

**TFG del Grado en Ingeniería  
Informática**

**Aplicación de ayuda para  
viajeros**

Presentado por Andrés Rodríguez Rosales  
en Universidad de Burgos — 7 de julio  
de 2021

Tutor: Jesús Manuel Maudes Raedo





D. nombre tutor, profesor del departamento de nombre departamento, área de nombre área.

Expone:

Que el alumno D. Andrés Rodríguez Rosales, con DNI 71314888F, ha realizado el Trabajo final de Grado en Ingeniería Informática titulado Aplicación de ayuda para viajeros.

Y que dicho trabajo ha sido realizado por el alumno bajo la dirección del que suscribe, en virtud de lo cual se autoriza su presentación y defensa.

En Burgos, 7 de julio de 2021

y descomentar lo siguiente Vº. Bº. del Tutor:

D. nombre tutor





## **Resumen**

En este proyecto se va a desarrollar una aplicación web que ayude a las personas que quieran viajar de un sitio a otro a elegir la mejor opción de hacerlo. Para ello se obtiene información real a través de diferentes API-REST relacionadas con los viajes en tren o bus, avión o blablacar.

La aplicación permite ver las opciones de viaje directo entre las ubicaciones, además de explorar posibles viajes con transbordos.

Se usa una base de datos relacional para almacenar los datos. A parte de los datos típicos relacionados con el mundo de los viajes almacenamos una serie de Nodos, que pueden ser aeropuertos o estaciones de bus o tren.

## **Descriptores**

Aplicación web, API-REST, Grafos, Viajes, Nodos, Blablacar, Trainline, Skyscanner, Google Maps . . .

## **Abstract**

In this project, a web application is developed to help people who want to travel from one place to another to choose the best option to do so. For this, real information is obtained through different API-REST related to travel by train, bus, plane or blablacar.

The application allows you to view direct travel options between locations, as well as explore possible journeys with transfers.

A relational database is used to store the data. Apart from the typical-related data, we store a series of Nodes, which can be airports or bus or train stations.

## **Keywords**

Web application, API-REST, Graphs, Travel, Nodes, Blablacar, Trainline, Skyscanner, Google Maps



---

# Índice general

---

<b>Índice general</b>	<b>III</b>
<b>Índice de figuras</b>	<b>V</b>
<b>Índice de tablas</b>	<b>VI</b>
<b>Introducción</b>	<b>1</b>
1.1. Material adjunto . . . . .	2
<b>Objetivos del proyecto</b>	<b>3</b>
2.1. Objetivos generales . . . . .	3
2.2. Objetivos técnicos . . . . .	3
2.3. Objetivos personales . . . . .	4
<b>Conceptos teóricos</b>	<b>5</b>
3.1. API REST . . . . .	5
3.2. Protocolo HTTP . . . . .	7
3.3. Algoritmo de Dijkstra . . . . .	7
3.4. Material adjunto . . . . .	8
<b>Técnicas y herramientas</b>	<b>9</b>
4.1. Técnicas de extracción de datos . . . . .	9
4.2. Herramientas utilizadas para el desarrollo web . . . . .	10
<b>Aspectos relevantes del desarrollo del proyecto</b>	<b>13</b>
5.1. Riesgos y retos . . . . .	13
5.2. Formación . . . . .	14
5.3. Problemas encontrados y soluciones aportadas . . . . .	15

<b>Trabajos relacionados</b>	<b>31</b>
6.1. Proyectos . . . . .	31
<b>Conclusiones y líneas de trabajo futuras</b>	<b>35</b>
7.1. Conclusiones . . . . .	35
7.2. Líneas de trabajo futuras . . . . .	38
<b>Bibliografía</b>	<b>39</b>

---

## Índice de figuras

---

3.1. Funcionamiento de una API REST [30] . . . . .	5
5.1. Adición o eliminación de API-REST [31]. . . . .	21
5.2. Formulario para añadir una nueva API-REST [31]. . . . .	22
5.3. Formulario para añadir una nueva Request(petición) [31]. . . . .	24
5.4. Adición o eliminación de una Request(petición) [31]. . . . .	25
5.5. Formulario de modificación de API-REST [31]. . . . .	26
5.6. Formulario para modificar una Request [31]. . . . .	27
5.7. Estructura para realizar una petición a Blablacar [31]. . . . .	28
6.1. Viaje con transbordo en la aplicación de Trainline [31]. . . . .	32

---

# Índice de tablas

---

5.1. Tiempo en segundos de las primeras peticiones en un ejemplo de búsqueda de viajes entre Burgos y Madrid [31] . . . . .	17
--	----

---

# Introducción

---

En la antigüedad cuando las personas se desplazaban, en la mayoría de casos, de su lugar de origen para buscar una mejor calidad de vida los viajes eran costosos en cuanto a tiempo, y en muchos casos no estaban al alcance de todos. Hoy en día, las razones por las cuales viajamos son muchas como pueden ser ocio o trabajo y los viajes son de menor duración y de menor coste económico.

Además de los medios de transporte tradicionales como autobús, tren o avión han surgido otros derivados del auge de la tecnología en los últimos años como puede ser el servicio de coche compartido blablacar. Esto aumenta nuestras posibilidades en cuanto a la búsqueda de viajes.

La forma de buscar viajes actualmente es a través de internet. Por este motivo, surge el problema de la gran cantidad de información que podemos encontrar en diversas páginas.

En este proyecto se trata de facilitar al usuario la decisión de elegir el mejor medio de transporte según el precio que se desee gastar o el día en el que desee realizar el trayecto. También trata de evitar tener que visitar demasiadas páginas, o tener que emplear demasiadas horas buscando viajes en muchas páginas o aplicaciones.

La aplicación utilizará fuentes de datos de diversos medios de transporte que nos permitirán ver los trayectos reales que habrá cada día. Gracias a ello, podremos encontrar diferentes tipos de viajes con un solo click e incluso encontrar viajes con transbordos, o evitar viajes a partir de un determinado precio.

El acceso a las fuentes de datos se ha realizado a través de API-REST. Esto ha conllevado riesgos, principalmente de desactualización ya que los accesos a este tipo de servicios suelen cambiar frecuentemente, y también el riesgo de que no se pudiera acceder a estos datos. Sin embargo, finalmente se ha demostrado que el proyecto es factible y puede ser ampliado con mejoras en un futuro.

## 1.1. Material adjunto

En el proyecto como material adjunto se incluye:

- Memoria
- Anexos

Además, el proyecto se pueden ver en el repositorio de GitHub:

<https://github.com/andriu99/TFGViajes>

Se puede acceder a la aplicación web del proyecto en la siguiente url:

<https://heroku-app1435.herokuapp.com/>

Es importante saber que algunas peticiones al link anterior fallan. Esto es debido a que Heroku [19] aborta las peticiones que tardan más de 30 segundos. Cuando las peticiones que se hacen a las fuentes de datos externas son costosas, es común que no nos muestre los resultados.

---

# Objetivos del proyecto

---

A continuación, se detallan los diferentes objetivos que han motivado la realización del proyecto.

## 2.1. Objetivos generales

- Desarrollar una aplicación que permita realizar búsquedas utilizando diferentes fuentes de datos externas.
- La aplicación será capaz de buscar viajes en diferentes medios de transporte.
- Identificar fuentes de datos externas que sean útiles para la resolución de los problemas abordados por el proyecto.
- El sistema será capaz de sugerir viajes con trasbordos al usuario.
- La aplicación desarrollada sera fácil de usar.
- Reducir al máximo la dependencia de cambios en las fuentes de datos externas.

## 2.2. Objetivos técnicos

- Disminuir el tiempo que tarda la aplicación en ejecutarse siempre que sea posible y conveniente.
- Hacer que la aplicación sea fácilmente mantenible en un futuro.
- Aprendizaje del framework de back-end Django y el lenguaje de programación Python para la construcción de la aplicación.
- Profundizar en el manejo de Git como sistema de control de versiones distribuido junto con la plataforma GitHub.

- Creación de una estructura de tablas para la base de datos relacional que nos permita añadir, borrar o modificar fuentes de datos fácilmente.
- Aprendizaje de la librería Requests para realizar las diferentes peticiones a las API-REST que se usan como fuente de datos.

## **2.3. Objetivos personales**

- Adquirir conocimientos acerca de desarrollo web, como pueden ser el manejo de HTML, CSS, Javascript o el framework Django.
- Consolidar algunos conocimientos adquiridos durante la carrera como los relativos al lenguaje de programación python, el manejo de bases de datos relacionales o la implementación del Algoritmo de Dijkstra.
- Aprender a realizar peticiones HTTP de tipo GET y POST a una API-REST.
- Ampliar mis conocimientos de bases de datos.



---

## Conceptos teóricos

---

A continuación se explicarán los conceptos teóricos más relevantes.

### 3.1. API REST

Una API (interfaz de programación de aplicaciones) está formada por un conjunto de reglas que describen cómo debe hacerse la comunicación entre diferentes aplicaciones.<sup>[20]</sup>

En concreto, las API REST utilizan solicitudes HTTP para comunicarse. De esta forma son capaces de realizar las operaciones básicas sobre una base de datos como pueden ser leer, actualizar y eliminar información. Usan las solicitudes de tipo GET para obtener datos, DELETE para eliminar, PUT para actualizar y POST para crear un registro <sup>[20]</sup>.

La información de un recurso en un momento concreto se puede representar de gran cantidad de formas (texto sin formato alguno, XML, JSON, etc.).

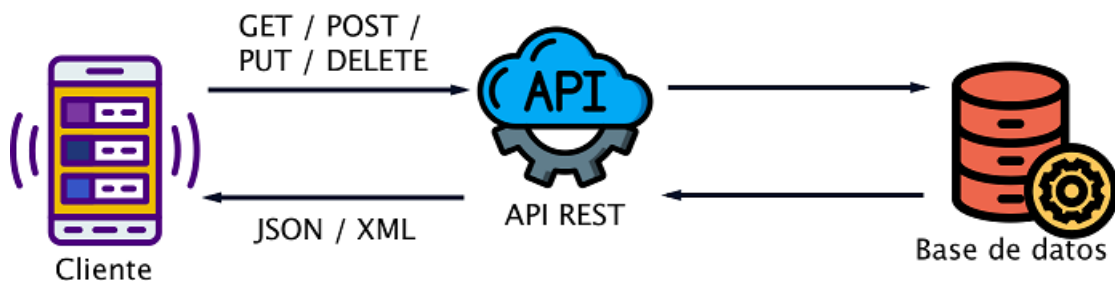


Figura 3.1: Funcionamiento de una API REST <sup>[30]</sup>

La forma que vamos a usar en nuestro caso, y también la más popular es JSON (JavaScript Object Notation) [9].

Según [34] podemos decir que históricamente las API RESTs surgieron a finales de los años 90. Siendo las empresas Amazon y Ebay las primeras que implementaron esta tecnología. Más tarde hacia 2004 otras empresas como Flickr lanzaron sus propias API RESTs. En la actualidad, las API RESTs se utilizan en infinidad de sitios web y son una de las columnas vertebrales de la web.

Entre los principios de las API RESTs podemos encontrar [34]:

1. Uso de una arquitectura de tipo Cliente-Servidor

Se separa la interfaz de usuario de la parte en la que almacenamos la información o los datos. De esta forma separamos los problemas relativos al cliente de los del servidor y mejoramos la portabilidad entre plataformas distintas.

2. La comunicación entre el cliente y el servidor siempre contiene la información necesaria para ejecutar la petición.

De esta forma el estado de sesión se encuentra siempre en el cliente no en el servidor. Si un acceso a una determinada información requiere de alguna forma la autenticación el cliente debe autenticarse.

3. Uso de la caché:

Se usará el almacenamiento en caché para aumentar la velocidad de las peticiones. Tanto el servidor, el cliente o los componentes intermedios pueden guardar datos en ella si conviene. Podemos clasificar los datos según sean almacenables en la caché o no dependiendo de las circunstancias.

4. Mismas reglas para todos los componentes

Todos los componentes de una API-REST han de seguir las mismas reglas para el intercambio de información. Esto facilita la comunicación entre distintos componentes del sistema.

5. Sistema de capas

Cada elemento individual sólo puede acceder a la información del nivel con el que interactúa. Por ejemplo, si un cliente interactúa con la API mediante un servicio proxy no sabrá lo que hay más allá de este servicio.

## 3.2. Protocolo HTTP

Es el usado por las API RESTs como hemos visto anteriormente. Es un protocolo que sirve para efectuar solicitudes y recoger respuestas. En el caso de un navegador web este será el cliente mientras que el servidor podría ser una aplicación que se ejecuta en el equipo. El cliente envía una petición al servidor y este responde con un código HTTP de finalización pudiendo incluir además la información solicitada [46].

Los códigos de respuesta que podemos encontrar en el protocolo HTTP los podemos agrupar en distintos grupos [41]:

- El código empieza por 1:  
Informa simplemente de que la solicitud se ha recibido.
- Empieza por 2:  
A parte de recibirse, la petición fue entendida y aceptada.
- Empieza por 3:  
Nos indica que son necesarias medidas extraordinarias para completar la petición.
- Empieza por 4:  
La petición que ha enviado el cliente tiene una formato incorrecto o no es posible satisfacerla.
- Empieza por 5:  
El servidor no ha sido capaz de satisfacer la petición del cliente aunque esta es válida sintácticamente.

## 3.3. Algoritmo de Dijkstra

Fue ideado por el científico de la computación Edsger Dijkstra quien en 1959 publicó por primera vez el algoritmo [12].

El objetivo del algoritmo es determinar la ruta más corta dado un grafo con un conjunto de nodos y diferentes pesos entre cada arco. Trata de minimizar la distancia. Se aplica en grafos para en los cuales los pesos de cada arco son mayores que 0. El coste global de un camino se considera como la suma de los arcos por los que se pasa para llegar desde el nodo de origen al de destino.

Entre sus aplicaciones más comunes podemos hablar de aquellas relacionadas con el mundo del transporte. Podemos destacar el caso de distribución de mercancía a través de una red de tiendas (que se considerarán nodos), reparto de correo entre diversos puntos, etc [10]. Entre sus características fundamentales se encuentra el hecho de que es un algoritmo voraz. Esto implica que en cada paso se toma la mejor solución sin tener en cuenta cómo esto pueda afectar en pasos posteriores y que el óptimo hallado en un determinado paso puede variar en sucesivas iteraciones.[10, 40]. Otros ejemplos de algoritmos voraces pueden ser Kruskal [23] o Prim [39].

Los pasos que sigue el algoritmo son los siguientes [10]:

1. Inicialmente nuestra solución estará formada por el nodo inicial. El camino hasta el nodo inicial se considerará 0, y para el resto de nodos infinito. En una lista guardaremos las distancias necesarias para llegar hasta cada nodo. También en otra lista guardaremos cada nodo con su nodo destino.
2. Iteramos a través de los arcos del nodo elegido, que será el que menor camino requiera recorrer para llegar hasta el y que no se haya analizado todavía, evidentemente en el inicio del algoritmo el nodo elegido será el nodo inicial. Si el camino hasta el nodo de destino desde el nodo elegido es menor que el ya guardado en la lista de distancias se guardará este como nuevo camino y el nodo elegido tendrá como nodo destino el que se está analizando en ese momento.
3. Repetimos 2 hasta que no queden nodos
4. Para obtener la solución dado un nodo destino cualquiera solamente debemos iterar la lista en la que asociamos cada nodo con su nodo destino desde el nodo origen hasta llegar hasta el.

### 3.4. Material adjunto

---

# Técnicas y herramientas

---

## 4.1. Técnicas de extracción de datos

Primeramente se barajaron diversas opciones de extracción de datos. Estas fueron principalmente **Web Scraping** [28] y el uso de peticiones HTTP a API RESTs. Como no había encontrado la forma de obtener datos de viajes en bus y en tren trate de usar web scraping con la herramienta **Scrapy** [33] para Python. Sin embargo, me encontré con el problema de que muchas veces el proceso de extracción de datos era bloqueado por la propia web.

Finalmente se consiguió extraer datos de viajes de las siguientes fuentes de datos:

1. API REST de BlablaCar[2]

Básicamente nos permite encontrar viajes realizando una petición que incluya coordenadas de origen y destino, y dos fechas entre las que se buscan los viajes.

2. API REST de Skyscanner[35]

Básicamente nos permite encontrar viajes entre dos nodos (aeropuertos). Para ello debemos obtener un token (a través de nuestra clave de la API) y una clave de sesión. Nos da información como puede ser el precio, la aerolínea, la url de pago, etc.

3. API REST de Trainline[16] Nos da información acerca de la fecha de inicio y finalización de cada viaje y precio. Para ello debemos proporcionarle una fecha y el código de la estación de origen y la de destino.

4. API REST de Google Maps[18] Nos proporciona servicios que nos permiten convertir coordenadas a direcciones y viceversa, establecer un mapa interactivo en nuestra web, etc.

En el caso de los viajes en Trainline y en avión necesitamos indicar a la API desde que nodo partimos y hasta que nodo llegamos. Todos esos datos de nodos los tenemos guardados en nuestra base de datos. En ambos casos obtenemos esos datos de forma distinta.

1. Aeropuertos [35]

En este caso obtenemos la información de los nodos directamente a través de la propia API-REST de skyscanner. Lo que hacemos es considerar el geocatalogo de skyscanner. Gracias a ello podemos tener guardada la información de cada aeropuerto y su código. En el presente trabajo, sólo consideramos los aeropuertos españoles del total de aeropuertos.

2. Estaciones de buses y trenes [14]

Se obtienen los datos de estaciones a través de un csv encontrado por internet con los datos de las estaciones que maneja Trainline. Una vez obtenido consideramos los nodos en los que el país sea España.

## 4.2. Herramientas utilizadas para el desarrollo web

En este punto se hablará acerca del conjunto de herramientas que se ha usado para el desarrollo web.

### Back-end

Para programar el lado del servidor se han usado las siguientes herramientas:

1. **Python**

Python es un lenguaje de programación de código abierto. Su objetivo principal es que el código sea legible y fácilmente interpretable [45].

Su historia se remonta a los años 80 cuando Guido van Rossum creó el lenguaje. Sin embargo, a partir de la versión 2.1 la fundación Python Software Foundation (PSF) se convirtió en dueña del proyecto [45].

## 4.2. HERRAMIENTAS UTILIZADAS PARA EL DESARROLLO WEB1

Entre sus principales características podemos encontrar según [29]

- Es multiparadigma  
Esto significa que permite diferentes estilos de programación como pueden ser programación orientada a objetos o funcional. Nos da libertad de elegir cuál nos conviene en cada caso.
- Es fácilmente extendible  
Significa que podemos utilizar otros lenguajes como C o C++ para el desarrollo de módulos para el.
- Tipado dinámico  
Significa que una determinada variable puede tomar valores de distintos tipos. También se aplica en el lenguaje de programación Ruby.
- Es multiplataforma  
Se puede ejecutar en diferentes sistemas operativos como Linux o Windows.

Hoy en día destaca por su infinidad de usos. Siendo los campos más importantes la inteligencia artificial y el maching learning.

## 2. Django

Django es un framework de back-end de código abierto para el lenguaje de programación Python. Se lanzó inicialmente en 2005. Sigue el patrón MTV model-template-views. Esto significa que se basa en un mapeador relacional de objetos conocido como ORM que hace de puente entre modelos (clases definidas en python), una base de datos relacional y que incluye un sistema de plantillas web para procesar peticiones de tipo HTTP y un Controler que es un distribuidor URL [42].

Entre sus objetivos fundamentales destacan la reusabilidad de los componentes de cada aplicación y que trata de que el desarrollo sea lo más rápido posible [8].

Usan Django diferentes sitios web de importancia global como podemos destacar Instagram [11] y The Washington Times [1].

La otra alternativa a Django para el desarrollo web en python es Flask [13]. Finalmente se decidió usar Django. Algunos de los motivos de esta decisión fueron la gran cantidad de librerías que vienen incorporadas en Django frente a Flask, que no contiene ninguna librería, y el hecho de que Django contiene un panel de gestión de bases de datos propio frente a Flask que no lo contiene [21].

### 3. SQLite

SQLite es un sistema gestor de bases de datos relacionales que podemos usar en Django. Es la base de datos por defecto en Django y viene incorporada con Python. Entre sus ventajas encontramos el hecho de que presenta un buen rendimiento y es poco propensa a errores. [47, 22]

## Front-end

Para este caso se han utilizado las herramientas clásicas de desarrollo web.

### 1. HTML

HyperText Markup Language es el lenguaje de marcado utilizado en las páginas web. Trata de que cada componente se diferencie del resto. Nos permite indicar la estructura de nuestra web solamente con etiquetas. [44]

### 2. CSS

CSS también conocida como "Hoja de estilos en cascada". Se ha usado para dar estética a los elementos HTML. [43]

### 3. Javascript

Es un lenguaje de programación de alto nivel y multiparadigma considerado uno de los ejes del desarrollo web. Se puede utilizar para programar el lado del cliente en una web o el lado del servidor con Node.js [27]. Sigue la especificación ECMAScript.

Javascript se ha utilizado para la incrustación de un mapa interactivo en la web.

## Otros

En esta sección hablaremos de Moovit [24]. Es una aplicación web. Sirve para dar instrucciones acerca de la forma de llegar a un determinado lugar en transporte público. En nuestro caso la utilizaremos para dar estimaciones al usuario acerca de como llegar a la estación o aeropuerto, o cualquier ubicación.



---

# Aspectos relevantes del desarrollo del proyecto

---

## 5.1. Riesgos y retos

En esta sección se van a relatar aquellos riesgos del desarrollo del proyecto y los retos que se han llevado a cabo.

### 1. Riesgo inicial de encontrar proveedores de información

Al principio, debíamos conseguir extraer información de viajes. Esta debía de ser en tiempo real y usando diferentes medios de transporte. Se trato de hacer en un principio mediante Web Scraping [28]. Pudimos ver algunas de sus desventajas: las webs se protegen contra este tipo de prácticas y que si la web cambia su estructura, el script de obtención de información quedará desactualizado. Finalmente, se consiguió acceder a datos a través del protocolo HTTP [46] y de las API-REST de Blablacar [2], Trainline [16] y Skyscanner [35].

### 2. Riesgo de cambio en la forma de acceso a datos por parte de una petición de una API-REST

Como se ha relatado anteriormente, el servicio de acceso a datos mediante API-REST puede cambiar en cualquier momento. Si esto ocurre la aplicación perderá cierto número de funcionalidades en ese momento. Este riesgo es imposible de evitar, no obstante tratamos de minimizarlo guardando la estructura con la que hacemos las peticiones a la API-REST en nuestra base de datos (posteriormente se explicará esta estructura en 5.3).

Guardamos la información relativa a la API-REST en la base de datos, como puede ser la URL Base de la API y la clave. También guardamos la información de cada petición tal como la parte que debemos añadir a la url base de la API para realizar la solicitud, el nombre de la función para extraer los datos devueltos o la estructura en la que debemos enviar los datos a la API. De esta forma, damos a todas las APIs y sus solicitudes una interfaz común. Además, en caso de cambio en la API o la petición, por ejemplo un cambio en la URL de la API, puede ser solucionado fácilmente por el administrador de la base de datos. Esto se conseguirá simplemente cambiando un campo en la base de datos. No hará falta parar la aplicación para ello.

### **3. Riesgo por cese de actividad de una API-REST**

Se podría dar el caso de que una API-REST ya no ofreciera más sus servicios. En este caso deberíamos buscar otra fuente de datos para evitar que la aplicación pierda funcionalidades y o quede inservible. En este caso, deberíamos borrar toda la información de la API que ha dejado de dar servicio, y añadir la nueva con sus peticiones correspondientes. Esto nos será facilitado por la estructura explicada en [5.3](#), que deberíamos de seguir.

Si necesitamos hacer una misma petición a la API desde dos funciones distintas, no tendremos que repetir la estructura de envío de datos a la API ya que la tendremos en la base de datos. En definitiva, la estructura de la que hablaremos en ?? nos ayuda a paliar este riesgo, y a reducir el tiempo de actualización de la aplicación en este caso.

### **4. Riesgo de que la API-REST responda demasiado lento a las peticiones**

Para solucionar este riesgo, se propone el uso de una memoria caché que guarde los resultados recibidos, y que al volver la misma petición a la API utilice el resultado de la caché en vez de hacer una petición a la API. Esto además nos evita problemas derivados de hacer demasiadas peticiones. Se explica más detalladamente en [5.3](#)

## **5.2. Formación**

Para llevar a cabo el proyecto se ha llevado a cabo un proceso formativo, ya que la práctica totalidad de las herramientas empleadas eran desconocidas antes para mí. A excepción del lenguaje de programación python del cuál he adquirido conocimientos a lo largo de la carrera.

### 5.3. PROBLEMAS ENCONTRADOS Y SOLUCIONES APORTADAS<sup>15</sup>

Para adquirir la formación que necesitaba he utilizado los siguientes recursos:

- Documentación de Django [6]
- Documentación de Google Maps [18]
- Tutorial de youtube de Django [48]
- Curso de HTML [4]
- Curso de CSS [3]
- Curso de Javascript [5]
- Comunidad de StackOverflow [37]

### 5.3. Problemas encontrados y soluciones aportadas

En esta sección vamos a hablar acerca de aquellas partes del proyecto en las cuáles hemos encontrado algún problema y que soluciones o alternativas se han propuesto.

#### Problema y solución dada por la lentitud en las solicitudes a la API REST de trainline

Las solicitudes a la API de Trainline [16] son muy costosas en cuanto a tiempo. Como podemos ver en el ejemplo que se muestra a continuación de este párrafo la mayoría de las peticiones tardan entre 0,5 y 2 segundos, esto se suma a que para averiguar los viajes entre dos coordenadas consideramos nodos de origen y nodos de destino, y hacemos una petición por cada nodo origen con todos los de destino para comprobar si hay viajes. Esto nos hace que el número de peticiones sea igual al número de nodos de origen por el número de nodos de destino. En definitiva, la búsqueda de viajes en bus y en tren es muy costosa.

A continuación podemos ver una imagen que ilustra un extracto de tiempos que tarda en encontrar viajes en un caso entre Madrid y Burgos. En la tabla 5.1, podemos ver que la media de tiempo que tarda la API en respondernos es aproximadamente 1,1 segundos. En un caso de viaje entre la ciudad de Burgos y la de Madrid, teniendo en cuenta que tenemos

3 nodos en Burgos y 25 posibles nodos en Madrid, nos encontramos ante 75 peticiones a la API, lo cuál hace un total de 1 min y 22 segundos. Esto es prohibitivo en cuestión de tiempo, por tanto, era necesaria una solución para paliar el problema

### 5.3. PROBLEMAS ENCONTRADOS Y SOLUCIONES APORTADAS<sup>17</sup>

Tiempo de cada solicitud
1.436
0.289
2.292
0.958
1.627
1.073
1.631
0.619
2.417
1.635
1.060
0.318
0.5018
0.405
0.939
0.358

Tabla 5.1: Tiempo en segundos de las primeras peticiones en un ejemplo de búsqueda de viajes entre Burgos y Madrid [31]

Para solucionar el problema, se combinará el acceso a datos de viajes (en bus y en tren) guardados en la base de datos con búsquedas en la API de Trainline. Cuando se vuelva a realizar una petición para la cuál se tienen datos ya guardados (en la fecha indicada por el usuario y para los nodos correspondientes), no se volverá a hacer una nueva petición a la API de trainline sino que se devolverán los resultados directamente de nuestra base de datos, evitando así realizar una nueva petición. En caso de no encontrar datos de viajes entre nodos origen y destino en nuestra Base de Datos local, sí que haríamos una solicitud a la API de Trainline.

La solución propuesta para evitar la desactualización de los datos de este tipo de viajes, es un script de python que se ejecutará una vez al día en segundo plano. Para ello se utilizará la librería app-scheduler para Django [32]. Lo que se hará básicamente será por cada par de nodos origen y destino encontrado, y cuya fecha de inicio del viaje no sea anterior a la actual se ejecutará la búsqueda de viajes entre esos dos nodos en esa fecha, y se guardará el resultado en la base de datos. Por cada viaje cuya fecha de inicio

sea anterior a la fecha actual, se procederá a eliminarlo de la base de datos para evitar ocupar espacio innecesario.

## **Problema y solución dada por la cantidad de nodos que podemos considerar**

Evidentemente, para cada búsqueda de viajes entre dos puntos no podemos considerar todos los nodos guardados en la base de datos. Esto sería un problema grave, ya que como se ha mencionado antes, alguna de las solicitudes que hacemos tardan demasiado tiempo.

Partimos de la base de que cada nodo tiene guardado el municipio al que pertenece y su provincia. Esto será lo que haremos en los diferentes casos para viajes directos:

En caso de los nodos de tipo aeropuerto, simplemente filtramos encontrando aquellos nodos cuyo municipio sea igual al que hemos marcado que queremos ir. Si no encontramos resultados consideramos los nodos aeropuerto de la provincia. Esto se aplicará tanto para origen y destino.

En caso de los nodos de tipo estación hacemos algo similar a los aeropuertos para origen y destino. En el caso de que encontremos nodos de tipo estación en el municipio en el que estemos, simplemente consideramos esos. Si no lo hacemos, consideramos todos los nodos de la provincia y conseguimos el tiempo que tardaríamos en ir en coche hasta cada uno de ellos. Al final consideraríamos los 10 nodos más cercanos, es decir, los 10 nodos a los cuales nos llevaría menos tiempo llegar.

## **Customización de los problemas o errores que puedan devolver las APIs y soluciones dadas**

Claramente las solicitudes a las APIs que utilizamos podrían dar error. Bien por que hayamos superado las cuotas de solitud, bien por cambios en las APIs, etc. Es por ello que en vez de mostrar al usuario toda la traza del error lo se hace es capturar excepciones. Para ello se ha utilizado el servicio de Django messages [7]. Se mostrará un error diferente al usuario según si falla la API de Blablacar, Trainline o Skyscanner.

## **Problemas por diferentes zonas horarias y solución dada**

Es sabido que en España tenemos dos diferentes zonas horarias. Es por ello y porque la aplicación en un futuro podría ser ampliada para utilizar viajes de diferentes países por lo que tenemos en cuenta este problema.

Lo que hacemos simplemente es establecer las fechas de salida y llegada con su zona horaria, y después pasar esas dos fechas a la zona horaria 0 para calcular la duración del viaje.

## **Problema en la aplicación de Dijkstra para los viajes con transbordo y adaptación del algoritmo a los tiempos de los viajes**

Es evidente que en un caso de viajes con transbordos, el tramo del viaje que sucede después ha de empezar después de que acabe el tramo anterior del itinerario. Sin embargo, en el algoritmo de Dijkstra que se usa normalmente no se hace teniendo en cuenta horarios. Es por ello que se ha tenido que implementar una versión propia del algoritmo en la cual, para actualizar la solución a parte de cumplir que sea mejor solución, en cuanto a precio, debe cumplir que cada tramo del viaje empieza después de que haya acabado el anterior.

Las limitaciones de este proceso son que para aplicar el algoritmo de Dijkstra no tenemos en cuenta ni viajes en Blablacar, ni viajes en avión. Esto queda pendiente como mejora futura.

## **Problemas a la hora de cambiar un proveedor de información por otro y, por cambio en el acceso a una API.**

La forma en la que realizamos peticiones vía API-REST puede cambiar. Se puede dar el caso en el que cambie la estructura que debemos enviar para obtener resultados, la URL de acceso a la API o la estructura del JSON que nos devuelve como resultado de nuestra petición. Esto es bastante frecuente en este tipo de servicios. Es por este motivo, y porque en este proyecto usamos gran cantidad de servicios de este tipo (API-REST de Skyscanner [35], API-REST de Trainline [16], API-REST de blablacar [2]), por lo que debemos facilitar a las personas encargadas de la gestión de la aplicación

que se pueda actualizar con celeridad y facilidad en caso de cambios en el acceso a las fuentes de datos externas.

Hemos diseñado una estructura con la cual podemos facilitar el proceso de añadir y borrar nuevas APIs y solicitudes. Además, nos permite que las modificaciones en la forma de hacer solicitudes, o en los elementos propios de la API que usamos sean realizadas fácilmente. A continuación, vamos a ilustrar como el panel de administración de bases de datos que incorpora **Django** [8] junto a esta estructura nos ayudan a ello.

Las APIs se componen de varias partes fundamentales para hacer las peticiones, las cuales se guardarán en la base de datos:

- **Nombre de la API-REST**

A cada API-REST guardada le daremos un nombre único. De esta forma, podemos identificar de forma legible para el programador cada una de ellas.

- **URL base de la API-REST**

Por regla general cada api rest tiene una url base o endpoint. Por ello almacenamos de forma separada esta información del resto.

- **API-KEY**

Clave única que usamos para comunicarnos con la API. Nos permite realizar peticiones.

A la hora de añadir una API solo debemos de seleccionar ADD REST API, y a la hora de borrar debemos de seleccionar una API y pulsar en la acción Delete.



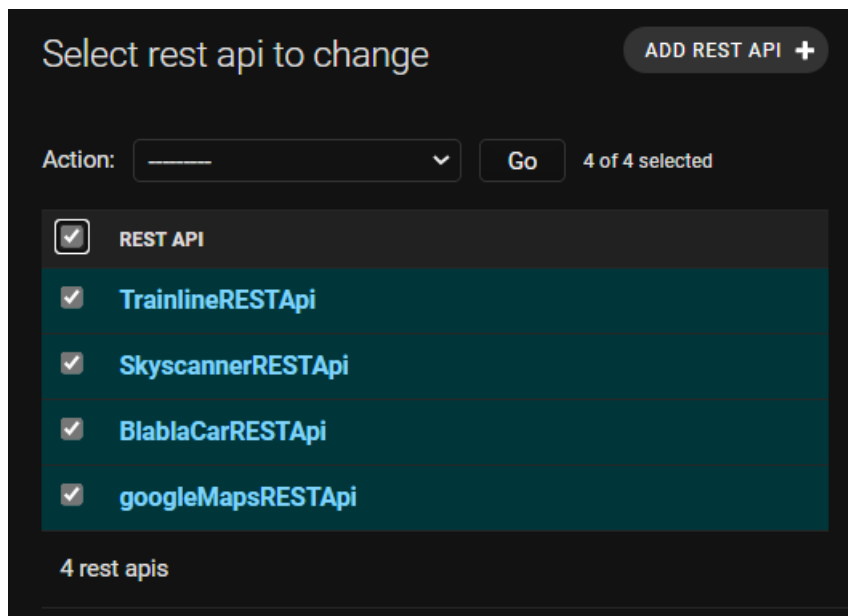


Figura 5.1: Adición o eliminación de API-REST [31].

Si queremos añadir una nueva API (o lo que es lo mismo, un nuevo proveedor de información) debemos de rellenar el siguiente formulario. BaseUrl y APIKey son los dos valores que usarán las peticiones. No nos deberemos preocupar por el valor de la URL o el de la API-Key en el código fuente.

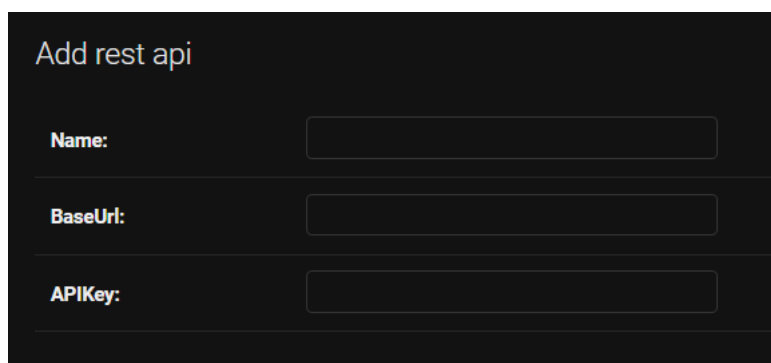
The image shows a dark-themed web form titled "Add rest api". It contains three input fields, each with a label to its left: "Name:", "BaseUrl:", and "APIKey:". Each label is in a light blue color, and the corresponding input field is a light blue rectangle. The form is set against a dark background.

Figura 5.2: Formulario para añadir una nueva API-REST [31].

En cuanto a las peticiones que cada API puede realizar, podemos distinguir varias partes:

- **Nombre de la petición**

A cada petición guardada le daremos un nombre único. De esta forma, podemos identificar de forma legible para el programador cada una de ellas.

- **Parte que debemos añadir a la url base**

Es una cadena de texto, que para cada petición es distinta y que debe de ser concatenada con la url base de la api rest.

- **Estructura del diccionario de Python que se usa para cada petición**

Cada petición tiene una serie de parámetros que deben de ser enviados. Almacenamos esta estructura vacía. Después debemos pasar una lista con los valores que queremos dar a cada clave del diccionario.

- **Nombre de la función con la cuál devolvemos los datos de la petición**

Con ese nombre ejecutamos la función que extraerá los datos del json devuelto por la API-REST.

### 5.3. PROBLEMAS ENCONTRADOS Y SOLUCIONES APORTADAS<sup>23</sup>

- **Tipo de petición** Diferenciamos según sean peticiones de tipo GET o POST.
- **API-REST asociada** Cada petición, evidentemente, tendrá una API asociada.

En caso de añadir una petición, nos vale con rellenar el siguiente formulario de adición marcando en la opción RApi, la que deseemos asociar a la petición.

The image shows a dark-themed web form titled "Add request". It contains several input fields and a dropdown menu. The fields are labeled as follows:

- Name:** A single-line text input field.
- Description:** A single-line text input field.
- PartToaddToBaseUrl:** A single-line text input field.
- FuncToExtractDataFromJs onName:** A single-line text input field.
- ParamsOrDataDictStructure:** A multi-line text area containing the word "null".
- TypeRequests:** A dropdown menu with a downward arrow.
- Headers:** A multi-line text area containing the word "null".
- RApi:** A dropdown menu with a downward arrow, a pencil icon, and a plus icon.

Figura 5.3: Formulario para añadir una nueva Request(petición) [31].

En caso de borrar una petición, nos vale con seleccionarla y marcar la opción Delete.

### 5.3. PROBLEMAS ENCONTRADOS Y SOLUCIONES APORTADAS<sup>25</sup>

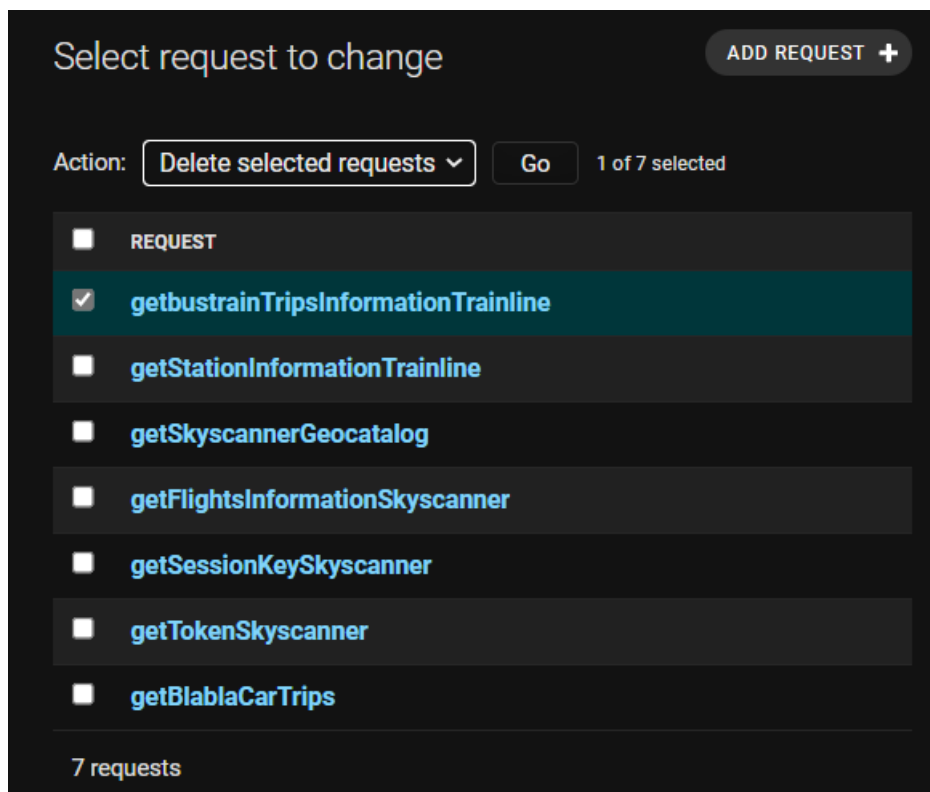
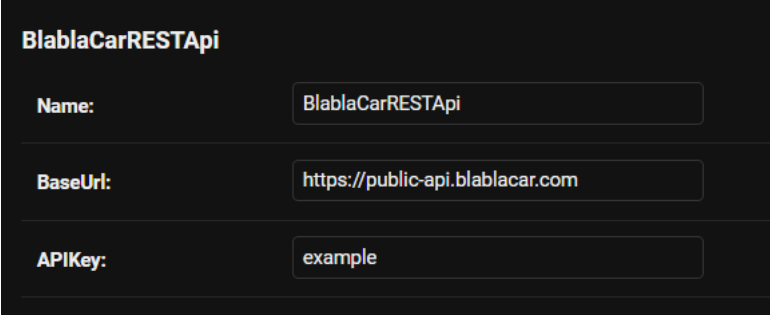


Figura 5.4: Adición o eliminación de una Request(petición) [31].

Para explicar la modificación de atributos y el significado de los mismos, vamos a ilustrar el siguiente ejemplo de petición. Para hacer una petición a la API de Blablacar necesitaremos incluir los siguientes datos:

- Clave (key)
- Coordenadas de origen (from\_coordinates)
- Coordenadas de destino (to\_coordinates)
- Moneda (currency)
- Fecha mínima de inicio del viaje (start\_date\_local)
- Fecha máxima de inicio del viaje (end\_date\_local)

De Blablacar podemos ver sus atributos en el modelo API-REST.



BlablaCarRESTApi

Name:	BlablaCarRESTApi
BaseUrl:	https://public-api.blablacar.com
APIKey:	example

Figura 5.5: Formulario de modificación de API-REST [31].

### 5.3. PROBLEMAS ENCONTRADOS Y SOLUCIONES APORTADAS27

Y sus atributos en el modelo Request:

**getBlablaCarTrips**

**Name:**

**Description:**

**PartToaddToBaseUrl:**

**FuncToExtractDataFromJs onName:**

**ParamsOrDataDictStructur e:**

```
{
  "key": "example",
  "from_coordinate": "41.3887900,2.1589900",
  "to_coordinate": "40.416775,-3.703791",
  "currency": "EUR",
  "start_date_local": "2021-05-05T00:00:00",
  "end_date_local": "2021-05-06T00:00:00"
}
```

**TypeRequests:**

**Headers:**

**RApi:**

Figura 5.6: Formulario para modificar una Request [31].

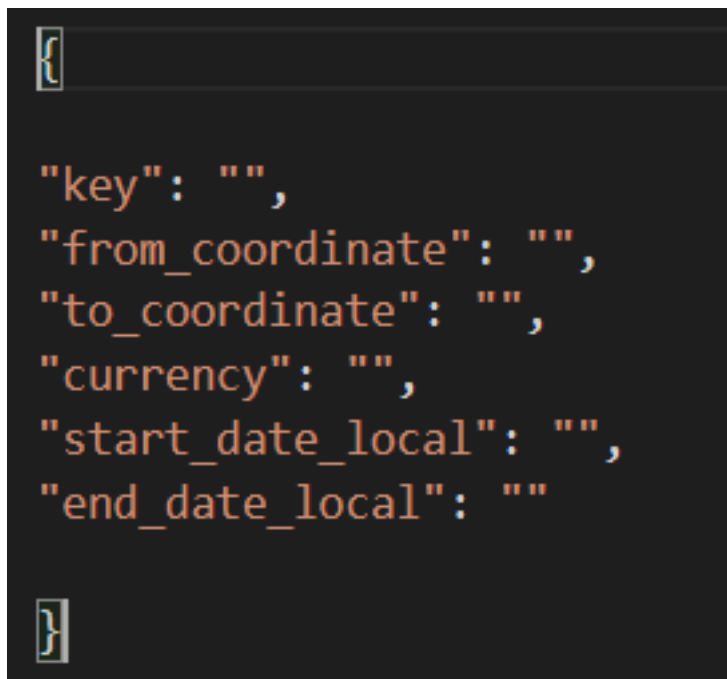
Con la estructura vista antes, para conseguir un iterador con los resultados de una consulta a la API de blablacar deberíamos hacer lo siguiente:

1. Siguiendo el orden de la estructura del diccionario de Python que se usa para nuestra petición (que veíamos en 5.3), creamos una estructura de tipo lista con los que serían los valores para cada uno.

Un ejemplo podría ser:

```
['api-key', '42.355011,-3.684135', '41.65087283744742,-0.9045939843749928',  
'EUR', '2021-07-06T00:00:00', '2021-07-07T00:00:00']
```

Este seguirá el orden siguiente, el cuál podemos observar en la imagen 5.6:



```
{  
  "key": "",  
  "from_coordinate": "",  
  "to_coordinate": "",  
  "currency": "",  
  "start_date_local": "",  
  "end_date_local": ""  
}
```

Figura 5.7: Estructura para realizar una petición a Blablacar [31].

De esta forma, si cambiará el nombre de cualquier clave en la estructura anterior sólo deberíamos cambiarla en la base de datos. En caso de incluir una nueva clave, aparte de las 6 anteriores, deberíamos añadirla a la estructura y pasar un argumento extra a la lista. En caso de necesitar borrar una clave, la quitaríamos de la estructura anterior y quitaríamos su valor de la lista anteriormente explicada.

2. La lista que hemos definido antes se lo pasamos a una función (executeFunction), la cual está disponible para cualquier petición. Esta hará varios pasos:
  - Ejecutará la petición a la API, y conseguirá un JSON que esta nos devolvera. Para ello, utilizará los datos de la URL base de la API-REST y de la parte que se ha de añadir a esta para realizar



### 5.3. PROBLEMAS ENCONTRADOS Y SOLUCIONES APORTADAS<sup>29</sup>

la petición. Además rellenará ordenadamente la estructura 6.1 con los datos de la lista.

- Segundo, recordamos que cada Request tiene asociada el nombre de una función que procesará el JSON devuelto. Entonces, habiendo conseguido ya el JSON devuelto, sólo debemos pasarle este JSON a la función cuyo nombre tenemos guardado. La función asociada a la Requests nos devolverá un iterador, que tendrá la información de cada viaje. Se lo devolvemos al usuario.

Evidentemente, de este punto podemos sacar varias conclusiones:

- a) Si cambiará la API Key o la URL de la API con sólo cambiarla en la base de datos (5.5) bastaría.
- b) En el caso de cambio en la parte de la URL a añadir a la de la API, bastaría con cambiarla también en la base de datos (5.5).
- c) Si cambia la estructura del JSON que nos devuelve la API, sólo deberíamos cambiar el código fuente de la función que lo procesa.

En conclusión, con esta estructura que hemos planteado anteriormente nos permite reaccionar rápidamente a cualquier pequeño cambio en la forma de hacer peticiones a la API REST. Esto es necesario debido a que estos servicios cambian frecuentemente. De esta forma si cambia la estructura del JSON que nos devuelve los resultados solo debemos cambiar la función con la que devolvemos los datos que estará separada del resto de elementos. Lo mismo ocurre con los demás elementos antes mencionados.

Deberíamos valorar si es adaptable a otros proyectos. Algunas de sus desventajas son: el hecho de que requiere muchas consultas a la base de datos y que requiere tener información guardada en la base de datos. Estos accesos hacen que la aplicación sea ligeramente más lenta. En casos en los que sólo se usa una sola API, o en los que el rendimiento es un factor vital no sería una buena opción.



---

## Trabajos relacionados

---

En este capítulo se hablará acerca de que proyectos se han utilizado como referencia para la construcción de la aplicación web.

### 6.1. Proyectos

Se han utilizado otros proyectos alojados en GitHub como referentes.

- **Repositorio de Trainline** [15]

Es el repositorio que nos ha ayudado a acceder a la API de trainline. Gracias a su código hemos podido ver cómo se debe de hacer para mandar peticiones a esta.

Permite ver datos de viajes vía Trainline entre una estación origen y una destino. Sin embargo, no permite explorar viajes con transbordos, como si lo hace nuestro proyecto. Tampoco da la opción de ver viajes vía Blablacar o en avión. Además, carece de interfaz gráfica.

- **Google Maps** [17]

La principal desventaja con el proyecto del que estamos hablando, es que no permite encontrar viajes con transbordos.

En cuanto a las ventajas, podemos observar una interfaz más amigable. También nos permite elegir otros medios de transporte, como coche particular o bicicleta. Otro factor destacable es que esta aplicación es más rápida que la nuestra a la hora de dar resultados.

- **Web de Trainline** [38]

Las desventajas con nuestra aplicación son:

1. Carece de un mapa para seleccionar los puntos de origen y llegada.
2. No encuentra viajes en Blablacar ni en avión.
3. No contempla opciones de ordenar viajes por precio o duración.
4. No hay ninguna casilla en la que podamos decir el precio máximo del viaje mostrado.
5. No nos informa del transporte público como si lo hace nuestro proyecto.

Y las ventajas con respecto a nuestro proyecto son:

1. En los viajes con transbordo nos dice el tiempo exacto que tenemos para llegar hasta el siguiente Nodo:

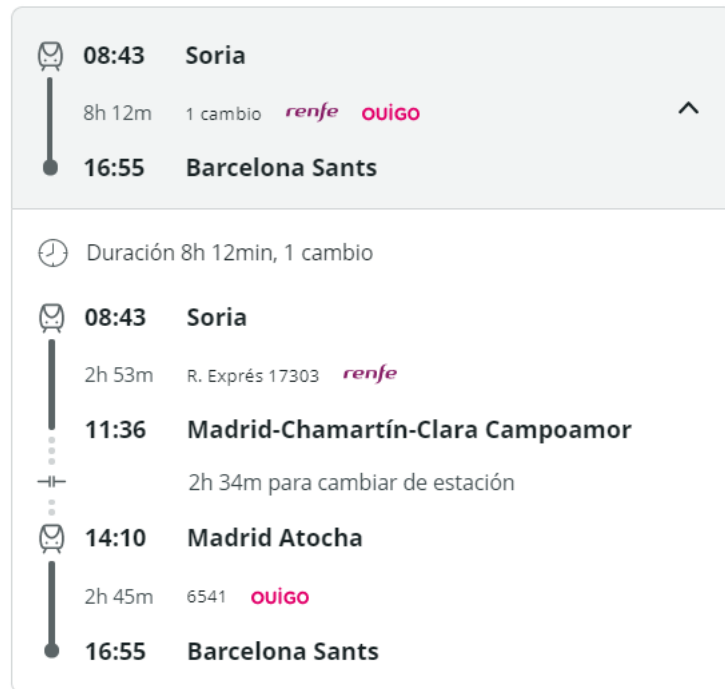


Figura 6.1: Viaje con transbordo en la aplicación de Trainline [31].

2. Nos dice cuál es el mejor viaje según precio y duración, y el más barato.
3. Podemos comprar el viaje en tren o bus directamente desde la aplicación.
4. Se pueden buscar viajes en países extranjeros.

- **Web de Skyscanner** [36]

Nos permite encontrar viajes con transbordos en aviones, aunque no nos permite encontrar ningún tipo de viaje en bus, tren o Blablacar como si hace nuestra aplicación.



---

# Conclusiones y líneas de trabajo futuras

---

En esta sección se va a hablar acerca de aquellas conclusiones del desarrollo diario del trabajo a parte de aquellas líneas futuras de ampliación del mismo.

## 7.1. Conclusiones

Podemos destacar las siguientes:

- El proyecto ha conllevado riesgos. Esto es debido fundamentalmente a que las fuentes de datos que se han utilizado son externas, y por tanto no tenemos el control sobre ellas. Es muy frecuente que este tipo de servicios cambien en cuanto a la forma en la que se hace las peticiones o que simplemente desaparezcan. Ante un cambio en un servicio la aplicación no se vería actualizada automáticamente sino que se quedaría desactualizada por un tiempo, lo cuál podría dejar el proyecto inservible durante un tiempo. Esto se suma a que no sabemos cuando los servicios cambiarán.
- Se han conseguido paliar los diferentes problemas encontrados correctamente. Entre las soluciones podemos destacar:
  - **La creación de un sistema de almacenamiento de los datos que nos son útiles para hacer peticiones a las fuentes de datos.**

Los datos, utilizados en las peticiones a las APIs, serán almacenados en nuestra base de datos. Básicamente dividimos cada

petición en sus partes fundamentales (Clave, petición de datos, URL, procesamiento de los datos devueltos). Esto, nos permite aislar problemas. Ante un cambio en una de esas partes, sólo tendremos que modificar un valor o en la base de datos o hacer cambios mínimos. También evitamos errores de los programadores, ya que al tener una estructura común para todas las peticiones será más fácil borrar o incluir nuevas.

En definitiva, de esta forma paliamos el riesgo más grave del proyecto que es la desactualización de la aplicación debido a proveedores externos.

- **La implantación de un sistema de memoria caché que permitirá paliar problemas de lentitud de determinadas APIs.**

Las búsquedas a la API de Trainline son lentas, y los tiempos son prohibitivos (1s de media por cada búsqueda aproximadamente). Como solución, hemos implantado un sistema que guarda los viajes encontrados en la base de datos. Imaginemonos que necesitamos los viajes de un determinado día, entre dos estaciones cualesquiera, y que esa búsqueda ya se ha realizado anteriormente. En ese caso, devolvemos los viajes directamente de la base de datos ahorrando por tanto un tiempo considerable.

De este sistema de memoria caché, surge un problema. Cabe la posibilidad de que los viajes de la caché estén desactualizados. La solución planteada es que cada 24 horas en segundo plano, se vuelvan a buscar los viajes que ya se encuentran en la caché y que no sean anteriores a la fecha actual.

- **Se ha desarrollado una forma para limitar el número de nodos que podemos considerar.**

En los viajes en tren, bus y avión, el problema de encontrar viajes es un problema de nodos (estaciones y aeropuertos). No podemos considerar todos en cada búsqueda. Por tanto, en caso de viajes en bus o en tren, consideramos los nodos que haya en la ciudad de origen (lo mismo para el destino) si hay, y si no hay consideramos las 10 estaciones más próximas en la provincia en la que estemos. En avión consideramos los nodos del municipio, o los de la provincia en caso de que no haya en el municipio.

De esta forma, evitamos la lentitud de la aplicación y hacemos factibles las diferentes búsquedas.



- **Se ha conseguido desarrollar un sistema para encontrar viajes con transbordos.**

Hemos conseguido desarrollar un sistema para encontrar viajes con transbordos en bus y en tren. Para ello, aplicamos el algoritmo de Dijkstra [12].

Un problema encontrado, que no cubre Dijkstra, es que cada viaje del itinerario debe empezar después de que acabe su predecesor. Esto se consigue comprobando si eso se cumple cada vez que consideramos una solución como buena en el algoritmo, si no se cumple esta solución no se considera válida.

- Los objetivos del proyecto finalmente se han cumplido, y es por tanto un proyecto factible. Se ha conseguido extraer datos reales de diferentes servicios acerca de viajes y procesar esa información, para esto en un principio se tuvo que llevar a cabo un trabajo de investigación acerca de cómo era posible realizar esto. Al principio se considero la técnica de Web Scraping [28], pero no fue factible. Finalmente, conseguimos acceder a fuentes de datos vía API-REST.

## 7.2. Líneas de trabajo futuras

Algunas de las mejoras que se pueden proponer al proyecto actual son:

- Considerar viajes en avión y Blablacar a la hora de llevar a cabo la búsqueda por el algoritmo Dijkstra.
- Ampliación de la aplicación incluyendo estaciones y aeropuertos de otros países a parte de España.
- Aplicar un sistema por el cuál se notifique a un email administrador los errores que den los accesos a las APIs cuando se produzcan. Esto se haría con el objetivo de que la aplicación sea actualizada lo más rápido posible ante cambios por parte de los proveedores de información.
- Que la aplicación sea posible ejecutarla como aplicación de Android y de iOS para dispositivos móviles a parte de ser una aplicación web.
- Implementación de una versión que utilice una base de datos NoSQL, de la familia de bases de datos de grafos. Concretamente Neo4j [26]. Gracias a ello, podríamos, por ejemplo, aplicar el algoritmo de Dijkstra con solo llamar a una función [25].
- Investigar otros algoritmos de optimización de rutas bioinspirados alternativos a Dijkstra (algoritmos genéticos, de la colonia de hormigas, o de tipo recocido), por ejemplo [12]. Esta línea sería interesante desarrollarla en combinación con la implementación en Neo4j.

---

## Bibliografía

---

- [1] Open Source at The Washington Times. Open source at the washington times. <http://opensource.washingtontimes.com/>, 2011. [Online; Accedido 25-Junio-2021].
- [2] BlablaCar. Search v3 - api documentation. <https://support.blabblacar.com/hc/en-gb/articles/360014199820--Search-V3-API-Documentation>, 2021. [Online; Accedido 23-Junio-2021].
- [3] Codecademy. Learn css. <https://www.codecademy.com/learn/learn-css>, 2021. [Online; Accedido 24-Junio-2021].
- [4] Codecademy. Learn html. <https://www.codecademy.com/learn/learn-html>, 2021. [Online; Accedido 24-Junio-2021].
- [5] Codecademy. Learn javascript. <https://www.codecademy.com/learn/introduction-to-javascript>, 2021. [Online; Accedido 24-Junio-2021].
- [6] Django. django. <https://www.djangoproject.com/>, 2021. [Online; Accedido 24-Junio-2021].
- [7] Django. The messages framework. <https://docs.djangoproject.com/en/3.2/ref/contrib/messages/>, 2021. [Online; Accedido 24-Junio-2021].
- [8] Djangoproject. Django makes it easier to build better web apps more quickly and with less code. <https://www.djangoproject.com/>, 2021. [Online; Accedido 25-Junio-2021].
- [9] ECMA. The JSON Data Interchange Syntax. page 5, 2017.

- [10] Ecured. Algoritmo de dijkstra. [https://www.ecured.cu/Algoritmo\\_de\\_Dijkstra](https://www.ecured.cu/Algoritmo_de_Dijkstra), 2021. [Online; Accedido 23-Junio-2021].
- [11] Instagram Engineering. What powers instagram: Hundreds of instances, dozens of technologies. <https://instagram-engineering.com/what-powers-instagram-hundreds-of-instances-dozens-of-technologies-adf2e22da2ad>, 2011. [Online; Accedido 25-Junio-2021].
- [12] E.W.Dijkstra. A Note on Two Problems in Connexion with Graphs. pages 1–3, 1959.
- [13] Flask. User’s guide. <https://flask.palletsprojects.com/en/2.0.x/>, 2021. [Online; Accedido 25-Junio-2021].
- [14] GitHub. Stations - a database of european train stations. <https://github.com/trainline-eu/stations>, 2021. [Online; Accedido 23-Junio-2021].
- [15] GitHub. Trainline. <https://github.com/tducret/trainline-python>, 2021. [Online; Accedido 25-Junio-2021].
- [16] GitHub. trainline-python. <https://github.com/tducret/trainline-python>, 2021. [Online; Accedido 23-Junio-2021].
- [17] Google. Google maps. <https://www.google.es/maps/>, 2021. [Online; Accedido 25-Junio-2021].
- [18] Google. Las api de google maps platform por plataforma. <https://developers.google.com/maps/apis-by-platform>, 2021. [Online; Accedido 24-Junio-2021].
- [19] Heroku. Heroku. <https://dashboard.heroku.com/>, 2021. [Online; Accedido 25-Junio-2021].
- [20] IBM. Rest apis— IBM Cloud Education. <https://www.ibm.com/cloud/learn/rest-apis>, 2021. [Online; Accedido 23-Junio-2021].
- [21] ILimit. Flask vs. django. <https://www.ilimit.com/blog/flask-vs-django/>, 2021. [Online; Accedido 23-Junio-2021].
- [22] JavaTpointg. Ventajas de sqlite. <https://www.javatpoint.com/sqlite-advantages-and-disadvantages>, 2021. [Online; Accedido 23-Junio-2021].

- [23] Joseph Kruskal. In *On the shortest spanning subtree and the traveling salesman problem*, pages 48–50, 1956.
- [24] Moovit. Embedded web app. <https://moovit.com/developers/embedded/>, 2021. [Online; Accedido 25-Junio-2021].
- [25] Neo4j. Dijkstra single-source. <https://neo4j.com/docs/graph-data-science/current/algorithms/dijkstra-single-source/>, 2021. [Online; Accedido 25-Junio-2021].
- [26] Neo4j. Neo4j documentation. <https://neo4j.com/docs/>, 2021. [Online; Accedido 25-Junio-2021].
- [27] Nodejs. Api reference documentation. <https://nodejs.org/en/docs/>, 2019. [Online; Accedido 25-Junio-2021].
- [28] Parsehub. What is web scraping and what is it used for? <https://www.parsehub.com/blog/what-is-web-scraping/>, 2021. [Online; Accedido 25-Junio-2021].
- [29] Python.es. ¿qué es python? <https://python.es/que-es-python-y-sus-caracteristicas/>, 2021. [Online; Accedido 24-Junio-2021].
- [30] Rocktech. Conectando una aplicación de angular con una api rest. <https://rd.rocktech.mx/publicaciones/entrada/conectando-una-aplicacion-de-angular-con-una-api-rest>, 2021. [Online; Accedido 25-Junio-2021].
- [31] Andrés Rodríguez. Tfgviajes. <https://github.com/andriu99/TFGViajes>, 2021. [Online; Accedido 25-Junio-2021].
- [32] Advanced Python Scheduler. Advanced python scheduler. <https://apscheduler.readthedocs.io/en/stable/>, 2021. [Online; Accedido 24-Junio-2021].
- [33] Scrapy.org. Scrapy. <https://scrapy.org/>, 2021. [Online; Accedido 25-Junio-2021].
- [34] Seobilitywiki. Rest api. [https://www.seobility.net/en/wiki/REST\\_API](https://www.seobility.net/en/wiki/REST_API), 2021. [Online; Accedido 23-Junio-2021].
- [35] Skyscanner. Api documentation. <https://skyscanner.github.io/slate/>, 2021. [Online; Accedido 23-Junio-2021].

- [36] Skyscanner. skyscanner. <https://www.skyscanner.es/>, 2021. [Online; Accedido 25-Junio-2021].
- [37] StackOverflow. Stackoverflow. <https://stackoverflow.com/>, 2021. [Online; Accedido 24-Junio-2021].
- [38] Trainline. trainline. <https://www.thetrainline.com/>, 2021. [Online; Accedido 25-Junio-2021].
- [39] Tutorialcup. Algoritmo de prim. <https://www.tutorialcup.com/es/entrevista/algoritmo/algoritmo-prims.html>, 2021. [Online; Accedido 25-Junio-2021].
- [40] Wikipedia. Algoritmo voraz. [https://es.wikipedia.org/wiki/Algoritmo\\_voraz](https://es.wikipedia.org/wiki/Algoritmo_voraz), 2021. [Online; Accedido 23-Junio-2021].
- [41] Wikipedia. Anexo:códigos de estado http. [https://es.wikipedia.org/wiki/Anexo:Códigos\\_de\\_estado\\_HTTP](https://es.wikipedia.org/wiki/Anexo:Códigos_de_estado_HTTP), 2021. [Online; Accedido 25-Junio-2021].
- [42] Wikipedia. Django (framework). [https://es.wikipedia.org/wiki/Django\\_\(framework\)](https://es.wikipedia.org/wiki/Django_(framework)), 2021. [Online; Accedido 23-Junio-2021].
- [43] Wikipedia. Hoja de estilos en cascada. [https://es.wikipedia.org/wiki/Hoja\\_de\\_estilos\\_en\\_cascada](https://es.wikipedia.org/wiki/Hoja_de_estilos_en_cascada), 2021. [Online; Accedido 23-Junio-2021].
- [44] Wikipedia. Html. [https://es.wikipedia.org/wiki/HTML#Etiquetas\\_HTML\\_b%C3%A1sicas](https://es.wikipedia.org/wiki/HTML#Etiquetas_HTML_b%C3%A1sicas), 2021. [Online; Accedido 23-Junio-2021].
- [45] Wikipedia. Python. <https://es.wikipedia.org/wiki/Python>, 2021. [Online; Accedido 24-Junio-2021].
- [46] Wikipedia. Rest api. [https://en.wikipedia.org/wiki/Hypertext\\_Transfer\\_Protocol#Technical\\_overview](https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Technical_overview), 2021. [Online; Accedido 23-Junio-2021].
- [47] Wikipedia. Sqlite. <https://en.wikipedia.org/wiki/SQLite>, 2021. [Online; Accedido 23-Junio-2021].
- [48] Youtube. Curso django desde cero. <https://www.youtube.com/watch?v=vXR5CAcRv5w&t=5051s>, 2021. [Online; Accedido 24-Junio-2021].