáticamente la última versión disponible de [Docker Hub](#), por si acaso cada vez que construyamos la imagen vamos a comprobar si hay actualizaciones disponibles para cualquier paquete del sistema operativo del que dispone la imagen. Además de esto, también se van a instalar como dependencias **inotify-tools**, una herramienta para monitorizar cambios en el código; y **dos2unix**, que se encarga de cambiar el tipo de final de línea de **CRLF** (Windows) a **LF** (Unix) [6]. Indicamos también que, dentro de la imagen, el directorio que vamos a usar para trabajar es **/app**, puesto que esto nos permite que cualquier comando que ejecutemos a continuación se realice dentro de ese directorio [3].

## Frontend

---

```
FROM node:alpine
ENV HOME=/app
RUN mkdir -p $HOME
WORKDIR $HOME
COPY package*.json ./
RUN npm install
EXPOSE 3000
CMD ["npm", "start"]
```

---

En la parte del *frontend* también disponemos de otro **Dockerfile**, en este caso con la imagen oficial de **Node**, ya que es el entorno de ejecución que hay debajo de React. He usado la etiqueta **alpine** ([Docker Hub](#)), ya que **Alpine** es una distribución de Linux orientada a ser lo más ligera posible, reduciendo el número de paquetes al mínimo y sustituyendo las herramientas GNU por **BusyBox** [7], un ejecutable que es capaz de emularlas.

Al igual que en el otro Dockerfile que acabamos de ver, he configurado el entorno de ejecución en el directorio `/app` del contenedor. A continuación he indicado que quiero que se copien los archivos que empiezan por `package` y acaban en `.json` al contenedor, ya que en un proyecto React son los archivos que indican todas las dependencias que necesita el proyecto para poder ejecutarse. Sobre el qué directorio va a usar Docker para coger estos archivos, lo veremos después en el archivo de configuración de **Docker Compose**, puesto que al construir la imagen con la etiqueta `build`: indicaremos el directorio al que puede acceder Docker durante la construcción de la imagen con el atributo `context`: [2].

Por último, la imagen va a instalar todas las dependencias del proyecto necesarias con `npm install`, se va a abrir el puerto `3000` del contenedor, ya que es el utilizado por React para poder acceder a la aplicación, y se va a iniciar la aplicación con el comando `npm start`.

## Docker Compose

### Backend

---

```
services:
  nutrimenumanage:
    build:
      context: ../webapps/nutrimenumanage
    restart: always
```

```

ports:
  - ${MANAGE_WEB_DOCKER_PORT}:${MANAGE_WEB_LOCAL_PORT}
  - ${MANAGE_RELOAD_DOCKER_PORT}:${MANAGE_RELOAD_LOCAL_PORT}
  - ${MANAGE_DEBUG_DOCKER_PORT}:${MANAGE_DEBUG_LOCAL_PORT}
networks:
  - nutrimenu-net
env_file:
  - .env
depends_on:
  - mysqlDb
working_dir: /app
command: sh run_docker.sh
volumes:
  - ./webapps/nutrimenumanage:/app
  - .m2:/root/.m2

```

---

Vamos a crear dos contenedores distintos, uno por cada parte del *backend* (parte de clientes y parte de gestión). Cada uno de ellos va a crear una imagen usando los *Dockerfiles* descritos en el anterior punto, y van a contener una serie de parámetros:

- Van a estar configurados para **reiniciarse de forma automática** en caso de fallo [4].
- Se van a **configurar los puertos** descritos en el apartado D.4.
- **Se van a conectar a una red** creada también en el fichero de Docker Compose, la cual veremos a continuación.
- Van a usar el archivo de **variables de entorno** descrito en el apartado D.4.
- **Dependen del contenedor de la base de datos al arrancar**, esto es que hasta que no arranque este contenedor, no lo van a hacer ellos, para evitar así problemas de conexión.
- Nada más arrancar **van a ejecutar el comando run\_docker.sh**. Este comando lo que hace es levantar el servicio de backend y estar a su vez a la espera de cambios en los ficheros del código fuente para, en caso de que suceda, recompilar el código de forma transparente.
- Se van a conectar dos volúmenes a cada contenedor: uno va a ser **el que contiene el código del servicio** (ya que si simplemente

copiáramos el código al contenedor, cada vez que hubiera cambios en el código habría que volver a desplegar los contenedores, y de esta forma podemos realizar cambios en tiempo real); y otro va a ser **el volumen encargado de almacenar las dependencias de Maven**, para no tener que descargarlas cada vez que ejecutemos el código.

## Base de datos

---

```
services:
  mysqlDb:
    image: "mysql:8.0"
    restart: always
    ports:
      - ${DB_DOCKER_PORT}:${DB_LOCAL_PORT}
    networks:
      - nutrimenu-net
    env_file:
      - .env
    volumes:
      - db_data:/var/lib/mysql
```

---

La base de datos también va a disponer de su propio contenedor, con ciertos parámetros similares a los de los servicios del *backend*. Podemos destacar de aquí algunos elementos:

- Vamos a usar la **imagen oficial de MySQL** ([Docker Hub](#)), concretamente la versión **8** (con los últimos parches).
- Se va a conectar un volumen a la base de datos, que no es más que **el volumen encargado de almacenar todos los datos** de la base de datos. Esto es para no tener que inicializar la base de datos cada vez que la despleguemos, y mantener así persistencia entre sesiones.

## Volúmenes

---

```
volumes:
  db_data:
```

---

Es el volumen creado para la base de datos, ya que a diferencia de los volúmenes creados para el *backend*, en los que realmente hemos conectado dos carpetas de nuestro equipo a los contenedores, este volumen se crea dentro de Docker.

## Redes

---

```
networks:
  nutrimenu-net:
```

---

Esta es la red creada para conectar todos los contenedores y que no existan problemas de conectividad entre ellos.

## Configuración del debugger remoto

Como veremos en el apartado D.4, se van a configurar distintos puertos por cada uno de los servicios, y uno de estos va a ser el puerto de debug. Como vamos a ejecutar la aplicación usando Docker Compose y no compilando el código directamente desde nuestro IDE, por defecto no podemos hacer debug como haríamos con cualquier código compilado localmente.

Sin embargo, con los pasos que vamos a ver a continuación [6], **vamos a ser capaces de conectarnos a nuestro entorno *contenerizado* y realizar debug del código como lo haríamos de forma normal**. En este caso voy a demostrar cómo funcionaría en IntelliJ IDEA, pero mientras que el IDE disponga de un *debugger remoto* y sigamos correctamente las instrucciones del desarrollador del IDE para usarlo, no debería de haber ningún problema.

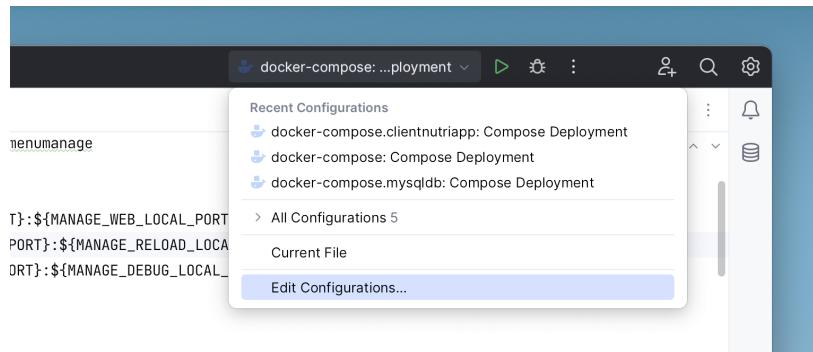


Figura D.1: Listado de todas las configuraciones de ejecución disponibles

Para comenzar, debemos de abrir el proyecto en IntelliJ y en la parte superior, al lado de los botones de Ejecución y Debug, tenemos un selector donde podemos seleccionar la configuración que deseemos. Debemos hacer click en *Edit Configurations....*

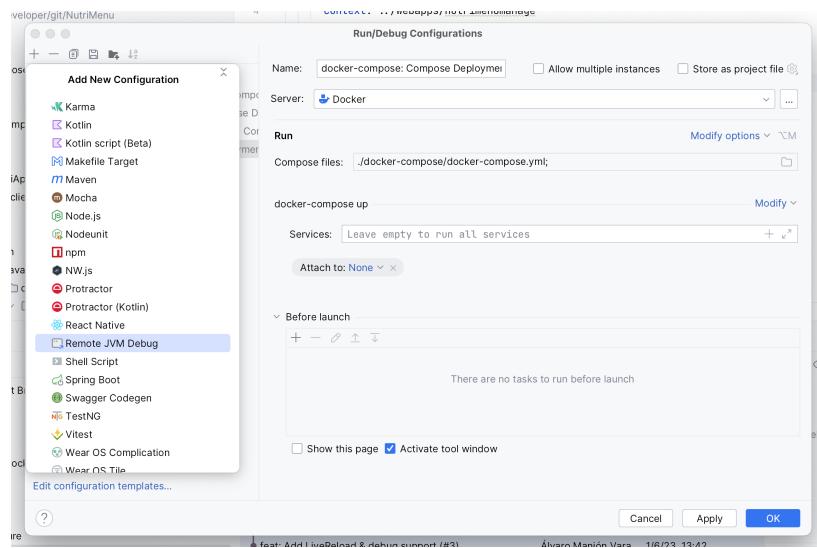


Figura D.2: Selección del tipo de configuración

Debemos hacer click en el icono con un + para añadir una nueva configuración, y seleccionar dentro del desplegable *Remote JVM Debug*.

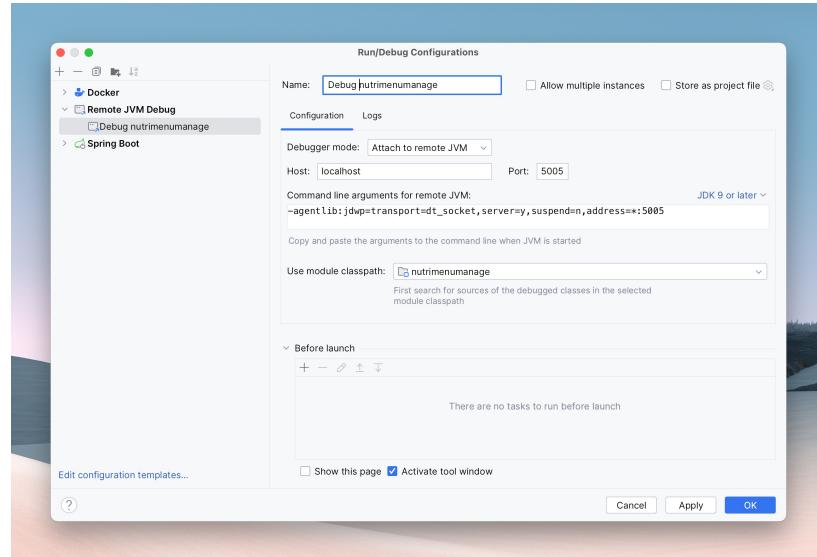


Figura D.3: Creación de la configuración del debugger remoto

A continuación daremos nombre a la configuración, indicaremos el puerto correspondiente al servicio que estemos configurando, y seleccionaremos el módulo de dicho servicio.

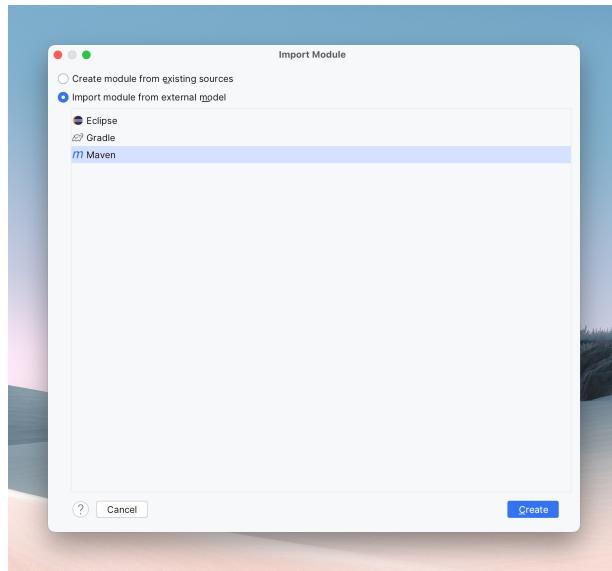


Figura D.4: Creación de un módulo para un servicio

En caso de que el módulo correspondiente no aparezca, deberemos seleccionar en la barra de herramientas *File > New > Module from Existing Sources...*, seleccionar el directorio en el que se encuentre la carpeta con el servicio correspondiente, y escoger Maven dentro de *Import module from external model*. Si sí que nos aparece el módulo podemos ignorar este paso.

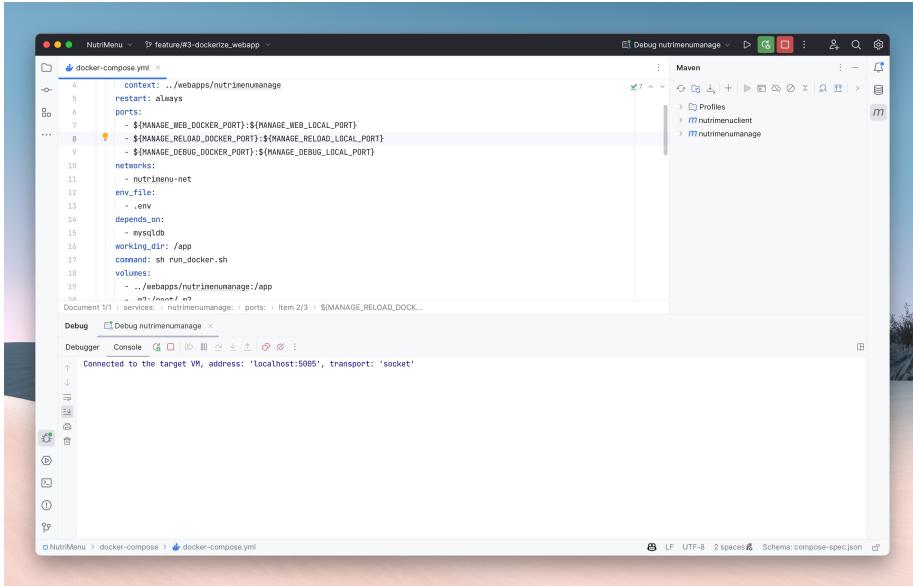


Figura D.5: Debug del servicio

Una vez hayamos completado estos pasos, podemos guardar la configuración y ejecutar el proyecto (asegurándonos de que la configuración deseada está seleccionada) desde el botón de Debug, como haríamos de normal. Si todo ha ido bien, el IDE nos devolverá en la consola el mensaje *Connected to the target VM, address: 'localhost:5005', transport: 'socket'*. Hay que tener en cuenta que estos pasos se deben repetir para cada uno de los servicios, seleccionando en cada caso el puerto y módulo indicados.

## D.4. Compilación, despliegue y ejecución del proyecto

Para realizar la compilación, despliegue y ejecución del proyecto es imprescindible disponer de **Docker CLI** y **Docker Compose** instalados en el equipo. La mejor forma de instalarlos es mediante **Docker Desktop**, ya que se encarga de instalar en el equipo el *daemon* de Docker, Docker CLI, Docker Compose, y demás dependencias y herramientas que no vamos a utilizar para este proyecto, pero que nos pueden ayudar en el futuro. Docker Desktop es compatible con Windows, macOS y Linux, y soporta tanto arquitecturas x86 como ARM64 (Apple Silicon).

## Instalación de Docker Desktop

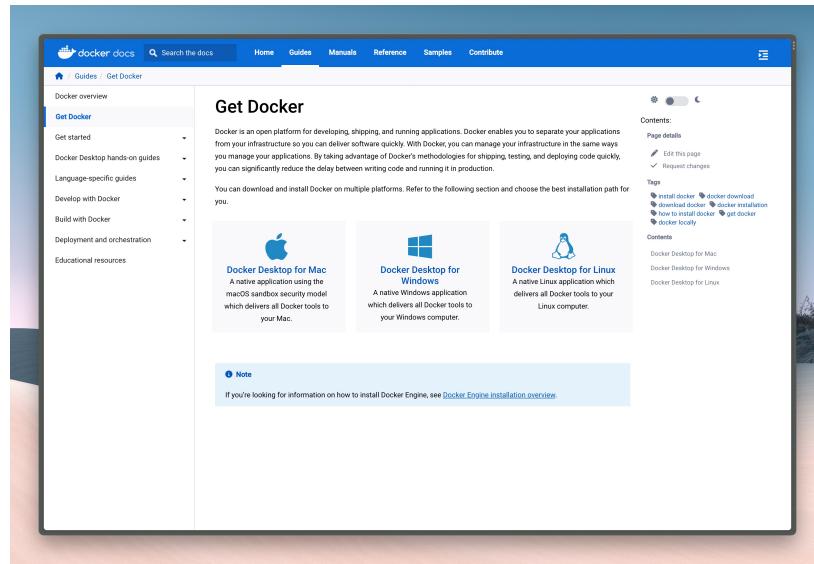


Figura D.6: Página de descarga de Docker Desktop

Para realizar la instalación de Docker Desktop tan sólo debemos dirigirnos a <https://docs.docker.com/get-docker/>, seleccionar la plataforma deseada, y descargar el instalador.

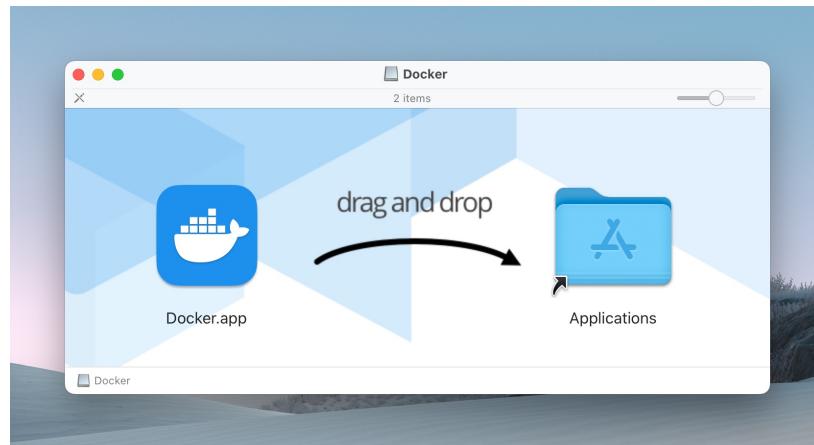


Figura D.7: Instalación de Docker Desktop en Mac

Una vez ejecutemos el instalador (en caso de macOS simplemente se debe arrastrar la aplicación a la carpeta de Aplicaciones), y hayamos seguido

todos los pasos hasta finalizar la instalación, nos encontraremos con el panel principal de Docker Desktop.

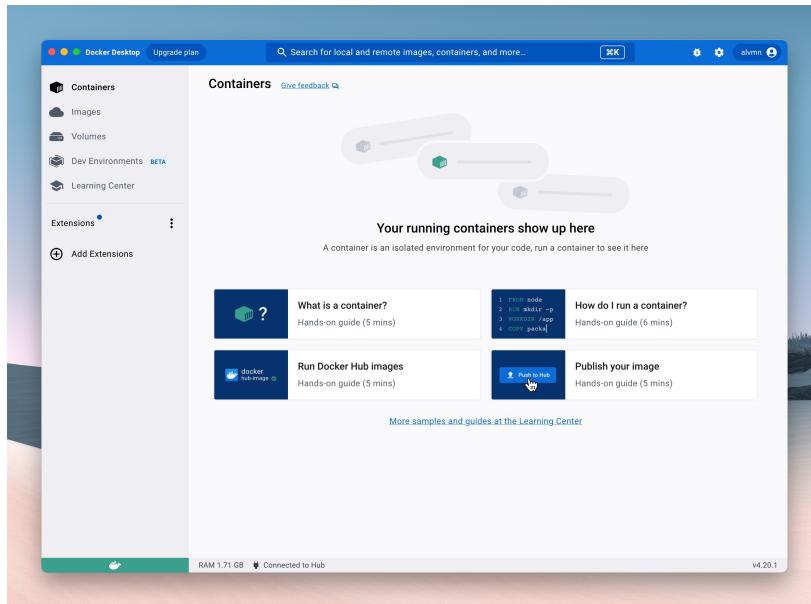


Figura D.8: Ventana principal de Docker Desktop

En esta ventana podemos ver todos los contenedores que están ejecutándose actualmente en el sistema, así como las imágenes descargadas y los volúmenes creados.

## Preparación del entorno de desarrollo

Para hacer más sencillo el cambio de valores en distintos parámetros de la aplicación, como el mapeado de puertos o la gestión de secretos, y facilitar así el despliegue de nuevos entornos o la escalabilidad de los servicios, estos valores se han externalizado en un sólo archivo `.env`, que va a ser un archivo que simplemente va a contener las variables junto a sus valores.

Al ser un fichero que contiene secretos, siguiendo lo que dictan las buenas prácticas no se sincronizará con el repositorio, ya que si se tratase de un entorno de producción o el código se hiciera público, cualquier usuario podría obtener acceso a los datos de los usuarios de la aplicación.

Por lo tanto, antes de desplegar los contenedores, para que la aplicación funcione correctamente el archivo se ha de crear dentro del directorio `/docker-compose`, y se deben de definir las siguientes variables dentro de él:

---

```

# Variables del backend de la parte de los clientes
CLIENT_WEB_DOCKER_PORT=8081
CLIENT_WEB_LOCAL_PORT=8081
CLIENT_RELOAD_DOCKER_PORT=35730
CLIENT_RELOAD_LOCAL_PORT=35730
CLIENT_DEBUG_DOCKER_PORT=5006
CLIENT_DEBUG_LOCAL_PORT=5006

# Variables del backend de la parte de gestión
MANAGE_WEB_DOCKER_PORT=8080
MANAGE_WEB_LOCAL_PORT=8080
MANAGE_RELOAD_DOCKER_PORT=35729
MANAGE_RELOAD_LOCAL_PORT=35729
MANAGE_DEBUG_DOCKER_PORT=5005
MANAGE_DEBUG_LOCAL_PORT=5005

# Variables de la base de datos
DATABASE_HOST=mysql ldb
MYSQL_DATABASE=nutri_db
MYSQL_USER=<MySQL user> # Introduce el nombre del usuario de MySQL
MYSQL_PASSWORD=<MySQL password> # Introduce la pass escogida para
    el usuario creado en el paso anterior
MYSQL_ROOT_PASSWORD=<MySQL root password> # Introduce la pass
    escogida para el usuario root de MySQL
DB_DOCKER_PORT=3306
DB_LOCAL_PORT=3306

```

---

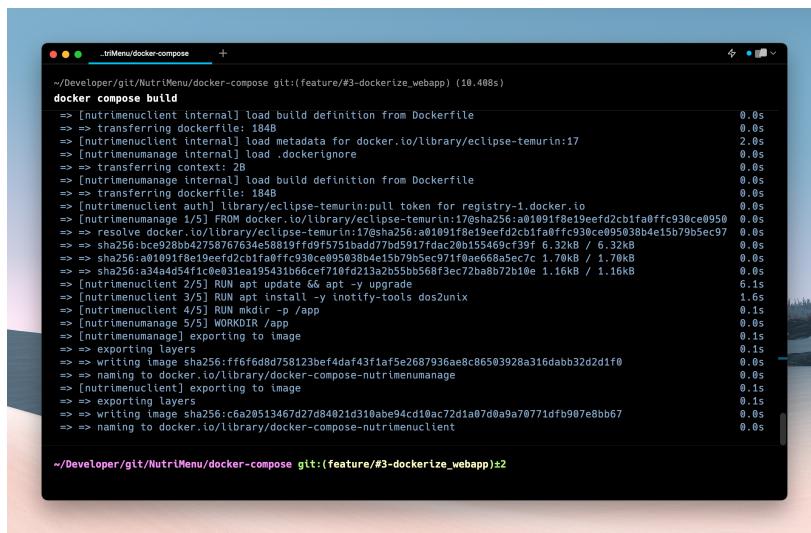
Los valores pueden ser modificados si es necesario, pero se recomienda usar los valores dados (indicando los valores escogidos en `MYSQL_USER`, `MYSQL_PASSWORD` y `MYSQL_ROOT_PASSWORD`). Con estos valores lo que estamos indicando es:

- Los puertos en los que se van a ejecutar cada una de las aplicaciones, que en este caso van a ser el **8080** para la aplicación de gestión y el **8081** para la aplicación de los clientes.
- Los puertos en los que va a estar escuchando el plugin *LiveReload* de Spring, que es el encargado de recargar el servidor en caliente cada vez que se produzcan cambios en el código, y no tener así que recompilar todo el código cada vez que cambiemos algo. En este caso los puertos

van a ser el **35729** para la aplicación de gestión y el **35730** para la aplicación de los clientes.

- Los puertos que se van a usar para permitir conectar un debugger remoto a la aplicación, y poder así hacer debug desde el IDE que usemos para el desarrollo. Los puertos serán el **5005** para la aplicación de gestión y el **5006** para la aplicación de los clientes.
- Todas las variables correspondientes a la base de datos, como el **nombre de la instancia**, **nombre de la base de datos**, **credenciales de los usuarios de MySQL**, y el puerto escogido para la base de datos, que en este caso va a ser el usado por defecto, **3306**.

## Despliegue de la aplicación



```
~/Developer/git/NutriMenu/docker-compose git:(feature/#3-dockerize_webapp) (10.408s)
docker compose build
--> [nutrimentclient] load build definition from Dockerfile          0.0s
--> => transferring dockerfile: 184B                                0.0s
--> [nutrimentclient] load metadata from docker.io/library/eclipse-temurin:17 2.0s
--> [nutrimentmanage] load .dockerrcignore                           0.0s
--> => transferring context: 29                                     0.0s
--> [nutrimentmanage] load build definition from Dockerfile          0.0s
--> => transferring dockerfile: 184B                                0.0s
--> [nutrimentclient] pull token for registry-1.docker.io           0.0s
--> [nutrimentmanage] 1/51 FROM docker.io/library/eclipse-temurin:17@sha256:a01091f8e19eef2cb1fa0ffc930ce095038b4e15b79b5ec97 0.0s
--> => sha256:bce920ba4275876753ae58819ff9f5751bad77bd5917fdac20b155469cf39f 6..32kB / 6..32kB 0.0s
--> => sha256:a01091f8e19eef2cb1fa0fc930ce095038b4e15b79b5ec971fa0e668a5ec7c 1..70kB / 1..70kB 0.0s
--> => sha256:a34a4d54f1cbe031ea195431b66cef710f4213ab2b55bb568f3ec72ba8b72b10e 1..16kB / 1..16kB 0.0s
--> [nutrimentclient] 2/51 RUN apt update && apt -y upgrade          6.1s
--> [nutrimentclient] 3/51 RUN apt install -y inotify-tools dos2unix 1.6s
--> [nutrimentclient] 4/51 RUN mkdir -p /app                         0.1s
--> [nutrimentclient] 5/51 WORKDIR /app                            0.0s
--> [nutrimentmanage] exporting to image                           0.1s
--> => exporting layers                                         0.1s
--> => writing image sha256:fff6fd8d758123bdef4daf43f1af5e2687936ae8c86503928a316dabb32d2d1f0 0.0s
--> => naming to docker.io/library/docker-compose-nutrimentmanage 0.0s
--> [nutrimentclient] exporting to image                           0.1s
--> => exporting layers                                         0.1s
--> => writing image sha256:c6a20513467d27db84021d310abe94cd10ac72d1a07d0a9a70771dfb997e8bb67 0.0s
--> => naming to docker.io/library/docker-compose-nutrimentclient 0.0s

~/Developer/git/NutriMenu/docker-compose git:(feature/#3-dockerize_webapp)±2
```

Figura D.9: Creación de las imágenes de los contenedores Docker

Una vez definidas las variables de entorno, podemos probar que todo funciona correctamente construyendo las imágenes de los contenedores de los servicios. Para ello, abriremos una consola y accederemos al directorio `/docker-compose`, donde ejecutaremos el siguiente comando:

---

```
docker-compose build
```

---

```

~/Developer/git/NutriMenu/docker-compose git:(feature/#3-dockerize_webapp) (10.408s)
docker compose build

~/Developer/git/NutriMenu/docker-compose git:(feature/#3-dockerize_webapp) (20.296s)
docker-compose up -d

[+] Running 12/12
  ✓ mysql 11 layers [██████████]
    ✓ 2ae143351a4f Pull complete
    ✓ 093b9ff06d48 Pull complete
    ✓ 1cf4f76b0363a Pull complete
    ✓ a8e5c746438f Pull complete
    ✓ 3b8559256f40 Pull complete
    ✓ 5b3ed925ef40 Pull complete
    ✓ d1388739c015 Pull complete
    ✓ 94a7daee25e1 Pull complete
    ✓ 17b0d0608032 Pull complete
    ✓ b4191a53e3e86 Pull complete
    ✓ 0d36888be7eb Pull complete

[+] Building 0.0s (0/0)
[+] Running 5/5
  ✓ Network docker-compose_nutrimenu-net      Created
  ✓ Volume "docker-compose_db_data"            Created
  ✓ Container docker-compose-mysqldb-1        Started
  ✓ Container docker-compose-nutrimenudb-1     Started
  ✓ Container docker-compose-nutrimenumanage-1 Started

~/Developer/git/NutriMenu/docker-compose git:(feature/#3-dockerize_webapp)±3
  
```

Figura D.10: Despliegue de los contenedores Docker

Para desplegar los contenedores y levantar el servicio, ejecutaremos el siguiente comando:

---

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

---

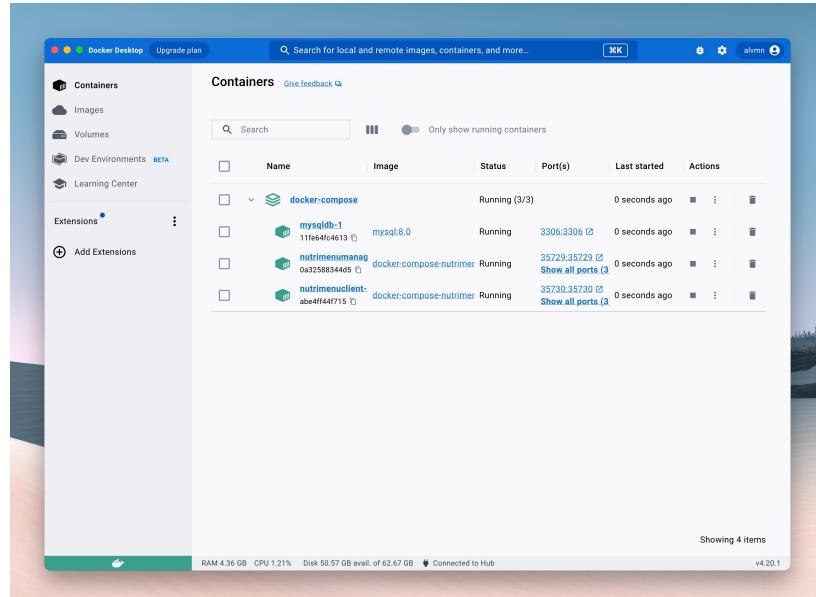


Figura D.11: Ejecución multi-contenedor de la aplicación

Si todo ha ido bien, podremos ver en Docker Desktop los contenedores corriendo con estado *Running*, y ya podremos acceder a la aplicación como haríamos con cualquier tipo de despliegue.

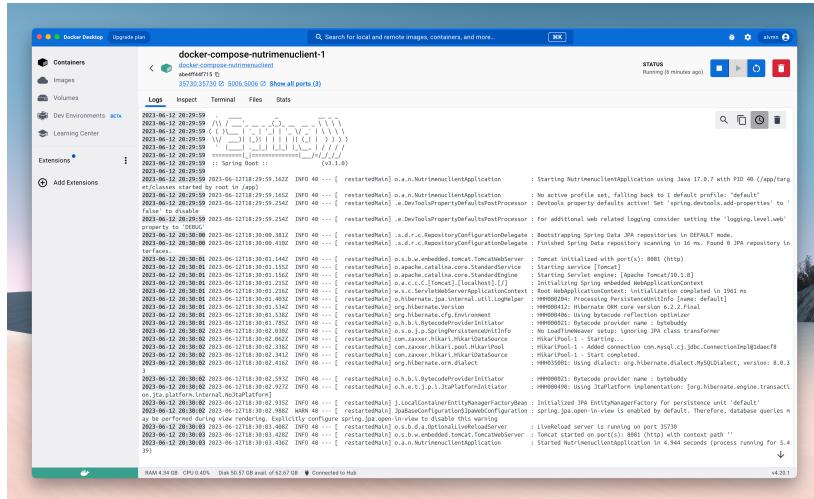


Figura D.12: Logs de la ejecución de un contenedor en Docker Desktop

Para asegurarnos de que no hay ningún tipo de problema, podemos seleccionar los 3 puntos verticales situados a la derecha de cada contenedor, y hacer click en *View details*, ya que esto nos permite poder ver el log de cada servicio en tiempo real.

## D.5. Pruebas del sistema



*Apéndice E*

---

## **Documentación de usuario**

---

- E.1. Introducción**
- E.2. Requisitos de usuarios**
- E.3. Instalación**
- E.4. Manual del usuario**



---

# Bibliografía

---

- [1] Erik Costlow and Simon Ritter. Foojay Podcast no. 4: Why So Many JDKs? <https://foojay.io/today/foojay-podcast-4/>, Oct 2021. [Internet; descargado 14-junio-2023].
- [2] Docker Docs. Build context. <https://docs.docker.com/build/building/context/>, 2023. [Internet; descargado 17-septiembre-2023].
- [3] Docker Docs. Dockerfile reference - WORKDIR. <https://docs.docker.com/engine/reference/builder/#workdir>, 2023. [Internet; descargado 17-septiembre-2023].
- [4] Christian Jaimes. Docker Compose Restart Policies. <https://www.baeldung.com/ops/docker-compose-restart-policies>, Oct 2022. [Internet; descargado 14-junio-2023].
- [5] Java. JDK Releases. <https://www.java.com/releases/>, 2023. [Internet; descargado 14-junio-2023].
- [6] Vibhor Mahajan. Developing Spring Boot Applications in Docker locally. <https://medium.com/trantor-inc/developing-spring-boot-applications-in-docker-locally-4ec922f4cb45>, Sep 2021. [Internet; descargado 13-junio-2023].
- [7] Wikipedia. BusyBox. <https://en.wikipedia.org/wiki/BusyBox>, 2023. [Internet; descargado 17-septiembre-2023].