Proyecto Cloud Computing

Máster Universitario en Computación en la Nube y de Altas Prestaciones - 2022/23

Salvador Chinesta Llobregat
Ernesto Gaspar Aparicio

1. Introducción

El cloud computing, es un modelo tecnológico que se está implementando en la mayoría de empresas en los últimos años. Sus ventajas son obvias, ya sea porque se usa Software como servicio (SaaS), Plataforma como servicio (PaaS) o Infraestructura como servicio (IaaS). Ello supone para las empresas un ahorro de costes, espacios seguros, fácil accesibilidad y una personalización del servicio según los requisitos del cliente.

El objetivo de este proyecto es el de construir un servicio complejo basado en microservicios, con el fin de mostrar las virtudes de este modelo tecnológico y asemejarse en la medida de lo posible al sistema de un entorno de producción real.

2. Descripción del proyecto

La finalidad de este trabajo es construir un sistema de colas de trabajo utilizando varios microservicios. Este sistema, basado en una API REST, deberá ser capaz de enviar trabajos a una cola de mensajes, a partir de la cual podrán ser extraídos por un *worker* que se encargará de realizar el cómputo de este trabajo. A su vez, este agente, deberá guardar la la salida estándar y la de error de este trabajo en un almacenamiento de tipo S3, que utilizará el cliente para obtener una respuesta de vuelta.

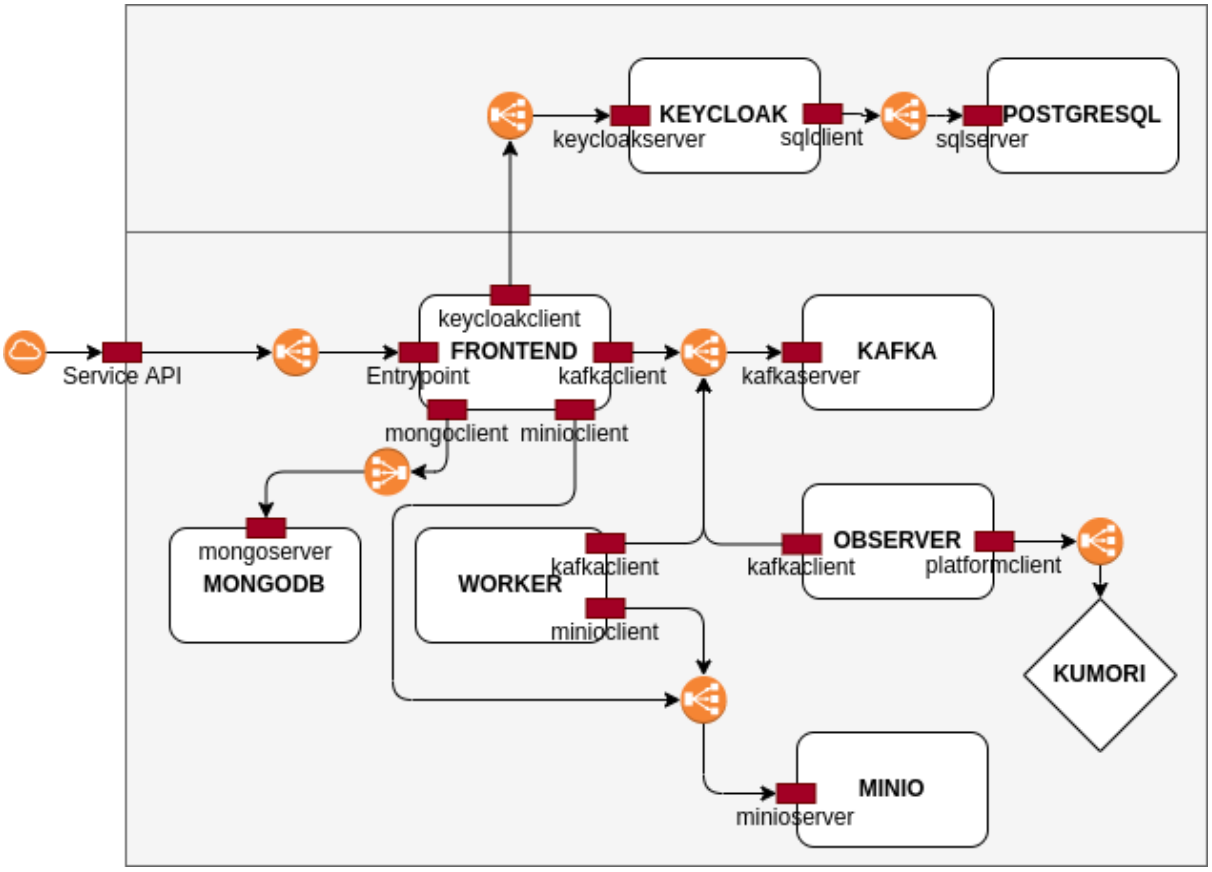
Se ha estructurado el trabajo con los siguientes objetivos:

2.1 Objetivos

1. Implementar los microservicios necesarios para cumplimentar la funcionalidad descrita.
2. Construir las imágenes de Docker necesarias para los microservicios.
3. Preparar un *docker-compose* con un fichero *.env* para desplegar el sistema entero en una sola máquina.
4. Desplegar el sistema en un KPaaS (Kubernetes Platform as a Service).
5. Diseñar una estrategia de elasticidad.
6. Implementar la estrategia de elasticidad.

3. Arquitectura

Una vez visto qué es lo que se pretende conseguir con este proyecto, pasaremos a diseñar la solución. Tras varias fases de refinamiento e investigación esta es la solución a la que se ha llegado.



Uno de los requisitos para esta solución, es que esta debe de ser elástica y resiliente. Para ello, aquellos servicios que se van a desarrollar deben de seguir algunos principios como podrían ser:

- **Comunicación desacoplada**, proporcionada por Kafka mediante el concepto de *Cola de trabajos*.
  Otro punto que proporciona esta característica, es el uso de *balanceadores de carga,* que permiten que los diferentes servicios y clientes se comuniquen entre las diferentes réplicas de las posibles instancias que existan, de forma que, no tienen que conocer las direcciones únicas de cada una de las instancias.

- **Stateless**, es decir, los diferentes servicios desarrollados no deben de tener estado ni ningún tipo de información referente a los trabajos ni usuarios, para esto, tenemos una capa de persistencia dedicada con *MongoDB*, que almacenará el estado de los diferentes trabajos y *Minio* que se encargará de almacenar los resultados obtenidos de ejecutar los diferentes trabajos.

Ahora se realizará una pequeña descripción de cada uno de los servicios que aparecen en el diagrama:

- **Frontend** -> Servicio que es la capa de presentación de la solución. En esta primera versión, se trata de una API de tipo REST la cual es visible para el público y que está protegida mediante **Keycloak** que persiste su información en **PostgreSQL**. Acepta crear y eliminar trabajos, así como, ver el estado de los mismos. Cuando se le pida añadir trabajos añadirá un mensaje en la cola de **Kafka** que corresponda. También se encarga de recoger el estado de los trabajos de **Kafka** y almacenarlo en **MongoDB**.

- **Worker** -> Servicio encargado de configurar, lanzar y recolectar los resultados de los trabajos, los resultados se guardarán en **Minio**. Otra de sus funciones es que cuando recibe un trabajo, emite mensajes sobre el estado del mismo.

- **Observer** -> Servicio encargado de observar cuál es el estado del sistema. Se dedica a escuchar las diferentes colas de **Kafka** y a partir de ahí proporcionar estadísticas sobre el uso/carga del sistema. Esto en un futuro podría servir para que, según se le indique, llegados a ciertos umbrales, este servicio se comunique con **Kumori**, la plataforma de despliegue, y le indique que despliegue nuevas instancias si el uso del sistema aumenta, o que repliegue si el uso del sistema disminuye.

3.1 Definición de un trabajo

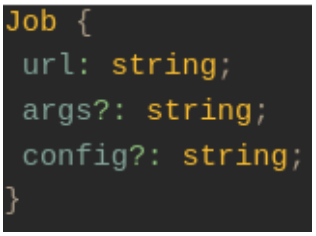
Un **trabajo** viene definido por unos *datos de entrada* y *unas acciones*, a aplicar sobre los datos proporcionados y se espera que genere un resultado.

Como se puede ver, es un concepto un tanto amplio y que puede dar lugar a muchas interpretaciones. Por eso, una parte muy importante del diseño de esta solución era la de diseñar un estándar que permitiera abarcar este abanico tan amplio sin tener que llegar a poner restricciones sobre, por ejemplo, las tecnologías con las que se tengan que programar las acciones los trabajos.

3.1.1 Nuestro contrato

Una vez vistas las necesidades que teníamos, se llegó a este estándar o contrato.
Cuando un cliente quiera añadir un trabajo, este, deberá de proporcionar la siguiente información sobre él:

- **url**: Dirección de un repositorio git el cuál tendrá o bien el código fuente o el binario con las acciones que se quieran realizar.
- **args**: Lista de argumentos con los que se quiere llamar al ejecutable
- **config:** Texto que será escrito en el archivo config.json en la carpeta raíz del proyecto. Puede contener desde claves necesarias para la ejecución del proyecto, hasta los datos de entrada con los que tendrá que tratar el ejecutable.



Visto que es aquello que forma un trabajo, vamos a ver cuales son las restricciones que aplicamos sobre cada uno de los campos mencionados.

Sobre el proyecto contenido en el **repositorio git**:

- Si el repositorio es privado la dirección que se tendrá que indicar debe de seguir el siguiente formato:
  https://<username>:<access-token>@github.com/<UserName>/<repo>.git
- Deberá de contar con un fichero *package.json*, el cual en su sección *scripts* tendrá una entrada *start*, en la cual se indicará el comando de aquello que se quiera ejecutar, es decir, dentro del Worker el trabajo se ejecutará con *npm run start*.
  Esto no supone ninguna restricción respecto a la tecnología en que esté programado el trabajo ya que el valor de *start* puede ser cualquier cosa que acepte un terminal.

  Por ejemplo:

```
# Para un proyecto de NodeJS (Soportado de forma nativa)
{
     ...
     "scripts": {
     "start": "npm install && node src/index.js"
     }
}
# Para un proyecto de Rust (NO Soportado de forma nativa, hay que
hacer la instalación de las dependencias)
{
     ...
     "scripts": {
          "installRust": "url --proto '=https' --tlsv1.2 -sSf
                         https://sh.rustup.rs | sh",
           "start": "npm run installRust
                    && cargo build src/main.rs --release &&
                    ./target/release/{projectName}"
     }
}
# Para ejecutar un comando de Bash(Soportado de forma nativa)
{
     ...
     "scripts": {
          "start": "apt install sysbench && sysbench"
     }
}
```

- Si el ejecutable genera algún archivo de salida, que se quiera obtener posteriormente, una vez finalizada la ejecución del trabajo, este deberá de guardarse en la carpeta *output* en la raíz del proyecto. Los archivos que se encuentren en esta

carpeta son los que se almacenarán en Minio para poder ser consultados posteriormente. Además de guardar los archivos de la carpeta indicada, de forma automática, se guardará también en Minio la salida estándar (*stdout*) y la salida de errores (*stderr*).

- En la raíz del proyecto no puede existir ningún archivo que tenga el nombre de *config.json* ya que este será sobrescrito con el contenido que se le haya indicado en el parámetro **config** del trabajo.

Sobre **args** no se aplica ningún tipo de restricción, puede ser cualquier argumento de entrada que acepte el programa a ejecutar.

Sobre **config** tampoco se aplican restricciones. Es simplemente texto en formato JSON el cual puede contener cualquier cosa que sea necesaria para la ejecución del trabajo. Desde claves para consumir alguna API externa, pasando por enlaces o datos de entrada del ejecutable. La estructura la define el programador del ejecutable.

Ejemplo de uso de **config** en el proyecto de ejemplo:

```
const translate = require("translate");
const fs = require("fs");
const config = require('./config.json')

async function translateText() {
  if (config.DEEPL_API_KEY === undefined || config.SOURCE_TEXT === undefined) {
    throw new Error(`
USO:
    1. En el archivo config.json debes de indicar:
      1.1. DEEPL_API_KEY ⟶ Que será la clave para poder habilitar la traducción.
      1.2. SOURCE_TEXT ⟶ Texto que quieres traducir
    `)
  }

  if (process.argv[2] === undefined) {
    throw new Error(`
USO:
  ${process.argv[0].split('/').splice(-1)} ${process.argv[1].split('/').splice(-1)} <language-to-translate>
    `)
  }

  translate.engine = "deepl";
  translate.key = config.DEEPL_API_KEY;

  const text = await translate(config.SOURCE_TEXT, process.argv[2]);
  fs.mkdirSync("./output", { recursive: true })
  fs.writeFileSync('./output/TranslatedText', text);
}

translateText();
```

En el proyecto de ejemplo, el código espera recibir *DEEPL_API_KEY* y *SOURCE_TEXT* desde el *config.json*, su aspecto sería el siguiente:

```
{
  "DEEPL_API_KEY": "[redacted]",
  "SOURCE_TEXT": "Work launched from the test area"
}
```

3.2 Estrategia de elasticidad

Como se ha comentado, el sistema tiene que ser elástico y resiliente. Aplicando los principios nombrados al principio de la sección, se consigue de forma sencilla que cada uno de los servicios programados puedan ser replicados proporcionando así mayor capacidad de cómputo en caso de aumentar la carga de trabajos, o disminuir el número de instancias en caso de una bajada de carga. Para poder tener una reducción de costes, o que puedan ser sustituidos sin ningún tipo de problema en caso de fallo de la instancia.

Teniendo elasticidad en el sistema, ahora hay que definir cómo utilizarla, ya que gracias a la elasticidad podremos ofrecer un nivel de **calidad** del servicio dado a los clientes y también controlar los gastos que produce el sistema, disminuyendo o aumentando la cantidad de réplicas de los diferentes componentes.

3.2.1 Definición de calidad

El primer paso para definir una estrategia de elasticidad es definir, en nuestro sistema, que es la calidad. En este sistema la calidad del servicio puede estar definida por los siguientes puntos:

- **Disponibilidad** → nos la ofrece la plataforma donde desplegaremos por lo que no es un punto importante en esta sección.
- **Tiempo de ejecución de un trabajo** → Dependiendo de las prestaciones del cluster donde se ejecuten las tareas pueden tardar más o menos.
- **Tiempo de servicio** →Tiempo que pasa entre que un cliente añade un trabajo a la cola de trabajos y este pasa a iniciar su ejecución.

Los siguientes puntos son los que restringen la elasticidad del sistema, afectando a la calidad:

- Presupuesto que se pueda destinar a gasto en computación.
- Tiempo de ejecución de un trabajo → Si los trabajos lanzados de por si tienen un coste computacional muy elevado pueden suponer un cuello de botella para el sistema.

Una posible solución en cuanto al “tiempo de ejecución de un trabajo”, que aparece tanto en el apartado de calidad como el de restricciones, sería limitar la cantidad de tiempo que puede estar un trabajo en ejecución o implementar un modelo de pago por horas de uso.

El punto de “tiempo de servicio” es un problema que la elasticidad puede solventar de forma sencilla. Agregando nuevas réplicas del servicio *Worker,* se puede conseguir que los trabajos que se encuentran en la cola de *Kafka* se lancen de forma paralela en vez de que se tuvieran que ir lanzando de forma secuencial. Aquí, se aplicaría la restricción de presupuesto, ya que dependiendo del modelo con el que se monetiza la solución sería interesante limitar la elasticidad del sistema para tener controlados los gastos del mismo.

Una vez visto que es aquello que podemos controlar con la elasticidad, presentaremos una posible estrategia que podría seguir este sistema.

3.2.2 Kafka

Un paso muy crítico para tener una elasticidad buena del sistema parte desde la configuración de los topics de Kafka. A modo de resumen, Kafka se configura como una cola de mensajes en las que hay un productor que los produce y uno o varios consumidores, agrupados por grupos, que los consumen.

Por defecto, un topic tiene una única partición la cual puede ser consumida sólo por un único consumidor, dentro del mismo grupo. Esto es un problema, porque en el caso de querer tener más de un consumidor (Worker) en el mismo grupo, solo uno recibirá trabajos, pero, si se opta por tener diferentes grupos no se asegura que los mensajes vayan a ser consumidos una única vez haciendo que se repita trabajo.

Una primera solución, aún, no del todo válida, para este problema, sería la de tener los consumidores dentro de un mismo grupo, para asegurar que los mensajes se consumen una única vez, y que se vaya creando una partición cada vez que se añade un consumidor.
Esto sigue sin ser del todo eficaz ya que cuando se crean particiones, éstas están vacías y entre particiones no se pueden repartir los mensajes que había en las particiones anteriores dejándonos prácticamente igual que estábamos, con una partición saturada y las nuevas sin trabajo que hacer.

Viendo este problema, parecería evidente entonces que desde un principio sería interesante tener un número muy elevado de particiones y aunque la solución podría ser válida, no es nada óptima ya que esto haría que el cluster de Kafka consumiese muchísimos recursos.

La solución que permite resolver este problema, se encuentra en un punto medio entre las dos posibles soluciones mostradas anteriormente, que consiste, en que deben de haber siempre más particiones que consumidores, ya que un consumidor puede consumir mensajes de más de una partición.
El número de particiones por consumidor podría venir dado por un valor que indique el Factor de Replicación, por ejemplo, 2, el cual nos indicaría que en todo momento nuestro sistema estaría listo para aumentar su capacidad 2 veces.

Por ejemplo, el sistema se inicia con 1 worker lo que nos daría como resultado la creación de 2 particiones (Nº Workers * Factor de Replicación). Entre estas se irán repartiendo los mensajes que vayan llegando. Llegados a cierto umbral, el sistema decide que necesita aumentar la capacidad de cómputo añadiendo un segundo worker, a este nuevo worker se le asignaría una de las particiones ya existentes, que ahora sí que tiene trabajos y además se crearían nuevas particiones, con lo que ahora tendríamos un total de 4 particiones (Nº Workers * Factor de Replicación) dándonos así capacidad de consumir los mensajes ya existentes previamente, la repartición de mensajes nuevos entre una mayor cantidad de particiones y la consecuente posibilidad de aumentar de forma correcta la capacidad del sistema con hasta 4 consumidores y así de forma iterativa.

3.2.3 La Estrategia

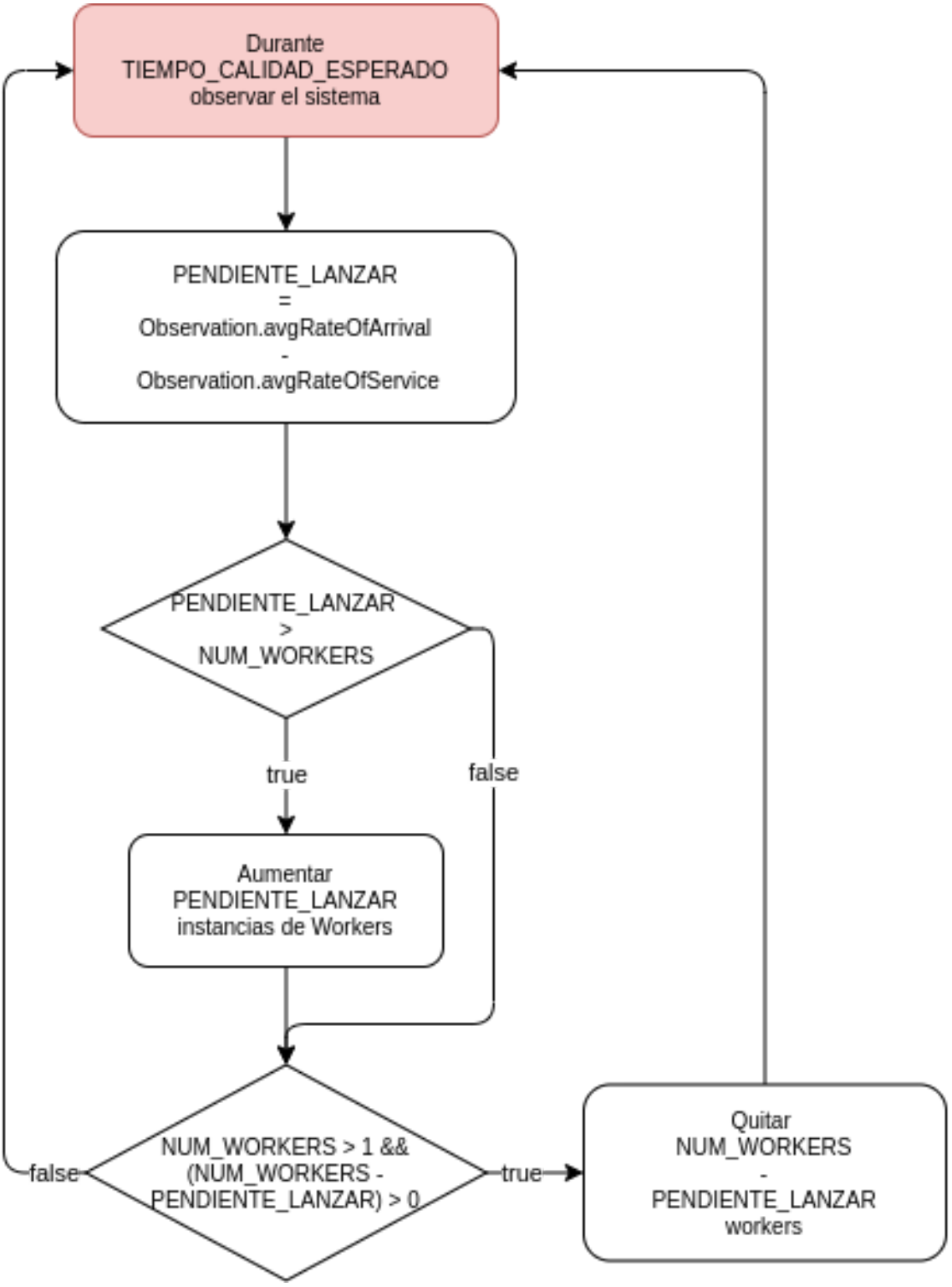
Ahora que ya tenemos un sistema perfectamente escalable, podemos pasar a tratar de ofrecer un cierto nivel de calidad a los clientes, enfocándonos principalmente en el tiempo que pasa entre que un cliente añade un trabajo a la cola de trabajos y este pasa a iniciar su ejecución.
Estos tiempos están controlados por el *Observer,* el cual periódicamente genera unas estadísticas con la siguiente estructura:

```
Observation {
 avgRateOfArrival: number;
 avgRateOfService: number;
 avgResponseTime: number;
 timestamp: number;
}
```

Donde:

- *avgRateOfArrival* -> Devuelve la cantidad de trabajos que se han añadido en el periodo.
- *avgRateOfService* -> Devuelve la cantidad de trabajos que han pasado de estado “En espera” a “Lanzado”.
- *avgRateOfFinished* -> Devuelve la cantidad de trabajos que han pasado de estado “Lanzado” a estado “Finalizado”.
- *avgResponseTime* --> Devuelve el tiempo medio de los trabajos terminados en el periodo.

El algoritmo sería el siguiente:



4. Implementación

4.1 Frontend

El frontend es la capa de presentación, es decir, la parte de la aplicación que interacciona con el usuario. Este servicio consiste en una API REST visible al público y protegida mediante *Keycloak*.

Este servicio ofrece las siguientes funcionalidades:

- Autenticación del usuario.
- Creación y eliminación de trabajos.
- Visualización del estado de uno o varios trabajos.
- Obtención del resultado de un trabajo.

En primer lugar, se establecen las rutas base de acceso, que tendrá la aplicación:

```
export function initRoutes(app: Express) {

  app.use('/health', statusRoutes)
  app.use('/api', jobRoutes)
  app.use('/auth', authRoutes)

};
```

Como se puede observar ofrece las direcciones, */health, /api, /auth*. Las funcionalidades principales se encuentran a través de la ruta */api*, pero es interesante echarle un vistazo al resto de rutas también.

4.1.1 Autenticación del sistema (/auth)

El usuario que quiera acceder al servicio, primero tendrá que hacer uso de esta ruta. Desde aquí, podrá iniciar sesión mediante la siguiente subdirección:

```
authRoutes.get('/:user/:password', async (req, res) => {
  const user = req.params.user;
  const password = req.params.password;
  let message = await login(user, password);

  res.send(message)
});
```

Mediante este sistema de inicio de sesión, se implementa una capa de seguridad la cual solo permite acceder al servicio a los usuarios registrados dentro de *KeyCloak*.
Para ello se llama a la función *login()*:

```
export async function login(user: string, password: string): Promise<any> {
  let message: any;
  try {
    const response = await getToken(user, password);
    message = response.data?.access_token;
  } catch (err: any) {
    const axiosError: AxiosError = err;
    message = axiosError;
  }
  return message;
}
```

Esta función verifica que efectivamente las credenciales del usuario se encuentran dentro del sistema y, además, si existe, devolverá un token que deberá utilizarse en la cabecera de las peticiones para obtener la autorización para hacer uso del servicio.

```
function getToken(user: string, password: string): Promise<AxiosResponse> {
  const kcconfig = JSON.parse(readFileSync("keycloak.json").toString())

  const data = qs.stringify({
    grant_type: 'password',
    client_id: kcconfig.resource,
    client_secret: kcconfig.credentials.secret,
    username: user,
    password,
  });

  const config = {
    method: 'post',
    url: `${kcconfig['auth-server-url']}realms/${kcconfig.realm}/protocol/openid-connect/token`,
    headers: {
      'Content-Type': 'application/x-www-form-urlencoded',
    },
    data,
  };

  return axios(config);
}
```

4.1.2 Estado del sistema (/health)

Esta ruta permite verificar el estado del sistema.

/health/

Devuelve una respuesta si el servidor está en funcionamiento.

/health/load

Devuelve la carga de trabajo que se encuentra en el sistema de colas actualmente.

4.1.3 Funcionalidades del sistema (/api)

En esta ruta se encuentran todas las funcionalidades principales del sistema. Es importante saber que todas estas funciones están protegidas con *Keycloak* mediante el middleware *keycloak.protect()*, lo cual permitirá proteger estos métodos en base a las políticas de acceso que tenga el usuario que los ejecuta.

/api/addJob

Permite al usuario agregar un nuevo trabajo. Si ha habido un error, devolverá un mensaje de error al usuario.
Este nuevo trabajo deberá seguir la misma nomenclatura de Nuestro Contrato.

```
jobRoutes.post('/addJob', keycloak.protect(), async (req, res) => {
  const url = req.body.url;
  const args = req.body.args;
  const jobConfig = req.body.config;
  const token = req.headers.authorization;
  const result = await addJob(url, args, jobConfig, token);
  res.send(result)
});
```

El sistema recoge los parámetros que introduce el usuario y llama a un método que se encargará de insertar un nuevo trabajo en la cola de *Kafka*.

Dentro de este método se verifica el repositorio de git indicado. Si no se indica un repositorio de git, se marcará el trabajo como “Fallido”, se avisará al usuario y no se ejecutará:

```
if (newJob.url === undefined) {
  jobStatus.status = "Fallido"
  await addJobStatus(jobStatus);

  throw new Error(ERROR_REPO_GIT)
}
```

Y si la aplicación falla por otro motivo al intentar insertar el trabajo, se devolverá un error:

```
} catch (err: any) {
  if (err.message === ERROR_REPO_GIT) {
    result = ERROR_REPO_GIT
  } else {
    result = 'Se ha producido un error al intentar crear el trabajo. Inténtalo más tarde'
  }
} finally {
  return result;
}
```

Si todo sale bien, y el resto de los parámetros son correctos se deberá poder insertar un nuevo trabajo a la cola de Kafka.

```
const messages = [
  { value: JSON.stringify(newJob) }
];
await producer.connect();
await producer.send({
  topic: 'jobs-queue',
  messages
});
result = `El id de tu trabajo es: ${newJob.id}`;
```

Dentro de la cola, el trabajo se mandará al topic *jobs-queue* y será consumido más tarde por un Worker.

/api/status/:id

Esta subruta implementa la funcionalidad de poder ver el estado de un trabajo concreto. Para ello, es necesario pasarle como parámetro el identificador del trabajo.

```
jobRoutes.get('/status/:id', keycloak.protect(), async (req, res) => {

  const jobId = req.params.id;
  const token = req.headers.authorization;

  let message: any = await checkJobStatus(jobId, token);

  res.send(message)
});
```

La función llama a otro método que se encarga de buscar en *MongoDB* un trabajo con el *id* dado que corresponda al usuario.

/api/status

Esta subruta devuelve el estado del conjunto de trabajos del usuario.

```
jobRoutes.get('/status', keycloak.protect(), async (req, res) => {
  const token = req.headers.authorization.replace('Bearer ', '')
  res.send(await showUserJobs(token));
});
```

/api/results/:id/:file

Permite obtener uno de los archivos de salida almacenados en *Minio* de un trabajo concreto.

```
jobRoutes.get('/results/:id/:file', keycloak.protect(), async (req, res) => {
  const jobId = req.params.id;
  const file = req.params.file;

  try {
    res.set('Content-disposition', 'attachment; filename=' + file);
    res.set('Content-Type', lookup(file)?.toString());
    res.send(await getFile(jobId + '/' + file))

  } catch (err: any) {
    res.removeHeader('Content-disposition')
    res.removeHeader('Content-Type')
    res.status(404)
    res.send(err.message)
  }
});
```

Este método llamará a una función que leerá directamente del bucket de *Minio* en busca del archivo pedido.

```
export async function getFile(filePath: string): Promise<string> {
  try {
    return (await minioClient.getObject(config.MINIO_BUCKET, filePath)).read().toString();
  } catch (error: any) {
    throw new Error("El archivo solicitado no se ha encontrado")
  }
}
```

/api/deleteJob/:id

Permite al usuario borrar un trabajo específico.

```
jobRoutes.get('/deleteJob/:id', keycloak.protect(), async (req, res) => {
  const jobId = req.params.id;
  const token = req.headers.authorization;
  let message: any = await deleteJob(jobId, token);
  res.send(message)
});
```

Como Kafka no permite el borrado de trabajos concretos de la cola de trabajos la solución por la que se ha optado es la de ignorar el trabajo. Para ello simplemente se llama a un método que se encarga de encolar este trabajo dentro de una cola *ignore-jobs* para que cuando más tarde tenga que ser procesado por el Worker directamente se ignore.

4.2 Worker

El worker es un servicio que ofrece las siguientes funcionalidades:

- Leer mensajes de la cola de trabajos de Kafka.
- A partir de la descripción de los trabajos, clonar el repositorio que se indique.
- Insertar la configuración del trabajo en el sitio indicado en Nuestro Contrato.
- Lanzar a ejecución el trabajo.
- Obtener la salida, tanto estándar como de errores, de la ejecución.
- Actualizar el estado del trabajo, cuando este empieza a ser lanzado y cuando haya finalizado su ejecución.
- Guardar en Minio los diferentes archivos de salida.
- Calcular el tiempo de ejecución de cada trabajo.

Ahora pasaremos a ver algunos de los detalles de implementación más interesantes.

```
await consumer.run({
  eachMessage: async ({ topic, partition, message }) => {
    const job: Job = JSON.parse(message.value.toString());
    try {
      await launchJob(job);
    } catch (err: any) {
      const jobStatus: JobStatus = {
        id: job.id,
        status: config.FALLO,
        username: job.username,
        url: job.url,
        config: job.config,
        args: job.args,
      }
      updateJobStatus(jobStatus);
      console.error(err)
    }
  },
})
```

A modo de resumen este sería el inicio del Worker, se trata de un consumidor de Kafka, el cual para cada mensaje que le llega llama a *lauchJob*, utilizando *await* nos aseguramos de que nos esperamos a que un trabajo acabe de lanzarse, antes de escuchar nuevos mensajes. En caso de que se produjera algún error, se actualizaría el estado del trabajo indicando que se ha producido un error.

El método *launchJob* básicamente se encarga de medir el tiempo que tarda en ejecutarse el trabajo y de actualizar el estado del trabajo según lo que le devuelva *executeJob*.

```
export async function launchJob(job: Job): Promise<JobStatus> {
  const startTime = performance.now()

  const status = await executeJob(job);

  const endTime = performance.now()
  const elapsedTime = (endTime - startTime) / 1000;

  status.elapsedTime = elapsedTime;

  updateJobStatus(status)
  return status
}
```

Del método *executeJob* veremos sólo las partes más importantes:

En primer lugar, antes de ejecutar el trabajo, se comprueba que este no tenga ninguna petición de eliminación del mismo, además se comprueba que quien pide no ejecutar el trabajo sea la misma persona que quien añadió el trabajo.

```
if (checkIfJobIsDeleted(job)) {
  jobStatus.status = "Eliminado"
  return jobStatus
}
```

```
function checkIfJobIsDeleted(job: Job): boolean {
  return jobsToIgnore.filter(ignoredJob => ignoredJob.id === job.id && ignoredJob.username === job.username).length !== 0
}
```

Una vez que sabemos que sí que queremos lanzar el trabajo, es momento de actualizar el estado de “En espera” a “Lanzado”, esto quiere decir que ocurrirá lo siguiente:

Primero se hará una limpieza de las carpetas donde el proyecto se clonará, simplemente por evitar que por algún casual la carpeta exista, posteriormente, se procederá a clonar el repositorio indicado, crear la carpeta *output* y poner en el sitio correspondiente la configuración del trabajo. Una vez realizado todo esto, estamos en disposición de ejecutar el trabajo, esto lo haremos mediante *npm run start* y este es el momento en el que se pasarán los argumentos indicados en el trabajo.

```
execSync(`rm -rf ${projectFolder}`)
execSync(`rm -rf ${tempProjectFolder}`)
execSync(`mkdir -p ${tempProjectFolder}/output`)
const clone = execSync(`mkdir -p ${config.WORKER_DATA_FOLDER};
                        cd ${config.WORKER_DATA_FOLDER}; pwd;
                        git clone ${job.url} ${projectFolder};`);
jobStdout += clone;
const cd = execSync(`cd ${projectFolder};`)
jobStdout += cd;
const mkdirOutput = execSync(`mkdir ${projectFolder}/output`)
jobStdout += mkdirOutput;
writeFile(`${projectFolder}/config.json`, job.config?.toString());

const result = execSync(`cd ${projectFolder} && npm run start -- ${job.args}`)
```

Finalmente, se escribirán los archivos de logs, se subirán a *Minio* los archivos de la carpeta output y se eliminará la carpeta del proyecto.

```
writeLogs(tempProjectFolder, jobStdout, jobStderr);
await uploadOutputFilesToMinio(projectFolder, tempProjectFolder, job, jobStatus);

jobStatus.responseTime = Date.now().toString();
execSync(`rm -rf ${projectFolder}`)
```

Aunque no se ha mostrado, por simplificar, en todo momento se ha hecho uso de cláusulas try/catch, las cuales, nos permiten controlar que las diferentes fases se completan de forma correcta y en caso de haber algún error poder dar el trabajo por fallido y poder actualizar el estado del mismo en el sistema.

4.3 Observer

El observer es un servicio sencillo, aunque vital para el correcto funcionamiento del sistema, y es que nada más y nada menos, es el encargado de gestionar el tamaño del mismo, es decir, es quien controla la elasticidad.

Básicamente este servicio se encarga de que cada cierto tiempo (*REFRESH_RATE*), a partir de los estados de los trabajos, que ha capturado durante ese periodo de tiempo, obtiene algunas estadísticas para generar una Observación la cuál contiene lo siguiente:

- avgRateOfArrival -> Cantidad de trabajos que se han añadido en el periodo.
- avgRateOfService -> Cantidad de trabajos que han pasado de estado “En espera” a “Lanzado”.
- avgRateOfFinished -> Cantidad de trabajos que han pasado de estado “Lanzado” a estado “Finalizado”.
- avgResponseTime --> Tiempo medio de los trabajos terminados en el periodo.

```
Observation {
 avgRateOfArrival: number;
 avgRateOfService: number;
 avgResponseTime: number;
 timestamp: number;
}
```

Una vez obtenidos estos datos ya estamos en disposición de ejecutar el algoritmo que implementa nuestra estrategia de elasticidad el cual tiene el siguiente aspecto:

```
async function elasticityStrategy(observation: Observation) {
  const pendienteLanzar = observation.avgRateOfArrival - observation.avgRateOfService
  const numWorkers = (await kafka.admin().describeGroups(['worker-group'])).groups[0].members.length

  console.log({pendienteLanzar, numWorkers});

  if (pendienteLanzar > numWorkers) {
    console.log(`Create ${pendienteLanzar} new Workers`)
  }

  if (numWorkers > 1 && (numWorkers-pendienteLanzar) > 0) {
    console.log(`Delete ${numWorkers-pendienteLanzar} Workers`)
  }
}
```

A modo de resumen, la estrategia trata de ver cuántos trabajos están en la cola de Kafka esperando a ser consumidos por un Worker. Cuando el número de trabajos pendientes es mayor al número de Workers simplemente se crearían nuevas réplicas de los mismos. En el caso contrario, en el que el número de Workers es mayor al número de trabajos en espera, se daría la orden de eliminar los Workers que se encuentran en desuso.
Como se puede ver, la lógica que se comunica con la plataforma Kumori no ha sido programada por limitaciones temporales.

5. Despliegue

El despliegue ha consistido de dos partes principalmente.

La primera de ellas consistía en hacer el despliegue en Docker mediante *docker-compose.* Dado que este entorno se ha utilizado en el entorno de desarrollo y no en el de producción, se ha optado por realizar un único *stack* de servicios y no dos como se pedía originalmente para así poder simplificar los despliegues durante el desarrollo.

Por lo tanto en el *docker-compose.yml* del proyecto, se encuentran los diferentes servicios que componen nuestra solución. Para ello, se ha definido un servicio para cada uno de los contenedores que se necesitan, configurando sus variables de entorno de la forma correspondiente y utilizando volúmenes en aquellos contenedores que lo necesitaban como pueden ser *PostgreSQL*, *MongoDB* o *Minio*.

La segunda parte del despliegue consistía en realizar el mismo sobre una plataforma, en concreto, **Kumori**. Para realizar los despliegues en esta plataforma, se debe de seguir un proceso que es el que explicaremos a continuación.

El primer paso, es definir los diferentes **componentes (component)** que formarán la solución. Un componente, es la definición de un contenedor donde se indica la imagen del mismo, los puertos de enlace, tanto de entrada como de salida, variables de entorno, etc.

Una vez tenemos definidos los componentes, estos pueden pasar a componerse en servicios, donde cada uno de los componentes tendrá un rol específico, y además, se realizará una definición de qué roles pueden interactuar entre ellos y cuáles no.

En el **servicio (service)**, si se desea, y así es como lo hemos realizado, tambíen se puede definir un rol de tipo *Inbound* el cual, irá enlazado al rol *Frontend* de nuestro servicio, para así poder exponerlo a Internet y que este pueda ser accedido desde el exterior.

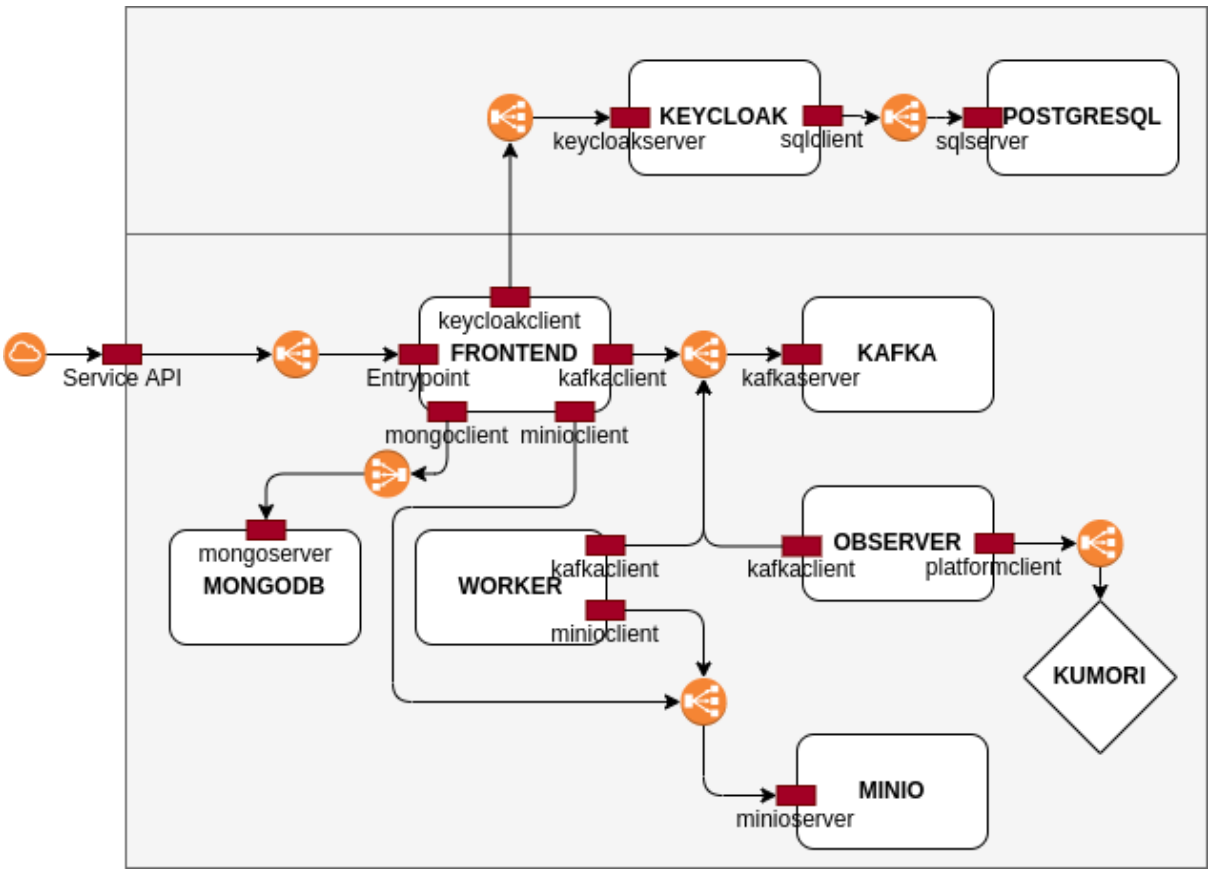
Por último, y una vez tenemos un servicio definido podemos pasar a desplegar este mediante un **despliegue (deployment)**.
Un despliegue, básicamente, consiste en un archivo que define cuáles serán los parámetros de configuración que deberán de utilizar cada uno de los roles dentro del servicio, así como indicar la cantidad de instancias de cada rol en el despliegue.

Una vez tenemos el despliegue, y si, sintácticamente está correcto y todos los recursos, como secretos y volúmenes existen en la plataforma, podremos dar la orden desde la herramienta *kumorictl* de hacer que se ponga en marcha el despliegue definido, levantando así los diferentes contenedores definidos en los roles del servicio que se despliega.

Como resultado de esta segunda forma de despliegue, se obtiene lo siguiente:

- 2 Deployments:
  - Uno de ellos para todo lo relacionado con la seguridad (*Keycloak*).
  - Y otro que representa la solución a desarrollar.
- 2 Services:
  - Uno define los roles utilizados en el despliegue de la seguridad.
  - Y otro que define los roles utilizados en el despliegue de la solución.
- 9 Components:
  - Fronted → API Rest que expone la funcionalidad del sistema.
  - Kafka → Cola de mensajes.
  - Keycloak → Seguridad.
  - Minio → Persistencia de resultados.
  - Mongo → Persistencia del estado de los trabajos.
  - Observer → Encargado de observar el estado del sistema y actuar según la carga del mismo.
  - Postgres → Base de datos para la persistencia de la configuración *Keycloak*.
  - Worker → Encargado de ejecutar los trabajos que se encuentran en *Kafka*.
  - Zookeeper → Servicio para el correcto funcionamiento de *Kafka*.

En la siguiente imagen, se puede observar de forma gráfica el despliegue realizado, donde en cada uno de los componentes se puede observar que endpoints tiene disponibles, así como sus nombres, para facilitar la “fontanería” entre cada uno de los componentes y el tipo de conexión que existe entre ellos, que en este caso, principalmente son balanceadores de carga.



6. Pruebas

Para comprobar el correcto funcionamiento del sistema, se han realizado pruebas unitarias sobre los diferentes componentes.

Para ello, se ha utilizado *Chai* y *Mocha*, librerías de testing para *NodeJs*, que permiten el testeo de aplicaciones de una forma sencilla y eficiente.

Para poder validar el correcto funcionamiento, se han realizado pruebas tratando de obtener la mayor cobertura de código posible.

7. Conclusiones

Llegados al final, vamos a analizar cuáles han sido los resultados que hemos obtenido basándonos en los objetivos que habíamos definido en un inicio.

1. Implementar los microservicios necesarios para cumplimentar la funcionalidad descrita.
    - Como se ha visto en la fase de implementación, se han implementado los servicios Frontend, Worker y Observer, los cuales, presentan un funcionamiento correcto y que además ofrecen funcionalidades bastante interesantes como la consulta de la carga del sistema, la capacidad de añadir o eliminar trabajos, así como ejecutar estos y, por supuesto, la recolección de los resultados.

2. Construir las imágenes de *Docker* necesarias para los microservicios.
    - De cada uno de los servicios se han generado las imágenes *Docker* necesarias mediante *Dockerfiles* para, posteriormente, poder desplegar estas en *Docker*. Además, estas han sido publicadas en un repositorio *Docker* público para poder hacer uso de las mismas sin la necesidad de construirlas desde el código fuente cada vez que se necesiten.

3. Preparar un *docker-compose* con un fichero *.env* para desplegar el sistema entero en una sola máquina.
    - En la parte de despliegue se ha comentado que se ha realizado esta configuración para facilitar el desarrollo y prueba de los diferentes componentes de la aplicación.

4. Desplegar la plataforma Kumori.
    - La segunda parte consistía en el despliegue de la solución en Kumori, que finalmente y tras varías dificultades encontradas, se ha podido hacer un despliegue exitoso, formado por dos servicios independientes los cuales son capaces de exponer a Internet toda la funcionalidad que se pedía. Se puede hacer uso de la solución desde https://proyectocc.vera.kumori.cloud.

5. Diseñar una estrategia de elasticidad.
    - En esta memoria, se ha diseñado una estrategía la cual permitiría cambiar la cantidad de instancias de Workers para así poder hacer frente a aumentos en la demanda del servicio.

6. Implementar la estrategia de elasticidad.
    - La implementación de la estrategia ha sido parcial. El encargado de esto es el Observer el cual tiene como función ver el estado del sistema y actuar en función de la carga del mismo. La estrategia de elasticidad ha sido implementada aunque realmente no ha sido conectada a la plataforma con lo que por el momento simplemente indica mediante logs, cuál sería la cantidad de instancias que se tendrían que crear o eliminar, todo esto, recordemos, que es en función de la carga de trabajo del sistema.

8. Trabajo Futuro

Como hemos visto una vez llegados aquí, hay parte del trabajo que podría evolucionar en un futuro y aquí enumeramos algunas funcionalidades que podrían implementarse en un futuro sobre la solución actual:

- Comunicar al Observer con la plataforma de despliegue para que realice las acciones necesarias para aumentar o disminuir el tamaño del sistema según se indique.
- Hacer una GUI desde la que observar de una forma más sencilla el estado de los trabajos, así como otras funcionalidades.
- Añadir una Base de datos en el worker que permita guardar ciertos estados para trabajos dependientes.
- Tener algún sistema de recolección de datos externos para que los trabajos puedan utilizarlos (*Injector*).

9. Anexo

9.1 Uso del proyecto (README.md)

Para probar el proyecto, puedes importar este workspace en Postman, así ya tendrás todas las peticiones preparadas para hacer uso del proyecto.
El primer paso, es ejecutar la petición de *login*, para ellos en el ejemplo existe el usuario *myuser* con contraseña *1234*. La petición devolverá un *Bearer Token*, el cual, tendrás que utilizar en el resto de peticiones para autenticarte.

Resumen de funcionalidad disponible:

- /health
  - / -> Indica si el servidor está en funcionamiento
  - /load -> Indica la carga del servidor
- /api
  - /addJob -> Añadir nuevo trabajo
  - /status -> Obtener estado de todos los trabajos del usuario
    - /status/:id -> Obtener estado del trabajo id del usuario
  - /results/:id/:file -> Obtener archivo file del trabajo id
  - /deleteJob/:id -> Eliminar trabajo de la cola de trabajos
- /auth
  - /:user/:password -> Inicio de sesión con usuario y contraseña, devuelve un *Bearer token*

/health

GET /

Response

response: <h1>Server is Running</h1>

GET /load

Devuelve la carga del sistema durante los últimos 10 periodos (20 segundos). Lista ordenada en orden inverso según timestamp

- avgRateOfArrival -> Devuelve la cantidad de trabajos que se han añadido en el periodo.
- avgRateOfService -> Devuelve la cantidad de trabajos que han pasado de estado *En espera* a *Lanzado*
- avgRateOfFinished -> Devuelve la cantidad de trabajos que han pasado a estado *Finalizado*
- avgResponseTime --> Devuelve el tiempo medio de los trabajos terminados en el periodo.

Request

headers:
  Authorization: Bearer yourBearerToken

Response

response: [
          {
            "avgRateOfArrival": number,
            "avgRateOfService": number,
            "avgRateOfFinished": number,
            "avgResponseTime": number,
            "timestamp": number
          },
          ...
        ]

/api

POST /addJob

Request

headers:
  Authorization: Bearer yourBearerToken
body: {
      "url": "https://username:password@github.com/your-username/your-repo.git",
           "args": "Arguments for your executable ## This string will be appended to your executable",
       "config": "Your custom config in JSON ## This content will be written to your root project folder on config.json"
    }

Response

response: {
          "jobId": "yourJobId"
        }

GET /status

Request

headers:
  Authorization: Bearer yourBearerToken

Response

response: [
          {
            "id": string,
            "status": string,
            "outputFiles": ["url://file1", "url://file2"]

```
    },
    {
      "id": string,
      "status": string,
      "outputFiles": ["url://file1", "url://file2"]
    },
    ...
  ]
```

GET /status/{jobId}

Request

headers:
  Authorization: Bearer yourBearerToken

Response

```
response: {
        "id": string,
        "status": string,
        "outputFiles": ["url://file1", "url://file2"]
      }
```

GET /results/{jobId}/{filename}

Request

headers:
  Authorization: Bearer yourBearerToken

Response

response: File Download

GET /deleteJob/{id}

Request

headers:
  Authorization: Bearer yourBearerToken

Response

response:
  `El trabajo ${jobId} ha sido eliminado.`
  o
  `El trabajo ${jobId} no existe o no te pertenece.`

/auth

GET /{username}/{password}

Response

response: {
"token": "yourBearerToken"
}
Definición de un trabajo

Cada trabajo necesita inicializar un proyecto npm porque necesitamos el archivo *package.json* para lanzarlo con npm run start. Esto no nos limita a solo lanzar proyectos Javascript/Typescript. Dentro del *package.json*, puedes definir cualquier comando a ejecutar. Por ejemplo:

```
# For a JS project you could start it as follows:
...
"scripts": {
  "start": "node src/index.js",
  ...
}
...

# For a Rust project you could start it as follows:
...
"scripts": {
  "start": "rustc main.rs && ./main.rs",
  ...
}
...
```

Como se explica en la definición de la API, si se pasa una configuración en la llamada a la API, esta, se escribirá en la carpeta raíz del proyecto como *config.json*.

Si el proyecto produce cualquier archivo de salida, tiene que ser escrito en el directorio de salida *output* en la raíz de tu proyecto. Todo lo que se guarde en esta carpeta será guardado en Minio para que así después se puedan obtener los resultados del trabajo.