UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Proyecto Laboratorio Cloud Computing

IMPLEMENTACIÓN DE UN FUNCTION AS A SERVICE

AUTORES: Alejandro Gálvez Ruiz, Victòria López Servera, Luca Pio Miano, José Serrano Díaz y Celia Solaz Vivó

PROFESOR: José Manuel Bernabeu Aubán

ASIGNATURA: Cloud Computing

Resumen:

Este articulo describe la implementación de una pequeña aplicación *Function as a Service* (FaaS) siguiendo las pautas de diseño propias de los servicios *Cloud*, como son el uso de microservicios contenerizados en imágenes *Docker* que interactúan entre ellos de manera reactiva a través de colas.

Palabras Clave:

Cloud, FaaS, NATS, API Rest, Oauth, Golang, Docker, Postman, Microservicios, Monitorización, Git

Abstract:

This article describes the implementation of a small application *Function as a Service* (FaaS) following the design guidelines of *Cloud* services, such as the use of microservices containerized in *Docker* images that interact with each other through reactive queues.

Key words:

Cloud, FaaS, NATS, API Rest, Oauth, Golang, Docker, Postman, Microservices, Monitoring, Git

Índice

1. Descripción del proyecto **6**

2. Tecnologías usadas en el proyecto **7**

3. Planteamiento del problema **9**

3.1. Diseño del sistema 9

3.2. Consideraciones 13

3.3. Reparto de tareas 13

4. Implementación de la solución. **15**

4.1. Creación del Oauth2-Proxy 15

4.2. Creación del NATS Server 15

4.3. Creación del Frontend 15

4.4. Creación del Worker 17

4.5. Creación del Observer 18

4.6. Creación del Injector 19

4.7. Contenerización 19

5. Guía de ejecución **20**

5.1. Entorno necesario 20

5.2. Despliegue del sistema 20

5.3. Prueba de la aplicación 20

5.3.1. Generar Cookies 20

5.3.2. Añadir la cookie en Postman 22

5.3.3. Crear Tarea 24

5.3.4. Obtener el estado de la tarea 25

5.3.5. Obtener el resultado de la tarea 26

5.3.6. Obtener el estado del sistema 26

5.3.7. Obtener todas las tareas enviada por un usuario 27

5.4. Pruebas 28

5.4.1. Pruebas de integración 28

5.4.2. Pruebas de congestión 28
5.5. Replegar el sistema 28

6. Mejoras a implementar **29**

7. Conclusión y discusión **30**

8. Anexo **31**

9. Bibliografía **32**

1. Descripción del proyecto

La finalidad del proyecto consiste en la realización de una pequeña aplicación *Function as a Service* (FaaS). Estos son una categoría de servicios de computación dentro de los servicios *serverless* que consiste en la ejecución de, típicamente, funciones de corta duración y que no mantienen en ningún caso el estado de cada sesión.

Por lo tanto, se deberán seguir una serie de pautas propias de estos sistemas característicos de la Nube:

- **Microservicios**. El sistema estará dividido en múltiples microservicios.
- **Naturaleza reactiva**. La interacción entre los microservicios será reactiva a través de eventos.
- **Colas**. La comunicación entre los microservicios se realizará a través de colas.
- **Contenerización**. Los microservicios se desplegarán a través de contenedores independientes con entornos propios.

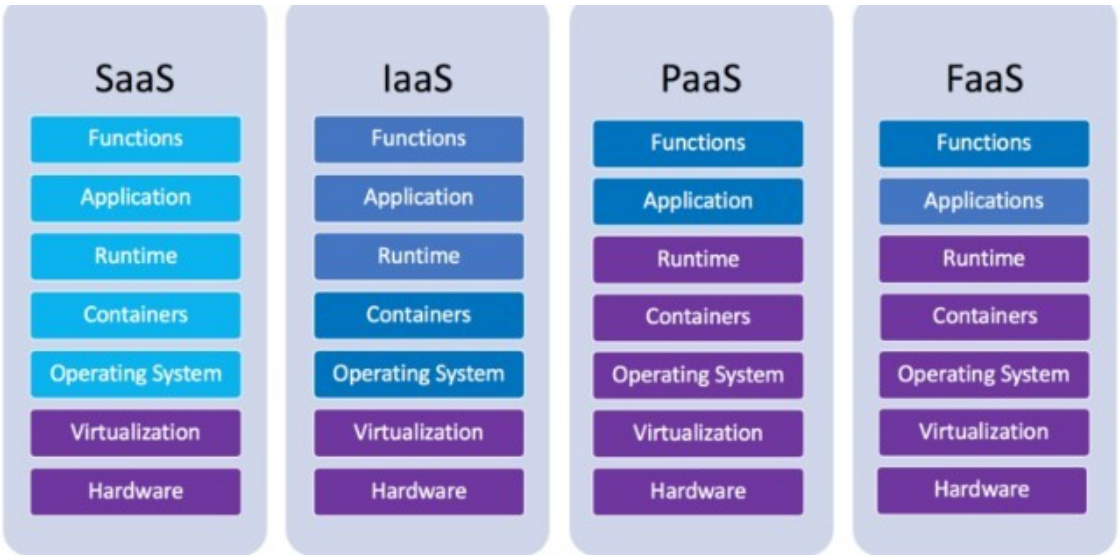

Figura 1: Comparación del *FaaS* con el resto de modelos de servicios *Cloud* [4]

2. Tecnologías usadas en el proyecto

- **Lenguaje de programación: Golang.** Es un lenguaje de programación relativamente reciente y desarrollado por Google que ofrece numerosas ventajas como:
  - **Concurrencia incorporada.** Ofrece soporte nativo para la concurrencia; esencial en servicios web.
  - **Seguridad.** Ofrece distintos mecanismos de seguridad que ayudan a prevenir la vulnerabilidad de nuestro sistema.
  - **Compilación rápida y fácil despliegue.** Ofrece la posibilidad de compilar un único ejecutable de rápida creación y distribución, sin depender de librerías externas.
  - **Gran ecosistema de herramientas.** Ofrece un gran ecosistema de herramientas y bibliotecas para no depender de librerías y/o *frameworks* de terceros.
- **Sistema de mensajería: Nats**. Es un sistema de mensajería de código abierto que está formado principalmente por dos partes:
  - ***Core NATS***. Ofrece las funcionalidades básicas de un sistema de mensajería como el uso de colas. Éstas permiten aplicar un patrón genérico de *PUB/SUB* donde unos Productores publican mensajes que son consumidos por unos *Consumidores* que están suscritos a dicha cola.
  - ***NATS con JetStream***. Añade una capa de persistencia adicional, como:
    - ***KV Buckets***. Almacenes de clave-valor.
    - ***Buckets* de Objetos**. Almacenes de objetos tales como ficheros.
- **Sistema de contenedores: Docker**. Es una plataforma que permite crear aplicaciones contenerizadas en entornos ligeros, portables y preparados con las herramientas necesarias para su puesta en marcha. De esta manera conseguimos crear

entornos aislados que facilitan tanto el desarrollo de cada microservicio como su despliegue y escalado.

3. Planteamiento del problema

3.1. Diseño del sistema

El diseño del sistema, que se puede comprobar en la Figura 17, consta de una serie de clases, librerias y microservicios.

En cuanto a las clases, se han ideado las siguientes:

- ***Task***.
  - *TaskId. UUID.*
  - *UserID. String (facilitado por OAuth).*
  - *RepoURL. String.*
  - *Parameters. []String*
  - *Status.*
- ***Status***. Puede tener distintos valores.
  - 100 - *PENDING.* La tarea está pendiente de ejecutar.
  - 200 - *EXECUTING.* La tarea se está ejecutando.
  - 300 - *FINISHED.* La ejecución de la tarea ha terminado.
  - 400 - *FINISHED_ERRORS.* La ejecución de la tarea ha terminado pero con errores.
  - 500 - *UNEXPECTED_ERRORS.* Un error inesperado ha ocurrido con la tarea.
- ***Result***.
  - *TaskId. UUID.*
  - *Files. [ ]String*(Ficheros con los resultados).
  - *TimeElapsed. Time.*

- ***RequestInjection***.
  - *File_name. String.*
  - *File_Content. [ ]bytes.*

En cuanto a las librerías, se han organizado en función de su finalidad:

- **TaskManager.** Contendrá las funciones encargadas de gestionar las tareas, como crearlas, obtener sus resultados, etc...
- **QueueManager.** Contendrá las funciones para interactuar directamente con la cola, como suscribirse a una cola, sacar los mensajes de una cola, etc...
- **StoreManager.** Contendrá las funciones encargadas de interactuar con los elementos persistentes de nuestro sistema, i.e. *KV Store y Object Store*.
- **InjectorManager.** Contendrá las funciones encargadas de gestionar las peticiones de inyección de ficheros del *Inyector* en un *bucket*.

Por otro lado, los microservicios son los siguientes:

- **Oauth.** Con el objetivo de proteger el servicio, se ha incluido un *oauth2-proxy* que realizará la autenticación de los usuarios, que una vez autenticados serán redirigidos al *Frontend*.
- **Servidor de NATS.** Será el encargado de correr el servicio de NATS. Este ofrecerá las colas de mensajería:
  - **Cola de Tareas.** Permite el envío de tareas entre el *Frontend y el Worker*.
  - **Cola del Injector.** Permite el envío de archivos entre el *Frontend y el Injector*.
  - **KV Buckets de usuarios.** Cada usuario tiene un *KV Bucket* donde se almacenarán el estado de sus tareas. Además, el número de *key-values* creados en este almacén nos ayudará a determinar un posible límite de tareas activas[1]. Pasado ese tiempo, todo rastro de tarea se eliminará y el cliente tendrá un nuevo *slot* disponible.

[1]Definimos tarea activa como toda aquella que lleva un tiempo específico creada

- **KV Bucket del *Observer***. Almacena información relevante sobre el estado del sistema para que pueda ser consultado desde el *frontend*.
- **Object Store para tareas.** El resultado de cada tarea es almacenado en un *Object Store* propio, de un número limitado de tamaño.

- **Worker:** Encargado de ejecutar las tareas que recibe directamente desde la *Cola de tareas*. La ejecución de dichas tareas se llevará a cabo de la siguiente forma:

  1. Clonar el repositorio de la tarea.
  2. Marca la tarea en el *KV Store* como ejecutando.
  3. Instala los módulos especificados en el *go.mod*.
  4. Ejecuta el archivo *main.go* con los parámetros especificados.
  5. Guarda las salidas y posibles errores en un *Object Store* creado específicamente para esa tarea.
  6. Marca la tarea en el *KV Store* como acabada.
  7. Elimina cualquier fichero/directorio creado.

- **Observer.** Saca una serie de métricas en función del estado del sistema y las almacena de modo que en cualquier momento, un administrador del sistema pueda comprobarlas a través del *frontend* y tomar las medidas oportunas necesarias.

- **Injector.** Permite a los administradores añadir ficheros potencialmente necesarios para nuestro sistema en un *bucket* del *Object Store*. Si bien en el escenario actual de nuestro sistema este componente no es necesario, se implementará por si lo fuera en una futura ampliación del servicio.

- **Frontend.** Servidor *HTTP* que recibe las peticiones *REST* de los clientes. La *API* ofrece las siguientes posibilidades:

  - **Crear una tarea.** Crea una tarea encolándola en una Cola de Tareas para que sea ejecutada por los *Workers*.
    - Inputs:
      - *Repositorio público* con el código a ejecutar.

- ◇ *Parámetros* necesarios para su ejecución.

* Outputs:
  - ◇ *ID* de la tarea creada.

- **Obtener el estado de una tarea.** Consulta en el *KV Store* el estado de una tarea.
  * Inputs:
    - ◇ *ID* de la tarea a consultar.
  * Outputs:
    - ◇ *Estado* de la tarea.
- **Obtener el resultado de una tarea.** Obtiene desde el *Object Store* los ficheros de salida de la tarea ejecutada.
  * Inputs:
    - ◇ *ID* de la tarea.
  * Outputs:
    - ◇ *Resultados* de la tarea.
- **Obtener una lista de los trabajos creados por un usuario.** Consulta en el *KV Store* todos los trabajos activos de un usuario.
  * Inputs:
    - ◇ *ID* del usuario.
  * Outputs:
    - ◇ *Lista de IDs* de los trabajos.
- **Obtener información recogida del *Observer.*** Consulta en el *KV Store* del *Observer* el estado del sistema. (Admin).
  * Outputs:
    - ◇ *Logs*
    - ◇ Estado del sistema.
- **Inyectar ficheros potencialmente necesarios**. Inserta en un bucket del *Object Store*, ficheros necesarios en nuestro sistema.(Admin).

    - Inputs:
        - Fichero.

3.2. Consideraciones

Hay una serie de atributos de nuestro sistema que deberán ser seleccionados en función de los requisitos del mismo, tal y como el número de tareas a crear por cada usuario y su tiempo de disponibilidad. A falta de mayor información sobre este ámbito, se han establecido una serie de valores predeterminados::

- El número máximo de tareas activas que puede tener un usuario es 80.
- Una tarea permanecerá activa desde el momento en que se crea hasta pasados 20 minutos, pasado ese tiempo se eliminará todo rastro de ella.
- El tiempo máximo que podrá ejecutarse una tarea son 20 segundos.

Todos estos valores se han *seteado* a través de variables de entorno de modo que pueden ser rápidamente modificadas en caso de que los requisitos del sistema cambien.

3.3. Reparto de tareas

Hecho el planteamiento de nuestro sistema, sus componentes y sus funcionalidades, se ha realizado el reparto de las diferentes tareas a llevar a cabo por cada miembro. La creación del *OAuth2 Proxy* y el *Servidor de NATS* será realizada conjuntamente por todos los miembros del grupo. Sin embargo, para el resto de microservicios se han realizado dos subgrupos:

- Alejandro Gálvez Ruiz y José Serrano Díaz. Son los encargados de realizar el *Worker y el Inyector.*
- Celia Solaz Vivo, Victòria López Servera y Luca Pio Miano. Son los encargados de realizar el *Frontend y el Observer.*

Como últimas consideraciones, debido a incompatibilidades de horario por causas externas Celia acabó realizando alguna tarea en el *Worker* mientras que Alejandro hizo lo propio sobre el *Frontend.*

4. Implementación de la solución.

4.1. Creación del Oauth2-Proxy

Siguiendo los pasos citados en la documentación de *Oauth2-Proxy* [1] debemos primeramente crear las credenciales para que Google pueda autenticar los usuarios a través del *Oauth2-Proxy*.

Para ello necesitamos ir a la *Google Cloud Console* y crear unas credenciales estableciendo las *URI de redireccionamiento autorizados*, que consistirá en las rutas a la que se podrá redireccionar a los usuarios autenticados. Hecho esto, Google nos facilitará un *ID* y *Secreto de Cliente* que usaremos en nuestro proxy junto con otra serie de valores:

```
environment:
  OAUTH2_PROXY_UPSTREAMS: "http://frontend:8080"
  OAUTH2_PROXY_CLIENT_ID: 936638621086-ukfhb28p7pkbjup42pgvjq4br9ce0h86.apps.googleusercontent.com
  OAUTH2_PROXY_CLIENT_SECRET: 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
  OAUTH2_PROXY_COOKIE_SECRET: _-vsNPgMZKXBRGWutZsIgeyB9Dzx-4aRNQwfX-zuDyo=
  OAUTH2_PROXY_HTTPS_REDIRECT: "false"
  OAUTH2_PROXY_EMAIL_DOMAINS: "*"
  OAUTH2_PROXY_PROVIDER: "google"
  OAUTH2_PROXY_REDIRECT_URL: "http://localhost:4180/oauth2/callback"
```

Figura 2: Configuración Oauth2-Proxy en Docker-Compose

4.2. Creación del NATS Server

La creación de este servicio es en general muy sencilla, pues sus propios desarrolladores ofrecen una imagen ya configurada [2], y la creación de colas y *buckets* se hace de manera dinámica durante la ejecución del sistema.

Por lo tanto únicamente hemos activado *JetStream* y la monitorización a través de variables de entorno.

4.3. Creación del Frontend

Para la creación del servidor se ha utilizado el *framework Gin* y se han definido las siguientes rutas para gestionar diferentes tipos de solicitudes HTTP:

- **/createTask.** Método *POST* para crear tareas, acepta como parámetros de entrada el URL del repositorio Git y los parámetros de ejecución.
    - Llama a la función postTask de la librería TaskManager, se encarga de comprobar que el cuerpo de la REQUEST es válido, revisa si el usuario esta autorizado y controla que no se haya superado el número de peticiones por usuario. Actualiza el estado de la tarea guardado en el Bucket del usuario a PENDING y encola la tarea. Por último, incrementa una variable guardada en el Bucket del Observer que contiene el número de mensajes totales que han sido enviados a la cola.
    - Si no hay errores devuelve el ID de la tarea en formato JSON y código 201.
- **/getTaskStatus.** Método *GET*, devuelve el *estado de una tarea*, acepta como parámetro el *ID de la tarea*.
    - Llama a la función getTaskStatus de la librería TaskManager que obtiene el estado de la tarea almacenada en el Bucket de dicho usuario.
    - Si no hay errores devuelve el Estado de la tarea en formato JSON y código 200.
- **/getAllTasks.** Método *GET*, devuelve el listado de *IDs de las tareas* que tiene el usuario en la cola.
    - Llama a la función getAllTasks de la librería de TaskManager y obtiene del *bucket* del usuario un listado de todos las tareas almacenadas.
    - Si no hay errores devuelve un listado de id de tareas en formato JSON y código 200.
- **/getSystemStatus.** Método *GET* que devuelve el *estado del sistema.*
    - Llama a la función getSystemStatus de la librería TaskManager controla que el usuario esté autorizado para sacar información del sistema. Accede al Bucket del Observer y extrae el estado del sistema y el número de Workers que se encuentran actualmente en ejecución. Saca del mismo Bucket los Logs producidos por el Observer relacionados con cambios en el estado del sistema.

- Si no hay errores devuelve el estado, el número de workers, los logs y código 200.

- **/injectFile.** Método *POST* para inyectar ficheros, acepta como parámetro de entrada un fichero.
  - Llama a la función PostInjection, que controla que el usuario este autorizado para realizar esta acción, lee el fichero y lo guarda en el Bucket del inyector, para la siguiente elaboración.
  - Devuelve codigo 200

Además se ha configurado un *middleware CORS* para que se pueda acceder al servidor desde cualquier origen y se ha establecido la conexión con el servidor NATS.

4.4. Creación del Worker

El funcionamiento del *Worker* es el siguiente:

1. Se conecta al *Servidor NATS.*
2. Espera a recibir tareas.
   a) Cuando llega una tarea por la *TaskQueue*, ejecuta una función *handleRequest()* que:
      1) Crea un directorio temporal para la tarea.
      2) Clona el repositorio en dicho directorio.
      3) Establece el estado de la tarea en *EXECUTING.*
      4) Ejecuta la tarea como un usuario con permisos limitados.
         a′ Primeramente haciendo un *'go mod download'.*
         b′ Seguido de un *go run main.go <params>.*
      5) Establece el estado de la tarea en *FINISHED.*
      6) Almacena el resultado de la tarea.
      7) Limpia los directorios temporales creados.

En caso de cualquier error durante alguno de los pasos descritos en el apartado a), el sistema gestionará el error creando un fichero de resultado que especifica dicho error, y estableciendo el estado de la tarea en *FINISHED_ERRORS*.

En caso de que algún error más ocurra durante esta gestión de fallos, el sistema declarará la tarea como inválida estableciendo su estado en *UNEXPECTED_ERROR* y volverá a esperar tareas de la cola.

4.5. Creación del Observer

Este componente se ocupará de gestionar el estado de congestión de la cola NATS. Al ser un aplicación distribuida hay que tener en cuenta la carga de trabajo sobre los diferentes componentes, en particular la cola de NATS donde se almacenan los trabajos a ejecutar por el *worker*.

El *observer* se ocupará de notificar al cliente el estado de congestión de la cola de NATS, y a la vez levantar otros *workers* para reducir la cargo sobre la misma. El funcionamiento es el siguiente: cada 5 segundos el *observer* lanza un *thread*. Este *thread* recupera desde el *KV store* el número de mensajes escritos y leídos en la cola de trabajos. Dichas variables son escritas por, respectivamente, el *frontend*, cada vez que llegue una *createTask*; y por el *worker*, cada vez que coja desde la cola un trabajo para ejecutarlo.

Se mide la diferencia entre *outMsgs y inMsgs* y se ejecuta el algoritmo:

- Si la diferencia es mayor que un límite (actualmente de 10) entonces el sistema estará a punto de congestionarse, si esta situación se verifica durante tres veces seguidas entonces el sistema está seguramente congestionado, por lo cual se incrementa el número de *workers* de uno.
- Si la diferencia es menor que $\frac{límite}{3}$ y el número de *workers* es > 3, entonces el sistema no está congestionado y se disminuye el número de *workers*.
- En los demás casos el sistema no está congestionado y no se hace nada.

Para cada evento que se verifique, excluyendo los casos en los que no se hace nada, el *observer* generara un log, guardado en un *KV store*, y actualizará las variables

number_of_workers con el número actual de *workers y status* con el estado del sistema {*congested, OK*}. Estos logs tienen una retención de 1 día, por lo que cada día se borran para crear nuevos.

El estado del sistema, con los logs producidos, son accesibles a través del *frontend* por la llamada */getSystemStatus*.

4.6. Creación del Injector

Este microservicio trabaja de forma similar al *Worker*:

1. Se conecta al *Servidor NATS*.
2. Espera a recibir peticiones de inyección.
   - *a*) Cuando llega una petición, ejecuta la función *handleRequest()* que:
     - 1) Almacena el fichero en el *bucket del injector*.
       - *a'* Si dicho *bucket* no existe, lo crea y vuelve a intentar.

4.7. Contenerización

Como ya se ha comentado, para la contenerización y puesta en marcha del sistema se ha utilizado *Docker*.

En los casos del *Oauth2-proxy y NATS Server* se optó por utilizar las imágenes ya creadas de dichos servicios, directamente configurables a través de sus variables de entorno, como se ha comentado previamente.

Para el resto de microservicios, se creó un *Dockerfile* general que partía de una imagen *Debian* y realizaba una configuración necesaria por todos los contenedores, como instalar una versión de *Golang* específica.

Hecho esto, cada microservicio hace uso de esta imagen creada para realizar las configuraciones específicas necesarias (como la creación de un usuario con menos permisos por parte del Worker), copiar una *build* de su código y ejecutarlo.

Todo esto se ha automatizado a través de un *Docker-compose* que nos facilitará la comunicación entre los contenedores, así como lanzar más de una copia del mismo microservicio, como el *Worker* que por defecto estará replicado tres veces.

5. Guía de ejecución

5.1. Entorno necesario

Para la ejecución del proyecto es necesario disponer de una instalación de *Docker* y los ficheros almacenados en el repositorio Git, https://github.com/vlopser/CC.

5.2. Despliegue del sistema

Se ha preparado un *Makefile* que se encargará de la compilación del código y la construcción de imágenes a través de *Docker-compose*. Por lo tanto, sitúese en el directorio raíz del proyecto y ejecute *make*.

Automáticamente en 30-60 segundos tendrá la aplicación lista para recibir peticiones.

5.3. Prueba de la aplicación

Para probar el sistema puede utilizar los *endpoints del Frontend* a través de distintas herramientas, en nuestro caso explicaremos como hacerlo a través de *Postman*.

5.3.1. Generar Cookies

En un escenario ideal, nuestro sistema contaria con un *frontend web* al que redirigiría las peticiones el *Oauth-Proxy*. Sin embargo, debido a que únicamente se ha implementado el *backend*, debemos iniciar sesión primeramente en dicho *Oauth* e inyectar la *cookie* que nos proporciona directamente en las llamadas *HTTP*. En este apartado se explica cómo crear dicha *cookie* para ejecutar las llamadas a los servicios.

Desde navegador web conectarse al URL http://localhost:4180, a continuación se abrirá una pantalla para hacer la verificación con *Oauth2-Proxy* (Fig. 3):

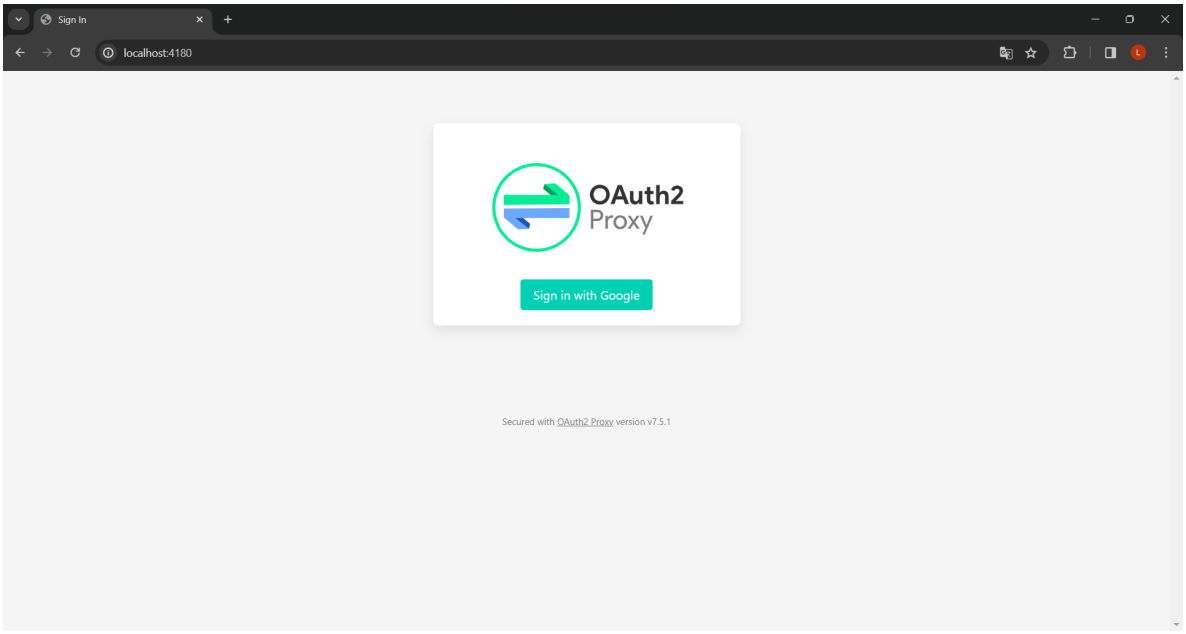

Figura 3: Autenticación con Oauth2-Proxy

Puedes iniciar sesión con cualquier cuenta de *gmail*, pero nosotros hemos añadido una como administrador:

- **Correo electrónico** = ccupv2023@gmail.com
- **Contraseña** = administrador2023

Al final del proceso saldrá una pantalla negra mostrando '*404 page not found*', pero no se trata de un problema debido a que igualmente nuestra *cookie* con el token *Oauth2-proxy* habrá sido creada. Para recuperar dicho token se puede hacer de dos maneras, según el navegador web utilizado:

- Chrome:
  - Hacer clic derecho en la misma página web, seleccionar *Inspeccionar* e ir a la pestaña *Application* (Aplicación).
  - Navegar a la sección *Cookies*:
    - Dentro de las Herramientas de Desarrollo, buscar la pestaña *Application* (Aplicación) en la parte superior.

    - En el panel izquierdo, buscar la sección *Cookies* bajo la subsección *Storage*. Seleccionar la *cookie* para http://localhost:4180, copiarla y guardar el valor de la *cookie* llamada *Oauth2proxy*.

- Firefox:

    - Hacer clic derecho en la misma página web, seleccionar *Inspeccionar* e ir a la pestaña *Almacenamiento*. En el panel izquierdo buscar la sección *Cookies*.
    - Seleccionar la *Cookie* para http://localhost:4180, copiarla y guardar el valor de la *Cookie* llamada *Oauth2-proxy*.

En el siguiente apartado explicaremos cómo debemos de añadir la *cookie* en *Postman*.

5.3.2. Añadir la cookie en Postman

1. Abrir *Postman*.
2. Pulsar sobre el botón *Cookies*

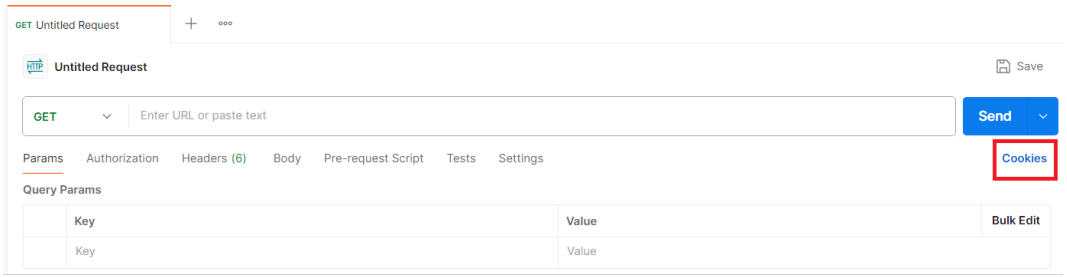

Figura 4: Botón para acceder a la creación de la *cookie* en *Postman*

3. Se abrirá una nueva ventana donde escribir el nombre que tendrá la *cookie*, en nuestro caso *oauth2-proxy*. Pulsaremos sobre el botón *Add domain*.

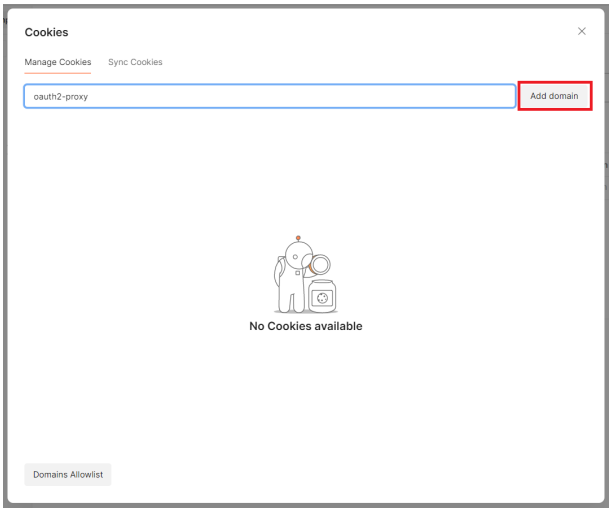

Figura 5: Nueva ventana con el botón *Add domain*

Obteniendo el siguiente resultado:

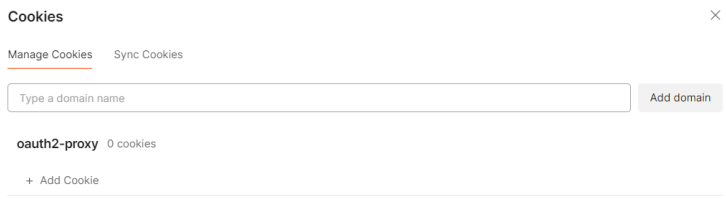

Figura 6: Resultado de añadir el nombre de nuestra *cookie*

4. Pulsaremos sobre *Add Cookie* y debemos de cambiar los siguientes parámetros:

   - El valor (*value*) y el nombre de la *cookie* que encontramos en el navegador web.

Figura 7: Primer argumento a cambiar

   - El parámetro *path* no necesita ser modificado.

Figura 8: Segundo argumento de la *cookie*.

- La fecha de expiración, *Expires*, la obtenemos de la *cookie* de nuestro navegador web.

Figura 9: Tercer argumento, fecha de expiración.

5. Ahora nos queda guardar la *cookie*, para ello pulsaremos sobre *Save*.

Figura 10: Guardamos la *cookie*.

Una vez tenemos la *cookie* en *Postman* podemos pasar a realizar la llamada a nuestro *frontend* de manera autenticada.

5.3.3. Crear Tarea

1. Llamar al *endpoint* desde Postman: http://localhost:4180/createTask.

2. Incluir en el cuerpo del *JSON* la siguiente estructura:

```
{
    "url" : "{git_url}",
    "parameters" : ["{param_1}", "{param_2"} ... "{param_n}"]
}
```

3. Pulsar *Send* para enviar la tarea y comprobar que el servidor responda correctamente (código 200).

4. Apuntar el *taskId* para las siguientes llamadas *APIs*.

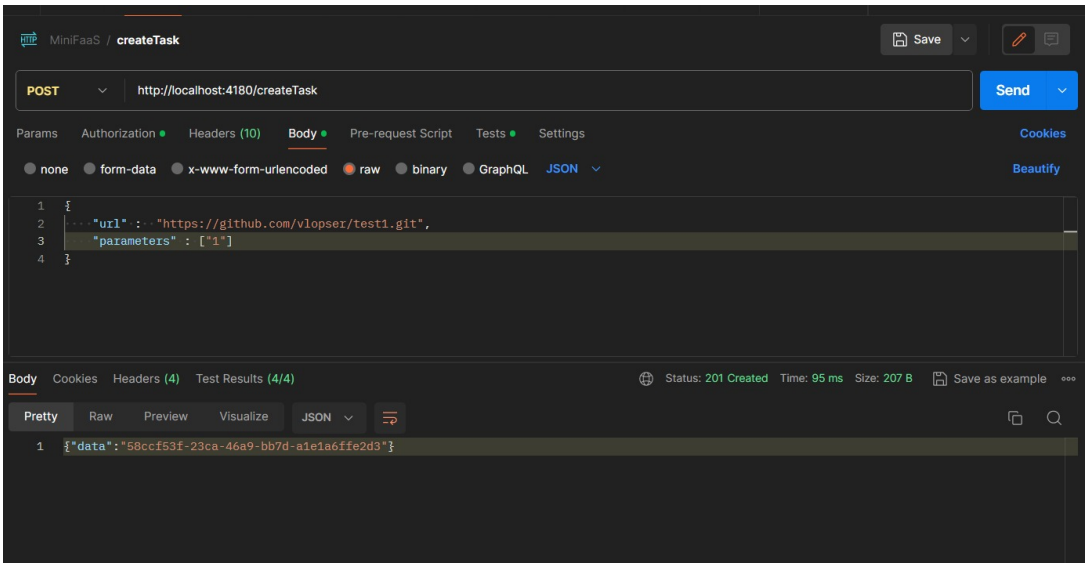

Figura 11: Crear tarea

5.3.4. Obtener el estado de la tarea

1. Llamar al *endpoint*: http://localhost:4180/getTaskStatus?taskId={{taskId}}

2. Substituir el parámetro *taskId* con el anteriormente apuntado.

3. Pulsar *Send* y comprobar que la llamada devuelva el estado de la tarea y código 200.

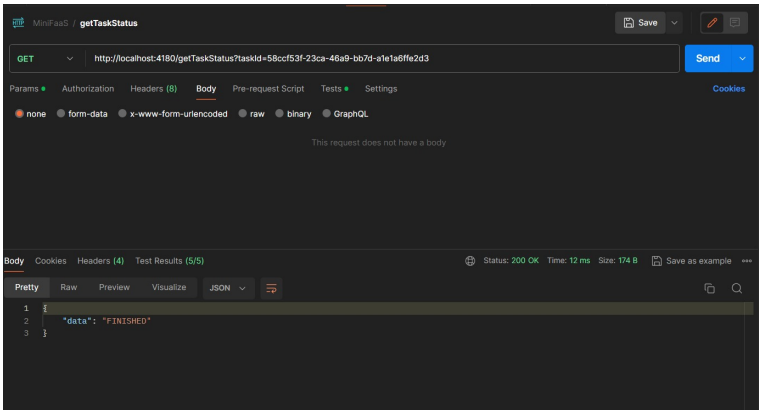

Figura 12: Obtener el estado de una tarea

5.3.5. Obtener el resultado de la tarea

1. Llamar al *endpoint*: http://localhost:4180/getTaskResult?taskId={{taskId}}
2. Substituir el parámetro *taskId* con el apuntado anteriormente.
3. Pulsar *Send* y comprobar que la llamada devuelva el resultado de la tarea y código 200.

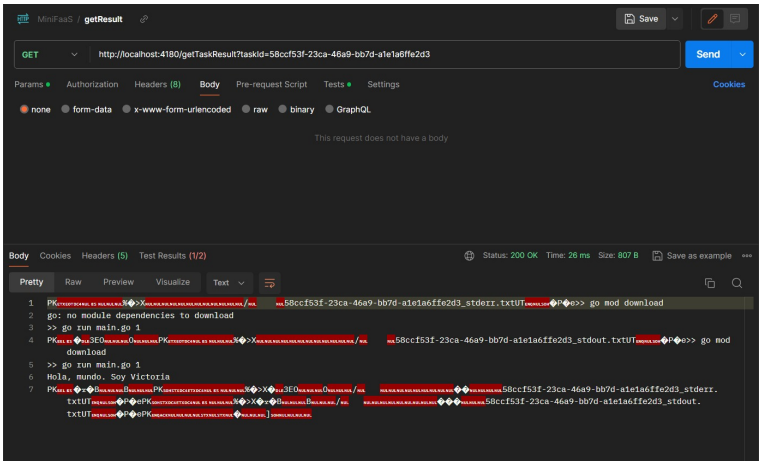

Figura 13: Obtener el resultado de una tarea

El resultado esperado es un fichero *ZIP*, por lo que es mejor ejecutar la llamada desde el propio navegador web.

5.3.6. Obtener el estado del sistema

1. Llamar al *endpoint*: http://localhost:4180/getSystemStatus.

2. Pulsar *Send* y comprobar que la llamada devuelva un *JSON* con el estado del sistema y código 200.

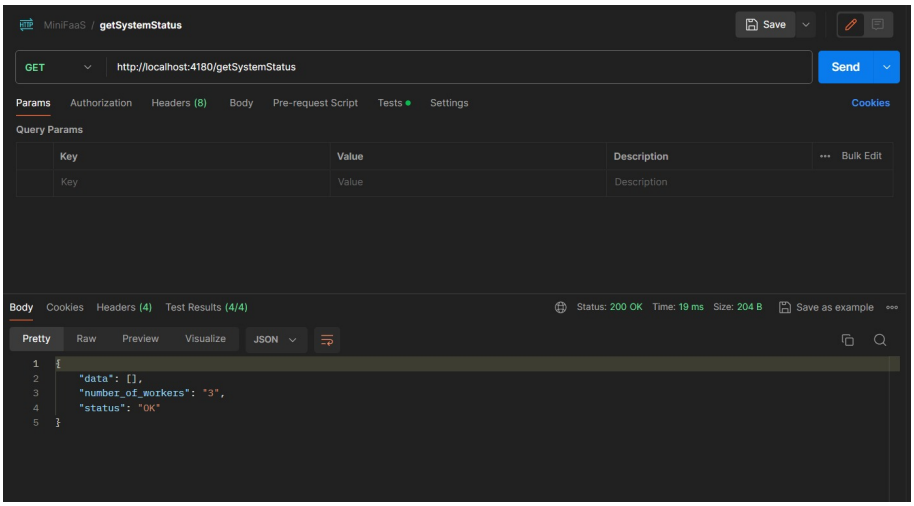

Figura 14: Obtener el estado del sistema

5.3.7. Obtener todas las tareas enviada por un usuario

1. Llamar al *endpoint*: http://localhost:4180/getAllTasks

2. Pulsar *Send* y comprobar que la llamada devuelva un *JSON* con un listado de todas las tareas enviada por el usuario y código 200.

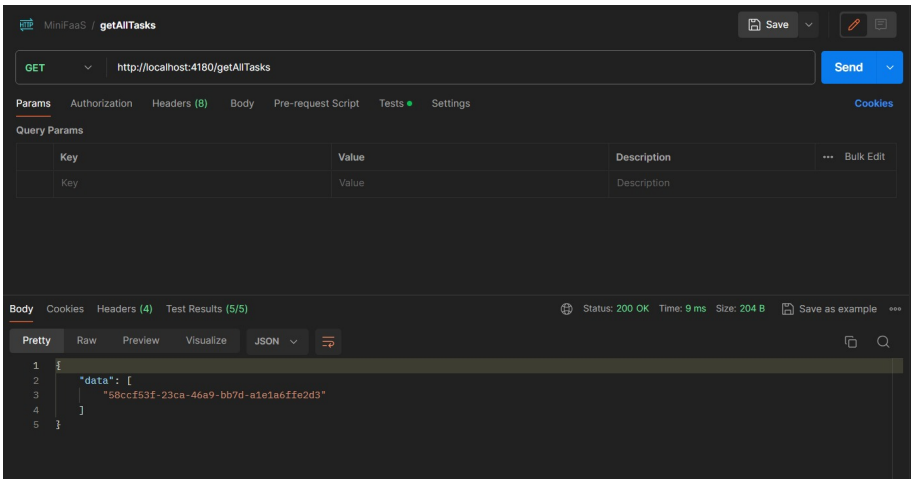

Figura 15: Obtener todas las tareas del Usuario

5.4. Pruebas

5.4.1. Pruebas de integración

Se ha incluido una *Collection* en *Postman* que puede importar y que realiza una serie de pruebas para comprobar el correcto funcionamiento del sistema. Para ello, deberá actualizar la variable *token* con el valor de la *cookie* proporcionada por el *Oauth Proxy*.

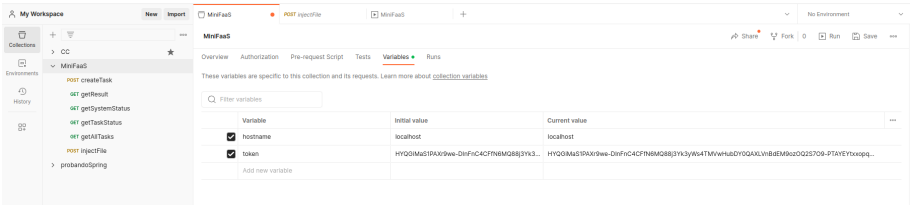

Figura 16: Token con la *cookie* proporcionada

A continuación puede lanzar dicha *collection* siguiendo los pasos que se incluyen en la documentación de *Postman* [3]. [2]

5.4.2. Pruebas de congestión

Para testear la creación de nuevos *workers* se ha proporcionado un *script.sh* que se ocupará de llamar la API *createTask* unas cuantas veces de manera que se simule la congestión del sistema. Para ello, ubicarse en la carpeta del proyecto y ejecutar el comando *./src/services/observer/test/script.sh <numero_de_ejecuciones> "<cookie_de_usuario>"*, especificando el número de ejecuciones deseadas, se sugiere 80. Puede comprobar a través del *endpoint /getSystemsStatus*, que el sistema se encuentra congestionado. Tras el paso de unos segundos, el sistema volverá a un estado de *OK*.

5.5. Replegar el sistema

De la misma forma, puede liberar los recursos creados mediante el comando *make down*.

[2] Para que el */injectFile* funcione debe adjuntar un fichero

6. Mejoras a implementar

Existen una serie de mejores que se podrían aplicar al sistema en futuras versiones.

- **Escalado dinámico.** Si bien el *observer* toma una serie de métricas, y en función de ellas es posible escalar o desescalar el sistema manualmente, un punto muy favorable sería migrar la plataforma a un gestor de contenedores como *Kubernetes* que nos permita, entre otras cosas, escalar la arquitectura dinámicamente.

- **Añadir más opciones de ejecución para las tareas.** Actualmente el sistema únicamente admite ejecuciones para repositorios con *Golang 1.21.6.* Un gran añadido al proyecto sería permitir más personalización en la versión de *Golang* a utilizar e incluir nuevos lenguajes de programación aceptados.

- **Más seguridad en las ejecuciones.** Si bien se han tomado consideraciones para una ejecución segura de las tareas mediante el uso de usuarios con menos privilegios. Una mejor aproximación hubiera sido el uso de espacios completamente aislados, como *namespaces*, donde ejecutar cada tarea.

7. Conclusión y discusión

En este estudio, hemos destacado la importancia crítica de la arquitectura de microservicios en la nube respaldada por la tecnología de contenerización Docker, específicamente en el contexto de un proyecto basado en Functions as a Service (FaaS). La adopción de microservicios ha demostrado ser esencial para la modularidad, facilitando el desarrollo, mantenimiento y escalabilidad de servicios de manera individual. La elección estratégica de Docker como plataforma de contenerización ha permitido la implementación consistente y aislada de estos microservicios, asegurando un despliegue eficiente y portabilidad en diversos entornos. En conclusión, la sinergia entre microservicios y Docker no solo ha optimizado el rendimiento operativo del proyecto FaaS, sino que también sienta las bases para futuras investigaciones en la continua evolución de las tecnologías en la nube.

8. Anexo

Repositorio GitHub: https://github.com/vlopser/CC

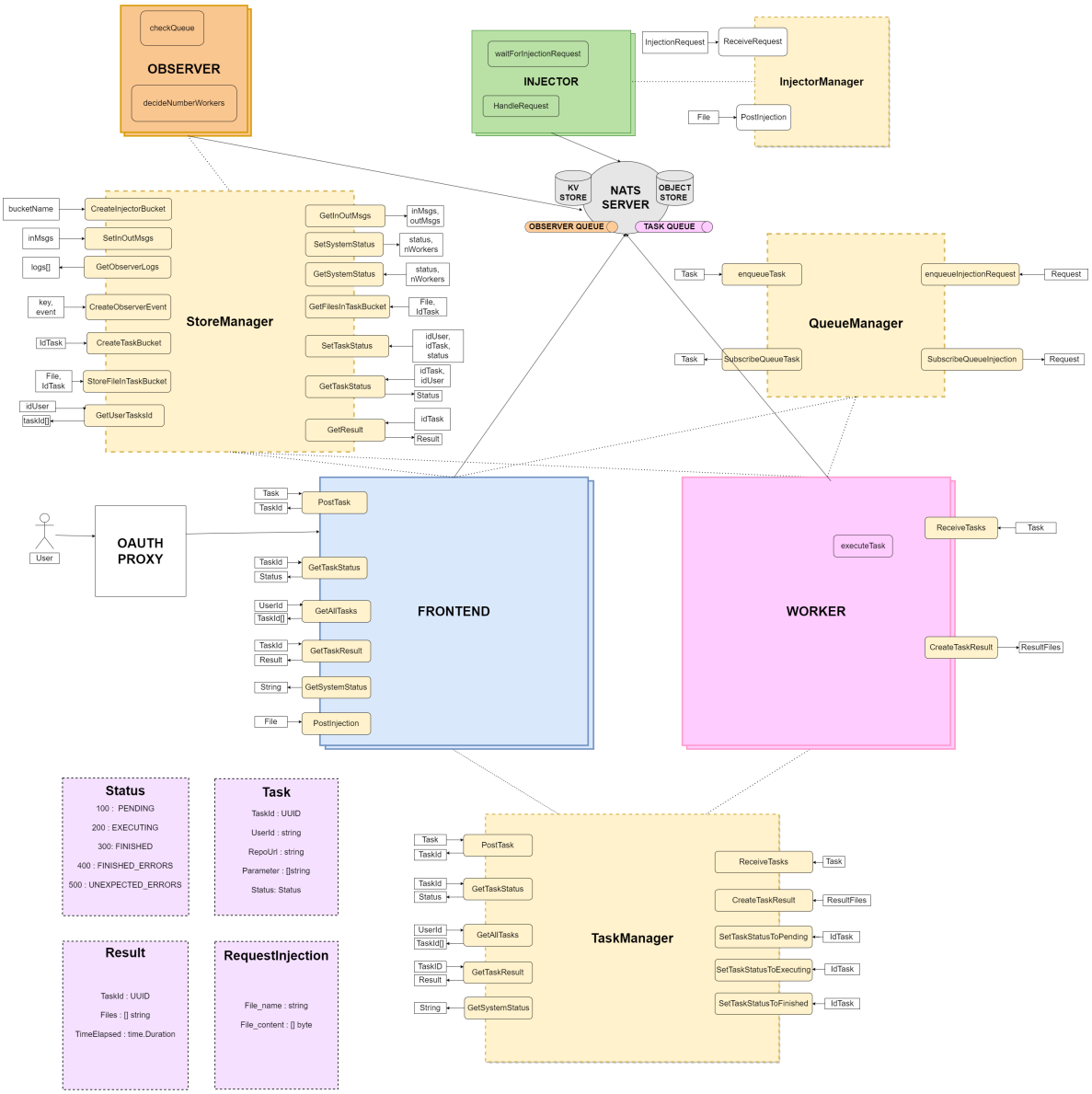

Figura 17: Diagrama del servicio en la nube con sus respectivos microservicios

9. Bibliografía

[1] Installation - oauth2 proxy - oauth2-proxy.github.io. https://oauth2-proxy.github.io/oauth2-proxy/.

[2] Nats and docker - docs.nats.io. https://docs.nats.io/running-a-nats-service/nats_docker.

[3] Run a collection using the postman cli - postman learning center - learning.postman.com. https://learning.postman.com/docs/postman-cli/postman-cli-run-collection/.

[4] What is function-as-a-service (faas)? - linkedin - linkedin.com. https://www.linkedin.com/pulse/what-function-as-a-service-faas-ipspecialistofficial.