ApplicationManual.md

Manual del Application Server

Introducción

Este proyecto consiste de un servidor Web HTTP accesible a través de una REST API para la aplicación **FIUBER**. **FIUBER** es una aplicación intencionada para conectar a los pasajeros con los conductores de vehículos que ofrecen servicio de transporte particular. Esta aplicación permite a los potenciales pasajeros: obtener estimaciones de costo un viaje antes de realizarlo, elegir al conductor que desean, solicitar al chofer y una vez que terminan el viaje realizar el pago utilizando cualquiera de los medios de pago disponibles.

Este sistema se basa en un diseño de 3 capas que permite el funcionamiento de la aplicación:

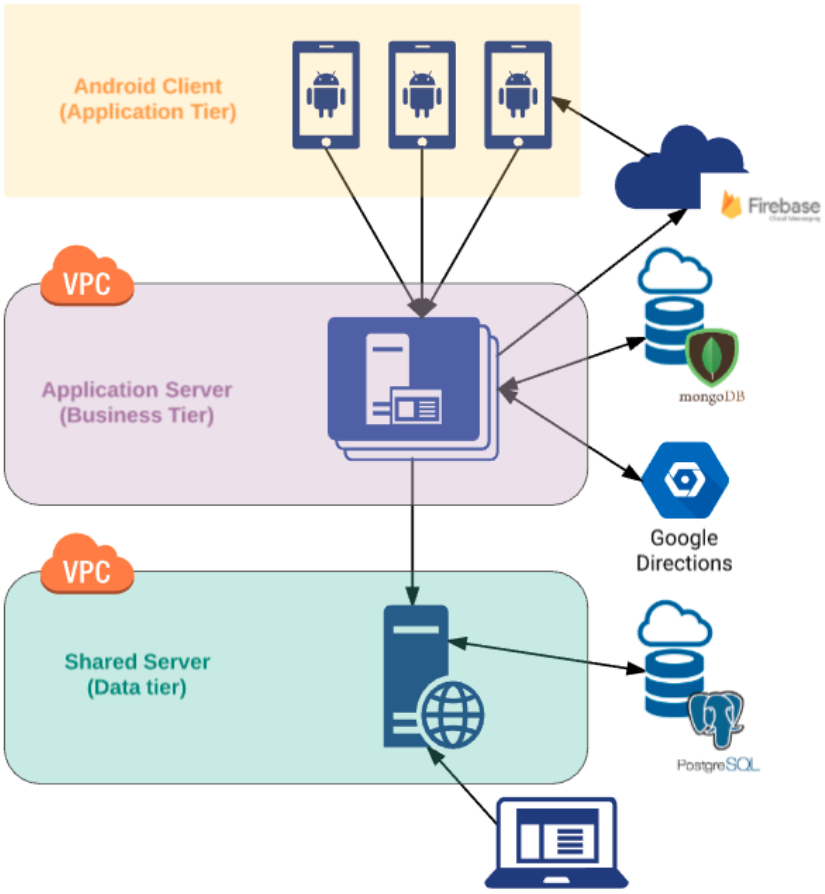
- Cliente
- **App Server**
- Shared Server

Este proyecto provee una implementación para la capa de App Server del sistema. En el archivo *simpleAPI.yml* puede encontrarse documentación detallada sobre la interfaz provista por el servidor para comunicarse con una aplicación cliente.

Relación con el cliente y el *Shared Server*

Para implementar esta aplicación se utilizó una arquitectura de 3 capas (3-Tier), donde el *App Server* representa la capa lógica o de negocios. La capa de datos es provista por el Shared Server y es allí donde se almacenan los datos de los usuarios de la aplicación, tanto conductores como pasajeros, los viajes y los servidores activos. La aplicación está pensada para permitir la coexistencia de múltiples *App Servers* que utilizan al *Shared Server* como servicio web para almacenar datos y como punto de acceso a la API de pagos, que se provee de forma externa.

Diagrama general de capas:



Por lo tanto, las funcionalidades implementadas estaban en parte limitadas por la API que se provee desde el *Shared Server*. A continuación se detallan la estructura del código del *App Server* así como los aspectos clave de la arquitectura y diseño.

Estructura de paquetes

La división del código en distintos módulos y paquetes se resume a continuación. Todo el código relevante a las distintas funcionalidades de la aplicación se encuentran en el directorio *src/main*. El código de las pruebas unitarias y de integración se encuentra en *src/test*.

- *src/main/myApp.py* : aquí se encuentran definidos todos los *endpoints* definidos para la REST API y su vinculación a los *recursos* designados para cada uno.
- *src/main/resources* : es el directorio que contiene todos los controladores de los *recursos*, encargados de manejar los requests realizados a los endpoints definidos en *myApp.py*. Los controladores de de endpoints similares están definidos en un mismo archivo.
- *src/main/com* : aquí residen los archivos que implementan módulos auxiliares. Éstos manejan tanto la comunicación con otras aplicaciones (i.e. shared-server o Google API), como el aislamiento de operaciones comunes a varios endpoints (e.g. validación de tokens del application server).
    - *ServerRequest.py* es el módulo que permite realizar requests al *Shared Server*.
    - *NotificationManager.py* es el módulo que permite conectarse con el servicio de *Firebase* para enviar notificaciones a los clientes.
    - *TokenGenerator.py* es el módulo encargado de la validación de los *tokens* de seguridad utilizados en la comunicación entre las tres capas.
    - *ResponseMaker.py* es un módulo que encapsula la generación de respuestas de la API: serialización de json y respuestas estándar de error.
    - *Distances.py* encapsula la lógica de cálculo de distancias entre coordenadas dadas en formato latitud-longitud.
- *src/main/mongodb* :

    - *MongoController.py* : es el módulo encargado de la conexión con la base de datos remota o local, si la anterior no estuviera definida.

- *src/main/mode/* : en este directorio se encuentran los archivos encargados de modelar las estructuras de las entidades conceptuales importantes del sistema para garantizar la compatibilidad entre los datos intercambiados entre el *Cliente* y el *Shared Server*.

    - *TripStates.py* : en este módulo se definen todos los estados posibles de un viaje. Permite mappear la información de un viaje desde el formato recibido desde el cliente al formato de viajes compatible con el *Shared Server*.

    - *Client.py* : en este módulo se definen todos los estados posibles de un usuario (tanto para pasajero como para chofer). Permite mappear los datos de los usuarios a un formato compatible con el utilizado por la base de datos local.

Arquitectura y diseño

Todo el diseño del *App Server* gira en torno a las dos entidades principales dentro del dominio de negocios: los *Users* (Usuarios) y los *Trips* (Viajes). Ambas entidades son modeladas dentro del *App Server* como diccionarios, con una sierie de campos requeridos y otras opcionales. Las razones detrás de esta decisión radican en 1) la facilidad del manejo de estos objetos en python y 2) la cercanía de los mismos al formato de comunicación con el *Shared Server*.

Users

Los *Users* dentro de la aplicación representan a una persona que utiliza un cliente en su teléfono. Así pues distinguimos dos roles dentro de los *Users*: los *Passengers* (pasajeros) y los *Drivers* (conductores). Ambos tienen la misma información básica (i.e. datos personales), pero se diferencian en algunos atributos y en cómo se relacionan con el resto de las entidades. El esquema de datos de un *User* tal y como se lo maneja en la aplicación y se lo guarda en la base de datos es el siguiente.

```
"user": {
  "_id": 7,
  "_ref": "74f3270c-f453-4fae-8eda-ac7f7a04a9ff",
  "applicationOwner": "1",
  "balance": [],
  "birthdate": "29-2-1971",
  "cars": [],
  "coord": {
    "lat": 34.161,
    "lng": 58.96
  },
  "country": "France",
  "email": "cosmefulanito3@thebest.com",
  "images": [
    "No tengo imagen"
  ],
  "name": "Cosme",
  "online": true,
  "rating": {
    "rate": 4,
    "rateCount": 2
  },
  "state": 11,
  "surname": "Fulanito",
  "tripId": "",
  "type": "driver",
  "username": "cosme_fulanito3"
}
```

Trips

Los *Trips* representan un viaje en cualquiera de sus estadíos: desde un viaje que se ha propuesto pero todavía carece de conductor, hasta uno que ha finalizado pero tiene pago pendiente. Al igual que los *Users* los *Trips* se modelan internamente como un diccionario y se guardan como tales en la base de datos. Todos los *Trips* tienen asociada necesariamente un objeto de tipo *Directions* que indica las direcciones del camino (i.e. puntos de origen y destino, tiempo estimado, distancia estimada y una serie de waypoints que marcan el camino paso a paso).

El estado de un viaje se codifica en un campo especial dentro de su estructura, y éste a su vez indica la validez y o necesidad de incluir los otros campos. La estructura completa de un viaje es la siguiente:

```
{
    "_id": "9b568ad2-d5f6-11e7-b2af-be5f70b87d11",
    "driverId": 4,
    "cost": {
        "currency": "ARS",
        "value": 70.95
    },
    "state": "finished_rated",
    "directions": {
        "duration": 740,
        "origin_name": "Ramón Franco 4142-4198, Remedios de Escalada, Buenos Aires, Argentina",
        "origin": {
            "lng": -58.4110197,
            "lat": -34.7313204
        },
        "path": [
            {
                "distance": 33,
                "duration": 6,
                "coords": {
                    "lng": -58.41132769999999,
                    "lat": -34.7314771
                }
            }
        ],
        "destination_name": "Yapeyú 801-899, B1828BBK Banfield, Buenos Aires, Argentina",
        "status": "OK",
        "destination": {
            "lng": -58.3801311,
            "lat": -34.7473679
        },
        "distance": 4300
    },
    "passengerId": 2,
    "time_start": "2017-11-30T17:48:55.104382",
    "time_start_waiting": "2017-11-30T17:48:55.108341",
    "real_route": [
        {
            "lng": -58.4110484,
            "timestamp": "2017-11-30T17:48:58.045657",
            "lat": -34.7316897
        }
    ],
    "time_finish": "2017-11-30T17:49:06.580658"
}
```


Diagramas de estados

Cada entidad tiene su propio set de estados, que indican las acciones que pueden efectuar y el punto en el que se encuentran dentro del flujo de eventos principal.

Los estados de un *Passenger* son los siguientes:

- *Idle*: el pasajero está recien logueado en la aplicación, puede ver conductores cercanos (GET a /users), averiguar por el costo de un viaje (POST /directions) o bien proponer un viaje nuevo y esperar por conductores (POST /trips)
- *Waiting Accept*: un pasajero que acaba de postear un *Trip* está en este estado, y representa la espera hasta que un *Driver* decida tomar el viaje. En cualquier momento se puede cancelar el viaje (POST a /trip/tripId/action, *'cancel'*)
- *Selecting Driver*: cuando un *Driver* acepte el *Trip* propuesto por un *Passenger*, éste último entrará en este estado. En este momento podrá ver la información del *Driver* (GET a /users/userId) y decidir si le parece bien (POS a /trip/tripId/action, *'confirm'*) o bien prefiere rechazarlo y esperar a otro *Driver* (POST a /trip/tripId/action, *'reject'*)
- *Waiting Driver*: el *Passenger* que haya aceptado a un *Driver* estará en este estado, e indica que está esperando a que lo pasen a buscar. Durante este estado se puede todavía cancelar el viaje (de la misma manera que se describió en *Waiting Accept*), o bien iniciarlo mediante un *two-way handshake* (POST a /trip/tripId/action, 'start').
- *Waiting Start*: para iniciar un viaje se necesita que tanto el *Driver* como el *Passenger* expresen que quieren iniciar el viaje a través del endpoint apropiado (/trip/tripId/action). Si fuera el *Passenger* el primero en querer iniciar el viaje, entra en este estado en el cual estará esperando a que el *Driver* también lo inicie. Esta es el último momento en que se puede cacelar un viaje sin que se realicen transferencias monetarias.
- *Travelling*: cuando el viaje inició tanto *Passengers* como *Drivers* entrarán en este estado, que representa el viaje en sí. Se trackearán las posiciones de ambos *Users* para definir el camino real del *Trip*. La única acción permitida en este

estado es la de finalizar el viaje (POST a /trip/tripId/action, *'finish'*). Al igual que en el mecanismo de inicio del viaje, se hace a través de un *two-way handshake*, teniendo ambos *Users* que estar de acuerdo en la finalización del *Trip*.

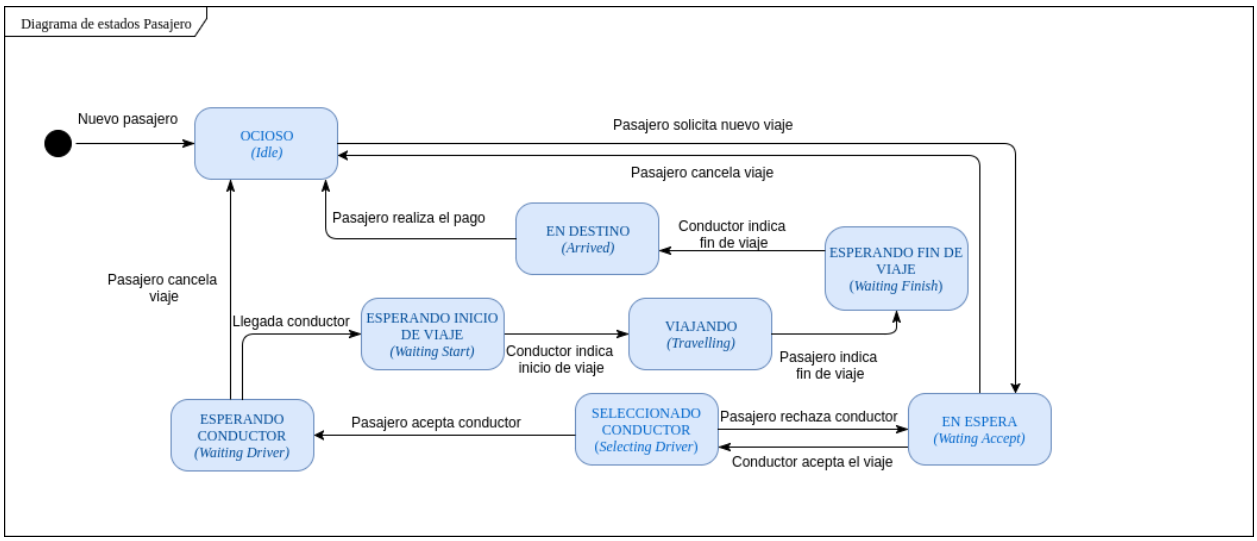
- *Waiting Finish*: es el análogo a *Waiting Start* pero para la finalización del viaje: se da sólo si el *Passenger* es el primero en terminar el *Trip* y debe esperar a que el *Driver* haga lo propio.
- *Arrived*: este es el estado final del *Passenger* en un viaje. Representa que el viaje terminó y que debe efectuar el pago del mismo. En este momento se permite realizar una reseña del conductor (POST a /trip/tripId/action, *'rate'*). El ciclo de vida del viaje termina cuando el *Passenger* efectúa el pago (POST a /trip/tripId/action, *'pay'*), volviendo al estado *Idle*.

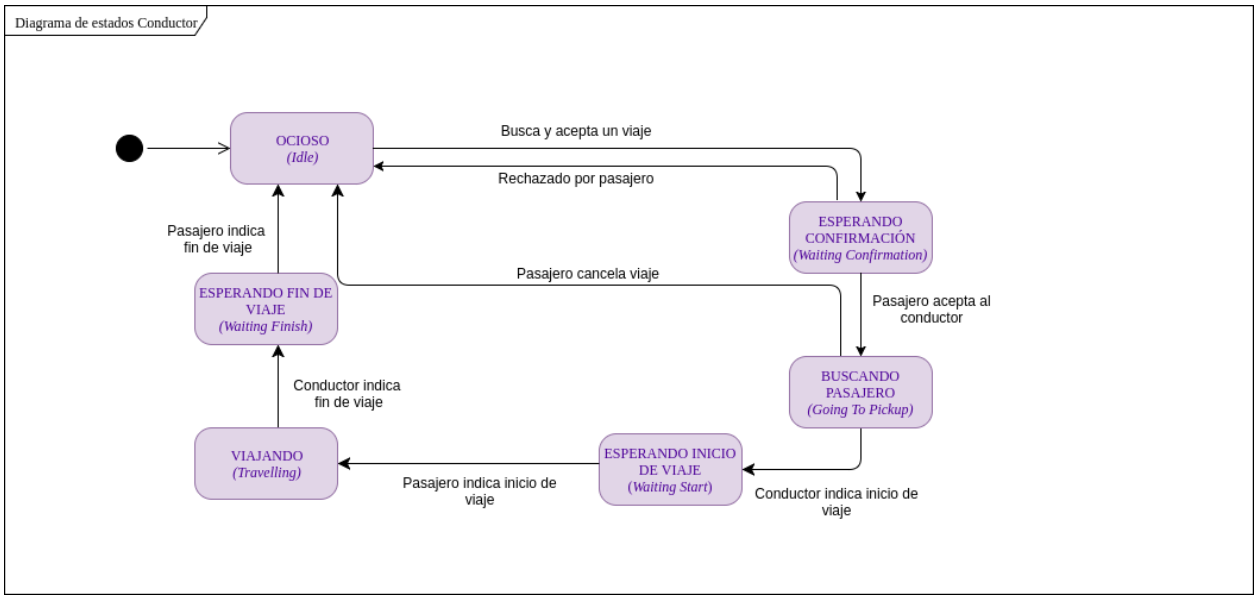
Los estados de un *Driver* son los siguientes:

- *Idle*: un *Driver* en este estado no está asociado a ningún viaje. Puede ver otros *Users* cercanos (GET /users), y obtener una lista de *Trips* propuestos que requieren de un conductor (GET /trips). Viendo la lista de viajes puede elegir tomar uno (POST /trips/tripId/action, *'accept'*).
- *Waiting Confirmation*: un *Driver* que decida aceptar un *Trip* todavía no estará asociado al mismo, debe esperar a que el *Passenger* que propuso el viaje le de su aprovación (ver estado *Selecting Driver* del pasajero). Este estado representa la espera del *Driver* a ser confirmado para tomar el viaje.
- *Going To Pickup*: cuando el *Driver* es confirmado, debe ir al punto de encuentro establecido al proponerse el *Trip*. Una vez ahí, deberá comenzar el viaje mediante el *two-way handshake* explicado en los estados del *Passenger* (POST a /trip/tripId/action, *'start'*.
- *Waiting Start*: si el *Driver* fue el primero en querer iniciar el viaje, entra en este estado en el cual estará esperando a que el *Passenger* también lo inicie.
- *Travelling*: cuando el viaje inició tanto *Passengers* como *Drivers* entrarán en este estado, que representa el viaje en sí. Ver el estado homónimo del *Passenger*.
- *Waiting Finish*: es el análogo a *Waiting Finish* pero si el *Driver* es el primero en terminar el viaje.

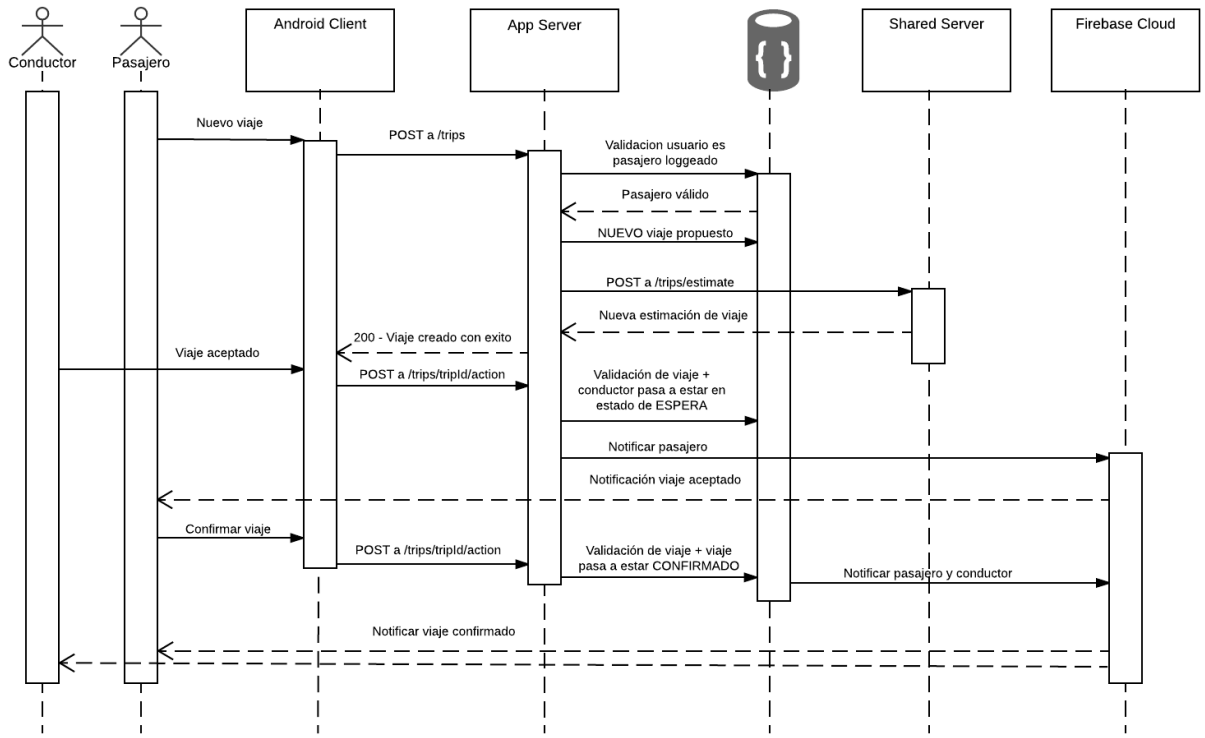
Los *Trips* tienen una referencia tanto al *Passenger* como al *Driver* (cuando lo haya), y una serie de estados que acompaña a los estados de los *Users*. Los estados posibles de un *Trip* son los siguientes.

- *Proposed*: corresponde a un *Trip* recién creado como producto de la acción de un *Passenger* (POST a /trips). Este estado es el único que tiene una referencia a *Driver* inválida, puesto que todavía no ha sido tomado por ningún conductor.
- *Accepted*: un *Trip* que haya sido tomado por un *Driver* pero aún no esté confirmado por el *Passenger* está en este estado.
- *Confirmed*: un *Trip* pasa a este estado cuando el *Passenger* confirma al *Driver*. Ambos *Users* pueden ahora iniciar el *two-way handshake* para dar inicio al *Trip*.
- *Started*, *Passenger Started* y *Driver Started*: son los tres estadíos del *two-way handshake* para iniciar un viaje. *Passenger Started* corresponde al estado en que el pasajero indicó que quiere comenzar el viaje, pero el conductor todavía no. El caso contrario es el del estado *Driver Started*. El estado *Started* corresponde a un viaje ya comenzado, y que se está llevando a cabo en este momento.
- *Finished*, *Passenger Finished* y *Driver Finished*: son los estados análogos al comienzo del viaje, pero para la finalización del mismo.
- *Finished Rated*: este es un estado de post-finalización que indica que el *Passenger* ha decidido dejar un rating al *Driver*. Sólo se puede puntuar al conductor si el viaje está en estado *Finished* y para evitar que haya más de un rating por *Trip* una vez hecha la puntuación el mismo pasa a este estado. Este estado no es parte del flujo de estados obligatorio de un *Trip*.
- *Completed* (a.k.a. *Payed*): este es un estado tácito puesto que los viajes finazliados y pagos no se almacenan el la base de datos propia del *Application Server*, sino que se realiza la alta apropiada en el *Shared Server*. Así, lógicamente todos los *Trips* que estén guardados en el *Shared Server* tendrán implícitamente este estado.





Las posibles transiciones entre los estados puede resumirse en el siguiente diagrama:

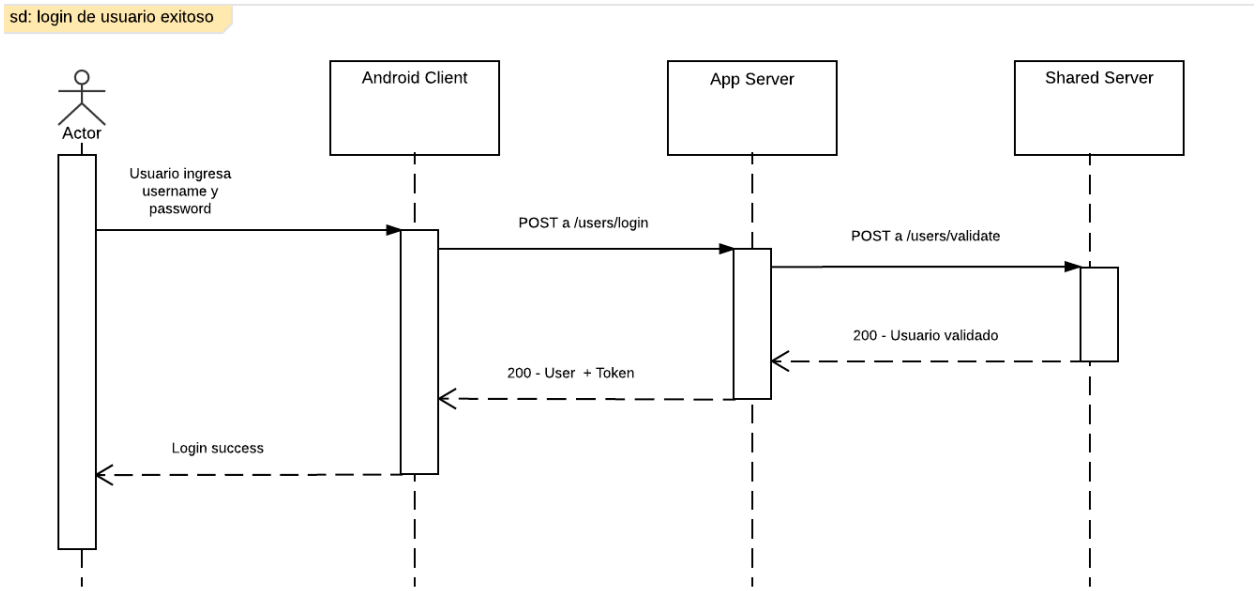


Login y Logout de *Users*

Todos los *Users* en la base de datos local tienen un flag que indica si están logueados o no. Esto permite mantener registro del estado de los usuarios incluso si se desconectan. Los pasos a seguir en caso del login de un *User* son:

1. Verificar las credenciales del usuario con el *Shared Server*, y obtener los datos del mismo.
2. Buscar el usuario en la base de datos y comparar los *_ref*, actualizar los campos en caso de ser necesario.
3. Si el usuario existía, recuperar su estado, sino ponerlo en estado *Idle*.
4. Marcar al user como *online*

El proceso de logout sólo borra al usuario de la base de datos local en caso de que el estado del mismo sea *Idle*, en caso contrario sólo se lo marca como *offline*.



Dependencias y herramientas

Esta sección está destinada a mencionar las herramientas, librerías y APIs más relevantes utilizadas en este proyecto. Destacamos que esta no pretende ser una descripción de todas las librerías utilizadas, sino una breve mención de aquellas más relevantes para la funcionalidad de este proyecto.

- **GUnicorn** + **Flask**

  Este proyecto consiste en un servidor HTTP basado en *GUnicorn*. Para confeccionar la REST API que permite la comunicación con el servidor, se utilizó un framework para Python llamado *Flask*.

- **Google Maps API**

  Se utilizó *Google Maps API* para obtener recorridos de viaje para los vehículos a partir del origen y el destino deseado. A partir de la información provista por Google Directions el usuario de la aplicación puede realizar un request al endpoint /directions para obtener tanto el recorrido como la estimación del costo de viaje.

- **Firebase**

  Se utilizó firebase como servicio web para poder proveer un servicio de mensajería (chat) entre el viajero y el conductor. Adicionalmente se utilizó Firebase para enviar notificaciones de eventos (como mensajes nuevos o la cancelación de un viaje) a los usuarios de la aplicación.

- **Docker**

  Para asegurar la flexibilidad y garantizar la compatibilidad en distintas plataformas, se utilizó Docker + Docker Compose. Existen 2 containers principales, uno para la base de datos y otro para la aplicación. Los archivos de configuración de docker pueden encontrarse en el directorio raíz del proyecto. En estos archivos se definen los nombres, puertos y propiedades principales de los containers, así como también las variables de entorno y dependencias de cada uno:

  - *Dockerfile*
  - *docker-compose.yml*

- **MongoDB**

  Mongo Database es el sistema de base de datos (NoSQL) seleccionado para este proyecto. Existe una base de test definida en el archivo *docker-compose.yml* que será la utilizada por el servidor como base default. El usuario puede optar por mantener esta base o definir la propia en el archivo citado. De no encontrarse una definición se utiliza el servicio de Mongo localmente.

Bugs conocidos y puntos a mejorar

Debido a restricciones temporales, existen algunos aspectos de este proyecto que requieren ser mejorados o concluidos. A continuación, realizamos una breve descripción de las falencias y bugs detectados hasta el momento:

- **Controles de concurrencia en la base de datos:** actualmente no existen controles de concurrencia sobre la base de datos. Como se mencionó anteriormente, uno de los objetivos de esta aplicación es que puedan coexistir varias instancias de *App Server* trabajando sobre una misma base hosteada en la web (salvo que por algún motivo se trabaje sobre una base de datos instanciada en forma local). Debido a que en MONGODB no existe el concepto de transacción para garantizar la integridad de los datos, estos controles deben realizarse en forma manual. En nuestra implementación, estos controles no están implementados, pudiendo ocasionarse una *race condition* si varios usuarios quisieran realizar modificaciones sobre la base al mismo tiempo.