Máster en Cloud Apps
Desarrollo y despliegue de aplicaciones en la nube

Curso académico 2023/2024

Trabajo de Fin de Máster

ZUIDUI

Implementación de una aplicación completa con las técnicas y tecnologías del máster

Autores: Manuel Lorente Almán
Juan Ángel Garrido Lupiañez
Tutor: Micael Gallego

Universidad
Rey Juan Carlos


CloudApps
MASTER

Tabla de contenidos

1. Introducción

1.1. Motivación del proyecto

Este proyecto tiene como objetivo principal aplicar las tecnologías y metodologías de desarrollo impartidas durante el máster. Para ello, se ha desarrollado una aplicación completa basada en varios microservicios. El funcionamiento de la aplicación se centra en tres microservicios principales:

- Registro y alta de jugadores y equipos: este microservicio se encarga del registro y la gestión de equipos y sus jugadores.
- Sistema de puntuación 360: este microservicio permite a los jugadores puntuar a los demás miembros de su equipo, proporcionando una evaluación integral del desempeño.
- Interfaz de consolidación de información: este microservicio actúa como intermediario, consolidando la información proporcionada por los otros dos microservicios y respondiendo al cliente con la información integrada.

La arquitectura basada en microservicios permite la futura inclusión de nuevos servicios independientes para extender las funcionalidades de la aplicación.

Desigualdad en la competitividad de los partidos
Frecuentemente, los partidos de deportes de equipo no son equilibrados, resultando en una experiencia menos satisfactoria para los participantes. La formación de equipos con jugadores de habilidades muy dispares afecta la competitividad y el disfrute del juego.

Complejidad en la organización de partidos
Organizar partidos competitivos y equilibrados es una tarea compleja, especialmente cuando se deben gestionar múltiples jugadores y sus habilidades.
Los organizadores necesitan una herramienta que simplifique este proceso y permita una gestión eficiente.

Necesidad de evaluación del desempeño
Tanto jugadores como organizadores desean un sistema de puntuación que permita evaluar el desempeño de los participantes.
Esto no solo ayuda a formar equipos equilibrados, sino que también proporciona una motivación adicional para que los jugadores mejoren sus habilidades.

1.2. Objetivos

Así como la motivación anterior es la que ha empujado al desarrollo de este proyecto, a lo largo del mismo se han cubierto una serie de objetivos técnicos relacionados con el máster que se detallan a continuación.

Desarrollo de un sistema basado en microservicios
Se desarrolló una aplicación compuesta por microservicios independientes, reutilizables y desplegables en un clúster de Kubernetes.

Esta arquitectura facilita la escalabilidad, el mantenimiento y aumenta la resiliencia del sistema.

Arquitectura basada en eventos

Para la comunicación entre microservicios, se implementó un sistema de mensajería que aseguró una integración fluida y eficiente. Se utilizó RabbitMQ para manejar las comunicaciones asincrónicas entre los servicios.

Comunicación eficiente entre servicios

Se utilizó GraphQL como lenguaje de consulta, proporcionando flexibilidad en las consultas, reduciendo el número de peticiones y permitiendo la adición de nuevas funcionalidades sin afectar a los servicios existentes.

GraphQL utiliza esquemas fuertemente tipados, lo que facilita la validación durante el desarrollo.

Desacoplamiento entre cliente y microservicios

Se desarrolló un microservicio que actuó como pasarela para traducir las peticiones del cliente vía API REST y construir los correspondientes mensajes GraphQL para los diferentes microservicios.

Esto desacopla el cliente de los microservicios, centraliza la autenticación, optimiza las peticiones, además de mejorar la escalabilidad, seguridad y simplificar el mantenimiento.

Por último, proporciona flexibilidad para evolucionar la API añadiendo nuevos tipos de clientes con otros protocolos.

Buenas prácticas del desarrollo de software

Se adoptaron buenas prácticas de diseño de software, incluyendo patrones de diseño como arquitectura de capas, aplicando los principios SOLID y Domain Driven Design (DDD). Esto garantizó que la aplicación fuera robusta, mantenible y extensible.

Se utilizaron contenedores de desarrollo (dev containers) para asegurar la consistencia y portabilidad en el desarrollo y las herramientas.

Además, se empleó Makefile para el análisis estático del código, generación de imágenes en el flujo de CI/CD y publicación de artefactos y releases.

Desarrollo orientado a pruebas

Se implementó Test Driven Development (TDD) para asegurar la calidad del software desde el principio. Se realizaron pruebas unitarias, funcionales e integrales para validar la funcionalidad y la integridad del sistema.

Automatización del ciclo de vida de la aplicación

Se siguió un flujo de desarrollo ágil utilizando GitHub Flow y GitHub Actions para integración continua y entrega continua.

GitOps se implementó mediante ArgoCD para despliegue continuo y Helm se utilizó para la gestión de paquetes en Kubernetes, facilitando la automatización y despliegue del sistema.

Despliegue en un entorno local

Inicialmente, la aplicación se desplegó en un clúster local orquestado por Kubernetes, como paso previo al despliegue en un proveedor cloud.

Despliegue en la nube

Para el entorno de producción, la aplicación se desplegó en la nube utilizando AWS como proveedor. Se utilizaron servicios gestionados como Amazon EKS para el clúster de Kubernetes y DockerHub para la gestión de imágenes Docker, proporcionando una infraestructura escalable y de alta disponibilidad.

Automatización del ciclo de vida del sistema

Se emplearon herramientas IaC como Terraform y CloudFormation para la automatización del despliegue de recursos de infraestructura.

2. Descripción del sistema

2.1. Repositorios y microservicios

El proyecto se ha creado como una organización en GitHub en lugar de en el repositorio oficial del máster para mejorar la colaboración y la gestión centralizada de los diferentes repositorios. Esta estructura facilita la continuidad del desarrollo en el futuro.

La organización está compuesta por diferentes repositorios, cada uno con detalles técnicos de la implementación y guías de despliegue. A continuación, se describen los repositorios principales:

- Repositorio de frontend
  Contiene un servidor Nginx que sirve los archivos estáticos y el código JavaScript necesario para ejecutar el cliente de la aplicación.

- Repositorio de recursos
  Incluye manifiestos para levantar recursos comunes de la aplicación, como el broker de mensajería RabbitMQ o un gestor de bases de datos como pgAdmin.

- Repositorio de servicio de pasarela.
  Aloja un servicio FastAPI que actúa como punto de entrada al clúster desde el cliente, orquestando las peticiones y sirviendo como pasarela REST-GraphQL para consolidar respuestas de los microservicios.

- Repositorio de servicio de gestión de equipos y jugadores
  Contiene un servicio FastAPI que permite la creación de equipos, la unión a los mismos y la creación de jugadores en la aplicación.

- Repositorio de servicio de puntuació
  Incluye un servicio FastAPI para la puntuación de los diferentes jugadores de la aplicación.

- Repositorio de infraestructura
  Contiene los manifiestos de Kubernetes para desplegar la aplicación y recursos de IaC como CloudFormation o Terraform para desplegar la infraestructura.

- Repositorio de documentación
  Memoria del proyecto y documentación relevante.

Los enlaces a estos repositorios están anexos a este documento para facilitar el acceso y la navegación.

2.2. Persistencia de datos

Dado que la aplicación está basada en microservicios, la persistencia de datos se ha implementado de manera aislada e independiente en cada uno de los servicios. Cada microservicio tiene asociada su propia base de datos para garantizar la independencia y la integridad de los datos.

Bases de datos
Se utilizan bases de datos relacionales PostgreSQL para los dos servicios principales. Esto asegura una gestión robusta y eficiente de los datos.

Gestor de las bases de datos
Adicionalmente, se ha desplegado una instancia común de administración de bases de datos con pgAdmin. Esta herramienta facilita la administración y supervisión de ambas bases de datos.

Volúmenes persistentes
Para garantizar la persistencia de datos en los contenedores de Kubernetes, se han montado volúmenes persistentes para cada una de las bases de datos desplegadas. En el despliegue en EKS, estos

volúmenes se integran con los componentes de AWS mediante el uso de *storageclass*, asegurando así una gestión eficiente y escalable del almacenamiento.

2.3. Entornos de desarrollo

El proyecto está diseñado para operar en dos entornos distintos, PRE y PRO, con el objetivo de asegurar la calidad y la estabilidad del sistema antes de su despliegue en producción.

Se utilizará un clúster de Kubernetes con diferentes *namespace* según el entorno, ya sea de desarrollo (PRE) o de producción (PRO). Dado que este proyecto no opera en un entorno real, la mayoría del desarrollo en el entorno PRE se llevará a cabo con Minikube.

Entorno de preproducción o PRE
El entorno PRE es el entorno de pruebas donde se validan las nuevas funcionalidades y cambios antes de su despliegue en producción.
En este entorno:

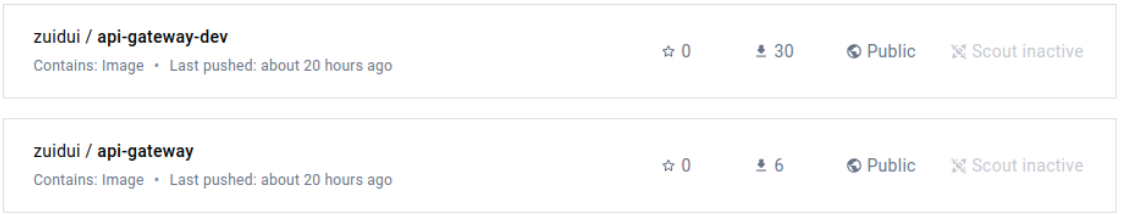
- Se realizará un despliegue automatizado cada vez que se haga push en ramas secundarias de características, refactorización o corrección de errores.
- Se utilizará EKS para la gestión del clúster y los recursos desplegados en AWS.

Entorno de producción o PRO
El entorno PRO es el entorno de producción donde se despliegan los cambios validados y aprobados.
En este entorno:

- Se realizará un despliegue automatizado después de la validación en PRE y la aprobación de pull requests hacia la rama principal.

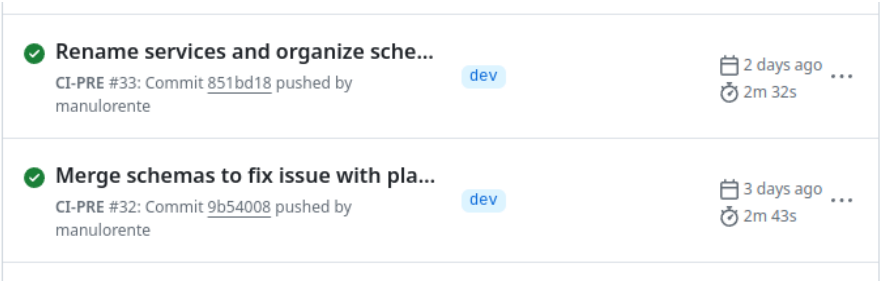
2.4. Flujo de integración continua y entrega continua

El flujo de integración y entrega continua (CI/CD) se implementa utilizando GitHub Actions, para asegurar la calidad del software y la eficiencia en el despliegue.



Integración en PRE
En el entorno PRE, cada push en la rama de desarrollo lanza las siguientes tareas:

- Validación del formato del código.
- Ejecución de pruebas unitarias y de aceptación.



Si la funcionalidad cumple con los requisitos:

- Se etiqueta la versión.
- Se genera una imagen con la etiqueta de la versión y *release-candidate* (rc).
- Se suben dos imágenes al registro de preproducción en DockerHub, etiquetado como *dev*; la versión etiquetada y se actualiza *latest* apuntando a la última versión de desarrollo integrada en preproducción.

Tag	OS	Type	Pulled	Pushed
latest		Image	---	14 hours ago
0.0.2-rc00		Image	---	14 hours ago
0.0.1-rc32		Image	21 hours ago	21 hours ago
0.0.1-rc31		Image	10 hours ago	a day ago
0.0.1-rc30		Image	10 hours ago	a day ago

Integración en PRO
En el entorno PRO, cada pull request a la rama de integración dispara las siguientes tareas:

- Ejecución de pruebas de integración.

Si las pruebas son exitosas:

- Se generan dos imágenes: una con la última versión validada en preproducción y otra etiquetada como latest.

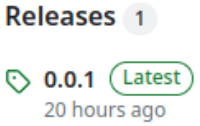
Tags

This repository contains 2 tag(s).

Tag	OS	Type	Pulled	Pushed
latest		Image	15 hours ago	20 hours ago
0.0.1		Image	15 hours ago	20 hours ago

See all

- Al integrarse en la rama principal, se genera en GitHub la release asociada a esa versión.



2.5. Flujo de despliegue continuo

El flujo de despliegue continuo se implementa utilizando la metodología GitOps mediante ArgoCD. Esta metodología asegura que los repositorios de Git actúan como únicas fuentes de verdad. Cualquier cambio versionado en la rama principal y, por lo tanto, actualizado en el registro de producción, se aplica automáticamente al entorno de Kubernetes mediante ArgoCD.

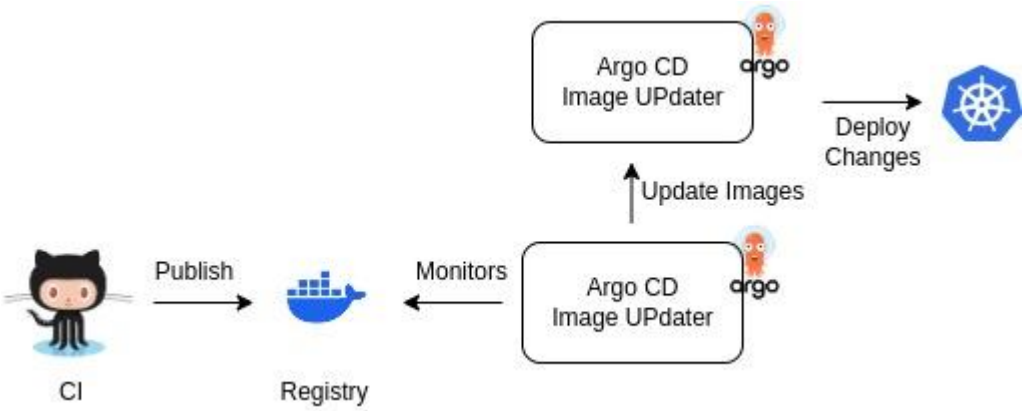
- Versionado y registro: cada cambio versionado en la rama principal se refleja en el registro de producción.
- Despliegue automatizado: ArgoCD monitorea los cambios en el repositorio de Git.
- Actualización en Kubernetes: ArgoCD aplica automáticamente estos cambios al entorno de Kubernetes, asegurando que el estado del clúster siempre esté alineado con el estado del repositorio.

Este enfoque garantiza que todas las actualizaciones se realicen de manera consistente y controlada.

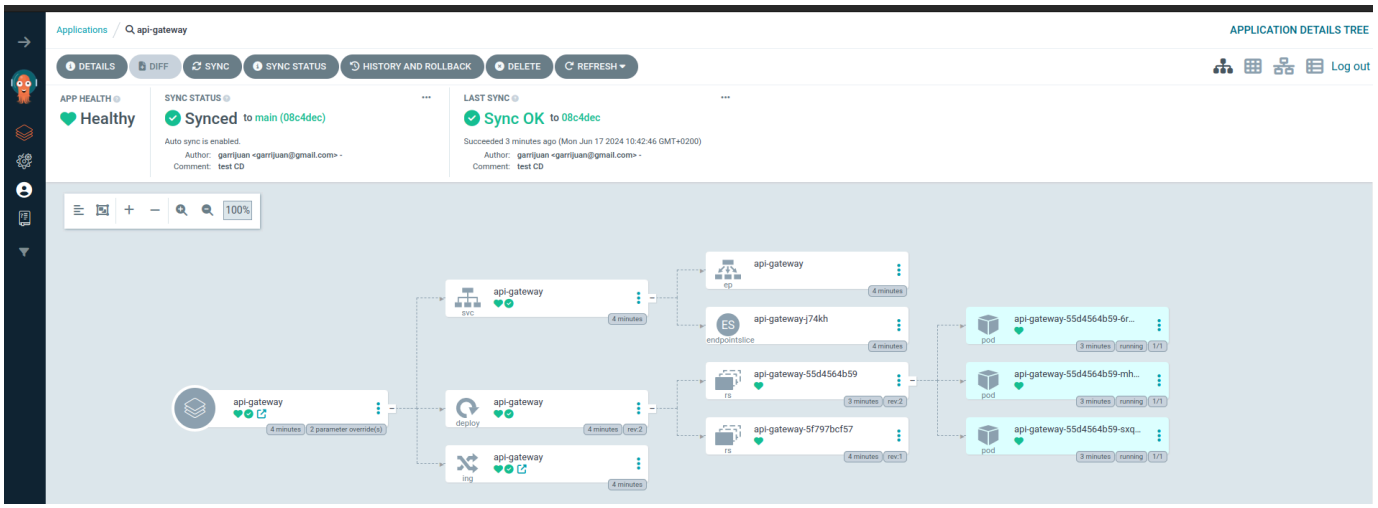
Hemos seguido dos metodologías a la hora de desplegar autónomamente imagenes en el cluster:

- Argo CD Image Updater:

  Argo Image Updater es una herramienta que automatiza la actualización de imágenes de contenedores en clústeres de Kubernetes gestionados por Argo CD. Monitorea repositorios de imágenes y actualiza las imágenes de los contenedores cuando detecta nuevas versiones.

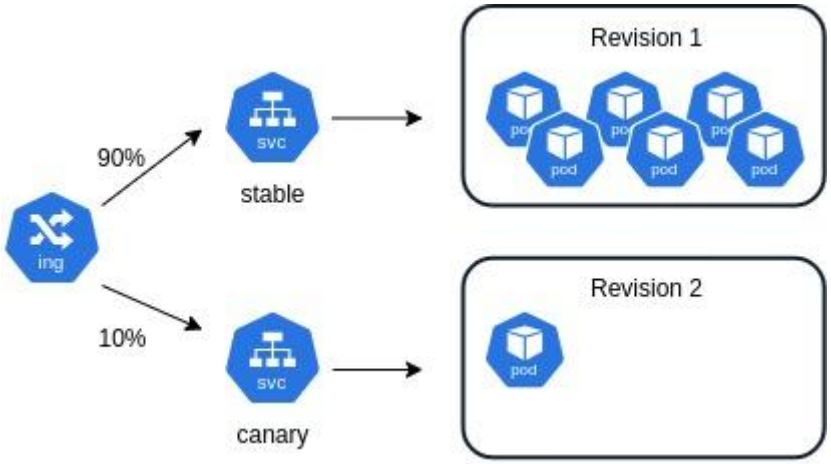


  A continuación, se verifica el despliegue de una nueva versión al detectar una imagen reciente en el repositorio de artefactos, que sucede a la previamente implementada. Posteriormente, se crea una nueva revisión que incorpora esta última imagen, asegurando que el sistema se mantenga actualizado con los cambios más recientes.



❖ Argo CD Rollout Canary:

Argo Rollouts es una herramienta de Kubernetes que facilita implementaciones avanzadas, como el despliegue Canary. En un despliegue Canary, una nueva versión de la aplicación se lanza a un pequeño subconjunto de usuarios para su evaluación antes de un despliegue completo. Argo Rollouts gestiona el tráfico y supervisa las métricas de salud para asegurar una transición segura y controlada a la nueva versión.



A continuación podemos observar las tres fases de despliegue de la canay en el servicio de api-gateway (20%,50%,100%):

❖ Se aplica el despliegue y se crea un pod canary (20%):

```
juanangel@PTTJUANANGEL:~/Escritorio/Repositorios Github/zuidui/api-gateway/chart$ kubectl argo rollouts get rollout api-gateway-rollout -n zuidui
Name:            api-gateway-rollout
Namespace:       zuidui
Status:          ॥ Paused
Message:         CanaryPauseStep
Strategy:        Canary
  Step:          1/5
  SetWeight:     20
  ActualWeight:  20
Images:          zuidui/api-gateway-dev:0.0.1-rc21 (stable)
                 zuidui/api-gateway-dev:0.0.1-rc22 (canary)
Replicas:
  Desired:       5
  Current:       5
  Updated:       1
  Ready:         5
  Available:     5

NAME                                        KIND        STATUS         AGE    INFO
⟳ api-gateway-rollout                       Rollout     ॥ Paused       3m24s
├──# revision:2
│  └──⧉ api-gateway-rollout-cf98955b9       ReplicaSet  ✔ Healthy      15s    canary
│     └──□ api-gateway-rollout-cf98955b9-p4vb4  Pod     ✔ Running      15s    ready:1/1
└──# revision:1
   └──⧉ api-gateway-rollout-84c5fd99d4       ReplicaSet  ✔ Healthy      3m24s  stable
      ├──□ api-gateway-rollout-84c5fd99d4-7gspn  Pod     ✔ Running      3m24s  ready:1/1
      ├──□ api-gateway-rollout-84c5fd99d4-7mlvq  Pod     ✔ Running      3m24s  ready:1/1
      ├──□ api-gateway-rollout-84c5fd99d4-8r9jl  Pod     ✔ Running      3m24s  ready:1/1
      ├──□ api-gateway-rollout-84c5fd99d4-bhxvz  Pod     ✔ Running      3m24s  ready:1/1
      └──□ api-gateway-rollout-84c5fd99d4-fsbq4  Pod     ◌ Terminating  3m24s  ready:1/1
```

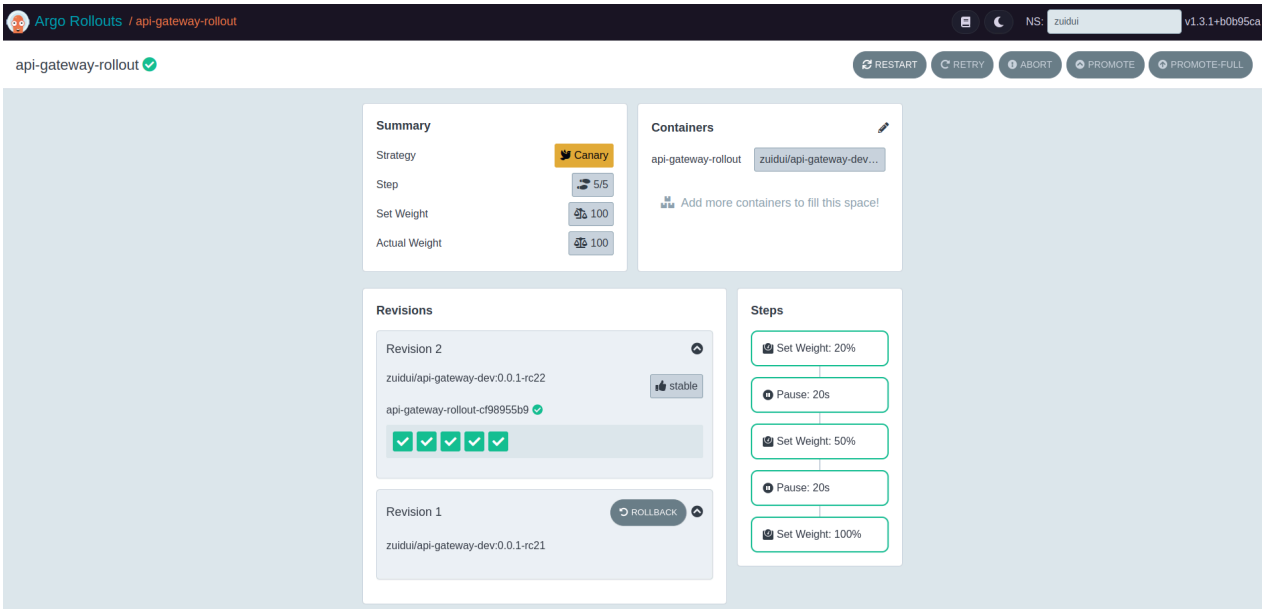
- Tras un intervalo de tiempo aumentamos número de pods al 50%:

```
juanangel@PTTJUANANGEL:~/Escritorio/Repositorios Github/zuidui/api-gateway/chart$ kubectl argo rollouts get rollout api-gateway-rollout -n zuidui
Name:            api-gateway-rollout
Namespace:       zuidui
Status:          ॥ Paused
Message:         CanaryPauseStep
Strategy:        Canary
  Step:          3/5
  SetWeight:     50
  ActualWeight:  50
Images:          zuidui/api-gateway-dev:0.0.1-rc21 (stable)
                 zuidui/api-gateway-dev:0.0.1-rc22 (canary)
Replicas:
  Desired:       5
  Current:       6
  Updated:       3
  Ready:         6
  Available:     6

NAME                                        KIND        STATUS         AGE    INFO
⟳ api-gateway-rollout                       Rollout     ॥ Paused       3m48s
├──# revision:2
│  └──⧉ api-gateway-rollout-cf98955b9       ReplicaSet  ✔ Healthy      39s    canary
│     ├──□ api-gateway-rollout-cf98955b9-p4vb4  Pod     ✔ Running      39s    ready:1/1
│     ├──□ api-gateway-rollout-cf98955b9-fkb8h  Pod     ✔ Running      7s     ready:1/1
│     └──□ api-gateway-rollout-cf98955b9-zzn6b  Pod     ✔ Running      7s     ready:1/1
└──# revision:1
   └──⧉ api-gateway-rollout-84c5fd99d4       ReplicaSet  ✔ Healthy      3m48s  stable
      ├──□ api-gateway-rollout-84c5fd99d4-7gspn  Pod     ✔ Running      3m48s  ready:1/1
      ├──□ api-gateway-rollout-84c5fd99d4-7mlvq  Pod     ✔ Running      3m48s  ready:1/1
      ├──□ api-gateway-rollout-84c5fd99d4-8r9jl  Pod     ◌ Terminating  3m48s  ready:1/1
      └──□ api-gateway-rollout-84c5fd99d4-bhxvz  Pod     ✔ Running      3m48s  ready:1/1
```

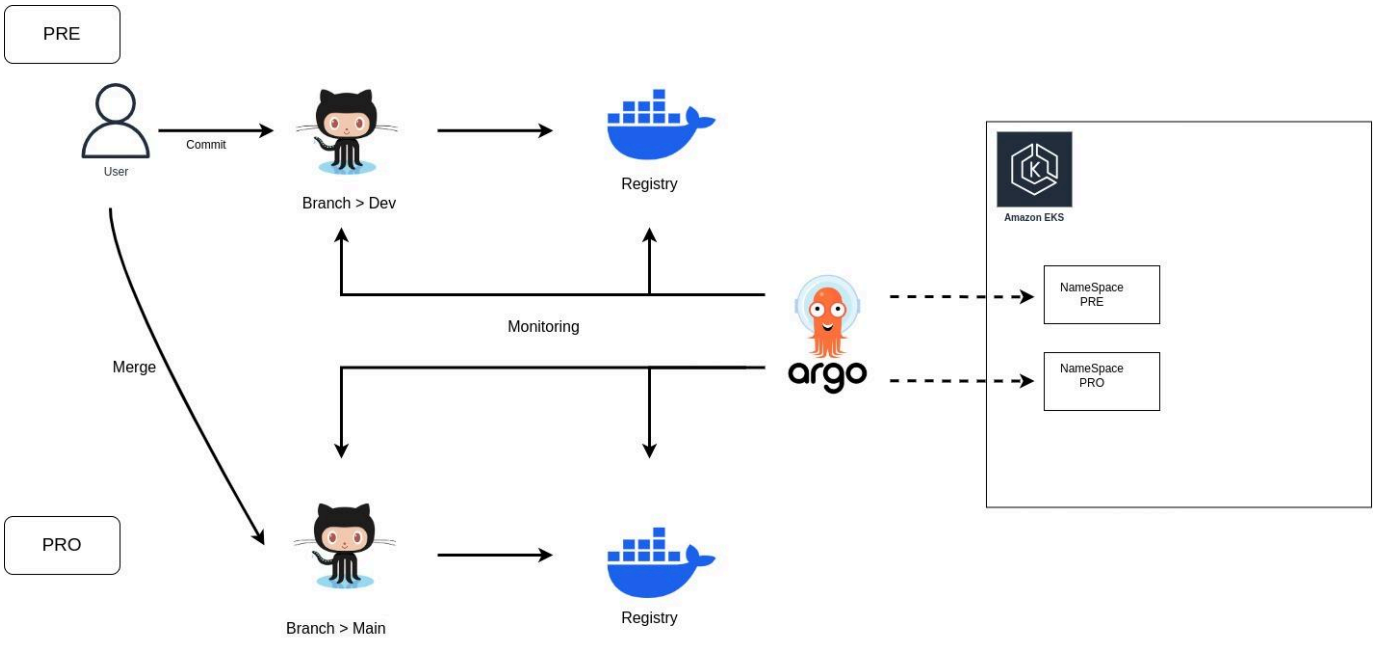
- Tras un intervalo de tiempo aumentamos número de pods al máximo, cambiamos la nueva release a version estable y se escala a cero la versión anterior:

```
juanangel@PTTJUANANGEL:~/Escritorio/Repositorios Github/zuidui/api-gateway/chart$ kubectl argo rollouts get rollout api-gateway-rollout -n zuidui
Name:            api-gateway-rollout
Namespace:       zuidui
Status:          ✔ Healthy
Strategy:        Canary
  Step:          5/5
  SetWeight:     100
  ActualWeight:  100
Images:          zuidui/api-gateway-dev:0.0.1-rc22 (stable)
Replicas:
  Desired:       5
  Current:       5
  Updated:       5
  Ready:         5
  Available:     5

NAME                                        KIND        STATUS         AGE    INFO
⟳ api-gateway-rollout                       Rollout     ✔ Healthy      8m59s
├──# revision:2
│  └──⧉ api-gateway-rollout-cf98955b9       ReplicaSet  ✔ Healthy      5m50s  stable
│     ├──□ api-gateway-rollout-cf98955b9-p4vb4  Pod     ✔ Running      5m50s  ready:1/1
│     ├──□ api-gateway-rollout-cf98955b9-fkb8h  Pod     ✔ Running      5m18s  ready:1/1
│     ├──□ api-gateway-rollout-cf98955b9-zzn6b  Pod     ✔ Running      5m18s  ready:1/1
│     ├──□ api-gateway-rollout-cf98955b9-dtcpt  Pod     ✔ Running      4m57s  ready:1/1
│     └──□ api-gateway-rollout-cf98955b9-vm7rv  Pod     ✔ Running      4m57s  ready:1/1
└──# revision:1
   └──⧉ api-gateway-rollout-84c5fd99d4       ReplicaSet  • ScaledDown   8m59s
```

- Interfaz gráfica donde se monitoriza el despliegue con la capacidad de revertir a la versión anterior si es necesario:



La integración de esta metodología DevOps en el entorno de trabajo se realiza de la siguiente manera: el desarrollador trabaja en su entorno local y realiza commit y push de los cambios a una rama de desarrollo. Un sistema de Integración Continua (CI) ejecuta pruebas automáticas y, si se superan satisfactoriamente, despliega los cambios en un entorno de desarrollo compartido.
Posteriormente, el desarrollador crea un pull request para fusionar los cambios en la rama principal (producción), el cual es revisado y aprobado por otros miembros del equipo. Una vez aprobado y fusionado el pull request, los cambios se despliegan automáticamente en el entorno de producción.



2.6. Pautas de desarrollo y diseño

Durante el desarrollo del proyecto, se han seguido las siguientes pautas:

Framework

- FastAPI: se ha utilizado FastAPI de Python para el desarrollo de los servicios.

Mensajería

- RabbitMQ: se ha utilizado RabbitMQ para la comunicación entre microservicios.

Pruebas

- Pirámide de pruebas: los microservicios han seguido la pirámide de pruebas, incluyendo pruebas unitarias y de integración.

Desarrollo

- Contenedores de desarrollo: se han utilizado contenedores de desarrollo para cada uno de los microservicios.
- Makefile: se han utilizado ficheros Makefile para la construcción de imágenes Docker y la automatización de tareas necesarias durante el desarrollo.

Metodología

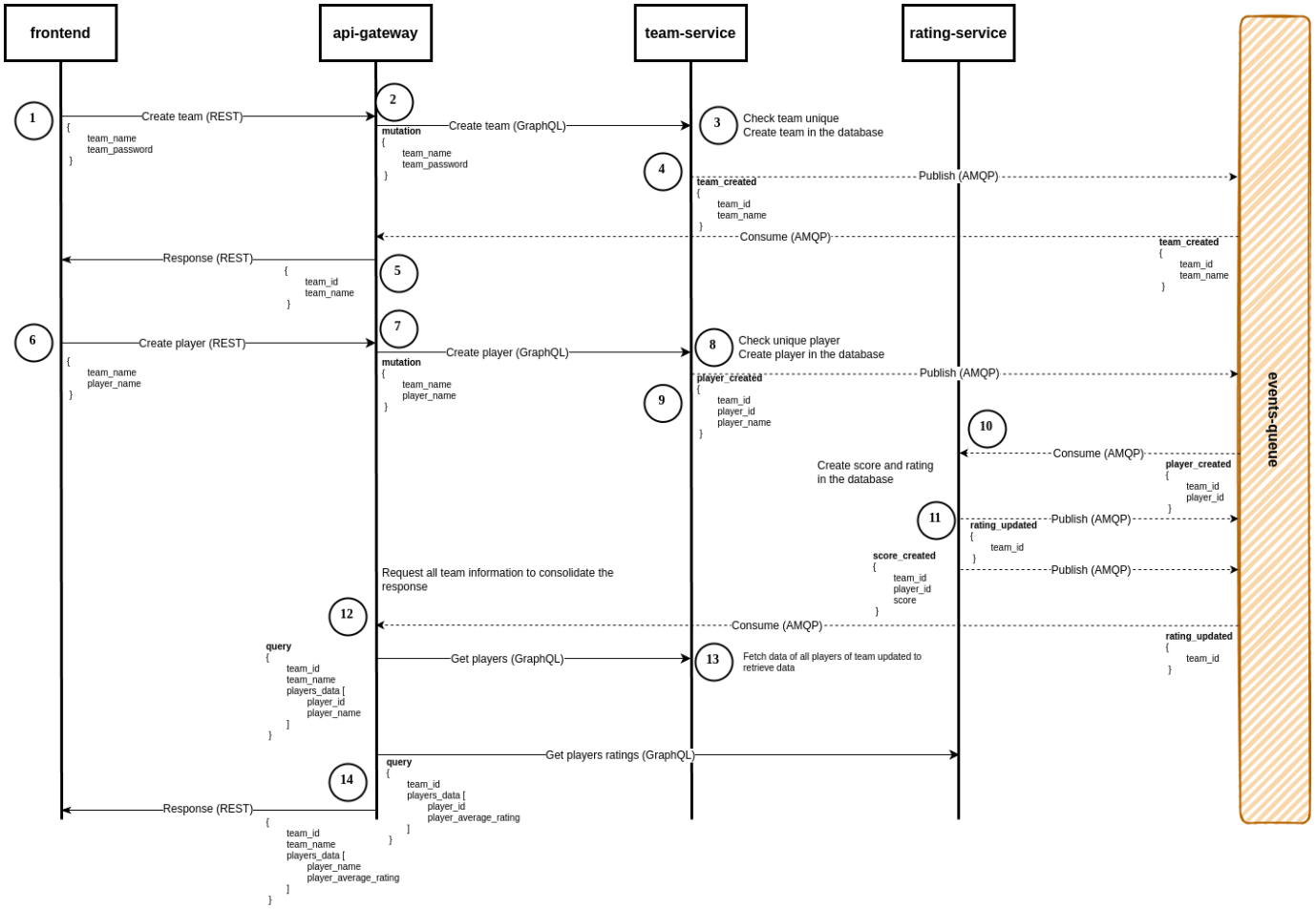
- GitHub Flow: se ha seguido la metodología ligera GitHub Flow, creando una rama específica de desarrollo durante el mismo y solicitudes de pull para revisión de código e integración en la rama principal de cada servicio.

Despliegue

- Charts de Helm y manifiestos de Kubernetes: se han utilizado charts de Helm y manifiestos de Kubernetes para el despliegue de los servicios.

3. Historias de usuario

3.1. Caso de uso #1



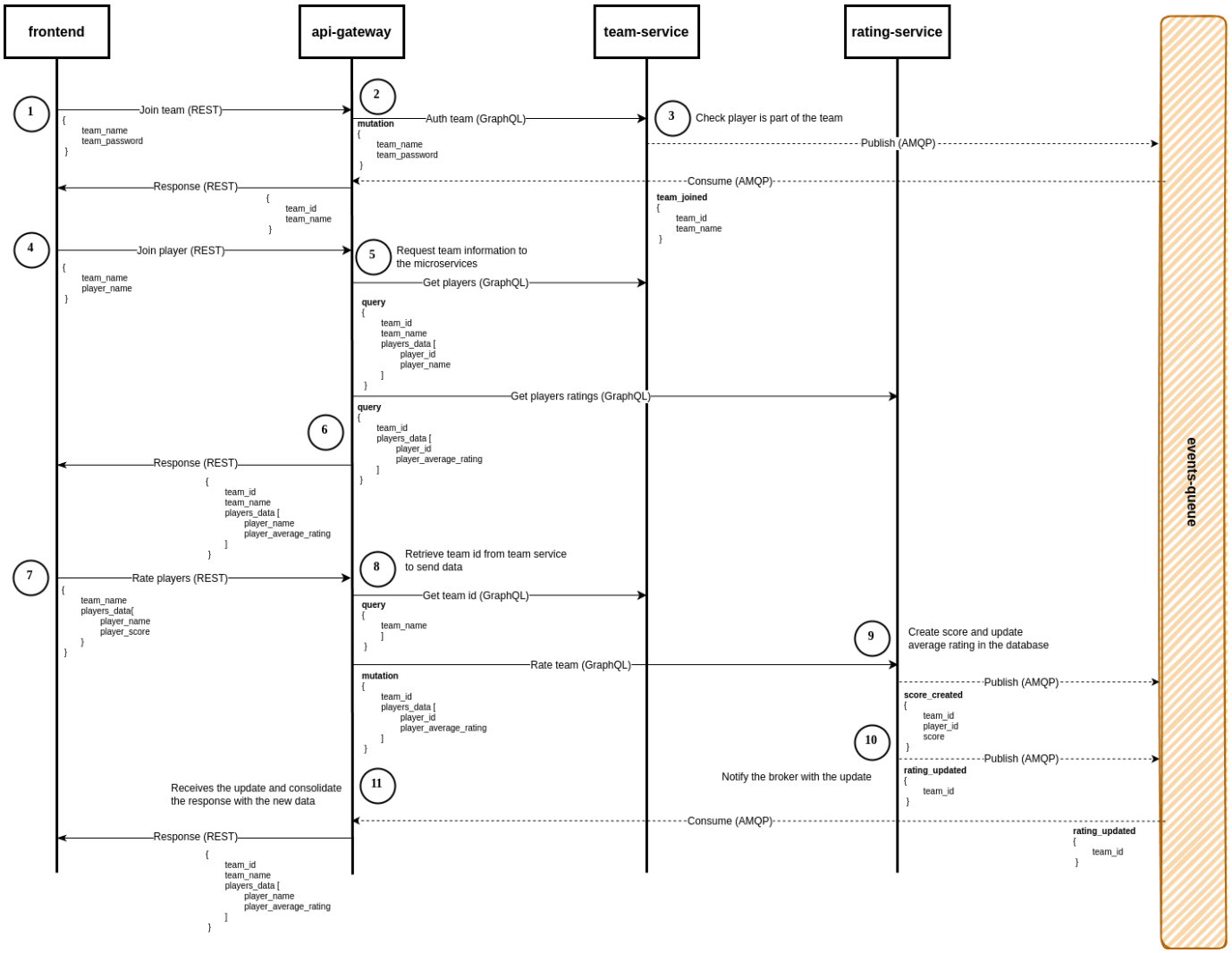
Un usuario accede a la aplicación por primera vez para crear un equipo y jugadores.

El flujo es el siguiente:

1. El frontend envía una solicitud REST para crear un equipo.
2. El API Gateway recibe la solicitud y la convierte en una mutación GraphQL, que es enviada al Team Service.
3. El Team Service verifica si el equipo es único y lo crea en la base de datos.
4. El Team Service publica un mensaje en la cola de eventos indicando que el equipo ha sido creado.
5. El API Gateway consume el mensaje de creación de equipo completada y responde vía REST al frontend.
6. El frontend envía una solicitud REST para crear un jugador.
7. El API Gateway recibe la solicitud y la convierte en una mutación GraphQL, que es enviada al Team Service.
8. El Team Service verifica si el jugador es único para ese equipo y lo crea en la base de datos.
9. El Team Service publica un mensaje en la cola de eventos indicando que el jugador ha sido creado.
10. El Rating Service consume el evento, crea una calificación y un rating promedio inicial para el jugador.

11. El Rating Service, una vez procesada la transacción, publica el evento de actualización en un equipo.
12. Cuando el API Gateway recibe el evento de equipo actualizado, lanza una query con el identificador del equipo actualizado para recuperar los datos de los jugadores.
13. El Team Service responde con los datos del equipo y sus jugadores.
14. El API Gateway envía otra query al Rating Service para recuperar las puntuaciones del equipo y actualizarlas al frontend en una respuesta compuesta por los datos recibidos de ambos microservicios.

3.2. Caso de uso #2



Un usuario accede accede a un equipo ya existente y puntúa al resto de integrantes.

El flujo es el siguiente:

1. El frontend envía una solicitud REST para unirse a un equipo existente.
2. El API Gateway recibe la solicitud y la convierte en una mutación GraphQL, que es enviada al Team Service.
3. El Team Service verifica las credenciales del equipo y publica un mensaje en la cola de eventos, que es consumido por el API Gateway para responder al frontend.
4. El frontend envía una petición REST para identificarse como miembro de un equipo.
5. El Team Service verifica las credenciales del usuario y solicita la información del equipo a los diferentes servicios.

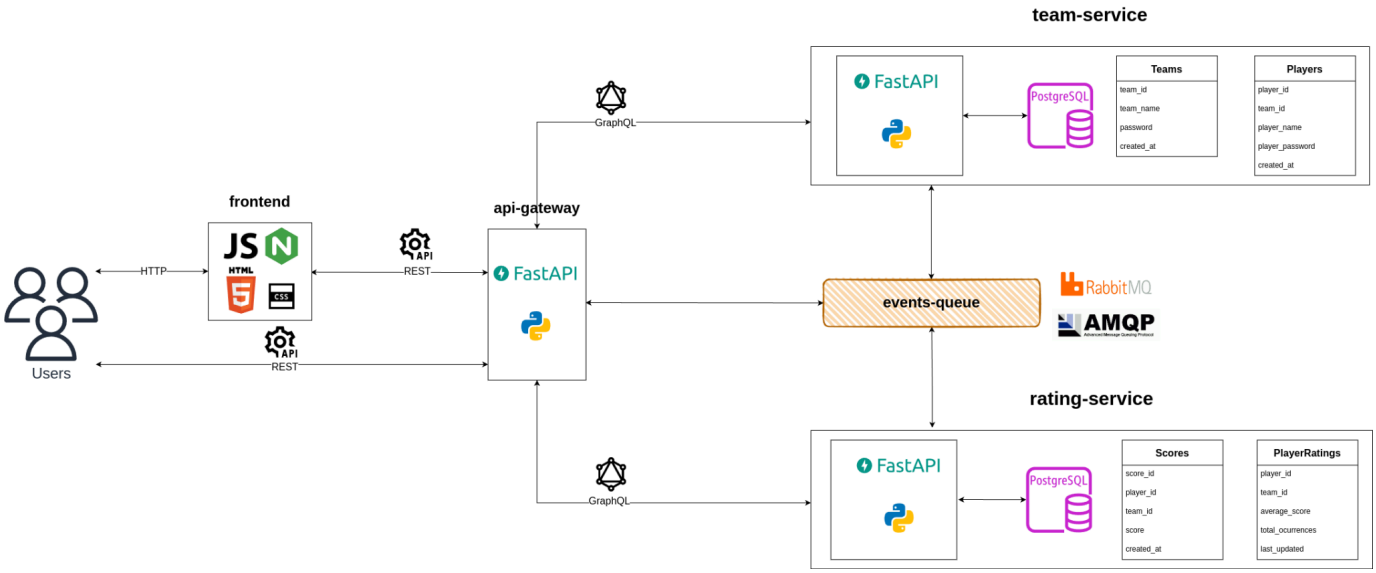
6. Cuando se recibe toda la información, se consolida el mensaje de respuesta al frontend para mostrar toda la información del equipo.
7. El usuario califica a los jugadores del equipo, enviando una solicitud REST al frontend.
8. El API Gateway convierte las calificaciones en una mutación GraphQL y las envía al Rating Service.
9. El Rating Service actualiza las calificaciones de los jugadores en la base de datos.
10. El Rating Service publica un mensaje en la cola de eventos indicando que las calificaciones han sido actualizadas.
11. El API Gateway consume el mensaje y consolida la respuesta con los datos actualizados al frontend.

4. Arquitectura

4.1. Arquitectura de la aplicación

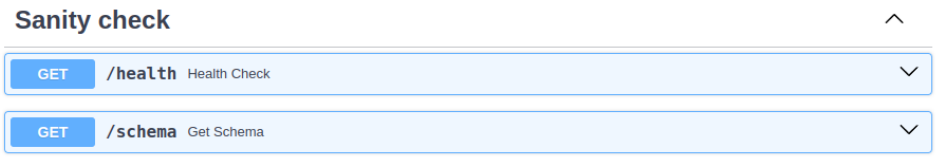
La aplicación está compuesta por cuatro microservicios y utiliza un exchange fanout en RabbitMQ para la comunicación entre servicios. Este enfoque facilita la incorporación de nuevos componentes en el futuro sin incrementar significativamente la complejidad de la aplicación.

Cada microservicio que requiere persistencia cuenta con su propia base de datos. Esta estrategia proporciona independencia y modularidad, permitiendo que los servicios operen de manera autónoma.



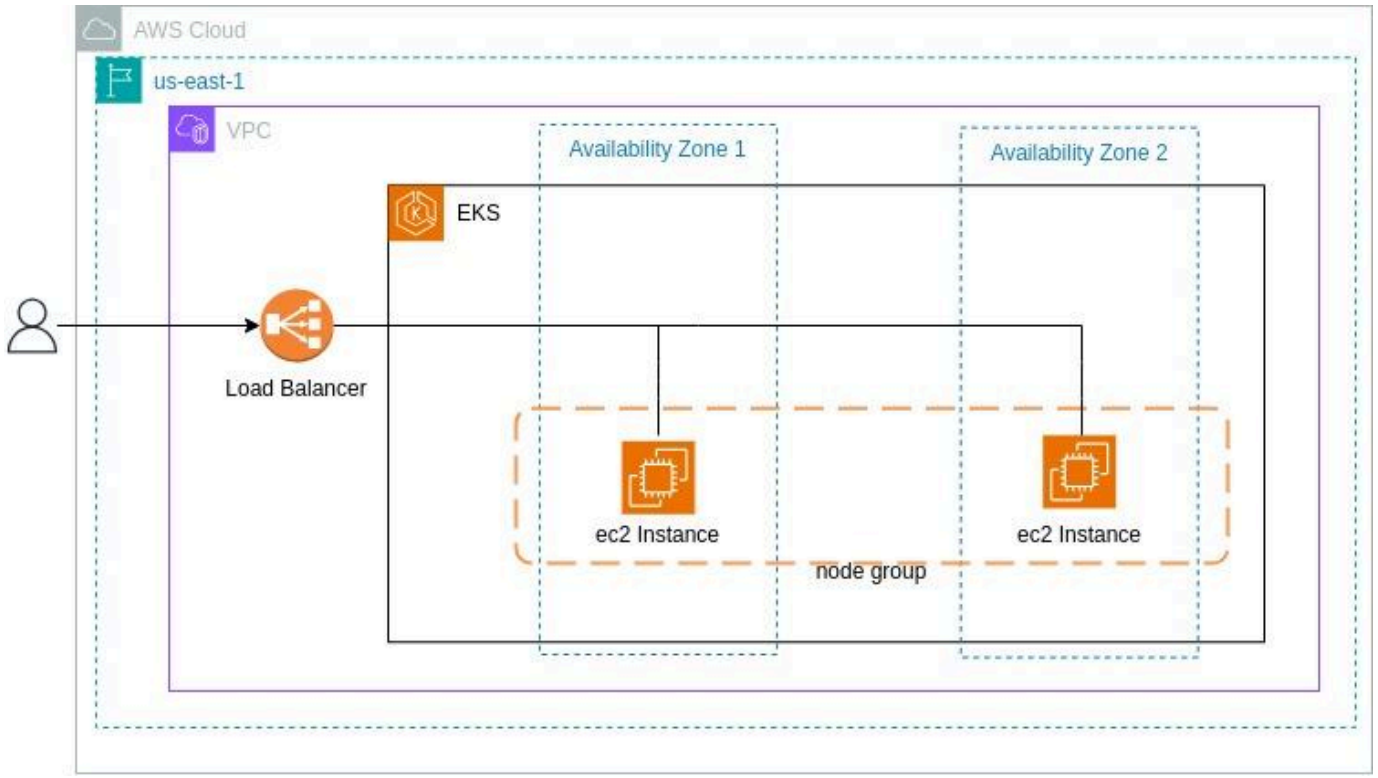
El uso de RabbitMQ configurado como un fanout exchange ha sido crucial. Esta configuración permite que todos los servicios puedan publicar y consumir mensajes sin que el orden sea un factor crítico. Como resultado, la aplicación es reactiva a cualquier evento, evitando bloqueos mientras espera otros recursos.

Finalmente, se han incluido endpoints de salud en todos los servicios para verificar el estado de cada microservicio.



4.2. Arquitectura del sistema

La arquitectura del sistema se basa en un clúster desplegado en Amazon Elastic Kubernetes Service (EKS) con dos nodos EC2.
Para implementar esta solución, hemos creado una Virtual Private Cloud (VPC) para un entorno de red seguro, configurado grupos de nodos (node groups) para alojar las instancias EC2 y definido los roles y permisos necesarios mediante AWS IAM.
Además, se ha implementado un balanceador de carga (Load Balancer) para distribuir eficientemente el tráfico de red, garantizando alta disponibilidad y rendimiento.
Esta arquitectura permite una gestión escalable y eficiente de aplicaciones en contenedores, asegurando la resiliencia y flexibilidad del sistema.



Toda la arquitectura está gestionada y mantenida mediante Infrastructure as Code (IaC) utilizando Terraform y AWS CloudFormation. Este enfoque asegura una administración consistente, automatizada y reproducible de la infraestructura, facilitando su despliegue, actualización y mantenimiento de manera eficiente y segura.
De este modo, logramos una aplicación totalmente autocontenida con su propio ciclo de vida dentro del entorno de la arquitectura, garantizando un desarrollo continuo y una operatividad autónoma.

5. Validación y pruebas

5.1. Validación de la aplicación

5.1.1. Sanidad de servicios y comunicación

Veamos primero toda la aplicación levantada en contenedores en local.

```
manu@msi:~$ docker ps -a --format "table {{.ID}}\t{{.Names}}\t{{.Image}}\t{{.Status}}"
CONTAINER ID   NAMES            IMAGE                          STATUS
6efcc94d9b27   api-gateway      api-gateway:0.0.2              Up About an hour
f938d3069452   rating-service   rating-service:0.0.2           Up 2 hours
59b5b5238a77   rating-db        postgres:13                    Up 2 hours
7f50fa5620dc   team-service     team-service:0.0.2             Up 2 hours
5dcb4e3d0d42   team-db          postgres:13                    Up 2 hours
d24262361798   frontend         frontend:0.0.2                 Up 2 hours
21619b9d8a24   events-store     rabbitmq:3.11-management       Up 2 hours
8347ed302e43   pgadmin          dpage/pgadmin4:latest          Up 23 hours
manu@msi:~$
```

Mediante peticiones GET a los endpoints de sanidad de los diferentes microservicios podemos comprobar que están disponibles.

```
manu@msi:~$ curl http://localhost:8081/health
{"status":"ok"}manu@msi:~$
manu@msi:~$ curl http://localhost:8082/health
{"status":"ok"}manu@msi:~$
manu@msi:~$ curl http://localhost:8083/health
{"status":"ok"}manu@msi:~$
```

Ahora, de la misma manera, vamos a comprobar la conectividad entre servicios desde dentro de los contenedores.

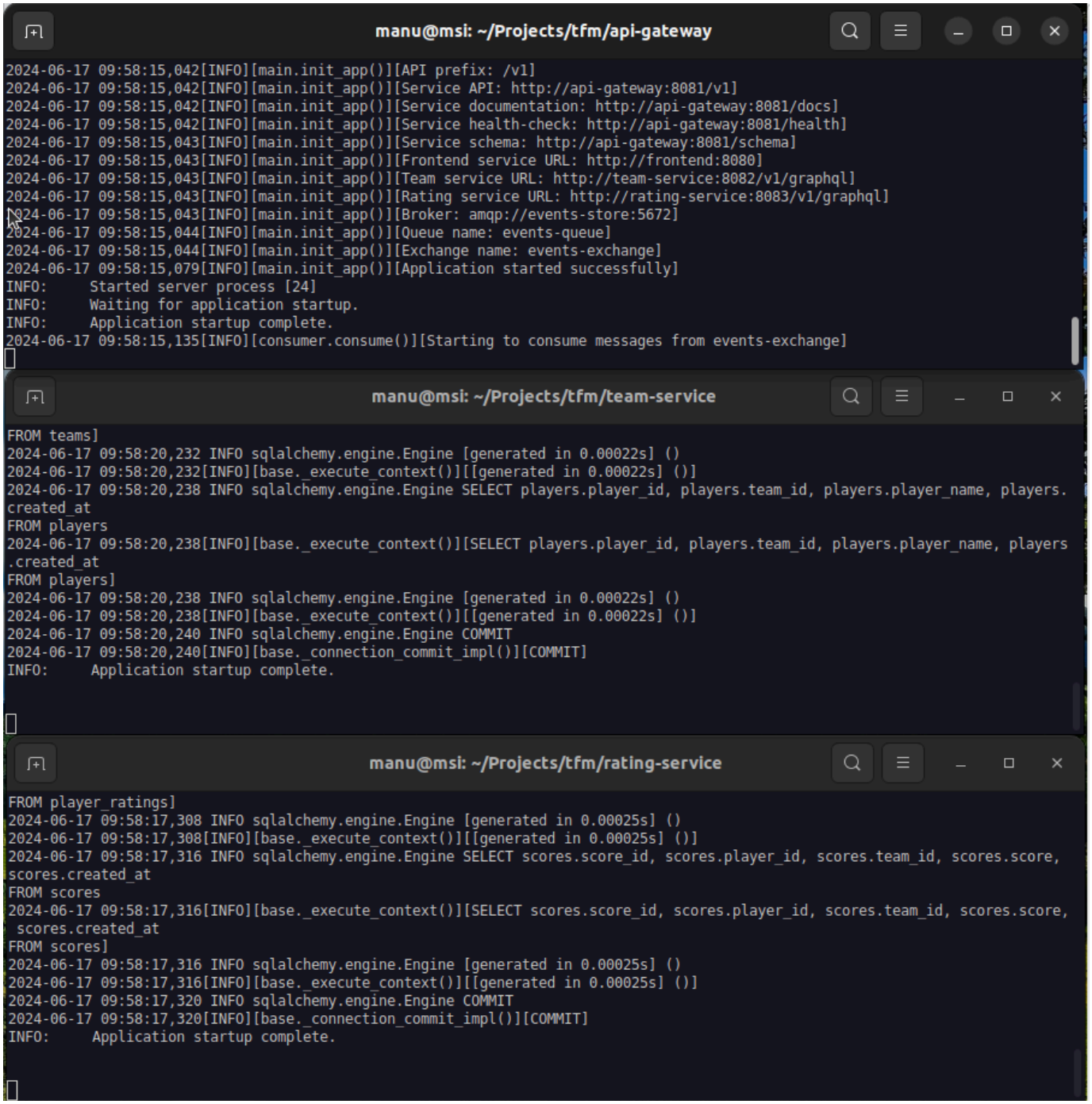
```
manu@msi:~$ docker exec -it frontend /bin/sh
/ # curl http://api-gateway:8081/health
{"status":"ok"}/ #
/ # curl http://team-service:8082/health
{"status":"ok"}/ #
/ # curl http://rating-service:8083/health
{"status":"ok"}/ #
/ # curl http://rating-service:8083/schema
"type Mutation {\n  placeholder: String!\n}\n\ntype PlayerRatingList {\n  team_id: Int!\n  players_data: [PlayerRatingOutput!]!\n}
\n\ntype PlayerRatingOutput {\n  player_id: Int!\n  player_average_rating: Float!\n}\n\ntype Query {\n  get_players_rating(team_id
: Int!): PlayerRatingList\n}"/ #
/ #
/ #
```

Con esto se daría por validado la sanidad y la comunicación entre servicios.

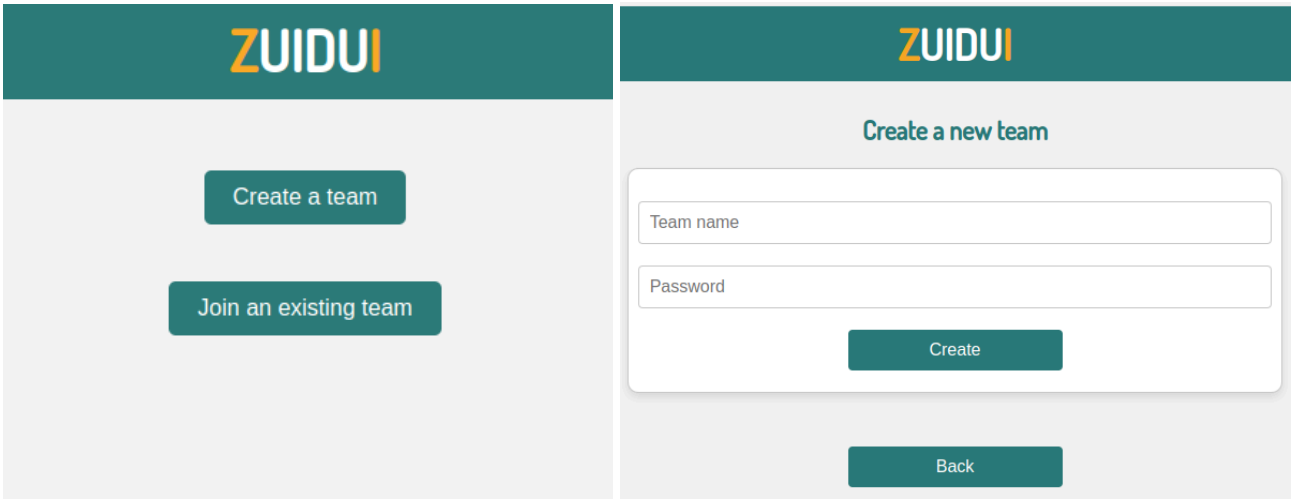
5.1.2. Caso de uso

A continuación vamos a mostrar los logs de los tres microservicios principales así como las diferentes vistas del frontend.

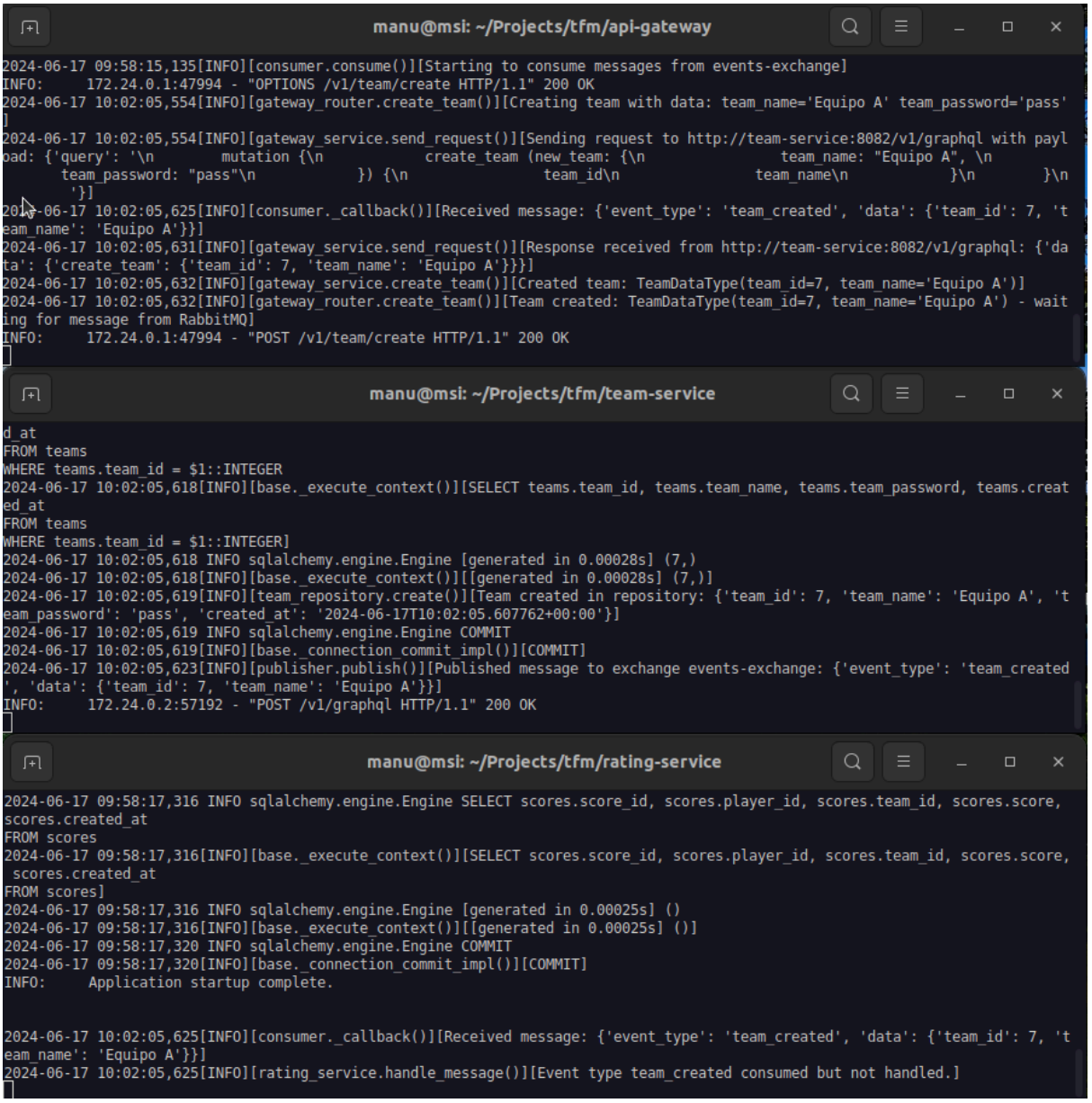
Cuando la aplicación arranca, todos los servicios están a la espera de peticiones. En el caso del team-service y rating-service han cargado al inicio de la aplicación unos datos de ejemplo para las pruebas.



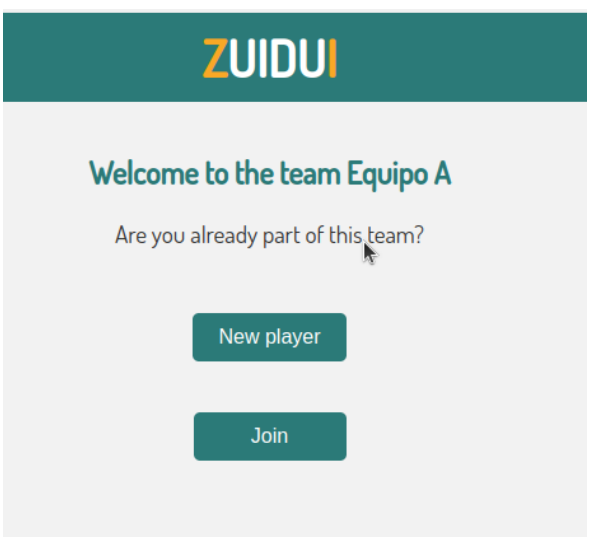
Y como se puede acceder a la aplicación a través del localhost. Y probaremos a crear un equipo.

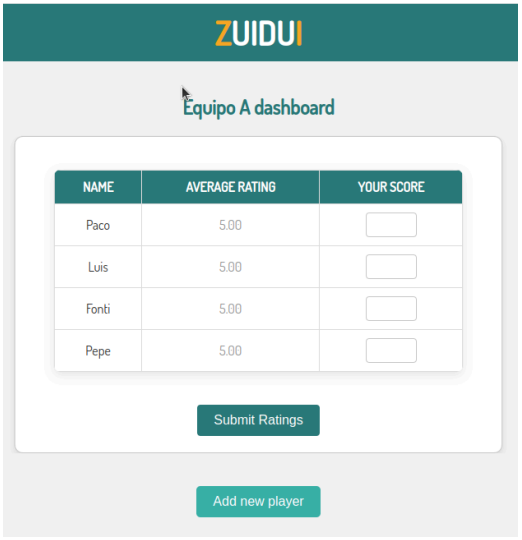


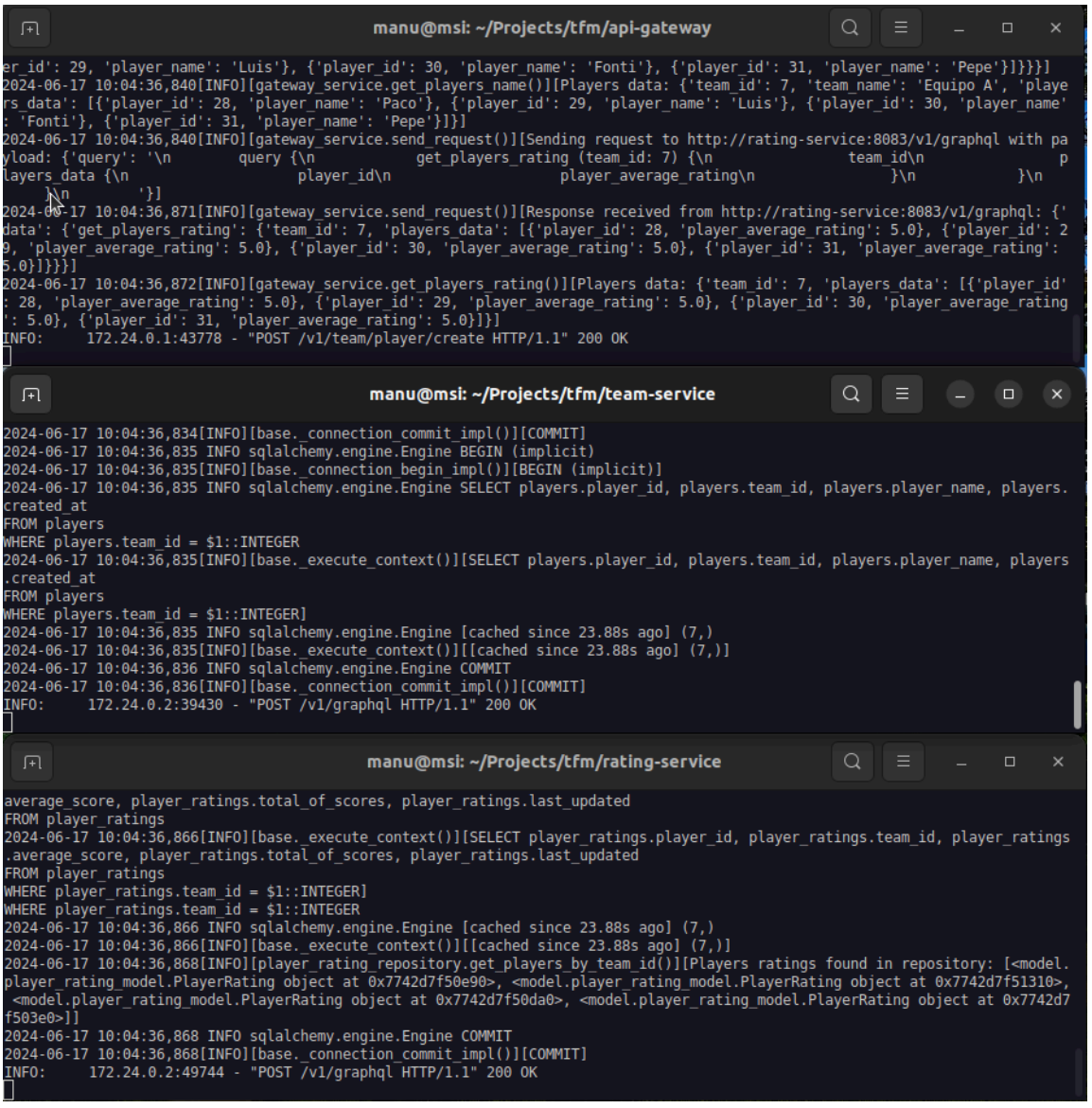
Cuando se crea un equipo vemos cómo se actualizan los logs con la transacción de mensajes y eventos.



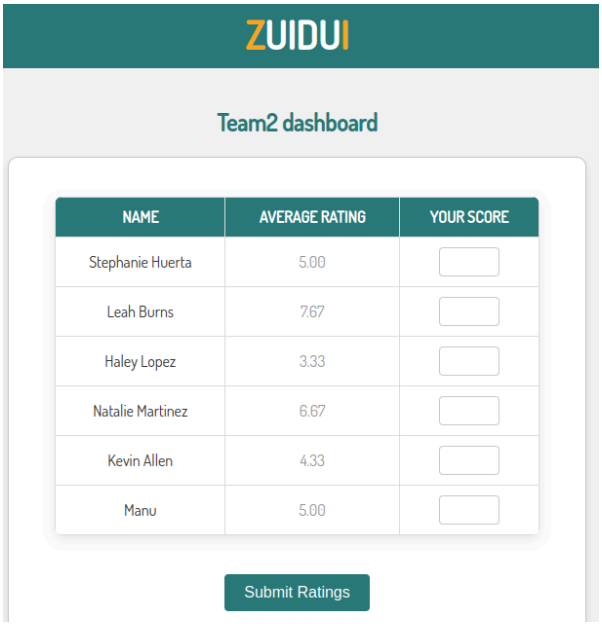
Vamos a probar a crear algunos jugadores de ejemplo, a los cuales se le asigna una puntuación media de 5 al principio y se verá cómo suceden las transacciones.

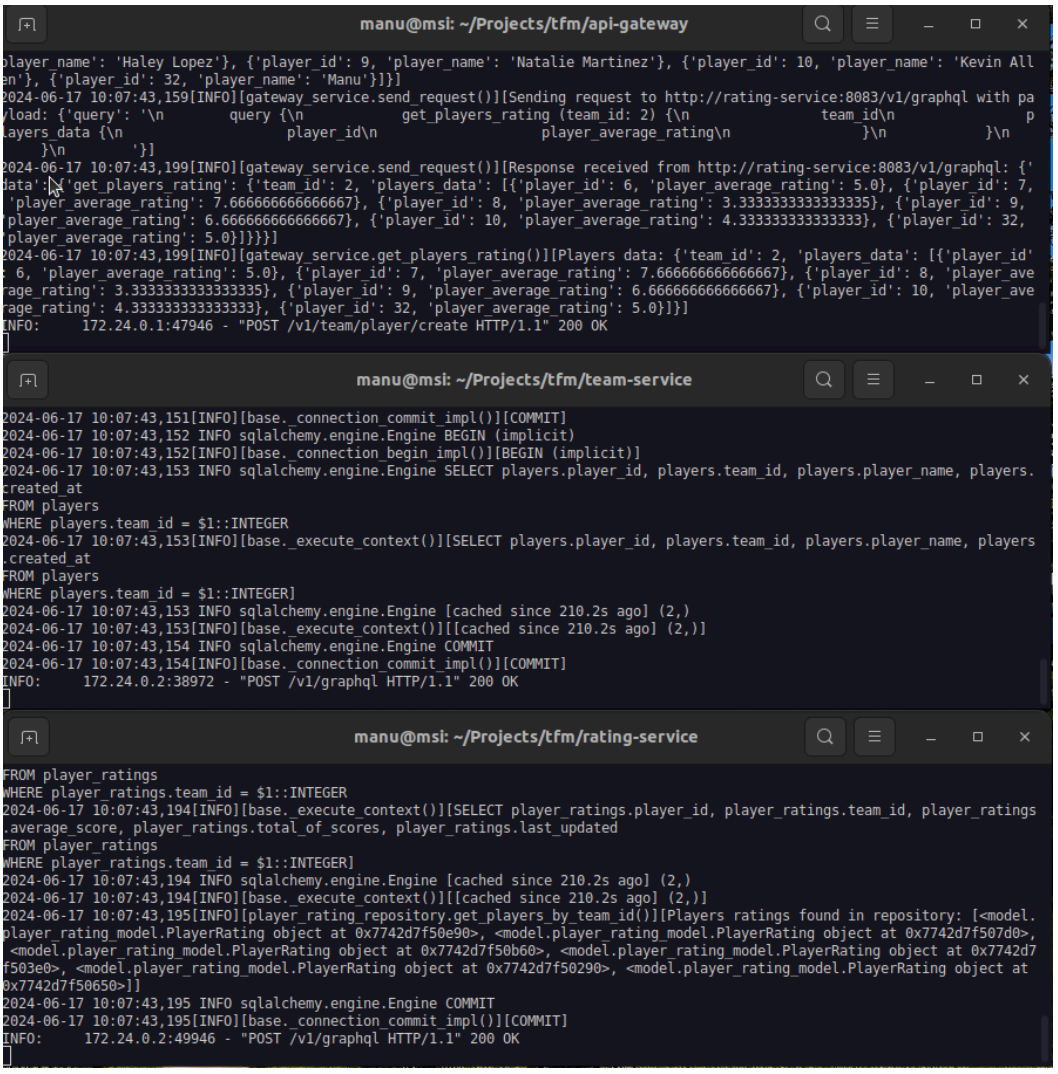






Por último vamos a unirnos a un equipo ya existente y votaremos los jugadores varias veces para que se modifique su puntuación media inicial. Con esto se daría por validados los dos casos de uso principales.

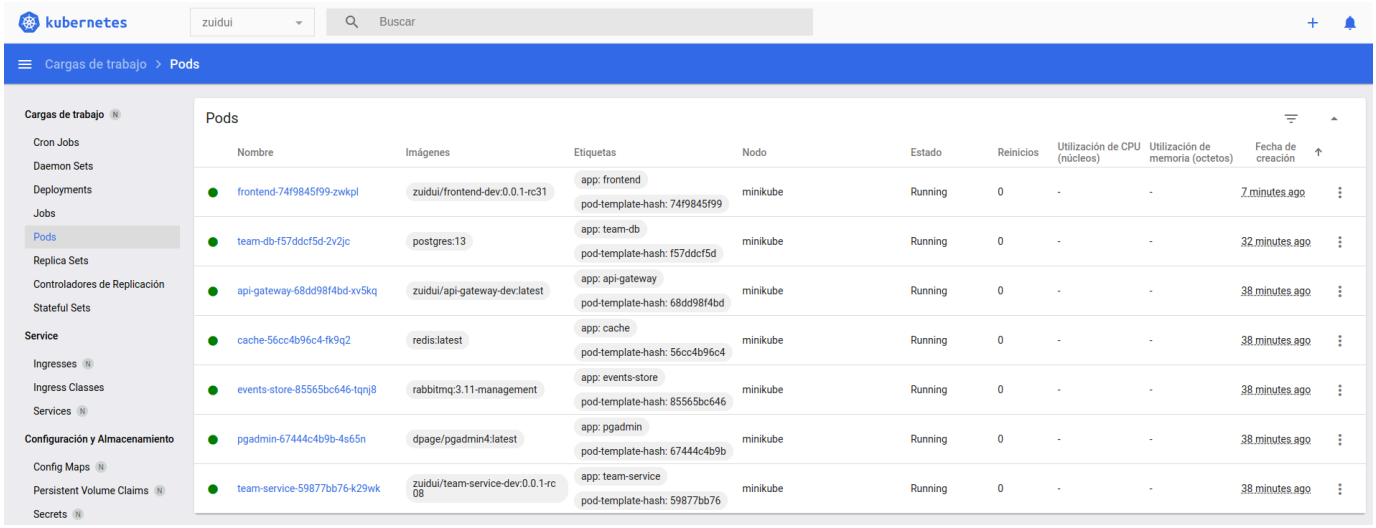


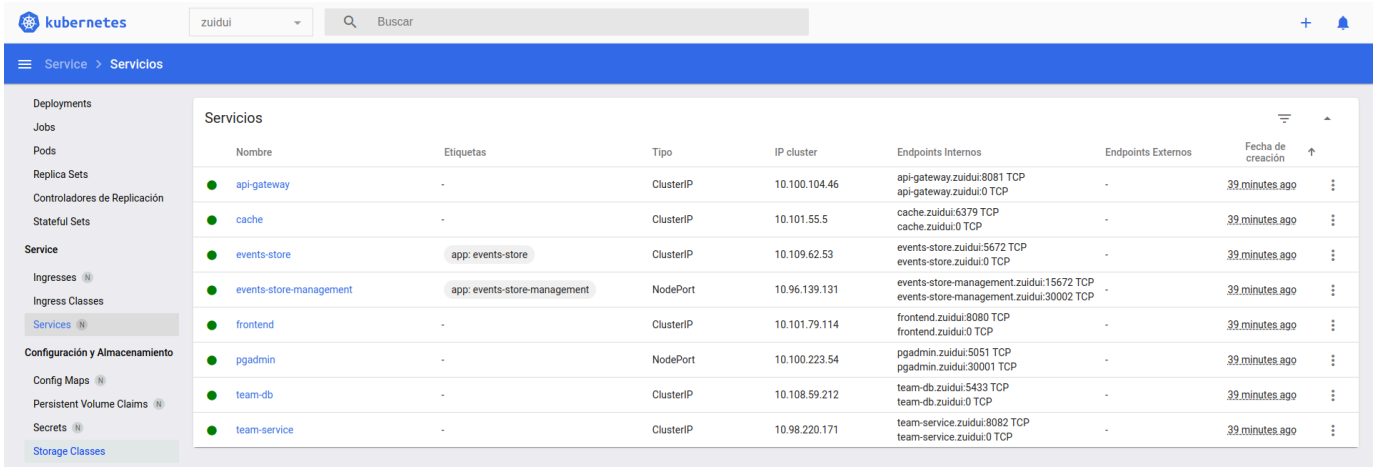


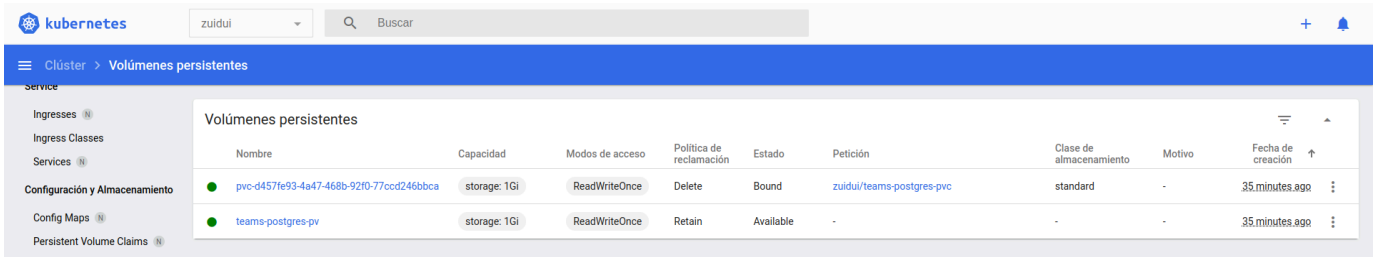
5.2. Validación del sistema

5.2.1. Local - Minikube

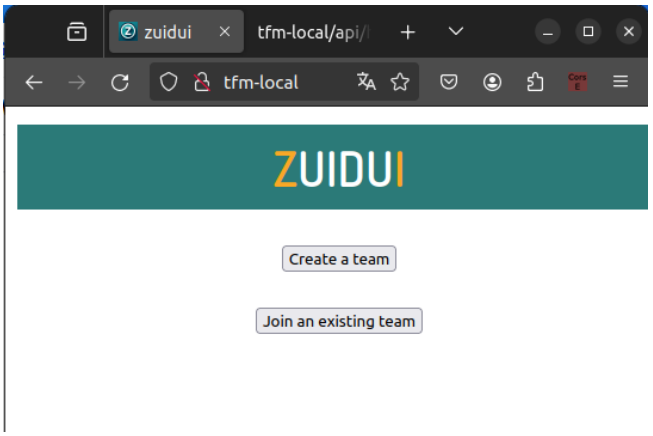
5.2.1.1. Despliegue de recursos

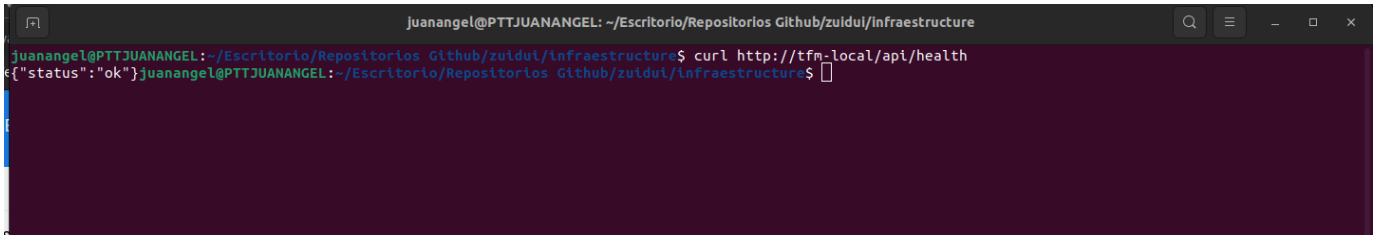






5.2.1.2. Sanidad de servicios





5.2.1.3. Comunicación entre servicios

JGL- captura de los logs

captura logs de los pods

5.2.1.4. Prueba de integración - caso de uso completo

JGL - capturas de los log sy frontend

5.2.2. Cloud - EKS

5.2.2.1. Despliegue de recursos

JGL
captura d

5.2.2.2. Sanidad de servicios

JGL

5.2.2.3. Comunicación entre servicios

JGL

5.2.2.4. Prueba de integración - caso de uso completo

JGL


CloudApps
MASTER

6. Conclusiones y líneas de mejoras

La principal conclusión del proyecto es que la adopción de una arquitectura basada en microservicios fuera del ámbito académico no se justifica para pequeños equipos de desarrollo o aplicaciones en sus primeras etapas. La complejidad adicional difícilmente se compensa con mejoras en el rendimiento a pequeña escala.
No obstante, esta arquitectura proporciona las bases necesarias para construir sistemas altamente escalables y fiables en producción.

La automatización del ciclo de vida de la aplicación y los flujos de CI/CD se han revelado como un valor añadido significativo, siendo estos enfoques fácilmente portables a otros contextos de desarrollo de software.

El proyecto también ha permitido trabajar con tecnologías emergentes como FastAPI, protocolos como GraphQL, y la integración de comunicación asíncrona mediante un broker de mensajería, abordando problemas de concurrencia inherentes.

En términos de infraestructura y cultura DevOps, hemos trabajado de manera asíncrona y utilizado herramientas IaC para automatizar el despliegue, tratando la infraestructura como un componente más del desarrollo, práctica aplicable a otros proyectos.

Como líneas de mejora se proponen las siguientes:

- Nuevas funcionalidades como servicios de administración y de generación de partidos balanceados dentro de un equipo en base a las puntuaciones media.
-
- Añadir sistema de cacheo en las respuestas con Redis para mejorar la latencia.
- Desarrollo del frontend con frameworks modernos como Angular o Vue.
- Implementación del patrón de Sagas para manejo de transacciones.
- Una pirámide de tests más exhaustiva, incluyendo pruebas end-to-end.
- Inclusión de *contact testing* para garantizar las interacciones entre servicios.
- Pruebas de carga utilizando herramientas como Artillery.
- Inclusión de herramientas de QA como SonarQube en el flujo de CI.
- Seguridad del clúster y control de comunicación entre servicios mediante network policies.
- Mejor observabilidad en Kubernetes con métricas usando Prometheus y Datadog para habilitar rollback automático.
- Escalado eficiente de nodos con Karpenter.

- ❖ Soporte en Terraform para múltiples proveedores de cloud.

Anexo - Tareas realizadas

Análisis y preparación

Planificación y organización.

- ☑ Identificar los puntos clave y áreas críticas de atención. Cronograma detallado del desarrollo.
- ☑ Configurar herramientas y entornos necesarios (GitHub, AWS, Minikube, etc.).

Gestión del Proyecto en GitHub.

- ☑ Crear una organización en GitHub para centralizar la gestión del proyecto y los repositorios necesarios.

Configuración inicial.

- ☑ Preparar registro de preproducción y producción.
- ☑ Configurar GitHub Actions para CI/CD, incluyendo validaciones de código y despliegue automático en el entorno de PRE.
- ☑ Análisis de requisitos y definición de historias de usuario.

Estrategia de despliegue.

- ☑ Planificar y configurar la estrategia de despliegue blue/green o canary en el entorno de PRO.
- ☑ Realizar pruebas iniciales de despliegue blue/green o canary en el entorno de PRE para asegurar la correcta implementación y minimizar riesgos.

Diseño de arquitectura.

- ☑ Diseñar la arquitectura de microservicios y definir comunicación entre ellos.
- ☑ Crear diagramas UML para visualizar la arquitectura y las relaciones entre componentes.

Desarrollo e implementación

Implementación de infraestructura.

- ☑ Configurar los recursos de infraestructura necesarios utilizando CloudFormation.
- ☑ Configuración de CRDs y herramientas para el despliegue continuo.

Desarrollo de microservicios.

- ☑ Preparación de los contenedores de desarrollo y scripts para automatizar construcción y pruebas.
- ☑ Desarrollo de los microservicios.

Diseño del frontend:

- ☑ Definir los componentes y la estructura de la interfaz de usuario.

- ☑ Desarrollar un diseño básico del frontend con HTML, CSS y JS.

Integración y despliegue

Integración de los componentes

- ☑ Integrar la comunicación del frontend con los microservicios a través del api-gateway.
- ☑ Integrar el sistema de mensajería para la comunicación basada en eventos entre los microservicios.

Pruebas y validación.

- ☑ Desplegar en Minikube para realizar pruebas de integración entre microservicios
- ☑ Validar la comunicación y la funcionalidad completa del sistema en el entorno de PRE.

Despliegue en producción.

- ☑ Desarrollo de los charts de HELM
- ☑ Realizar el despliegue final en el entorno de PRO utilizando ArgoCD y EKS.

Entrega

- ☑ Incluir guías de configuración, despliegue e información técnica sobre los servicios en los ficheros README de cada repositorio.
- ☑ Desarrollo de la memoria, documentación y refactorización del código.

Anexo - Enlaces de interés y bibliografía

[Repositorios del proyecto]
https://github.com/orgs/zuidui/repositories

[Registro de imágenes del proyecto]
https://hub.docker.com/repositories/zuidui

[Apuntes del máster]
https://www.codeurjc.es/mastercloudapps/

[Stack Overflow]
https://stackoverflow.com/

[Documentación oficial de FastAPI]
https://fastapi.tiangolo.com/

[Documentación oficial de GraphQL]
https://graphql.org/

[Documentación oficial GitHub Actions]
https://docs.github.com/es/actions

[Documentacion oficial Nginx]
https://nginx.org/en/docs/

[Documentación oficial de HELM]
https://helm.sh/es/docs/

[Documentación oficial de Kubernetes]
https://kubernetes.io/es/docs/home/

[Documentación oficial de ArgoCD]
https://argoproj.github.io/cd/

[Documentación ArgoCD Image Updater]
https://argocd-image-updater.readthedocs.io/en/stable/

[Documentación ArgoCD rollout]
https://argo-rollouts.readthedocs.io/en/stable/features/canary/