FUNDAMENTOS DE INGENIERÍA DEL SOFTWARE

MEMORIA TRABAJO PRÁCTICO

GRUPO 5

Máster en Ingeniería del Software: Datos, Cloud y Gestión TI

Universidad de Sevilla

Fafago

César García Pascual & Roberto Hermoso Núñez

Alfonso Bravo Llanos & Pablo García Barco

MCarmen Arenas Zayas & Jesús Monda Caña

Alejandro Guerrero Díaz & Juan Miguel Blanco Ferreira

Organización GitHub: https://github.com/fafagorg

Curso 2020-21

Índice

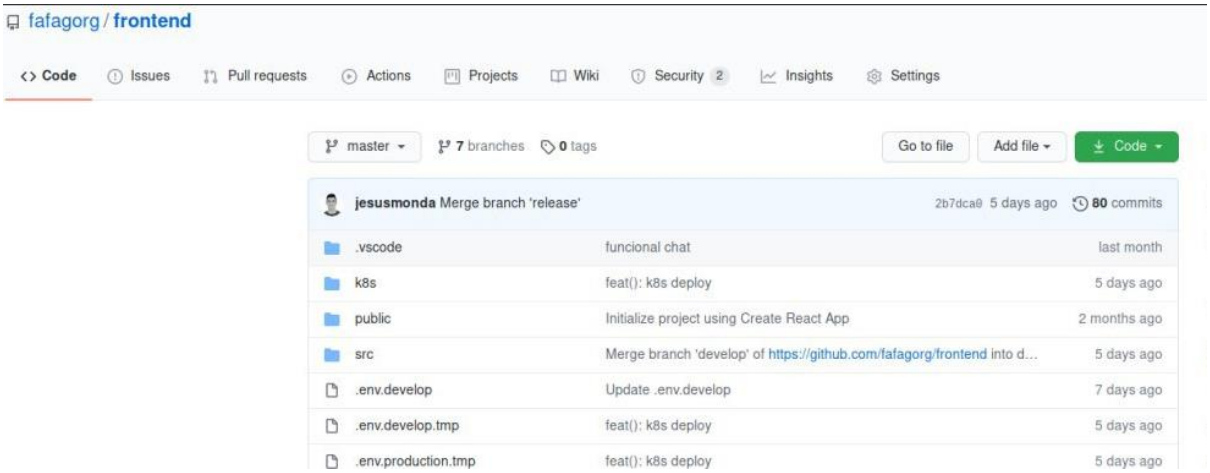
1. Requisitos de la aplicación basada en microservicios avanzada

En este apartado se han cumplimentado los siguientes requisitos:

1. ***Tener un front end común que integre los front ends de cada uno de los microservicios.***

Desde un inicio se acordó tener un front end común sobre el que trabajar todos los grupos, en lugar de hacer un front end por cada pareja y después integrarlos.

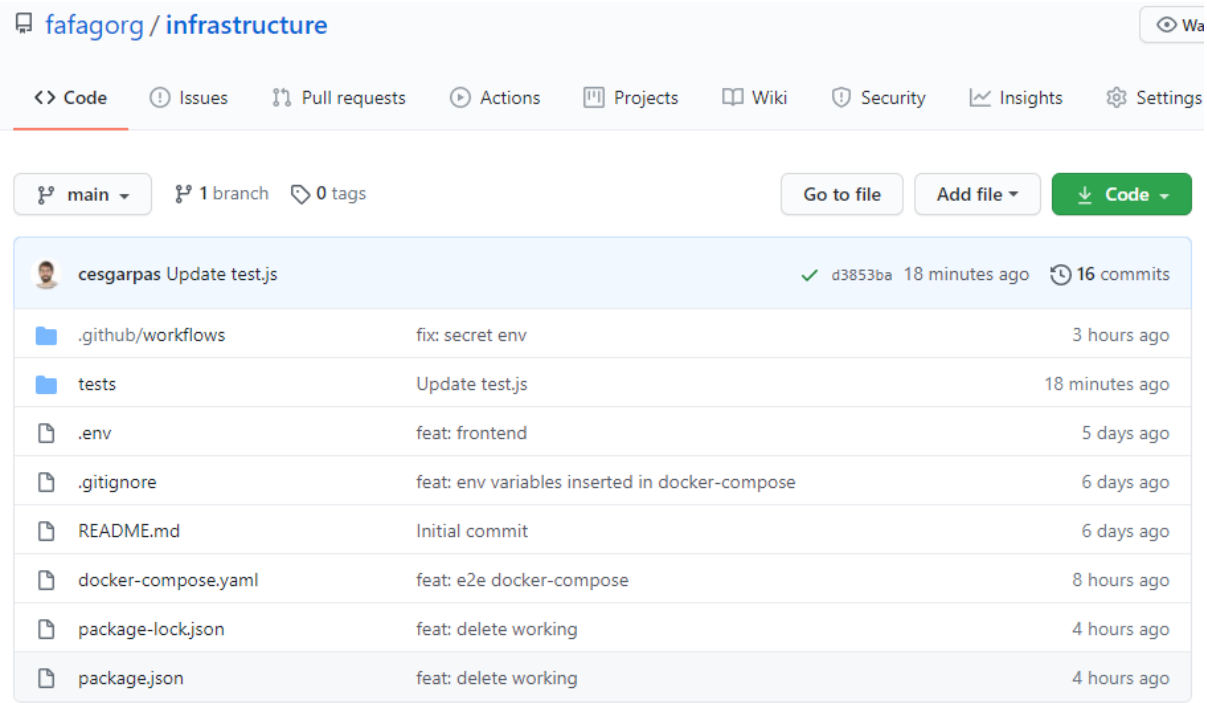
Para ello se creó un repositorio dentro de la organización de GitHub, llamado "frontend": https://github.com/fafagorg/frontend



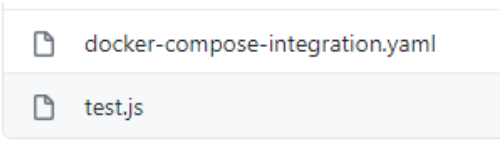
Este repositorio contiene una app hecha en react, que se puede ejecutar de forma sencilla gracias al empleo de docker-compose que levanta los contenedores necesarios y utiliza los microservicios desplegados. Para desplegarlo en producción se utiliza CloudBuild, que es un servicio gestionado por Google y perteneciente a Google Cloud.

2. ***Realizar pruebas de integración automatizadas con los otros microservicios utilizando el sistema de integración continua.***

Para las pruebas de integración, se ha montado un repositorio además de los otros microservicios dentro de la organización llamado "infrastructure" (https://github.com/fafagorg/infrastructure).



Este repositorio contiene un docker-compose con los distintos contenedores (No utilizado actualmente debido al despliegue de Kubernetes que se explica en el siguiente punto) y es un repositorio de node.js. En el package.json están los distintos paquetes y scripts requeridos para ejecutar las pruebas. Accediendo a la carpeta de tests se pueden ver 2 archivos distintos:

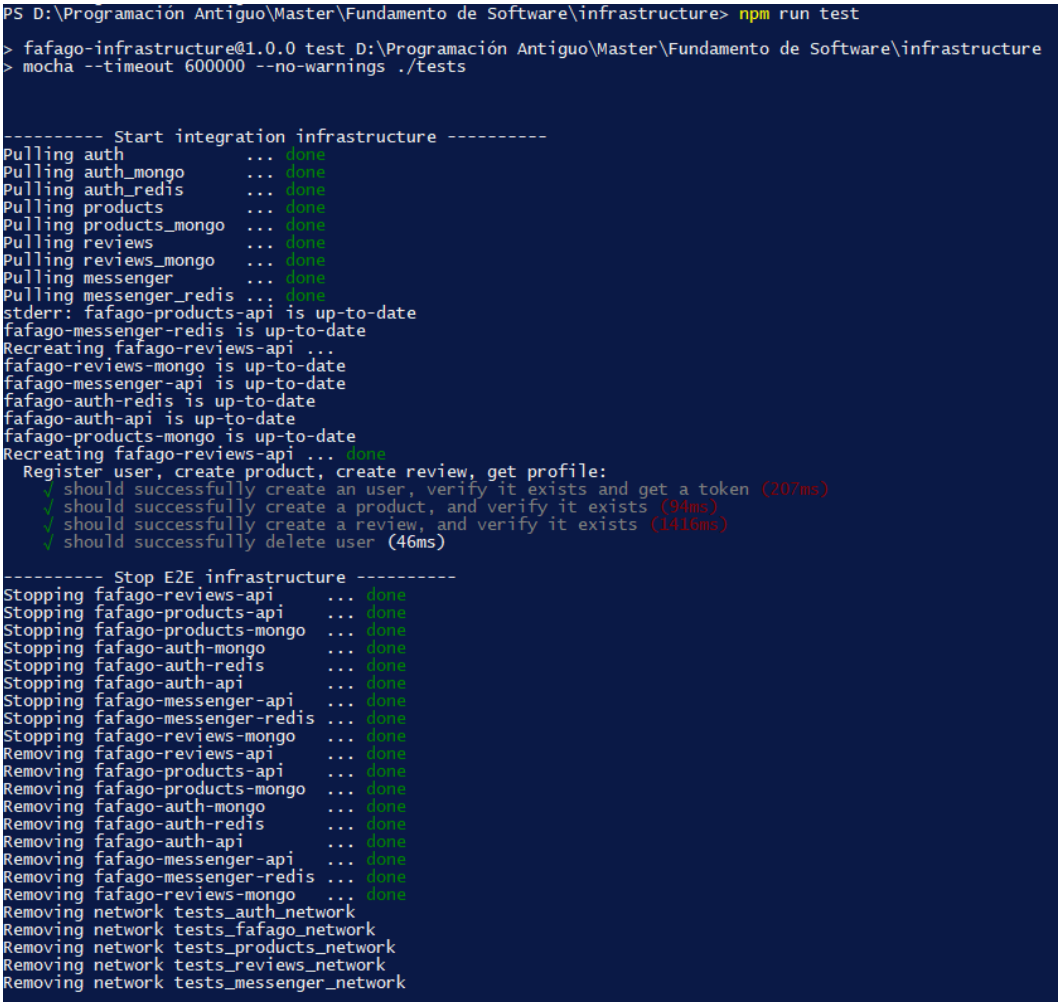


El archivo docker-compose-integration.yaml es un docker-compose con los 4 microservicios y sus respectivas bases de datos configurados para arrancar en local utilizando los puertos 8081-8084 para desplegar los 4 microservicios. No es necesario desplegarlo ya que el archivo test.js se encarga de ello.

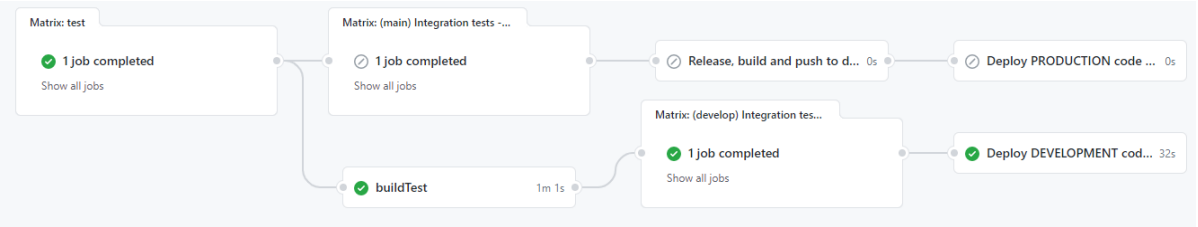
En el archivo test.js, programado con mocha, se pueden diferenciar 4 partes.

- La parte superior contiene los distintos paquetes y librerías necesarias para el funcionamiento de los tests. También hay variables con los distintos objetos necesarios al ejecutarse el test.
- Acto seguido hay un before() el cual hace un pull de las distintas imágenes requeridas para arrancar esta infraestructura. Estas imágenes se actualizan automáticamente mediante el sistema de CI/CD configurado en GitHub actions.
- Después podemos encontrar los tests. Es la sección más larga y en estos se comprueba el correcto funcionamiento de los componentes así como sus integraciones, dividiendo en varias secciones it() para mejorar la legibilidad.
- Por último, en la función after(), se usa docker-compose down para apagar la infraestructura.

Una vez ejecutados, el output sería el siguiente:



Estos tests están integrados con el sistema de CI/CD para verificar antes de desplegar y crear imágenes de producción, que las integraciones entre componentes no han sido invalidadas.

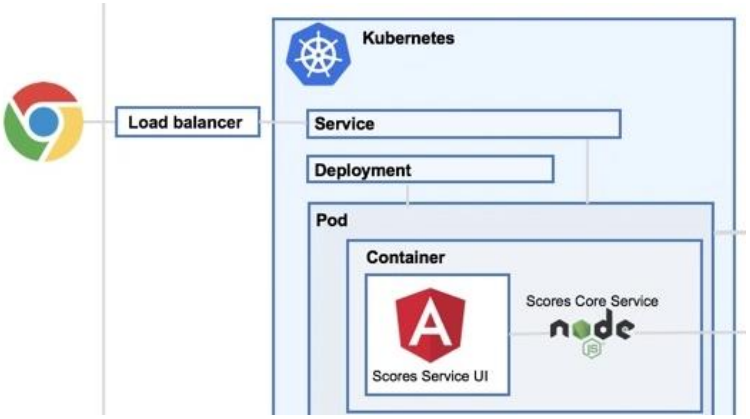


Este workflow se ejecuta en cada componente en cada cambio a la rama de prueba o a la de desarrollo. Este workflow será explicado detenidamente en el punto 6 de este documento.

3. ***Uso de Kubernetes para el despliegue completo de todos los microservicios.*** La aplicación se ha desplegado mediante Kubernetes, que es un sistema para la automatización del despliegue que opera a nivel de contenedores. Para poder utilizarlo se necesita un cluster y como mínimo un nodo (donde se ejecuta un conjunto de contenedores).

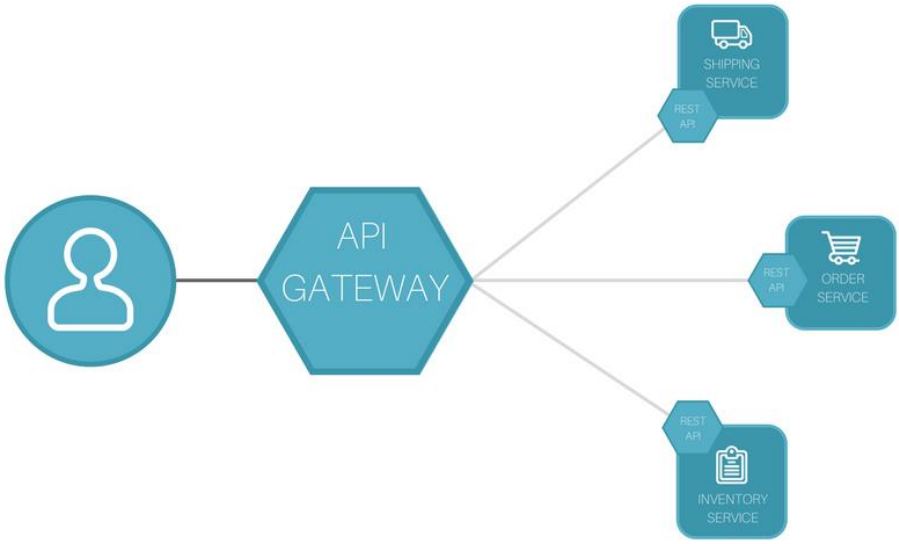
   El cluster ha sido creado utilizando el servicio GKE de Google Cloud, es gestionado por Google y esto ofrece muchas ventajas. Para poder hacer el despliegue, hemos necesitado crear un deployments, un servicio para exponer internamente el servicio y por último un volumen para la persistencia de datos de MongoDB y Redis.

   Un ejemplo de despliegue es el siguiente:



4. ***Hacer uso de un API Gateway.***

   Un API Gateway es el punto de entrada por el que llega todo el tráfico, y que se encarga de unificar la publicación de API en un único endpoint. Para ello se ha creado un ingress que internamente genera un ingress que redirecciona el tráfico desde fafago-dev.alexgd.es a cada uno de los servicios asociado a microservicios.



5. ***Implementación de un mecanismo de autenticación homogéneo para todos los microservicios.***

Se ha realizado una autenticación común en toda la aplicación, usando el microservicio de autenticación. Este módulo dispone de una operación de registro y otra de login, esta última operación usa el paquete jwt para generar los token de autenticación, estos token son los que nos permite saber si el usuario está logueado correctamente o no.

```
module.exports.authLogin = function authLogin (req, res, next) {
  const login = req.user.value;
  databaseRepository.login(login.username, login.password).then((token) => {
    if (token !== undefined) {
      res.status(200).send({
        ok: true,
        user: login.username,
        token: token
      });
    } else {
      res.status(401).send({
        err: 'Username or password wrong'
      });
    }
```

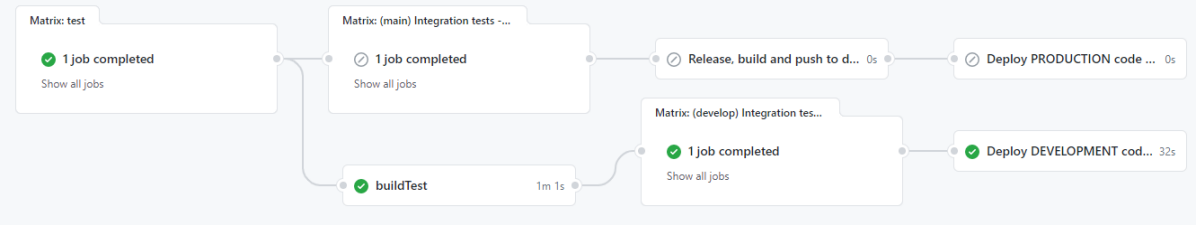
El resto de módulos usarán la operación de "validate", esta operación devolverá un estado u otro dependiendo si el token es válido. De tal manera que, para las operaciones que requieran estar autenticado, los microservicios validan el token que reciben desde el frontend usando esta operación.

```
module.exports.authValidate = function authValidate (req, res, next) {
  const token = req.user.value.token;
  if (token) {
    databaseRepository.validateToken(token).then((valid) => {
      if (valid) {
        res.status(200).send({ userId: valid.user.username });
      } else {
        res.status(403).send({ err: 'Token not valid' });
      }
    });
  } else {
    res.status(401).send({ err: 'Token not provided' });
  }
};
```

La función "validateToken" usará la función "verify" que nos proporciona jwt para desencriptar el token y comprobar que el token es válido.

6. Versionado automático de proyectos y despliegue.

Se ha realizado un workflow complejo utilizando Github Actions para testear el componente y las pruebas de integración, la construcción y subida de imágenes a Docker Hub, despliegue en el servidor usando Kubernetes y la creación de tags y releases en Docker. Usando la captura anterior del punto 2:



Como se ve en este workflow, en primer lugar se lanzarán las pruebas del propio componente y, en caso positivo, dependiendo de la rama se ejecutaría un flujo u otro:

- Rama de desarrollo: En primer lugar se buildea una imagen y se hace push a Docker Hub. Acto seguido se correrían las pruebas de integración con esa nueva imagen y, en caso positivo, se despliega al servidor de Kubernetes.

- Rama de producción: Como en la rama de producción es necesario hacer un PR desde develop, en primer lugar se ejecuta las pruebas de integración con la imagen anteriormente creada en el flujo de desarrollo para confirmar su correcto funcionamiento. Acto seguido se realizaría una release de GitHub, utilizando el número de la versión para buildear y tagear una imagen de docker y subirla a Docker Hub. Por último si todo ha sido satisfactorio se despliega el servidor de producción.

En esta segunda imagen se ve el workflow satisfactorio cuando se hace un merge en la rama de desarrollo. Esto ha producido los siguientes cambios:

- Creación de release: Una vez terminado el flujo, en la imagen de abajo se puede ver como una nueva release se ha creado. En caso de utilizar “Conventional Commits”, una especificación para los mensajes de commit de git, se creará un texto indicando los distintos cambios en un formato ordenado de manera automática.

v1.1.0

cadf913

Compare

v1.1.0

github-actions released this 8 minutes ago · 0 commits to 1d8d12ff4929da87aac17281092fc68c71a5cf28 since this release

Bug Fixes

- action matrix-version (6562841)
- api specification coherence (f4fdaad)
- authorization header (9cf3e19)
- badges (9d90cbe)
- Branch name deployment (4a2f906)
- build and push action (1acb417)
- build id (bdd9607)

Features

- add procfile for heroku deploy (613a28a)
- added codecov to pipeline (d3e5a89)
- Adding continuos deployment (a64469a)
- auth validation token (db99d09)
- badges (05ccbe5)
- CI integration and release (5b36b07)
- code style (3d41aa4)
- cors (ac40eeb)

- Creación de imagen en Docker Hub con ese tag de la release asociada: Como se puede ver en la siguiente imagen, se ha creado una imagen nueva con la release como tag para, en caso de actualizar una versión en el servidor, pueda utilizarse y no perder nunca la release dockerizada.