Desarrollo de aplicación web para compartir vehículo en viajes recurrentes

Memoria de Proyecto Final Máster

Máster Universitario en Desarrollo de Sitios y Aplicaciones Web

Área de informática, multimedia y telecomunicaciones.

Autor: Ernesto Gimeno Sánchez

Consultora: Anna Ferry Mestres

Profesor: Carlos Casado Martínez

Dedicatoria

A mi familia, por estar siempre ahí.

Abstract

En el presente proyecto se desarrolla una aplicación para permitir que diversos usuarios puedan compartir vehículo en viajes que se realizan de manera recurrente, como los que tienen lugar por ejemplo para ir y volver del trabajo.

Para el desarrollo de la aplicación se utilizará un *backend*[1] desarrollado en Ruby On Rails que gestionará la autenticación/autorización de peticiones, así como todas las operaciones de lectura/escritura con base de datos PostgresSQLy la serialización de las respuestas JSON que suministrará al *frontend*[2], desarrollado en Angular 10.

La aplicación se desarrollará de manera *responsive*[3] para su uso adecuado preferiblemente en móviles y ordenadores, haciendo uso de HTML/CSS(SASS).

Palabras clave: Trabajo Final de Máster, aplicación web, compartir vehículo, Angular, Ruby on Rails.

[1] Parte de la aplicación encargada de gestionar la lógica de negocio, conectar con la base de datos y el servidor y que ejerce de intermediario entre la capa de datos y la de visualización.
[2] Parte de la aplicación que gestiona la interfaz de usuario, la “cara visible” de la aplicación.
[3] Técnica de diseño web que busca la correcta visualización de una misma página en distintos dispositivos.

Abstract(english)

In the present document it will be developed a web application to give users the ability to share their vehicles in recurrent travels, such as the ones that take place to go to their jobs on a daily basis.

The application will be developed by using Ruby on Rails as the backend that will be in charge of authenticating/authorizing requests as well as to manage all the needed CRUD operations with the postgresSQL database and also the serializing of the JSON output supplied to the Angular 10 frontend.

The application will be developed following the responsive design principles focusing on cell phones and desktop/laptop computers, making use of HTML/CSS(SASS).

keywords: Master final project, web application, sharing vehicle, Angular, Ruby on Rails.

Agradecimientos

A mi consultora Anna por su soporte y sus certeros consejos.
A todos los creadores de contenido y guías que desinteresadamente contribuyen al acceso al conocimiento por parte de todos.
A mi jefe José Ignacio por poner los medios para que pueda dar rienda suelta a mi pasión por el desarrollo y poner en práctica lo aprendido.

Índice

Figuras y tablas

Índice de figuras

Índice de tablas

1. Introducción

Durante el presente documento, se detallará el proceso de desarrollo de una aplicación web con el propósito de ayudar a las personas para colaborar a la hora de llevar a cabo viajes recurrentes, teniendo presente de manera fundamental el caso de uso en que diversas personas deciden compartir vehículo para acudir al puesto de trabajo.
Para ayudar en este proceso, se trabajará en una herramienta que permitirá a un usuario determinado poner su vehículo a disposición de otros, y otro grupo de personas podrá unirse a estos para realizar el viaje de manera conjunta.
Esta herramienta será una aplicación web completa que constará de dos partes diferentes que pueden considerarse aplicaciones separadas:

- Backend: API REST conectada a base de datos PostgresSQL, que se desarrollará con Ruby on Rails 6 y se conectará con los servicios del api de HERE Maps.
- Frontend que se desarrollará con Angular 10, HTML, SASS y la librería de componentes Angular Material.

1.1 Contexto y justificación

Gran parte de los desplazamientos que se llevan a cabo a diario tienen lugar con motivo de desplazarnos al centro de trabajo. Muchos de estos desplazamientos tienen lugar a su vez mediante turismos y otro tipo de vehículos privados.
Por ejemplo, según la encuesta sintética de movilidad del año 2014 de la Comunidad de Madrid, la población que viajaba por motivos de trabajo ascendía a 2,26 millones, y En el caso de Barcelona, la encuesta de movilidad en día laborable de 2015 (EMEF12), el número de viajes con destino el trabajo en un día laborable era de 2,5 millones de desplazamientos, que representaban el 13,3 % del total de viajes, con un reparto por género de 55,3 % de hombres y 44,7 % de mujeres. El modo de transporte mayoritario era el vehículo privado con un 56,2 % ([4]).
Además, la posibilidad de conseguir que las personas se agrupen compartiendo vehículo en este tipo de trayectos conlleva una gran cantidad de beneficios tanto a nivel ecológico como social siendo las siguientes los más evidentes:

- Reducción de la huella de emisiones de CO2 ligadas a la movilidad.
- Reducción de costes al compartir los costes derivados del trayecto.
- Mejora en la inclusión social de personas que no disponen de medio de transporte para desplazarse a zonas mal conectadas con transporte público, como es el caso de muchos polígonos.

[4]https://www.idae.es/sites/default/files/la_movilidad_al_trabajo_un_reto_pendiente_dgt_idae_junio_2019.pdf

2. Objetivos

En este epígrafe, se valoran los objetivos principales y secundarios que se desean acometer con el presente trabajo.

2.1 Principales

- Practicar y demostrar los conocimientos adquiridos a lo largo del máster mediante el desarrollo de una app completa.
- Desarrollar una interfaz de usuario fluida y moderna de tipo SPA.
- Desarrollar una API REST completa que incluya autenticación/autorización y serialización de datos.
- Aplicar los principios del diseño responsive para lograr una adecuada experiencia de usuario en dispositivos móviles además de en pantallas grandes.

2.2 Secundarios

- Desarrollar una aplicación escalable y mantenible, haciendo uso de buenas prácticas y tests.
- Desplegar la aplicación en un entorno de producción.
- Conectarse a servicios externos.
- Hacer uso de flujos de trabajo modernos utilizando herramientas como Git/Github, linters, formateadores de código etc.
- Llevar a cabo una adecuada documentación, tanto del propio proceso de desarrollo de este trabajo como de diferentes elementos críticos de la aplicación, como los *endpoints* (urls de una api o *backend* que responden a una petición enviada desde un cliente) disponibles en la API.

3. Enfoque y método seguidos

Durante el presente Proyecto, se llevará a cabo un enfoque basado en Agile. La filosofía Agile se basa en separar un trabajo completo en partes más pequeñas que deben generar entregables cada poco tiempo.

Esta metodología, en contraposición el modelo clásico en cascada busca ser más flexible y permitir mediante entregas rápidas obtener feedback que ayude a iterar mejorando el resultado, en lugar de seguir una programación rígida. Agile intenta partir de un comienzo menos planificado, entrando menos en el detalle concreto, pero permitiendo ser más flexible a la hora de introducir cambios necesarios que van percibiéndose mejor conforme el desarrollo avanza y los entregables generan feedback.

Para poder ir generando entregables de tipo *vertical slice*[5], la idea es llevar a cabo el desarrollo del frontend y el backend en paralelo.

Además, se llevará a cabo un tablero Kanban mediante la aplicación Trello que permita ayudar en la gestión del Proyecto.

[5] Hitos en la realización de un proyecto que hacen énfasis en demostrar progresos en todas las áreas de desarrollo.

4. Planificación

A continuación, se muestra la planificación de las tareas a desarrollar con la duración de las tareas en días, así como un diagrama de Gantt para su correcta visualización

TAREA	FECHA INICIAL	FECHA FINAL	DURACIÓN
Definición del tema	16-9-2020	22-9-2020	6
Desarrollo PEC 1	23-9-2020	28-9-2020	5
Entrega PEC 1	◆	28-9-2020	
Prototipado	29-9-2020	5-10-2020	6
Definición de la rquitectura de software	6-10-2020	8-10-2020	2
Diseño de modelos/bd	9-10-2020	11-10-2020	2
Backend: Setup de proyecto y Sistema Autenticación JWT	12-10-2020	22-10-2020	10
Desarrollo PEC2	23-10-2020	28-10-2020	5
Entrega PEC 2	◆	28-10-2020	
Backend CRUD/endpoints users/viajes	29-10-2020	3-11-2020	5
Frontend; Setup de proyecto y landing Page	4-11-2020	7-11-2020	3
Frontend; componente Login y guards para auth.	8-11-2020	13-11-2020	5
Backend: Integración servicios ext. mapas/lugares	14-11-2020	19-11-2020	5
Frontend: Página de generación de nuevo viaje	20-11-2020	24-11-2020	4
Frontend: Página de busqueda y vista de lista de viajes	25-11-2020	30-11-2020	5
Frontend: Página de detalle de viaje	30-11-2020	2-12-2020	2
Grabación video 1ª versión	2-12-2020	5-12-2020	3
Desarrollo PEC 3	5-12-2020	6-12-2020	1
Entrega PEC 3	◆	6-12-2020	
Backend: Desarrollo sistema de chat en viaje	7-12-2020	11-12-2020	4
Frontend: Implementación Chat	12-12-2020	16-12-2020	4
Tests y corrección de errores	17-12-2020	21-12-2020	4
Despliegue	21-12-2020	23-12-2020	2
Autoinforme	23-12-2020	23-12-2020	0
Grabación video definitivo	23-12-2020	24-12-2020	1
Presentación para público	24-12-2020	29-12-2020	5
Finalización memoria	30-12-2020	4-1-2021	5
Entrega Final	◆	4-1-2021	

Tabla 1. Planificación

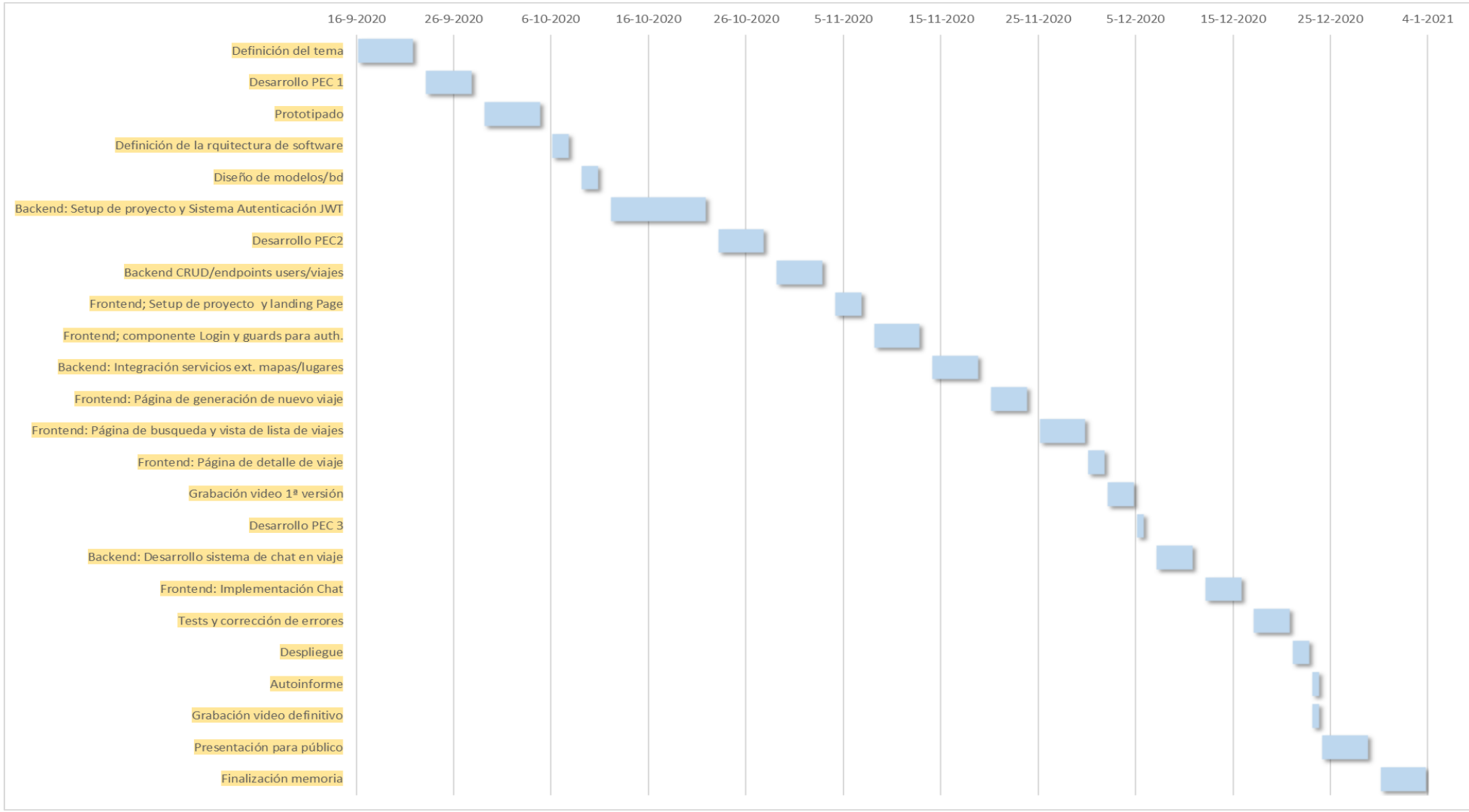


Ilustración 1. Diagrama de Gantt

5. Arquitectura de la aplicación

Se implementará una solución basada en el patrón cliente/servidor, con dos aplicaciones diferentes y una base de datos. Este tipo de arquitectura tiene como mayor ventaja el desacoplamiento entre la capa de visualización y la de datos, así como la posibilidad de disponer de diversos clientes en cualquier tipo de plataforma consumiendo los mismos datos (web, móvil, desktop etc).
De forma más esquemática:

- Servidor: Gestiona y expone los datos como recursos protegidos.
- Cliente: Hace peticiones al servidor para interactuar con estos recursos protegidos.

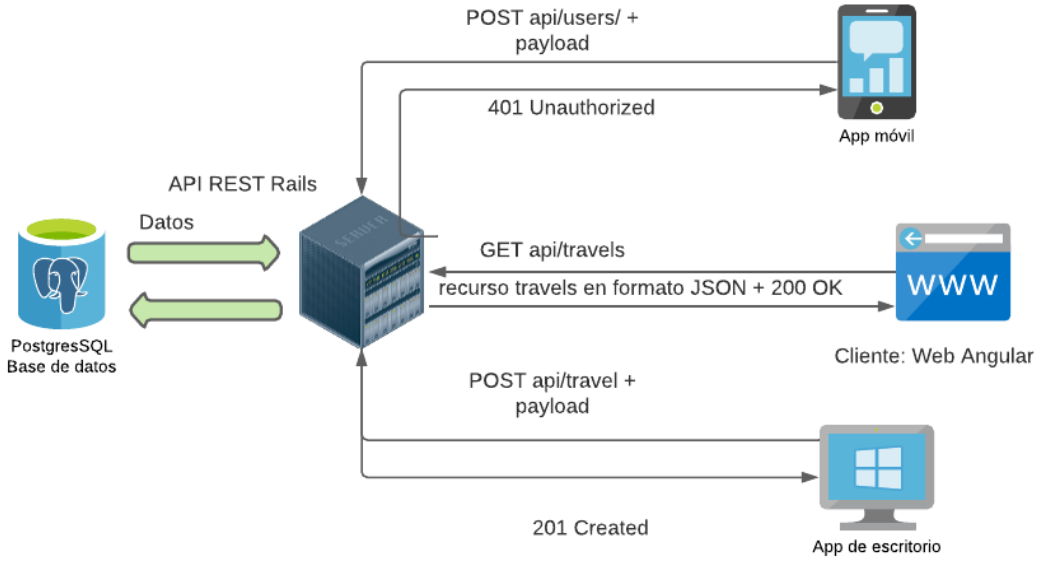


Ilustración 2. Esquema básico de arquitectura y uso potencial de un mismo backend por distintos clientes.

5.1 Lado del servidor o backend.

En el lado del servidor, tendremos una aplicación desarrollada con Ruby On Rails que consistirá en una API REST que responderá a peticiones del cliente con objetos JSON y códigos de estado. Esta aplicación se ocupará de gestionar la lógica de negocio y la comunicación con la base de datos, así como la comunicación con los diversos servicios externos que la aplicación necesita para hacer geolocalización.

Rails es un framework web en lenguaje Ruby muy usado, con una gran comunidad y ecosistema detrás que facilita encontrar soluciones a los retos más típicos encontrados en el desarrollo de una aplicación, así como de ocuparse ya de salida de muchos aspectos

relacionados con la seguridad. Empresas tan importantes como Github, Shopify o Basecamp lo usan actualmente en entornos de producción.

De forma breve, alguna de las características más reseñables de este framework son:

- Patrón MVC (modelo-vista-controlador)
- Filosofía “convention over configuration”. Se sacrifica la flexibilidad para construir una herramienta integral donde el desarrollo se facilita en gran medida siempre que el desarrollador se adhiera al camino marcado por los desarrolladores de Rails (el “*Rails way*”)
- Uso del patrón *ActiveRecord.* La comunicación con la base de datos se realiza mediante una abstracción usando Active Record, que mapea las tablas y campos de la base de datos en objetos Ruby y evita la necesidad de escribir código SQL directamente.
- En su versión 6, usa el weberver *Puma.*

Está aplicación Rails se comunicará con la base de datos PostgresSQL que se ocupará de la persistencia de datos.

5.2 Lado del cliente o frontend

En el lado del cliente, se dispondrá de una aplicación Angular 10 que cargará archivos estáticos HTML, CSS y JavaScript desde el servidor y los completará con datos que se conseguirán mediante peticiones http de tipo AJAX a la API desarrollada con Rails y explicada en el punto anterior. Está aplicación será del tipo SPA (*Single Page Application*).
Las características esenciales de Angular son las siguientes:

- Salvando las distancias, su filosofía es similar a la de Rails, proporcionando de serie herramientas y soluciones elegidas por los desarrolladores del framework para llevar a cabo las acciones más típicas en una aplicación web: formularios reactivos, cliente http, directivas etc.
- Multiplataforma, pudiendo desarrollar *Progresive Web Apps.*
- Arquitectura MVC.
- Uso de TypeScript (superset de Javascript con tipos) y RxJS. (librería que implementa la programación reactiva y los observables en Javascript o TypeScript, en este caso).

5.3 Autenticación/Autorización

Como esquema de autenticación/autorización se usará el estándar JWT (JSON WEB Tokens), ampliamente usado en aplicaciones con una arquitectura similar a la que se implementa aquí: API REST + SPA.

JWT es un estándar de código abierto. El esquema fundamental de uso es el siguiente: El cliente envía las credenciales de acceso o login al servidor. El servidor determina si las credenciales son correctas, y en caso de que lo sean, envía un token (un jwt) al cliente. El cliente almacena este token (por ejemplo en localStorage), y posteriormente adjunta este token en todas las peticiones a recursos protegidos que haga al servidor (por ejemplo, como *Authorization Header* de la petición). El servidor determina la autenticidad del token y permite el acceso al recurso si aplica, enviando de vuelta la respuesta en formato JSON. Estos tokens pueden ser invalidados e incluidos en una lista negra en cualquier momento, y tienen un periodo de validez determinado.

Los jwt se componen de estas partes:

- Header: Declara el tipo, que es “JWT” y el algoritmo de encriptación.
- Payload: Información adicional que el token puede contener. Cada parte de información se denomina “*claim*”, y las hay de tipo “registrado” (tienen nombre reservado según el estándar), “públicas” (las que crea el desarrollador de la aplicación, nombre, email etc.) o privadas.
- La firma. Permite verificar la autenticidad del token por parte del servidor.

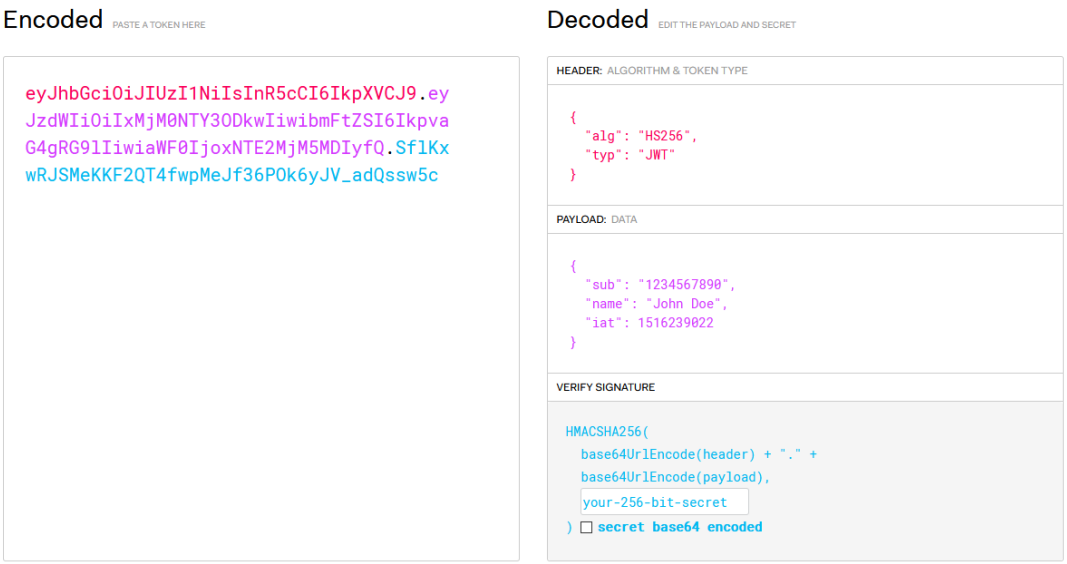


Ilustración 3. Anatomía de un JWT (de jwt.io).

Algunas ventajas del uso de esta estrategia de autenticación es:

- No tiene estado: El backend solo necesita verificar la autenticidad del token, no necesita guardar sesiones y cada solicitud se verifica independientemente del resultado de la anterior. Cada token es independiente y autónomo, y el servidor solo tiene que firmar y verificar, no almacenar. Esto hace que nuestra aplicación sea más escalable ya que necesita menos recursos en el servidor.

- El JWT permite incorporar datos públicos que use el frontend. Por ejemplo, una url con el avatar, el nombre de usuario etc. Es importante no enviar ningún dato sensible como parte de las claims públicas.
- Permite usar el mismo esquema de autorización tanto en plataforma web como en una hipotética aplicación móvil, o al menos de una forma mucho más sencilla que la clásica basada en cookies e id de sesión.
- Mayor facilidad de implementación en escenarios de diferentes dominios que utilizando cookies.

6. Casos de uso

A continuación se muestra un diagrama de caso de uso de la función básica de la aplicación:

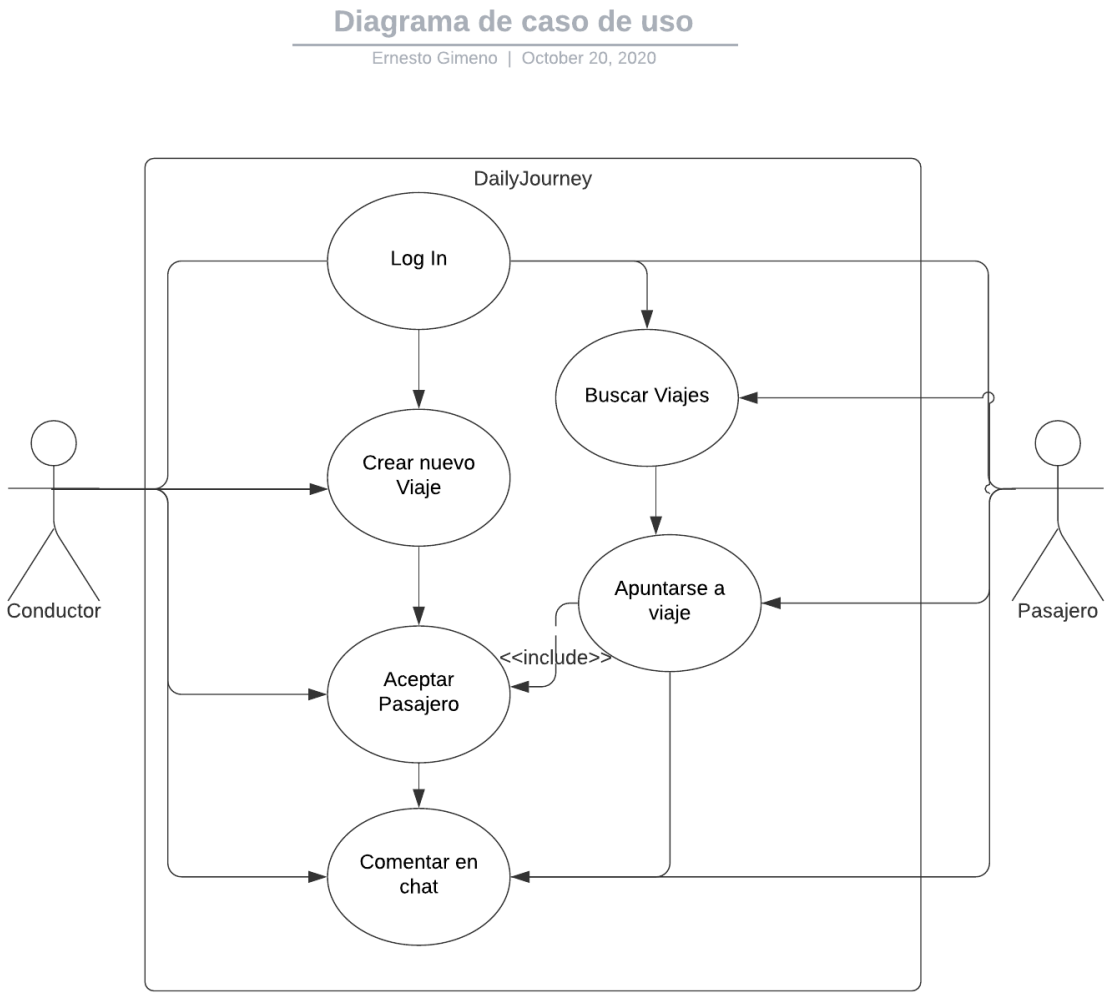


Ilustración 4. Casos de Uso

La funcionalidad esencial de la que dispondrá la aplicación se enumera a continuación:

- Un usuario puede registrarse, acceder con sus credenciales y salir de la aplicación cerrando la sesión.
- Un usuario puede proponer un nuevo viaje, indicando su dirección de partida, plazas disponibles, hora de salida y punto de llegada.
- Un usuario puede buscar viajes disponibles buscando por lugar de llegada, en un radio determinado de km, y visualizarlos en un mapa.
- Un usuario puede apuntarse a un viaje disponible.
- Un usuario creador de un viaje puede confirmar o no la plaza de un usuario que se ha apuntado a su viaje.
- Un usuario puede ver una vista de detalle de un viaje que ha creado o al que se ha apuntado.
- En la vista de detalle de viaje puede haber un chat en el que los viajeros apuntados comenten incidencias de cualquier tipo entre ellos.

- Un usuario que crea un viaje puede aportar detalles extra a través de un campo de observaciones.

La técnica de los casos de uso consiste en describir como los usuarios realizan las acciones de la aplicación. Usando el punto de vista del usuario, se hace un esbozo del comportamiento del sistema y su respuesta ante una solicitud. A continuación, se detallan algunos de los casos de uso más interesantes dentro de la aplicación:

Caso de uso: Crear un nuevo viaje	
Descripción	Un usuario registrado y logueado crea un nuevo viaje en el sistema.
Actores	Conductor
Precondiciones	Ser un usuario registrado y haberse logueado previamente
Curso del caso de uso	
1. Un usuario pulsa en el link del navbar "Nuevo Viaje" 2. Se navega a la ruta de creación de viaje y se muestra un formulario con todos los campos necesarios para generar un viaje. 3. El usuario imputa los datos. 4. El usuario pulsa el botón "Crear viaje". 5. Si el viaje se ha podido guardar correctamente, se muestra snackbar de notificación de éxito y se navega a la página de detalle del viaje 6. Si no se ha podido guardar, se muestra un mensaje de error con detalles	
Postcondiciones	Se crea un nuevo viaje en la base de datos y el viaje aparece en la lista de viajes del usuario de su perfil

Tabla 2.Caso de uso: Crear un nuevo viaje

Caso de uso: Modificar Datos de Perfil	
Descripción	Un usuario registrado y logueado modifica sus datos de perfil
Actores	Conductor y Participante
Precondiciones	Ser un usuario registrado y haberse logueado previamente
Curso del caso de uso	
1. Un usuario pulsa en link de su perfil en navbar. 2. Selecciona la opción "Editar perfil". 3. Se navega a la ruta de edición de perfil. 4. Se escoge la pestaña "Datos generales" 5. Se muestra formulario de edición de datos generales. 6. El usuario puede modificar su nombre en el input de usuario. 7. El usuario puede modificar su avatar. 8. El usuario guarda los nuevos datos. 9. Si la operación ha tenido éxito, se muestra snackbar de notificación de éxito. 10. . Si no se ha podido guardar, se muestra un mensaje de error con detalles	
Postcondiciones	Los datos de usuario son los modificados.

Tabla 3.Caso de uso: Editar perfil de usuario

Caso de uso: Aprobar el registro de un usuario en un viaje	
Descripción	Un conductor puede aprobar la solicitud de un usuario para participar del viaje
Actores	Conductor
Precondiciones	Ser creador de un viaje y que un usuario se haya registrado en ese viaje para ser participante
Curso del caso de uso	
1. Un usuario viaja a la página de detalle del viaje (por ejemplo desde su perfil) 2. Se navega a la ruta de del viaje de detalle. 3. Se pulsa el botón "aceptar" en la línea correspondiente al participante dentro de la tabla de "Viajeros" 4. Si la operación ha tenido éxito, se muestra snackbar de notificación de éxito. 5. Si no se ha podido guardar, se muestra un mensaje de error con detalles.	
Postcondiciones	El participante queda aprobado y cuenta para el cálculo del aforo de viaje.

Tabla 4.Caso de uso: Aprobar el registro de un usuario en un viaje

7. Diagrama de la base de datos

En el diagrama a continuación, se muestra de forma esquemática el diseño de la base de datos así como las relaciones entre los modelos.

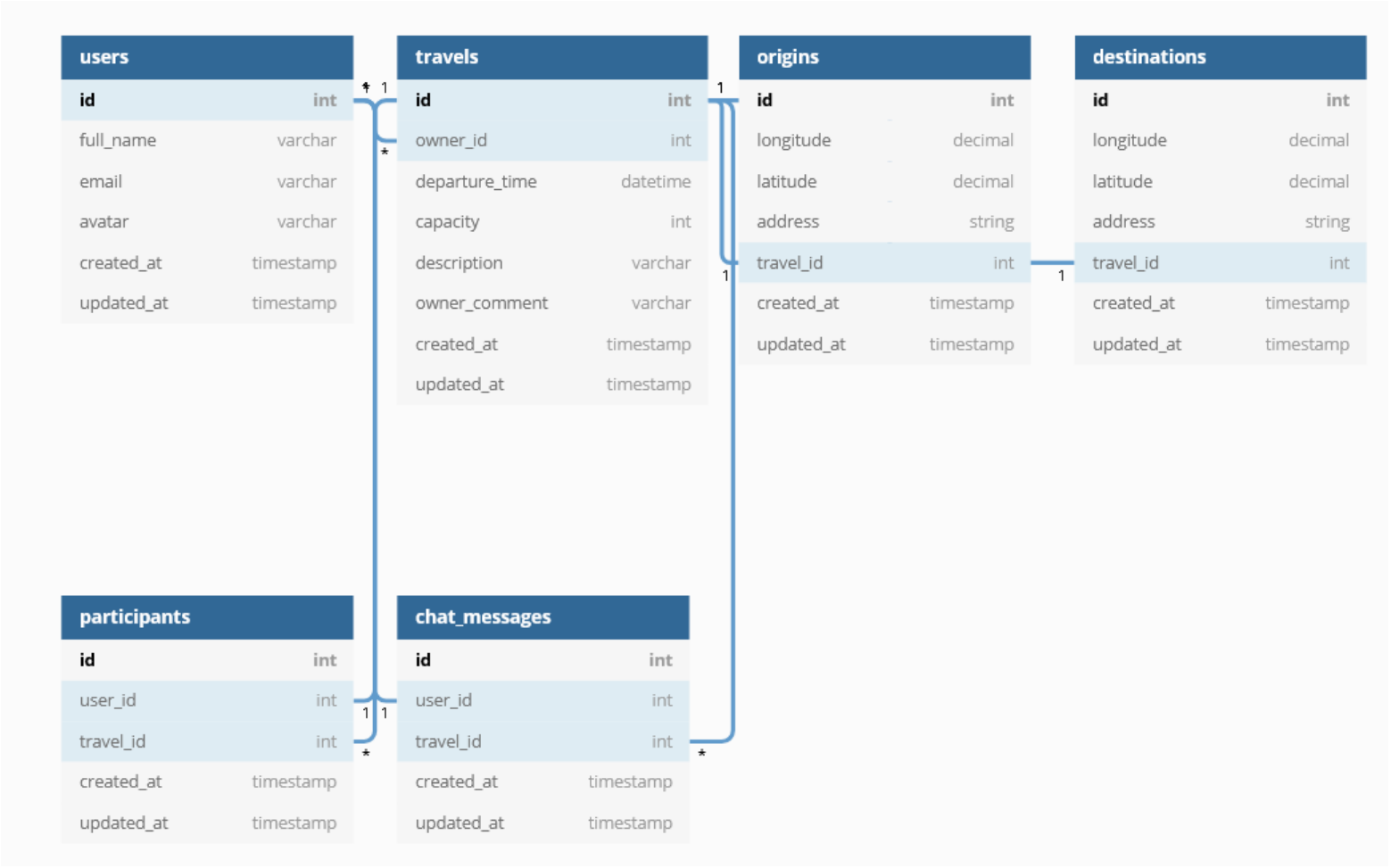


Ilustración 5. Diagrama de modelos y relaciones

8. Prototipos

En los próximos puntos se muestran los prototipos de alta fidelidad de las páginas esenciales en formato desktop. Para la confección de los prototipos se ha usado Figma.

8.1 Landing Page

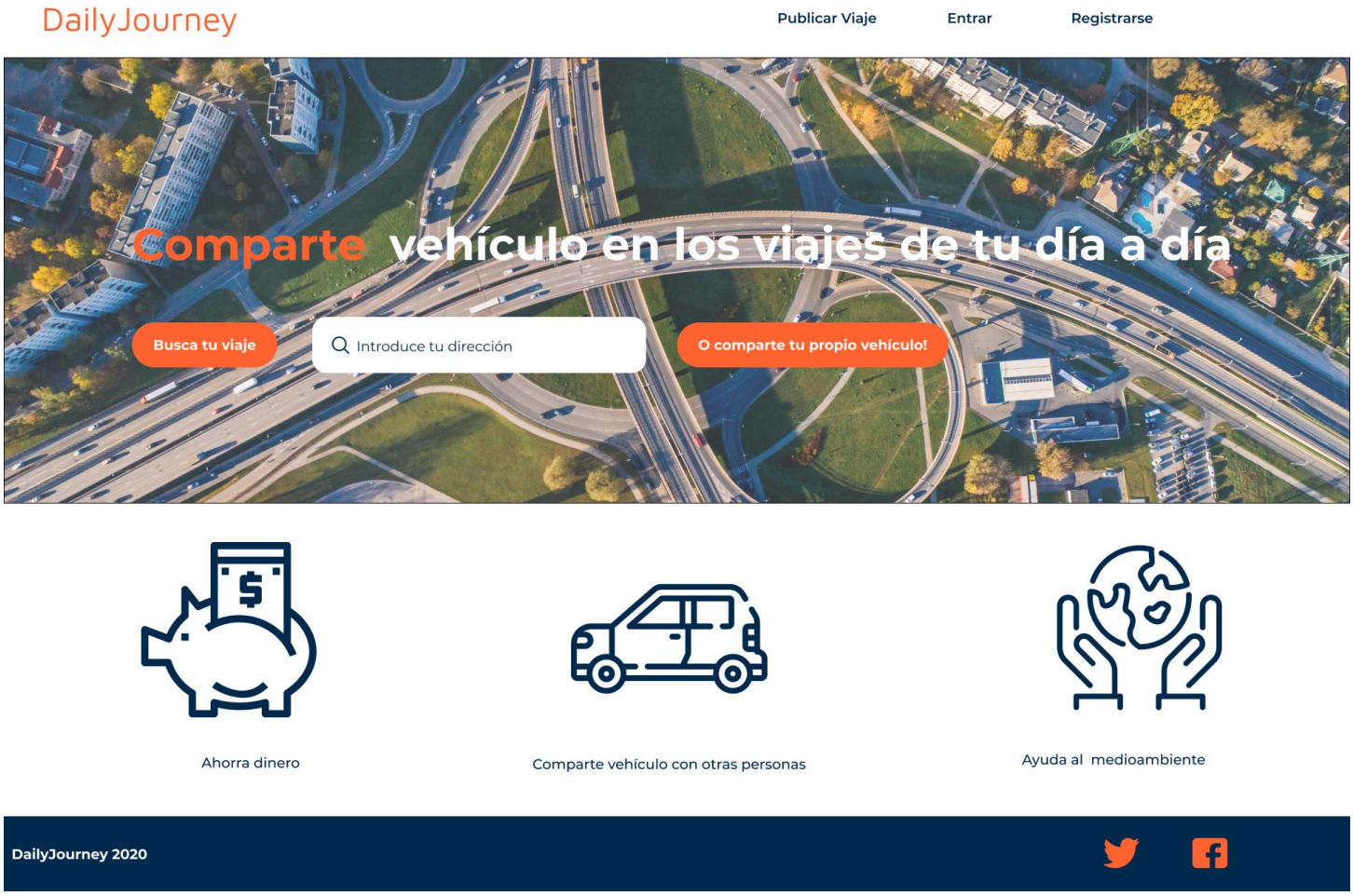


Ilustración 6. Prototipo de la Landing Page

8.2 Página de generación de nuevo viaje

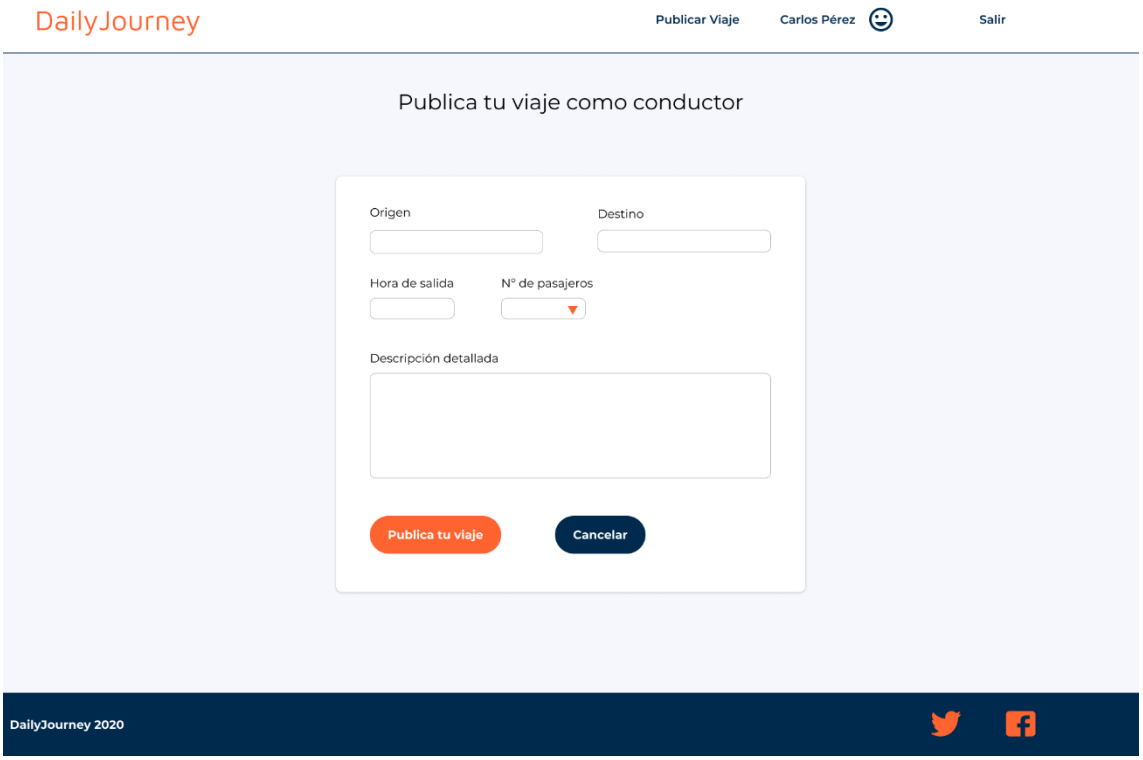


Ilustración 7. Prototipo de página para creación de nuevo viaje.

8.3 Búsqueda de viajes

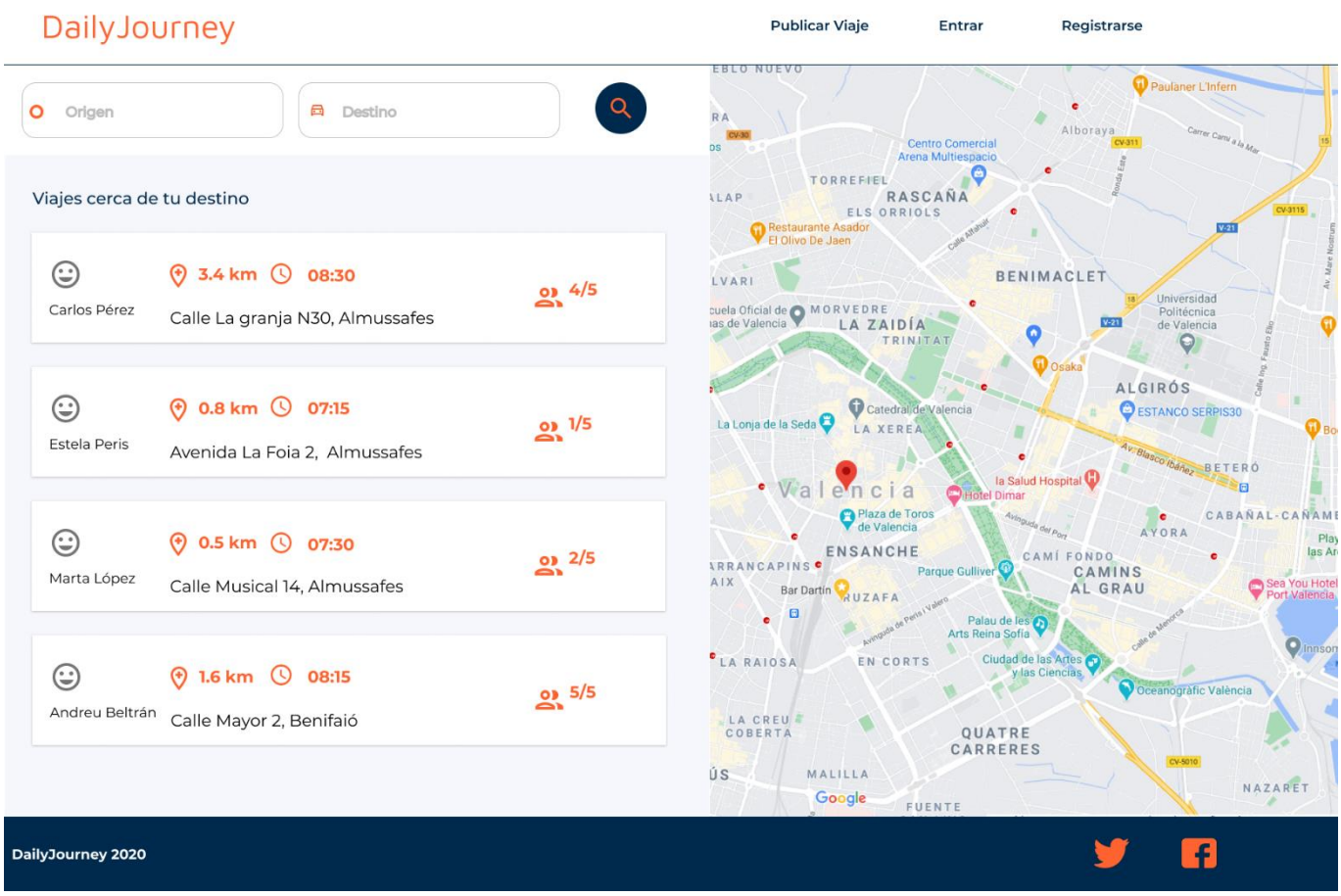


Ilustración 8. Búsqueda de viajes

8.4 Página de detalle de viaje

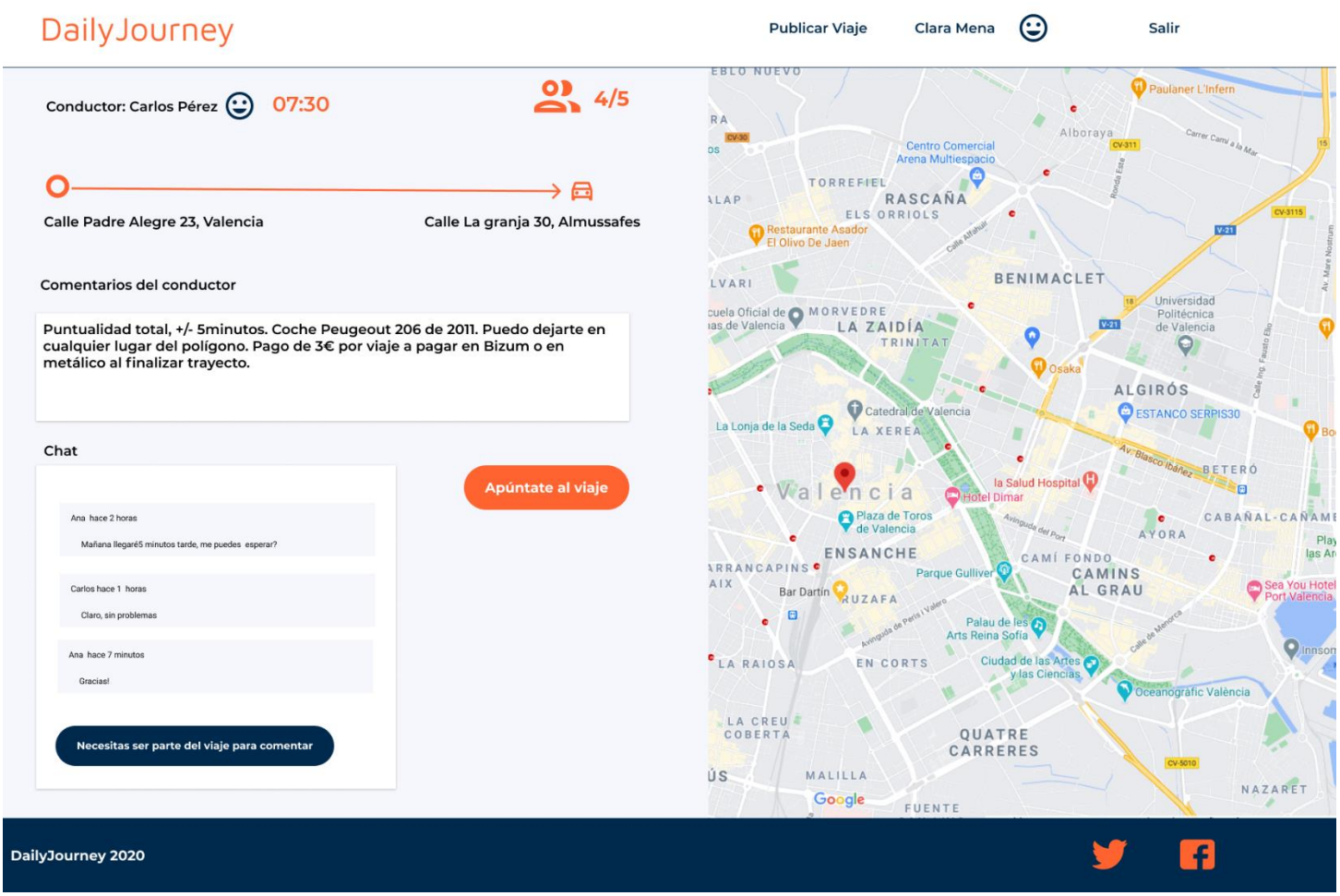


Ilustración 9. Prototipo de página de detalle de viaje

9. Plataforma de desarrollo

En el desarrollo del presente proyecto se han utilizado los siguientes recursos:

Elementos de Hardware

- Intel Core i7 4770
- 8Gb RAM DDR3
- Disco SSD 512Gb.

Sistema operativo

Se ha desarrollado utilizando el Windows Subsystem Linux 2 (WSL2), en concreto con una distribución Ubuntu 20.04 LTS.

Elementos de Software

- Visual Studio Code como editor de código con diversos plugins relacionados con el desarrollo de Ruby, Rails, Javascript y Angular, así como formateadores de código como Prettier.
- Insomnia como herramienta para testear las peticiones HTTP al api del backend.
- Git y Github para control de versiones y apoyo al despliegue.

Características del servidor de despliegue

- Backend Rails: Debe desplegarse en un servidor Linux con PostgresSQL instalado. Se despliega en Heroku, que utiliza el stack Heroku 20 con Ubuntu 20.04.
- Frontend: Al ser una aplicación que consta solo de ficheros estáticos puede desplegarse en cualquier servidor. Concretamente se despliega en Netlify.

10. APIs utilizadas.

Resulta indispensable la integración con un servicio de mapas para poder llevar a cabo la validación de las direcciones que los usuarios introducen y para convertirlas a parámetros de longitud y latitud con que se pueda trabajar posteriormente para calcular distancias y posicionar marcadores en los mapas.
Existen muchas alternativas comerciales en el mercado, siendo posiblemente la más conocida Google Places API, pero en este caso se ha decidido utilizar el servicio Here Maps: https://developer.here.com/.
Los motivos por los que se ha tomado esta decisión son esencialmente dos:

- Facilidad de uso, con una documentación completa y posibilidad de utilizar su API REST, además de diversos SDK.
- Especialmente, su generoso plan gratuito “Freemium”, que permite llevar a cabo una gran cantidad de peticiones sin necesidad de ingresar ningún método de pago.

El endpoint principal del que se hace uso es el llamado “Discover” dentro de la suite de endpoints llamada “Geocoding and Search API v7”:

Este endpoint procesa una *query*(consulta) en formato texto libre y devuelve una lista de lugares de diversos tipos (entre ellos direcciones”, en orden de relevancia.

Una petición típica sería un GET *request* a la url:
https://discover.search.hereapi.com/v1/discover

La petición recibe varios parámetros, tanto obligatorios como opcionales:

- q: Texto libre a buscar.
- at: coordenadas que especifican el contexto de búsqueda para ayudar a ordenar por relevancia los lugares devueltos.
- limit: Número de resultados máximos a devolver.

También es necesario añadir a la petición la apiKey, que obtenemos al dar la aplicación de alta en el panel de control de Here. Dado que esta clave es secreta no se deben hacer las peticiones directamente desde el navegador, en el lado cliente, ya que quedaría expuesta, por ello se implementa un servicio en el backend al que se llamará desde el cliente, y que será el que gestione las solicitudes directamente con la plataforma Here. En el desarrollo de este servicio se ha implementado el patrón de diseño Adaptador, que se detallará después.

11. Instrucciones de instalación/despliegue.

11.1 Instrucciones de Instalación. Backend.

Se detallan las instrucciones para obtener un entorno de desarrollo completo en Ubuntu 20.04 que permite lanzar un servidor Rails con la API REST completamente funcional. Teóricamente Rails puede ejecutarse también en otros sistemas operativos, pero convencionalmente se utilizan sistemas UNIX, ya que tanto el framework como las herramientas están originalmente adaptadas a estos sistemas, y desarrollar Rails en Windows no es aconsejable porque pueden aparecer problemas y errores inesperados, e incluso puede haber gemas (librerías) que no sean compatibles. También está ganando mucha tracción el desarrollo de Rails en contenedores Docker.

- Instalación de dependencias esenciales

```
sudo apt install curl
curl -sL https://deb.nodesource.com/setup_12.x | sudo -E bash -
curl -sS https://dl.yarnpkg.com/debian/pubkey.gpg | sudo apt-key add -
echo "deb https://dl.yarnpkg.com/debian/ stable main" | sudo tee /etc/apt
/sources.list.d/yarn.list

sudo apt-get update
sudo apt-get install git-core zlib1g-dev build-essential libssl-dev
libreadline-dev libyaml-dev libsqlite3-dev sqlite3 libxml2-dev
libxslt1-dev libcurl4-openssl-dev software-properties-common libffi-
dev nodejs yarn
```

- Instalación de Ruby. Versión 2.7.2 recomendada. Se instala el gestor de versiones rbenv y posteriormente Ruby:

```
cd
git clone https://github.com/rbenv/rbenv.git ~/.rbenv
echo 'export PATH="$HOME/.rbenv/bin:$PATH"' >> ~/.bashrc
echo 'eval "$(rbenv init -)"' >> ~/.bashrc
exec $SHELL

git clone https://github.com/rbenv/ruby-build.git ~/.rbenv/plugins/ruby-
build
echo 'export PATH="$HOME/.rbenv/plugins/ruby-
build/bin:$PATH"' >> ~/.bashrc
exec $SHELL

rbenv install 2.7.2
rbenv global 2.7.2
ruby -v
```

- Instalación de bundler

```
gem install bundler
```

- Configuración de Git y Github. Nuestro código estará alojado en Github y por tanto es necesario configurar nuestro entorno de trabajo. Se configura también el certificado ssh:

```
git config --global color.ui true
git config --global user.name "YOUR NAME"
git config --global user.email "YOUR@EMAIL.com"
ssh-keygen -t rsa -b 4096 -C "YOUR@EMAIL.com"

cat ~/.ssh/id_rsa.pub
ssh -T git@github.com
```

- Instalación de Rails

```
gem install rails -v 6.0.3.4
```

- PostgresSQL. Rails trae de serie una base de datos SQLLite para desarrollo, pero es preferible usar postgres en desarrollo ya que de esta manera el entorno de desarrollo y el de despliegue son lo más parecido posible, evitando de esta manera problemas posteriores.

```
sudo apt install postgresql-12 libpq-dev
```

Creación del usuario postgres:

```
sudo -u postgres createuser ernesto -s

# Si se desea añadir un password al user
sudo -u postgres psql
postgres=# \password ernesto
```

Podemos levantar nuestro servicio postgres con el comando

```
sudo service postgresql start
```

Una vez tenemos todas las herramientas listas, se debe obtener el código del repositorio:

```
git clone https://github.com/EGimenoS/dailyjourneyapi.git
```

A continuación, y estando dentro de la carpeta del proyecto, deben descargarse e instalarse las dependencias de la aplicación (las gemas de Ruby, librerías):

```
bundle install
```

En este punto, queda crear la base de datos, aplicar las migraciones para generar las tablas y campos de esta y llenarla de datos:

```
rails db:create
rails db:migrate
rails db:seed
```

Finalmente, en este punto, podemos levantar nuestro servidor local de Rails con la aplicación, accesible en localhost:3000 por defecto:

```
rails s
```

Para la generación de datos falsos, se ha hecho uso de la gema Faker. Las instrucciones de generación de estos datos son como sigue, generando tanto usuarios como viajes.

```ruby
# seed users
100.times do
  User.create(
    email: Faker::Internet.free_email,
    name: Faker::Name.name,
    avatar: Faker::Avatar.image,
    password: 'test1234',
    password_confirmation: 'test1234'
  )
end

# seed travels around Valencia
500.times do
  origin_latitude = rand(39.16975..39.56975)
  origin_longitude = rand(-0.408723..-0.37639)
  destination_latitude = rand(39.16975..39.56975)
  destination_longitude = rand(-0.408723..-0.37639)
  origin_address = Geocoder.search([origin_latitude, origin_longitude]).first.address
  destination_address = Geocoder.search([destination_latitude, destination_longitude]).first.address
  Travel.create(
    departure_time: '08:30',
    owner_comment: Faker::Lorem.paragraph(sentence_count: 2, supplemental: true),
    capacity: Faker::Number.between(from: 1, to: 4),
    owner_id: Faker::Number.between(from: 1, to: 100),
    origin_attributes: { address: origin_address, longitude: origin_longitude, latitude: origin_latitude },
    destination_attributes: { address: destination_address, longitude: destination_longitude, latitude: destination_latitude },
    periodicity: 'Entre Semana'
  )
end
```

Ilustración 10 Detalle de fichero seeds.rb

Para poder acceder a todas las funcionalidades de la app es necesario disponer de una cuenta de desarrollador en el servicio Here Maps, o bien modificar el adaptador existente o añadir uno nuevo.

Una vez obtenida la apiKey de Here, es necesario configurarla como variable ENV.

El código del adaptador se encuentra en la carpeta app/adapters, en el fichero adapter.rb

En el controlador basta con instanciar un objeto del adaptador (en el ejemplo la clase HereMaps) y hacer uso del método autosuggest_addresses del adaptador que deseemos:

```ruby
class Api::V1::SearchAddressesController < Api::V1::BaseController
  def index
    url = 'https://discover.search.hereapi.com/v1/discover'
    results = Adapter::HereMaps.new(url, params['q'], params['at']).autosuggest_addresses
    render_object(results, :ok)
  end
end
```

Ilustración 11. Ejemplo de donde modificar llamada al adaptador correspondiente

11.2 Instrucciones de Instalación. Frontend.

Se proporcionan instrucciones de instalación para Angular. En este caso, no resulta tan crítica la elección de sistema operativo, y no suele haber problemas al desarrollar en Windows. En cualquier caso, en el presente proyecto el frontend se ha trabajado también con Ubuntu 20.04 en el WSL2 de Windows 10.

- Instalación de Node y npm. Si no tenemos instalado Node/npm, es imprescindible, y aunque es posible hacerlo por varias vías distintas, es aconsejable instalarlo mediante su gestor de versiones nvm, ya que permite disponer de varias instalaciones de Node a la vez lo que puede ser conveniente cuando trabajamos con diferentes proyectos:

```
wget -qO- https://raw.githubusercontent.com/nvm-sh/nvm/v0.37.2/install.sh | bash
source ~/.bashrc
nvm install v14.2.0
```

- Instalación de Angular. Mediante Angular CLI.

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

- Descargamos el proyecto desde git:

```
git clone https://github.com/EGimenoS/dailyjourney-angular.git
```

- Posicionándonos en la carpeta del proyecto, instalamos las dependencias:

```
npm install
```

Dependiendo de a qué versión del backend queramos apuntar (en un principio, la versión corriendo en nuestro server local o la desplegada, pero en el futuro también potenciales versiones diferentes del mismo), debemos adecuar los valores de los archivos environment.ts y environment.prod.ts.

Finalmente, para levantar el servidor:

```
ng serve
```

11.3 Instrucciones de despliegue. Backend.

Existen múltiples opciones para desplegar una aplicación Rails, de diferente coste y nivel de complejidad, pero generalmente todas las opciones se resumen en dos vías:

1. Utilizar un servicio PaaS (*platform as a service*, o plataforma como servicio) que abstraiga la complejidad inherente al despliegue a cambio de limitar la flexibilidad y encarecer el mantenimiento a largo plazo.
2. Gestionar nuestro propio servidor Linux, bien en local gestionando físicamente los servidores o más comúnmente en un VPS como Digital Ocean, AWS Beanstalk, Linode etc.

Existe una opción intermedia que es utilizar contenedores que se alojan en VPS o server propio. Incluso hay iniciativas open source que intentan aportar los beneficios de la simplicidad de desplegar en PaaS con los de reducción de coste del VPS, como Dokku,

Para este proyecto, se decide desplegar en PaaS, en este caso Heroku: www.heroku.com
Heroku es un clásico en los despliegues Rails, ya que simplifica enormemente el proceso, y en estadios iniciales el coste es reducido e incluso 0. Es una vía habitual en muchas de las startups que utilizan Rails comenzar en Heroku para evitar el gasto de tiempo y recursos que supone configurar un VPS o aprender lo necesario de gestión de servidores, para posteriormente y con el aumento de tráfico y recursos moverse a otras opciones.

El primer paso es crear una cuenta en Heroku: https://signup.heroku.com/

La manera más sencilla de desplegar en Heroku es utilizando un repositorio Github.

- Instalar Heroku Toolbelt:

```
sudo snap install --classic heroku
```

- Desde el directorio de la app, creación de la nueva app en Heroku. Este comando añade “heroku” como remoto de git.

```
heroku apps:create dailyjourney-api
```

- Ahora se puede hacer push desde la rama master del proyecto al nuevo remote Heroku:

```
git push heroku master
```

En este momento, se despliega la app en Heroku de forma casi automática: Al detectar que el gemfile contiene la gema pg, incluso se aprovisiona el add-on Heroku Postgres para la base de datos y añade las variables de entorno necesarias. También ejecuta el comando bundle install e instala las dependencias, configura el servidor web, etc. Si no hay ningún problema, la app está desplegada y solo queda ejecutar los comandos apropiados para inicializar la estructura y datos de la base de datos (no crearla, ya que ese paso lo hace Heroku al desplegar)

```
heroku run rails db:migrate
```

```
heroku run rails db:seed
```

En el momento actual, la api del presente proyecto se encuentra desplegada en esta dirección: https://dailyjourney-api.herokuapp.com/api/v1

11.4 Instrucciones de despliegue. Frontend.

El despliegue de una SPA que no hace uso de técnicas de Server Side Rendering es un proceso mucho más sencillo, y los nuevos servicios existentes como Netlify y Vercel lo simplifican aún más.
En el caso que nos ocupa, se despliega en Netlify, y de nuevo se aprovechará el uso de Git/Github para simplificar y automatizar el despliegue:

1. Creación de una cuenta de usuario en Netlify.
2. Tras loguearse, pinchar en link “New Site From Git”.
3. En la página que se abre, se debe seleccionar el proveedor de git, en este caso seleccionar Github.
4. En este punto, debe especificarse lo siguiente:
    a. Rama de git desde la que se desea desplegar, típicamente master.
    b. Comandos que debe ejecutar Netlify para generar la build de producción. Al tratarse de una aplicación Angular, se debe indicar lo siguiente por defecto:
        i. ng build –prod
        ii. dist/dailyjourney
5. Al llevar a cabo estos pasos, se desplegará la primera build, y además cada vez que se haga push a la rama master de Github, se llevará a cabo un despliegue automático de la app.
6. Es necesario realizar un ajuste para que pueda navegarse directamente a rutas que no son index.html sin que se visualice el mensaje “404 Page Not Found”. Al ser la app una SPA, las diferentes rutas son creadas por angular router en el cliente, no existen en el servidor, por lo que hay que indicar que se redireccione a index.html cuando se intente acceder a una ruta de manera directa:
    a. Añadir nuevo fichero en el proyecto Angular src/_redirects, y añadir: “/index.html 200” en este fichero.
    b. Editar fichero angular.json para que este nuevo archivo _redirects se incluya en la build. Por ejemplo, añadiéndolo al array de assets. Hacer commit a master de estos cambios para que tengan visbilidad en Netlify.

En el momento actual la app se encuentra desplegada en: https://dailyjourney.netlify.app

12. Proceso de desarrollo.

En este capítulo, se resumen algunos de los elementos más interesantes a reseñar del desarrollo, así como algunas elecciones de arquitectura y patrones a nivel software.

12.1 Backend

Pasos Iniciales

Generación de la aplicación. La CLI con que cuenta Rails permite generar un esqueleto de la aplicación muy completo, en línea con las convenciones que sustentan el framework. Esta CLI admite opciones para cambiar ciertos detalles de configuración, tales como la base de datos, o el framework de *testing* que se desea utilizar (por defecto con Rails, minitest). Rails también permite un modo "solo api" para no instalar los componentes de Rails que en un proyecto solo api no tienen sentido, tales como por ejemplo el componente ActionView que en un proyecto tradicional se encarga de gestionar las vistas y los *templates (plantillas en formato html.erb)*.
Por tanto, se crea la aplicación con el siguiente comando que indica crear una aplicación con las estas opciones:

- Modo solo API
- Base de datos postgres
- No instalar infraestructura de tests (en el futuro se desea utilizar Rspec en lugar de minitest como framework de *testing*)

```
rails new --api -database=postgresql --skip-test dailyjourney-api
```

Una vez creada la aplicación, puede observarse que de partida Rails ya introduce una arquitectura MVC muy clara en su estructura de carpetas. Se añade además la carpeta "adapters" dentro del folder "app", que Rails precarga, para añadir posteriormente el adaptador del servicio de mapas. El uso de este patrón de diseño de software permite abstraer el código responsable de la llamada a la API, de forma que el controlador no tiene por qué conocer detalles acerca de la implementación del servicio y su integración con la aplicación.
Por último, dentro de estos primeros pasos, debe crearse la base de datos y configurar el archivo database.yml con los datos necesarios de la instalación local postgres. Al indicar durante la creación de la aplicación que se iba a utilizar PostgresSQL, Rails se ha ocupado de incorporar la gema 'pg' y de preconfigurar el archivo database.yml.

Definir esquema de la base de datos

Rails trabaja con migraciones para generar la estructura de la base de datos, en contraposición a la alternativa clásica de trabajar directamente usando comandos SQL sobre la base de datos. Las migraciones funcionan como una especie de control de versiones de la base de datos, siendo algunas de sus ventajas:

- Permite resetear y recrear el estado de la base de datos en cualquier momento.
- Se dispone de una noción clara de en qué estado actual está nuestra base de datos.
- Se migra de manera determinista desde nuestra actual versión de la base de datos a una nueva.
- Es muy sencillo tener sincronizado el estado de la base de datos tanto en local como en producción.

Una vez hemos generado los ficheros de migraciones necesarios indicando los cambios a la base de datos que se desean (creación de tablas, cambios de nombre o tipo en columna, añadir índices etc), pueden ejecutarse estas migraciones, y Rails genera un archivo 'schema.rb' con los detalles:

db > schema.rb

```ruby
ActiveRecord::Schema.define(version: 20_201_129_161_129) do
  enable_extension 'plpgsql'

  create_table 'chat_messages', force: :cascade do |t|
    t.bigint 'travel_id', null: false
    t.bigint 'user_id', null: false
    t.text 'message'
    t.datetime 'created_at', precision: 6, null: false
    t.datetime 'updated_at', precision: 6, null: false
    t.index ['travel_id'], name: 'index_chat_messages_on_travel_id'
    t.index ['user_id'], name: 'index_chat_messages_on_user_id'
  end

  create_table 'destinations', force: :cascade do |t|
    t.float 'longitude'
    t.float 'latitude'
    t.string 'address'
    t.bigint 'travel_id'
    t.datetime 'created_at', precision: 6, null: false
    t.datetime 'updated_at', precision: 6, null: false
    t.index ['travel_id'], name: 'index_destinations_on_travel_id'
  end

  create_table 'origins', force: :cascade do |t|
    t.float 'longitude'
    t.float 'latitude'
    t.string 'address'
    t.bigint 'travel_id'
    t.datetime 'created_at', precision: 6, null: false
    t.datetime 'updated_at', precision: 6, null: false
    t.index ['travel_id'], name: 'index_origins_on_travel_id'
  end

  create_table 'participants', force: :cascade do |t|
    t.bigint 'user_id'
    t.bigint 'travel_id'
    t.datetime 'created_at', precision: 6, null: false
    t.datetime 'updated_at', precision: 6, null: false
    t.index ['travel_id'], name: 'index_participants_on_travel_id'
    t.index ['user_id'], name: 'index_participants_on_user_id'
  end
```

Ilustración 12 Detalle de una parte de schema.rb

Además de disponer de este esquema, es necesario configurar los modelos para que sustenten las asociaciones entre ellos utilizando las propiedades de los modelos de Rails has_many, belongs_to etc.

Añadir gemas necesarias para el desarrollo

Se añaden al archivo 'gemfile.rb' las gemas necesarias para llevar a cabo el desarrollo, en adición a las que Rails ya ha incluido por defecto.

- Gemas del grupo *development*. Son las que solo se instalarán para desarrollo local y no se incluirán en el bundle de producción:
    - Rspec. Framework de testing.
    - Factory_bot_rails. Alternativa a *fixtures* (datos creados automáticamente para inyectar en los tests).
    - Shoulda-matchers: Facilita el testeo de características básicas de Rails, como las validaciones de los modelos.
    - Rubocop: Linter y formateo de código.
    - Awesome-print: Muestra de manera más legible los resultados de llamadas a métodos de ActiveRecord en consola.
- Gemas que si se incluyen en el bundler:
    - Bcrypt: Permite utilizar el método has_secure_password en el modelo User.
    - Api-guard: Gema que maneja los aspectos fundamentales de la autenticación mediante JWT.
    - Rack-cors. Configuración de cors.
    - Active Model Serializers. Permite controlar el cómo se serializan las respuestas de nuestra api.
    - Faker: Generación de datos tipo seed para desarrollo,
    - Figaro: Gestión de variables ENV.
    - Geocoder: Operaciones con direcciones y coordenadas.
    - Rest Client. Cliente para hacer llamadas a API's externas.

Autenticación/Autorización

Tal como se indicó en el apartado dedicado a la arquitectura de la aplicación, se utiliza JWT para la gestión de la autenticación/autorización y protección de recursos de nuestra api. Para facilitar la tarea, se hace uso de la gema 'api_guard' modificando ciertos aspectos necesarios, que se ocupa de los siguientes detalles de implementación del esquema de autenticación:

- Generación de endpoints para registro, login, eliminación y cambio de password del usuario.
- Generación de los tokens jwt que envía en la cabecera de una respuesta exitosa a una solicitud de login o registro, y descodificación de este cuando desde el cliente se adjunta el jwt en una petición.
- Determinación del "current_user" (usuario que ha llevado a cabo la petición), en base al id del jwt recibido. Protección de los recursos solicitados en la petición en base a dicho usuario.
- Gestión del refresco, validez, tiempo de expiración del token.

- Mantenimiento de listas negras de tokens invalidados.
- Configuración los campos que se desea que se integren en el payload del token

Integración con servicio de mapas

Para integrarnos con el servicio de mapas Here utilizamos el patrón adapter como se ha indicado anteriormente. En nuestro caso, definimos una configuración inicial del cliente y un método que llama a la API de Here, recibe los resultados, los formatea apropiadamente y devuelve solo los campos que son necesarios para la operación. En caso de querer disponer de más un proveedor de servicio de mapas, es sencillo añadir otro adaptador que devuelva datos con la misma estructura, de manera que el controlador no debe conocer detalles acerca de la implementación del servicio de mapas ni preocuparse por el hecho de que en un momento dado utilicemos uno u otro. Se añaden a continuación elementos esenciales del código de este adaptador por su interés.

```ruby
module Adapter
  class HereMaps
    API_KEY = ENV['here_maps_api_key']

    def initialize(base_url, query, at = '39.48728,-0.36593')
      @base_url = base_url
      @config = set_config
      @query = query
      @at = at
    end

    def autosuggest_addresses
      req_params = { at: @at, q: @query }
      results = RestClient.get @base_url, { params: @config.merge(req_params) }
      results = JSON.parse(results)
      results['items'].map do |item|
        { address: item['title'], latitude: item['position']['lat'], longitude: item['position']['lng'] }
      end
    end

    private

    def set_config
      { apikey: API_KEY, limit: 5 }
    end
  end
end
```

Ilustración 13. Código de adapter.rb

Desarrollo de CRUD de recursos.

En la presente aplicación se cuenta con diversos recursos que deben permitir diversas operaciones tipo CRUD. En Rails, CRUD (acrónimo para create, read, update, delete), se definen con los métodos de los controladores index, show, update, create y delete, que a su vez se corresponden por convención con determinadas rutas de nuestra aplicación que serán accesibles como endpoints REST desde cualquier cliente. Estos recursos se exponen utilizando módulos y *namespaces* de forma que la api es accesible desde rutas del tipo api/v1,

lo que facilita el versionado y actualización de la api sin romper la funcionalidad para clientes que no quieran o puedan actualizarse.

```
Rails.application.routes.draw do
  api_guard_routes for: 'users', controller: {
    registration: 'users/registration',
    authentication: 'users/authentication',
    passwords: 'users/passwords',
    tokens: 'users/tokens'
  }
  namespace :api do
    namespace :v1 do
      resources :travels
      resource :user, only: [:update]
      resources :search_addresses, only: [:index]
      resources :chat_messages, only: %i[create index]
      resources :participants
      resources :profile_travels, only: [:index]
    end
  end
end
```

Ilustración 14. Rutas de la API que definen los endpoints

11.2 Frontend

Pasos iniciales

Generación de la aplicación. Angular también cuenta con una CLI muy potente que permite generar una estructura base de la aplicación, así como posteriormente los distintos elementos (servicios, componentes, pipes, directivas). Se indican también diversas opciones iniciales: inclusión de routing, uso de scss etc.

Posteriormente, se define y configuran las herramientas necesarias para una adecuada experiencia de desarrollo:

- Prettier y tslint. Prettier es un formateador de código y tslint un *linter*[6]. Aunque muchas veces sus responsabilidades se solapan, nosotros deseamos que prettier se ocupe solo del formato del código, y tslint de detalles relacionados con la calidad de este. Para ello, se hace uso del plugin tslint-config-prettier.
- Husky: Permite añadir una función que formatea y analiza el código antes de cada *commit*.[7]

[6] Herramienta que ayuda a seguir buenas prácticas o guías de estilo en nuestro código.
[7] Confirmar un conjunto de cambios provisionales de forma permanente en el sistema de control de versiones, en este caso Git.

- Se añaden scripts necesarios para poder efectuar estas operaciones manualmente en array de scripts de package.json.
- Se añaden los plugins necesarios al IDE (Visual Studio Code) para que también se ejecuten automáticamente al guardar un archivo.

Después, añadimos la librería Angular Material, que permite el uso de componentes bajo el libro de estilo Google Material adaptados a Angular. En este proyecto no se va a hacer un uso extensivo del mismo ya que no se desea que los estilos sean del tipo “Material”, pero la librería incluye ciertos aspectos muy convenientes especialmente durante la gestión de formularios, así como incluir de partida componentes como modales, tooltips, datepickers o alertas que serán de gran utilidad.

Una vez llevado a cabo esto, se procede a incluir en el fichero de estilos globales elementos tales como:

- Reset CSS.
- Variables del theming (colores, fuentes).
- Estilos de algunos elementos que se usan en toda la aplicación.

Se definen también los archivos “environment.ts” y “environment.prod.ts” para que incluyan las direcciones base de los endpoints a los que los servicios del frontend atacarán

Definición de arquitectura

Se define la arquitectura en base a la siguiente figura:

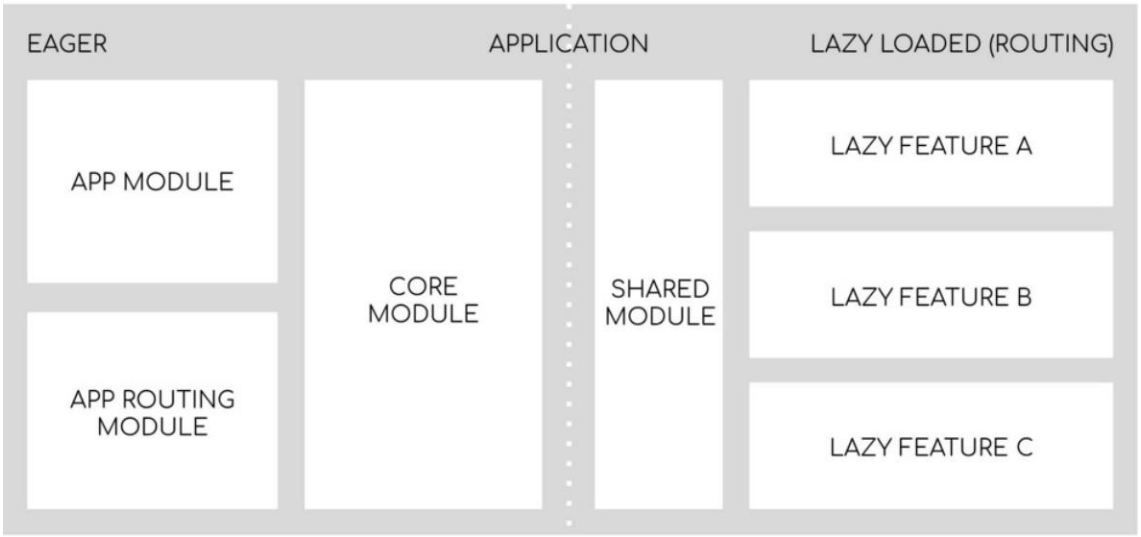


Ilustración 15 Arquitectura de app Angular[8]

Nuestra aplicación constará de dos partes principales.

1. Una que se cargará directamente desde el principio (la parte main.js del *bundle* o paquete de producción que genera Angular). Contendrá el módulo principal

[8] Fuente: https://medium.com/@tomastrajan/how-to-build-epic-angular-app-with-clean-architecture-91640ed1656

AppModule, las rutas principales y un CoreModule que se ocupará de los componentes que conforman el layout principal (navbar, footer), los componentes Login y Register, los servicios que pueden ser inyectados en cualquier componente de la aplicación y otros elementos propios de Angular/TypeScript compartidos en toda la aplicación como las *interfaces*, los *custom pipes* o los *validators*.

2. Otra parte que incluirá los componentes cargados en modo “lazy”, es decir, que solo serán cargados cuando un usuario navegue a las pantallas que integran estas características. También se importará un módulo llamado SharedModule que incluirá los componentes reutilizables usados en más de uno de estos componentes.

La elección de esta arquitectura supone un buen punto medio entre obtener un *bundle* de producción con un tamaño razonable, favoreciendo la primera carga, y una experiencia de desarrollo no excesivamente compleja.

A continuación, se muestra un detalle de las rutas según estructura comentada, así como la estructura general de carpetas:

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { AuthGuard } from './core/guards/auth.guard';

const routes: Routes = [
  { path: '', pathMatch: 'full', redirectTo: 'home' },
  {
    path: 'home',
    loadChildren: () => import('./features/home/home.module').then((m) => m.HomeModule),
  },
  {
    path: 'search-results',
    loadChildren: () =>
      import('./features/search-results/search-results.module').then((m) => m.SearchResultsModule),
  },
  {
    path: 'add-travel',
    loadChildren: () =>
      import('./features/add-travel/add-travel.module').then((m) => m.AddTravelModule),
    canActivate: [AuthGuard],
  },
  {
    path: 'travel-detail/:id',
    loadChildren: () =>
      import('./features/travel-detail/travel-detail.module').then((m) => m.TravelDetailModule),
  },
  { path: '**', redirectTo: 'home' },
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule],
})
export class AppRoutingModule {}
```

Ilustración 16 Detalle de rutas y lazy loading de componentes.

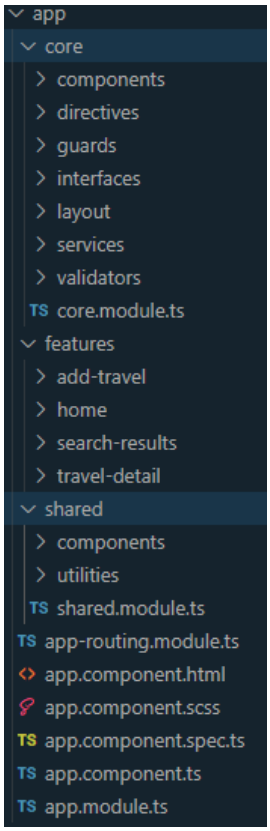


Ilustración 17 Estructura de carpetas de la aplicación Angular

Otras dependencias añadidas

Además de la anteriormente reseñada Angular Material, de las dependencias reseñadas durante la configuración de las herramientas y de las dependencias que vienen de serie con Angular, se añaden las siguientes librerías en formato paquetes npm:

- Ngx-leaflet: Implementación para Angular de la conocida librería de visualización de mapas Leaflet. Se utilizan para visualizar los mapas proporcionados por OSM (Open Street Maps).
- Angular-jwt: Librería que simplifica el proceso de trabajar con jwt en el cliente. Añade por defecto el token en las peticiones a los endpoints que se desee, permite descifrar el contenido de los tokens en el cliente, añadir listas de dominios a los que adjuntar el token etc.

13. Usabilidad/UX

A lo largo de esta sección se detallarán algunos aspectos relativos a la usabilidad de la aplicación, en relación tanto con aspectos del diseño responsive como de patrones de diseño utilizados.

13.1 Pautas y herramientas utilizadas

Como primer punto a tener en consideración, debe apuntarse que el diseño de la aplicación se ha llevado a cabo teniendo en cuenta su uso en distintos dispositivos, siguiendo las siguientes pautas:

- Diseños específicos y optimizados para dos tamaños, desktop/laptop y móvil grande/Tablet pequeña.
- Para tamaños intermedios, se ha podido asegurar la usabilidad gracias al uso extensivo de herramientas como Flex.

Estas pautas se han concretado del siguiente modo:

- Uso de herramientas propias de CSS nativo para gestionar los layouts. Se decidió no incluir librerías muy populares como Bootstrap ya que se considera que las modernas herramientas que de manera nativa incluyen los navegadores permiten generar layouts de manera óptima sin necesidad de añadir nuevas dependencias.
- Uso de media queries con un punto de corte, 768px, utilizando el modelo “mobile first”, de modo que la condición de la media query es “min-width: 768px”.

Se incluye también la librería de componentes Angular Material, con objetivo no de hacer uso extensivo de sus componentes ya que se desea que el diseño sea personal, si no de utilizar algunos de los más convenientes y que mantienen una perfecta interacción con Angular, como por ejemplo ventanas modales, *snackbars* de alerta, menús desplegables o inputs de formulario.

13.2 Patrones de diseño web adaptables

De acuerdo con los patrones identificados originalmente por Luke Wroblewski[9], se han utilizado los siguientes en el desarrollo de esta aplicación.

MOSTLY FLUID

Este patrón se basa en el uso de una cuadrícula fluida. En pantallas grandes o medianas se mantiene el mismo ancho de los elementos y se ajusta el margen, mientras que para pantallas pequeñas los elementos se apilan de manera vertical.

[9] https://www.lukew.com/ff/entry.asp?1514

Este patrón puede observarse en varios lugares de la web, como por ejemplo la pantalla de edición de perfil del usuario, con el formulario central manteniendo el mismo tamaño y ajustándose los márgenes:

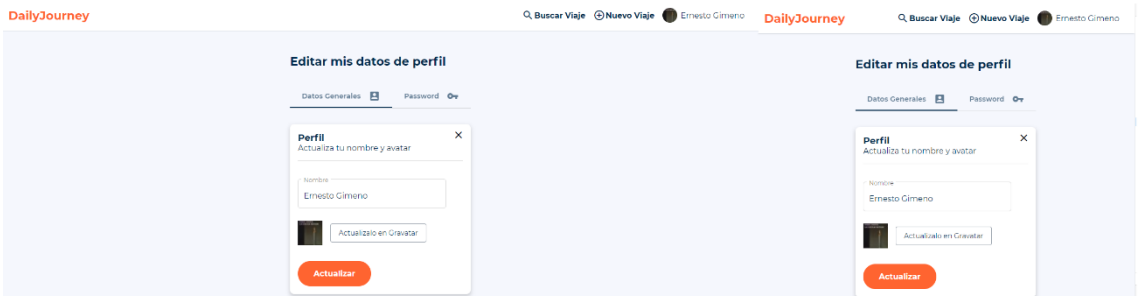


Ilustración 18. Ejemplo de uso de patrón Mostly Fluid en componente UserProfile

COLOCACIÓN DE COLUMNAS

Conforme el ancho de pantalla se hace más reducido, los elementos se apilan verticalmente Un ejemplo de uso de este patrón puede observarse por ejemplo en el formulario de creación de nuevo viaje, con los inputs apilándose verticalmente en dispositivos pequeños:

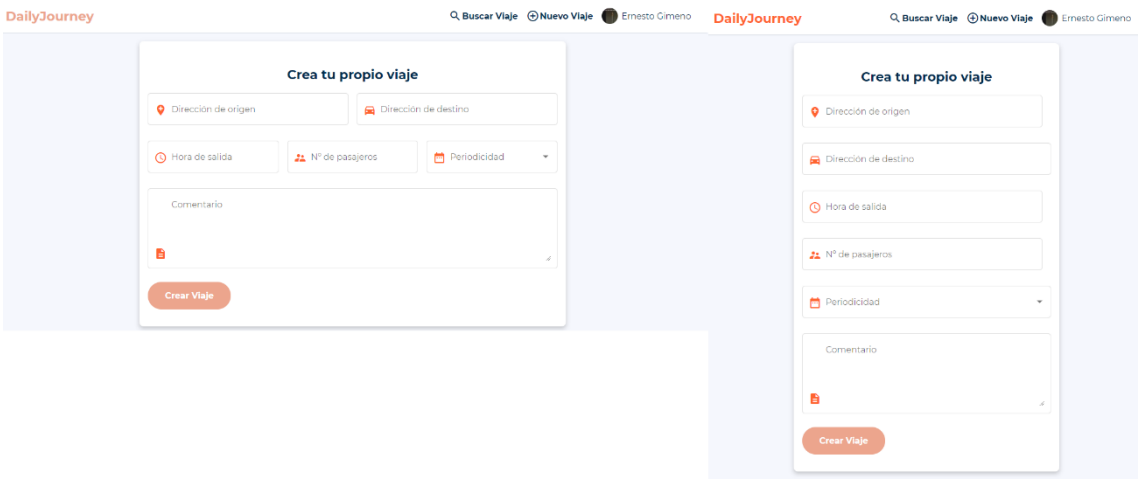


Ilustración 19. Ejemplo de uso de patrón Colocación de columnas en página de creación de nuevo viaje

TINY TWEAKS

En este patrón se ajustan en forma de pequeños cambios elementos como el relleno o el tamaño de las fuentes. Este patrón de usa también en páginas como la de detalle de viaje, para mostrar la información de manera similar, por ejemplo, en la página de detalle de viaje en diversos elementos el *padding* (relleno) o el tamaño de la fuente se reduce en pantallas pequeñas, o no mostrando el avatar de los participantes en el viaje en la vista del conductor para evitar que los elementos queden demasiado constreñidos.

OFF CANVAS

En este patrón ciertos elementos se colocan fuera de pantalla en dispositivos pequeños. Es el caso del mapa en el que se dibujan los markers con los destinos, que no se visualiza en pantallas pequeñas.

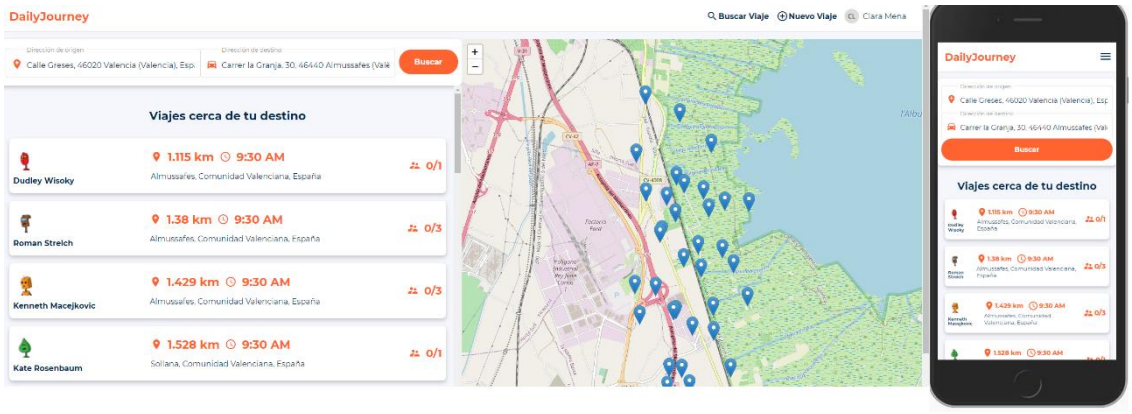


Ilustración 20. Ejemplo de uso de patrón Off Canvas

13.3 Patrones de diseño

El uso de patrones de diseño permite mejorar la experiencia de usuario incluyendo elementos e interacciones a los que el usuario ya está acostumbrado y que identifica y entiende de manera sencilla e incluso en algunos casos a nivel subconsciente. Estos patrones son soluciones reusables a problemas de usabilidad. A continuación, se detallan algunos de estos patrones usados en la aplicación.

Patrones de imputación de datos

- Input Feedback. La aplicación provee de feedback al usuario cuando imputa datos en los campos con necesidad de validación, así como después de enviar la información y ser validada en el servidor.

Ilustración 21. Ejemplo de validaciones en formulario de nuevo viaje

- Time Picker. El usuario usa un selector de hora para imputar una hora válida en el formulario de creación de nuevo viaje.

Ilustración 22. Ejemplo de uso de patrón Timepicker en creación de nuevo viaje

- Formato indulgente. En los inputs de entrada de direcciones, el usuario puede teclear la información en el formato que desee, siendo el sistema el que sugiere localizaciones correctas en función del texto libre imputado por el usuario. Esto permite que el usuario no tenga que preocuparse por teclearse la dirección en un formato determinado (tipo de vía, vía, nombre, número, localidad, provincia etc.).

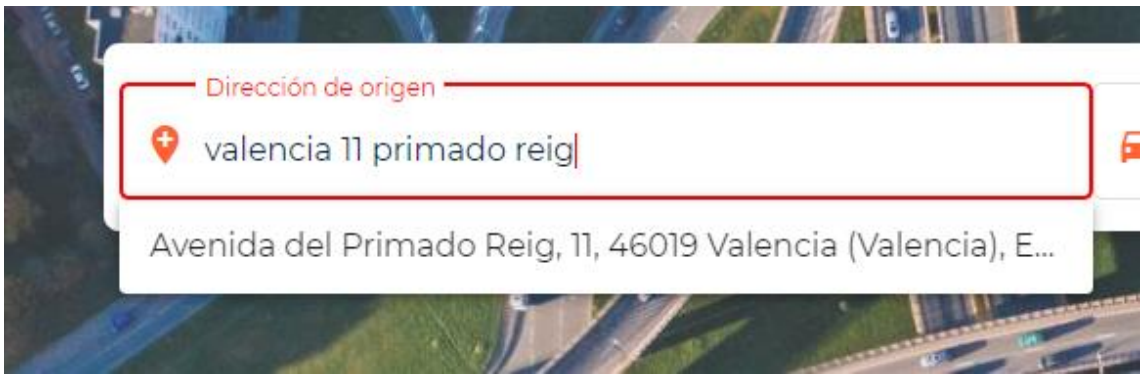


Ilustración 23, Ejemplo de uso del patrón "forgiving format" en formulario de obtención de direcciones

- Controles cambiantes. Este patrón se usa para presentar solo los controles relativos a las acciones disponibles en ese momento al usuario. En la aplicación se usa por ejemplo en la página de detalle de viaje, mostrando solo los controles de edición o eliminación de viaje cuando el usuario es el creador del viaje, o en caso de que el usuario no sea el creador de viaje, mostrando el botón de “Apuntarse en viaje” o “Cancelar Registro” dependiendo del estado previo del usuario.

Patrones de navegación

- Menú “burguer”. En pantallas pequeñas el menú de navegación se colapsa y es accesible clickando en un botón tipo “burguer”:

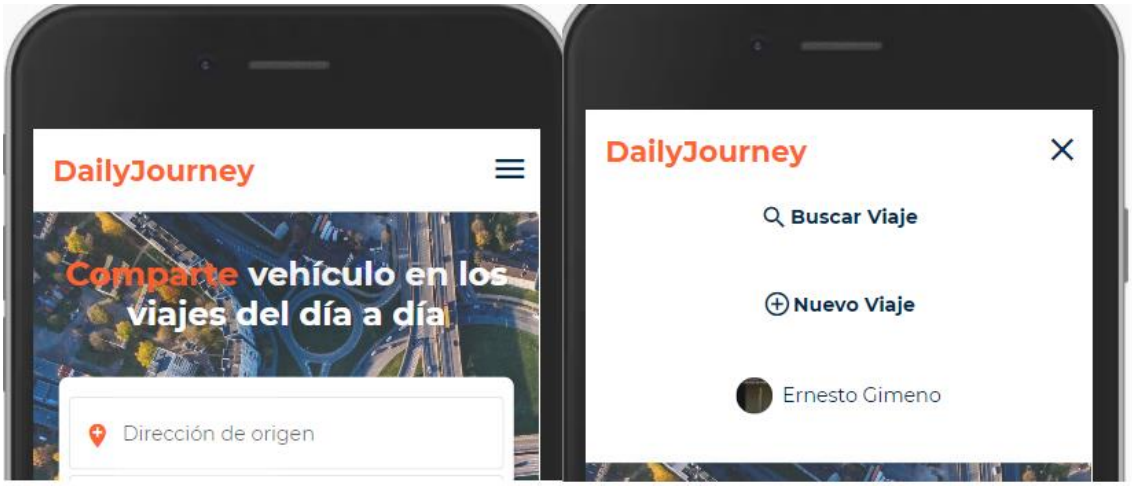


Ilustración 24 Ejemplo de menú burguer en navegación móvil

- Tabs (pestañas) de navegación. El contenido se separa por pestañas. Usado en la pantalla de edición de perfil de usuario.
- Menú desplegable de atajo. Usado cuando el usuario necesita acceder a una porción especifica de funcionalidad de manera rápida saltándose la jerarquía de navegación. Usada en la vista de viajes en los que participa el usuario:

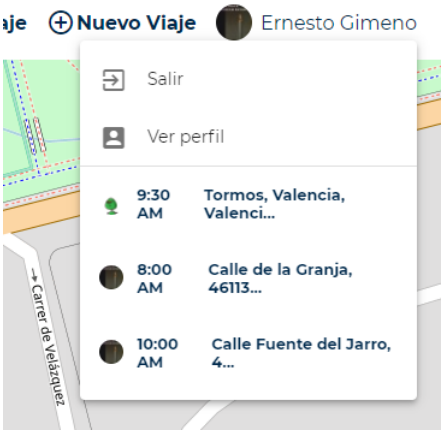


Ilustración 25. Uso del patrón "shortcut dropdown"

- Modal. Usado cuando el usuario necesita tomar una acción y desactivar las demás hasta tomar una decisión. Por ejemplo, modal de login que aparece cuando el usuario intenta acceder a una ruta que necesita de autenticación.

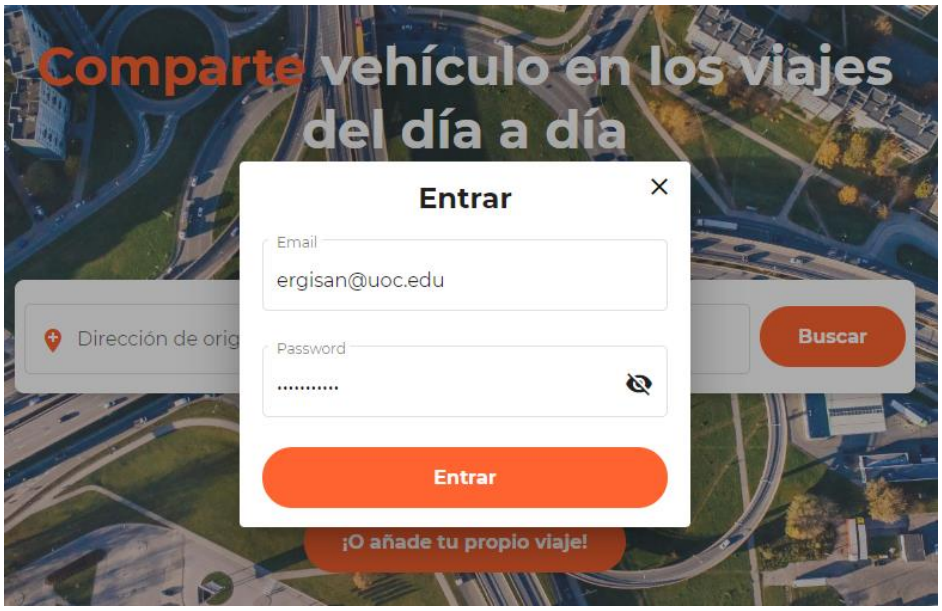


Ilustración 26 Modal de login.

- Link a “home”. Ruta rápida siempre accesible para que el usuario retorne a la página principal o “home”, independientemente de donde está. En la aplicación se encuentra como link en el logo.
- Menú desplegable vertical. Usado cuando el usuario necesita navegar entre 2 o más secciones de la web, pero el espacio es reducido para mostrar todos los links de navegación.

Tratando con datos

- Autocompletar. Usado de nuevo el formulario de imputación de direcciones.

De Interacción social

- Chat. Usado cuando se desea que los usuarios interactúen de forma privada en grupos o con otros individuos dentro del sistema. En la aplicación se usa el chat como elemento de comunicación entre el conductor en los viajes y los participantes que este ha aprobado.

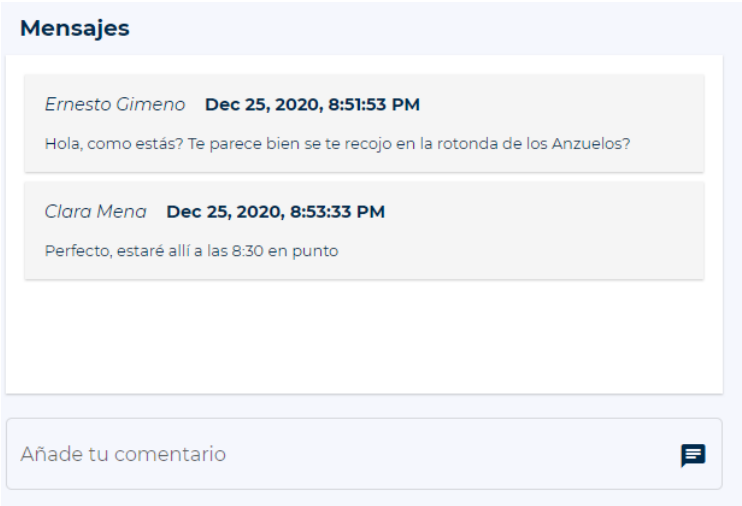


Ilustración 27. Chat en página de detalle de viaje

De onboarding[10]

- Registro “vago”. Se permite al usuario probar y usar inmediatamente algunas de las características sin necesidad de registro formal, reduciendo barreras de entrada. En el caso de la presente aplicación, el usuario puede buscar viajes y ver partes del detalle de estos sin tener que llevar a cabo ningún tipo de registro.

13.4 Árbol de navegación

En la siguiente ilustración se muestra el árbol de navegación de la web.

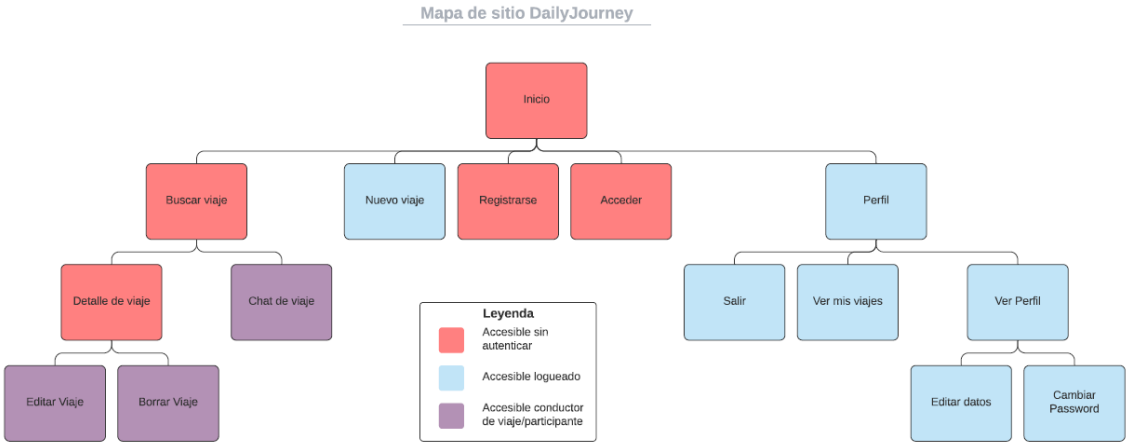


Ilustración 28. Mapa de sitio de la web

[10] En el contexto del marketing web, se conoce el “onboarding” como la experiencia del usuario al acceder por primera vez a una web.

14. Seguridad

En este punto se repasan algunos de los conceptos esenciales a nivel de seguridad que se han tenido en cuenta durante el desarrollo de esta aplicación:

A nivel general

- Uso de HTTPS en backend y frontend. Tanto Heroku como Netlify permiten añadir certificados SSL de forma gratuita a nuestros sitios. Rails además permite redirigir por defecto el tráfico a HTTPS usando la configuración 'config.force_ssl = true'
- No incluir credenciales en el repositorio.

A nivel de backend (aplicación tipo solo-API Rails)

- Chequeo de acceso no autorizado. Tal como se detalló en el punto 5.3, la aplicación hace uso de un sistema de autenticación/autorización basado en JWT. Todos los recursos que lo requieren están protegidos a nivel de backend, algunos son accesibles simplemente estando autenticado, y otros requieren además autorización dependiendo de qué usuario esté realizando la petición (por ejemplo, para editar un viaje). En ambos casos todas las peticiones http desde el cliente a recursos protegidos deben incluir un token jwt en la cabecera Authorization, que será descodificado en el servidor permitiendo o no el acceso a los recursos. Todas las solicitudes no autorizadas responden con código 401 – Unauthorized.
- Encriptación de passwords en base de datos. El uso de la funcionalidad "has_secure_password" de Rails (que utiliza bcrypt) junto con un "salt" (bits aleatorios que se usan como entrada de en una función derivadora de claves) permite almacenar las contraseñas como hashes seguros, de forma que un atacante con acceso a la tabla de usuarios nunca dispondría de las contraseñas reales.
- Filtro de passwords y otros datos sensibles en los logs. Rails permite filtrar datos como el password en los logs que se muestran en consola o se guardan para consulta en caso de errores.

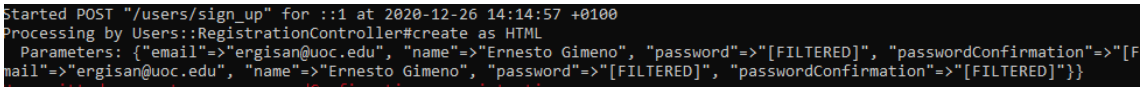


Ilustración 29. Ejemplo de cómo se filtra el password en logs de Rails

- Cross Site Request Forgery (CSRF). Rails protege por defecto contra ataques de este tipo.
- Parámetros fuertes. Rails no permite el uso de "mass asignment" o asignación masiva de parámetros en acciones update o create, debiendo explícitamente el desarrollador indicar qué parámetros exactamente permite recibir y utilizar en estas operaciones,

evitando que un usuario pueda intentar inyectar en sus peticiones parámetros que no se desean procesar.

- Inyección SQL. Todas las consultas contra la base de datos se hacen utilizando la interfaz de ActiveRecord, y teniendo cuidado de no utilizar sintaxis que permita SQL injection, especialmente al utilizar parámetros en las condiciones WHERE.

A nivel de frontend (aplicación Angular)

- Prevenir cross-site-scripting (XSS). Evitar que un usuario malicioso pueda inyectar sus propios scripts en elementos del DOM. Para prevenir esto, cualquier valor introducido en la aplicación debe ser saneado, algo que Angular hace por defecto como un mecanismo incluido con el framework, siempre que usemos las formas seguras que nos facilita el framework (template binding) y no tratemos de manipular el DOM directamente usando los métodos de la API nativa del navegador.
- Evitar el uso de apiKeys directamente desde el cliente. Utilizar cualquier tipo de apiKey desde el cliente es peligroso porque debe considerarse que están totalmente expuestas. En el caso de la aplicación DailyJourney, se sigue esta política con la clave de acceso del servicio HereMaps, que solo se utiliza en el backend, nunca llamando directamente al servicio desde el navegador y utilizando el backend como intermediario.
- Se utilizan *guards* de Angular para proteger el acceso a rutas autenticadas y autorizadas, aunque la protección real de los recursos se hace a nivel de backend.

15. Tests

Se han llevado a cabo tests manuales de manera recurrente para comprobar que el funcionamiento de la aplicación era el adecuado posteriormente a cualquier cambio.

Dado el tamaño y alcance actual de la aplicación realizar este tipo de pruebas manuales no es especialmente costoso en términos de tiempo, pero ante un escenario de una aplicación de la que se espera un recorrido largo y que por lo tanto va a verse expuesta irremediablemente al cambio, disponer de una suite de test automatizada es prácticamente una obligación para garantizar la mantenibilidad. Queda como ejercicio posterior a la presente entrega el llevar a cabo el desarrollo de dicha batería de pruebas.

Como trabajo ya realizado, se ha dejado lista la infraestructura necesaria para llevar a cabo el desarrollo de pruebas unitarias de la aplicación de Rails, utilizando la librería Rspec, junto con librerías auxiliares como Factorybot, Faker o Shoulda Matchers (se explicó en qué consistía cada una en el punto 12.1).

Para la aplicación Angular también se prevé llevar a cabo a modo de ejercicio posterior a la entrega tanto pruebas unitarias como end to end, mediante Jasmine y Cypress.

Durante el desarrollo, se ha hecho uso intensivo del programa “Insomnia”, cliente HTTP de escritorio similar a Postman muy útil para desarrollar la API y testear los endpoints de la misma.

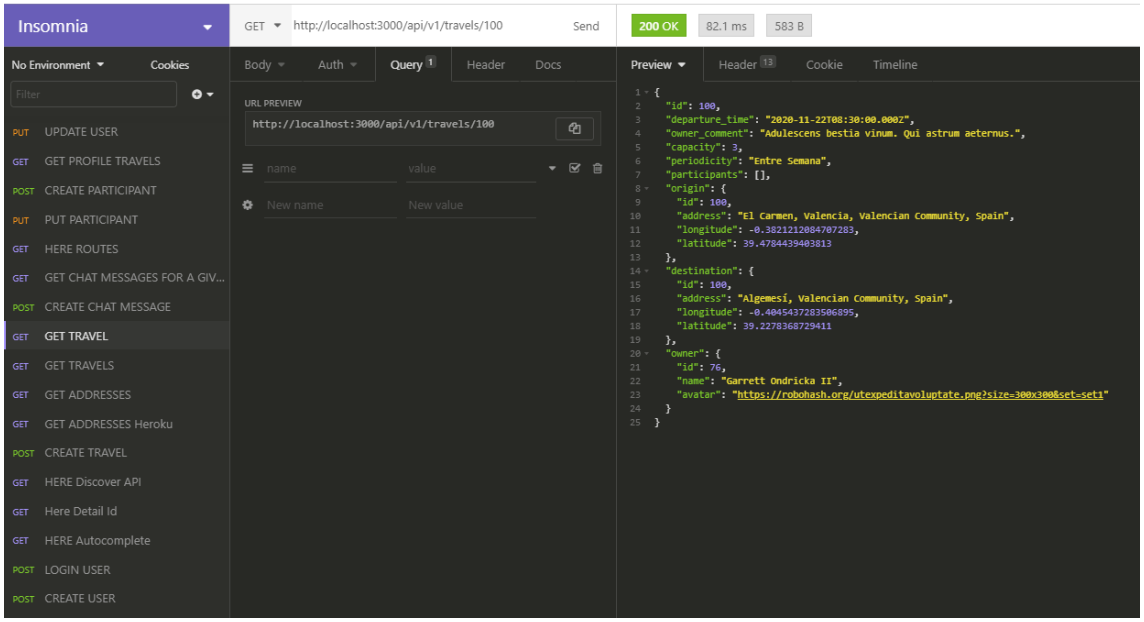


Ilustración 30. Ejemplo de petición http con Insomnia.

Por último, se ha testeado la aplicación ampliamente en las últimas versiones de los navegadores Chrome y Firefox, y es en estos navegadores en los que se garantiza una óptima experiencia.

16. Instrucciones de uso

En primer lugar, el usuario debe acceder a la aplicación por medio de su url: https://dailyjourney.netlify.app/

Una vez en la aplicación, su primer destino es la página de Home. Pese a que el usuario puede llevar a cabo búsquedas sin estar registrado, es oportuno crear un nuevo usuario para poder acceder a todas las posibilidades que ofrece la web. Para ello, el usuario debe hacer click en el link “Registrarse” de la barra de navegación superior.

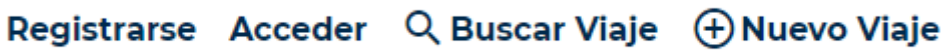


Ilustración 31. Opciones iniciales del menú de navegación

Al hacer click, se muestra una pantalla modal con el formulario de registro.

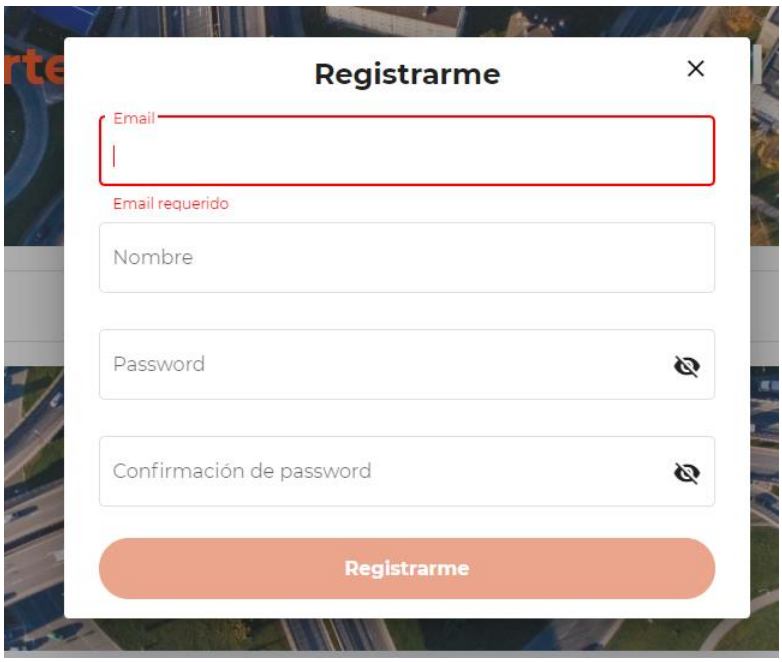


Ilustración 32. Formulario de registro

En este formulario el usuario debe imputar su dirección de email (que servirá como identificador de acceso), su nombre y el password deseado, confirmándolo para evitar errores.
Una vez registrado, el usuario queda logueado en el sistema automáticamente. En caso de que el usuario ya se haya registrado previamente, puede obviar este paso y hacer click en el link de la barra de navegación superior “Acceder”, para mostrar el formulario modal de acceso:

Ilustración 33. Formulario de login

En cualquiera de los dos casos, el usuario queda logueado en el sistema, y las opciones mostradas en la barra de navegación superior cambian:

Ilustración 34. Barra de navegación superior para usuario logueado

Se muestra el nombre de usuario y el avatar. Este avatar se ha generado automáticamente durante el registro, de forma que si el usuario dispone de una cuenta registrada en el servicio Gravatar con esa dirección de email se muestra dicho avatar, y en caso contrario se muestra un avatar por defecto personalizado con las letras iniciales de su nombre.
Al hacer click sobre el avatar o el nombre, se despliega un menú con estas opciones:

- Salir: El usuario desea hacer logout del sistema.
- Ver Perfil: Para ver y editar sus datos personales.

Ilustración 35. Detalle menú desplegable de pefil

Si el usuario hace click en “Ver Perfil”, navega a la página de detalle de perfil, donde tiene acceso a estas opciones:

1. Datos Generales. Puede editar su nombre o avatar. Para editar su avatar, debe actualizarlo en el servicio Gravatar y posteriormente hacer click en “Actualizar”. Para editar el nombre basta con editar el valor del input de texto con la etiqueta “Nombre”.

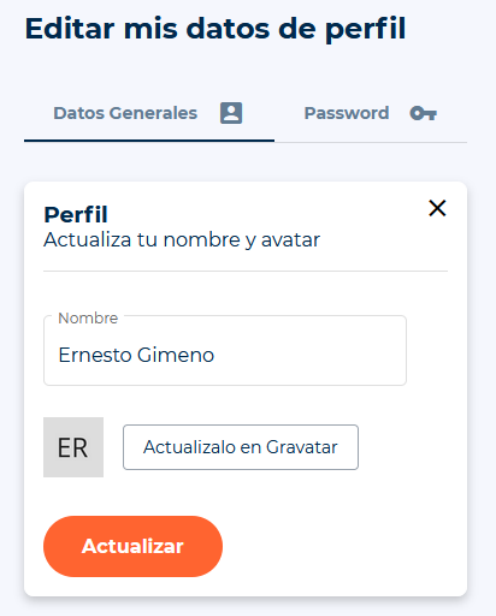


Ilustración 36. Editar datos generales de perfil.

2. Password. Para editar su contraseña actual.

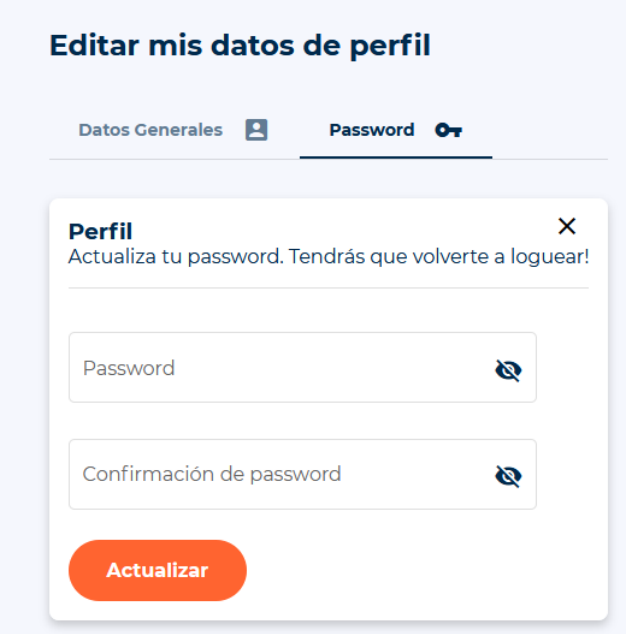


Ilustración 37.Editar contraseña

Pasando ya a las acciones esenciales de la aplicación, un usuario puede tomar dos caminos:

1. Crear su propio viaje, compartiendo su vehículo y marcando las restricciones que desee (origen, destino, horario etc)
2. Realizar una búsqueda para ver qué viajes hay disponibles en un radio inferior a 3km de su destino deseado.

Si el usuario desea crear su propio viaje puede hacerlo por dos vías:

1. Haciendo click en el link “Nuevo Viaje” de la barra de navegación.
2. Haciendo click en el botón “¡O añade tu propio viaje!” que se encuentra sobre la imagen principal de la página “Home”.

En cualquiera de los dos casos el usuario accede al formulario de creación de nuevo viaje:

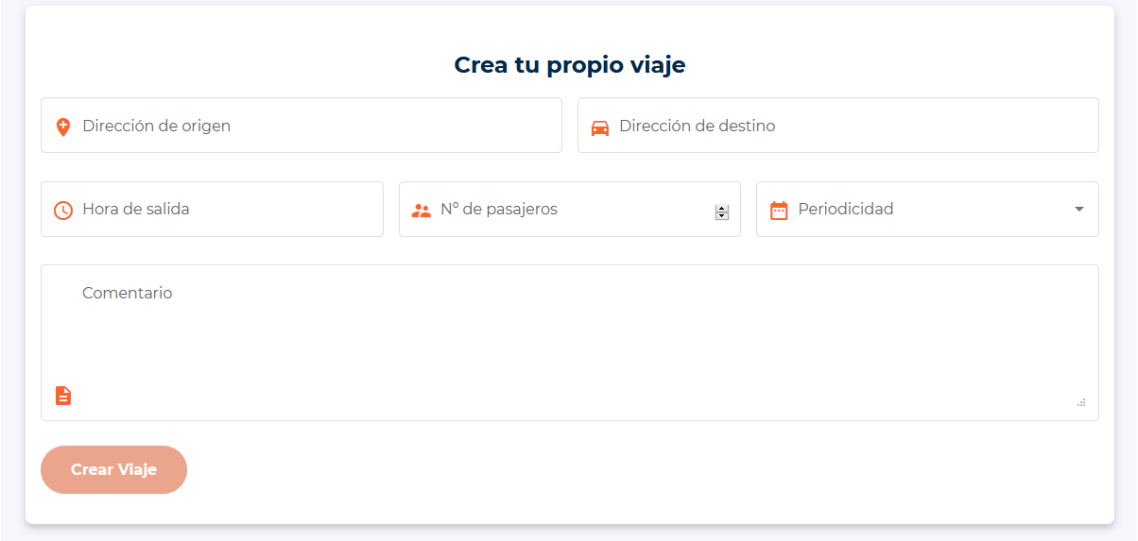


Ilustración 38. Formulario de creación de nuevo viaje.

En este formulario deben indicarse los siguientes valores dentro de los campos correspondientes:

- Dirección de origen y destino. El usuario debe escribir su valor deseado en el campo correspondiente y elegir una de las direcciones que se sugieren en el desplegable que aparece y cambia según el usuario escribe.
- Hora de salida. El usuario debe indicar la hora de salida del viaje usando el control “timepicker” que se muestra al hacer click en el campo.
- Nº de pasajeros. El usuario debe elegir el aforo máximo de su viaje, sin tenerle en cuenta a él.
- Periodicidad. El usuario indica si su viaje se lleva a cabo todos los días de la semana, todos los días entre semana, sábado y domingo o de manera semanal, debiendo indicar en este caso el día o días de la semana en que planea realizar el viaje en los comentarios.

- Comentario. Cualquier otro tipo de dato que el usuario considere relevante: tiempo de espera, coste por persona, organización de los pagos, tiempo de espera de cortesía, localización exacta donde se recogerá a las personas etc.

Una vez el usuario ha rellenado todos estos campos y son válidos, el botón “Crear Viaje” se activa y el usuario puede finalizar la creación de su viaje. Al hacerlo, se navega a la página de este nuevo viaje:

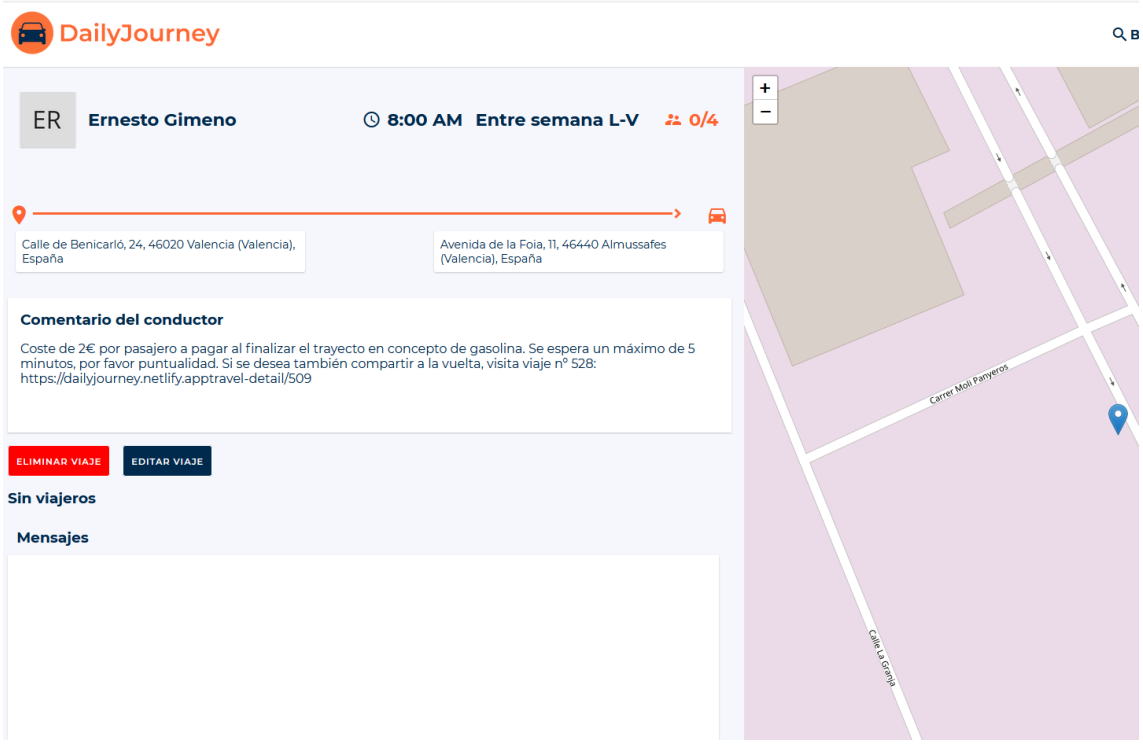


Ilustración 39. Página de detalle de viaje: Vista del conductor

Si en cambio el usuario desea buscar que viajes ya existentes están cerca de su destino, puede acceder a la búsqueda de viaje por dos vías:

1. En los links de la barra de navegación superior, haciendo click en buscar viaje.
2. En el formulario de búsqueda que se puede observar en sobre la imagen principal de la página “Home”.

En cualquiera de los dos casos el usuario debe imputar las direcciones de origen y de destino que desea, y pulsar el botón “Buscar”. Al hacerlo se muestra la página de resultados con viajes con destino dentro de un radio de 3km con respecto al indicado:

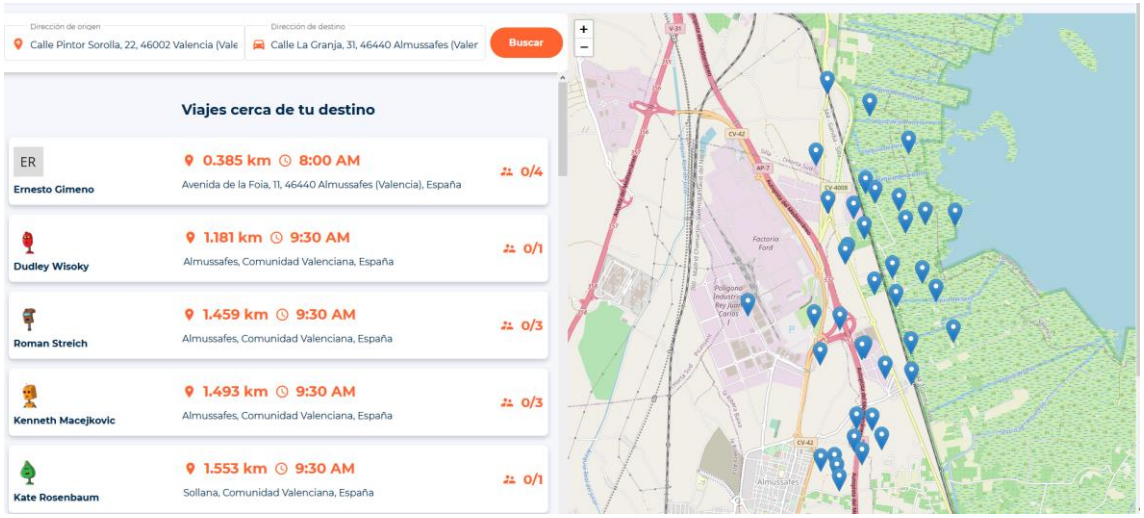


Ilustración 40. Página de búsqueda de viajes

El usuario puede pinchar en los marcadores que señalan los puntos de destino del viaje para obtener:

- Origen del viaje en el que se ha hecho click
- Scroll en la lista de viajes marcándose el viaje en el que ha hecho click.

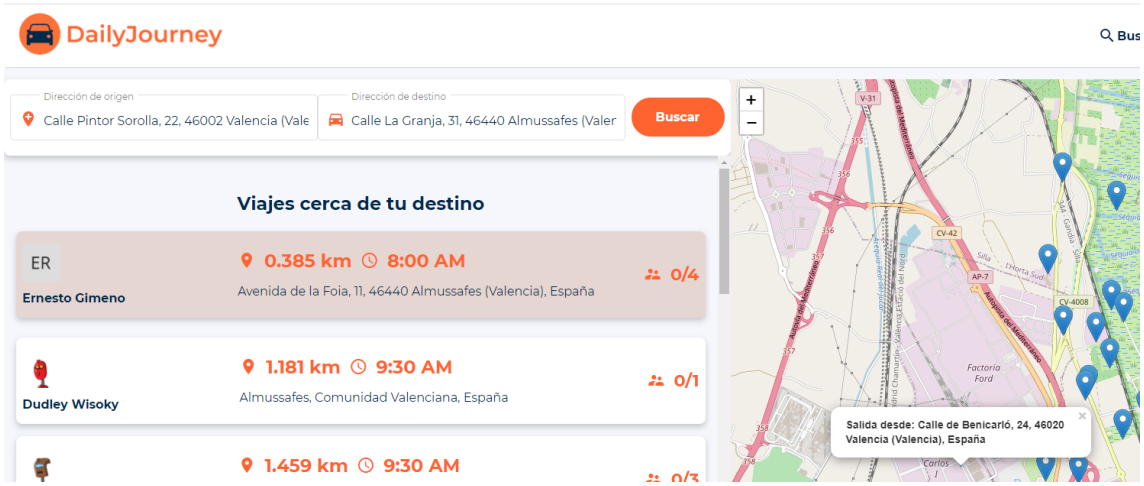


Ilustración 41. usuario hace click en un marcador del mapa

El usuario también puede hacer click en el viaje que desea ver en más detalle dentro de la lista de la derecha y acceder a la página de detalle de viaje:

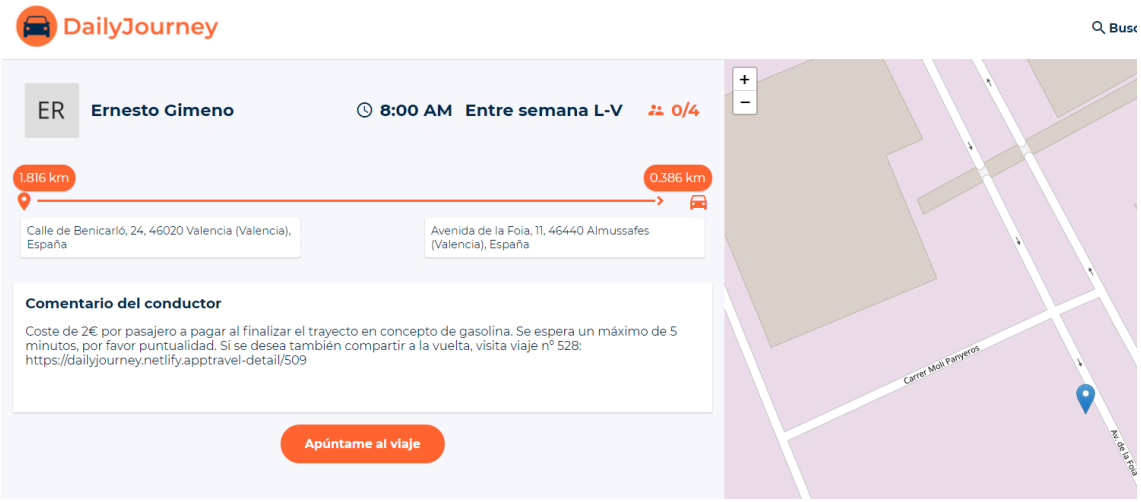


Ilustración 42. Detalle de viaje, vista de participante

El usuario puede ver en detalle:

1. Dirección de salida del viaje y distancia con respecto a su punto de origen antes especificado.
2. Dirección de destino del viaje y distancia con respecto a su punto de destino antes especificado.
3. Detalle de hora de salida, periodicidad, estado del aforo.
4. Comentario del conductor.

Si el usuario desea formar parte del viaje, debe pulsar en el botón “Apúntame al viaje”. El usuario queda a la espera de que el conductor apruebe su participación:

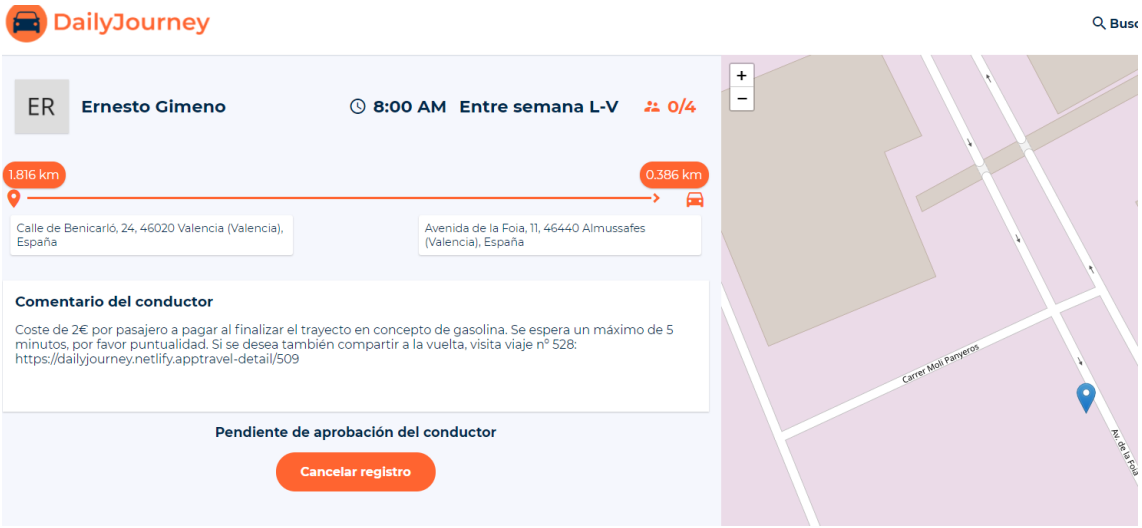


Ilustración 43. Participante pendiente de aprobación.

Además, el viaje ahora aparece en el desplegable del menú de navegación superior, como forma rápida de acceder a los viajes de los que forma parte el usuario, tanto como como conductor como viajero:

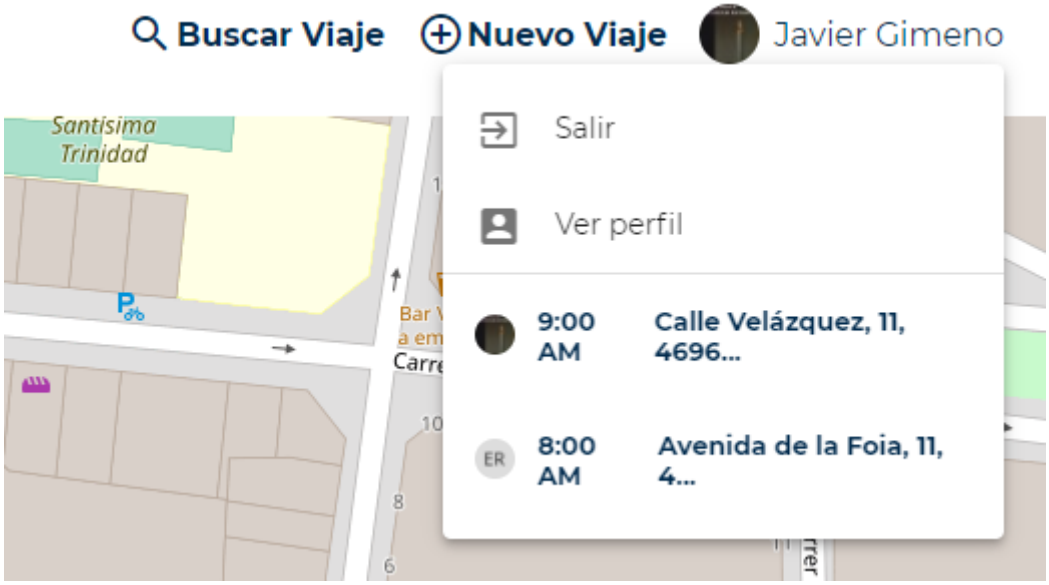


Ilustración 44. Viajes del usuario. Se observa un viaje propio y otro ajeno

Por otro lado, el usuario creador del viaje puede ahora observar en su vista de la página de detalle que hay un usuario pendiente de aprobación utilizando el botón verde, o rechazarlo con el botón rojo:

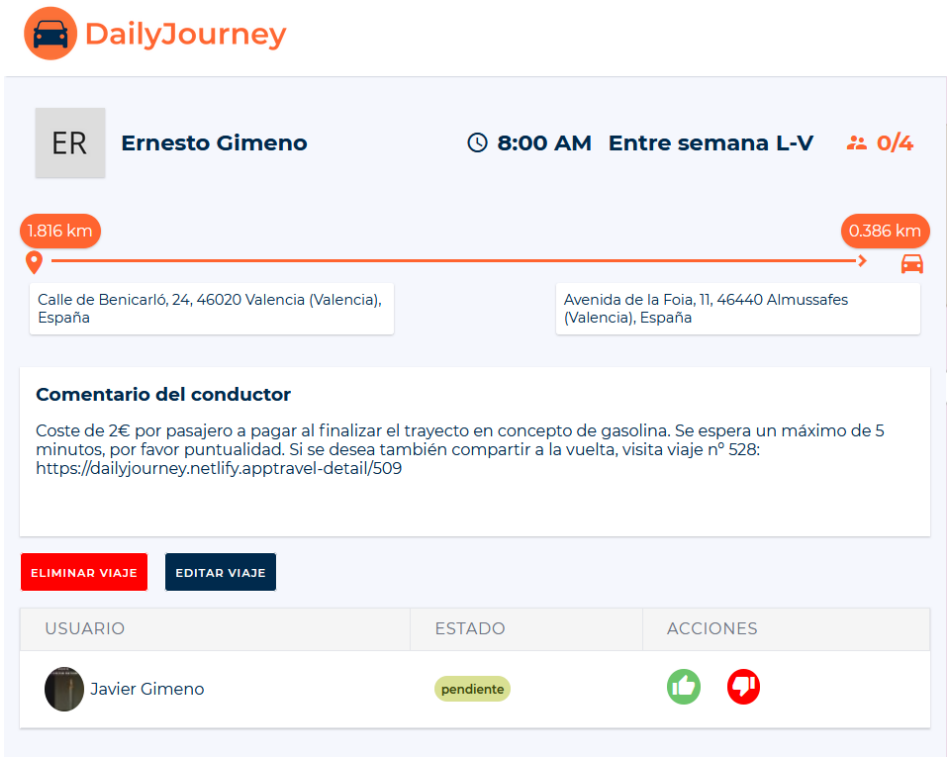


Ilustración 45. Detalle de viaje, vista del conductor con participantes apuntados

El conductor puede aprobar al usuario, cambiando el aforo del viaje en este caso de 0 a 1 y cambiando el estado del participante:

Además, tanto el creador del viaje como los viajeros aprobados pueden ver y añadir mensajes en el chat del viaje:

Ilustración 46. Chat de viaje

Por último, el creador de viaje puede tomar estas dos acciones con respecto al viaje:

1. Borrar el viaje.
2. Editar el viaje. Al hacer click en este botón se accede a la pantalla de edición de viaje, similar a la pantalla de creación de viaje, pero con los datos de partida rellenados:

Ilustración 47.Pantalla de actualización de detalle de viaje

17. Proyección a futuro.

En esta sección se detallan algunos aspectos que conforman las potenciales mejoras futuras que desean añadirse a la aplicación.

- Mejoras de escalabilidad/mantenibilidad/experiencia de desarrollo.
    - Generar una suite de tests unitarios y e2e en la que poder confiar a la hora de llevar a cabo cambios, refactorizaciones o implementar nuevas funcionalidades.
    - Refactorizar componentes demasiado grandes en Angular. Ciertos componentes han quedado demasiado grandes y hacen demasiadas cosas. Algunos de los componentes se han desarrollado de manera que ha sido posible su reutilización en al menos dos páginas diferentes de la app (formulario de imputación de direcciones y mapa de destinos), pero otros claramente mejorarían extrayendo funcionalidad en componentes más pequeños y con responsabilidades más acotadas.
    - Extraer queries complejas de los controladores a otro tipo de objetos (service objects, query objects) en aplicación Rails.
    - Implementar sistema de integración/entrega continúa (CI/CD), por ejemplo, con Github Actions.
- Mejoras tecnológicas/de plataforma
    - Uso de websockets. Algunas funcionalidades mejorarían claramente la experiencia de usuario si se beneficiasen del uso de websockets para hacerlas más reactivas, por ejemplo, el chat.
    - Utilizar un esquema de autenticación/autorización más robusta, como Oauth2.
    - Convertir la aplicación en una PWA[11] que permita por ejemplo su instalación.
    - Explorar el uso de Angular Universal para mejorar el SEO convirtiendo la app en una SSR[12].
- Nuevas características
    - Registro/Login mediante proveedores sociales como Google o Facebook.
    - Implementar sistema de notificaciones in-app con acciones relevantes a revisar por el usuario.
    - Mejorar los flujos secundarios de autenticación: Olvido de password, confirmación de cuenta etc.
    - Permitir al usuario subir su propio avatar y almacenarlo en AWS S3 o servicio similar.

[11] Progressive Web Application o Aplicación Web Progresiva, ofrecen una experiencia en móviles lo más parecida a la de una aplicación nativa.
[12] Server Side Rendering, es una técnica utilizada para representar una aplicación de página única (SPA) en el servidor.

  - Añadir más filtros en las búsquedas de viaje, por ejemplo, permitir definir el radio de distancia en km de la búsqueda.
  - Integrar instrucciones de ruta del servicio Here Maps, por ejemplo, para que un usuario pueda descargar las instrucciones para llegar al punto de origen del viaje desde el suyo propio.
  - Añadir valoraciones al conductor.
  - Añadir calendario de viajes para un control más granular de la periodicidad de los viajes.
  - Añadir verificación de los conductores tal como se hace en otras aplicaciones similares de economía colaborativa.
  - Introducir editor de texto “Lo que ves es lo que obtienes” (WYSIWYG es el acrónimo en inglés con el que se conoce a estos editores) para mejorar el formato de los comentarios del conductor.

- A nivel de producto
  - Explorar formas de gestionar la monetización de la app, quizá admitiendo los pagos de los participantes al conductor directamente en la aplicación y cobrando una comisión.

18. Presupuesto

En esta sección se presenta la estimación de presupuesto de la realización y mantenimiento del presente trabajo. Como precondiciones, se tiene en cuenta que el equipo informático con el que se ha llevado a término el desarrollo constituido por PC completo y monitor, así como las licencias de sistema operativo y suite de ofimática usados en la redacción de este documento ya han sido ampliamente amortizadas en el pasado por lo que no se les atribuye un coste. De la misma manera, ninguna de las herramientas de software para el desarrollo de las que se ha hecho uso presentan un coste por ser generalmente open-source o de uso libre.

Por tanto, los costes que definen el presupuesto derivan del coste en mano de obra del trabajo y los costes de mantenimiento derivados del despliegue. Los costes de desarrollo se incluyen completos por ser una unidad única y discreta. Los costes de mantenimiento de la infraestructura son recurrentes durante la vida de la aplicación, y se estiman los costes correspondientes a 1 año. Como nota adicional, estos costes son el estándar, pero en particular no se ha incurrido en ningún coste durante el desarrollo de este trabajo gracias al uso del Github Students Pack, donde se ofrece 1 dyno gratis de Heroku durante 2 años entre muchas otras cosas: https://education.github.com/pack

Se tienen en cuenta además los siguientes aspectos:

1) El add-on de Heroku PostgresSQL es de pago a partir de 10.000 filas en la base de dato, por lo que se considera que debe entrar en la estimación de costes desde el primer momento.
2) El tier gratis del servicio HereMaps comprende 250.000 peticiones mensuales al servicio, teniendo en cuenta que durante el desarrollo del proyecto apenas se ha consumido un 0,5% de ese límite, se entiende que podría empezar a operarse sin necesidad de pagar dicho servicio. En caso de superar dicho límite, el coste por cada 1000 peticiones extra al servicio es 1€.

Con respecto al coste de infraestructura, cabe añadir que realizando una inversión en aprendizaje de administración de sistemas/devops sería posible reducir el coste desplegando en un VPS propio en lugar de un servicio como Heroku, tal como ya reseñado en el punto 11.3

CONCEPTO - COSTE RECURSOS HUMANOS	UDS (HORAS)	COSTE UNITARIO €	TOTAL
Definición de idea, requisitos y prototipos	30	35,00 €	1.050,00 €
Desarrollo de API REST Rails	40	35,00 €	1.400,00 €
Desarrollo de aplicación cliente Angular	100	35,00 €	3.500,00 €
TOTAL			**5.950,00 €**

Tabla 5. Presupuesto de recursos humanos

CONCEPTO - COSTE INFRAESTRUCTURA	UDS	COSTE UNITARIO €	TOTAL
Coste de dominios	1	15 €	15 €
Coste de dyno web en Heroku (7$/mes)	12	5,67 €	68,04 €
Coste add-on Postgres tier Hobby Basic (9$/mes)	12	7,29 €	87,48 €
TOTAL			**170,52 €**

Tabla 6. Presupuesto anual de infraestructura

19. Análisis de mercado

Actualmente, hay diversas apps que llevan a cabo un servicio similar, pero con suficientes matices como para que el nicho buscado con DailyJourney (viajes recurrentes de diario) pueda ser atractivo.

Dos de las principales alternativas podrían ser las siguientes:

- BlablaCar. BlaBlaCar es un gigante con una enorme implantación en España. El concepto inicial de conductor que busca gente que le ayude a compartir los gastos y propone un viaje es similar, pero su uso está mucho más extendido en viajes no recurrentes de media y larga distancia.
- Journify. Este producto sí que está ideado para compartir vehículo en trayectos recurrentes, pero a fecha de la redacción de este documento solo disponen de aplicación móvil disponible en las tiendas de los dispositivos, lo cual puede reducir su cuota de mercado por la fricción extra que supone para el usuario acceder una aplicación desde App Store o Google Play.

20. Conclusiones

La valoración final a nivel particular es tremendamente positiva, especialmente por el conjunto de conocimientos adquiridos durante el proceso, no solo a nivel meramente tecnológico sino también de procedimiento. He tenido la oportunidad en el pasado de desarrollar aplicaciones completas, pero la necesidad de llevar a cabo el trabajo de modelado formal de las diferentes fases del proyecto me ha permitido hacerme una idea mucho más certera de lo que es y los pasos que articulan el desarrollo de un producto.

A nivel tecnológico, ha sido muy enriquecedor poder trabajar con Rails en modo API, ya que pese a que he trabajado con el framework en el pasado siempre lo había hecho en su formato tradicional, el de framework para desarrollar aplicaciones clásicas del lado servidor. Pese a que muchos de los conceptos que definen el desarrollo con Rails siguen aplicando, la necesidad de aprender sobre nuevos aspectos como es la autenticación/autorización de una api, serializar las respuestas o profundizar en la comprensión de las bases de las aplicaciones basadas en REST ha sido muy positivo.

También he tenido la oportunidad de mejorar enormemente mis habilidades con Angular, y estoy especialmente satisfecho de la mayor comprensión obtenida acerca de las buenas prácticas en general y de la arquitectura y organización de la aplicación en particular.

Con respecto a los objetivos iniciales expuestos al inicio de esta memoria, han podido cumplirse todos los principales y la mayoría de los secundarios, y la planificación planteada también ha podido llevarse a cabo en gran manera, por lo que el balance también es bueno.

En definitiva, la realización de este trabajo ha supuesto una gran mejora en mis conocimientos técnicos y de gestión de un proyecto de desarrollo de esta índole, y como nota personal además supone la constatación de que el desarrollo de aplicaciones web es algo que me apasiona y que espero que en algún momento pueda convertirse además en mi carrera profesional.

Anexo 1. Entregables

Repositorios de código:

- API Rails https://github.com/EGimenoS/dailyjourneyapi
- Cliente Angular https://github.com/EGimenoS/dailyjourney

Direcciones de despliegue

- API: https://github.com/EGimenoS/dailyjourneyapi
- Producto Final: https://dailyjourney.netlify.app

Memoria: PAC_FINAL_mem_GimenoSanchez_Ernesto.pdf
Código: PAC_FINAL_prj_GimenoSanchez_Ernesto.rar
Vídeo: PAC_FINAL_vid_GimenoSanchez_Ernesto.mp4
Presentación: PAC_FINAL_prs_GimenoSanchez_Ernesto.pdf
Autoinforme de evaluación: PAC_FINAL_autoeval_GimenoSanchez_Ernesto.pdf

Anexo 2. Código fuente (Extractos)

Ejemplo de modelo en Rails con validaciones, relaciones y un scope (consulta nombrada)

```ruby
class Travel < ApplicationRecord
  validates_presence_of :departure_time, :capacity, :owner_comment, :owner_id
  validates_associated :origin, :destination
  validates :origin, presence: true
  validates :destination, presence: true
  has_many :participants, dependent: :destroy
  has_many :chat_messages, dependent: :destroy, inverse_of: :travel
  belongs_to :user, class_name: 'User', foreign_key: 'owner_id'
  has_one :origin,  dependent: :destroy
  has_one :destination,  dependent: :destroy
  accepts_nested_attributes_for :origin, :destination

  scope :near_of, ->(target_lat, target_lng) { joins(:destination).merge(Destination.near([target_lat, target_lng], 3)) }
end
```

Ilustración 48. Fichero travel.rb

Ejemplo de serializer en Rails

```ruby
class TravelSerializer < BaseSerializer
  attributes :departure_time, :owner_comment, :capacity, :distance, :bearing
  has_many :participants
  has_one :origin
  has_one :destination
  belongs_to :user, key: :owner
end
```

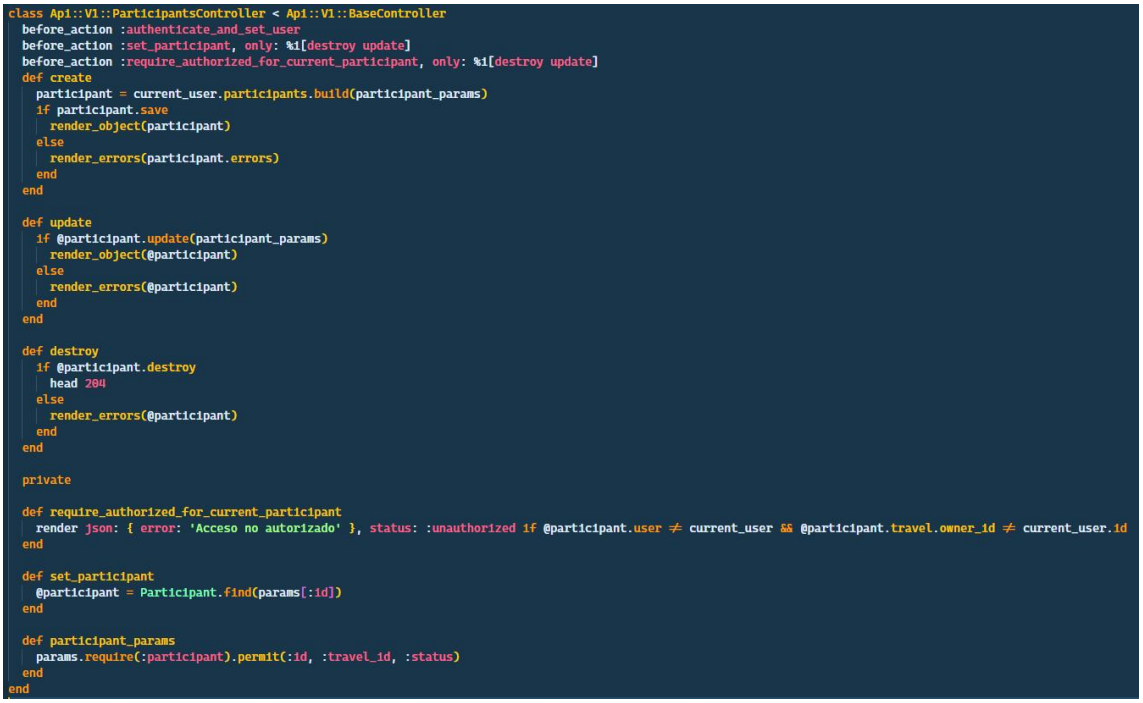
Ilustración 49 Fichero travel_detail_serializer.rb

Adaptador de servicio de mapas

```ruby
module Adapter
  class HereMaps
    API_KEY = ENV['here_maps_api_key']

    def initialize(base_url, query, at = '39.48728,-0.36593')
      @base_url = base_url
      @config = set_config
      @query = query
      @at = at
    end

    def autosuggest_addresses
      req_params = { at: @at, q: @query }
      results = RestClient.get @base_url, { params: @config.merge(req_params) }
      results = JSON.parse(results)
      results['items'].map do |item|
        { address: item['title'], latitude: item['position']['lat'], longitude: item['position']['lng'] }
      end
    end

    private

    def set_config
      { apikey: API_KEY, limit: 5 }
    end
  end
end
```

Ilustración 50 Fichero adapters/adapter.rb

Ejemplo de controlador en Rails. En las primeras líneas se observan los callbacks que tienen lugar para las acciones del controlador indicadas y que definen la autenticación/autorización de los mismos.



```ruby
class Api::V1::ParticipantsController < Api::V1::BaseController
  before_action :authenticate_and_set_user
  before_action :set_participant, only: %i[destroy update]
  before_action :require_authorized_for_current_participant, only: %i[destroy update]
  def create
    participant = current_user.participants.build(participant_params)
    if participant.save
      render_object(participant)
    else
      render_errors(participant.errors)
    end
  end

  def update
    if @participant.update(participant_params)
      render_object(@participant)
    else
      render_errors(@participant)
    end
  end

  def destroy
    if @participant.destroy
      head 204
    else
      render_errors(@participant)
    end
  end

  private

  def require_authorized_for_current_participant
    render json: { error: 'Acceso no autorizado' }, status: :unauthorized if @participant.user ≠ current_user && @participant.travel.owner_id ≠ current_user.id
  end

  def set_participant
    @participant = Participant.find(params[:id])
  end

  def participant_params
    params.require(:participant).permit(:id, :travel_id, :status)
  end
end
```

Ilustración 51. Fichero participants_controller.rb

Modelo User

```ruby
class User < ApplicationRecord
  after_update :set_user_gravatar
  after_create :set_user_gravatar
  has_many :travels, foreign_key: :owner_id, class_name: 'Travel', dependent: :destroy
  has_many :chat_messages, dependent: :destroy
  has_many :participants, dependent: :destroy
  has_secure_password
  validates :email, uniqueness: true, presence: true
  validates_presence_of :name, on: :create

  def jwt_token_payload
    { name: name, avatar: avatar }
  end

  def set_user_gravatar
    gravatar_id = Digest::MD5.hexdigest(email.downcase)
    update_columns(avatar: "http://secure.gravatar.com/avatar/#{gravatar_id}?s=32&d=https%3A%2F%2Fui-avatars.com%2Fapi%2F/#{name}/128")
  end
end
```

Ilustración 52 Fichero User.rb

Ejemplo de servicio Angular

```typescript
import { Injectable } from '@angular/core';
import { HttpClient, HttpErrorResponse } from '@angular/common/http';
import { Observable, of } from 'rxjs';
import { AutocompleteAddress } from '../interfaces/autocomplete-address';
import { endpoint } from '../../../../config';
import { catchError } from 'rxjs/operators';
import { ErrorsService } from './errors.service';
import { UserLocationService } from './user-location.service';

@Injectable({
  providedIn: 'root',
})
export class AutocompleteAddresesService {
  url = `${endpoint}/search_addresses`;
  defaultAt = '39.48728,-0.36593';
  position: any = null;
  constructor(
    private http: HttpClient,
    private errorsService: ErrorsService,
    private userLocationService: UserLocationService
  ) {
    this.userLocationService.deviceLocation.subscribe(
      (location) => (this.position = `${location.coords.latitude},${location.coords.longitude}`)
    );
  }

  getValidAddreses(query: string): Observable<AutocompleteAddress[]> {
    const at = this.position || this.defaultAt;
    return this.http
      .get<AutocompleteAddress[]>(this.url, { params: { q: query, at } })
      .pipe(
        catchError((error: HttpErrorResponse) => {
          this.errorsService.handleError(error, 'Obteniendo direcciones');
          return of(null);
        })
      );
  }
}
```

Ilustración 53. autocomplete-addresses.service.ts

Ejemplo de servicio CRUD Angular

```
import { HttpClient, HttpErrorResponse, HttpHeaders } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { endpoint } from 'config';
import { Observable, of } from 'rxjs';
import { catchError } from 'rxjs/operators';
import { ApiResponse } from '../interfaces/api-response';
import { ChatMessage } from '../interfaces/chat-message';
import { ErrorsService } from './errors.service';

@Injectable({
  providedIn: 'root',
})
export class ChatMessagesService {
  url = `${endpoint}/chat_messages`;
  headers = new HttpHeaders({
    'Content-type': 'application/json',
  });
  constructor(private http: HttpClient, private errorsService: ErrorsService) {}

  public getChatMessagesByTravel(travelID: string): Observable<ChatMessage[]> {
    return this.http
      .get<ChatMessage[]>(this.url, {
        params: { travel_id: travelID },
      })
      .pipe(
        catchError((error: HttpErrorResponse) => {
          this.errorsService.handleError(error, 'Obteniendo mensajes de chat');
          return of(null);
        })
      );
  }

  public createNewChatMessage(travelID: string, message: string): Observable<ApiResponse> {
    return this.http
      .post<ApiResponse>(
        this.url,
        { travel_id: travelID, message },
        {
          headers: this.headers,
        }
      )
      .pipe(
        catchError((error: HttpErrorResponse) => {
          this.errorsService.handleError(error, 'Creando mensaje');
          return of(null);
        })
      );
  }
}
```

Ilustración 54. chat-messages-service.rb

Ejemplo de componente Angular, formulario de creación/edición de viaje

```
import { formatDate } from '@angular/common';
import { Component, OnInit } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
import { ActivatedRoute } from '@angular/router';
import { NgxMaterialTimepickerTheme } from 'ngx-material-timepicker';
import { Observable } from 'rxjs';
import { debounceTime, filter, switchMap } from 'rxjs/operators';
import { AutocompleteAddress } from 'src/app/core/interfaces/autocomplete-address';
import { Travel } from 'src/app/core/interfaces/travel';
import { AutocompleteAddresesService } from 'src/app/core/services/autocomplete-addreses.service';
import { TravelsService } from 'src/app/core/services/travels.service';
import { longlatPresence } from 'src/app/core/validators/longlat-presence';

@Component({
  selector: 'app-add-travel',
  templateUrl: './add-travel.component.html',
  styleUrls: ['./add-travel.component.scss'],
})
export class AddTravelComponent implements OnInit {
  isUpdating = false;
  periodicityOptions = ['Diario L-D', 'Semanal', 'Entre semana L-V', 'Fin de semana S-D'];
  newTravelGroupForm: FormGroup;
  validOriginAddreses: Observable<AutocompleteAddress[]>;
  validDestinationAddreses: Observable<AutocompleteAddress[]>;

  theme: NgxMaterialTimepickerTheme = {
    container: {
      bodyBackgroundColor: 'white',
      buttonColor: '#002a4d',
    },
    dial: {
      dialBackgroundColor: '#ff6430',
    },
    clockFace: {
      clockFaceBackgroundColor: '#ff6430',
      clockHandColor: '#002a4d',
      clockFaceTimeInactiveColor: '#fff',
    },
  };

  constructor(
    private fb: FormBuilder,
    private validAddreses: AutocompleteAddresesService,
    private travelsService: TravelsService,
    private route: ActivatedRoute
  ) {}

  onSubmit(): void {
    this.isUpdating
      ? this.travelsService
          .updateTravel(this.route.snapshot.params.id, this.newTravelGroupForm.value)
          .subscribe()
      : this.travelsService.createNewTravel(this.newTravelGroupForm.value).subscribe();
  }
  createNewTravelGroupForm(): FormGroup {
    return this.fb.group({
      origin_attributes: ['', [longlatPresence()]],
      destination_attributes: ['', [longlatPresence()]],
      departure_time: ['', Validators.required],
      capacity: ['', Validators.required],
      periodicity: ['', Validators.required],
      owner_comment: ['', Validators.required],
    });
  }
  ngOnInit(): void {
    this.newTravelGroupForm = this.createNewTravelGroupForm();

    if (this.route.snapshot.params.id) {
      this.isUpdating = true;
      this.fillForm(this.route.snapshot.params.id);
    }
    this.validOriginAddreses = this.newTravelGroupForm.get('origin_attributes').valueChanges.pipe(
      debounceTime(300),
      filter((value) => value.length > 0),
      switchMap((value) => this.validAddreses.getValidAddreses(value))
    );
    this.validDestinationAddreses = this.newTravelGroupForm
      .get('destination_attributes')
      .valueChanges.pipe(
        debounceTime(300),
        filter((value) => value.length > 0),
        switchMap((value) => this.validAddreses.getValidAddreses(value))
      );
  }
  displayFn(address: AutocompleteAddress): string {
    if (address) {
      return address.address;
    }
  }

  private fillForm(id: string): void {
    this.travelsService.getTravelDetail(id).subscribe((travel: Travel) => {
      this.newTravelGroupForm.patchValue({
        ...travel[0],
        origin_attributes: travel[0].origin,
        destination_attributes: travel[0].destination,
        departure_time: formatDate(travel[0].departure_time, 'hh:mm', 'en'),
      });
    });
  }
}
```

Ilustración 55. fichero add-travel.ts

Ejemplo interceptor http para gestionar spinner de carga

```ts
import { Injectable } from '@angular/core';
import {
  HttpRequest,
  HttpHandler,
  HttpEvent,
  HttpInterceptor,
  HttpResponse,
} from '@angular/common/http';
import { Observable, throwError } from 'rxjs';
import { catchError, tap } from 'rxjs/operators';
import { UiService } from '../services/ui.service';

@Injectable()
export class LoadingInterceptor implements HttpInterceptor {
  constructor(private uiService: UiService) {}
  excludedEndpoints = ['search_address', 'chat_messages', 'profile_travels'];
  intercept(request: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    if (this.excludedEndpoints.some((url) => request.url.includes(url))) {
      return next.handle(request);
    }
    this.uiService.setLoading(true, request.url);
    return next.handle(request).pipe(
      tap((evt: HttpEvent<any>) => {
        if (evt instanceof HttpResponse) {
          this.uiService.setLoading(false, request.url);
        }
        return evt;
      }),
      catchError((err) => {
        this.uiService.setLoading(false, request.url);
        return throwError(err);
      })
    );
  }
}
```

Ilustración 56. loading.interceptor.ts

Anexo 3. Librerías/Código externo utilizado

- Backend:
  - Ruby On Rails 6.0.3. Framework web MVC para Ruby
  - Pg 2.0 (gema[13] de potsgres para Ruby)
  - Puma 4.1 (gema del servidor web Puma para Ruby)
  - Api_guard 0.5.0. Librería de autenticación para apis con jwt.
  - Rest-client 2.1.0. Cliente para hacer peticiones http par Ruby.
  - Faker: Implementación para Ruby de la gema para generar datos falsos a usar durante desarrollo.
  - Rubocop: Linter y formateador de código para Ruby
  - Active_model_serializers: Gema para serializar respuestas json de la api.
  - Figaro. Gema para gestionar secretos y credenciales.
  - Geocoder: Gema para hacer geolocalización y cálculos relacionados.
  - Rspec: Framework de testing para Ruby
  - FactoryBot: Reemplazo de fixtures para tests
- Frontend
  - Angular 10. Framework de Google para desarrollar SPA's con TypeScript.
  - Angular Material: Librería de componentes basado en Material Design para Angular.
  - Leaflet/ngx-leaflet: Librería para visualización de mapas dinámicos.
  - Angular-jwt: Librería open source de la compañía auth0 para facilitar el trabajo con jwt en el cliente.
  - Geolib: Cálculo de distancias en el cliente.
  - Rxjs: Implementación para Javascript/TypeScript de la librería Rx para programación reactiva.
  - Material-design-icons: Iconos de Angular Material.
  - Prettier: Formateador de código.

[13] Una gema es una librería de Ruby, de forma análoga a por ejemplo los paquetes de npm de Javascript.

Anexo 4. Libro de estilo

- Logo

Ilustración 57. Logo de DailyJourney

- Paleta de colores

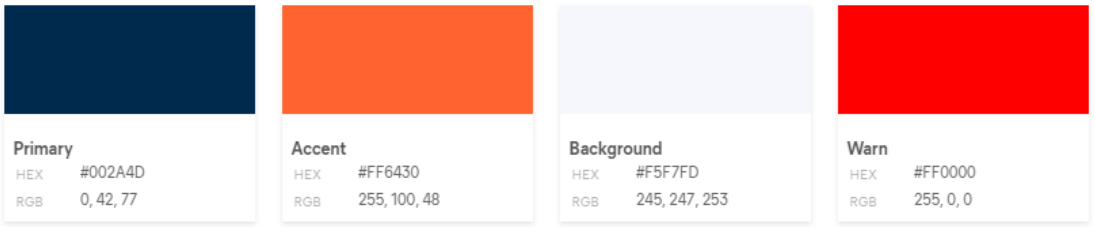


Ilustración 58. Paleta de colores principal de la aplicación

- Fuente: Montserrat https://fonts.google.com/specimen/Montserrat#standard-styles

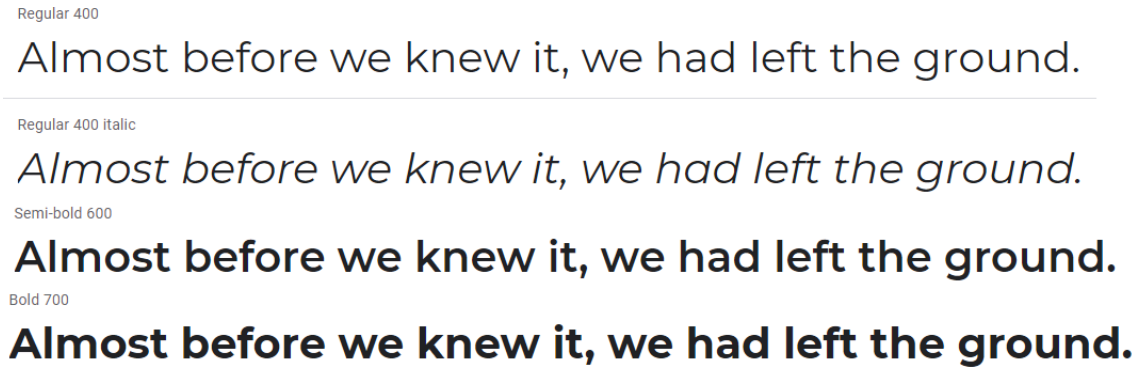


Ilustración 59. Fuente Montserrat

- Iconos: Angular Material Icons https://material.io/resources/icons/?style=baseline

Anexo 5. Bibliografía

- Pete Lepage, Patrones de Diseño Web Adaptables: https://developers.google.com/web/fundamentals/design-and-ux/responsive/patterns?hl=es
- Tomas Trajan, How to build epic Angular apps with clean architecture: https://medium.com/@tomastrajan/how-to-build-epic-angular-app-with-clean-architecture-91640ed1656
- Guías de Rails: https://guides.rubyonrails.org/
- Guías de Angular: https://angular.io/docs
- Documentación de Angular Material: https://material.angular.io/components/categories
- Patrones de diseño: http://ui-patterns.com/patterns