



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 — 14 de junio
de 2023

Tutor: nombre tutor

Índice general

Índice general	i
Índice de figuras	iii
Índice de tablas	iv
Apéndice A Plan de Proyecto Software	1
A.1. Introducción	1
A.2. Planificación temporal	1
A.3. Estudio de viabilidad	1
Apéndice B Especificación de Requisitos	3
B.1. Introducción	3
B.2. Objetivos generales	3
B.3. Catalogo de requisitos	3
B.4. Especificación de requisitos	3
Apéndice C Especificación de diseño	5
C.1. Introducción	5
C.2. Diseño de datos	5
C.3. Diseño procedimental	5
C.4. Diseño arquitectónico	5
Apéndice D Documentación técnica de programación	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	14
D.5. Pruebas del sistema	20
Apéndice E Documentación de usuario	21
E.1. Introducción	21
E.2. Requisitos de usuarios	21
E.3. Instalación	21
E.4. Manual del usuario	21
Bibliografía	23

Índice de figuras

D.1. Listado de todas las configuraciones de ejecución disponibles	11
D.2. Selección del tipo de configuración	12
D.3. Creación de la configuración del debugger remoto	12
D.4. Creación de un módulo para un servicio	13
D.5. Debug del servicio	14
D.6. Página de descarga de Docker Desktop	15
D.7. Instalación de Docker Desktop en Mac	15
D.8. Ventana principal de Docker Desktop	16
D.9. Creación de las imágenes de los contenedores Docker	18
D.10. Despliegue de los contenedores Docker	19
D.11. Ejecución multi-contenedor de la aplicación	19
D.12. Logs de la ejecución de un contenedor en Docker Desktop	20

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

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

```
NutriMenu
├── docker-compose
│   └── docker-compose.yml
└── webapps
    ├── nutrimenuclient
    │   ├── Dockerfile
    │   └── run-docker.sh
    └── nutrimentumanage
        ├── Dockerfile
        └── run-docker.sh
```

D.3. Manual del programador

Dockerfiles

```
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
```

Cada uno de los servicios del *backend* dispone de un **Dockerfile**, que a la hora del despliegue es el encargado de definir cómo se va a crear la imagen, y qué parámetros y dependencias va a necesitar.

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 todos los binarios relacionados. 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 [3]. Existen muchas implementaciones del JDK de Java [1], pero en este caso he usado Eclipse Temurin principalmente por el 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.

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) [4].

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 [2].
- 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 [4], **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.

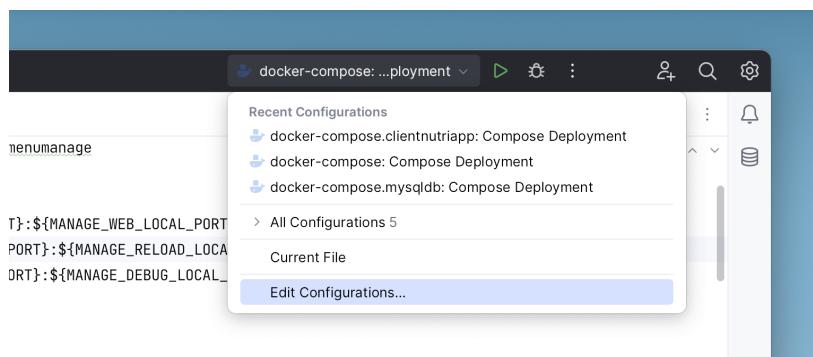


Figura D.1: Listado de todas las configuraciones de ejecución disponibles

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...*

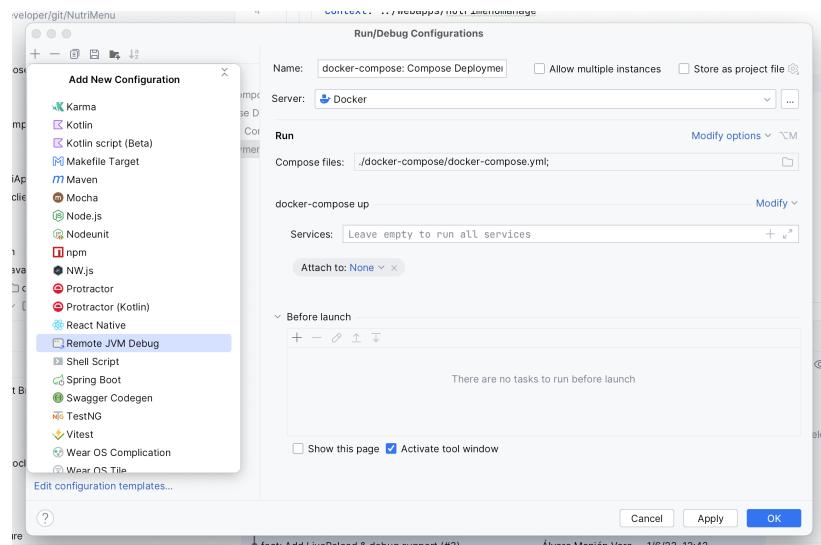


Figura D.2: Selección del tipo de configuración

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

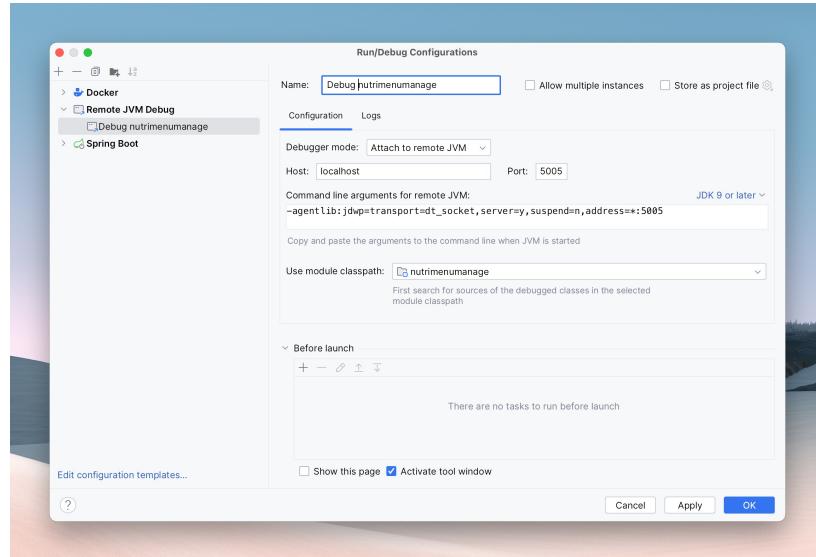


Figura D.3: Creación de la configuración del debugger remoto

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

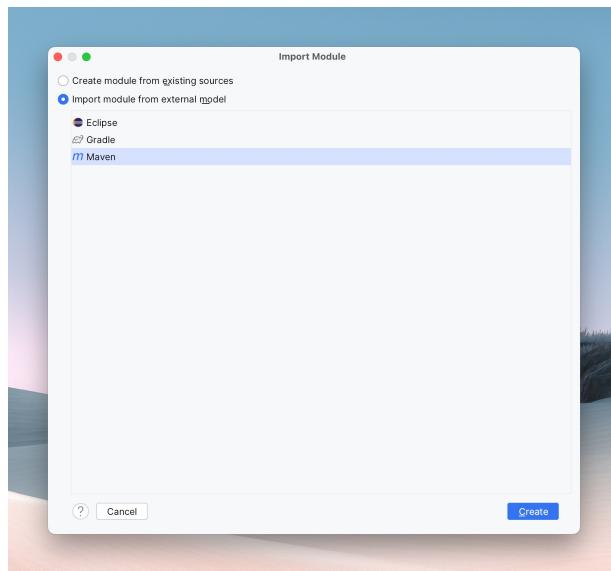


Figura D.4: Creación de un módulo para un servicio

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.

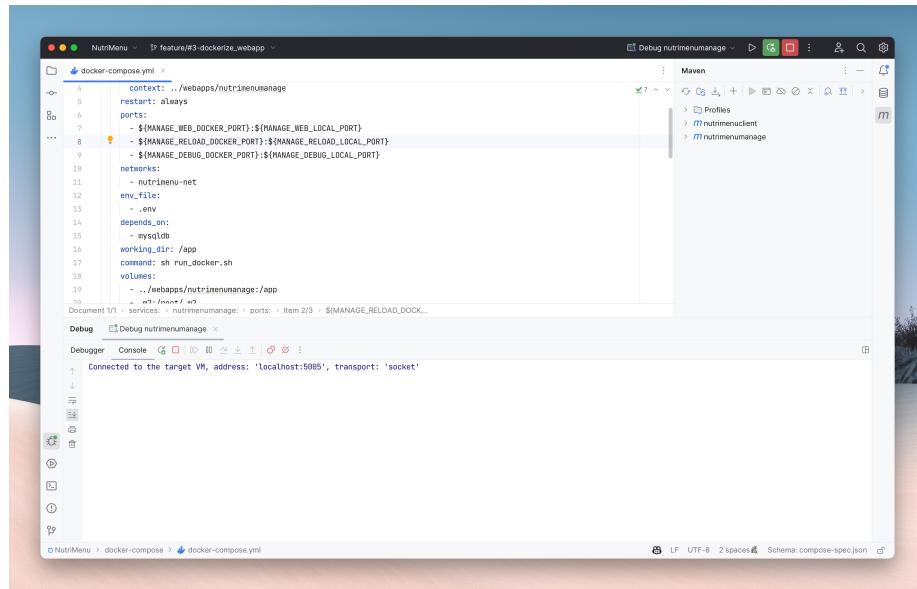


Figura D.5: Debug del servicio

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

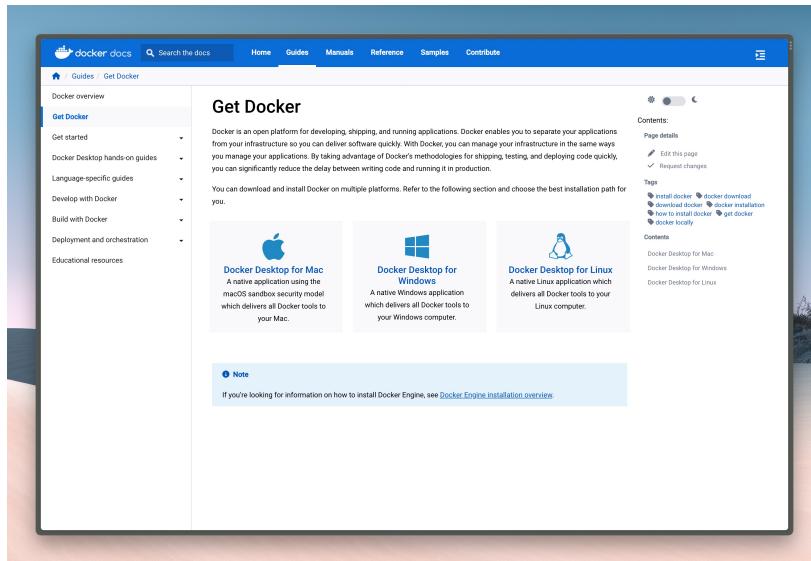


Figura D.6: Página de descarga de Docker Desktop

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.

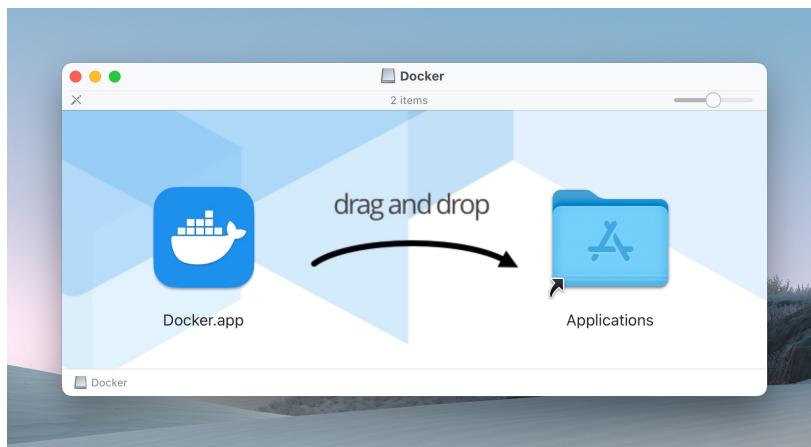


Figura D.7: Instalación de Docker Desktop en Mac

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.

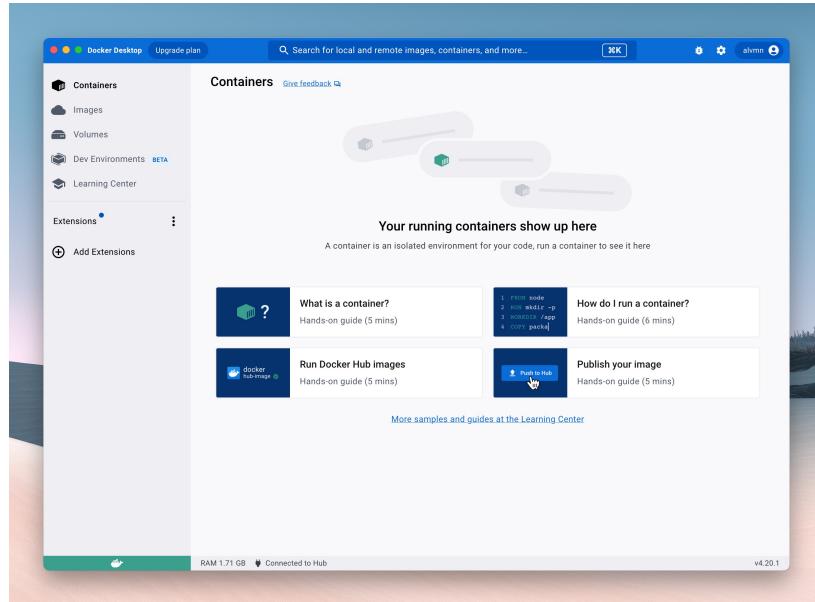


Figura D.8: Ventana principal de Docker Desktop

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=mysqlDb
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
```

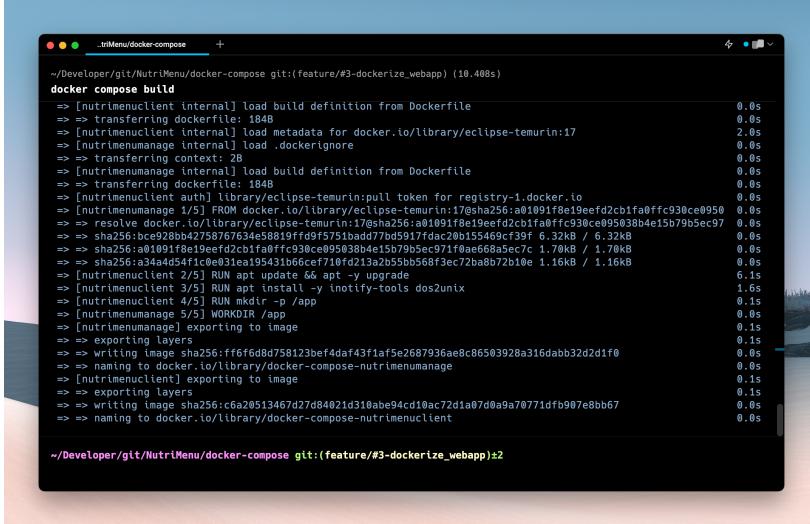
Los valores pueden ser modificados si es necesario, pero se recomienda usar los valores dados (indicando los valores escogidos en `MYSQL_USER`, `MYSQL_PASSWORD` y `MYSQL_ROOT_PASSWORD`). Con estos valores lo que estamos indicando es:

- 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
--> [nutrimenuclient internal] load build definition from Dockerfile          0.0s
--> => transferring dockerfile: 184B                                         0.0s
--> [nutrimenuclient internal] load metadata for docker.io/library/eclipse-temurin:17 2.0s
--> [nutrimenumanage internal] load .dockignore                           0.0s
--> => transferring file: 2B                                              0.0s
--> [nutrimenumanage internal] load build definition from Dockerfile          0.0s
--> => transferring dockerfile: 184B                                         0.0s
--> => resolve docker.io/library/eclipse-temurin:pull token for registry-1.docker.io 0.0s
--> [nutrimenuclient 1/5] FROM docker.io/library/eclipse-temurin:17@sha256:a01091f8e19eefd2cb1fa0ffc930ce095038b4e15b79b5ec97 0.0s
--> => sha256:bcc928bb42758767634e58819ffdf9751badd77bd5917dac20b155469cf39f 6.32kB / 6.32kB 0.0s
--> => sha256:a01091f8e19eefd2cb1fa0ffc930ce095038b4e15b79b5ec971f0ae668a5ec7c 1.70kB / 1.70kB 0.0s
--> => sha256:a3a44d54f1c0e831ea195431b66ccf710fd213a2b55bb568f3ec72ba0b72b10e 1.16kB / 1.16kB 0.0s
--> [nutrimenuclient 2/5] RUN apt update && apt -y upgrade                   6.1s
--> [nutrimenuclient 3/5] RUN apt install -y inotify-tools dos2unix           1.6s
--> [nutrimenuclient 4/5] RUN mkdir -p /app                                     0.1s
--> [nutrimenumanage 5/5] WORKDIR /app                                      0.0s
--> [nutrimenumanage] exporting to image                                     0.1s
--> => exporting layers                                                 0.1s
--> => writing image sha256:c6a20513467d27d84021d310be94cd10ac72d1a09a70771dfb907e8bb67 0.0s
--> => naming to docker.io/library/docker-compose-nutrimenumanage            0.0s
--> [nutrimenuclient] exporting to image                                     0.1s
--> => exporting layers                                                 0.1s
--> => writing image sha256:c6a20513467d27d84021d310be94cd10ac72d1a09a70771dfb907e8bb67 0.0s
--> => naming to docker.io/library/docker-compose-nutrimenuclient           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

[+] Running 12/12
✓ mysqlDb 11 layers [██████████]    0B/0B      Pulled          19.25
✓ 2ae143351a4f Pull complete
✓ 093b9ff96d48 Pull complete
✓ 1cf4f7bd933a Pull complete
✓ a8614746438f Pull complete
✓ 38803a0a201e Pull complete
✓ 5b3e9326fe81 Pull complete
✓ d1388739c015 Pull complete
✓ 94a7da0a25e1 Pull complete
✓ 17b0d0608a32 Pull complete
✓ b4191a53e86e Pull complete
✓ 0d368800e7eb 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-nutrimenuclient-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
```

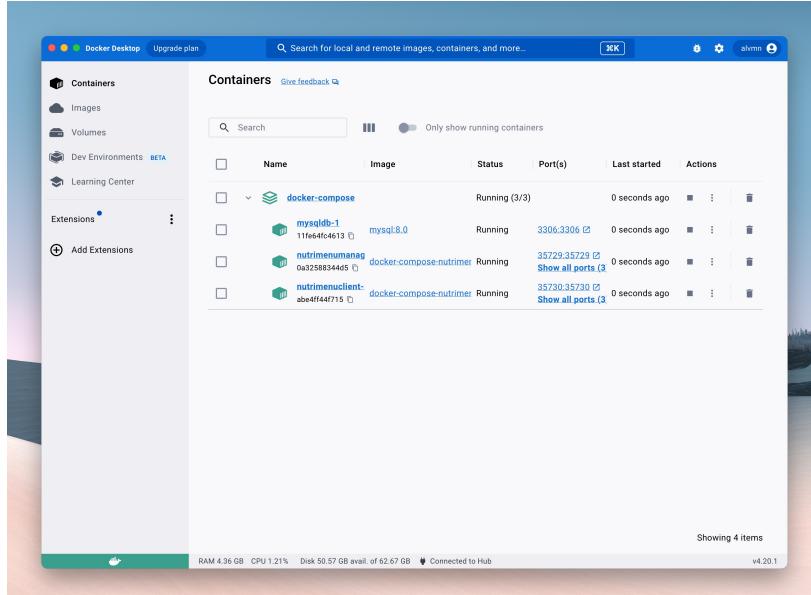


Figura D.11: Ejecución multi-contenedor de la aplicación

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.

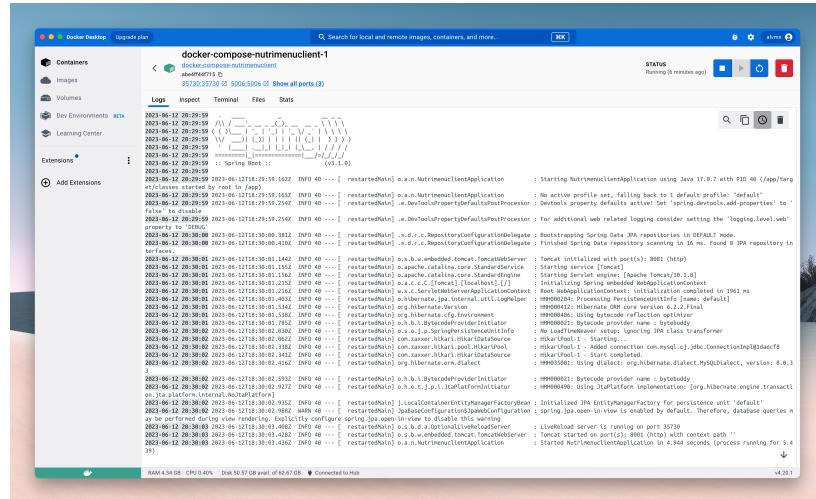


Figura D.12: Logs de la ejecución de un contenedor en Docker Desktop

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] Christian Jaimes. Docker Compose Restart Policies. <https://www.baeldung.com/ops/docker-compose-restart-policies>, Oct 2022. [Internet; descargado 14-junio-2023].
- [3] Java. JDK Releases. <https://www.java.com/releases/>, 2023. [Internet; descargado 14-junio-2023].
- [4] 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].