GRADO EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

IMPLEMENTACIÓN DE UNA ARQUITECTURA DE MICROSERVICIOS PARA UNA RED DE SENSORES IOT SOBRE ARDUINO

Autor: Roberto Gesteira Miñarro

Director: Atilano Ramiro Fernández-Pacheco Sánchez-Migallón

Madrid

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título

IMPLEMENTACIÓN DE UNA ARQUITECTURA DE MICROSERVICIOS PARA UNA RED DE SENSORES IOT SOBRE ARDUINO

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el

curso académico 2019/20 es de mi autoría, original e inédito y

no ha sido presentado con anterioridad a otros efectos.

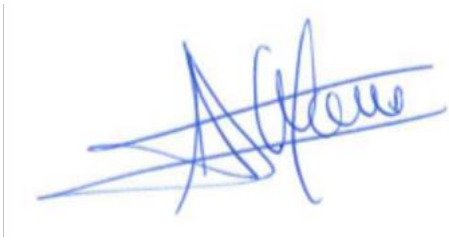
El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido

tomada de otros documentos está debidamente referenciada.

Fdo.: Roberto Gesteira Miñarro Fecha: 01/07/2020

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO



Fdo.: Atilano Ramiro Fernández-Pacheco Sánchez-Migallón Fecha: 01/07/2020

GRADO EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

IMPLEMENTACIÓN DE UNA ARQUITECTURA DE MICROSERVICIOS PARA UNA RED DE SENSORES IOT SOBRE ARDUINO

Autor: Roberto Gesteira Miñarro

Director: Atilano Ramiro Fernández-Pacheco Sánchez-Migallón

Madrid

Agradecimientos

A mis familiares, amigos y profesores, por apoyarme y estar siempre a mi lado.

IMPLEMENTACIÓN DE UNA ARQUITECTURA DE MICRO-SERVICIOS PARA UNA RED DE SENSORES IOT SOBRE ARDUINO

Autor: Gesteira Miñarro, Roberto.
Director: Fernández-Pacheco Sánchez-Migallón, Atilano Ramiro.
Entidad Colaboradora: ICAI – Universidad Pontificia Comillas

RESUMEN DEL PROYECTO

En este proyecto se analiza la viabilidad de utilizar una arquitectura de microservicios para gestionar una red de sensores conectados a una placa de Arduino. Para ello, se ha desarrollado una aplicación web de domótica que se apoya en una arquitectura de microservicios desplegada en Kubernetes para comunicarse con los sensores.

Palabras clave: IoT, microservicios, Kubernetes, Docker, Arduino, *Cloud*.

1. Introducción

Los mundos de la tecnología y el *software* están en continua evolución. Los avances en la electrónica analógica y digital han favorecido a la domótica y al IoT. Por otro lado, el surgimiento de *Cloud Computing* y sus servicios IaaS, Paas y SaaS han cambiado por completo los modelos de negocio de las empresas, siendo más conveniente contratar proveedores *cloud* que implementar sus propios servicios [1].

2. Definición del proyecto

Este proyecto está motivado por las nuevas tecnologías de IoT y *Cloud Computing*. En los servicios *Cloud* se emplean técnicas de virtualización, apoyándose de máquinas virtuales y contenedores. Una manera muy innovadora de desplegar aplicaciones es mediante contenedores, precisamente, ya que son muy eficientes en cuanto al uso de recursos.

Las arquitecturas de microservicios están pensadas para ser pequeñas aplicaciones desplegadas en un *cluster*, que se comunican por medio de la red. Así, se diferencian de las arquitecturas monolíticas en el hecho de que la funcionalidad completa está distribuida en distintos microservicios, cada uno con una parte de esa funcionalidad global [2], como se muestra en la Fig. 1.

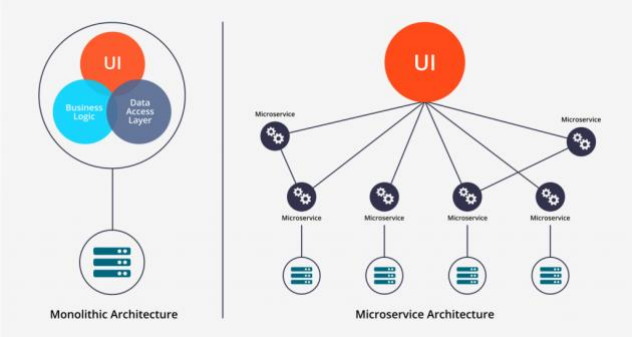


Fig. 1. Comparación entre arquitectura monolítica y microservicios [3]

Docker es una herramienta utilizada para la creación de contenedores, mientras que Kubernetes es un orquestador de contenedores. Kubernetes se encarga de desplegar, detener y escalar estos contenedores según indiquen una serie de archivos conocidos como manifiestos que se añaden a Kubernetes.

Los objetivos de este proyecto son:

- Implementar una arquitectura de microservicios mediante Docker y Kubernetes.
- Montar una red de sensores IoT con una placa de Arduino.
- Desarrollar un *dashboard* con Angular para gestionar los sensores.

3. Descripción del sistema

Lo más importante del sistema desarrollado es la arquitectura de microservicios. Como se puede ver en la Fig. 2, se han implementado 10 microservicios (la placa de Arduino no es un microservicio). Cada uno de estos presenta una función concreta e independiente en cierta medida de los otros microservicios. Además, se pueden ver las comunicaciones establecidas entre microservicios.

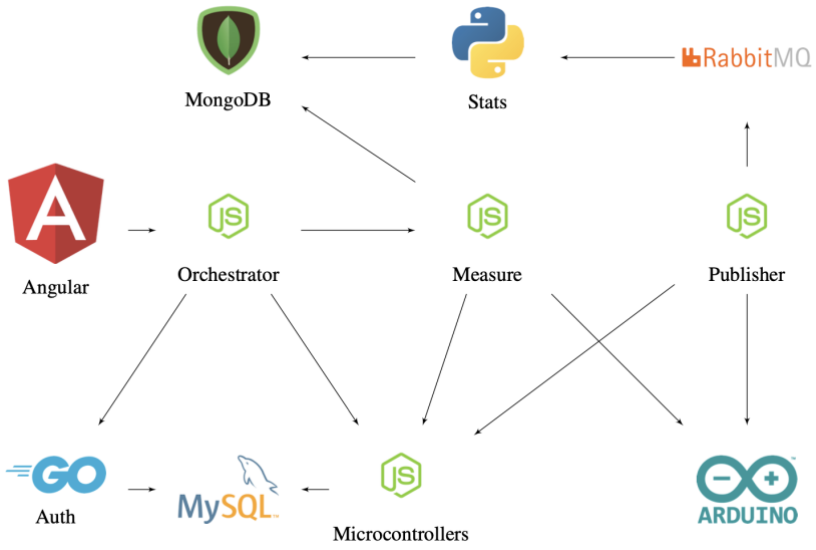


Fig. 2. Arquitectura de microservicios desarrollada

A modo de resumen, las tareas de los microservicios son:

- Angular: Aplicación web con el *dashboard* del usuario.
- Orchestrator: API *gateway*, entrada a la arquitectura de microservicios.
- Auth: Autenticación de usuarios en el sistema.
- Microcontrollers: Gestión de los microcontroladores de los usuarios.
- Measure: Recogida de medidas de los sensores de los usuarios.
- Publisher: Publicación de medidas en colas de mensajes de RabbitMQ.
- RabbitMQ: Bróker de mensajes encargado de gestionar colas de mensajería.
- Stats: Cálculo de estadísticas con las medidas de los mensajes de las colas.
- MySQL: Almacenamiento de usuarios y microcontroladores.
- MongoDB: Almacenamiento de estadísticas de las mediciones tomadas.

En la placa de Arduino se han utilizado un sensor de temperatura y un sensor de humedad de la tierra; y se ha simulado una bombilla inteligente mediante un LED.

4. Resultados

Los resultados obtenidos han sido satisfactorios, ya que se han logrado los objetivos propuestos al comienzo del proyecto.

Además, se ha construido una arquitectura de microservicios con Docker y Kubernetes, aprovechando al máximo todas las ventajas que proporciona respecto de una arquitectura monolítica.

El uso de diferentes lenguajes de programación y la independencia existente entre microservicios hace que esta arquitectura sea muy potente para desarrollo de aplicaciones en equipos de desarrollo de software, ya que facilitan el trabajo en equipo, la división de tareas y el uso de distintas tecnologías y metodologías.

Por otro lado, se han realizado pruebas unitarias y de integración y pruebas manuales a los microservicios desarrollados, obteniendo así un software de mejor calidad. Además, se utilizaron herramientas de monitorización para visualizar los recursos consumidos por cada microservicio en producción.

5. Conclusiones

Una vez finalizado el proyecto, y en base a los resultados obtenidos, se concluye que es viable utilizar una arquitectura de microservicios para el caso de uso de gestionar una red de sensores de domótica.

Además, se han añadido distintas funcionalidades a la aplicación para demostrar que realmente facilita el desarrollo el hecho de utilizar microservicios, sobre todo a la hora de desplegarlos y de agregar nuevas funciones o servicios.

6. Referencias

[1] *La importancia del Cloud Computing en las empresas*, 2018. [En línea]. Disponible en: https://www.beservices.es/importancia-cloud-computing-empresas-n-5317-es (Accedido: 26-06-2020).

[2] K. B. Roland Barcia y R. Osowski, *Guía de Microservicios. Punto de vista*, IBM. [En línea]. Disponible en: https://www.ibm.com/downloads/cas/5O8YOPKN (Accedido: 09-06-2020).

[3] *Arquitectura de microservicios: qué es, ventajas y desventajas*. [En línea]. Disponible en: https://decidesoluciones.es/arquitectura-de-microservicios/ (Accedido: 25-06-2020).

IMPLEMENTATION OF A MICROSERVICES ARCHITECTURE FOR AN IOT SENSOR NETWORK OVER ARDUINO

Author: Gesteira Miñarro, Roberto.
Supervisor: Fernández-Pacheco Sánchez-Migallón, Atilano Ramiro.
Collaborating Entity: ICAI – Universidad Pontificia Comillas

ABSTRACT

In this project, the feasibility of using a microservices architecture to manage a sensor network connected to an Arduino board is analyzed. For this task, a home automation web application has been developed. This webapp is based on a microservices architecture deployed in Kubernetes to communicate with the sensors.

Keywords: IoT, microservices, Kubernetes, Docker, Arduino, Cloud.

1. Introduction

The worlds of technology and software are in continuous evolution. The advances in analog and digital electronics have favored home automation and IoT. On the other hand, business models have been completely changed due to the rise of Cloud Computing and its IaaS, PaaS and SaaS services, making it more convenient to hire cloud providers rather than implementing their own services [1].

2. Project definition

This project is motivated by new technologies, such as IoT and Cloud Computing. Virtualization techniques are commonly used in cloud services, using virtual machines and containers. A very innovative way to deploy applications is through containers, precisely, since they are extremely efficient in terms of the use of resources.

Microservices architectures are intended to be small applications deployed in a cluster, which communicate through the network. Thus, they differ from monolithic architectures due to the fact that the complete functionality is distributed in different microservices, each of them with a single part of that global functionality [2], as it is depicted in Fig. 1.

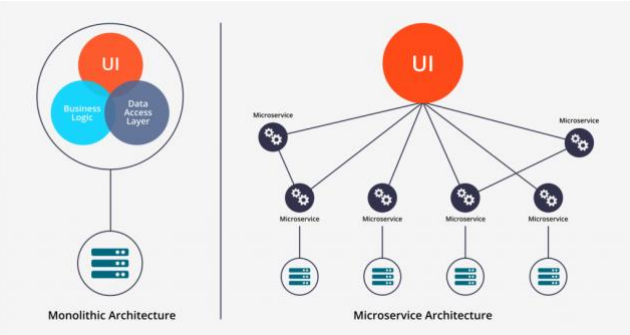


Fig. 1. Comparison between monolithic architecture and microservices [3]

Docker is a tool used for creating containers, while Kubernetes is a container orchestrator. Kubernetes is responsible for deploying, stopping, and scaling these containers as indicated by a series of files known as manifests that are added to Kubernetes.

The objectives of this project are:

- Implementing a microservices architecture using Docker and Kubernetes.
- Building an IoT sensor network with an Arduino board.
- Developing a dashboard with Angular to manage the sensors.

3. System description

The most important part of the developed system is the microservices architecture. As shown in Fig. 2, a total of 10 microservices have been implemented (the Arduino board is not a microservice). Each of these has a specific function and to some extent independent of the other microservices. In addition, the communications established between microservices are also depicted in the diagram.

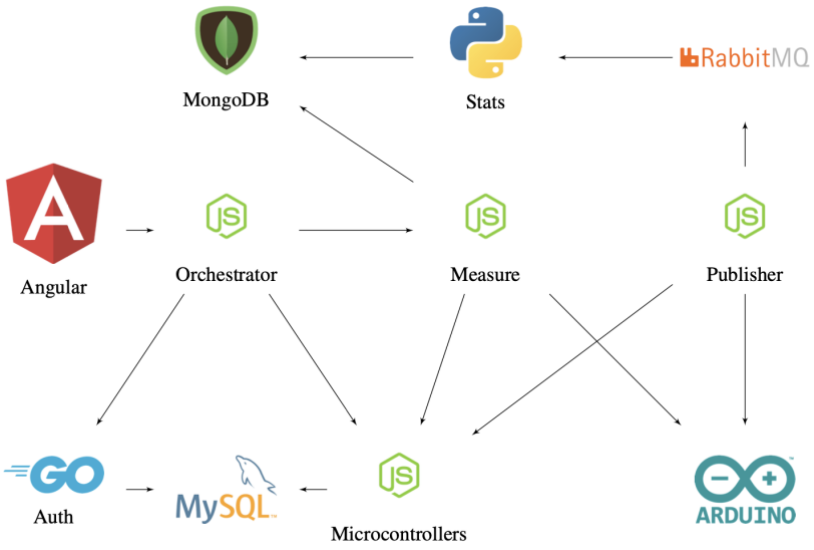


Fig. 2. Developed microservices architecture

In summary, the tasks of the microservices are:

- Angular: Web application containing the user's dashboard.
- Orchestrator: API gateway, entry to the microservices architecture.
- Auth: User authentication in the system.
- Microcontrollers: Management of users' microcontrollers.
- Measure: Collection of measurements from users' sensors.
- Publisher: Publishing measures in RabbitMQ message queues.
- RabbitMQ: Message broker in charge of managing message queues.
- Stats: Calculation of statistics with the measures of the messages in the queues.
- MySQL: Users and microcontrollers storage.
- MongoDB: Storage for statistics of measurements.

On the Arduino board, a temperature sensor and an earth humidity sensor have been used; and a smart light bulb has been simulated using an LED.

4. Results

The results obtained have been satisfactory, since the objectives proposed at the beginning of the project have been achieved.

In addition, a microservices architecture has been built with Docker and Kubernetes, taking full advantage of all the benefits it provides compared to a monolithic architecture.

The use of different programming languages and the independence existing between microservices makes this architecture highly powerful for application development in software development teams, since they facilitate teamwork, division of tasks and the use of distinct technologies and methodologies.

On the other hand, unit and integration tests and manual tests have been carried out on the developed microservices, thus obtaining high-quality software. In addition, monitoring tools were used to visualize the resources consumed by each microservice in production.

5. Conclusions

Once the project is completed, and based on the results obtained, it is concluded that it is feasible to implement a microservices architecture for the use case of managing a home automation sensor network.

In addition, different functionalities have been added to the application to demonstrate that using microservices really facilitates development tasks, especially when it comes to deploying and adding new functions or services.

6. References

[1] *La importancia del Cloud Computing en las empresas*, 2018. [Online]. Available: https://www.beservices.es/importancia-cloud-computing-empresas-n-5317-es (Accessed: Jun 26, 2020).

[2] K. B. Roland Barcia and R. Osowski, *Guía de Microservicios. Punto de vista*, IBM. [Online]. Available: https://www.ibm.com/downloads/cas/5O8YOPKN (Accessed: Jun 09, 2020).

[3] *Arquitectura de microservicios: qué es, ventajas y desventajas*. [Online]. Available: https://decidesoluciones.es/arquitectura-de-microservicios/ (Accessed: Jun 25, 2020).

Índice de la memoria

Índice de figuras

Índice de códigos

Índice de tablas

Capítulo 1

Introducción

Desde el nacimiento de Internet, las tecnologías web no han dejado de evolucionar. La utilización de servidores web, el uso de páginas y aplicaciones web y los lenguajes y paradigmas de programación han mejorado en gran medida para conseguir que los sistemas desarrollados sean más eficientes, robustos, seguros y escalables, entre otras características.

Por otro lado, el mundo de la domótica también ha avanzado mucho, debido principalmente a las mejoras de los circuitos electrónicos tanto analógicos como digitales, a la tecnología de *Internet of Things* (IoT) y a los algoritmos de *Machine Learning* e Inteligencia Artificial.

Todos estos cambios forman parte de la conocida como transformación digital. Este es un proceso que llevan a cabo las empresas para actualizar sus tecnologías, sus recursos y, en definitiva, sus modelos de negocio para no quedarse atrás respecto de la competencia en su sector. Además, estos cambios buscan proporcionar soluciones y valor añadido a las empresas, precisamente, con lo que obtendrán un mejor desempeño en el mercado [1].

Quizás, lo más destacado hoy en día es la utilización de los servicios de *Cloud Computing* para obtener soluciones de diversa naturaleza. Ejemplos de estos servicios en la nube abarcan desde un simple *software* como servicio hasta una compleja infraestructura con una gran cantidad de elementos conectados entre sí. Las principales ventajas que proporcionan servicios en la nube para las empresas tienen relación con la reducción de costes de mantenimiento, administración, energía y seguridad; ya que es el proveedor *cloud* quien se encarga [2].

CAPÍTULO 1. INTRODUCCIÓN

Actualmente, las empresas disponen de servidores, páginas web y aplicaciones. Esto se ha visto favorecido por el uso de las tecnologías *cloud* y de las arquitecturas de microservicios. Estos dos aspectos mejoran el desempeño de los equipos de desarrollo, obteniendo resultados de mejor calidad.

Uno de los principales conceptos que se implementa en la nube es la virtualización mediante máquinas virtuales y contenedores, con tecnologías como Docker y Kubernetes. Esto proporciona beneficios en cuanto al uso de recursos y la administración de sistemas.

En el presente Trabajo Fin de Grado se van a relacionar estos puntos anteriores entre sí para analizar la viabilidad de utilizar una arquitectura de microservicios para gestionar una red de sensores IoT con Arduino. Se desarrollará una aplicación web de domótica como caso de uso y se utilizará Docker y Kubernetes para construir la arquitectura de microservicios.

El proyecto está motivado por el uso de conceptos y tecnologías innovadoras que están en la vanguardia del mundo del desarrollo web y del desarrollo de *software*. Es un hecho el que estos nuevos paradigmas y métodos de despliegue de aplicaciones serán los predominantes en el futuro cercano, ya que aportan muchas ventajas a día de hoy.

En este documento, se explican en primer lugar las tecnologías utilizadas. Posteriormente, se desarrolla el estado del arte en la actualidad y la definición del trabajo realizado. Finalmente, se explica en detalle el sistema desarrollado junto con un análisis de los resultados obtenidos, conclusiones y posibles trabajos adicionales.

Como añadido, se incluyen una guía de instalación y un manual de usuario como anexos. Además, se incluye un tercer anexo con una breve reflexión sobre la importancia de los Objetivos de Desarrollo Sostenible y su relación con el proyecto desarrollado.

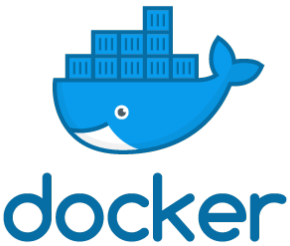
Capítulo 2

Descripción de las tecnologías

En este capítulo se describen las principales tecnologías que se usan en el presente Trabajo Fin de Grado. Se ha centrado la atención en Docker y Kubernetes, ya que son las herramientas más complejas del proyecto y de las que se hablará en gran medida en los próximos capítulos.

2.1. Docker

Docker es una tecnología de creación de contenedores que permite la creación y el uso de contenedores de Linux. Con este *software* se pueden utilizar los contenedores como máquinas virtuales extremadamente livianas. Además, se tiene la posibilidad de ejecutarlos, pararlos, modificarlos, copiarlos y moverlos a distintos entornos, tal y como dice la comunidad de Red Hat [3].



Más adelante, en la Sección 3.2, se explicará en detalle qué es un contenedor, para qué sirve y qué diferencias tiene respecto de una máquina virtual.

Para la comprensión de la presente sección basta con saber que un contenedor de Docker es un entorno virtual de Linux desde el que se pueden ejecutar comandos propios de la terminal de Linux.

Los contenedores de Docker son ampliamente utilizados en equipos de desarrollo de *software*, tanto en las fases de desarrollo como en la fase de producción. Por ejemplo, Docker permite que todos los desarrolladores dispongan del mismo entorno a la hora de programar, independientemente del sistema operativo que se use para escribir el código. Y no solo eso, también da la posibilidad de no tener que instalar todas las dependencias de un proyecto localmente, ya que estas se pueden instalar en un contenedor de Docker. Lo mismo pasa con instanciar una base de datos, esta se puede configurar en un contenedor de Docker, en vez de descargarla y configurarla en el ordenador que se utilice para codificar.

Por otro lado, Docker también se utiliza en producción. De este tema también se hablará en la Sección 3.4. El hecho es que los contenedores de Docker son muy ligeros y permiten ejecutar aplicaciones de manera muy sencilla. Además, con las nuevas tecnologías de integración continua y entrega continua (CI/CD) y las arquitecturas de microservicios, el utilizar contenedores de Docker en producción es mucho más viable que otras alternativas de despliegue de aplicaciones.

2.1.1. Imágenes de Docker

Utilizar un contenedor de Docker es como utilizar una máquina virtual dentro de un sistema operativo *host*. Esto da una flexibilidad increíble, sobre todo porque el entorno de Docker está aislado del sistema *host*, por lo que todo lo que se ejecute o instale en el contenedor permanecerá ahí hasta que se elimine.

En Docker existe un concepto denominado "imagen" que hace referencia a un modelo de contenedor, es decir, que una imagen de Docker representa un contenedor. Docker, como sistema de código abierto, provee un repositorio denominado Docker Hub en el cual se pueden subir, descargar y versionar imágenes de Docker.

Este repositorio de imágenes es público, e implementa un sistema de control de versiones de las imágenes. De esta manera, a las imágenes de Docker se les puede

asignar una etiqueta que indique su versión. Si no se indica etiqueta, por defecto se pondrá *latest*, asumiendo que es la versión más reciente.

En Docker Hub existen imágenes de Docker verificadas (como las imágenes de Ubuntu, MySQL, MongoDB, Node.js, NGINX, RabbitMQ o Python). De esta forma, al ejecutar un contenedor de una imagen de MySQL, por ejemplo, toda la configuración de MySQL como base de datos ya estará realizada por defecto. Así, el usuario/desarrollador no tiene que preocuparse de instalar paquetes mediante línea de comandos al iniciar el contenedor. En este sentido tienen importancia las versiones de las imágenes, ya que será distinto utilizar `mysql:8` que `mysql:5.7`. También sería posible emplear `mysql:latest`, pero no se considera buena práctica, ya que esta imagen puede ser actualizada con el tiempo y deje de funcionar algo si la versión de MySQL es determinante.

Una utilidad muy potente que ofrece Docker es la posibilidad de crear imágenes de Docker personalizadas. Esto se realiza mediante un archivo de texto denominado `Dockerfile`. Un ejemplo de este tipo de archivo puede verse en el Código 2.1.

Los `Dockerfile` consisten en una serie de instrucciones que se ejecutan secuencialmente desde un contenedor de Docker que se toma como base. En el ejemplo del siguiente Código 2.1, el contenedor base se crea a partir de la imagen de `node:12.18-alpine`, la cual contiene ya configurado lo justo y necesario para ejecutar programas utilizando Node.js (versión 12.18).

El sufijo `alpine` indica que el contenedor se ejecuta sobre Alpine Linux, una distribución de Linux muy ligera, simple y segura [4]. De esta manera, el contenedor es muy liviano y consumirá menos recursos. Se trata de una etiqueta bastante común en las imágenes de Docker Hub.

```
FROM node:12.18-alpine

WORKDIR /orchestrator-ms

COPY *.json ./
RUN npm install --production

COPY src src

CMD [ "npm", "start" ]
```

Código 2.1. Ejemplo de `Dockerfile`

Algunas instrucciones básicas de los `Dockerfile` son las siguientes:

- `FROM`: Suele ser la primera instrucción del `Dockerfile`. Indica la imagen del contenedor que se utilizará para ejecutar las siguientes líneas.

- `RUN`: Se utiliza para ejecutar un comando de la terminal de Linux. Es útil para instalar las dependencias de un proyecto (por ejemplo, en Node.js se utiliza `npm install`).

- `COPY`: Sirve para añadir archivos desde el *host* hasta un directorio del contenedor. Con esta instrucción se pueden añadir archivos de configuración o códigos fuente.

- `ADD`: Extiende la funcionalidad de `COPY`. Por ejemplo, permite añadir archivos desde una URL o descomprimir archivos comprimidos directamente en el contenedor. Docker recomienda utilizar `COPY` como buena práctica antes que `ADD` si no es estrictamente necesario.

- `CMD`: Esta instrucción única en el `Dockerfile` indica el comando que se ejecutará por defecto al iniciar el contenedor (una posibilidad para una aplicación de Node.js sería `npm start`).

- `ENV`: Con este comando se indican variables de entorno (clave-valor) que tenga que haber en el contenedor. No obstante, las variables de entorno también se pueden introducir al iniciar el contenedor, como se verá más adelante.

- `WORKDIR`: Se puede utilizar para cambiar el directorio de trabajo. A efectos de uso es como el comando `cd` de Linux.

El significado de estas instrucciones y otras más avanzadas puede consultarse en la documentación oficial de Docker, accesible en [5].

2.1.2. Contenedores de Docker

El proceso a seguir para ejecutar un contenedor personalizado es realizar un `Dockerfile`, construir la imagen de Docker y después ejecutar el contenedor con dicha imagen, como se muestra en la Fig. 2.1.

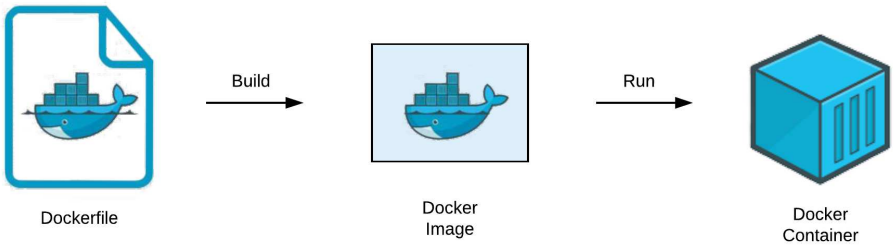


Fig. 2.1. Proceso de creación de contenedores personalizados

Para construir la imagen se debe ejecutar `docker build -t` *`nombre-imagen`* `.` desde el mismo directorio en el que se encuentra el archivo `Dockerfile` (indicado por el punto presente en el propio comando, que simboliza el directorio actual).

Existe una versión experimental de este comando que se utiliza para que la imagen construida funcione correctamente en distintas arquitecturas. Por ejemplo, se puede especificar que funcione en arquitecturas linux/amd64, linux/arm64, linux/arm/v7 y linux/arm/v6, entre otras. Para utilizar este comando es necesario activar las funcionalidades experimentales de Docker y realizar ciertas configuraciones del comando `docker buildx`.

Posteriormente, para usarlo se puede escribir el comando `docker buildx build --platform linux/amd64,linux/arm64 -t` *`nombre-imagen`* `.` desde el directorio del `Dockerfile`. La especificación de este comando está disponible en la documentación de Docker, accesible en [5].

El comando `docker buildx` se utiliza en este proyecto para que las imágenes de Docker construidas funcionen tanto en un ordenador (con arquitectura AMD) como en una Raspberry Pi 4 (con arquitectura ARM).

Una vez que la imagen se ha construido correctamente, se puede ejecutar `docker run -it` *`nombre-imagen`* para ejecutar un contenedor con la imagen construida. Si se quiere ejecutar una imagen de Docker verificada (como por ejemplo `ubuntu` o `mysql`), basta con poner este nombre en el comando anterior para ejecutar un contenedor con dicha imagen. Docker se descargará la imagen de Docker Hub si no la tiene ya descargada.

La línea de comandos de Docker ofrece muchas posibilidades a la hora de iniciar un contenedor. Por ejemplo, al ejecutar el comando `docker run` con la opción `-p` *`puerto-host`*`:`*`puerto-contenedor`* se puede configurar un mapeo de puertos entre el *host* y el contenedor de Docker para poder acceder al contenedor desde el *host*. Las opciones `-i` y `-t` se utilizan para que al ejecutar el contenedor con la imagen indicada en el comando se abra una consola de comandos interactiva conectada al contenedor. Otra opción interesante es `-e` *`clave=valor`*, que se utiliza para configurar variables de entorno en el contenedor.

Los contenedores de Docker están pensados para no persistir datos. De hecho, cada vez que se arranca un contenedor, este inicia siempre con las mismas instrucciones del `Dockerfile` con el que se construyó la imagen, a no ser que no se borre el contenedor, lo cual no es una práctica habitual. No obstante, antes se dijo que Docker es útil para instanciar bases de datos como MySQL o MongoDB.

Existe otro término relevante en Docker denominado "volumen". Esto es un sistema de archivos que se monta en el contenedor y se mapea con un directorio del *host*. De esta manera, ambos contenedor y *host* comparten el mismo directorio y así, si el contenedor necesita persistir datos, estos quedarán almacenados en el *host*. Cuando el contenedor arranque, tendrá disponibles los datos que necesitaba guardar y la aplicación que esté corriendo en el contenedor realizará las tareas correspondientes con dichos datos. Los volúmenes de Docker se pueden configurar con la opción `-v` *`directorio-host`*`:`*`directorio-contenedor`* del comando `docker run`.

Al igual que antes, el significado de los comandos y sus múltiples opciones puede encontrarse en la documentación oficial de Docker, accesible en [5].

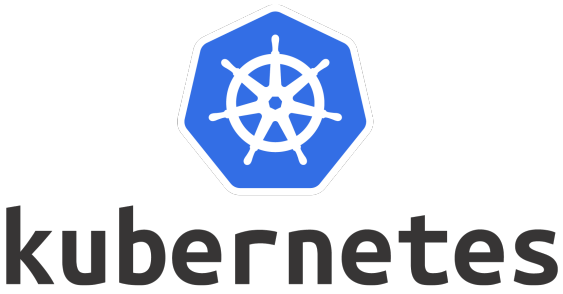
Para listar los contenedores de Docker en ejecución o los contenedores guardados se utiliza la instrucción `docker container ls`. Por otro lado, para listar las imágenes guardadas en Docker se debe ejecutar `docker image ls`. Y para ver los procesos en ejecución de Docker, se puede utilizar `docker ps -a`.

Para finalizar, cabe mencionar la utilidad de los siguientes comandos: `docker rm` *`id-contenedor`* para eliminar un contenedor de Docker; `docker rmi` *`id-imagen`* para borrar una imagen; y `docker push` *`usuario/nombre-imagen`* para subir una imagen a Docker Hub (para esto hay que disponer de un usuario en Docker Hub y haber construido antes la imagen *`usuario/nombre-imagen`* localmente).

2.2. Kubernetes

Kubernetes (K8s) es una plataforma de código abierto para automatizar la implementación, el escalado y la administración de aplicaciones en contenedores [6]. Se trata de un proyecto desarrollado en Go y mantenido por Google.

Actualmente Kubernetes está siendo utilizado por un gran número de empresas. Algunas de estas son: Spotify, Ebay, Adidas, ING, IBM, Nokia, Philips, Pinterest y Huawei [7].



2.2.1. Conceptos de Kubernetes

En este apartado se introducen los conceptos principales de Kubernetes, extendiendo aquellos que son más relevantes para el presente proyecto. La mayor parte de la información aquí mostrada procede de la documentación de Kubernetes, que puede consultarse en [8].

Kubernetes se suele utilizar a través de línea de comandos, con `kubectl`. Este comando estará conectado a un *cluster* de Kubernetes, y su administrador podrá comunicarse con su nodo maestro (*master*). Un *cluster* no es más que un conjunto de nodos, siendo un nodo un servidor físico, un ordenador o una máquina virtual.

Dentro del nodo *master* (plano de control) se ejecutan los siguientes tres procesos: `kube-apiserver`, `kube-controller-manager` y `kube-scheduler`. En los demás nodos del *cluster* (denominados *minions*) se ejecutan `kubelet` y `kube-proxy`. En la Fig. 2.2 se muestra la arquitectura de un *cluster* de Kubernetes desplegado en la nube. Lo interesante de esta imagen es ver los distintos procesos de Kubernetes que se están ejecutando en cada nodo.

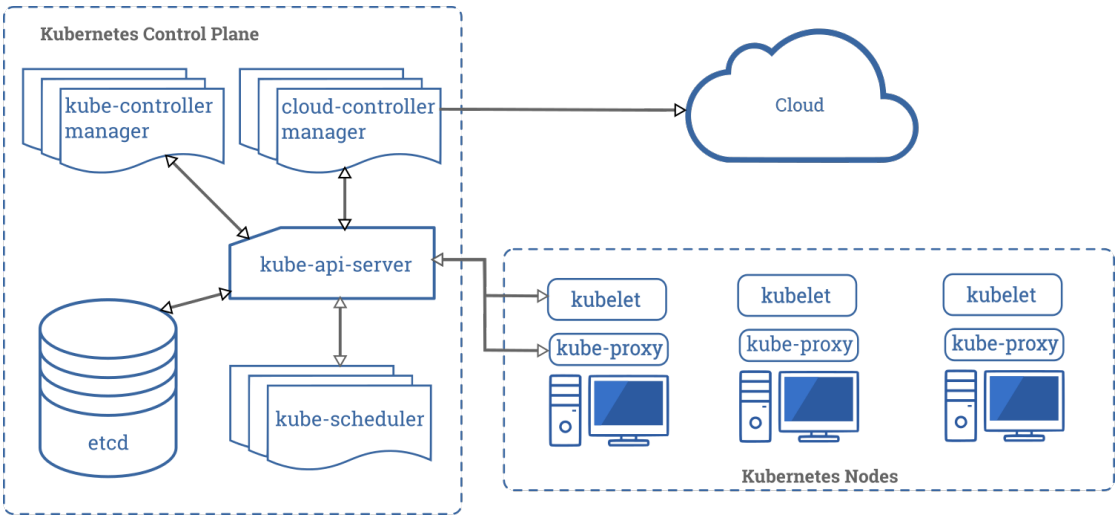


Fig. 2.2. Arquitectura de un *cluster* de Kubernetes [8]

A continuación se explica cada uno de estos procesos:

- `kube-apiserver`: Es un componente del plano de control de Kubernetes (nodo *master*) que expone la API de Kubernetes, es decir, es el *front-end* del plano de control.

- `etcd`: Se trata de una base de datos de tipo clave-valor para almacenar los datos del *cluster*.

- `kube-scheduler`: Este componente está pendiente de nuevos `Pods` creados sin un nodo asignado y les asigna uno teniendo en cuenta diversos aspectos como los recursos requeridos por el `Pod`.

- `kube-controller-manager`: Se encarga de gestionar distintos *controllers*, como los *node controller*, los *replication controller* o los *endpoints controller*.

- `cloud-controller-manager`: Permite enlazar el *cluster* a la API de un proveedor *cloud*, como Amazon Web Services, Google Cloud Platform, IBM Cloud o Microsoft Azure.

- `kubelet`: Componente que se ejecuta en cada nodo del *cluster* que se asegura de que todos los contenedores desplegados están ejecutándose correctamente en los `Pods`.

- `kube-proxy`: Se trata de un *proxy* de red que se ejecuta en cada nodo del *cluster* para mantener todas las reglas de red establecidas, como por ejemplo, permitir la comunicación entre los `Pods`.

2.2.2. Objetos de Kubernetes

Una vez conocida la estructura interna de Kubernetes, conviene explicar los principales tipos de objetos que se pueden crear en Kubernetes:

- `Pod`: Un `Pod` es la unidad básica de Kubernetes. Consiste en un conjunto de uno o más contenedores (habitualmente contenedores de Docker) que comparten la misma red y el mismo almacenamiento. No obstante, el caso de uso más común es ejecutar un contenedor por `Pod`. En Kubernetes, los `Pods` se consideran efímeros, es decir, desechables y sin estado (del inglés, *stateless*). De hecho, se suele decir que los `Pods` son "mortales".

- `Deployment`: Proporciona actualizaciones declarativas para `ReplicaSets`. De esta manera, en un `Deployment` se describe el estado deseado de los `Pods` y un *controller* se encargará de alcanzar y mantener este estado. El caso de uso habitual es crear un `Deployment` para definir un `ReplicaSet`, siendo este un objeto que simplemente crea un número determinado de réplicas de `Pods`. El `Deployment` permitirá, por ejemplo, escalar (aumentar o disminuir) el número réplicas de `Pods` en ejecución según se requiera.

- `StatefulSet`: Se trata de una especie de `Deployment` que trata a los `Pods` generados como instancias concretas y no efímeras. Un `StatefulSet` se puede usar para crear volúmenes de almacenamiento de información (por ejemplo, bases de datos).

- `Service`: Es la manera de conectar distintos `Pods` de un `Deployment` a la red de Kubernetes. Esto se realiza para posibilitar la comunicación entre distintos `Pods` de otros `Deployments`. Al enlazar un `Service` a un `Deployment`, a cada `Pod` se le asigna una dirección IP; y al `Deployment` se le asigna un único nombre de DNS. A efectos de conectar `Pods`, se suele utilizar el nombre de DNS, ya que no va a variar; las direcciones IP pueden cambiar si algún `Pod` nace o muere. Existe la opción de establecer un `NodePort` para mapear el puerto del `Service` a un puerto del nodo y poder acceder a la red de Kubernetes a través de dicho puerto y la dirección IP del nodo.

- `Ingress`: Se trata de un objeto que gestiona el acceso a los `Services` del *cluster*, típicamente vía HTTP. También permite balanceo de carga y SSL. Para poder utilizarlo, es necesario disponer de un *ingress controller* como NGINX o Traefik ejecutándose en el *cluster*. De esta manera, se puede utilizar un nombre de DNS para acceder a `Services` de tipo `NodePort` sin la necesidad de especificar dirección IP y puerto del nodo.

- `PersistentVolumeClaim`: Consiste en una petición de almacenamiento (un `PersistentVolume`) por parte de un `Deployment` o `StatefulSet`. El volumen en el que se almacenen los datos puede ser un directorio del nodo o algún tipo de almacenamiento en la nube (como instancias de AWS o Azure).

- `ConfigMap`: Se trata de un objeto que permite describir datos en formato clave-valor. Es realmente útil para definir variables de entorno que serán utilizadas por varios `Pods`, de manera que se vincula la configuración de estos `Pods` a un `ConfigMap`.

- `Secret`: Es una especie de `ConfigMap` que se utiliza para almacenar valores sensibles, tales como credenciales de usuario o claves de API. Los datos se pueden describir en formato clave-valor (como en un `ConfigMap`), pero los valores han de estar codificados en Base64.

- `CronJob`: Consiste en la ejecución de un `Job` en un intervalo de tiempo regular. Un `Job` no es más que un `Pod` que realiza una serie de tareas y que muere cuando termina. El intervalo de tiempo de un `CronJob` se especifica en formato Cron (Unix). Por ejemplo, la línea `*/5 * * * *` indicaría al `CronJob` que ejecute un `Job` cada 5 minutos.

Existen otros muchos conceptos de Kubernetes más avanzados y específicos. Hasta este punto se han explicado unas bases sobre Kubernetes. Sobre estas bases se realiza este Trabajo Fin de Grado.

2.2.3. Despliegue de aplicaciones en Kubernetes

Todos los objetos de Kubernetes se configuran mediante archivos YAML, que es un lenguaje de descripción, parecido a JSON pero más sencillo. También es posible emplear JSON, pero YAML resulta más legible y es lo más utilizado.

Al igual que JSON, la descripción de objetos en YAML se basa en el formato clave-valor. Mientras que en JSON se definen objetos mediante llaves y listas mediante corchetes; en YAML se definen mediante tabulaciones (típicamente de 2 espacios) y guiones, respectivamente. Además, en YAML no es necesario utilizar comillas para definir los nombres de las claves o valores de tipo *string*.

En el Código 2.2 se muestra el `Service` de `angular-ms` a modo de ejemplo de archivo YAML y objeto de Kubernetes. En este archivo se especifican varios puertos: el puerto del `Pod` (clave `port`), el puerto del contenedor (clave `targetPort`) y el puerto del nodo (clave `nodePort`). Este último es necesario para exponer el `Pod` hacia el exterior del *cluster*.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: angular-ms
spec:
  ports:
  - nodePort: 31600
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: angular-ms
    tier: frontend
  type: NodePort
```

Código 2.2. Descripción del `Service` de `angular-ms`

Otros parámetros interesantes son las etiquetas (en este caso `app` y `tier`, que podrían tener cualquier nombre, pero se considera buena práctica elegir nombres representativos). Estas etiquetas sirven para vincular el `Service` con un `Deployment` y sus `Pods`. Una vez vinculados, se podrá acceder a los `Pods` de este `Deployment` desde dentro del *cluster* mediante la dirección `http://angular-ms`, por ser este el nombre del `Service` (clave `name`). También es posible utilizar las direcciones IP de los `Pods`, asignadas por el `Service`, pero no es buena práctica ya que los `Pods` son mortales y estas direcciones podrían cambiar.

Por otro lado, en el Código 2.3 se puede observar la descripción del `Deployment` de `angular-ms`. Su definición es más extensa que la del `Service`, ya que es necesario especificar más características.

En primer lugar, se ve que las etiquetas del `Service` anterior están escritas tanto en la especificación del `Deployment` como en su plantilla (descripción de los `Pods`).

Otro aspecto interesante es que se indica el número de réplicas de `Pods` (en este caso solo una réplica). Para describir los `Pods` se utiliza la clave `template`. Como se vio antes, un `Pod` es un conjunto de contenedores. Estos contenedores se definen mediante una lista YAML con la clave `containers` (en este caso la lista es de un elemento, por lo que solo habrá un contenedor en el `Pod`). Dicho contenedor utiliza una imagen llamada `7rocky/angular-ms`, y Kubernetes la descarga de Docker Hub, como repositorio de imágenes por defecto.

También es recomendable establecer los límites de memoria y CPU que puede utilizar cada `Pod` del `Deployment`, mediante la clave `resources`; aunque no es estrictamente necesario.

Existe también otro concepto relacionado con la inicialización de los `Pods`: los `initContainers`. Estos son un conjunto de contenedores (no `Pods`) que se ejecutan antes de que se levante el `Pod` en sí. Por ejemplo, en el `Deployment` del Código 2.3 se utiliza un contenedor con la imagen `busybox:1.28` para ejecutar unos comandos. En este caso, se utiliza el comando `nslookup` de UNIX para conectarse al microservicio Orchestrator (cuyo nombre de DNS viene especificado en la variable de entorno `$ORCHESTRATOR_MS_HOSTNAME`), de manera que este `initContainer` no finaliza hasta que el microservicio Orchestrator esté disponible. Así, se consigue que el microservicio Angular no inicie antes de tiempo.

Existen más claves configurables en un `Deployment`, como la especificación de volúmenes o variables de entorno. Estas configuraciones se mostrarán más adelante.

Una vez escritos ambos objetos `Service` y `Deployment`, para desplegarlos en el *cluster* de Kubernetes es necesario ejecutar el siguiente comando: `kubectl apply -f` *`nombre-archivo`*`.yaml`. Es posible utilizar un solo archivo YAML para incluir varios objetos, utilizando un separador determinado (tres guiones).

Una vez aplicado el archivo YAML (conocido también como manifiesto), Kubernetes comenzará a crear todo lo que se ha requerido. Mediante comandos como `kubectl get all`, `kubectl get pods`, `kubectl get deployments` o `kubectl get services` se podrá ver la información relativa a los objetos desplegados en el *cluster*.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: angular-ms
spec:
  replicas: 1
  selector:
    matchLabels:
      app: angular-ms
      tier: frontend
  template:
    metadata:
      labels:
        app: angular-ms
        tier: frontend
    spec:
      containers:
      - image: 7rocky/angular-ms
        imagePullPolicy: Always
        name: angular-ms
        ports:
        - containerPort: 80
        resources:
          limits:
            cpu: 100m
            memory: 64Mi
          requests:
            memory: 8Mi
      initContainers:
      - command:
        - 'sh'
        - '-c'
        - 'until nslookup $ORCHESTRATOR_MS_HOSTNAME; do :; sleep 2; done;'
        env:
        - name: ORCHESTRATOR_MS_HOSTNAME
          valueFrom:
            configMapKeyRef:
              key: ORCHESTRATOR_MS_HOSTNAME
              name: env-configmap
        image: busybox:1.28
        name: init-angular-ms
```

Código 2.3. Descripción del `Deployment` de `angular-ms`

Para eliminar los objetos del *cluster*, se puede usar `kubectl delete deployment` *`nombre-deployment`* y `kubectl delete service` *`nombre-service`*. Aunque es más sencillo realizar `kubectl delete -f` *`nombre-archivo`*`.yaml`, que borrará del *cluster* todos los objetos descritos en dicho archivo.

Como Kubernetes utiliza Docker por defecto como entorno de ejecución de contenedores, muchos de los comandos de Kubernetes se parecen a los de Docker. Por ejemplo, `kubectl exec -it` *`nombre-pod`* `--` *`comando`* se utiliza para ejecutar un comando dentro de un `Pod` (comúnmente se ejecuta `sh` o `bash`). Este comando puede ser útil para depurar errores al desplegar una aplicación en Kubernetes.

Otros comandos útiles son `kubectl describe` *`nombre-objeto`* para obtener información detallada de un determinado objeto de Kubernetes; o `kubectl logs` *`nombre-pod`* para visualizar los registros generados por cierto `Pod` en ejecución.

Cabe mencionar también que no es necesario escribir un archivo YAML para desplegar aplicaciones en Kubernetes. Se pueden utilizar los comandos `kubectl run` y `kubectl expose` para desplegar un `Deployment` y exponerlo mediante un `Service`, respectivamente. Aun así, es preferible utilizar archivos YAML, ya que estos pueden ser versionados mediante un sistema de control de versiones, a diferencia de usar simples comandos.

2.2.4. Minikube

Minikube es una solución para crear un *cluster* local de Kubernetes formado por un solo nodo. Se trata de una manera sencilla de iniciarse en el mundo de Kubernetes. Está disponible para Windows, macOS y Linux.

Minikube aparece en la página oficial de Kubernetes como la herramienta idónea para iniciarse en el entorno de Kubernetes y utilizar sus funcionalidades más importantes.

Para generar el *cluster*, Minikube utiliza un gestor de máquinas virtuales (como VirtualBox) y crea una máquina virtual para instanciar el único nodo del *cluster* de Kubernetes (que será el nodo *master*). En el sistema operativo *host*, Minikube configura la línea de comandos `kubectl` para conectarse al *cluster* creado, de manera que no es necesario entrar en la máquina virtual.

Cuando se establezca un `Service` de tipo `NodePort`, el puerto configurado se mapeará al puerto correspondiente del nodo. La dirección IP del nodo puede cambiar dependiendo del gestor de máquinas virtuales, en VirtualBox es una dirección a partir de la 192.168.99.100. No obstante, es posible ejecutar `kubectl port-forward` *`nombre-service puerto-host`*`:`*`puerto-nodo`* para poder mapear el puerto del nodo con el puerto del *host*, de manera que se podría acceder al *cluster* por medio de *localhost* y el puerto configurado.

Los comandos básicos de Minikube son:

- `minikube start`: Inicia un *cluster* de Kubernetes, arranca la máquina virtual y configura `kubectl`.
- `minikube ip`: Muestra la dirección IP del nodo del *cluster*.
- `minikube ssh`: Se utiliza para conectarse al nodo a través de SSH.
- `minikube stop`: Termina la ejecución del *cluster* y apaga la máquina virtual.
- `minikube dashboard`: Muestra un *dashboard* en el navegador con los datos del *cluster* (`Pods`, `Deployments`, `Services`...).
- `minikube service` *`nombre-servicio`*: Abre un determinado `Service` en el navegador. Resulta útil cuando el `Service` es de tipo `NodePort`.
- `minikube delete`: Se utiliza para eliminar el *cluster* (junto con la máquina virtual y todos los datos guardados en ella).

Minikube se utiliza en este proyecto para desplegar el *cluster* de Kubernetes en un ordenador macOS con VirtualBox.

2.2.5. MicroK8s

MicroK8s es otra solución para desplegar Kubernetes en un entorno local. A diferencia de Minikube, es una solución más completa. MicroK8s está pensado para utilizarse en entornos Linux.

Esto se debe a que Kubernetes necesita de entornos Linux para funcionar. De ahí que Minikube haga uso de máquinas virtuales (al menos para sistemas Windows y macOS). MicroK8s no utiliza máquinas virtuales y, además, soporta la posibilidad de crear un *cluster* multi-nodo. Los comandos de MicroK8s son prácticamente iguales a los de Minikube. Su instalación también es bastante sencilla.

En este proyecto se utiliza MicroK8s para desplegar el *cluster* de Kubernetes en una Raspberry Pi 4 con Ubuntu Server 20.04 LTS como sistema operativo. El cambio de Minikube a MicroK8s se debe a que desarrollar la arquitectura de microservicios localmente mientras está el *cluster* desplegado se hace muy tedioso, ya que VirtualBox necesita bastantes recursos para mantener el *cluster*, y además cada microservicio ejecuta un proceso distinto del ordenador.

Por este motivo, la migración del *cluster* a la Raspberry Pi 4 favoreció el desarrollo, ya que desde ese momento, la máquina macOS iba a utilizarse principalmente para desarrollar; y la Raspberry Pi 4, para desplegar el *cluster*.

No obstante, Minikube sigue siendo una solución válida para desplegar el *cluster*, pero no lo suficientemente apropiada para desarrollar al mismo tiempo.

2.2.6. Monitorización en Kubernetes

Existen múltiples herramientas para obtener distintas métricas de los objetos de un *cluster* de Kubernetes. En primer lugar, está el *dashboard* de Kubernetes, el cual presenta de manera gráfica toda la información que se podría obtener por línea de comandos, además de dar la posibilidad de conectarse a los `Pods` por medio de una consola (en la propia página web) y de crear o actualizar objetos de Kubernetes escribiendo los manifiestos en YAML directamente en la página.

Una vista de este *dashboard* se presenta en la Fig. 2.3. Como curiosidad, esta página web está desarrollada con Angular.

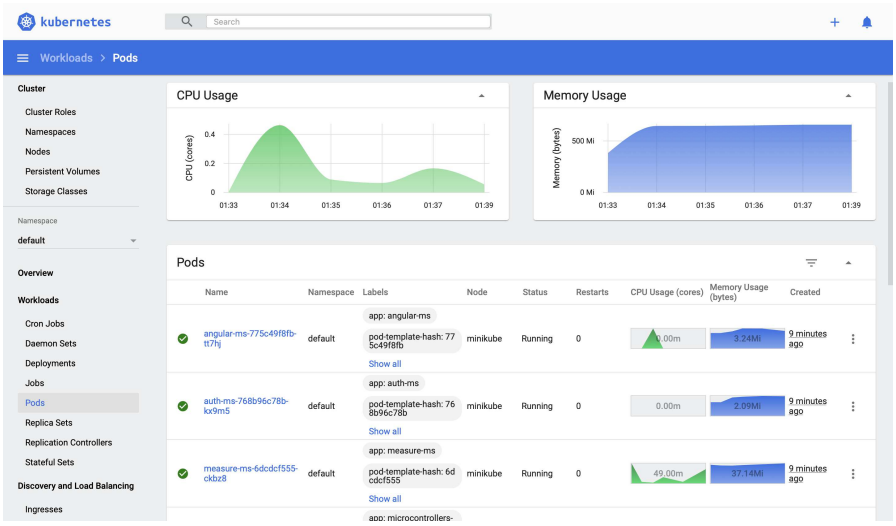


Fig. 2.3. Vista del *dashboard* de Kubernetes

Otras herramientas que aportan más información útil para los administradores de sistemas encargados de los despliegues en Kubernetes son Prometheus y Grafana.

La primera herramienta, Prometheus, obtiene datos de los `Pods` de Kubernetes y calcula distintas métricas. Prometheus se integra muy bien con Grafana, una herramienta de visualización de datos y creación de *dashboards* de control. Un ejemplo de estas herramientas aplicadas a un `Pod` se puede ver en la Fig. 2.4, la cual muestra el tráfico de red y la memoria y CPU utilizadas por el `Pod`.

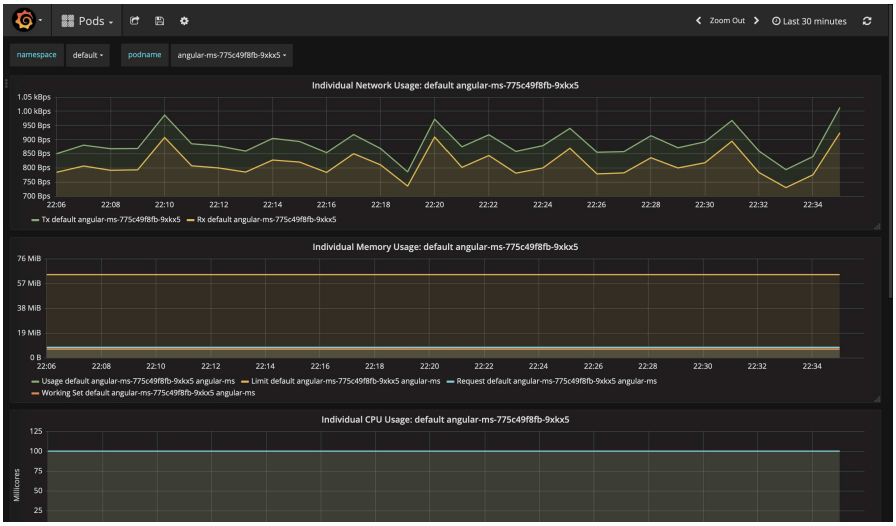


Fig. 2.4. Vista del *dashboard* de Grafana y Prometheus

2.3. Angular

Angular es un *framework* de diseño de aplicaciones y una plataforma de desarrollo para crear eficientes y sofisticadas aplicaciones web de una sola página [9] (SPA, *Single Page Application*).

Fig. 2.5. Logotipo de Angular

En este proyecto, se utiliza Angular (versión 9) para desarrollar un *dashboard* de control de los sensores por medio de la arquitectura de microservicios.

Angular es fundamentalmente un *framework* de desarrollo web *front-end*. Es un proyecto creado y mantenido por Google y está desarrollado en TypeScript (que es un lenguaje que extiende las funcionalidades de JavaScript).

2.3.1. Entorno de desarrollo de Angular

Los principales tipos de archivos que se utilizan en Angular son archivos HTML, CSS y TS. Sin embargo, los navegadores no entienden TypeScript, sino JavaScript; por lo que Angular realiza la transpilación de TypeScript a JavaScript automáticamente.

Para utilizar Angular, es conveniente instalar Angular CLI mediante `npm`. Al iniciar un proyecto de Angular, el entorno de desarrollo se configura por defecto para que cada vez que se guarde un archivo, Angular CLI realice las conversiones de archivos TS a JS y además se actualice la página del navegador automáticamente para cargar los nuevos cambios. Esto aumenta considerablemente la experiencia de desarrollo y la productividad del desarrollador.

Por otro lado, es posible utilizar preprocesadores de CSS (como LESS, SASS o Stylus). Los archivos escritos con preprocesadores tienen que ser convertidos a CSS para que el navegador los pueda entender. El uso de preprocesadores añade diversas funcionalidades a las hojas de estilo convencionales (por ejemplo, utilización de variables, reutilización de clases o estilos anidados) [10].

Angular utiliza una herramienta llamada Webpack para realizar estar traducciones de código y para refrescar automáticamente el navegador al realizar un cambio en un archivo.

2.3.2. Funcionamiento de Angular

La arquitectura de una aplicación de Angular se basa en módulos (denominados `NgModules`), que proporcionan un contexto de compilación para sus componentes (`Components`). Siempre tiene que existir un módulo raíz que permite arrancar la aplicación, al cual típicamente se añaden otros módulos.

Angular utiliza componentes para definir las vistas. Cada componente tiene una funcionalidad subyacente. Los componentes en Angular poseen una plantilla en HTML, una o varias hojas de estilo y un controlador en TypeScript.

Para utilizar los componentes en HTML, Angular crea etiquetas personalizadas con el nombre del componente. Además, existe un enlace bidireccional entre el archivo HTML y el archivo TS, es decir, que desde la plantilla se puede acceder a los atributos y métodos del controlador y viceversa.

Los componentes utilizan servicios (`Services`) para realizar funciones que no están directamente relacionadas con la vista. Es común utilizar servicios para realizar peticiones HTTP a arquitecturas de tipo API REST, principalmente.

Cabe mencionar que todas las peticiones HTTP que se realizan desde Angular se hacen mediante AJAX (*Asynchronous JavaScript And XML*). Esta característica es la que hace que las aplicaciones de Angular sean SPA, es decir, aplicaciones web que no recargan la página.

Estos conceptos y otros más avanzados pueden consultarse en la documentación oficial de Angular, accesible en [9].

2.3.3. Librerías de Angular

Entre las muchas librerías de Angular, existe Angular Material, la cual consiste en una serie de componentes que implementan patrones de interacción con el usuario de acuerdo con la normativa de Material Design [11]. Algunos de los componentes de Angular Material son: cajas de texto, botones, iconos, selectores, calendarios, menús desplegables, ventanas de diálogo y barras de progreso, entre otros.

Una librería que utiliza Angular por defecto es RxJS, la cual proporciona funciones reactivas de JavaScript y TypeScript. La característica clave de esta librería es que utiliza objetos denominados `Observables`, que facilitan el uso de código asíncrono basado en *callbacks* [12].

Otra funcionalidad que se utiliza en el presente proyecto es la API de Google Charts, para poder dibujar gráficos de manera sencilla. Esta librería se usa para mostrar la evolución en el tiempo de las distintas magnitudes medidas por los sensores.

Existen otras librerías de Angular, incluidas por defecto, como *@angular/forms* o *@angular/common/http* que sirven para implementar funcionalidades en formularios HTML y realizar peticiones HTTP, respectivamente.

Por otro lado, en Angular también es posible realizar pruebas. Angular incluye por defecto las librerías *karma* y *protractor*. La primera de ellas sirve para realizar pruebas unitarias y de integración; mientras que la segunda se utiliza para realizar pruebas *end-to-end* (E2E).

2.4. Node.js

Node.js es un entorno de ejecución para JavaScript construido sobre el motor de JavaScript V8 de Chrome [13]. Se trata de una plataforma orientada a eventos, ya que JavaScript es un lenguaje principalmente asíncrono. De hecho, Node.js está pensado para utilizar un único *thread* (esto es, un único proceso de ejecución). Lo que hace de Node.js un entorno escalable es el hecho de utilizar *callbacks* y promesas de JavaScript, para poder gestionar métodos asíncronos.

Es muy común el uso de Node.js para desarrollar servidores web, arquitecturas

API REST y microservicios, ya que el código que se genera es relativamente ligero. Además, JavaScript es uno de los lenguajes más rápidos en la actualidad.

En este proyecto se utiliza Node.js para desarrollar los microservicios relacionados con las distintas magnitudes medidas por los sensores. Por tanto, serán estos microservicios los que se conecten directamente a las placas de los usuarios. El API *gateway* también se desarrolla con Node.js.

Fig. 2.6. Logotipo de Node.js

2.4.1. Módulos y librerías de Node.js

Node.js dispone de una gran cantidad de módulos desarrollados para extender sus funcionalidades. Sin duda, el módulo más conocido de Node.js es *express*, el cual se utiliza para crear servidores HTTP y arquitecturas de tipo API REST de manera extremadamente sencilla y escalable. Además, *express* permite utilizar otras funcionalidades como autenticación mediante sesión HTTP o JWT (JSON Web Token), entre otras.

Existen muchos otros módulos programados para Node.js. Para utilizar estos módulos con Node.js, se suele utilizar `npm` como administrador de paquetes (se dice que `npm` significa *Node.js Package Manager*, ya que las siglas se corresponden).

Los principales comandos de `npm` son:

- `npm init`: Este es el comando para iniciar un nuevo proyecto en Node.js. El resultado de esta instrucción será un archivo `package.json` creado en el directorio de trabajo. Este archivo contiene los datos principales del proyecto (nombre, versión, dependencias, repositorios, etc.).

- `npm install` *`nombre-módulo`*: Se utiliza para añadir módulos al proyecto de Node.js en el que se esté trabajando. Al ejecutar este comando, se actualizará el `package.json` con las nuevas dependencias, se creará un archivo `package-lock.json`, que contiene las versiones exactas de las dependencias instaladas y se generará también un directorio denominado `node_modules` en el que se encuentran todos los módulos del proyecto.

- `npm install`: Se utiliza para instalar las dependencias de un determinado proyecto (listadas en los archivos `package.json` y `package-lock.json`).

- `npm start`: Consiste en ejecutar un determinado *script* llamado *start*, indicado en el `package.json`. Lo habitual es iniciar una aplicación de Node.js, mediante la instrucción `node index.js`, por ejemplo.

- `npm test`: Se utiliza para ejecutar los *tests* unitarios del proyecto. Esto también es un *script* del `package.json`, que esta vez se llama *test*. Se pueden añadir *scripts* personalizados al `package.json`, y para ejecutarlos habrá que utilizar `npm run` *`nombre-script`*.

Para realizar pruebas unitarias y de integración, existen múltiples librerías como *mocha* y *chai*, o *jest*. En este proyecto se ha usado *jest*, ya que combina las funciones de *framework* de pruebas (como *mocha*) y las de librería de aserciones (como *chai*) en un solo paquete.

2.5. Arduino

Arduino es una plataforma de electrónica de *hardware* y *software* libre. Las placas de Arduino son microcontroladores, capaces de leer entradas (por ejemplo de sensores) y de transformarlas a unas salidas [14]. Esto se realiza mediante un código escrito en un lenguaje de programación propio de Arduino, basado en C++.

La estructura básica de un programa de Arduino (denominado *sketch*) es la del siguiente código:

```
void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:

}
```

La función `setup()` se utiliza para realizar las configuraciones iniciales del microcontrolador (definir pines de entrada y salida, configurar conexiones, etc.). Por otro lado, en la función `loop()` se escribe el código que deberá realizar el microcontrolador continuamente, ya que los programas diseñados para microcontroladores siempre deben entrar en bucle infinito.

En este proyecto se utiliza la placa de Arduino Uno WiFi Rev2 [15], ya que posee una antena WiFi incorporada. De esta manera no es necesario adquirir un módulo de conexión WiFi aparte de la placa de Arduino.

Para utilizar la interfaz de conexión WiFi se usa una librería de Arduino llamada `WiFiNINA.h`. Es fácil de aprender cómo se usa esta librería a partir de la documentación de Arduino y de los ejemplos que proporciona el Arduino IDE referentes a dicha librería, accesibles en [16].

Los sensores que se utilizan en este proyecto son:

- *Grove - Temperature Sensor V1.2*: Se trata de un sensor de temperatura de la empresa Seeed. Utiliza un termistor para detectar la temperatura ambiente,

de manera que la resistencia del termistor aumenta cuando baja la temperatura ambiente. El rango de temperaturas detectable por el sensor es desde −40 °C hasta 125 °C, con una precisión de ±1'5 °C. Más características de este dispositivo pueden encontrarse en [17].

- *Grove - Moisture Sensor V1.4*: Este es un sensor de humedad también de la empresa Seeed. Se puede utilizar principalmente para detectar la humedad del suelo, por ejemplo, para saber si una planta necesita más humedad en la tierra de la maceta o menos. La documentación de este dispositivo se encuentra en [18].

2.6. Python

Python es un lenguaje de programación multipropósito. Actualmente es muy utilizado tanto para *Machine Learning* como para desarrollo web. Se trata de un lenguaje interpretado que busca que el código sea legible.

Fig. 2.7. Logotipo de Python

En este proyecto, se utiliza Python (versión 3) para el microservicio de estadísticas. Básicamente, lo que realiza este servicio es calcular ciertas estadísticas con los datos de los sensores del microcontrolador y guardarlos en una base de datos MongoDB. Una de las muchas características de Python es que permite el uso de *threads* (hilos de ejecución simultáneos), por lo que podrá realizar distintos cálculos al mismo tiempo. Este es el principal motivo para elegir Python para este microservicio.

Para realizar pruebas unitarias, se utiliza *unittest*, una librería de Python que proporciona diversas funciones para realizar aserciones.

2.7. Go

Go es un lenguaje de programación de código abierto desarrollado por Google. Se trata de un lenguaje expresivo, conciso, limpio y eficiente [19]. Además, es un lenguaje compilado y de tipado estático que intenta asemejarse a C o C++, aunque posee características de lenguajes dinámicos como Python o JavaScript.

En este proyecto se utiliza Go para desarrollar el microservicio de autenticación y registro de usuarios. Se ha elegido este lenguaje debido a que tiene un buen desempeño y es sencillo de utilizar. Además, tanto Docker como Kubernetes están programados en Go, por lo que ambos tienen soporte para este lenguaje.

2.8. MongoDB

MongoDB es una base de datos distribuida orientada a documentos y de propósito general [20]. Está optimizada para utilizar enormes cantidades de datos. Se trata de una base de datos NoSQL, es decir, no relacional.

Fig. 2.8. Logotipo de MongoDB

Los documentos que se almacenan en MongoDB tienen formato JSON (*JavaScript Object Notation*), como el que se muestra a continuación[1]:

```
{
  "_id": "5cf0029caff5056591b0ce7d",
  "firstname": "Jane",
  "lastname": "Wu",
  "address": {
    "street": "1 Circle Rd",
    "city": "Los Angeles",
    "state": "CA",
    "zip": "90404"
  },
  "hobbies": [
    "surfing",
    "coding"
  ]
}
```

Como se puede ver, el formato JSON es sencillo de leer, ya que consiste en un conjunto de parejas clave-valor. Este formato es más flexible que el formato de las bases de datos relacionales (formato de tablas), ya que permite escribir objetos y listas dentro de documentos (como `"address"` y `"hobbies"` en el ejemplo anterior, respectivamente) y porque no es necesario que todos los documentos tengan el mismo formato.

MongoDB estructura la base de datos en colecciones. Cada colección contiene un conjunto de documentos. Como se dijo antes, estos documentos pueden tener estructuras variadas; sin embargo, no es buena práctica configurar modelos de documento demasiado dispares dentro de una misma colección.

Se ha pensado utilizar esta base de datos para almacenar la información recogida de los distintos sensores IoT y las estadísticas calculadas en base a ellos. MongoDB es el servicio adecuado, ya que la cantidad de datos que se almacenarán es relativamente grande. Además, en Docker Hub existe una imagen oficial de MongoDB y existen librerías para conectarse escritas en Node.js y Python, las cuales facilitan el desarrollo en gran medida.

[1]Documento extraído de [20]

2.9. MySQL

En este proyecto se utiliza MySQL como base de datos relacional. Se trata de la base de datos *open-source* más popular con un alto desempeño y una gran escalabilidad [21].

La utilidad que se le ha dado a este servicio de base de datos SQL es la de almacenar datos relacionados con usuarios. Se ha elegido MySQL principalmente porque hay una gran cantidad de librerías desarrolladas para conectarse a esta base de datos desde lenguajes de programación como Node.js, Java o Python. Además, dispone de una imagen en Docker Hub.

Con la versión 8 de MySQL, este servicio ha añadido un sistema de base de datos de tipo NoSQL. Podría haberse sustituido el servicio de MongoDB por este. No obstante, se trata de un servicio bastante reciente (la versión es del año 2018) y, por ello, se ha considerado como opción más confiable la de utilizar MongoDB como sistema de base de datos no relacional.

2.10. RabbitMQ

RabbitMQ es lo que se conoce como un bróker de mensajes. Se trata de un servicio *open-source* fácil de desplegar en entornos *cloud* que soporta distintos protocolos de mensajería, entre ellos AMQP.

2.10.1. AMQP

AMQP, de sus siglas en inglés *Advanced Message Queueing Protocol*, es un protocolo abierto de Internet de mensajería diseñado para transferir mensajes entre aplicaciones u organizaciones. Es un protocolo de mensajería que destaca por su ubicuidad, su seguridad, su fidelidad, su aplicabilidad, su interoperabilidad y su manejabilidad [22].

El uso que se le da a este protocolo en el presente Trabajo Fin de Grado es el de comunicar distintos microservicios. RabbitMQ será el servicio encargado de gestionar distintas colas de mensajes. Por su parte habrá microservicios que manden mensajes a una cola de RabbitMQ, y otro microservicio recibirá los mensajes de esas colas para realizar su actividad.

Lo que se consigue con este sistema de colas es la posibilidad de que los microservicios se despreocupen la recepción de los mensajes que envían. Al estar RabbitMQ en medio, este almacenará los mensajes que sean necesarios en la colas correspondientes hasta que el destinatario pueda recibirlos.

2.11. NGINX

NGINX es un servidor HTTP y *proxy* reverso de código abierto [23]. Es un servidor web ligero y de alto rendimiento. Esta tecnología se ha utilizado para alojar la aplicación web desarrollada con Angular dentro de un contenedor de Docker con la imagen oficial de NGINX.

Lo interesante de utilizar NGINX es que se pueden especificar reglas de direccionamiento según la URL que llegue al servidor. Así, se pueden filtrar las peticiones en función del origen de la petición, por ejemplo.

En este proyecto se ha utilizado NGINX principalmente porque permite redirigir peticiones HTTP, es sencillo de usar y dispone de una imagen de Docker oficial.

2.12. Git

Git es un sistema de control de versiones distribuido. Es un *software* de código libre diseñado para gestionar desde pequeños hasta grandes proyectos con rapidez y eficiencia [24].

Los sistemas de control de versiones, en general, están pensados para utilizarse en equipos de desarrollo, donde varias personas pueden estar modificando muchos archivos a la vez. Si se utiliza un sistema de control de versiones como Git, cada miembro del equipo estará al tanto de los cambios realizados por cada desarrollador y se evitarán un gran número de conflictos.

Aunque este Trabajo Fin de Grado ha sido realizado de manera individual, se ha empleado Git para poder utilizar otras de sus múltiples virtudes.

2.12.1. Ramas de Git

Cuando se inicializa un repositorio de Git, por defecto se crea la rama `master`. A partir de aquí se puede empezar a programar. Toda rama nueva parte desde una rama de origen. En este caso, si se crea una rama nueva (por ejemplo `mi-rama`), partirá desde `master` y contendrá todo el código de esta rama.

En este momento, todos los cambios realizados en `mi-rama` no estarán presentes en `master`. Todas las ramas son independientes entre sí. Si se vuelve a la rama `master`, se volverá al código inicial, pero sin perder el código de `mi-rama`.

Esta característica de Git es útil cuando se quiere realizar una nueva funcionalidad en el código. El proceso es: crear una nueva rama, desarrollar la funcionalidad y fusionar dicha rama con la rama origen. De esta manera se tiene el código más estructurado y se consigue que la rama `master` tenga siempre código funcional.

2.12.2. GitHub

Los conceptos de Git son sencillos; sin embargo, para utilizarlo es necesario usar una interfaz de línea de comandos, lo cual puede llegar a ser tedioso y confuso. Por ello, existen soluciones *on-line* como GitHub, GitLab o BitBucket que implementan una interfaz gráfica de Git bastante más amigable que una consola.

En este proyecto se ha utilizado GitHub para almacenar el código desarrollado. El repositorio se encuentra en la siguiente dirección: https://github.com/7Rocky/IoT_Microservices.

Capítulo 3

Estado del arte

Desde el surgimiento de Internet alrededor del año 1990, las tecnologías del desarrollo web han evolucionado en gran medida. En la actualidad existen cientos de formas de desarrollar un sitio web; tantas, que a veces es difícil elegir qué tecnología utilizar.

Por otro lado, las arquitecturas de computadores y las arquitecturas de redes también han mejorado. Se ha optimizado notablemente la rapidez de procesamiento de los sistemas y comunicaciones.

De estos avances de tecnologías surgió *Internet of Things* (IoT), que hace referencia a la conexión de objetos cotidianos a la red de Internet (por ejemplo, un reloj, una televisión, una lámpara, una persiana o un radiador, normalmente apodados con el término *smart*).

En este capítulo se abordarán las últimas tendencias en cuanto a desarrollo web y desarrollo de aplicaciones, ahondando en conceptos como las arquitecturas de microservicios o las aplicaciones basadas en contenedores.

Posteriormente, se hablará de las tecnologías de *Cloud Computing* y de *Internet of Things*.

Finalmente, se hará mención al sector de la domótica poniendo como ejemplo a una empresa que utiliza estas nuevas tecnologías en su modelo de negocio.

3.1. Microservicios

Las aplicaciones web se basan en un tipo de arquitectura cliente-servidor, esto es, el cliente realiza peticiones al servidor y este le envía una respuesta en función de la petición recibida.

Una aplicación web monolítica es aquella en la que toda la funcionalidad se implementa en un único programa, que se aloja y se ejecuta en un único servidor físico.

Por otro lado, una arquitectura de microservicios consiste en dividir una aplicación monolítica en programas más pequeños que implementan una sola funcionalidad (principio de responsabilidad única). De este modo, se utilizan distintos servidores donde se alojan y se ejecutan cada uno de estos microservicios, manteniendo una comunicación entre sí por medio de la red (comúnmente mediante REST y colas de mensajes) [25].

En la siguiente Fig. 3.1, se muestra de manera visual lo que se ha explicado sobre la arquitectura monolítica y la arquitectura de microservicios.

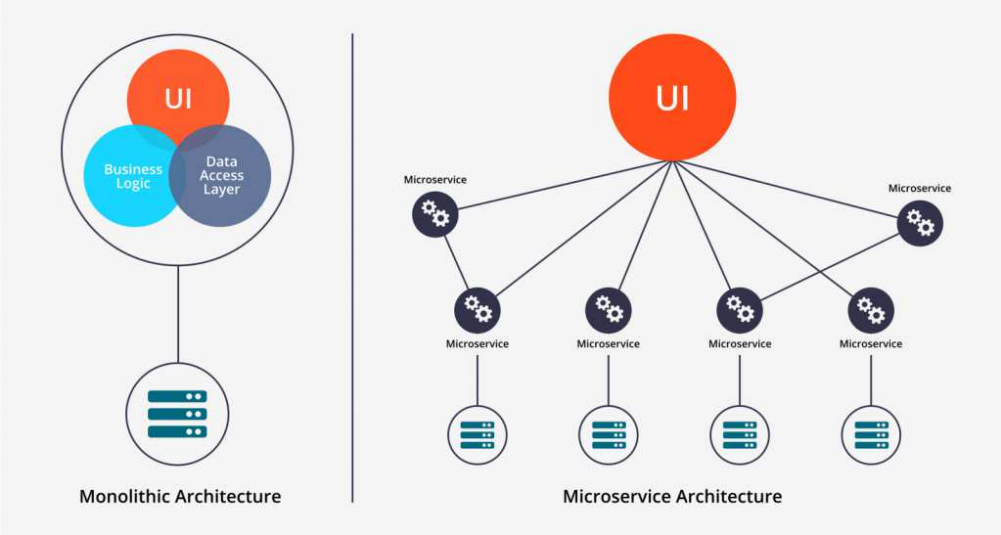


Fig. 3.1. Comparación entre arquitectura monolítica y microservicios [26]

Inicialmente, las aplicaciones web se desarrollaban con una arquitectura monolítica. Era común el uso del *stack* LAMP (Linux, Apache, MySQL, PHP) o similares,

en los que toda la aplicación estaba contenida en un solo servidor web Apache (montado en Linux) que ejecutaba los *scripts* de PHP para procesar una petición y consultaba una base de datos de MySQL. También era frecuente el uso de arquitecturas J2EE, basadas en el lenguaje Java Enterprise Edition, con un servidor de aplicaciones como Apache Tomcat.

Con el auge de las aplicaciones web y móvil, este tipo de arquitectura se torna insostenible cuando el tamaño de la aplicación aumenta o cuando el tráfico recibido es muy elevado.

La principal desventaja de una aplicación monolítica es que la cantidad de código aumenta considerablemente, lo que conlleva más trabajo en el desarrollo de la aplicación y en la realización de pruebas y depuración de errores, que pueden afectar a nuevas funcionalidades o incluso a funcionalidades anteriores que se han visto perjudicadas al incluir nuevo código. De la misma manera, un cambio en una funcionalidad del programa puede afectar al resto. Además, el entorno de una aplicación monolítica suele ser muy limitado y rígido, ya que está sujeto a un único lenguaje de programación en el servidor [27]. Por tanto, la escalabilidad a nivel vertical no es muy razonable.

Desde el otro punto de vista, la arquitectura de microservicios presenta ventajas como que cada microservicio puede estar escrito en un lenguaje de programación diferente, ya que cada uno se ejecuta en un servidor distinto. Con esto, se obtiene el poder de decidir qué lenguaje utilizar según la funcionalidad que se quiera implementar. También es destacable la escalabilidad de este tipo de arquitectura (tanto vertical como horizontal).

Otra ventaja de la arquitectura de microservicios es que favorece la división del trabajo dentro de un equipo de desarrollo [27] y la depuración de errores, ya que, al tratarse de módulos más o menos independientes, cada programador del equipo puede centrarse en ciertos servicios; y, al incluir menos código, conseguir que la depuración de errores sea más sencilla. También ayuda a que la modificación de funcionalidades no afecte a los demás microservicios.

No obstante, existen también inconvenientes. Por ejemplo, a la hora de integrar toda la funcionalidad de la aplicación, puede resultar costoso corregir fallos procedentes de la comunicación entre microservicios.

A modo de resumen, en las siguientes tablas se presentan las ventajas e inconvenientes de la arquitectura monolítica y de la arquitectura de microservicios [26].

Tabla 3.1. Ventajas de la arquitectura monolítica

Ventajas	Descripción
Rapidez	Al estar toda la lógica contenida en un solo servidor, todas las comunicaciones entre módulos se realizan en un tiempo óptimo
Seguridad y estabilidad	Como toda la aplicación está en un único servidor, el sistema es más seguro y estable en cuanto a comunicaciones por la red o fallos del sistema

Tabla 3.2. Ventajas de la arquitectura de microservicios

Ventajas	Descripción
Escalabilidad	Permite la creación de módulos y la separación de funcionalidades de manera sencilla. Además, solo es necesario desplegar el servicio modificado o añadido
Trabajo en equipo	Cada desarrollador puede centrarse en unos pocos microservicios y encargarse de las pruebas unitarias y del despliegue de estos microservicios
Distintos lenguajes	Aporta poder de decisión para elegir el lenguaje de programación más apropiado para el microservicio, en función de los requisitos que sean necesarios
Automatización	Es habitual utilizar la arquitectura de microservicios con un proceso de automatización, como CI/CD u orquestación de servicios con Kubernetes
Reutilización	Resulta sencillo adaptar el código de un microservicio ya funcional para generar otro microservicio parecido, y así ahorrar tiempo en el proceso de desarrollo
Versiones	Es posible utilizar varias versiones (típicamente dos) de un microservicio funcionando a la vez en producción, lo cual permite evitar fallos de regresión

Tabla 3.3. Inconvenientes de la arquitectura monolítica

Inconvenientes	Descripción
Lenguaje de programación	Puede llegar a ser costoso migrar a otro lenguaje de programación o incluso a una nueva versión del lenguaje utilizado
Exclusividad	Las aplicaciones monolíticas no suelen ser reutilizables ya que están diseñadas a medida para algún propósito concreto
Despliegue	Como todas las funcionalidades están en un solo servidor, al modificar o añadir una función, se debe desplegar la aplicación entera

Tabla 3.4. Inconvenientes de la arquitectura de microservicios

Inconvenientes	Descripción
Pruebas de integración	Puede ser costoso corregir errores procedentes de la comunicación entre distintos módulos, ya que los microservicios tienden a ser independientes entre sí
Tiempo de petición	Cuando una petición realiza muchas conexiones entre módulos, es probable que la latencia no sea adecuada
Complejidad	Aumentar el número de microservicios de la aplicación puede elevar considerablemente su nivel de complejidad

3.2. Contenedores y máquinas virtuales

Las distintas maneras de alojar una aplicación web son: en un servidor físico, en una máquina virtual o en un contenedor.

La opción de alojar una aplicación en un servidor físico es la más sencilla de implementar. Esta era la manera predominante en los inicios de Internet (con el *stack* LAMP y las aplicaciones J2EE), ya que las aplicaciones que se desarrollaban entonces se diseñaban con una arquitectura monolítica. No obstante, esta opción

es la menos escalable, ya que sería necesario disponer de un mayor número de servidores físicos si se quiere replicar la aplicación (escalabilidad horizontal). Y, por otro lado, incluir nuevas funcionalidades se vuelve más complicado conforme aumenta la envergadura de la aplicación, ya que un fallo puede afectar a todo el sistema (escalabilidad vertical).

Una solución para mejorar la escalabilidad horizontal es crear máquinas virtuales dentro del propio servidor y alojar cada réplica de la aplicación en una máquina virtual. Sin duda, esta opción mejora la escalabilidad, pero dificulta las tareas de configuración y de aprovechamiento de recursos, ya que cada máquina virtual posee un sistema operativo completo; y el propio servidor físico también tiene su propio sistema operativo.

Entonces, una solución a esto es utilizar distintos contenedores que hagan uso del sistema operativo del servidor para funcionar. Por este motivo, los contenedores son más livianos que las máquinas virtuales, ya que los contenedores tienen los mínimos archivos de configuración que necesitan y no un sistema operativo completo.

De manera esquemática, se muestran estas dos últimas opciones de alojamiento de aplicaciones web en la siguiente Fig. 3.2.

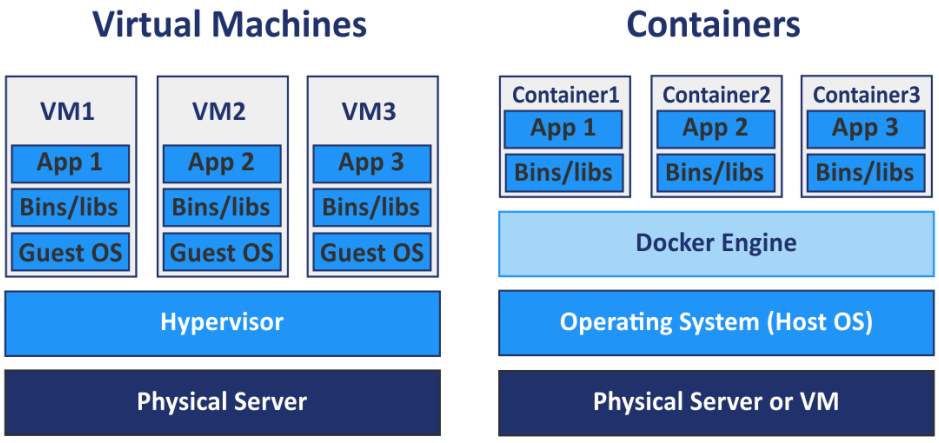


Fig. 3.2. Comparación entre máquinas virtuales y contenedores

Al dividir una aplicación monolítica en distintas funcionalidades y al incluir cada una de estas en un contenedor independiente se obtiene como resultado una arquitectura de microservicios como la que se vio en la Fig. 3.1.

Por otro lado, en la actualidad es muy común el uso de computación sin servidor (también conocida como *serverless*). Cada vez más, se recurre a proveedores de servicios en la nube, como Amazon Web Services, Google Cloud Platform, IBM Cloud o Microsoft Azure. Esta tecnología es ampliamente utilizada para el desarrollo de arquitecturas de tipo API REST.

Por ejemplo, con AWS Lambda [28] se puede ejecutar un código cuando Lambda esté activo, de modo que es Lambda el que administra y aprovisiona sus propios servidores. De esta manera, el código solo será ejecutado cuando sea requerido, lo cual optimiza los recursos utilizados y minimiza el coste del servicio.

Para el presente proyecto, se van a utilizar distintos contenedores de Docker para almacenar microservicios; y estos contenedores serán administrados y gestionados por Kubernetes, un sistema de orquestación de servicios.

3.3. Cultura DevOps

El término DevOps proviene de la conjunción de *Development* y *Operations*. El objetivo de esta cultura moderna es integrar ambos roles de un equipo de desarrollo para facilitar el despliegue de aplicaciones.

El personal especializado en *Development* se dedica a la programación de aplicaciones (diseño visual, lógica de negocio, consultas a bases de datos, etc.) y tiene un amplio conocimiento en distintos lenguajes de programación.

Por otro lado, el grupo de *Operations* se encarga de manejar los despliegues de las aplicaciones que realizan los desarrolladores. Por tanto, los operadores deben saber configurar servidores, bases de datos, servidores en la nube, máquinas virtuales y contenedores de aplicaciones, además de realizar distintos tipos de métricas sobre las aplicaciones en producción, para detectar puntos de mejora.

El problema llega cuando la aplicación falla en producción y es necesario arreglarla cuanto antes. Si los desarrolladores corrigen el error, pero no saben desplegar la aplicación a producción, tendrán que esperar a que un operador pueda hacerlo.

Lo que se intenta con la filosofía DevOps es integrar ambas ramas de desarrollo y operaciones; y, sobre todo, automatizar tareas, pruebas y despliegues. Con esto, se obtiene una mayor eficiencia a la hora de poner aplicaciones en producción y

una menor posibilidad de errores, obteniendo así un *software* de mejor calidad. La Fig. 3.3 muestra el ciclo de vida del *software* que persigue la cultura DevOps.

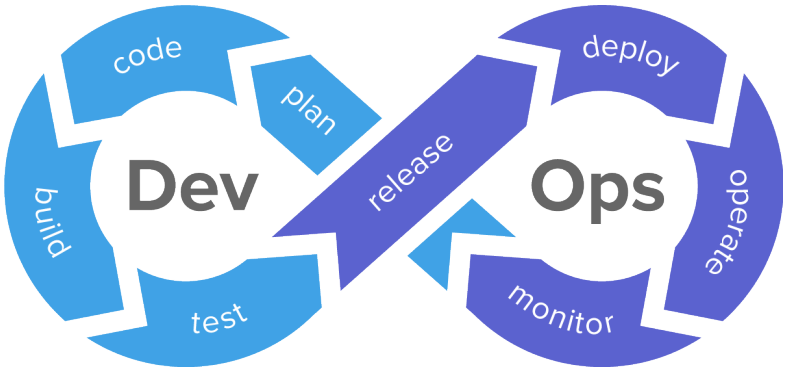


Fig. 3.3. Ciclo de vida del *software* según la cultura DevOps [29]

A raíz de DevOps, surge el paradigma de integración continua y entrega continua (CI/CD, *Continuous Integration / Continuous Delivery*), el cual es un modo de desarrollar *software* que trata de ejecutar de forma automática una serie de pruebas antes de integrar un cambio con la aplicación desarrollada hasta el momento (CI) y antes de ponerlo en producción (CD). De esta manera, si las pruebas fallan, los cambios no se incluirán, reduciendo la probabilidad de que aparezcan errores.

La metodología CI/CD es muy útil cuando se desarrolla una arquitectura de microservicios, ya que las actualizaciones de cada microservicio deberían ser independientes de los demás (y, por tanto, tener unos procesos de CI/CD distintos). En cambio, en una arquitectura monolítica, si falla una sola función, falla todo el sistema. Esto se puede ver esquematizado en la Fig. 3.4.

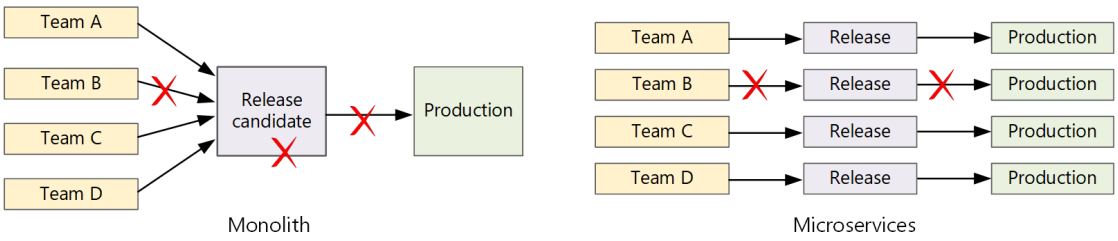


Fig. 3.4. Errores en las actualizaciones según la arquitectura [30]

3.4. *Cloud Computing*

Cloud Computing (término comúnmente conocido como "la nube") se refiere a un conjunto de servicios proporcionados por un proveedor que presentan una alta disponibilidad y escalabilidad, entre otras características.

Existen muchos servicios que se atribuyen a *Cloud Computing*, los cuales se pueden clasificar en *software* como servicio (SaaS, *Software as a Service*), plataforma como servicio (PaaS, *Platform as a Service*) e infraestructura como servicio (IaaS, *Infrastructure as a Service*), principalmente.

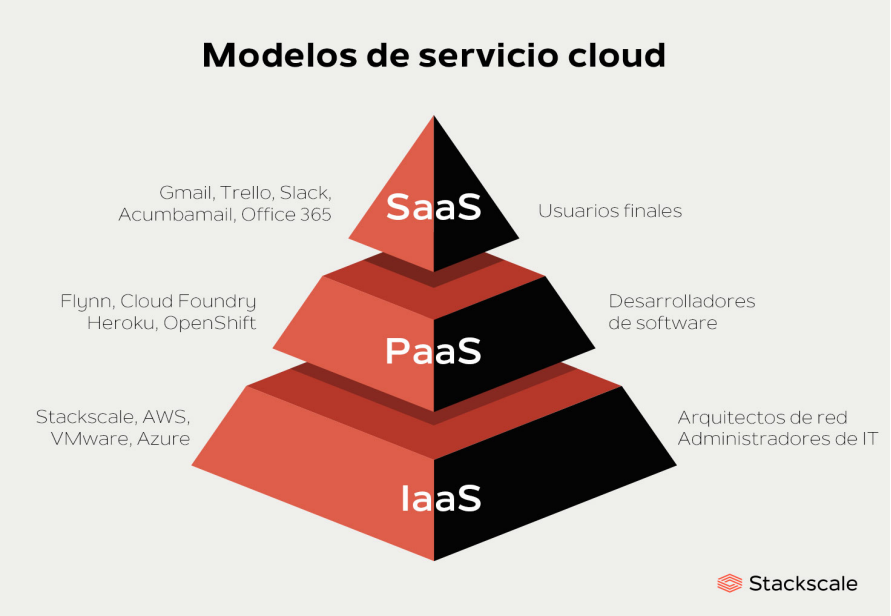


Fig. 3.5. Tipos de servicios de *Cloud Computing* [31]

SaaS consiste en un servicio proporcionado por los proveedores *cloud* listo para que los usuarios lo utilicen por medio de Internet. De esta manera, los clientes solo pagan por el uso de este servicio. Algunos ejemplos de SaaS son Gmail (un servicio de correo electrónico) o Slack (un sistema de mensajería orientado al ámbito profesional), como se puede ver en la Fig. 3.5.

PaaS hace referencia a un entorno de desarrollo gestionado por el proveedor en el que el cliente solo tiene que proporcionar los datos y el *software* necesario pa-

ra desarrollar o probar una aplicación. Algunos de los PaaS más conocidos son Cloud Foundry (disponible en IBM Cloud) y Heroku, muy útiles pare desplegar aplicaciones web.

Por último, IaaS es una solución para montar una arquitectura de computadores y servicios conectados entre sí, siendo el cliente el que elige qué hacer con cada servidor, con cierto rango de libertad. Amazon Web Services y Microsoft Azure son los dos principales proveedores *cloud* que ofrecen IaaS.

En la Fig. 3.6 se muestra una comparativa entre estos tres servicios de *Cloud Computing* y la opción de construir una solución local, sin utilizar la nube. Es interesante ver cómo el SaaS es gestionado al completo por el proveedor, y el cliente solo tiene que utilizarlo; las PaaS, en cambio, requieren que los clientes aporten datos y aplicaciones; mientras que en la IaaS, el proveedor solo se encarga de gestionar sus propios servidores, el almacenamiento y la red.

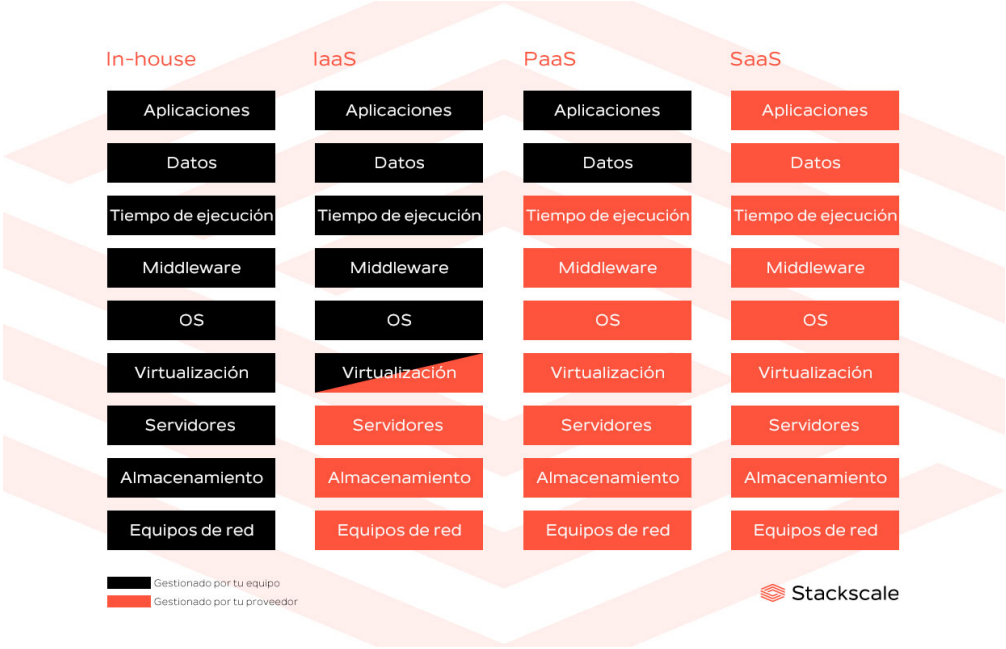


Fig. 3.6. Comparación entre los servicios de *Cloud Computing* [31]

Lo interesante es que *cloud* públicas como Amazon Web Services, Microsoft Azure, Google Cloud Platform o IBM Cloud ofrecen servicios para utilizar Kubernetes y Docker en el despliegue de aplicaciones (incluyéndose en las PaaS). Y este servicio es el que se implementa en este proyecto, pero de manera local.

3.5. *Internet of Things* (IoT)

El término IoT hace referencia a la conexión de objetos cotidianos a Internet. Estos dispositivos suelen incluir sensores, antenas y *software* para poder aprovechar su capacidad de conexión.

Actualmente, resulta viable el uso de IoT debido a que [32]:

- Los sensores electrónicos son dispositivos de baja potencia y bajo coste.
- Los microcontroladores y sensores disponen de antenas WiFi para poder utilizar distintos protocolos de red (como HTTP[2] o MQTT[3]).
- Existen servicios de *Cloud Computing* que favorecen el uso de IoT.
- Su combinación con algoritmos de *Machine Learning* es muy efectiva, ya que los dispositivos IoT son una fuente de datos muy útil para estas técnicas de aprendizaje automático.
- La utilización de inteligencia artificial (por ejemplo para el procesamiento del lenguaje natural) ha hecho que los dispositivos IoT sean empleados en entornos domésticos.

La implementación de IoT en este proyecto consiste en la utilización de una placa de Arduino con conectividad WiFi y unos pocos sensores y LED. La comunicación con el exterior se realiza mediante HTTP.

[2] *HyperText Transfer Protocol*
[3] *Message Queuing Telemetry Transport*

3.6. Domótica

La domótica (del latín, *domus*, 'casa') consiste en la utilización de tecnología para automatizar una vivienda, ya sea gestionando la energía y los recursos o proporcionando bienestar y seguridad.

Para proveer estos servicios, la domótica se sirve de las tecnologías de IoT y *Machine Learning*, principalmente. De esta manera, se consigue la conectividad de los distintos dispositivos inteligentes de una vivienda (televisiones, lámparas, neveras, radiadores, persianas, aspiradores, etc.).

A estos servicios de domótica se les suele añadir una aplicación de control y gestión, para que los usuarios puedan manejar sus dispositivos a su antojo y disponer de una gran cantidad de datos relacionados con el estado de su vivienda.

3.6.1. Wink

Wink es una empresa cuya actividad principal es la domótica. Los usuarios que utilizan Wink disponen de una *app* desde la que pueden gestionar los aparatos inteligentes de su casa, ya que estos se pueden conectar a Wink.

Lo interesante de Wink es que realizaron una migración de sus servicios a Kubernetes. De hecho, publicaron un caso de estudio en la página de Kubernetes, accesible en [33]; y un artículo en Cloud Native Computing Foundation, en [34].

En estos artículos se presenta el problema que tenía Wink y lo que querían lograr. Buscaban construir una infraestructura con baja latencia y alta disponibilidad y confiabilidad para poder servir las comunicaciones entre los millones de dispositivos conectados a Wink. La solución a la que llegaron fue implementar el *stack* Kubernetes-Docker-CoreOS[4] para montar una red de microservicios.

[4]CoreOS es un sistema operativo *open-source* basado en el *kernel* de Linux y pensado para utilizarse en los nodos de un *cluster*, como por ejemplo, un *cluster* de Kubernetes

3.6.2. Usos de la domótica en la actualidad

Siguiendo con el ejemplo de Wink, y añadiendo productos similares de otras empresas, como SmartThings de Samsung, en esta sección se presenta el estado actual de la domótica a nivel de uso.

En primer lugar, para que los usuarios puedan conectar sus dispositivos inteligentes, es necesario que dispongan de un concentrador o *hub* (como Wink Hub), que es un aparato que está conectado a los servidores del servicio correspondiente y a los dispositivos del usuario por medio de la red, de manera que actúa de intermediario.

Por otro lado, los usuarios disponen de una aplicación móvil conectada al proveedor y realizan sus gestiones directamente desde la aplicación, de forma que esas gestiones viajan hasta el *hub* de su vivienda, que está conectado a sus dispositivos.

La siguiente Fig. 3.7 muestra la gran variedad de sensores y aparatos inteligentes que se pueden conectar a Wink por medio del Wink Hub (representado también en la figura). Para ello, la empresa ofrece una lista de dispositivos compatibles con su sistema. Por ejemplo, bombillas de Philips o incluso asistentes personales como Amazon Alexa o Google Assistant [35].

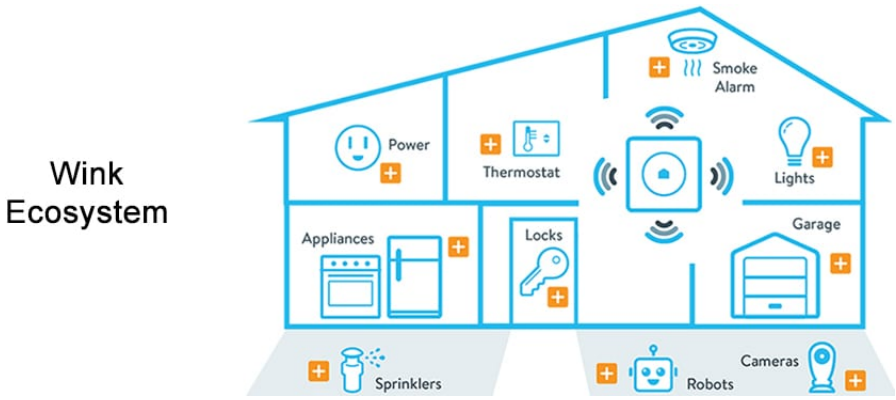


Fig. 3.7. Vivienda inteligente gestionada con Wink [34]

Entre las utilidades de estos sistemas de domótica, destaca la posibilidad de configurar un hogar inteligente de manera personalizada, creando escenarios y programaciones de tareas [36]. Con esto, se consigue una vivienda sostenible, además de ahorrar energía, ya que los recursos se pueden configurar según las necesidades del usuario.

Un ejemplo claro de tarea programada puede ser subir las persianas o encender las luces a una determinada hora. Y un escenario bastante común, puede ser un entorno de cine utilizando una *smart* TV y una iluminación tenue.

Todo esto se realiza mediante la aplicación móvil en unos pocos pasos, o incluso hablando con un asistente personal que esté conectado al *hub*, lo cual hace que la experiencia de usuario sea excelente.

Para lograr esto, se debe cumplir además que el retardo de tiempo desde que se da la instrucción hasta que se realiza la acción no sea excesivo. Este parámetro es lo que consiguió optimizar Wink mediante el *stack* Kubernetes-Docker-CoreOS; en concreto, lograron un retardo de menos de 200 milisegundos en encender una luz desde la aplicación móvil, cerrar una puerta o cambiar la temperatura de un termostato, según se dice en [33].

Capítulo 4

Definición del trabajo

En este capítulo se explica el porqué de la realización de este proyecto. Se describirán la motivación, los objetivos a conseguir, la metodología utilizada y la planificación y estimación económica del proyecto.

4.1. Motivación

El propósito de este proyecto es analizar la viabilidad de un caso de uso muy específico. Sin embargo, este proyecto está motivado por el uso de tecnologías muy recientes e innovadoras de las que, sin duda, se hablará en un futuro no muy lejano.

Primeramente, el desarrollo web está en auge. Actualmente, gran parte de las empresas disponen de una página web moderna, con una interfaz de usuario excelente. Las empresas necesitan publicitarse en la red, y qué mejor manera de realizarlo con una página web (o una *app* móvil) que atraiga la atención de un cliente potencial.

En segundo lugar, la programación en la nube es una tecnología que están implementando hoy en día muchas empresas para almacenar sus propios servicios, ya que proporciona una capa de abstracción de su sistema y no tienen que preocuparse en gran medida por su gestión. Este proceso de migración a los servicios en la nube forma parte de la transformación digital de las empresas.

Kubernetes es una tecnología que ofrecen prácticamente todos los proveedores de

servicios en la nube, de ahí que sea algo importante. Este sistema se utiliza en conjunción con Docker para montar arquitecturas de microservicios, principalmente. El uso de microservicios ha aumentado por parte de las empresas tecnológicas, ya que proporcionan un gran número de ventajas, como se vio anteriormente. Está claro que las últimas tendencias en desarrollo de *software* giran en torno a la orquestación de contenedores y a las arquitecturas de microservicios.

Finalmente, también está adquiriendo mucha importancia la tecnología IoT. De hecho, desde el punto de vista de la ciberseguridad, es una tecnología que está siendo muy analizada. No obstante, el caso de uso que se presenta en este proyecto tiene que ver con la domótica, ámbito que también está en continuo avance.

En este proyecto se desarrolla una aplicación que integra funciones de *Cloud Computing*, de IoT y de domótica, para demostrar que es una buena forma de implementar un sistema de control de sensores. Como estas tecnologías son muy novedosas y de interés para la comunidad de desarrolladores, la realización de este proyecto puede ser de gran utilidad para conocer su estado y madurez al implementarlas en una aplicación concreta.

4.2. Objetivos

En el presente trabajo, se analizará la viabilidad de utilizar una arquitectura de microservicios para conectar con una red de sensores IoT, utilizando una aplicación de domótica como caso de uso concreto. Los objetivos principales del proyecto son:

- Construir una arquitectura de microservicios: Se tratarán de aprovechar al máximo las ventajas que ofrecen los microservicios para poder analizar si este tipo de arquitectura es más beneficiosa que una arquitectura monolítica para el caso de uso seleccionado.

- Configurar una red de sensores IoT: Se utilizarán sensores adecuados para la domótica y se conectarán a un Arduino. Este enviará los datos de los sensores a sus correspondientes microservicios mediante una red WiFi.

- Diseñar un *dashboard* de control: Se implementará una página web en la que el usuario de la red de sensores pueda consultar y gestionar los datos que procesan los distintos microservicios de manera intuitiva. Para el diseño de esta funcionalidad se usará Angular.

4.3. Metodología

Para este proyecto, se utilizará una metodología ágil, como es frecuente en el mundo del desarrollo de *software*. De esta manera, el desarrollo del proyecto se estructura en ciclos o iteraciones conocidos como *sprints*. En cada *sprint*, se tratará de abordar una funcionalidad y se realizará una documentación sobre la misma, para facilitar la posterior etapa de pruebas y corrección de errores.

Una de las ventajas de la metodología ágil es la velocidad de desarrollo de *software* que se impone para disponer de un producto mínimo viable al final de cada *sprint*.

4.4. Planificación y estimación económica

Respecto a la planificación del proyecto, esta se muestra de forma visual mediante un diagrama de Gantt en la siguiente Fig. 4.1.

Fig. 4.1. Planificación del proyecto

Cabe mencionar que los tiempos establecidos para cada tarea son flexibles, en el sentido de que hay tareas que se iniciaron antes de lo estipulado y tareas que continuaron más allá de la fecha de finalización indicada. No obstante, este dia-

grama de Gantt sí describe los pasos seguidos para la realización del proyecto y la importancia de cada tarea, de manera orientativa.

Por otro lado, se ha utilizado Trello para la gestión de tareas concretas. Trello es una herramienta de gestión de proyectos que permite crear tableros y tarjetas de distinta temática. En este sentido, se crearon listas de tareas por hacer, tareas en proceso y tareas hechas, para poder conocer el progreso del proyecto.

Respecto a la estimación económica, se han tenido en cuenta los costes del *hardware* utilizado y de los trabajadores. Los costes de *software* son nulos, ya que se han usado tecnologías *open-source*. La Tabla 4.1 muestra los costes considerados.

Tabla 4.1. Estimación de los costes del proyecto

Concepto	Justificación	Coste (€)
Ordenador	MacBook Pro (13-inch, 2017), macOS Catalina, procesador Intel i5, memoria RAM de 8 GB, disco duro de 256 GB	2 200.00
Arduino UNO WiFi Rev2	Microcontrolador con conexión WiFi que gestiona la red de sensores	38.90
Sensor de temperatura	Para medir la temperatura ambiente	2.90
Sensor de humedad	Para medir la humedad de la tierra	4.90
Raspberry Pi 4	Utilizada para desplegar la arquitectura de microservicios desarrollada en un *cluster* de Kubernetes con MicroK8s	64.99
Accesorios de Raspberry Pi	Cable de alimentación, disipadores, ventilador de refrigeración, carcasa	16.99
Tarjeta microSD de 64 GB	Necesaria para cargar una imagen de Ubuntu en la Raspberry Pi	9.95
Cable micro-HDMI a HDMI	Se utilizó para conectar la Raspberry Pi a un monitor y poder realizar la configuración inicial	6.99
Cable Ethernet	Utilizado para conectar la Raspberry Pi a Internet	2.90
Desarrolladores	Un desarrollador para llevar a cabo el proyecto, con un coste de 50 € por hora	20 000.00
Software	Programas y tecnologías utilizados	0.00
Total		**22 348.52**

Capítulo 5

Sistema desarrollado

En este capítulo se explica en detalle el sistema desarrollado en el presente proyecto.

En primer lugar, se explicará la arquitectura de microservicios implementada, indicando los distintos procesos y conexiones existentes, así como su despliegue con Docker y Kubernetes.

En segundo lugar, se analizarán los diferentes sensores utilizados en la placa de Arduino para poder obtener distintas mediciones desde la aplicación desarrollada.

Y, por último, se describirán las funcionalidades más relevantes de la aplicación desarrollada, poniendo especial atención en los aspectos de diseño e implementación de las distintas tecnologías involucradas.

5.1. Arquitectura de microservicios

En la Fig. 5.1 se muestra el esquema diseñado. Cada una de estas entidades forman parte del *cluster* de Kubernetes, a excepción de los microcontroladores (representados mediante el logotipo de Arduino).

El único acceso al *cluster* es la aplicación web de Angular. Esto se debe a que el `Service` asignado al `Deployment` de Angular en Kubernetes tiene configurado un `NodePort` que expone un puerto del nodo por el cual se puede acceder al *cluster*.

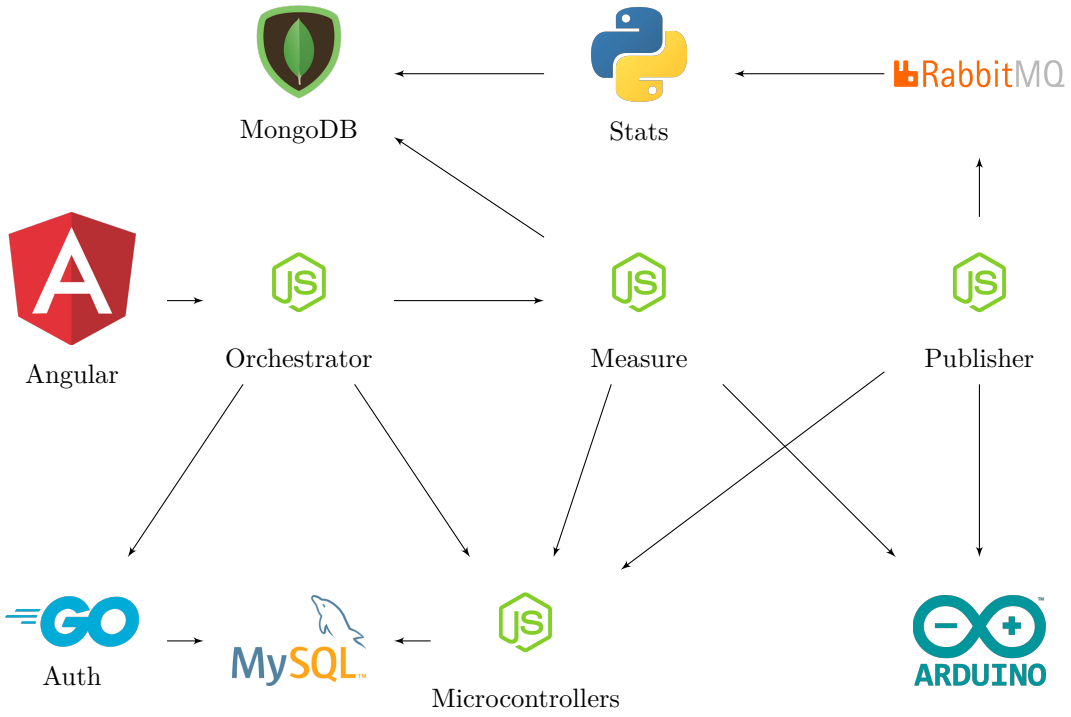


Fig. 5.1. Diseño de la arquitectura de microservicios implementada

Con esto, se consigue que el *cluster* esté protegido contra el acceso no autorizado, ya que no es posible acceder a microservicios como MySQL o MongoDB directamente desde el exterior.

El microservicio denominado Orchestrator hace la función de orquestador o API *gateway* (término bastante común en arquitecturas de microservicios). Esto es una entidad que sirve de entrada a la arquitectura de microservicios y redirige los mensajes que recibe al microservicio correspondiente [37]. Este microservicio Orchestrator es la única entrada al *back-end* disponible para la aplicación de Angular (es decir, el *front-end*). Así, se consigue proteger los demás microservicios, ya que están detrás del API *gateway* y solo son accesibles desde este.

Por otro lado, se ha implementado un microservicio por gestionar las distintas magnitudes medidas y funcionalidades de la placa de Arduino (por ejemplo, valores de temperatura y humedad o posibilidad de encender y apagar un LED). Este microservicio, llamado Measure, tiene varias conexiones: una conexión con la placa de Arduino (vía direccionamiento IP), una conexión con la base de datos de

MongoDB, una conexión con el microservicio Microcontrollers y una conexión con el microservicio Orchestrator.

El microservicio llamado Auth se encarga de controlar la autenticación y el registro de usuarios, conectándose a una base de datos de MySQL. Esta base de datos es accedida también por el microservicio Microcontrollers para gestionar los microcontroladores de cada usuario.

Por otro lado, está el microservicio Publisher, que se encarga de recopilar las mediciones de todos los microcontroladores almacenados en la base de datos de MySQL y publicar estos datos en distintas colas de RabbitMQ. Esta tarea se realiza periódicamente, cada minuto.

Existe otro microservicio, denominado Stats, que se encarga de tomar los mensajes de las colas de RabbitMQ, calcular estadísticas en función del tipo de magnitud y almacenar los resultados en la base de datos de MongoDB.

Los motivos de la elección de las distintas tecnologías y lenguajes de programación empleados se explicaron en el Capítulo 2.

5.2. Implementación en Kubernetes

Una vez diseñada la arquitectura de microservicios, para implementarla es necesario generar imágenes de Docker para cada microservicio y escribir los manifiestos de Kubernetes.

5.2.1. Construcción de imágenes de Docker

En este punto, es necesario escribir varios archivos `Dockerfile` y utilizar los comandos explicados en la Sección 2.1.

En el microservicio de Angular, lo que se hace es utilizar una imagen del servidor web NGINX y añadirle la configuración del servidor y los archivos estáticos de la aplicación de Angular (es decir, los archivos HTML, CSS, JS y demás), optimizados previamente para producción mediante el comando `ng build --prod` de Angular CLI.

Para los microservicios de RabbitMQ y MongoDB, simplemente se utilizan sus imágenes de Docker oficiales. En cambio, para MySQL, es necesario escribir un archivo SQL para inicializar la base de datos y los esquemas de las tablas. Este fichero SQL se introduce en el contenedor de MySQL a través de un `Dockerfile`, por lo que hay que crear una imagen de MySQL personalizada que contenga estos datos al inicializarse.

En los demás microservicios, la construcción de imágenes se limita a insertar los códigos fuente en el contenedor, instalar las dependencias y compilarlos o ejecutarlos (dependiendo de la tecnología utilizada), mediante un `Dockerfile`.

Las imágenes resultantes se han de subir a Docker Hub para que Kubernetes pueda utilizarlas, aunque existe una manera de que Kubernetes utilice imágenes de Docker locales, pero no es lo más común.

5.2.2. Manifiestos de Kubernetes

Para cada microservicio es necesario generar un `Deployment` o `StatefulSet`, para especificar los `Pods`, y un `Service` para poder conectarlos en el *cluster*.

Así, para los microservicios de MongoDB, MySQL y RabbitMQ conviene utilizar `StatefulSets`, ya que son microservicios importantes para el funcionamiento de la aplicación. Y, para los demás, se utilizan `Deployments`. En estos `Deployments` y `StatefulSets` se indican las imágenes de Docker anteriormente comentadas.

Todos los `Services` escritos son muy parecidos. En el `Service` de Angular y NGINX se especifica que es de tipo `NodePort`, para poder exponer el *cluster* de Kubernetes.

Para persistir datos en los microservicios de MongoDB y MySQL, es necesario declarar un `PersistentVolumeClaim` para cada uno. En el Código 5.1 se muestra el `PersistentVolumeClaim` de MongoDB, el cual solicita una capacidad máxima de 1 GiB.

Por otro lado, existen ciertas configuraciones que no deberían estar escritas directamente en el código de los microservicios. Para no tener estas configuraciones *hard-coded*, se utilizan variables de entorno.

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mongo-pv-claim
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: standard
  volumeMode: Filesystem
```

Código 5.1. `PersistentVolumeClaim` para MongoDB

Kubernetes, por defecto, proporciona una serie de variables de entorno a los `Pods` con información de todos los `Services` existentes en el *cluster* (por ejemplo, los puertos de escucha). Sin embargo, existen otras variables de entorno que serían muy útiles para los microservicios. Mediante un `ConfigMap` se pueden especificar variables de entorno y que los `Pods` utilicen aquellas que necesiten. En el Código 5.2 se muestran las variables de entorno consideradas.

```yaml
apiVersion: v1
data:
  ANGULAR_MS_HOSTNAME: angular-ms
  AUTH_MS_HOSTNAME: auth-ms
  MEASURE_MS_HOSTNAME: measure-ms
  MICROCONTROLLERS_MS_HOSTNAME: microcontrollers-ms
  MONGO_DATABASE_NAME: iot
  MONGO_HOSTNAME: mongo
  MYSQL_DATABASE_NAME: iot
  MYSQL_HOSTNAME: mysql
  ORCHESTRATOR_MS_HOSTNAME: orchestrator-ms
  QUEUE_HUMIDITY_NAME: humidities
  QUEUE_LIGHT_NAME: lights
  QUEUE_TEMPERATURE_NAME: temperatures
  RABBITMQ_HOSTNAME: rabbitmq
  STATS_MS_HOSTNAME: stats-ms
kind: ConfigMap
metadata:
  name: env-configmap
  namespace: default
```

Código 5.2. `ConfigMap` con variables de entorno

Por ejemplo, el microservicio Auth necesita las variables de `$MYSQL_HOSTNAME` y `$MYSQL_DATABASE_NAME`, que se corresponden con el nombre de DNS del `Service` de MySQL y el nombre de la base de datos, respectivamente. El puerto del `Service` está guardado en una variable `$MYSQL_SERVICE_PORT`, y es de las que provee Kubernetes por defecto.

```
apiVersion: v1
data:
  MONGO_INITDB_ROOT_USERNAME: cm9vdA==
  MONGO_INITDB_ROOT_PASSWORD: c2VjcmV0
  MYSQL_ROOT_PASSWORD: bXktc2VjcmV0LXB3
  MYSQL_ROOT_USERNAME: cm9vdA==
  RABBITMQ_DEFAULT_USER: dXNlcg==
  RABBITMQ_DEFAULT_PASS: cGFzc3dvcmQ=
kind: Secret
metadata:
  name: secrets
  namespace: default
type: Opaque
```

Código 5.3. `Secret` con credenciales de acceso

Otros parámetros que conviene especificar como variables de entorno son las credenciales de acceso a las bases de datos y a RabbitMQ. Para ello, en vez de usar un `ConfigMap`, se utiliza un `Secret`, el cual no presenta estas credenciales de forma explícita, sino codificadas en Base64. El Código 5.3 contiene estas credenciales en un `Secret`.

Así, el microservicio Auth utilizará las variables de `$MYSQL_ROOT_USERNAME` y `$MYSQL_ROOT_PASSWORD` para conectarse a MySQL. Estas variables estarán disponibles en el `Pod` ya decodificadas.

5.3. Sensores IoT

En esta sección se presentan los sensores utilizados en la placa de Arduino para conectarlos a la aplicación web de Angular por medio de la arquitectura de microservicios.

Aunque los sensores reciban el apellido de IoT (*Internet of Things*), estos no se conectan directamente a Internet, sino que necesitan de un microcontrolador con un módulo de conexión para ello. En este caso, el microcontrolador elegido es una placa de Arduino, en concreto, Arduino Uno WiFi Rev2, disponible en [15].

Cabe mencionar que no es necesario que la placa sea de Arduino, es posible utilizar cualquier otro microcontrolador o dispositivo que tenga de una antena WiFi para conectarse por HTTP a los microservicios.

5.3.1. Sensor de temperatura

Para medir la temperatura ambiente, se ha utilizado un sensor llamado *Grove - Temperature Sensor* [17], que utiliza un termistor para realizar la medida.

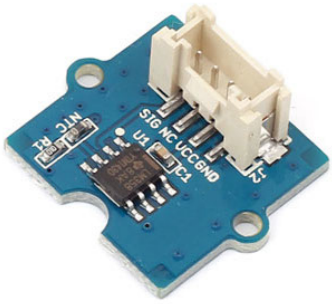


Fig. 5.2. Sensor *Grove - Temperature Sensor*

El rango de operación de este sensor es de $[-40, 125]$ °C y tiene una precisión de $\pm 1'5$ °C. Estos parámetros son aceptables para el entorno en el que se va a utilizar.

El sensor se conecta a la placa de Arduino a través de un puerto analógico (además de alimentarlo a una tensión entre 3'3 y 5 voltios y conectarlo a tierra). Por este motivo, el valor que se lee desde el Arduino es un valor de 10 bits (un valor decimal entre 0 y 1023).

En la página web del componente se muestra una función para transformar el valor digital leído del puerto analógico a un valor de temperatura en grados centígrados. Esta función es la siguiente:

$$T(a) = \frac{1}{\frac{1}{B} \cdot \ln\left(\frac{1023}{a} - 1\right) + \frac{1}{298'15}} - 273'15 \tag{5.1}$$

Donde a es el valor analógico leído por el Arduino; y B representa una constante del termistor cuyo valor está entre 4250 y 4299 K. Una estimación razonable para esta constante supone un valor de 4275 K. La gráfica de esta función se muestra en la Fig. 5.3.

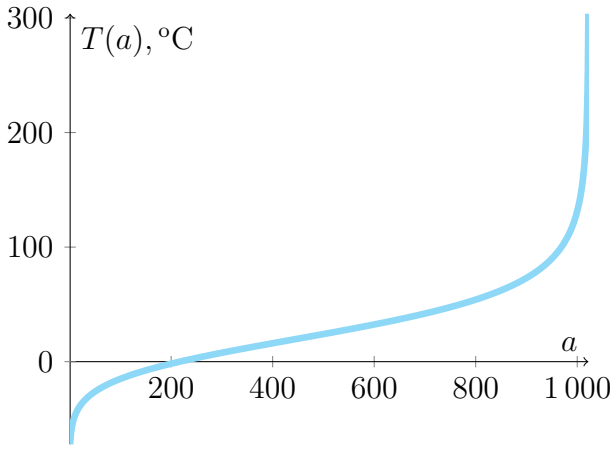


Fig. 5.3. Función de conversión de temperatura

5.3.2. Sensor de humedad

Otro sensor utilizado es el *Grove - Moisture Sensor* [18], que se puede utilizar para medir la humedad del suelo (por ejemplo, de la tierra de una maceta).

Como se puede ver en la Fig. 5.4, el sensor está compuesto por dos sondas metálicas que permiten que la corriente circule a través del suelo, para poder calcular el valor de una resistencia que mide la cantidad de humedad presente en el suelo.

Según el valor que reciba el Arduino por el puerto analógico, se puede distinguir si el suelo está seco, húmedo o mojado. Según las características del dispositivo presentes en su página web, se tienen los rangos de valores expuestos en la Tabla 5.1.

Tabla 5.1. Rango de valores de humedad en función del tipo de suelo

Estado del suelo	Valor mínimo	Valor máximo
Seco	0	300
Húmedo	300	700
Mojado	700	950

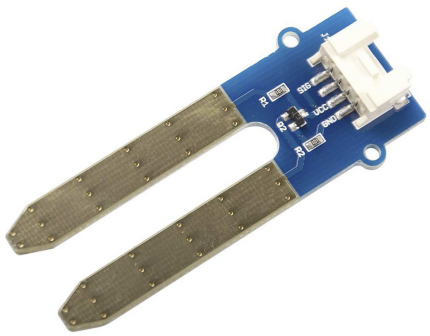


Fig. 5.4. Sensor *Grove - Moisture Sensor*

5.4. Obtención de mediciones en tiempo real

Se ha considerado como funcionalidad principal de la aplicación la de obtener el valor de una determinada medición en tiempo real.

En el siguiente diagrama de secuencia se muestran los distintos microservicios que intervienen y la comunicación necesaria entre ellos para lograr la funcionalidad descrita, en este caso, para obtener la temperatura en tiempo real (ver Fig. 5.5).

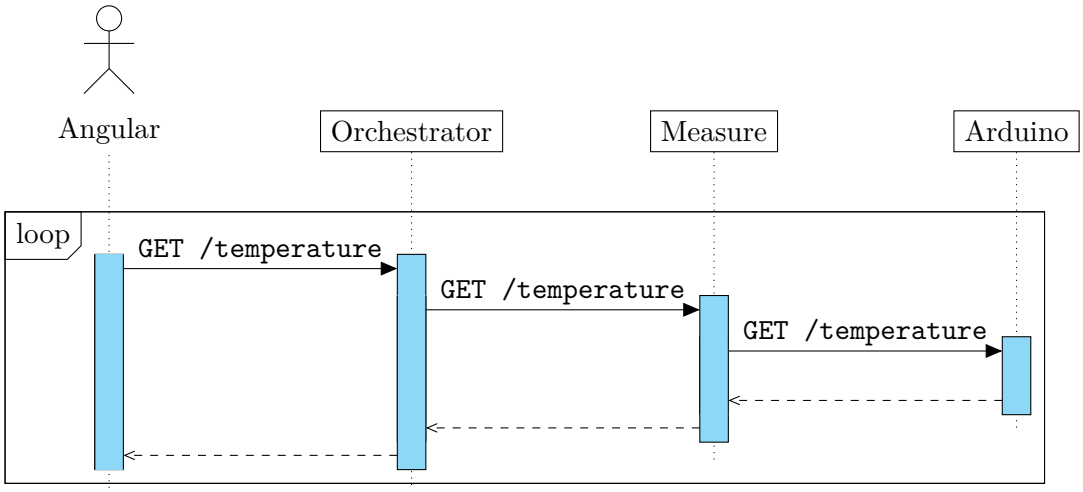


Fig. 5.5. Obtención de la temperatura en tiempo real

Cada una de las interacciones que se presentan en el diagrama anterior se realizan mediante HTTP (es decir, son peticiones y respuestas HTTP). Como se vio anteriormente, una de las ventajas del uso de microservicios es que las comunicaciones se realizan con HTTP, el cual es un protocolo de comunicación muy ligero.

Desde la aplicación web de Angular se realizan estas peticiones utilizando AJAX (*Asynchronous JavaScript And XML*). Esta manera de utilizar HTTP permite cargar información procedente del *back-end* de manera asíncrona, sin tener que recargar la página (es decir, sin volver a procesar archivos HTML y CSS, archivos de JavaScript, imágenes y tipografías). Con esto se consigue que las peticiones y respuestas contengan menos datos y, por tanto, sean más rápidas en transmisión y procesamiento.

Para realizar las peticiones HTTP desde los microservicios Orchestrator y Measure, se utiliza una librería de Node.js llamada *axios*. No obstante, para que la comunicación sea exitosa, es necesario que el microservicio Measure pueda localizar la placa de Arduino. Por este motivo, es necesario que el microservicio Measure disponga de la dirección IP o una dirección de internet accesible.

5.4.1. Comunicación con Arduino

Para que los microservicios puedan comunicarse con la placa de Arduino, esta tiene que actuar como un servidor HTTP.

En la función `setup()` del *sketch* Arduino se configura la conexión WiFi y la dirección IP, además de inicializar el servidor. Por otro lado, en la función `loop()` la placa se queda escuchando en el puerto 80 (puerto TCP asignado al HTTP por defecto) hasta recibir una petición.

En el momento en el que se recibe una nueva petición, el programa lee su contenido y utiliza las funciones del Código 5.4 para enviar la respuesta.

En primer lugar, con la función `handleGetRequest()` se analiza la conocida como *Request-Line* (que contiene el método, la URI y la versión de HTTP), que deberá coincidir con alguna de las rutas establecidas para poder decidir qué sensor utilizar (y, por tanto, saber en qué pin está conectado).

En segundo lugar, se envía la respuesta con la función `sendResponse()`. De esta

manera, si el pin de la placa no es conocido, se envía un error "404 Not Found"; y si el pin es válido, se envía el valor medido en formato JSON, junto con un código "200 OK". A continuación se puede ver un ejemplo:

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Connection: keep-alive
Content-Type: application/json

{"temperature":503}
```

Nótese que las cabeceras HTTP tienen que ser escritas por cuenta propia en el *sketch* de Arduino, manteniendo el formato descrito en la RFC 2616 (por ejemplo, ha de existir un salto de línea entre las cabeceras y el cuerpo en un mensaje HTTP).

```
boolean handleGetRequest(String line, int* pindex) {
  for (int i = 0; i < NUM_ENDPOINTS; i++) {
    if (line.equals("GET /" + MEASURES[i] + " HTTP/1.1")) {
      *pindex = i;
      break;
    }
  }

  return *pindex != -1;
}

void sendResponse(WiFiClient client, int index) {
  if (index == -1) printHttpError(client);
  else if (index == 2) printHttpResponse(client, digitalRead(PINS[index]), MEASURES[index]);
  else printHttpResponse(client, analogRead(PINS[index]), MEASURES[index]);
}

void printHttpHeaders(WiFiClient client, int httpStatus) {
  if (httpStatus == 200) client.println("HTTP/1.1 200 OK");
  if (httpStatus == 404) client.println("HTTP/1.1 404 Not Found");

  client.println("Access-Control-Allow-Origin: *");
  client.println("Connection: keep-alive");
  client.println("Content-Type: application/json\n");
}

void printHttpError(WiFiClient client) {
  printHttpHeaders(client, 404);
  client.println("{\"error\":404,\"message\":\"Not Found\"}");
}

void printHttpResponse(WiFiClient client, int value, String measure) {
  printHttpHeaders(client, 200);
  client.print("{\"" + measure + "\":" + value + "}");
}
```

Código 5.4. Utilización de Arduino como servidor HTTP

5.4.2. Descubrimiento de microcontroladores

Como se dijo anteriormente, los microservicios dedicados a gestionar las mediciones de la placa de Arduino necesitan de su dirección IP para poder comunicarse.

En un primer momento se utilizó una variable de entorno en dichos microservicios con el valor de la dirección IP, de manera que pudiera ser accedida desde el código. Posteriormente, junto con el servicio de usuarios se almacenó en base de datos toda la información relativa a los microcontroladores pertenecientes a los distintos usuarios, junto con sus direcciones IP.

Esta nueva manera de obtener el valor de la dirección IP de la placa de Arduino hizo que la aplicación web pudiera ser utilizada por varios usuarios. Además, se tuvo en cuenta que un usuario puede disponer de varias placas de Arduino (o cualquier otro tipo de microcontrolador o dispositivo que pueda actuar como servidor web) con varios sensores conectados.

Esta información permanece almacenada en una tabla de la base de datos de MySQL llamada MICROCONTROLLERS. Esta tabla se relaciona con otra denominada USERS de la forma presentada en la Fig. 5.6. El significado de esta relación es que un usuario puede tener varios microcontroladores (o ninguno). Esto se debe a que `username` es clave foránea con respecto a la tabla USERS.

Por otro lado, la clave primaria de la tabla MICROCONTROLLERS es el conjunto (`ip`, `measure`). Se entiende que cada microcontrolador debe tener una dirección IP única, y que un microcontrolador solamente podrá disponer de un sensor para cada magnitud medida. Por este motivo, el conjunto (`ip`, `measure`) no podrá repetirse.

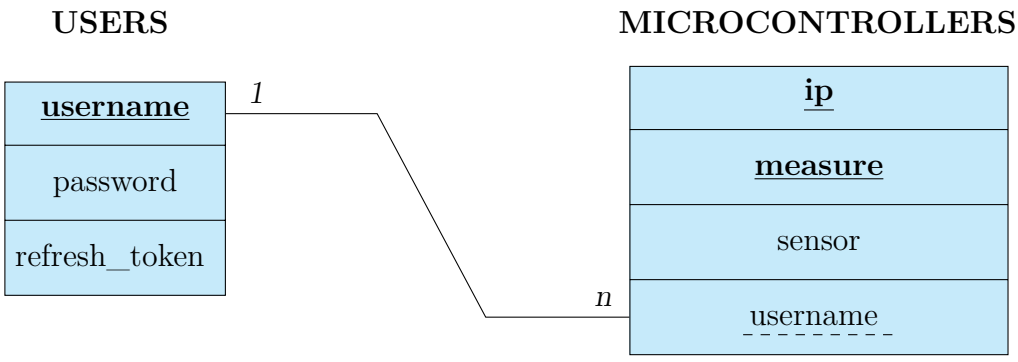


Fig. 5.6. Relaciones entre las tablas de la base de datos de MySQL

Con esta solución, los microservicios como Measure y Publisher se comunicarán con el microservicio Microcontrollers para obtener una lista de los microcontroladores con algún sensor concreto, como se muestra en la Fig. 5.7.

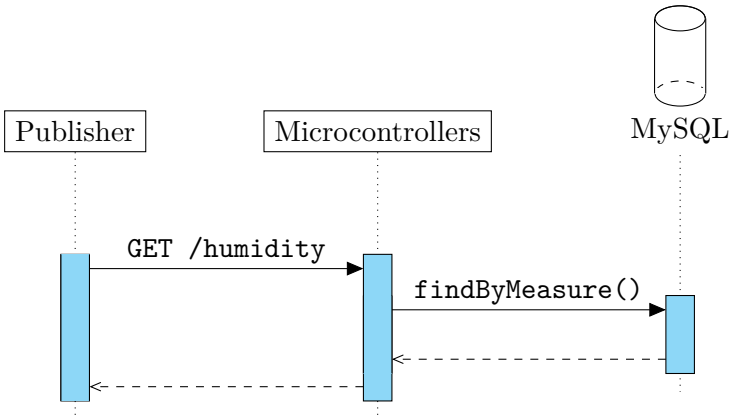


Fig. 5.7. Descubrimiento de microcontroladores con sensor de humedad

Este flujo de peticiones HTTP se realizará cada vez que los microservicios Measure o Publisher necesiten conocer los datos de los microcontroladores para llevar a cabo las peticiones. De hecho, en el flujo de la Fig. 5.5, anteriormente comentado, se ha omitido este paso intermedio por parte del microservicio Measure.

Como el número de inserciones será mucho menor que el número de consultas a la base de datos, para que el flujo sea más eficiente es conveniente implementar una memoria caché en el microservicio Microcontrollers. Así, las peticiones sucesivas serán respondidas rápidamente. Cabe mencionar que la memoria caché ha de borrarse cada vez que se inserte, se actualice o se elimine un microcontrolador de la tabla de la base de datos.

Desde otro punto de vista, una vez iniciada la sesión en la aplicación de Angular, el usuario dispone de una lista de sus microcontroladores y las mediciones realizadas por estos. Para ello, la aplicación debe obtener una lista de los microcontroladores del usuario. El flujo de mensajes HTTP se muestra en la Fig. 5.8.

Como se puede observar, la aplicación de Angular llama al microservicio Orchestrator (ya que es el API *gateway*) y este redirige la petición al microservicio Microcontrollers, que es el que gestiona la tabla de MySQL relativa a los microcontroladores de los usuarios.

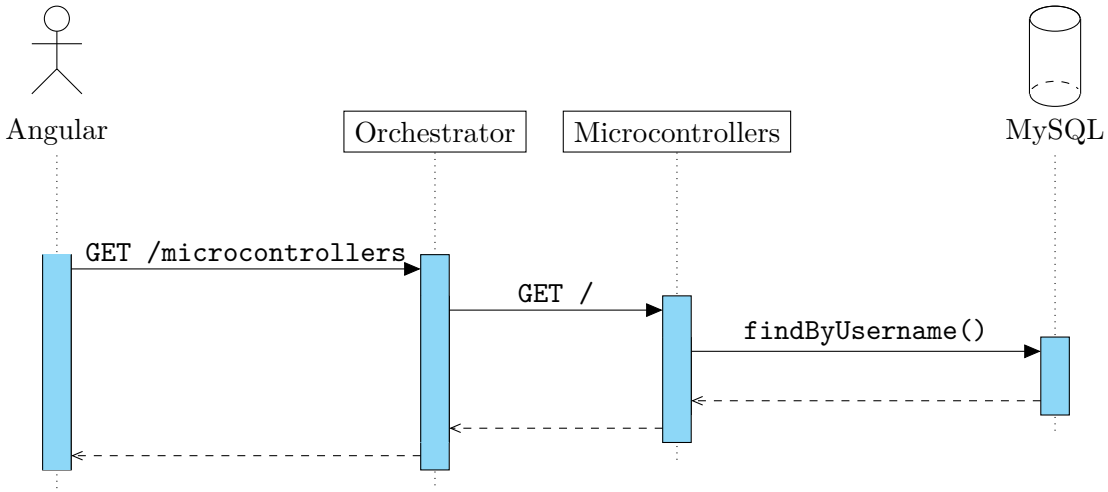


Fig. 5.8. Descubrimiento de microcontroladores de un usuario

5.4.3. Visualización de datos

Para implementar una visualización de datos amigable y útil para el usuario, se ha utilizado la API de Google Charts. Esta librería provee una gran cantidad de tipos de diagramas y gráficos, así como múltiples opciones configurables por cada uno de estos. Algunos ejemplos de Google Charts se muestran en la Fig. 5.9.

Sin duda, la interfaz y la experiencia de usuario que ofrece Google Charts son excelentes. Se trata de una API en la que el desarrollador simplemente provee los datos y las configuraciones del gráfico, y las funciones de la API ya realizan el resto. La documentación de Google Charts es pública y está disponible en [38].

Esta librería está pensada para usarse en una página web, mediante JavaScript. Sin embargo, existen opciones para integrar Google Charts en una aplicación de Angular (que está desarrollada en TypeScript), como la librería *angular-google-charts* o la librería *ng2-google-chart*.

Los datos que se mostrarán mediante Google Charts son las distintas mediciones realizadas por los sensores de la placa de Arduino, ya sean medidos en tiempo real o sean datos anteriores guardados en base de datos. Los principales diagramas que se utilizan son los denominados `AreaChart`, `LineChart` y `GaugeChart`.

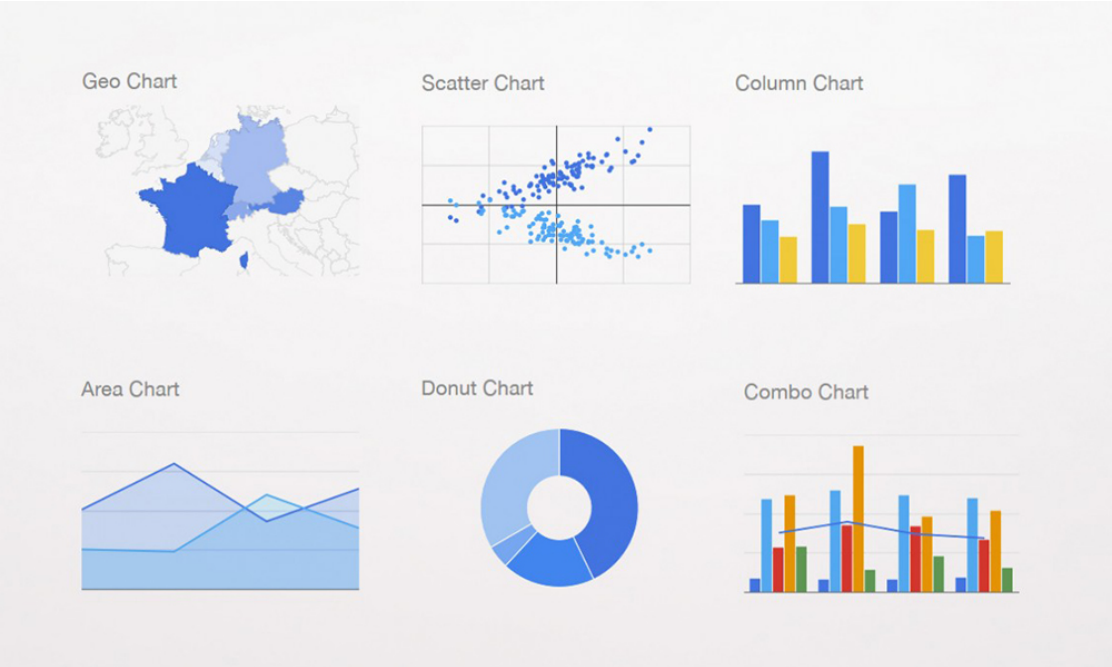


Fig. 5.9. Ejemplo de gráficos de Google Charts

5.4.4. Componentes y servicios de Angular

Para conseguir la visualización de datos anterior, desde Angular es necesario estructurar la aplicación en componentes y servicios.

Un componente de Angular representa una parte visual de la página web y está asociado a un controlador que se maneja con TypeScript. Por otro lado, un servicio de Angular es un objeto que se instancia al iniciar la aplicación (utiliza internamente el patrón de diseño Singleton) y está pensado para proveer datos y demás funcionalidades a los distintos componentes de la aplicación.

Así, para implementar la visualización de datos en tiempo real se han utilizado los siguientes objetos de Angular:

- `DashboardComponent`: Este componente representa el *dashboard* completo del usuario, y es contenedor de los siguientes componentes, es decir, es el componente padre.

- `DashboardMicrocontrollerComponent`: Contiene los datos relativos a un microcontrolador, como son el nombre del sensor, la magnitud de medida, la dirección IP del microcontrolador y el estado de su conexión.

- `HumidityChartComponent`: En este componente se introduce el componente de `GoogleChart` adecuado para la magnitud (`GaugeChart` para la humedad, en concreto) y se controlan los datos para dibujar las gráficas.

- `HumidityStatsComponent`: Este componente calcula estadísticas (de humedad, en este caso) según se van recibiendo datos de un sensor.

- `ArduinoService`: Se trata de un servicio que realiza las distintas llamadas HTTP al API *gateway* y contiene la lista de microcontroladores del usuario.

La interacción entre estos objetos de Angular se muestra en el diagrama de la Fig. 5.10.

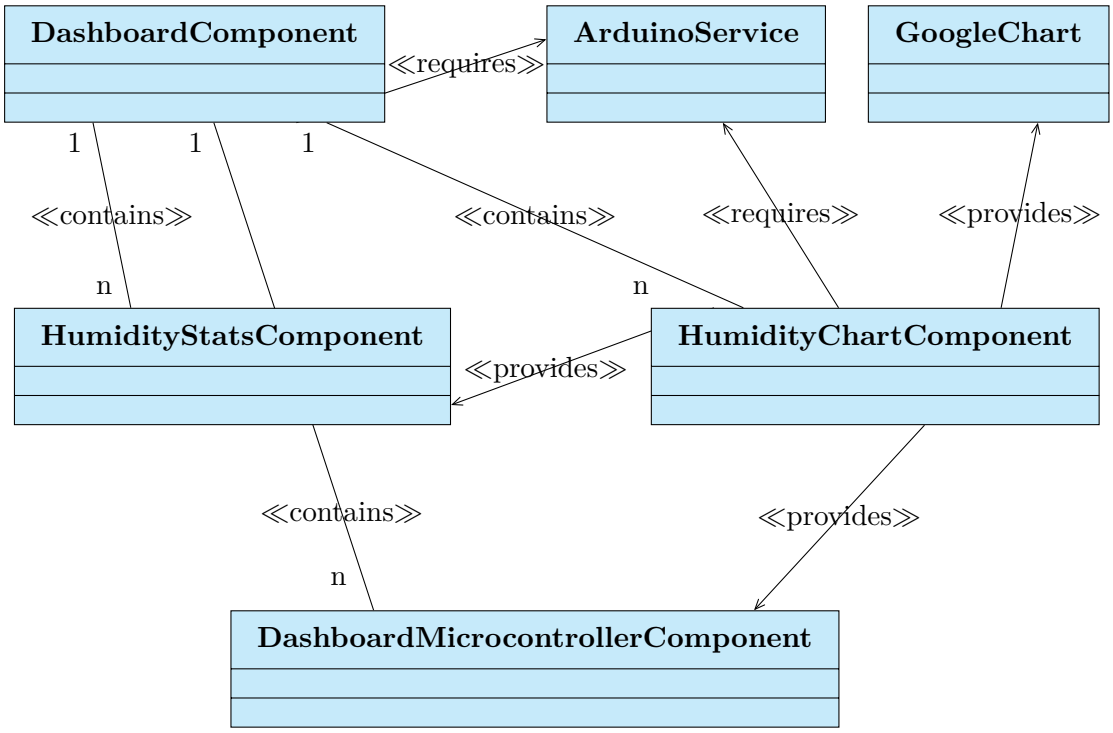


Fig. 5.10. Interacción entre objetos del *dashboard*

Como puede observarse, el componente `DashboardComponent` es el padre de los otros componentes del *dashboard*, pudiendo contener varios de estos componentes.

Por otro lado, el servicio `ArduinoService` es requerido por `DashboardComponent` (para obtener la lista de microcontroladores) y por `HumidityChartComponent` (para realizar las peticiones al microservicio Orchestrator referentes a un microcontrolador concreto).

Finalmente, el componente `HumidityChartComponent` es el que provee los datos obtenidos del servicio `ArduinoService` a sus componentes hijo (`GoogleChart`) y hermanos (`HumidityStatsComponent` y `DashboardMicrocontrollerComponent`) de manera que cada componente pueda llevar a cabo su acometido con la información provista.

Gracias al funcionamiento de Angular, cuando se actualiza un valor en un componente padre, este se actualiza automáticamente en los componentes hijos, por medio de distintos modos de comunicación entre componentes.

5.4.5. Utilización de NGINX

NGINX es el servidor HTTP utilizado para alojar la aplicación web de Angular. Como se indicó anteriormente, NGINX permite redirigir peticiones HTTP. Para realizar esto, se utiliza un archivo de configuración como el que aparece en el Código 5.5 (fragmento del archivo implementado en realidad).

Estas redirecciones son necesarias para que la página web pueda comunicarse con el *cluster* de Kubernetes. El microservicio de Angular es el único que tiene un `NodePort`, por lo que es el único accesible desde el exterior del *cluster*. Por este motivo, si se realizan peticiones al microservicio Orchestrator, estas van a fallar.

La manera de solucionarlo es redirigir las peticiones en el servidor NGINX. Con esto, se consigue que la petición llegue desde el navegador hasta NGINX y sea este el que realice la petición al microservicio Orchestrator, de manera que la comunicación tenga lugar dentro del *cluster*.

Otra solución válida habría sido configurar el microservicio Orchestrator como `NodePort` en el *cluster* de Kubernetes y sacar la aplicación web afuera del *cluster*.

```
upstream orchestrator-ms {
  server orchestrator-ms;
}

server {

  listen 80;

  location / {
    root /usr/share/nginx/html/;
    index index.html;
    try_files $uri $uri/ /index.html;
  }

  location = /humidity {
    if ($http_referer = '') {
      return 403;
    }
    proxy_pass http://orchestrator-ms/humidity;
  }

  location = /light {
    if ($http_referer = '') {
      return 403;
    }
    proxy_pass http://orchestrator-ms/light;
  }

  location = /temperature {
    if ($http_referer = '') {
      return 403;
    }
    proxy_pass http://orchestrator-ms/temperature;
  }

  location = /microcontrollers {
    if ($http_referer = '') {
      return 403;
    }
    proxy_pass http://orchestrator-ms/microcontrollers;
  }

}
```

Código 5.5. Archivo de configuración de NGINX

Cabe mencionar que en el Código 5.5 solamente se muestra la localización del archivo `index.html` y la redirección de las rutas involucradas en esta funcionalidad (faltarían las rutas relacionadas con la autenticación de usuarios); y que NGINX devolverá un error "403 Forbidden" en caso de que la URL sea accedida directamente desde la barra de búsqueda de un navegador web (así solo se permiten las peticiones realizadas directamente por el código de la aplicación web).

5.5. Servicio de usuarios

Para gestionar la autenticación y el registro de usuarios en la aplicación se ha utilizado el estándar JSON Web Token (JWT), especificado en la RFC 7519 [39].

Este método de autenticación es muy utilizado en arquitecturas REST y arquitecturas de microservicios, ya que permite que el *back-end* sea *stateless* (es decir, que no retenga información sobre los usuarios). Además, en este tipo de arquitecturas el *front-end* está separado del *back-end*; es decir, que el *back-end* no controla la vista que muestra el *front-end*, ya que este no responde con una página HTML, sino solo con datos (normalmente en formato JSON) para que el *front-end* los gestione como corresponda.

Existen otras maneras de autenticar a los usuarios. Por ejemplo, es muy común el uso de la sesión HTTP en aplicaciones web convencionales. Sin embargo, esta manera no es aconsejable si se utilizan microservicios, ya que estos deberían dedicarse a otras tareas distintas de almacenar datos de los usuarios en memoria.

5.5.1. Implementación de JSON Web Token (JWT)

Para comenzar, es necesario explicar lo que es un *token*. Esto no es más que una cadena de caracteres que contiene cierta información codificada.

Un *token* de JWT está constituido por tres partes, separadas por puntos. A continuación se muestra un ejemplo tomado de [40]:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibm
FtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.jdKVAStGjdb8oZrTPQyVAZ
qcjwVRc7josQ1MlpYyjog
```

Las tres partes del *token* JWT son las que aparecen coloreadas. La primera parte representa la cabecera, y contiene información sobre el tipo de *token* y el algoritmo utilizado para verificar su autenticidad. En este caso, la cabecera sería el siguiente objeto JSON:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Por otro lado, la segunda parte, denominada *payload*, contiene la información de usuario que porta el *token*. En este ejemplo, es este objeto JSON:

```
{
  "sub": "123456789",
  "name": "John Doe",
  "iat": "1516239022"
}
```

Por último, la tercera parte del *token* JWT representa la firma del *token*. Para obtener esta firma, primero se codifican los objetos JSON anteriores (analizados como JSON *strings*) en Base64 independientemente y se concatenan por medio de un punto (con esto, se consiguen las dos primeras partes del *token*). A continuación se añade una clave secreta al resultado anterior y se realiza un *hash* del conjunto (con el algoritmo especificado en la cabecera, en este caso HS256).

Se debe saber que la codificación Base64 no es una herramienta que proporcione seguridad, ya que una cadena de caracteres puede ser codificada y decodificada sin necesidad de utilizar una clave. Este algoritmo no es más que un sistema de numeración posicional que utiliza como base el número 64, evidentemente, utilizando 64 de los caracteres imprimibles del código ASCII (entre ellos, las letras del alfabeto, mayúsculas y minúsculas, y los dígitos del 0 al 9).

Por tanto, a partir del *token* JWT se puede obtener la información que contiene (la cabecera y el *payload*) de manera sencilla. De ahí que no se deba incluir información sensible en el cuerpo del *token*.

La manera de brindar seguridad de un *token* JWT es mediante la firma. Para validar un token, se realiza un proceso similar al de su creación: se cogen las dos primeras partes del *token*, se le añade la clave secreta y se computa el *hash*. Un

token es válido si el *hash* obtenido coincide con la tercera parte de dicho *token*.

Cabe mencionar que una función de *hash* es unidireccional, lo que quiere decir que aunque se conozca un *hash* no se podrá conocer el texto con el que se calculó dicho *hash*. Además, las funciones de *hash* tienen la propiedad de que un cambio en el texto de entrada da como resultado un *hash* completamente distinto.

Por este motivo, si se pretende cambiar el contenido del *token*, la firma ya no será válida, puesto que se ha cambiado el texto de entrada a la función de *hash*. Y aunque el algoritmo de *hash* utilizado es conocido (aparece en la cabecera del *token*), no se puede recalcular dicho *hash* correctamente, ya que es necesaria una clave secreta que solo debe poseer el servidor que gestiona los *tokens*.

En el estándar JWT existen los llamados JWT *Claims*, los cuales agregan información al *token* sobre la creación del mismo. Estos JWT *Claims* no son obligatorios, pero sí es recomendable utilizarlos. En este proyecto se han utilizado `iat` (*issued at*) y `exp` (*expiration time*), los cuales representan la marca de tiempo de creación y de expiración, respectivamente.

Los *tokens*, en general, tienen un tiempo de vida relativamente corto (típicamente del orden de minutos). Si el *token* ha expirado, se considera inválido y el usuario deberá obtener un nuevo *token*. Esta acción implicaría tener que iniciar sesión de nuevo, lo cual puede suponer una mala experiencia de usuario.

Por este motivo, existe otro tipo de *token* denominado *refresh token*, que sirve únicamente para regenerar un *token* de acceso sin necesidad de introducir las credenciales de usuario de nuevo. Los *refresh tokens* tienen, por lo general, un mayor tiempo de vida, ya que se suelen almacenar en base de datos y se relacionan con un usuario concreto.

El uso que se le suele dar a los *tokens*, y el que se le da en este proyecto, sigue el siguiente proceso:

- Cuando un usuario inicia sesión en la aplicación, este obtiene un *token* y un *refresh token*.

- Cada vez que se realice una consulta desde el *front-end* al *back-end*, el *token* de acceso viajará en la petición HTTP (típicamente en una cabecera `Authorization: Bearer` *`token`*) para que el servidor pueda validar que el usuario ha iniciado sesión correctamente (puesto que tiene un *token* generado

por el mismo servidor) y proceder a enviar una respuesta HTTP satisfactoria.

- Cuando el *token* ha expirado, el *front-end* realiza, de manera transparente al usuario, una petición de refresco al *back-end* para regenerar el *token* de acceso, incluyendo además el *refresh token* en la petición. El *back-end* comprobará que el *token* es válido fijándose en la firma y que el *refresh token* es el que le corresponde al usuario desde la última vez que inició sesión y que no ha expirado. Finalmente, el *back-end* responde con unos nuevos valores de *token* y *refresh token*.

- En caso de que el *refresh token* hubiera expirado, el *back-end* deberá denegar la petición y responder con un error para que el *front-end* indique al usuario que inicie sesión de nuevo.

5.5.2. Gestión de *tokens*

La gestión de los *tokens* de acceso y de los *refresh tokens* se ha incluido en el microservicio Orchestrator. Al ser este microservicio el API *gateway*, esta es la entrada al *back-end*. Por este motivo, es el punto ideal para aceptar o denegar peticiones dependiendo de si el *token* de acceso es válido o inválido.

Para esto, se ha utilizado el módulo de Node.js denominado *express-jwt*, el cual se puede integrar con la librería de *express*, utilizada para crear un servidor web.

Por otro lado, para generar *refresh tokens*, se han utilizado cadenas de caracteres aleatorias de 256 caracteres (utilizando el módulo *rand-token* de Node.js). Este valor se guarda en base de datos, como se vio en la Fig. 5.6, para poder permitir el refresco del *token* de acceso de un usuario concreto si provee el mismo valor de *refresh token* guardado en la base de datos.

En la Fig. 5.11 se muestra el proceso de refresco de un *token* de acceso mediante un *refresh token*, en el cual intervienen principalmente los microservicios Orchestrator y Auth, y la base de datos de MySQL.

En este diagrama de secuencia se ve cómo es el microservicio Orchestrator el que genera el *refresh token* y lo envía al microservicio Auth para que este actualice el campo `refresh_token` en la tabla USERS de MySQL. Si no se incluye un *refresh token* válido, el microservicio Orchestrator envía un mensaje de error.

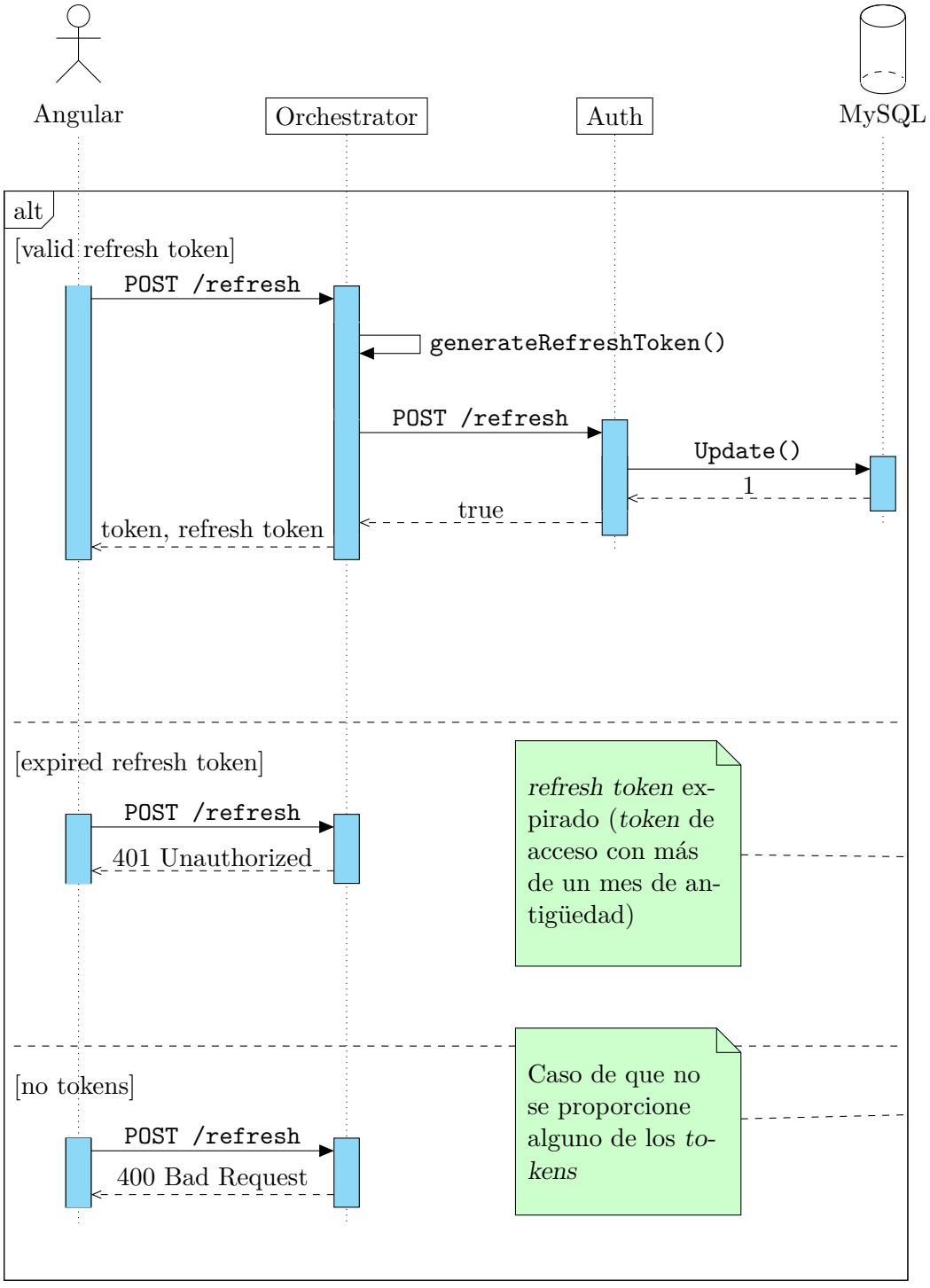


Fig. 5.11. Regeneración del *token* de acceso por medio del *refresh token*

5.5.3. Inicio de sesión y registro de usuarios

El microservicio encargado de gestionar la tabla de USERS de MySQL es el microservicio Auth. Este microservicio está desarrollado en Go, y en él se implementa un simple servidor web con las funcionalidades de inicio de sesión, registro de usuarios y refresco de *tokens* de acceso.

En los diagramas de secuencia de la Fig. 5.12 y Fig. 5.13 se describen los procesos de registro e inicio de sesión de usuarios, respectivamente.

En ambos procesos se puede ver cómo es el microservicio Orchestrator el que genera un nuevo *refresh token* para el usuario y se lo envía al microservicio Auth junto con las credenciales de usuario.

El método de autenticación utilizado en esta aplicación es mediante usuario y contraseña. Para proporcionar una capa de seguridad al almacenar las contraseñas de los usuarios en la base de datos de MySQL, no se han guardado las contraseñas como tal (en texto plano), sino que se ha realizado un *hash* de estas contraseñas.

Como se ha explicado anteriormente en relación a la firma de un *token* JWT, un *hash* es una función unidireccional, lo cual implica que aunque se conozca el *hash* no se podrá obtener el texto inicial con el que se calculó dicho *hash*.

Por este motivo, nada más llegan al microservicio Orchestrator las credenciales de usuario, se calcula el *hash* de la contraseña para mandarlo al microservicio Auth, junto con el nombre de usuario y el nuevo *refresh token* generado.

La manera que tiene el microservicio Auth de saber si la contraseña de un determinado usuario es correcta es comparando el *hash* que obtiene del Orchestrator y el *hash* almacenado en la base de datos. Si estos valores coinciden, la contraseña es correcta; en caso contrario, la contraseña no es correcta (recuérdese que un ligero cambio en el texto de entrada a la función de *hash* produce un resultado completamente distinto).

Una diferencia que presenta el proceso de inicio de sesión con respecto al proceso de registro de un nuevo usuario es que una vez verificado que el usuario y la contraseña son correctas, se ha de actualizar el valor del campo `refresh_token` de dicho usuario en la tabla USERS de MySQL, por seguridad.

En el caso del registro de usuarios, si el usuario a registrar ya existe (proceso

verificado directamente por las restricciones de MySQL), se enviará un mensaje de error. Y en el caso de la autenticación de usuarios, si las credenciales no son correctas, también se mandará un mensaje de error.

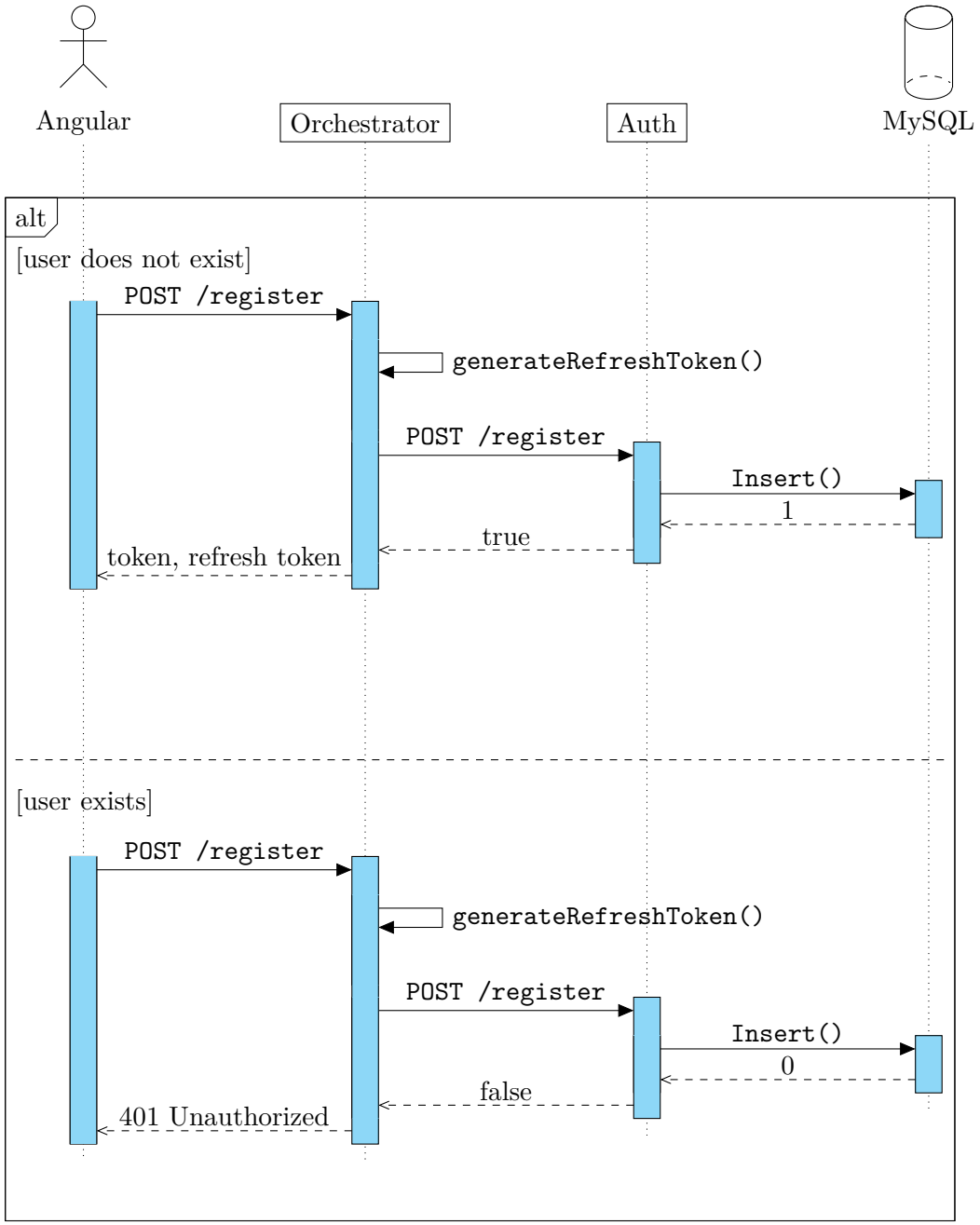


Fig. 5.12. Registro de usuarios en la aplicación

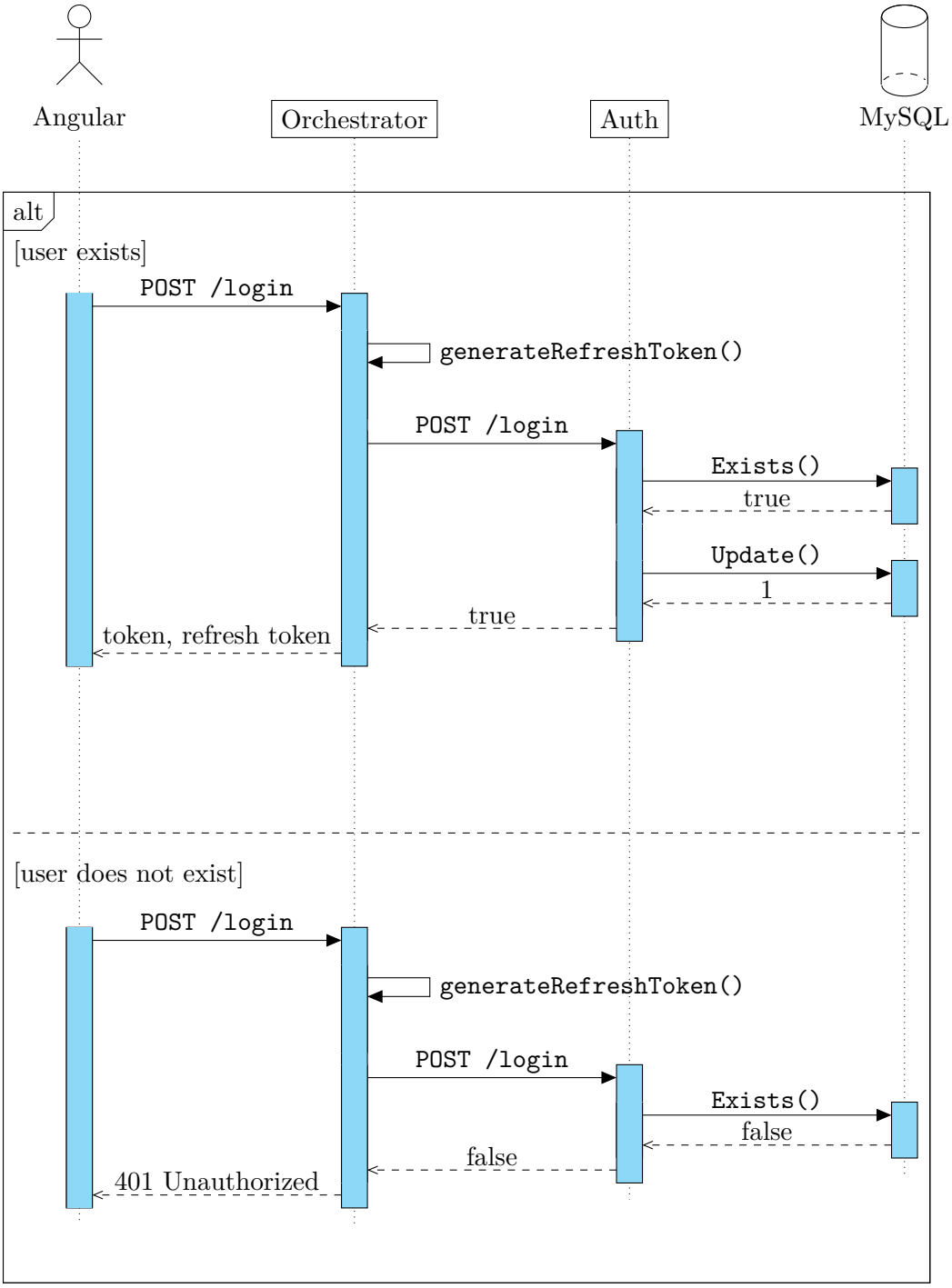


Fig. 5.13. Autenticación de usuarios en la aplicación

5.5.4. Autenticación mediante JWT en Angular

Para gestionar los *tokens* de usuario dentro de una aplicación de Angular, es conveniente utilizar objetos de Angular conocidos como guardias e interceptores.

Un guardia de Angular realiza ciertas comprobaciones cuando el usuario navega entre las distintas rutas de la aplicación (dependiendo de la ruta). Un interceptor se encarga de aplicar una cierta lógica a las peticiones HTTP realizadas por Angular antes de que sean enviadas.

Así, se ha implementado un guardia en este proyecto para impedir que el usuario acceda a rutas a las que no puede acceder porque no está autenticado correctamente en la aplicación. El guardia redirige al usuario a la página de inicio y muestra una ventana de diálogo con el formulario de inicio de sesión.

Respecto al interceptor, este es realmente útil, por ejemplo, para incluir la cabecera `Authorization: Bearer` ***`token`*** en aquellas peticiones que la requieran (básicamente todas, excepto las de inicio de sesión y registro de usuarios).

Otro uso que se le ha dado al interceptor en este proyecto es el de gestionar el refresco del *token* de acceso. El proceso es el que se muestra en el diagrama de la Fig. 5.14.

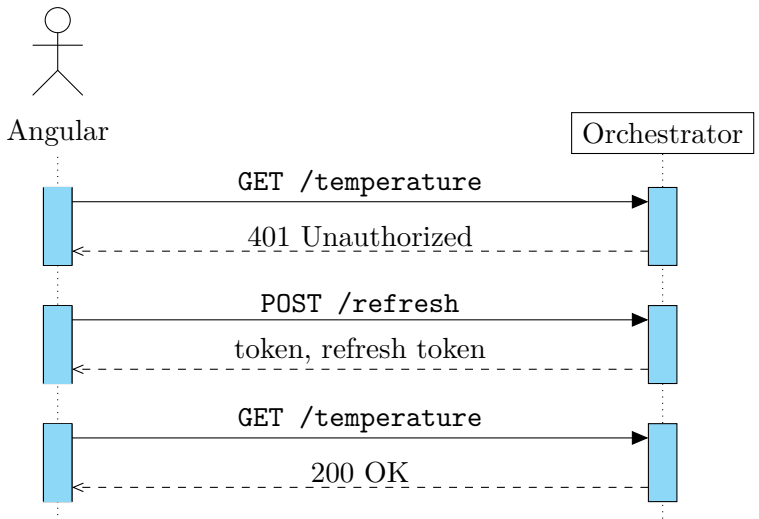


Fig. 5.14. Regeneración del *token* de acceso desde Angular

Como se puede ver, cuando una petición retorna un error de autenticación, el interceptor lo captura y realiza la petición de refresco. Una vez recibe los nuevos *tokens*, vuelve a realizar la petición que antes dio error. Cabe mencionar que este proceso se realiza de manera totalmente transparente al usuario.

Por otro lado, se ha elegido utilizar el almacenamiento local del navegador para guardar los valores del *token* de acceso y del *refresh token* de un usuario. Los navegadores poseen una API denominada `localStorage` que permite la obtener, insertar, actualizar y borrar valores en el almacenamiento local de manera sencilla, a modo de clave-valor.

5.6. Historial de mediciones

Otra funcionalidad importante de la aplicación es la posibilidad de recuperar datos anteriores de temperatura, humedad o luz, almacenados en base de datos.

Para ello, desde la aplicación de Angular se ha de seguir el diagrama de secuencia de la Fig. 5.15: se ha de llamar al microservicio Measure, por medio del microservicio Orchestrator, para pedirle los datos requeridos entre dos fechas concretas. Nótese que la URL de petición de Angular es simplemente `/temperature`. Para indicar que esta petición se refiere a un histórico de temperaturas, se utiliza un parámetro.

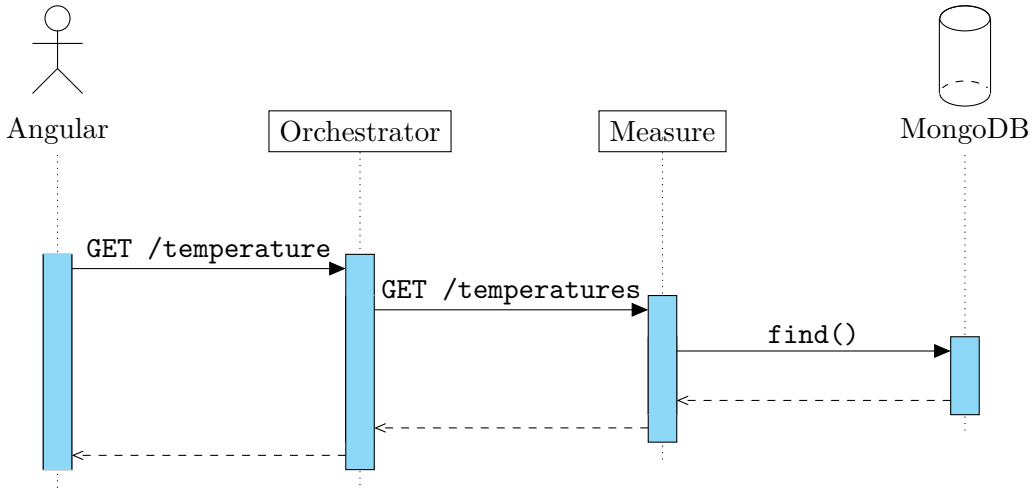


Fig. 5.15. Obtención de un historial de temperaturas

Para poder almacenar diferentes mediciones en la base de datos de MongoDB, el microservicio Measure tendría que realizar peticiones periódicas al Arduino para obtener la temperatura en un momento concreto. En un primer momento, se realizaron estas peticiones a cada minuto, utilizando los intervalos de Node.js.

No obstante, no es demasiado razonable guardar estas mediciones directamente en la base de datos, puesto que estos ocuparían un gran volumen (se está hablando de 60 documentos por hora, es decir 1440 documentos al día). La decisión que se tomó para reducir la cantidad de documentos almacenados en la base de datos fue la de resumir los 60 valores de temperatura en un único documento.

En este documento resumen se guardan los valores de temperatura en una lista ordenada, además de parámetros estadísticos como la media, la desviación típica o los valores máximo y mínimo (dependiendo del tipo de magnitud). Para realizar estos cálculos, se ha utilizado el microservicio Stats, programado en Python.

Este microservicio Stats se encarga de ir almacenando los valores de las distintas mediciones que le llegan en una lista, y cuando el número de elementos de la lista llega a 60, realiza el cálculo de las estadísticas y crea un documento para insertarlo en la base de datos de MongoDB.

Para realizar la comunicación entre el microservicio encargado de pedir datos al Arduino y el microservicio Stats, se ha hecho uso de RabbitMQ, un sistema de mensajería basado en colas de tipo publicación-suscripción. Así, el microservicio Measure publica los valores de temperatura que va obteniendo del Arduino en la cola "temperatures", mientras que el microservicio Stats está suscrito a los mensajes de dicha cola y otras más (como las colas "humidities" o "lights"), de manera que cada vez que haya un mensaje en una cola de RabbitMQ, este será recibido por el microservicio Stats.

RabbitMQ ofrece la funcionalidad de almacenar los mensajes en las colas hasta que el suscriptor esté disponible para recibirlos. Esto es una ventaja con respecto a otros protocolos de comunicación, como HTTP. Así, si por cualquier motivo se cae el microservicio Stats o está realizando algún proceso bloqueante, el mensaje no se perderá. Además, genera una capa de abstracción para el microservicio Measure, ya que simplemente envía los datos a RabbitMQ y se desentiende de si son recibidos por el microservicio Stats, ya que este trabajo es de RabbitMQ.

En la Fig. 5.16 se muestra el proceso anteriormente descrito.

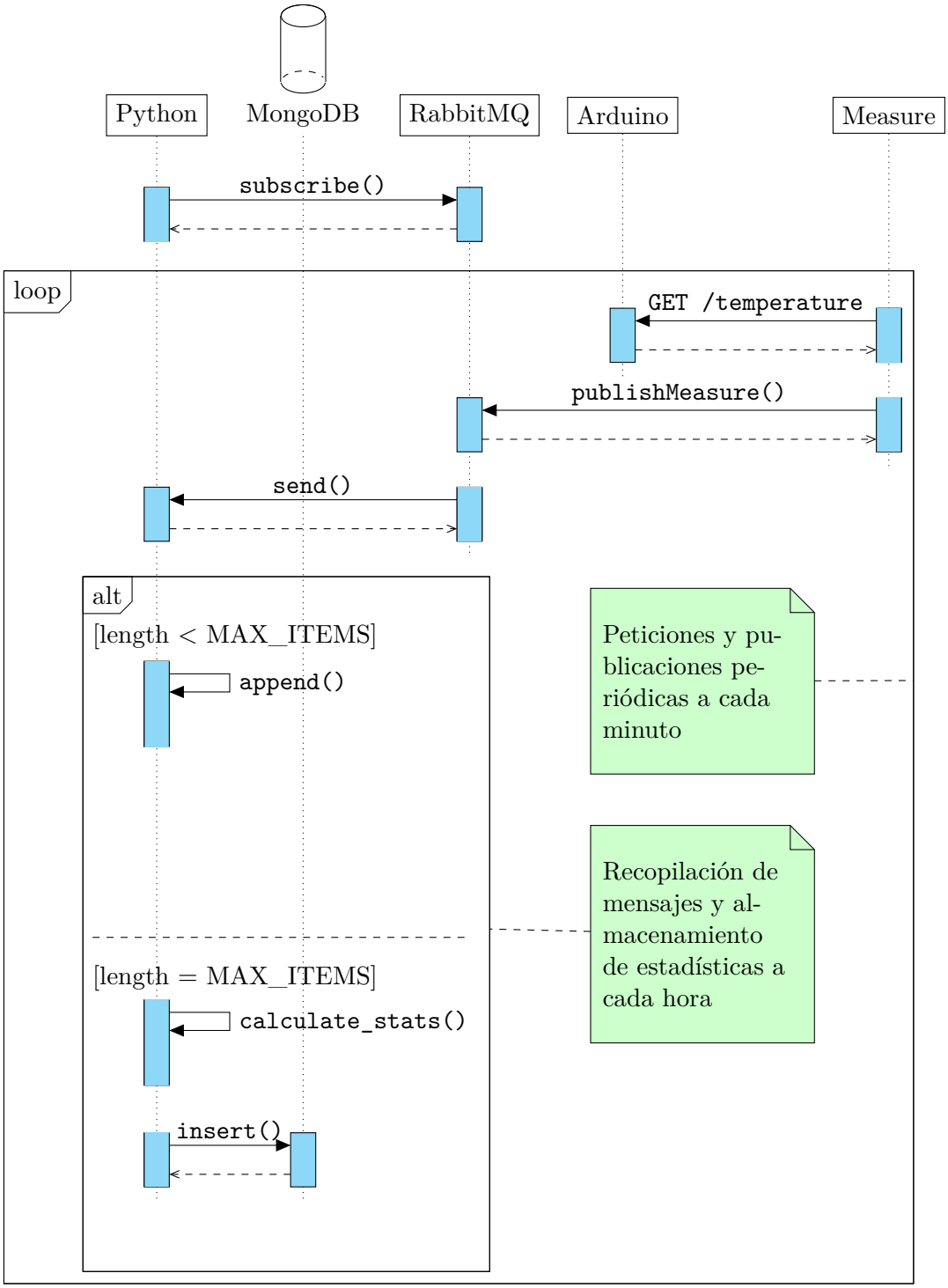


Fig. 5.16. Cálculo de estadísticas

Cabe mencionar que el microservicio Measure pide los datos de temperatura a todos los microcontroladores que poseen un sensor de temperatura y envía cada uno de estos datos a RabbitMQ.

El microservicio Stats genera un *thread* por cada tipo de medición (temperatura, humedad o luz), de manera que cada *thread* recibe datos relativos a un único tipo de medición. Dentro de cada *thread* se generan listas de valores diferentes según la dirección IP del microcontrolador (recuérdese que el par de valores (`ip`, `measure`) es único), de manera que no se mezclan valores medidos por distintos microcontroladores. Así, este microservicio Stats puede mantener distintas cuentas de manera simultánea.

La utilización de *threads* es un elemento clave para el correcto funcionamiento del sistema desarrollado, motivo por el cual se eligió Python para programar el microservicio Stats.

Otro punto importante de esta funcionalidad es el registro del tiempo en el que se realizaron las mediciones. En el documento resumen se guardan tanto la marca de tiempo en milisegundos (desde el 1 de enero de 1970, UNIX Epoch) como el valor de la fecha en formato UTC (tiempo universal coordinado) de la primera y de la última medición.

Ambos valores de tiempo son necesarios: el primero, al ser numérico, resulta más sencillo para que un programa interactúe con él; el segundo, en cambio, resulta más legible para el usuario, por lo que es el que se utilizará en la aplicación de Angular a la hora de mostrar los datos.

Como se dijo antes, desde Angular se realiza la petición de los valores medidos entre dos fechas concretas, las cuales se envían en formato UTC. En el microservicio Measure, se convierte la fecha a marca de tiempo para realizar la búsqueda en la base de datos de MongoDB, ya que la utilización de valores numéricos es más eficiente.

Una vez recibidos los datos en la aplicación de Angular, estos se pintan en una gráfica utilizando el objeto `GoogleChart` y el gráfico `LineChart`, como se explicó en la Sección 5.4.3, dando la posibilidad de visualizar los valores medios, máximos y mínimos a lo largo de un intervalo de tiempo en la misma gráfica.

Si existe un subintervalo temporal en el que no hay datos registrados, la API de Google Charts simplemente trazará una línea recta entre sus extremos.

5.6.1. Implementación de un `CronJob` de Kubernetes

Una vez terminada esta funcionalidad y después de comprobar que realmente funciona y es una solución válida, se vio que el microservicio Measure comenzaba a tener muchas funcionalidades.

Además, el uso de intervalos de JavaScript no era perfecto, y siempre existía un pequeño incremento de tiempo que hacía que el intervalo esperado de un minuto fuera algo mayor.

Por este motivo, se decidió trasladar la funcionalidad de publicar mediciones en las colas RabbitMQ a un nuevo microservicio denominado Publisher.

Este microservicio consiste en un programa de inicio y fin que realiza los siguientes pasos:

1. Obtiene todos los microcontroladores con un determinado sensor (por medio del microservicio Microcontrollers).

2. Realiza las peticiones de la magnitud correspondiente a estos microcontroladores.

3. Publica los valores obtenidos en una cola concreta de RabbitMQ.

Este proceso se repite por cada magnitud disponible en la aplicación (humedad, temperatura y luz).

La manera de integrar este microservicio con la arquitectura de microservicios es mediante un `CronJob` de Kubernetes. Recuérdese que un `CronJob` consiste en la ejecución de un `Job` (es decir, una serie de tareas) en un intervalo de tiempo concreto.

Así, este microservicio Publisher será levantado por Kubernetes automáticamente cada minuto. El microservicio realizará la secuencia de tareas anterior y finalizará su ejecución.

Sin duda, esta es una solución más elegante y aprovecha las ventajas de Kubernetes y las arquitecturas de microservicios, desacoplando funcionalidad de otros y disminuyendo la complejidad del sistema.

5.7. Otras funcionalidades

En esta sección se explican otras funcionalidades implementadas en la aplicación pero que no tienen tanta relevancia con respecto al objetivo del proyecto.

5.7.1. Encendido y apagado de LED

En este punto, se trató de simular el control de bombillas inteligentes a través de la aplicación web de Angular. Para ello, se utilizó un LED de la placa de Arduino.

Como se puede ver en el diagrama de la Fig. 5.17, las peticiones se realizan con el método POST, indicando en el cuerpo de la petición el estado deseado de la bombilla (ON/OFF). Finalmente, el microservicio Measure pone este estado explícitamente en la URL, para que sea más fácil de manejar por el microcontrolador.

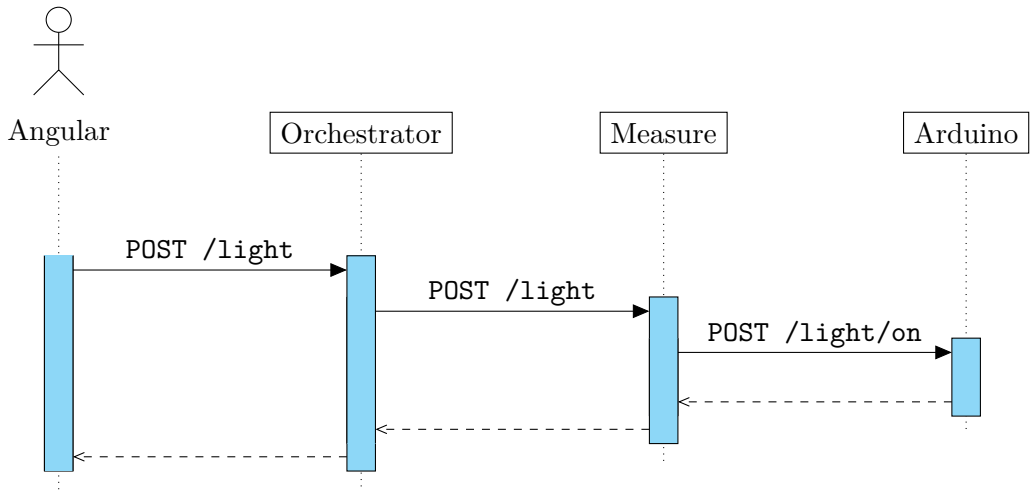


Fig. 5.17. Encendido de un LED

En la aplicación de Angular se conoce en todo momento el estado de las bombillas inteligentes de un usuario, ya que la funcionalidad explicada en la Sección 5.4 también se aplica a las luces, aunque solo hay dos valores posibles (ON/OFF).

5.7.2. Fallos de conexión de microcontroladores

Si un microcontrolador se desconecta o no se ha conectado correctamente, el usuario debe saber que uno de sus microcontroladores está dando fallos de conexión.

Para esto, desde la aplicación de Angular, si no llega un valor de medición de un microcontrolador, es que este está inactivo. En este caso, se pondrá un distintivo para que el usuario sepa cuáles de sus microcontroladores no están conectados.

Cuando la aplicación de Angular obtenga un valor de un microcontrolador que no estaba bien conectado, quitará la marca de inactivo para que el usuario sepa que el dispositivo ha vuelto a funcionar correctamente.

Capítulo 6

Análisis de resultados

En este capítulo se presentan y analizan los resultados más relevantes del proyecto realizado.

Recuérdese que el objetivo principal de este Trabajo Fin de Grado es analizar la viabilidad de utilizar una arquitectura de microservicios para gestionar una red de sensores con Arduino. Para esto, se ha implementado la arquitectura de microservicios anteriormente descrita, junto con una aplicación web para poder interactuar con la placa de Arduino.

Una vez terminado el desarrollo de la aplicación y de los microservicios, se ve que el uso de una arquitectura de este tipo es totalmente viable e incluso presenta más ventajas que inconvenientes con respecto a la arquitectura monolítica (como se vio en la Sección 3.1).

La aplicación web contiene varios aspectos de domótica (ya que este es precisamente el caso de uso considerado para el proyecto), es completamente funcional y está perfectamente integrada con la placa de Arduino a través de la arquitectura de microservicios.

A continuación, se explican distintos puntos a analizar del proyecto desarrollado y los resultados obtenidos.

6.1. Análisis de la arquitectura de microservicios

El haber implementado esta aplicación de domótica con una arquitectura de microservicios ha proporcionado una serie de beneficios que no habrían sido posibles con una arquitectura monolítica.

Quizás lo más destacado (ver el esquema de la Fig. 5.1) sea el poder utilizar diferentes lenguajes de programación en cada microservicio y poder integrarlos de manera efectiva.

Como anécdota, el microservicio Auth se programó primeramente en Java, utilizando Spring-Boot como *framework* de desarrollo de aplicaciones y Apache Tomcat como servidor de aplicaciones web. Sin embargo, este microservicio daba problemas al interactuar con MySQL y tardaba mucho en arrancar. Por este motivo, se decidió implementar el microservicio Auth con Go, el cual no dio tantos problemas, debido a la sencillez del microservicio y del lenguaje.

La resolución de este problema ha sido posible precisamente por utilizar microservicios. En una aplicación monolítica desarrollada al completo con Java, se tendría que haber corregido el error, o haber buscado otra base de datos, o haber migrado toda la aplicación a otro lenguaje, lo cual es muy costoso.

Por tanto, es factible modificar un microservicio y asegurar su compatibilidad con el resto, ya que su integración se realiza mediante protocolos de Internet como HTTP y REST. Esto conlleva la posibilidad de mantener distintas versiones de los microservicios en producción y que sean compatibles todos con todos y sin afectar a la funcionalidad completa de la aplicación. En un sistema monolítico, este hecho no se puede dar, ya que solo se puede tener una única versión del sistema en producción, lo cual podría causar fallos de regresión.

Otro punto interesante es el poder utilizar Python en el microservicio Stats para el cálculo de estadísticas y así aprovechar el potencial de este lenguaje en temas de *Machine Learning*.

La utilización de RabbitMQ también ha sido un punto clave en la arquitectura de microservicios, ya que servía como nexo entre el `CronJob` del microservicio Publisher y el microservicio Stats. En una aplicación monolítica, el sistema de colas de RabbitMQ tendría que haberse implementado con variables globales o algún método similar, lo cual puede causar problemas con la utilización de *threads*.

6.2. Utilización de Docker y Kubernetes

Docker y Kubernetes han sido herramientas muy importantes para implementar la arquitectura de microservicios. Se trata de una manera de desplegar aplicaciones un tanto peculiar, pero que aporta muchos beneficios.

En este proyecto solo se han usado unas pocas ventajas de Kubernetes, ya que se han utilizado Minikube y MicroK8s, versiones de Kubernetes ligeras pensadas para el desarrollo en entornos locales. Lo normal es utilizar un proveedor *cloud* como Amazon Web Services, Google Cloud Platform o Microsoft Azure para desplegar un *cluster* de Kubernetes en sus servidores. Estos servicios no son gratuitos (al menos no de manera indefinida), por lo que se ha preferido montar el *cluster* de Kubernetes en un entorno local.

En el caso del presente proyecto, solamente existe un nodo dentro del *cluster*, y el número de `Pods` que contiene se puede contar fácilmente. De hecho, en los todos `Deployments` se ha especificado una sola réplica del `Pod`, y no se ha habilitado el autoescalado horizontal con el HPA (*Horizontal `Pod` Autoscaler*) de Kubernetes, ya que no existe una necesidad real para implementarlo.

En los *cluster* de las empresas que utilizan Kubernetes existen cientos de miles de `Pods`. Por ejemplo, el servicio de GKE (Google Kubernetes Engine) de Google Cloud Platform admite el despliegue de un *cluster* de hasta 5 000 nodos, con hasta 110 `Pods` por nodo y con un total de 300 000 contenedores en ejecución [41]. En un entorno como este, es vital el uso de herramientas de automatización y gestión para Kubernetes (como Rancher) y herramientas de monitorización (como Prometheus y Grafana) por parte de los administradores de sistemas.

En el presente proyecto no tiene sentido implementar estas tecnologías, ya que la administración del *cluster* es perfectamente realizable de forma manual y, por tanto, no aportarían ninguna ventaja decisiva.

6.3. Pruebas unitarias y de integración

Una de las maneras de comprobar que el *software* escrito funciona correctamente es mediante pruebas unitarias y de integración.

Los microservicios testeados fueron los desarrollados en Node.js y el microservicio Stats, desarrollado con Python, ya que eran los microservicios más relevantes para el funcionamiento de la aplicación.

En Node.js, se utilizó la librería *jest*, la cual permite escribir códigos de prueba (conocidos como *mocks*) de las librerías de terceros. Así, al ejecutar los *tests*, si la aplicación requería una librería externa, utilizaría el *mock*.

Por ejemplo, normalmente el microservicio Measure realiza una petición a la placa de Arduino utilizando la librería *axios*. El *mock* de *axios* podría devolver la respuesta que daría el Arduino, sin realizar la petición HTTP. Así, se supone que todos los *inputs* al microservicio Measure son los que deberían ser (gracias a la utilización de *mocks*), y lo que se verifica realmente es que los *outputs* son los correctos. Esta manera de realizar *tests* se utiliza en todos los microservicios de Node.js, añadiendo *mocks* a las librerías externas.

En el microservicio Stats se realiza un proceso parecido al seguido con los microservicios de Node.js, pero con el módulo *unittest*. El microservicio Auth no fue necesario probarlo mediante pruebas unitarias, ya que su funcionalidad era muy sencilla y podía probarse manualmente.

Una vez escritos algunos *tests*, al añadir nueva funcionalidad o modificar alguna parte de un microservicio, se ejecutaban estas pruebas automáticas para comprobar que todo seguía funcionando correctamente. Este método de desarrollo es parecido al TDD (*Test-Driven Development*) y con él se consigue generar un *software* funcional y de mejor calidad.

Durante el desarrollo en local, para comprobar que funcionaban las comunicaciones entre microservicios, se utilizó Postman, el cual es un cliente de arquitecturas de tipo REST. Postman es una herramienta muy útil y sencilla para realizar peticiones HTTP con la posibilidad de indicar URL, método, parámetros, cuerpo y cabeceras, entre otras cosas.

Por otro lado, aunque Angular provee un entorno de pruebas con las librerías *karma* y *protractor*, no se consideró la realización de *tests* de manera programática, ya que no se trata de una aplicación de gran complejidad. No obstante, durante el desarrollo de la aplicación web, sí se realizaron pruebas manualmente mediante las herramientas de desarrollador del navegador (inspector de elementos, consola y depurador de JavaScript, paneles de red y almacenamiento, etc.).

Una de las desventajas de la arquitectura de microservicios (presentada en la Sección 3.1) hacía referencia a las pruebas de integración y a la comunicación entre microservicios. Utilizando pruebas unitarias y Postman ha sido posible minimizar este inconveniente para que no afectara al desarrollo de la aplicación.

Si se hubiera desarrollado la aplicación de manera monolítica, la realización de pruebas unitarias y de integración habría sido mucho más complicada, ya que no sería tan fácil modularizar las distintas funcionalidades del sistema o añadir *mocks*. Además, un solo punto de fallo puede implicar el fallo completo del sistema, lo cual puede ser difícil de depurar.

Cabe mencionar la existencia de servicios de automatización como Jenkins, que son herramientas que se encargan de ejecutar una batería de pruebas automáticas antes de integrar el nuevo *software* al entorno de producción para verificar que no hay ningún fallo o regresión. Además, se pueden configurar para versionar el código y desplegarlo, haciendo uso del paradigma CI/CD.

6.4. Monitorización de microservicios

Otro inconveniente de las arquitecturas de microservicios es que utilizan la red para comunicarse entre sí (mediante HTTP, REST o colas de mensajería). Este punto puede ser crítico si se necesita que la latencia de red sea mínima.

Para observar el tráfico de red de cada microservicio, se ha utilizado Prometheus y Grafana (explicados en la Sección 2.2.6) para obtener un *dashboard* de monitorización con métricas de los `Pods` del *cluster* (ver Fig. 6.1).

Las métricas de los `Pods` se corresponden con el ratio de *bytes* transmitidos (líneas de color verde) y el ratio de *bytes* recibidos (líneas de color ámbar) en un intervalo de 30 minutos.

Durante los primeros 10 minutos, aproximadamente, se tenía abierto el *dashboard* del usuario en la aplicación de Angular. Esto significa que los datos de los sensores estaban llegando a la aplicación de manera periódica (cada 10 segundos). Por este motivo, se ve que el tráfico de los microservicios Angular, Orchestrator, Microcontrollers y Measure son elevados y más o menos constantes.

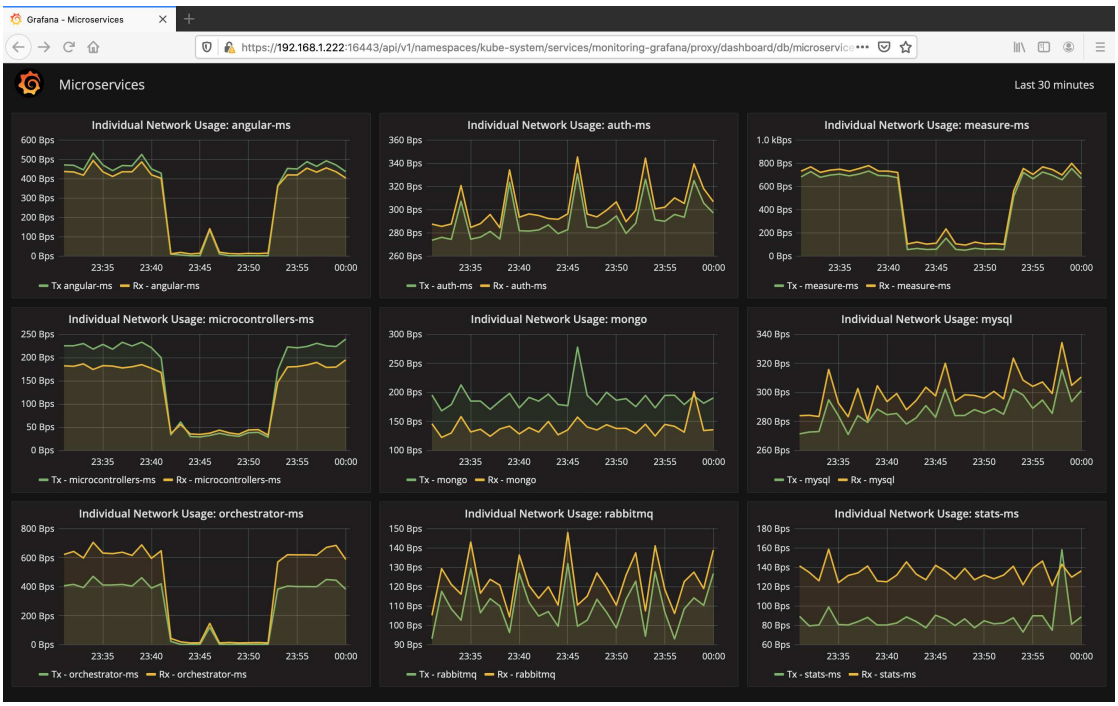


Fig. 6.1. Tráfico de red de los microservicios con Grafana y Prometheus

Por otro lado, se puede ver cómo el microservicio Auth presenta actividad de red cada 5 minutos, aproximadamente. Esto se debe a que los *tokens* de acceso tienen una caducidad de 5 minutos, precisamente, por lo que la aplicación de Angular realizará una petición de refresco cada vez que expiren los *tokens*. De hecho, estos picos tan característicos de la gráfica de Auth se pueden observar también en las gráficas de MySQL, Angular y Orchestrator (no tan pronunciados porque estos microservicios tienen más actividad).

Entre las 23:42 y las 23:52, aproximadamente, se abrió la ventana de historial de mediciones en la aplicación de Angular. En esta ventana ya no existen peticiones sucesivas para obtener datos en tiempo real de los sensores; y, por este motivo, se observa que el tráfico de Angular, Orchestrator, Microcontrollers y Measure se ha reducido considerablemente. El microservicio Microcontrollers sigue teniendo algo de actividad porque es consultado por el microservicio Publisher (no mostrado en el *dashboard* por ser un `CronJob`).

Alrededor de las 23:46, se solicitó un historial de mediciones desde la aplicación de Angular. En este punto, aparece un pico en la gráfica de Angular, así como en la de Orchestrator, Measure y MongoDB. Esto se debe a que el historial de mediciones

está almacenado en una colección de MongoDB, que es consultada mediante el microservicio Measure. Nótese que el microservicio Auth también presenta un pico a las 23:46, estando a más de 5 minutos del pico anterior. El motivo es que el *token* de acceso había expirado en el momento de hacer la petición del historial, por lo que antes de llevar a cabo la petición, se solicita un refresco de *token*.

Otro aspecto interesante es que se ha capturado el momento en el que el microservicio Stats reúne un total de 60 mediciones por magnitud, genera los documentos de resumen con las estadísticas correspondientes y los guarda en la base de datos MongoDB. Esto ocurre en torno a las 23:58 y puede observarse en el pico verde de la gráfica de Stats, que coincide con el pico ámbar de la gráfica de MongoDB.

Cabe mencionar que la mayoría de curvas de *bytes* transmitidos y recibidos solo se diferencian en un desplazamiento vertical, ya que la mayoría de microservicios se dedican a enviar lo que reciben de otros microservicios. Por tanto, esta diferencia se corresponde con los datos propiamente dichos (obviando las cabeceras de los mensajes HTTP).

Capítulo 7

Conclusiones y trabajos futuros

En este capítulo se presentan las conclusiones más importantes de la realización de este Trabajo Fin de Grado, analizando los objetivos propuestos y los resultados obtenidos. Además, se indicarán una serie de trabajos futuros en relación con el proyecto desarrollado.

7.1. Conclusiones y resultados principales

El objetivo esencial del presente proyecto se ha cumplido, y se concluye que es completamente viable implementar una arquitectura de microservicios para gestionar una red de sensores.

Respecto a los tres objetivos expuestos en la Sección 4.2:

- Se ha logrado implementar una arquitectura de microservicios funcional y escalable, aprovechando al máximo las ventajas que ofrece este tipo de arquitectura con respecto a una arquitectura monolítica. Además, el uso de Docker y Kubernetes ha supuesto un valor añadido en cuanto al despliegue de los microservicios, como se ha comentado en previos capítulos.

- Se ha programado una placa de Arduino para controlar un sensor de temperatura ambiente, un sensor de humedad de la tierra y un LED que simula una bombilla inteligente. Estas tres funcionalidades son compatibles con un

sistema de domótica; y, además, se utiliza la tecnología de IoT al estar la placa de Arduino conectada a Internet mediante una antena WiFi.

- Se ha conseguido desarrollar un *dashboard* de control como una aplicación web con Angular. En este *dashboard*, el usuario puede ver en tiempo real los datos de sus sensores y puede visualizar un histórico de estos datos, además de obtener ciertas estadísticas útiles para la gestión de su vivienda.

Adicionalmente, la aplicación se ha desarrollado para admitir múltiples usuarios y múltiples microcontroladores por usuario. No obstante, el sistema solo funciona en un entorno local; es decir, las direcciones IP de los microcontroladores de los usuarios han de ser públicas y accesibles, o bien han de estar en la misma red de área local que el dispositivo que contiene el *cluster* de Kubernetes.

Esto último se puede mejorar incluyendo un *hub* que actúe de intermediario (como se comentó en la Sección 3.6.2); lo cual no tiene que ver con el objetivo principal del proyecto, ya que existiría el mismo problema con una arquitectura monolítica.

Como solo se disponía de una placa de Arduino, para simular otra placa de Arduino se utilizó un servidor web sencillo de Node.js (denominado *Fake-Arduino*) que devolviera valores aleatorios de temperatura y humedad y que tuviera una variable binaria para emular una bombilla inteligente.

Por otro lado, la aplicación web desarrollada no se limita solo a obtener datos de los microcontroladores. También se ha implementado un servicio de usuarios y autenticación con el estándar JWT, muy utilizado en arquitecturas de microservicios y arquitecturas tipo REST. A raíz de esto surgió la idea de poder utilizar varios microcontroladores por usuario.

Además, se ha probado el funcionamiento de la aplicación mediante *tests* unitarios y de integración para mejorar la calidad y la eficiencia del *software* desarrollado; y se han visualizado algunas métricas del *cluster* de Kubernetes para estar al tanto de los recursos y el tráfico de red de cada `Pod`.

7.2. Trabajos futuros

Una vez terminado el presente proyecto, surgen nuevas ideas para agregar a la aplicación desarrollada. A pesar de que es un proyecto de gran envergadura y que

contiene conceptos abstractos y avanzados (sobre todo relacionados con Docker y Kubernetes), se pueden considerar las siguientes tareas adicionales:

- Implementar *Machine Learning*, por ejemplo con modelos de regresión, para realizar predicciones de valores de temperatura o humedad. Sería un proyecto bastante asequible, ya que se tiene ya una base de datos con mediciones de temperatura y humedad que puede servir como *dataset* de entrenamiento al modelo. Además, al utilizar microservicios, la idea sería desarrollar uno nuevo en Python (por ejemplo) que tuviera esta nueva funcionalidad.

- Añadir cálculos acerca del consumo de las bombillas inteligentes, por ejemplo, para proporcionar dicha información a los usuarios y conseguir así una aplicación más familiarizada con el desarrollo sostenible. En base a estos costes energéticos, se podrían implementar algoritmos para gestionar los recursos y lograr un consumo responsable.

- Investigar sobre la seguridad en la conexión entre red de sensores y la arquitectura de microservicios. En el presente proyecto no se ha considerado este aspecto. De hecho, el sistema desarrollado permite hacer peticiones a la placa de Arduino desde cualquier dispositivo conectado a la misma red de área local (ya que la placa funciona como un servidor web). Por este motivo, un punto a considerar es mejorar la conectividad de la placa de Arduino para que solo pueda conectarse a la red de microservicios, mejorando así la seguridad de los sensores y los datos que proporcionan. Por otro lado, también convendría investigar sobre los servicios que ofrece Kubernetes con respecto a la seguridad en el *cluster*.

- Desplegar el *cluster* de Kubernetes desarrollado en un entorno de producción como Amazon Web Services, Google Cloud Platform o Microsoft Azure. Para esto, habría que investigar además cómo realizar la conexión entre la placa de Arduino y el *back-end* (posiblemente mediante un *hub*). Una vez conseguido esto, se podría aprovechar al máximo la utilización de Kubernetes y emplear herramientas de automatización y monitorización, las cuales tienen más sentido en entornos de producción.

Estas extensiones mejorarían el proyecto desarrollado en este Trabajo Fin de Grado en el sentido de que sería un sistema más robusto y funcional, y también, más cercano a la realidad y a los modelos de negocio de las empresas actualmente.

Bibliografía

[1] *¿Qué es la transformación digital?*, Red hat. [En línea]. Disponible en: https://www.redhat.com/es/topics/digital-transformation/what-is-digital-transformation (Accedido: 28-06-2020) (citado en p. 13).

[2] *La importancia del Cloud Computing en las empresas*. [En línea]. Disponible en: https://www.beservices.es/importancia-cloud-computing-empresas-n-5317-es (Accedido: 26-06-2020) (citado en p. 13).

[3] *¿Qué es Docker?*, Red Hat. [En línea]. Disponible en: https://www.redhat.com/es/topics/containers/what-is-docker (Accedido: 06-04-2020) (citado en p. 15).

[4] *index / Alpine Linux*, Alpine Linux. [En línea]. Disponible en: https://alpinelinux.org (Accedido: 20-06-2020) (citado en p. 17).

[5] *Reference documentation | Docker Documentation*, Docker. [En línea]. Disponible en: https://docs.docker.com/reference/ (Accedido: 22-04-2020) (citado en pp. 18, 19, 20).

[6] *Orquestación de contenedores para producción - Kubernetes*, Kubernetes. [En línea]. Disponible en: https://kubernetes.io/es/ (Accedido: 22-04-2020) (citado en pp. 21, 116).

[7] *Case Studies - Kubernetes*, Kubernetes. [En línea]. Disponible en: https://kubernetes.io/case-studies/ (Accedido: 25-04-2020) (citado en p. 21).

[8] *Conceptos - Kubernetes*, Kubernetes. [En línea]. Disponible en: https://kubernetes.io/es/docs/concepts/ (Accedido: 22-04-2020) (citado en pp. 21, 22).

[9] *Angular - Introduction to the Angular Docs*, Angular. [En línea]. Disponible en: https://angular.io/docs (Accedido: 25-04-2020) (citado en pp. 32, 33).

[10] *Preprocesador CSS - Glosario | MDN*, Mozilla. [En línea]. Disponible en: https://developer.mozilla.org/es/docs/Glossary/Preprocesador_CSS (Accedido: 25-04-2020) (citado en p. 33).

[11] *Components | Angular Material*, Angular. [En línea]. Disponible en: https://material.angular.io/components/categories (Accedido: 25-04-2020) (citado en p. 34).

[12] *RxJS*. [En línea]. Disponible en: https://rxjs-dev.firebaseapp.com (Accedido: 25-04-2020) (citado en p. 34).

[13] *Node.js*, OpenJS Foundation. [En línea]. Disponible en: https://nodejs.org/es/ (Accedido: 14-04-2020) (citado en p. 34).

[14] *Arduino - Introduction*, Arduino. [En línea]. Disponible en: https://www.arduino.cc/en/Guide/Introduction (Accedido: 24-04-2020) (citado en p. 36).

[15] *ARDUINO UNO WiFi REV2 | Arduino Official Store*, Arduino. [En línea]. Disponible en: https://store.arduino.cc/arduino-uno-wifi-rev2 (Accedido: 24-04-2020) (citado en pp. 37, 68).

[16] *WiFiNINA*, Arduino. [En línea]. Disponible en: https://www.arduino.cc/en/Reference/WiFiNINA (Accedido: 24-04-2020) (citado en p. 37).

[17] *Grove - Temperature Sensor - Seeed Studio*. [En línea]. Disponible en: https://www.seeedstudio.com/Grove-Temperature-Sensor.html (Accedido: 30-05-2020) (citado en pp. 38, 69).

[18] *Grove - Moisture Sensor - Seeed Studio*. [En línea]. Disponible en: https://www.seeedstudio.com/Grove-Moisture-Sensor.html (Accedido: 30-05-2020) (citado en pp. 38, 70).

[19] *Documentation - The Go Programming Language*. [En línea]. Disponible en: https://golang.org/doc/ (Accedido: 22-04-2020) (citado en p. 39).

[20] *The most popular database for modern apps*, MongoDB. [En línea]. Disponible en: https://www.mongodb.com/ (Accedido: 13-04-2020) (citado en pp. 39, 40).

[21] *MySQL Editions*, MySQL. [En línea]. Disponible en: https://www.mysql.com/products/ (Accedido: 13-04-2020) (citado en p. 41).

[22] *AMQP v1.0*, 2011. [En línea]. Disponible en: http://www.amqp.org/sites/amqp.org/files/amqp.pdf (Accedido: 13-04-2020) (citado en p. 42).

[23] *nginx*, NGINX. [En línea]. Disponible en: https://nginx.org/en/ (Accedido: 25-04-2020) (citado en p. 42).

[24] *Git*. [En línea]. Disponible en: https://git-scm.com/ (Accedido: 10-04-2020) (citado en p. 43).

[25] K. B. Roland Barcia y R. Osowski, *Guía de Microservicios. Punto de vista*, IBM. [En línea]. Disponible en: https://www.ibm.com/downloads/cas/5O8YOPKN (Accedido: 09-06-2020) (citado en p. 46).

[26] *Arquitectura de microservicios: qué es, ventajas y desventajas*. [En línea]. Disponible en: https://decidesoluciones.es/arquitectura-de-microservicios/ (Accedido: 25-06-2020) (citado en pp. 46, 47).

[27] *Arquitectura de microservicios vs arquitectura monolítica*. [En línea]. Disponible en: https://www.viewnext.com/arquitectura-de-microservicios-vs-arquitectura-monolitica/ (Accedido: 06-06-2020) (citado en p. 47).

[28] *Características de AWS Lambda*, Amazon Web Services. [En línea]. Disponible en: https://aws.amazon.com/es/lambda/features/ (Accedido: 07-06-2020) (citado en p. 51).

[29] *¿Qué Es DevOps?*, 19 de dic. de 2019. [En línea]. Disponible en: https://nub8.net/es/que-es-devops/ (Accedido: 09-06-2020) (citado en p. 52).

[30] *CI/CD para arquitecturas de microservicios*, Microsoft Azure, 27 de mar. de 2019. [En línea]. Disponible en: https://docs.microsoft.com/es-es/azure/architecture/microservices/ci-cd (Accedido: 09-06-2020) (citado en p. 52).

[31] *SaaS, PaaS e IaaS: los principales modelos de servicio cloud*, 14 de abr. de 2020. [En línea]. Disponible en: https://www.stackscale.com/es/blog/modelos-de-servicio-cloud/ (Accedido: 08-06-2020) (citado en pp. 53, 54).

[32] *¿Qué es IoT?*, ORACLE. [En línea]. Disponible en: https://www.oracle.com/es/internet-of-things/what-is-iot.html (Accedido: 09-06-2020) (citado en p. 55).

[33] *Case Study: Wink. Cloud-Native Infrastructure Keeps Your Smart Home Connected*, Kubernetes. [En línea]. Disponible en: https://kubernetes.io/case-studies/wink/ (Accedido: 09-06-2020) (citado en pp. 56, 58).

[34] *Wink: Connecting Your Smart Home Using Cloud Native Infrastructure*, Cloud Native Computing Foundation. [En línea]. Disponible en: https://www.cncf.io/blog/2017/07/26/wink-connecting-smart-home-using-cloud-native-infrastructure/ (Accedido: 09-06-2020) (citado en pp. 56, 57).

[35] *Wink | Buy and View Smart Home Products*, Wink. [En línea]. Disponible en: https://www.wink.com/products/ (Accedido: 27-06-2020) (citado en p. 57).

[36] *SmartThings. Más dispositivos inteligentes, una aplicación inteligente*, Samsung. [En línea]. Disponible en: https://www.samsung.com/es/apps/smartthings/ (Accedido: 28-06-2020) (citado en p. 57).

[37] *What is API gateway? - Definition from WhatIs.com*. [En línea]. Disponible en: https://whatis.techtarget.com/definition/API-gateway-application-programming-interface-gateway (Accedido: 03-05-2020) (citado en p. 64).

[38] *Using Google Charts | Google Developers*, Google. [En línea]. Disponible en: https://developers.google.com/chart/interactive/docs/ (Accedido: 05-05-2020) (citado en p. 76).

[39] *JSON Web Token (JWT)*, IETF, 2015. [En línea]. Disponible en: https://tools.ietf.org/html/rfc7519 (Accedido: 07-05-2020) (citado en p. 81).

[40] *JSON Web Tokens - jwt.io*. [En línea]. Disponible en: https://jwt.io (Accedido: 07-05-2020) (citado en p. 81).

[41] *Elige el tamaño y el permiso de los clústeres de Google Kubernetes Engine*, Google Cloud. [En línea]. Disponible en: https://cloud.google.com/solutions/scope-and-size-kubernetes-engine-clusters?hl=es-419 (Accedido: 19-06-2020) (citado en p. 99).

[42] *Desarrollo Sostenible – United Nations Sustainable Development Sites*, ONU. [En línea]. Disponible en: https://www.un.org/sustainabledevelopment/es/ (Accedido: 10-06-2020) (citado en p. 155).

[43] *Desarrollo Sostenible. ¿Qué es y cómo alcanzarlo?*, Acciona. [En línea]. Disponible en: https://www.acciona.com/es/desarrollo-sostenible/ (Accedido: 10-06-2020) (citado en p. 155).

[44] *¿Cómo puede la Domótica hacer de tu Vivienda un Entorno más Sostenible?* [En línea]. Disponible en: https://inarquia.es/domotica-vivienda-sostenible (Accedido: 10-06-2020) (citado en p. 158).

[45] *Cómo ayuda la domótica en edificios en nuestro día a día*. [En línea]. Disponible en: https://inarquia.es/como-ayuda-la-domotica-en-edificios-en-nuestro-dia-a-dia (Accedido: 10-06-2020) (citado en p. 158).

[46] *El camino hacia las TIC sostenibles*. [En línea]. Disponible en: https://revistabyte.es/tema-de-portada-byte-ti/el-camino-hacia-unas-tic-sostenibles/ (Accedido: 10-06-2020) (citado en p. 158).

Anexo A. Guía de instalación

En este anexo se presentan los distintos programas necesarios para desarrollar la aplicación descrita en la memoria, así como su instalación y su modo de uso.

A.1. Docker

Para utilizar Docker, es necesario instalar Docker Desktop, que es un programa que permite crear imágenes de Docker personalizadas, ejecutarlas y descargar imágenes de Docker Hub o algún otro registro, entre otras tareas.

Esta herramienta es necesaria para crear imágenes propias y subirlas a Docker Hub. Kubernetes buscará estas imágenes en Docker Hub y las utilizará para sus funciones.

Docker Desktop se puede descargar directamente desde la página oficial de Docker (https://www.docker.com/products/docker-desktop). A continuación, se muestran algunos pasos de la instalación.

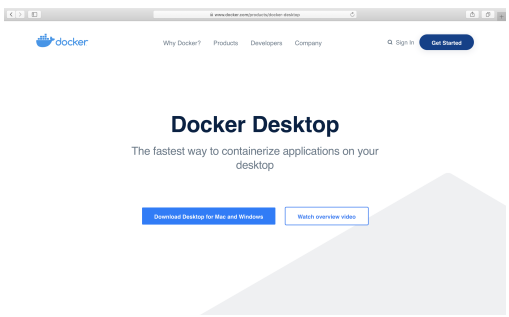


Fig. A.1. Instalación de Docker Desktop (1)

Es necesario registrarse con una cuenta de Docker para instalar Docker Desktop. Si no se dispone de cuenta, habrá que crearse una nueva. Esta cuenta será la misma que se utiliza en Docker Hub.

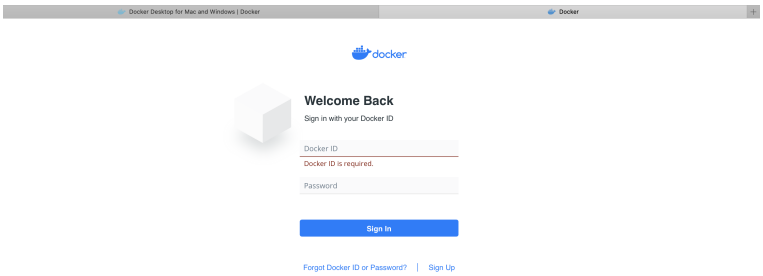


Fig. A.2. Instalación de Docker Desktop (2)

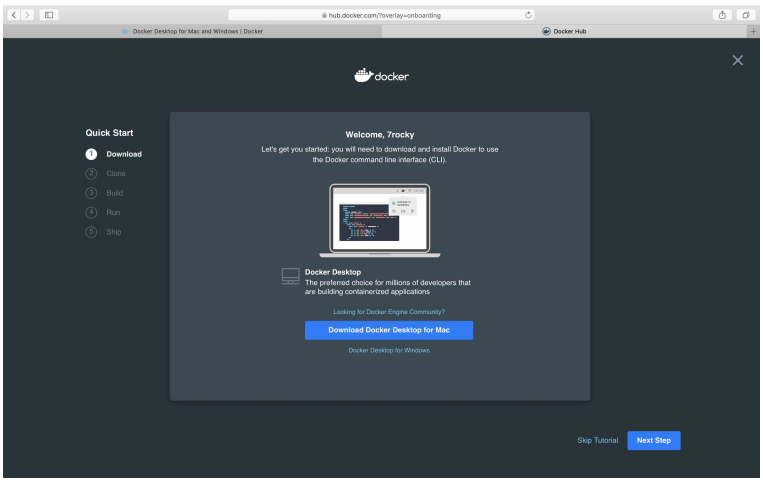


Fig. A.3. Instalación de Docker Desktop (3)

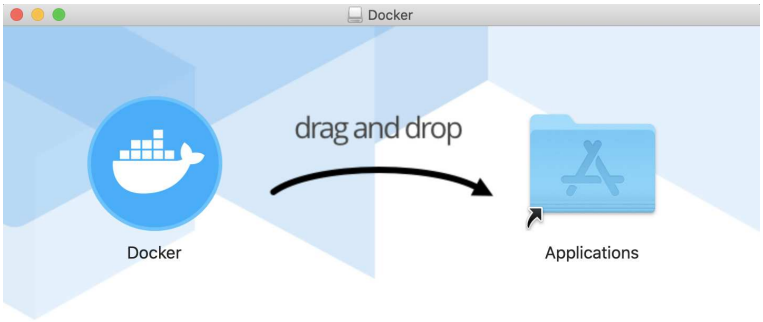


Fig. A.4. Instalación de Docker Desktop (4)

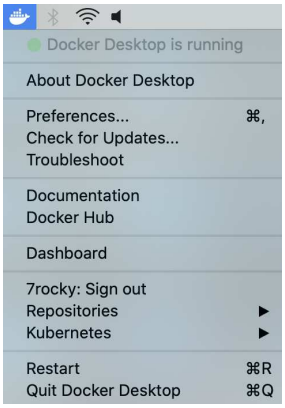


Fig. A.5. Instalación de Docker Desktop (5)

Para verificar la correcta instalación de Docker, se ha de iniciar la aplicación de Docker Desktop. Esta aplicación no abre una ventana con una interfaz gráfica, sino que activa un icono en la barra de herramientas (ver Fig. A.5). De hecho, Docker es comúnmente usado mediante línea de comandos. Cuando aparezca el mensaje de *Docker Desktop is running* al pinchar sobre dicho icono, se debe abrir una consola y escribir el siguiente comando (ver Fig. A.6):

```
docker --version
```

Fig. A.6. Verificación de la instalación de Docker

Si aparece la versión instalada quiere decir que el programa se ha instalado correctamente. En caso contrario, la consola no reconocería el comando.

En el menú desplegable de la Fig. A.5 se puede configurar el usuario de Docker Hub, se puede abrir una interfaz de usuario (*Dashboard*) para visualizar los contenedores en ejecución e incluso se puede inicializar un *cluster* de Kubernetes de un único nodo.

A.2. Kubernetes

La instalación de Kubernetes es bastante larga. Se aconseja seguir detalladamente los pasos indicados en la página oficial de Kubernetes, accesible en [6], teniendo en cuenta además el sistema operativo que se esté utilizando.

En primer lugar, para el desarrollo de este proyecto se ha utilizado Minikube. Este programa genera un *cluster* de Kubernetes de un único nodo en un entorno local (utilizado normalmente para desarrollo). Minikube soporta las características más habituales de Kubernetes, por lo que es ideal para iniciarse con Kubernetes.

Para instalar Minikube, se deben seguir los pasos mostrados en https://kubernetes.io/docs/tasks/tools/install-minikube/. El presente proyecto ha sido desarrollado en un ordenador con sistema operativo macOS, por lo que los pasos indicados a continuación se ajustan a macOS.

En primer lugar, se debe verificar que la virtualización es soportada por la versión de macOS que se está utilizando. Para ello, hay que ejecutar el siguiente comando:

```
sysctl -a | grep -E --color 'machdep.cpu.features|VMX'
```

Si aparece `VMX` en color rojo, entonces la virtualización está habilitada y se puede instalar Minikube (ver Fig. A.7).

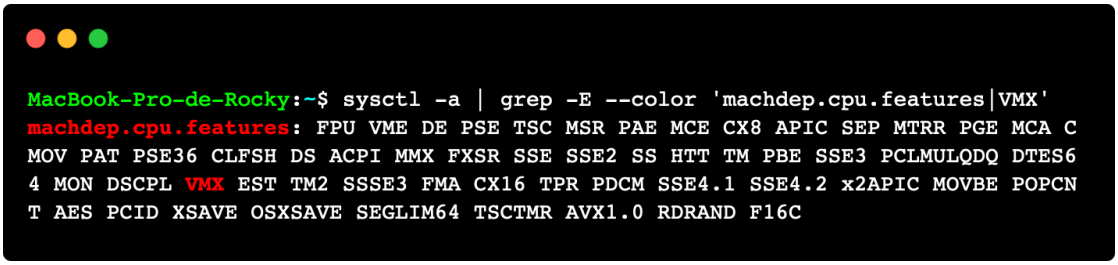


Fig. A.7. Instalación de Minikube (1)

A continuación, hay que instalar la línea de comandos de Kubernetes: `kubectl`. Para ello, se debe acudir a https://kubernetes.io/docs/tasks/tools/install-kubectl/#install-kubectl-on-macos. Puede instalarse con `curl`, con HomeBrew y con MacPorts. Sin embargo, resulta sencillo de instalar con HomeBrew, un gestor de paquetes de macOS:

```
brew install kubernetes-cli
```

```
kubectl version --client
```

```
MacBook-Pro-de-Rocky:~$ brew install kubernetes-cli
==> Downloading https://homebrew.bintray.com/bottles/kubernetes-cli-1.18.3.catal
ina.bottle.tar.gz
==> Downloading from https://akamai.bintray.com/c3/c359b0082317f065a9ecb67fd35e4
######################################################################## 100.0%
==> Pouring kubernetes-cli-1.18.3.catalina.bottle.tar.gz
==> Caveats
Bash completion has been installed to:
  /usr/local/etc/bash_completion.d

zsh completions have been installed to:
  /usr/local/share/zsh/site-functions
==> Summary
🍺  /usr/local/Cellar/kubernetes-cli/1.18.3: 232 files, 49.2MB
MacBook-Pro-de-Rocky:~$ kubectl version --client
Client Version: version.Info{Major:"1", Minor:"18", GitVersion:"v1.18.3", GitCom
mit:"2e7996e3e2712684bc73f0dec0200d64eec7fe40", GitTreeState:"clean", BuildDate:
"2020-05-21T14:51:23Z", GoVersion:"go1.14.3", Compiler:"gc", Platform:"darwin/am
d64"}
```

Fig. A.8. Instalación de `kubectl`

Después, será necesario instalar uno de los siguientes gestores de máquinas virtuales: HyperKit, VirtualBox o VMWare Fusion. En este proyecto se ha utilizado VirtualBox. Se puede entrar simplemente en la página https://www.virtualbox.org y pinchar en "Descargar". A continuación, se elige el sistema operativo del ordenador en el que se vaya a utilizar y la versión de VirtualBox deseada (la más reciente). Se descargará un instalador (ver las siguientes figuras):

Fig. A.9. Instalación de VirtualBox (1)

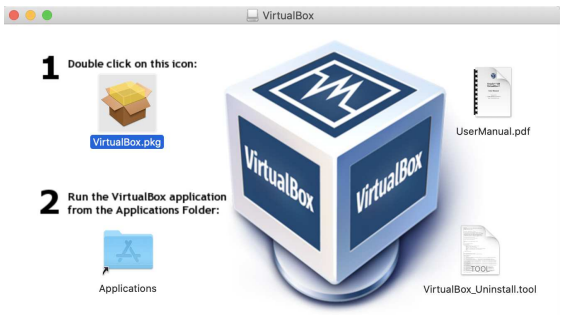


Fig. A.10. Instalación de VirtualBox (2)

Fig. A.11. Instalación de VirtualBox (3)

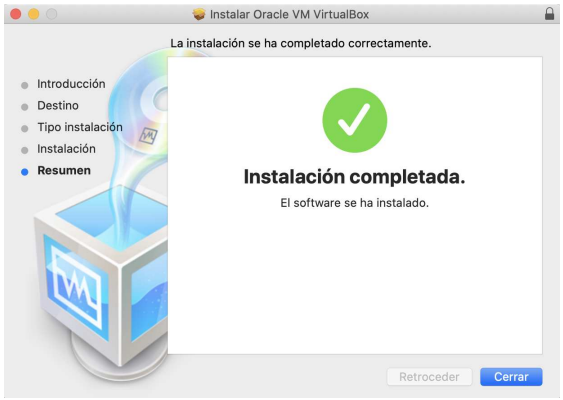


Fig. A.12. Instalación de VirtualBox (4)

Por último, se instala Minikube. De nuevo, se puede instalar con HomeBrew o con `curl`. Usar HomeBrew es más sencillo, ya que solo se ha de escribir el siguiente comando en una consola:

```
brew install minikube
```

Para comprobar que se ha instalado correctamente, se puede ejecutar:

```
minikube version
```

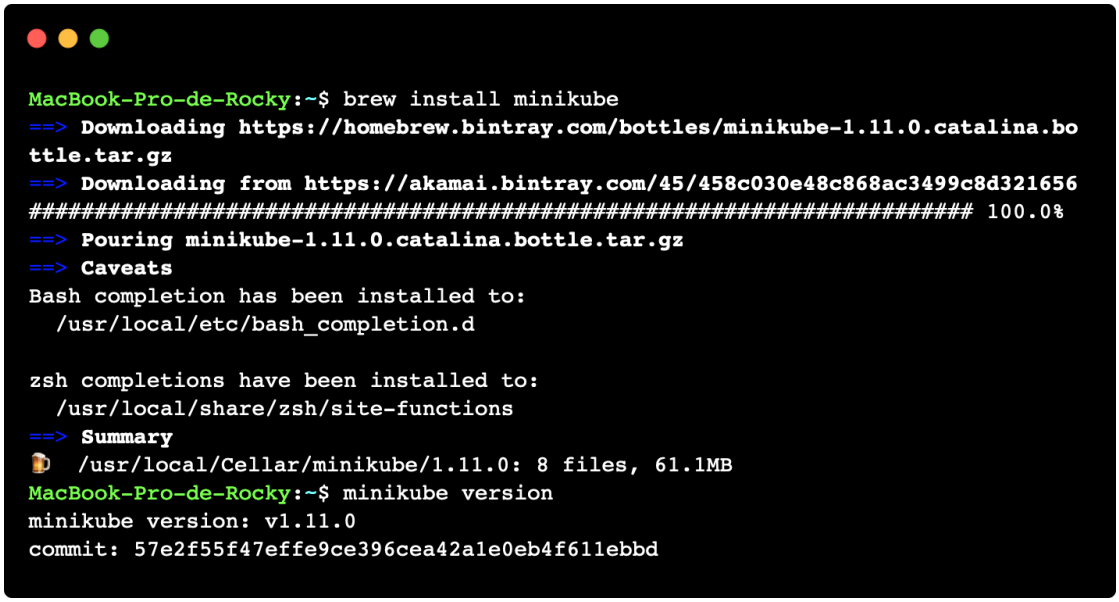


Fig. A.13. Instalación de Minikube (2)

Para inicializar un *cluster* de Kubernetes con Minikube, será necesario especificar VirtualBox como *driver* (solo la primera vez). Para ello, se debe escribir lo siguiente:

```
minikube start --vm-driver=virtualbox
```

Después se podrá omitir la especificación del `vm-driver`. Para detener la ejecución de Minikube, se ha de ejecutar:

```
minikube stop
```

La salida de estos últimos comandos se puede ver en la Fig. A.14.

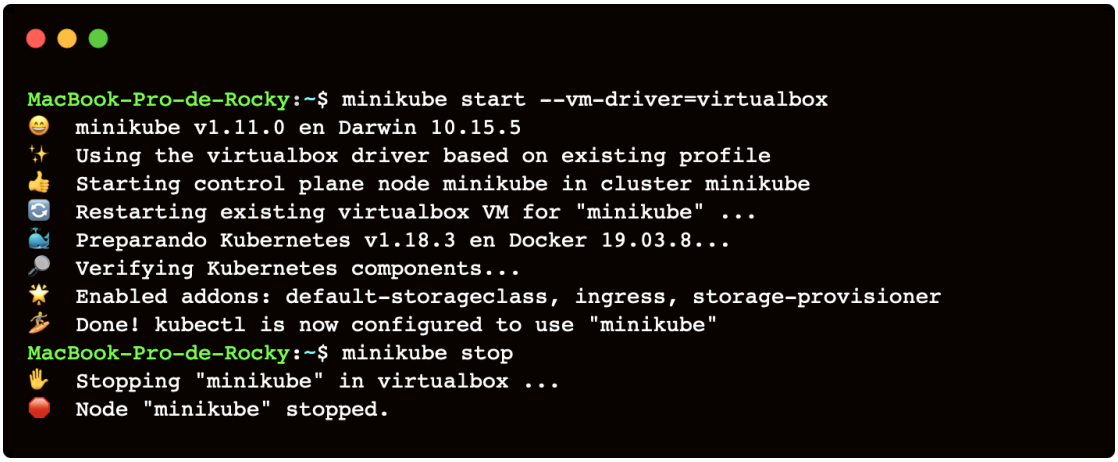


Fig. A.14. Verificación de la instalación de Minikube

Minikube es una herramienta que permite generar un *cluster* de Kubernetes de un solo nodo, de manera que este nodo se ejecuta en una máquina virtual de VirtualBox (o de otro gestor disponible para Kubernetes). Desde el sistema *host* (en este caso macOS) se puede controlar el *cluster* mediante `kubectl`, ya que al iniciar Minikube, este configura la línea de comandos de Kubernetes.

Con Minikube basta para poder desarrollar este proyecto. No obstante, al desplegar el *cluster*, el ordenador consumirá muchos recursos y podrá ralentizarlo haciendo tedioso el desarrollo de la arquitectura de microservicios en entorno local. Por este motivo, se ha utilizado una Raspberry Pi 4 para desplegar el *cluster* de Kubernetes.

Existe otro modo de desplegar Kubernetes en local, esto es, utilizando MicroK8s. Este programa se instaló en Ubuntu Server 20.04 LTS en una Raspberry Pi 4. Para ello, es necesario ejecutar los comandos que aparecen en https://microk8s.io/docs/install-alternatives#arm, que son específicos para la arquitectura ARM (propia de la Raspberry Pi). Habiendo realizado esto y reiniciado el dispositivo, se pueden seguir los pasos de instalación que aparecen en https://microk8s.io/docs/.

Una vez hecho esto, la Raspberry Pi está preparada para desplegar un *cluster* de Kubernetes (en este caso de un solo nodo).

Por otro lado, será necesario utilizar una funcionalidad experimental de Docker (`docker buildx`) para construir imágenes multiplataforma, ya que Raspberry Pi utiliza una arquitectura ARM, mientras que los ordenadores utilizan AMD. Este comando aparece en https://docs.docker.com/buildx/working-with-buildx/.

A.3. Node.js

Node.js es un entorno de ejecución de JavaScript ajeno a un navegador web. Esta tecnología proporciona gran versatilidad y rapidez a la hora de desarrollar servidores web. En este proyecto, varios microservicios se han programado con este lenguaje.

Para desarrollar con Node.js, es necesario instalar Node.js y `npm`, el cual es el gestor de paquetes por defecto de Node.js. En la página de https://www.nodejs.org/ se pueden instalar ambos programas mediante un único instalador, como se puede ver en las siguientes figuras:

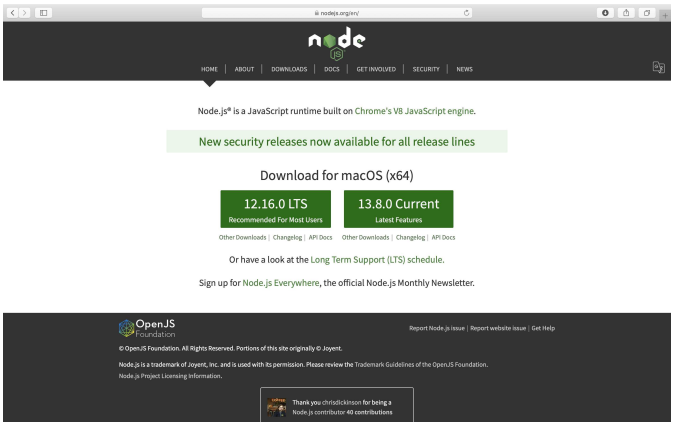


Fig. A.15. Instalación de Node.js y `npm` (1)

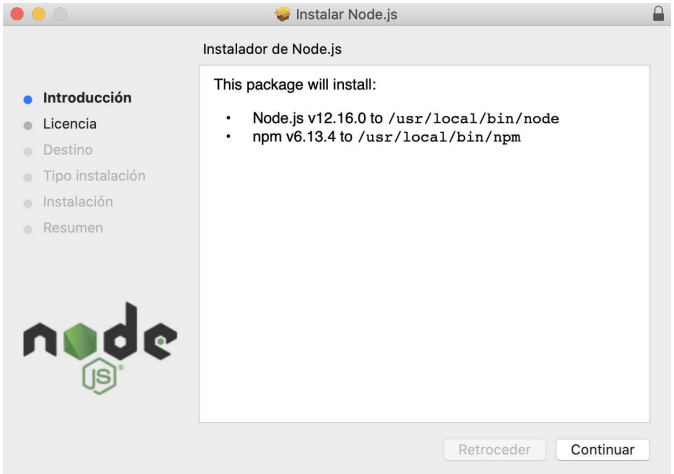


Fig. A.16. Instalación de Node.js y `npm` (2)

Fig. A.17. Instalación de Node.js y `npm` (3)

Para verificar que la instalación ha sido exitosa, se deben introducir los siguientes comandos en una consola:

```
node --version
```

```
npm --version
```

```
MacBook-Pro-de-Rocky:~$ node --version
v12.16.0
MacBook-Pro-de-Rocky:~$ npm --version
6.13.4
```

Fig. A.18. Verificación de la instalación de Node.js y `npm`

El resultado de estos comandos debería ser la versión instalada anteriormente con el instalador (ver Fig. A.18). Es posible que haya versiones más nuevas de Node.js o de `npm`. Si se desea actualizarlos a sus últimas versiones, se debe ejecutar:

```
sudo n stable
```

```
sudo npm install -g npm
```

Fig. A.19. Actualización de Node.js y `npm`

A.4. Angular CLI

Angular es un *framework* de desarrollo web del lado del cliente. Está desarrollado principalmente en TypeScript y es mantenido por Google. Se trata de un *framework* multiplataforma pensado para crear aplicaciones web de una sola página.

Para instalar Angular es necesario disponer de Node.js y `npm` en el sistema (ver Sección A.3). Para usar Angular resulta muy útil la línea de comandos de Angular (conocida como Angular CLI). Para ello, se debe abrir una consola y escribir:

```
npm install -g @angular/cli
```

Con este comando se instalará Angular CLI de manera global en el sistema operativo. A partir de este momento, se podrá utilizar el comando `ng` para crear un nuevo proyecto o para crear nuevos componentes y servicios de Angular, entre otras funcionalidades. Para verificar que Angular CLI se ha instalado correctamente, se puede ejecutar:

```
ng version
```

Este comando mostrará la versión instalada de las librerías principales de Angular (ver Fig. A.20).

Fig. A.20. Verificación de la instalación de Angular

A.5. Python

Python es un lenguaje de programación multipropósito. Se trata de un lenguaje interpretado, no compilado que se utiliza principalmente en *Machine Learning* y desarrollo web y que destaca por su sencillez y legibilidad.

Para instalar Python y poder programar con este lenguaje, conviene seguir los pasos mostrados en las siguientes figuras. En primer lugar, hay que entrar en la página https://www.python.org/downloads/ (ver Fig. A.21). Desde ahí se puede descargar un instalador. Aparte de Python, se instalarán otros programas como IDLE o Python Launcher, pero que no serán necesarios para desarrollar en Python.

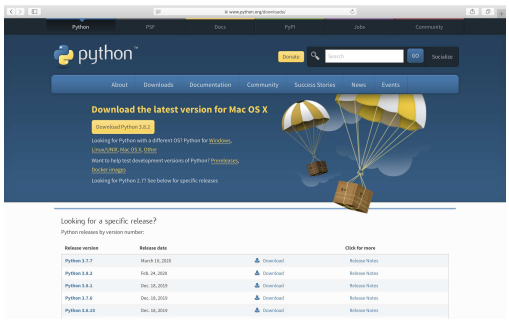


Fig. A.21. Instalación de Python (1)

Fig. A.22. Instalación de Python (2)

Fig. A.23. Instalación de Python (3)

Una vez instalado Python, se debe entrar en una consola y escribir lo siguiente:

```
python --version
```

Para poder instalar librerías de Python, se puede utilizar `pip`, el cual es un gestor de paquetes de Python. Para instalarlo, hay que descargar primero un *script* de Python mediante `curl` y ejecutar dicho *script*:

```
curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
```

```
python get-pip.py
```

Una vez hecho esto, para comprobar la correcta instalación de `pip`, se puede ejecutar el siguiente comando:

```
pip --version
```

En la Fig. A.24 se ven los resultados de estos últimos comandos. La versión de Python utilizada en este proyecto es la versión 3, ya que es la que tiene más soporte actualmente.

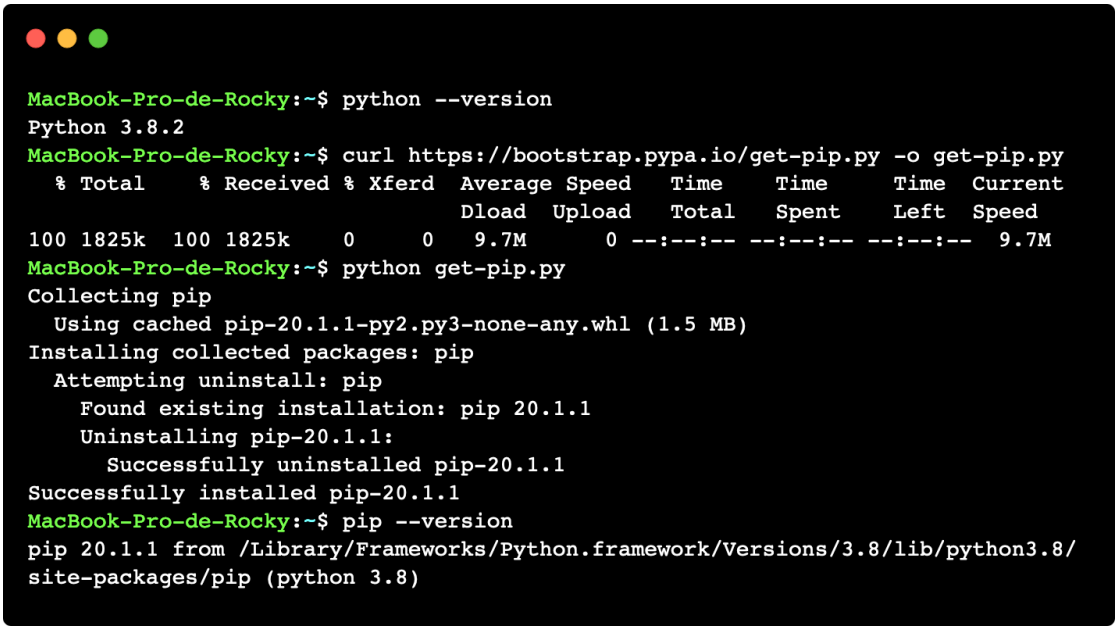


Fig. A.24. Verificación de la instalación de Python y `pip`

A.6. Go

Go es un lenguaje de programación desarrollado por Google. Se trata de un lenguaje compilado de alto rendimiento, similar a C o C++.

Se puede instalar Go de varias maneras. En la página oficial de Go (https://golang.org/dl/) se puede descargar un instalador o un archivo comprimido con los binarios del programa.

No obstante, en macOS también se puede instalar mediante HomeBrew, lo cual es bastante más sencillo y fácil de mantener y actualizar. Se debe ejecutar el siguiente comando:

```
brew install go
```

Para verificar que se ha instalado correctamente, se puede escribir lo siguiente en una consola:

```
go version
```

El resultado de estos dos comandos se muestra en la Fig. A.25. Aparte del número de versión, también aparecerá la arquitectura de la máquina en la que se ha instalado Go (en el caso de macOS, esta es darwin/amd64).

```
MacBook-Pro-de-Rocky:~$ brew install go
==> Downloading https://homebrew.bintray.com/bottles/go-1.14.3.catalina.bottle.t
ar.gz
==> Downloading from https://akamai.bintray.com/e3/e3c4f834acba4e4db3f2b848860a0
######################################################################## 100.0%
==> Pouring go-1.14.3.catalina.bottle.tar.gz
🍺  /usr/local/Cellar/go/1.14.3: 9,441 files, 424.6MB
MacBook-Pro-de-Rocky:~$ go version
go version go1.14.3 darwin/amd64
```

Fig. A.25. Instalación de Go

A.7. Visual Studio Code

Visual Studio Code es un editor de código *open-source* que dispone de un gran número de extensiones para desarrollar en distintos lenguajes de programación. Además, dispone de IntelliSense (función de autocompletado de código).

Se ha utilizado este editor de código porque proporciona muchas ventajas a la hora de programar. Por ejemplo, disponer de una consola integrada, poder programar en varios lenguajes o poder utilizar Git para la gestión de versiones del proyecto.

Instalar Visual Studio Code es muy sencillo. La instalación se puede realizar entrando en https://code.visualstudio.com/download y seleccionando el sistema operativo de la máquina en la que se vaya a usar (ver Fig. A.26). El archivo descargado será la aplicación como tal.

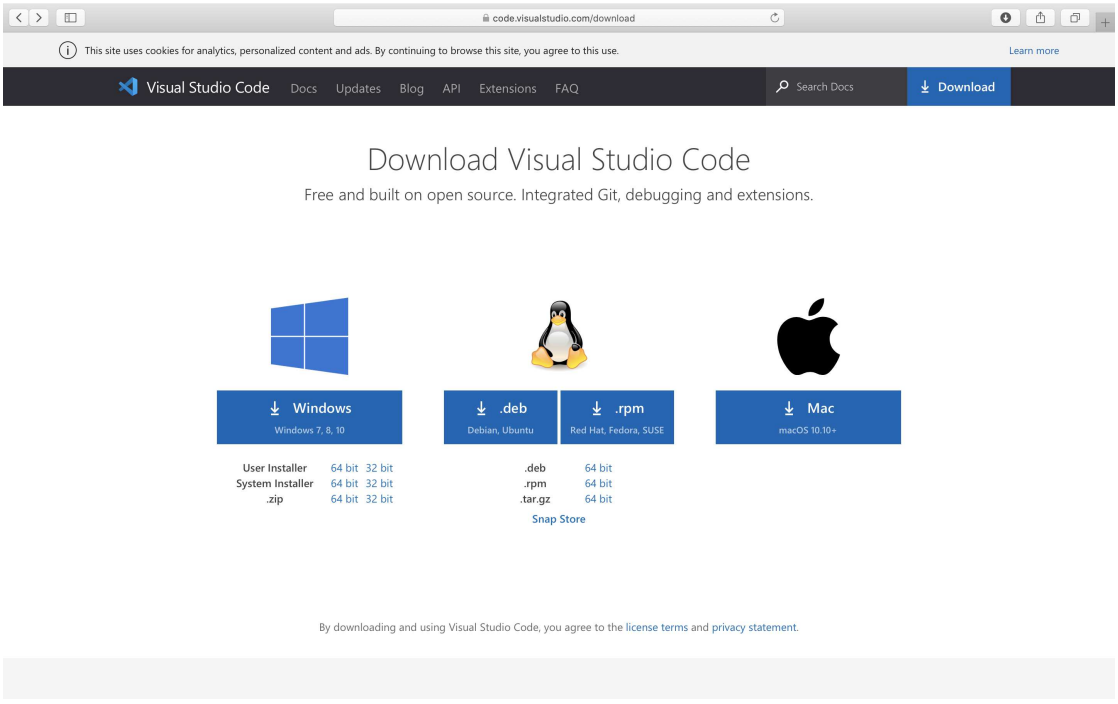


Fig. A.26. Instalación de Visual Studio Code

Para desarrollar el presente proyecto, se han utilizado las siguientes extensiones de Visual Studio Code:

- Anaconda Extension Pack. Ofrece utilidades que mejoran la experiencia al programar con Python.
- Angular Language Service. Útil para desarrollar aplicaciones de Angular.
- Docker. Proporciona una sencilla interfaz de usuario para manejar contenedores e imágenes de Docker, además de conectarse a Docker Hub.
- HTML CSS Support. Ayuda a desarrollar con lenguajes de marcado como HTML, CSS o MD, entre otros.
- Go. Muy útil para programar en Go, ya que posee autocompletado de código y un *linter* habilitado por defecto. No obstante, al ser una extensión bastante nueva, aún no soporta ciertas funcionalidades, como los módulos de Go.
- Python. Útil para desarrollar con Python y obtener sugerencias de lenguaje.
- YAML. Proporciona soporte para los archivos de manifiesto de Kubernetes.

No obstante, ninguna de estas extensiones es imprescindible para trabajar con el proyecto; aunque facilitan bastante el desarrollo, sobre todo para codificar en Angular y YAML.

A.8. Arduino IDE

Arduino IDE es el entorno de desarrollo integrado más común para programar un microcontrolador de Arduino. Aunque existen extensiones para Visual Studio Code que proveen prácticamente las mismas funcionalidades que ofrece Arduino IDE, se ha usado esta aplicación para utilizar la placa de Arduino.

Este entorno de desarrollo ofrece un editor de código, múltiples programas de ejemplo, librerías de código y una consola para capturar la salida serie del Arduino (conectado por USB al ordenador).

Para instalar Arduino IDE, se debe entrar a la página https://www.arduino.cc/en/main/software y descargar la aplicación para el sistema operativo que se esté utilizando (ver Fig. A.27).

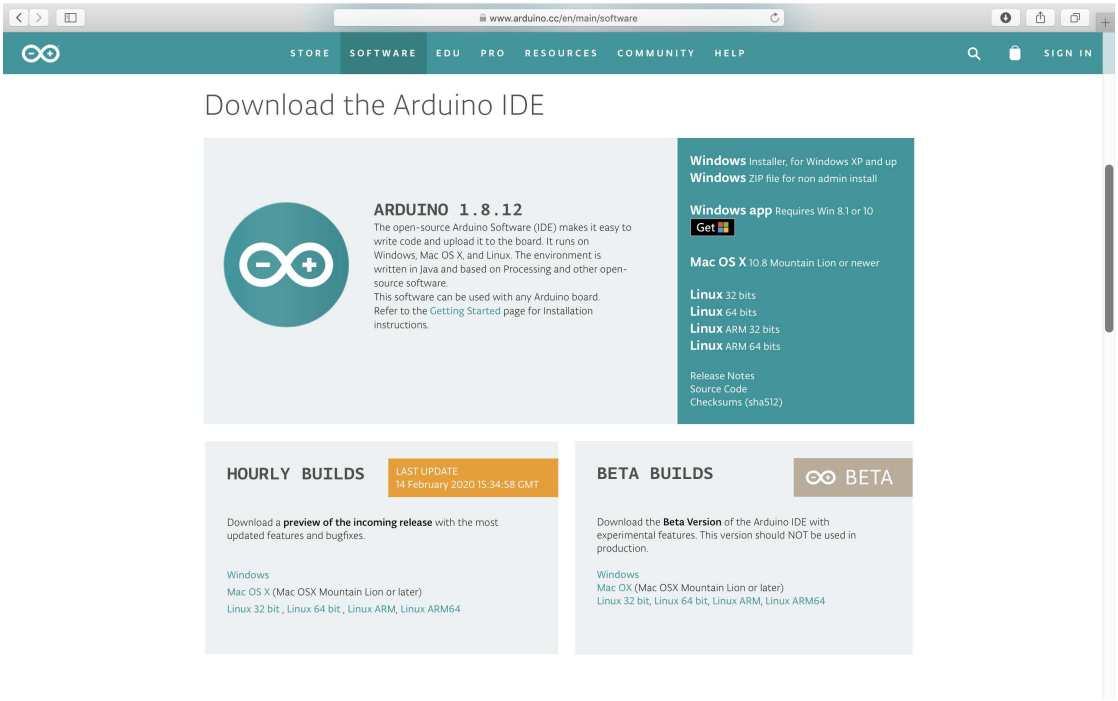


Fig. A.27. Instalación de Arduino IDE

A.9. Postman

Postman es una plataforma muy útil para el desarrollo de API REST. Se puede utilizar para realizar peticiones HTTP especificando la URL, el método HTTP, los parámetros de consulta o el cuerpo de la petición en distintos formatos, cabeceras HTTP, *tokens*, *cookies*, etc.

Además, permite realizar pruebas con las respuestas obtenidas, proporcionando un gran número de funciones para realizar aserciones. También permite integración con lenguajes de programación como Node.js con librerías como *newman*, para obtener los resultados de las pruebas de manera programática y poder, por ejemplo diseñar una plantilla de resultados personalizada.

En este proyecto se utiliza Postman para realizar llamadas a las rutas de los distintos microservicios de manera independiente, en entorno de desarrollo. De esta manera se puede ver el contenido de los mensajes entre microservicios de forma directa.

Para descargar Postman, simplemente hay que entrar en https://www.postman.com/downloads/ y darle al botón de descarga (ver Fig. A.28). Este programa está disponible para Windows, macOS y la mayoría de distribuciones de Linux. El archivo descargado será la aplicación en sí.

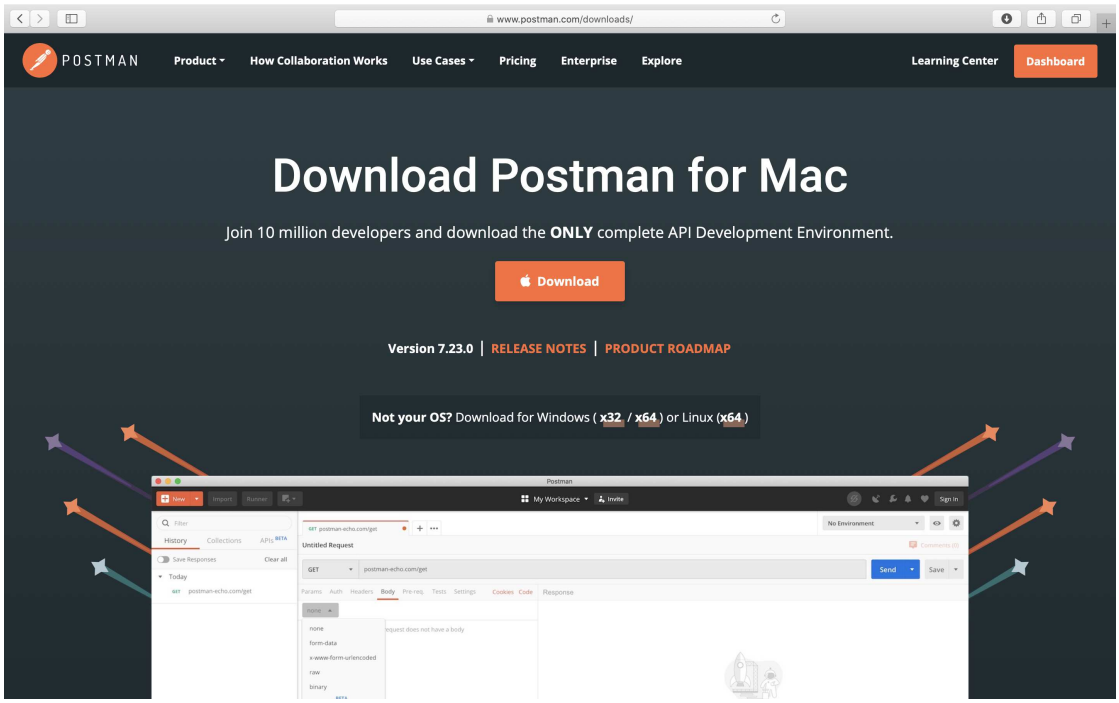


Fig. A.28. Instalación de Postman

Anexo B. Manual de usuario

En este anexo se presenta un manual para utilizar la aplicación desarrollada en el presente Trabajo Fin de Grado. Este manual se ha dividido en cuatro puntos: un manual para desplegar la arquitectura de microservicios, un manual para usuarios finales de la aplicación, un manual de uso del *cluster* y un cuarto manual para desarrolladores.

Cabe mencionar que el código desarrollado está disponible en un repositorio de GitHub, accesible en https://github.com/7Rocky/IoT_Microservices, y que las imágenes de Docker utilizadas están disponibles en Docker Hub, en https://hub.docker.com/u/7rocky.

B.1. Manual de despliegue

En primer lugar, se deberá clonar o descargar el repositorio de GitHub con el código escrito para este proyecto. Suponiendo que se tiene Git instalado[5], se puede ejecutar lo siguiente en una consola:

```
git clone https://github.com/7Rocky/IoT_Microservices

cd IoT_Microservices/
```

El resultado obtenido será algo similar a la Fig. B.1. Una vez en el directorio de trabajo, se pueden listar los contenidos del mismo mediante el comando `ls -la`, como muestra la Fig. B.2.

[5]Si no se dispone de Git, se puede descargar un archivo comprimido desde el repositorio de GitHub

```
MacBook-Pro-de-Rocky:~$ git clone https://github.com/7Rocky/IoT_Microservices
Clonando en 'IoT_Microservices'...
remote: Enumerating objects: 854, done.
remote: Counting objects: 100% (854/854), done.
remote: Compressing objects: 100% (453/453), done.
remote: Total 3119 (delta 479), reused 686 (delta 370), pack-reused 2265
Recibiendo objetos: 100% (3119/3119), 1.83 MiB | 3.99 MiB/s, listo.
Resolviendo deltas: 100% (1766/1766), listo.
MacBook-Pro-de-Rocky:~$ cd IoT_Microservices/
MacBook-Pro-de-Rocky:~/IoT_Microservices$
```

Fig. B.1. Clonación del repositorio de GitHub

```
MacBook-Pro-de-Rocky:~/IoT_Microservices$ ls -la
total 40
drwxr-xr-x  18 rocky  staff   576 21 jun 14:26 .
drwxr-xr-x+ 63 rocky  staff  2016 21 jun 14:26 ..
drwxr-xr-x  12 rocky  staff   384 21 jun 14:26 .git
-rw-r--r--   1 rocky  staff   426 21 jun 14:26 .gitignore
-rw-r--r--   1 rocky  staff   738 21 jun 14:26 LICENSE
-rw-r--r--   1 rocky  staff  1872 21 jun 14:26 README-raspberryPi.md
-rw-r--r--   1 rocky  staff  7742 21 jun 14:26 README.md
drwxr-xr-x   6 rocky  staff   192 21 jun 14:26 angular-ms
drwxr-xr-x   3 rocky  staff    96 21 jun 14:26 arduino-iot
drwxr-xr-x   4 rocky  staff   128 21 jun 14:26 auth-ms
drwxr-xr-x   6 rocky  staff   192 21 jun 14:26 fake-arduino-iot
drwxr-xr-x   5 rocky  staff   160 21 jun 14:26 manifests-k8s
drwxr-xr-x   8 rocky  staff   256 21 jun 14:26 measure-ms
drwxr-xr-x   8 rocky  staff   256 21 jun 14:26 microcontrollers-ms
drwxr-xr-x   4 rocky  staff   128 21 jun 14:26 mysql-iot
drwxr-xr-x   8 rocky  staff   256 21 jun 14:26 orchestrator-ms
drwxr-xr-x   8 rocky  staff   256 21 jun 14:26 publisher-ms
drwxr-xr-x   7 rocky  staff   224 21 jun 14:26 stats-ms
```

Fig. B.2. Directorio de trabajo del proyecto desarrollado

Para configurar el comando de `kubectl`, basta con iniciar los servicios de Minikube o MicroK8s (`minikube start` o `microk8s start`). Véase la Sección A.2 para más información sobre la instalación de Kubernetes.

Para desplegar el *cluster* en producción, en primer lugar hay que aplicar los manifiestos de configuración de Kubernetes. En el directorio `manifests-k8s` hay tres

subdirectorios: `config`, `prod` y `dev` (este último no es necesario).

En el directorio `config` hay manifiestos con variables de entorno y manifiestos de `PersistentVolumeClaim` para MySQL y MongoDB. Existe otro archivo llamado `angular-ingress.yaml`, que contiene un `Ingress`, pero que no es necesario desplegar.

En cuanto a los `PersistentVolumeClaims`, existen dos por cada tipo de base de datos. Esto es porque en Minikube y en MicroK8s, el despliegue es ligeramente distinto. En Minikube habría que aplicar `mysql-pv-claim.yaml`, mientras que en MicroK8s habría que utilizar `mysql-pv-claim_rp.yaml`. Y lo mismo con el `PersistentVolumeClaim` de MongoDB.

El ejemplo mostrado se realiza con MicroK8s en una Raspberry Pi 4 con Ubuntu Server 20.04 LTS, en la dirección IP 192.168.1.222 de red de área local. Pero sería un procedimiento muy parecido en Minikube. Entonces, será necesario ejecutar:

```
cd manifests-k8s/config/

kubectl apply -f env-configmap.yaml -f secrets.yaml

cd pvc-k8s/

kubectl apply -f mongo-pv-claim_rp.yaml -f mysql-pv-claim_rp.yaml
```

```
ubuntu@ubuntu:~/IoT_Microservices$ cd manifests-k8s/config/
ubuntu@ubuntu:~/IoT_Microservices/manifests-k8s/config$ kubectl apply -f env-con
figmap.yaml -f secrets.yaml
configmap/env-configmap created
secret/secrets created
ubuntu@ubuntu:~/IoT_Microservices/manifests-k8s/config$ cd pvc-k8s/
ubuntu@ubuntu:~/IoT_Microservices/manifests-k8s/config/pvc-k8s$ kubectl apply -f
 mongo-pv-claim_rp.yaml -f mysql-pv-claim_rp.yaml
persistentvolumeclaim/mongo-pv-claim created
persistentvolumeclaim/mysql-pv-claim created
```

Fig. B.3. Configuración del *cluster* de Kubernetes

Una vez hecho esto, habrá que desplegar los manifiestos del directorio `prod`, los cuales contienen `Deployments`, `StatefulSets`, `Services` y un `CronJob` con las

imágenes de Docker creadas para cada microservicio (disponibles en Docker Hub). Los comandos a utilizar son:

```
cd manifests-k8s/

kubectl apply -f prod
```

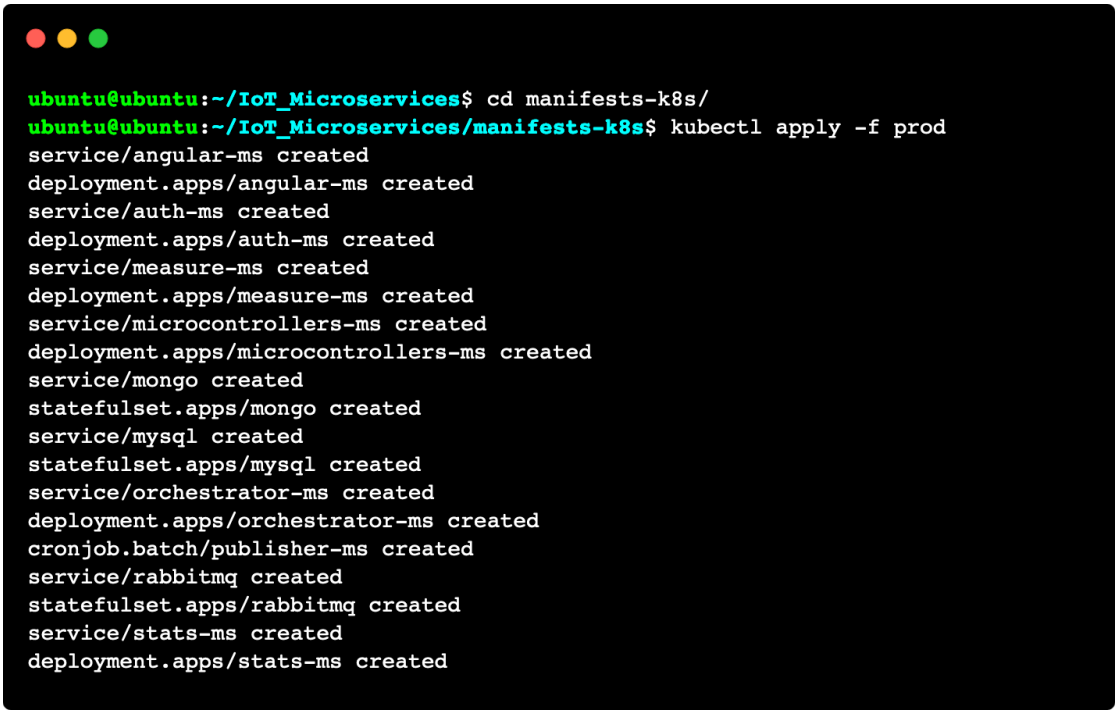


Fig. B.4. Despliegue del *cluster* de Kubernetes en producción

La manera más eficaz de saber que todos los objetos de Kubernetes están ejecutándose es mediante `kubectl get all`.

Hasta este punto, el *cluster* de Kubernetes ya estaría desplegado. Si se accede ahora a la dirección http://192.168.99.100:31600 (en el caso de Minikube[6]) o a la dirección http://192.168.1.222:31600 (en el caso de MicroK8s[7]), se podrá visualizar la aplicación de Angular desarrollada en el proyecto.

[6]Puede ser que la dirección no sea exactamente la 192.168.99.100. En caso de duda, se puede utilizar `minikube ip` para obtener la dirección concreta; o bien, `minikube service angular-ms` para que se abra el navegador automáticamente en la dirección apropiada

[7]Se ha de especificar la dirección IP de la máquina en la que se está ejecutando MicroK8s

Para detener la ejecución del *cluster* se pueden utilizar los siguientes comandos, similares a los anteriores:

```
cd manifests-k8s/

kubectl delete -f prod
```

No es necesario eliminar los manifiestos de configuración. De hecho, si se borran los `PersistentVolumeClaims` de MySQL y MongoDB, se perderán los datos almacenados.

B.2. Manual para usuarios finales

En primer lugar, para poder utilizar la aplicación es necesario que el *cluster* de Kubernetes esté desplegado (ver Sección B.1). Una vez desplegado, se puede acceder a la dirección correspondiente (según si se utiliza Minikube o MicroK8s) y visualizar la primera página de la aplicación:

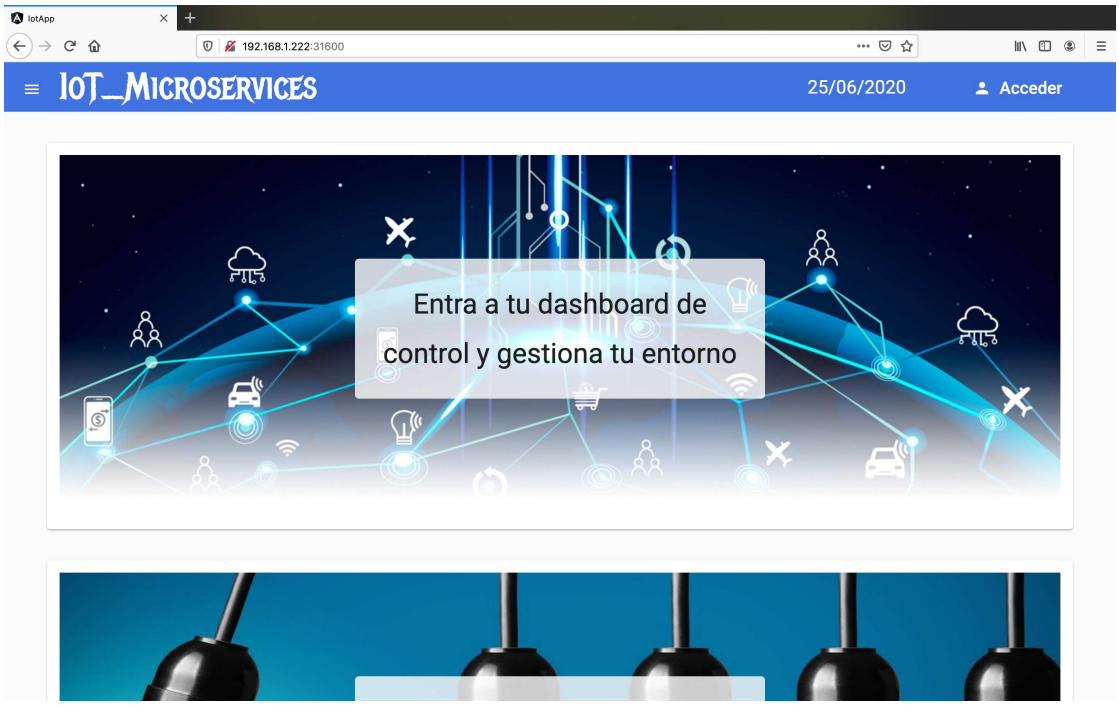


Fig. B.5. Página de bienvenida de la aplicación desarrollada

En esta página se pueden ver tres tarjetas (haciendo *scroll*) con las cuales se puede acceder al *dashboard* personal del usuario, a su centro de control de bombillas inteligentes o a su listado de sensores y microcontroladores.

Además, existe un menú desplegable a la izquierda, como se ve en la Fig. B.6, el cual presenta algunas opciones más, como visualizar los sensores de temperatura o los de humedad del usuario.

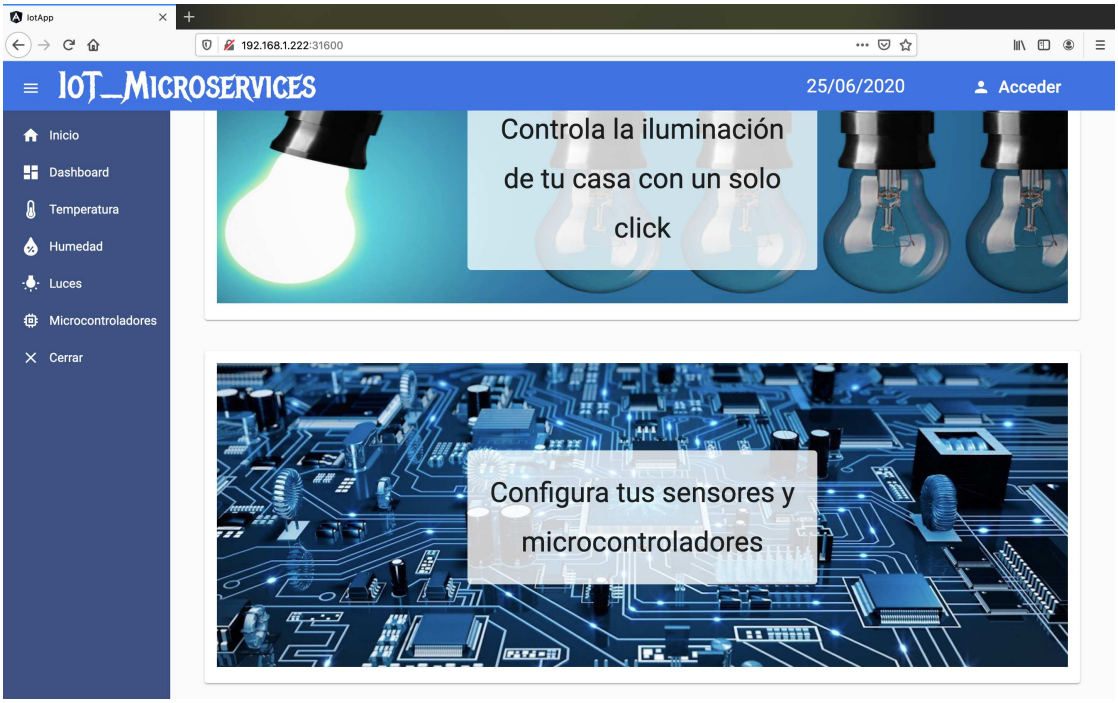


Fig. B.6. Página de bienvenida con menú desplegado

Todos estos enlaces están protegidos, ya que contienen información personal del usuario. Por ello, es necesario que el usuario esté registrado y autenticado antes de acceder a este contenido. Para iniciar sesión, se tiene un botón en la esquina superior derecha de la pantalla, dentro de la barra de navegación (como se puede ver en las figuras anteriores).

Otra opción es intentar acceder al contenido personal, ya que la aplicación detectará que el usuario no está autenticado y desplegará una ventana de diálogo con el formulario de inicio de sesión o registro antes de que pueda entrar, como se muestra en la siguiente Fig. B.7.

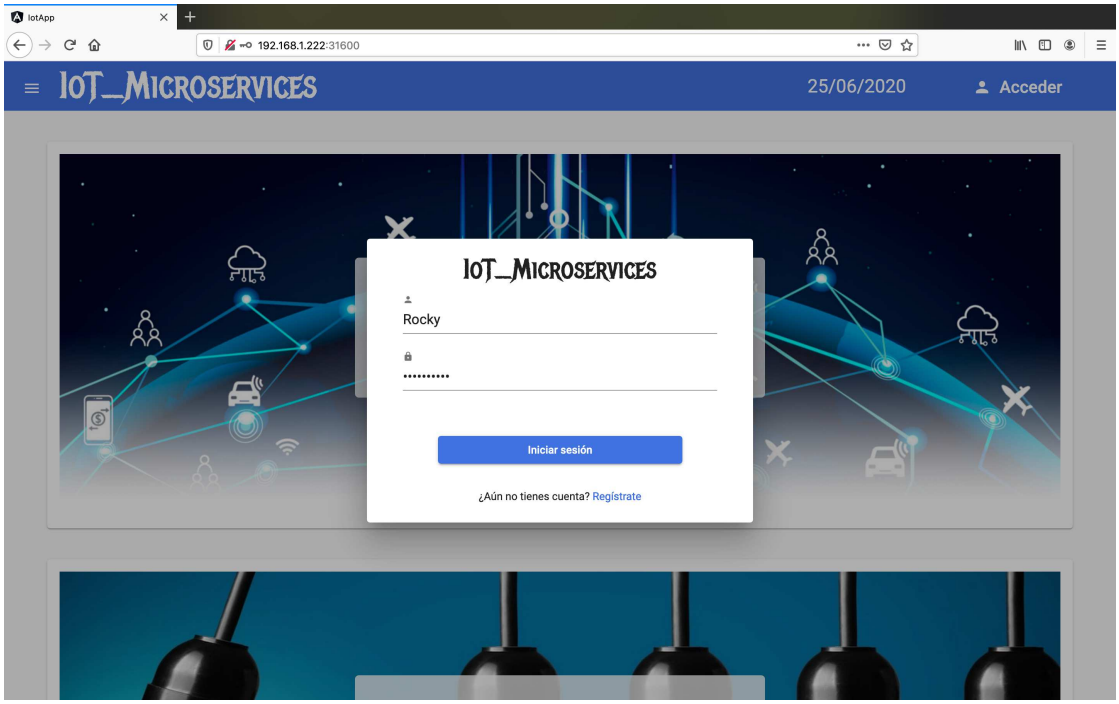


Fig. B.7. Ventana de diálogo para iniciar sesión o registrarse

Por comodidad del uso, y debido al sistema de autenticación utilizado, la página recordará al usuario siempre que este no cierre su sesión explícitamente o no acceda a la plataforma en mucho tiempo.

Una vez autenticado, el usuario podrá visualizar su nombre en la esquina superior derecha. Además, podrá acceder a su contenido sin ningún tipo de restricción.

En su *dashboard* podrá observar las medidas que están tomando todos sus sensores en tiempo real (con un refresco automático cada 10 segundos o manual).

Como se puede observar en la Fig. B.8, existen unas franjas de color azul que indican el tipo de medida, el sensor que la está realizando, en qué dirección IP se encuentra, y el estado de su conexión.

En la Fig. B.8 se observa que la conexión es estable. Mientras que en la Fig. B.9 el dispositivo no está conectado correctamente al sistema. En esta Fig. B.9 se muestran también las opciones que aparecen al desplegar el menú de la franja de color azul (“Estadísticas” y “Editar”).

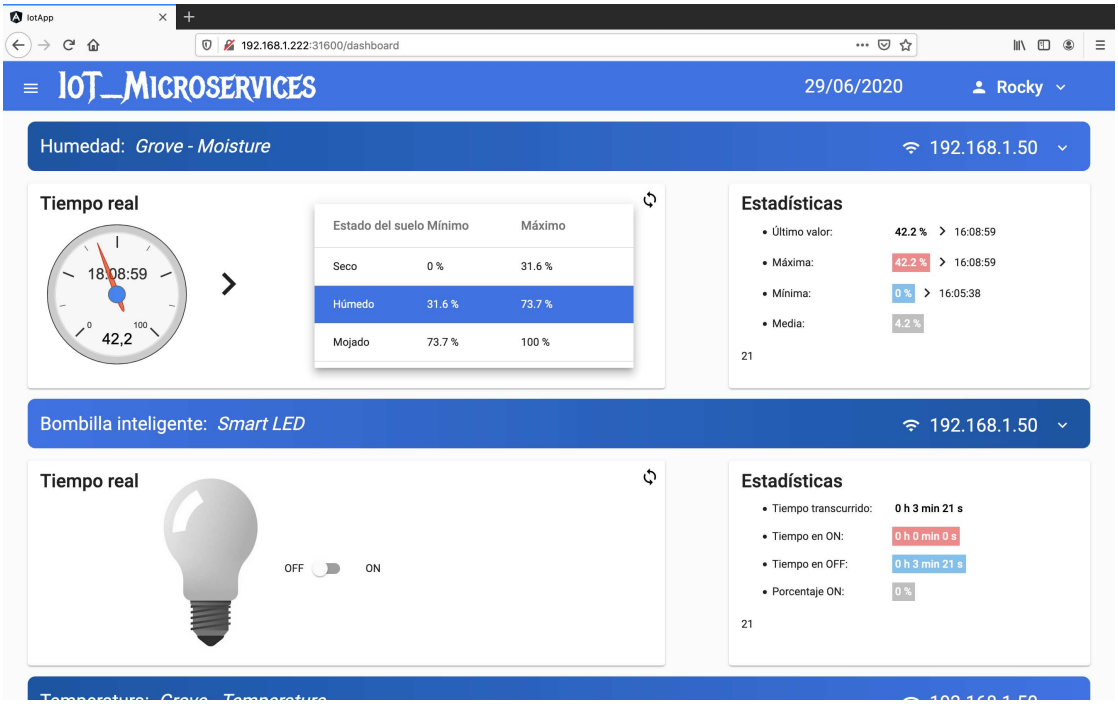


Fig. B.8. Vista del *dashboard* del usuario (1)

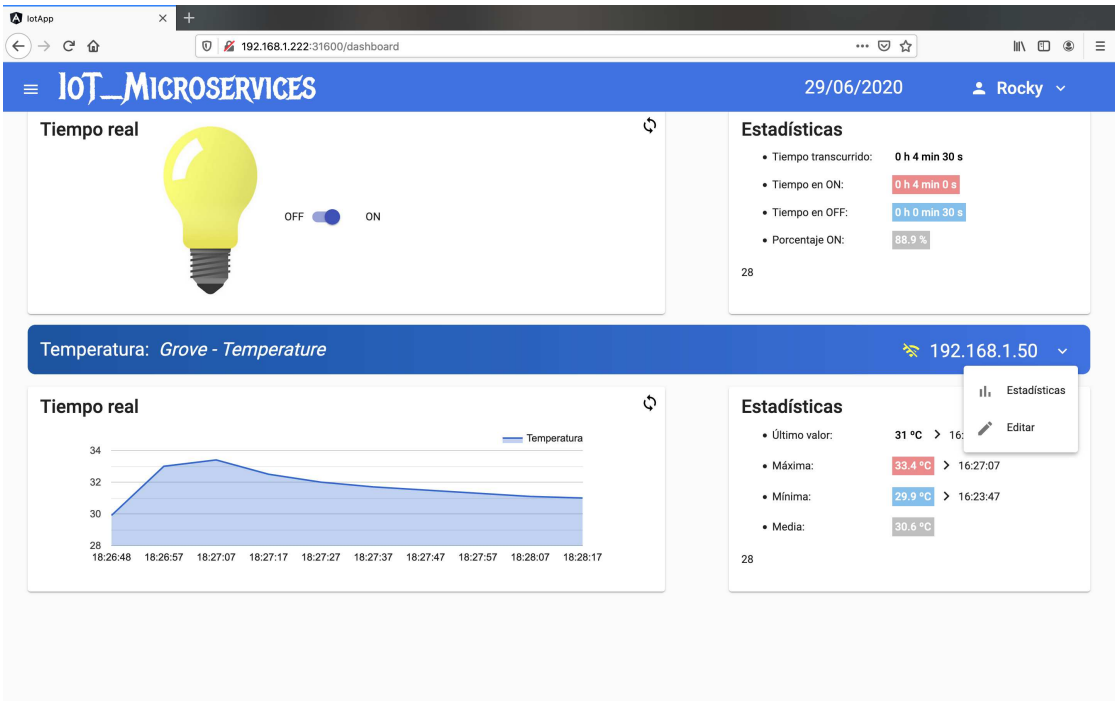


Fig. B.9. Vista del *dashboard* del usuario (2)

Como se puede observar, la estructura de la página del *dashboard* consta de una franja con la información del sensor y dos tarjetas debajo de esta. La tarjeta de la izquierda muestra visualmente las medidas del sensor en tiempo real, mientras que en la tarjeta de la derecha se observan diferentes valores estadísticos medidos conforme se van recibiendo datos.

Como se vio en la Fig. B.8, la manera de mostrar los valores de humedad es mediante un `GaugeChart` que muestra el porcentaje de humedad existente y el instante en el que se tomó la medida. Además, se añade una tabla con los tipos de suelo según el porcentaje de humedad, señalando el tipo correspondiente al porcentaje recibido.

Por otro lado, la bombilla inteligente se muestra mediante una imagen de una bombilla apagada o encendida (según el estado real de la misma). También se incluye un interruptor para poder apagar o encender la bombilla desde la aplicación.

En la Fig. B.9 se ve que los datos de temperatura se muestran en un `AreaChart`, de manera que se van añadiendo los últimos datos a la gráfica para visualizar la evolución de la temperatura en el tiempo.

Las estadísticas se limitan a calcular el valor medio de todos los datos recibidos, los valores máximo y mínimo y el último dato recibido, añadiendo el instante de tiempo correspondiente. En el caso de la bombilla inteligente, se muestra el tiempo que está la bombilla encendida y apagada y la relación entre ambos tiempos.

Al seleccionar la opción de "Estadísticas" del menú desplegable que aparece en la Fig. B.9 (disponible para humedad, luz y temperatura), se abre una página con la vista del historial de mediciones.

En esta página, el usuario selecciona dos fechas distintas (una de inicio y una de fin) para cargar los datos guardados de una magnitud en concreto entre las fechas indicadas[8]. Al pulsar el botón de "Cargar", se mostrará mediante un `LineChart` una gráfica con los valores medios de todas las horas guardadas en base de datos.

Adicionalmente, el usuario puede seleccionar si quiere visualizar los valores máximos, mínimos o medios (o una combinación de ambos) en la misma gráfica. Los valores máximos se representan mediante una línea de color rojo; los valores mínimos, con una línea de color azul; mientras que los valores medios se representan con una línea de color verde (como se muestra en la Fig. B.10).

[8]Por defecto, la fecha final será la fecha actual y la fecha inicial será un día anterior

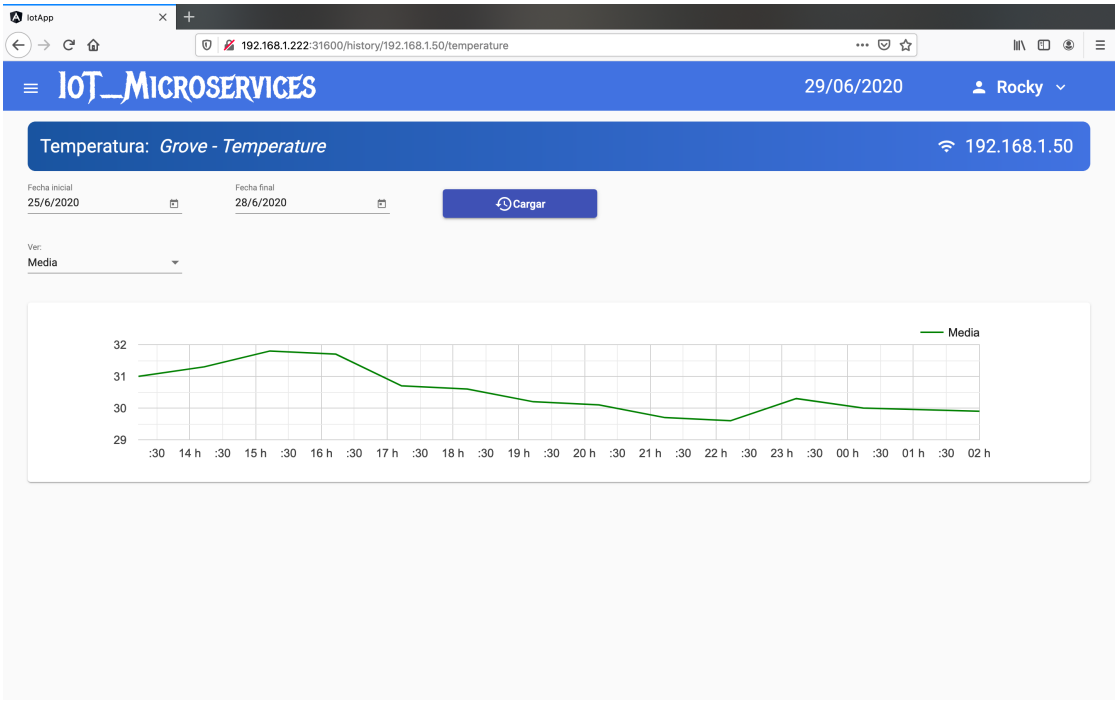


Fig. B.10. Historial de mediciones

Si no hubiera datos guardados en algún intervalo de tiempo, aparecerá una ventana de diálogo informando al usuario.

Aunque es evidente, el usuario no podrá indicar una fecha de inicio que sea posterior o igual a la fecha final, ni posterior a la fecha actual.

En la página de microcontroladores (accesible desde el menú y desde la página de inicio) se muestra la lista de sensores del usuario, indicando el tipo de medida que realizan, el nombre del sensor y la dirección IP que tiene el microcontrolador, como se muestra en la Fig. B.11.

Cabe mencionar que las direcciones IP han de pertenecer a la misma red de área local que la máquina en la que se ejecuta el *cluster* de Kubernetes (en este caso, la red es la 192.168.1.0/24). Esto es debido a que la aplicación solamente funciona en entorno local.

En la tabla de la Fig. B.11, se muestran también opciones para editar o eliminar el sensor seleccionado. También existe un botón debajo de la tabla con la opción de añadir un nuevo sensor a la lista.

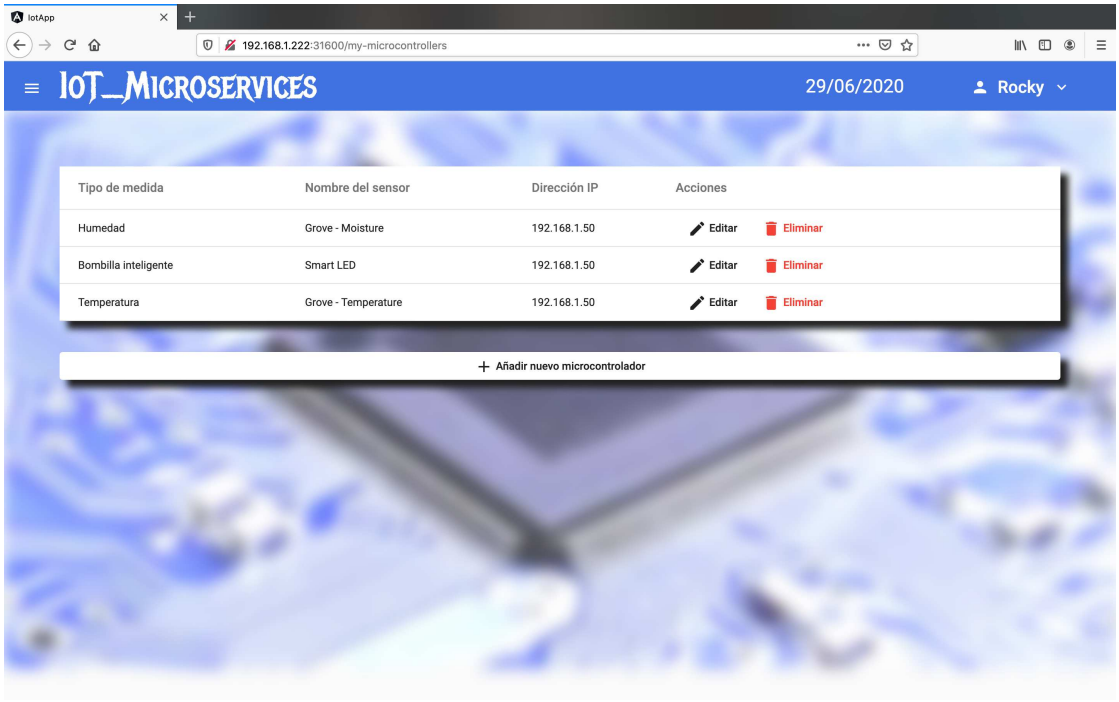


Fig. B.11. Vista de los microcontroladores del usuario

Al pinchar en "Editar" (al igual que en el menú desplegable de la Fig. B.9), se abre una página con un formulario para poder modificar los valores del sensor. Como se ve en la Fig. B.12, se trata de un formulario guiado que consta de cuatro pasos.

En el paso número 3, se muestra una lista de los sensores compatibles con el sistema desarrollado y la magnitud elegida en el paso número 1. Así, se evitan conflictos entre magnitudes y sensores.

Si se pulsa el botón de "Eliminar", evidentemente, se elimina el sensor seleccionado. De todas maneras, en caso de equivocación es sencillo volver a añadirlo.

Al momento de añadir un nuevo sensor, también se abre el formulario de la Fig. B.12. No obstante, en este caso, el formulario estará vacío, y no se podrá pasar al siguiente paso hasta que se hayan rellenado los campos anteriores. Así, se evita que el usuario envíe un formulario incompleto.

Cabe mencionar que las páginas mostradas están acondicionadas para pantallas de teléfono móvil, de manera que la aplicación web desarrollada sea *responsive* y pueda ser utilizada desde distintos tipos de dispositivo.

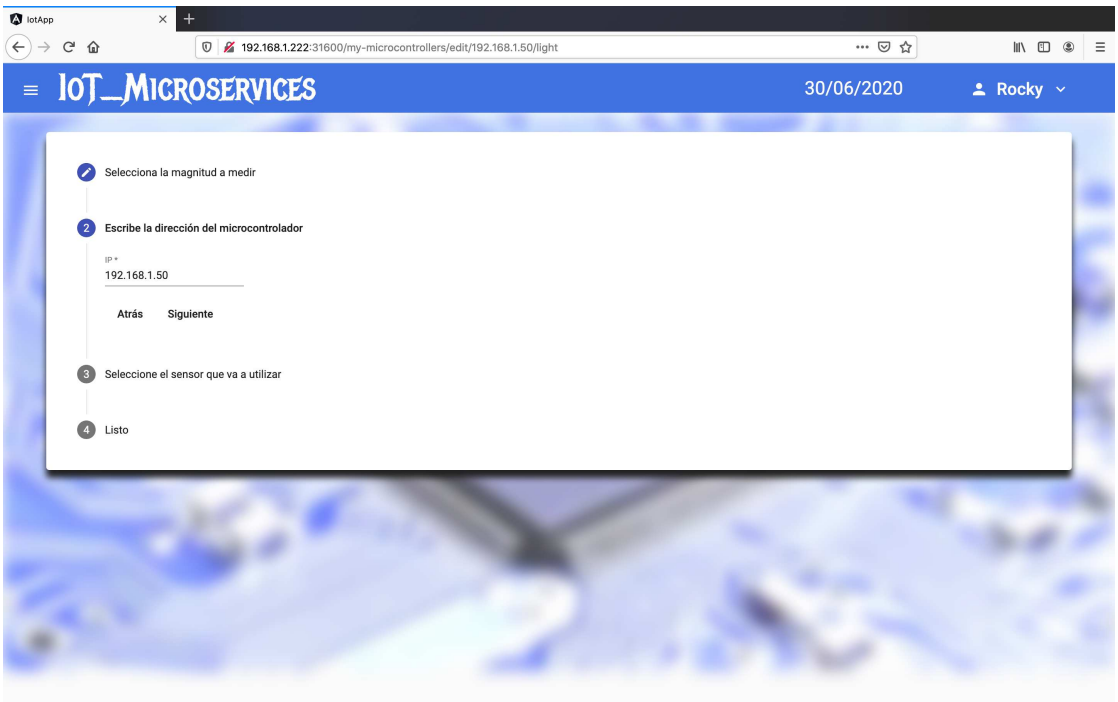


Fig. B.12. Formulario de registro y edición de microcontroladores

B.3. Manual de uso del *cluster*

Una manera de depurar errores en los `Pods` se consigue con los siguientes comandos de `kubectl`:

- `kubectl logs pod/<identificador-pod> [-f]`
- `kubectl describe pod/<identificador-pod>`
- `kubectl exec -it pod/<identificador-pod> -- <comando>`

El primer comando mostrará los registros del `Pod` correspondiente. Si se especifica la opción `-f`, la consola se quedará escuchando indefinidamente, recibiendo los registros en tiempo real. La siguiente instrucción muestra la información del `Pod`, y entre esta información, se pueden ver los distintos estados por los que ha pasado dicho `Pod`. Y el último comando sirve para ejecutar comandos desde dentro del `Pod`, típicamente se ejecuta `sh` o `bash`, o algún comando propio del `Pod` (como el

comando `mongo -u root -p` en un `Pod` de MongoDB).

Una segunda manera de depurar errores, más amigable, consiste en abrir el *dashboard* de Kubernetes. Desde este *dashboard* se puede realizar todo lo anterior, pero con interfaz gráfica. Por ejemplo, se puede entrar en los `Pods` mediante una consola integrada en la propia página web y visualizar los registros y el estado de cada `Pod`, entre otras posibilidades.

Para utilizar el *dashboard* de Kubernetes es necesario disponer de un *addon* tanto en Minikube como en MicroK8s:

```
minikube addons enable dashboard
```

```
microk8s enable dashboard
```

En Minikube es muy sencillo abrir este *dashboard*, simplemente hay que utilizar un comando para que se abra automáticamente una pestaña del navegador:

```
minikube dashboard
```

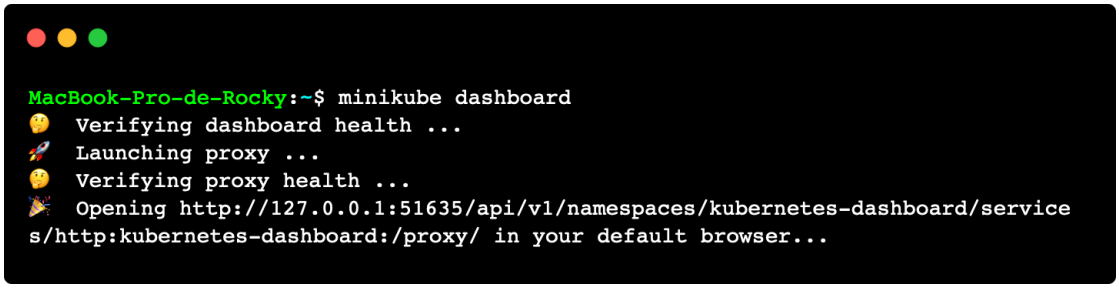


Fig. B.13. *Dashboard* de Kubernetes desde Minikube

En MicroK8s es algo más complicado de desplegar. Es necesario ejecutar primero:

```
token=$(microk8s kubectl -n kube-system get secret |
    grep default-token | cut -d " " -f1)
```

```
microk8s kubectl -n kube-system describe secret $token
```

Estos comandos generan un *token* de acceso al *dashboard* (ver Fig. B.14). En la página de MicroK8s, recomiendan utilizar lo siguiente:

```
microk8s kubectl port-forward -n kube-system
    service/kubernetes-dashboard 10443:443
```

Este comando funciona para acceder al *dashboard* desde la máquina en la que se ejecuta MicroK8s. Como este no era el caso, se encontró una solución alternativa, en la que se cambia el tipo de puerto al `service/kubernetes-dashboard` para poner un `NodePort` y que así pueda ser accedido desde fuera. Será necesario ejecutar:

```
kubectl -n kube-system edit service/kubernetes-dashboard
```

Este abrirá un editor de la consola (como Vim) con el manifiesto en YAML. Habrá que cambiar una línea (`type: ClusterIP` por `type: NodePort`) y añadir un valor de `NodePort` válido[9] a la lista de `ports`, como se muestra en el Código B.1.

```
ubuntu@ubuntu:~$ token=$(microk8s kubectl -n kube-system get secret | grep defau
lt-token | cut -d " " -f1)
ubuntu@ubuntu:~$ microk8s kubectl -n kube-system describe secret $token
Name:         default-token-bhwnz
Namespace:    kube-system
Labels:       <none>
Annotations:  kubernetes.io/service-account.name: default
              kubernetes.io/service-account.uid: 2a0a7012-4318-4f73-b147-b1d5263
              9a07e

Type:  kubernetes.io/service-account-token

Data
====
ca.crt:     1103 bytes
namespace:  11 bytes
token:      eyJhbGciOiJSUzI1NiIsImtpZCI6IlVKMFZpckQxRTdzWFBYbTEyWC1FR2dfenJSbUc3
WUtMRjdLVUxZZnFnaEEifQ.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZXJuZ
XRlcy5pby9zZXJ2aWNlYWNjb3VudC9uYW1lc3BhY2UiOiJrdWJlLXN5c3RlbSIsImt1YmVybmV0ZXMua
W8vc2VydmljZWFjY291bnQvc2VjcmV0Lm5hbWUiOiJkZWZhdWx0LXRva2VuLWJod256Iiwia3ViZXJuZ
XRlcy5pby9zZXJ2aWNlYWNjb3VudC9zZXJ2aWNlLWFjY291bnQubmFtZSI6ImRlZmF1bHQiLCJrdWJlc
m5ldGVzLmlvL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UtYWNjb3VudC51aWQiOiIyYTBhNzAxMi00MzE4L
TRmNzMtYjE0Ny1iMWQ1MjYzOWEwN2UiLCJzdWIiOiJzeXN0ZW06c2VydmljZWFjY291bnQ6a3ViZS1ze
XN0ZW06ZGVmYXVsdCJ9.dPzmWkPgAxW977niAEaBH9RrhV22t2-FtKBskIGXpNDqOpxDiKh_vq3vwX8W
DcfQ703yzBnbN0EC5iyRNBOqwahl6xROdWw5iVyekh6WYdQsx9hNt8Q9G5-YcC-PcCDeWHlbIJ1LI0nQ
a__RDxKdAlcO2_SyFw1NgwNlDc7lbwK405e6aMOg6NdCZixgzAv41LERZmITFw5HS1tEeuOyfgnJDezH
XfnLdXgxsQjQRoQoqMFtYnXqZrOpN3tFP9ukETHJa3YlSXel39Z7cl1Z7SrKVnGVlsBNC8Gza_26FlIM
PbG0RJ54bN3xS0T1Sf1KmHhmOhe19QuXFEYt9yF8xQ
```

Fig. B.14. *Token* de acceso al *dashboard* de Kubernetes desde MicroK8s

[9]Son válidos los puertos entre el 30 000 y el 32 767

```
spec:
  clusterIP: 10.152.183.73
  externalTrafficPolicy: Cluster
  ports:
  - port: 443
    protocol: TCP
    targetPort: 8443

  selector:
    k8s-app: kubernetes-dashboard
  sessionAffinity: None
  type: ClusterIP
```

```
spec:
  clusterIP: 10.152.183.73
  externalTrafficPolicy: Cluster
  ports:
  - port: 443
    protocol: TCP
    targetPort: 8443
    nodePort: 30443
  selector:
    k8s-app: kubernetes-dashboard
  sessionAffinity: None
  type: NodePort
```

Código B.1. Cambios en la instancia `service/kubernetes-dashboard`

Ahora, al acceder a la dirección de la máquina que ejecuta MicroK8s, poniendo como puerto el `NodePort` especificado (30 443, en este ejemplo), el navegador lanzará una alerta de que el sitio web no es seguro. No obstante, no existe ninguna amenaza, puesto que se trata de un entorno local y la página web a la que se accede está controlada mediante Kubernetes.

Una vez rechazada la alerta del navegador, aparecerá una ventana para introducir el *token* anteriormente generado (ver Fig. B.14). Al ingresar el *token*, se podrá visualizar el *dashboard* de Kubernetes.

Por último, cabe mencionar que si se quiere realizar alguna modificación en los microservicios y desplegarlos en producción, será necesario construir nuevas imágenes de Docker, cambiando el `Dockerfile` y utilizando los comandos `docker build` o `docker buildx`, subirlas a Docker Hub y modificar la clave `image` de aquellos `Deployments` que se quieran modificar en el *cluster* de producción. Estos comandos y su funcionamiento se explican más detalladamente en la Sección 2.1 y en la Sección 2.2.

B.4. Manual para desarrolladores

En este punto, se asume que el desarrollador tiene instaladas todas las tecnologías que quiere utilizar, ayudándose del anexo anterior si fuera necesario.

Para que funcione al completo el entorno local de desarrollo, es necesario tener instancias de MySQL, MongoDB y RabbitMQ en ejecución. Estas tres instancias se lanzan en Kubernetes mediante `StatefulSets` y `Services` que exponen puertos de tipo `NodePort` para que puedan ser accedidos desde el exterior (es decir, desde el entorno local). En el directorio `manifests-k8s` hay tres subdirectorios: `config`, `dev` y `prod` (este último no es necesario para desarrollar en entorno local).

Una vez iniciados los servicios de Minikube o MicroK8s y teniendo disponible el comando `kubectl`, será necesario aplicar los manifiestos del directorio `config` si no se aplicaron anteriormente (ver Sección B.1).

Una vez aplicados estos manifiestos, se pueden aplicar los del directorio `dev`:

```
cd manifests-k8s/

kubectl apply -f dev
```

```
ubuntu@ubuntu:~/IoT_Microservices$ cd manifests-k8s/
ubuntu@ubuntu:~/IoT_Microservices/manifests-k8s$ kubectl apply -f dev
service/mongo-dev created
statefulset.apps/mongo-dev created
service/mysql-dev created
statefulset.apps/mysql-dev created
service/rabbitmq-dev created
statefulset.apps/rabbitmq-dev created
```

Fig. B.15. Despliegue del *cluster* de Kubernetes para desarrollo

Para verificar que los manifiestos se han aplicado correctamente, se puede ejecutar `kubectl get all` (ver Fig. B.16).

Otros comandos útiles para ver que todo está correcto son `kubectl get` y `kubectl describe`, especificando el tipo de objeto de Kubernetes. Por ejemplo, se podría escribir `kubectl get pods` o `kubectl describe configmaps`.

Para acceder a los `Pods` del *cluster*, se ha de utilizar el comando `kubectl exec -it pod/<identificador-pod> -- <comando>`. Por ejemplo, para conectarse a MongoDB (o MySQL) o para entrar en el sistema de archivos de un `Pod` (mediante `sh` o `bash`), se pueden ejecutar los siguientes comandos:

```
ubuntu@ubuntu:~/IoT_Microservices/manifests-k8s$ kubectl get all
NAME                  READY   STATUS    RESTARTS   AGE
pod/mongo-dev-0       1/1     Running   0          2m6s
pod/mysql-dev-0       1/1     Running   0          2m6s
pod/rabbitmq-dev-0    1/1     Running   0          2m5s

NAME                    TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)
    AGE
service/kubernetes      ClusterIP   10.152.183.1     <none>        443/TCP
    24d
service/mongo-dev       NodePort    10.152.183.30    <none>        27017:32000/TC
P   2m6s
service/mysql-dev       NodePort    10.152.183.209   <none>        3306:31000/TCP
    2m6s
service/rabbitmq-dev    NodePort    10.152.183.57    <none>        5672:31300/TCP
    2m6s

NAME                             READY   AGE
statefulset.apps/mongo-dev       1/1     2m6s
statefulset.apps/mysql-dev       1/1     2m6s
statefulset.apps/rabbitmq-dev    1/1     2m6s
```

Fig. B.16. Comprobación de las instancias desplegadas en Kubernetes

`kubectl exec -it pod/mongo-dev-0 -- mongo -u root -p` [10]

`kubectl exec -it pod/rabbitmq-dev-0 -- sh`

En referencia a MySQL, existe un directorio denominado `mysql-iot` en el que se incluye un archivo SQL para escribirlo en un `Dockerfile` y así construir una imagen personalizada de MySQL con una base de datos inicializada con las tablas utilizadas en la aplicación.

Como se están utilizando microservicios, cada subdirectorio de la Fig. B.2 (excepto unos pocos) es una parte del proyecto que se despliega de manera independiente.

Los microservicios desarrollados con Node.js son: `measure-ms`, `orchestrator-ms`, `microcontrollers-ms` y `publisher-ms` (aunque este último está diseñado para funcionar exclusivamente con Kubernetes).

En estos microservicios se deben descargar las dependencias de cada uno (escritas

[10] Tras escribir este comando, la consola solicitará la contraseña del usuario "root"

en los archivos `package.json` y `package-lock.json`) mediante el comando `npm install`. Una vez descargadas, se puede arrancar cada microservicio escribiendo `npm start` o bien `npm run dev`[11]. Los resultados de estos comandos en uno de los microservicios de Node.js mencionados pueden verse en la Fig. B.17.

```
MacBook-Pro-de-Rocky:~/IoT_Microservices$ cd orchestrator-ms/
MacBook-Pro-de-Rocky:~/IoT_Microservices/orchestrator-ms$ npm install

> nodemon@2.0.4 postinstall /Users/rocky/IoT_Microservices/orchestrator-ms/node_
modules/nodemon
> node bin/postinstall || exit 0

Love nodemon? You can now support the project via the open collective:
 > https://opencollective.com/nodemon/donate

added 663 packages from 411 contributors and audited 664 packages in 7.897s

19 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities

MacBook-Pro-de-Rocky:~/IoT_Microservices/orchestrator-ms$ npm start

> orchestrator-ms@1.0.0 start /Users/rocky/IoT_Microservices/orchestrator-ms
> node src/index.js

orchestrator-ms at http://localhost:3000
^C
MacBook-Pro-de-Rocky:~/IoT_Microservices/orchestrator-ms$ npm run dev

> orchestrator-ms@1.0.0 dev /Users/rocky/IoT_Microservices/orchestrator-ms
> nodemon src/index.js

[nodemon] 2.0.4
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node src/index.js`
orchestrator-ms at http://localhost:3000
```

Fig. B.17. Desarrollo de microservicios con Node.js

Si se quieren ejecutar los *tests*, se puede utilizar el comando `npm test` desde el

[11]Con este comando se ejecutará la librería *nodemon*, el cual es un programa que está atento a los archivos para reiniciar el servidor al guardar los cambios

mismo directorio en el que se encuentra el archivo `package.json`, que llamará a la librería *jest* para realizar todas las pruebas especificadas en el subdirectorio `test` de cada microservicio.

El microservicio Stats está programado con Python. Para comenzar, es necesario instalar algunas librerías utilizando `pip`, y después se podrán ejecutar los *scripts*, escribiendo las siguientes instrucciones:

```
cd stats-ms/

pip install -r requirements.txt

python -m src
```

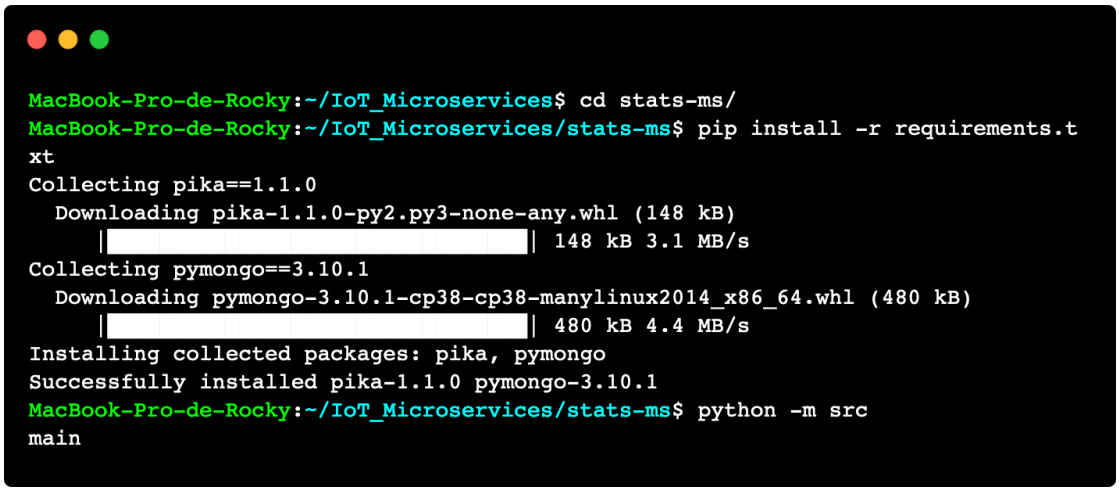


Fig. B.18. Desarrollo del microservicio Stats con Python

Para ejecutar los *tests* unitarios, se puede utilizar el siguiente comando:

```
python -m unittest discover
```

Para que este microservicio realice su función de suscriptor, es necesario que el servicio de RabbitMQ esté levantado y que exista un publicador. Este es el microservicio Publisher, el cual está desarrollado en Node.js y está pensado para utilizarse directamente en Kubernetes mediante un `CronJob`.

Para utilizar este microservicio Publisher en modo de desarrollo local, será necesa-

rio ejecutar específicamente `npm run dev`, el cual utiliza un archivo `index_dev.js`, ligeramente distinto del archivo `index.js`, que es el utilizado en producción por medio de `npm start`.

Para el microservicio Auth (desarrollado en Go), es necesario descargar algunos módulos (especificados en los archivos `go.mod` y `go.sum`) y después ejecutar el código, con las siguientes instrucciones:

```
cd auth-ms/src/

go mod download

go run main.go
```

Fig. B.19. Desarrollo del microservicio Auth con Go

Para desarrollar la aplicación de Angular es conveniente disponer de Angular CLI, aunque no es estrictamente necesario. Basta con utilizar los siguientes comandos (como en ejemplos anteriores):

```
cd angular-ms/iot-app/

npm install

npm start
```

Estos comandos (iguales que en cualquier proyecto de Node.js) instalarán todas las dependencias de Angular, TypeScript y demás librerías, y arrancarán un servidor de desarrollo con Webpack para observar los cambios en los archivos de la aplicación, para además actualizar automáticamente la pestaña del navegador con la aplicación abierta.

Por último, para programar la placa de Arduino, se puede utilizar Arduino IDE y

abrir el proyecto situado en `arduino-iot/main/main.ino`. Cabe mencionar que existe un archivo `main-template.h` en el que han de ponerse los datos de la red WiFi a la que se conectará el Arduino y la dirección IP que se quiere utilizar.

Si no se dispone de una placa de Arduino, se puede utilizar un servidor web de Node.js, situado en el directorio `fake-arduino-iot`. Este se arranca con los comandos `npm install` y `npm start`, como los microservicios de Node.js.

Una vez terminada la tarea de desarrollo, se puede detener la ejecución de cada microservicio mediante ^C (Control + C). Para parar los `Pods` de Kubernetes, se puede utilizar `kubectl delete -f dev` (no es necesario eliminar los manifiestos de configuración, de hecho, si se borran los `PersistentVolumeClaims`, se perderán los datos almacenados)[12].

```
ubuntu@ubuntu:~/IoT_Microservices$ cd manifests-k8s/
ubuntu@ubuntu:~/IoT_Microservices/manifests-k8s$ kubectl delete -f dev
service "mongo-dev" deleted
statefulset.apps "mongo-dev" deleted
service "mysql-dev" deleted
statefulset.apps "mysql-dev" deleted
service "rabbitmq-dev" deleted
statefulset.apps "rabbitmq-dev" deleted
```

Fig. B.20. Eliminación de instancias del *cluster* de desarrollo

Como recomendación, resulta de gran utilidad emplear editores de código con consolas integradas. Para este proyecto se ha utilizado Visual Studio Code, el cual permite la utilización de múltiples consolas al mismo tiempo, para poder visualizar los datos de cada microservicio.

Por otro lado, un cliente REST como Postman puede ser muy útil para realizar peticiones HTTP a cada microservicio y mejorar la productividad y la depuración de errores en la comunicación entre microservicios.

[12]Cuando se dice "eliminar los manifiestos" se hace referencia a quitarlos de Kubernetes. No confundir con borrar los archivos YAML

Anexo C. Objetivos de Desarrollo Sostenible

En este capítulo se muestra la relación que tiene el presente Trabajo Fin de Grado con los Objetivos de Desarrollo Sostenible (ODS) aprobados por la ONU en 2015, en la denominada Agenda 2030 sobre el Desarrollo Sostenible [42].

Cada uno de estos 17 objetivos (ver Fig. C.1) consiste en un conjunto de metas específicas que deben alcanzarse en los próximos 10 años (hasta el año 2030). La adopción de estos objetivos por parte de los líderes mundiales tiene el fin de erradicar la pobreza, proteger el planeta y asegurar la prosperidad.

Tal y como se dice en [43], "la sostenibilidad es el desarrollo que satisface las necesidades del presente sin comprometer la capacidad de las futuras generaciones, garantizando el equilibrio entre el crecimiento económico, el cuidado del medio ambiente y el bienestar social".

Los 17 Objetivos de Desarrollo Sostenible persiguen este concepto de sostenibilidad. Estos ODS se pueden clasificar según la dimensión con la que se relacionan. Así, existe una dimensión económica, una dimensión social y una dimensión medioambiental (referida a veces con el término biosfera). En la Fig. C.2 se muestran estas tres dimensiones y los ODS que intervienen en cada una.

Puede parecer que este concepto de sostenibilidad no tenga relación con el desarrollo de *software* y las nuevas tecnologías. Sin embargo, sí que existe una pequeña relación. Por este motivo, el proyecto realizado en este Trabajo Fin de Grado persigue algunos de los 17 Objetivos de Desarrollo Sostenible. En la Tabla C.1 se muestra el ODS de cada dimensión que más relación tiene con el proyecto desarrollado, su nivel de importancia, y las metas específicas que más se ajustan.

Fig. C.1. Objetivos de Desarrollo Sostenible

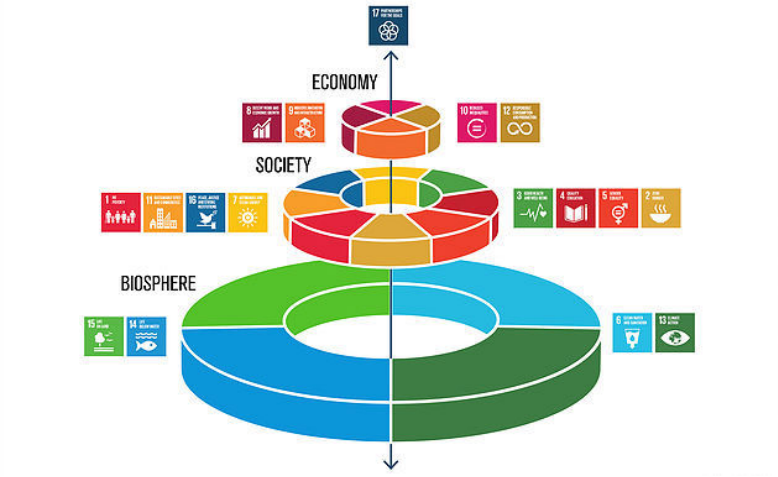


Fig. C.2. Dimensiones de los ODS

Se ha considerado que el ODS9 es el que más relación tiene con el proyecto desarrollado. Esto se debe a que las tecnologías que se utilizan en la aplicación son innovadoras y contribuyen a realizar un mejor uso del *software* y de las infra-

estructuras. El auge de *Cloud Computing* permite que las empresas contraten servicios más eficaces que requieren un menor coste por su parte. La adopción de las arquitecturas de microservicios y de la cultura DevOps promueve una serie de buenas prácticas que beneficia a las empresas a mejorar sus procesos de producción de *software* y a automatizar las tareas.

Por otro lado, el ODS11 tiene algo de relación con la domótica, IoT y *Machine Learning*. Estas tecnologías pueden servir para desarrollar viviendas sostenibles. Al ser muy sutil la relación de este ODS con el proyecto desarrollado, se le ha dado una importancia secundaria.

Tabla C.1. Objetivos de desarrollo sostenible relacionados con el proyecto

Dimensión	ODS	Rol	Meta
Biosfera[13]	-	-	-
Sociedad	**ODS11.** Ciudades y comunidades sostenibles	Secundario	**11.3.** De aquí a 2030, aumentar la urbanización inclusiva y sostenible y la capacidad para la planificación y la gestión participativas, integradas y sostenibles de los asentamientos humanos en todos los países
Economía	**ODS9.** Industria, innovación e infraestructura	Primario	**9.4.** De aquí a 2030, modernizar la infraestructura y reconvertir las industrias para que sean sostenibles, utilizando los recursos con mayor eficacia y promoviendo la adopción de tecnologías y procesos industriales limpios y ambientalmente racionales, y logrando que todos los países tomen medidas de acuerdo con sus capacidades respectivas

[13]No ha sido posible encontrar un Objetivo de Desarrollo Sostenible relacionado con el medio ambiente que se ajuste al presente proyecto

Por último, es interesante mencionar algunos datos numéricos sobre tecnologías utilizadas en este proyecto relacionadas con la sostenibilidad.

La domótica permite que las viviendas sean sostenibles, gracias a la gestión de la energía y la seguridad que llevan a cabo los dispositivos inteligentes conectados a Internet mediante IoT.

Las bombillas inteligentes suelen estar fabricadas con tecnología LED. Esta forma de iluminación permite alcanzar ahorros de hasta el 70 % frente a bombillas convencionales [44].

Según el Instituto para la Diversificación y el Ahorro de la Energía (IDAE), el consumo de energía en España por parte de las familias supone un 20 % del consumo total del país, en los últimos años. Esta cifra tiene una tendencia decreciente: el consumo de energía se está reduciendo poco a poco [45]. El motivo de esta reducción puede deberse, en parte, al uso de domótica y dispositivos inteligentes que permiten realizar un uso responsable de la energía mediante técnicas de regulación automática, por ejemplo.

Por otro lado, el uso de *Cloud Computing* también contribuye a la sostenibilidad. La utilización de servicios *cloud* en lugar de implementar servicios y centros de procesamiento de datos (CPD) propios hace que se reduzcan los recursos energéticos por parte de las empresas, lo que conlleva una reducción de las emisiones de CO_2 a la atmósfera. Esta reducción de emisiones de gases de efecto invernadero supone entre un 30 % y un 90 % menos, en comparación con la implementación de un CPD propio [46].

En definitiva, aunque *a priori* los proyectos de *software* y tecnología no parezcan tener relación con los Objetivos de Desarrollo Sostenible, al analizar en detalle los beneficios que producen se ve más claro que sí están relacionados. Con esto, se consigue aportar valor a la sostenibilidad perseguida por la ONU.