Whaya

Una aproximación a la arquitectura de microservicios y datos espaciales

Manel Mena[a,1,*], Antonio Corral[a,1,**], Luis Iribarne[a,1,**]

[a] *Universidad de Almería, Ctra. Sacramento, s/n, 04120 La Cañada, Almería*

Abstract

En esta memoria vamos a reflejar con detalle el proceso de consecución de la aplicación Whaya (**WH**ere **A**re **YA**?), una aplicación híbrida"Mobile First"que permite la comunicación entre personas contando con el geoposicionamiento de sus usuarios como principal factor diferenciador. Para ello, aplicaremos técnicas de desarrollo basado en microservicios, junto con metodologías apropiadas para gestión y creación de los mismos. Haremos todo el despliegue apoyándonos en una infraestructura back-end basada en *Dockers* (Infrastructure as Code) y un *front-end* realizado mediante tecnologías híbridas, más concretamente utilizando el framework Ionic.

Keywords: Datos espaciales, Datos en tiempo real, Arquitectura de microservicios, Persistencia políglota, Patrones de sistemas, IaC, Programación móvil, Programación híbrida, Ionic, Docker, Netflix OSS, Spring.

[*] Autor de trabajo de fin de máster

[**] Director/Co-Director de trabajo de fin de máster

Email addresses: manel.mena@ual.es (Manel Mena), acorral@ual.es (Antonio Corral), luis.iribarne@ual.es (Luis Iribarne)

[1] ACG - Applied Computing Group de la Universidad de Almería, TIC-211

Agradecimientos

Lo primero me gustaría realizar una serie de agradecimientos:

- A los directores de este trabajo Antonio Corral y Luis Iribarne, sin los que la consecución de este proyecto no hubiese sido posible.
- Al Departamento de Informática, en especial a Manolo Torres y José Antonio Martínez, por ofrecerme recursos la la nube privada del Departamento de Informática, junto con la gran tarea que conlleva el mantenimiento de la misma.
- A mis compañeros de grupo de computación aplicada, que me aguantan cada día de manera incansable.
- A mi familia sin los cuales no podría ni si quiera estar aquí.

Índice

Índice de figuras

Listings

1. Introducción

1.1. Preámbulo

En este proyecto se pretende realizar una herramienta en forma de aplicación móvil de comunicación entre personas, en el que se incluirá como principal componente el geoposicionamiento de sus usuarios en todo momento. Todo ello, como justificación para el desarrollo de una infraestructura en el que se aplican novedosos avances actuales en áreas relacionadas con las arquitecturas de sistemas, orquestación de servicios, seguridad de servicios web, aplicaciones híbridas móviles, etc.

En el mismo, vamos a intentar desarrollar primero una capa de servicios en la que primen la seguridad, la alta disponibilidad y la posible tolerancia a fallos. Debido a ello, hemos decidido aproximarnos a la solución desarrollando una arquitectura de microservicios, los cuales serán pequeñas piezas funcionales lo mas independientes entre si, en forma de servicios RESTful, que contarán con distintas infraestructuras de datos dependiendo de los datos que manejen. Por ejemplo, el servicio que se encarga de posicionar puntos de encuentro entre usuarios contará con una base de datos *Neo4j* con extensiones espaciales para controlar la persistencia e indexado de los mismos.

A su vez, para garantizar la disponibilidad de cada uno de los microservicios, aplicaremos una serie de patrones de sistemas, ampliamente probados a explicar en el memoria, contando para ello con la ayuda de algunos paquetes de Netflix OSS. La infraestructura se desplegará mediante sistemas de contenedores (*Dockers*), lo cual nos permite de manera muy rápida realizar réplicas de la misma ejecutando simplemente un script de código, e incluso levantar réplicas individuales de cada uno de los microservicios para garantizar una mayor disponibilidad.

Para probar esa infraestructura y mostrar su potencia, vamos a realizar el desarrollo de una aplicación móvil híbrida mediante el framework de Ionic, que nos ayudará a desarrollarla para que se pueda desplegar en casi cualquier plataforma con solo una única implementación (code once deploy everywhere), estando esta diseñada con una aproximación *mobile first*, aunque no se descarte el despliegue en plataformas de corte no móvil.

1.2. Motivaciones

El proyecto nace en primer lugar como una manera de mostrar los conocimientos adquiridos a lo largo de la consecución del máster, sin olvidar la idea

de mostrar mi capacidad como ingeniero informático para resolver problemas. En él, intento mostrar mis conocimientos del mayor número posible de tecnologías, así como metodologías, para hacer una puesta en valor de mis capacidades cara al ámbito de la empresa privada.

En segundo lugar, con este proyecto busco abarcar una pequeña iniciación a la temática de una posible línea de investigación de doctorado, en este caso una iniciación a los datos espaciales, básicamente responder a qué son, cómo se guardan y algunas consultas básicas en los mismos. Para ello, en este punto de introducción, vamos a dedicar una sección a hablar de los mismos.

1.3. Objetivos

Teniendo en cuenta las motivaciones de las que nace este trabajo de fin de Master, los objetivos marcados comprenden una serie de puntos detallados a continuación:

a) Realizar un estudio del estado del arte tanto en temas de datos espaciales como en desarrollo de arquitecturas de microservicios que me permita una aproximación al problema de manera adecuada.

b) Aprendizaje de las tecnologías pertinentes a la consecución de los objetivos del proyecto. Como por ejemplo, Dockers, Ionic, Angular 4, Websockets, Spatial Neo4j, etc.

c) Enfrentarme a las problemáticas intrínsecas a las arquitecturas devenidas de sistemas de alta disponibilidad. Por ejemplo, temas relacionados con la seguridad de la infraestructura, tolerancia a fallos de la misma, posibilidad de levantar replicas de los servicios de manera rápida y visionado de la misma.

d) Desarrollo de Whaya (Where Are YA?), una aplicación móvil híbrida que cuente entre su funcionalidad como mínimo con un sistema de geoposicionamiento propio y para terceros, así como un chat y la posibilidad de generar encuentros entre usuarios amigos.

e) Elaboración de un artículo valido concluyendo así memoria del proyecto de trabajo de fin de máster.

1.4. Cronograma

Para cumplir los objetivos descritos anteriormente vamos a mostrar las tareas que nos marcamos en la consecución del proyecto, así como un diagrama de Gantt para mostrar la temporización de las mismas. Comenzando por las tareas, son las que se muestran a continuación:

a) Estudio del problema y análisis. Primero delimitamos lo que queríamos mostrar con la solución dada, así como las tareas preliminares a realizar para la consecución del mismo.

b) Estudio de arquitecturas de microservicios y diseño de la arquitectura. Estudiamos el punto en el que están las arquitecturas de microservicios, qué decisiones tomar referentes a cómo iban a ser orquestados estos microservicios, y qué tecnologías usaríamos al implementarlos. Junto con ello, tomamos decisiones referentes al diseño de nuestra arquitectura de microservicios.

c) Aprendizaje de las tecnologías implementadas en el *back-end*. Spring, Netflix OSS, Socket.io, Docker, etc.

d) Estudio de tecnologías utilizadas en *front-end* y diseño de la aplicación. En este punto tomamos la decision de qué framework utilizar en el *front-end*, cuál sería el diseño de la misma, así como la manera más adecuada de establecer la comunicación con nuestro *back-end*.

e) Aprendizaje de tecnologías utilizadas en el *front-end*. Ionic, Angular 4, etc.

f) Implementación del *back-end*.

g) Implementación del *front-end*.

h) Depuración, juegos de prueba y corrección de errores.

i) Preparación de la memoria y presentación.

Para tener una visión aproximada de lo que ha supuesto la consecución del proyecto (Figura 1) veamos un diagrama de Gantt aproximado del mismo.

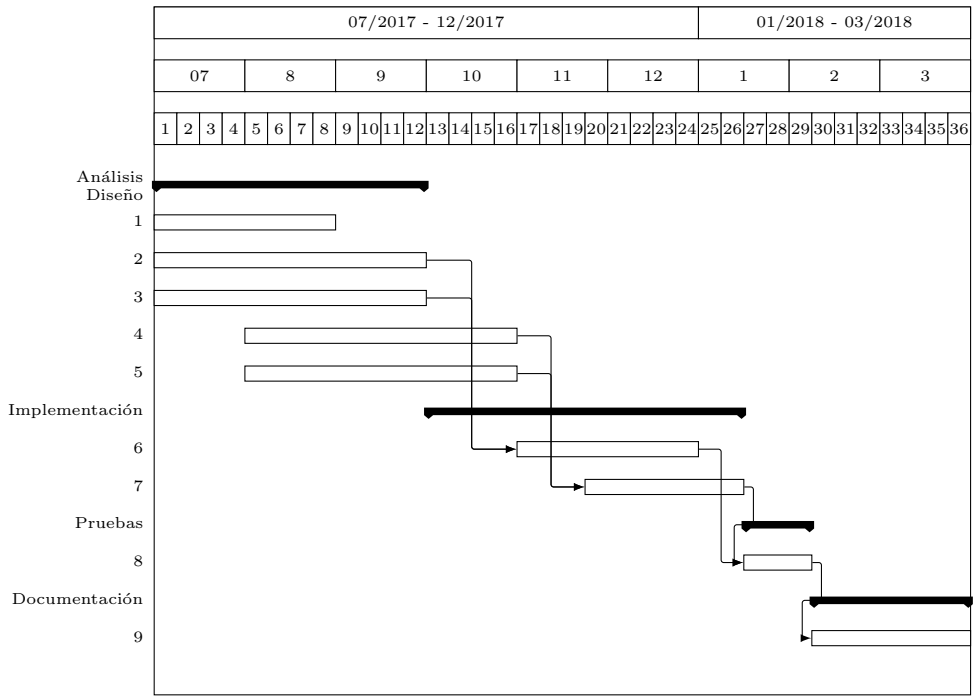


Figura 1: Cronograma del trabajo de fin de master.

1.5. Datos espaciales

Los datos espaciales son aquellos que representan la localización, el tamaño o la forma de un objeto partiendo de un sistema de referencia geográfica. Estos datos también pueden incluir atributos sobre el objeto real que pretende ser representado. Para acceder, visualizar o manipular este tipo de datos, se suelen utilizar frameworks o herramientas de soporte de Sistemas de Información Geográfica (GIS).

En este proyecto vamos a centrarnos en dos de los frameworks de sistemas *GIS* más utilizados. Por un lado **PostGIS**, que es una extensión del sistema de base de datos *PostgreSQL* que la convierte en una base de datos espacial, cuenta con las características descritas en la especificación *OpenGIS* del Open Geospatial Consortium [5]. Este sistema es utilizado por infinidad de organizaciones a nivel mundial, su primera versión estable 1.0 data del año 2005. Las características principales de la misma son:

a) Incluye tipos geométricos para `Points`, `LineStrings`, `Polygons`, `MultiPoints`, `MultiLineStrings`, `MultiPolygons` y `Geometry Collections`.

b) Predicados espaciales que ayudan a determinar la interacción de geometrías.

c) Operadores espaciales para determinar medidas de este tipo de datos, como pueden ser medidas de área, distancia, etc. A su vez también operaciones entre los datos como pueden ser unión, diferencia, etc.

d) De base trabajan con un **R-tree-over-GiST** (Generalized Search Tree) para las consultas que requieran de estos índices espaciales.

e) Soporte para la mezcla de índices espaciales y no espaciales mediante planes de consulta.

f) Para datos de rasterización cuentan con *PostGIS Raster*.

La ventaja de *PostGIS* es que al ser una implementación basada en *geometrías ligeras* e índices simples, está optimizada para tener un uso de memoria y disco mínimo, lo cual hace que la cantidad de datos que caben en la RAM sea sustancialmente mayor que en otros sistemas GIS lo que provoca que se mejore la velocidad de las consultas.

Por otro lado también vamos a hablar de *Neo4j Spatial* que es también una extensión en este caso de una base de datos *Neo4j*. Al contrario de *PostgreSQL*, que es una base de datos relacional, *Neo4j* es una base de datos de las comúnmente llamadas NoSQL, más concretamente una base de datos orientada a grafos, en la que se guardan varios tipos de objetos como entidades (nodos) y sus relaciones (aristas). Para situaciones en las que los datos se ajustan a este modelo, hacen que esta solución pueda ser mucho más rápidas que otras. Un ejemplo de datos que se ajustan al modelo serian los que nosotros estamos tratando, ya que por ejemplo, podemos considerar algunos nodos como los lugares, y las relaciones como la rutas de unión de esos lugares.

La mayoría de bases de datos NoSQL se diseñan teniendo en cuenta que deben de cumplirse al menos 2 de las características que son intrínsecas al teorema de CAP, que son:

a) **Consistency**. Que viene a significar que las transacciones son ACID, lo cual nos permite confirmar que todos los objetos tienen los mismos datos al mismo tiempo.

b) **Availability**. Todos los datos están disponibles y las transacciones que se inician tienen que terminar en un tiempo razonable. También se asegura que si un nodo falla los otros continúan.

c) **Partition-tolerance**. Existe tolerancia a fallos entre componentes individuales de un sistema distribuido. Lo cual viene a decirnos que el sistema continuará incluso si mensajes se pierden entre algunos de sus componentes.

En nuestro caso *Neo4j Spatial*, al igual que *PostGIS*, también cumple con las especificaciones marcadas por *OpenGIS*, por lo que comparte gran parte de las características que tiene *PostGIS*, cómo son los tipos de datos, operaciones, e incluso como se guardan los índices espaciales. La principal diferencia es que gracias al modelo que utiliza *Neo4j* existen consultas que son increíblemente rápidas, casi sin importar el tamaño de los datos, como pueden ser análisis de rutas, consultas de proximidad, etc. Básicamente consultas que siguen patrones jerárquicos.

En la tesis *NoSQL Spatial - Neo4j versus PostGIS* [1] podemos ver comparativas y pruebas de distintas operaciones en las dos bases de datos, aquí descritas. Como se puede observar en la Figura 2 el factor más importante a la hora de elegir cual de los dos sistemas utilizar, es el tipo de consulta que vamos a hacer, y la cantidad de RAM con la que contamos en el sistema, ya que la principal desventaja de *Neo4j* es que tiene una huella en memoria mucho mayor.

En nuestro caso hemos elegido *Neo4j* dado que en un principio la capacidad de memoria no iba a ser un problema, ya que contamos con una serie de recursos muy generosos aportados por el **Departamento de Informática** de la Universidad de Almería, así como los recursos aportados por el grupo de investigación **ACG** (Applied Computing Group) de la misma universidad. A su vez, debido a que la temática de la aplicación principal que va a hacer uso de la infraestructura se basa, sobre todo, en datos poco cambiantes y consultas de proximidad, decidimos utilizar *Neo4j* dado que también se ajusta muy bien a las relaciones entre otros datos necesarios para la aplicación, como son las relaciones de amistad y el control de popularidad de las quedadas.

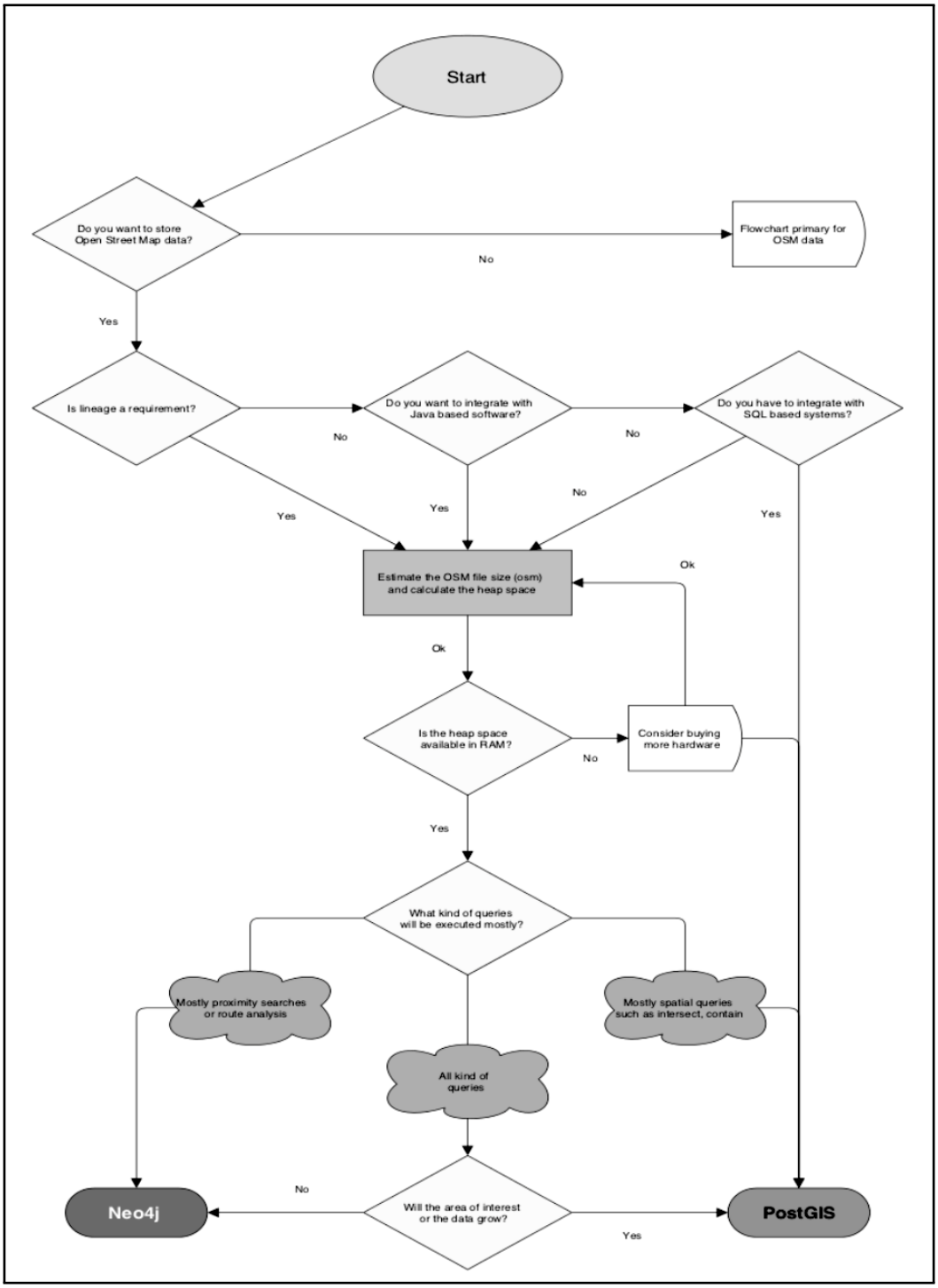


Figura 2: Neo4j Spatial vs PostGIS [1]

2. Whaya - Desarrollo de Back-End

En la siguiente sección, vamos a hablar de todo lo relacionado con el *back-end* de Whaya, incluyendo el tipo de arquitectura, patrones utilizados, tecnologías aplicadas, etc. Junto con ello, haremos mención tanto de la implementación, como de la seguridad que interviene en el funcionamiento de nuestro sistema.

2.1. Introducción a los microservicios

En primer lugar, vamos a intentar definir lo que es un microservicio. Aunque no hay una definición del todo clara, es interesante pensar en ellos como conjuntos auto contenidos que usualmente dan respuesta a pequeños problemas, normalmente mediante servicios, que se construyen alrededor de piezas claves para nuestro negocio. Estos microservicios deben ser procesos propios independientes capaces de comunicarse unos con otros con mecanismos ligeros, normalmente como recursos HTTP de una API. La mayoría de las veces se escriben en diferentes lenguajes de programación y no tienen por qué tener como base las mismas tecnologías de persistencia de datos.

Una arquitectura de microservicios se basa en la orquestación de los mismos de manera adecuada. Esto nos da la ventaja de poder levantar réplicas de manera independiente, al contrario que en arquitecturas monolíticas, en las que toda la lógica de negocio se implementa en una solución única. En la Figura 3 se pueden observar las diferencias entre las arquitecturas monolíticas y las arquitecturas de microservicios. El artículo de James Lewis y Martin Fowler [2] que la incluye originalmente, resulta muy interesante para comprender el funcionamiento de los mismos.

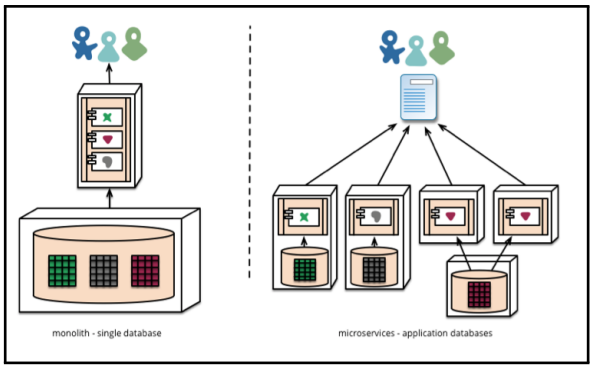


Figura 3: Arquitecturas monolíticas vs microservicios. [2]

Para nuestra implementación contamos con la ayuda de contenedores de tipo *Dockers*, y para orquestar los mismos utilizaremos la herramienta *Docker-compose* que explicaremos detalladamente a continuación.

2.2. Infraestructura como código(IaC)

Como hemos comentado anteriormente para orquestar estos microservicios utilizamos *Docker*. Esto nos facilita mucho la vida a la hora de levantar réplicas de nuestros servicios de manera rápida, con el potencial añadido de incluso también poder llevar un control muy sencillo de las versiones para los mismos, todo ello utilizando el concepto de infraestructura como código (IaC) [6]. *Docker* es un software que proporciona contenedores, estos nos permiten tener una serie de recursos virtualizados que son completamente independientes entre sí, cada uno con características propias. Por ejemplo, espacios de nombres, sistema de archivos, etc. Todo ello hace que varios contenedores puedan funcionar en una sola instancia de Linux evitando así la sobrecarga de tener que iniciar y mantener máquinas virtuales de manera independiente. Un sistema de *Dockers* se basa en crear una capa que describe aquello que hay que hacer sobre las imágenes base de los contenedores en concreto. Sin meternos muy en profundidad en todas las opciones que *Docker* nos ofrece, lo mejor para entender cómo se aprovisionan y se ejecutan contenedores mediante *Docker* es ver un ejemplo:

```
1 docker pull alpine
  docker run -d --restart=always -p 80:80 alpine:version
```

En el código anterior, vemos cómo lo primero que debemos hacer es un `pull` de una imagen del repositorio oficial, que en nuestro caso es la que se ejecuta cuando ejecutamos el comando `docker run`. En el código anterior vemos cómo lo que hacemos es ejecutar un nuevo contenedor de esa imagen, a su vez, en nuestra máquina, cualquier petición que vaya al puerto 80 se redirigirá a ese contenedor también por el puerto 80. Otra de las opciones que vemos es que si hay una interrupción de nuestro contenedor, éste se reiniciará siempre gracias a la orden `restart=always`. La instrucción anterior es un ejemplo muy simple, dado que el cliente base de *Docker* tiene muchas más órdenes que podemos aprender en la documentación oficial del mismo [7]. Incluso, es posible acceder a un nivel más interno de personalización de las máquinas que levantemos, siempre y cuando definamos el mismo de otra manera, en este caso con un `dockerfile`. cómo ejemplo de `dockerfile` (Listing 1) veamos cómo a partir de la imagen de *Neo4j*, utilizamos este

archivo para personalizarla, instalando un complemento de datos espaciales en *Neo4j* y las extensiones APOC.

```
FROM neo4j:3.1.4
MAINTAINER Manel Mena Vicente manel.mena@ual.es

ENV APOC_RELEASE 3.1.3.7
ENV APOC_RELEASES_URL https://github.com/neo4j-contrib/neo4j-
    apoc-procedures/releases/download
ENV APOC_PLUGIN_JAR apoc-${APOC_RELEASE}-all.jar
ENV APOC_PLUGIN_URL ${APOC_RELEASES_URL}/${APOC_RELEASE}/${
    APOC_PLUGIN_JAR}

RUN apk update \
    && apk add ca-certificates wget \
    && update-ca-certificates \
    && apk add openssl curl

RUN apk add --update zip unzip && rm -rf /var/cache/apk/*

RUN curl -s -L -o /var/lib/neo4j/plugins/neo4j-spatial-server-
    plugin.jar https://github.com/neo4j-contrib/spatial/files
    /1227950/neo4j-spatial-0.24-neo4j-3.1.4-server-plugin.zip

RUN curl --output /var/lib/neo4j/plugins/${APOC_PLUGIN_JAR} --
    location "${APOC_PLUGIN_URL}?raw=true"
```

Listing 1: Ejemplo `dockerfile`.

En el archivo `dockerfile` podemos apreciar qué a partir de la imagen de `neo4j:3.1.4`, primero generamos una serie de variables de entorno, que después utilizamos para ejecutar órdenes básicas de Linux.

Junto con *Docker*, es interesante utilizar una serie de herramientas que permiten aprovisionar contenedores de manera más sencilla, y así aprovechar todos los recursos de hardware de los que disponemos. Más concretamente *Docker-compose* y *Docker-swarm*. Debido a los recursos disponibles a la hora de ejecutar nuestro proyecto (solo un equipo físico con recursos limitados), utilizar *Docker-swarm* es bastante desmesurado, aun así es interesante conocer para que se utiliza el mismo. *Docker-swarm* u otras herramientas similares como *Kubernetes*, es interesante usarlas cuando contamos con bastantes re-

cursos hardware, esto nos permite hacer un clúster de recursos donde todos ellos se agrupan para formar un único *Docker engine*. A partir de la versión 1.12 de *Docker*, el modo swarm ya está integrado en la herramienta base de *Docker*.

El siguiente punto, y donde vamos a hacer un poco más de hincapié, es en ver y entender la herramienta *Docker-compose*, la cual nos permite realizar la definición de la configuración de nuestra infraestructura en un archivo YAML (**Y**et **A**nother **M**arkup **L**anguage) el cual configura los servicios de la aplicación y realiza todos procesos de creación y de arranque de los contenedores con un solo comando.

Primero veamos cómo es la organización de carpetas o scaffolding (Figura 4) de nuestra infraestructura.

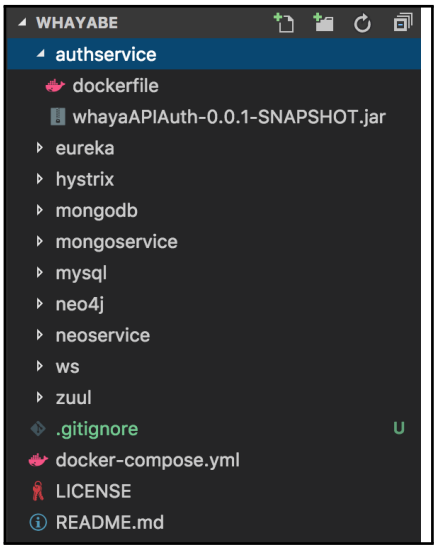


Figura 4: Scaffolding de WhayaBe.

Vemos que en la raíz contamos con el archivo `docker-compose.yml` en el que definimos toda la infraestructura. Aparte, los servicios que requieran configuración propia o lo que es lo mismo que no nazcan directamente de las imágenes base de *Docker* sin configurar, también cuentan con un `dockerfile` en el que se ejecutarán las órdenes de iniciación del contenedor en concreto.

Lo mejor para entender cómo funciona, es ver un pequeño fragmento del archivo *Docker-compose* que orquesta nuestra infraestructura (Listing 2).

```
version: '3'
services:
  neo4j:
    container_name: neo4j
    build: neo4j
    restart: always
    ports:
     - 7474:7474
     - 7687:7687
     - 7473:7473
    environment:
     - NEO4J_AUTH=neo4j/secreto
     - NEO4J_dbms_memory_heap_maxSize=2048M
     - NEO4J_dbms_memory_pagecache_size=2048M
    volumes:
     - ./neo4j/data:/data # provision the volumes
  mongo:
    container_name: mongodb
    image: bitnami/mongodb:latest
    restart: always
    ports:
     - 27017:27017
    environment:
     - MONGODB_USERNAME=mongodb
     - MONGODB_PASSWORD=secreto
     - MONGODB_DATABASE=Whaya
    volumes:
     - ./mongodb/data:/bitnami
  neoservice:
    container_name: neoservice
    build: neoservice
    restart: always
    depends_on:
      - neo4j
      - zuul
      - eureka
      - rabbitmq
      - authservice
```

Listing 2: Ejemplo `docker-compose.yml`.

Los tres servicios elegidos para mostrar cómo funciona *docker-compose* son muy ilustrativos, dado que utilizan una serie de distintos tipos de configuración. En primer lugar, definimos la versión del cliente de *docker-compose* que va a utilizar nuestro archivo de configuración. Para ello, utilizamos la palabra clave `version` a partir de ahí definimos cada uno de los servicios los cuales detallamos a continuación.

a) **Neo4j**: Primero asignamos un nombre con `container_name`, la palabra clave `build` indica que para levantar este contenedor se debe utilizar el archivo `dockerfile` de la carpeta *Neo4j*, `restart:always` nos permite indicar que queremos que si por alguna razón en el contenedor se produce un error insalvable o se cae por algún motivo este va a intentar siempre reiniciarse. En `ports` declaramos los puertos de escucha del contenedor; a posteriori con `environment` declaramos las variables de entorno que corresponden a ese contenedor y en `volumes` indicamos que se genere una copia 1 a 1 de una carpeta física de nuestra máquina a una carpeta virtual del contenedor, por lo que si se modifica algo en una carpeta de volumen, bien por la máquina virtual o por la física, esta se vea reflejada en la otra y viceversa. Gracias a ello podemos realizar una copia de la carpeta en otro dispositivo, y al levantar la infraestructura en ese otro dispositivo tengamos ya los datos que generó el contenedor anterior.

b) **MongoDb**: Este servicio se diferencia del anterior tan solo en que no parte de la palabra clave `build`, si no que nace desde la imagen original de *MongoDb* proporcionada por la compañía *Bitnami* a partir del comando `image`. En ella no realizamos ninguna personalización mas allá de modificar los valores por defecto de algunas variables de entorno de la misma.

c) **Neoservice**: Ésta vuelve a ser como la primera imagen, pero vemos como se aprecia un nuevo comando, mas concretamente `depends_on` el cual nos proporciona la posibilidad de iniciar el contenedor en concreto después de lo contenedores que están bajo `depends_on`.

Es interesante fijarnos en que *Docker* nos permite levantar todo tipo de servicios virtuales, no solo servidores web, si no también bases de datos, o casi cualquier otra cosa que se nos ocurra. De hecho, el mundo en general de la virtualización está virando hacia el ecosistema *Docker* de manera fulgurante.

Compañías como Microsoft con su nube Azure, Google con ComputeEngine y CloudSoftware con OpenStack, ya ofrecen la posibilidad de utilizar archivos de configuración *Docker* para levantar de manera rápida servicios en sus Clouds.

A continuación vamos a nombrar cada uno de los contenedores gestionados con *Docker* en nuestra arquitectura, así como una breve descripción de la función cada uno de ellos:

- **Neo4j**: Base de datos que guarda los usuarios junto con relaciones entre los mismos y los encuentros organizados.
- **Neo4jService**: Servicio que gestiona los datos relacionados con los usuarios y encuentros organizados por los mismos.
- **MongoDb**: `Data Lake` en el que se guardan de manera anónima toda la información de las posiciones de los usuarios mes a mes.
- **MongoService**: Servicio que gestiona el histórico de datos de geoposicionamiento de nuestro sistema.
- **MySql**: Base de datos donde guardamos todo lo referente a la autorización y autenticación de los usuarios en nuestro sistema respetando la definición de seguridad OAuth 2.0.
- **AuthService**: Servicio que recibe las peticiones de nuestro sistema OAuth 2.0.
- **Ws**: Servicio de Socket.io que se encarga de manejar todos los datos en tiempo real de nuestra aplicación.
- **Eureka**: Componente de Netflix OSS que actúa de registro de servicios.
- **Zuul**: Componente de Netflix OSS que proporciona una puerta de enlace para acceder al resto de servicios.
- **Hystrix**: Componente de Netflix OSS que controla el estado de las peticiones de nuestro sistema que nos ayuda a implementar el patrón *Circuit Breaker* en el mismo. Este patrón lo explicaremos posteriormente.

- **Rabbitmq**: Sistema de gestión de cola de mensajes que permite la comunicación entre los componentes de nuestra arquitectura. Dando soporte a *Hystrix*, componente esencial para la detección de errores en la comunicación en arquitecturas compatibles con Netflix OSS.

La mayoría de estos servicios serán explicados con más detalle posteriormente. Esto tan solo es una primera aproximación para que tengamos una idea de los contenedores que gestionamos mediante la herramienta *Docker*.

2.3. Arquitectura de microservicios

Una buena práctica es comenzar partiendo de un pequeño esquema sobre cómo vamos a partir nuestros microservicios. Esto lo hacemos para delimitar el tipo de servicios que utilizamos en nuestra arquitectura. Para ello, nuestro subsistema de *back-end* contará en un primer momento con 2 niveles básicos o capas que se detallan a continuación:

- **DB Services**. Estos servicios son básicamente aquellos que levantan las bases de datos que controlan la persistencia de los datos manejados en nuestra lógica de negocio.

- **API Services**. Estos servicios son los que controlan por un lado nuestra lógica de negocio y por otro exponen la funcionalidad de manera externa, permitiendo a aplicaciones de terceros hacer uso de los mismos (siempre y cuando sean *trusted apps*).

Dentro de estos dos primeros niveles básicos, contamos con los que son replicables a nivel de infraestructura, véase los servicios incluidos en la capa de API Services, y los que no, aquellos que pertenecen a la capa de DB Services. En esta última tenemos la persistencia de datos, y replicarlos a nivel de infraestructura podría dar lugar a inconsistencia en los mismos. Por tanto, si se desea hace una réplica de estos servicios se debe hacer a nivel de base de datos. Esto hace que nuestra arquitectura básica objetivo sea la representada en la Figura 5.

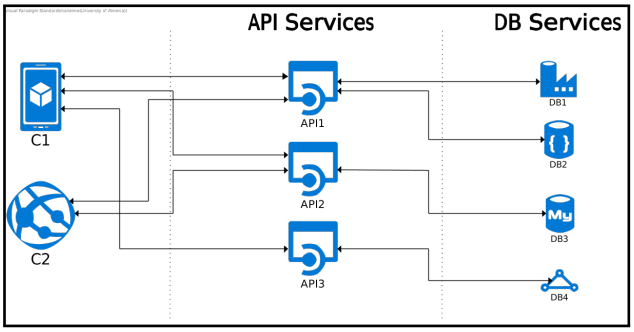


Figura 5: Arquitetura básica objetivo Whaya.

A parte de lo que seria la arquitectura básica de nuestros microservicios, contamos con un nivel más que denominamos como "capa de orquestación de microservicios", la cual será explicada mas adelante.

El hecho de desplegar la arquitectura mediante microservicios conlleva una serie de problemas que es conveniente plantearse incluso antes de desarrollar nuestro sistema, como por ejemplo:

- **Número de microservicios que desplegamos**. Cada microservicio tiene su propia configuración y ocupa sus puertos, lo cual aumenta la complejidad de nuestro sistema en general y esto se puede convertir en un problema de envergadura a la hora de mantener el sistema si lo hacemos de manera manual, por lo que tenemos que intentar alcanzar un compromiso entre la separación de microservicios y la granularidad que le vamos a dar a los mismos.

- **Seguimiento de estado de los microservicios**. Otra problemática es que cuando tenemos un sistema con tantas pieza pequeñas es difícil realizar un seguimiento de cada una de ellas y comprobar que todas están listas para ser utilizadas y no hay ningún problema. Este problema se da en menor medida cuando contamos con una arquitectura monolítica.

- **Mantener el seguimiento de la información y evitar fallos en cadena**. Dado que intrínsecamente los microservicios pueden y deben comunicarse entre sí, es interesante intentar llevar un seguimiento del flujo y estado de la información. Dado que si no tenemos cuidado con este punto se puede dar el caso de que, o bien el tiempo de respuesta de nuestros microservicios se vea muy mermado si la información pasa

por muchos microservicios, o bien se produzca un fallo en uno de ellos y genere una reacción en cadena.

- **Asegurarnos que solo ciertos servicios estan expuestos al exterior**. Es necesario que solo ciertos servicios sean expuestos al exterior. En nuestro caso los únicos visibles serán los servicios de tipo API Services, aunque como explicaremos posteriormente, solo tendremos un servicio expuesto a puerto publico (*Zuul*, un servicio de la capa de orquestación).

- **Cómo proveer de seguridad nuestros servicios API**. Al contar con más de un microservicio en nuestro sistema, tenemos que plantearnos cómo anteponer la seguridad desde otra perpectiva distinta a como lo haríamos en un servicio monolítico. En nuestro caso contamos con dos alternativas que serán comentadas a la hora de explicar la orquestación de la arquitectura.

Para dar solución a estos problemas contamos con una nueva capa en nuestra arquitectura, la que hemos denominado la **capa de orquestación**.

2.4. Orquestación de microservicios/patrones de sistemas

En esta capa vamos a levantar una serie de microservicios que nos ayuden a gestionar, monitorizar y exponer de forma segura el resto de microservicios así como la comunicación entre los mismos. Para ello contamos con la ayuda de un framework de componentes altamente probados **Netflix OSS**. Nuestra solución de orquestación se basa en los siguientes componentes:

a) **Servicio de descubrimiento**. *Eureka* es un microservicio que levantamos para ayudarnos a llevar un seguimiento de los microservicios. Cuando levantamos los mismos, estos hacen una llamada a ese servicio de descrubrimiento para decirle quienes son y en qué estado están. A su vez, un microservicio podrá preguntar por otro siempre y cuando ambos estén registrados en *Eureka*, gracias a una API de descubrimiento de servicios que se incluye en el mismo.

b) **Circuit Breaker**. *Hystrix* es otro microservicio de nuestra capa de orquestación que nos ayuda a implementar el patrón *Circuit Breaker*.

Este patrón se utiliza para evitar posibles errores en cadena, para entenderlo de mejor manera es recomendable leer la entrada al blog de Martin Fowler, *Circuit Breaker* [3]. Pero por tener una idea básica, *Hystrix* se encarga de ver todas las peticiones y sus respuestas en nuestra arquitectura. Si alguna de estas peticiones muestra de manera continua errores, *Hystrix* se encargará de lanzar una respuesta predefinida en puesto de sobrecargar nuestro servicio con peticiones que van a ser erroneas, basicamente se abre el circuito. No obstante, cada cierto periodo de tiempo *Hystrix* se va a encargar de comprobar ese método del servicio en concreto que tiene abierto el circuito, a ver si vuelve a funcionar correctamente, momento en el que cerrará el circuito. En la Figura 6 se ilustra esto mediante un diagrama de secuencias.

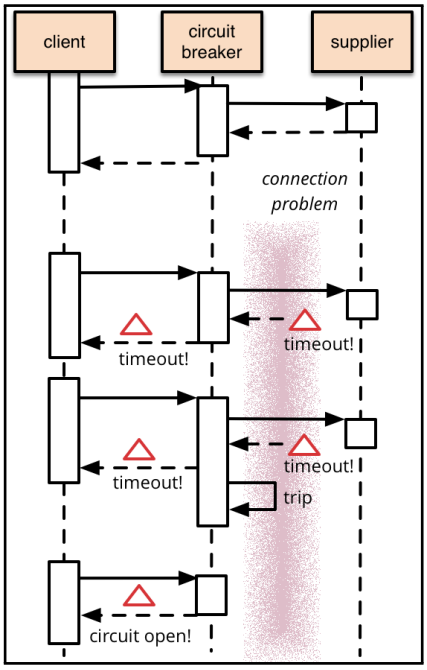


Figura 6: Patrón *Circuit Breaker*. [3]

c) **Monitorización**. Gracias a que tenemos esos *circuits breakers* podemos llevar una monitorización al detalle de cada uno de los recursos expuestos en nuestros microservicios. Para ello utilizamos *Turbine*, que se encarga de monitorizar todos los métodos de nuestros microservicios que tengan la posibilidad de cortocircuitarse.

d) **Edge Service**. *Zuul* es otro microservicio proveniente del framework de Netflix OSS que nos ayuda a implementar el patrón de sistemas puerta de enlace. A través del mismo expondremos al exterior el resto de servicios para prevenir acceso no autorizado a los recursos, el mismo utilizará de base, sistemas de balanceo de carga (con ayuda de Ribbon) y enrutamiento dinámico dentro de nuestra arquitectura. Dado que como comentamos anteriormente podremos tener réplicas de los servicios API dentro de nuestro sistema, todo registrado en nuestro sistema de descubrimiento. *Zuul* actuará como un servidor dinámico de proxy reverso, el cual no será necesario actualizar manualmente cuando un nuevo servicio se añada a nuestra arquitectura.

e) **OAuth 2.0**. `AuthService` es un microservicio que está entre la capa de orquestación y la capa de servicios API. Dado que si bien es un servicio que ofrece una serie de recursos al exterior, en mayor medida lo usaremos en cada uno de los servicios internos API para comprobar si el usuario esta autenticado y autorizado para acceder a un recurso en concreto. Mas adelande explicaremos detalladamente este microservicio así como el protocolo de seguridad que implementa.

Una vez hemos enumerado cada uno de los servicios de la capa de orquestación de nuestra arquitectura, vamos a ilustrarlo en la Figura 7.

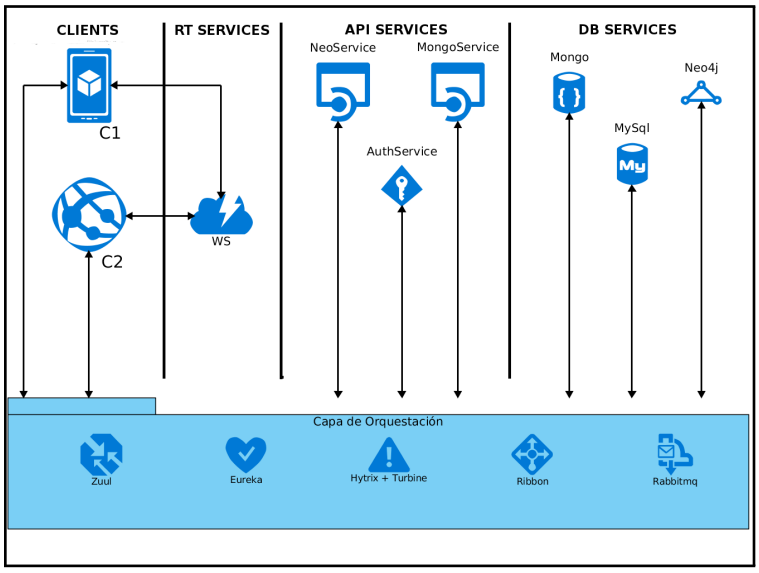


Figura 7: Arquitetura final de Whaya.

Esta figura pretende mostrar que los clientes deben de utilizar la capa de orquestación para comunicarse con los servicios API, y estos a su vez deberán comunicarse con los servicios DB también mediante la capa de orquestación. Todos ellos tomando como punto de confluencia de comunicación a *Zuul*. Éste, gracias a el resto de servicios de orquestación, será capaz de determinar las rutas y el estado de todos ellos. En el caso de los servicios API, de manera dinámica, y en el caso de los servicios DB con rutas ya preestablecidas.

El caso del servicio de datos en tiempo real es un tanto especial, ya que es el único que no utiliza nuestra capa de orquestación. A modo resumen pasamos a enumerar características comunes a los servicios de cada capa.

a) Servicios API. En esta capa contamos con los microservicios que responden a las peticiones de los clientes, entre sus características contamos con que todos ellos procederán a registrarse en *Eureka* cuando sean levantados; *Zuul* genera de manera dinámica rutas para su comunicación con otros elementos; la replicación de los mismos se puede realizar a nivel de infraestructura por lo que gracias a *Ribbon*, *Zuul* podrá de manera dinámica levantar réplicas si lo considera oportuno para aliviar el nivel de carga de los mismos.

b) Servicios DB. Esta capa cuenta con lo servicios relacionados con la persistencia de datos, como características principales la replicación de los mismos se realiza a nivel de base de datos, dado que de lo contrario podría producirse inconsistencia en los datos. Al contrario que los servicios de API estos deben establecer rutas predefinidas en *Zuul* dado que en primera instancia no están preparados para registrarse por sí solos en *Eureka*, por lo que *Zuul* no tiene visibilidad de manera automática al estado de los mismos.

c) Servicio RealTime. No cuenta con nigún tipo de replicación, al contrario que los servicios de las otras dos capas. Comparte con los servicios DB el hecho de que no se registra en *Eureka*, pero a su vez cuenta con una salvedad, al contrario que estos la comunicación de este servicio utiliza el protocolo *WebSocket*, esto conlleva que no se lleve bien con *Zuul*. Las características de este servicio provocan que por lo pronto se quede fuera de la capa de orquestación.

d) Servicios Orquestación. Todos los servicios de esta capa, excepto *Zuul*, tienen la capacidad de replicarse a nivel de infraestructura, a su vez

todos ellos son los que se encargan de controlar la comunicación entre el resto de capas estableciendo control en el flujo de información.

2.5. Seguridad de microservicios

Para proteger la seguridad de nuestra arquitectura de microservicios contamos principalmente con dos técnicas complementarias.

2.5.1. Comunicación segura HTTPs

En primer lugar conviene recordar que nuestra arquitectura solo tiene un punto de acceso, más concretamente *Zuul*, esto nos ayuda a que simplemente en ese punto tengamos levantado todo lo relacionado con `https` de nuestros servicios, ya que solo aquí tendremos comunicaciones con sistemas externos. Por lo tanto, no hay necesidad de tener que implementar, en la red interna generada por *Docker*, comunicación mediante protocolos seguros, ya que tan solo en esa red tendremos visibilidad a los datos manejados en los microservicios.

Para poder firmar la seguridad del protocolo `https` hemos utilizado la herramienta proporcionada por la **Linux Foundation** `Let's Encrypt` [8]. Ya que aunque siempre tenemos la opción de generar un certificado nosotros mismos, y dado que no somos una agencia certificadora de confianza en ningún navegador, estos nos detectarían como una página no segura, imposibilitando el acceso a nuestro *back-end*. Gracias a `let's encrypt` podemos firmar de forma segura y con confianza nuestro protocolo de seguridad, y como hemos comentado antes, tan solo debemos de hacerlo en un punto, *Zuul*.

Por otro lado, para controlar tanto la autentificación como la autorización de nuestros microservicios, hemos realizado la implementación del protocolo OAuth 2.0 [9]. Para ello, implementamos un servidor OAuth (como microservicio), y distintos clientes en cada uno de los microservicios que hacen uso de este servidor de OAuth para verificar que el usuario tiene acceso a los recursos a los que realiza la petición. La ventaja de utilizar este tipo de protocolo de serguridad es que contamos con la posibilidad de que nuestros microservicios realicen la comprobación de credenciales, no solo con nuestro servidor, sino también con otros servidores de confianza como pueden ser Google, Facebook, LinkedIn, etc. Ya que estos ofrecen la posibilidad mediante API de actuar como servidores de autenticación con las cuentas de sus usuarios.

Para comprender este protocolo y entender cómo implementarlo ,existen gran variedad de recursos entre los que se incluye *OAuth 2.0 Cookbook: Protect your web applications using Spring Security* [10]. No obstante en el siguiente punto vamos a intentar explicar de manera sencilla cómo funciona este protocolo.

2.5.2. OAuth 2.0

OAuth 2 es la segunda versión del protocolo o framework de OAuth. Este protocolo permite acceso limitado a recursos de servicios HTTP, bien a través del "Resource Owner" o bien permitiendo el acceso a recurso de terceros para que estos puedan acceder a la misma todo a través de un cliente autorizado.

El protocolo cuenta con cuatro figuras básicas, a saber:

a) **Resource Owner**: Normalmente nosotros.

b) **Resource Server**: Éste viene a ser el servidor que aloja los datos que estan protegido por el protocolo.

c) **Cliente**: La aplicacion que pide el acceso a el `Resource Server`. Basicamente lo que viene a ser nuestra aplicación web, movil, etc.

d) **Authorization Server**: El server que proporciona el token de acceso al `cliente`. Este token será utilizado por el cliente para acceder al `Resource Server`. En algunos casos `Authorization Server` esta embebido en el mismo `Resource Server`, no siendo nuestro caso ya que nuestro servidor de autorización es un microservicio a parte.

Lo siguiente es hablar de qué es un token, y de qué tipos de tokens contamos en OAuth 2. Los tokens vienen a ser cadenas de texto aleatorias generadas por `Authorization Server`, las cuales los clientes deben pasar en las peticiones a los `Resource Servers`. Contamos con dos tipos:

a) **Access Token**: Permite que los datos de usuarios se accedan desde una aplicacion de terceros. Este token es mandado por el cliente bien como un parámetro, o bien como un componente de la cabecera de una petición hacia el `Resource Server`. Es importante comentar que estos tokens en el protocolo OAuth 2 tienen un tiempo de vida limitado que el `Authorization Server` define. Tiene que ser lo más confidencial posible pero muchas veces esto se convierte en algo muy difícil de asegurar,

ya que en la mayoría de los casos los clientes tienden a ser aplicaciones web o derivados, que para no estar pidiendo la autorización de usuarios continuamente, guardan los token en `session`.

b) **Refresh Token**: Este token al contrario que el anterior, tiene que ir de la mano de un `Access Token`, y se utiliza solo en el caso de que este último haya expirado. Cuando se realize esta petición de refresco de token, nuestro `Authorization Server` comunicará al cliente un nuevo par de `Access Token - Refresh Token`.

En el protocolo contamos con varias capas de autorización. La primera de ellas es la llamada `scope`, que es un parámetro que va asociado con los tokens generados que permiten limitar los derechos de acceso a nuetros recursos a nivel token. El servidor de autorización es el que limita los `scopes` generados. El cliente debe de mandar el `scope` que quiere utilizar cuando realiza una petición.

Otra capa de autorización sería el registro como cliente de nuestra aplicación. Nuestro servicio de autorización no cuenta tan solo con la autentificación de usuarios, sino que también debe ser autorizada la aplicación que esta usando el mismo. Esto confirma que se puede dar el caso de que un usuario esté registrado en el sistema, pero que al intentar entrar con un cliente no autorizado, no se le permita el acceso a recursos. El registro de clientes (aplicaciones) cuenta con cuatro parametros principales a la hora de realizarse:

a) **Application Name**: Básicamente el nombre por el que identificaremos la aplicación.

b) **Grant Type(s)**: Tipos de autorización que serán usadas por el cliente.

c) **Redirect URLs**: URLs del cliente que este utilizará a la hora de recibir códigos de autorización.

d) **Javascript Origin(opcional)**: El nombre del host al cual le será permitido pedir datos al `Resource Server` mediante XMLHttpRequest.

Una vez se haga el registro del cliente, a éste se le asociará un `Client Id`, que viene a ser una cadena de texto única y un `Client Secret` una clave secreta para identificar a ese cliente que se ha de guardar en sitio seguro.

Estos dos parametros tendrán que ser enviados con cada petición que se realize de recuperación o consulta de tokens.

Anteriormente hablamos de los tipos de autorización, en el caso del protocolo OAuth 2 existen cuatro tipos basicos de tipos de autorización, de los cuales presentaremos una pequeña explicación de cuando se usan y un diagrama de secuencia que ilustre un escenario de uso de cada uno de los mismos:

a) **Authorization Code Grant** (Figura 8). Este tipo debería ser utilizado cuando el cliente sea un web server. Ya que nos permite acceso a un token que puede ser renovado continuamente con el refresh token siempre y cuando este lo permita. Normalmente el token de acceso se guarda como una variable de `session` en nuestro navegador y el cliente nunca tiene porqué verlo.

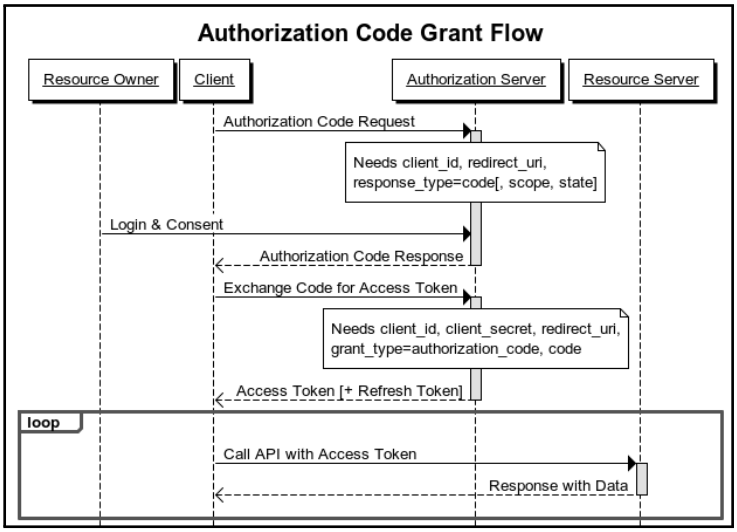


Figura 8: Authorization Code Grant Flow.

b) **Implicit Grant** (Figura 9). Se usa normalmente también cuando el cliente es un navegador utilizando un lenguaje del estilo JavaScript. Pero al contrario que el anterior no se permite la renovación del token, por lo que suelen ser tokens proporcionados para un solo uso.

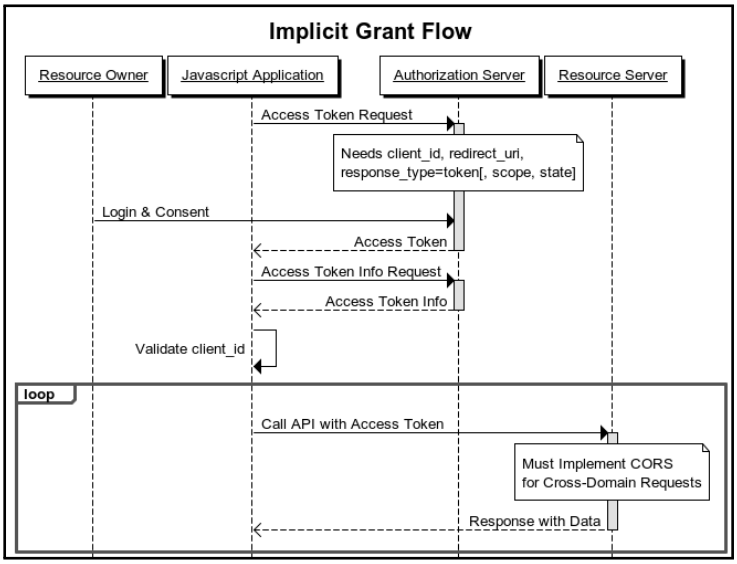


Figura 9: Implicit Grant Flow.

c) **Resource Owner Password Credentials Grant** (Figura 10). Se suele utilizar cuando el servidor de autorización y el cliente parten de la misma entidad, ya que las credenciales se mandan al cliente y éste a su vez las lanza al servidor de autorización. Bajo nuestro punto de vista aún cuando haya completa confianza entre el cliente y el servidor de autorización sería mas serio utilizar `Authorization Code Grant`.

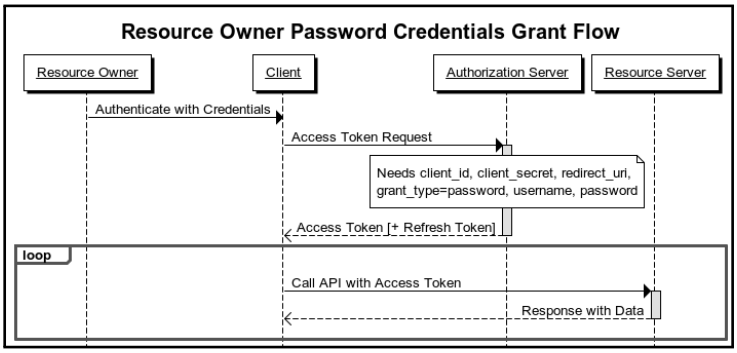


Figura 10: Resource Owner Password Credentials Grant Flow.

d) **Client Credentials Grant** (Figura 11). Normalmente este tipo de autorización se usa cuando el cliente es a la vez el `Resource Owner`. En

este caso no se requiere autentificación por parte del usuario.

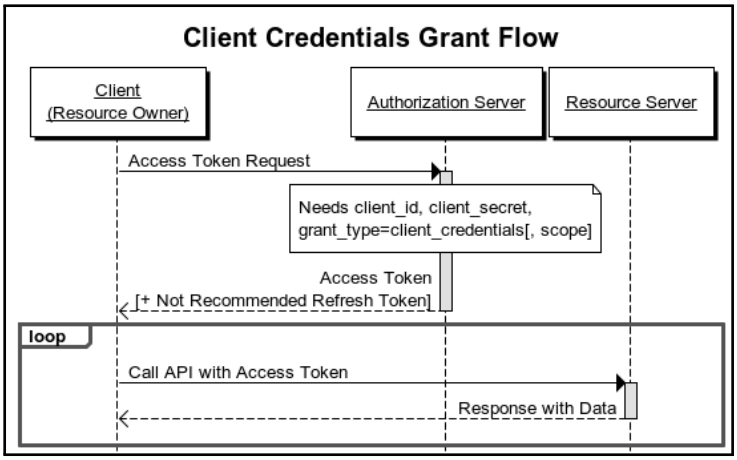


Figura 11: Client Credentials Grant Flow.

En nuestro sistema utilizamos **Authorization Code Grant** ya que es el más seguro y a la vez es una manera de utilizar el caso de uso más común de este protocolo de seguridad, pues simplemente cambiando un par de líneas de código podríamos habilitar la autenticación mediante Google, Facebook o cualquier otro sistema usado para la autenticación.

2.6. Implementación de microservicios

Después de hablar de los aspectos de diseño y analizar la orquestación de nuestros microservicios, vamos a hablar de la implementación de los mismos, así como de las herramientas utilizadas. Principalmente hemos usado JAVA, más concretamente el framework *Spring* de ese lenguaje, dado el alto acoplamiento y la facilidad de implementación de la capa de Netflix OSS en ese framework. Para ver cómo hemos realizado la implementación lo mejor es ver como hemos implementado alguno de estos microservicios, a su vez veremos pinceladas del funcionamiento de alguno de los componentes de nuestra capa de orquestación.

El hecho de utilizar *Spring* viene motivado principalmente por ser considerado el framework de JAVA más utilizado a la hora de realizar desarrollo de *back-end* en el *Developer Survey* de Stackoverflow tanto en el año 2016

[11] como en el 2017 [12]. Por lo que me motivó la idea de aprender una nueva tecnología que está muy bien considerada a la hora de realizar desarrollo web en JAVA. Además, este framework cuenta con una herramienta muy util llamada *Spring Boot* [13], la cual nos permite crear un proyecto arquetipo de manera rápida ya funcional, con todas las dependencias que consideremos oportunas, ahorrándonos posibles problemas con las versiones de las librerías que utilicemos. También proporciona un pequeño servidor *Tomcat* embebido lo cual hace que nuestro microservicio sea casi completamente independiente a la hora de ejecutarse.

Una vez ya comentado cómo se han creado cada uno de los microservicios, nos centraremos en una de ellos con más detenimiento, concretamente **Neo4jService**. Lo primero es ver el Scaffolding del mismo para tener una idea de como estructuramos nuestro microservicio en la Figura 12.

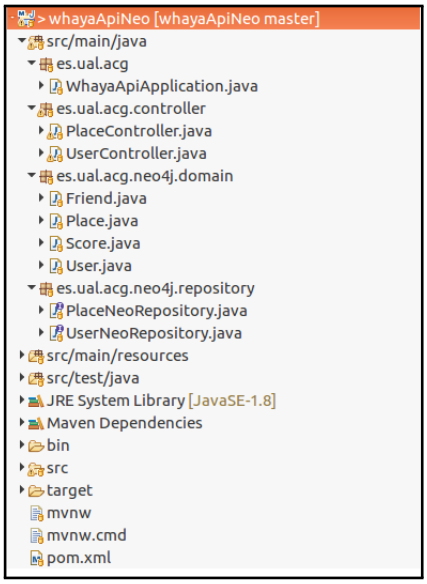


Figura 12: Scaffolding WhayaAPIneo.

Sobre la anterior figura vamos a dilucidar una serie de puntos para intentar explicar cada uno de los elementos que se pueden observar:

a) En **src/main/java** encontraremos los archivos fuente de nuestra aplicación, donde observamos una serie de subcarpetas que tienen más que ver con la separación de componentes de nuestra aplicación, a saber:

- `es.ual.acg` encontramos `WhayaApiApplication.java` que es basicamente el punto de entrada de nuestra aplicación donde tenemos implementado también todo lo relacionado con la llamada al servicio de autenticación. En el código que mostramos a continuación (Listing 3) podemos ver una serie de decoradores en clases y métodos que nos ayudan a definir ciertas propiedades de nuestra aplicación de *Spring*. Prestamos especial atención a algunos, como es el caso de `@EnableNeo4jRepositories` que nos dice que esta app va a usar elementos de *Spring Data Neo4j* como persistencia de datos. También la anotación `@EnableEurekaClient` que indica que este servidor de API pertenece a una infraestructura gobernada por `Eureka`. `@SpringBootApplication` nos viene a decir que esta aplicación es una aplicación de *Spring Boot* con un servidor web embebido. Más abajo, en `@EnableResourceServer` y `@EnableGlobalMethodSecurity`, indicamos que estamos utilizando una configuración de seguridad global con un microservicio externo para la ejecución de la misma.

```
package es.ual.acg;
import org.springframework.boot.SpringApplication; ...
@EnableTransactionManagement @ComponentScan
@EnableNeo4jRepositories @EnableEurekaClient
@SpringBootApplication
public class WhayaApiApplication {
    public static void main(String[] args) {
        SpringApplication.run(WhayaApiApplication.class, args);
    }
    @Primary @Bean
    public RemoteTokenServices tokenService() {
        RemoteTokenServices tokenService = new RemoteTokenServices();
        tokenService.setCheckTokenEndpointUrl(
            "http://zuul:8080/authservice/oauth/check_token");
        tokenService.setClientId("ClientIdPassword");
        tokenService.setClientSecret("secret");
        return tokenService;
    }
    @Configuration @EnableResourceServer
    @EnableGlobalMethodSecurity(prePostEnabled = true)
    public class OAuth2ResourceServerConfig
        extends GlobalMethodSecurityConfiguration {
        @Override
        protected MethodSecurityExpressionHandler createExpressionHandler() {
            return new OAuth2MethodSecurityExpressionHandler();
        }
    }
}
```

Listing 3: `WhayaApiApplication.java`.

- `es.ual.acg.neo4j.domain` cuenta con archivos POJO (Plain Old Java Object) que definen los Nodos y relaciones de nuestra base de datos *Neo4j*. Como ejemplo el siguiente fragmento de código (Listing 4), donde vamos a declarar ciertas propiedades de nuestro nodo User y decoramos las relaciones que si son complejas contarán con ciertas clases que representan propiedades de las relaciones, con sus *getters* y *setters*. En la misma vemos cómo contamos con un índice declarado implicitamente **email**, este índice es muy importante en nuestro sistema ya que es el que da consistencia, y por el cual relacionamos todas las distintas bases de datos que utilizamos en nuestro sistema.

```
package es.ual.acg.neo4j.domain;

import java.util.ArrayList; ...
@NodeEntity
public class User {

    @GraphId private Long id;
    @Index(unique=true) private String email;
    private String name;
    private String avatar;
    private List<Friend> friend;
    private List<Score> score;
    private List<Place> places;

    public User() {
        score = new ArrayList<Score>();
        friend = new ArrayList<Friend>();
        this.places = new ArrayList<Place>();
    }
    ...
}
```

Listing 4: Ejemplo POJO `User.java`.

- `es.ual.acg.neo4j.repository` en esta carpeta contamos con la ayuda de *Spring Data*, vemos que el archivo ejemplo (Listing 5)es una interfaz JAVA que hereda de `GraphRepository<Place>` lo cual por defecto ya nos ofrece todas las operaciones CRUD sobre la entidad `Place`, y además, vemos que definimos una serie de métodos en esa interfaz, por lo que contamos con algo más que las operaciones CRUD. Por un lado, el método `findByName(String name)`, es también parte de una serie de ayudas que nos ofrece *Spring Data* para no tener que implementar métodos simples. En este caso `findBy+nombre de propiedad` indica a *Spring Data* que

genere un código por debajo que tenga la funcionalidad de buscar un lugar en este caso por su nombre. Por otro lado, vemos que tenemos dos métodos que tienen un decorador `@Query`, más concretamente los métodos `addToSpatialIndex()` y `closePlaces()`, esto indica a *Spring Data* que genere unos métodos que cumplan con el comportamiento de la query que se indica. El primero de estos dos últimos métodos se encargan de introducir un lugar en el índice espacial de nuestra base de datos y el otro de ofrecer los lugares que tenemos a cierta distancia.

```
1  package es.ual.acg.neo4j.repository;
   import java.util.List; ..
3
   @Component
5  public interface PlaceNeoRepository extends GraphRepository<Place> {

7      @Query("MATCH (n:Place) WHERE NOT (n)<-[:RTREE_REFERENCE]-() AND n.name = {
           name} CALL spatial.addNode('LocationsLayer', n) YIELD node RETURN node
           ")
       Place addToSpatialIndex(@Param(value = "name") String name);
9
       Place findByName(String name);
11

13     @Query("CALL spatial.withinDistance('LocationsLayer',{longitude:{longitude
           },latitude:{latitude}},{distance})")
       List<Place> closePlaces(@Param(value = "latitude") double latitude, @Param(
           value = "longitude") double longitude, @Param(value = "distance")
           double distance);
15 }
```

Listing 5: Ejemplo repository `PlaceNeoRepository.java`.

- `es.ual.acg.controller`. En esta carpeta tenemos controladores de usuario y lugares. En sí esto no es necesario si solo utilizamos las operaciones CRUD, ya que las interfaces que heredan de los tipos `Repository` nos permitirian de manera automática mostrar estas operaciones si fuesen las únicas que necesitamos. En nuestro caso, contamos con comportamientos un tanto especiales en alguno de los metodos, como por ejemplo puede ser el *delete*, en el cual no borramos el nodo en concreto, sino que lo colocamos como no visible para seguir teniendo consistencia en la base de datos. Por lo tanto, solo creamos controladores porque queremos ir un poco mas allá de lo que serían operaciones básicas, y para indicar que es un controlador, lo hacemos con `@RestController`. En el ejemplo que se muestra en el siguiente código (Listing 6) comprobamos que utili-

zamos otros decoradores, como los decoradores de mapeo de rutas `@PostMapping...` para indicar el tipo de petición y ruta al que corresponde cada método, `@lstinline{@Autowired}` que indica que automáticamente va a configurar que injectar, sin tener que realizar ninguna instanciación de nuestra parte, o `@HystrixCommand` que indica el método que va a saltar cuando *Hystrix* lo requiera.

```
1
package es.ual.acg.controller;
3 import java.util.ArrayList; ...

5
@RestController
7 @RequestMapping("/users")
public class UserController {
9
    @Autowired
11  private UserNeoRepository userNeoRepository;

13  @HystrixCommand(fallbackMethod = "defaultUser")
    @PostMapping("")
15  public ResponseEntity createUser(@Param(value = "email") String email,
        @Param(value = "name") String name, @Param(value = "avatar") String
        avatar) {

17      try {
            User aux = new User(email, name, avatar);
19          User compareAux = this.userNeoRepository.findByEmail(email);
            if (compareAux != null) {
21              return new ResponseEntity("User already created", HttpStatus.
                    CONFLICT);
            }
23          userNeoRepository.save(aux);
            return new ResponseEntity(aux, HttpStatus.OK);
25      } catch (Exception e) {
            return new ResponseEntity(e, HttpStatus.INTERNAL_SERVER_ERROR);
27      }

29  }
    public ResponseEntity defaultUser(@Param(value = "email") String email,
        @Param(value = "name") String name) {
31      return new ResponseEntity(null, HttpStatus.BAD_GATEWAY);
    }
33 }
```

Listing 6: Ejemplo controlador `UserController.java`.

b) En la carpeta **resources** introducimos archivos con los parámetros de configuración global de nuestro microservicio, cosas como datos de conexión a *Neo4j* o *Rabbitmq*, apuntamos la dirección de *Eureka*, etc. En el siguiente código (Listing 7) se puede ver un ejemplo sencillo.

```
spring.application.name=neoservice
server.port=0

spring.data.neo4j.uri=bolt://neo4j:7687
spring.data.neo4j.username=neo4j
spring.data.neo4j.password=passP

eureka.client.serviceUrl.defaultZone: ${EUREKA_URI:http://zuul:8080/eureka/eureka}
eureka.instance.instance-id=${spring.application.name}:${random.int}
eureka.instance.preferIpAddress=true

spring.rabbitmq.host=rabbitmq
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
```

Listing 7: Ejemplo `application.properties`.

A parte de este tipo de microservicios contamos con los de la capa de orquestación, los cuales prácticamente *Spring Boot* nos los da ya implementados. Simplemente seleccionando el arquetipo oportuno y cambiando opciones de configuración tendremos levantada la capa de orquestación. Por poner un ejemplo, una vez creamos el arquetipo *Eureka Server* de *Spring Boot*, literalmente tendremos ya levantado el servidor de *Eureka*, solo teniendo que configurar el archivo `application.properties` del mismo con la configuración del puerto donde queremos levantarlo. Para ver lo simple que es, en el siguiente código (Listing 8) tenemos la implementación de nuestro servidor de *Eureka*.

```

package es.ual.acg;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication
@EnableEurekaServer
@EnableDiscoveryClient
public class EurekaApplication {

    public static void main(String[] args) {
        SpringApplication.run(EurekaApplication.class, args);
    }
}
```

Listing 8: `EurekaApplication.java`.

Por último, contamos con un servicio en particular que es el que trata con datos en tiempo real, mediante el framework de **Socket.io**, implementado

en `Node.js`. El framework nos permite enviar mensajes sobre el protocolo WebSocket y básicamente trata de responder a ciertos mensajes con otros. Para intentar entenderlo mejor vamos a ver un fragmento del código (Listing 9) de nuestro microservicio.

```
1
var express = require('express');
3 var app = express();
app.set('port', process.env.PORT || 9000);
5 var request = require('request');
var server = require('http').Server(app);
7 var io = require('socket.io')(server);
var port = app.get('port');
9
app.use(express.static('public'));
11
server.listen(port, function () {
13     console.log("Server listening on: http://localhost:%s", port);
});
15
function User(email, name, socketId) {
17     this.email = email;
    this.name = name;
19     this.socketId = socketId;
}
21 var users = [];

23 io.sockets.on('connection', function (socket) {

25     socket.on('connected', function (data) {

27
        socket.user = new User(data.email, data.name, socket.id);
29         users.push(socket.user);
        console.log('users:' + JSON.stringify(users))
31
    });
33
    socket.on('addConnectionFriend', function (data) {
35
        for (var i = 0; i < users.length; i++) {
37
            if (data.friend == users[i].email) {
39                 socket.to(users[i].socketId).emit("userConnected", { 'email': socket.user.
                    email, 'name': socket.user.name, 'socketId': socket.id });
                socket.emit("userConnected", { 'email': users[i].email, 'name': users[i].name
                    , 'socketId': users[i].socketId });
41             }
        }
43     });
});
```

Listing 9: Ejemplo socket.io `app.js`.

En los métodos mostrados de este servicio vemos lo que hacemos cuando nos conectamos al mismo. Simplemente nos incluimos en una variable global del servidor donde estan todos los usuarios conectados. A continuación con el mensaje *addConnectionFriend* comprobamos si el usuario amigo al que queremos decirle que estamos conectados está en la lista de usuarios conectados, y de ser así mandaremos un mensaje a ese usuario con nuestro `socketId` y a la vez recuperamos para nosotros el `socketId` del usuario. En ese momento tendremos la posibilidad de comunicarnos de manera directa entre ambos. Este servicio queda fuera de la capa de orquestación, dado que no permite réplica, ya que existe la problemática de que si tenemos varias réplicas esa lista de usuarios global cabría la posibilidad de que estuviese particionada provocando que se diese el caso de que dos usuarios estuviesen conectados sin que sean visibles entre sí. Esta problemática puede tener distintas soluciones pero se sale del contexto de este proyecto. Este servicio es el que se encarga de llevar todo lo relacionado con cuando se conectan tus amigos al sistema, el chat y por ejemplo el que pasa tu posición a tus amigos. Al realizar este microservicio es interesante ver como hemos realizado la conexión y la comunicación de datos 1 a 1 para aumentar la privacidad del sistema. El siguiente diagrama de secuencia (Figura 13) ilustra el funcionamiento del código anterior.

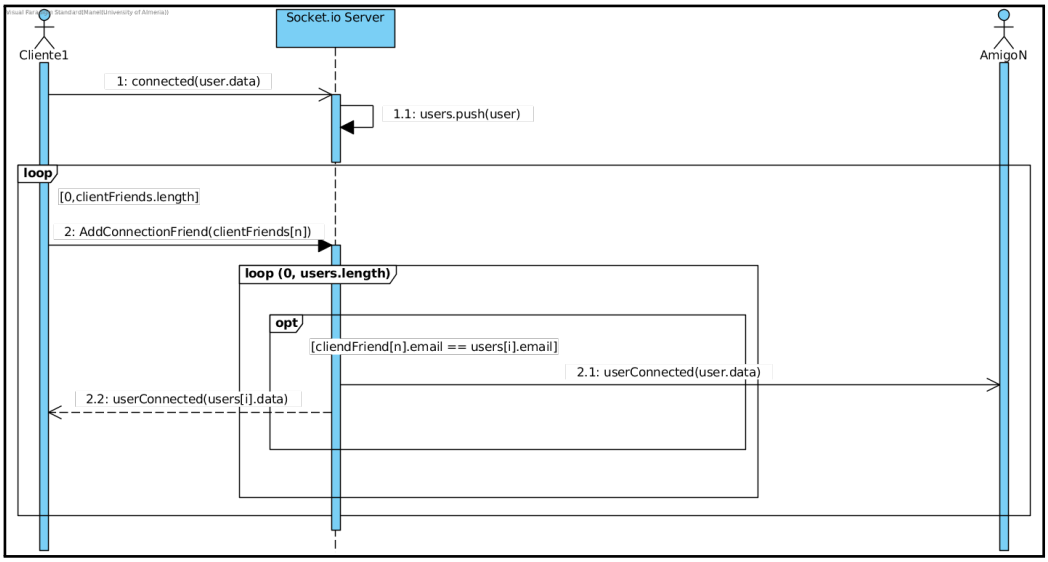


Figura 13: Diagrama de secuencia WS.

3. Whaya - Desarrollo de Front-End

A continuación, vamos a realizar una breve introducción a las tecnologías híbridas para el desarrollo de aplicaciones móviles, en la cual detallaremos las ventajas e inconvenientes de cada una de ellas. Después, nos centramos en Ionic, la tecnología elegida para el desarrollo de Whaya. Describiremos una introducción a la misma, así como sus particularidades. Por último, nos centraremos en el desarrollo de nuestra aplicación en sí.

3.1. Introducción

Al realizar una aplicación para probar la arquitectura propuesta partimos de la premisa de que la misma fuese híbrida, dado que resultaba interesante tan solo desarrollar una vez y ser capaz de desplegar en varias plataformas. Para ello vamos a intentar explicar las cuatro alternativas más utilizadas en este tipo de desarrollo. A continuación veremos cuáles son los puntos fuertes y débiles de cada una de estas alternativas.

3.1.1. Apache Cordova

Apache Cordova es una aproximación opensource al desarrollo multiplataforma, su mayor ventaja es la simplicidad con respecto a las otras alternativas. Permite crear aplicaciones de todo tipo de corte, no solo móviles, sino también de escritorio o incluso web. Para ello utiliza Web APIs, y es capaz de envolver aplicaciones web en una aplicación nativa de la plataforma en cuestión y utiliza por sí sola la API correspondiente de la plataforma en cuestión cuando se la llama.

Las aplicaciones *Cordova* son simplemente una colección de páginas HTML que son renderizadas en el teléfono móvil como WebViews. Para desarrollar las aplicaciones solo te hace falta HTML5, CSS y JavaScript. Entre las expuestas es la única que sigue completamente la filosofía "write once, run anywhere".

Ventajas

- Pequeña y simple API nativa que permite ser usada en diferentes entornos de desarrollo.
- Alta reusabilidad con HTML, CSS y JavaScript.

- Cualquier cosa escrita como una página web puede ser fácilmente envuelta como una aplicación nativa de la plataforma para la que estamos desarrollando.
- Soporte para todas las plataformas y sistemas operativos que incluyen Android, iOS, Windows Phone, Blackberry, Firefox OS y Ubuntu.
- La mayoría de desarrolladores están acostumbrados a codificar en HTML/CSS/JavaScript por lo que tienen muy fácil el hecho de empezar a trabajar en *Cordova*.

Desventajas

- Peor rendimiento que aplicaciones de código nativo ya que al fin y al cabo son aplicaciones web que se lanzan con el motor del navegador del dispositivo.
- Muchas librerías muy fragmentadas con frameworks de nivel básico.
- Interfaces de Usuario que pueden distar mucho de la apariencia nativa de la plataforma, aunque esto depende mucho del desarrollador en sí.

3.1.2. Titanium Appcelerator

Titanium es una plataforma de desarrollo basada en JavaScript. Titanium utiliza JavaScript para escribir código de aplicaciones que después utilizan la API y la UI de una y solo una de las plataformas a las que va destinada. Es decir, al contrario que *Cordova*, esta plataforma de desarrollo no está orientada a la filosofía "write once and run anywhere", aunque da la posibilidad de reutilizar las partes comunes entre las plataformas que intentamos desarrollar.

Tiene la desventaja que al contrario que *Cordova* y Xamarin tendremos que aprender parte de la API de cada una de las plataformas que vamos a desarrollar y no solo eso, sino que tendremos que aprender la capa que envuelve a esas APIs en JavaScript que es como las terminaremos utilizando.

Ventajas

- Mejor rendimiento, ya que se hace un uso de las APIs de cada plataforma de manera nativa. También tenemos la ventaja que da acceso específico a características propias de cada plataforma.

- La apariencia de las aplicaciones realizadas con Titanium es puramente nativo en cada plataforma, y no tendremos que dedicar mucho tiempo a que esto sea así.
- Con JavaScript se asegura un desarrollo fácil y rápido.

Desventajas

- No tiene soporte para librerías de terceros.
- Dificultad a la hora de desarrollar aplicaciones complejas.
- Ya que no utiliza HTML5 o CSS, la animaciones y los elementos del DOM muchas veces actúan con un poco de retardo, en general responden de manera más lenta.

3.1.3. Microsoft Xamarin

Xamarin, conocido originalmente como MonoTouch es otro framework multiplataforma que se ha hecho un hueco en el mercado con su propio IDE. Funciona con C# todo dentro del ".NET Framework" y te permite realizar aplicaciones nativas utilizando las APIs y UIs de cada plataforma.

Xamarin viene con la librería Xamarin.Forms que nos permite escribir UIs para una plataforma de manera nativa y después compartirlas con el resto de plataformas que queremos desarrollar. Xamarin soporta iOS, Android y plataformas Windows.

Ventajas

- Xamarin incluye TestCloud que nos permite probar aplicaciones de manera automática.
- Proporciona código 100 % reutilizable en el desarrollo de UIs gracias a Xamarin.Forms, lo cual nos ahorrará mucho tiempo y esfuerzo.
- Soporta patrones como MVC y MVVM. Xamarin.Android soporta Google Glass, Android Wear y Firephone.
- La curva de aprendizajes es muy relativa. Si conoces C# resulta bastante fácil comenzar a desarrollar con Xamarin.

Desventajas

- No permite acceso directo a controles específicos de elementos de la UI de Android.
- Utilizar Xamarin provoca ciertos retrasos en los tiempos de carga cuando la app está en ejecución.
- No permite compartir ni desarrollar código fuera del entorno de desarrollo Xamarin.

3.1.4. Ionic

Ionic [14] es un framework desarrollado sobre *Apache Cordova*, por lo que comparten sus puntos fuertes y débiles. No obstante el desarrollo se basa en dos frameworks que van de la mano. Por un lado *Angular* [15], framework desarrollado por Google, el cual está completamente basado en componentes auto-contenidos, básicamente todo lo que desarrollamos con él (páginas, servicios, items, etc) terminan convirtiéndose en componentes, a veces individuales y otras como parte de otros componentes. Por otro lado, el propio *Ionic* es otro framework que se construye sobre el propio *Angular* donde todos sus componentes vienen a ser componentes *Angular*, pero que en el caso de componentes gráficos estos se adaptan a la plataforma en la cual son desplegados. Una ventaja es que podemos mezclar indistintamente componentes de ambos lenguajes (tanto *Angular* como *Ionic*). En ambos trabajamos en TypeScript, un lenguaje de alto nivel extensión de Javascript.

La idea es que desarrollamos una aplicación web basándonos en componentes *Angular* y *Ionic*, en la que el cliente de *Ionic* a la hora de construir el proyecto va a realizar optimizaciones basándose en la plataforma para la que están siendo construidos. Con lo cual es una solución que se acerca mucho más al comportamiento de los componentes nativos incluso donde *Cordova* tiene problemas, que es normalmente en los componentes gráficos. Respecto al uso de componentes de tipo dispositivo, dentro de la plataforma donde estamos desarrollando, véase cosas como cámara, gps, acelerómetros, etc. el comportamiento es idéntico al de *Cordova*, no hay ningún impacto significativo con respecto al uso nativo de los mismos.

Ventajas

- Aplicaciones mucho más rápidas a la hora de desarrollar.
- Las aplicaciones cuentan con un aspecto que corresponde a las plataformas en las cuales van a ser desplegadas.
- Gran cantidad de plugins desarrollados para hacer uso de los sensores internos de los dispositivos.
- El catálogo de librerías compatibles con *Ionic* es muy extenso.
- Cliente de *Ionic* muy robusto, lo que permite generar un código inicial de los componentes bastante completo.

Desventajas

- Peor rendimiento que aplicaciones de código nativo.
- Necesidad de seguir la nomenclatura y la estructura de proyecto que exige *Ionic*.
- No recomendable para el desarrollo de videojuegos o de compoentes 3D ya que se ralentiza enormemente el rendimiento.

En la tabla de la Figura 14 se observan las principales diferencias entre las distintas tecnologías de desarrollo híbrido.

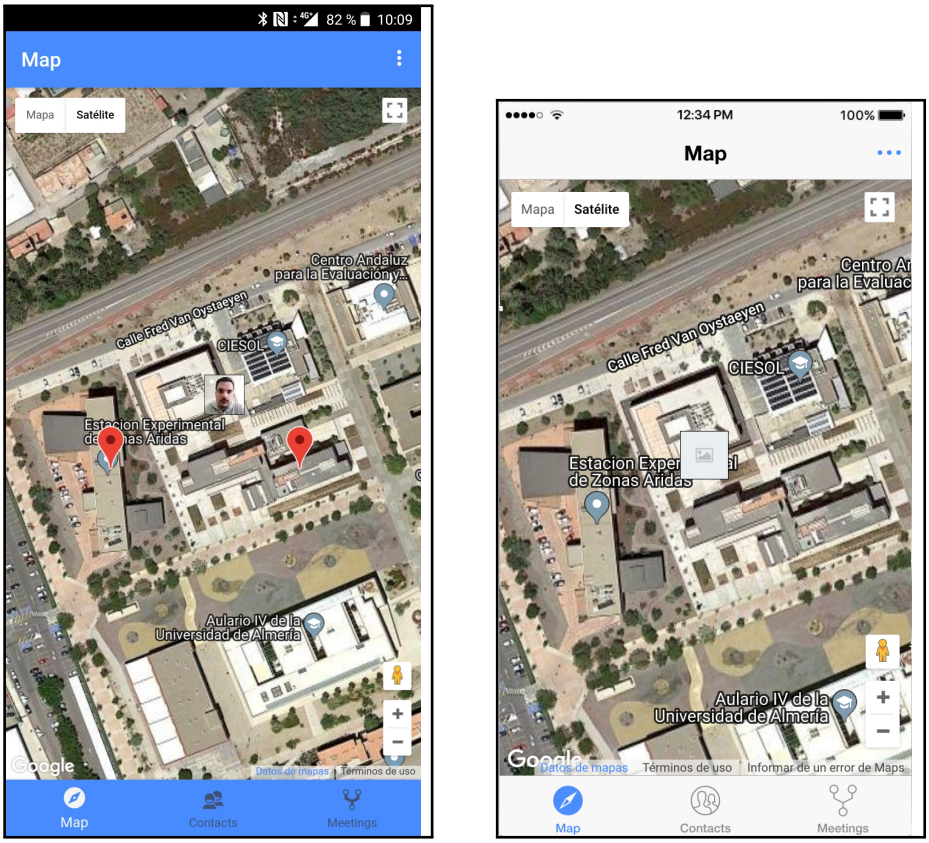
Figura 14: Tabla comparativa de tecnologías híbridas.

	Apache Cordova / Ionic	**Appcelerator**	**Xamarin**
Plataformas soportadas	iOS, Android, Windows Phone, Blackberry, OSX, Windows, Linux,Ubuntu Phone...	Blackberry	Android, iOS, Windows
Lenguaje de programación	HTML5, CSS, JavaScript / Angular	JavaScript	C#
UI	Web UI / Parcialmente nativa	Nativa	Nativa
Acceso a API del sistema	Limitada a componentes desarrollados para la plataforma en concreto	Completa	Completa
Soporte de estandares web	SI	NO	NO
Soporte del DOM	SI	NO	SI
Desempeño nativo	NO	SI	SI
Usado por	IBM, Sony, Mozilla, Intel...	Cisco, VMWare, MitsubishiElectric...	GitHub, Microsoft, FourSquare...

Una vez vista las principales diferencias entre las distintas tecnologías para el desarrollo de aplicaciones híbridas decidimos utilizar la alternativa de *Ionic* ya que nos permite sin excesivos esfuerzos desarrollar una aplicación básica para probar nuestro back-end, que a su vez, sin mucho esfuerzo, va a tener un aspecto nativo a la plataforma donde se despliegue.

3.2. ¿Que es Whaya?

Whaya es una aplicación desarrollada con el objetivo principal de probar la arquitectura de microservicios propuesta anteriormente. A pesar de ello, la aplicación presenta una cierta funcionalidad importante como detallaremos más adelante. Para desarrollarla, como comentábamos anteriormente, hemos elegido *Ionic* que es un framework de desarrollo híbrido de aplicaciones que hace uso de *Angular* para generar aplicaciones móviles basadas en componentes. De manera nativa *Ionic* nos ofrece la posibilidad de utilizar todo tipo de funcionalidades de nuestro dispositivo móvil mediante la instalación de plugins, que no vienen a ser otra cosa que nuevos componentes que nos permiten seguir montando el **rompecabezas** mediante el cual desarrollamos nuestra aplicación. Dado que no todo es el uso de los recursos hardware de nuestro dispositivo también contamos por defecto con componentes gráficos, los cuales *Ionic* se encarga a la hora de compilar nuestra aplicación hacer que esta luzca de la manera que esperamos en dispositivo en el que la vamos a desplegar. La Figura 15 muestra una comparativa entre la interfaz de la versión Android y la iOS, en ellas podemos apreciar particularidades de cada una de las plataformas, como pueden ser dónde están situados los títulos de las cabeceras de las ventanas de la aplicación, iconos de las mismas, menús adaptados a plataformas, etc.



(a) Interfaz android (b) Interfaz ios

Figura 15: Android vs iOS.

La posibilidad de que, de manera automática se adapte la interfaz a la plataforma donde se ejecuta la aplicación al utilizar los componentes nativos de *Ionic*, proporciona una descarga de trabajo sustancial de cara a diseñar la interfaz de usuario, y aún así proporcionar a los usuarios que utilicen la aplicación una experiencia de uso familiar.

Después de comentar las particularidades de *Ionic* vamos a pasar a nombrar los casos de uso que nuestra aplicación desarrolla, los cuales presentamos en la Figura 16.

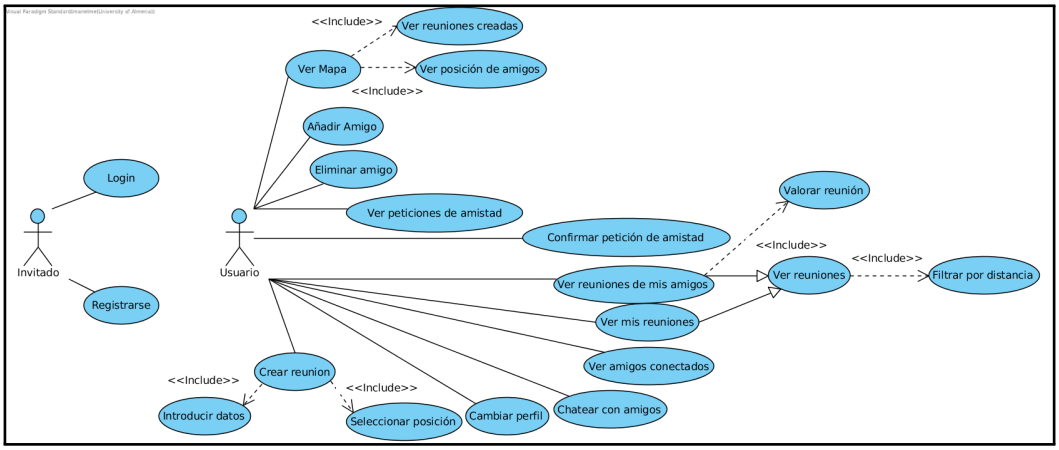


Figura 16: Diagrama de casos de uso Whaya.

Consideramos el nombre de cada uno de los casos de uso lo suficientemente descriptivo como para no tener que entrar en profundidad en cada uno de ellos, ya que queda un poco fuera del alcance de este proyecto.

3.3. Desarrollo de Whaya

Para entender mejor como se desarrolla una aplicación con *Ionic* vamos a intentar explicar cómo hemos desarrollado alguna de las páginas de nuestra aplicación, así como consideraciones que hay que tener en cuenta a la hora de realizar implementaciones con *Ionic*.

En primer lugar tenemos que instalar `Node.js`, para a continuación instalar de manera global el intérprete de *Ionic* junto con su base *Cordova*. Esto ya se encarga de descargar todas las dependencias necesarias para hacer funcionar la aplicación en nuestro sistema. Para ello, y dado a que nuestra máquina de desarrollo está basada en Ubuntu, lo único que tenemos que hacer es ejecutar las ordenes pertinentes de instalación.

```
curl -sL https://deb.nodesource.com/setup_9.x | sudo -E
   bash -
sudo apt-get install -y nodejs
npm install -g cordova Ionic
```

Una vez realizamos la instalación de los intérpretes necesarios, nos situamos en la carpeta donde queremos guardar nuestro proyecto y ejecutamos

la orden de iniciación del proyecto. En la misma tenemos que disparar también el arquetipo que deseamos utilizar de inicio. En nuestro caso iniciamos nuestra aplicación con el arquetipo tabs, y declaramos el nombre de nuestra aplicación.

```
1    Ionic start Whaya tabs
```

Esto nos va a generar todo la estructura de carpetas junto con los archivos básicos necesarios en el proyecto. En la Figura 17 apreciamos cuales son los archivos y carpetas mínimos necesarios, y sabiendo que en la raíz del proyecto hay que prestar especial atención a 4 cosas:

a) **config.xml**. En este archivo se guarda la configuración del proyecto, tanto de manera global, como configuraciones de plataformas en concreto. Configuraciones correspondientes al nombre y versión de la aplicación, así como mensajes de permisos de uso de un componente hardware de la aplicación.

b) **package.json**. En este archivo tenemos las dependencias directas de nuestro proyecto

c) **resources/**. En esta carpeta se guardan archivos de recursos generados por el cliente de *Ionic*, del estilo de iconos de la aplicación o pantallas de carga de la misma.

d) **src/**. Esta carpeta contiene los componentes en los cuales implementamos nuestra aplicación, páginas, servicios, etc.

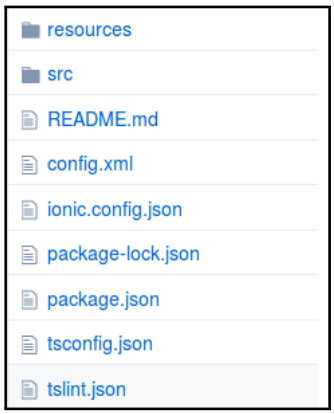


Figura 17: Scaffolding de Whaya.

El siguiente paso antes de iniciar el proceso de programación de nuestra aplicación será indicarle al proyecto *Ionic* cuales son las plataformas objetivo, en nuestro caso Android e iOS. Para ello utilizamos la siguiente orden.

```
Ionic cordova platform add android
```

Esta orden, de manera automática, va a generar una serie de líneas correspondientes a la plataforma que hemos elegido dentro del archivo `config.xml` teniendo en cuenta, plugins que hayamos añadido previamente al proyecto, configurándolos de manera adecuada a la plataforma. Una vez hemos realizado estos pasos previos entramos en faena con la aplicación propiamente dicha.

Como hemos dicho antes, toda aplicación *Ionic* se basa en el desarrollo de componentes, estos son pequeños fragmentos de código que aportan una funcionalidad. En nuestra aplicación hemos hecho uso principalmente de 4 tipos, a saber:

a) **Pages** (Listing 10). Componentes que engloban todo lo necesario con la interacción con el usuario, en el siguiente fragmento de código apreciamos cómo se declaran estos componentes y a la vez de qué archivos se componen. Como se puede observar, lo primero que vemos después de los `import` es la inclusión de una serie de decoradores, `@IonicPage` indica que estamos hablando de un componente página. Ademas, podemos apreciar que tiene dos archivos principales: la "vista", que la podemos encontrar donde indica `templateUrl` y el "controlador"que viene a ser la clase que se exporta, que de cara a archivos externos la utilizaremos con el nombre indicado en el campo `selector`.

```
import { Component, ViewChild, ElementRef } from '@angular/core';
import { IonicPage, NavController, NavParams } from 'Ionic-angular';
import { GeolocProvider } from '../../providers/geoloc/geoloc'; ...
@IonicPage()
@Component({
  selector: 'page-meetings',
  templateUrl: 'meetings.html',
})
export class MeetingsPage {
  @ViewChild('ranger') ranger: ElementRef;
  constructor(public navCtrl: NavController, public navParams: NavParams,
       private storage: Storage, private geolocProvider: GeolocProvider,
       private neoService: NeoServiceProvider) {}
  ...
}
```

Listing 10: Ejemplo componente Page `MeetingsPage.ts`.

b) **Providers** (Listing 11). En el mundo *Angular* se denominan servicios, y cumplen con la funcionalidad de comunicarse con servicios web o con servicios internos del móvil como pueden ser la cámara, GPS, etc. En este caso vemos que simplemente hemos de declararlo como un archivo `@Injectable`, lo cual indica que está preparado para ser usado por otros componentes, cómo pueden ser por ejemplo las `pages`.

```
import { Injectable } from '@angular/core';
import { Http, Headers } from '@angular/http';...

@Injectable()
export class NeoServiceProvider {
    private headers = new Headers();
private userAuth: any;
constructor(public http: Http, public storage: Storage) {}
...
}
```

Listing 11: Ejemplo componente Provider `NeoServiceProvider.ts`.

c) **Pipes** (Listing 12). Este tipo de componentes son los que se encargan básicamente de transformar datos antes de mostrarlos al usuario. En el ejemplo vemos como se declaran con el decorador @Pipe y a la vez lo dotamos de un nombre relativo.

```
import { Pipe, PipeTransform } from '@angular/core';
import * as moment from 'moment';
@Pipe({
  name: 'relativeTime',
})
export class RelativeTime implements PipeTransform {
  /**
   * Takes a value and makes it lowercase.
   */
  transform(value: string, ...args) {
      return moment(value).fromNow();
  }
}
```

Listing 12: Ejemplo componente Pipe `RelativeTime.ts`.

d) **Components** (Listing 13). En nuestro caso hemos utilizado otro tipo de componente que queda un poco fuera de *Ionic*, y podríamos considerarlo más como un componente *Angular*. Básicamente, es una manera de generar otra capa de separación o abstracción. En nuestro caso lo uso para separar aquellos componentes (visuales) que por sí solos no significan nada, y que dependen del uso que otros hagan de ellos. Por

ejemplo, un `@Component` puede ser la relación que tiene `emojiPicker` con `ChatPage`. Como ejemplo de `@Component` tenemos el siguiente, que es un componente que nos permite elegir un emoji para el chat.

```
import { Component, forwardRef } from '@angular/core';
import { EmojiProvider } from "../../providers/emoji";
import { ControlValueAccessor, NG_VALUE_ACCESSOR } from "@angular/forms";

export const EMOJI_PICKER_VALUE_ACCESSOR: any = {
    provide: NG_VALUE_ACCESSOR,
    useExisting: forwardRef(() => EmojiPickerComponent),
    multi: true
};

@Component({
    selector: 'emoji-picker',
    providers: [EMOJI_PICKER_VALUE_ACCESSOR],
    templateUrl: './emoji-picker.html'
})
export class EmojiPickerComponent implements ControlValueAccessor {

    emojiArr = [];

    _content: string;
    _onChanged: Function;
    _onTouched: Function;
constructor(emojiProvider: EmojiProvider) {
this.emojiArr = emojiProvider.getEmojis();
}
...
}
```

Listing 13: Ejemplo componente Component `EmojiPickerComponent.ts`.

Como vemos, el lenguaje utilizado para definir los distintos componentes es TypeScript, extensión de Javascript que proporciona ciertas funcionalidades adicionales de ES6 (EcmaScript 6), como es el tipado de objetos web para detectar errores en tiempo de compilación, el uso de clases, etc. También hemos hecho uso, aunque solo donde es adecuado, de variables de tipo `Promises`, las cuales nos permiten dotar de cierta sincronía a algunas peticiones web, y variables `Observables` que dotan a la aplicación de la posibilidad de actuar frente a cambios de valor de una variable.

También cabe destacar que tenemos dos maneras distintas de cargar nuestros componentes *Ionic* en la aplicación:

- Por un lado, contamos con la forma tradicional, que es cargar el componenente al cargar la aplicación, para ello tenemos que declararlo como un entryComponent en el archivo `app.module.ts`. Este archivo actúa

como director de la aplicación y en él declararemos todos los componentes que nuestra aplicación hará uso de manera global. En el siguiente ejemplo (Listing 14) podemos ver como el componente HomePage es declarado como un entryComponent en nuestra aplicación, esto quiere decir que se cargará cuando la misma se ejecute.

```
import { MyApp } from './app.component';
import { HomePage } from '../pages/home/home';
...
@NgModule({
    declarations: [MyApp, HomePage],
    imports: [ ... ],
    bootstrap: [IonicApp],
    entryComponents: [MyApp, HomePage],
    providers: [ ... ]
})
export class AppModule {}
```

Listing 14: Ejemplo módulo de aplicación `app.module.ts`.

- En segundo lugar podemos cargar componentes en nuestra aplicación al estilo *LazyLoading* (Listing 15). Esto nos permite cargar el componente solamente cuando va a ser usado, para ello el componente tendrá que estar preparado para este método de carga. Lo que debemos hacer es proveer al mismo con su propio `NgModule` que básicamente proporciona al componente la capacidad de encapsular todo lo que necesita para funcionar en el mismo. Una vez esté preparado el componente, tan solo tendremos que importarlo donde vayamos a utilizarlo.

```
import { NgModule } from '@angular/core';
import { IonicPageModule } from 'Ionic-angular';
import { HomePage } from './home';
@NgModule({
    declarations: [HomePage],
    imports: [IonicPageModule.forChild(HomePage)],
})
export class HomePageModule { }
```

Listing 15: Ejemplo módulo de componente `home.module.ts`.

Siempre que sea posible, sería interesante que los componentes se cargasen de la segunda forma, ya que esto permite que la aplicación haga uso de una cantidad de memoria menor, pues solo cargará los componentes que son necesarios cuando los esté utilizando. No obstante, la mayor desventaja es que a veces es interesante que los componentes estén "pre-cacheados.ªntes incluso de que se haga uso de ellos, todo para que la experiencia de usuario

sea mucho más rápida, en este caso es mucho más interesante cargar los componentes de manera tradicional.

Otro punto a tener en cuenta es el ciclo de vida que tiene un componente *Ionic*. Para ello, cualquier componente de *Ionic* tiene de base los metodos que se muestran en la Figura 18

Figura 18: Tabla de eventos de ciclo de vida *Ionic*

Page Event	**Return**	**Descripción**
ionViewDidLoad	void	Se ejecuta cuando la página se ha cargado. Este evento solo ocurre una vez por página creada. Si salimos de la página pero ésta sigue cacheada el evento no se volverá a disparar cuando la misma vuelva a verse.
ionViewWillEnter	void	Se ejecuta justo antes de entrar a la página.
ionViewDidEnter	void	Se ejecuta cuando la página se convierte en la página activa.
ionViewWillLeave	void	Se ejecuta justo antes de que la página deje de ser la activa.
ionViewDidLeave	void	Se ejecuta cuando la página deja de ser la página activa.
ionViewWillUnload	void	Se ejecuta cuando la página va a ser destruida y sus elementos eliminadas
ionViewCanEnter	Promise	Se ejecuta cuando la página puede ser accedida. Se concibe como un metodo de estilo "guarda", que permite por ejemplo que solo se entre a ciertas páginas si estas autenticado.
ionViewCanLeave	Promise	Se ejecuta cuando se puede dejar la página. Se concibe como un metodo de estilo "guarda"que permite por ejemplo que solo se salga de la página si estas autenticado.

Es importante que entendamos bien cuando y cómo se ejecutan cada uno de los métodos del ciclo de vida de nuestra aplicación *Ionic*, dado que buena

parte del funcionamiento de la misma estará basado en la utilización correcta de estos métodos. Para ilustrar mejor estos métodos del ciclo de vida podemos echar un vistazo a la Figura 19, la cual está extraída de una entrada del blog oficial de *Ionic* [4] donde explican de manera muy clara cómo funcionan.

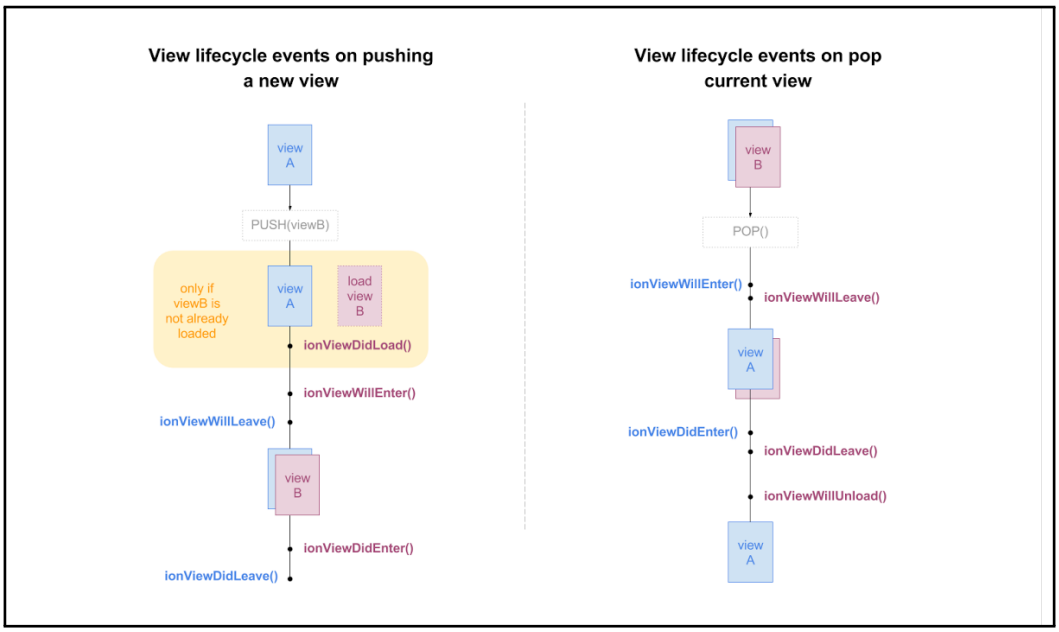


Figura 19: Ionic view Lifecycle Events [4].

Por último, para ir probando la aplicación tenemos dos opciones. Por un lado usando la primera orden de *Ionic* que se muestra a continuación, que nos permite de manera rápida mostrar la interfaz de la aplicación en las distintas plataformas. O bien realizar un despliegue en la plataforma en concreto, lo cual nos permite compilar la aplicación como proyecto correspondiente a la plataforma donde la estamos desplegando. En nuestro caso cuando la lancemos en Android se generará una carpeta dentro de `platforms/` que contendrá un proyecto de Android Studio, o en el caso de iOS un proyecto de XCode.

```
Ionic serve --lab
Ionic cordova run android
```

4. Conclusiones y trabajo futuro

A lo largo de la consecución de este trabajo de fin de master, consideramos que los objetivos propuestos han sido debidamente cumplidos. Especialmente interesante ha sido establecer una versión un tanto particular de arquitecturas de microservicios, temática particularmente interesante a día de hoy. He aprendido el uso de nuevas técnicas y tecnologías punteras de los campos propuestos, dando cabida a temáticas muy variadas relacionadas con orquestación de microservicios, persistencia políglota, infraestructura como código, desarrollo híbrido de aplicaciones, etc. Otra buena experiencia didáctica ha sido encontrarnos con dificultades y poder resolverlas, dada la presencia de datos en tiempo real, particularmente como responder a cambios en los mismos.

Como trabajo futuro sería interesante dotar de mayor funcionalidad a la aplicación móvil, incluir datos medioambientales, estado meteorológico o temas relacionados con auto check-in en las reuniones a las que vamos mediante coordenadas. Un reto que se ha quedado en el tintero ha sido replicación del servicio de datos en tiempo real, una solución factible sería que cuando los usuarios se conecten a este servicio, se realizase una llamada a una base de datos en la que se guarden tanto el id del usuario que se ha conectado, cómo en que servidor réplica está, y cuando se mande algún mensaje entre usuarios, el usuario1 compruebe primero en que servidor réplica está el usuario2, que id tiene en ese otro servidor, para poder de esa manera pasar el mensaje del servidor del usuario1 al servidor del usuario2. También sería interesante ver si pudiéramos hacer funcionar nuestro servicio de datos en tiempo real con la capa de orquestación buscando alternativas a *Zuul*, como puede ser por ejemplo *Consul*.

5. Bibliografía

[1] BLP Baas. Nosql spatial–neo4j versus postgis. Master's thesis, 2012.

[2] Martin Fowler. Microservices: a definition of this new architectural term. https://martinfowler.com/articles/microservices.html.

[3] Martin Fowler. Circuit breaker. https://martinfowler.com/bliki/CircuitBreaker.html.

[4] Ionic - lifecycle events. https://blog.ionicframework.com/navigating-lifecycle-events/.

[5] Open geospatial consortium - ogc. http://www.opengeospatial.org/.

[6] Kief Morris. *Infrastructure as code: managing servers in the cloud*. OReilly Media, Inc., 2016.

[7] Official docker documentation. https://docs.docker.com/, Jan 2018.

[8] Let's encrypt: Free ssl/tls certificates. https://letsencrypt.org/.

[9] Oauth.net. https://oauth.net/2/.

[10] Adolfo Nascimento. *OAuth 2.0 Cookbook*. Packt Publishing, Birmingham, 2017.

[11] Stackoverflow developer survey 2016. https://insights.stackoverflow.com/survey/2016#technology.

[12] Stackoverflow developer survey 2017. https://insights.stackoverflow.com/survey/2017#technology.

[13] Spring boot. https://projects.spring.io/spring-boot.

[14] Arvind Ravulavaru. *Learning Ionic 2, Second Edition*. Packt Publishing, 2017.

[15] Ari Lerner, Felipe Coury, Nate Murray, and Carlos Taborda. *ng-book 2: The Complete Guide to Angular 2*. Fullstack.io, 2016.