SD	Sistemas Distribuidos
20/21	Práctica
	Memoria de la práctica

➢ Introducción

Esta práctica ha sido realizada utilizando la pila MEAN para su desarrollo. Esto es, se ha utilizado MongoDB como base de datos, Express como servidor y Node.js para la ejecución de código JavaScript. Quedaría el uso de Angular para la parte del front-end, pero no ha sido realizado en esta práctica.

Para poner en marcha esta práctica, primero se ha procedido con la instalación de los paquetes necesarios. Esta se desarrolló en **Ubuntu 20.10**, utilizando el gestor de paquetería de este SO, apt-get, por lo que los comandos a utilizar en la instalación fueron los siguientes:

```
$ sudo apt install nodejs
$ sudo apt install npm
```

Una vez tenemos los paquetes necesarios instalados, procedemos a la instalación del gestor de paquetes para node:

```
$ npm init
```

A partir de este momento ya podemos trabajar sobre código que queramos ejecutar, como por ejemplo realizar una pequeña versión de un servidor.

- Express

  El siguiente paso es instalar Express mediante npm. Para ello, realizamos el siguiente comando:

  ```
  $ npm i -S express
  ```

  En el comando anterior, el parámetro *i* sirve para indicar que se va a proceder a la instalación de un paquete, y el parámetro *-S* sirve para indicar a npm que guarde Express en los archivos de configuración, de forma que si en algún momento, desde otro dispositivo, realizamos *npm init*, se instale express de forma automática.

express

- Nodemon

El siguiente paquete de npm que vamos a instalar es Nodemon. Este se va a encargar de reiniciar nuestros endpoints cada vez que realicemos un cambio en ellos, agilizando el desarrollo de estos. Para instalaro, realizamos el mismo comando explicado anteriormente.

- Morgan

Ahora vamos a instalar el paquete que se va a encargar de proporcionarnos información, mediante logs, de todo lo que pase en nuestra aplicación, mediante el registro de peticiones y respuestas en nuestras aplicaciones Express. Para instalaro:

```
$ npm i -S morgan
```

morgan

- MongoJS y MongoDB

Como hemos explicado antes, vamos a utilizar como base de datos MongoDB. Este simple módulo nos permite conectarnos a la base de datos, realizar diferentes tipos de llamadas, como lecturas, escrituras y modificaciones. Para su instalación realizamos los siguientes comandos:

```
$ npm i -S mongodb
$ npm i -S mongojs
```

mongojs

- Node-Fetch

Como se explicará más adelante, nuestro sistema dispone de un GateWay que se encarga de conectarse a los diferentes endpoints y recibir datos de estos. Para ello requerimos de un módulo que nos permita conectarnos a estos, y hacerles las diferentes llamadas get, post, update y delete. Para instalar este paquete realizamos los mismos pasos que en los anteriores:

```
> npm i node-fetch
```

node-fetch

- Url

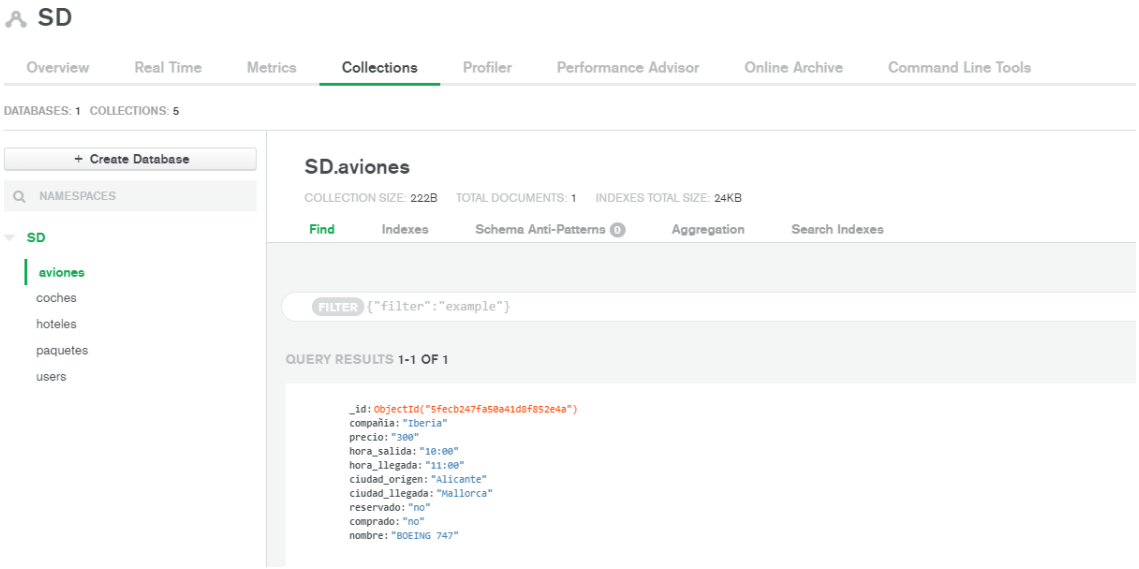
Si queremos leer los parámetros que nos pasen los usuarios por parámetro necesitamos este módulo, que nos permite extraer la información pasada. Para instalarlo, ejecutamos el siguiente comando y ya lo tenemos:

```
> npm i url
```

node-url

- Base de Datos

En los laboratorios de la EPS no viene instalado por defecto MongoDB como base de datos, por lo que, para poder usar una base de datos se ha accedido a MongoDB Atlas y se ha creado un cluster llamado SD, con una base de datos llamada SD, con 5 colecciones: aviones, coches, hoteles, paquetes y users:



➢ Servicios/End-Points

Para la implementación de la práctica se han implementado varios End-Points que definen la arquitectura de la práctica. De esta forma, se ha implementado un End-Point para los aviones, para los hoteles, para los coches y para la reserva de paquetes.

Los 3 End-Points principales, Coches, Aviones y Hoteles, disponen de una colección en la base de datos en la nube, por lo que cada uno de ellos se conectará a una de estas colecciones para consultar/modificar datos. Además, el End-Point que se encarga de crear, consultar y reservar paquetes, a su vez se conecta a estas bases de datos, con el objetivo de crear estos. La definición gráfica de estos se muestra en la arquitectura conceptual.

Por último, se ha creado un GateWay encargado de conectarse a los anteriores End-Points usando la herramienta fetch.

La siguiente tabla muestra los diferentes servicios que se han creado, con un breve resumen de cada uno de ellos:

Servicio	Tipo	Breve descripción
Reserva de Coches	WS tipo REST	Permite gestionar las reservas de coches.
Reserva de Hoteles	WS tipo REST	Permite gestionar las reservas de los hoteles
Reserva de Aviones	WS tipo REST	Permite gestionar las reservas de aviones
Reserva de Paquetes	WS tipo REST	Permite gestionar los paquetes
GateWay	WS tipo REST	Conecta con el resto de servicios.

A continuación, se pasan a detallar los métodos http de cada servicio:

- ### API-REST-GW

La siguiente tabla resume los diferentes tipos de llamadas que se pueden realizar al GateWay:

Verbo HTTP	Ruta	Descripción
GET	/api/paquetes	Obtiene todos los paquetes creados hasta la fecha.
GET	/api/{colecciones}	Obtenemos todos los elementos de la tabla {coleccion}. Estas colecciones pueden ser coches, aviones o hoteles.
GET	/api/{colecciones}/{id}	Obtenemos el elemento indicado en {id} de la tabla {coleccion}. Para cada colección, el {id} ha de ser el nombre del objeto a buscar.

Verbo HTTP	Ruta	Descripción
GET	/api/{colecciones}/id/{id}	Obtiene el elemento de la colección indicada, pero a partir de su ID.
POST	/api/paquetes	Crea un paquete a través de los nombres de las ciudades (de coche, avión y hotel) pasados en la URL de la llamada. Si ya existe un paquete con los objetos seleccionados, devuelve error.
POST	/api/paquetes/reservar/{id}	Se encarga de reservar un paquete a partir de su {id}. Si el paquete ya está reservado, devuelve error.
POST	/api/{colecciones}	Este método se encarga de llamar al método post de la {colecciones} pasada y crear nuevos objetos. Se pueden crear coches, aviones y hoteles.
POST	/login	Este método es el encargado de realizar el proceso de iniciar sesión a los usuarios. Si los datos introducidos son correctos, devuelve el token creado y lo guarda en la base de datos del usuario.
POST	/register	Este método es el encargado de crear un usuario en la base de datos. Se encarga de comprobar que el email introducido no exista, y guarda el usuario en la BD.
PUT	/api/{colecciones}/{id}	Se encarga de modificar la colección pasada en la url (coches, aviones, hoteles). El {id} ha de coincidir con el nombre del objeto a modificar.
DELETE	/api/{colecciones}/{id}	Elimina el elemento {id} de la colección pasada. A diferencia del resto de métodos, este método requiere del ID del producto a eliminar.
DELETE	/api/paquetes/{id}	Elimina el paquete cuyo id coincide con el pasado en {id}.

- API-REST-Coches/API-REST-Aviones/API-REST-Hoteles

Para la explicación de las rutas de estos 3 servicios se va a realizar la tabla de uno de ellos, coches, ya que son todos exactamente iguales y no varían sus métodos. La tabla:

Verbo HTTP	Ruta	Descripción
GET	/api/{colecciones}	Obtiene todos los coches creados hasta la fecha.
GET	/api/{colecciones}/{nombre}	Obtenemos el coche cuyo nombre es {nombre}.

Verbo HTTP	Ruta	Descripción
GET	/api/{colecciones}/id/{id}	Obtenemos el elemento indicado en {id} de la tabla coches. El {id} ha de ser el id del coche a buscar.
POST	/api/{colecciones}	Crea un nuevo coche.
PUT	/api/{colecciones}/{nombre}	Modifica el coche cuyo nombre coincide con {nombre}
DELETE	/api/{colecciones}/{id}	Elimina el coche cuyo id coincide con {id}

- API-REST-Paquetes

Verbo HTTP	Ruta	Descripción
GET	/api/paquetes	Obtiene todos los paquetes creados hasta la fecha.
POST	/api/paquetes	Crea un paquete a través de los nombres de las ciudades (de coche, avión y hotel) pasados en la URL de la llamada. Si ya existe un paquete con los objetos seleccionados, devuelve error.
POST	/api/paquetes/reservar/{id}	Se encarga de reservar un paquete a partir de su {id}. Si el paquete ya está reservado, devuelve error.
DELETE	/api/paquetes/{id}	Elimina el paquete cuyo id coincide con el pasado en {id}.

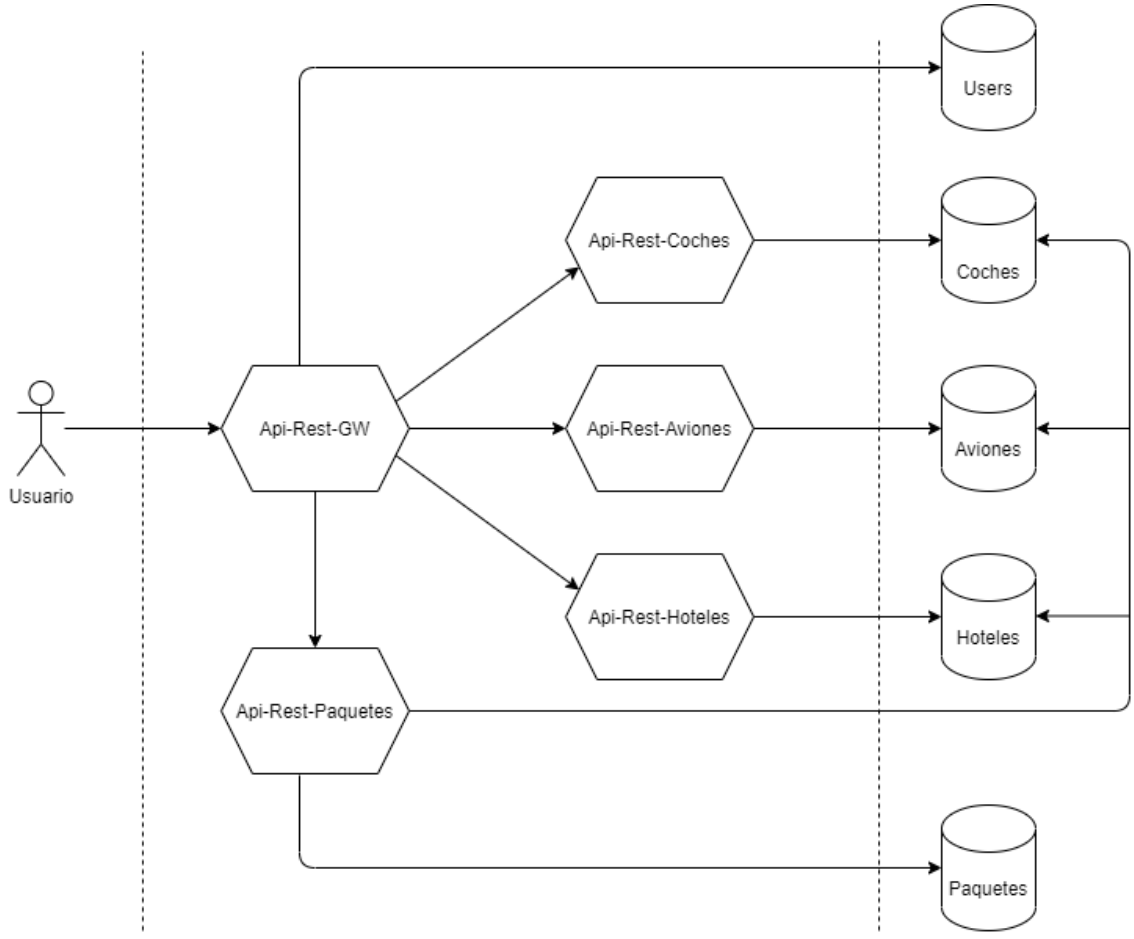
En cuanto al uso de mensajes, los diferentes servicios se comunican mediante mensajes tipo JSON. Este, al ser utilizado por el lenguaje JavaScript, es una potente herramienta que nos permite comunicar los diferentes end-points.

➢ Arquitectura conceptual

En cuanto a la arquitectura conceptual, se ha desarrollado como se ha descrito anteriormente. Se dispone de un Api-Rest-GW, el cual es el encargado de comunicarse con el resto de servicios, mediante llamadas GET, POST, DELETE y UPDATE a estas. Así, a su vez, cada End-Point se comunica con su base de datos almacenando, editando y eliminando los diferentes objetos.

De esta forma se consigue una independencia respecto a los diferentes servicios, de forma que, si uno de estos cae, el resto va a poder seguir resolviendo peticiones. En este caso la BD se encuentra toda en el mismo Cluster (debido a los problemas en los laboratorio se ha optado por esta opción), pero en un diseño real, se podría instalar cada base de datos en un cluster diferente, creando así una protección ante caídas parciales del sistema.

La arquitectura conceptual se define en el siguiente esquema:

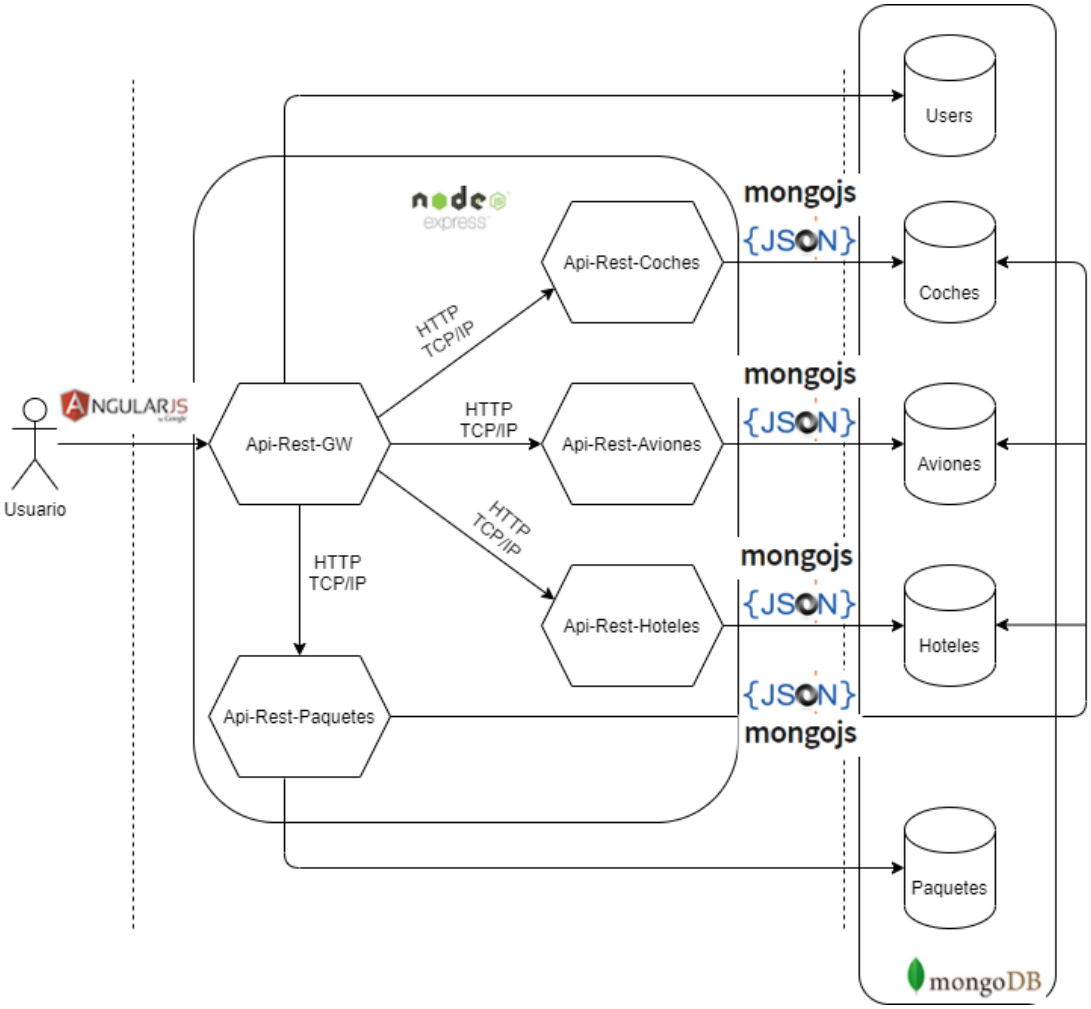


Como se puede observar en el diagrama, no hay un módulo encargado de los usuarios, puesto que este se encuentra directamente en el GateWay. Es decir, será este el que se encargue de toda la lógica presente en los usuarios.

Además, no hay ningún servicio encargado de las transacciones. Esto se ha implementado en el Api-Rest-Paquetes, donde se procede a la reserva de los diferentes objetos presentes en las bases de datos de Hoteles, Aviones y Coches. También comentar que este paquete, a parte de su propio Id, contiene los Ids de los diferentes objetos presentes en el.

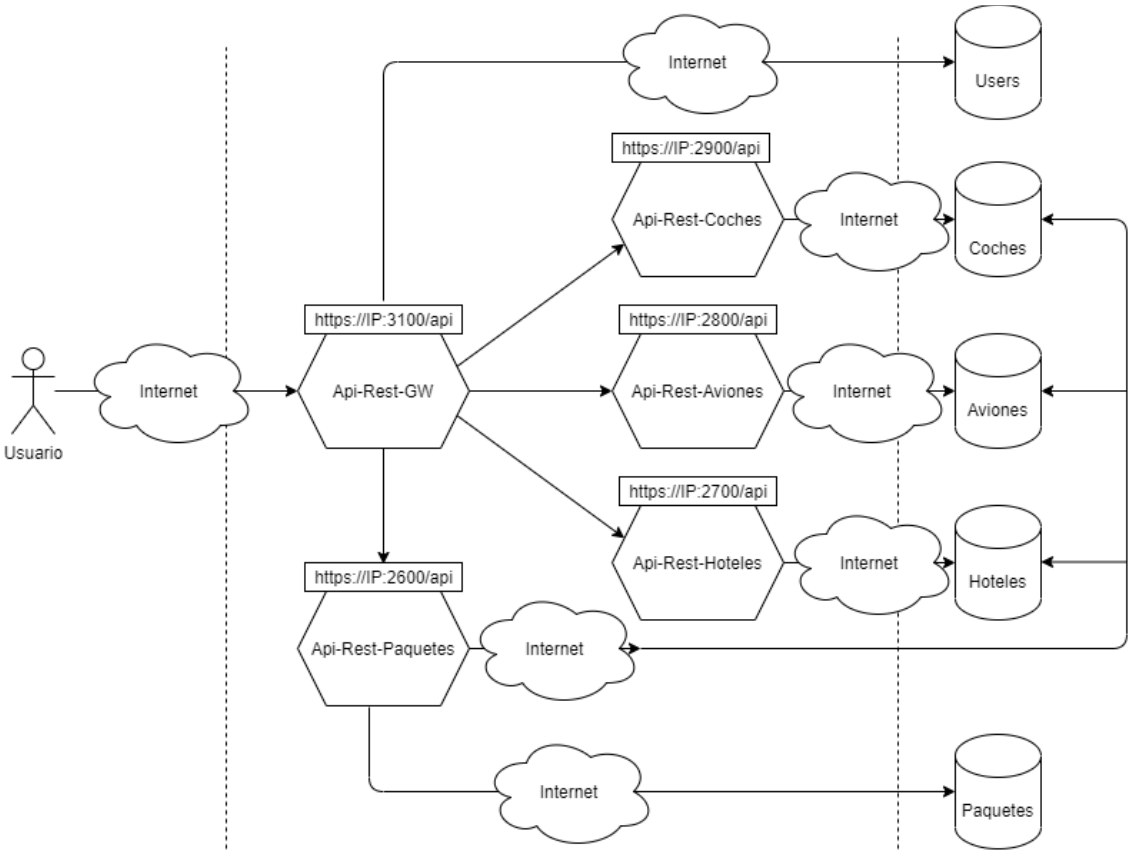
➢ Arquitectura técnica

En la introducción ya se ha nombrado sobre la arquitectura técnica y las diferentes tecnologías utilizadas para la realización de la práctica, pero en la siguiente imagen aparece, para cada parte del sistema, que tecnología ha sido utilizada:



➢ Arquitectura de despliegue

La arquitectura de despliegue de nuestro sistema es la siguiente:



Como se puede ver, cada servicio dentro de la red dispone de un puerto propio que lo distingue de los demás, además de una IP. Esta no se especifica puesto que, hasta el día de despliegue, donde se asigne la IP del equipo donde se encuentra el servicio, no se puede disponer de IP.

Además, como ya sabemos, la base de datos está en la nube, por lo que para acceder a ella debemos acceder a través de Internet. A esta se accede a través de una cadena de conexión que se especifica en el código.

➢ Conclusiones

Se ha desarrollado el sistema de agencias utilizando la pila MEAN, desarrollando cada uno de los sistemas de forma independiente, utilizando su IP, su puerto y su base de datos correspondiente. De esta forma se ha conseguido un sistema escalable, tolerable a fallos y con seguridad basada en el uso de certificados y https.

➢ Referencias

- MongoJS

  https://www.npmjs.com/package/mongojs

- Node-Fetch

  https://www.npmjs.com/package/node-fetch

- Express

  https://www.npmjs.com/package/express

- URL

  https://www.npmjs.com/package/url

- Nodemon

  https://www.npmjs.com/package/nodemon

- Morgan

  https://www.npmjs.com/package/morgan

- SSL y Node

  https://www.sitepoint.com/how-to-use-ssltls-with-node-js/

- Uso de Web Tokens

  https://www.npmjs.com/package/jwt-simple

  https://www.npmjs.com/package/bcrypt

➢ Entrega

- Proyecto

  El link del proyecto es el siguiente: https://github.com/Baidal/SDP3

  Indicar que el desarrollo de este se hizo en Bitbucket a través de varios proyectos, pero esta plataforma no deja agregar a usuarios a proyectos, por lo que tuve que coger el proyecto y migrarlo a github. Es esta la razón de que, en esta plataforma, no hayan commits.

- Base de Datos

  Usuario: pmacia Contraseña: Pmacia2020

String para conectarse desde el Shell:

```
mongo "mongodb+srv://sd.7nmy6.mongodb.net/SD" --username <pmacia>
```

String para conectarse desde NodeJS:

```
mongodb+srv://pmacia:Pmacia2020@sd.7nmy6.mongodb.net/<dbname>?retryWrites=true&w=majority
```