Facultad
de
Ciencias

AUTOMATIZACIÓN DE LOS PROCESOS DE CONSTRUCCIÓN DE SOFTWARE MEDIANTE LA APLICACIÓN DE TÉCNICAS DEVOPS

(Automation of software construction processes by applying DevOps techniques)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Verónica Fernández González

Director: José Carlos Palencia Gutiérrez

Codirector: Manuel Macho del Río

06 - 2022

INDICE

Resumen TFG español

Partiendo de una aplicación monolítica con procesos de construcción manuales, en este trabajo se desarrollan pruebas para verificar su correcto funcionamiento para posteriormente evolucionar dicha aplicación, separarla en pequeños servicios independientes y automatizar su despliegue empleando técnicas DevOps.

Se lleva a cabo un ciclo completo de software, centrado en la construcción del entorno de trabajo, gestión de repositorios Git, orquestación de servicios mediante contenedores, testing, integración continua (IC) y, por último, monitorización.

Resumen TFG inglés

Starting from a monolithic application with manual construction processes, tests are developed to verify its correct operation to later evolve said application, separating it into miniservices and automating its deployment using DevOps techniques.

A complete cycle of software is carried out, focused on the construction of the work environment, management of Git repositories, orchestration of services through containers, testing, continuous integration (CI), and finally monitoring.

Palabras clave español

DevOps – Contenedores – Automatización – Calidad – Eficiencia

Palabras clave inglés

DevOps – Containers – Automation – Quality – Efficiency

Objetivos TFG

Los objetivos a desarrollar en el presente trabajo fin de grado son los siguientes:

- Modelar, configurar y desarrollar el entorno de trabajo para tareas de desarrollo de aplicaciones empresariales web de gestión.
- Orquestar los servicios en contenedores mediante Docker y Docker-Compose.
- Elaborar tests de integración como pruebas de testing con ayuda de JUnit y Mockito.
- Realizar la integración continua utilizando Jenkins con SonarQube y evaluar la calidad del código intentando encontrar puntos de mejora.
- Independizar del diseño algunas funcionalidades, separándolas en imágenes mediante Docker y luego orquestándolos a través de Kubernetes.
- Supervisar el comportamiento adecuado de la aplicación y comprobar que no haya problemas de seguridad, utilizando Prometheus y Grafana como herramientas de monitorización

1. ¿Qué es DevOps?

DevOps [1][2][3] es la unión de personas, procesos y tecnología para contemplar de forma integrada de extremo a extremo todo el ciclo de vida de una aplicación, siendo un compuesto de desarrollo (Dev) y operaciones (Ops). Se unen todos los equipos (desarrolladores, operaciones, calidad y seguridad) con un fin común, entregar productos de mayor calidad en el menor tiempo posible.

Es una filosofía de trabajo conjunta y compartida, es decir, colaborativa, centrada en la comunicación, colaboración e integración entre desarrolladores software y profesionales en IT, con automatización, mejora continua, centrada en el cliente y con un fin claro, mejorar la calidad del software y acelerar la entrega del mismo.

Por ello los beneficios que se pueden conseguir al usar DevOps son alto rendimiento, mejores productos de mayor calidad y entrega más rápida por lo que los clientes suelen estar más satisfechos. Además, mejora los objetivos comerciales de la compañía, aporta diferentes métricas de la producción y mayor nivel de automatización, dando lugar a un trabajo más cómodo y seguro pues al automatizar un proceso se disminuye la probabilidad del error humano.

Aparece el rol SER (Site Reliability Engineer) encargado de implementar todas las prácticas DevOps. Está en la mitad entre desarrollo y operaciones, ayudando de forma conjunta con ambos equipos (50% con cada uno). Este perfil será el que ponga en práctica en el desarrollo del proyecto.

1.1 Aspectos claves de DevOps (al implicar tanto desarrollo como operaciones)

- Control de versiones: Git, cómo administrar los repositorios de código.
- Integración continua: pipelines en Jenkins, permitiendo automatizar compilaciones y pruebas unitarias al hacer "commit" hacia el repositorio de código.
- Entrega continua: suministro de software rápido y confiable, de manera que se pueda desplegar en producción automatizando dicho proceso.

  Estos dos conceptos están muy unidos, buscando que todo el código que añaden los desarrolladores se pueda integrar de forma constante y entregar en producción de forma continua.

- Infraestructura como código: mediante archivos ".yaml" se realiza la definición de infraestructura en un texto que permite revertir, desmontar y recrear entornos de forma sencilla/automática ejecutando el archivo de texto generado.
- Supervisión y registro: Prometheus y Grafana, utilizados para la monitorización recopilando métricas y datos de rendimiento para optimizar el desempeño de las aplicaciones y detectar fallos.
- Aprendizaje validado: análisis de datos para mejor los procesos en cada nuevo ciclo de desarrollo software.

1.2 ¿Qué herramientas se van a usar en el desarrollo del proyecto DevOps?

- Desarrollador: Repositorios, Maven, microservicios y testing.
- Operaciones: Docker, Kubernetes, Prometheus, Grafana, SonarQube y YAMLs.

Más adelante se trata cada una de ellas, se aportan definiciones y se indica para qué se utilizan en el desarrollo.

2. Esquema de trabajo inicial

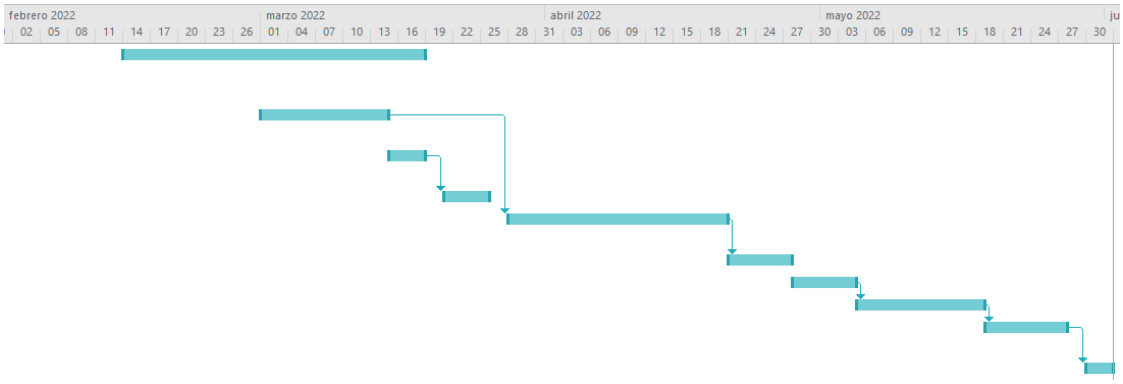
2.1 Tareas a realizar

El esquema de trabajo se divide en las siguientes tareas:

- Desarrollo del entorno de trabajo remoto, que se basará en la elaboración de una máquina virtual con Ubuntu 20.04 y la configuración de esta para soportar el despliegue con Docker, Docker-Compose, etc.
- Configuración del entorno local tanto en cuanto a herramientas (Eclipse, Visual Studio Code, Sourcetree, etc.) como en la creación de los componentes necesarios (.jar ejecutable y empaquetado web).
- Despliegue local del proyecto.
- Configuración del entorno remoto para poder desplegar el proyecto (YAMLs, Dockerfiles, etc.).
- Despliegue remoto del proyecto.
- Desarrollo de alguna prueba de testing (JUnit y Mockito).
- Integración continua (ejecución de tests mediante Jenkins y generación del compilado).
- Despliegue con Kubernetes (independizando de Docker el frontend y backend de la aplicación).
- Inclusión de herramientas de monitorización (Prometheus y Grafana).

2.2 Diagrama de Gantt

Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
Formación en herramientas y técnicas DevOps	25 días	lun 14/02/22	vie 18/03/22	-
Desarrollo entorno de trabajo remoto	10 días	mar 01/03/22	lun 14/03/22	-
Configuración entorno local	4 días	mar 15/03/22	vie 18/03/22	-
Despliegue local	5 días	lun 21/03/22	vie 25/03/22	3
Configuración entorno remoto	18 días	lun 28/03/22	mié 20/04/22	2
Despliegue remoto	5 días	jue 21/04/22	mié 27/04/22	5
Testing	5 días	jue 28/04/22	mié 04/05/22	
Integración continua	10 días	jue 05/05/22	mié 18/05/22	7
Independización Kubernetes	7 días	jue 19/05/22	vie 27/05/22	8
Monitorización	3 días	lun 30/05/22	mié 01/06/22	9



3. Desarrollo del entorno de trabajo, ¿por qué usar Vagrant?

Vagrant [4] es una herramienta que permite crear, configurar, administrar e instalar máquinas virtuales a partir de un simple archivo de configuración. La clave es que lo hace de forma automática, y que los entornos generados son fáciles de reconfigurar si fuera necesario, reproducir, y son portables. Además, permite configurar automáticamente carpetas compartidas, conexiones SSH, etc.

Es decir, Vagrant permite configurar un entorno de desarrollo completo con las herramientas necesarias para comenzar a trabajar evitando tener que perder el tiempo en instalar dichas herramientas de una en una. Así cualquiera que se incorpore nuevo a un trabajo, proyecto o desarrollo podrá hacerlo de una forma mucho más rápida y eficiente. Además, también facilita los cambios y nuevas configuraciones dentro del entorno de trabajo.

Vagrant es una de las herramientas DevOps imprescindibles, y está diseñada para que todo el mundo pueda crear un entorno virtual de forma fácil y rápida apoyándose en VirtualBox.

VirtualBox funciona en segundo plano dando soporte a Vagrant, y Vagrant funciona como una línea de comandos para VirtualBox.

Contextualizada la situación de trabajo, se pasa a la primera tarea, configuración del entorno de trabajo mediante Vagrant.

3.1 Aprovisionamiento y generación del entorno de trabajo [5]

Mediante Vagrant, se crea una máquina virtual partiendo de la imagen de Ubuntu 20.04 oficial. Se elige esta versión ya que es la última con soporte LTS.

```
vagrant init ubuntu/focal64
```

Descargada la base, se pasa a configurar el Vagrantfile para aprovisionar la máquina con una serie de herramientas necesarias para el desarrollo del proyecto: Docker, Docker-Compose y Kubernetes (kubectl & minikube). Esto se ha realizado mediante scripts personalizados en bash que contemplan todas las acciones necesarias.

En Vagrant se dispone de dos opciones para llevar a cabo el aprovisionamiento [6], a través comandos con una shell inline o mediante paths a scripts externos.

```
66    config.vm.provision "shell", inline: <<-SHELL
67        apt-get update
68        apt-get upgrade -y
69        scripts/instalar-docker.sh
70    SHELL
71  end
```

Inline Shell

```
70    config.vm.provision "shell", path: "scripts/instalar-docker.sh"
71  end
```

Path

Se elige la segunda opción, puesto que es más clara y sencilla en cuanto a aspectos organizativos tener una carpeta con todos de scripts. Estos scripts de aprovisionamiento estarán, además de localmente, en una carpeta en un repositorio remoto (Git) junto con el archivo Vagrantfile que los llama y define también otros aspectos de la máquina.

Sin embargo, pensando a la larga y en el aprovisionamiento de otros elementos que no sean herramientas, como una versión de base de datos, será más cómodo para alguien nuevo que se introduzca en el proyecto poderse descargar la imagen de Vagrant con todo instalado, sin preocuparse por su aprovisionamiento. Además, de este modo no se corre el riesgo de que al aprovisionar la máquina en el equipo de otra persona y en otro momento de tiempo se descargue una versión más actualizada de algún componente que pueda generar fallos. Por este motivo, una vez aprovisionada la máquina se generará una imagen de la máquina ya aprovisionada para subir a Vagrant Cloud.

Para realizar el aprovisionamiento se ejecuta el siguiente comando:

```
vagrant up --provision
```

Y para comprobar si se han instalado las diferentes herramientas, se hace una conexión mediante ssh con la máquina y se ejecutan los siguientes comandos:

```
vagrant ssh
docker version
docker-compose --version
kubectl version --client
minikube version
minikube start      (para parar → minikube stop)
```

Durante la instalación de las herramientas surgió un problema, y es que Minikube requiere 2 CPUs y 2GB de memoria RAM como requisitos mínimos para funcionar. Por ello, también se modificó el Vagrantfile, para darle dichas capacidades a la máquina.

Para conseguir mayor utilidad, practicidad y facilidad en la administración del sistema y futura utilización de este también se provee la máquina con una interfaz gráfica. En este caso se ha decido instalar la interfaz de Gnome, instalada con ayuda de la herramienta "tasksel".

```
vagrant up
vagrant ssh → sudo apt-get update
              sudo apt tasksel -y
              sudo tasksel install ubuntu-desktop
              sudo reboot
```

Una vez instalado, también se puede comprobar el estado del escritorio con el siguiente comando:

```
vagrant ssh
systemctl status gdm
```

Finalizado el aprovisionamiento de la máquina, se pasa a generar la imagen que posteriormente añadiré a Vagrant Cloud.

La autenticación de Vagrant funciona de la siguiente manera. Al arrancar en un equipo local se generan un par de llaves pública y privada propias únicamente de dicho equipo, por lo que si se genera la imagen de la máquina sin modificar estas llaves cualquier otro usuario no podrá utilizar la máquina por problemas de autenticación.

Para solucionarlo, se sustituye la llave pública de la máquina por la llave pública de Vagrant obteniéndola del repositorio de Mitchell Hashimoto (fundador de HashiCorp y creador de Vagrant). Se consigue con los siguientes comandos:

```
vagrant ssh
wget --no-check-certificate
https://raw.github.com/mitchellh/vagrant/master/keys/vagrant.pub -O
/home/vagrant/.ssh/authorized_keys
chmod 0600 /home/vagrant/.ssh/authorized_keys
chown -R vagrant /home/vagrant/.ssh
```

Realizado este cambio, la imagen generada a partir de esta máquina funcionará correctamente en cualquier usuario, puesto que en el arranque de la máquina detectará la llave pública insegura de Vagrant y la sustituirá generando un par de llaves (pública y privada) locales que permitan la autenticación.

```
==> default: Waiting for machine to boot. This may take a few minutes...
    default: SSH address: 127.0.0.1:2222
    default: SSH username: vagrant
    default: SSH auth method: private key
    default: Warning: Connection reset. Retrying...
    default: Warning: Connection aborted. Retrying...
    default:
    default: Vagrant insecure key detected. Vagrant will automatically replace
    default: this with a newly generated keypair for better security.
    default:
    default: Inserting generated public key within guest...
    default: Removing insecure key from the guest if it's present...
    default: Key inserted! Disconnecting and reconnecting using new SSH key...
==> default: Machine booted and ready!
```

Para crear la imagen Vagrant que se subirá al Vagrant Cloud se usa el siguiente comando (estando situados en el directorio donde se quiere guardar la imagen generada):

```
vagrant package --base TFG_Entorno_... --output Entorno.1x64.box
```

Siendo base el nombre en VirtualBox de la máquina aprovisionada, y output el nombre del fichero de salida, que es el archivo box empaquetado que se subirá a Vagrant Cloud. [7]

Una vez subida la imagen, para descargarla y comprobar su funcionamiento se ejecutaría:

```
vagrant init veronicafg/devops-box
vagrant up          (arranca la máquina)
vagrant ssh         (entra en la máquina, donde se puede ejecutar cualquier comando)
vagrant halt        (apaga la maquina)
```

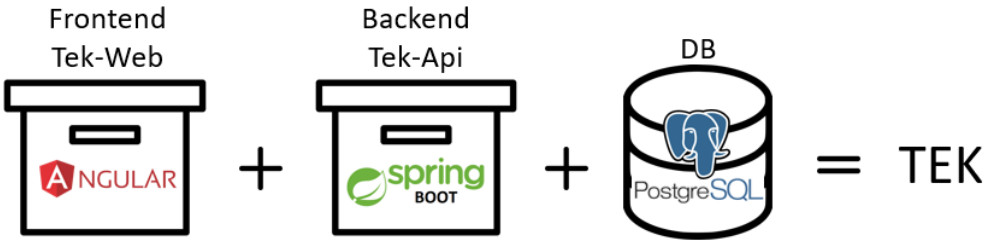
4. Configuración de las herramientas para despliegue local

El proyecto sobre el que trabajaré y cuya infraestructura voy a proceder a configurar recibe el nombre Tek. Se corresponde con una plataforma de gestión de comunidades energéticas locales (CEL) desarrollada para una importante empresa de distribución de energía eléctrica y sus clientes, pues también sirve para proporcionar servicios de información y gestión a los miembros de una CEL, es decir, al conjunto de clientes que comparten una o varias instalaciones fotovoltaicas pertenecientes a una misma comunidad energética.

Con la aplicación desarrollada se permite por tanto la gestión de CELs (Comunidades Energéticas Locales), gestión de los contratos/notificaciones y de los datos de consumo/generación de los miembros de una CEL, y gestión de votaciones.

Para ello se desarrolla una API (formada por un frontend y un backend), y se diseña una base de datos relacional que almacena los datos asociados a las diferentes CEL y sus miembros.

Visto en resumen, la aplicación se conforma de un frontend en Angular al que se denomina tek-web, y un backend en SpringBoot (basado en Java 1.8.0_171) al que se denomina tek-api, y utiliza una base de datos PostgreSQL. Tanto el front como el back se encuentran almacenados en repositorios independientes de GitLab.

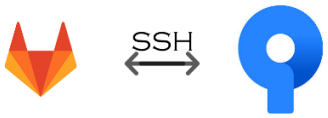


Por ello para manejar y desplegar el proyecto se requieren herramientas para la gestión de los repositorios (GitLab con Sourcetree), dos herramientas de gestión de código (Visual Studio Code y Eclipse), Java, herramientas para Angular que son Node.js (entorno de ejecución para JavaScript) y npm (Node Package Manager, sistema de gestión de paquetes por defecto para Node.js) y por último PgAdmin para gestionar la base de datos, Postman para probar conexiones y Nginx para hacer el despliegue.

La instalación y configuración de estas herramientas realizará sobre el equipo local Windows para poder desplegar el proyecto localmente y analizar su funcionamiento. Así próximamente podré generar las imágenes y contenedores necesarios para poder hacer su despliegue de forma automática mediante Docker, usando para ello como servidor una máquina virtual remota (la generada en la primera parte del trabajo).

El proceso de configuración se describe a continuación.

4.1 Uso de repositorios GitLab administrados a través de Sourcetree [8]



Cada parte se encuentra en un repositorio del GitLab interno de la empresa, CIC. Para poder desplegar el proyecto, por tanto, desde la herramienta de gestión de repositorios Sourcetree se realiza el “clone” de los mismos.

Sin embargo, para poder manejar los repositorios se requiere autenticación mediante claves ssh. Por este motivo previamente genero un par de llaves utilizando putty e incluyendo la llave en los tokens de GitLab y de la herramienta Sourcetree.

4.2 Instalación de Node.js y npm [9]

En este caso, basta con descargar del repositorio oficial de Node.js el instalador de la herramienta y seguir su proceso de instalación. Una vez finalizado el proceso se puede comprobar la instalación mirando la versión de las herramientas:

```
Windows PowerShell
Copyright (C) Microsoft Corporation. Todos los derechos reservados.

Instale la versión más reciente de PowerShell para obtener nuevas características y mejoras.
https://aka.ms/PSWindows

PS C:\Users\vfernandez> node -v
v16.14.2
PS C:\Users\vfernandez> npm -v
8.5.0
PS C:\Users\vfernandez>
```

4.3 Configuración Java [10]

Descargo el instalador de la versión concreta de Java requerida para el proyecto de los repositorios oficiales de Oracle. Después prosigo con el proceso de instalación, tomando nota del directorio en el que se instalará para poder establecer la variable de entorno Java del sistema apuntando a dicho directorio.

Una vez establecida la variable de entorno, compruebo la instalación al igual que en el caso anterior comprobando la versión de Java instalada:

```
Microsoft Windows [Versión 10.0.22000.613]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\vfernandez>java -version
java version "1.8.0_171"
Java(TM) SE Runtime Environment (build 1.8.0_171-b11)
Java HotSpot(TM) 64-Bit Server VM (build 25.171-b11, mixed mode)

C:\Users\vfernandez>
```

4.4 Gestión de código mediante Eclipse y Visual Studio Code [11][12]

Estas herramientas se instalan obteniendo sus instaladores de sus páginas oficiales y siguiendo el proceso de instalación. Lo único extra que hay que configurar específicamente es en eclipse, añadiendo el jdk de la versión Java instalada previamente, que es la que utiliza el proyecto.

4.5 PgAdmin, Postman y Nginx [13][14][15]

PgAdmin y Postman se utilizan para, una vez levantada la aplicación, comprobar su funcionamiento, pero no requieren configuración específica. PgAdmin se utiliza para poder visualizar la base de datos PostgreSQL, y Postman para probar la conexión correcta entre la web (front) y el API (back).

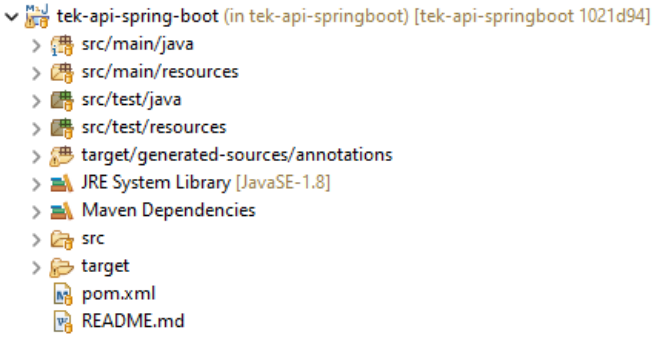
Nginx sin embargo se trata de servidor web/proxy inverso ligero de alto rendimiento, y es sobre el que se levantará la web. Sin embargo, al contrario que las aplicaciones anteriores, sí requiere una pequeña configuración modificando su fichero de configuración "nginx.conf" para indicar la ruta hacia el empaquetado de la web.

5. Despliegue del proyecto en infraestructura local

Presentadas todas las herramientas necesarias para el despliegue del proyecto, expongo el proceso que hay que llevar a cabo para desplegarlo.

Como he mencionado, los dos componentes principales del proyecto se descargan de los repositorios GitLab de la corporación CIC y se incluyen en la herramienta Sourcetree, que permite una navegación más cómoda y visual entre versiones, ramas, etc.

Hecho esto, con Eclipse se abren el backend, es decir "tek-api-spring-boot". Con esta pieza se realiza el despliegue de la parte de SpringBoot. El contenido se muestra a continuación:



Como ficheros destaca el "pom.xml" y el README.

El primero, "pom.xml", se corresponde con un fichero donde se encuentran todas las dependencias, ya que se utiliza Maven para gestionarlas.

El README contiene una pequeña explicación del proyecto y de SpringBoot.

Cabe destacar que este proyecto se ha adaptado para el desarrollo DevOps, siendo uno de los cambios más significativos la evolución del uso de Spring a SpringBoot.

Otro componente importante, aparte de las librerías de dependencias Maven y la versión JRE adecuada para el compilado (Java 1.8.0_171), es la carpeta "src". Dentro de esta carpeta se encuentra todo el código de funcionamiento de la API, que cuenta entre otros aspectos con Hibernate, Flyway para realizar la carga de los datos SQL, Actuator para obtener información del funcionamiento y saber si es correcto, y muchos otros componentes. Destacar que para la seguridad se usa OAuth2.

Por comodidad, y debido a que son los archivos más importantes, se generan accesos directo a "src/main/java", "src/main/resources", "src/test/java" y "src/test/resources".

El desarrollo de la API se basa en un modelo vista-controlador. Es decir, dentro de "src/main/java" se tienen los diferentes paquetes correspondientes a la implementación (código) de los controladores, que son accedidos siempre que se haga referencia alguno de sus métodos (crear, modificar, eliminar, actualizar, etc.) para modificar una entidad (CEL, contrato, notificación, votación, usuario, dato de generación/consumo, etc.). Para ello se pasa de los controladores a los servicios, de los servicios a los repositorios, y de los repositorios a la entidad. La peculiaridad de este proyecto es que la parte de las vistas se desarrolla en Angular (en la parte web/front del API).

Por otro lado, en "src/main/resources" está la definición de por ejemplo las propiedades del proyecto, y los datos SQL de la base de datos.

Por último, en las carpetas "src/test/…" se tiene el contenido necesario para los test, que se explicará más adelante.

Con todos estos elementos se generará el .jar ejecutable de la aplicación, apoyándose para ello en Maven. El .jar se situará dentro de la carpeta denominada "target".

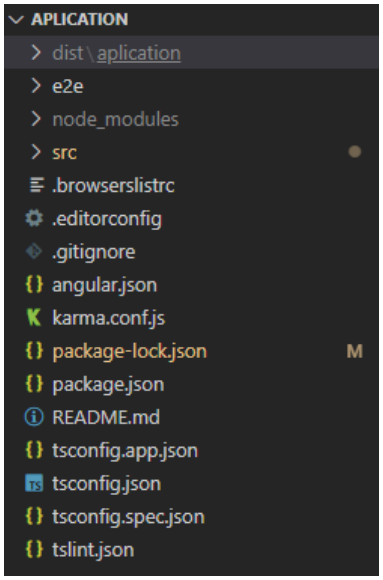
Por otro lado, se abre el frontend, es decir “tek-web”, con Visual Studio Code. Al estar desarrollada en Angular con dos sencillos comandos se puede tanto arrancar como generar su empaquetado. Este se situará dentro de una carpeta llamada “dist”.

Angular se apoya en las herramientas instaladas previamente NPM y Node.js, y se hará el primer uso de ellas para instalar la detección de comandos de Angular (para lo que se introduce en terminar Windows “npm install -g @angular/cli”).

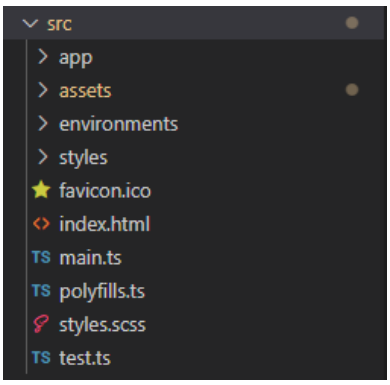
En segundo lugar, desde la terminal de Visual Studio Code, se instalan las dependencias del proyecto ejecutando “npm install”. A continuación, se arranca la web del proyecto para lo que se usa el comando “npm run start”. Finalizada la ejecución de este comando, siempre que esté levantado también el back, la web se despliega y se puede acceder a ella indicando el puerto en un navegador.

Por último, para generar el empaquetado de la web se ejecuta “ng build --prod --base-href=”/tek-web/””. El empaquetado se situará dentro de la subcarpeta denominada “dist”, que se genera al formar el primer empaquetado.

El frontend se conforma de distintas carpetas y ficheros:



- dist: empaquetado de la aplicación.
- e2e: elementos del IDE (entorno de desarrollo integrado), en este caso Visual Studio Code.
- node_modules: paquetes que necesita el proyecto (que es lo que se descarga con “npm install”) que se cargan en base al fichero “package.json” que tiene todo información acerca del proyecto (los scripts, comandos de compilación para arrancar el proyecto, construirlo, etc.; dependencias, paquetes que hay que instalarse para que funcione el proyecto; herencias, ya que cada paquete puede tener a su vez otras dependencias que se indican en “package-lock.json”)
- src: todos los componentes de la web, siendo lo más importante su subcarpeta app, donde se definen todos los componentes.



Dentro de src hay diferentes subcarpetas y ficheros mostrados en la imagen, pero la más importante es “app”

“app” contiene las vistas (pages) que representan todo lo que se mostrará en la web (contratos, comunidades, pantalla de login, etc.), servicios (services) que son llamadas get, post, create, etc. para obtener la información a mostrar, modelos (models) que sirven para navegar entre objetos, y el client se define en el constructor. Las páginas llaman a los servicios y son los servicios los que tienen las funciones para llamar y comunicarse con el api.

Explicados los componentes de la aplicación, generado el .jar y el empaquetado web, se procede al despliegue completo. Para ello se utiliza el servidor web Nginx.

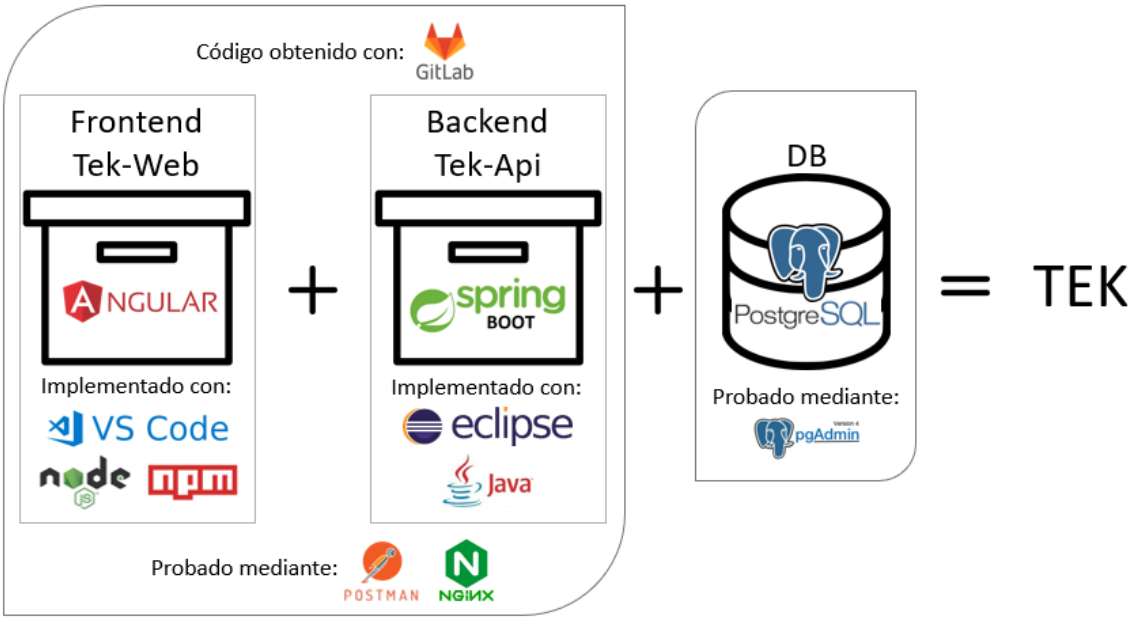
En primer lugar, se abre un terminal Windows en el path del .jar y se ejecuta “java -jar”. Tras esto, se despliega el back.

En segundo lugar, se incluye el empaquetado web dentro del servidor Nginx (en la carpeta html). A continuación, se ejecuta en un terminal Windows en el path de Nginx “start nginx”.

Con estos dos pasos, se arranca tanto el back como el front, y se pueden visualizar usando para ello un navegador e indicando como URL “localhost:<puerto>/tek-api” para comprobar el back, y “localhost:<puerto>/tek-web” para ver el front.

Además, para probar el funcionamiento de la conexión entre back y front se puede utilizar Postman, y para ver la base de datos se tiene PgAdmin.

Por tanto, de forma local la aplicación queda operativa de la siguiente manera:



Tras el despliegue y análisis de los componentes del proyecto queda claro que se requerirán al menos tres contenedores Docker, siendo uno la base de datos PostgreSQL, otro el backend “tek-api”, y otro el frontend “tek-web”. Y por este mismo motivo las imágenes necesarias serán la imagen con la versión PostgreSQL del proyecto (postgres:11.12), la imagen del back con SpringBoot arrancando desde el .jar, la imagen del front con Angular arrancando de su empaquetado web, una imagen Nginx para el servidor web, y la imagen Java con la versión del proyecto (openjdk-8).

Es decir, la serie de requisitos técnicos que se tendrían que instalar y disponer en cada máquina en la que se quisiera desplegar Tek son:

- Servidor web (Nginx)
- Java 1.8
- Base de datos (PostgreSQL)

Actualmente la instalación de estos requisitos se tendría que hacer manualmente en cada máquina, sin embargo, al finalizar este trabajo gracias a las técnicas DevOps el despliegue se hará de forma automática. Entre otros aspectos esto mejora el mantenimiento y portabilidad de la aplicación puesto que solo se tiene que mantener actualizada de forma manual una máquina, para el resto el despliegue se hace a partir de la configuración de la primera máquina de forma automática.

6. Configuración de la infraestructura remota

6.1 ¿Qué es Docker? ¿Por qué su uso como técnica DevOps es importante?

Docker [16][17] es una tecnología de virtualización que sirve para empaquetar aplicaciones haciendo que sean portables y más fáciles de distribuir. Es decir, es una plataforma de software que permite crear, probar e implementar aplicaciones de forma rápida automatizando su despliegue usando contenedores.

Su utilización como técnica DevOps evita tener que instalar todas las herramientas necesarias para probar una aplicación de forma manual. En su lugar, se incluye todo en el Docker que se descarga, y que nada más instalarse permite arrancar la aplicación. Se asemeja a trabajar con VirtualBox, pero no se instala una máquina virtual, sino que se usa el sistema operativo subyacente.

Por este motivo también ayuda en el escalado, de manera que si fuera necesario añadir más servidores o sustituir un servidor actual por un fallo la configuración de los nuevos servidores se realizaría de forma sencilla y automática, estando listos para ponerse en funcionamiento rápidamente.

En resumen, usando Docker Engine la configuración cambia. Se incluyen en un componente Docker Image los requerimientos de la aplicación (Java, servidor web, base de datos, etc.), y a partir de dicha imagen se generan contenedores en cada nodo (equipo, servidor, etc.), siendo necesario únicamente configurar como herramienta en cada nodo Docker.

6.2 ¿Qué es Docker-Compose?

Docker Compose [18] es una herramienta para definir y ejecutar aplicaciones de Docker de varios contenedores a partir de un archivo YAML que configura los servicios de la aplicación. Es decir, es una herramienta con la que se pueden generar las imágenes con diferentes contenidos de una aplicación.

Por ejemplo, si se quiere generar una imagen con PostgreSQL se puede hacer directamente un “pull” de la imagen deseada que se encuentra en Docker Hub (repositorio de imágenes Docker) y “run”, pero también se puede generar mediante un archivo YAML que es un script que generará la imagen tras ser ejecutado con Docker Compose.

6.3 Despliegue en remoto de un único servicio

Para facilitar la comprensión del funcionamiento de estas herramientas explico a continuación cómo se generaría la imagen y el contenedor para la base de datos. Además, aprovecho para exponer como se procederá a funcionar posteriormente (exposición de puertos para poder ver la app desde equipo local, no solo en el servidor).

Para comenzar, como se va a trabajar desde una máquina remota, se utiliza Vagrant para obtener la imagen de la máquina y configurar en enrutamiento de puertos. Esto se debe a que la máquina virtual gestiona sus puertos de forma interna, y como queremos que lo desplegado dentro de la VM sea accesible de forma remota, es decir desde el equipo local, se deben exponer los puertos sobre los que se levanten los diferentes contenedores o servicios que se quieran exponer.

En este caso, se va a desplegar únicamente el servicio de base de datos que se levanta en el puerto 5432, por ello se incluyen en el Vagrantfile las siguientes líneas marcadas:

```
1  # -*- mode: ruby -*-
2  # vi: set ft=ruby :
3
4  # All Vagrant configuration is done below. The "2" in Vagrant.configure
5  # configures the configuration version (we support older styles for
6  # backwards compatibility). Please don't change it unless you know what
7  # you're doing.
8  Vagrant.configure("2") do |config|
9    # The most common configuration options are documented and commented below.
10   # For a complete reference, please see the online documentation at
11   # https://docs.vagrantup.com.
12
13   # Every Vagrant development environment requires a box. You can search for
14   # boxes at https://vagrantcloud.com/search.
15   config.vm.box = "veronicafg/devops-con-interfaz-esp"
16
17   # Disable automatic box update checking. If you disable this, then
18   # boxes will only be checked for updates when the user runs
19   # `vagrant box outdated`. This is not recommended.
20   # config.vm.box_check_update = false
21
22   # Create a forwarded port mapping which allows access to a specific port
23   # within the machine from a port on the host machine. In the example below,
24   # accessing "localhost:8080" will access port 80 on the guest machine.
25   # NOTE: This will enable public access to the opened port
26   # config.vm.network "forwarded_port", guest: 80, host: 8080
27   config.vm.network "forwarded_port", guest: 5432, host: 5432
```

Con estas líneas se define que se parta de la imagen desarrollada en la primera parte de este trabajo, que es una VM con todas las herramientas necesarias para funcionar con Docker, y que se exponga en la máquina local en el puerto 5432 el puerto remoto 5432. El resto del Vagrantfile se mantiene en su estado por defecto, comentado.

Tras esto, se despliega la máquina:

```
vagrant up
```

Se puede comprobar si se están exponiendo los puertos declarados en el Vagrantfile con el siguiente comando:

```
vagrant port
```

Por comodidad, como se aprovisionó la máquina de interfaz gráfica, se puede optar por seguir con la configuración mediante terminal, o mostrar la interfaz a través de VirtualBox y trabajar de forma más gráfica. En este caso, se proseguirá por terminal, por lo que nos conectamos con:

```
vagrant ssh
```

A continuación, se obtiene la imagen docker con PostgreSQL en la versión del proyecto, PostgreSQL 11.12. Para realizar la obtención, se utiliza Docker Hub, que es un repositorio de imágenes de todo tipo, bases de datos, servidores web, etc. y por ello será la plataforma que usemos próximamente para obtener también las imágenes de otros componentes como Nginx.

Antes de realizar el "pull" de la imagen, hay que cerciorarse de que es oficial, puesto que una imagen es un conjunto de ejecutables y podría contener algún tipo de malware.

Realizada esta comprobación, se realiza la descarga de la imagen:

```
docker pull postgres:11.12
```

Para comprobar que la imagen se ha descargado, se pueden listar las imágenes disponibles:

```
docker images
```

Para levantar el contenedor de la base de datos, se debe indicar el puerto "-p", un nombre "--name", una contraseña "-e" y la imagen base. El comando por tanto queda de la siguiente manera:

```
docker run -p 5432:5432 --name postgres -e "POSTGRES_PASSWORD=veronica11" postgres:11.12
```

Para comprobar que existe el contenedor, y ver su estado (si está arrancado (up), o no (exited)) se utiliza:

```
docker ps -a
```

```
vagrant@ubuntu-focal:~$ docker ps -a
CONTAINER ID   IMAGE           COMMAND                  CREATED              STATUS                      PORTS     NAMES
62f011df2670   postgres:11.12  "docker-entrypoint.s…"   About a minute ago   Exited (0) 8 seconds ago              postgres
```

Como puede verse, el contenedor no está arrancado. Para arrancarlo:

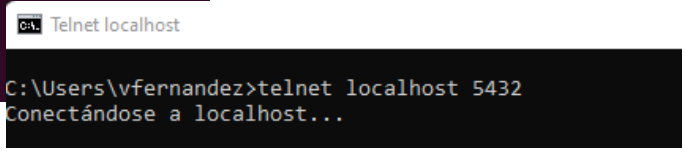
```
docker start <nombreContenedor>
```

```
vagrant@ubuntu-focal:~$ docker start postgres
postgres
vagrant@ubuntu-focal:~$ docker ps -a
CONTAINER ID   IMAGE           COMMAND                  CREATED          STATUS          PORTS                                       NAMES
62f011df2670   postgres:11.12  "docker-entrypoint.s…"   10 minutes ago   Up 5 minutes    0.0.0.0:5432->5432/tcp, :::5432->5432/tcp   postgres
```

Ahora el estado del contenedor es "Up" (arrancado), y además se muestra en qué puertos se ha levantado, en este caso en el 5432. Para volver a parar el contenedor, se ejecuta el mismo comando anterior sustituyendo "start" por "stop".

Por último, se comprueba que el contenedor está operativo realizando una conexión telnet con el contenedor a través del puerto 5432 tanto local (desde la máquina virtual) como remota (desde el equipo local Windows).

```
vagrant@ubuntu-focal:~$ telnet localhost 5432
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
```



Como se puede ver, el contenedor es accesible desde ambos extremos y está operativo puesto que la conexión que se realiza con éxito en ambos casos.

Aquí finaliza el despliegue del servicio de PostgreSQL, el cual al estar disponible de forma remota se podría probar mediante PgAdmin tras configurar la base de datos. A continuación, procedo a explicar cómo se realiza el despliegue de la infraestructura completa para el proyecto Tek, que al requerir de varios servicios difiere un poco de la configuración explicada previamente.

7. Despliegue del proyecto Tek en infraestructura remota

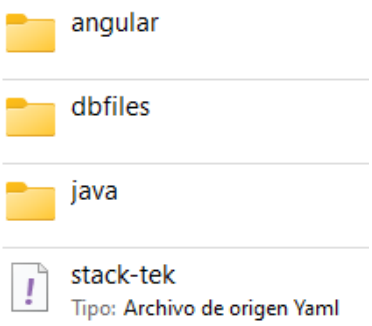
La orquestación completa de la aplicación es más compleja, ya que requiere de varios servicios y diferentes contenedores. Cada contenedor debe arrancar de una imagen base diferente, el back debe arrancar a partir del .jar, el front a partir del empaquetado web, y la base de datos de la imagen PostgreSQL. Además, los datos de la base de datos se deben enviar a un volumen persistente, para que en caso de que se borre un contenedor para por ejemplo escalarlo o reconfigurarlo, los datos no se pierdan.

Para definir toda esta configuración de los diferentes servicios se utiliza un fichero .yaml, el cual posteriormente se ejecutará con Docker-Compose levantando así todos los servicios definidos que son los necesarios para la aplicación. Y además se tienen diferentes Dockerfile para definir las imágenes de back y front.

Como se mencionó al explicar el funcionamiento del proyecto en local, y como se ha vuelto a recalcar, se requerirán al menos tres contenedores Docker y cinco imágenes. Los contenedores serán uno la base de datos, otro el frontend, y otro el backend. Las imágenes serán una la versión PostgreSQL necesaria para el proyecto (11.12), otra el frontend que se genera desde su empaquetado, una imagen Nginx que es el servidor sobre el que se levanta la web (se utiliza Nginx Alpine), otra será el backend que se generará a partir del .jar y por último la imagen con el Java apropiado para el proyecto (Java 8).

A continuación, se explican los diferentes componentes que conforman por tanto el empaquetado completo del proyecto para desplegarlo mediante Docker y conseguir montar toda su infraestructura en una máquina remota, pero haciendo que la aplicación esté disponible desde equipos externos.

El empaquetado contiene 4 elementos:



- angular: carpeta que contiene el empaquetado web necesario para generar la imagen del frontend, el fichero Dockerfile de configuración de la misma, y el archivo de configuración del servidor web Nginx.
- dbfiles: carpeta que contiene el script de creación y configuración de la base de datos PostgreSQL. Posteriormente se explicará cómo obtener los datos SQL, existiendo dos alternativas y suponiendo una de ellas una alteración sobre el contenido de esta carpeta.
- java: carpeta que contiene el .jar para levantar la imagen backend, y el fichero Dockerfile de configuración de esta.
- stack-tek.yaml: fichero de configuración de todos los servicios/contenedores que conforman la aplicación y que se levantarán mediante Docker-Compose.

Ahora se analiza cada elemento por separado:

1. Carpeta angular:
    A. tek-web: se corresponde con el empaquetado generado localmente tras ejecutar “ng build --prod --base-href=”/tek-web/””, aspecto explicado previamente.

    B. Dockerfile: configuración de la imagen front, en la que se indica que se utilizará el servidor web Nginx Alpine, que se debe copiar el empaquetado

de la web dentro de la carpeta de configuración del servidor llamada "html" localizada en "/usr/share/nginx/html", vaciando por si acaso previamente dicha carpeta pues solo puede contener los ficheros necesarios para una web, también se copia el fichero de configuración del servidor, y por último se otorgan permisos al usuario nginx para asegurar que podrá ejecutar la carpeta "html" y por tanto desplegar la web exponiéndola en el puerto 80.

Su contenido se muestra a continuación:

```
FROM nginx:alpine
# use a volume is more efficient and speed the filesystem
VOLUME /tmp
RUN rm -rf /usr/share/nginx/html/*
COPY nginx.conf /etc/nginx/nginx.conf
COPY tek-web /usr/share/nginx/html/tek-web
RUN chown -R nginx:nginx /usr/share/nginx/*
#expose app in 80
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

C. nginx.conf: fichero de configuración del servidor web, en el que se indica que se escuchará por el puerto 80, que se deben incluir ficheros de estilo para la web (que lo indica el "include mime.types"), y dónde se encuentran los ficheros del empaquetado web, que es en la carpeta "html".

Su contenido se muestra a continuación:

```
worker_processes  1;

events {
    worker_connections  1024;
}

http {
    server {
        listen 80;
        include /etc/nginx/mime.types;
        location / {
            root /usr/share/nginx/html;
            index index.html index.htm;
        }
    }
}
```

2. Carpeta dbfiles:
Como se ha mencionado previamente existen dos opciones para cargar los datos SQL en la base de datos, y según la opción elegida el contenido de esta carpeta varía. En la opción A se opta por cargarlos utilizando Flyway desde SpringBoot, la opción B muestra como quedaría la carpeta en caso de optar por cargar los datos a través de Docker.

Cabe destacar que en ambos casos los datos de la base de datos se almacenan en un volumen persistente para que en caso de que se borre el contenedor por algún motivo los datos se mantengan, y que además la ejecución de la creación de la base de datos, esquema e inserción de los datos en sí solo se realiza en la

primera ejecución, en el resto se comprueba si los datos existen ya y lo único que se hace es actualizar las tablas si se ha añadido algo nuevo.

Destacar también que no hay una opción mejor que otra, se debe elegir en función de quien se quiere que tenga la responsabilidad sobre los datos, si desarrollo, para lo que se elegiría la opción A, u operaciones, para lo que se elegiría la opción B. Y podría haber incluso una opción C, que sería añadir Flyway como otro contenedor de Docker, pero no se ha probado puesto que también dejaría la responsabilidad de los datos a operaciones, y en este caso se quiere dejar la responsabilidad a los desarrolladores.

Para variar entre la opción A y B solo se debe modificar el JAR del backend, puesto que para delegar el Docker se debe deshabilitar Flyway.

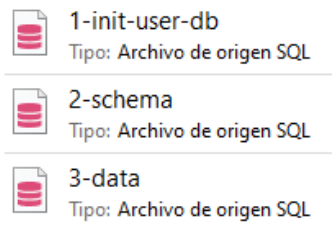
A. init-user-db.sh: script de creación de base de datos y configuración de un usuario para la misma. Se define un ámbito de configuración SQL donde lo primero se crea un usuario para la base de datos llamado "tek-api" con contraseña "tek", a continuación se le dan permisos de superusuario (necesarios para generar el esquema de la base de datos), se crea la base de datos "tek" y por último se dan todos los privilegios al usuario "tek-api" dentro de la base de datos "tek".

   El contenido del script por tanto es el siguiente:

```
#!/bin/bash
set -e

psql -v ON_ERROR_STOP=1 --username "$POSTGRES_USER" --dbname "$POSTGRES_DB" <<-EOSQL
    CREATE USER "tek-api" WITH ENCRYPTED PASSWORD 'tek';
    ALTER USER "tek-api" WITH SUPERUSER;
    CREATE DATABASE tek;
    GRANT ALL PRIVILEGES ON DATABASE "tek" TO "tek-api";
EOSQL
```

B. Tres ficheros SQL: para este caso se generan tres ficheros SQL, el primero se encarga de la inicialización de la base de datos con la creación de un usuario, el segundo genera el esquema de la base de datos, es decir todas las tablas, y el tercero realiza los "insert" de los datos en sus respectivas tablas.

   Los ficheros se nombran como se muestra a continuación ya que en esta alternativa los ficheros se ejecutarán en orden alfabético, y por ello es importante establecer este orden de ejecución.



3. Carpeta angular:
   A. Dockerfile: configuración de la imagen back, en la que se indica que usará una versión Java 8, se genera un usuario llamado "javams" y su grupo "devopsc" que es el usuario con el que se revisan las variables de entorno, se selecciona el .jar ejecutable de la aplicación y se ejecuta dejando expuesto el back en el puerto 8080.

El contenido por tanto es el mostrado a continuación:

```
FROM openjdk:8-jdk-alpine
RUN addgroup -S devopsc && adduser -S javams -G devopsc
USER javams:devopsc
ENV JAVA_OPTS=""
ARG JAR_FILE
ADD ${JAR_FILE} app.jar
 # use a volume is more efficient and speed the filesystem
VOLUME /tmp
EXPOSE 8080
ENTRYPOINT [ "sh", "-c", "java $JAVA_OPTS -Djava.security.egd=file:/dev/./urandom -jar /app.jar" ]
```

B. JAR ejecutable: .jar generado localmente con el proceso explicado anteriormente.

4. Archivo de configuración stack-tek.yaml:
En este fichero se definen todos los servicios, y será el utilizado por Docker-Compose para realizar la orquestación y administración de los mismos.

El primer aspecto importantes es que la versión indicada de Docker-Compose (primera línea del yaml) sea compatible con la versión de Docker-Engine que se utiliza.

Comprobado esto, se pasa a la definición de los servicios. Inicialmente se definen cuatro servicios:

1) Servicio de la base de datos PostgreSQL – postgres_db: El contenedor de este servicio se llamará postgres, y arranca de la imagen de PostgreSQL necesaria para el proyecto (11.12) que se encuentra en Docker Hub. Se levantará en los puertos 5432, y se generan los volúmenes para almacenar los datos SQL de forma persistente. Por último, se establece un usuario y contraseña para la base de datos con los que se podrá acceder a ella desde el administrador, Adminer.

```
version: '3.1'

services:
#database engine service
  postgres_db:
    container_name: postgres
    image: postgres:11.12
    restart: always
    environment:
    ports:
      - 5432:5432
    volumes:
       #allow *.sql, *.sql.gz, or *.sh and is executed only if data directory is empty
      - ./dbfiles:/docker-entrypoint-initdb.d
      - /var/lib/postgres_data:/var/lib/postgresql/data
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: veronica11
      POSTGRES_DB: postgres
```

2) Servicio de un administrador para la base de datos Adminer – adminer: El contenedor de este servicio se llamará adminer, arranca de la imagen adminer de Docker Hub. Como es un administrador de la base de datos depende de que dicho servicio esté operativo, y se expone en el puerto local (del contenedor) 8080, y el puerto externo (de la máquina virtual) 9090. Este servicio se añade por tener un extra de funcionalidad de manera que los

usuarios de esta aplicación puedan tener acceso al motor de base de datos de forma más gráfica mediante esta interfaz.

```yaml
#database admin service
  adminer:
    container_name: adminer
    image: adminer
    restart: always
    depends_on:
      - postgres_db
    ports:
      - 9090:8080
```

3) Servicio backend – tek-api-back: El contenedor de este servicio se llamará tekApi-back, y como arranca desde el .jar incluido en la carpeta java lo indico con los parámetros seguidos de "build". Al estar la carpeta en el mismo directorio, el path hasta el .jar queda como "./java", si estuviera localizado en otro lugar se debe indicar la ruta completa. Se definen algunas variables de entorno para establecer que se reconozca el contenedor de la base de datos, ya que al estar en otro contenedor es como si estuviera en un servidor externo, es decir, no vale con referirse a él como "localhost:5432" hay que llamarlo con "<nombreServicio>:puerto" en este caso por tanto "postgres_db:5432". También se define que Java tenga una capacidad 512MB. Al igual que ocurre con adminer, depende de que el servicio de la base de datos esté operativo, y así se indica. Por último, este contenedor se expone en el puerto local 8080 (del contenedor) y puerto externo 8081 (de la máquina virtual).

```yaml
#tek app backend service
  tek-api-back:
    build:
      context: ./java
      args:
        - JAR_FILE=*.jar
    container_name: tekApi-back
    environment:
      - "TEK_API_IP_EXTERNAL=localhost"
      - "DB_URL=jdbc:postgresql://postgres_db:5432/tek"
      - JAVA_OPTS=
        -Xms512M
        -Xmx512M
    depends_on:
      - postgres_db
    ports:
      - 8081:8080
```

4) Servicio frontend – tek-api-front: El contenedor de este servicio se llamará tekApi-front, y como arranca desde el empaquetado web incluido en la carpeta del mismo directorio "./angular" se indica de esta manera el path. El front depende del back, por lo que se establece una dependencia con el servicio del backend. Por último, se expone en el puerto local 80 (del contenedor) y en el puerto externo 8080 (de la máquina virtual).

```yaml
#tek app frontend service
  tek-api-front:
    build:
      context: ./angular
    container_name: tekApi-front
    depends_on:
      - tek-api-back
    ports:
      - 8080:80
```

Explicado el empaquetado necesario para realizar el despliegue en remoto, se debe configurar la máquina de manera que se expongan los puertos necesarios para acceder a los cuatro servicios que se levantarán.

Se utilizará la máquina generada al inicio de este trabajo, pero modificando su Vagrantfile para exponer un puerto para cada servicio. Para realizar el enrutamiento de puertos se utiliza como puerto local el puerto en el que se exponen los servicios en la máquina virtual, es decir, el 5432 para PostgreSQL, 9090 para Adminer, 8081 para backend y 8080 para frontend. Por comodidad, se establecen como puertos externos los mismos, de manera que para acceder desde el equipo local Windows a los servicios nos conectamos de la misma manera, a través de los puertos 5432, 9090, 8081 y 8080.

El fichero Vagrantfile queda por tanto de la siguiente manera (estando el resto de líneas por defecto, comentadas):

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

# All Vagrant configuration is done below. The "2" in Vagrant.configure
# configures the configuration version (we support older styles for
# backwards compatibility). Please don't change it unless you know what
# you're doing.
Vagrant.configure("2") do |config|
  # The most common configuration options are documented and commented below.
  # For a complete reference, please see the online documentation at
  # https://docs.vagrantup.com.

  # Every Vagrant development environment requires a box. You can search for
  # boxes at https://vagrantcloud.com/search.
  config.vm.box = "veronicafg/devops-con-interfaz-esp"

  # Disable automatic box update checking. If you disable this, then
  # boxes will only be checked for updates when the user runs
  # `vagrant box outdated`. This is not recommended.
  # config.vm.box_check_update = false

  # Create a forwarded port mapping which allows access to a specific port
  # within the machine from a port on the host machine. In the example below,
  # accessing "localhost:8080" will access port 80 on the guest machine.
  # NOTE: This will enable public access to the opened port
  # config.vm.network "forwarded_port", guest: 80, host: 8080
  config.vm.network "forwarded_port", guest: 5432, host: 5432
  config.vm.network "forwarded_port", guest: 8080, host: 8080
  config.vm.network "forwarded_port", guest: 8081, host: 8081
  config.vm.network "forwarded_port", guest: 9090, host: 9090
```

Configurada la máquina, pasamos a ejecutar los comandos necesarios para hacer el despliegue de los servicios y por tanto la orquestación completa de toda la aplicación con imágenes independientes y en una máquina remota.

Como se ha indicado en varias ocasiones, para la orquestación se utiliza Docker-Compose llamando a su fichero de configuración “stack-tek.yaml”. Por tanto, estando situados en el directorio que contiene todo el contenido de la orquestación, se ejecutan los siguientes comandos:

En primer lugar, se construyen todas las imágenes de los servicios:

```
docker-compose -f stack-tek.yml build
```

En segundo lugar, se inicializan los contenedores de los servicios:

```
docker-compose -f stack-tek.yml up -d
```

El arranque se realiza en segundo plano, pero siempre se pueden revisar los logs de los contenedores bien de forma estática o bien de forma dinámica (con opción -f).

```
docker logs (-f) <nombreContenedor>
```

Finalizados estos comandos, se puede comprobar que todas las imágenes necesarias se han descargado haciendo un listado de las imágenes disponibles en el equipo, y también se pueden ver los contenedores y su estado de la misma manera.

```
docker images
docker ps -a
```

Para arrancar y parar todos los contendores se ejecuta lo siguiente:

```
docker-compose -f stack-tek.yml start
docker-compose -f stack-tek.yml stop
```

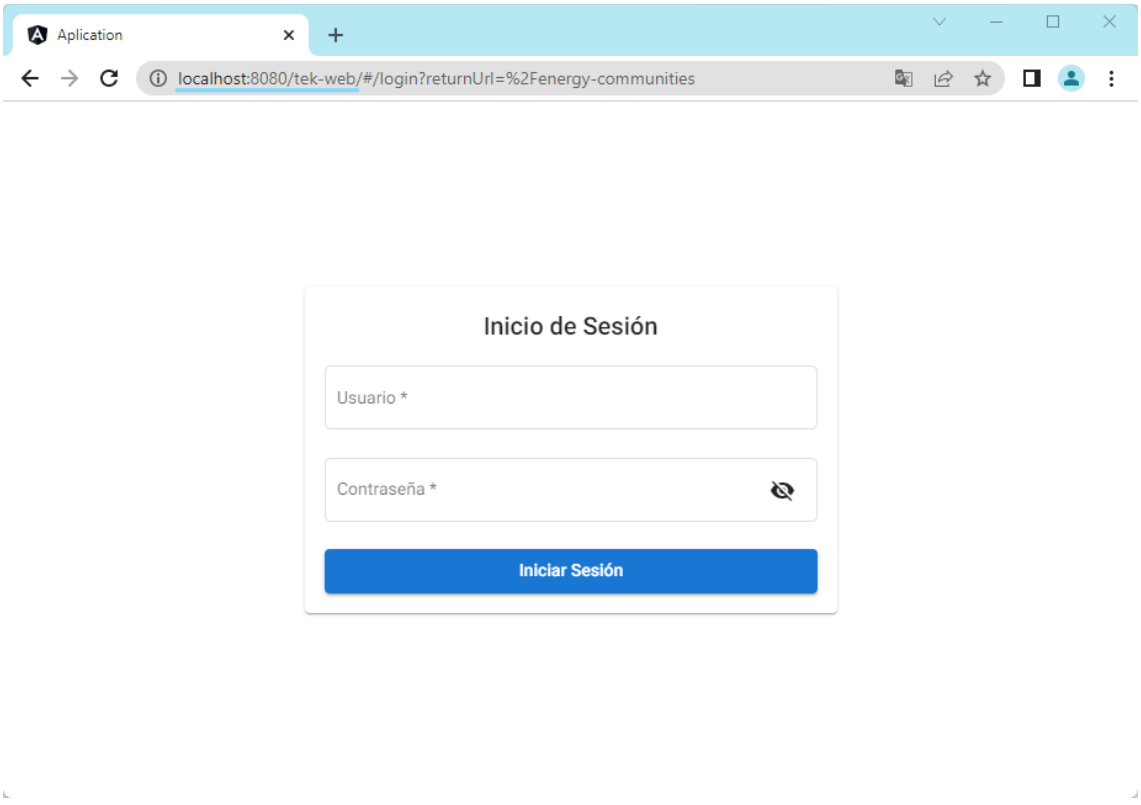
En caso de querer eliminar los contenedores además de pararlos entonces se ejecuta:

```
docker-compose -f stack-tek.yml stop
```

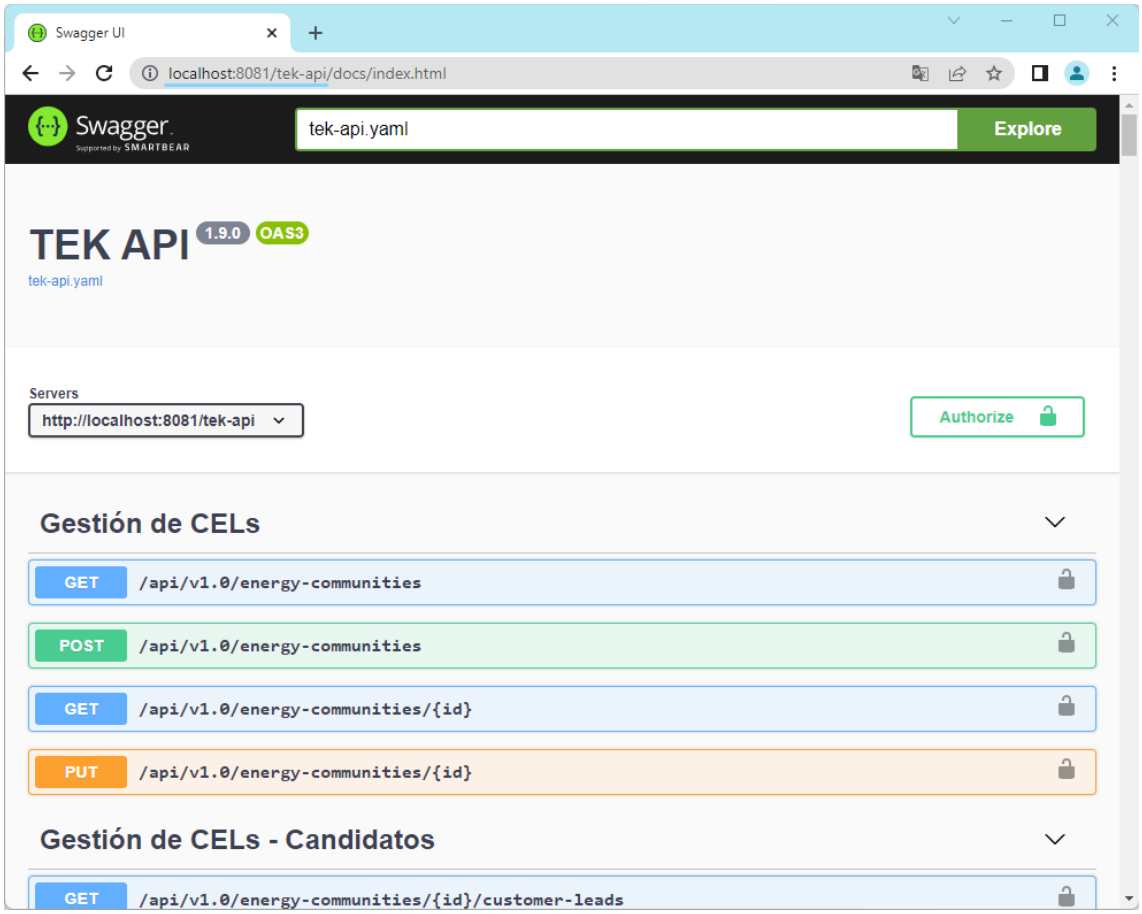
Tras este comando, en el siguiente “up” se recrearán los contenedores.

Por último, para probar que todo funciona mediante un navegador se accede a las URLs correspondientes (indicando localhost (pues en la máquina local están todos en el mismo server), puerto correspondiente para cada servicio y dominio).

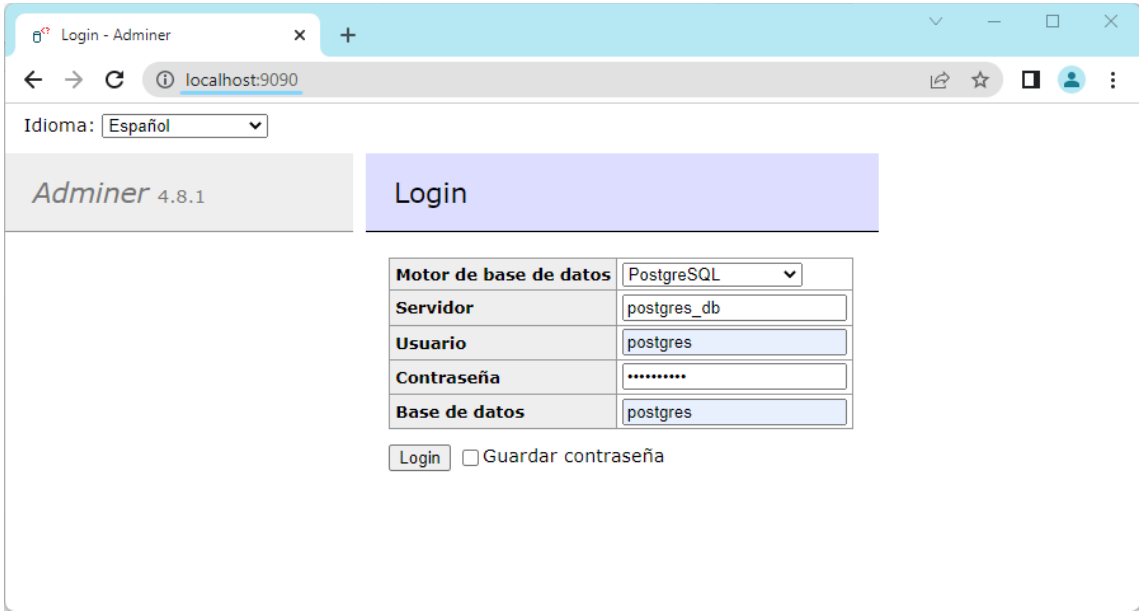
Así navegando a “localhost:8080/tek-web” se puede ver la web, que por defecto redirige a la página de log-in.



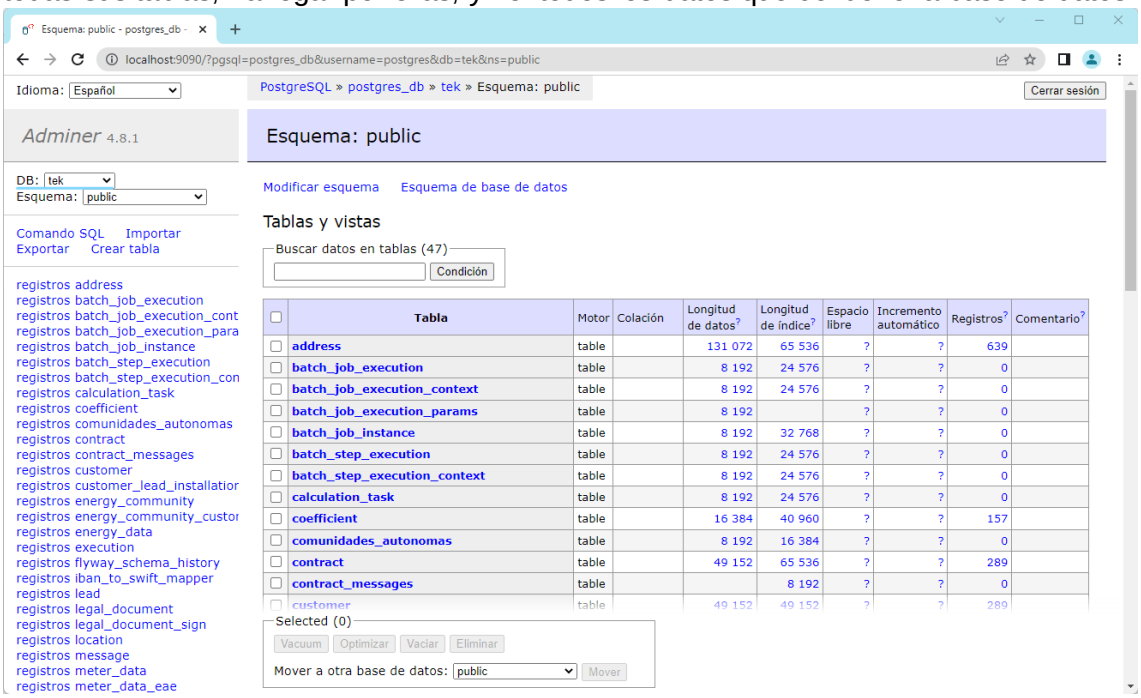
Navegando a “localhost:8081/tek-api” se puede ver el back, que se muestra a través de Swagger.



Y por último navegando a “localhost:9090” se accede al Adminer, con el que si introducimos las credenciales establecidas al momento de crear el servicio “postgres_db” se puede acceder a la base de datos.



Una vez dentro, solo hay que seleccionar nuestra base de datos “tek” y ya podemos ver todas sus tablas, navegar por ellas, y ver todos los datos que contiene la base de datos.



Alternativamente también se puede comprobar si todo está en orden entrando dentro de cada contenedor. Por ejemplo, si quisiéramos comprobar si todos los datos se han cargado correctamente, entramos en el contenedor postgres ejecutando el siguiente comando:

```
docker exec -it postgres /bin/bash
```

Tras esto, entramos al usuario postgres (que es el que definimos como usuario para poder acceder la base de datos al configurar el servicio postgres_db) y nos conectamos a la Shell de PostgreSQL:

```
root@53084ffc1dca:/# su - postgres
postgres@53084ffc1dca:~$ psql
psql (11.12 (Debian 11.12-1.pgdg90+1))
Type "help" for help.
```

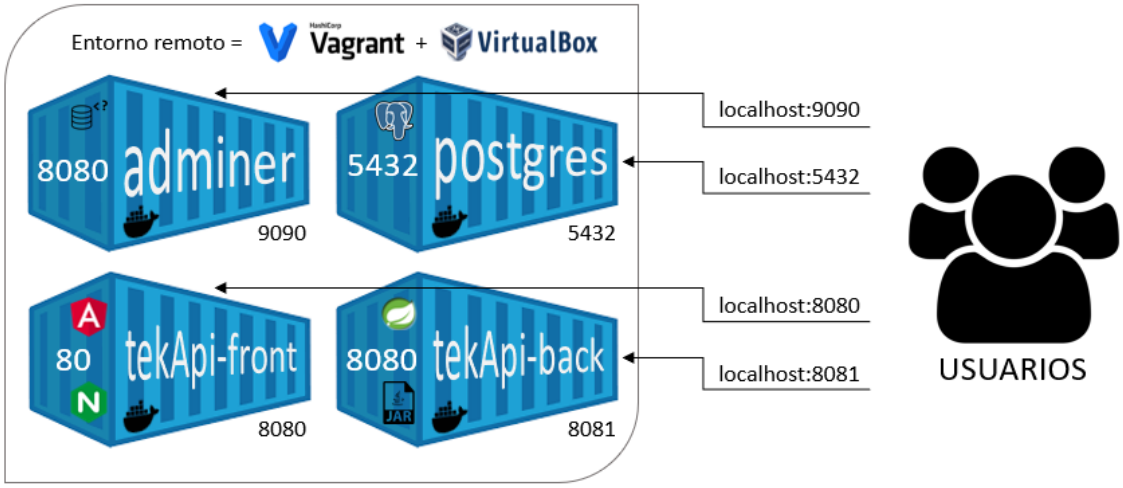
Una vez dentro de la Shell, nos conectamos a la base de datos “tek” y con un sencillo comando podemos listar todas sus tablas:

```
postgres=# \c tek
You are now connected to database "tek" as user "postgres".
tek=# \dt
                    List of relations
 Schema |                  Name                  | Type  |  Owner
--------+----------------------------------------+-------+---------
 public | address                                | table | tek-api
 public | calculation_task                       | table | tek-api
 public | coefficient                            | table | tek-api
 public | comunidades_autonomas                  | table | tek-api
 public | contract                               | table | tek-api
 public | contract_messages                      | table | tek-api
 public | customer                               | table | tek-api
 public | customer_lead_installations            | table | tek-api
 public | energy_community                       | table | tek-api
 public | energy_community_customer_lead_address | table | tek-api
```

Y finalmente, para comprobar el contenido de una tabla ejecutamos directivas SQL y se nos muestra el contenido:

```
tek=# select * from municipios;
  id   | cod_ine |                  descripcion                  | provincia_id
-------+---------+-----------------------------------------------+--------------
  1001 | 01001   | ALEGRÍA-DULANTZI                              |            1
  1002 | 01002   | AMURRIO                                       |            1
  1003 | 01003   | ARAMAIO                                       |            1
  1004 | 01004   | ARTZINIEGA                                    |            1
  1006 | 01006   | ARMIÑÓN                                       |            1
  1008 | 01008   | ARRATZUA-UBARRUNDIA                           |            1
  1009 | 01009   | ASPARRENA                                     |            1
  1010 | 01010   | AYALA/AIARA                                   |            1
  1011 | 01011   | BAÑOS DE EBRO/MAÑUETA                         |            1
```

Por tanto, el despliegue remoto de Tek queda de la siguiente manera:



Dentro de la máquina virtual se despliegan los tres contenedores que conforman la aplicación Tek (frontend, backend y base de datos) más un contenedor para la interfaz de acceso a la base de datos (Adminer), cada uno con sus componentes, y se permite su acceso desde el exterior mediante la exposición de puertos.

8. Proxy Inverso

8.1 Comunicación entre contenedores

Cada contenedor funciona de manera independiente. Esto quiere decir que tienen sus propios puertos, conexiones, localhost, etc.

Para la aplicación Tek, debido a que el API se divide en dos partes, backend y frontend, diseñados en lenguajes diferentes y cada cual con una funcionalidad completamente distinta, se levantan en dos contenedores separados, uno con el empaquetado Java y otro con el empaquetado web en Angular.

Sin embargo, al pertenecer ambos a un mismo componente, la API, aparece la necesidad de comunicar ambos contenedores por ejemplo para realizar las tareas de autenticación de usuarios que acceden a través de la interfaz web, pero cuyas credenciales se comprueban mediante funcionalidad Java.

Esto aparentemente es un problema, pues al desplegarse cada parte en un contenedor diferente no se pueden comunicar, y se genera un escenario no contemplado previamente ya que en los despliegues locales todo está sobre un mismo servidor y la comunicación se lleva a cabo sin problema. Este problema recibe el nombre de CORS (intercambio de recursos de origen cruzado). Sin embargo, existen una serie de medidas que podemos adoptar, como establecer una red interna entre contenedores o utilizar un proxy inverso, que solucionan este aspecto. Se opta por implementar este último.

Un proxy inverso es una herramienta que permite la publicación de servicios bajo un mismo dominio de red de manera que a través de ese dominio se puedan comunicar diferentes componentes de una infraestructura, como contenedores. Para implementar el proxy inverso se utiliza Traefik.

8.2 Implementación Traefik

Lo primero, se genera un nombre de dominio llamado "tfg.local", que se declara tanto para probar la comunicación interna en la máquina virtual entre contenedores (declarando el nombre de dominio en "/etc/hosts"), como para permitir la comunicación en conexiones remotas entre contenedores (declarando el nombre de dominio en este caso en la máquina local Windows en "System32/drivers/etc/hosts").

En la siguiente imagen se muestra la línea que hay que añadir para Windows, y en Linux sería lo mismo.

```
# localhost name resolution is handled within DNS itself.
#   127.0.0.1       localhost
#   ::1             localhost
127.0.0.1       tfg.local
```

En segundo lugar, se configuran los contenedores añadiendo a aquellos que se quieren comunicar (los contenedores del backend y frontend en este caso) las siguientes líneas que configuran Traefik:

```
#tek app backend service
  tek-api-back:
    build:
      ...
    labels:
      - "traefik.enable=true"
      - "traefik.http.routers.tek-api-back.rule=Host(`tfg.local`) && PathPrefix(`/tek-api`)"
      - "traefik.http.routers.tek-api-back.entrypoints=web"
#tek app frontend service
  tek-api-front:
    build:
      ...
    labels:
      - "traefik.enable=true"
      - "traefik.http.routers.tek-api-front.rule=Host(`tfg.local`) && PathPrefix(`/tek-web`)"
      - "traefik.http.routers.tek-api-front.entrypoints=web"
```

Por último, se declara un contenedor para el propio proxy inverso que recibe el nombre de "traefik" y se configura con las líneas mostradas a continuación:

```
services:
#proxy inverso
  traefik:
    image: "traefik:v2.6"
    container_name: traefik
    command:
      - "--log.level=DEBUG"
      - "--api.insecure=true"
      - "--providers.docker=true"
      - "--providers.docker.exposedbydefault=false"
      - "--entrypoints.web.address=:80"
    ports:
      - "80:80"
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock:ro
```

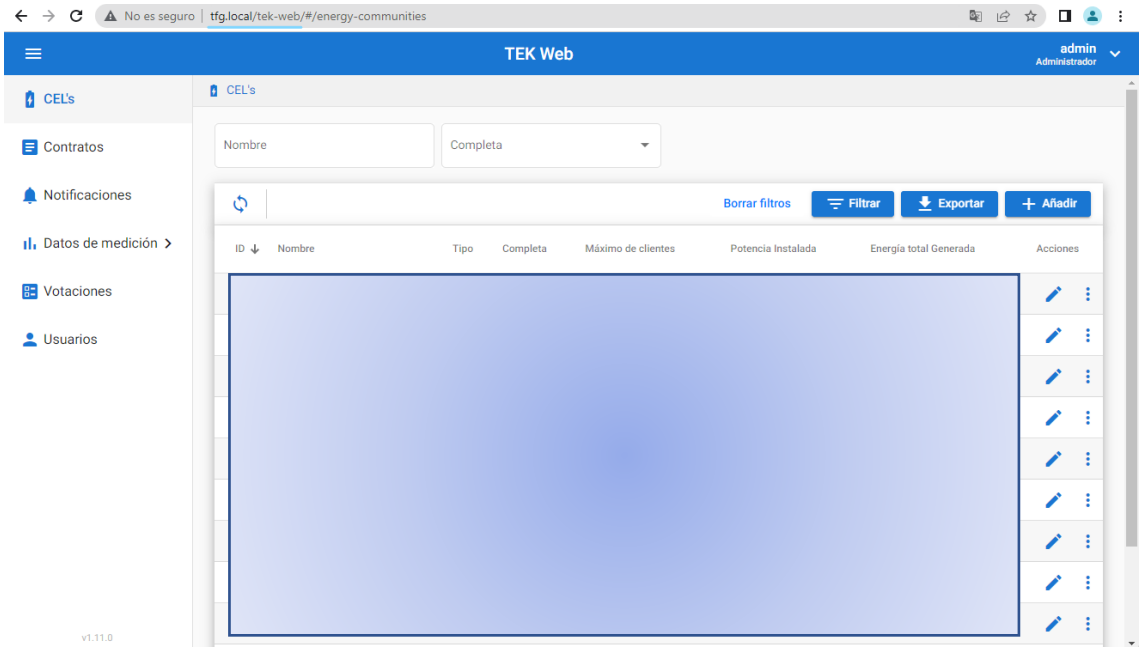
Con estas líneas se define que la imagen a descargar de Docker Hub para Traefik sea la versión 2.6, el nombre del contenedor será “traefik”, y se establece que en su arranque se pueda analizar a nivel debug, y otros parámetros de los que el más importante es que se expondrá en el puerto 80, tanto local como externamente.

Además, en este caso para que la parte web del api utilice el nombre de dominio generado para llamar y poder comunicarse con la parte Java se modifica también la configuración de la misma editando por tanto el “assets/config.json” realizando la siguiente modificación:

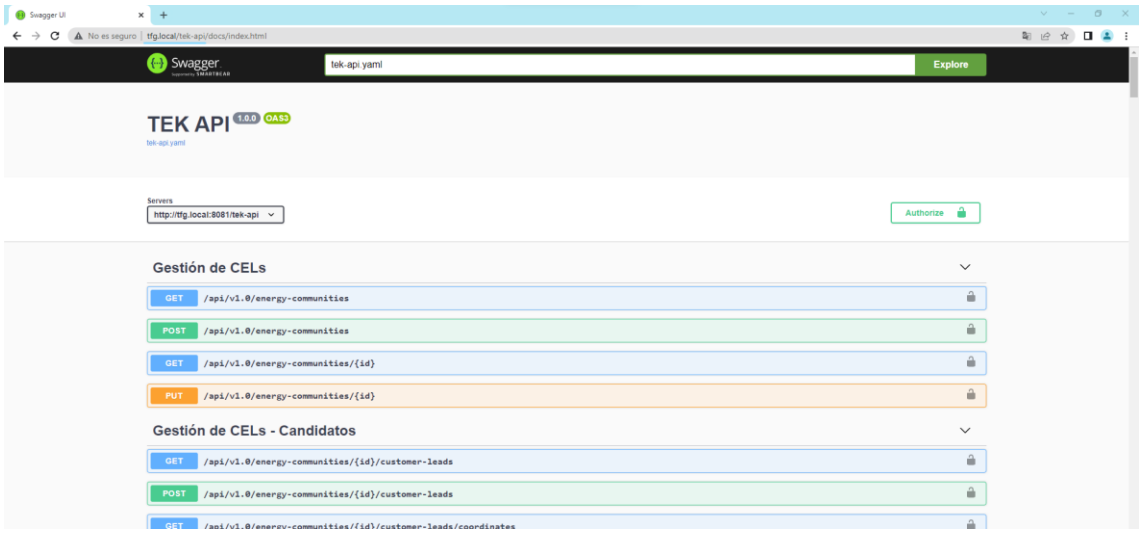
```
{
  "serviceBaseUrl": "http://tfg.local/tek-api/api/v1.0/",
  "serviceAuthUrl": "http://tfg.local/tek-api/",
  "cliId": "guest",
  "cliSec": "guest",
  "maxElementsToExport": 1000,
  "baseHref": "/tek-web/"
}
```

Finalizada la configuración, se accede tanto a la interfaz web como al Swagger del API mediante "tfg.local/<URL>", en lugar de utilizando "localhost/<URL>", y se permite hacer el login en la web.

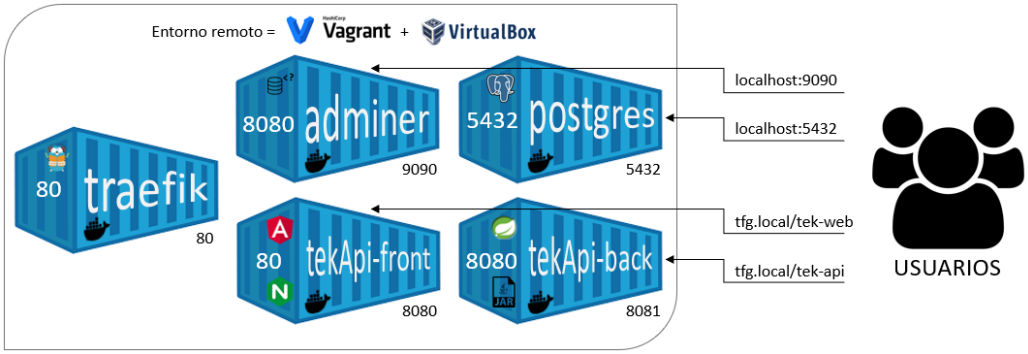
A continuación, se muestra la interfaz web dentro del usuario administrador junto con el path mediante el que se consigue acceso a través del navegador.



Y también se puede mostrar el Swagger:



Se puede comprobar por tanto como en efecto esta medida soluciona el problema de conexión de manera que el back y front se pueden comunicar y se puede realizar la autenticación. Y, por lo tanto, el esquema de acceso queda de la siguiente manera:



9. Testing

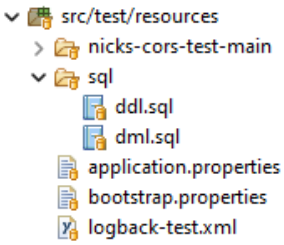
9.1 Desarrollo de los tests

Para la parte de testing se desarrollan tests de integración que ponen a prueba la seguridad del proyecto, basada en OAuth2. Esto se hace con SpringBoot, por lo que lo primero se incluyen una serie de dependencias en el "pom.xml" que permiten añadir de anotaciones para test y ejecutarlos. Gracias a estas anotaciones se puede saber que se trata de tests de integración ya que se levanta un entorno web simulado para ejecutarlos.

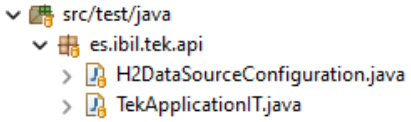
```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.DEFINED_PORT, classes =
{ TekApplicationIT.TekTestConfig.class })
@TestPropertySource("/bootstrap.properties")
@EnableAutoConfiguration
```

La definición de los test se hace en dos carpetas dentro de "tek-api-spring-boot": "src/test/java" que contiene el código de los test y "src/test/resources" que contiene aspectos más de configuración, de propiedades de los test.

De las propiedades de los test ("src/test/resources") lo más importante es que se define que para los tests se utilice la base de datos en memoria H2, en lugar de la base de datos PostgreSQL, y dentro de "/sql" se encuentran unos de scripts SQL con los que se realiza la configuración de la base de datos H2, y se le añaden una serie de contenidos para probar los test.



Por otro lado, se tiene la definición de los test ("src/test/java"). Dentro hay dos ficheros. El primero se corresponde con un método que ejecuta la configuración de la base de datos H2, para lo que llama a los scripts SQL mencionados previamente.



El segundo define la funcionalidad de los tests a ejecutar, que se explica a continuación:

- Lo primero se carga la seguridad del proyecto que se quiere probar (OAuth2), para lo que se utiliza la notación "@Before".
- Lo segundo se definen los test:
  - El primer test trata de validar un usuario sin proporcionar su contraseña, por tanto debe considerarse un usuario no autorizado.

```
@Test
public void givenNoToken_whenGetSecureRequest_thenUnauthorized() throws Exception
{

mockMvc.perform(get(GET_USERS)).andExpect(status().isUnauthorized());
}
```

  - El segundo test trata de validar un usuario teniendo sus credenciales completas, por tanto el usuario debe validarse correctamente.

```
@Test
public void givenToken_whenGetSecureRequest_thenOk() throws Exception
{

final String accessToken = obtainAccessToken(USERNAME, PASSWORD);
mockMvc.perform(get(GET_USERS).header("Authorization", "Bearer " + accessToken)).andExpect(status().isOk());
}
```

Los datos de validación de los usuarios se declaran en variables e incluyen en la base de datos H2. El resultado de los test se proporciona con códigos HTTP.

9.2 Comprobación local

Para ejecutar los tests de forma local sencillamente nos apoyamos en JUnit situándonos sobre el proyecto, dando botón derecho y "Run As → JUnit Test", y podemos ver desde la interfaz de Eclipse si la salida es correcta, y cuánto tiempo se tardan en ejecutar.

En efecto, la salida es correcta y se mostrará posteriormente ejecutando los test de forma automática desde Jenkins.

El aspecto más importante es que la ejecución de los tests debe ser rápida, puesto que condicionan el tiempo que se tarda en generar el empaquetado ya que se deben ejecutar con éxito antes de realizar el compilado.

10. Integración Continua (IC)

En primer lugar, se debe comprender qué se entiende por integración continua (IC) y su importancia como técnica DevOps. [19]

La integración continua es una práctica de desarrollo software mediante la que se automatiza la integración de código desarrollado por múltiples contribuidores a un repositorio central, es decir, automatiza la gestión de cambios del código que tiene múltiples desarrolladores pero pertenece a un mismo proyecto software. Tras esto se ejecutan una serie de pruebas automáticas, y finalmente si todo va correctamente se realiza la compilación del proyecto.

En resumen, se entiendo por integración continua la ejecución de pruebas y la compilación de un proyecto tras estar actualizado con su repositorio remoto.

Se corresponde con una de las prácticas más relevantes en DevOps ya que automatizando estas tareas se pueden encontrar y arreglar errores con más rapidez, se mejora la calidad del software y se consigue reducir el tiempo de entrega de código validado y actualizado.

El componente más importante es el repositorio de código, gestionado por una estrategia bien trunk-based o gitflow. Sobre este se crean ramas sobre las que trabajan los diferentes desarrolladores, y una vez su trabajo es estable y se valida mediante integración continua se envían los cambios al repositorio central.

Sin IC cada desarrollador tendría que subir su código y avisar a un administrador que valida su código y en caso de ser correcto actualiza con él el repositorio central. Sin embargo, con IC, utilizando un servidor de automatización como Jenkins, cada desarrollador puede enviar su código sin miedo al git, y será servidor IC quien detecte los cambios y lleve a cabo de forma automática la descarga, validación y en su caso la actualización del repositorio central, sin necesidad de avisar a nadie.

10.1 Jenkins

Como se ha mencionado, la herramienta con la que se lleva a cabo la integración continua en este caso es con un servidor de automatización opensource llamado Jenkins. [20][21]

La implementación de Jenkins se hace a través de Docker, generando un contenedor a partir del siguiente Dockerfile.

```
1   FROM jenkinsci/blueocean
2   USER root
3   RUN apk update && apk add wget
4   RUN  wget --no-verbose -O /tmp/apache-maven-3.6.3-bin.tar.gz https://downloads.apache.org/maven/maven-3/3.6.3/binarie
5   RUN tar xzf /tmp/apache-maven-3.6.3-bin.tar.gz -C /opt/
6   RUN ln -s  /opt/apache-maven-3.6.3 /opt/maven
7   RUN ln -s /opt/maven/bin/mvn /usr/local/bin
8   RUN rm /tmp/apache-maven-3.6.3-bin.tar.gz
9   RUN chown jenkins:jenkins /opt/maven;
10
11  ENV MAVEN_HOME=/opt/mvn
12
13  USER jenkins
```

Como se puede ver, se utiliza la imagen de Docker Hub de Jenkins que contiene también Blueocean, que es una interfaz más amigable para observar las tareas y su resultado. Además, para la instalación de Jenkins se requiere Maven, por lo que también se realiza su descarga e instalación.

Definida la imagen, a continuación se muestra cómo se define el contenedor en el Docker-Compose que se utilizó previamente para declarar el resto de los elementos del proyecto:

```
#jenkins service
  jenkins:
    build:
      context: ./jenkins
    container_name: jenkins
    ports:
      - "8090:8080"
    volumes:
      - jenkins-data:/var/jenkins_home
    labels:
      - "traefik.enable=true"
      - "traefik.http.routers.jenkins.rule=Host(`tfg.local`)"
      - "traefik.http.routers.jenkins.entrypoints=web"
      - "traefik.http.routers.jenkins.middlewares=jenkins"
      - "traefik.http.middlewares.jenkins.stripprefix.prefixes=/jenkins"
```

Como se puede observar, para generar el servicio se parte del Dockerfile anterior que se incluye en una carpeta llamada “jenkins”, se expone por el puerto 8080 local y 8090 externo, se utiliza un volumen para guardar toda su configuración de forma persistente, y también se configura Traefik para poder acceder al contenedor desde fuera a través del proxy inverso.

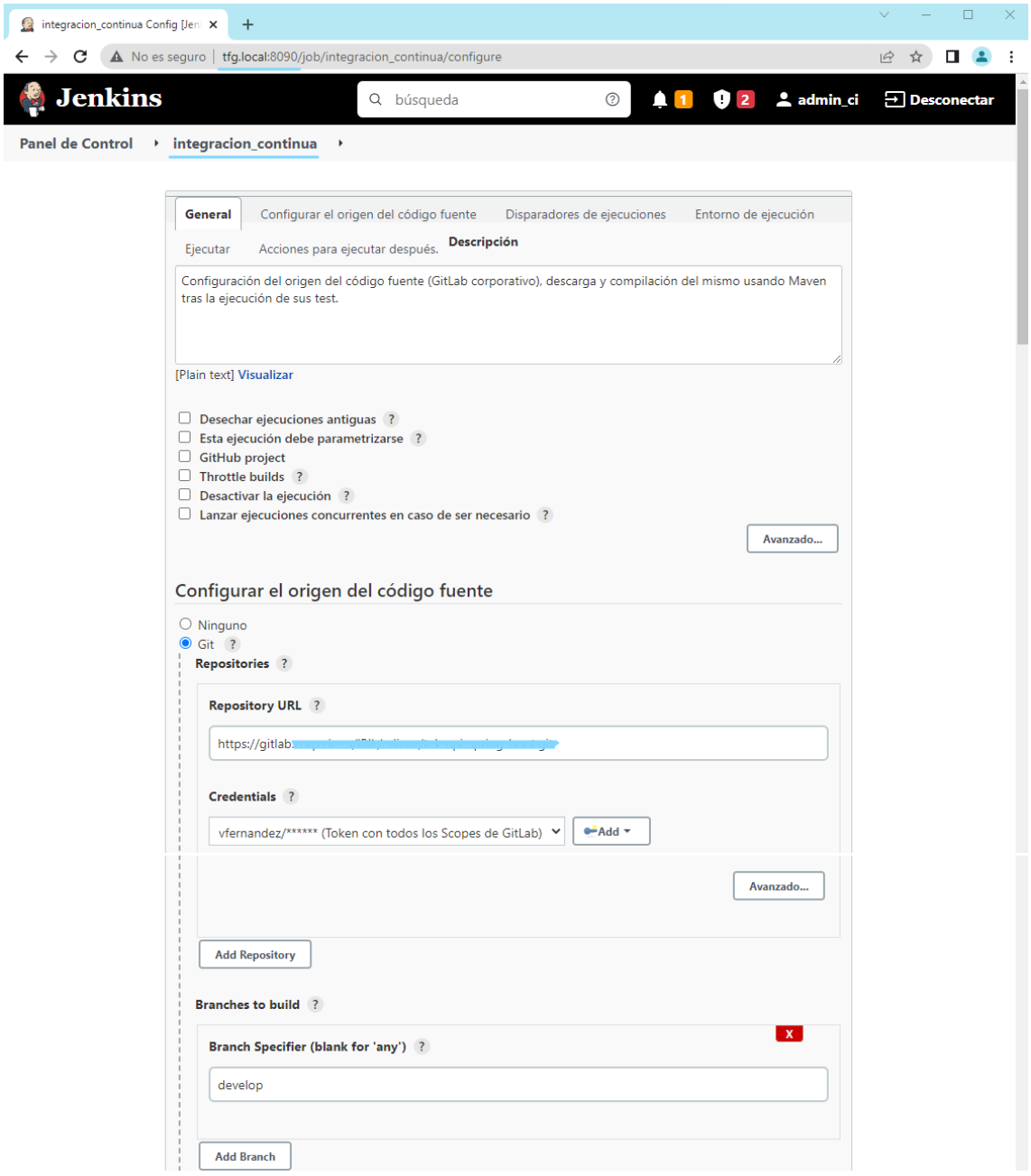
Para configurar Jenkins, una vez se levanta el contenedor por primera vez genera una contraseña que se debe guardar para hacer el primer login en la interfaz, tras este se declaran usuario y contraseña deseados para validarse, y ya se pueden declarar tareas.

10.2 Tareas Jenkins

a) Tarea sencilla, con Maven se compila la API y genera el .jar

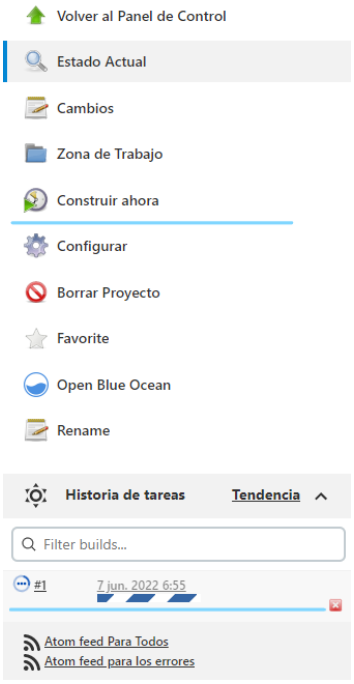
En esta primera tarea se realiza únicamente la compilación del código Java y por tanto se finaliza generando el .jar ejecutable. En su declaración de indica de forma breve una descripción de lo que realiza, que en este caso es la configuración del repositorio de código fuente que se encuentra en el GitLab corporativo, y tras su descarga se procede a la compilación mediante Maven.

Para establecer el repositorio, como podemos ver en la siguiente imagen, se indica que se usará Git, y se indica una URL al repositorio y las claves de acceso. Además, se indica que trabaje sobre la rama “develop” del repositorio.

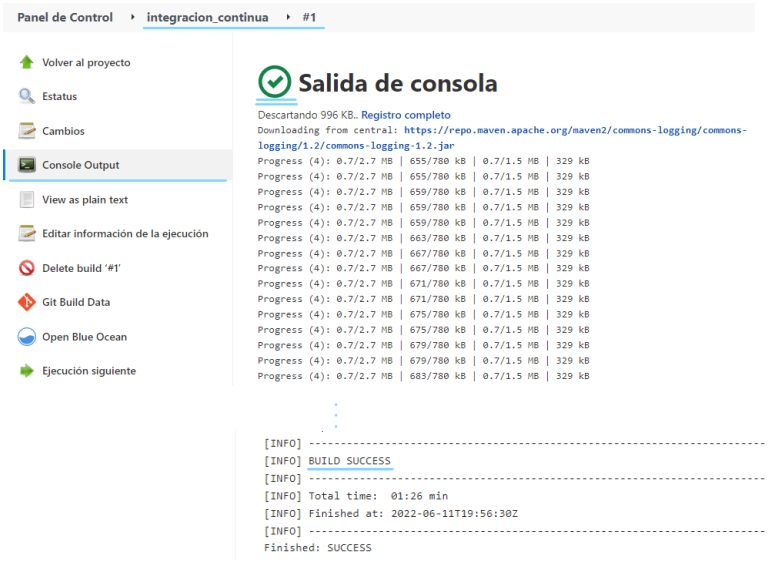


Por último, se indican las directivas Maven que se deben ejecutar para compilar el código y generar el .jar ejecutable. Además, por simplicidad, se indica la ruta al pom.

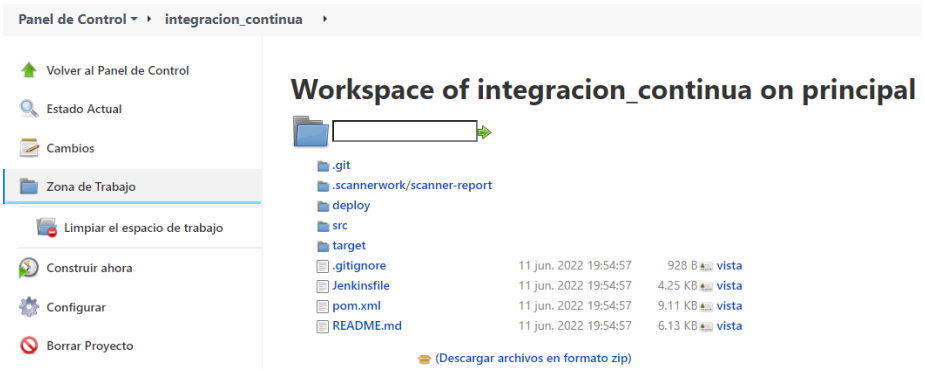
Configurada la tarea, se ejecuta usando la opción “Construir ahora”, y si se clica en la ejecución se puede ver la salida por consola con todos los pasos (directivas git para obtener el código y resto de funciones para generar el .jar ejecutable).



Finalizada la ejecución se ve si ha sido correcta porque aparece un tick verde y “build success”. También se puede comprobar examinando la “Zona de Trabajo”.



Dentro de “Zona de Trabajo” está todo el código que se ha descargado, y la salida de la compilación por tanto está en la carpeta “target”.



Y por último hay otra opción de comprobar que se ha generado el .jar conectándose al contenedor y navegando dentro del volumen hasta su path.

```
docker exec -it jenkins /bin/bash
       ls -l /var/jenkins_home/...
```

b) Tarea que ejecuta los test antes de generar el compilado (.jar)

Se añade a la tarea anterior la ejecución de los test. Para ello se ha configurado el plugin de “surefire” dentro del “pom.xml” del proyecto, que es el encargado de generar un informe de salida informando acerca de los test.

La modificación que hay que hacer en Jenkins es cambiar las directivas Maven que debe incluir la ejecución de los test y generación del informe:

Ahora se puede comprobar en la salida de la ejecución qué ocurre con los test, que en este caso se ejecutan con éxito:

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
```

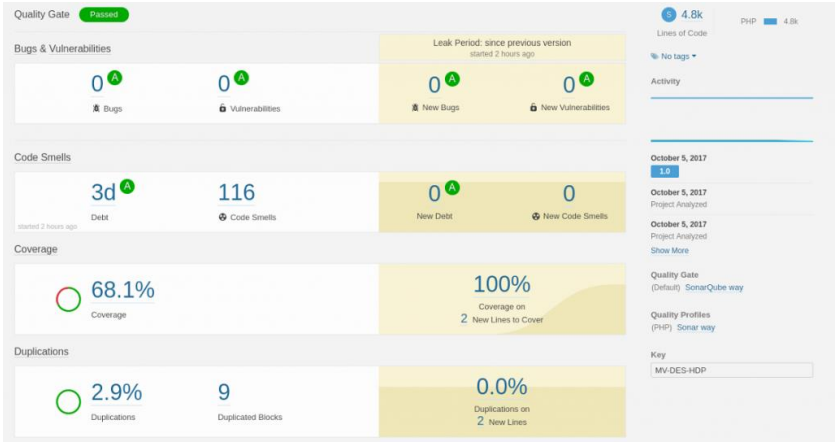
c) Tarea que examina la calidad del código mediante SonarQube, ejecuta los test, y genera el compilado (.jar)

Se añade a la tarea el escáner del código para evaluar su calidad mediante SonarQube.

SonarQube [22] es una plataforma opensource para evaluar código fuente. Usa diversas herramientas de análisis estático y trabaja en base a unas reglas definidas y umbrales de calidad para obtener métricas que ayudan a mejorar la calidad del código de un programa. Los aspectos que evalúa entre otras cosas son los siguientes:

- Bugs: fallos de código que se podrían mejorar o que si no se arreglan pueden terminar siendo un problema.
- Vulnerabilidades y Hotspot: aspectos de seguridad.
- Code Smells: código que se puede optimizar o mejorar.
- Coverage: cobertura de test, siendo lo mejor estar cerca de 100%.
- Duplications: código duplicado, siendo lo mejor estar cerca de 0%.
- Lines: líneas de código.

A continuación se muestra un ejemplo de salida de un análisis de código fuente excelente:



Para implementar SonarQube se instala como un contenedor de Docker:

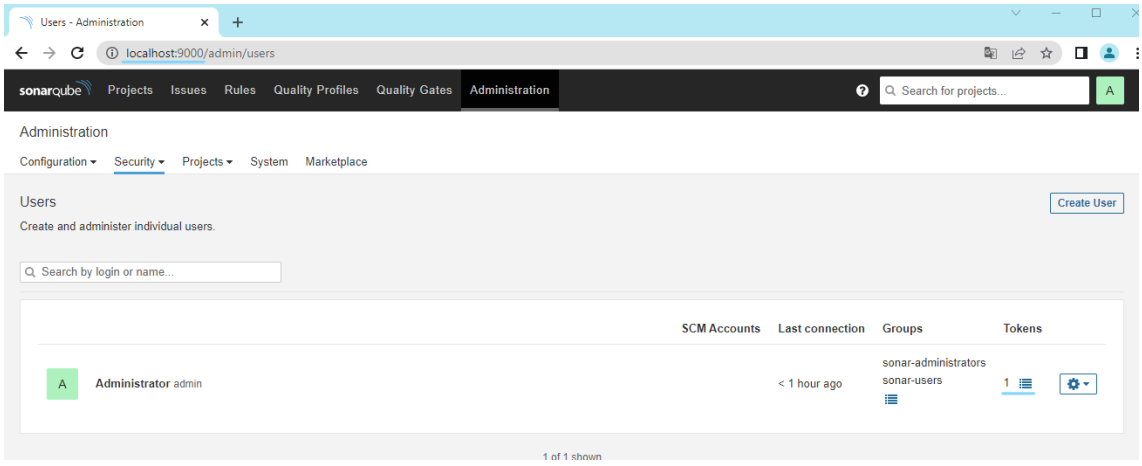
```yaml
#sonar service
sonar:
  image: sonarqube:community
  container_name: sonarqube
  depends_on:
    - sonar-db
  environment:
    SONAR_JDBC_URL: jdbc:postgresql://sonar-db:5432/sonar
    SONAR_JDBC_USERNAME: sonar
    SONAR_JDBC_PASSWORD: sonar
  ports:
    - "9000:9000"
  volumes:
    - sonar-data:/opt/sonarqube/data
    - sonar-extensions:/opt/sonarqube/extensions
    - sonar-logs:/opt/sonarqube/logs
sonar-db:
  image: postgres:14
  container_name: sonar-db
  environment:
    POSTGRES_USER: sonar
    POSTGRES_PASSWORD: sonar
  volumes:
    - postgresql-config:/var/lib/postgresql
    - postgresql-data:/var/lib/postgresql/data
```

Como se puede ver se utiliza de imagen de SonarQube la última versión “community”, y se apoya en una base de datos PostgreSQL versión 14 para la que se define un usuario de acceso. Ambos elementos utilizan volúmenes para guardar los datos y configuraciones.

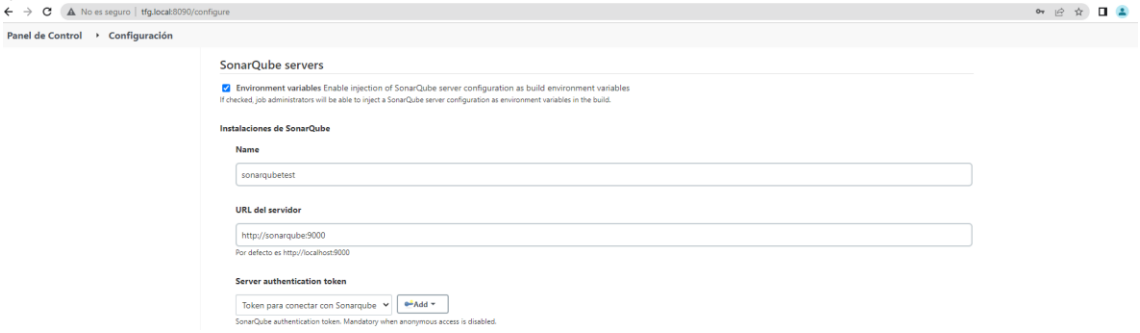
Posteriormente, tras levantar el contenedor, se configuran sus plugins dentro de Jenkins [23] para finalmente añadirlo a la tarea para que funcione de forma automática y realice el análisis del código. Como se debe comunicar con Jenkins, en este caso se genera una red virtual entre ambos contenedores con los siguientes comandos.

```
docker network create jenkins_sonarqube      → creación red
docker network connect jenkins_sonarqube jenkins    → inclusión contenedores
docker network connect jenkins_sonarqube sonarqube
```

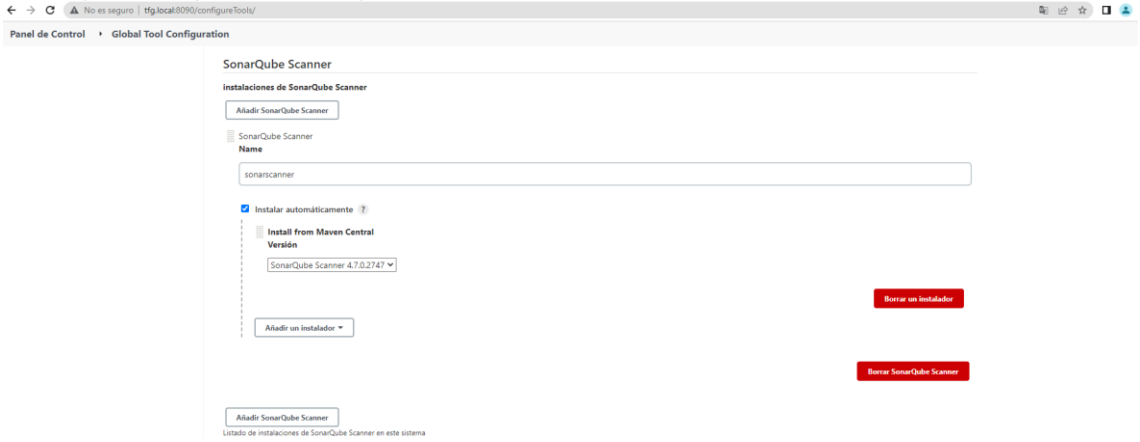
Para realizar la conexión con SonarQube desde Jenkins se genera un token entrando en la interfaz de SonarQube, conectándonos a través del puerto en el que se ha expuesto al crear el contenedor (9000).



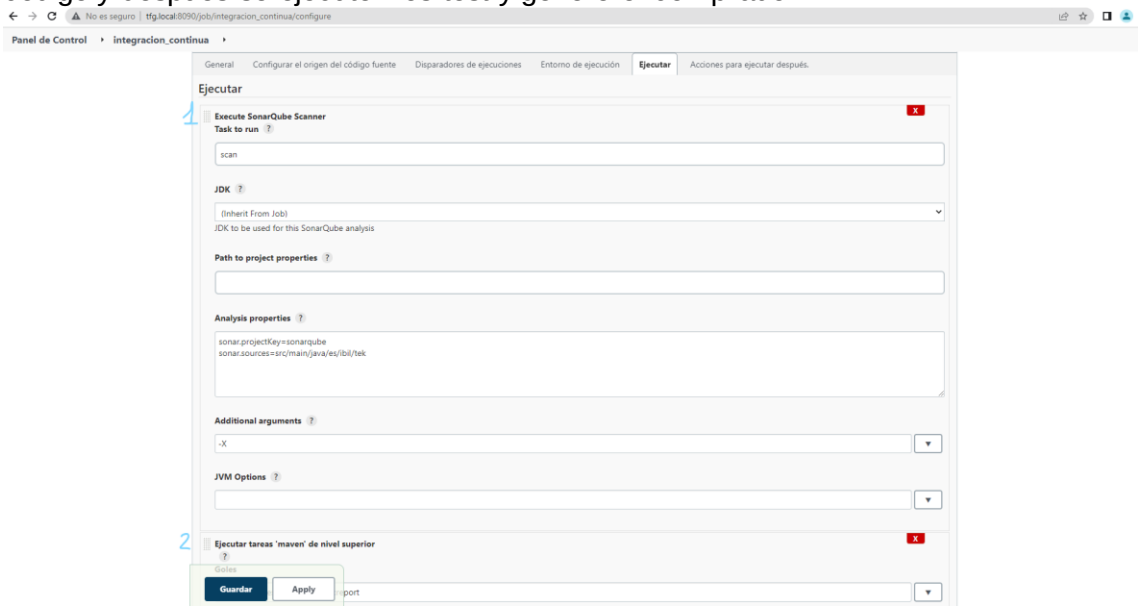
Hecho esto, e instalado el plugin de SonarQube en Jenkins, lo primero se configura el servidor SonarQube que es el contenedor, y al que se consigue conexión con el token generado anteriormente.



En segundo lugar, se configura el escáner.



Por último, se edita la tarea generada anteriormente añadiendo otra subtarea a ejecutar, que define dónde está el código a analizar (/src/main/java/…). Además, se puede establecer el orden de ejecución de las subtareas, y se fija que primero se analice el código y después se ejecuten los test y genere el compilado.



Tras esto se ejecuta la tarea y una vez finaliza se muestra la ejecución de análisis de código en SonarQube tanto en Jenkins, como en la interfaz de SonarQube donde podemos ver más en detalle los resultados para analizar posibles puntos de mejora.

11. Kubernetes

Kubernetes [24] es una plataforma de código abierto portable y extensible para administrar cargas de trabajo y servicios. Para ello gestiona clústeres (compuestos por un número de nodos (máquinas) adaptado a la capacidad de cómputo necesaria), y con ellos facilita la automatización y explota la escalabilidad horizontal y maneja el balanceo de carga buscando conseguir alta disponibilidad.

La escalabilidad es la capacidad de los sistemas para adaptarse al crecimiento de estos tanto por complejidad como por demanda. Por ejemplo, de repente se pasa de 10 a 1000 usuarios y por tanto aumenta la complejidad haciendo necesarios más recursos (más RAM, mejor velocidad de procesamiento, almacenamiento, etc.). Kubernetes ayuda a gestionar este aspecto permitiendo generar más nodos dentro de un clúster para adaptarse a la carga de trabajo de forma rápida y sencilla.

Los componentes que se generan con Kubernetes se denominan Objetos, y se interactúa con ellos a través de una API a la que se envían diferentes funciones a ejecutar como la definición de los Objetos, el envío de los Objetos a un clúster, etc.

Otros elementos de Kubernetes importantes son:

- PODs: Componente clave en el clúster de Kubernetes, siendo la unidad más pequeña que se encarga de gestionar. Es un grupo de uno o más contenedores que comparten almacenamiento, red y las especificaciones de cómo ejecutarse. Además, son efímeros, en cualquier momento de puede matar/eliminar un POD y recrearlo.
- Deployments: Definición de cómo será la implementación de Kubernetes, por ejemplo, el número de replicas que se manejan para conseguir alta disponibilidad. Define el estado deseado de la implementación, ejecuta las múltiples réplicas de la aplicación y reemplaza las defectuosas/que no responden.
- Services: Definición de cómo exponer una aplicación que se ejecuta en un conjunto de PODS al usuario final a través de la red. Por defecto el balanceo de carga se hace mediante Round-Robin.
- Config Map: Ficheros de configuración utilizados por los PODs. No soportan la encriptación, por tanto, no se deben meter datos sensibles.
- Labels: Parejas clave-valor para organizar los objetos.
- Selectores: Peticiones para hacer consultas a las labels (por ejemplo, obtener todos los PODs con ciertas características).

Por último, se debe entender la arquitectura interna de un clúster de Kubernetes, que es donde se van a ejecutar las aplicaciones. Dentro del clúster tenemos dos tipos de nodos, una serie de nodos “workers”, que albergan los PODs (componentes con carga de trabajo de la aplicación a desplegar) y un nodo “master”, que gestiona los “workers”.

11.1 Generación de imágenes Docker

En primer lugar, situándonos en los directorios que contienen los Dockerfile que definen las imágenes de los contenedores del backend y frontend, se generan las imágenes Docker que se utilizarán por Kubernetes.

Esto se consigue con el siguiente comando:

```
docker build -t tek-api   → desde /java
docker build -t tek-web   → desde /angular
```

No se sigue la nomenclatura oficial para poder publicar las imágenes en Docker Hub debido a que por cuestiones de privacidad no se van a subir al repositorio. Esto se debe a que por defecto los repositorios de Docker Hub son públicos, para hacerlos privados hay que pagar.

11.2 Despliegue

Para desplegar los Kubernetes se hace uso de minikube y kubectl.

En primer lugar, se levanta minikube, se comprueba que está arrancado y se accede a su dashboard que sirve de apoyo visual para ver que los PODs y servicios se generan correctamente.

Para esto se ejecuta:

```
minikube start      → arranca el clúster de Kubernetes
minikube status     → aparece running
minikube dashboard  → arrancamos interfaz visual Kubernetes
```

A continuación, se generan los PODs partiendo de las imágenes Docker creadas previamente:

```
kubectl run api --image=tek-api:latest --port=8080 8080
kubectl run web --image=tek-web:latest --port=80 80
```

Tras estos comandos se nos informa por consola de que los POD se han creado, y además se puede comprobar en el dashboard de Kubernetes.

Además, se pueden listar todos los PODs, mostrar todos los detalles de uno de ellos o borrar mediante los siguientes comandos:

```
kubectl get pods
kubectl describe pod <pod-name>
kubectl delete pod <pod-name>
```

En este momento, no se puede acceder a los PODs creados porque no son visibles desde fuera del clúster de Kubernetes. Para hacerlos visibles, se crean sus servicios.

Para crear los servicios se ejecuta:

```
kubectl expose pod api --port=8081 --target-port=8080
kubectl expose pod web --port=81 --target-port=80
```

Tras estos comandos se nos informa por consola de que los servicios se han creado, y además se puede comprobar en el dashboard de Kubernetes. Y al igual que ocurre con los PODs, podemos listar los servicios, verlos en detalle y borrarlos:

```
kubectl get services
kubectl describe services <pod-name>
kubectl delete service <pod-name>
```

Por último, para poder acceder a los servicios desde el exterior se les asigna una IP externa.

```
minikube service api --url
minikube service web --url
```

Y se accede a los servicios desde el navegador con la URL que aparece.

Para que funcione se debe exponer el puerto del .jar ejecutable del API y, al igual que con Docker, crear un proxy inverso para comunicar ambos PODs. Sin embargo, en este caso solo se quiere mostrar como funcionaría la exposición por Kubernetes, pero no se hace funcional debido a que se escapa del alcance de este trabajo y conlleva gran complejidad.

Para mostrar de forma más gráfica todo el proceso que se ha seguido hasta este punto se presenta el siguiente diagrama:

El desarrollador sube su código al GitLab, de donde Jenkins lo obtiene, hace un análisis de la calidad del código mediante SonarQube y procede a ejecutar los test con JUnit para comprobar que el código es correcto. Si todo es correcto se genera el .jar ejecutable mediante Maven. Este mismo proceso se haría también con el código del front, no solo con el código del back, para verificar que todo es correcto, pero esto no se ha llevado a cabo debido a que excedía el alcance del trabajo.

Una vez verificado el código y generados los empaquetados, se generan las imágenes Docker actualizadas, y dichas imágenes son las que se levantan mediante Kubernetes desplegando así la aplicación. Para que este proceso se hiciera de la forma más óptima se implementaría entrega continua, o lo que es lo mismo despliegue continuo (CD), integrando la generación de imágenes y el despliegue en Jenkins, subiendo cada versión de imagen generada a Docker Hub. Sin embargo, debido a motivos de privacidad de código, este aspecto no se puede implementar.

Por último, para facilitar el mantenimiento y control del correcto funcionamiento de los diferentes servicios que conforman la aplicación se añade monitorización.

12. Monitorización mediante Prometheus y Grafana

Gracias a la monitorización se permite controlar el correcto funcionamiento de todos los servicios y se realizan diagnósticos para saber si algo funciona mal y buscar soluciones, pudiendo detectar por ejemplo anomalías o si algún servicio se cae, lo que podría provocar grandes fallos en el sistema.

Es decir, la monitorización se puede hacer a diferentes niveles, desde a una infraestructura física hasta a un componente funcional (servicio). Además, se pueden programar alertas en tiempo real, notificaciones, etc.

12.1 Prometheus

Prometheus [25] es un sistema de monitorización de eventos y alertas opensource. Fue creado por SoundCloud en 2012 y hoy en día lo mantiene Cloud Native Computing Foundation (Linux Foundation). Aporta ventajas en monitorización de infraestructuras complejas/grandes.

El componente principal es Prometheus Server. Este componente realiza la extracción y almacenamiento de métricas en tiempo real y maneja la persistencia. Por otro lado, también es importante el Client Library, que instrumenta el código de aplicaciones (por ejemplo, si al construir un microservicio se le quiere dotar de eventos/notificaciones, esto se embebe en el servicio mediante el Client Library), y Alert Manager, que maneja las alertas basadas en reglas (por ejemplo, envía una notificación cuando se alcanza cierto consumo de memoria o utilización de CPU por un servicio).

Pero, en resumen, el funcionamiento de Prometheus se basa en recuperar información de los servicios haciendo barridos periódicos y almacenando los datos extraídos en bases de datos. Esto se consigue mediante jobs.

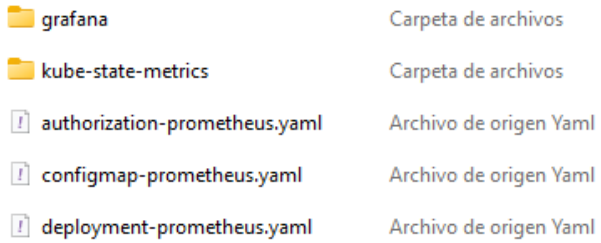
12.2 Grafana

Grafana [26] es una plataforma de web opensource de análisis de visualización interactiva, proporciona gráficos, tableros de mando y alertas web al conectarse a una fuente de datos. Soporta múltiples fuentes de datos (AWS, Prometheus, MySQL, PostgreSQL, etc.).

En resumen, Grafana genera una vista más agradable e interactiva de los datos de monitorización recopilados por una fuente de datos como Prometheus.

12.3 Implementación mediante Kubernetes y ficheros de configuración YAML

Para probar estas herramientas [27] se utilizan una serie de ficheros de configuración de tipo YAML incluidos en una carpeta de nombre “monitoring” que contiene lo siguiente:



Las reglas que se utilizarán son las incluidas en la subcarpeta “kube-state-metrics”, y que se definen mediante ficheros de configuración YAML.

Empezamos con la configuración de Prometheus. En primer lugar, se crea un clúster virtual (namespace) dentro de un clúster físico de Kubernetes con el nombre de "monitoring".

```
kubectl create namespace monitoring
```

Se otorga este nombre porque es con el que se referencia al servicio de monitorización en los ficheros YAML.

En segundo lugar, vamos aplicando los diferentes ficheros de configuración.

1. Primero, se ejecuta "authorization-prometheus.yaml", que crea un usuario "prometheus" que define qué acciones se pueden ejecutar sobre el clúster de Prometheus utilizando para ello sus permisos.

Para aplicar este primer fichero se ejecuta:

```
kubectl apply -f authorization-prometheus.yaml
```

Se puede comprobar que se ha creado este nuevo usuario/rol consultando el dashboard de Kubernetes y consultando los "Cluster Roles" disponibles.

2. Segundo, vamos a definir la configuración global que define los jobs, creando para ello un volumen para que la configuración se mantenga de forma persistente. Esto se define en el fichero "configmap-prometheus.yaml". Además, también en este fichero, se define una alerta que se repetirá cada minuto y los diferentes jobs con sus funciones (recopilación de información de PODs, recopilación de información de endpoints de los servicios, etc.).

Para aplicar este segundo fichero se ejecuta:

```
kubectl apply -f configmap-prometheus.yaml
```

Se puede comprobar que se ha creado consultando en el dashboard de Kubernetes los "Config Maps" existentes filtrando con "monitoring".

3. Tercero, se genera un contenedor Prometheus partiendo de una imagen de Docker Hub y definiendo el servicio a partir de esta imagen, al igual que se ha hecho con otros contenedores explicados con anterioridad. Al igual que con otros contenedores, se define un volumen para almacenar sus datos de forma persistente, y se expone para ser accesible desde fuera del clúster de Kubernetes. Todo esto se encuentra en "deployment-prometheus.yaml".

Para aplicar este tercer fichero se ejecuta:

```
kubectl apply -f deployment-prometheus.yaml
```

Una vez aplicados todos los ficheros, se puede comprobar que se tiene todo lo necesario para usar Prometheus ejecutando el siguiente comando:

```
kubectl get all --namespace=monitoring
```

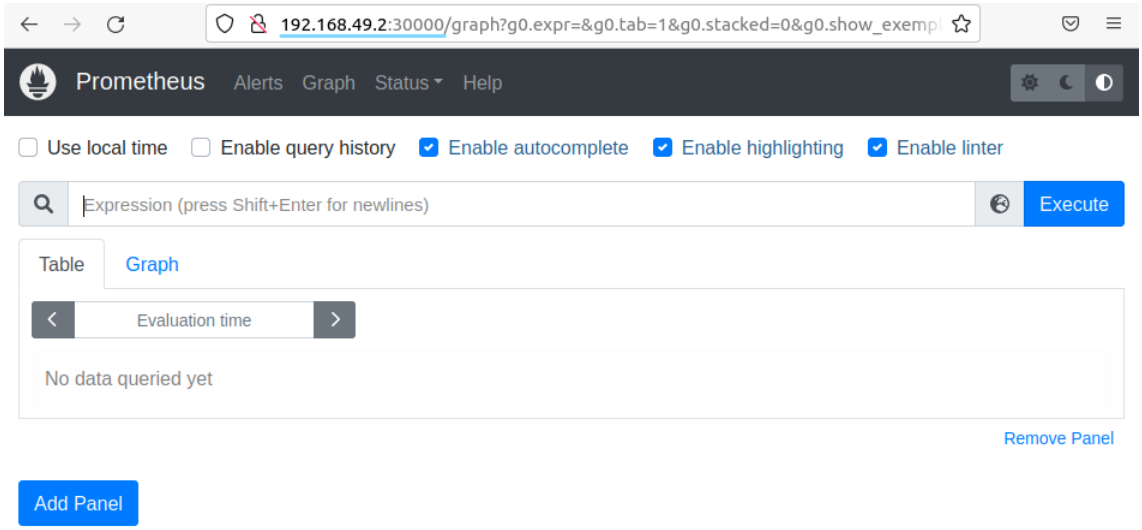
```
vagrant@ubuntu-focal:~/Documents/monitoring$ kubectl get all --namespace=monitoring
NAME                                         READY   STATUS              RESTARTS   AGE
pod/prometheus-deployment-b65d5d898-kj7g7    0/1     ContainerCreating   0          16s

NAME                           TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
service/prometheus-service     NodePort   10.104.47.33    <none>        8280:30000/TCP   16s

NAME                                    READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/prometheus-deployment   0/1     1            0           16s

NAME                                               DESIRED   CURRENT   READY   AGE
replicaset.apps/prometheus-deployment-b65d5d898    1         1         0       16s
```

Lo más importante, es el puerto por el que se escucha el servicio, que se establece en el 30000. A continuación, se obtiene la IP en la que se expone con:

```
minikube ip
```

```
vagrant@ubuntu-focal:~/Documents/monitoring$ minikube ip
192.168.49.2
```

E introduciendo en un buscador "<IP>:<puerto>", es decir, "192.168.49.2:30000" se accede al servicio de Prometheus.



Ahora, se ejecutan las reglas para obtener métricas de los objetos. Para ello se ejecutan todos los ficheros de configuración de reglas incluidos en "kube-state-metrics" con:

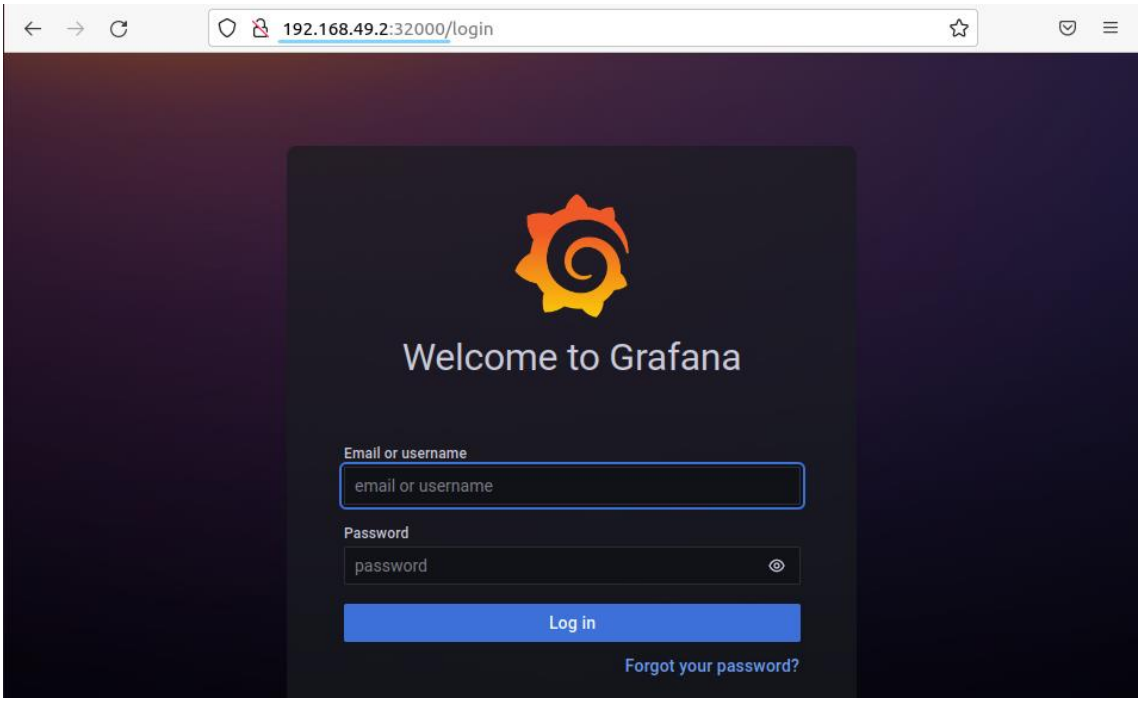
```
kubectl apply -f kube-state-metrics/
```

Para evaluar las métricas, se implementa Grafana, puesto que es una interfaz mucho más amigable para analizarlas. Para ello se hace uso de los ficheros de la subcarpeta "grafana". Dentro están por un lado "configmap-grafana.yaml" con el que se define la fuente de datos recopilados que se deben mostrar, siendo en este caso lo proporcionado por el servicio Prometheus. Por otro lado está "deployment-grafana.yaml", con el que se define el contenedor de Grafana que crea el servicio junto con un volumen para guardar sus datos de forma persistente, los puertos en los que se expone, etc.

Para aplicar la configuración se ejecuta:

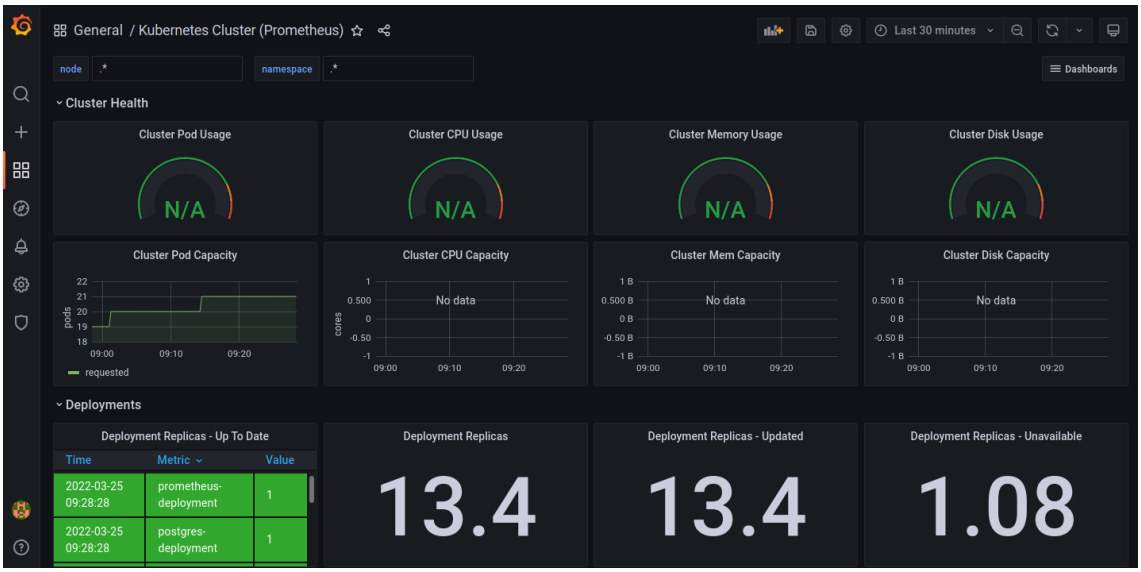
```
kubectl apply -f grafana/
```

Para acceder a la interfaz de Grafana se introduce en el navegador “<IP>:32000”.



Finalmente, solo nos queda conectar el dashboard de Grafana con Prometheus [28] para que muestre los datos recopilados por él. Para ello, desde la interfaz de Grafana se hace “Create → Import → URL plantilla deseada (6417 en este caso)”.

Tras esto, se puede visualizar el dashboard de Grafana con los datos recopilados por Prometheus acerca de los servicios del sistema mostrando uso de CPU, almacenamiento, etc.



13. Análisis final, futuras mejoras y conclusiones

El desarrollo de este trabajo ha sido un proceso completo y complejo, puesto que se afrontan tareas tanto del perfil de desarrollador software como de administrador de sistemas, estudiando por tanto múltiples escenarios y aspectos de la informática, gracias al empleo de técnicas DevOps.

Visto de forma numérica, antes de mi trabajo como SER (Site Reliability Engineer), cualquier nuevo desarrollador que se incorporase al proyecto tenía que configurar de forma manual su equipo para poder trabajar con la aplicación Tek teniendo que descargarse las versiones adecuadas de los diferentes componentes y este proceso conlleva entre dos y tres días. Además, este nuevo desarrollador puede estar a cargo de mantener también otras aplicaciones que pueden compartir componentes con Tek, como por ejemplo Java, pero que se utilicen en diferentes versiones, lo que podría ser una fuente de complicaciones.

Sin embargo, tras mi desarrollo DevOps, el equipo de este nuevo desarrollador estaría listo para desplegar la aplicación en a lo sumo media hora teniendo que descargarse únicamente Vagrant para desplegar el entorno de trabajo completo, y dentro de este entorno se haría el despliegue sin tener que descargar nada de forma local.

Por lo tanto, analizando el trabajo desde el punto de vista empresarial se aporta un gran valor mediante el uso de estas técnicas ya que se tiene a los diferentes desarrolladores listos para trabajar en mucho menor tiempo, y además se garantiza un trabajo de mayor calidad y organización.

Con las técnicas DevOps se tiene garantía de que todo lo que un desarrollador suba al repositorio estará actualizado y corriendo en producción gracias al ciclo de trabajo desarrollado. Y, además, este proceso se realiza de forma automática, y se monitoriza facilitando el mantenimiento.

Para dar soporte a las tareas de tan diferente índole, se han manejado múltiples y variadas herramientas que he tenido que ir aprendiendo configurar correctamente como es Jenkins, Prometheus, Grafana, etc., además de lenguajes puesto he manejado y aprendido a comprender código de SpringBoot, Angular, directivas Maven, configuración de Docker y Docker-Compose, etc.

Gracias a contar con consejo profesional de compañeros de la empresa, la selección de herramientas ha sido más sencilla, pues para cada función existe más de una alternativa. De no haber sido así, habría tenido que implementar y analizar cada herramienta existente para llevar a cabo cada tarea para posteriormente elegir de entre todas la mejor opción.

Como es natural, durante el desarrollo han ido surgiendo algunos problemas, que he tenido que enfrentar y solucionar empleando diferentes técnicas, que han modificado la planificación inicial. Los problemas a destacar son los mencionados a continuación.

El primer problema se produjo al desarrollar el entorno de trabajo debido a aspectos de autenticación mediante llaves públicas/privadas.

El segundo problema ocurrió en la configuración del entorno remoto, es decir, en la definición de los diferentes servicios, puesto que se produjo un problema de CORS, ya que los contenedores de frontend y backend debían responder bajo un mismo nombre de dominio. También supuso un problema el hecho de que el proyecto no está desarrollado para DevOps por lo que los logs están configurados de un modo que dificulta su acceso, y esto dificultó encontrar el problema que se estaba produciendo.

El último imprevisto surgió en la integración continua al añadir la evaluación de la calidad del código debido a que las últimas versiones, más optimizadas y seguras, de SonarQube requieren Java 11, y el proyecto estaba desarrollado en Java 8, por lo que se tuvo que actualizar.

Sin embargo, puesto que se tuvo en cuenta un margen al planificar las diferentes tareas, se ha podido cumplir con el plazo de entrega, pues aunque inicialmente estaba planeado para finalizarse el 1 de junio, con la desviación provocada por los problemas mencionados previamente se finalizó el 6 de junio.

Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
Formación en herramientas y técnicas DevOps	25 días	lun 14/02/22	vie 18/03/22	
Desarrollo entorno de trabajo remoto	13 días	mar 01/03/22	jue 17/03/22	
Configuración entorno local	4 días	mar 15/03/22	vie 18/03/22	
Despliegue local	5 días	lun 21/03/22	vie 25/03/22	3
Configuración entorno remoto	20 días	lun 28/03/22	vie 22/04/22	2
Despliegue remoto	7 días	lun 25/04/22	mar 03/05/22	5
Testing	5 días	jue 28/04/22	mié 04/05/22	
Integración continua	13 días	jue 05/05/22	lun 23/05/22	7
Independización Kubernetes	7 días	mar 24/05/22	mié 01/06/22	8
Monitorización	3 días	jue 02/06/22	lun 06/06/22	9

En cuanto a líneas de mejora para el futuro, se añadiría “testcontainers”, que es una extensión de JUnit para controlar los contenedores Docker. También se contempla añadir al desarrollo el uso de “kind” para Kubernetes, puesto que consigue alta disponibilidad. Y por último se valora además pasar a utilizar un proveedor cloud como Azure o AWS para no tener que instalar nada de forma manual, sino partir de máquinas ya aprovisionadas y con mantenimiento.

En conclusión, debo destacar la importancia de aplicar técnicas DevOps en el desarrollo de proyectos software puesto que facilita mucho el trabajo y además hace que sea más eficiente, pues se consigue un uso óptimo de los recursos y un desarrollo de software de calidad en menor tiempo. Además, facilita el mantenimiento de las aplicaciones y su portabilidad, al contener todos los elementos necesarios para el despliegue en una máquina que es la que se debe mantener, y cuyos componentes se replican en otros servidores, pero al ser réplicas del servidor original no suponen esfuerzo de mantenimiento.

BIBLIOGRAFÍA

[1] ¿Qué es DevOps? – microsoft.com - https://azure.microsoft.com/es-es/overview/what-is-devops/

[2] Concepto DevOps – redhat.com - https://www.redhat.com/es/topics/devops

[3] ¿Qué es DevOps? – amazon.com - https://aws.amazon.com/es/devops/what-is-devops/

[4] ¿Qué es Vagrant? – vagrantup.com - https://www.vagrantup.com/intro

[5] Documentación Vagrant – vagrantup.com - https://www.vagrantup.com/docs

[6] Aprovisionar Vagrant – vagrantup.com - https://www.vagrantup.com/docs/provisioning/shell

[7] Repositorio Vagrant con imágenes desarrolladas - https://app.vagrantup.com/veronicafg

[8] Guía instalación Sourcetree – sourcetree.com - https://www.sourcetreeapp.com/

[9] Guía instalación Node.js y npm – npmjs.com - https://docs.npmjs.com/downloading-and-installing-node-js-and-npm

[10] Guía instalación Java – java.com - https://www.java.com/es/download/help/windows_manual_download.html

[11] Descarga Eclipse – eclipse.org - https://www.eclipse.org/downloads/

[12] Descarga VS Code – visualstudio.com - https://code.visualstudio.com/download

[13] Descarga PgAdmin – pgadmin.org - https://www.pgadmin.org/download/

[14] Descarga Postman – postman.com - https://www.postman.com/downloads/

[15] Descarga y uso Nginx – nginx.com - https://www.nginx.com/resources/wiki/start/topics/tutorials/install/

[16] Primeros pasos Docker – docker.com - https://www.docker.com/

[17] Documentación Docker – docker.com - https://docs.docker.com/

[18] Documentación Docker-Compose – docker.com - https://docs.docker.com/compose/

[19] ¿Qué es IC (integración continua)? – amazon.com - https://aws.amazon.com/es/devops/continuous-integration/

[20] Descarga Jenkins – jenkins.io - https://www.jenkins.io/

[21] Documentación Jenkins – jenkins.io - https://www.jenkins.io/doc/

[22] Documentación SonarQube – sonarqube.org - https://docs.sonarqube.org/latest/

[23] Configuración Sonar en Jenkins – sonarqube.org - https://docs.sonarqube.org/latest/analysis/jenkins/

[24] ¿Qué es Kubernetes? – kubernetes.io - https://kubernetes.io/es/docs/concepts/overview/what-is-kubernetes/

[25] ¿Qué es Prometheus? – prometheus.io - https://prometheus.io/docs/introduction/overview/

[26] ¿Qué es Grafana? – grafana.com - https://grafana.com/grafana/

[27] Configuración Prometheus y Grafana – grafana.com - https://grafana.com/docs/grafana/latest/getting-started/getting-started-prometheus/

[28] Integración Grafana en Prometheus – prometheus.com - https://prometheus.io/docs/visualization/grafana/