

# VentBook

Una aplicacion para  
gestionar eventos y reservas

Autor

Rubén Rodríguez Esteban

11/08/2025

# Índice

<b>1. Propuesta de VentBook</b>	<b>V</b>
1.1. Análisis tecnológico . . . . .	V
1.1.1. Despliegue de la APP . . . . .	V
1.1.2. Control de virtualización . . . . .	VI
1.1.3. Sistema operativo virtual . . . . .	VII
1.1.4. Herramientas de desarrollo . . . . .	VII
<b>2. Primeros pasos de VentBook</b>	<b>IX</b>
2.1. Instalación de VirtualBox . . . . .	IX
2.2. Gestión de VirtualBox Guest Additions . . . . .	XIII
2.3. Adición de software de desarrollo . . . . .	XV
2.4. Bienvenida a Dokcer y Docker Compose . . . . .	XVI
<b>3. VentBook: un software modular</b>	<b>XVII</b>
3.1. Capa arquitectural . . . . .	XVII
3.1.1. Fichero Makefile . . . . .	XVIII
3.1.2. Fichero docker-compose.yml . . . . .	XIX
3.2. Capa de aplicación . . . . .	XXII
3.2.1. Puesta punto del servidor . . . . .	XXII
3.2.2. Estructura de carpetas . . . . .	XXIII
3.3. Capa de infraestructura . . . . .	XXV
3.3.1. Entidades . . . . .	XXV
3.3.2. Data Transfer Objects . . . . .	XXVII
3.3.3. Ensambladores . . . . .	XXIX
3.3.4. Servicios . . . . .	XXX
3.3.5. Controladores . . . . .	XXXI
3.4. Capa de persistencia . . . . .	XXXIII
3.4.1. Configuración de la BD MySQL 8.0.23 . . . . .	XXXIII
3.4.2. Configuración de Redis 7.0 . . . . .	XXXVIII

<b>4. Documentación de EventBook</b>	<b>XLII</b>
4.1. Código PHP . . . . .	XLII
4.2. Ficheros SQL . . . . .	XLII
4.3. Lógica de negocio . . . . .	XLIII
4.4. API interactiva con Swagger UI . . . . .	XLIII
<b>5. Testing de EventBook</b>	<b>XLV</b>
<b>6. Problemas encontrados</b>	<b>XLVII</b>
6.1. Incompatibilidad entre PHP y Symfony . . . . .	XLVII
6.2. Configuración de Swagger UI . . . . .	XLVII
6.3. Realización de tests automatizados . . . . .	XLVIII
6.4. Comunicación con la base de datos MySQL 8.0.26 . . . . .	XLVIII

# Contexto

El objetivo del proyecto es el desarrollo de una API REST implementada en lenguaje PHP para la gestión de eventos y reservas. La herramienta debe diseñarse teniendo en cuenta que debe ser mantenible, estructurada y bien organizada. Además, de cara a la implementación será necesario prestar especial atención a las buenas prácticas de programación, a los principios SOLID y DRY, así como hacer uso de una arquitectura hexagonal desacoplada con diseño por capas.

## Eventos

En lo que al ámbito funcional se refiere, la API debe incluir endpoints para gestionar eventos, incluyendo la creación, visualización y listado de eventos, con la opción de aplicar filtros específicos. Adicionalmente, deberá permitir la creación y cancelación de reservas para eventos, así como el listado de reservas basado en la información del solicitante.

A continuación, se puede visualizar qué información se espera que devuelva la API acerca de un evento concreto:

```
{  
    "id": "unique event reference (integer)",  
    "name": "event name (string)",  
    "description": "event description (string)",  
    "from": "initial event date (string formato yyyy-mm-dd)",  
    "to": "final event date (string formato yyyy-mm-dd)",  
    "available": "available seats (integer)"  
}
```

Para el manejo y gestión de eventos se deben tener en cuenta las siguientes consideraciones técnicas:

- **Eventos diarios y rango de fechas:** se permite una única reserva por evento diario dentro del rango de fechas especificado. Por ejemplo, si el evento ofrece un rango desde 2030-01-01 hasta 2030-01-15, se podrá reservar una fecha concreta dentro de las fechas indicadas. Ambas fechas del rango están incluidas.

- **Filtro en la búsqueda de eventos:** la búsqueda de eventos se puede filtrar por nombre, fecha de inicio, fecha de fin y número de plazas disponibles. El filtro por plazas disponibles excluye eventos con menos plazas que las requeridas.
- **Listado y detalles de eventos:** el listado de eventos excluye la descripción del evento. La descripción solo se muestra en el endpoint del detalle.

## Reservas

Seguidamente, se muestra qué información se espera que la API responda sobre una reserva en concreto:

```
{
  "reference": "unique booking reference (string)",
  "eventId": "related event reference (integer)",
  "eventDate": "event date selected (string formato yyyy-mm-dd)",
  "attendees": "number of people included (string)",
  "identification": "buyer's personal id (string formato DNI)"
}
```

Para el manejo y gestión de eventos se deben tener en cuenta las siguientes consideraciones técnicas:

- **Restricción de reservas por DNI:** un comprador no puede realizar más de una reserva para un mismo día, independientemente del evento.
- **Actualización de asientos disponibles:**
  - Una reserva disminuye el número de asientos disponibles para ese evento.
  - Una cancelación restaura los asientos reservados al evento.

## Endpoints

Por último, se muestran dos tablas que detallan los endpoints que la API debe permitir realizar para la gestión de eventos y de reservas.

Ruta	Tipo	Desscripción
/events	POST	Crea un nuevo evento
/events	GET	Listar eventos existentes con filtros opcionales
/events{id}	GET	Obtener los detalles de un evento concreto

Tabla 1: Endpoints para la gestión de eventos

Ruta	Tipo	Descripción
/bookings	POST	Generar una reserva
/bookings	GET	Listar las reservas por el DNI del comprador
/bookings{id}/cancel	GET	Cancelar una reserva

Tabla 2: Endpoints para la gestión de reservas

## Restricciones en el uso de ORMs y frameworks

- **Evitar el uso de ORMs:** se prohíbe expresamente el uso de *Object-Relational Mappers (ORMs)* para la interacción con la base de datos. Las consultas deben realizarse directamente en SQL explícito para mantener la transparencia y el control sobre las operaciones.
- **Limitar el uso de frameworks:** se permite el uso de frameworks únicamente para:
  - **Enrutamiento:** definición y gestión de rutas (endpoints) de la API, facilitando la asignación de URLs a controladores o acciones.
  - **Inyección de dependencias:** desacoplar componentes y gestionar dependencias de forma eficiente.
- **Implementación personalizada:** toda funcionalidad fuera de enrutamiento o inyección de dependencias, acceso a base de datos, lógica de negocio, creación de objetos de dominio, etcétera, debe implementarse sin asistencia de frameworks, asegurando control y comprensión del código.

## Otras directrices

- **Validación de datos:** implementar validaciones rigurosas de todos los datos recibidos por la API (tipo, formato, longitud, restricciones adicionales) para garantizar integridad y seguridad.

- **Objetos inmutables:** usar objetos inmutables siempre que sea posible para evitar efectos secundarios y facilitar la concurrencia.

## Entorno de desarrollo y tests

- Docker Compose: uso de Docker Compose para definir y gestionar servicios como base de datos, servidor web y otros componentes, garantizando consistencia entre entornos.
- **Imágenes de Docker:** se permite el uso de imágenes de Docker para PHP, Apache, Nginx, Redis u otros necesarios para la API.
- **Tests:** la aplicación debe incluir tests para garantizar calidad, detectar errores y facilitar mantenimiento.

# Capítulo 1

## Propuesta de VentBook

La elaboración de la prueba técnica ha sido llevada a cabo a través de un enfoque mucho más didáctico y orientado a un ámbito de negocio. Se ha procedido a elaborar el diseño de una API para una APP ficticia, llamada VentBook, que se va a caracterizar por realizar y cumplir todas las funcionalidades y especificaciones resumidas en el apartado anterior. Así las cosas, la elaboración de la prueba ha sido mucho más amigable y el programador ha tenido una mayor sensación de comodidad.

En suma a lo anterior, se ha realizado alguna que otra variación sobre las especificaciones de diseño relacionadas con las estructuras de datos a emplear o con las propiedades de los endpoints. La razón estriba en que, por un lado, la gestión de la información ha resultado ser mucho más sencilla, y por otro, que algún requisito propio de las APIs REST no cumplía del todo con el estándar. A lo largo del documento, y en los apartados pertinentes, se irán comentando más en detalle estas pequeñas variaciones efectuadas.

### 1.1. Análisis tecnológico

Antes de comenzar con el diseño e implementación, se realizó una investigación exhaustiva sobre las distintas tecnologías disponibles para despliegues, posibles soluciones de virtualización, programación de funcionalidades y almacenamiento persistente de datos.

#### 1.1.1. Despliegue de la APP

Para coordinar despliegues de aplicaciones con una arquitectura modular y altamente portable, Docker se posiciona como la plataforma tecnológica ideal. Su funcionamiento se

basa en el uso de imágenes y contenedores, donde cada contenedor actúa como una unidad independiente que incluye la aplicación junto con todas sus dependencias, lo que simplifica considerablemente su gestión. Además, esto garantiza que la aplicación se ejecute de forma idéntica sin importar ni el sistema operativo ni el tipo de entorno, ya sea desarrollo, pruebas o producción, asegurando así una alta portabilidad, ya que puede correr tanto en la nube como en máquinas locales.

### 1.1.2. Control de virtualización

Una vez se seleccionó la tecnología utilizada para desplegar la arquitectura de VentBook, el siguiente paso fue determinar sobre qué sistema se iba a correr la arquitectura. La máquina personal trabaja con Windows 10, lo cual no es ventajoso, ya que para desplegar Docker no hay nada mejor que tener una máquina Linux.

Si bien es cierto que Docker es compatible con ambas, para hacerlo funcionar en sistemas Windows hay que realizar diversos “empalmes” para que todo se comporte adecuadamente. Hay que instalar software adicional:

- Docker Desktop: para crear, compartir y ejecutar aplicaciones y microservicios en los contenedores por medio de sus respectivas imágenes.
- WSL: para poder instalar sobre Windows (sistema anfitrión) y poder correr un sistema nativo basado en Linux sin tener que recurrir a procesos de dual boot o arranque dual, ya que pueden poner en peligro las particiones de disco, y con ello, la integridad de la información almacenada en ellas.

Por fortuna o desgracia, se ha trabajado con Docker en ambos sistemas y se puede afirmar, sin duda alguna, que como ejecutarlo directamente en Linux no hay nada. La instalación en Windows es relativamente simple porque hay mucho soporte, hay manuales muy explícitos con los pasos, así como los posibles errores que nos vamos a encontrar, etcétera. Sin embargo, al buildear, bootear y ejecutar la APP podemos derivar en problemas de virtualización, sobrecarga del sistema anfitrión, problemas de memoria RAM, lentitud operativa, entre otros.

Por este motivo, se ha optado por emplear una VM gestionada por medio de VirtualBox que va a permitir correr un sistema operativo Linux sin todos estos problemas, puesto que la máquina va a estar optimizada en recursos desde su creación. Así las cosas, aunque se

invierta algo más de tiempo en el arranque, luego se dispone de una distro Linux mucho más flexible, totalmente personalizable, administrable plenamente en cuanto a recursos que consume y con total independencia del kernel de Windows.

### 1.1.3. Sistema operativo virtual

El siguiente paso ha sido determinar qué distro Linux utilizar dentro de la VM para correr la APP con Docker. Tras reflexionar sobre las opciones más comunes, se ha optado por emplear la distro Ubuntu 24.04 LTS, ya que cuenta con muy buenas referencias por parte de la comunidad, es altamente estable, ofrece mucha seguridad, y en última instancia, ofrece una interfaz gráfica muy visual que hace que trabajar con ella sea mucho más ergonómico de cara al usuario.

### 1.1.4. Herramientas de desarrollo

Finalmente, se ha realizado una evaluación detallada para seleccionar las tecnologías adecuadas que permitan desarrollar la totalidad de la funcionalidad de la aplicación. En consecuencia, la operatividad del software queda respaldada por las siguientes herramientas y componentes:

- PHP versión 8.2.0: se optó por utilizar esta versión del lenguaje en lugar de versiones más recientes, como la 8.4, debido a que las imágenes de Docker con FPM ofrecen una mayor estabilidad y seguridad comprobada. Además, facilitan la gestión de servicios y frameworks, como Symfony o Laravel, ya que las versiones más recientes aún presentan diversos problemas de compatibilidad.
- Symfony 6.2: uno de los frameworks por excelencia para trabajar con PHP en el desarrollo de aplicaciones web. Con una gran cantidad de funcionalidad a sus espaldas, Symfony es uno de los frameworks más cotizados del mercado.
- MySQL 8.0.26: se utiliza este sistema gestor de bases de datos relacional como la herramienta principal para construir las entidades de la aplicación, definiendo tablas, columnas y claves foráneas que estructuran los datos. Además, MySQL es el sistema fundamental para la ejecución de consultas SELECT y la actualización de información, garantizando la persistencia y consistencia de los datos bajo propiedades ACID.

Paralelamente, se ha empleado la interfaz gráfica de DBeaver como alternativa a la gestión mediante terminal, facilitando la administración y visualización de las bases de datos.

- Redis 7.0: se emplea un sistema adicional de persistencia en memoria o disco para el almacenamiento en caché de consultas. Está basada en Alpine y es sumamente ligera. Pese a presentar algunas vulnerabilidades de seguridad, resulta una opción adecuada debido a su amplia compatibilidad. Su función principal es optimizar el rendimiento de las consultas que podrían experimentar un aumento significativo en el tiempo de ejecución conforme crecen la complejidad y el volumen de datos de la aplicación. Por esta razón, en este contexto tan introductorio y cerrado, cumple un papel secundario y complementario de soporte.
- Swagger UI 5.27.1: se usa para documentar la API, ofreciendo información detallada sobre los distintos tipos de endpoints disponibles, los parámetros que aceptan, los tipos de solicitudes HTTP y las posibles respuestas junto con sus códigos de estado. Todo esto se presenta mediante una interfaz gráfica intuitiva que facilita la comprensión y visualización de la documentación. Además, permite realizar pruebas en tiempo real de los endpoints utilizando datos personalizados por el usuario, lo que mejora significativamente el proceso de validación y desarrollo.
- PHPUnit 9.6.23: ha permitido gestionar la ejecución de pruebas automatizadas: unitarias, de integración y funcionales, diseñadas a partir de un análisis de pruebas de caja negra, con el objetivo de verificar el correcto funcionamiento del código implementado.

Para las cuestiones de programación, se ha trabajado con el IDE de VSC para llevar a cabo toda la implementación de la API. Se han añadido diversas extensiones adicionales para trabajar con PHP y Symfony, principalmente, en términos de formato y estilo para asegurar los estándares de clean code y las buenas prácticas.

Por último, se ha empleado la herramienta Postman para realizar pruebas eficientes de la API. Esta permite enviar solicitudes HTTP a distintos endpoints, configurar parámetros, encabezados y cuerpos de las peticiones, así como recibir y analizar las respuestas. De este modo, se han podido ampliar y complementar las pruebas realizadas previamente con Swagger.

# Capítulo 2

## Primeros pasos de VentBook

Tras comentar todas las decisiones pertinentes que se tomaron en relación a las bases tecnológicas del proyecto, el siguiente paso fue llevar a cabo las instalaciones y configuraciones de todos los servicios detallados anteriormente.

### 2.1. Instalación de VirtualBox

El primer punto fue proceder con la instalación de VirtualBox (versión 7.1.12) de la página oficial. Para ello, se accedió a la siguiente URL: <https://www.virtualbox.org/>. Accediendo al botón ‘‘Download’’ y clickando en el archivo binario de acuerdo a nuestro sistema se descarga el fichero ejecutable (véase figura 2.1).

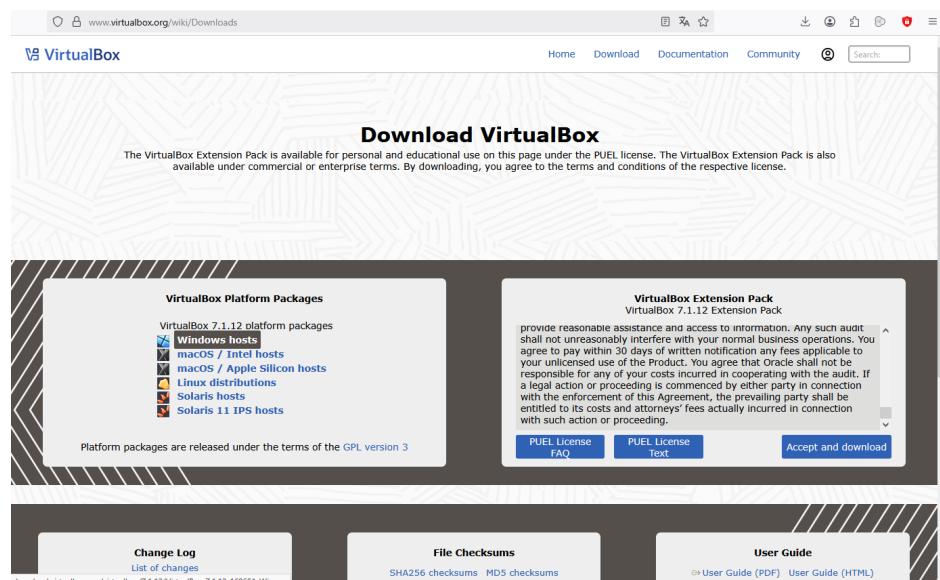


Figura 2.1: Instalación de VirtualBox 7.1.12 versión oficial

Tras verificar que VirtualBox funcionaba correctamente, se procedió a hacer lo mismo con la distro de Ubuntu. Se obtuvo de la página oficial: <https://ubuntu.com/download/desktop>. Se nos descarga un archivo ejecutable con extensión .deb correspondiente a la versión estable 24.04 LTS (véase figura 2.2)

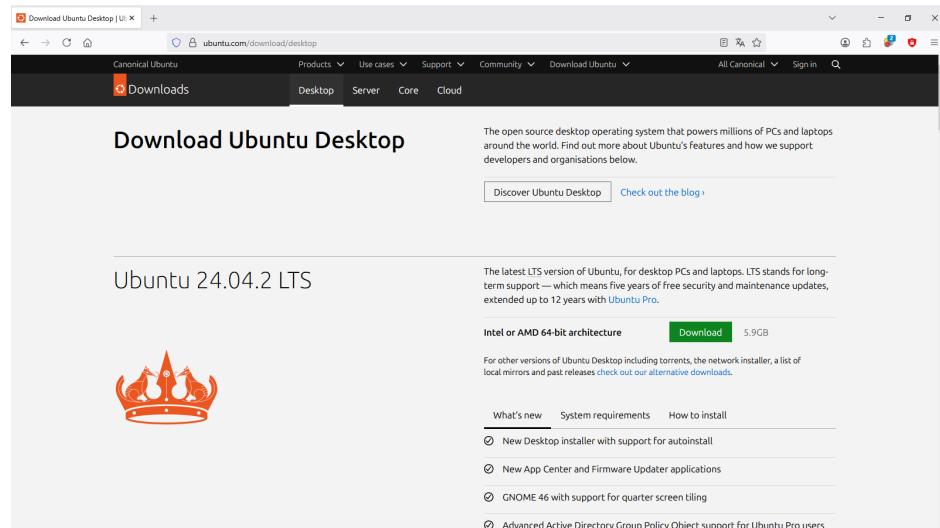


Figura 2.2: Instalación de Ubuntu 24.04 LTS versión oficial

Una vez descargado el ejecutable, se procedió con la creación de la VM. Desde el propio VirtualBox se creó la máquina de nombre Avoris, se especifica el directorio donde queremos que se guarde (recomendado el que se crea por defecto por el asistente de instalación) y la imagen ISO del sistema que vamos a alojar. En nuestro caso, un sistema Ubuntu de 64-bit (véase figura 2.3)

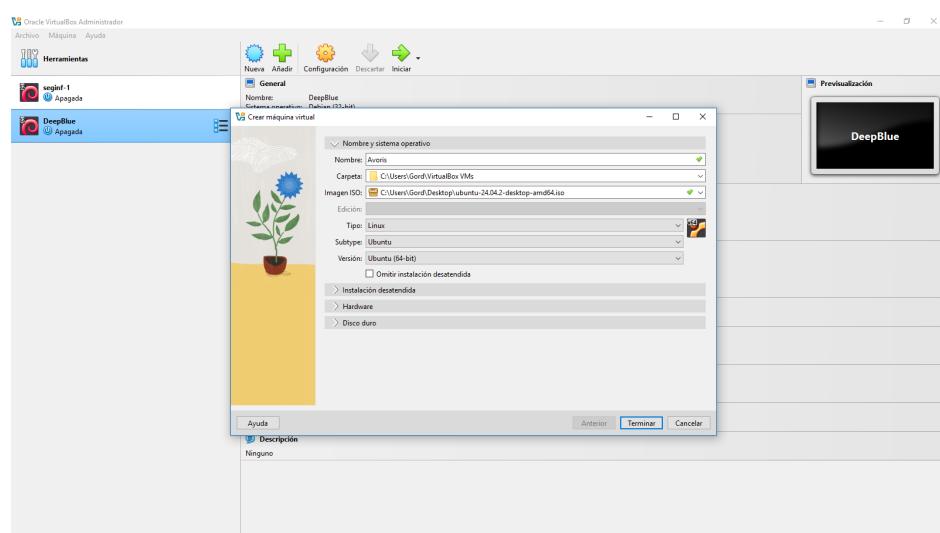


Figura 2.3: Configuración básica de la VM

Se procedió con la configuración de las credenciales de la VM. De todos los parámetros de configuración, interesa sobre todo el usuario y la password: **user** y **resu**, respectivamente. Se nos pedirá meterla al arrancar la VM, al bloquearse, cerrar sesión, para realizar acciones como superusuario, y vaga la redundancia, para ejecutar sudo (véase figura 2.4).

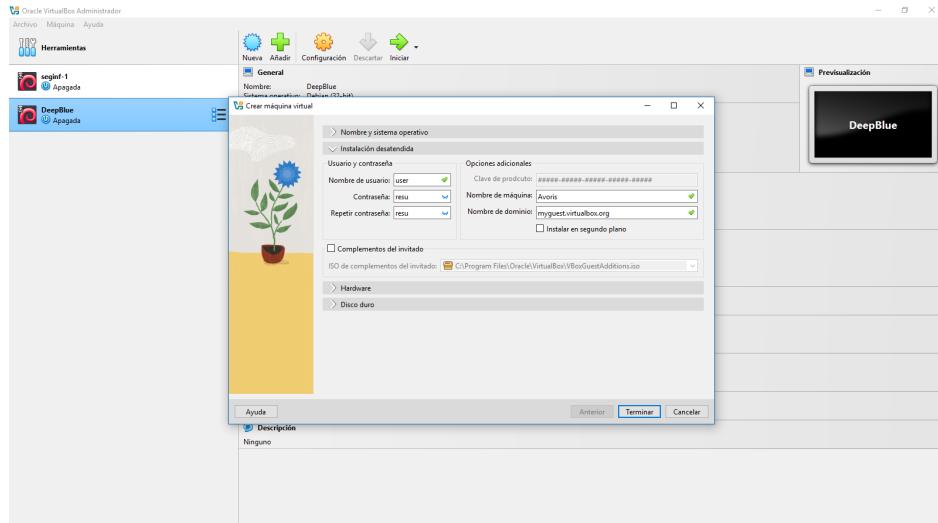


Figura 2.4: Configuración de credenciales de usuario administrador

A continuación, se configuraron los parámetros más relevantes de la máquina virtual. En este caso, se estableció una capacidad de disco duro de 25 GB (valor predeterminado), más que suficiente para el entorno requerido, junto con un procesador y una memoria RAM máxima de 8 GB. Tal y como se observa, la VM se ejecuta en condiciones óptimas, sin que el sistema operativo anfitrión experimente una disminución notable en su rendimiento (véanse figuras 2.5 y 2.6, respectivamente).

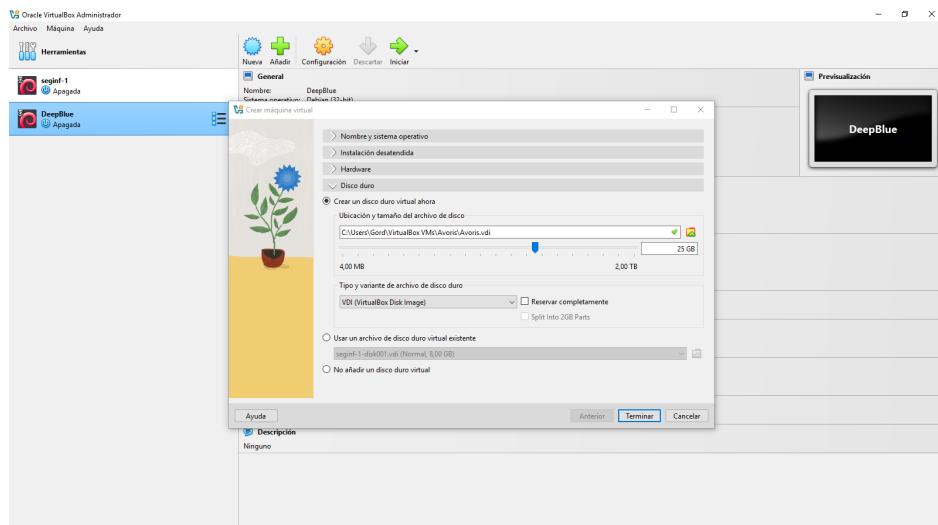


Figura 2.5: Configuración de tamaño del disco duro

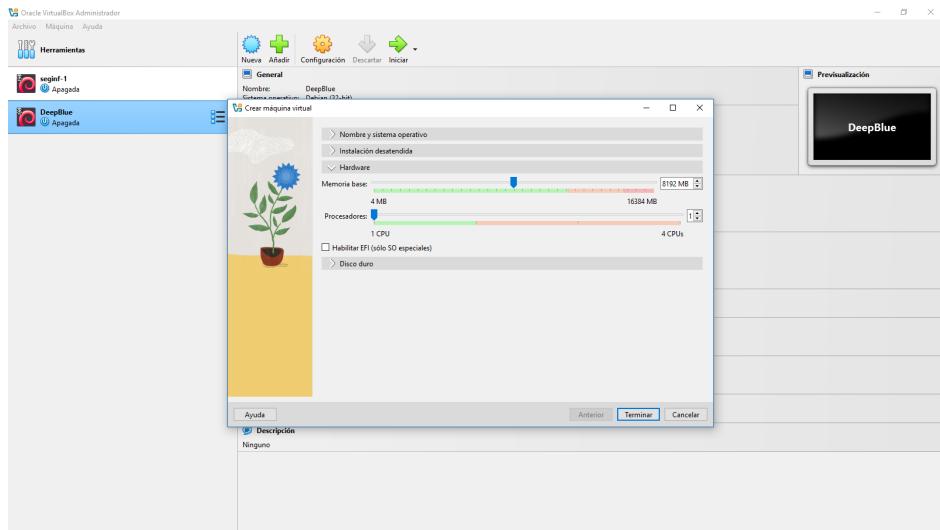


Figura 2.6: Configuración de recursos: RAM y los procesadores

Tras completar todos estos pasos, la máquina virtual ejecutó el asistente de instalación del sistema operativo con los ajustes previamente configurados. Una vez finalizado el proceso, la VM quedó prácticamente lista para su uso, quedando únicamente pendiente una tarea esencial: la instalación de las VirtualBox Guest Additions (véase figura 2.7).

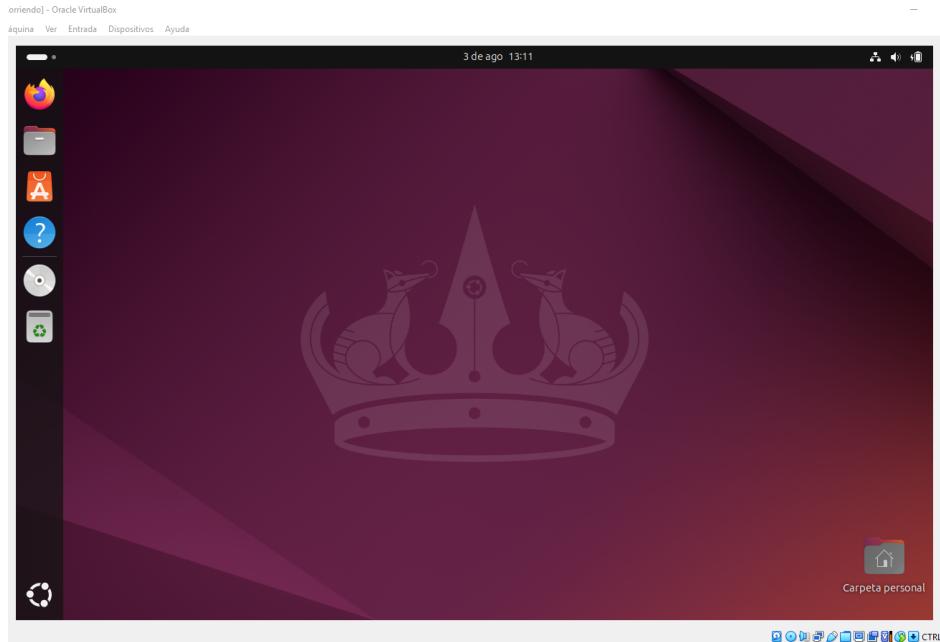


Figura 2.7: Máquina virtual ya en funcionamiento

## 2.2. Gestión de VirtualBox Guest Additions

Este conjunto de controladores y utilidades optimiza la integración entre el sistema operativo invitado y el anfitrión, incorporando mejoras como la redimensionación automática de la pantalla, el soporte para carpetas compartidas, una aceleración gráfica más eficiente, la sincronización del portapapeles y un rendimiento general más fluido de la máquina virtual. Además, contribuye a una gestión más eficiente de los recursos asignados, evitando que el consumo se acerque a los límites establecidos en la configuración y reduciendo así el riesgo de afectar la estabilidad del sistema operativo anfitrión.

Lo primero que hubo que hacer es asegurar que VirtualBox reconozca la funcionalidad óptica del paquete en el panel de administración de dispositivos ópticos y verificar que está marcado (aparece por defecto). Esto debe hacerse siempre con la VM apagada, aunque no se toque nada, puesto que al ser una zona de configuración altamente sensible pueden surgir problemas.

Seguidamente, con la VM arrancada, se accede a la sección de dispositivos de la interfaz de VirtualBox para montar las VirtualBox Guest Additions accediendo a la unidad desde terminal (véase figura 2.8).

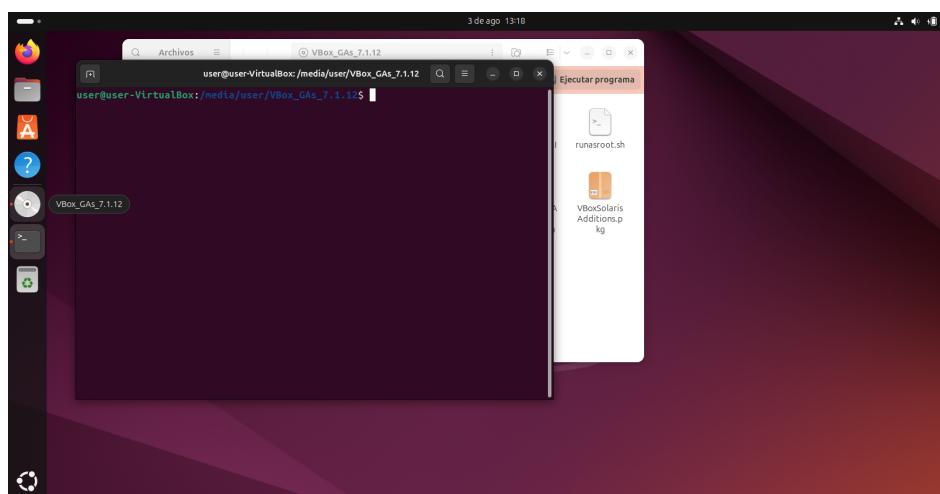


Figura 2.8: Montaje de las VirtualBox Guest Additions

Llegados a ese punto, estamos preparados para ejecutar los comandos que instalan la susodicha funcionalidad.

```
sudo apt-get update
sudo apt-get install bzip2
sudo apt-get install tar
sudo apt-get install gcc
sudo apt-get install build-essential gcc make perl dkms
sudo /sbin/recvboxadd quicksetup all
reboot
```

A grandes rasgos, el proceso consiste en actualizar la lista de paquetes disponibles desde los repositorios, instalar las herramientas necesarias para descomprimir los ficheros incluidos en la imagen ISO, instalar GCC para compilar los módulos del kernel que integran las Guest Additions, y añadir utilidades básicas de compilación como `make` y cabeceras estándar para recompilar los kernel headers. Finalmente, se reinicia el sistema para que los cambios tengan efecto.

Como paso adicional, se configuró una carpeta compartida para facilitar la transferencia de archivos (capturas de pantalla utilizadas) entre el sistema operativo anfitrión y la máquina virtual. En el SO anfitrión, se creó una carpeta en la ubicación y con el nombre deseados, habilitando las opciones de automontaje y montaje permanente para que la carpeta se monte automáticamente en cada inicio de la VM (véase figura 2.9). Posteriormente, desde el menú de carpetas compartidas de VirtualBox, se añadió dicha carpeta, denominada en este caso SharedFolder. Una vez completada la configuración, se accedió a la máquina virtual y se ejecutaron los siguientes comandos en la terminal:

```
sudo mkdir /mnt/SharedFolder
sudo mount -t vboxsf SharedFolder /mnt/SharedFolder
sudo usermod -aG vboxsf $(whoami)
```

Con esto lo que se hace es crear la carpeta SharedFolder en el directorio `/mnt` (carpeta donde se montan las unidades externas del sistema), se vincula la carpeta de la VM al enlace de la residente en el sistema anfitrión y se modifican los permisos porque por defecto solamente se tiene acceso como superusuario. Es importante enfatizar que con estos comandos los cambios no persisten tras reinicios o apagados. Para ello, hay que editar el fichero `/etc/fstab` y añadir el segundo comando de los anteriores. Así las cosas, la carpeta se nos automontará postapagado o postreinicio del sistema (véase figura 2.10).

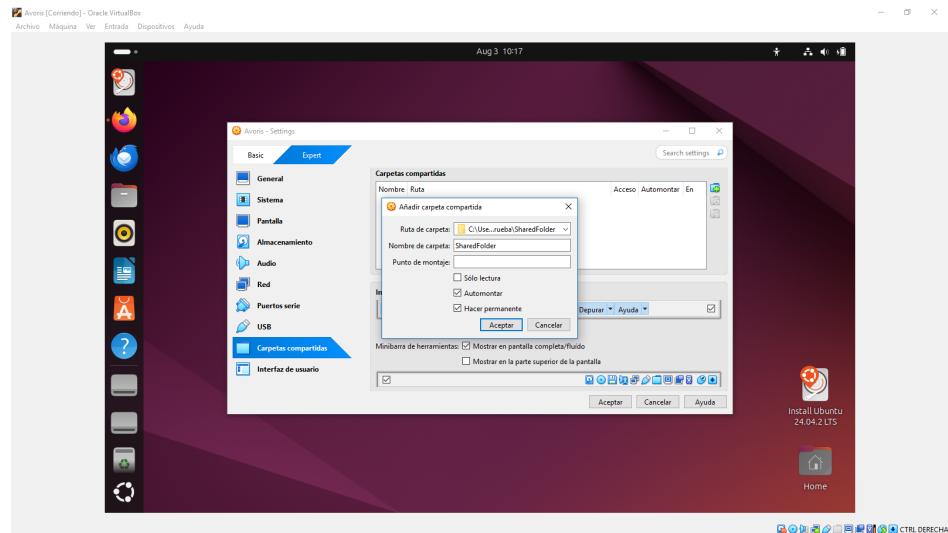


Figura 2.9: Configuración de la carpeta compartida en sistema anfitrión

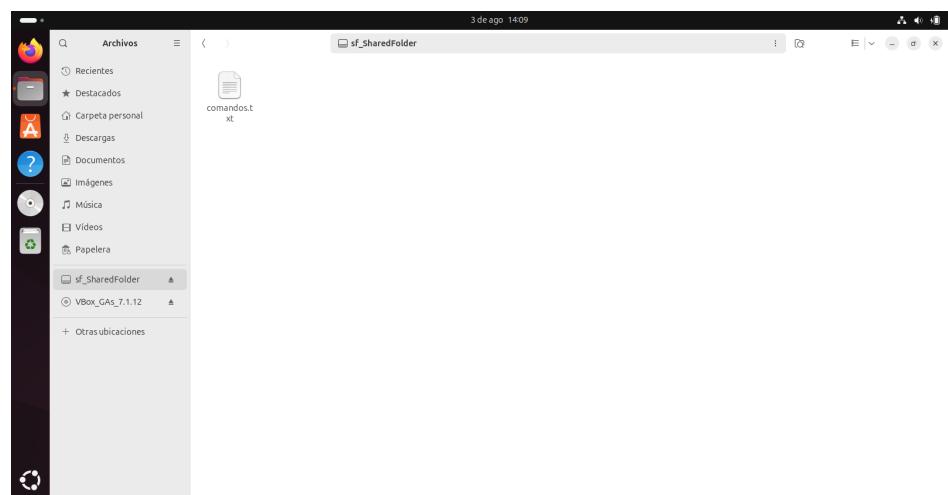


Figura 2.10: Visualización de la carpeta compartida desde la VM

## 2.3. Adición de software de desarrollo

Con todos los componentes básicos instalados, se procedió a configurar el resto de las herramientas de desarrollo. En este punto, se instalaron Google Chrome, el IDE Visual Studio Code junto con las extensiones para PHP y Symfony, así como Postman para realizar pruebas de los endpoints y validar la funcionalidad web de la lógica de negocio.

Dado que estos comandos no aportan un valor significativo al flujo principal del proyecto y no son relevantes para su desarrollo, se omiten aquí y se pasa directamente a la instalación de Docker y Docker Compose.

## 2.4. Bienvenida a Dokcer y Docker Compose

La instalación de Docker fue de lo más sencilla y sin problemas. Se ejecutó siguiendo las directrices de la página oficial de la herramienta. Dentro de la carpeta personal del usuario user, se creó una carpeta llamada *Avoris*: el directorio raíz del proyecto. Ahí es donde se llevó a cabo la instalación de Docker por medio de los siguientes comandos:

```
# Actualizar lista de paquetes
sudo apt update

# Preparar dependencias, keyrings y clave GPG
sudo apt install -y ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg \
-o /etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc

# Anyadir repositorio oficial de Docker a APT
echo \
"deb [arch=$(dpkg --print-architecture) signed-by=" \
"/etc/apt/keyrings/docker.asc] " \
https://download.docker.com/linux/ubuntu \
$(. /etc/os-release && echo \"$VERSION_CODENAME\") stable" | \
sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

# Actualizar lista de paquetes con el repositorio Docker
sudo apt update

# Instalar Docker y sus componentes principales
sudo apt-get install -y docker-ce docker-ce-cli containerd.io \
docker-buildx-plugin docker-compose-plugin

# Verificar que Docker se haya instalado correctamente
docker --version
docker compose version
sudo docker run hello-world

# Permitir usar Docker sin sudo
sudo usermod -aG docker $USER
newgrp docker
id -nG
```

Después de instalar ambas herramientas, en el directorio */Avoris* se generaron los archivos de configuración de Docker, quedando así todo listo para proceder con la configuración de despliegue de la aplicación VentBook.

# Capítulo 3

## VentBook: un software modular

Con todas las herramientas de despliegue y programación instaladas, es momento de detallar cómo se llevó a cabo la puesta a punto de la API de VentBook. Para asegurar que no se pierda ninguna restricción del diseño, evitar omisiones, mantener un flujo metodológico claro y facilitar su comprensión y explicación a terceros, se optó por una secuenciación modular basada en capas o fases. Además, se definieron una serie de hitos o puntos de control que, una vez alcanzados, permitieron avanzar de una fase a la siguiente. La secuencia de pasos es la siguiente.

### 3.1. Capa arquitectural

Esta primera fase del proyecto refleja la complejidad y el volumen de configuraciones necesarias para garantizar el correcto funcionamiento de VentBook. La gran cantidad de ajustes y comandos ejecutados evidencia el esfuerzo requerido para poner en marcha toda la arquitectura.

No obstante, la parte más esencial del despliegue reside en dos archivos ubicados en la carpeta principal /Avoris: el `Makefile` y el `docker-compose.yml`.

Estos dos archivos son la base fundamental del software desarrollado. El `Makefile` contiene las reglas que facilitan la interacción y gestión cómoda con Docker y sus contenedores, mientras que el `docker-compose.yml` define las instrucciones para construir la red Docker, crear las imágenes de los distintos servicios que componen el sistema, así como gestionar las dependencias necesarias para su correcto funcionamiento. A continuación, se explica muy brevemente el contenido de sendos archivos.

### 3.1.1. Fichero Makefile

Este Makefile está diseñado para facilitar la gestión del entorno de desarrollo basado en contenedores Docker, así como la ejecución de comandos comunes relacionados con la aplicación backend Symfony. Su objetivo principal es simplificar y estandarizar la ejecución de comandos complejos mediante alias fáciles de recordar y utilizar (véase figura 3.1).

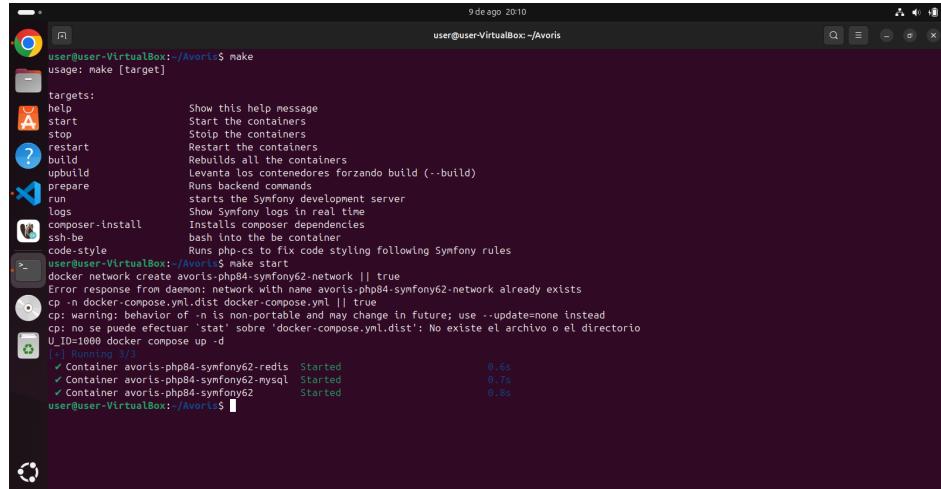
#### – Variables

- `UID`: captura el ID del usuario actual del sistema operativo. Esto es imprescindible para ejecutar comandos dentro de contenedores Docker con el mismo UID que el usuario host, evitando problemas de permisos sobre archivos creados o modificados dentro del contenedor. ¡Esto es super importante! Toma el valor 1000, puesto que `user` es el siguiente usuario creado en el sistema después de `root` ( $ID = 0$ ). Por convenio, el siguiente usuario tiene  $ID = 1000$ , y los demás que se añadan, tendrán IDs sucesivos.
- `DOCKER_BE`: define el nombre del contenedor backend (PHP/Symfony) para ser utilizado en comandos `docker exec` y otros. Toma como valor la constante '`avoris-php84-symfony62`'.

Paralelamente, contiene una serie de reglas que contribuyen a que la gestión de los contenedores Docker sea mucho más ergonómica:

- `help`: muestra una ayuda automática que lista todos los comandos disponibles en el Makefile junto con sus descripciones. Esto se logra con una expresión regular que extrae comentarios de los targets. Al ser la primera regla, simplemente poniendo `make` también funciona.
- `start`: crea una red Docker llamada '`avoris-php84-symfony62`' si no existe, copia el archivo `docker-compose.yml.dist` a `docker-compose.yml` (sin sobrescribir si ya existe) y levanta los contenedores en segundo plano (-d). Usa la variable `UID` para correr el comando con el UID del usuario.
- `stop`: detiene los contenedores activos definidos en el `docker-compose` actual.
- `restart`: combina el target `stop` seguido de `start` para reiniciar el entorno. En palabras llanas, apaga y reinicia los contenedores.

- build: construye las imágenes Docker sin levantar contenedores, asegurando que la red Docker application\_network exista.
- upbuild: construye las imágenes forzando la reconstrucción completa sin cache (`-no-cache`) y luego levanta los contenedores en modo detached.
- prepare: target de alto nivel que ejecuta tareas preparatorias; actualmente solo invoca la instalación de dependencias con `composer`.
- run: ejecuta el servidor de desarrollo Symfony dentro del contenedor backend, en modo demonio (`-d`).
- logs: muestra los logs en tiempo real del servidor Symfony dentro del contenedor backend.



```

user@user-VirtualBox:~/Avoris$ make
usage: make [target]

targets:
  help           Show this help message
  start          Start the containers
  stop          Stop the containers
  restart        Restart the containers
  build          Rebuilds all the containers
  upbuild        Levanta los contenedores forzando build (--build)
  prepare        Runs backend commands
  run            starts the Symfony development server
  logs           Show Symfony logs in real time
  composer-install  Installs composer dependencies
  ssh-be         bash into the be container
  code-style     Runs php-cs to fix code styling following Symfony rules
user@user-VirtualBox:~/Avoris$ make start
docker network create avoris-php84-symfony62-network || true
Error response from daemon: network avoris-php84-symfony62-network already exists
cp: in docker-compose.yml.dist docker-compose.yml || true
cp: warning: behavior of 'cp' is non-portable and may change in future; use --update=none instead
cp: no se puede efectuar 'stat' sobre 'docker-compose.yml.dist': No existe el archivo o el directorio
U_ID=1000 docker compose up -d
[+] Running 2/3
  ✓ Container avoris-php84-symfony62-redis Started              0.6s
  ✓ Container avoris-php84-symfony62-mysql Started                0.7s
  ✓ Container avoris-php84-symfony62 Started                      0.8s
user@user-VirtualBox:~/Avoris$ 

```

Figura 3.1: Ejemplos de opciones del comando `make`

### 3.1.2. Fichero docker-compose.yml

Ahora es el turno del fichero `docker-compose.yml`. Este archivo contiene las reglas que definen los diferentes contenedores Docker junto con los diferentes servicios y dependencias necesarios para que operen y se comuniquen correctamente. Hay que tener en cuenta que este fichero ha estado en continua modificación desde su origen, dado que el flujo metodológico empleado ha exigido ir añadiendo los diferentes servicios secuencialmente conforme los anteriores se creaban adecuadamente.

Pese a todo, se ha optado por hacer una sencilla descripción del contenido, que posteriormente, será complementada con una visión más profunda, conforme

paulatinamente, vayamos adentrándonos en los diferentes servicios que componen la arquitectura de EventBook.

– **Servicios definidos:**

- **avoris-php84-symfony62**
  - **Tipo:** contenedor backend PHP con Symfony 6.2.
  - **Construcción:** se construye desde un contexto local (`./docker`) pasando un argumento UID para mantener la coherencia de permisos entre host y contenedor.
  - **Montaje de volumen:** el directorio raíz del proyecto se monta dentro del contenedor en `/appdata/www`, permitiendo sincronización en tiempo real del código.
  - **Variables de entorno:**
    - ◊ Configuración para Xdebug con host y puerto cliente definidos para depuración remota.
    - ◊ URL de conexión a Redis, apuntando al servicio Redis interno.
    - ◊ Entorno de Symfony en modo desarrollo con debugging habilitado.
  - **Exposición de puertos:** mapea el puerto interno 8000 a 1000 en host, facilitando acceso al servidor web Symfony.
  - **Dependencias:** arranca después de que MySQL y Redis estén disponibles para asegurar integridad en la inicialización.
  - **Red:** se conecta a la red personalizada y artificial de nombre `avoris-php84-symfony62-network` donde se añadirán el resto de servicios.
- **avoris-php84-symfony62-mysql**
  - **Tipo:** base de datos MySQL versión 8.0.26.
  - **Exposición de puertos:** expone puerto 3306 dentro del contenedor hacia el puerto 3336 en el host.
  - **Configuración de entorno:**
    - ◊ Define base de datos y contraseña `root` para inicialización.
  - **Persistencia:** usa un volumen Docker nombrado para persistir datos en `/var/lib/mysql`.
  - **Configuración adicional:** establece conjunto de caracteres `utf8mb4` y colación para soporte multilenguaje y emojis.

- **Red:** conectado a la red personalizada común.
- `avoris-php84-symfony62-redis`
  - **Tipo:** contenedor Redis versión 7.0 en imagen ligera Alpine.
  - **Configuración de persistencia:** se habilita appendonly para registrar cada cambio en un archivo de log con sincronización cada segundo, asegurando durabilidad.
  - **Montaje de volumen:** persistencia de datos Redis en carpeta local `./redis-data`.
  - **Exposición de puertos:** Redis escucha en el puerto estándar 6379 y se expone igual en host.
  - **Red:** conectado a la red común de los servicios.
- **Redes y Volúmenes:**
  - **Red personalizada:** `avoris-php84-symfony62-network` usa el driver bridge para aislamiento y comunicación entre contenedores.
  - **Volumen Docker:** `avoris-php84-symfony62-mysql-data` reservado para almacenamiento persistente de datos MySQL.

El siguiente diagrama (véase figura 3.2) representa la arquitectura, indicando contenedores, tecnologías, versiones y puertos, lo que facilita la comprensión de la interconexión del sistema. Cabe señalar que no se han incluido en el diagrama ni Swagger ni PHPUnit porque no forman parte directa de la infraestructura de despliegue.

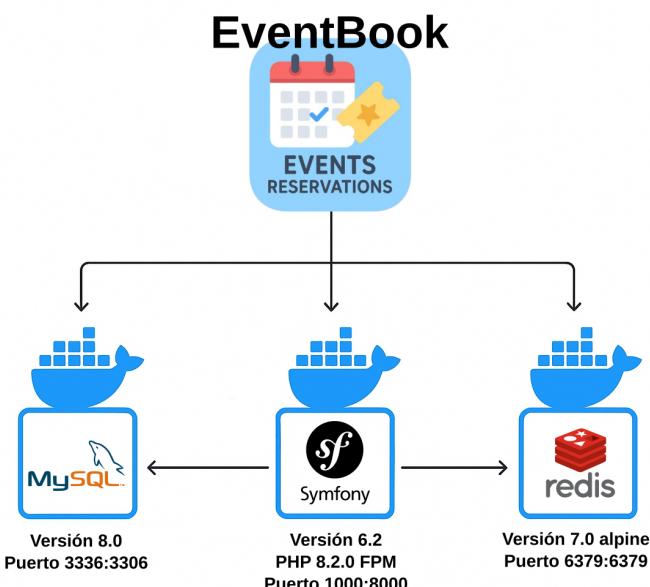


Figura 3.2: Diagrama de la arquitectura hexagonal de EventBook

## 3.2. Capa de aplicación

Una vez confeccionada la arquitectura hexagonal del sistema, se procedió a establecer conexión con el contenedor principal que va a almacenar toda la lógica de negocio. El framework encargado de esa tarea es **Symfony** (versión 6.2). Esta tecnología es la responsable de la implementación, por medio de PHP y de todos los recursos que ofrece, de gestionar los diversos casos de uso, entidades, servicios y reglas de negocio, así como los controladores y rutas necesarias para que la API, y en definitiva, EventBook funcionen de un modo agnóstico y modular.

### 3.2.1. Puesta punto del servidor

Para iniciar el proyecto Symfony, se accedió al directorio Avoris y, mediante el comando `make ssh-be`, que establece una conexión SSH con el contenedor, se ejecutaron los siguientes comandos dentro del mismo:

```
composer create-project symfony/skeleton:"6.2.*" myapp && \
cd myapp && rm -rf .git .gitignore
composer require symfony/routing symfony/dependency-injection
```

Estos comandos crean un proyecto Symfony con la arquitectura skeleton, ligera y sin dependencias pesadas. A continuación, se instalan los paquetes `symfony/routing` y `symfony/dependency-injection` para gestionar el enrutamiento y garantizar una correcta inyección de dependencias, elementos esenciales para el desarrollo de controladores y sus respectivas rutas.

Seguidamente, se ejecutó el comando `make run` (véase figura 3.3) para iniciar el servidor local de Symfony en el puerto 1000, deshabilitando la validación de certificados SSL/TLS (`-no-tls`) y permitiendo conexiones desde todas las interfaces (`-allow-all-ip`).

Finalmente, al acceder a la URL `http://localhost:1000/` se muestra la página principal de Symfony (véase figura 3.4) junto con la ruta donde está alojado el proyecto: `/appdata/www/myapp/`. En esta estructura, `/appdata/www` corresponde al directorio raíz de trabajo del contenedor, mientras que `myapp` contiene la instalación de Symfony, junto con el `composer` y el resto de sus dependencias.

```

user@user-VirtualBox:~/Avoris$ make run
U_ID=1000 docker exec -it --user 1000 avoris-php84.symfony62 bash -c "cd myapp && symfony local:serve:start --no-tls --allow-all-ip"
Following Web Server log file (/home/user/.symfony5/log/47fcf12651f97856293de636a8edf810d2b7d.log)
Following PHP-FPM log file (/home/user/.symfony5/log/47fcf12651f97857056293de636a8edf810d2b7d/53fb8e204547646acb3461995e4da5a54cc7575.log)

[WARNING] The local web server is optimized for local development and MUST never be used in a production setup.

[OK] Web server listening
The Web server is using PHP FPM 8.2.0
http://127.0.0.1:8000

[Web Server] Aug 9 19:59:38 | DEBUG | PHP Reloading PHP versions
[Web Server] Aug 9 19:59:38 | DEBUG | PHP Using PHP version 8.2.0 (from default version in SPATH)
[Application] Aug 8 18:45:45 | INFO | REQUESTS Matched route "event_display". method="GET" request_url="http://localhost:1000/events/1" route="event_display" route_parameters={"controller": "App\\Controller\\EventDisplayController", "route": "event_display", "id": "1"}
[Application] Aug 8 18:45:45 | INFO | SECURI Checking for authenticator support. authenticators=@firewall_name="main"
[Application] Aug 8 18:49:28 | INFO | REQUESTS Matched route "event_display". method="GET" request_url="http://localhost:1000/events/6" route="event_display" route_parameters={"controller": "App\\Controller\\EventDisplayController", "route": "event_display", "id": "6"}
[Application] Aug 8 18:49:28 | DEBUG | SECURI Checking for authenticator support. authenticators=@firewall_name="main"
[Application] Aug 8 18:49:40 | INFO | REQUESTS Matched route "event_display". method="GET" request_url="http://localhost:1000/events/7" route="event_display" route_parameters={"controller": "App\\Controller\\EventDisplayController", "route": "event_display", "id": "7"}
[Application] Aug 8 18:49:40 | DEBUG | SECURI Checking for authenticator support. authenticators=@firewall_name="main"
[Application] Aug 8 18:49:46 | INFO | REQUESTS Matched route "event_display". method="GET" request_url="http://localhost:1000/events/8" route="event_display" route_parameters={"controller": "App\\Controller\\EventDisplayController", "route": "event_display", "id": "8"}
[Application] Aug 8 18:49:46 | DEBUG | SECURI Checking for authenticator support. authenticators=@firewall_name="main"
[Application] Aug 9 18:09:01 | DEBUG | PHP Warning: stat failed for /appdata/www/myapp/var/cache/de/App_KernelDevDebugContainer.php
[Application] Aug 9 18:09:01 | ERROR | REQUESTS Uncought PHP Exception Symfony\Component\HttpFoundation\Exception\NotFoundHttpException: "No route found for "GET http://localhost:1000/" at /appdata/www/myapp/vendor/symfony/http-kernel/EventListener/RouterListener.php line 127
[Web Server] Aug 9 19:50:38 | INFO | PHP listening path="/usr/local/sbin/php-fpm" php="8.2.0" port=5807
[Web Server] Aug 9 19:50:38 | DEBUG | PHP Using PHP version 8.2.0 (from default version in SPATH)
[PHP-FPM] Aug 9 19:50:38 | DEBUG | RUNNER Waiting for channels (first boot) cmd="PHP-FPM"

```

Figura 3.3: Lanzamiento del servidor local de Symfony

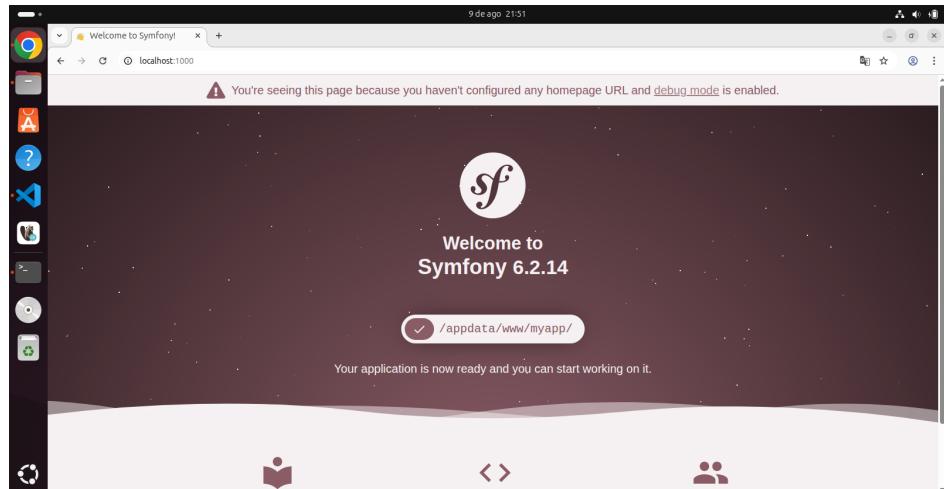


Figura 3.4: Página principal de funcionamiento de Symfony

### 3.2.2. Estructura de carpetas

El siguiente paso consistió en construir el árbol de directorios base que utilizaría la aplicación EventBook. Dicha estructura fue evolucionando progresivamente a medida que se añadían nuevas funcionalidades, módulos y dependencias. Por este motivo, se ha optado por representar gráficamente la estructura final obtenida tras la culminación del proyecto.

Cabe señalar que no se han incluido en esta representación los ficheros de configuración, servicios, DTOs, ensambladores, pruebas unitarias (tests), directorios de caché, entre otros, ya que su inclusión generaría una visualización excesivamente compleja y poco manejable. Estos elementos se irán mencionando de forma específica conforme se expliquen las subcapas de la lógica de negocio. A continuación, se muestra el esquema de carpetas.

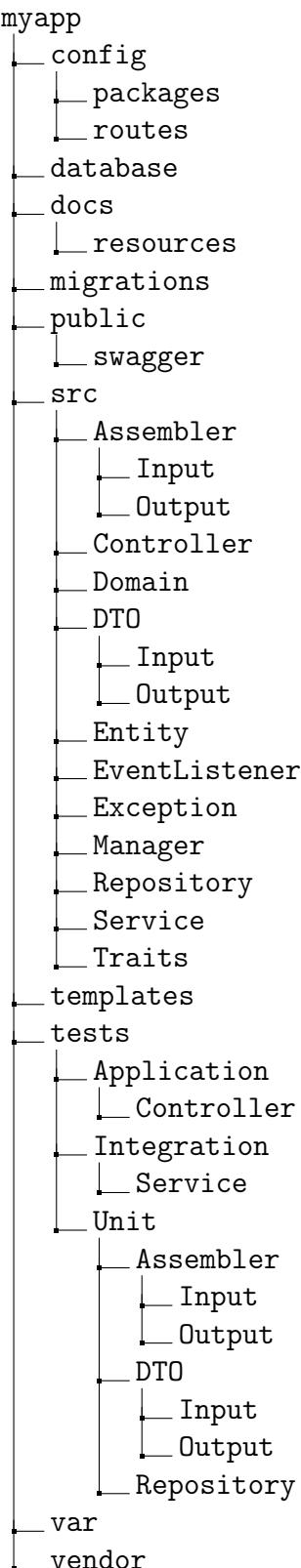


Figura 3.5: Árbol de directorios base de la aplicación EventBook.

### 3.3. Capa de infraestructura

Una vez definido el proyecto con su configuración base, que garantiza su correcto funcionamiento, el siguiente paso es desarrollar la estructura de entidades, DTOs, ensambladores, servicios, controladores y demás componentes necesarios para que el sistema cumpla las especificaciones. En síntesis, la capa de aplicación sostiene y orquesta toda la lógica de negocio, mientras que la capa de infraestructura implementa dicha lógica a nivel técnico. Aunque ambos conceptos están estrechamente relacionados y pueden parecer equivalentes, esta separación permite un mayor control, claridad en el flujo de trabajo y una planificación más precisa de los siguientes hitos.

En continuidad con lo anterior, la fase de infraestructura ha consistido en la implementación del conjunto completo de clases, entidades y estructuras requeridas, siguiendo las buenas prácticas de Symfony, las restricciones definidas al inicio del documento y los principios de programación recomendados. A continuación, se detallan las diferentes estructuras implementadas y su correspondencia dentro de la jerarquía de directorios del proyecto.

#### 3.3.1. Entidades

La primera subcapa abordada dentro de la infraestructura fue el diseño de las identidades con las que operará la API, concretamente Event y Booking, ubicadas en `src/Domain`.

Es importante señalar que las entidades se han ubicado en el directorio `Domain` en lugar de `Entity`, que según la documentación de Symfony es la ubicación estándar. Esta decisión obedece a que el directorio `Entity` está orientado a proyectos que utilizan `Doctrine`, donde se emplean anotaciones ORM y migraciones para reflejar en la base de datos los cambios en las entidades.

En este proyecto se optó deliberadamente por no instalar `Doctrine`, con el fin de evitar el uso de un ORM. En consecuencia, y siguiendo las recomendaciones de Symfony para entornos sin `Doctrine`, se ha utilizado el directorio `Domain` como ubicación de las entidades.

Paralelamente, otra de las especificaciones que se pedían era prestar atención a la inmutabilidad de las entidades para hacer más fácil el comportamiento del código. En los atributos de los constructores de sendas entidades se ha utilizado la palabra reservada de PHP `readonly`.

Por último, cabe señalar que se realizaron ciertos ajustes de diseño en los atributos de las estructuras Evento y Reserva, concretamente en la nomenclatura de algunas propiedades. A continuación, se presenta una comparativa entre la estructura definida en el enunciado y la propuesta final. Tal y como se aprecia, la cantidad de propiedades y sus tipos se mantienen inalterados, habiéndose modificado únicamente algunos nombres con el fin de dotarlos de una mayor claridad y capacidad descriptiva.

```
{  
    "id": "unique event reference (integer)",  
    "name": "event name (string)",  
    "description": "event description (string)",  
    "from": "initial event date (string formato yyyy-mm-dd)",  
    "to": "final event date (string) formato yyyy-mm-dd)",  
    "available": "available seats (integer)"  
}
```

```
{  
    "id": "unique event reference (integer)",  
    "name": "event name (string)",  
    "description": "event description (string)",  
    "fromDate": "initial event date (string formato yyyy-mm-dd)",  
    "toDate": "final event date (string) formato yyyy-mm-dd)",  
    "availableSeats": "available seats (integer)"  
}
```

Figura 3.6: Comparativa de la estructura Evento (original vs propuesta)

```
{
  "reference": "unique booking reference (string)",
  "eventId": "related event reference (integer)",
  "eventDate": "event date selected (string formato yyyy-mm-dd)",
  "attendees": "number of people included (string)",
  "identification": "buyer's personal id (string formato DNI)"
}
```

```
{
  "reference": "unique booking reference (string)",
  "eventId": "related event reference (integer)",
  "eventDate": "event date selected (string formato yyyy-mm-dd)",
  "attendees": "number of people included (string)",
  "buyerId": "buyer's personal id (string formato DNI)"
}
```

Figura 3.7: Comparativa de la estructura Reserva (original vs propuesta)

### 3.3.2. Data Transfer Objects

Tras definir las entidades, el siguiente aspecto fue proceder con el diseño de los DTOs, es decir, las estructuras de datos que van a posibilitar el intercambio de información entre los diferentes elementos que subyacen dentro de la capa de infraestructura. Los DTOs se han clasificado en los siguientes grupos:

- **DTOs de entrada:** estructuras de datos que reciben las distintas capas de la lógica de negocio. Tras ser validadas de forma estricta, se utilizan para efectuar los cálculos, transformaciones o modificaciones requeridas. Se ubican en `src/DTO/Input`.
- **DTOs de salida:** estructuras generadas como resultado de los cálculos, procesamientos y operaciones efectuadas por la lógica de negocio. Se almacenan en `src/DTO/Output`.

La siguiente tabla muestra un breve resumen de todos los DTOs que se han creado, indicando si son de entrada o salida, así como su cometido dentro de la lógica de infraestructura, es decir, la información que representan:

Nombre DTO	Input o Output	Descripción
BookingCancelDTO	Input	Cancelación de una reserva
BookingCreateDTO	Input	Creación de una reserva
BookingListingDTO	Input	Listado de reservas
EventCreateDTO	Input	Creación de un evento
EventDisplayDTO	Input	Detalles de evento
EventOptionalFilterDisplayDTO	Input	Opciones de filtrado de eventos
BookingInfoDataDTO	Output	Información de una reserva
EventInfoDataDTO	Output	Salida de un evento

Tabla 3.1: Conjunto de DTOs diseñados

En relación a los DTOs, otro aspecto de diseño que no se ha recomendado seguir es la inmutabilidad. La arquitectura propuesta está basada en un flujo de validación de datos de entrada y salida mediante ensambladores. Si bien es cierto que los DTOs inmutables son más restrictivos en cuanto a comportamientos no deseados, el hecho de que sean mutables ofrece diversas ventajas, como menos código repetido, mucha más flexibilidad para construir los DTOs e irlos inicializando según se necesite, y además, es un comportamiento más natural para Symfony, ya que hace posible la serialización de estos para, por ejemplo, convertirlos a JSON al enviar la respuesta en las peticiones.

Ahora bien, pese a que son mutables, la siguiente decisión de diseño ha pasado por incorporar cláusulas de validación por medio de anotaciones basadas en atributos (véase figura 3.8) para controlar aspectos de tipado y valores posibles críticos.

```

13 class EventCreateDTO
14 {
15     /**
16      * Description of the event.
17      *
18      * @var string
19      */
20     #[AssertNotBlank]
21     #[AssertNotNull]
22     #[AssertType('string')]
23     public string $description;
24
25     /**
26      * Start date of the event.
27      *
28      * @var DateTimeImmutable
29      */
30     #[AssertNotBlank]
31     #[AssertNotNull]
32     #[AssertExpression(
33         "this->fromDate->lte(this->toDate)",
34         "The start date must be less than or equal to the end date."
35     )]
36     public DateTimeImmutable $fromDate;
37
38     /**
39      * End date of the event.
40      *
41      * Must be greater than or equal to the start date.
42      *
43      * @var DateTimeImmutable
44      */
45     #[AssertNotBlank]
46     #[AssertNotNull]
47     #[AssertExpression(
48         "this->fromDate->lt(this->toDate)",
49         "The start date must be less than or equal to the end date."
50     )]
51     public DateTimeImmutable $toDate;
52
53     /**
54      * Number of available seats for the event.
55      *
56      * @var int
57      */
58     #[AssertExpression(
59         "this->seats >= 0",
60         "The number of seats must be greater than or equal to zero."
61     )]
62     public int $seats;
63 }

```

Figura 3.8: Ejemplo de DTO con anotaciones de atributos

La razón por la que se han usado anotaciones basadas en atributos es porque son muy fiables y consistentes. Además, desde PHP 8.0 se les ha dado soporte nativo en el lenguaje,

enfatizando el papel positivo que ejercen. Asimismo, tienen el mismo rol que las anotaciones basadas en ORM, las cuales poco a poco están quedando deprecadas, y se aprovecha para recordar que no pueden utilizarse.

### 3.3.3. Ensambladores

Tras la creación de los DTOs, se continuó con las estructuras encargadas de la validación de tipado y contenido de estos: las clases ensambladoras. Al igual que ocurrió con los DTOs, los ensambladores están clasificados de la misma manera, es decir, en entrada o salida. Los ensambladores de entrada se localizan en `src/Assembler/Input`, mientras que los de salida residen en `src/Assembler/Output`. La siguiente tabla informativa reseña las estructuras tanto de entrada como de salida que se han creado y los DTOs que validan.

Nótese que los nombres de los `assemblers` siguen el patrón de nomenclatura propuesto por el estándar de `Symfony`, que consiste en añadir a la clase de DTO que construyen y validan el sufijo `Assembler` (véase figura 3.9). De este modo, se visualiza de manera fácil e intuitiva la relación entre ambos conceptos.

Nombre Assembler	Input o Output
<code>BookingCancelDTOAssembler</code>	Input
<code>BookingCreateDTOAssembler</code>	Input
<code>BookingListingDTOAssembler</code>	Input
<code>EventCreateDTOAssembler</code>	Input
<code>EventDisplayDTOAssembler</code>	Input
<code>EventOptionalFilterDisplayDTOAssembler</code>	Input
<code>BookingInfoDataDTOAssembler</code>	Output
<code>EventInfoDataDTOAssembler</code>	Output

Tabla 3.2: Conjunto de `assemblers` diseñados

```

10 de ago 13:01
File Edit Selection View Go Run Terminal Help < - > ⌘ Avisos
gCreateService.php docker-compose.yml Makefile nelmio_cors.yaml Booking.php Event.php BookingCreateDTOAssembler.php ...
myapp>src>Assembler>Input>BookingCreateDTOAssembler.php > ...
13 class BookingCreateDTOAssembler
21 * @param ValidatorInterface $validator Symfony validator component.
22 */
23 public function __construct(ValidatorInterface $validator)
24 {
25     $this->validator = $validator;
26 }
27 /**
28 * Creates and validates a BookingCreateDTO from the given parameters.
29 *
30 * @param array $params Raw request parameters.
31 * @return BookingCreateDTO Validated DTO.
32 * @throws ValidationFailedException if any constraint violations occur.
33 */
34 public function transform(array $params): BookingCreateDTO
35 {
36     $booking = new BookingCreateDTO();
37     $booking->setReference($params['reference']);
38     $booking->setEventId(new \DateTimeImmutable($params['event_id']));
39     $booking->setEventDate(new \DateTimeImmutable($params['event_date']));
40     $booking->setAttendees($params['attendees']);
41     $booking->setIdentification($params['identification']);
42     $violations = $this->validator->validate($booking);
43     if (count($violations) > 0) {
44         throw new ValidationFailedException($booking, $violations);
45     }
46     return $booking;
47 }
48 }
49 }
50 }
51 }
52 }

```

Ln 9, Col 1 Spaces: 4 UTF-8 LF () PHP

Figura 3.9: Ejemplo de ensamblador de entrada

### 3.3.4. Servicios

Posteriormente, se ha diseñado la implementación de los servicios, es decir, las clases encargadas de llevar a cabo el cómputo y la lógica del negocio. Estos elementos son los encargados de invocar a los ensambladores descritos en el bloque anterior para poder trabajar con los DTOs en condiciones óptimas. Y por supuesto, también se encargan de operar con los repositorios (definidos en la capa de persistencia) para efectuar cambios en la BD: inserciones, borrados, actualizaciones y proyecciones o consultas.

Se han diseñado un total de seis servicios distintos, a razón de uno por funcionalidad a implementar. Estos componentes, como veremos más adelante, son las unidades invocadas en los controladores de la API. Los servicios, según el estándar de Symfony, se ubican en la carpeta /src/Service. La siguiente tabla muestra el listado de servicios que se han creado junto con su cometido. Para más información se puede consultar el código de cada uno de ellos.

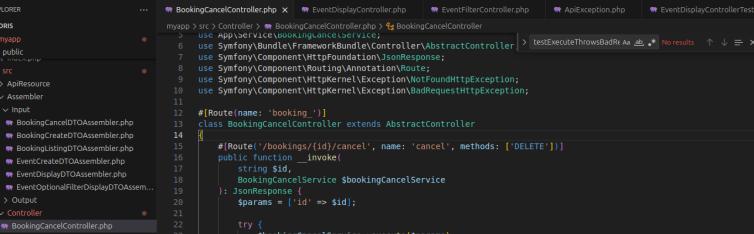
Nombre del servicio	Descripción
BookingCancelService	Cancelación de una reserva
BookingCreateService	Creación de una reserva
BookingListingService	Listado de reservas por DNI de cliente
EventCreateService	Creación de un evento
EventDisplayService	Muestra detalles de un evento
EventOptionalFilterDisplayService	Listado de eventos según una serie de filtros

Tabla 3.3: Conjunto de servicios diseñados

### 3.3.5. Controladores

Por último, se llevó a cabo la implementación de los controladores encargados de invocar a los servicios anteriormente descritos. Los controladores son, dentro de la capa de infraestructura, los últimos elementos que se han creado, puesto que dependen de todos los bloques anteriores, y además, son los puntos de entrada de los clientes a la API. Dentro del proyecto se localizan en `/src/Controller`.

Al igual que se hizo con los DTOs, se han utilizado anotaciones basadas en atributos para establecer las rutas de los endpoints (véase figura 3.10). Por ello, no ha sido necesario utilizar las anotaciones ORM para garantizar los enrutamientos.



The screenshot shows a PHP development environment with the following details:

- File Explorer (Left):** Shows the project structure with nodes like **AVORIS**, **myapp**, **public**, **src**, **bin**, **Controller**, **BookingCancelController.php** (selected), **EventDisplayController.php**, **EventFilterController.php**, **ApiException.php**, and **EventDisplayControllerTest.php**.
- Code Editor (Center):** Displays the **BookingCancelController.php** file content. The code handles booking cancellation and filtering events.
- Search Results (Top Right):** Shows a search bar with "Avoris" and a results table with one entry: "BookingCancelController.php" at line 14, with the message "No results".
- Status Bar (Bottom):** Shows the date and time as "10 de ago 21:52" and the file path as "BookingCancelController.php".

Figura 3.10: Ruta y tipo de controlador con anotaciones en atributos

Haciendo uso del comando `php bin/console debug:router` dentro del contenedor Docker, en el proyecto Symfony, podemos obtener un listado con las rutas activas (véase figura 3.11).

Llegados a este punto, es importante comentar otro cambio de diseño en relación a las rutas propuestas. Inicialmente, el endpoint de borrado se planteó como GET, pero esto no cumple con las especificaciones de una API REST bien construida, ya que un endpoint cuyo cometido es el borrado de un recurso debe ser de tipo DELETE.

Así, los endpoints que realizan operaciones de lectura deben ser GET, los que crean recursos deben ser POST, y los que actualizan, POST o PUT, según se desee.

```

user@user-VirtualBox:~/Avenir$ make ssh-be
U_ID=1000 docker exec -t user 1000 avoris-php84-symfony62 bash
user@2680e50edd67:/appdata/www/myapp/
user@2680e50edd67:/appdata/www/myapp$ php bin/console debug:router
  Name        Method Scheme Host Path
-----+-----+-----+-----+-----+
  api_genid      ANY    ANY    ANY    /api/.well-known/genid/{id}
  api_errors      ANY    ANY    ANY    /api/errors/{status}
  api_validation_errors ANY    ANY    ANY    /api/validation_errors/{id}
  api_entrypoint   ANY    ANY    ANY    /api/{index}.{_format}
  api_doc         ANY    ANY    ANY    /api/docs.{_format}
  api_jsonld_context ANY    ANY    ANY    /api/exchange/{language}.{_format}
  error_max_error  ANY    ANY    ANY    _error/{code}.{_format}
  booking_cancel  DELETE  ANY    ANY    /bookings/{id}/cancel
  booking_create   POST   ANY    ANY    /bookings
  booking_list     GET    ANY    ANY    /bookings/{id}
  event_create     POST   ANY    ANY    /events
  event_display    GET    ANY    ANY    /events/{id}
  event_filter_list GET    ANY    ANY    /events

```

Figura 3.11: Rutas disponibles de los controladores

Para poder hacer pruebas exhaustivas de los endpoints, se ha instalado la herramienta de Postman (véase figura 3.12). De esta forma, se han probado los endpoints con las URLs, las distintas casuísticas, así como las pertinentes respuestas obtenidas en base a los diversos casos de entrada propuestos.

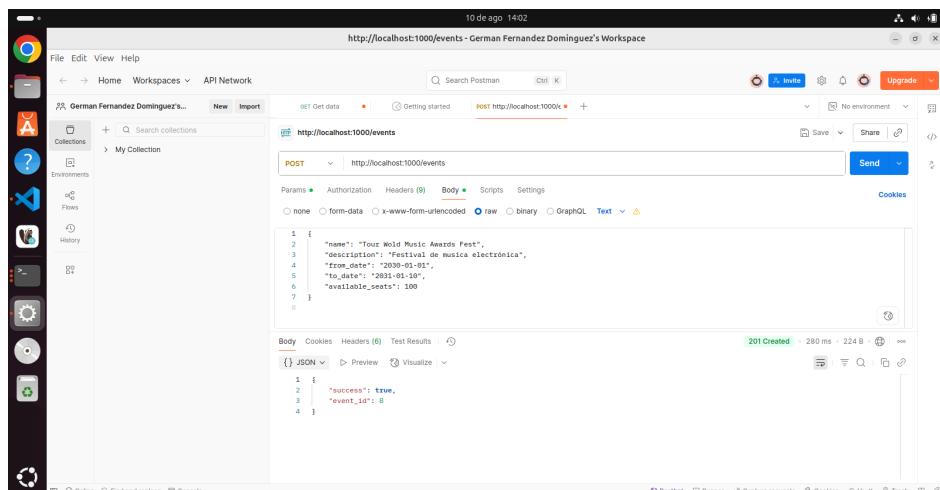


Figura 3.12: Ejemplo de petición hecha con Postman

La captura evidencia una petición a la URL: `http://localhost:1000/events`, que es de tipo POST. Adicionalmente, se especifica en los headers que los parámetros insertados en el body son de tipo JSON mediante el campo `application/json`. La respuesta es también un JSON que ha devuelto un flag de éxito y el último ID del evento creado. Además, está retornando como código de respuesta HTTP el valor 201, que de acuerdo al estándar de API REST se identifica con el estado de recurso creado con éxito.

## 3.4. Capa de persistencia

Tras tener ya creadas las capas de arquitectura e infraestructura, se procedió a terminar de depurar los servicios de persistencia de datos, es decir, la capa de persistencia.

En líneas generales, este módulo está integrado por dos bases de datos: una en MySQL 8.0.26 para controlar todo el procesamiento de guardado y manipulación de datos primarios, y otra en Redis 7.0, que funciona como un componente auxiliar para optimizar los resultados de consulta mediante un sistema de caché basado en QueryResult, es decir, en el almacenamiento y optimización de los resultados de consulta. En los siguientes apartados se explican cómo se configuraron ambos servicios.

### 3.4.1. Configuración de la BD MySQL 8.0.23

El primer paso fue añadir en el fichero docker-compose.yml las líneas de código pertinentes para poder levantar el contenedor Docker correspondiente.

#### Estructura del docker-compose.yml

```
Docker Compose YAML

avoris-php84-symfony62-mysql:
  container_name: avoris-php84-symfony62-mysql
  image: mysql:8.0.26
  ports:
    - '3336:3306'
  environment:
    MYSQL_DATABASE: mysql_symfony_avoris
    MYSQL_ROOT_PASSWORD: avoris
  volumes:
    - avoris-php84-symfony62-mysql-data:/var/lib/mysql
  networks:
    - avoris-php84-symfony62-network
  command: [ 'mysqld', '--character-set-server=utf8mb4',
    --collation-server=utf8mb4_unicode_ci' ]
```

Este fragmento define un servicio de base de datos MySQL en el archivo docker-compose.yml para nuestro entorno de desarrollo Symfony con PHP 8.2:

- **Nombre del servicio:** avoris-php84-symfony62-mysql, que también es el

nombre asignado al contenedor Docker mediante `container_name` para facilitar su identificación.

- **Imagen:** utiliza la imagen oficial `mysql:8.0.26`, una versión estable de MySQL 8.
- **Puertos:** el puerto local 3336 se mapea al puerto 3306 del contenedor, que es el puerto estándar de MySQL, permitiendo conexiones externas al servidor MySQL dentro del contenedor.
- **Variables de entorno:** se configura una base de datos inicial llamada `mysql_symfony_avoris` y se establece la contraseña de `root` como `avoris`.
- **Volúmenes:** se monta un volumen Docker llamado `avoris-php84-symfony62-mysql-data` para persistir los datos de MySQL en `/var/lib/mysql`, asegurando que los datos no se pierdan al reiniciar el contenedor.
- **Redes:** el contenedor se conecta a la red personalizada cuyo nombre es `avoris-php84-symfony62-network` para comunicarse con otros servicios en el entorno Docker.
- **Comando personalizado:** se inicia el servidor MySQL con configuración específica para usar el conjunto de caracteres `utf8mb4` y la colación `utf8mb4_unicode_ci`, que son recomendadas para soportar correctamente caracteres Unicode, incluidos emojis y otros símbolos.

Esta configuración proporciona un entorno reproducible y aislado para el desarrollo y pruebas de aplicaciones Symfony que requieren acceso a una base de datos MySQL.

Mediante esta operativa, se consigue que el contenedor de aplicación con Symfony y PHP esté totalmente desacoplado del empleado para MySQL. Así las cosas, desde el directorio raíz /Avoris podemos acceder al susodicho contenedor (véase figura 3.13) y ver que la comunicación con él es exitosa.

```

4 de ago 00:09
user@user-VirtualBox:~/Avoris$ ss -tln
          bign := [established|syn-sent|fin-wait-{1,2}|closed|close-wait|last-ack|listening|closing]
NetId State      Recv-Q Send-Q Local Address:Port          Peer Address:Port      Process
[-]  NetId      State      Recv-Q Send-Q Local Address:Port          Peer Address:Port      Process
  udp  UNCONN    0      0      127.0.0.54:53          0.0.0.0:*          0.0.0.0:*
  udp  UNCONN    0      0      127.0.0.53%lo:53          0.0.0.0:*
  udp  UNCONN    0      0      0.0.0.0:33749          0.0.0.0:*
  udp  UNCONN    0      0      0.0.0.0:353            0.0.0.0:*
  udp  UNCONN    0      0      [:]:5353              [:]:*
  udp  UNCONN    0      0      [:]:52507             [:]:*
  tcp  LISTEN   0      4096   127.0.0.54:53          0.0.0.0:*
  tcp  LISTEN   0      4096   127.0.0.1:631          0.0.0.0:*
  tcp  LISTEN   0      4096   0.0.0.0:3336          0.0.0.0:*
  tcp  LISTEN   0      4096   0.0.0.0:1000          0.0.0.0:*
  tcp  LISTEN   0      4096   127.0.0.53%lo:53          0.0.0.0:*
  tcp  LISTEN   0      59     [::ffff:127.0.0.1]:37615  [::]:*
  tcp  LISTEN   0      4096   [::]:3336             [:]:*
  tcp  LISTEN   0      4096   [:]:631              [:]:*
  tcp  LISTEN   0      4096   [:]:1000             [:]:*
user@user-VirtualBox:~/Avoris$ docker exec -it avoris php84-symfony62 mysql bash
root@eb7a5ea0600:/# mysql -u root -p
Enter password:
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 8
Server version: 8.0.26 MySQL Community Server - GPL

Copyright (c) 2000, 2021, Oracle and/or its affiliates.

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>

```

Figura 3.13: Comunicación con el contenedor de MySQL 8.0.23

## Instalación de DBeaver

La visión y gestión de una BD a través de consola puede ser bastante molesta. Por ello, se ha decidido instalar el SGBD de DBeaver, una de las herramientas más completas en cuanto al manejo de BDs, ya que permite crear instancias de numerosos tipos de bases de datos.

Tras la instalación del software, se estableció una conexión con la BD a través del programa (véase la figura 3.14).

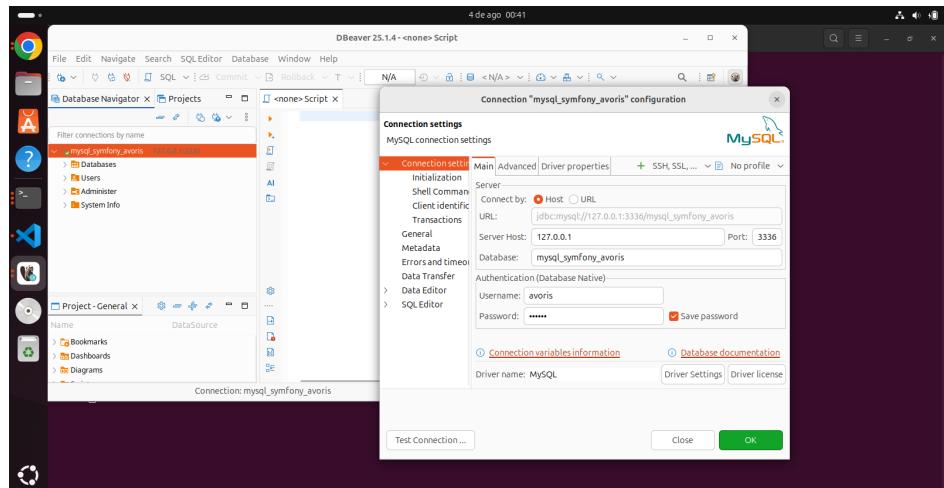


Figura 3.14: Conexión con la BD de MySQL 8.0.23 usando DBeaver

## Schema y poblado de la BD MySQL 8.0.23

El siguiente hito fue crear el schema de la BD, es decir, el conjunto de tablas que conforman la estructura de negocio a nivel de persistencia, especificando las columnas, los tipados y restricciones en rango de valores, así como las relaciones y claves foráneas entre ellas. En este sentido, un evento puede tener muchas reservas, mientras que una reserva pertenece a un único evento. Por consiguiente, la relación entre ambas entidades es del tipo 1:N.

Otro aspecto importante a recalcar es que no se tiene ni Doctrine ni sus dependencias instaladas. Así las cosas, esto en el punto de migrar y poblar es muy importante. Los proyectos de Symfony con Doctrine permiten, mediante el comando `php bin/console doctrine:migrations:diff`, comparar el schema actual con la estructura de las entidades que hay en `/src/Entity` y generar nuevas migraciones para ejecutar en la ruta `/src/migrations`. En este proyecto, nada de esto figura porque no hay nada instalado, es decir, el directorio `/src/migrations` está vacío.

No se pueden usar ni anotaciones ORM ni tampoco anotaciones de ORM mediante atributos. Por tanto, el uso de entidades queda un poco inservible, ya que el quid de todo esto es poder ejecutar cambios automáticamente por medio de las migraciones. Al no disponer de esta funcionalidad, ha sido necesario crear los scripts en SQL de creación de BD y poblado: `schema.sql` y `seed.sql`, respectivamente. Estos ficheros se localizan en la carpeta `/src/database`. Tras crearlos y ejecutarlos se observó a través de DB que todo había ido bien (véase figura 3.15).

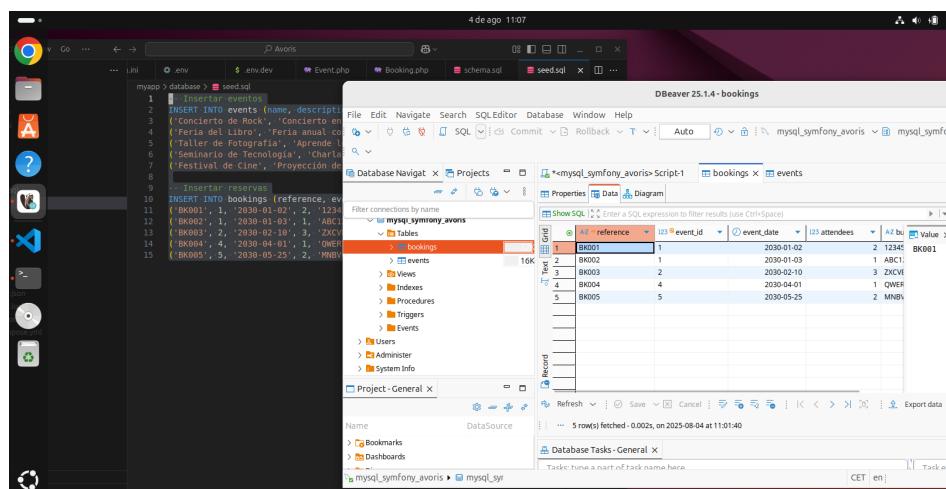


Figura 3.15: Poblado de la BD de MySQL 8.0.23 usando DBeaver

## Adaptadores de escritura y lectura

El siguiente paso ha consistido en desacoplar aún más la arquitectura hexagonal de EventBook configurando adaptadores de lectura y escritura para conectar con la BD.

Este matiz es un gran aliciente de funcionalidad, debido a que garantiza aún más la separación entre persistencia y aplicación. Paralelamente, si en algún momento del futuro se quisiese actualizar la versión de Symfony o de MySQL, no habría que tocar ningún aspecto de sendas lógicas, primero porque cada una está almacenada en un contenedor Docker distinto, y en segundo lugar, porque la creación de adaptadores permite, precisamente, que el único punto de conexión entre ambos servicios sean ellos mismos: los adaptadores.

La elaboración de los adaptadores ha consistido en la implementación de una clase llamada `ConnectionManager` que se ejecuta al momento de arrancar los contenedores una única vez. De esta forma, se asegura que la conexión es recicitable y que no se están creando nuevas instancias cada vez que se hacen operaciones sobre ella.

Paralelamente, en el fichero `/config/services.yml` (véase sección `parameters`) se definen los valores de conexión necesarios para poder establecer comunicación con el contenedor, mientras que en la sección inferior (véanse fragmentos `PDO.read` y `PDO.write`) se definen los adaptadores como tal. Nótese que, al estar en entorno de desarrollo (`dev`), ambos adaptadores trabajan sobre la misma instancia. Sin embargo, si esto se llevase a entornos de test y producción, habría que configurar entornos diferentes y mecanismos de replicación entre los nodos para garantizar consistencia entre lectura y escritura. Para este caso tan cerrado se ha considerado que así era suficiente, ya que permitía demostrar la idea.

## Creación de los repositorios

El último punto de esta minisección se ha reservado para la orquestación de los repositorios. Gracias a estas clases, los servicios comentados en la capa de infraestructura pueden comunicarse haciendo uso de los adaptadores con la BD.

Cabe destacar que los repositorios se crearon al mismo tiempo que el resto de módulos que componen el área de infraestructura, pero se han explicado en este apartado por su intrínseca relación con la gestión de adaptadores y BD. Así las cosas, se han creado dos únicos repositorios almacenados en el directorio `/src/Repository`. La siguiente tabla muestra los repositorios en cuestión, así como la finalidad de cada uno.

Nombre del repositorio DTO	Funcionalidad que desempeña
BookingRepository	Gestión de reservas
EventRepository	Gestión de eventos

Tabla 3.4: Repositorios creados para el proyecto

La comunicación entre los repositorios y los adaptadores se ha realizado de un modo tan simple como invocando a la clase `ConnectionManager` e inyectándola como dependencia directa en los constructores. De este modo, mediante dos propiedades internas, los repositorios tienen vinculados sendos adaptadores de forma disjunta (véase el código para más información).

Otra decisión de diseño adoptada ha sido que las operaciones de BD de tipo proyección o `SELECT` se ejecuten exclusivamente sobre el adaptador de lectura, mientras que las operaciones de modificación de datos: `CREATE`, `UPDATE` y `DELETE`, se realicen a través del adaptador de escritura. Esta separación refuerza el desacoplamiento de la arquitectura y mejora su escalabilidad.

Para finalizar este apartado, es importante destacar que existen operaciones, como la creación y cancelación de reservas, que no solo actualizan datos de una tabla, sino de dos. Para garantizar la cohesión e integridad entre ambas, se ha asegurado que dichas operaciones se ejecuten dentro de un ámbito atómico, haciendo uso de los métodos `beginTransaction`, `commit` y `rollback`.

Estas invocaciones se han implementado en los servicios, y no en los repositorios, ya que los servicios son los únicos componentes que tienen acceso a ambos adaptadores. Los repositorios, en cambio, únicamente deben gestionar el acceso a sus propios datos y no coordinar operaciones que involucren múltiples fuentes de persistencia (véanse los servicios para más detalle).

### 3.4.2. Configuración de Redis 7.0

En relación al apartado de persistencia, el último punto pasa por explicar cómo se llevó a cabo la integración de esta tecnología. Para ello, es necesario retonar de nuevo al fichero `docker-compose.yml`.

## Estructura del archivo docker-compose.yml

### Docker Compose YAML — Redis

```
avoris-php84-symfony62-redis:
  container_name: avoris-php84-symfony62-redis
  image: redis:7.0-alpine
  command: redis-server --appendonly yes --appendfsync everysec
    ↳ --loglevel warning
  ports:
    - "6379:6379"
  volumes:
    - ./redis-data:/data
  networks:
    - avoris-php84-symfony62-network
```

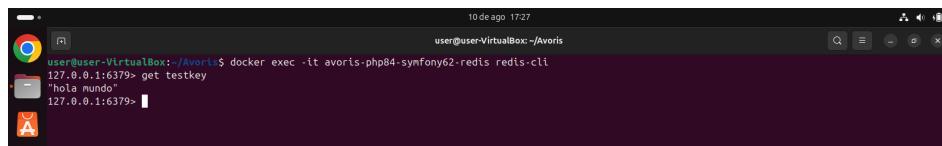
- **Servicio** `ontainer_name`: define el servicio de Redis y fija el nombre del contenedor para facilitar operaciones Docker (logs, exec, etc.).
- **image**: `redis:7.0-alpine` usa la variante alpine (imagen ligera) de Redis 7.0.
- **command**:
  - `-appendonly yes`: habilita AOF (Append Only File) para persistencia de todas las operaciones de escritura en `appendonly.aof`.
  - `-appendfsync everysec`: realiza un fsync cada segundo (compromiso entre durabilidad y rendimiento y menor latencia que `always`).
  - `-loglevel warning`: reduce el ruido de logs dejando sólo advertencias y errores.
- **ports**: mapea el puerto 6379 del contenedor al 6379 del host, permitiendo conexiones desde el host o servicios externos. (En entornos de producción es habitual evitar exponer el puerto públicamente y usar red interna).
- **volumes**: `./redis-data:/data` persiste los datos AOF en el host. Es importante para que los datos sobrevivan reinicios del contenedor.
- **networks**: conecta el contenedor a la red Docker definida con nombre (`avoris-php84-symfony62-network`) para comunicarse con los demás servicios del stack.

## Testeo del funcionamiento de Redis

Tras completar la configuración y reiniciar todos los contenedores, se procedió a verificar tanto la accesibilidad del servicio como el correcto funcionamiento del protocolo de persistencia configurado.

Desde el directorio /Avoris, se estableció conexión con el contenedor de Redis y, una vez dentro, se insertó un ejemplo de clave denominada “Hola Mundo”. Posteriormente, se reiniciaron los contenedores y se volvió a establecer conexión. Al consultar las claves almacenadas, se comprobó que “Hola Mundo” seguía presente (véase figura 3.16).

Cabe recordar que, por defecto, Redis almacena los datos únicamente en memoria, lo que implica que el contenido se pierde al reiniciar o detener el contenedor. Sin embargo, en este caso, la persistencia de la clave tras el reinicio confirma que la configuración establecida, descrita previamente, está operativa y que la integración con el servicio se ha realizado correctamente.

A screenshot of a terminal window titled "user@user-VirtualBox:~/Avoris\$". The window shows the command "docker exec -it avoris-php84-symfony62-redis redis-cli" being run. Below it, the output of the "get testkey" command is shown: "127.0.0.1:6379> get testkey" followed by the response "holá mundo". The timestamp at the top right of the terminal window is "10 de ago 17:27".

```
user@user-VirtualBox:~/Avoris$ docker exec -it avoris-php84-symfony62-redis redis-cli
127.0.0.1:6379> get testkey
"holá mundo"
127.0.0.1:6379>
```

Figura 3.16: Testeo de la conexión y persistencia de Redis

## Uso del sistema de cacheo en Redis

Para concluir este apartado, se ha incorporado el servicio de caché de Redis en el código de los repositorios, cuyas especificaciones se detallaron previamente.

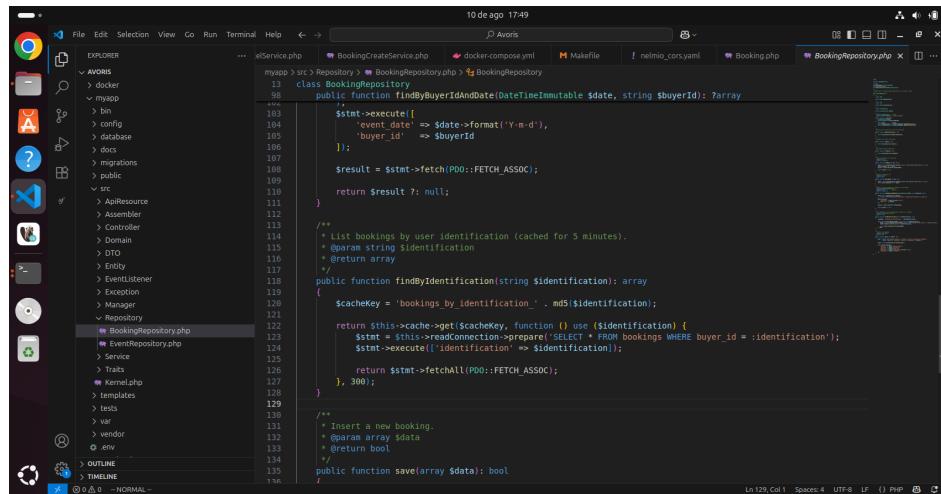
En coherencia con la estrategia seguida para la integración de los adaptadores de lectura y escritura del ConnectionManager, la inclusión de Redis se ha realizado mediante inyección de dependencias en los constructores de cada repositorio. De este modo, los repositorios pueden acceder al servicio de caché de forma desacoplada y centralizada, favoreciendo la reutilización y manteniendo la consistencia en la arquitectura (véase el código para más detalle).

Este enfoque permite mejorar el rendimiento general de la aplicación al reducir la carga sobre la base de datos para consultas frecuentes o costosas, almacenando sus resultados en caché. De esta forma, se minimizan los tiempos de respuesta y se incrementa la escalabilidad, anticipando el crecimiento futuro de la aplicación.

De acuerdo con la funcionalidad a diseñar, se optó por analizar qué puntos requerirían optimización en caso de un crecimiento deliberado de la APP. La conclusión fue centrarse en la proyección de reservas de un cliente mediante su DNI, así como en el listado de eventos, independientemente de los filtros de búsqueda aplicados.

Se implementó un mecanismo de cacheo con Redis para almacenar temporalmente los resultados de consultas. La primera ejecución de la consulta accede a la base de datos, y su resultado se guarda en la caché con una expiración de 300 segundos (véase figura 3.17). Durante ese período, futuras solicitudes recuperan los datos desde la caché, mejorando la eficiencia y reduciendo la carga en la base de datos.

Tras la expiración, la caché se invalida y la consulta vuelve a ejecutarse contra la base de datos para refrescar la caché. El sistema de caché emplea un esquema clave-valor donde la clave es un hash generado a partir de la consulta.



```
10 de ago 17:49
File Edit Selection View Go Run Terminal Help <- > Avisos
... \eService.php BookingCreateService.php docker-compose.yml Makefile nelmio_cors.yaml Booking.php BookingRepository.php ...
myapp > src > Repository > BookingRepository.php
13 98
class BookingRepository
14 public function findByBuyerIdAndDate(DateTimeImmutable $date, string $buyerId): ?array
15 {
16     $stmt = $this->execute([
17         'event_date' => $date->format('Y-m-d'),
18         'buyer_id' => $buyerId
19     ]);
20
21     $result = $stmt->fetch(PDO::FETCH_ASSOC);
22
23     return $result ?: null;
24 }
25
26 /**
27 * List bookings by user identification (cached for 5 minutes).
28 * @param string $identification
29 * @return array
30 */
31 public function findByIdentification(string $identification): array
32 {
33     $cacheKey = 'bookings_by_identification_' . md5($identification);
34
35     return $this->cache->get($cacheKey, function () use ($identification) {
36         $stmt = $this->readConnection->prepare("SELECT * FROM bookings WHERE buyer_id = :identification");
37         $stmt->execute([':identification' => $identification]);
38
39         return $stmt->fetchAll(PDO::FETCH_ASSOC);
40     }, 300);
41 }
42
43 /**
44 * Insert a new booking.
45 * @param array $data
46 * @return bool
47 */
48 public function save(array $data): bool
49 {
```

Figura 3.17: Ejemplo de caché de Redis en código

# Capítulo 4

## Documentación de EventBook

Una vez finalizada toda la parte de implementación y diseño, se ha procedido con la documentación de todo el código, decisiones de diseño, lógica de negocio, scripts de base de datos, y en última instancia, la propia API diseñada. El proceso de documentación ha pasado por las etapas adyacentes.

### 4.1. Código PHP

La primera parte que se abordó fue comentar todos los elementos que componen la capa de infraestructura y persistencia. Por consiguiente, se han documentado los DTOs, los assemblers, los repositorios de base de datos, los servicios que contienen la lógica de negocio, así como los endpoints que constituyen los casos de uso del software diseñado.

La documentación del código PHP se ha realizado por medio del estándar de PHPDoc. Además, se han reformateado todos los ficheros programados para que cumplan las directivas y especificaciones del PSR-12.

### 4.2. Ficheros SQL

Seguidamente, se documentó el contenido de los ficheros SQL creados para la definición y el poblado de la base de datos. Se elaboró un archivo detallado en formato Markdown que recoge las especificaciones y decisiones de diseño adoptadas, incluyendo tablas, columnas, tipos de datos, restricciones y claves foráneas. Este documento, denominado README.md, se encuentra ubicado en la carpeta /database, junto con los scripts correspondientes.

## 4.3. Lógica de negocio

Finalizada la documentación de la capa de persistencia, se documentó exhaustivamente la capa de implementación, abarcando la lógica de negocio de los seis servicios desarrollados. Para ello, se creó un archivo `markdown` por servicio y un índice como menú para facilitar la navegación. Toda la documentación se encuentra en la carpeta `/docs/resources`.

## 4.4. API interactiva con Swagger UI

Tras completar las etapas anteriores, se implementó una documentación interactiva para la API diseñada utilizando Swagger, la plataforma líder para crear documentaciones visuales y detalladas de APIs. Para ello, se accedió al repositorio oficial de Swagger UI en <https://github.com/swagger-api/swagger-ui>, y desde la sección Releases se descargó la última versión estable, la 5.27.1.

Una vez descargado y descomprimido el archivo `zip`, se copió el contenido de la carpeta `swagger` al directorio `public` del proyecto. A continuación, se editó el fichero `openapi.yml` para introducir la documentación de la API. Se configuró la propiedad `server` con el valor `http://localhost:1000/`, permitiendo que la interfaz gráfica de Swagger realice peticiones directas a la API para facilitar las pruebas. Finalmente, tras reiniciar los contenedores, se pudo acceder a la documentación interactiva a través de `http://localhost:1000/swagger/` (véase figura 4.1).

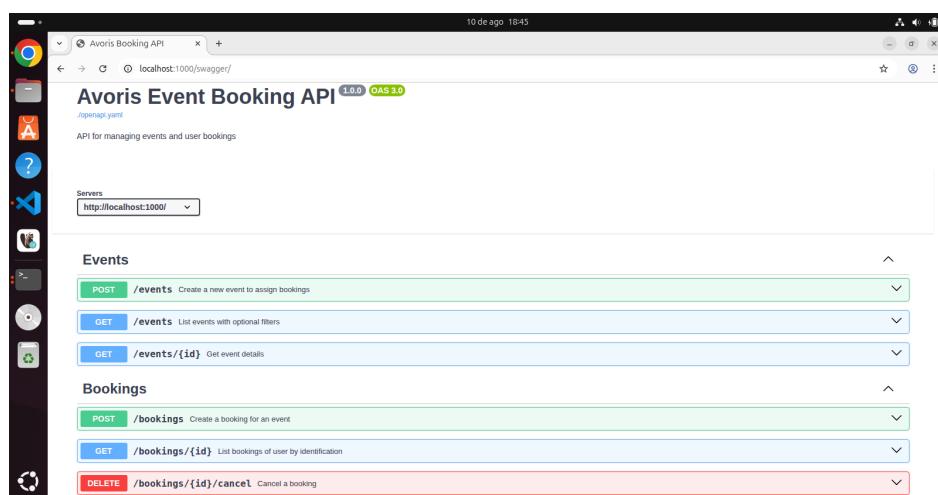


Figura 4.1: Documentación interactiva de la API con Swagger

Paralelamente, también se realizaron algunas pruebas de endpoint a través de la interfaz para ver que la comunicación entre ambos servicios funcionaba correctamente (véanse las figuras 4.2 y 4.3).

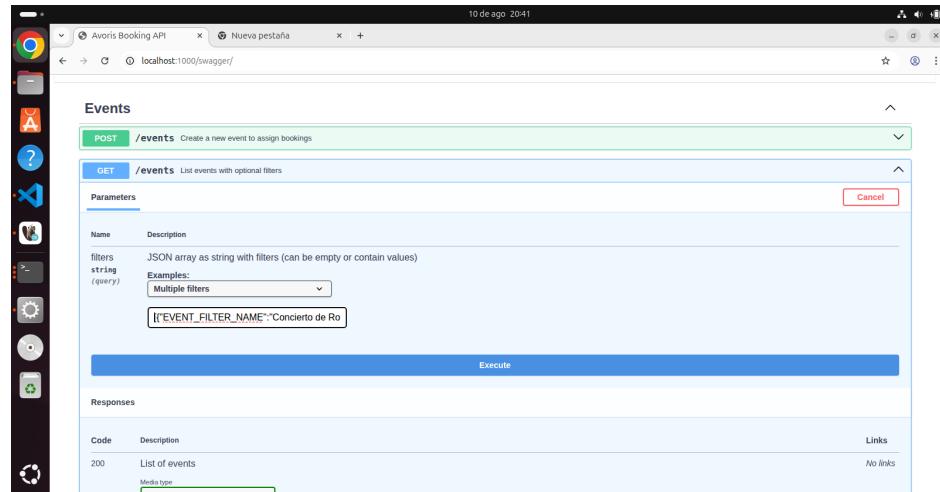


Figura 4.2: Ejemplo de petición API con Swagger

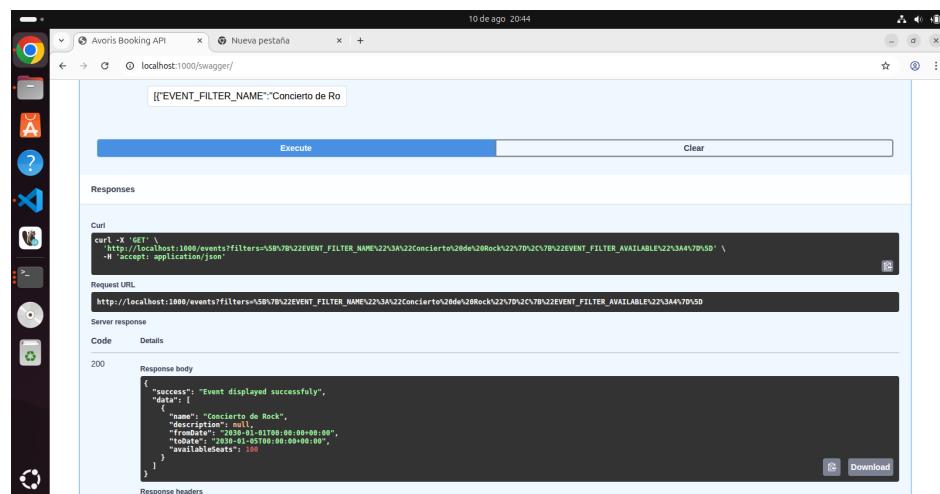


Figura 4.3: Ejemplo de respuesta API con Swagger

# Capítulo 5

## Testing de EventBook

El último paso en el diseño de EventBook ha sido la implementación de una exhaustiva batería de tests para validar y depurar las distintas áreas desarrolladas. Se han cubierto pruebas unitarias, de integración y de aplicación, todas localizadas en el directorio /tests del proyecto principal.

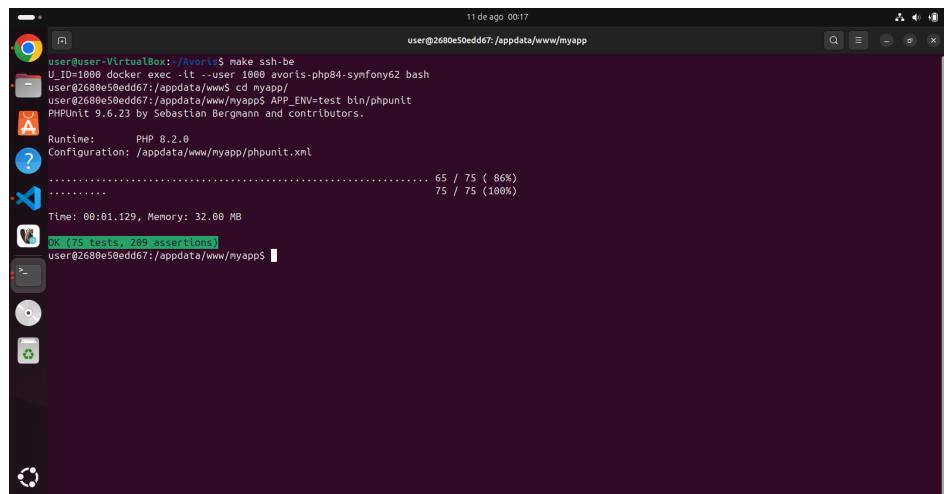
Es importante remarcar que, en el caso de pruebas de integración o de aplicación, no se ha trabajado con datos reales, sino con simulaciones mediante mockups. Esta decisión obedece a las buenas prácticas, que recomiendan la separación estricta de entornos prod, test y dev. Aunque el código base es el mismo, cada entorno dispone de configuraciones independientes: librerías, sistema de caché y ficheros de entorno (.env), adaptadas a sus necesidades.

La ausencia de un entorno de pruebas plenamente configurado, unida a las particularidades de Symfony y PHPUnit, complica evitar conflictos de cacheo y errores de ejecución. Por este motivo, se ha optado por el uso de mocks y por establecer manualmente la variable de entorno APP\_ENV con el valor test, asegurando así la correcta ejecución de las pruebas. Para la instalación de PHPUnit se han ejecutado los siguientes comandos:

```
composer require --dev phpunit/phpunit:^9.5
ln -s "$(pwd)/vendor/bin/phpunit" bin/phpunit
ls -l bin/phpunit
chmod +x vendor/bin/phpunit
```

Una vez instalado, para ejecutar los tests basta con situarse en el directorio raíz del proyecto Symfony y lanzar el siguiente comando (estableciendo el entorno de pruebas):

```
APP_ENV=test bin/phpunit
```



A screenshot of a terminal window titled "user@2680e50edd67:/appdata/www/myapp". The window shows the output of a PHPUnit test run. The command "make ssh-be" was run, which executed "ssh-be" and then "phpunit". The output indicates a successful run with 75 assertions passed (100%) in 00:01.129, using 32.08 MB of memory. The configuration file used is "/appdata/www/myapp/phpunit.xml".

```
user@user-VirtualBox:/Avoris$ make ssh-be
user@2680e50edd67:/appdata/www$ cd myapp/
user@2680e50edd67:/appdata/www/myapp$ APP_ENV=test bin/phpunit
PHPUnit 9.6.23 by Sebastian Bergmann and contributors.

Runtime:      PHP 8.2.0
Configuration: /appdata/www/myapp/phpunit.xml

..... 65 / 75 ( 86%)
..... 75 / 75 (100%)

Time: 00:01.129, Memory: 32.08 MB
25/25 assertions
user@2680e50edd67:/appdata/www/myapp$
```

Figura 5.1: Ejecución de los tests con PHPUnit

# Capítulo 6

## Problemas encontrados

Durante el desarrollo del proyecto, se presentaron diversas dificultades. A continuación, se detallan los principales problemas afrontados.

### 6.1. Incompatibilidad entre PHP y Symfony

Un desafío muy relevante fue la incompatibilidad entre las versiones de PHP y Symfony. Inicialmente, el contenedor utilizado ejecutaba PHP 8.4, con la intención de aprovechar su versión más reciente. Sin embargo, surgieron múltiples conflictos, especialmente relacionados con la gestión de errores y advertencias de deprecación y warnings. Estas discrepancias derivaban en problemas al modificar el archivo `php.ini`, generando errores críticos en la ejecución. Finalmente, se optó por utilizar PHP 8.2, versión compatible y estable con Symfony 6.2, lo que resolvió las incompatibilidades. Nótese que los contenedores y redes Docker llevan el alias `php84` y no `php82`, a modo de recordatorio de algo que nunca pudo ser.

### 6.2. Configuración de Swagger UI

La integración de la documentación interactiva de la API presentó dificultades significativas. El método inicialmente previsto, utilizando API Platform o el bundle `NelmioApiDocBundle` para Swagger, generaba errores relacionados con dependencias y problemas de caché en `composer.json`. Tras varios intentos fallidos, fue necesario purgar y reconstruir los contenedores, implementando finalmente la solución manual descrita en la sección de documentación, que permitió una correcta integración de Swagger UI.

### **6.3. Realización de tests automatizados**

El desarrollo de pruebas automatizadas también constituyó un reto considerable, debido a la falta de recursos adecuados para su ejecución en condiciones óptimas. En consecuencia, se optó por la simulación mediante `mockups`, lo que implicó un proceso complejo de adaptación y aprendizaje. Para ello, se recurrió a la documentación oficial y al apoyo de herramientas como ChatGPT para resolver las incidencias que surgían durante la implementación de los tests.

### **6.4. Comunicación con la base de datos MySQL 8.0.26**

Este problema resultó especialmente relevante debido a que la configuración inicial de la base de datos estaba correctamente realizada. Sin embargo, surgieron dificultades para establecer una conexión desde DBeaver. La conexión directa al contenedor funcionó sin inconvenientes, pero al intentar conectar mediante la interfaz gráfica de DBeaver, no se lograba el acceso y la causa no era inicialmente evidente.

Tras analizar y modificar diversos parámetros de configuración, se identificó que el origen del problema residía en el puerto utilizado. Concretamente, DBeaver estaba intentando conectar a la base de datos utilizando el puerto interno del contenedor, válido únicamente para la red interna de Docker, en lugar del puerto remapeado expuesto en el host. Ajustando la conexión para usar el puerto del host se resolvió la incidencia, permitiendo una comunicación adecuada con la base de datos.