Universidad de Valladolid

ESCUELA DE INGENIERÍA INFORMÁTICA (SG)
Grado en Ingeniería Informática de Servicios y Aplicaciones

Aplicación de Reserva de Entradas para Eventos Culturales con Integración Continua y Arquitectura de Microservicios con Javascript

Alumno: Carlos Martín Ruiz
Tutor: Fernando Díaz Gómez

1. Introducción **4**
Identificación del TFG 4
Motivación 4
Objetivos y alcance 6
Usuarios de la aplicación 7
Equipo de desarrolladores 7
Equipo de operaciones y despliegue (DevOps) 7
Equipo de QA 7
Estructura de la documentación 8
2. Gestión del proyecto y presupuesto **10**
Metodología 10
Planificación del proyecto 10
Presupuesto 11
Presupuesto hardware 11
Presupuesto de infraestructura 12
Presupuesto de recursos humanos 12
Presupuesto total 12
3. Tecnologías asociadas **14**
Virtualización 14
Hipervisores 14
Contenedores 15
Docker 15
Arquitectura básica 15
Construir imágenes 16
Construyendo imágenes a partir de un Dockerfile 16
Registros 17
Clustering y manejo de contenedores 18
Kubernetes 19
Escalabilidad y Alta Disponibilidad en Kubernetes 20
Docker Compose 20
Cypress 20
4. Arquitectura basada en microservicios y soporte CI/CD **21**
Arquitectura basada en microservicios 21
Introducción al CI/CD 23
Principios y prácticas 23
Entornos 23
Pipeline 24
Proceso de construcción 24
Testing 24

Despliegue 25
Jenkins: el servidor de automatización de código abierto 25
Conceptos básicos de Jenkins 25
Pipeline como código 25
Imágenes de contenedor como artefactos 26

5. Desarrollo del caso de estudio 27
Análisis 27
Actores 27
Casos de uso 27
Usuario no registrado 27
Usuario registrado 28
Administrador del sitio 29
Administrador de la infraestructura 32
Requisitos funcionales 32
Requisitos de información 33
Diagrama Entidad-Relación 34
Diseño 35
Implementación 37
Webapp 37
Estructura del microservicio 38
Detalles de implementación 41
Dockerfile 42
Main API 43
Estructura del microservicio 44
Detalles de implementación 45
Dockerfile 48
Payments API 48
Estructura del microservicio 49
Detalles de implementación 49
Dockerfile 50
Pruebas 50

6. Propuesta de arquitectura con soporte CI/CD para nuestro caso de estudio 54
Entornos 54
Pipeline 54

7. Resumen y conclusiones 64

Referencias 65

1. Introducción

Identificación del TFG

Este Trabajo de Fin de Grado ha sido realizado por Carlos Martín Ruiz, alumno de Ingeniería Informática de Servicios y Aplicaciones en la Escuela de Ingeniería Informática del Campus María Zambrano, perteneciente a la Universidad de Valladolid.

Motivación

Nos encontramos en una época en la que el mundo evoluciona a una velocidad vertiginosa. Debido a esto, necesitamos que los cambios y mejoras se produzcan de una manera rápida, acorde a las necesidades del ser humano. En el mundo digital esta situación se manifiesta de igual manera, por lo que el *software* debe mejorar de una manera rápida, segura y fiable. Como respuesta a este problema surgen las técnicas de integración contínua y de despliegue continuo (CI/CD) y la arquitectura basada en microservicios.

En las arquitecturas de aplicación tradicionales, todo el código se encuentra empaquetado en una única entidad, es decir, se basa en una arquitectura monolítica. Los desarrolladores deben conocer una gran parte del código para poder realizar un cambio, por pequeño que sea. Además, pueden surgir otros problemas. Por ejemplo, supongamos que un componente de la aplicación requiera de una versión concreta de un paquete del sistema para poder funcionar correctamente, pero, por otro lado, una mejora solicitada de la aplicación puede requerir que ese paquete se actualice a una versión más reciente. Esto supone un desafío. Hay que decidir entre actualizar ese paquete, y que el componente de la aplicación que lo utiliza deje de funcionar, o no actualizar el paquete, lo que puede traducirse en vulnerabilidades de seguridad del sistema en su conjunto, entre otras cosas.

Utilizar una arquitectura basada en microservicios resuelve este problema: la aplicación se divide en aplicaciones más pequeñas e independientes entre sí, con propósitos muy concretos, que se comunican entre ellas. Estas pequeñas aplicaciones se empaquetan en contenedores con todo lo necesario para funcionar. Así, cada equipo de desarrollo puede encargarse de un microservicio, sin interferir con los demás, agilizando, así, todo el proceso de desarrollo y reduciendo la posibilidad de que se produzcan errores.

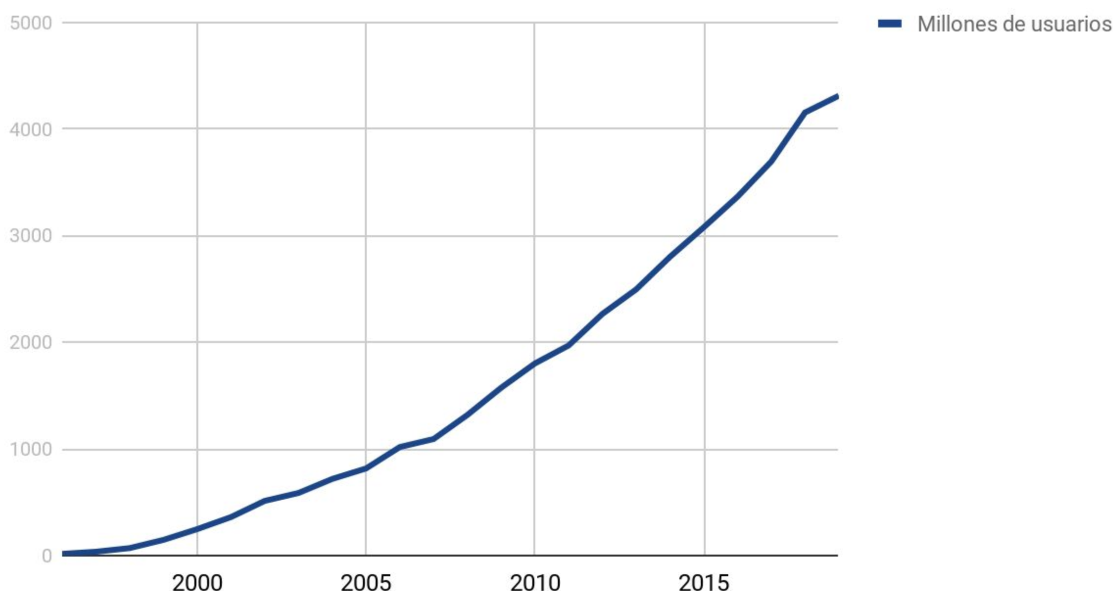

Figura 1. Evaluación anual del crecimiento de usuarios de Internet (Fuente: https://www.internetworldstats.com/emarketing.htm)

El número de usuarios no ha dejado de crecer desde que se creó Internet, por lo que los sitios web cada vez tienen más tráfico que soportar. Páginas web muy importantes han sufrido caídas del servicio debido a picos de tráfico que no supieron prever. Para solventar este tipo de problemas, han surgido las tecnologías de Alta Disponibilidad y Escalabilidad, que alcanzan su máximo exponente cuando se habla de proveedores Cloud como, por ejemplo, Amazon Web Services, Google Cloud Platform o Microsoft Azure.

De manera tradicional, la escalabilidad y la alta disponibilidad se llevaban a cabo teniendo varias copias de la aplicación monolítica y balanceando el tráfico entre ellas, para que en caso de que unas de ellas cayera, el resto pudiera seguir respondiendo las peticiones de los usuarios. En arquitecturas cloud, combinadas con los microservicios, esto puede realizarse de manera mucho más sencilla. Este proceso lo lleva a cabo el orquestador de contenedores de manera automática e individual para cada microservicio, escalando únicamente aquellos servicios que estén recibiendo una mayor cantidad de tráfico y manteniendo siempre un número mínimo de copias de cada uno.

La motivación de este TFG es aproximarse lo máximo posible a la arquitectura basada en microservicios, a la Integración y Despliegue Continuo y a la Alta Disponibilidad y Escalabilidad,

aplicando estas tecnologías a una aplicación web desplegada en un entorno con exigencias reales, para poder así observar las características y potenciales usos que conllevan este tipo de metodologías en el desarrollo y despliegue de la misma. Disponemos de un equipo que cuenta a su vez con tres equipos más pequeños, con labores bien diferenciadas. Estos equipos son el equipo de desarrolladores, el equipo de operaciones y el equipo de QA (Quality Assurance). Para el caso de este TFG, las labores de los tres roles han sido desempeñadas por mí, representando el stack completo de la aplicación.

Para ello supondremos que una empresa nos ha solicitado la construcción de una sitio web para la reserva y gestión de entradas para eventos culturales en toda España. El cliente requiere que la aplicación pueda soportar grandes cantidades de tráfico entrante en días en los que se produzcan anuncios de grandes eventos y que los recursos utilizados por la aplicación se minimicen en situaciones en las que la aplicación no esté soportando un número muy alto de peticiones. Además es necesario que el tiempo de disponibilidad de la aplicación sea el más alto posible, ya que la caída del servicio podría repercutir seriamente en los ingresos del cliente.

Objetivos y alcance

Uno de los objetivos principales es que se puedan realizar cambios en el código de una manera muy ágil y poder estar seguros de que los cambios realizados no contienen fallos que puedan rebajar la confianza de los usuarios en el servicio. De esta manera, los cambios pueden estar desplegados en el entorno de Kubernetes de producción en pocos minutos, con un 100% de confianza en que no introducen fallos en la aplicación. Además, el equipo de desarrollo puede realizar cambios en uno de los microservicios de manera que no afecten al resto.
El segundo objetivo es asegurar la disponibilidad y la escalabilidad, realizando despliegues de nuevas versiones con un *downtime* mínimo y siendo resilientes ante fallos y picos de tráfico. Estos objetivos generales se desgranan en los siguientes objetivos específicos.

OBj - 01	Ejecutar la construcción de los diferentes microservicios por separado cada vez que se produce un cambio en el código de cualquiera de ellos
OBj - 02	Ejecutar la integración de los diferentes servicios cada vez que se produce un cambio en el código de cualquiera de ellos
OBj - 03	Ejecutar los tests de integración de los diferentes servicios cada vez que se produce un cambio en el código de cualquiera de ellos
OBj - 04	Realizar el despliegue de la aplicación a un entorno de Kubernetes cada vez que se produce un cambio en el repositorio de código de cualquiera de los microservicios y los tests de integración se pasan sin ningún fallo
OBj - 05	Asegurar que el servicio está disponible el 85% del tiempo
OBj - 06	Asegurar que el servicio siga estando disponible para los usuarios cuando se produzcan picos de tráfico entrante

OBj - 07	Asegurar que los cambios y mejoras puedan ser desplegados de una manera rápida, estando seguros de que no se introducen fallos en la aplicación

Usuarios de la aplicación

Los usuarios de la aplicación están definidos desde el punto de vista del equipo humano que llevará a cabo el proyecto. Así, dentro del proyecto, pueden distinguirse los siguientes roles:

Equipo de desarrolladores

El equipo de desarrolladores está subdividido en tres equipos más pequeños, tantos como microservicios componen este proyecto. Existen dos equipos especializados en tecnologías de *backend*, encargados de construir los dos microservicios que conformarán el *backend* de la aplicación. El equipo de desarrolladores restante sería el encargado de desarrollar el *frontend*. Entre ellos hay especialistas en *frameworks* de Javascript, así como diseñadores expertos en *User Interface* (UI) y *User Experience* (UX).
Estos tres subequipos acuerdan las interfaces de comunicación que serán usadas por el *frontend* y por los microservicios de *backend* para comunicarse entre ellos.

Equipo de operaciones y despliegue (*DevOps*)

El equipo de *DevOps* será el encargado de llevar a cabo todas las labores de despliegue y administración de la arquitectura. Su principal objetivo es abstraer al equipo de desarrollo de la infraestructura sobre la que se ejecutará el código, para que no tengan que preocuparse de nada más que de programar su microservicio y de las interfaces que expondrán para la comunicación entre ellos, agilizando así en gran manera la velocidad con la que progresa el proyecto.
A más bajo nivel, son los encargados de empaquetar las aplicaciones en contenedores y de definir las configuraciones de despliegue para cada una de las fases por las que pasará el desarrollo del producto, en este caso, desarrollo, integración y producción.
Además, su labor también incluye la monitorización y el alarmado de la aplicación una vez que la aplicación pase a producción.

Equipo de QA

El equipo de QA (*Quality Assurance*) es el encargado de asegurar que implementación del código y la arquitectura están alineadas con los requisitos de la aplicación y que esta está libre de errores antes de ser entregada al cliente.
Para ello llevan a cabo exhaustivos tests para poder detectar cualquier error que pueda tener el proyecto y trasladarlos a los equipos de desarrollo o al de operaciones, según corresponda. Estos tests son realizados de manera manual y programática, ayudándose de librerías específicas para testear aplicaciones.

Estructura de la documentación

Este documento está organizado siguiendo la siguiente estructura:

1. **Introducción**: en este capítulo se presenta el TFG y se expone el contexto necesario para entender el objetivo de este proyecto. A su vez, este capítulo tiene los siguientes apartados:
    a. Identificación del TFG: apartado donde se muestra información sobre la persona que ha realizado este TFG.
    b. Motivación: apartado donde se exponen los motivos para llevar a cabo este TFG.
    c. Objetivos y alcance: apartado donde se definen los objetivos que se esperan cumplir con este proyecto.
    d. Usuarios de la aplicación: usuarios involucrados en el proyecto, desde el punto de vista de su desarrollo.
    e. Estructura de la documentación: apartado donde se explica la estructura que seguirá este documento.
2. **Gestión del proyecto y presupuesto:** en este capítulo se explica cómo se llevará a cabo la gestión del proyecto y el presupuesto detallado.
    a. Metodología: en este apartado se explica la metodología de desarrollo aplicada en este proyecto.
    b. Planificación del proyecto
    c. Presupuesto
3. **Tecnologías asociadas:** en este capítulo se detallan algunos conceptos y tecnologías necesarias para entender correctamente esta documentación.
4. **Desarrollo de la arquitectura:** en este capítulo se explican, de forma general, principios y herramientas que se utilizan para desarrollar arquitecturas basadas en microservicios y flujos de integración y despliegue continuos.
5. **Desarrollo de la arquitectura CI/CD**: en este capítulo se expone la arquitectura en la que se basará nuestra aplicación y su flujo de integración y despliegue, junto con el código que lo hace posible.
6. **Desarrollo del caso de estudio:** en este capítulo se desarrollan los pasos que se han seguido para construir la aplicación web, desde un punto de vista tradicional.
    a. Análisis: en este apartado se realiza el análisis de requisitos de la aplicación, se muestran los actores y se modelan los casos de uso.
    b. Diseño: en este apartado se define la que será la arquitectura sobre la que se ejecutarán los microservicios.
    c. Implementación: en este apartado se muestran algunos detalles de implementación que se consideran importantes, además de la estructura que utiliza el código de cada uno de los microservicios de la aplicación.
    d. Pruebas: en este apartado se exponen los tests programáticos que se realizan sobre la aplicación como parte del *pipeline* de integración y despliegue continuos, para asegurar que no se introducen errores.

7. **Conclusiones:** en este capítulo se exponen las conclusiones obtenidas de aplicar este tipo de arquitecturas y de herramientas de integración y despliegue continuos.
8. **Referencias:** en este capítulo se hace referencia a toda la documentación y recursos consultados para la realización de este proyecto.

2. Gestión del proyecto y presupuesto

En este capítulo se explica cómo se llevará a cabo la gestión del proyecto y el presupuesto detallado.

Metodología

Las metodologías ágiles buscan un modelo de mejora continua en el que se planifica, se crea, se comprueba el resultado y se mejora. Algo que es constante y rápido, con plazos de entrega reducidos que buscan evitar la dispersión y centrar toda la atención en la tarea encomendada. Los principios y valores en los que se basan las metodologías ágiles tienen como principal característica realizar entregas rápidas y continuas de *software* funcional. Por ejemplo, en Scrum, el proyecto se divide en pequeñas partes que tienen que completarse y entregarse en plazos cortos, llamados *sprints*. De esta manera, solo se hacen cambios en la parte implicada y en muy poco tiempo.
Otra de las principales características de este tipo de metodologías es el uso de equipos multidisciplinares de trabajo conjunto, ayudando a obtener *feedback* de manera mucho más rápida y fiable. Los equipos ágiles realizan sesiones diarias llamadas *dailies*, en los que cada miembro explica brevemente las tareas que ha completado, las que va a hacer, y si hay impedimentos que no le hayan permitido avanzar.

Planificación del proyecto

Para este proyecto se seguirá una metodología de desarrollo Scrum. Esto quiere decir que el desarrollo estará dividido en *sprints* y que cada día se tendrá una *daily* para poner en común el trabajo realizado. El tiempo de desarrollo de la aplicación completa está estimado para durar 6 meses y los *sprints* tendrán todos la misma duración. Para llevar a cabo el proyecto, se necesitarán 4 *sprints*, cuya duración será de mes y medio. Este número de *sprints* será ampliable en caso de que el cliente requiera nuevas funcionalidades o mejoras de la aplicación. Estos cuatro *sprints* se dedicarán a:

1. Desarrollo de la arquitectura.
2. Desarrollo de los procesos de Integración Continua y Despliegue Continuo.
3. Desarrollo del módulo para administradores de la aplicación.
4. Desarrollo del módulo para los clientes de la aplicación.

Las dos primeros *sprints* involucran principalmente al equipo de operaciones y despliegue y al de QA (*Quality Assurance*), que serán los encargados de validar que el código desarrollado por el equipo de operaciones cumple su función y no presenta errores. Durante los dos primeros *sprints* los equipos de desarrollo deberán prestar atención a estas fases para conocer el funcionamiento básico de la infraestructura sobre la que se ejecutará el código de los microservicios y los procesos de integración que se lanzarán cada vez que se realice un cambio en el código de la aplicación.

Los siguientes dos *sprints* estarán enfocados a desarrollar el código de la aplicación web, es decir, el código que se ejecutará dentro de los microservicios. En estas dos fases, los principales involucrados serán los equipos de desarrollo de cada microservicio y el equipo de QA, que será en el encargado de realizar tests manuales y programar los tests de integración. El equipo de operaciones se dedicará a dar soporte y a resolver cualquier error que pueda descubrirse en el código de la infraestructura o en el del proceso de CI/CD.

Dentro de cada *sprint* se dedicarán estos porcentajes de tiempo a cada una de estas tareas:

Actividad	Porcentaje
Análisis	15%
Diseño	25%
Implementación	35%
Pruebas	25%

El desarrollo del proyecto se ha planificado para comentar a principios de diciembre y acabar a finales del mes de mayo. El trabajo se llevará a cabo aplicando media jornada, es decir, 4 horas al día, descansando los viernes.

Presupuesto

En este apartado se hace un desglose de todos los gastos que supone el proyecto.

Presupuesto *hardware*

Para el desarrollo de este proyecto se han utilizado los siguientes elementos *hardware*:

- PC:

Elemento	Modelo	Precio
Procesador	Intel Core i7-7700K	499,00€
Memoria RAM	Kingston HyperX Fury Black 8GB DDR4	41,00€
Sistema Operativo	Windows 10 Home	145,00€
Tarjeta gráfica	NVIDIA GeForce RTX 2060 OC	349,00€

Coste total: 499,00+41,00+145,00+349,00=1034,00€
Vida útil del PC: 4 años
Uso durante 6 meses, 12,5% de vida útil = 129,25€

- Internet: 300Mb de fibra óptica al mes por 6 meses = 228,00€
- Material de oficina: ratón, teclado, folios = 40,00€

El presupuesto *hardware* total se eleva a:
129,25+228,00+40,00=**397,25€**

Presupuesto de infraestructura

Para llevar a cabo el proyecto se utilizarán servidores virtuales alojados en servidores. La opción inicial era utilizar los servicios del proveedor cloud Amazon Web Services. Los requisitos del proyecto son:

Servidor	**Características**	**Precio en AWS (mensual)**
Servidor de integración	4 núcleos, 8 GiB de memoria RAM y 30 GB de disco	78,18€
Clúster de Elastic Kubernetes Services con un solo nodo	El nodo es de 4 núcleos, 8 GiB de memoria RAM y 30 GB de disco	64,40€+78,18€=142,58€

El precio total de la infraestructura para un mes = 78,18 + 142,58 = 220,76€
El precio total de la infraestructura para 6 meses = 220,76 * 6 = **1.324,56€**

Gracias al Departamento de Informática de la Universidad de Valladolid, que me ha prestado sus servidores virtuales para la realización del proyecto, he podido ahorrar la cantidad de 1.324,56€ en la realización de este TFG.

Presupuesto de recursos humanos

Para calcular el presupuesto que supondrá desarrollar la aplicación, supondremos que ha sido desarrollada íntegramente por Carlos Martín Ruiz, desempeñando todos los roles dentro del equipo de nuestra empresa. Por esto, será necesario abonar sueldo a una sola persona. Para ello, supondremos que el sueldo medio de un ingeniero informático junior equivale a 20.000€ brutos al año.

20.000 / 12 = 1.666,66€ al mes
1.666,66 * 6 = 10.000,00€

Pagar a un ingeniero informático durante los 6 meses de duración del proyecto supondrá 10.000€.

Presupuesto total

El presupuesto total para el proyecto es de:

Concepto	**Cantidad**
Presupuesto *hardware*	397,25€
Presupuesto de infraestructura	1.324,56€
Presupuesto de recursos humanos	10.000,00€
TOTAL:	**11.721,25€**

3. Tecnologías asociadas

Este proyecto hace uso de varias tecnologías que se explican detalladamente en este capítulo.

Virtualización

La virtualización de servidores hace posible ejecutar múltiples instancias de sistemas operativos concurrentemente en el mismo *hardware* físico. El *software* de virtualización divide la CPU, la memoria y los dispositivos de entrada y salida, distribuyendo dinámicamente su uso entre los diferentes sistemas operativos virtualizados y resolviendo los conflictos por el acceso a los recursos. Desde el punto de vista del usuario, un servidor virtual se comporta de manera idéntica a como lo haría un servidor físico.
Esta tecnología trae como consecuencia varias ventajas. Los servidores virtuales son mucho más flexibles que sus equivalentes físicos. Son totalmente portables y pueden ser gestionados programáticamente. El *hardware* sobre el que se ejecutan es usado más eficientemente ya que puede ser utilizado por varias máquinas virtuales simultáneamente.

Hipervisores

Un hipervisor (*Hypervisor*), también llamado monitor de máquinas virtuales, es una capa de *software* que media entre las máquinas virtuales y el *hardware* físico sobre el que están ejecutándose. Es el encargado de compartir los recursos del sistema entre los sistemas operativos virtuales, los cuales están aislados los unos de los otros y cuyo acceso a los recursos de la máquina física se hace exclusivamente a través del hipervisor. Existen dos tipos de hipervisores:

- Hipervisor de tipo 1: se ejecuta directamente sobre el *hardware* de la máquina física sin ningún sistema operativo que le de soporte, y por este motivo a veces es conocido como hipervisor nativo.
- Hipervisor de tipo 2: son aplicaciones que se ejecutan en la capa de usuario sobre un sistema operativo de propósito general.

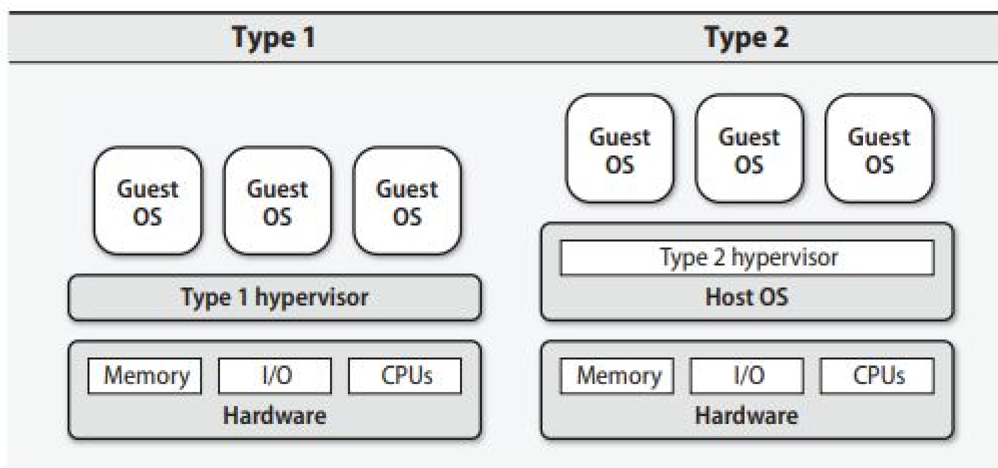

Figura 2. Comparación entre hipervisores de tipo 1 y tipo 2. (Fuente: Nemeth, E., Snyder, G., Hein, T. R., Whaley, B., & Mackin, D. (2017). *UNIX and Linux System Administration Handbook (5th Edition)* (5.a ed.). New York, USA: Addison-Wesley Professional)

Contenedores

Para ilustrar la utilidad de los contenedores, consideraremos una aplicación web típica desarrollada en cualquier *framework* o lenguaje moderno. Como mínimo, los siguientes ingredientes serán necesarios para instalar y ejecutar la aplicación:

- El código de la aplicación.
- Librerías y otras dependencias, cada una con un número de versión concreto.
- Un intérprete o *runtime* para ejecutar el código, también con una versión concreta.
- Cuentas de usuario, variables de entorno y servicios provistos por el sistema operativo.

Mantener la uniformidad en cada una de estas áreas es un desafío constante. Además, en sitios donde los desarrolladores y los administradores de sistemas son equipos completamente separados, una cuidadosa coordinación es necesaria para poder identificar las responsabilidades de cada equipo.

Una imagen de contenedor simplifica estas tareas, empaquetando la aplicación y sus requisitos en un fichero estándar y portable. Cualquier máquina, con un motor compatible para contenedores, puede crear estos contendores cogiendo la imagen como plantilla. Decenas y centenas de contenedores pueden ser ejecutados simultáneamente en la misma máquina, sin conflictos.

En esencia, un contenedor en un grupo de procesos aislados que están restringidos a un sistema de ficheros y a un *namespace* de procesos privados. Los procesos "containerizados" comparten el *kernel* y otros servicios del sistema operativo de la máquina física, pero por defecto no tienen acceso a ficheros o recursos fuera del contenedor. Las aplicaciones que están siendo ejecutadas dentro de un contenedor no tienen consciencia de ello y no requieren ninguna modificación. Estos contenedores se ejecutan exactamente igual, independientemente de cual sea la infraestructura sobre la cual están corriendo. Los contenedores aíslan el *software* de su entorno y aseguran que funcionen uniformemente, a pesar de las diferencias que puedan existir, por ejemplo, entre las distintas fases del proyecto, como son la de desarrollo, la de integración y la producción.

Docker

El producto principal de Docker, Inc.'s es una aplicación cliente/servidor que construye y maneja contenedores. El motor de contenedores Docker está escrito en Go y es altamente modular.

Arquitectura básica

`docker` es un comando ejecutable que permite el manejo de tareas del sistema Docker. `dockerd` es el proceso demonio que implementa las operaciones con los contenedores y las imágenes. Este

proceso es el encargado de crear contenedores invocando a las llamadas al sistema apropiadas, preparando los sistemas de ficheros y ejecutando procesos.
Los administradores interactúan con `dockerd` desde la línea de comandos ejecutando subcomandos del cliente de `docker`.
Una imagen es una plantilla para un contenedor. Incluye los ficheros de los cuales dependen los procesos ejecutándose dentro del contenedor, como librerías, binarios del sistema operativo y aplicaciones.
Un contenedor depende de la imagen como base para su ejecución. Cuando `dockerd` ejecuta un contenedor, crea una capa de sistema de ficheros de escritura que está separada de la imagen de origen. El contenedor puede leer todo lo necesario de la imagen, pero cualquier operación de escritura está restringida a la capa de escritura propia del contenedor.
Un registro de imágenes es una colección centralizada de imágenes. `dockerd` se comunica con estos registros cuando se hace un `docker pull` de una imagen que no está presente en la máquina local o cuando se hace un `docker push` de una imagen propia. El registro por defecto es Docker Hub, que almacena imágenes de aplicación muy populares.

Construir imágenes

Además de utilizar imágenes construidas por otros y almacenadas en registros públicos, también se pueden "containerizar" aplicaciones propias construyendo imágenes que incluyan el código de la aplicación. El proceso de construcción comienza con la imagen base. Se puede añadir la aplicación añadiendo cambios como una nueva capa y almacenando la imagen en la base de datos de imágenes local. Así, se podrá ejecutar un contenedor a partir de esa imagen o subirla a un registro de imágenes para hacerla accesible a otros sistemas que ejecuten Docker.

Construyendo imágenes a partir de un Dockerfile

Un Dockerfile es una receta para construir una imagen. Contiene una serie de instrucciones y comandos de "shell". El comando `docker build` lee el Dockerfile, ejecuta sus instrucciones en secuencia y guarda el resultado como una imagen.
La primera instrucción de un Dockerfile especifica la imagen que se utilizará como base. Cada instrucción subsequente genera un cambio y lo guarda en una nueva capa, que es utilizada como base para la siguiente instrucción. Cada capa incluye únicamente los cambios sobre la capa anterior. Después, se unen las capas para formar el sistema de ficheros raíz. Al final, una imagen es un archivo ejecutable, y un contenedor es una instancia de esa imagen en ejecución.

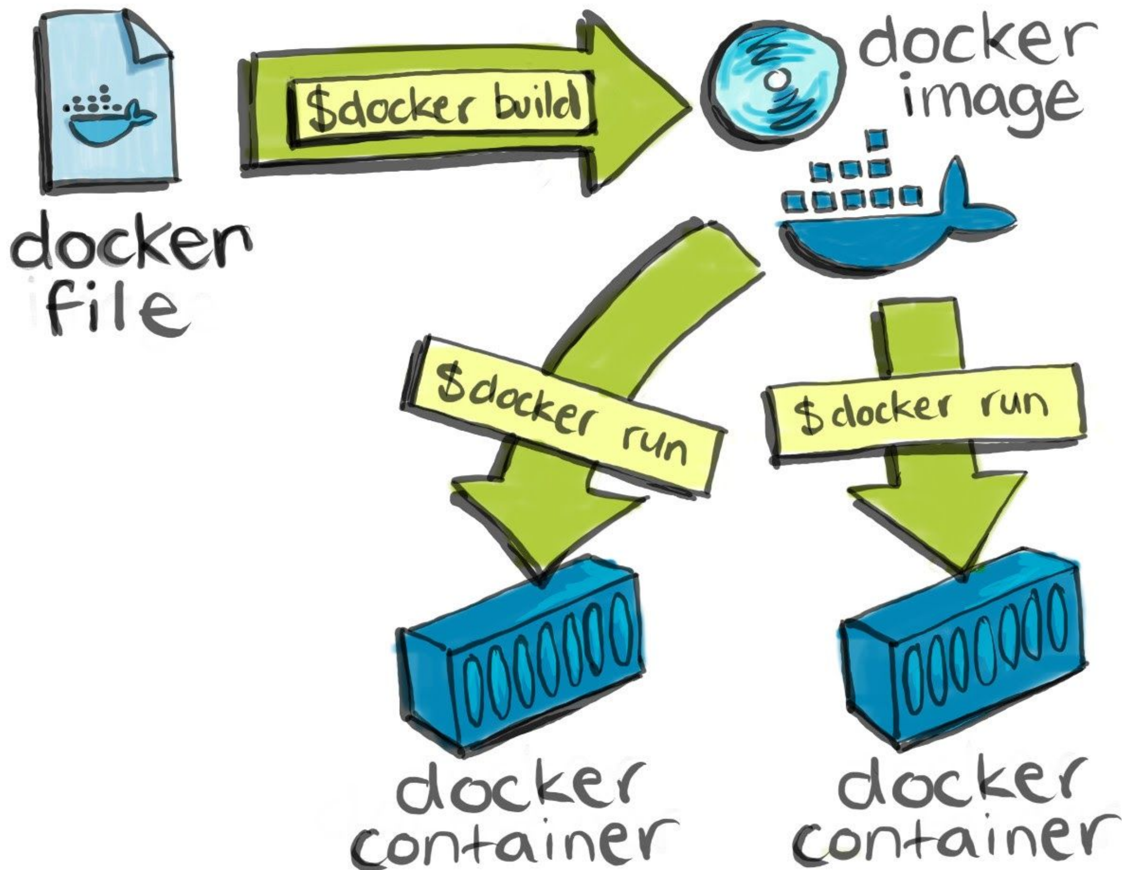

Figura 3. Flujo de trabajo de Docker, desde un Dockerfile a un contenedor en ejecución. (Fuente: https://cultivatehq.com)

Registros

Un registro es un índice de imagénes de Docker al cual `dockerd` puede acceder a través de HTTP. Cuando una imagen es solicitada y no está almacenada en el disco local, `dockerd` la descarga del registro.

Docker Hub es un registro hosteado por Docker, Inc. Almacena imágenes de muchas distribuciones y proyectos de código abierto, y además permite subir imágenes propias para uso público. Cualquiera puede descargar imágenes públicas de Docker Hub, pero con una suscripción pagada, además puedes crear repositorios privados. Haciendo `docker login` puedes acceder a estos repositorios privados para poder almacenar y descargar imágenes propias. También permite ejecutar la construcción de imágenes como respuesta a un cambio en un repositorio de Github.

Docker Hub no es el único registro basado en suscripciones pagadas. Existen otros como Google Container Registry y Elastic Container Registry.

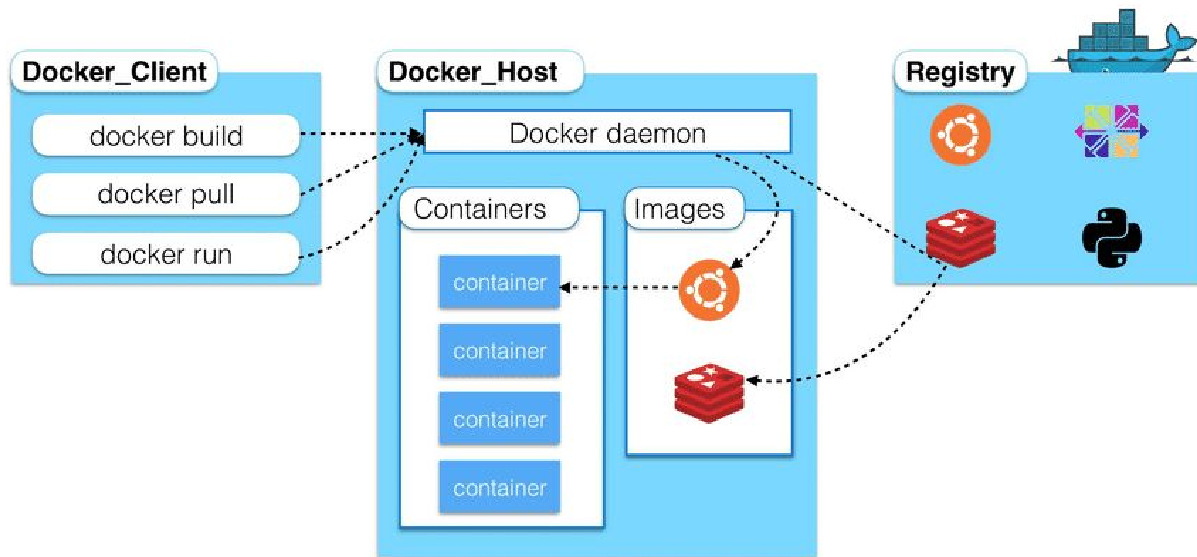

Figura 4. Arquitectura básica de Docker. (Fuente: https://www.researchgate.net/profile/Yahya_Al-Dhuraibi/publication/308050257/figure/fig1/AS:433709594746881@1480415833510/High-level-overview-of-Docker-architecture.png)

Clustering y manejo de contenedores

El motor de contenedores Docker es responsable de los contenedores de manera individual, no responde a la pregunta de ejecutar muchos contenedores en máquinas distribuidas con una configuración de alta disponibilidad.

En lo relativo al despliegue de contenedores a lo ancho de la red, es necesario un *software* orquestador de contenedores. Para entender cómo funcionan estos sistemas, pensemos en los servidores como una granja de recursos de computación. Cada nodo dentro de la granja ofrece sus recursos al orquestador. Cuando el orquestador recibe una petición de ejecutar un contenedor o un conjunto de ellos, los sitúa en un nodo que tenga los recursos necesarios disponibles. Como el orquestador sabe donde ha situado este contenedor, puede empezar a enrutar tráfico hacia los nodos correctos dentro del clúster. Los administradores del clúster interactúan con el orquestador, en lugar de hacerlo con los motores de contenedores de cada uno de los nodos.

Los sistemas orquestadores de contenedores proveen varias características útiles:

- Los algoritmos de selección de nodo eligen el mejor nodo para ejecutar el contenedor en función de las necesidades de este.
- Las APIs expuestas permiten asignar tareas al clúster de una manera simple, facilitando, por ejemplo, el uso conjunto de herramientas de integración contínua y el despliegue continuo (CI/CD).
- Configuraciones de alta disponibilidad de una manera muy sencilla, como ejecutar una réplica de cada contenedor en cada región geográfica.
- Control de la disponibilidad de los servicios ofrecido de forma nativa. Por ejemplo, si un servicio deja de estar disponible, es realojado y redesplegado de forma automática.
- Facilidad para escalar recursos, simplemente añadiendo o eliminando nodos de cómputo.

- El sistema de orquestación puede interactuar con un balanceador de carga para enrutar el tráfico externo de una manera uniforme a dentro del clúster.

Uno de los desafíos que supone la utilización de este tipo de herramientas es mapear el nombre de los servicios a los contenedores, ya que los contenedores se consideran como entidades efímeras y pueden tener puertos asignados dinámicamente. Este problema es llamado descubrimiento de servicios.
Uno de los orquestadores de contenedores más utilizados en la actualidad es Kubernetes.

Kubernetes

Kubernetes (también llamado *k8s*) se originó dentro de Google y fue lanzado por algunos de los desarrolladores que trabajaron en Borg, el administrador de clústeres interno de Google. Fue liberado como código abierto en 2014 y ahora tiene más de mil contribuidores activos.
Kubernetes consiste en algunos servicios que se unen para formar un clúster. Los bloques básicos son:

- El API server, para operar las peticiones.
- Un programador, para emplazar las tareas dentro del clúster.
- Un *controller manager*, para seguir el estado del clúster.
- El *kubelet*, un agente que se ejecuta en todos los nodos.
- cAdvisor, para monitorizar las métricas de los contenedores.
- Un proxy, para enrutar las peticiones entrantes al contenedor apropiado.

Los tres primeros ítems se ejecutan en los *masters* del clúster (que también pueden comportarse como nodos) para una alta disponibilidad. Los procesos de *kubelet* y cAdvisor se ejecutan en cada nodo, manejando las peticiones del *controller manager* y reportando estadísticas sobre la salud de las tareas que están ejecutando.
En Kubernetes, los contenedores son desplegados como *pods*, los cuales contienen uno o más contenedores. Todos los contenedores dentro de un *pod* se ejecutarán juntos dentro de un mismo nodo. Los *pods* tienen asignados IPs únicas dentro del clúster, y son etiquetados para labores de identificación y posicionamiento.
Estos *pods* son efímeros. Si un nodo muere, el controlador programará un *pod* de reemplazo en un nodo diferente y con una IP distinta.
Los servicios son colecciones de *pods* relacionados con una dirección IP que no cambia. Si un *pod* asociado a un servicio muere o falla su *healthcheck* (comprobación de la salud del *pod*), el servicio elimina ese *pod* de su rotación y deja de enviarle tráfico.
Kubernetes tiene soporte integrado para el descubrimiento de servicios, manejo de secretos, despliegue y autoescalamiento de *pods*. También puede soportar aplicaciones con estados moviendo los volúmenes de un nodo a otro según sea necesario.

Escalabilidad y Alta Disponibilidad en Kubernetes

En Kubernetes, los *masters* del clúster son los encargados de guardar la configuración que hace referencia al estado deseado de los servicios ejecutándose dentro. Uno de estos parámetros de configuración es el número de réplicas deseadas de cada servicio. El *master* será el encargado de mantener ese número indicado de réplicas de cada servicio, planificando el despliegue de *pods* de ese servicio a través de todo el clúster. En el momento que uno o varios de los *pods* dejen de estar disponibles, el *master* comparará el número de réplicas deseado de cada servicio en su configuración con el número de *pods* ejecutándose actualmente. En caso de que esos números no coincidan, el *master* planificará tantos *pods* nuevos como sea necesario para mantener ese número de *pods* deseados de cada servicio, manteniendo así una alta disponibilidad de los servicios.
En caso de la carga de los microservicios aumente puntualmente, el *master* escalará horizontalmente, y de manera automática, el número de réplicas de esos servicios. Este escalamiento horizontal puede realizarse de dos maneras principalmente:

- Escalado de la aplicación: este escalado se realiza planificando más o menos *pods* a través de todos los nodos del clúster, en función de las necesidades de cada servicio. El *master* aumentará o disminuirá el número de réplicas, monitorizando los recursos consumidos en cada *pod*. Adicionalmente, cuando el número de réplicas de un servicio es mayor que uno, puede asegurarse que los despliegues de nuevas versiones seguirán manteniendo la alta disponibilidad, al actualizar los *pods* uno a uno, en lugar de todos a la vez.
- Escalado del clúster: esta opción no es soportada nativamente por Kubernetes, pero puede ser llevada a cabo con la ayuda de proveedores cloud como Amazon Web Services o Google Cloud Platform. De esta manera, cuando se detecta que los nodos del clúster tienen una gran parte de sus recursos consumidos, automáticamente se levanta un nuevo nodo y se incluye en el clúster, disminuyendo así el porcentaje de recursos consumidos dentro del clúster.

Docker Compose

Docker Compose es una herramienta utilizada para definir y ejecutar aplicaciones multi contenedor para contenedores construidos sobre Docker. Se utilizar un fichero YAML para definir la configuración que utilizará cada contendor y las relaciones que puede haber entre ellos.

Cypress

Cypress es una herramienta de nueva generación para testear aplicaciones de *frontend* construido para la web moderna. Cypress puede testear cualquier pieza de código que es ejecute en un navegador o incluso APIs REST.
Detrás de Cypress hay un proceso de Nodejs. Cypress y el proceso de Nodejs se comunican constantemente, se sincronizan, y realizan tareas en nombre de cada uno. Teniendo acceso a las dos partes (*frontend* y *backend*) nos da la habilidad de responder a los eventos de la aplicación en tiempo real, mientras se trabaja fuera del navegador para tareas que requieren mayor privilegio. Cypress también trabaja en la capa de red, leyendo y alterando el tráfico web al vuelo.

4. Arquitectura basada en microservicios y soporte CI/CD

Arquitectura basada en microservicios

Tradicionalmente el diseño de *software* se ha realizado con arquitectura monolítica, en la que el *software* se estructura de forma que todos los aspectos funcionales quedan acoplados en un mismo programa. En este tipo de arquitectura, toda la información está alojada en un solo servidor, por lo que no hay separación entre módulos y las diferentes partes del programa están muy sujetas unas a otras. Esto acaba generando un problema a largo plazo, ya que se trata de un sistema no escalable de manera sencilla. La arquitectura basada en microservicios aparece como respuesta a este problema.

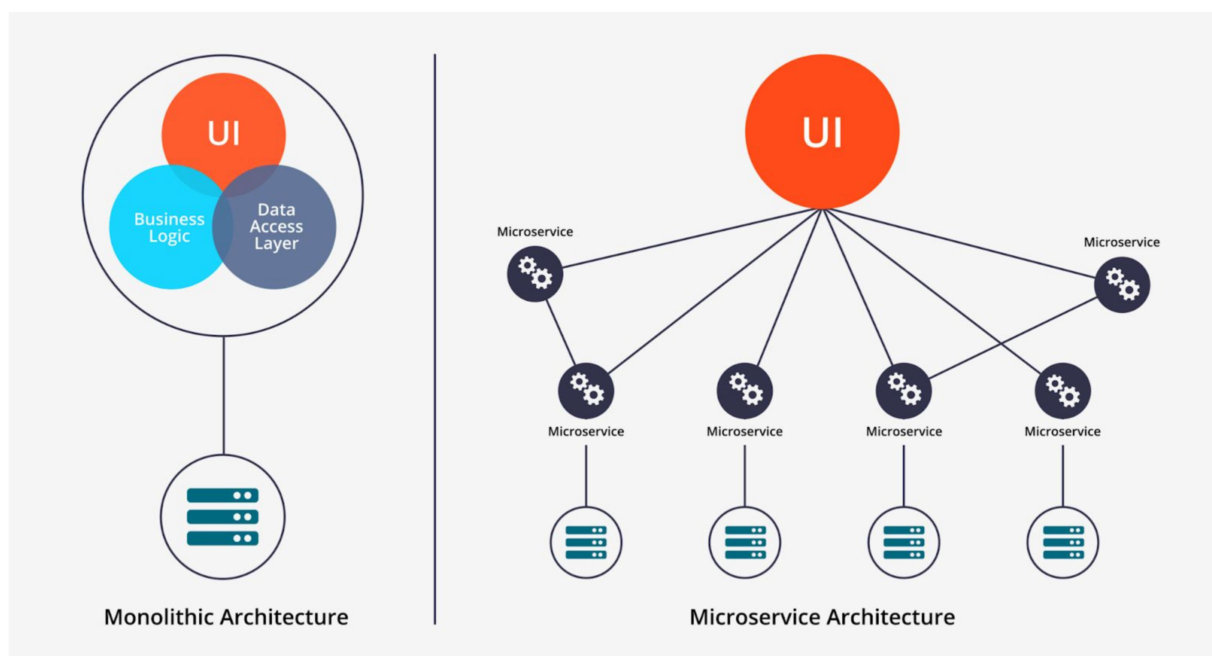

Figura 5. Diferencias entre una arquitectura monolítica y una arquitectura basada en microservicios. (Fuente: Malav, B. (2018, 16 junio). Microservices vs Monolithic architecture - Hash#Include. Recuperado 13 de junio de 2020, de https://medium.com/startlovingyourself/microservices-vs-monolithic-architecture-c8df91f16bb4)

La arquitectura basada en microservicios es un método de desarrollo de aplicaciones *software* que funciona como un conjunto de pequeños servicios que se ejecutan de manera independiente y autónoma, proporcionando una funcionalidad de negocio completa. En ella, cada microservicio es un código que puede estar en un lenguaje de programación diferente, y que desempeña una función específica. Los microservicios se comunican entre sí a través de protocolos ligeros como HTTP, y suelen contar con sistemas de almacenamiento propios, lo que evita la sobrecarga y la posible caída de la aplicación.

Esto ayuda a crear infraestructuras más adaptables y flexibles, debido a que, si se quiere modificar solamente un servicio, no es necesario alterar el resto de la infraestructura. Cada uno de los

microservicios se puede desplegar y modificar sin afectar a los demás servicios de la aplicación. Este tipo de arquitecturas traen como consecuencia una serie de ventajas:

- Modularidad: al tratarse de microservicios completamente autónomos y desacoplados, se pueden desarrollar y desplegar de forma independiente. Además un error en uno de los servicios no debería afectar al resto.
- Escalabilidad: cada módulo o servicio de la aplicación se puede escalar horizontalmente según sea necesario y de forma automática, aumentando la capacidad de los módulos que están soportando una mayor carga y disminuyéndola cuando la carga se reduce.
- Versatilidad: se pueden usar diferentes tecnologías y lenguajes de programación, lo que permite adaptar cada funcionalidad a la tecnología más adecuada y rentable.
- Rapidez de actuación: reducir el tamaño de los servicios permite un desarrollo y despliegue menos costosos.
- Mantenimiento simple y barato: esto se consigue al hacerse mejoras de un solo módulo y no tener que intervenir en la arquitectura completa.

Este tipo de arquitectura también trae consigo algunas desventajas, que se enumeran a continuación:

- Alto consumo de memoria y CPU, al tener cada microservicio sus propios recursos y bases de datos.
- Inversión de tiempo inicial para crear una arquitectura fragmentada e implementar la comunicación entre ellos.
- Complejidad en la gestión: al contar con un gran número de microservicios, se hace más complicado la gestión e integración de los mismos.
- No uniformidad: aunque esto pueda parecer una ventaja, puede acabar convirtiéndose en un desafío.
- Dificultad en la realización de pruebas: debido a que los componentes de la aplicación están distribuidos.

Las ventajas de utilizar arquitectura basada en microservicios y contenedores de forma conjunta saltan a la vista. Docker empaqueta cada servicio en un contenedor junto con todas sus dependendencias y variables de entorno. De esta manera, todos los microservicios de una aplicación pueden ejecutarse dentro de un mismo servidor, reduciendo el consumo de recursos y el coste, al tiempo que se mejora la eficiencia y la portabilidad de los microservicios.

Este método satisface las necesidades de administración de varios contenedores dentro un único servidor, pero se queda corta a la hora de administrar varios contenedores ejecutándose a través de varios servidores en aplicaciones distribuidas complejas. Para estos casos, puede ser necesario un orquestador de contenedores, que además provee, de una manera rápida y sencilla, soporte para alta disponibilidad y escalabilidad. Para este proyecto se ha decidido usar Kubernetes, como se explicará en el siguiente capítulo.

Para conseguir un rápido desarrollo e integración de todos los microservicios en este tipo de arquitecturas, pueden utilizarse técnicas y principios llamados Integración Continua y Despliegue Continuo (CI/CD).

Introducción al CI/CD

Muchos términos relacionados con la Integración Continua y el Despliegue Continuo (CI/CD) suenan similares y tienen los significados solapados. Vamos a echar un vistazo primero a las diferencias entre la integración continua, la entrega continua y el despliegue continuo:

- Integración continua: es el proceso de colaborar en una base de código compartida, introduciendo cambios en un sistema de control de versiones, creando y probando automáticamente las nuevas versiones del código.
- Entrega continua: es el proceso de desplegar automáticamente los cambios en el código a un entorno que no sea de producción, una vez que el proceso de integración está completo.
- Despliegue continuo: proceso de desplegar una aplicación a los entornos de producción, sin intervención de ningún operador.

Principios y prácticas

La agilidad del negocio es uno de los beneficios claves del CI/CD. El despliegue continuo facilita la liberación de nuevas características, bien testeadas, a producción en minutos u horas, en lugar de semanas o meses. Las siguientes secciones cubren algunas reglas básicas a tener en mente.

- Usar control de versiones: todo el código debería estar en un sistema de control de versiones. Pero no solo el código de aplicación en sí, sino también el código de la infraestructura, siguiendo los principios de la infraestructura como código (*Infrastructure as code*), para poder también hacer un seguimiento de las versiones que ha seguido la arquitectura de la aplicación.
- Contruir una vez, desplegar muchas: cada versión del código se construye una sola vez, pero debe ser desplegada y testeada en todos los entornos en los que sea posible y sean lo más similares al entorno de producción.
- Automatizar el proceso de principio a fin (*end-to-end*): construir, testear y desplegar el código de la aplicación sin intervención humana, es clave para hacer actualizaciones fiables para nuestras aplicaciones.
- Ejecutar la integración tras cada cambio en el repositorio: la idea detrás de la integración continua es que cada cambio en el repositorio de código en la rama de integración resulte, automáticamente, en una nueva construcción. En caso de que la construcción falle, es muy fácil ver qué líneas son las que han introducido el fallo y corregirlas de manera rápida.

Entornos

Las aplicaciones no se ejecutan de forma aislada. Dependen de recursos externos como bases de datos, cachés, registros de DNS, APIs HTTPs remotas y otras aplicaciones. Un entorno de ejecución incluye todos estos recursos y todo lo demás que la aplicación necesite para ejecutarse. La mayoría de los sitios usan estos tres entornos:

- Desarrollo o `dev`: donde se integran los cambios de múltiples desarrolladores, se prueban cambios en la infraestructura y se comprueban errores obvios. En el contexto de CI/CD, este entorno puede ser creado y reseteado varias veces al día.
- Staging o `stage`: es el entorno donde se ejecutan las pruebas tanto automáticas como manuales. Los *testers* u otros interesados en el producto usan este entorno para probar nuevas características o fallos.
- Producción o `prod`: este entorno es donde se implementa el servicio para los usuarios reales. Un corte en el servicio de este entorno debe ser resuelto inmediatamente, por lo que se monitoriza de una manera exhaustiva.

Pipeline

Un *pipeline* de CI/CD es una serie de pasos, llamados *stages*, que se ejecutan en secuencia. Cada *step* es esencialmente un *script* que realiza unas tareas específicas sobre el proyecto. En su nivel más básico, un *pipeline* de CI/CD:

- Construye y empaqueta la aplicación
- Ejecuta una serie de tests en busca de errores
- Despliega el código en varios entornos, culminando en el entorno de producción.

Las siguientes secciones explican las tres etapas con más detalle.

Proceso de construcción

El *build* o proceso de construcción transforma el código en una pieza de *software* instalable. Los *builds* pueden ser ejecutados como respuesta a un cambio en el repositorio de código del proyecto, a una ejecución manual o a un evento que se lanza periódicamente.
Los pasos específicos del proceso de construcción dependen del lenguaje y del *software* utilizado. Para algunos lenguajes puede traducirse en la compilación del código y, para aplicaciones basadas en contenedores, en la construcción de las imágenes de contenedor. Los resultados de esta fase son llamados artefactos, y, una vez completada la fase, son almacenados, por ejemplo, en servidores SFTP o en registros de imágenes de Docker. Estos artefactos deben estar disponibles para los sistemas que vayan a realizar la fase de despliegue.

Testing

En esta fase es donde se ejecutan los tests programados. Si el *build* falla en alguno de los tests, los demás *stages* del *pipeline* no se ejecutan. El equipo debe determinar por qué ha fallado el *build* y encontrar el error dentro del código. Como el *pipeline* se ejecuta debido a cualquier cambio en el código, es fácil aislar el problema al último cambio que se haya producido. Cuantas menos líneas de código nuevas introduzca cada cambio, más fácil será encontrar el error.
Los fallos en este *stage* no siempre provienen de un fallo en el código. Pueden ocurrir debido a fallos en la red, fallos en la infraestructura o en recursos de terceros utilizados por la aplicación.
Existen varios tipos de tests que suelen ser ejecutados en este *stage*:

- Análisis de código estático: detecta errores de sintaxis o violaciones de las guías de estilo del código.
- Tests unitarios: testean los componentes mínimos (unidades) de la aplicación, como funciones y métodos. Son escritos por los mismos desarrolladores del código.
- Tests de integración: ejecutan la aplicación con todos los *frameworks* y con las dependencias externas como APIs, bases de datos, colas y caches.
- Tests de aceptación: reflejan el punto de vista del usuario de la aplicación.
- Tests de rendimiento
- Tests de infraestructura

Despliegue

El despliegue es el acto de instalar *software* y prepararlo para el uso dentro de un entorno de servidor. Cada proceso de despliegue puede ser diferente dependiendo de la pila de tecnologías que se estén utilizando. El despliegue comprende el proceso de descargar el artefacto de donde se encuentre almacenado, instalarlo en el servidor y los pasos necesarios para levantar el servicio. El proceso de despliegue concluye cuando la nueva versión está ejecutándose y la antigua ha sido deshabilitada.

Jenkins: el servidor de automatización de código abierto

Jenkins es un servidor de automatización escrito en Java. Es, por mucho, el *software* más utilizado para implementar CI/CD. Gracias a la gran cantidad de *plugins* disponibles, puede ser utilizado en una gran cantidad de casos de uso.
Jenkins tiene *plugins* para casi cualquier tarea concebible. Los *plugins* se usan para mandar notificaciones, coordinar versiones y ejecutar *jobs* programados, entre otras cosas.

Conceptos básicos de Jenkins

En su núcleo, Jenkins es un servidor de coordinación que une varias herramientas en una cadena o *pipeline*. Jenkins es solo el organizador. El resto del trabajo depende de servicios externos como repositorios de código, compiladores, herramientas de construcción, de *testing* y sistemas de despliegue.
Un *job* de Jenkins, o proyecto, es una colección de *stages* unidos. Estos *stages* pueden ejecutarse en paralelo, en serie o, incluso, de manera condicional, dependiendo de los resultados del *stage* anterior.
Cada *job* de Jenkins debe estar conectado a un repositorio de código. Una vez hecho esto, se pueden definir los disparadores que ejecutarán ese *job*. Después de esto, se especifican los *stages* que seguirá ese *job*, que suelen coincidir con los *stages* de un proceso de CI/CD.

Pipeline como código

Hemos descrito el proceso de crear proyectos de Jenkins uniendo *stages* desde la interfaz de usuario de Jenkins. Es la forma más rápida de empezar, pero desde el punto de vista de la infraestructura es

un poco opaca. Todo el código es manejado por Jenkins, y si el servidor de Jenkins se pierde, se deberán cargar los datos desde el *backup* (copia segura de los datos) más reciente.
La versión 2 de Jenkins introduce una nueva característica, llamada *pipeline* de Jenkins. Un *pipeline* de Jenkins codifica los pasos de un proyecto de manera declarativa, con un lenguaje de dominio específico basado en el lenguaje de programación Groovy.

Imágenes de contenedor como artefactos

El producto de un *build* puede ser una imagen desplegable en un sistema de orquestación de contenedores. Los contendores son ligeros y portables. Mover imágenes entre sistemas, utilizando un registro de imágenes, es fácil y rápido. Cualquier herramienta de CI/CD puede adoptar la estrategia de producir imágenes como artefactos. Los pasos a seguir son:

1. Construir la aplicación dentro de contenedores.
2. Crear una o varias imágenes de contenedor que incluyan la aplicación y sus dependencias.
3. Subir las imágenes a un registro de imágenes.
4. Desplegar las imágenes en un entorno de ejecución que soporte contenedores.

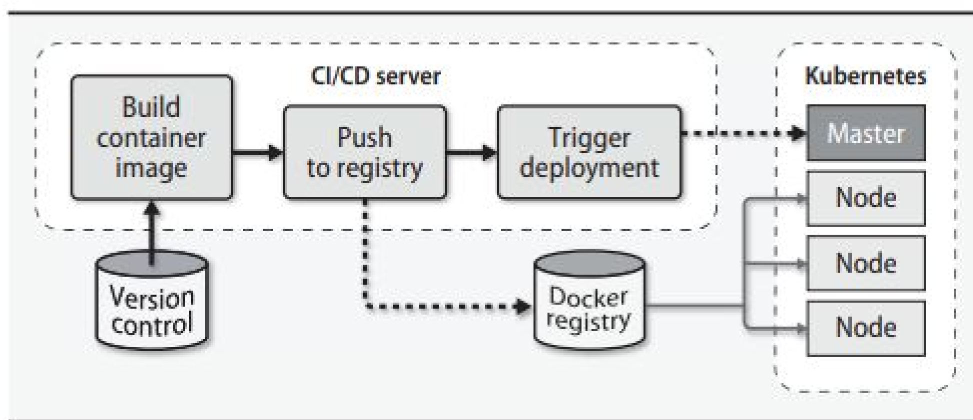

Figura 6. Proceso de despliegue basado en contenedores. (Fuente: Nemeth, E., Snyder, G., Hein, T. R., Whaley, B., & Mackin, D. (2017). *UNIX and Linux System Administration Handbook (5th Edition)* (5.a ed.) New York, USA: Addison-Wesley Professional)

5. Desarrollo del caso de estudio

En este capítulo se detallan las etapas del proceso de desarrollo de un producto *software*.

Análisis

En este apartado se realiza el análisis de requisitos de la aplicación, se muestran los actores y se modelan los casos de uso.

Actores

Respecto a la aplicación existen tres tipos de actores:

- Usuario no registrado: este tipo de usuarios representa al usuario cliente de la aplicación que no tiene una sesión iniciada en el portal.
- Usuario registrado: representa al usuario cliente que tiene una cuenta asociada a su correo electrónico y ha iniciado sesión en la aplicación.
- Administrador del sitio: representa a un usuario registrado, pero con permisos de administrador sobre la aplicación web.
- Administrador de la infraestructura: representa a un encargado del equipo de operaciones y despliegue de nuestra empresa.

Casos de uso

En esta sección se muestran los casos de uso extraídos de los requisitos solicitados por el cliente, divididos en función de los usuarios de la aplicación.

Usuario no registrado

US - 01	El usuario no registrado podrá registrarse en la aplicación
US - 02	El usuario no registrado podrá ver los tours disponibles y su información
US - 03	El usuario no registrado podrá ver los eventos disponibles y su información
US - 04	El usuario no registrado podrá realizar búsquedas por artista o por nombre de tour

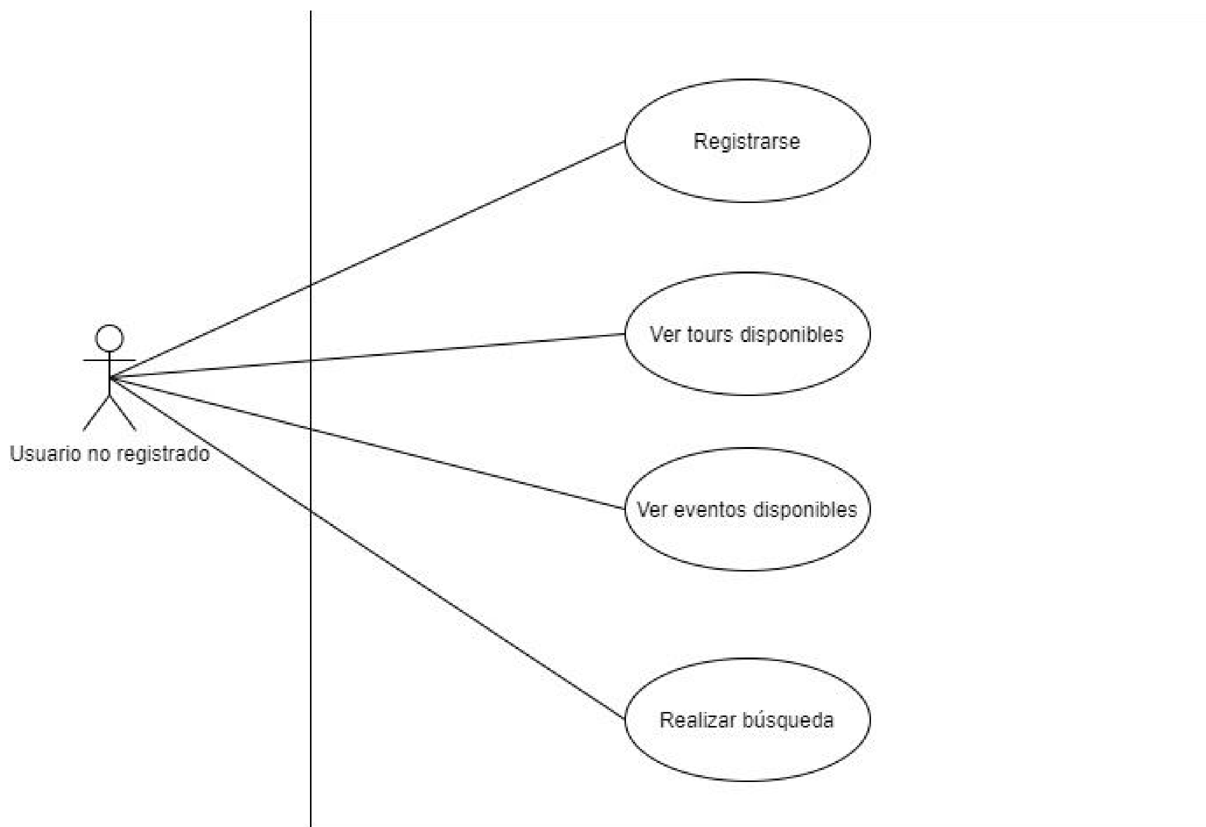

Figura 7. Casos de uso para usuario no registrado

Usuario registrado

US - 05	El usuario registrado podrá iniciar sesión en la aplicación
US - 06	El usuario registrado podrá cerrar sesión en la aplicación
US - 07	El usuario registrado podrá visualizar su información de usuario y editarla
US - 08	El usuario registrado podrá comprar entradas a eventos
US - 09	El usuario registrado podrá descargar en PDF su entrada
US - 10	El usuario registrado podrá visualizar los eventos de los que ha comprado entradas

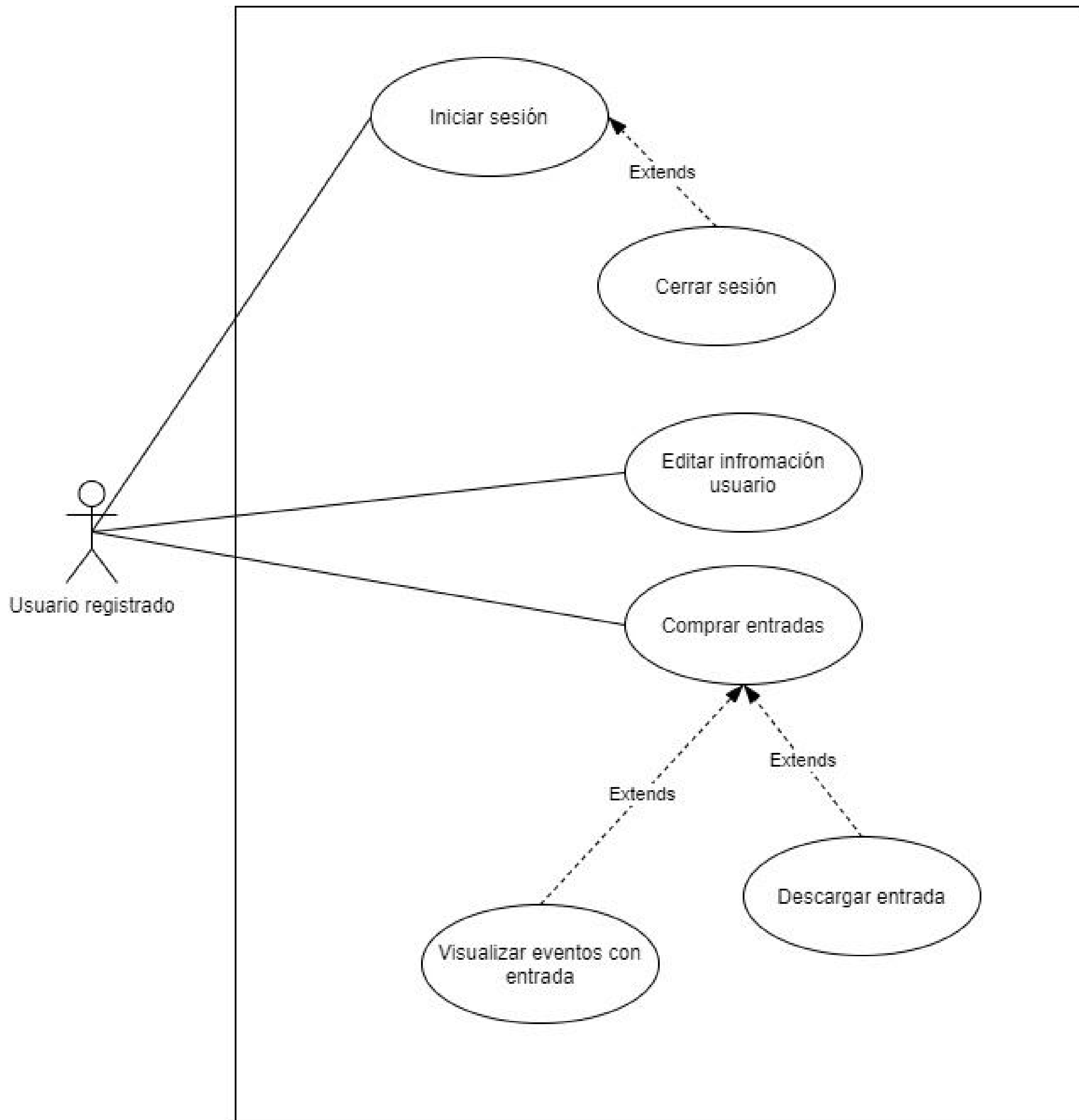

Figura 8. Casos de uso para usuario registrado

Administrador del sitio

US - 11	El usuario administrador podrá visualizar las localizaciones disponibles
US - 12	El usuario administrador podrá crear localizaciones
US - 13	El usuario administrador podrá editar localizaciones
US - 14	El usuario administrador podrá eliminar localizaciones
US - 15	El usuario administrador podrá visualizar los artistas disponibles

US - 16	El usuario administrador podrá crear artistas
US - 17	El usuario administrador podrá editar artistas
US - 18	El usuario administrador podrá eliminar artistas
US - 19	El usuario administrador podrá visualizar los tours disponibles
US - 20	El usuario administrador podrá crear tours
US - 21	El usuario administrador podrá editar tours
US - 22	El usuario administrador podrá eliminar tours
US - 23	El usuario administrador podrá visualizar los eventos disponibles
US - 24	El usuario administrador podrá crear eventos
US - 25	El usuario administrador podrá editar eventos
US - 26	El usuario administrador podrá eliminar eventos
US - 27	El usuario administrador podrá visualizar los usuarios registrados de la aplicación
US - 28	El usuario administrador podrá eliminar cuentas de usuarios que no sean administradores

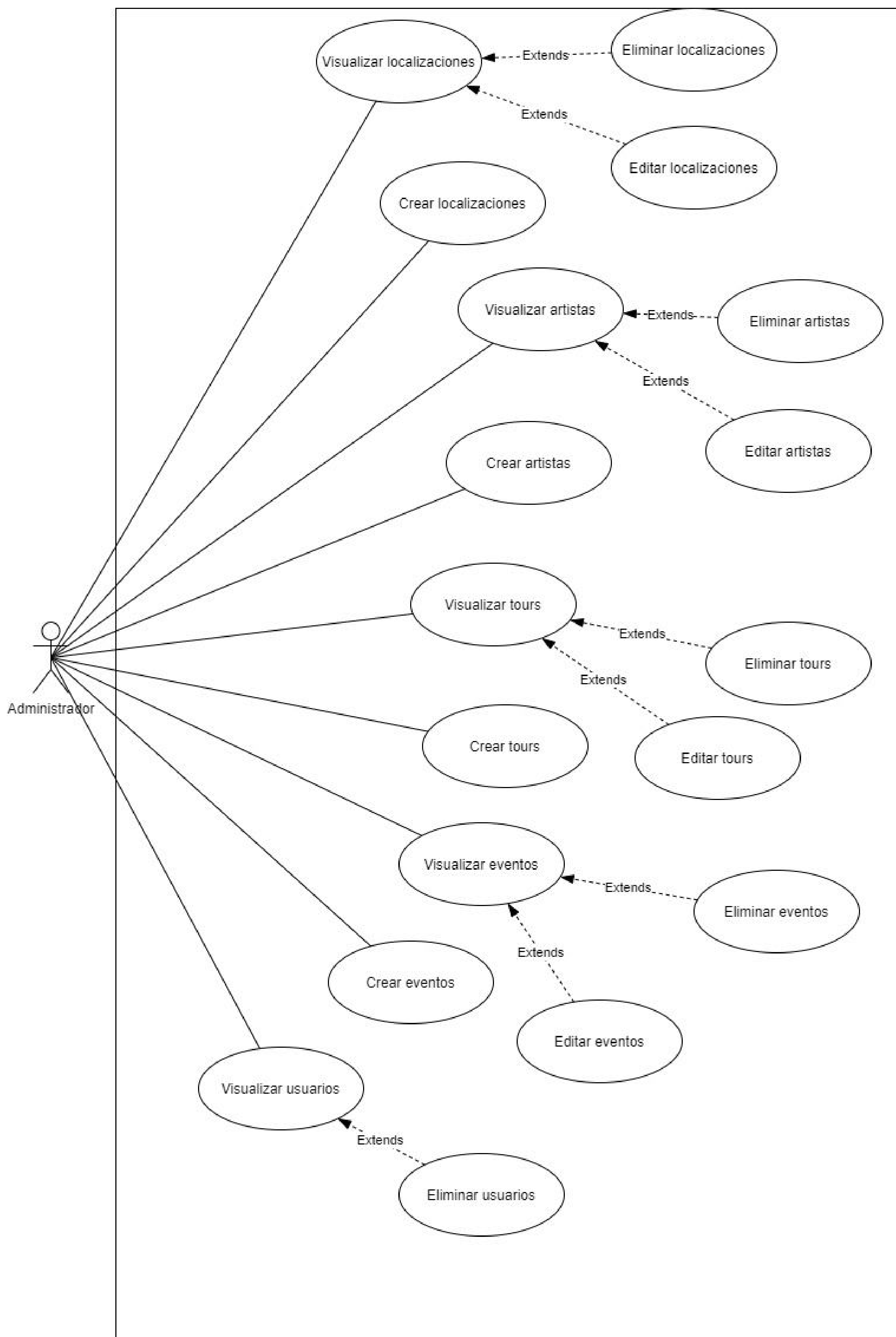
Visualizar localizaciones
Extends
Eliminar localizaciones
Extends
Editar localizaciones
Crear localizaciones
Visualizar artistas
Extends
Eliminar artistas
Extends
Editar artistas
Crear artistas
Visualizar tours
Extends
Eliminar tours
Extends
Editar tours
Crear tours
Administrador
Visualizar eventos
Extends
Eliminar eventos
Extends
Editar eventos
Crear eventos
Visualizar usuarios
Extends
Eliminar usuarios

Figura 9. Casos de uso para usuario administrador del sitio

Administrador de la infraestructura

US - 29	El administrador de la infraestructura podrá crear cuentas de administrador del sitio

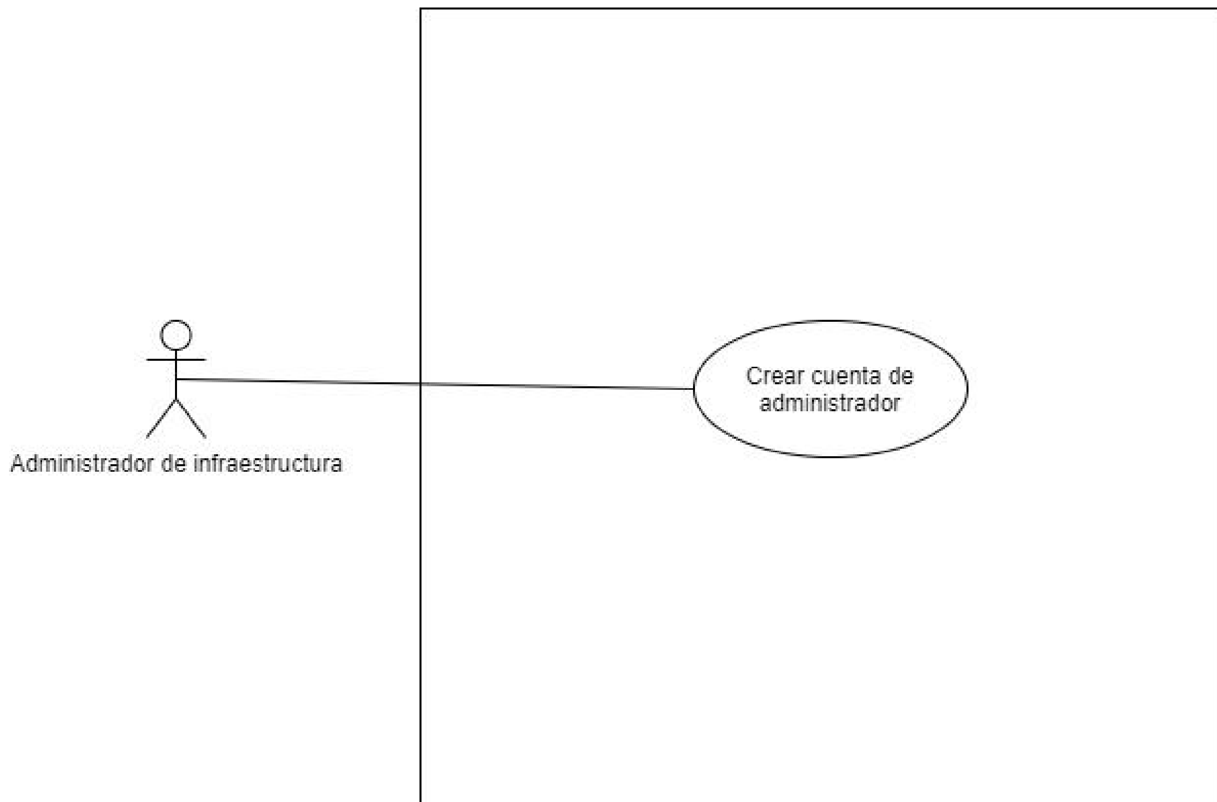

Figura 10. Casos de uso para usuario administrador de la infraestructura

Requisitos funcionales

La siguiente tabla define los requisitos funcionales que deberá cumplir el sistema.

RF - 01	El sistema muestra las localizaciones disponibles
RF - 02	El sistema muestra los artistas disponibles
RF - 03	El sistema muestra los tours disponibles
RF - 04	El sistema muestra los eventos disponibles
RF - 05	El sistema muestra los usuarios disponibles

RF - 06	El sistema muestra un formulario para crear y editar localizaciones
RF - 07	El sistema muestra un formulario para crear y editar artistas
RF - 08	El sistema muestra un formulario para crear y editar tours
RF - 09	El sistema muestra un formulario para crear y editar eventos
RF - 10	El sistema muestra un formulario para seleccionar el número de entradas a comprar
RF - 11	El sistema muestra un formulario para introducir los datos de la tarjeta de crédito
RF - 12	El sistema muestra un formulario para editar la información de usuario
RF - 13	El sistema muestra un formulario para registrarse como usuario
RF - 14	El sistema muestra un formulario para iniciar sesión
RF - 15	El sistema muestra los eventos para los que el usuario tiene entradas
RF - 16	El sistema muestra un formulario para realizar búsquedas de eventos y artistas
RF - 17	El sistema muestra notificaciones con el resultado de las operaciones realizadas
RF - 18	El sistema permite descargar las entradas como PDF
RF - 19	El sistema actualiza, introduce y elimina datos de la base de datos
RF - 20	El sistema valida datos antes de introducirlos en la base de datos
RF - 21	El sistema permite navegar hacia la página anterior
RF - 22	El sistema permite ir a la página principal

Requisitos de información

A continuación se muestran la información que debe mantener la aplicación para cumplir los requisitos del cliente.

RF - 01	El sistema debe almacenar el identificador de la localización
RF - 02	El sistema debe almacenar la ciudad en la que se sitúa la localización
RF - 03	El sistema debe almacenar la dirección en la que se sitúa la localización
RF - 04	El sistema debe almacenar la capacidad de aforo con la que cuenta la localización
RF - 05	El sistema debe almacenar el identificador del artista
RF - 06	El sistema debe almacenar el nombre del artista
RF - 07	El sistema debe almacenar la imagen del artista
RF - 08	El sistema debe almacenar el identificador del tour

RF - 09	El sistema debe almacenar el nombre del tour
RF - 10	El sistema debe almacenar la descripción del tour
RF - 11	El sistema debe almacenar el identificador del artista al que se refiere el tour
RF - 12	El sistema debe almacenar un conjunto de imágenes referentes al tour
RF - 13	El sistema debe almacenar el identificador del evento
RF - 14	El sistema debe almacenar el identificador de la localización sobre la que se realizará el evento
RF - 15	El sistema debe almacenar la fecha en la que se realizará el evento
RF - 16	El sistema debe almacenar el número de entradas que se han vendido del evento
RF - 17	El sistema debe almacenar el precio de la entrada del evento
RF - 18	El sistema debe almacenar el identificador del tour al que se refiere el evento
RF - 19	El sistema debe almacenar el identificador del usuario
RF - 20	El sistema debe almacenar el nombre del usuario
RF - 21	El sistema debe almacenar el apellido del usuario
RF - 22	El sistema debe almacenar el correo electrónico del usuario
RF - 23	El sistema debe almacenar la contraseña del usuario
RF - 24	El sistema debe almacenar la fecha de nacimiento del usuario
RF - 25	El sistema debe almacenar el rol del usuario
RF - 26	El sistema debe almacenar los eventos para los que el usuario dispone de entradas
RF - 27	El sistema debe almacenar los cargos que se han realizado en la aplicación a los usuarios

Diagrama Entidad-Relación

En esta sección se muestra el diagrama Entidad-Relación seguido por la aplicación. La estructura de datos de la aplicación puede ser fácilmente transportable a una estructura relacional, a pesar de que será implementada sobre un sistema de base de datos no relacional, como es MongoDB, que está orientado a documentos.

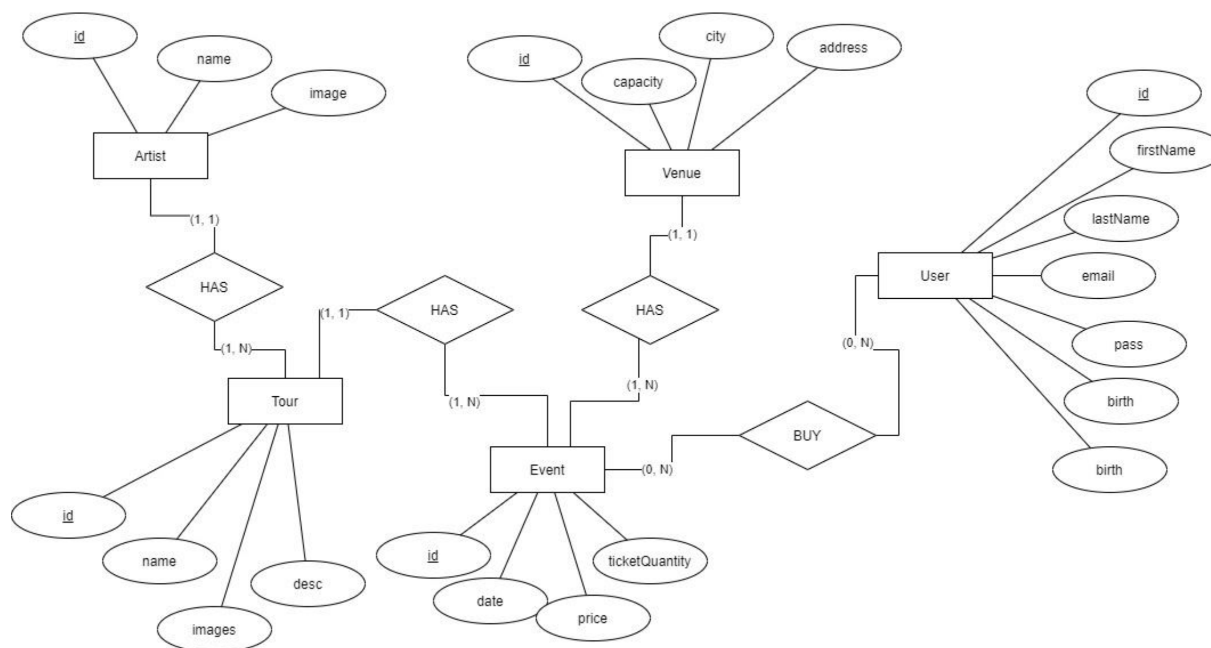

Figura 11. Modelo Entidad-Relación de la aplicación

Diseño

Para este proyecto, se ha optado por una solución con una arquitectura basada en microservicios, atendiendo a algunas de las ventajas que esta arquitectura nos brinda, entre ellas, mayor velocidad de desarrollo, gracias a la independencia entre los equipos de desarrollo, debido al desacoplamiento existente entre los diferentes módulos de la aplicación.

La aplicación se ha diseñado para tener tres microservicios principales. Estos microservicios serán:

- *Webapp*: será el encargado de servir los ficheros estáticos de la aplicación web, generados por React. Este microservicio se comunicará con las dos APIs REST realizando peticiones HTTP.
- *Main API*: será la encargada de la autenticación y autorización de usuarios, además de gestionar localizaciones, artistas, tours y eventos. Para todo ello tendrá conexión directa con la base de datos MongoDB.
- *Payments API*: será la encargada de interactuar con Stripe para manejar los pagos realizados por los clientes.

La arquitectura y la comunicación entre servicios será exactamente igual en todos los entornos de ejecución de la aplicación, ya sea levantando los contenedores con Docker Compose o con Kubernetes. La implementación de la arquitectura se mostrará en el siguiente capítulo. Esta comunicación se realizará a través de HTTP, exponiendo cada uno de los dos microservicios de *backend* su API REST. La arquitectura lógica de la aplicación, para cada uno de estos dos casos, se muestra a continuación:

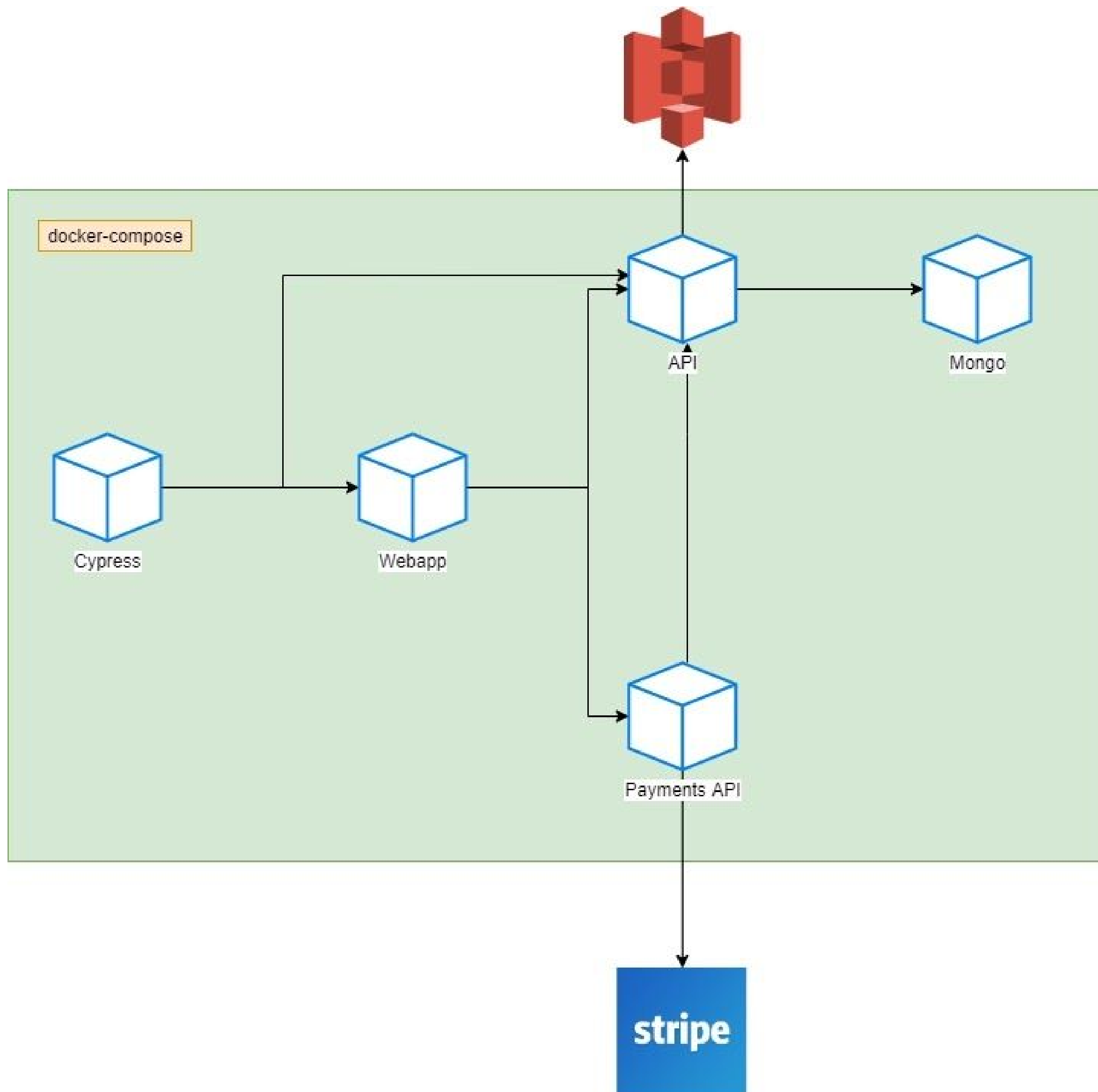

Figura 12: representación visual de la arquitectura lógica de la aplicación para el entorno de integración.

En la figura 12 se puede apreciar el contenedor de Cypress, ya que los tests se ejecutan también desde dentro de un contenedor.

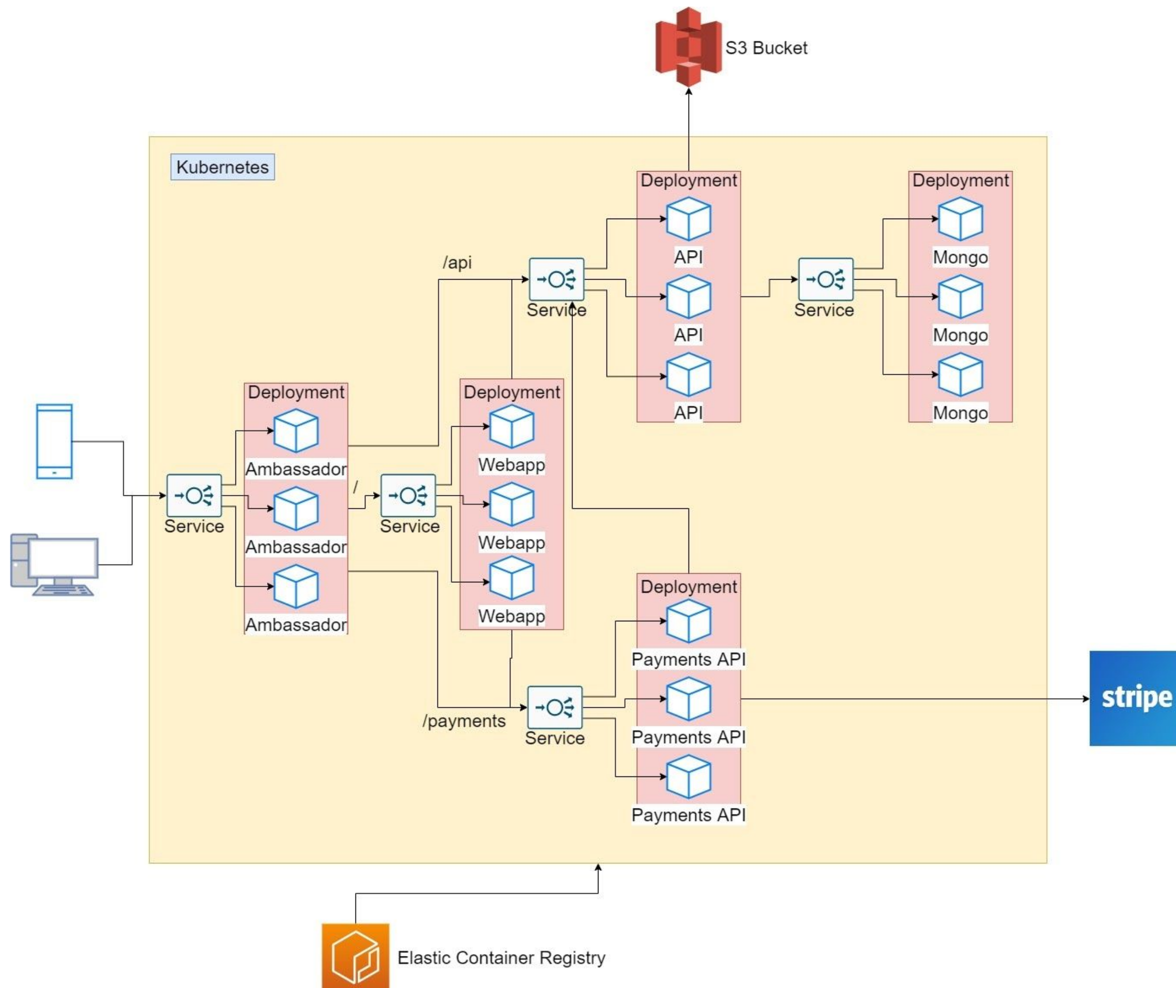

Figura 13: representación visual de la arquitectura lógica de la aplicación para el entorno de producción.

Implementación

Webapp

Este microservicio es el encargado servir los ficheros estáticos de la aplicación, almacenados en un Nginx, que es un servidor web de alto rendimiento. Estos ficheros estáticos son los ficheros que contienen el HTML, CSS y Javascript de la página.

Los ficheros Javascript son construidos por React, que es una librería de Javascript de código abierto utilizada construir interfaces de usuario. Es mantenida por Facebook y por la comunidad y está enfocada a crear aplicaciones “single-page”. Está basada en componentes completamente reutilizables y hace uso de JSX o Javascript XML, que es una extensión a la sintaxis del lenguaje Javascript. Es muy similar a HTML y provee una manera de estructurar el renderizado de componentes usando una sintaxis familiar para muchos desarrolladores.

Se utiliza React Bootstrap como framework CSS, que es el Bootstrap original reescrito en forma de componentes para trabajar de una forma óptima con React, y sin dependencias innecesarias como jQuery. Además, el código CSS escrito por nuestro equipo de frontend es empaquetado y

minimizado dentro de los ficheros estáticos por una librería llamada Styled-components, que está especialmente diseñada y optimizada para trabajar con componentes React.

Uno de los requisitos impuestos por la empresa cliente es que la interfaz de usuario sea una aplicación "single-page" (SPA). Esto quiere decir que la página web que interactúa con el navegador reescribiendo dinámicamente pequeñas partes del DOM (Document Object Model) con la nueva información provista por el servidor web (los dos microservicios del backend en este caso), en lugar del método tradicional, que consiste en cargar la página web entera. El objetivo es que las transiciones entre páginas sean más rápidas y la apariencia sea de una aplicación nativa. Así se mejora mucho el rendimiento del sitio, ya que todos los ficheros estáticos se descargan solamente la primera vez que se carga la página y se almacenan en la caché del navegador. La información extra se envía y recibe únicamente como respuesta a acciones concretas del usuario, de una forma ligera y rápida, usualmente en formato JSON a través de APIs REST.

Estructura del microservicio

La estructura inicial de este microservicio ha sido creada con "Create React App", que es una librería que crea un entorno de desarrollo para poder usar las últimas características de React. Por debajo usa Babel, que es una funcionalidad principalmente utilizada para convertir código escrito en ECMAScript 2015 o superior a código Javascript compatible con navegadores antiguos y actuales, y Webpack, que es una librería de código abierto utilizada como empaquetador de módulos. Webpack toma los módulos de la aplicación con sus dependencias, crea un grafo de dependencias y genera ficheros estáticos representando esos módulos.

```
daxevents/webapp/
├── Dockerfile
├── Dockerfile-develop
├── nginx.conf
├── package.json
├── package-lock.json
├── public
│   ├── config.js
│   ├── favicon.ico
│   ├── index.html
│   ├── logo192.png
│   ├── logo512.png
│   ├── manifest.json
│   └── robots.txt
├── README.md
└── src
    ├── App.jsx
    ├── App.test.js
    ├── components
    │   ├── Admin
    │   │   ├── Admin.component.jsx
    │   │   ├── Admin.container.js
```

```
│   │   └── Admin.styles.js
│   ├── AdminArtists
│   │   ├── AdminArtists.component.jsx
│   │   └── AdminArtists.container.js
│   ├── AdminCreateArtist
│   │   ├── AdminCreateArtist.component.jsx
│   │   └── AdminCreateArtist.container.js
│   ├── AdminCreateEvent
│   │   ├── AdminCreateEvent.component.jsx
│   │   └── AdminCreateEvent.container.js
│   ├── AdminCreateTour
│   │   ├── AdminCreateTour.component.jsx
│   │   └── AdminCreateTour.container.js
│   ├── AdminCreateVenue
│   │   ├── AdminCreateVenue.component.jsx
│   │   └── AdminCreateVenue.container.js
│   ├── AdminEditArtist
│   │   ├── AdminEditArtist.component.jsx
│   │   └── AdminEditArtist.container.js
│   ├── AdminEditEvent
│   │   ├── AdminEditEvent.component.jsx
│   │   └── AdminEditEvent.container.js
│   ├── AdminEditTour
│   │   ├── AdminEditTour.component.jsx
│   │   └── AdminEditTour.container.js
│   ├── AdminEditVenue
│   │   ├── AdminEditVenue.component.jsx
│   │   └── AdminEditVenue.container.js
│   ├── AdminEvents
│   │   ├── AdminEvents.component.jsx
│   │   └── AdminEvents.container.js
│   ├── AdminTours
│   │   ├── AdminTours.component.jsx
│   │   └── AdminTours.container.js
│   ├── AdminUsers
│   │   ├── AdminUsers.component.jsx
│   │   └── AdminUsers.container.js
│   ├── AdminVenues
│   │   ├── AdminVenues.component.jsx
│   │   └── AdminVenues.container.js
│   ├── CheckoutForm
│   │   └── CheckoutForm.component.jsx
│   ├── EventItem
│   │   ├── EventItem.component.jsx
│   │   ├── EventItem.container.js
│   │   └── EventItem.styles.js
```

```
│   ├── Events
│   │   ├── Events.component.jsx
│   │   └── Events.container.js
│   ├── Footer
│   │   ├── Footer.component.jsx
│   │   ├── Footer.container.js
│   │   └── Footer.styles.js
│   ├── FormErrComponent
│   │   ├── FormErrComponent.component.jsx
│   │   └── FormErrComponent.container.js
│   ├── Header
│   │   ├── Header.component.jsx
│   │   ├── Header.container.js
│   │   └── Header.styles.js
│   ├── Home
│   │   ├── Home.component.jsx
│   │   └── Home.container.js
│   ├── Login
│   │   ├── Login.component.jsx
│   │   └── Login.container.js
│   ├── Main
│   │   ├── Main.component.jsx
│   │   ├── Main.container.js
│   │   └── Main.styles.js
│   ├── NotificationPanel
│   │   ├── NotificationPanel.component.jsx
│   │   ├── NotificationPanel.container.js
│   │   └── NotificationPanel.styles.js
│   ├── PaymentInfo
│   │   ├── PaymentInfo.component.jsx
│   │   ├── PaymentInfo.container.js
│   │   └── PaymentInfo.styles.js
│   ├── Signup
│   │   ├── Signup.component.jsx
│   │   ├── Signup.container.js
│   │   └── Signup.styles.js
│   ├── TourItem
│   │   ├── TourItem.component.jsx
│   │   ├── TourItem.container.js
│   │   └── TourItem.styles.js
│   ├── Tours
│   │   ├── Tours.component.jsx
│   │   ├── Tours.container.js
│   │   └── Tours.styles.js
│   ├── UpcomingEvents
│   │   ├── UpcomingEvents.component.jsx
```

```
│   │   └── UpcomingEvents.container.js
│   ├── UserEvents
│   │   ├── UserEvents.component.jsx
│   │   └── UserEvents.container.js
│   └── UserInfo
│       ├── UserInfo.component.jsx
│       └── UserInfo.container.js
├── index.css
├── index.js
├── logo.svg
├── serviceWorker.js
└── utils
    ├── isAdmin.js
    ├── moment.js
    ├── sample-data
    │   ├── events.json
    │   └── tours.json
    └── styles
        ├── global.js
        └── helpers.js

37 directories, 96 files
```

Detalles de implementación

Como requisito del cliente para la aplicación, el usuario registrado debe poder descargar en un PDF la información de su entrada en forma de código QR. Este código se mostrará a la entrada del evento y un lector de códigos QR validará que la entrada es válida. El código que permite hacer esto es el siguiente:

```
<tr className="qr-code">
    <td>QR code:</td>
    <td>
        <QRCode
            id="qr-canvas"
            value={`user: ${
                user._id
            }\nticketQuantity: ${ticketQuantity}\namount: ${amount /
                100}€\ntour: ${tour._id}\nevent: ${event._id}`}
        />
    </td>
</tr>
```

Este código pertenece al componente *PaymentInfo*, que es el encargado de mostrar en pantalla la información de la entrada, una vez que se ha hecho la compra, para verificar el resultado. Utilizando

una librería de NPM llama `qrcode.react`, dibujamos en un elemento `canvas` el código QR con la información de la entrada contenida en formato cadena de caracteres.
Gracias a otra librería llamada `jspdf`, podemos introducir ese código QR generado anteriormente en un archivo PDF descargable por el usuario. Este es el código que permite esa funcionalidad, que pertenece al mismo componente:

```
<Button
    onClick={e => {
        const filename = 'DaxEvents-Tickets.pdf';

        let pdf = new jsPDF();
        pdf.addImage(
document.querySelector('#qr-canvas').toDataURL('image/jpeg'),
            'JPEG',
            10,
            10,
            50,
            50
        );
        pdf.save(filename);

        e.preventDefault();
    }}
>
```

Dockerfile

En esta sección se detalla el Dockerfile utilizado para construir la imagen de contenedor de este microservicio. El Dockerfile utiliza como imagen de base node:13-slim, que es una imagen que contiene Nodejs v13.14 instalado sobre una versión Linux basada en Debian. "Slim" quiere decir que la imagen base trae únicamente los archivos necesarios para ejecutar Nodejs, lo que se traduce en una imagen final con un tamaño muy reducido y un número relativamente bajo de vulnerabilidades de seguridad.
Como se puede apreciar, este Dockerfile tiene dos sentencias `FROM`. Esto quiere decir que estamos ante lo que en Docker se conoce como *multistage builds*. La imagen se construye en dos fases:

1. Se copian los ficheros que contienen el código del microservicio dentro de la imagen, se instalan todas las dependencias necesarias para el proceso y después se ejecuta el *build*, que indica a React que debe generar todos los ficheros estáticos (HTML, CSS y Javascript) de la aplicación, que una vez construidos, se guardan en la carpeta `/app/build`.
2. La segunda fase toma toma como imagen de base una imagen mantenida por Nginx, que contiene Nginx v1.17.5 instalado en su interior. Se toman los archivos estáticos generados en la fase anterior y se copian al interior de la imagen, así como el fichero que define la configuración de Nginx. Se expone el puerto 80 de la imagen al exterior y se arranca Nginx en segundo plano.

```
FROM node:13-slim as react-build
WORKDIR /app
COPY . .

RUN apt-get update; apt-get install git -y; \
    npm i --production && npm run build

FROM nginx:1.19.0

COPY --from=react-build /app/build /usr/share/nginx/html

COPY nginx.conf /etc/nginx/conf.d/default.conf

EXPOSE 80

CMD ["/bin/bash", "-c", "nginx -g \"daemon off;\""]
```

Main API

La API principal es la encargada de interactuar con la base de datos para manejar los datos persistentes de la aplicación. Esta interacción con la base de datos MongoDB se realiza a través de Mongoose, que es un ODM (Object Document Mapper) para Nodejs. Gracias a Mongoose, los datos de la aplicación se exponen a través de esta API para poder ser accedidos desde otros microservicios, principalmente desde el microservicio del *frontend*, Webapp.
Esta exposición se realiza usando el protocolo HTTP, gracias la API REST construída sobre Express. Estas rutas HTTP públicas gestionan las operaciones de listado, creación, modificación y borrado de las localizaciones, los artistas, los tours y los eventos. Además, también se encarga de recibir las peticiones de registro y de autenticación y autorización de usuarios desde el *frontend* y de validarlas contra la base de datos.
La gestión de usuarios registrados se realiza utilizando JSON Web Tokens (JWT). Cuando se recibe una petición de inicio de sesión, utilizando el correo electrónico y la contraseña, estos datos son comparados con los almacenados en la base de datos. Las contraseñas en la base de datos se guardan, es decir, se almacena el hash de la contraseña y de un valor aleatorio que asegura que dos contraseñas iguales no generen nunca el mismo hash. Si los valores de usuario y contraseña de la base de datos coinciden con los introducidos por el usuario, el microservicio genera un *token* JWT cifrado y lo envía al usuario. El navegador del usuario almacena este JWT en una cookie y lo envía cada vez que el usuario realiza una petición. Por cada petición, el microservicio descifra el *token* y comprueba que el usuario y sus permisos son válidos para acceder al recurso seleccionado.
Además este microservicio se conecta con el servicio de almacenamiento de archivos gestionado por AWS, Simple Storage Service (S3), y almacena las imágenes asociadas a los artistas y los tours para no ocupar espacio extra en los contenedores.

Estructura del microservicio

La estructura de carpetas de este microservicio ha sido generada con express-generator, una librería de Nodejs para generar fácilmente los archivos necesarios para utilizar con Express, que es un framework de Nodejs para crear aplicaciones web.

```
daxevents/api/
├── Dockerfile
├── Dockerfile-develop
├── package.json
├── package-lock.json
├── requests
│   ├── artists
│   │   ├── create-artist.http
│   │   ├── delete-artist.http
│   │   ├── list-artist.http
│   │   └── update-artist.http
│   ├── auth
│   │   └── login.http
│   ├── events
│   │   ├── create-event.http
│   │   ├── delete-event.http
│   │   ├── list-event.http
│   │   └── update-event.http
│   ├── request.sh
│   ├── tours
│   │   ├── create-tour.http
│   │   ├── delete-tour.http
│   │   ├── list-tour.http
│   │   └── update-tour.http
│   ├── users
│   │   ├── create-user.http
│   │   ├── delete-user.http
│   │   ├── list-users.http
│   │   └── update-user.http
│   └── venues
│       ├── create-venue.http
│       ├── delete-venue.http
│       ├── list-venues.http
│       └── update-venue.http
└── src
    ├── api
    │   ├── models
    │   │   ├── artist.js
    │   │   ├── event.js
    │   │   ├── tour.js
```

```
│   │   ├── user.js
│   │   └── venue.js
│   └── routes
│       ├── artists.js
│       ├── auth.js
│       ├── events.js
│       ├── tours.js
│       ├── users.js
│       └── venues.js
├── app.js
├── bin
│   └── www.js
├── manage
│   └── manage.js
├── populate
│   ├── populate-db.js
│   └── users.csv
└── utils
    ├── error.js
    ├── isAdmin.js
    ├── passport.js
    ├── remove.js
    └── s3.js

15 directories, 47 files
```

Detalles de implementación

Para la autenticación y autorización de usuarios, nuestra aplicación utiliza JSON Web Tokens. Para el manejo de estos *tokens* se utiliza dos librerías de NPM, llamadas `passport` y `jsonwebtoken`. El código encargado de validar el *token* enviado en cada petición y autorizar al usuario contiene lo siguiente:

```
import { Strategy as JwtStrategy } from "passport-jwt";
dotenv.config();

let cookieExtractor = function(req) {
    var token = null;
    if (req && req.cookies) {
        token = req.cookies["jwt"];
    }
    return token ? token.split(" ")[1] : null;
};

const opts = {
    jwtFromRequest: cookieExtractor,
```

```
    secretOrKey: process.env.JWT_SECRET
};

module.exports = passport => {
    passport.use(
        new JwtStrategy(opts, async (payload, done) => {
            try {
                const user = await User.findById(payload.user._id);
                if (user) done(null, user);
                else done(null, false);
            } catch (err) {
                console.log(err);
            }
        })
    );
};
```

Adicionalmente, el almacenamiento de las imágenes asociadas a tours y artistas se realiza utilizando S3, el servicio de almacenamiento de AWS. Para la conexión de nuestra aplicación con dicho servicio se ha utilizado el SDK que AWS provee para Nodejs. La clase utilizada para ello contiene el siguiente código:

```
import AWS from "aws-sdk";
import fs from "fs";
const dotenv = require("dotenv");

class S3Manager {
    constructor() {
        dotenv.config();

        AWS.config.update({ region: "eu-west-1" });
        this.s3 = new AWS.S3({ apiVersion: "2006-03-01" });
    }

    deleteFile(file, folder) {
        return new Promise((resolve, rejects) => {
            const params = {
                Bucket: process.env.BUCKET_NAME,
                Key: `${folder}/${file}`
            };

            this.s3.deleteObject(params, function(err, data) {
                if (err) rejects(err);
                else resolve(data); // successful response
            });
```

```
        });
    }

    uploadFile(file, folder) {
        return new Promise((resolve, rejects) => {
            const params = {
                Bucket: process.env.BUCKET_NAME,
                Key: `${folder}/${file.originalname}`, // File name you
want to save as in S3
                Body: fs.createReadStream(file.path),
                ACL: "public-read"
            };

            // Uploading files to the bucket
            this.s3.upload(params, function(err, data) {
                if (err) {
                    rejects(err);
                }
                resolve(data);
            });
        });
    }

    getFile(key) {
        return new Promise((resolve, rejects) => {
            var getParams = {
                Bucket: process.env.BUCKET_NAME, //replace example
bucket with your s3 bucket name
                Key: key // replace file location with your s3 file
location
            };

            this.s3.getObject(getParams, function(err, data) {
                if (err) {
                    rejects(err);
                } else {
                    resolve(data.Body.toString());
                }
            });
        });
    }
}

export default S3Manager;
```

Esta clase nos permite borrar archivos, guardar archivos y obtener la URL a esos archivos para mostrarlos en nuestra aplicación.

Dockerfile

Este Dockerfile utiliza como imagen base la versión slim de Nodejs v13. Sobre ella instala los paquetes necesarios para compilar la librería de NPM `bcrypt`, la encargada se securizar las contraseñas. Una vez hecha la compilación, se instalan las dependencias de la aplicación desde NPM y se eliminan los paquetes que ya no son necesarios para evitar vulnerabilidades y que la imagen pese más de lo necesario. En los siguientes pasos se copia el código de la aplicación dentro de la imagen y se ejecuta la construcción, que transforma el código de la última versión del lenguaje Javascript a una versión compatible con Nodejs. Este paso es necesario para poder utilizar las últimas funcionalidades del lenguaje Javascript que aún no están soportadas nativamente por los navegadores ni por Nodejs. Después se expone un puerto fuera del contenedor para comunicación con el exterior y se arranca el proceso de servidor de Express.

```
FROM node:13-slim

WORKDIR /app

COPY package*.json ./

RUN apt-get update; \
    apt-get install -y build-essential python; \
    npm i node-gyp;\
    npm i --production;\
    apt-get purge --auto-remove -y build-essential python

COPY src /app/src
COPY .babelrc /app/.babelrc

RUN npm run build

EXPOSE 3000

CMD [ "npm", "run", "serve" ]
```

Payments API

Esta API está diseñada para administrar toda la información relacionada con los pagos. Es una API REST construida sobre Express, que integra la librería de Stripe para Nodejs como pasarela de pagos. Esta librería de Stripe maneja la comunicación con su API se una manera fácil y eficiente, de manera que podamos realizar cargos en las tarjetas de crédito de los clientes a la hora de comprar entradas y llevar un registro de los pagos que se han recibido.

Estructura del microservicio

La estructura de carpetas de este microservicio ha sido generada con express-generator, una librería de Nodejs para generar fácilmente los archivos necesarios para utilizar con Express, que es un framework de Nodejs para crear aplicaciones web.

```
daxevents/payments-api/
├── Dockerfile
├── Dockerfile-develop
├── package.json
├── package-lock.json
├── requests
│   └── create-payment.http
└── src
    ├── api
    │   └── index.js
    ├── app.js
    └── bin
        └── www.js

4 directories, 8 files
```

Detalles de implementación

Para la comunicación con Stripe, hemos utilizado la librería de NPM con el mismo nombre. El siguiente código nos permite crear cargos en las tarjetas de crédito introducidas por los clientes:

```
req.stripe.charges
    .create({
        amount: Math.round(price * ticketQuantity * 100),
        currency: "eur",
        description: `Charge for ${email}`,
        source: token.id,
        metadata: {
            user: userId,
            event: eventId
        }
    })
    .then(resp => {
        if (resp.status === "succeeded") {
            res.json({
                charge: resp,
                body: {
                    ticketQuantity: ticketQuantity,
                    user,
                    event
```

```
                }
            });
        } else {
            res.status(500).json(resp);
        }
    })
    .catch(err => {
        res.status(500).json(err);
    });
```

Dockerfile

Este Dockerfile utiliza como imagen base la versión slim de Nodejs v13. Después se instalan las dependencias de la aplicación desde NPM. En los siguientes pasos se copia el código de la aplicación dentro de la imagen y se ejecuta la construcción, que transforma el código de la última versión del lenguaje Javascript a una versión compatible con Nodejs. Este paso es necesario para poder utilizar las últimas funcionalidades del lenguaje Javascript que aún no están soportadas nativamente por los navegadores ni por Nodejs. Después se expone un puerto fuera del contenedor para comunicación con el exterior y se arranca el proceso de servidor de Express.

```
FROM node:13-slim

WORKDIR /app

COPY package*.json ./

RUN npm i --production

COPY src /app/src
COPY .babelrc /app/.babelrc

RUN npm run build

EXPOSE 5000

CMD [ "npm", "run", "serve" ]
```

Pruebas

Este proyecto cuenta con una *suite* de tests automatizados que comprueban la integración entre los diferentes componentes de la aplicación, como son los propios microservicios, Stripe y S3. Estos tests están escritos en Javascript y son ejecutados con Cypress sobre Nodejs. Cypress es utilizado tanto para ejecutar los tests como para realizar la carga de datos, tanto a través del *frontend* como contra el *backend*. La estructura de carpetas utilizada para esta tarea es la siguiente:

```
daxevents/e2e/
├── cypress
│   ├── fixtures
│   │   ├── admin-user.json
│   │   ├── artists.json
│   │   ├── customer-user.json
│   │   ├── events.json
│   │   ├── images
│   │   │   ├── aitana.jpg
│   │   │   ├── aitana-tour2.jpg
│   │   │   ├── aitana-tour.jpg
│   │   │   .
│   │   │   .
│   │   │   .
│   │   │   ├── wiz.jpg
│   │   │   ├── wiz-tour2.jpg
│   │   │   └── wiz-tour3.jpg
│   │   ├── tours.json
│   │   └── venues.json
│   ├── integration
│   │   ├── 1-admin
│   │   │   ├── events-spec.js
│   │   │   └── users-spec.js
│   │   └── 2-user
│   │       ├── 1-signin-login.js
│   │       ├── 2-home-page.js
│   │       ├── 3-tickets-spec.js
│   │       ├── 4-search-spec.js
│   │       ├── 5-profile-spec.js
│   │       └── 6-pastevents-spec.js
│   ├── plugins
│   │   └── index.js
│   ├── population
│   │   └── populate.js
│   └── support
│       ├── commands.js
│       └── index.js
├── cypress.json
├── Dockerfile
├── package.json
├── package-lock.json
├── populate-local.json
└── remote.json

9 directories, 65 files
```

Los tests están divididos en dos partes, la parte del administrador y la parte de los clientes, que se corresponden con las carpetas `1-admin` y `2-user`, respectivamente. Dentro de los tests para la parte del administrador, se realizan los siguientes tests:

- ✓ Comprobar que se pueden listar las localizaciones.
- ✓ Comprobar que se pueden crear localizaciones.
- ✓ Comprobar que se pueden eliminar las localizaciones creadas.
- ✓ Comprobar que se pueden listar los artistas.
- ✓ Comprobar que se pueden crear artistas.
- ✓ Comprobar que se pueden eliminar los artistas creados.
- ✓ Comprobar que se pueden listar los tours.
- ✓ Comprobar que que se pueden crear tours.
- ✓ Comprobar que se pueden eliminar los tours creados.
- ✓ Comprobar que se pueden listar los eventos y que se ordenan por fecha.
- ✓ Comprobar que se pueden crear eventos.
- ✓ Comprobar que no se pueden crear eventos que no cumplan las condiciones, como por ejemplo con fechas ya pasadas.
- ✓ Comprobar que se pueden eliminar los eventos creados.
- ✓ Comprobar que se pueden listar los usuarios registrados.
- ✓ Comprobar que se pueden eliminar los usuarios con rol de cliente pero no de administrador.

Dentro de los tests para la parte del cliente, se realizan los siguientes tests:

- ✓ Comprobar que se pueden registrar usuarios con rol de cliente.
- ✓ Comprobar que se puede iniciar sesión.
- ✓ Comprobar que se puede cerrar sesión.
- ✓ Comprobar que no se pueden crear usuarios que no cumplan ciertas condiciones, como tener menos de 18 años.
- ✓ Comprobar que se pueden listar los tours desde la página principal.
- ✓ Comprobar que se pueden comprar entradas y la información se muestra correctamente.
- ✓ Comprobar que se pueden descargar las entradas como PDF.
- ✓ Comprobar que aparecen los eventos para los que el usuario dispone de entradas en su panel de usuario.
- ✓ Comprobar que se pueden realizar búsquedas por nombre de tour y de artista y los resultados son correctos.
- ✓ Comprobar que se puede modificar la información de usuario, como poder cambiar la contraseña y después volver a iniciar sesión.
- ✓ Comprobar que los eventos con fecha pasada se muestran en rojo en el panel de administración.
- ✓ Comprobar que los eventos con fecha pasada no se muestran para los clientes.

A continuación se muestran algunos extractos de código para mostrar cómo se realizan algunas tareas concretas dentro de una herramienta de tests automatizados:

- Iniciar sesión

```
cy.request("POST", `${Cypress.config("apiUrl")}/auth/login`, {
    email,
    pass,
}).then(({ body }) => {
    cy.setCookie("user", JSON.stringify(body.user));
});
```

- Cerrar sesión

```
cy.contains(`Hi, ${firstName}`).click();
cy.contains(`Log out`).click();
cy.get(".button.login").should("be.visible");
cy.get(".button.signup").should("be.visible");
```

- Crear evento

```
cy.get(".create-event-form #date").clear().type(event.date);
cy.get(".create-event-form #tour-dropdown").click();
cy.contains(event.tour).click();
cy.get(".create-event-form #venue-dropdown").click();
cy.contains(event.venue).click();
cy.get(".create-event-form #ticketQuantity")
    .clear()
    .type(event.ticketMaxQuantity);
cy.get(".create-event-form #price").clear().type(event.price);
cy.get(".create-event-form").submit();
```

- Comprobar número de usuarios

```
cy.request(`${Cypress.config("apiUrl")}/users`).as("users");
cy.get("@users").then(({ body }) => {
   cy.get(".users-list .users-list-item").should(
        "have.length",
        body.length
    );
});
```

6. Propuesta de arquitectura con soporte CI/CD para nuestro caso de estudio

Como se ha expuesto en la sección de diseño, para este proyecto se ha utilizado una arquitectura basada en microservicios. Estos microservicios serán empaquetados en contenedores de Docker utilizando Dockerfiles escritos por nuestro equipo, para convertir estos servicios en imágenes de contenedor, que se trasladarán entre diferentes entornos de ejecución. Para ejecutar estas imágenes en cada uno de los diferentes entornos se utilizarán tanto Docker Compose como Kubernetes.
En esta sección se expondrá cómo se han aplicado el flujo de CI/CD al proyecto de construir una aplicación web para la gestión y venta de entradas para eventos culturales solicitada por el cliente.

Entornos

Para la realización de este proyecto se utilizarán tres entornos de ejecución. Estos tres entornos tienen las siguientes características:

- Entorno de desarrollo o `dev`: este entorno representa la máquina de cada uno de los desarrolladores del equipo. Gracias a Docker Compose, cada desarrollador puede levantar todos los contenedores de los que está compuesta la aplicación en su máquina y ejecutar los tests en local antes de subir los cambios al repositorio de código.
- Entorno de integración o `stage`: este entorno se ejecuta en el servidor de integración, es decir, el servidor Jenkins de la empresa. Jenkins utiliza también Docker Compose para levantar los contenedores que componen la aplicación, representando un entorno de ejecución lo más parecido posible al entorno final de producción. Desde este entorno se ejecutan los tests necesarios para asegurar la calidad de la aplicación.
- Entorno de producción o `prod`: este entorno se aloja en el clúster de Kubernetes de la empresa y será el utilizado por los usuarios finales de la aplicación.

Pipeline

Todos los *stages* del *pipeline* de CI/CD de nuestro proyecto se ejecutan sobre Jenkins, es decir, sobre nuestro servidor de integración.

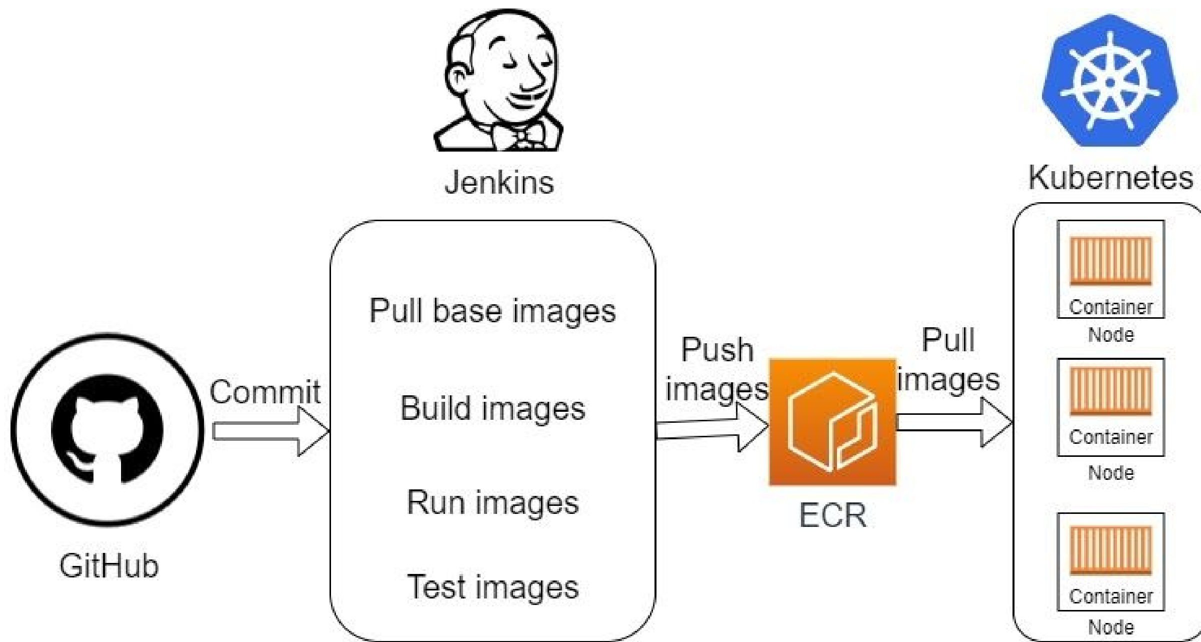

Figura 14. Flujo que siguen las imágenes de Docker, desde que se construyen hasta que se despliegan en el entorno de producción.

Como se muestra en la figura 14, las imágenes de contenedor siguen un flujo bien definido antes de llegar al entorno de producción. Desde un punto de vista de abstracción alto, las imágenes pasan por las siguientes fases, atendiendo al proceso de *pipeline* descrito en el capítulo anterior:

1. La construcción de las imágenes de Docker se ejecuta como respuesta a un cambio en la rama de desarrollo en el repositorio de código del proyecto en Github.
2. Las imágenes construidas en el paso anterior son testeadas en búsqueda de cualquier error que pueda haber sido introducido con los últimos cambios en el código.
3. Si los tests se han pasado correctamente, estas imágenes son desplegadas en el entorno de producción y son testeadas de nuevo, para descubrir cualquier posible error en la configuración de despliegue de Kubernetes.

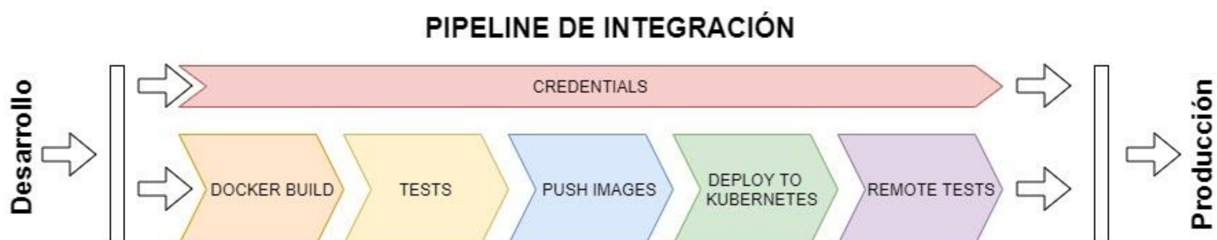

Figura 15. *Stages* que se ejecutan en el *pipeline* de la aplicación

Desde el punto de vista de la implementación, el *pipeline* de CI/CD de este proyecto comprende los siguientes *stages*:

1. *Credentials*: se escriben en ficheros de texto las variables de entorno de los que leerán los contendores para obtener su configuración. Estas variables de entorno sirven para la

comunicación entre microservicios, pero además guardan variables que deben permanecer secretas. Estas variables están almacenadas de forma segura en Jenkins y gestionadas por el servidor, cargándose en tiempo de ejecución.

2. *DockerBuild*: se descargan las imágenes base necesarias (`docker-compose pull`) para cada una de nuestras imágenes personalizadas. Estas imagénes base están definidas Dockerfile de cada microservicio, y se muestran en el apartado de implementación de cada uno de ellos. Una vez las imágenes base se encuentran en nuestro servidor de integración, se procede a ejecutar los Dockerfiles de cada microservicio (`docker-compose build`) para construir nuestras imágenes. Estos dos pasos se realizan con la ayuda del fichero `docker-compose.yml`, que se presenta al final de este capítulo.
3. *Tests*: una vez construidas nuestras imágenes, estas son ejecutadas en forma de contenedores dentro del propio servidor de integración. Estas imágenes son testeadas con la ayuda de un navegador embebido que provee Cypress, llamado Chromium. Cypress se ejecuta dentro de un contenedor propio (`docker-compose run e2e`), de manera totalmente aislada del entorno. Este paso se realiza con la ayuda del fichero `docker-compose.yml`, que se presenta al final de este capítulo.
4. *Push images*: en caso de que el anterior *stage* haya finalizado con éxito, las mismas imágenes que han superado los tests se almacenan en el Elastic Container Registry (ECR), el registro de imágenes de Docker administrado de AWS.
5. *Deploy to Kubernetes*: el servidor de Jenkins se conecta a Kubernetes y ejecuta los ficheros de despliegue e infraestructura, que se bajan de ECR las imágenes recién construidas, testeadas y almacenadas.
6. *Remote tests*: Cypress vuelve a ejecutar los tests dentro de un contenedor, pero esta vez apuntando contra las URLs del entorno de producción, es decir, contra Kubernetes. De esta manera, no solamente se testean las imágenes, como en el *stage* 3, sino que además se testea los ficheros de despliegue e infraestructura de Kubernetes.

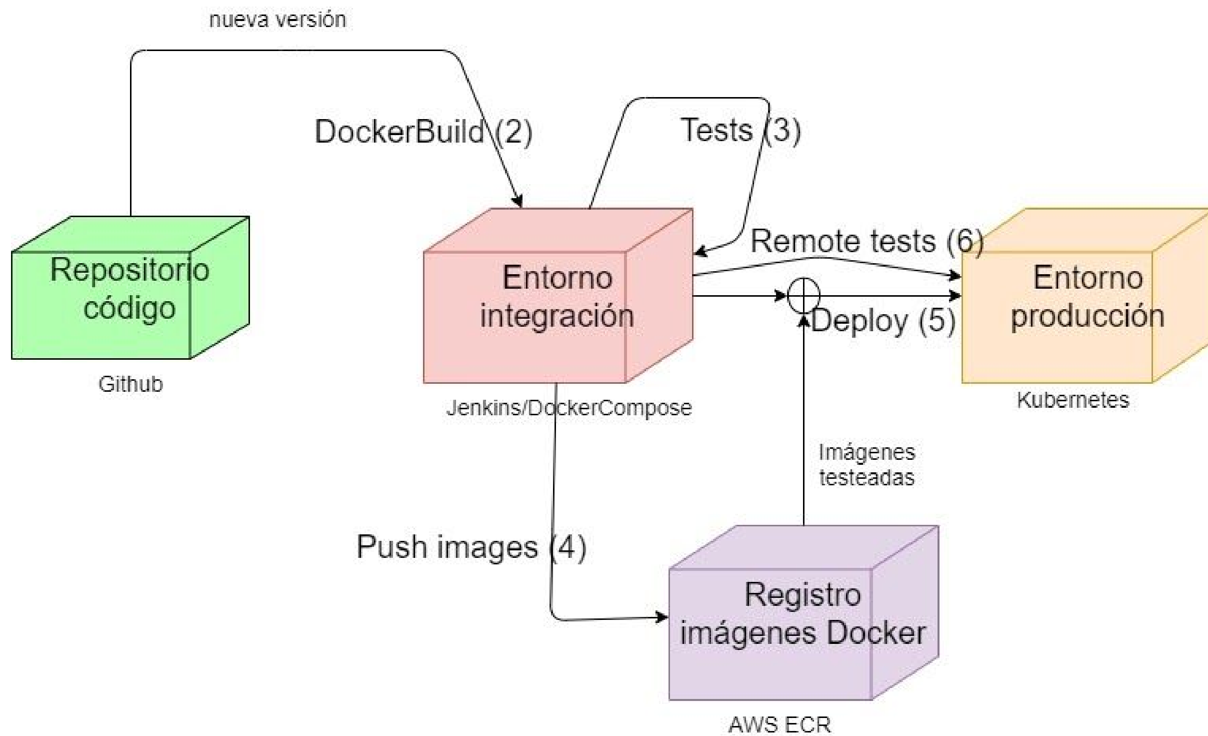

Figura 16. Diagrama de pasos que sigue el *pipeline* de la aplicación

El *pipeline* utilizado en este proyecto viene definido por el siguiente fichero, llamado `Jenkinsfile`:

```
pipeline {
    agent any
    stages {
        stage (Credentials) {
            steps {
                withCredentials([[$class: 'AmazonWebServicesCredentialsBinding',
accessKeyVariable: 'AWS_ACCESS_KEY_ID', credentialsId:
'daxevents-aws-credentials', secretKeyVariable: 'AWS_SECRET_ACCESS_KEY']]) {
                    sh '''#!/bin/bash
                     echo "AWS_ACCESS_KEY_ID=${AWS_ACCESS_KEY_ID}
                        AWS_SECRET_ACCESS_KEY=${AWS_SECRET_ACCESS_KEY}
                        BUCKET_NAME=daxevents
                        JWT_SECRET=${JWT_SECRET}
                        STAGE=test
                        WEBAPP_URL=http://webapp
                        DB_HOST=mongodb://mongo:27017/daxevents" > .env.api

                     echo "DB_HOST=mongodb://mongo-payments:27017/payments
                        STRIPE_KEY=${STRIPE_KEY}
                        WEBAPP_URL=http://webapp
                        API_URL=http://api" > .env.payments-api
```

```
                     echo "window.REACT_APP_API_URL = 'http://172.17.0.1:5000';
                        window.REACT_APP_STRIPE_KEY =
'pk_test_exEPQSqsD0P32gai1ZrrMX4z';
                        window.REACT_APP_ENV = 'LOCAL';
                        window.REACT_APP_PAYMENTS_API_URL =
'http://172.17.0.1:5001';" > ./webapp/public/config.js

                     echo '{
                        "viewportWidth": 1024,
                        "chromeWebSecurity": false,
                        "viewportHeight": 660,
                        "baseUrl":"http://webapp",
                        "apiUrl":"http://172.17.0.1:5000"
                     }' > ./e2e/cypress.json
                    '''
                }
            }
        }

        stage (DockerBuild) {
            steps {
                ansiColor('xterm') {
                    sh '''#!/bin/bash
                     docker-compose pull
                     docker-compose build
                     docker-compose run api npm run manage createuser Carlos
"Martín Ruiz" admin@correo 1234 2020-02-02 666777888 admin
                    '''
                }
            }
        }

        stage (Tests) {
            steps {
                ansiColor('xterm') {
                    sh 'docker-compose run e2e'
                }
            }
        }

        stage ('Push images') {
            steps {
                sh '''#!/bin/bash
                 docker tag daxeventsci_webapp
934890826375.dkr.ecr.eu-west-1.amazonaws.com/webapp:$(git rev-parse --short HEAD)
                 docker tag daxeventsci_webapp
934890826375.dkr.ecr.eu-west-1.amazonaws.com/webapp:latest

                 docker tag daxeventsci_api
934890826375.dkr.ecr.eu-west-1.amazonaws.com/api:$(git rev-parse --short HEAD)
```

```
                docker tag daxeventsci_api
934890826375.dkr.ecr.eu-west-1.amazonaws.com/api:latest

                docker tag daxeventsci_payments-api
934890826375.dkr.ecr.eu-west-1.amazonaws.com/payments-api:$(git rev-parse --short
HEAD)
                docker tag daxeventsci_payments-api
934890826375.dkr.ecr.eu-west-1.amazonaws.com/payments-api:latest
               '''

               sh '''#!/bin/bash
                $(aws ecr get-login --no-include-email --region eu-west-1)

                docker push
934890826375.dkr.ecr.eu-west-1.amazonaws.com/webapp:$(git rev-parse --short HEAD)
> /dev/null
                docker push
934890826375.dkr.ecr.eu-west-1.amazonaws.com/webapp:latest > /dev/null

                docker push
934890826375.dkr.ecr.eu-west-1.amazonaws.com/api:$(git rev-parse --short HEAD) >
/dev/null
                docker push
934890826375.dkr.ecr.eu-west-1.amazonaws.com/api:latest

                docker push
934890826375.dkr.ecr.eu-west-1.amazonaws.com/payments-api:$(git rev-parse --short
HEAD) > /dev/null
                docker push
934890826375.dkr.ecr.eu-west-1.amazonaws.com/payments-api:latest > /dev/null
               '''
           }
       }

       stage ('Deploy to Kubernetes') {
           steps {
               script {
                   withCredentials([sshUserPrivateKey(credentialsId:
'daxevents-key', keyFileVariable: 'keyfile', passphraseVariable: '',
usernameVariable: 'user')]) {
                       def remote = [:]
                       remote.user = user
                       remote.identityFile = keyfile
                       remote.name = "server"
                       remote.host = "virtual.infor.uva.es"
                       remote.port = 65161
                       remote.allowAnyHosts = true


                       sshPut remote: remote, from: 'kube', into: '.'
                       sshScript remote: remote, script: "kube/deploy.sh"
```

```
                        sshCommand remote: remote, failOnError: false, command:
"sudo kubectl exec svc/ci-api -n ci-daxevents-ns -- npm run manage createuser
Carlos \"Martín Ruiz\" admin@correo 1234 2020-02-02 666777888 admin"
                    }
                }
            }
        }

        stage ('Remote tests') {
            steps {
                ansiColor('xterm') {
                    sh label: 'cypress', script: 'docker build -t cypress ./e2e;
docker run --rm -e CYPRESS_video=false -v $WORKSPACE/e2e/cypress:/app/cypress -v
$WORKSPACE/e2e/remote.json:/app/remote.json cypress npm run run:remote'
                }
            }
        }
    }

    post {
        always {
            sh '''#!/bin/bash
                docker-compose down
                docker system prune -f --volumes
            '''
        }
    }
}
```

El resultado en de este fichero en Jenkins es el siguiente:

Figura 17: representación visual de un pipeline de Jenkins. Fuente: servidor de Jenkins

Dentro del *stage* de *Tests*, en el entorno de integración, se utiliza Docker Compose para levantar los microservicios que componen la aplicación, entre ellos también el de Cypress, que ejecuta los tests. El fichero utilizado por Docker Compose para esta tarea se define a continuación:

```
version: "3"
services:
  webapp:
    build: ./webapp
    ports:
      - "80:80"
    volumes:
      - "./webapp/public/config.js:/usr/share/nginx/html/config.js"
    depends_on:
      - api
      - payments-api
  api:
    env_file: .env.api
    environment:
      - WEBAPP_URL=http://webapp
      - DB_HOST=mongodb://mongo:27017/daxevents
    build: ./api
    ports:
      - "5000:3000"
    depends_on:
      - mongo
  payments-api:
    env_file: .env.payments-api
    environment:
      - WEBAPP_URL=http://webapp
      - API_URL=http://api:3000
    build: ./payments-api
    ports:
      - "5001:5000"
    depends_on:
      - mongo-payments
  mongo:
    image: mongo
    volumes:
      - mongo:/data/db
  e2e:
    ipc: host
    image: cypress/base:10.18.1
    build: ./e2e
    environment:
      - CYPRESS_video=false
    depends_on:
      - webapp
    container_name: cypress
    command: npm run run
```

```
    volumes:
      - ./e2e/cypress:/app/cypress
      - ./e2e/cypress.json:/app/cypress.json

volumes:
  mongo:
  payments-db:
```

Para el *stage* de despliegue a Kubernetes, es decir, al entorno de producción, se utiliza Kustomize, la herramienta nativa de configuración para el manejo de los recursos de Kubernetes. La estructura de carpetas para esta tarea es:

```
daxevents/kube/
├── base
│   ├── ambassador
│   │   ├── ambassador-crds.yaml
│   │   └── ambassador-rbac.yaml
│   ├── config.yaml
│   ├── deployments
│   │   ├── api.yml
│   │   ├── mongo-payments.yml
│   │   ├── mongo.yml
│   │   ├── payments-api.yml
│   │   └── webapp.yml
│   ├── kustomization.yaml
│   ├── mappings
│   │   ├── api.yml
│   │   ├── payments-api.yml
│   │   └── webapp.yml
│   ├── namespaces
│   │   └── ns.yml
│   ├── secrets
│   │   ├── .aws.env
│   │   ├── .dockerconfigjson
│   │   ├── .jwt.env
│   │   └── .stripe.env
│   ├── services
│   │   ├── ambassador.yml
│   │   ├── api.yml
│   │   ├── mongo-payments.yml
│   │   ├── mongo.yml
│   │   ├── payments-api.yml
│   │   └── webapp.yml
│   └── volumes
│       └── mongo-pv.yml
├── ci
```

```
│   ├── config.js
│   ├── .hosts.env
│   ├── kustomization.yaml
│   ├── mappings-patch.yaml
│   └── ports-patch.yaml
├── deploy.sh
├── kustomization.yaml
├── nginx.conf
└── prod
    ├── config.js
    ├── .hosts.env
    ├── kustomization.yaml
    ├── mappings-patch.yaml
    └── ports-patch.yaml

10 directories, 37 files
```

Atendiendo a la estructura de carpetas anterior, Kustomize utiliza como plantilla la carpeta `base` para generar volúmenes, secretos y demás recursos de Kubernetes. Estos recursos son sobreescritos con los valores alojados en las carpetas de `ci` y `prod`, representando cada unos de los posibles entornos que podríamos tener dentro de Kubernetes. Para este proyecto se ha utilizado un solo entorno, correspondiente a la carpeta `ci` (a pesar de que representa el entorno de producción).

7. Resumen y conclusiones

Aunque yo haya sido la única persona que ha trabajado en el proyecto, he podido ver cuales son las principales ventajas y desventajas de aplicar el CI/CD y la arquitectura basada en microservicios a un proyecto. La desventaja más destacable se pone de manifiesto a la hora de programar los tests. Los tests pueden llegar a ser muy difíciles de implementar, si no se tiene experiencia y una idea clara de todo lo que se quiere comprobar, debido a que, por ejemplo, el orden de los tests puede alterar significativamente el resultado. Además, estos pueden fallar por diversos factores, algunos de ellos ajenos a la propia aplicación que se está testeando, como, por ejemplo, debido a la lentitud de la conexión de red en la máquina en la que se estén ejecutando. A pesar de este problema y algunos otros, el uso de estas dos técnicas trae más ventajas que desventajas. Esto me ha permitido realizar cambios en uno de los microservicios, sin tener que preocuparme de estar afectando al resto de ellos. Adicionalmente, si alguno de los cambios recién hechos en el código introducía algún error en la aplicación, solo tenía que esperar a que los tests se ejecutarán, una vez subidos los cambios al repositorio de código. Si efectivamente los tests comenzaban a fallar, únicamente era necesario deshacer esos cambios, devolviendo a aplicación a un estado totalmente estable. Esto se traduce en la posibilidad de desarrollar y actualizar la aplicación a una gran velocidad y con una alta fiabilidad.
A nivel personal, este proyecto ha supuesto para mí un gran desafío. Lo he llevado a cabo mientras realizaba prácticas en Madrid y cursaba las últimas asignaturas de la carrera. Esto me ha obligado a organizar mi tiempo de una manera mucho más eficiente y a mejorar mucho mi concentración, para aprovechar al máximo todo el tiempo del que disponía. Pero gracias a haber estado realizando las prácticas en un equipo de DevOps en Telefónica, he podido aplicar todos los conocimientos que he adquirido allí en el TFG, como puede ser el manejo de Amazon Web Services, y también aplicar todo el conocimiento que he adquirido gracias al TFG a las tareas de mi equipo en las prácticas, como la utilización de Docker. También he aprendido a manejar la frustración, ya que eran casi constantes los días en los que aparecían errores totalmente nuevos para mí, y podía llegar a perder hasta tres días en encontrar una solución al problema.
A nivel técnico, decidí realizar el proyecto sobre tecnologías que desconocía en su gran mayoría, ya que los conocimientos utilizados no se imparten en las clases de la universidad. Esto me ha servido para adquirir esos conocimientos, pero sobre todo, para obtener la capacidad de ser autodidacta. En contraposición, esto ha propiciado que el desarrollo del proyecto se alargara más de lo esperado.
He adquirido conocimientos, sobre todo, de React, Nodejs, Amazon Web Services, Kubernetes, Docker, Docker Compose, MongoDB, Cypress y Jenkins. Pero lo más instructivo para mí, con diferencia, ha sido manejar el ciclo de vida completo de la aplicación, es decir, desde la construcción de la interfaz web del usuario, hasta diseñar e implementar la infraestructura de Kubernetes sobre la que se ejecutan todos los microservicios.

Referencias

- *Internet Growth Statistics 1995 to 2019 - the Global Village Online*. (s. f.). Internet Growth Statistics 1995 to 2019. Recuperado 2 de junio de 2020, de https://www.internetworldstats.com/emarketing.htm
- colaboradores de Wikipedia. (2020, 2 junio). *Docker (software)*. Wikipedia, la enciclopedia libre. https://es.wikipedia.org/wiki/Docker_(software)
- *What is a Container?* (s. f.). Docker. Recuperado 6 de junio de 2020, de https://www.docker.com/resources/what-container
- Wikipedia contributors. (2020, 18 junio). *Jenkins (software)*. Wikipedia. https://en.wikipedia.org/wiki/Jenkins_(software)
- *Overview of Docker Compose*. (2020, 24 junio). Docker Documentation. https://docs.docker.com/compose/
- Wikipedia contributors. (2020b, junio 24). *React (web framework)*. Wikipedia. https://en.wikipedia.org/wiki/React_(web_framework)
- S. (s. f.). *styled-components: Basics*. styled-components. Recuperado 16 de junio de 2020, de https://styled-components.com/docs/basics#motivation
- Wikipedia contributors. (2020a, junio 10). *Single-page application*. Wikipedia. https://en.wikipedia.org/wiki/Single-page_application
- *Create a New App –*. (s. f.). React. Recuperado 18 de junio de 2020, de https://reactjs.org/docs/create-a-new-react-app.html
- *What is Babel? · Babel*. (s. f.). Babel. Recuperado 20 de junio de 2020, de https://babeljs.io/docs/en/
- Wikipedia contributors. (2020a, junio 2). *Webpack*. Wikipedia. https://en.wikipedia.org/wiki/Webpack
- Nemeth, E., Snyder, G., Hein, T. R., Whaley, B., & Mackin, D. (2017). *UNIX and Linux System Administration Handbook (5th Edition)* (5.a ed.). New York, USA: Addison-Wesley Professional. Consultado el 06/06/2020.
- Express - Node.js web application framework. (s. f.). Recuperado 9 de junio de 2020, de https://expressjs.com/
- Astorga, P. P. (2020, 27 marzo). Arquitectura de microservicios: qué es, ventajas y desventajas. Recuperado 13 de junio de 2020, de https://decidesoluciones.es/arquitectura-de-microservicios/
- Malav, B. (2018, 16 junio). Microservices vs Monolithic architecture - Hash#Include. Recuperado 13 de junio de 2020, de https://medium.com/startlovingyourself/microservices-vs-monolithic-architecture-c8df91f16bb4
- Tena, M. (2020, 21 mayo). ¿Qué es la metodología «agile»? Recuperado 21 de junio de 2020, de https://www.bbva.com/es/metodologia-agile-la-revolucion-las-formas-trabajo/