



# Agenda cultural de Euskadi

Administración de Sistemas

4º Curso

Grado en Ingeniería Informática de Gestión y Sistemas  
de Información

Francisco Fernández Condado

24 de noviembre de 2024

# Índice general

<b>1. Introducción</b>	<b>2</b>
<b>2. Listado de tareas realizadas</b>	<b>4</b>
<b>3. Instalación y despliegue del proyecto</b>	<b>5</b>
3.1. Despliegue mediante Docker . . . . .	6
3.1.1. Imagen <i>updater</i> . . . . .	9
3.1.2. Imagen <i>flask_app</i> . . . . .	9
3.2. Despliegue mediante Kubernetes . . . . .	9
<b>4. Manual de uso y funcionalidades disponibles</b>	<b>14</b>
<b>5. Declaración sobre el uso de asistentes virtuales</b>	<b>16</b>
<b>6. Bibliografía</b>	<b>17</b>

# Introducción

Este proyecto propone la creación de una aplicación web funcional basándose en los contenedores de Docker y, de forma opcional, realizar el despliegue equivalente de Kubernetes.

En este caso, la idea es crear un portal web interactivo que permita consultar la Agenda Cultural de Euskadi con actualizaciones en tiempo real. Para ello, se ha hecho uso de la información proporcionada en el portal Open Data del Servicio web del Gobierno Vasco, que provee diferentes formas de acceso a la información completa.

Se ha elegido hacer uso de un servidor web basado en nginx, que se encargará de redirigir las peticiones a un servidor Flask. La información de los eventos queda registrada en una base de datos PostgreSQL y, mediante un entorno adicional basado en Python, se actualizan los datos de los eventos a partir de la API de Open Data para contar con actualizaciones en tiempo real. Además, se cuenta con un entorno adicional que permite exportar los detalles de los eventos a un formato PDF, siendo más cómodo para imprimir o compartir en caso de ser necesario.

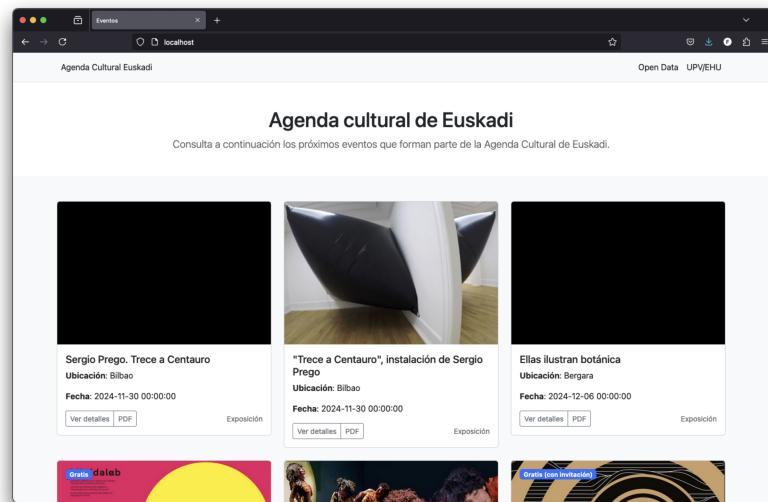


Figura 1.1: Vista general de la aplicación

De este modo, la aplicación web permite acceder a un catálogo actualizado en tiempo real de los eventos presentes en la Agenda Cultural de Euskadi, con un diseño sencillo basado en la librería Bootstrap. Accediendo al detalle completo, será posible consultar toda la información (fechas, descripción, coste, categoría, lugar de realización, enlaces adicionales, etc.), así como a un mapa que muestra la ubicación en la que se realizará el evento.

El despliegue completo de la aplicación puede encontrarse en el siguiente repositorio de GitHub para su uso y descarga: [https://github.com/ffernandezco/as\\_proyecto](https://github.com/ffernandezco/as_proyecto).

# Listado de tareas realizadas

- Tareas obligatorias:

- Desarrollo de una aplicación funcional utilizando, al menos, un servidor web, un servidor de BBDD y una imagen a libre elección, basada en datos de repositorios públicos.
- Creación de las imágenes Docker necesarias para el proyecto, subiéndose a un repositorio público de Docker Hub.
- Creación de un entorno Docker Compose que permite ejecutar la aplicación desarrollada.

- Tareas adicionales:

- Crear un despliegue Kubernetes que permita ejecutar la aplicación desarrollada.
- Inclusión de imágenes o contenedores adicionales que aporten nuevas funcionalidades a la aplicación sobre Docker Compose o Kubernetes.
- Uso de funcionalidades de Kubernetes no vistas en clase.

# Instalación y despliegue del proyecto

El proyecto cuenta con un total de 6 servicios, que deben ser desplegados de forma conjunta con el fin de garantizar todas las funcionalidades disponibles:

- **Servicio *nginx*:** propone el uso de un servidor *nginx* con una configuración personalizada que redirige las peticiones al servicio Flask, sobre el que se hablará más adelante. Se basa en la última versión disponible en Docker Hub de la imagen: [https://hub.docker.com/\\_/nginx](https://hub.docker.com/_/nginx).
- **Servicio *postgres*:** construye un servidor de bases de datos *PostgreSQL*, al que posteriormente se accederá mediante Flask para mostrar la información de los eventos, así como por el contenedor Python encargado de actualizar la información y por el gestor de bases de datos. Se basa en la última versión disponible en Docker Hub de la imagen: [https://hub.docker.com/\\_/postgres](https://hub.docker.com/_/postgres).
- **Servicio *adminer*:** aunque no aporta ninguna funcionalidad útil de cara al público, el gestor Adminer permite de una forma sencilla gestionar las bases de datos del contenedor PostgreSQL mencionado anteriormente directamente desde una interfaz web basada en PHP. Se basa en la última versión disponible en Docker Hub de la imagen: [https://hub.docker.com/\\_/adminer/](https://hub.docker.com/_/adminer/).
- **Servicio *updater*:** propone un contenedor basado en la última versión de la imagen de Python que se encargará de ejecutar un script que, a través de la API de Open Data Euskadi, obtendrá los registros más recientes y actualizará la base de datos PostgreSQL mencionada anteriormente incluyendo los registros más recientes. Se trata de una imagen personalizada que, frente a la oficial, incluye la instalación de las librerías necesarias para que el script funcione, así como un *cron* personalizado, que se encarga de comprobar si hay actualizaciones disponibles cada un minuto. Se puede consultar la imagen de forma pública en Docker Hub: <https://hub.docker.com/r/franciscofdez/updater-amd64>.
  - La imagen fue creada originalmente en la arquitectura ARM, contando con disponibilidad limitada al ejecutarse en entornos AMD64: <https://hub.docker.com/r/franciscofdez/updater>

- Para ejecutarse en un entorno Kubernetes, se ha creado la siguiente imagen dedicada, que en este caso está optimizada para ser instanciada mediante un *Cronjob* y, por tanto, son necesarios menos pasos para generarla: <https://hub.docker.com/r/franciscofdez/events-updater-2>
- **Servicio flask:** construye un segundo contenedor personalizado, de nuevo basado en la última versión de la imagen de Python, que se encargará de proporcionar a nginx la salida necesaria según la consulta realizada sobre el puerto 80. En este caso, se ha optado por una imagen personalizada que instala las dependencias necesarias, incluyendo el motor Flask, y que cuenta con un fichero *app.py* que se encargará de gestionar las peticiones, bien por consultas a la base de datos PostgreSQL y ofreciendo salidas a través de las plantillas diseñadas, o bien realizando una solicitud al servicio Gotenberg que se expondrá a continuación. Se puede consultar la imagen de forma pública en Docker Hub: [https://hub.docker.com/r/franciscofdez/flask\\_app-amd64](https://hub.docker.com/r/franciscofdez/flask_app-amd64).
  - La imagen fue creada originalmente en la arquitectura ARM, contando con disponibilidad limitada al ejecutarse en entornos AMD64: [https://hub.docker.com/r/franciscofdez/flask\\_app](https://hub.docker.com/r/franciscofdez/flask_app)
  - Para ejecutarse en un entorno Kubernetes, se ha creado la siguiente imagen dedicada: <https://hub.docker.com/r/franciscofdez/flask-k8s>
- **Servicio gotenberg:** se trata de una funcionalidad adicional añadida, que permite obtener la información de un evento en formato PDF. El servicio se basa en el motor Gotenberg, que permite crear una API sencilla para obtener ficheros en formato PDF. Las consultas a dicha API estarán gestionadas por el servicio Flask mencionado anteriormente, permitiendo al usuario final descargar cualquier fichero generado si así lo desea. Se basa en la versión 8 de la imagen oficial disponible en Docker Hub: <https://hub.docker.com/r/gotenberg/gotenberg/>.

En dos directorios separados, *Docker* y *Kubernetes*, se han incluido los ficheros necesarios para realizar el despliegue mediante Docker (con su entorno Compose), así como en Kubernetes. A continuación se exponen instrucciones dedicadas para ambos servicios.

### 3.1. Despliegue mediante Docker

El fichero *docker-compose.yml* permite un despliegue automático a partir de las imágenes mencionadas anteriormente. Sin embargo, en primer lugar, será necesario crear un secreto, que se encargará de colocar la contraseña necesaria para la base de datos PostgreSQL. Dicho secreto será usado, además, por los contenedores de Flask y Python, que necesitarán acceder a la base de datos correspondiente.

Para crear el secreto, basta con crear un fichero nuevo con el nombre *postgres\_password* en el directorio *secrets*. Asumiendo que primero se ha clonado el repositorio y que se ha realizado la instalación de Docker y Docker Compose en el equipo, ejecutando los siguientes comandos se puede replicar el escenario:

```
$ cd Docker
$ mkdir secrets
```

```
$ echo "micontraseña" > ./secrets/postgres_password  
$ docker compose up --build
```

En concreto, el comando *docker compose up -build*, permitirá crear una instancia completa a partir del fichero *docker-compose.yml*. En la Tabla 3.1 se exponen los diferentes servicios que este fichero pone en marcha al iniciar el contenedor, incluyendo los puertos que pueden usarse para consultar los diferentes servicios disponibles. Mencionar en este caso que, por cómo se ha generado, es posible que la primera vez que se realice el despliegue, se tarde unos minutos en poder ver la página principal hasta que el *update* se encargue de obtener los datos de la API de Open Data Euskadi.

Una vez creado la primera vez, el despliegue pasa a ser más rápido, ya que además de los contenedores se ha definido el volumen *postgres\_data* para almacenar los valores de la base de datos, de tal forma que después el *update* solo se encarga de actualizar la información.

En este caso, todas las imágenes usadas están preparadas para poderse ejecutar de forma inmediata sin ningún fichero adicional, a excepción del contenedor *nginx*. Para este contenedor, en el fichero */nginx/nginx.conf* se ha definido la configuración que debe usar el servidor para obtener las solicitudes de Flask y dirigirlas al puerto 80. Así, al acceder a la dirección IP del servidor o, en su defecto, a *localhost*, cargará directamente la página principal.

Servicio	Imagen y descripción	Puerto original	Puerto expuesto
nginx	<p><b><i>nginx:latest</i></b></p> <p>Servidor basado en nginx. Redirige las peticiones solicitadas mediante la interfaz web al servicio correcto a partir del fichero <i>/nginx/nginx.conf</i>.</p>	80	80
postgres	<p><b><i>postgres:latest</i></b></p> <p>Servidor de base de datos basado en PostgreSQL, usado para almacenar la información de los eventos. Crea automáticamente un usuario y una base de datos con la contraseña definida en el secreto correspondiente.</p>	5432	5432
adminer	<p><b><i>adminer:latest</i></b></p> <p>Permite gestionar desde una interfaz visual la base de datos PostgreSQL mencionada anteriormente.</p>	8080	8081
updater	<p><b><i>francisofdez/updater-amd64:latest</i></b></p> <p>Contenedor basado en Python que realiza, de forma periódica, consultas a la API de Open Data Euskadi para descargar los eventos más recientes de la agenda cultural.</p>	—	—
flask	<p><b><i>francisofdez/flask_app-amd64:latest</i></b></p> <p>Contenedor basado en Python con Flask usado para mostrar los detalles de los eventos en base a plantillas diseñadas con Bootstrap y dirigir las solicitudes a los servicios adecuados.</p>	5000	5001
gotenberg	<p><b><i>gotenberg/gotenberg:8</i></b></p> <p>Permite, mediante una API, realizar múltiples acciones relacionadas con documentos PDF. Se usa para convertir ficheros HTML con información sobre los eventos en este formato.</p>	3000	3000

Cuadro 3.1: Servicios definidos en el despliegue mediante Docker Compose

En lo que respecta a las imágenes propias, su código fuente se encuentra en el directorio anexo *docker-images*, y pueden clonarse de forma directa al estar disponibles de forma pública en Docker Hub. A continuación se expone cómo han sido creadas y su funcionalidad en el proyecto:

### 3.1.1. Imagen *Updater*

El propósito de esta imagen es recoger, a partir de la API de Open Data Euskadi, los datos correspondientes a la agenda cultural del año en curso. Para gestionar esto, a partir de la imagen *python:latest*, se crea un contenedor en el que se instalan las dependencias *requests*, *psycopg2* y *datetime*, necesarias para ejecutar el código del programa, que se encuentra descrito en el fichero *main.py*.

Para gestionar la actualización automática, se cuenta con el siguiente fichero *crontab* personalizado, que se encarga de ejecutar el script cada minuto con el fin de comprobar si hay o no actualizaciones disponibles a partir de la API del Gobierno Vasco:

```
* * * * * root POSTGRES_PASSWORD_FILE=/run/secrets/postgres_password  
/usr/local/bin/python /app/main.py >> /var/log/cron.log 2>&1
```

Por la estructura del contenedor, se deben especificar las variables de entorno en el propio fichero del cron. Además, dejaremos el resultado en un fichero *cron.log* para poder depurar errores. Toda esta configuración se expresa en el fichero *Dockerfile* usado para crear la imagen.

### 3.1.2. Imagen *flask\_app*

La imagen personalizada *flask\_app* se encarga de controlar un servidor basado en Flask, que se usará de cara al frontend de la aplicación. Para ello, se parte de nuevo de la imagen *python:latest*. A continuación, se realiza la instalación de las dependencias *Flask*, *flask-cors*, *psycopg2* y *requests*. Estas dependencias serán usadas por el fichero *app.py*, que será el encargado de proporcionar las funcionalidades de Flask.

En el directorio */templates*, se cuenta con diferentes plantillas, que serán las que tome en cuenta Flask. Además, es el encargado de gestionar las consultas del frontend a la base de datos PostgreSQL, así como de las solicitudes a la API de Gotenberg en el caso de que el usuario solicite un documento PDF.

El fichero *Dockerfile* se encargará de gestionar todo esto, instalando todo lo necesario e iniciando *app.py* como es debido. Por cómo está construida la imagen, será necesario especificar en dicho fichero las variables de entorno que Flask necesitará más adelante.

## 3.2. Despliegue mediante Kubernetes

De cara al despliegue utilizando Kubernetes, la instancia no es tan rápida como mediante Docker, al ser necesario configurar cada uno de los objetos de tipo ConfigMap, Deployment, Service, CronJob e Ingress que se han implementado.

En este caso, se ha optado por realizar el despliegue en un entorno local, desplegado haciendo uso del motor de Kubernetes incluido en Docker Desktop, sobre el que puede

obtenerse más información en su sitio web oficial. Además, los accesos han sido configurados a partir de la red con subdominios del tipo *localdev.me* con propósitos de pruebas, si bien podría ajustarse para responder a otros dominios en un entorno basado en la nube.

El motivo principal por el que se ha optado por realizar el despliegue haciendo uso de un entorno local guarda relación con el coste de la implementación en otros entornos como *Google Kubernetes Engine*. Al hacer uso de un cluster de tipo *Autopilot*, que gestionará de forma automática los contenedores y la creación de las máquinas, y al tener en este caso 5 máquinas funcionando en todo momento y una que se ejecuta de forma periódica haciendo uso de un objeto *CronJob*, el coste podría dispararse. No obstante, en un entorno de producción, resulta una opción recomendable que facilita la implementación.

Teniendo en cuenta estos detalles, para poder comenzar será necesario crear un secreto que contenga el nombre de usuario de la base de datos PostgreSQL así como su contraseña. Para hacer esto, debemos ejecutar el siguiente comando en nuestro entorno, sustituyendo “*micontraseña*” por la contraseña deseada:

```
$ kubectl create secret generic postgres-secret \
$ --from-literal=POSTGRES_USER="as" \
$ --from-literal=POSTGRES_PASSWORD="micontraseña"
```

A continuación, para poder crear una instancia básica del proyecto, será necesario ir ejecutando cada uno de los siguientes comandos sobre el directorio, que se encargan de configurar cada uno de los ficheros creados y, además, configurará el controlador de nginx necesario para ejecutar correctamente la configuración del Ingress para la red:

```
$ kubectl apply -f postgres-configmap.yaml
$ kubectl apply -f postgres-pvc.yaml
$ kubectl apply -f postgres-deployment.yaml
$ kubectl apply -f postgres-service.yaml
$ kubectl apply -f adminer-deployment.yaml
$ kubectl apply -f adminer-service.yaml
$ kubectl apply -f updater-cronjob.yaml
$ kubectl apply -f gotenberg-deployment.yaml
$ kubectl apply -f gotenberg-service.yaml
$ kubectl apply -f flask-deployment.yaml
$ kubectl apply -f flask-service.yaml
$ kubectl apply -f nginx-configmap.yaml
$ kubectl apply -f nginx-deployment.yaml
$ kubectl apply -f nginx-service.yaml
$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/
ingress-nginx/controller-v1.11.3/deploy/static/provider/cloud/
deploy.yaml
$ kubectl apply -f ingress.yaml
```

Una vez realizado, ejecutando el comando *kubectl get pods* podrán verse los diferentes contenedores creados, mientras que *kubectl get jobs* devolverá las tareas del *updater* definido en el CronJob correspondiente, y *kubectl get ingress* mostrará la configuración

de la red. Si todo ha funcionado de forma correcta, debería mostrarse algo similar a lo expuesto en la Figura 3.1.



```
(base) francisco@francisco Kubernetes % kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
adminer-5bd749f5b7-pwkzv   1/1     Running   0          29h
adminer-deployment-555bbc688c-xn98l   1/1     Running   0          29h
events-updater-28865440-sdnv6   0/1     Completed  0          28m
events-updater-28865450-9hnql   0/1     Completed  0          18m
events-updater-28865460-mtqz5   0/1     Completed  0          8m42s
flask-deployment-5f66657b58-xjm85   1/1     Running   0          14h
gotenberg-deployment-6bffd95769-8znmw   1/1     Running   0          16h
nginx-deployment-b8d5dc4b-w6jm2   1/1     Running   0          14h
postgres-fd5ff55c-g6968   1/1     Running   0          29h
(base) francisco@francisco Kubernetes % kubectl get jobs
NAME            STATUS  COMPLETIONS DURATION   AGE
events-updater-28865440  Complete 1/1      29s       28m
events-updater-28865450  Complete 1/1      46s       18m
events-updater-28865460  Complete 1/1      32s       8m48s
(base) francisco@francisco Kubernetes % kubectl get ingress
NAME        CLASS      HOSTS                                     ADDRESS      PORTS  AGE
ingress    nginx     adminer.localdev.me,agendacultural.localdev.me,gotenberg.localdev.me  localhost  80     14h
(base) francisco@francisco Kubernetes %
```

Figura 3.1: Objetos instanciados por Kubernetes

A continuación se expone cada uno de los ficheros de configuración que han sido utilizados para lograr el despliegue equivalente en Kubernetes, explicando su funcionalidad:

■ **Servidor PostgreSQL:**

- ***postgres-configmap.yaml*:** define las variables necesarias para el servidor de la base de datos PostgreSQL, encargándose de crear una base de datos con el nombre *”database”* de definir *”as”* como el nombre de usuario por defecto.
- ***postgres-pvc.yaml*:** se encarga de realizar la reclamación de volumen persistente para el servidor de la base de datos.
  - **Modo de acceso:** se ha definido como *ReadWriteOnce* para permitir el funcionamiento correcto de PostgreSQL.
  - **Almacenamiento:** se ha definido 1 GB de espacio en el campo *storage*, que debería ser suficiente para almacenar la base de datos completa.
- ***postgres-deployment.yaml*:** define el funcionamiento general del servidor PostgreSQL, de acuerdo a las características detalladas a continuación.
  - **Imagen base:** postgres:latest
  - **Número de réplicas:** 1
  - Utiliza el secreto creado anteriormente como contraseña para la base de datos generada, así como la reclamación de volumen del campo anterior para almacenar la información.
- ***postgres-service.yaml*:** gestiona el servicio a través del que será posible acceder al contenedor.

- **Puerto de origen:** 5432
- **Puerto de salida:** 5432

▪ **Servicio Adminer:**

- ***adminer-deployment.yaml*:** gestiona el funcionamiento general del servicio Adminer, usado para gestionar visualmente la base de datos. Se basa en las características detalladas a continuación.
  - **Imagen base:** adminer:latest
  - **Número de réplicas:** 1
  - Define el servidor por defecto de la base de datos como *postgres-service*, ruta definida por el servicio PostgreSQL.
- ***adminer-service.yaml*:** se encarga de gestionar el servicio a través del que será posible acceder al contenedor.
  - **Puerto de origen:** 8080
  - **Puerto de salida:** 30001
  - *Se ha modificado el puerto de salida con respecto al despliegue de Docker por no dar problemas de compatibilidad con el despliegue de Kubernetes.*

▪ **Servicio Updater**

- ***updater-cronjob.yaml*:** define un objeto de tipo *Cronjob, funcionalidad de Kubernetes no vista en clase*, que se encargará de instanciar, cada 10 minutos, el servicio de actualización de datos de acuerdo con las siguientes configuraciones.
  - **Imagen base (propia):** franciscofdez/events-updater-2:latest. Se utiliza una imagen diferente a la de Docker pero con la misma funcionalidad, debido a que no es necesario configurar un *cronjob* al usar, en su lugar, la funcionalidad de Kubernetes, así como para adaptarse a la forma de gestionar los secretos.
  - Utiliza como parámetros los datos de la base de datos del servicio PostgreSQL para conectarse, así como el secreto configurado previamente.

▪ **Servicio Gotenberg**

- ***gotenberg-deployment.yaml*:** configura el funcionamiento general del servicio Gotenberg, usado para la generación de documentos PDF, de acuerdo con las características detalladas a continuación.
  - **Imagen base:** gotenberg/gotenberg:8
  - **Número de réplicas:** 1
- ***gotenberg-service.yaml*:** se encarga de gestionar el servicio a través del que será posible acceder al contenedor.
  - **Puerto de origen:** 3000
  - **Puerto de salida:** 30001
  - *Se ha modificado el puerto de salida con respecto al despliegue de Docker por no dar problemas de compatibilidad con el despliegue de Kubernetes.*

## ■ Servicio Flask

- **flask-deployment.yaml:** objeto encargado de gestionar el contenedor, basado en Flask, que se mostrará en el frontend de la aplicación. Se encarga de mostrar, en base a las plantillas ya creadas, el contenido de la base de datos, o de llamar al servicio Gotenberg cuando un usuario solicita la creación de un documento PDF.
  - **Imagen base (propia):** franciscofdez/flask-k8s:latest. Se utiliza una imagen diferente a la de Docker pero con la misma funcionalidad, debido a cómo funciona el entorno Python en Kubernetes, así como para adaptarse a la forma de gestionar los secretos.
  - **Número de réplicas:** 1
  - Utiliza como parámetros los datos de la base de datos del servicio PostgreSQL para conectarse, así como el secreto configurado previamente.
- **flask-service.yaml:** tiene como finalidad gestionar el servicio a través del que será posible acceder al contenedor Flask a partir de nginx o de forma directa.
  - **Puerto de origen:** 5000
  - **Puerto de salida:** 30002
  - *Se ha modificado el puerto de salida con respecto al despliegue de Docker por no dar problemas de compatibilidad con el despliegue de Kubernetes.*

## ■ Servicio nginx

- **nginx-configmap.yaml:** se encarga de configurar los parámetros necesarios de cara al servidor nginx. Contiene los mismos datos almacenados previamente en el fichero *nginx.conf* de Docker.
- **nginx-deployment.yaml:** objeto encargado de gestionar el servidor nginx, que se usará para realizar consultas al contenedor de Flask cuando sea necesario.
  - **Imagen base:** nginx:latest.
  - **Número de réplicas:** 1
- **nginx-service.yaml:** se encarga de exponer el servidor nginx al exterior, usando para ello el puerto 80 por ser el predefinido para tráfico web HTTP.
  - **Puerto de origen:** 80
  - **Puerto de destino:** 80

## ■ Configuración de red

- **ingress.yaml:** objeto encargado de gestionar la conectividad de los diferentes contenedores. Está optimizado para desarrollo local, por lo que se utilizan los siguientes dominios.
  - *adminer.localdev.me*: apunta al servicio Adminer.
  - *agendacultural.localdev.me*: apunta al servicio Flask de forma directa, cargando la página principal.
  - *gotenberg.localdev.me*: apunta directamente al servicio Gotenberg, pudiendo usarse a partir de su API propia.

# Manual de uso y funcionalidades disponibles

Accediendo a la página principal, se mostrará automáticamente un listado completo de los eventos de la Agenda Cultural de Euskadi. Como se puede ver en la Figura 4.1 cada uno de ellos, se verán algunos datos básicos, incluyendo el tipo, el nombre, la fecha, el lugar o una imagen.

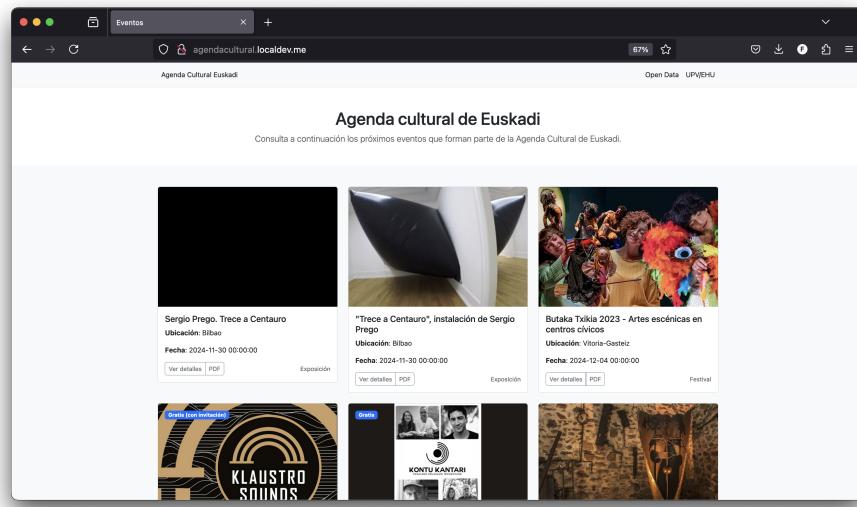


Figura 4.1: Página principal de la aplicación web

Pulsando el botón de “Ver detalles”, será posible acceder a la página de detalles de los eventos, tal y como se muestra en la Figura 4.2. El contenido depende en este caso de la información proporcionada por la API de Open Data Euskadi para el evento en cuestión. Si contiene campos como el precio o un enlace recomendado se mostrarán. Además, de cara a los eventos con una localización asociada, se mostrará el mapa correspondiente, basado en los mapas de OpenStreetMap y Leaflet para su consulta.

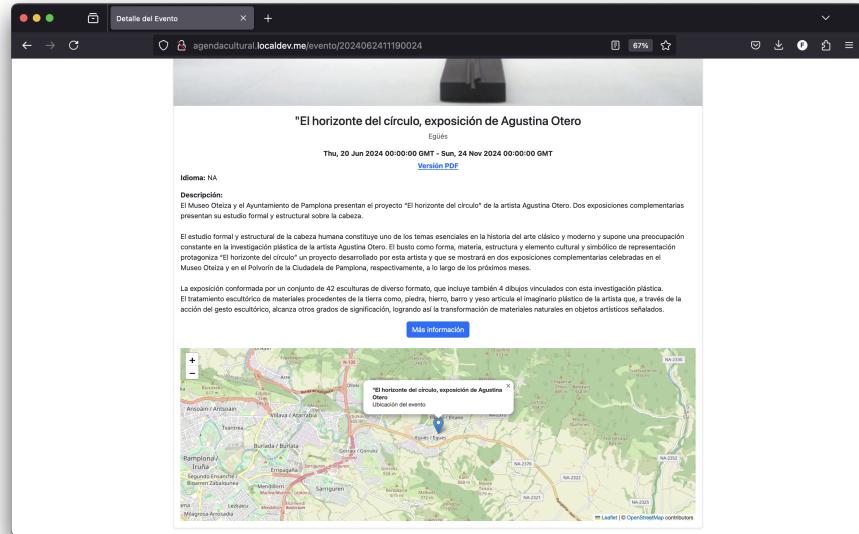


Figura 4.2: Ejemplo de detalle de un evento

Por último, al presionar sobre el botón “PDF” de la página de inicio, o al presionar sobre el enlace “Versión PDF” dentro del detalle de un evento, se genera un documento PDF que, tal y como describe la Figura 4.3, muestra todos los detalles del evento, facilitando compartirlo o la impresión.

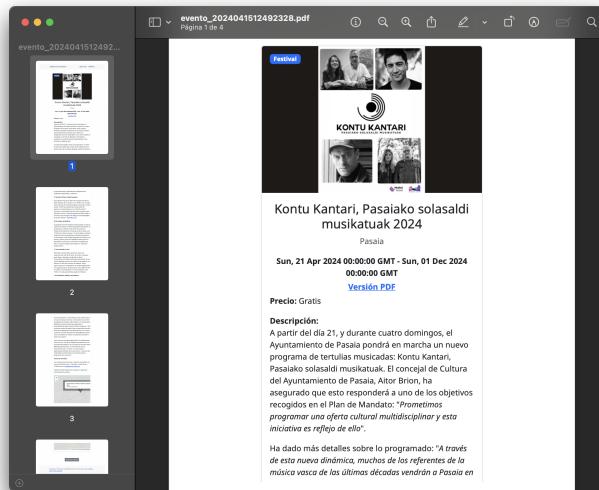


Figura 4.3: Ejemplo de documento PDF generado por la aplicación

# **Declaración sobre el uso de asistentes virtuales**

Para la realización de este proyecto, se han utilizado en combinación los asistentes virtuales ChatGPT (OpenAI), Gemini (Google) y Copilot (Microsoft). No obstante, indicar en este caso que no han sido de ayuda en todo momento. Por ejemplo, la configuración de la tarea cron a partir de la imagen de Python de Docker ha tenido que crearse de forma manual a través de la información de múltiples repositorios de GitHub y preguntas de foros, pues ninguno de ellos sabía detallar cómo podían usarse librerías externas sobre el código. Tampoco gestionan bien el paso de Docker a Kubernetes, y proporcionan ficheros obsoletos, por ejemplo indicando la versión en Docker. No obstante, han sido de ayuda a la hora de optimizar tareas como el diseño, las consultas a la base de datos y, en general, para definir la estructura del proyecto.

# Bibliografía

- *Docker Docs.*
- *adiii717/docker-python-cronjob/cron-numpy*. GitHub.
- *cron en un dockerfile*. Stack Overflow en español.
- *Adminer.org*.
- *Containerization of Python Flask Nginx in docker*. Medium (@habibullah.127.0.0.1).
- *Deploy flask app inside container and setup Nginx reverse proxy*. Medium (@prashantsde).
- *Flask Tutorial*.
- *Python — Build a REST API using Flask*. GeeksForGeeks.
- *Gotenberg Documentation*.
- *Kubernetes Documentation*.
- *Deploy on Kubernetes with Docker Desktop*. Docker Docs.
- *How to Deploy Postgres to Kubernetes Cluster*. DigitalOcean Community.
- *How to Deploy Postgres on Kubernetes*. CloudyTuts.
- *A Hands-On Guide to Kubernetes CronJobs*. Medium (@muppedaanvesh).
- *How to Deploy Flask App on Kubernetes?*. GeeksForGeeks.