UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

GESTOR DE ARCHIVOS GTFS PARA TRANSAPP

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

JOAQUÍN ANTONIO ROMERO MUNIZAGA

PROFESORA GUÍA:
JOCELYN SIMMONDS WAGEMANN

MIEMBROS DE LA COMISIÓN:
GONZALO NAVARRO BADINO
ANDRÉS MUÑOZ ORDENES

SANTIAGO DE CHILE
2021

Resumen

TranSapp es una aplicación que permite a sus usuarios visualizar y realizar reportes sobre los paraderos, buses y recorridos del transporte público de Santiago. La especificación General Transit Feed Specification (GTFS) fue creada por Google como un estándar de archivos que describen redes de transporte. Con todo esto en mente, el equipo de TranSapp a veces requiere crear/editar archivos GTFS para poder agregar nuevas redes de transporte y alterar las que ya utilizan.

Existen aplicaciones que permiten editar archivos GTFS, sin embargo, las que el equipo de TranSapp ha logrado instalar resultan inadecuadas para la edición de ciertos componentes del modelo, principalmente los que utilizan información geográfica. Por esto, se busca implementar un editor de archivos GTFS que se ajuste a las necesidades del equipo de TranSapp con mapas interactivos para los modelos que utilizan información geográfica.

Para implementar la solución se decidió crear una aplicación web usando Django para el backend y Vue para el frontend. Como primer paso se realizó el diseño de un diagrama de flujo de aplicación y mockups, los cuales fueron validados con el cliente. Tomando en cuenta la especificación GTFS, se procedió a diseñar e implementar un modelo de datos con el sistema de modelos de Django. Con estos modelos se implementaron endpoints de API para el acceso y edición de datos, incluida la subida y bajada de archivos. Una vez completada la API se procedió a implementar vistas básicas para la visualización y edición de datos, vale decir, vistas generales para acceder a cada proyecto y vistas específicas de edición para cada tabla. Con estas vistas finalizadas, se procedió a implementar vistas con un mapa interactivo utilizando Mapbox para asistir al usuario de forma visual (Ver los paraderos en el mapa por ejemplo). Una vez terminadas las vistas avanzadas se procedió a refinar los distintos componentes de la aplicación.

Con las vistas implementadas el usuario es capaz de manipular los datos correspondientes a todas las tablas presentes en el estándar GTFS. Además, agregan varias funcionalidades que hacen que trabajar sobre ellas sea más amigable a trabajar con un editor genérico como Excel.

La metodología de trabajo consistió en reuniones diarias con el equipo de TranSapp, en las que se definieron objetivos de corto plazo y se realizó validación continua durante el proceso de desarrollo. La aplicación logró cumplir con los objetivos planteados al definir el proyecto.

En este documento se describe el proceso de creación de una aplicación que en su versión actual resuelve algunas necesidades previamente no resueltas del equipo de TranSapp, aportando valor al equipo. Gracias a la aplicación ahora pueden de manera más fácil editar archivos GTFS pre-existentes y crear nuevos archivos GTFS.

Agradecimientos

Quiero agradecer a mi familia, quienes me han acompañado y apoyado a lo largo de este duro proceso. Cuando he pasado malos ratos siempre han estado siempre han estado a mi lado y me han brindado soporte. Cuando he pasado buenos ratos siempre han celebrado conmigo.

También quiero agradecer a mi profesora guía, Jocelyn Simmonds. Su disposición y apoyo han sido fundamentales para mi en esta última etapa.

Por otra parte, quiero agradecer a mis amigos, quienes han estado junto a mi en las buenas y en las malas.

Por último, quiero darle las gracias a la gente de TranSapp. Fue un agrado trabajar con ellos, tanto en el aspecto personal como el profesional.

Tabla de Contenido

Índice de Ilustraciones

Capítulo 1

Introducción

1.1. Antecedentes

TranSapp es una aplicación para Android e iOS desarrollada para los usuarios de Red, el sistema de transporte público de Santiago, por un grupo alumnos del Departamento de Ciencias de la Computación de la Universidad de Chile en el curso de Proyecto de Software y extendida por el equipo de TranSapp [1], quienes han formado una empresa con el mismo nombre. La aplicación permite a los usuarios obtener información en línea sobre la ubicación y los recorridos de los buses, visualizar sus rutas y realizar reportes sobre el estado de paraderos y buses.

En la figura 1.1 podemos apreciar una vista de un paradero con los buses que van en camino a él. La información de la red publicada por el Directorio de Transporte Público Metropolitana (DTPM) es complementada con información de los dispositivos GPS instalados en los buses, que transmiten una señal de posición cada 30 segundos, y de lo que reportan los propios usuarios, en más de 55.000 reportes registrados al mes [2].

1.2. Problema

Los datos básicos utilizados por TranSapp corresponden a un archivo que describe la red de transporte. Este archivo viene en un formato llamado GTFS [3], el cual se encuentra descrito en detalle en el capítulo 2. Cuando TranSapp se ve en la necesidad de agregar información, como por ejemplo al integrar un servicio de transporte que no es manejado por el DTPM, se ven en la necesidad de editar o crear un GTFS.

Un caso particular de esto es cuando decidieron integrar los buses comunales de La Reina, donde tuvieron que usar un editor de GTFS pre-existente. Los editores existentes de GTFS son algo limitados, y fue un proceso engorroso y bastante manual para crear e integrar los *Shapes* correspondientes a estos recorridos. En términos simples, un *Shape* corresponde a una polilínea en un mapa, la cual se utiliza para representar el trazado de un viaje.

Figura 1.1: Vista mostrando recorridos de TranSapp

1.3. Objetivos

A continuación se presenta el objetivo general y los objetivos específicos de esta memoria.

1.3.1. Objetivo General

El objetivo planteado consiste en la creación de un sistema que permita la creación, edición, validación y publicación de archivos GTFS.

1.3.2. Objetivos Específicos

1. Modelamiento de la base de datos para permitir manipular y crear datos correspondientes a un GTFS.
2. Diseño e implementación de una interfaz que permita la edición de estos datos.
3. Incorporación de interfaces con mapas interactivos para los datos que ocupan información geográfica.
4. Diseño e implementación de mecanismos para la exportación e importación de GTFS.
5. Validación del producto con los clientes.

1.4. Solución

Para cumplir con los objetivos se desarrolló una aplicación web utilizando Django para el backend y Vue para el frontend. En el backend se implementó una API REST que permite realizar sobre todos los modelos de un GTFS. En el frontend se utilizó Vue con Vuetables

para crear tablas editables para los modelos del GTFS. Además, se usó Mapbox para crear mapas interactivas para los modelos que ocupan información geográfica.

1.5. Metodología

Para desarrollar el proyecto se emplearon prácticas ágiles. Se operó en un esquema de reuniones diarias con el cliente, permitiendo al cliente dar retroalimentación inmediata sobre el producto. Se utilizó un Kanban virtual en Trello, dividiendo las tareas en columnas de backlog, deseable, importante, fundamental, en proceso y hecho. Por último, se implementaron unit tests en el backend, probando todos los endpoints implementados. Se dispone de una batería de 235 tests, los cuales verifican los distintos métodos de la API, incluyendo archivos CSV para la validación de la subida y bajada de archivos CSV de cada una de las tablas.

1.6. Estructura de la memoria

Este documento está estructurado como sigue: en el capítulo 2 se explicará en mayor detalle el formato GTFS y por qué las herramientas pre-existentes no han sido capaces de satisfacer las necesidades de TranSapp. En el capítulo 3 se describirá el diseño de la solución, explicando el trabajo previo a la implementación. En el capítulo 4 se entrará en el detalle de implementación. En el capítulo 5 se hablará sobre la validación de la aplicación. Por último, en la sección de conclusiones se hará una recapitulación y se hablará sobre posibles mejoras al proyecto.

Capítulo 2

Edición de GTFS

En este capítulo se describirá el estándar GTFS y se van a mencionar algunas de las herramientas pre-existentes.

2.1. Estándar GTFS

El acrónimo GTFS significa General Transit Feed Specification. Corresponde a una especificación creada por Google [4] para describir sistemas de transporte. Un archivo GTFS es un archivo *.zip* (*GTFS.zip*) que contiene varios archivos *.txt* que tienen formato de CSV y describen las distintas entidades del sistema de transporte.

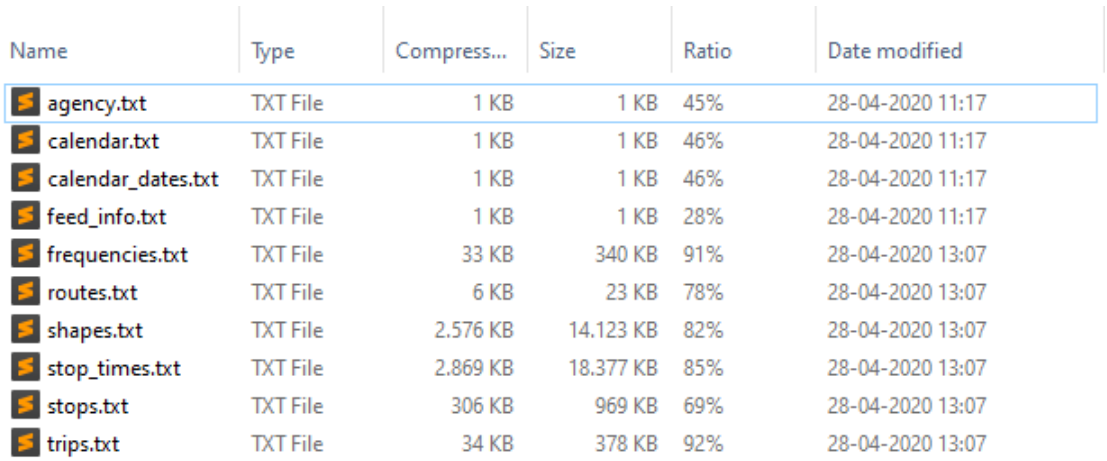
El estándar GTFS se ha vuelto un estándar de facto, vale decir, pese a no haber sido validado o certificado por alguna institución (por ejemplo ISO), es el estándar que se ocupa en la industria para compartir información de redes de transporte.

2.2. Formato de un archivo GTFS

Para explicar el formato de un archivo GTFS estudiaremos el que publica el Directorio de Transporte Público Metropolitano (DTPM). El DTPM es una institución creada el año 2013 con el objetivo de "analizar de forma integral el sistema de transporte público capitalino y velar por la adecuada coordinación de los diferentes modos que participan en el transporte público de la ciudad de Santiago". Esta institución publica en su sitio web un archivo GTFS [3] que describe la Red Metropolitana de Movilidad, el sistema de transporte público de santiago. Esto incluye a los buses urbanos, el Metro y el MetroTren Nos.

El archivo GTFS publicado por el DTPM, cuyo contenido se puede observar en la figura 2.1, describe el sistema de transporte público de Santiago. Su tamaño total es de 33,4 MB sin comprimir y 5,8 MB comprimido. Para ver ejemplos de la estructura de estos archivos, referirse al apéndice A.

A continuación, se explica el contenido de varios de los archivos que aparecen en la figura 2.1, omitiendo los más sencillos:



Name	Type	Compress...	Size	Ratio	Date modified
agency.txt	TXT File	1 KB	1 KB	45%	28-04-2020 11:17
calendar.txt	TXT File	1 KB	1 KB	46%	28-04-2020 11:17
calendar_dates.txt	TXT File	1 KB	1 KB	46%	28-04-2020 11:17
feed_info.txt	TXT File	1 KB	1 KB	28%	28-04-2020 11:17
frequencies.txt	TXT File	33 KB	340 KB	91%	28-04-2020 13:07
routes.txt	TXT File	6 KB	23 KB	78%	28-04-2020 13:07
shapes.txt	TXT File	2.576 KB	14.123 KB	82%	28-04-2020 13:07
stop_times.txt	TXT File	2.869 KB	18.377 KB	85%	28-04-2020 13:07
stops.txt	TXT File	306 KB	969 KB	69%	28-04-2020 13:07
trips.txt	TXT File	34 KB	378 KB	92%	28-04-2020 13:07

Figura 2.1: GTFS entregado por el DTPM

- `stops.txt` define los paraderos del sistema de transporte, incluye las coordenadas de los paraderos. Cuenta con 11.412 entradas.
- `routes.txt` define las rutas, como por ejemplo las líneas de metro o los recorridos de los buses, como la línea 506. Cuenta con 372 entradas.
- `shapes.txt` define polilíneas que corresponden a los trazados de los recorridos. Una polilínea consiste en una serie de puntos que representan una secuencia de líneas uniendo los puntos consecutivos. Este archivo tiene 375.369 entradas, que corresponden a aproximadamente 890 shapes.
- `trips.txt` define viajes de cada recorrido y les asocia un *shape*. Dispone de 9.393 entradas.
- `stop-times.txt` le asigna a los viajes de `trips.txt` una secuencia de paraderos en conjunto con horas de partida y llegada para cada uno. Cuenta con 471.221 entradas.
- `frequencies.txt` define horas de partida para cada *trip*, básicamente establece que el viaje se repite durante un intervalo de horas cada cierto tiempo. Cuenta con 9.306 entradas.

Notar que por su naturaleza `shapes` y `stop-times` requieren varias filas para describir a un único elemento. Esto contribuye a que sean los archivos más grandes del GTFS. Además, el archivo de frequencies es opcional pues se puede crear un viaje con sus respectivos `stop-times` por cada bus que parte, en vez de repetir un viaje varias veces.

2.3. Herramientas existentes

Hay herramientas que permiten crear y editar archivos GTFS, pero en TranSapp no han logrado usarlas por diversas causas, ya sea por problemas en la instalación o porque carecen de características importantes para cumplir con los requerimientos de TranSapp. Un ejemplo es Transit Data Tools [5], esta aplicación no ha podido ser instalada por el equipo de TranSapp, ya que no han logrado que la interfaz se conecte al de servidor y la documentación no ha sido suficiente. Otro ejemplo es static GTFS Manager [6]. Esta aplicación permite editar archivos

GTFS, pero no permite editar el archivo `shapes.txt`, requiriendo que el usuario utilice una aplicación externa para crear un archivo `.geojson` y lo suba a la aplicación. Adicionalmente, ninguna herramienta que han instalado dispone de un mapa interactivo que les permita editar de forma más fácil la posición de los paraderos, los `shapes` y los `stop-times`.

Capítulo 3

Análisis y Diseño

En esta sección se listan las principales historias de usuario de la solución, la arquitectura propuesta y el diseño de la solución (flujos, mockups y modelo de base de datos). Por último, se describen las herramientas de manejo de información geográfica ocupadas en este proyecto.

3.1. Historias de Usuario

Para entender en el problema veremos algunas historias de usuario. Estas historias de usuario han sido creadas para ilustrar lo que los clientes esperaban de la aplicación.

Historias de Usuario

H1: Como usuario quiero crear, editar y borrar entradas para crear archivos GTFS.

H2: Como usuario quiero descargar los datos que edito como un archivo GTFS para usarlo en otras aplicaciones.

H3: Como usuario quiero subir un archivo GTFS y editar sus datos.

H4: Como usuario quiero subir y bajar un CSV para editar tablas específicas.

H5: Como usuario quiero arrastrar un paradero para ajustar su posición.

H6: Como usuario quiero crear un Shape agregando unos pocos puntos de referencia y obtener un Shape que, pasando solo por calles, pase por todos ellos.

H7: Como usuario quiero agregar paraderos a un viaje haciéndole click a los paraderos con ayuda del Shape asociado.

3.2. Arquitectura de la solución

A grandes rasgos, la solución consiste en una aplicación web estándar usando el framework Django [7] para el backend y Vue [8] para el frontend. El acceso a la base de datos se realiza a

través del object-relational mapping (ORM) de Django, también conocido como modelos [9]. Este componente de Django se encarga de traducir operaciones sobre objetos de python a SQL, creando automáticamente las tablas necesarias para almacenar los datos en base a la definición de clases y realizando todas las operaciones de acceso, creación y modificación necesarias al realizar operaciones sobre los objetos. Esta permite entonces evitar tener que trabajar directamente con SQL, delegando el manejo de datos a los modelos. Al realizar una consulta sobre un modelo de Django, la librería entrega un Queryset, el cual corresponde a un objeto que representa una consulta de SQL. Uno puede iterar sobre los objetos de un Queryset o puede encadenar operaciones sobre este, como por ejemplo ordernarlo o filtrarlo según algún criterio. Django no realiza la consulta de SQL hasta intentar acceder a los objetos contenidos en el Queryset. Django permite configurar distintas bases de datos para levantar el ORM. Para la aplicación desarrollada se ha decidido utilizar PostgreSQL. En un archivo de configuración se le entrega a Django la dirección (URL), el nombre de usuario y contraseña de la base de datos y Django se encarga de conectarse y gestiona las consultas.

3.2.1. Backend

El backend levanta una API REST (REpresentational State Transfer) en base a los modelos. REST es un estándar para APIs diseñado por Roy Fieldings [10], que se ha convertido en uno de los principales patrones para diseño de aplicaciones cliente servidor. REST aprovecha los métodos de HTML para definir sus operaciones: GET para obtener datos de la API; PUT para reemplazar objetos completos; DELETE para borrar datos; PATCH para actualizar valores; y POST para crear entradas (esto es una simplificación, pero resulta suficiente para comprender su uso). REST se basa en el envío de mensajes sin estado, vale decir, cada mensaje debe ser capaz de ser interpretado por si solo (si el cliente debe "responder" a un mensaje el contexto debe ser entregado por el cliente, no manejado por el servidor). Una API REST expone endpoints, estos corresponden a las distintas URLs que maneja la API, cada una con sus respectivas operaciones. Por ejemplo al ingresar la url */api/projects/* del backend, se obtiene una lista de todos los proyectos almacenados.

El backend está implementado como un proyecto de Django que utiliza la librería Django REST Framework [11] para levantar una API REST en base a los modelos implementados. Django REST Framework posee Serializers, los cuales permiten realizar de forma más sencilla la conversión de modelo de Django a JavaScript Object Notation (JSON)[12] y vice-versa. Adicionalmente cuenta con ViewSets, las cuales son clases que definen una vista con un conjunto de operaciones permitidas. Como en TranSapp se utiliza Amazon Web Services (AWS)[13], el proyecto está diseñado para correr en un contenedor de Docker[14]. Docker es un software que permite crear sistemas operativos virtuales llamados contenedores. Esto permite fácilmente levantar la misma aplicación en distintos sistemas sin tener que preocuparse por problemas de compatibilidad.

3.2.2. Frontend

El frontend utiliza la API definida en el backend, permitiendo visualizar y manipular los datos que representan al GTFS en edición, cargando los datos para mostrarlos y actualizando/creando datos cuando el usuario los edita. El frontend ha sido implementado utilizando Vue. Vue es un framework que se centra en definir componentes con su propio HTML, lógica

y Cascading Style Sheets (CSS) [8]. Esto permite encapsular mejor el comportamiento de las partes de la aplicación y tener una mejor abstracción del código, pues en vez de tener todo el HTML en un archivo y código en JavaScript y CSS que opera sobre ese HTML, el JavaScript solo opera sobre el mismo archivo, generando mayor localidad del código.

Para implementar tablas con Vue se utilizó Vuetable, esta librería permite crear tablas en base a datos en formato JSON [15]. Vuetable utiliza un parámetro llamado field para permitir configurar los tipos de campos que se muestran en la tabla. Los campos que Vuetable ofrece por defecto no son editables, así que para este proyecto, se requirió extender la definición de field para permitir el ingreso de datos. Esto se hizo mediante la inclusión de inputs de HTML, lo que permite que ahora se pueda operar sobre los datos presentados en las tablas directamente.

3.3. Flujos de la aplicación

Al inicio del proyecto se creó un diagrama de flujo que fue validado con la gente de TranSapp, incluido el Diseñador de UX. Además, en base a ese flujo se diseñaron mockups para las vistas.

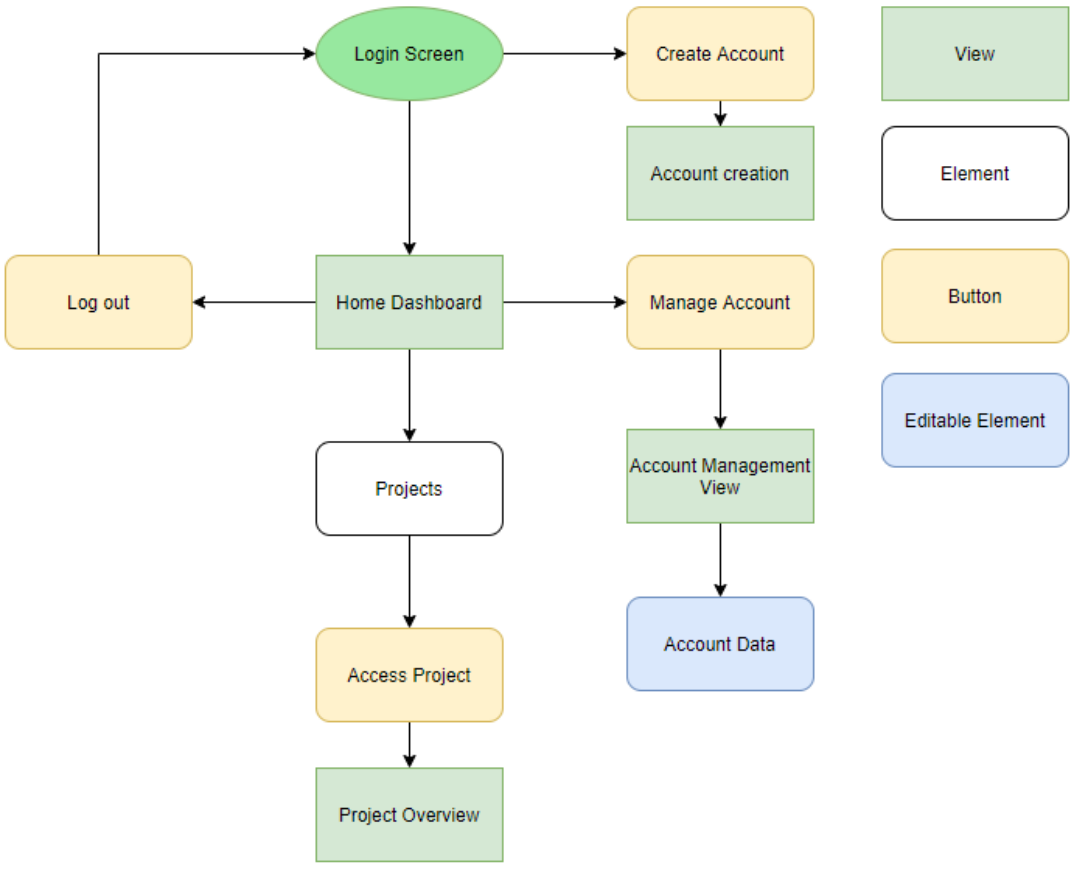


Figura 3.1: Diagrama de flujo del inicio

En la figura 3.1 podemos ver la sección del diagrama flujo correspondiente a la lógica de

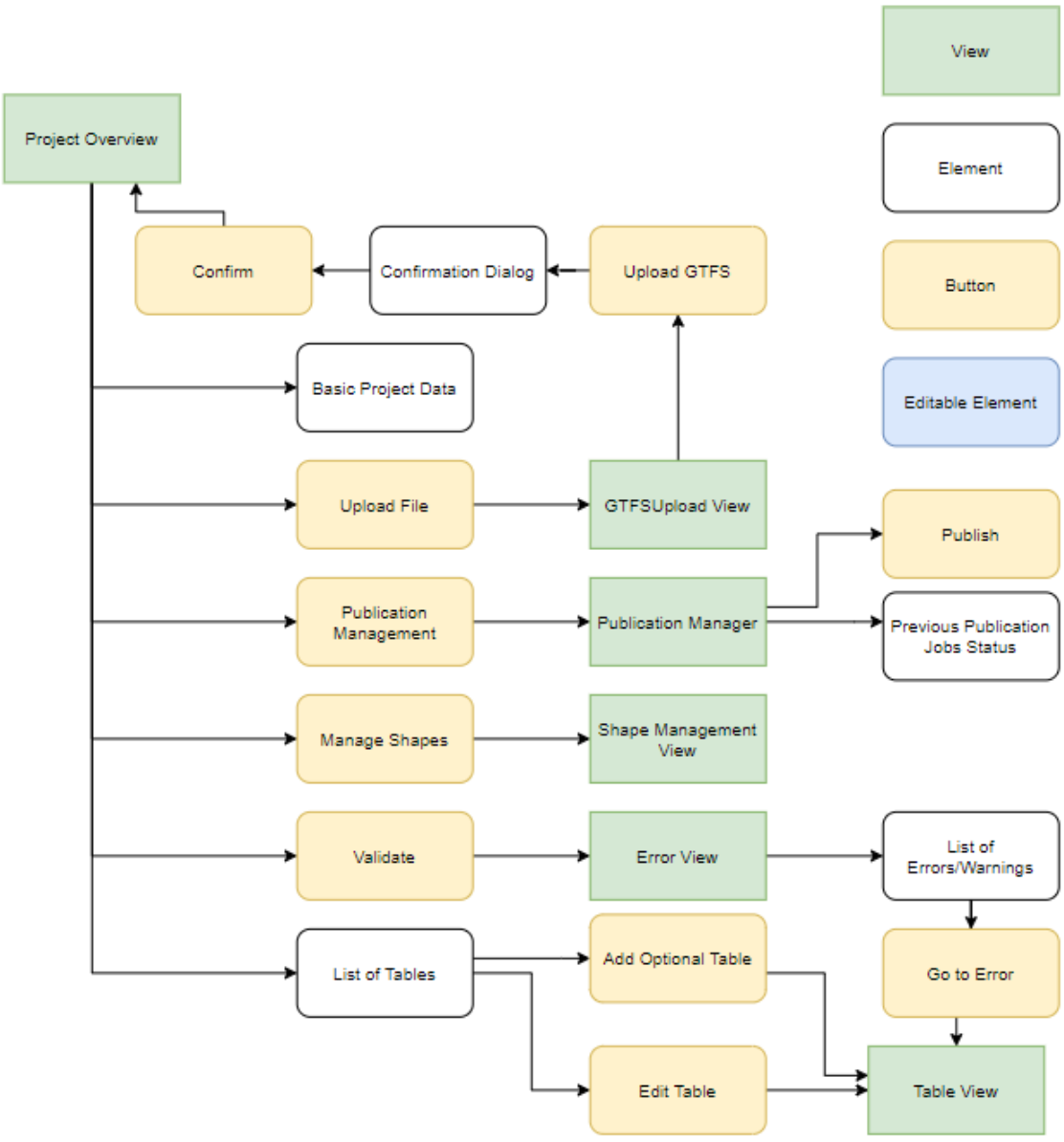


Figura 3.2: Diagrama de flujo del proyecto

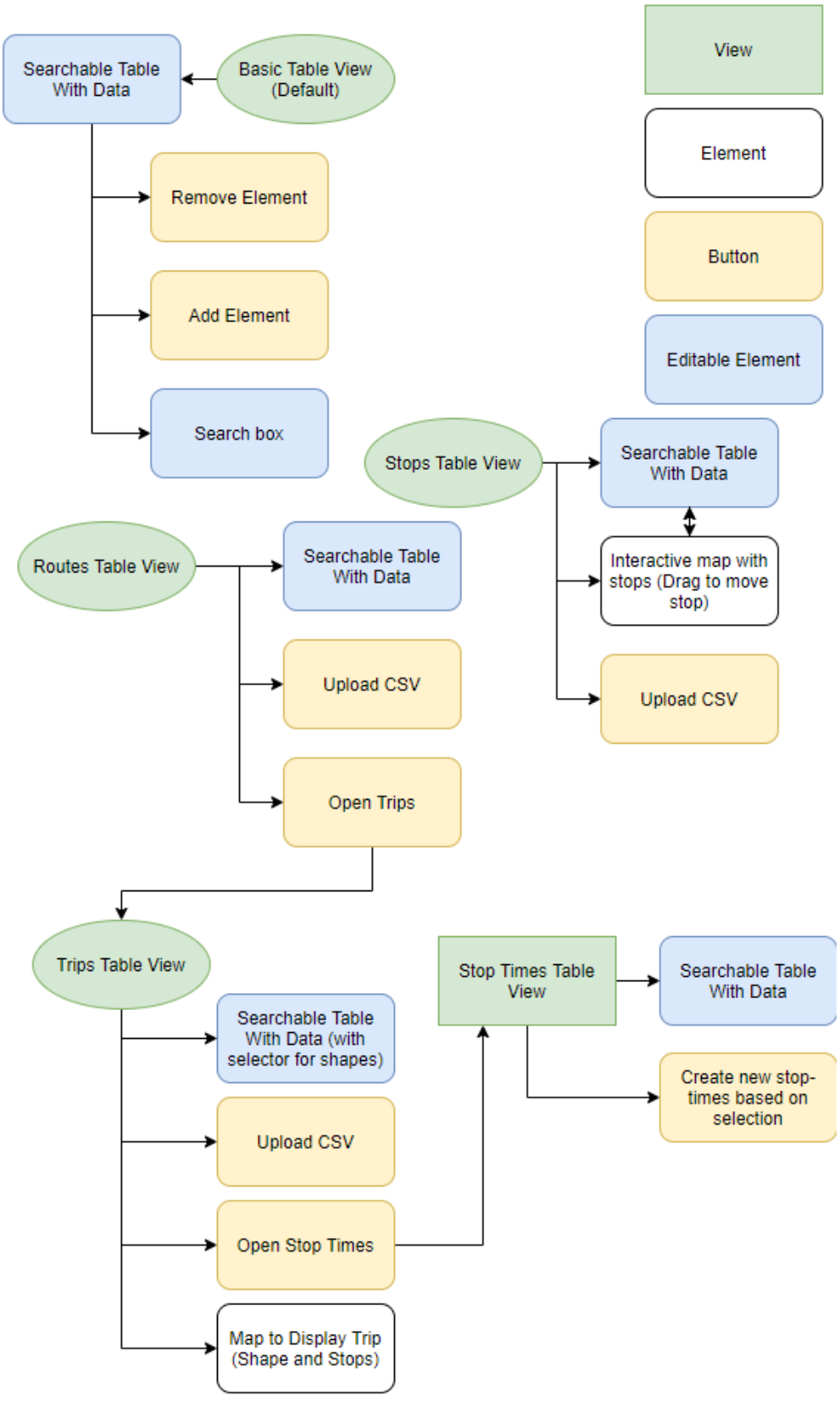


Figura 3.3: Diagrama de flujo de las tablas

ingreso a la aplicación. El manejo de cuentas no fue implementado, pues las prioridades del proyecto cambiaron durante su curso.

El diagrama de flujo de la figura 3.2 representa el flujo entre las vistas asociadas a un proyecto, excluyendo las de edición de los datos. `Project Overview` corresponde a un dashboard del proyecto desde se puede acceder a información básica y se tiene acceso a varias acciones que se pueden ejecutar sobre proyecto, además de poder acceder a las tablas para editar los datos. De las acciones básicas tenemos la vista `GTFSUploadView` para permitir al usuario subir un archivo GTFS, para las tareas de publicación existe la vista `Publication Manager`, la cual permite al usuario ver el estado de las tareas de publicación preexistentes y además dispone de un botón para crear una nueva tarea de publicación. Para la validación tenemos la vista `Error View`, la cual nos muestra el resultado de la última validación. Se puede ir directamente a la tabla que haya producido el error desde esta vista. `Shape Management View` reemplaza a la vista de tabla de `shapes` nos provee una vista que nos permite acceder a nuestros `shapes` como resumen, desde donde se podría ingresar a editar el `shape`.

La figura 3.3 corresponde al diagrama de flujo de las tablas. Como la mayoría de las tablas a editar son relativamente sencillas, se definió `Basic Table View` como vista por defecto para la mayoría de los casos y he expandido sobre esta para las tablas con mayor interactividad. Las vistas de forma circular son aquellas que pueden ser accedidas directamente desde el dashboard del proyecto. Cabe notar que los modelos de las tablas de `routes`, `trips` y `stop-times` son creados de manera jerárquica, por esto se decidió permitir al usuario acceder desde una tabla a la siguiente. Si partimos desde `Routes Table View` podemos para una ruta específica acceder a sus viajes, lo que nos llevaría a `Trips Table View`. En `Trips Table View` contamos con un mapa para mostrar los paraderos y el `shape` de cada ruta, pero si quisiéramos editar el `stop-times` asociado nos lleva a la vista `Stop Times Table View`, desde la cual podemos editar el `stop-times`. `Stops Table View` es una tabla básica con la particularidad de que además cuenta con un mapa interactivo en el que se pueden editar los paraderos.

3.4. Mockups

Para planificar las vistas se diseñó una serie de mockups, los cuales fueron validados con el equipo de TranSapp. Se elaboraron 15 mockups, 5 de ellos corresponden a vistas de generales y de proyecto y 10 de ellos corresponden a vistas de edición de tablas. Estas vistas se encuentran detalladas en el apéndice B. Para ilustrar la idea del diseño estudiaremos tres vistas clave: la de visión general del proyecto, la de edición de `stops` y la de edición de `stop-times`. Esto pues estas son las vistas que pasaron por más revisiones.

En la figura 3.4 se puede apreciar el concepto original de cómo se vería el dashboard de un proyecto. En la parte A podemos apreciar la lista de tablas, la cual nos indica la cantidad de valores en la tabla y nos indica la última hora de edición. Al hacer click en una de las filas la aplicación nos debiera enviar a la vista de edición de la tabla correspondiente. En el sector B tenemos botones que nos permiten realizar acciones sobre el proyecto, ya sea configurarlo, subir o bajar el archivo GTFS, validar los datos o crear una tarea de publicación. El sector C del mockup nos entrega cierta información sobre el proyecto y nos permite ver un resumen del resultado de la última validación y acceder a esa información en detalle.

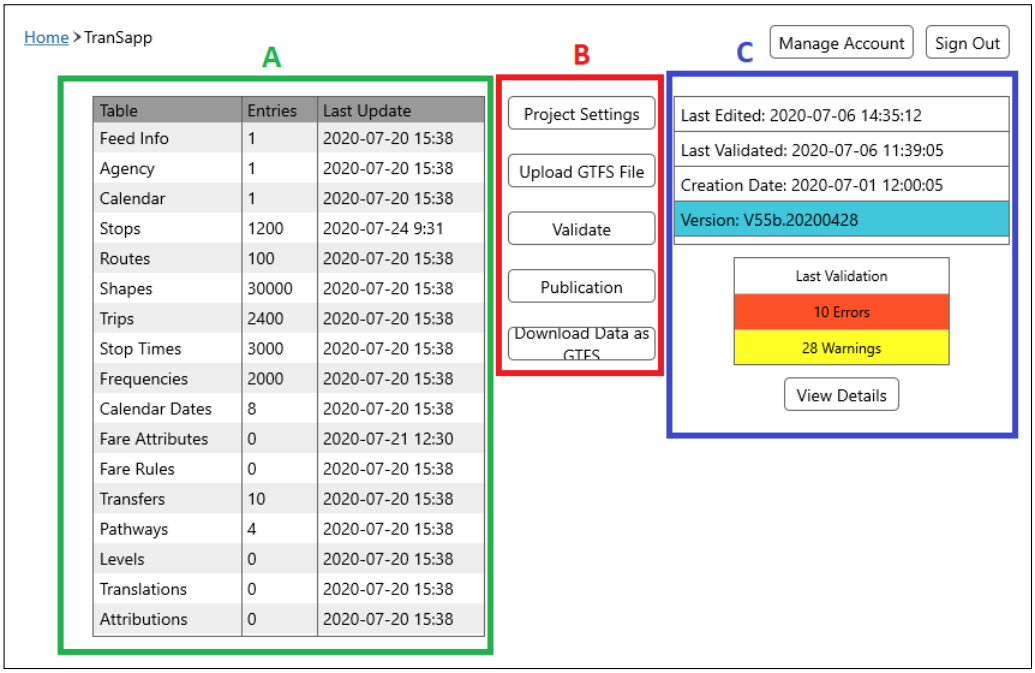


Figura 3.4: Mockup de vista de visión general de proyecto

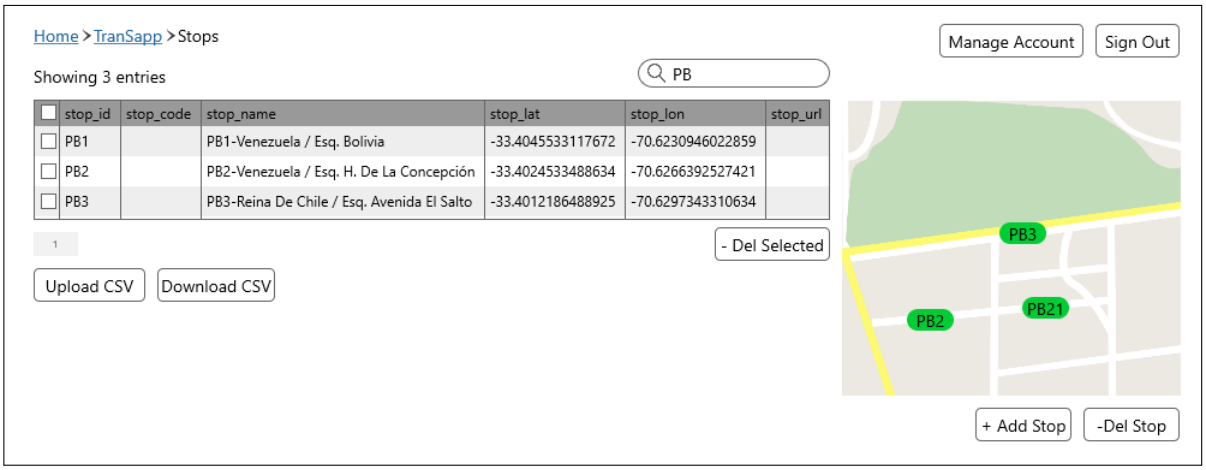


Figura 3.5: Mockup de vista de edición de stops

El mockup presentado en la figura 3.5 corresponde a la edición de la tabla de stops. Como podemos ver esta vista se compone de una tabla con los paraderos acompañada de un mapa interactivo, en el cual el usuario puede ver las ubicaciones de los paraderos y editarlas arrastrando los paraderos en el mapa.

En la figura 3.6 está el mockup de la vista de edición de `stop-times`. En la parte A tenemos un selector de trip, el cual nos permite elegir a cuál trip corresponde el `stop-times` que se está editando (cabe destacar que un trip solo puede tener una secuencia de `stop-times` asociada). El mapa del sector A muestra los paraderos para poder asociarlos al viaje seleccionado y, si dispusiera de uno, también muestra el `shape` para ayudar a elegir los paraderos. En el sector B tengo una tabla sencilla que muestra la secuencia de paraderos para el trip seleccionado y las horas de llegada y partida a cada uno de ellos. En el sector C hay

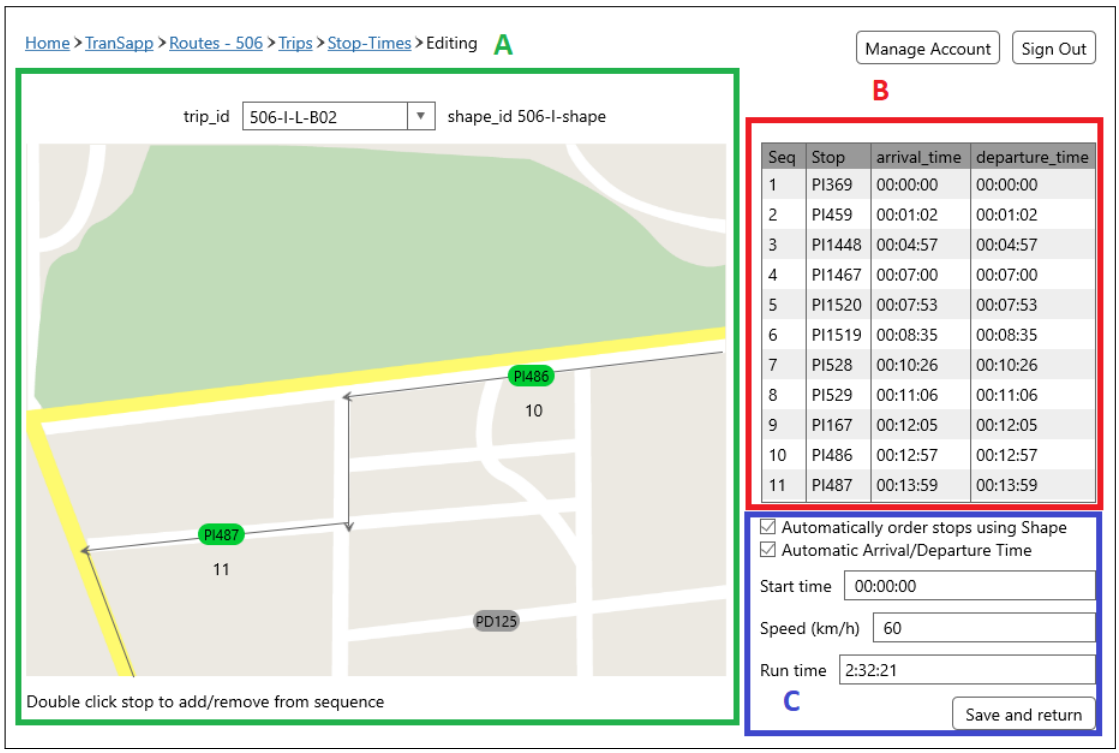


Figura 3.6: Mockup de vista de edición de `stop-times`

algunas opciones útiles para la edición de `stop-times`, las cuales son dependientes de que el trip tenga un `shape` asociado. Si no hubiese un `shape` asociado se oscurecerían para indicar que no son utilizables. La primera opción permitiría realizar un mapeo de las posiciones en paraderos al `shape`, permitiendo entonces ordenar los paraderos según su posición en el `shape`. La segunda opción utiliza lo mismo en conjunto con ya sea la velocidad o la duración del viaje para calcular cuándo pasaría por cada paradero. Al llenar el campo de velocidad el de duración del viaje se llena automáticamente y vice-versa.

3.5. Modelo de datos

Habiendo analizado los datos entregados por el DTPM y estudiado la referencia de archivos GTFS de Google, se ha diseñado un modelo relacional de la base de datos para ser utilizado por la aplicación. El modelo contiene 19 relaciones, la mayoría de las cuales corresponden a las tablas del archivo GTFS. En la figura 3.7 podemos ver las principales relaciones del modelo de datos. Las tablas omitidas son aquellas tablas sin llaves foráneas que no referencian `projects` y aquellas que simplemente establecen una relación entre objetos pre-existentes, por ejemplo `transfers` indica que se puede realizar una transferencia entre dos paraderos. En este diagrama las líneas simbolizan relaciones 1-n, marcando con un punto el "n". Una línea continua indica que la llave foránea es utilizada como llave primaria, mientras que la línea punteada indica que no. Como Django no permite usar distintos esquemas o bases de datos para los distintos proyectos, se ha tenido que incluir el proyecto como llave foránea para todas las tablas mostradas, ya sea directamente o a través de una referencia indirecta (por ejemplo, `frequencies` utiliza el proyecto del `trip` que referencia).

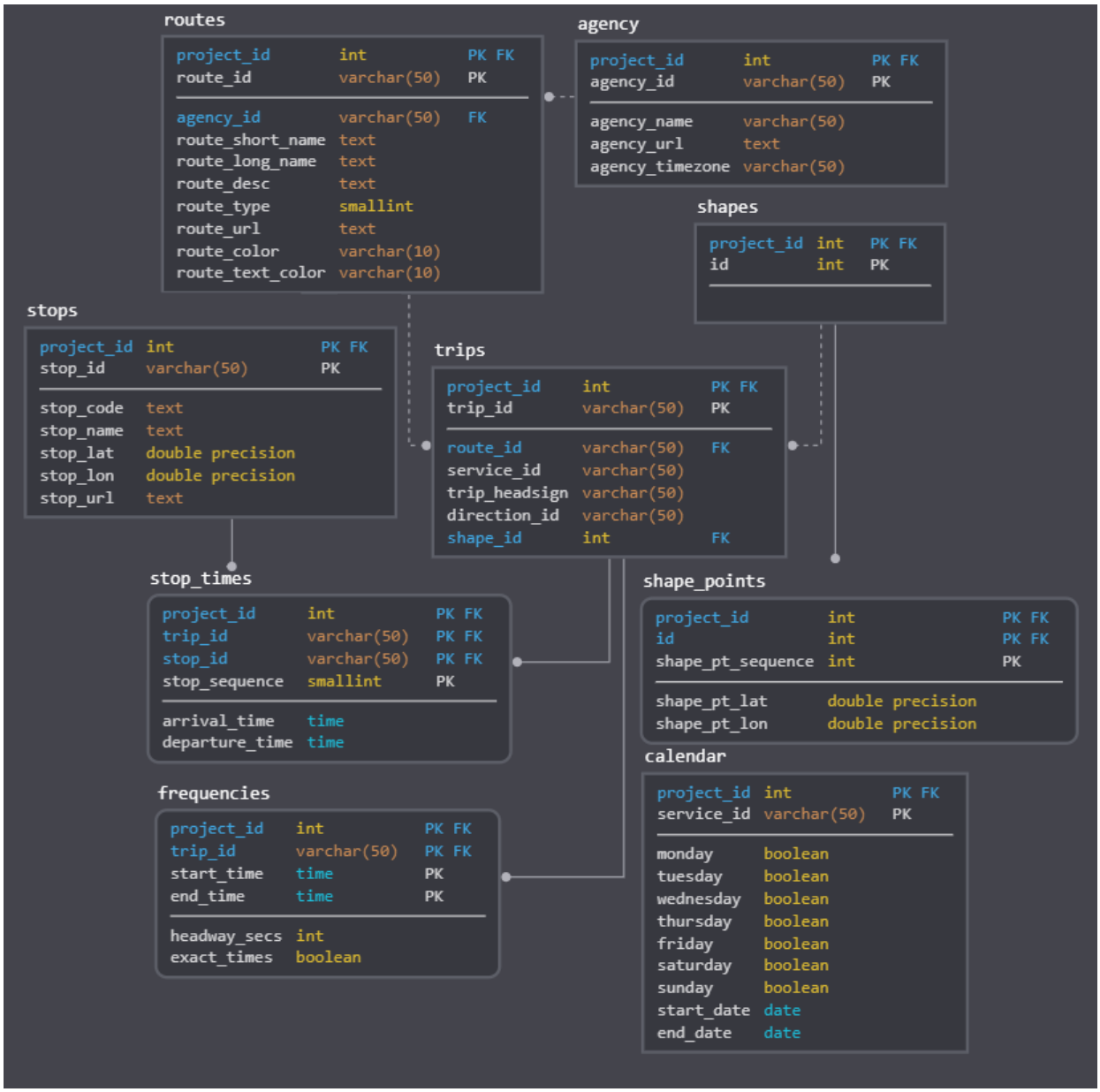


Figura 3.7: Esquema relacional describiendo las principales tablas

3.6. Herramientas geográficas

En esta sección se describirán algunas de las herramientas del proyecto que han resultado cruciales a la hora de desarrollar los mapas interactivos. Todas estas herramientas operan con datos en formato GeoJSON (RFC 7946), un formato basado en JSON diseñado para describir datos geográficos. Permite distintos tipos de elementos geográficos y admite propiedades para ellos.

3.6.1. MapBox

MapBox es un producto desarrollador por una empresa del mismo nombre que, utilizando los datos de OpenStreetMap (OSM), ofrece herramientas para visualizar, interactuar, navegar, entre otros. OSM es un proyecto abierto con el objetivo de crear un mapa editable de todo el mundo. Para desarrollar el editor de GTFS se utilizó la herramienta de mapas, la cual permite generar visualizaciones sobre un mapa navegable y agregar funcionalidad a este mapa, como por ejemplo poner los paraderos en un mapa y que el usuario pueda arrastrarlos para cambiar sus posiciones.

3.6.2. Map-Matching

Map-Matching es una herramienta ofrecida por MapBox que permite generar rutas en base a un mapa. La API de Map-Matching es un componente del servicio de navegación de MapBox que en base a una secuencia de puntos y un margen de distancia genera un GeoJSON describiendo una polilínea que pasa por todos los puntos.

3.6.3. Turf.js

Turf.js es una librería de Javascript que contiene varias herramientas para trabajar con información geográfica. En particular, se utiliza el método *nearestPointOnLine*, el cual toma una polilínea de GeoJSON y un punto como argumentos y entrega el punto en la polilínea más cercano, con una propiedad indicando la posición en la polilínea desde el origen en kilómetros del punto entregado.

Capítulo 4

Implementación

En este capítulo analizaremos la implementación del proyecto. Con esto en mente, realizaremos un análisis de cómo han sido implementados los distintos componentes del editor de GTFS. En la sección 4.1 veremos la implementación del backend, llegando hasta las APIs REST. Dentro de la sección 4.2 estudiaremos el frontend, empezando por las interfaces. Por último, en la sección 4.3 se describirán las vistas con mapa interactivo.

4.1. Backend

A grandes rasgos el backend del proyecto consiste en una aplicación de Django que levanta una API REST, exponiendo endpoints para consultar, modificar, crear y eliminar las entradas de un GTFS. Esta API utiliza modelos de Django y Django REST framework para manejar los datos.

4.1.1. Los Modelos

En base al modelo de datos explorado en el capítulo anterior se han implementado modelos de Django. Los modelos de Django corresponden a clases de python que describen tipos de datos, los cuales Django es capaz de convertir a SQL [9]. Al correr el comando `makemigrations`, Django traduce los cambios en los modelos a migraciones, las cuales corresponden a archivos de python que describen cómo actualizar la base de datos para que represente los cambios realizados en los modelos[1]. Tras haber creado una o más migraciones, el comando `migrate` ejecuta los comandos de SQL necesarios para actualizar la base de datos. Los modelos apoyan con la solución de las historias de usuario **H1** y **H3**, esto pues los modelos permiten la manipulación de los datos manejados por la aplicación.

Con el objetivo de permitir ciertas operaciones comunes sobre varios modelos, se implementaron managers, los cuales corresponden a objetos que definen ciertas funciones que operan sobre todas las entradas de un modelo. Los modelos de Django vienen por defecto con un manager llamado `Objects`. Dado que en el modelo todos los objetos van asociados

[1]Las migraciones de este proyecto se pueden encontrar en `https://github.com/InspectorIncognito/gtfs-editor-backend/tree/dev/rest_api/migrations`

a un proyecto se creó un `Manager` llamado `FilterManager`, cuya implementación se puede observar en el snippet 4.1. El método `filter_by_project` de la clase entrega los objetos filtrados por un proyecto particular. El parámetro `project_filter`, que aparece en la línea 4, define el string utilizado por el filtro de Django para filtrar según el proyecto. Por ejemplo, como el modelo de `Frequencies` tiene una referencia a `Trips`, entonces el filtro de proyecto sería `trip__project__project_id`. Un doble guión bajo sirve para acceder a un attributo de una llave foránea, o sea, `trip__project` mira el proyecto asociado al Trip.

Snippet 4.1: FilterManager

```
1  from django.db import models
2
3  class FilterManager(models.Manager):
4      def __init__(self, project_filter='project_id'):
5          super().__init__()
6          self.project_filter = project_filter
7
8      def filter_by_project(self, project_id):
9          return self.get_Queryset().filter(**{self.project_filter: project_id})
```

Adicionalmente, muchas de las tablas de un GTFS tienen una ID asociada a cada entrada, por lo que se definió un `InternalIDFilterManager`, cuya implementación corresponde al código del snippet 4.2, el cual tiene métodos para seleccionar objetos. En la línea 6 tenemos el método `select_by_internal_id`, este toma como parámetro el ID del proyecto y un ID de GTFS y entrega un Queryset con este objeto. El método `multiselect_by_internal_ids` funciona similarmente, pero toma una lista de IDs como argumento y devuelve un Queryset con todos los objetos cuya ID se encuentre en esa lista.

Snippet 4.2: InternalIDFilterManager

```
1  class InternalIDFilterManager(FilterManager):
2      def __init__(self, id_filter, project_filter='project_id'):
3          super().__init__(project_filter)
4          self.id_filter = id_filter
5
6      def select_by_internal_id(self, project_id, internal_id):
7          return self.filter_by_project(project_id).filter(**{self.id_filter: internal_id})
8
9      def multiselect_by_internal_ids(self, project_id, internal_ids):
10         internal_ids = list(internal_ids)
11         return self.filter_by_project(project_id).filter(**{self.id_filter + '__in':
           ↪  internal_ids})
```

Como vimos en el capítulo anterior, hay tablas con mayor complejidad, vale decir, aquellas que son referenciadas o referencian a otra tabla. A continuación estudiaremos la implementación de algunas de ellas con modelos de Django. Como puedo tener distintos GTFS en edición, cada modelo posee una referencia a su respectivo proyecto.

Agencies

El modelo de agencias, presente en el snippet 4.3, es bastante sencillo. Contiene una llave foránea a un proyecto y una restricción de unicidad de su ID con su proyecto, vale decir, un proyecto no puede tener agencias distintas con el mismo ID. Se utiliza el parámetro *on_delete=models.PROTECT* para las llaves foráneas pues así si intento borrar un objeto, Django lanza un error advirtiendo que está siendo referenciado. Esto es útil para entregar un mensaje al usuario si no he podido borrar una entrada.

Snippet 4.3: Modelo de Agency

```
class Agency(models.Model):
    project = models.ForeignKey(Project, on_delete=models.PROTECT)
    agency_id = models.CharField(max_length=50)
    agency_name = models.CharField(max_length=50)
    agency_url = models.URLField()
    agency_timezone = models.CharField(max_length=20)
    agency_lang = models.CharField(max_length=10, null=True, blank=True)
    agency_phone = models.CharField(max_length=20, null=True, blank=True)
    agency_fare_url = models.URLField(max_length=255, null=True, blank=True)
    agency_email = models.EmailField(max_length=255, null=True, blank=True)
    objects = InternalIDFilterManager('agency_id')

    class Meta:
        unique_together = ['project', 'agency_id']
```

Routes

El modelo de rutas no es mucho más complejo que el de agencias, pero como podemos ver en el snippet 4.4 no dispone de una referencia al proyecto. En el modelo de datos la llave foránea al proyecto queda explícita, pero al referenciar esto queda implícito, encargándose Django automáticamente de manejarla.

Snippet 4.4: Modelo de Route

```
class Route(models.Model):
    agency = models.ForeignKey(Agency, on_delete=models.PROTECT)
    route_id = models.CharField(max_length=50)
    route_short_name = models.CharField(max_length=50, null=True, blank=True)
    route_long_name = models.CharField(max_length=200, null=True, blank=True)
    route_desc = models.CharField(max_length=50, null=True, blank=True)
    route_type = models.IntegerField()
    route_url = models.URLField(null=True, blank=True)
    route_color = models.CharField(max_length=10, null=True, blank=True)
    route_text_color = models.CharField(max_length=10, null=True, blank=True)
    objects = InternalIDFilterManager('route_id', 'agency__project__project_id')
```

StopTimes

En el snippet 4.5 tenemos el modelo de StopTimes. Este modelo establece una relación entre paraderos y viajes, estableciendo una secuencia de paraderos por las que pasa el viaje. Por esto tengo una restricción de unicidad del viaje y el `stop_sequence`, un viaje no puede tener dos segundos paraderos. También se establece un *related_name* para la referencia a *Trip*, esto permite de forma más sencilla obtener los StopTimes asociados a un viaje.

Snippet 4.5: Modelo de StopTimes

```python
class StopTime(models.Model):
    trip = models.ForeignKey(Trip, on_delete=models.PROTECT, related_name='stop_times')
    stop = models.ForeignKey(Stop, on_delete=models.PROTECT)
    stop_sequence = models.IntegerField()
    arrival_time = models.DurationField(null=True, blank=True)
    departure_time = models.DurationField(null=True, blank=True)
    stop_headsign = models.CharField(max_length=50, null=True, blank=True)
    pickup_type = models.IntegerField(null=True, blank=True)
    drop_off_type = models.IntegerField(null=True, blank=True)
    continuous_pickup = models.IntegerField(null=True, blank=True)
    continuous_dropoff = models.IntegerField(null=True, blank=True)
    shape_dist_traveled = models.FloatField(null=True, blank=True)
    timepoint = models.IntegerField(null=True, blank=True)
    objects = FilterManager('trip__project')

    class Meta:
        unique_together = ['trip', 'stop_sequence']
```

4.1.2. Serializers

Los serializadores de Django son clases que convierten objetos de Django a JSON y viceversa, esto con el fin de permitir la transferencia de datos desde y hacia la API. Para esto se debe configurar cuales son los parámetros con los que opera el serializer. El snippet 4.6 muestra el serializador de el modelo de agencias. En la clase *Meta* se define el modelo, los campos y cuáles son de solo lectura. Además, se ha incluido un validador de zonas horarias. Con este validador se revisa que la zona horaria corresponda a un formato soportado, arrojando un error si se intenta utilizar una zona horaria en otro formato.

Snippet 4.6: Serializador de Agency

```python
class AgencySerializer(NestedModelSerializer):
    agency_timezone = serializers.CharField(validators=[validators.timeZoneValidator])
    class Meta:
        model = Agency
        fields = ['id', 'agency_id', 'agency_name', 'agency_url', 'agency_timezone',
        ↪  'agency_lang', 'agency_phone', 'agency_fare_url', 'agency_email']
        read_only = ['id']
```

Para el modelo de Trips defino un serializador que permite la edición y creación de StopTimes directamente, esto pues muchas veces al editar los StopTimes se deben editar varios de los asociados a un Trip simultáneamente y esto simplifica estas ediciones pues permite que la operación falle o tenga éxito de forma atómica. Un ejemplo es que al insertar un paradero en medio de un viaje esto desplaza los *stop_sequence* en 1 para todos los StopTimes posteriores. En el snippet 4.7 está el código fuente de esto, el método *update* saca los *stop_times* de el objeto a actualizar, ejecuta la acción y reemplaza los *stop_times* antiguos por los nuevos. Como se ocupó *transaction.atomic()*, un error produce que todos los cambios se reviertan, evitando estados inconsistentes de los datos.

Snippet 4.7: Serializer de StopTimes

```
1  class TripSerializer(NestedModelSerializer):
2      route_id = serializers.CharField(source='route.route_id', read_only=True)
3      shape_id = serializers.CharField(source='shape.shape_id', read_only=True)
4      stop_times = SimpleStopTimeSerializer(many=True, required=False)
5      ...
6      def simplify_data(self, data):
7          return {k: v for (k, v) in data.items() if k != 'stop_times'}
8      ...
9      def update(self, instance, validated_data):
10         try:
11             with transaction.atomic():
12                 instance = super().update(instance, self.simplify_data(validated_data))
13                 if 'stop_times' in validated_data:
14                     StopTime.objects.filter(trip=instance).delete()
15                     stop_times = map(lambda st: StopTime(trip=instance, **st),
                       ↪  validated_data['stop_times'])
16                     StopTime.objects.bulk_create(stop_times)
17                 return instance
18         except IntegrityError as error:
19             raise ValidationError(error)
```

El Serializer de Shape es análogo al de Trips, pero posee una variante que entrega información simplificada para evitar sobrecargar la API (ver snippet 4.8). Su clase Meta define el campo *point_count* que no existe en el modelo, por esto, en la línea 2 se define un atributo *point_count* y se emplea *SerializerMethodField* para establecer que su valor se obtiene de un método, el cual simplemente devuelve la cantidad de puntos asociados al Shape.

Snippet 4.8: Serializador de Shape simple

```
1  class ShapeSerializer(NestedModelSerializer):
2      point_count = serializers.SerializerMethodField()
3
4      class Meta:
5          model = Shape
6          fields = ['id', 'shape_id', 'point_count']
7          read_only = ['id']
```

```
8
9       def get_point_count(self, obj):
10          return obj.points.count()
```

4.1.3. Views

Las vistas de Django definen las operaciones que se realizan al llamar un endpoint de la API REST. En el archivo `urls.py` se define un `Router`, el cual procesa las URLs y establece cual vista es responsable de atender la solicitud. Luego las vistas se hacen cargo de procesar la solicitud, realizar las operaciones requeridas y entregar un resultado. Django cuenta con un `ModelViewSet`, clase que contiene mixins para permitir varias operaciones básicas sobre el modelo en base a configuración de la clase. En el caso de python, un mixin es un patrón de diseño que corresponde a una clase sin estado utilizada para agregar funcionalidad a un objeto, en el caso de los `Serializers` de Django REST utilizan una clase `Meta` de configuración para parametrizar las vistas, requiriendo al menos el modelo y los campos. Las operaciones son: 1) listar los objetos, 2) obtener el detalle de un objeto específico, 3) crear un objeto, 4) eliminar un objeto y 5) actualizar un objeto. Para las necesidades de la API, esta clase resultó insuficiente, por lo que fue necesario extender estas clases para generalizar ciertos comportamientos importantes para la aplicación, tales como la subida y bajada de archivos CSV. Estas extensiones han sido realizadas a través de mixins que implementan estos comportamientos comunes en base a parámetros. La creación de estos mixins ha sido sumamente importante para el proyecto, esto pues reduce la duplicación de código y simplifica la edición de vistas a un problema de configuración en vez de implementación. Estas vistas se encuentran ligadas a todas las historias de usuario, esto pues son la manera de la que la interfaz interactúa con los datos, pero sobre todo las historias de usuario **H1**, **H3** y **H4**. Las historias **H1** y **H3** requieren las vistas básicas para la manipulación de los datos, mientras que las vistas de subida y bajada de CSV contribuyen a la solución de las historias, **H3** y **H4**.

Las tareas de subida y bajada de archivos CSV son bastante complejas. Esto pues, al estar los datos organizados como tablas en archivos CSV, se debe traducir entre el modelo SQL/Django y el formato requerido. Puesto que estas operaciones modifican varias entradas, es necesario realizar operaciones masivas para evitar que se generen una o más consultas a la base de datos por cada entrada modificada. Como Django ocupa IDs numéricas internas para representar los valores, cada vez que hay una referencia a un valor foráneo se debe traducir la ID de Django (número) a la ID del CSV (texto), y vice-versa, pero agrupando las operaciones por la escala. Otra consideración importante es que si el CSV modifica datos pre-existentes se deben mantener las IDs. Por ejemplo, si genero un nuevo CSV con un nuevo formato de nombres para los paraderos, es importante no borrar los paraderos para que las referencias a ellos en stop-times se mantengan. También existe el caso de la autoreferencia a Stops en el atributo `parent_stop`.

En el snippet 4.9 se muestra el mixin implementado para la descarga de modelos como CSV. Se ha separado el procesamiento la escritura del CSV del llamado al método `download` (línea 24) de la vista porque para crear un GTFS es necesario escribir varios archivos `.txt` y luego comprimirlos, y esto permite simplemente utilizar los `write_to_file` de las vistas para crear el GTFS. En la línea 33 se utiliza un decorador de python que define download como una

de las vistas del ViewSet, luego download crea un `HttpResponse` y llama a `write_to_file`. `write_to_file` se encuentra en la línea 4 del snippet y simplemente escribe el encabezado del CSV y luego para cada objeto obtiene los valores en el orden del encabezado y los escribe en una línea.

Snippet 4.9: Mixin de descarga como CSV

```
1  class CSVDownloadMixin:
2
3      @staticmethod
4      def write_to_file(out_file, meta_class, qs):
5          meta = meta_class()
6          header = meta.csv_header
7          writer = csv.writer(out_file)
8          csv_fields = [e for e in getattr(meta, 'csv_fields', meta.csv_header)]
9          csv_field_mappings = getattr(meta, 'csv_field_mappings', {})
10         for k in csv_field_mappings:
11             csv_fields[csv_fields.index(k)] = csv_field_mappings[k]
12         # First we write the header
13         writer.writerow(header)
14         for obj in qs.values(*csv_fields):
15             # We transform the types that need transforming, for instance the booleans
16             # into 0-1 and the dates get formatted
17             meta.convert_values(obj)
18             row = list()
19             for k in csv_fields:
20                 row.append(obj[k])
21             writer.writerow(row)
22
23     @action(methods=['get'], detail=False, renderer_classes=(BinaryRenderer,))
24     def download(self, *args, **kwargs):
25         try:
26             meta = self.Meta()
27             filename = meta.csv_filename
28             header = meta.csv_header
29             qs = self.get_Queryset()
30         except AttributeError as err:
31             print(err)
32             return HttpResponse('Error: endpoint not correctly implemented, check Meta
               ↪  class.\n{0}'.format(str(err)),
33                                 status=status.HTTP_501_NOT_IMPLEMENTED)
34         response = HttpResponse(content_type='text/csv')
35         response['Content-Disposition'] = 'attachment;
           ↪  filename="{}.csv"'.format(filename)
36         self.write_to_file(response, self.Meta, qs)
37         return response
```

Para la subida de CSV se dispone de un mixin que es bastante más complejo que el de bajada, compartiendo como rasgos el decorator para definir la vista y la separación del llamado a la vista y la ejecución de las operaciones (para la subida de GTFS esta vez). La implementación de este mixin está documentada de forma abreviada en el snippet 4.10. La primera implementación de este mixin tenía problemas de eficiencia. En cargar y procesar las más de 375.000 líneas del CSV de Shapes de Santiago se demoraba un poco más de media hora. Tras analizar los tiempos de ejecución de las distintas partes se determinó que gran parte de ese tiempo se gastaba en las operaciones de actualización de los modelos. Tras investigar posibles soluciones se descubrió que Django posee operaciones de creación y actualización masivas, `bulk_create` y `bulk_update`. Para arreglar este problema se decidió agrupar estas operaciones en bloques. Esto permite a la aplicación realizar menos consultas de SQL.

El método *update_or_create_chunk* del snippet 4.10 está encargado de procesar los datos para su creación y/o actualización. En las líneas 5-10 se crean diccionarios de python con las llaves foráneas de forma eficiente, esto pues Django internamente trabaja con números como identificadores por defecto y realizar una consulta por entrada resulta bastante lento. En las líneas 11-35 se realiza la conversión de los datos para realizar las operaciones. Tras traducir las IDs del CSV (texto) a IDs de Django (números), se aplican métodos de preprocesamiento de los datos (por ejemplo convertir strings de fecha a datetime), se cambian los nombres de las columnas cuyo nombre interno no corresponde al del estándar y se incluye el id del proyecto si corresponde. En las líneas 36-57 se determina cuáles entradas van a ser actualizadas y creadas, esto es importante pues, ya que los modelos contienen llaves foráneas, no se puede simplemente borrar y volver a crear las entradas, sino que se deben actualizar las que ya existen y crear las que no. Para esto se genera un id_map, revisando cuales entradas ya existen. Como el modelo de GTFS utiliza IDs internas para referencias entre tablas, aquellas tablas que no las utilizan no pueden ser referenciadas por otra tabla. Esto permite simplemente eliminarlas todas y crearlas nuevamente. Por último, en las líneas 59-64, simplemente se ejecutan las operaciones de `bulk_create` y `bulk_update`.

Snippet 4.10: Mixin de subida de CSV abreviado

```
1  class CSVUploadMixin:
2      def update_or_create_chunk(self, chunk, project_pk, id_set, meta):
3          ...
4          # For each foreign key create a hashmap mapping the GTFS IDs to django model IDs
5          for fk in foreign_key_mappings:
6              foreign_key_maps[fk['csv_key']] = create_foreign_key_hashmap(chunk,
               ↪  fk['model'], project_pk, fk['csv_key'], fk['model_key'])
7              if 'internal_key' not in fk:
8                  fk['internal_key'] = fk['model_key']
9          for row in chunk:
10             # First we replace the foreign keys
11             for fk in foreign_key_mappings:
12                 k = fk['csv_key']
13                 id_map = foreign_key_maps[k]
14                 if k not in row:
15                     continue
```

```
                val = id_map[row[k]]
                if fk['csv_key'] != fk['internal_key']:
                    del row[k]
                row[fk['internal_key']] = val
            # Then we do all processing required on the data (like date formatting)
            for k in preprocess_funcs:
                if k in row and row[k] is not None:
                    row[k] = preprocess_funcs[k](row[k])
            # Rename then entries that need it
            for k in rename_fields:
                val = row[k]
                del row[k]
                row[rename_fields[k]] = val
            # Include project_id if the model requires it
            if include_project_id:
                row['project_id'] = project_pk

        to_create = list()
        to_update = list()
        # if using internal IDs we have to choose whether we create or update each row
        if use_internal_id:
            # using the name of the GTFS ID we create a map for the model itself, to be
            ↪ used in the update
            internal_id = model.objects.get_internal_id_name()
            id_map = create_foreign_key_hashmap(chunk, model, project_pk, internal_id,
            ↪ internal_id)

            for row in chunk:
                # We store the internal ID so we don't delete the entries afterwards
                id_set.add(row[internal_id])
                if row[internal_id] in id_map:
                    row['id'] = id_map[row[internal_id]]
                # Create a model but don't save it! we don't want to perform one SQL
                ↪ operation per entry
                obj = model(**row)
                # if the row already existed we prepare it for updating
                if row[internal_id] in id_map:
                    to_update.append(obj)
                # otherwise we prepare it for creation
                else:
                    to_create.append(obj)
        # If not using internal IDs we just create every row
        else:
            for row in chunk:
                to_create.append(model(**row))
        # Then we simply create the new objects and update the existing ones
        t1 = time.time()
```

```
60          model.objects.bulk_create(to_create, batch_size=1000)
61          t2 = time.time()
62          if use_internal_id:
63              model.objects.bulk_update(to_update, params, batch_size=1000)
64              t3 = time.time()
```

Para ilustrar como se configuran las vistas, veremos 2 ejemplos en esta sección. El primer ejemplo corresponde al snippet 4.11. Se define el nombre del CSV generado (línea 5), cuáles son los encabezados del CSV (línea 6), el modelo y el diccionario de preprocesamiento que traduce los strings de fecha a objetos de tipo fecha (líneas 7-12). Además se define el Queryset por defecto en la línea 16, esto es una consulta de Django que es usada por las vistas. Este Queryset viene ordenado por defecto para mantener consistencia en el orden de los datos, tanto para efecto de consistencia para el usuario como para simplificar el testeo.

Snippet 4.11: Viewset de Calendar

```
1  class CalendarViewSet(CSVHandlerMixin, MyModelViewSet):
2      serializer_class = CalendarSerializer
3
4      class Meta(ConvertValuesMeta):
5          csv_filename = 'calendars'
6          csv_header = ['service_id', 'monday',  'tuesday', 'wednesday', 'thursday',
           ↪  'friday', 'saturday', 'sunday', 'start_date', 'end_date']
7          model = Calendar
8          foreign_key_mappings = {}
9          upload_preprocess = {
10             'start_date': lambda date: datetime.datetime.strptime(date, '%Y%m%d'),
11             'end_date': lambda date: datetime.datetime.strptime(date, '%Y%m%d'),
12         }
13
14     @staticmethod
15     def get_qs(kwargs):
16         return
           ↪  Calendar.objects.filter(project=kwargs['project_pk']).order_by('service_id')
```

En el snippet 4.12 se utilizan algunos de los parámetros avanzados, con `csv_fields` se define que `agency_id` en el CSV se traduce a `agency` en el modelo. En las líneas 14-21 se define el arreglo `foreign_key_mappings`, este arreglo contiene dicts de python que configuran la interpretación de llaves foráneas. `csv_key` corresponde al encabezado de la columna en el CSV correspondiente al campo, `model` es el modelo que la llave foránea referencia, `model_key` establece el nombre del ID interno en el modelo e `internal_key` indica como obtener el ID del objeto.

Snippet 4.12: Viewset de Route

```
1  class RouteViewSet(CSVHandlerMixin, MyModelViewSet):
2      serializer_class = RouteSerializer
```

```
3
4       class Meta(ConvertValuesMeta):
5           csv_filename = 'routes'
6           csv_header = ['route_id', 'agency_id', 'route_short_name', 'route_long_name',
            ↪  'route_desc', 'route_type', 'route_url', 'route_color', 'route_text_color']
7           csv_fields = [e for e in csv_header]
8           csv_fields[1] = 'agency'
9           csv_field_mappings = {
10              'agency': 'agency__agency_id'
11          }
12          model = Route
13          include_project_id = False
14          foreign_key_mappings = [
15              {
16                  'csv_key': 'agency_id',
17                  'model': Agency,
18                  'model_key': 'agency_id',
19                  'internal_key': 'agency_id'
20              }
21          ]
22          search_fields = ['route_id', 'agency__agency_id']
23
24      @staticmethod
25      def get_qs(kwargs):
26          return Route.objects.filter(agency__project =
            ↪  kwargs['project_pk']).order_by('route_id')
```

Una vez configurados los ViewSets, estos se agregaron al archivo `urls.py`. Django tiene Routers, objetos que realizan la traducción de una URL relativa a una vista de Django. Es un patrón de Django tener un archivo `urls.py`, en el cual se registran las vistas de la aplicación asociadas a URLs. Como un ViewSet define varias vistas, al registrarlo se agregan automáticamente todas las vistas contenidas en el ViewSet. Al agregar todos los ViewSets definidos al Router, se generan endpoints para todos los modelos creados.

Originalmente se creó un ViewSet para la bajada como GTFS. Para esto se utilizaron los métodos de `write_to_file` del mixin de descarga (línea 4 del snippet 4.9). El proceso, implementado en el snippet 4.13, consiste en crear un archivo `.zip` vacío, escribir cada uno de los archivos usando las respectivas vistas y entregar el archivo generado. Como la tarea era lenta, Felipe Hernández se hizo cargo de refactorizarlo y separar la tarea de generación del archivo y la tarea de descarga. Además, Hernández hizo algo similar para la subida de GTFS, aprovechando el mixin de subida de CSV. Estos ViewSets permiten que el backend realice las operaciones requeridas para las historias de usuario **H2** y **H3**.

Snippet 4.13: Proceso de descarga

```
1   # files es un dict que mapea nombres de archivos a vistas ('agency': AgencyViewSet)
2   zf = zipfile.ZipFile(s, "w", zipfile.ZIP_DEFLATED, False)
```

```
for gtfs_filename in files:
    out = StringIO()
    view = files[gtfs_filename]
    qs = view.get_qs({'project_pk': project_obj.pk})
    view.write_to_file(out, view.Meta, qs)
    zf.writestr('{}.txt'.format(gtfs_filename), out.getvalue())
zf.close()
```

4.2. Frontend

En esta sección inspeccionaremos la implementación del frontend de la aplicación. Con este frontend se busca cumplir lo planteado en el segundo objetivo específico, crear las vistas que permiten al usuario manipular los datos de un GTFS. Como ha sido descrito en la sección 3.2.2, esta implementación utiliza el framework Vue con algunas librerías. Como la librería Vuetable no soporta tablas editables por defecto, se tuvieron que incorporar inputs de distintos tipos para los campos para permitir la edición de datos. Los campos que hacen referencia a otros modelos tienen un selector que obtiene las opciones de la API y los campos que tienen una cantidad limitada de posibles valores ocupan un selector con opciones estáticas.

4.2.1. Vista básica de tabla

Todos los modelos de un GTFS excepto `feed-info`, `shapes` y `StopTimes` poseen vistas de tabla editable similares, por esto se decidió crear un componente de Vue que generaliza este comportamiento y a través de configuración es capaz de definir cómo mostrar y editar los datos. Para ilustrar cómo operan estas tablas, veremos la que representa al modelo de Routes.

En la figura 4.1 podemos ver la tabla de Routes. Las columnas de acciones de las tablas básicas poseen por defecto la acción de eliminar una entrada, además de cualquier acción extra que haya añadido. En el caso particular de la tabla de Routes, se añadió una acción que permite abrir los Trips asociados a esa ruta, la cual es añadida usando un `slot` de Vue. Un `slot` de Vue corresponde a un tag de HTML propio de Vue que puede ser reemplazado por el padre de un componente. El HTML interno del slot es usado si el padre no lo utiliza y reemplazado si lo utiliza. Las líneas 8-9 del snippet 4.16 corresponden al `slot` que permite agregar botones adicionales, mientras que las líneas 6-8 del snippet 4.14 contienen un ejemplo de uso de un slot, reemplazando el contenido por defecto por un botón con un ícono.

Snippet 4.14: HTML de vista de ruta

```
<template>
  <div>
    <EditableTable :fields="fields" :url="url" :updateMethod="update"
    ↪  :deleteMethod="remove" :createMethod="create"
      :downloadURL="downloadURL" :uploadCSV="uploadCSV" :searchable="true"
      ↪  :infoURL="infoURL">
      <template slot="additional-actions" slot-scope="props">
```

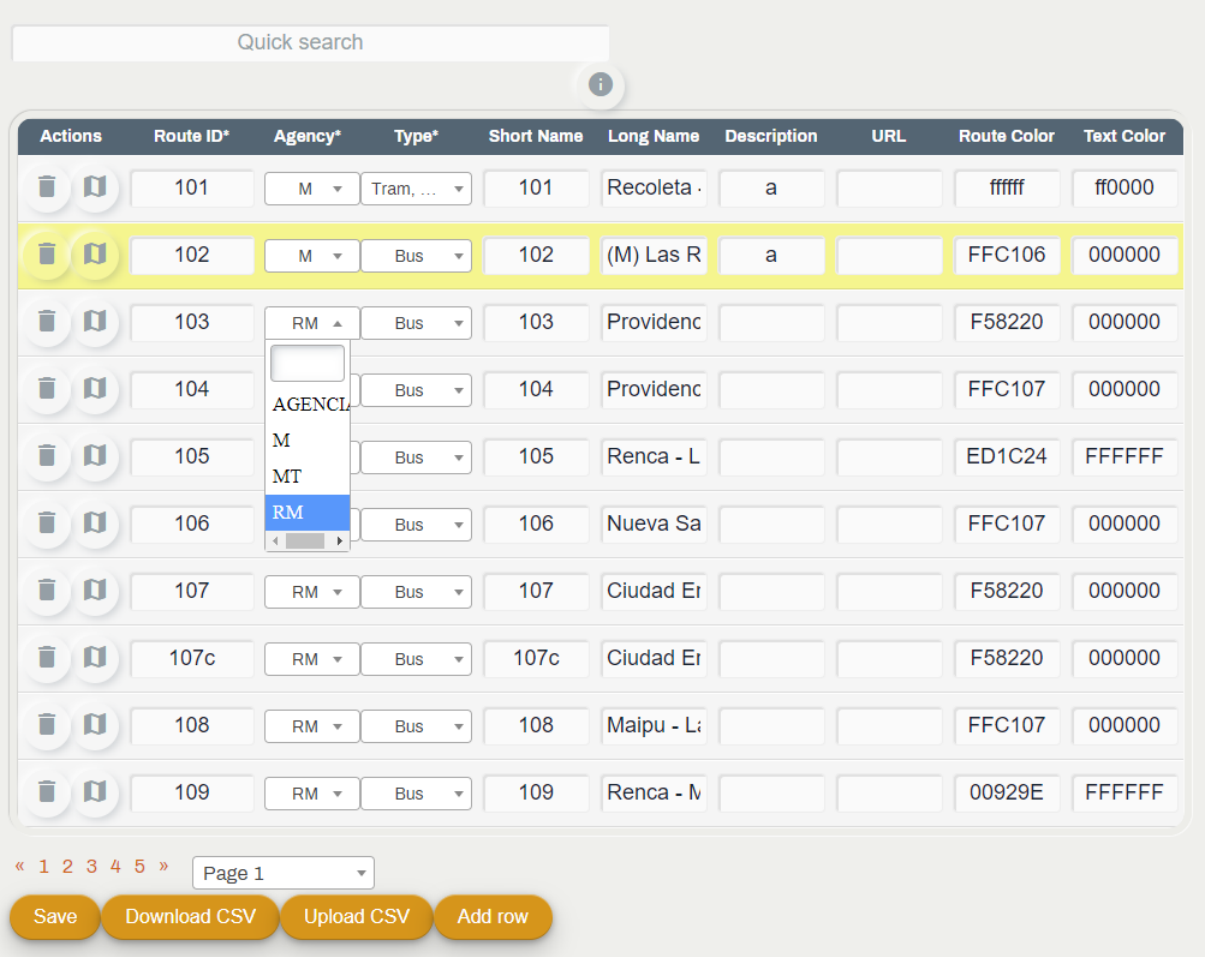


Figura 4.1: Vista de tabla de rutas

```
6           <button class="btn icon" @click="goToRoute(props)" alt="Go to trips">
7             <span class="material-icons">map</span>
8           </button>
9         </template>
10      </EditableTable>
11    </div>
12  </template>
```

Al final de cualquier tabla se muestran cuatro botones (ver figura 4.1). El fondo de una fila se pone amarillo cuando hay cambios sin guardar, y rojo si ha habido un error al guardar (al ocupar el botón *Save*). Adicionalmente, los datos de todas las tablas pueden ser descargados y subidos como CSV con los botones de *Download CSV* y *Upload CSV*, abriendo un Modal con un selector de archivo para la subida. El botón de *Add row* abre un Modal con un formulario estándar de HTML que permite agregar una nueva entrada a la tabla.

Para configurar las columnas de una tabla editable se utiliza un arreglo de JSON que es entregado como parámetro a la tabla editable. Este formato cumple con la especificación de Vuetable, pero también incluye parámetros adicionales para configurar los inputs. En el

snippet 4.15 podemos apreciar parte del JSON que configura la tabla de rutas. El parámetro `name` define el nombre interno del atributo, `title` define como se muestra en la tabla y `sortField` define si es ordenable (es redundante con name pero es requerido por Vuetable). Además, disponemos de tres tipos de input, los cuales serán descritos a continuación. El input por defecto que es un *input* de HTML con un tipo asignado, contamos con un ejemplo de como se configura en las líneas 5-10 del snippet 4.15. El input con selector traduce valores de el GTFS a texto para facilitar la selección. En las líneas 26-34 tenemos un ejemplo de este tipo, el selector muestra el texto correspondiente a las llaves del mapa de opciones, mientras que los valores del mapa son los valores internos. Los nombres y valores de las opciones configuradas han sido obtenidas de la especificación de GTFS. Por último, tenemos el input con llave foránea, el cual se encuentra configurado en las líneas 13-23. En este tipo de input configuramos una consulta de AJAX para obtener los posibles valores para cargar los datos para el selector, además de configurar cuales son los valores internos y cuales usamos para mostrar la tabla. Esta parametrización de los campos nos permite configurar cada tabla editable teniendo que definir solo el JSON que la describe, simplificando la inclusión de nuevas tablas y modificación de tablas existentes (por ejemplo en caso que cambie la especificación).

Snippet 4.15: Fields de Route

```
1  let fields = [
2    // actions indica la posicion de los botones.
3    'actions',
4    // route_id es un campo requerido estandar
5    {
6      name: 'route_id',
7      title: 'Route ID',
8      sortField: 'route_id',
9      required: true,
10   },
11   // agency_id nos muestra como se configura un campo
12   // con selector usando la API para opciones
13   {
14     name: 'agency_id',
15     title: 'Agency',
16     sortField: 'agency_id',
17     foreignKey: true,
18     id_field: 'agency',
19     required: true,
20     ajax_params: {
21       url: agenciesAPI.agenciesAPI.getFullBaseURL(this.$route.params.projectid),
22     }
23   },
24   // selector con opciones pre-definidas
25   {
26     name: 'route_type',
27     title: 'Type',
28     type: "select-simple",
```

```
    required: true,
    options: {
      'Tram, Streetcar, Light rail': 0,
      'Subway, Metro': 1,
      'Rail': 2,
      'Bus': 3,
      ...
    },
  },
  ...
]
```

4.2.2. Tabla editable

Para crear una tabla editable se creó una tabla de Vuetable, cuyo HTML es observable en el snippet 4.16. Como las tablas de Vuetable no permiten editar los datos, se ha definido un componente `GeneralizedInput` que, utilizando el descriptor del campo (field), determina cómo mostrar y cómo guardar los datos correspondientes a esa columna. Estos inputs son insertados en el slot de Vue correspondiente a las casillas de la tabla en las líneas 12-16 del snippet ya mencionado. Para tener los datos ingresados por el usuario se utilizó `v-model` en los GeneralizedInput, un parámetro de Vue que permite atar el valor de un componente a una variable de JavaScript, actualizando el valor de la variable cuando el componente se actualiza y actualizando el componente cuando la variable cambia. Como algunas tablas tienen miles de entradas, se vio en la necesidad de incorporar paginamiento y búsqueda, tal como se puede apreciar en la figura 4.1. Esta vista básica de tabla editable entrega al usuario la interfaz que requiere para las operaciones de las historias de usuario **H1**, **H3** y **H4**, esto pues permite la edición, creación y eliminación de datos, además de implementar los botones de subida y bajada de CSV.

Snippet 4.16: HTML de una tabla editable

```
1  <vuetable ref="vuetable" :api-url="url" :multi-sort="true"
   ↪  :fields="getTitledFields(fields)"
2    data-path="results" pagination-path="pagination"
     ↪  @vuetable:pagination-data="onPaginationData"
3    :query-params="makeQueryParams" :row-class="getRowClass">
4    <div slot="actions" slot-scope="props" style="display: flex; flex-direction: row;">
5      <button class="btn icon" @click="beginDeleteRow(props.rowData)" alt="Delete entry.">
6        <span class="material-icons">delete</span>
7      </button>
8      <slot name="additional-actions" v-bind:rowData="props.rowData"
       ↪  v-bind:rowField="props.rowField"
9        v-bind:rowIndex="props.rowIndex"></slot>
10   </div>
11   <!- This is where the fields are converted into inputs to make the table editable ->
12   <GeneralizedInput :key="index" v-for="(field, index) in getProperFields(fields)"
13   :slot="getFieldName(field)" slot-scope="properties" :data="properties.rowData"
```

```
14    :field="properties.rowField"
      ↪  v-model="properties.rowData[getFieldID(properties.rowField)]"
15    v-on:input="changeHandler(properties)">
16    </GeneralizedInput>
17  </vuetable>
```

El componente EditableTable cuenta con 360 líneas de JavaScript que agregan funcionalidad a la tabla. Buena parte de esto corresponde a la funcionalidad de los distintos botones y Modals, además de los llamados a API y los eventos que ellos desencadenan en caso de éxito o fallo. En el snippet 4.17 se encuentran algunos de los principales métodos de este componente. En la línea 1 tenemos el método `setDefaultCreationValue`, el cual toma un campo y un objeto y establece el valor como el valor por defecto para ese tipo de campo. El método `saveChanges`, presente en la línea 21, itera sobre las filas de la tabla y realiza un poco de preprocesamiento para luego enviar un update a la API REST, mostrando la fila en color rojo si hay algún error. El método que comienza en la línea 52 es `deleteRow`, método que es llamado desde el Modal de borrado e intenta borrar la entrada en cuestión. Si es exitoso, refresca la tabla y cierra el Modal. Si falla, guarda el mensaje en el objeto que representa al Modal, lo que Vue automáticamente muestra en el Modal, tal como podemos apreciar en la figura 4.2.

Las líneas 63-96 del snippet 4.17 contienen el método `createEntry`, el cual manda una solicitud a la API para crear una nueva entrada en la tabla, mostrando el error si ha fallado y cerrando el Modal de creación si ha sido exitoso. Los métodos `onChangePage` y `onPaginationData` de las líneas 97 y 113 se encargan de manejar el paginamiento. El método `onPaginationData` se encarga de procesar la información de paginamiento, ignorándola si no ha cambiado la página, esto pues tenemos 2 selectores de paginamiento que podrían actualizarse mutuamente y `onChangePage` se asegura de que no hayan cambios sin guardar antes de cambiar la página. En la línea 124 tenemos el método `makeQueryParams`, el cual se encarga de crear un objeto de configuración para las consultas a la API, revisando los parámetros de ordenamiento, búsqueda y paginamiento.

Snippet 4.17: Algunos de los métodos de EditableTable

```
1  setDefaultCreationValue(field, data = this.createModal.data) {
2    if (field === "actions") {
3      return;
4    }
5    if (field.foreignKey) {
6      data[field.name] = null;
7    } else if (field.options) {
8      data[field.name] = Object.values(field.options)[0];
9    } else if (field.data_type) {
10     let def = "";
11     if (field.data_type === "checkbox") {
12       def = false;
13     } else if (field.data_type === 'color') {
14       def = "#000000"
```

```
    }
    data[field.name] = def;
  } else {
    data[this.getFieldName(field)] = "";
  }
},
async saveChanges() {
  let data = this.$refs.vuetable.tableData;

  data.filter(row => row.changed).map(
    (row) => {
      for (let i = 0; i < this.fields.length; i++) {
        let field = this.fields[i];
        let fieldName = this.getFieldName(field);

        if (row[fieldName] === '') {
          row[fieldName] = null;
        }
        if (field.data_type === 'color') {
          if (row[fieldName].charAt(0) === '#') {
            row[fieldName] = row[fieldName].slice(0, 1)
          }
        }
      }
      this.updateMethod(row).then(response => {
        row.changed = false;
        row.error = false;
        console.log(response);
      }).catch(error => {
        let response = error.response;
        row.error = true;
        console.log(response.data);
      });
    }
  );
},
deleteRow() {
  let data = this.deleteModal.data;
  this.deleteMethod(data).then(() => {
    this.reloadTable();
    this.deleteModal.visible = false;
    this.deleteModal.data = {};
    this.deleteModal.message = "";
  }).catch((err) => {
    let data = err.response.data;
    this.deleteModal.message = data.message;
  });
```

```
62  },
63  createEntry() {
64    let data = {
65      ...this.createModal.data
66    }
67    this.fields.forEach(field => {
68      if (data[field.name] === "") {
69        data[field.name] = null;
70      }
71      if (field.data_type === 'color') {
72        if (data[field.name].charAt(0) === '#') {
73          data[field.name] = data[field.name].slice(1, 7)
74        }
75      }
76    });
77    console.log(data);
78    this.createMethod(data).then(response => {
79      console.log(response);
80      this.createModal.visible = false;
81      this.reloadTable();
82      this.createModal.errors = {};
83      this.fields.filter(field => field !== "actions").forEach(field => {
84        if (field instanceof Object) {
85          if (!field.remember_creation_value) {
86            this.setDefaultCreationValue(field);
87          }
88        } else {
89          this.setDefaultCreationValue(field);
90        }
91      });
92    }).catch(error => {
93      console.log(error.response.data);
94      this.createModal.errors = error.response.data;
95    });
96  },
97  onChangePage(page) {
98    if (page == this.current_page) {
99      return;
100   }
101   if (this.hasUnsavedChanges()) {
102     let answer = window.confirm("There are unsaved changes, are you sure you want to
        ↪  proceed?");
103     if (!answer) {
104       return;
105     }
106   }
107   this.$refs.vuetable.changePage(page);
```

```
108     this.$nextTick(() => this.$refs.vuetable.$data.tableData.forEach(row => {
109       row.changed = false;
110       row.error = false;
111     }));
112   },
113   onPaginationData(paginationData) {
114     if (paginationData.current_page !== this.current_page || paginationData.last_page !==
      ↪  this.last_page) {
115       this.current_page = paginationData.current_page;
116       this.last_page = paginationData.last_page;
117       this.$refs.pagination.setPaginationData(paginationData);
118       this.$refs.paginationDropDown.setPaginationData(paginationData);
119       this.$nextTick(() => {
120         $(".vuetable-pagination-dropdown").val(this.current_page).trigger('change');
121       });
122     }
123   },
124   makeQueryParams(sortOrder, currentPage, perPage) {
125     let sorting = ""
126     if (sortOrder.length > 0) {
127       sorting = sortOrder[0].sortField + "|" + sortOrder[0].direction;
128     }
129     return {
130       sort: sorting,
131       page: currentPage,
132       per_page: perPage,
133       search: this.quickSearch,
134     }
135   },
```

4.2.3. Inputs

Para poder convertir las celdas de la tabla a componentes editables se diseñaron 3 componentes de input y un componente GeneralizedInput que selecciona el correcto para el campo. SimpleInput corresponde a un input estándar de HTML, entregando el tipo definido en caso que corresponda como parámetro y reemplazando el input por un checkbox en caso que corresponda. SimpleSelect es un select con un conjunto de opciones predefinido. Para entender el formato en el que vienen las opciones, referirse a la línea 30 del snippet 4.15. En los selectores se utilizó la librería `Select2`[16], la cual permite generar un selector con dropdown buscable y permite crear selectores con opciones provenientes de una API.

La implementación del componente `GeneralizedInput` se encuentra en el snippet 4.18. En las líneas 3-10 se realiza la selección del tipo de input según la configuración del campo que representa, y con el método `onInput` (línea 42) se toman los cambios de valor del input creado por el componente y son comunicados al componente padre.

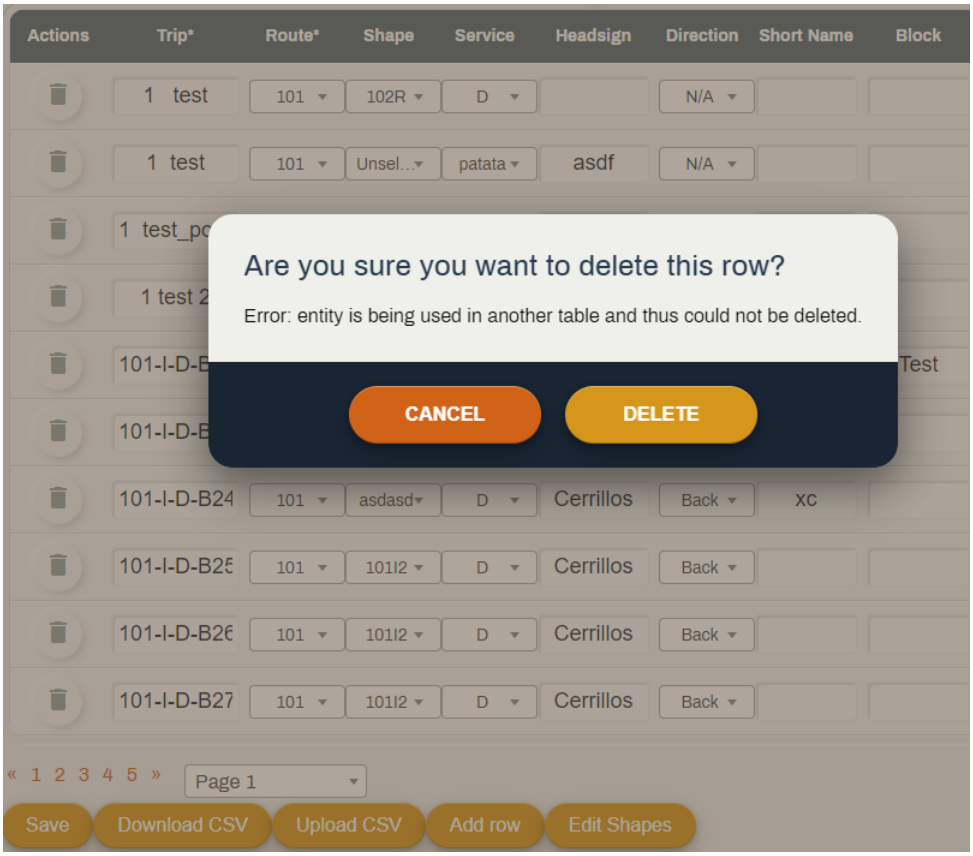


Figura 4.2: Modal de borrado de filas

Snippet 4.18: Componente de input genérico

```
<template>
  <!- Foreign key ->
  <FKSelect v-if="field.foreignKey" :field="field" :data="data" v-model="val"
  ↪  @input="onInput" :hasErrors="has_errors()">
  </FKSelect>
  <!- Options ->
  <SimpleSelect v-else-if="field.options" :field="field" v-model="val" @input="onInput"
  ↪  :hasErrors="has_errors()">
  </SimpleSelect>
  <!- Default ->
  <SimpleInput v-else :field="field" v-model="val" @input="onInput"
  ↪  :hasErrors="has_errors()">
  </SimpleInput>
</template>

<script>
  import ...
  export default {
    ...
```

```
17      props: {
18        field: {
19          type: Object,
20          required: true,
21        },
22        data: {
23          type: Object,
24          required: true,
25        },
26        errors: {
27          type: Array,
28          default: ()=>[],
29        },
30      },
31      data() {
32        let name = this.getFieldName(this.field);
33        return {
34          name,
35          val: this.data[this.getFieldID(this.field)],
36        }
37      },
38      methods: {
39        has_errors() {
40          return (this.errors instanceof Array)? this.errors.length>0:false;
41        },
42        onInput(event){
43          this.$emit('input', event);
44        }
45      },
46    }
47  </script>
```

El caso del input simple es bastante sencillo, por lo que no entraremos en detalles. Simplemente se crea input básico de HTML y se le asigna el tipo que viene configurado, preocupándose de actualizar el valor cuando cambia la variable interna y de lanzar un evento de input cuando el usuario cambia el valor del input.

Para el selector con opciones se creó el componente SimpleSelect, cuya implementación podemos ver en el snippet 4.19. En la línea 3 se procesan las opciones y se genera el HTML correspondiente. En las líneas 12-17 tenemos un watcher para el atributo value, el cual crea un nuevo selector con el nuevo valor seleccionado por defecto. Fue necesario destruir y regenerar el selector mejorado, pues de lo contrario se volvía mucho más complejo el manejo de eventos. El principal problema era que al actualizar el valor eso gatillaba el evento de input, lo que hacía que la fila se mostrara con cambios cuando no correspondía, y como Vue altera el HTML en base a las variables, resultaba complejo desactivar los eventos y esperar a que se actualizara el HTML para volver a activarlos. Cuando el usuario selecciona una nueva

opción se lanza el método onChange (línea 31), el cual revisa que el lanzamiento de un evento con cambio se encuentre habilitado y emite un evento de input para que el v-model padre se actualice. El único caso en que estaría deshabilitado es que el input se acaba de crear y se está estableciendo la opción por defecto, deshabilitándolo en el observador de value en la línea 13, cambiando el valor en la línea 15 y volviendo a habilitarlo en el método datafy, en la línea 40, cuando se ha creado ya el selector de `Select2`.

Snippet 4.19: Componente de selector simple

```
1  <template>
2    <select class="select-simple" ref="select" v-model="val">
3      <option v-for="(option, key) in field.options" :key="key"
       ↪  :value="option">{{key}}</option>
4    </select>
5  </template>
6
7  <script>
8  ...
9  export default {
10 ...
11 watch: {
12   value() {
13     this.changeEnabled = false;
14     $(this.$refs.select).\texttt{Select2}('destroy');
15     this.val = (this.value === "")? null:this.value;
16     this.$nextTick(this.datafy);
17   }
18 },
19 data() {
20   return {
21     changeEnabled: true,
22     val: this.value,
23     name: this.getFieldName(this.field),
24   }
25 },
26 mounted() {
27   this.datafy();
28   $(this.$refs.select).on('change', this.onChange);
29 },
30 methods: {
31   onChange(evt) {
32     this.val = evt.target.value;
33     if (this.changeEnabled) {
34       this.$emit("input", this.val);
35     }
36   },
37   datafy() {
```

```
38      let select = this.$refs.select;
39      $(select).\texttt{Select2}();
40      this.changeEnabled = true;
41    },
42  },
43  }
44  </script>
```

4.3. Vistas de mapa interactivo

Como hemos estudiado en la sección 3.2.2, algunas de las tablas hacen referencias a datos geográficos y, por esto mismo, se ven beneficiadas por el acceso a un mapa interactivo que asiste al usuario en las tareas de creación y edición de datos, esto en concordancia con el tercer objetivo específico. Las vistas en cuestión son la vista de stops, la vista de shapes y la vista de StopTimes.

4.3.1. Stops

La vista de Stops busca entregar al usuario las herramientas necesarias para las acciones de la historia de usuario **H5**. En la figura 4.3 podemos ver un Stop en edición. Para editar un Stop el usuario debe arrastrarlo o hacerle click para abrir un pop-up con la información del paradero en formato de formulario. Para guardar los cambios, el mapa manda una solicitud de PATCH a la API del modelo, entregando errores si no ha sido posible guardar los cambios. Como los paraderos se utilizan en conjunto con los Shapes en StopTimes, se agregó un selector de Shapes que permite al usuario ver un Shape en conjunto con los paraderos, tal como podemos apreciar en la figura 4.4. El cómo se muestra un Shape se encuentra descrito en la sección 4.3.2.

Para evitar sobrecargar la API con consultas de paraderos, la aplicación carga de inmediato todos los paraderos y los agrega al mapa de Mapbox. Al ser un número relativamente grande de paraderos (11.437 en el caso de Santiago), Mapbox se volvía muy lento al agregarlos como marcadores directamente. Por esto se tuvo que utilizar el formato GeoJSON[17] para poder mostrarlos en el mapa, tal como podemos apreciar en el snippet 4.20. El estándar GeoJSON es simplemente un JSON que sigue un formato que permite describir información geográfica. El método `generateStopGeoJson` toma un paradero en el formato de la API como argumento y genera el GeoJSON correspondiente a este paradero. En la línea 21 se le agrega un `source` al mapa, vale decir, un GeoJSON que describe un elemento o conjunto de elementos que que se desea representar en el mapa. Una vez configurado nuestro `source`, se le agregan `layers`, los cuales corresponden a las distintas formas en que se visualizan los datos. En el caso particular de los paraderos se les agrega un layer correspondiente al punto azul y otro correspondiente al texto. Estos `layers` pueden ser observados en la figura 4.3. El primer `layer`, agregado en la línea 29 del snippet, agrega el círculo. Con el parámetro stops de circle-radius se definen los distintos tamaños del círculo a distintos niveles de zoom. El otro `layer` corresponde al texto, el cual utiliza el atributo label del GeoJSON y se muestra debajo del centro del objeto. Se utiliza el parámetro `minzoom` para que al estar lejos no se llene la vista de texto con los nombres de todos los paraderos.

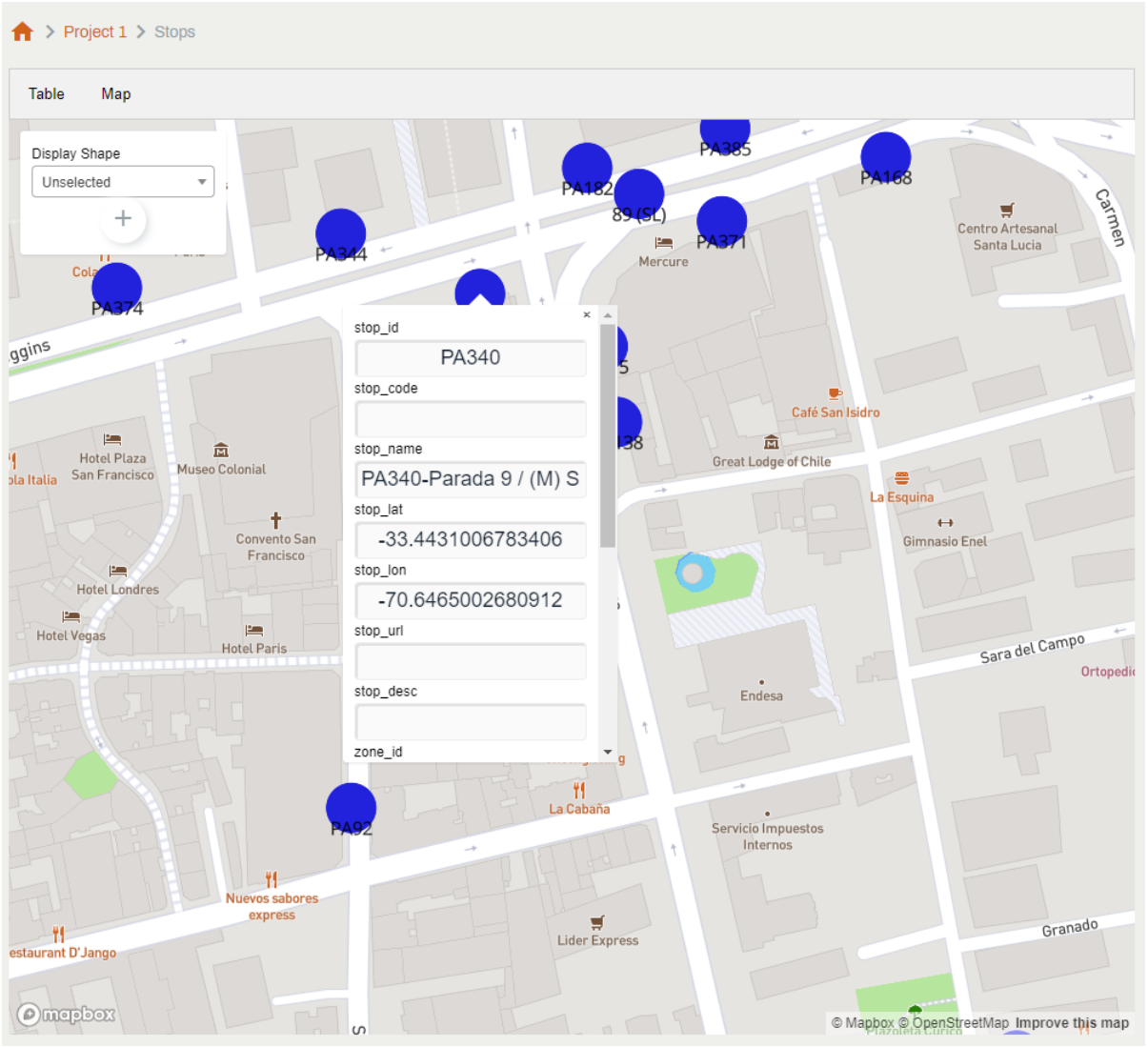


Figura 4.3: Vista de edición de paraderos con mapa

Snippet 4.20: Adición de los paraderos al mapa

```
generateStopGeoJson(stop) {
    return {
      type: 'Feature',
      geometry: {
        type: 'Point',
        coordinates: [
          stop.stop_lon,
          stop.stop_lat,
        ]
      },
      properties: {
        stop_id: stop.id,
        label: this.createLabel(stop),
```

```
      }
    }
}
...
// We create the geojson and add it to the map
this.geojson = {
  type: 'FeatureCollection',
  features: this.stops.map(this.generateStopGeoJson),
};
this.map.addSource('stops', {
  type: 'geojson',
  data: this.geojson,
});

// We add an icon and text to the geojson
this.map.addLayer({
  id: "layer-stops-icon",
  type: "circle",
  source: "stops",
  paint: {
    "circle-radius": {
      base: 2,
      stops: [
        [12, 1.5],
        [14, 4],
        [20, 180]
      ]
    },
    "circle-color": "#2222DD"
  }
});
this.map.addLayer({
  id: "layer-stops-label",
  type: "symbol",
  source: "stops",
  minzoom: 16,
  layout: {
    "text-field": "{label}",
    "text-anchor": "top",
    "text-offset": [0, 0.5],
    "text-allow-overlap": true,
  }
});
```

Para poder agregar interactividad al mapa se han empleado listeners de Mapbox. En los listeners se puede especificar no solo el tipo de evento, sino que también el layer que lo

escucha. Esto resulta útil, por ejemplo, a la hora de distinguir si el usuario hace click en el mapa y arrastra para mover un paradero o para mover el mapa. En el snippet 4.21 podemos ver un ejemplo de esto: a grandes rasgos, si el usuario presiona click con el cursor sobre un paradero lo cambiamos a una mano para indicar que está arrastrando el paradero y cuando levanta revisamos que lo haya movido, si lo ha movido entonces actualizamos la posición del paradero.

Snippet 4.21: Eventos de arrastrar paradero en el mapa

```
map.on('mousedown', 'layer-stops-icon', function (evt_down) {
  // Prevent the default map drag behavior.
  evt_down.preventDefault();
  canvas.style.cursor = 'grab';
  let activeStop = evt_down.features[0]
  console.log(evt_down.features);

  map.once('mouseup', evt_up => {
    let coords = evt_up.lngLat;
    let distance = self.calcDistance(evt_down, evt_up);
    if (!distance) {
      return;
    }
    self.updateStop(activeStop, coords);
    canvas.style.cursor = '';
  });
});
```

Para la creación de un paradero se hace algo similar que para la edición. El usuario puede agregar un punto al mapa, el cual puede arrastrar para ajustar la posición del paradero que desea crear. Al agregar este punto se abre un formulario de creación de paradero, tal como se ve reflejado en la figura 4.4. Al llenar los datos necesarios para la creación del paradero, el usuario aprieta un botón y se envía una solicitud de creación a la API. En caso de fallo los errores se formatean y muestran en el formulario de creación.

En la vista de paraderos el usuario puede cambiar a una vista de tabla, lo que puede resultar útil en caso que necesite editar varios datos no geográficos. Esta vista es simplemente una tabla editable con una acción adicional que abre el mapa y lo centra y se acerca al paradero correspondiente a la fila cuyo botón ha sido presionado.

4.3.2. Shapes

El editor de Shapes cuenta con 2 vistas principales. La primera es una vista que permite al usuario visualizar Shapes y seleccionar cuál desea editar y la segunda es la vista de edición de Shapes. Este editor busca resolver la necesidad presentada en la historia de usuario **H6**.

La vista general de Shapes corresponde a la figura 4.5. Es una vista que tiene botones para visualizar, editar y borrar cada Shape, además de uno que permite crear un nuevo Shape.

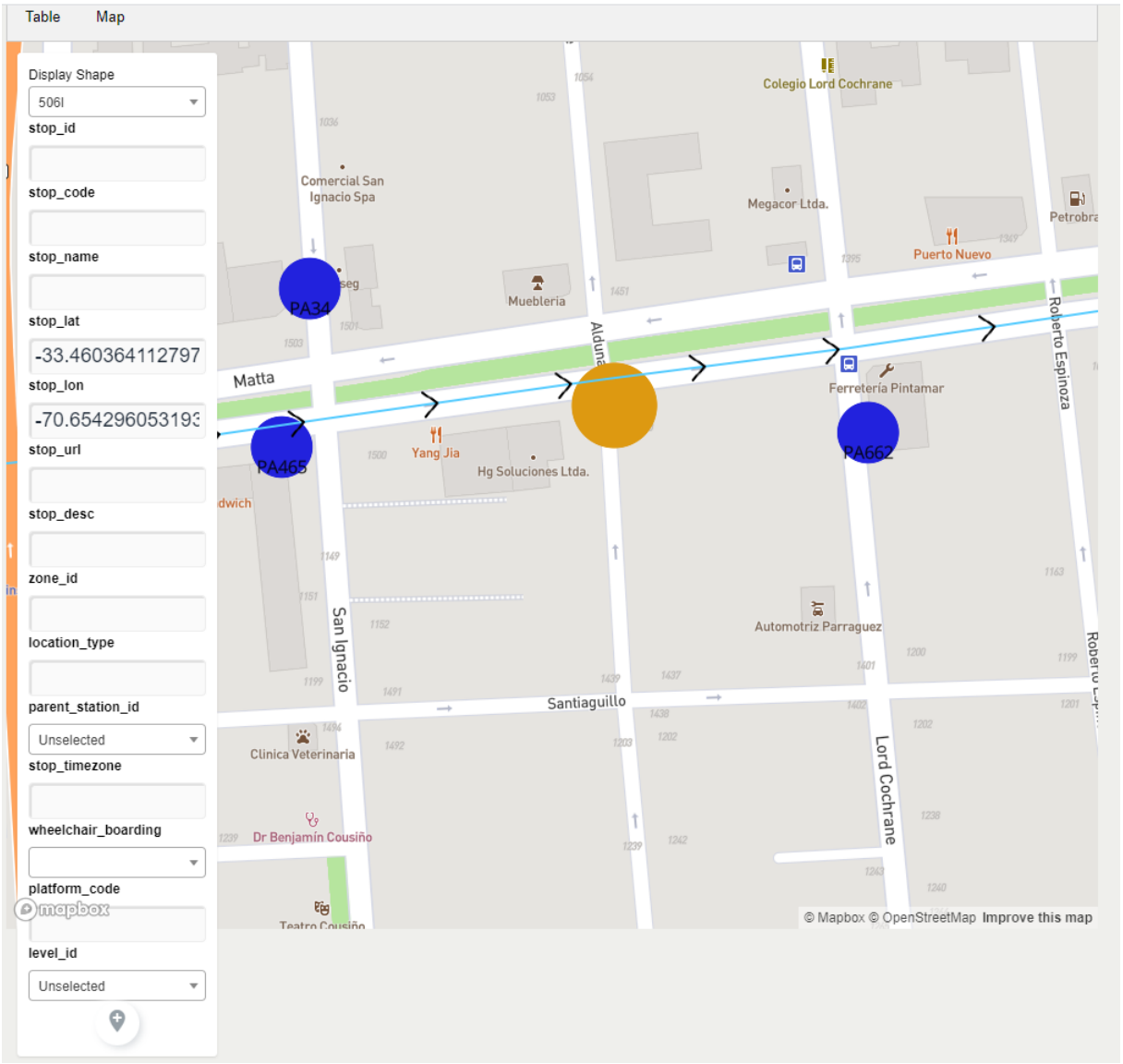


Figura 4.4: Creación de paradero con mapa

Para mostrar el shape en el mapa se agregó un source que representa a los puntos con un layer para mostrarlos, siguiendo un proceso similar al descrito en la sección 4.3.2. La línea y las flechas indicando la dirección del Shape requieren un source con un formato distinto. En el snippet 4.22 creo un JSON que describe la línea usando un arreglo de coordenadas y luego lo agrego al mapa. Una vez añadido el source se agregó un layer que genera la línea y otro que ubica las flechas a lo largo del Shape. Al cambiar el shape se debe actualizar los GeoJSON y con ellos actualizar los sources, pero los layers se mantienen. Esto significa que tenemos que configurar una única vez como se muestran los datos, pues Mapbox dibuja los layers con los nuevos datos al actualizar el source. Un detalle importante del layer con imagen es que para el espaciamiento considera el tamaño de la imagen antes de escalarla. Por ejemplo, cuando se tenía una imagen mucho más grande, con una escala de 0,065, no lograba que se mostraran juntas. Una vez confirmada la sospecha de que el problema era el tamaño de la imagen, se probó con una imagen más pequeña, con una escala de 1, y así se pudo configurar correctamente el espaciado.

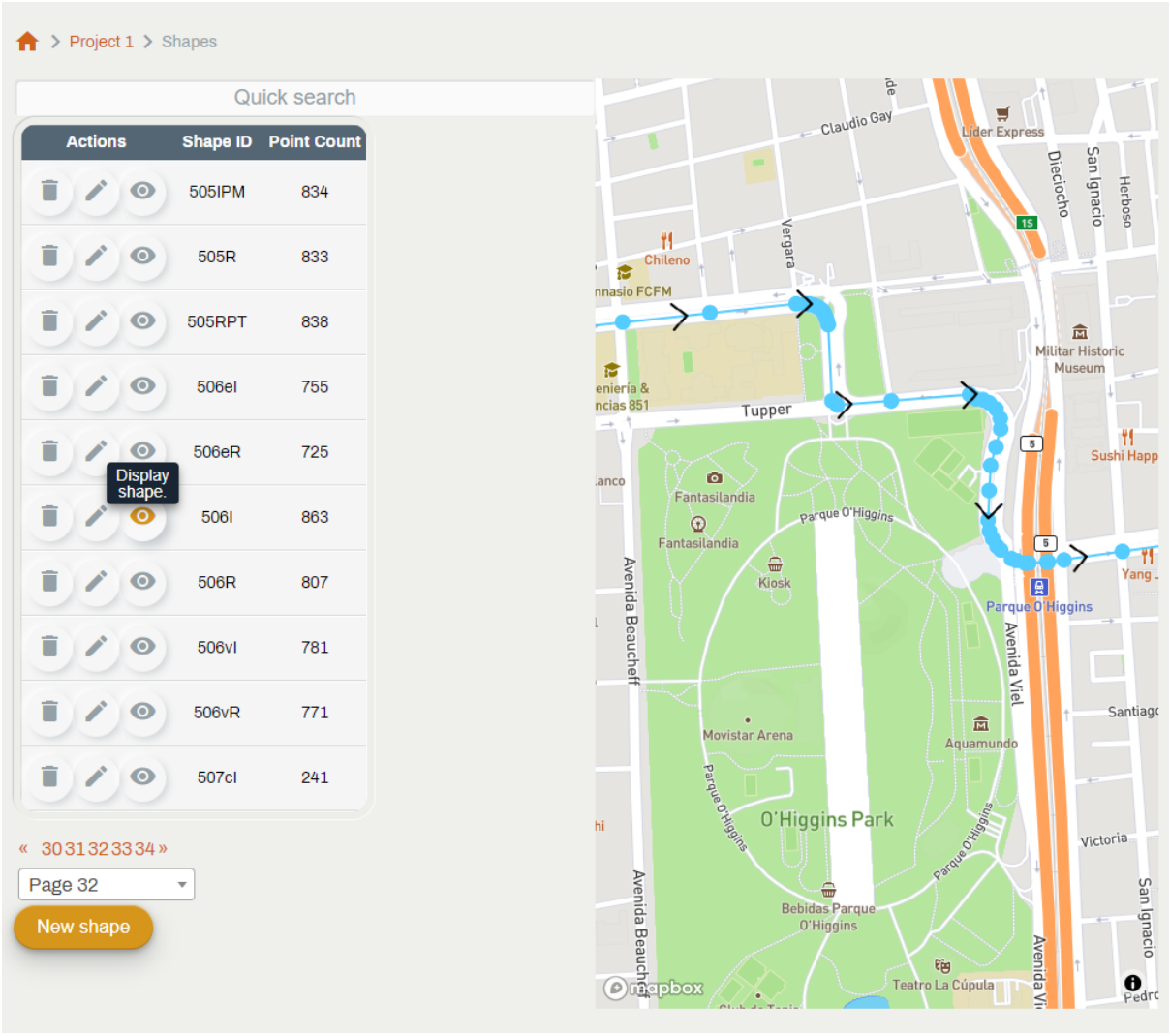


Figura 4.5: Vista general de Shapes

Snippet 4.22: Layers de línea de Shape

```
lineGeojson: {
    'type': 'Feature',
    'properties': {},
    'geometry': {
        'type': 'LineString',
        'coordinates': []
    }
}
...
this.map.addSource('line', {
    'type': 'geojson',
    'data': this.lineGeojson,
});
...
```

```
this.map.addLayer({
    'id': 'line-layer',
    'type': 'line',
    'source': 'line',
    'layout': {
        'line-join': 'round',
        'line-cap': 'round'
    },
    'paint': {
        'line-color': '#3333CC',
        'line-width': 2
    }
});
...
this.map.addImage('arrow', image);
this.map.addLayer({
    'id': 'point-arrow',
    'type': 'symbol',
    'source': 'points-seq',
    'layout': {
        'symbol-placement': 'line',
        'symbol-spacing': 100,
        'icon-allow-overlap': true,
        'icon-ignore-placement': true,
        'icon-image': 'arrow',
        'icon-size': 1,
        'visibility': 'visible'
    }
});
```

Al hacer click en el botón de edición en la vista general de Shapes se abre el Modal de la figura 4.6, el cual permite al usuario seleccionar un modo de edición. El modo de `Replace entire Shape` permite reemplazar el shape por uno nuevo, manteniendo las referencias a este. El modo de `Edit Shape directly` abre el editor con el Shape completo en caso que el usuario quiera realizar cambios menores. El selector de rango (tercer botón) exige al usuario seleccionar 2 puntos y le permite editar los puntos entre los que haya seleccionado, incluyéndolos. Por último, `Duplicate shape` abre el editor de shapes en modo de nuevo shape pero con los puntos ya cargados, esto es útil si el usuario desea crear una variante de un shape (como por ejemplo el recorrido de vuelta o una versión corta/extendida del shape).

El editor de Shapes es un mapa interactivo como el visto en la figura 4.7, con varias acciones disponibles. Haciendo doble click en el mapa, el usuario agrega un punto a continuación en el shape. Los puntos son arrastrables, permitiendo al usuario ajustar las posiciones de los shapes. Hacer click en la línea del Shape crea un punto en esa posición, dividiendo la línea en dos y permitiendo ajustar las partes intermedias del shape. El editor de Shapes cuenta además con varias acciones en la forma de botones en el cuadro en la parte superior izquierda

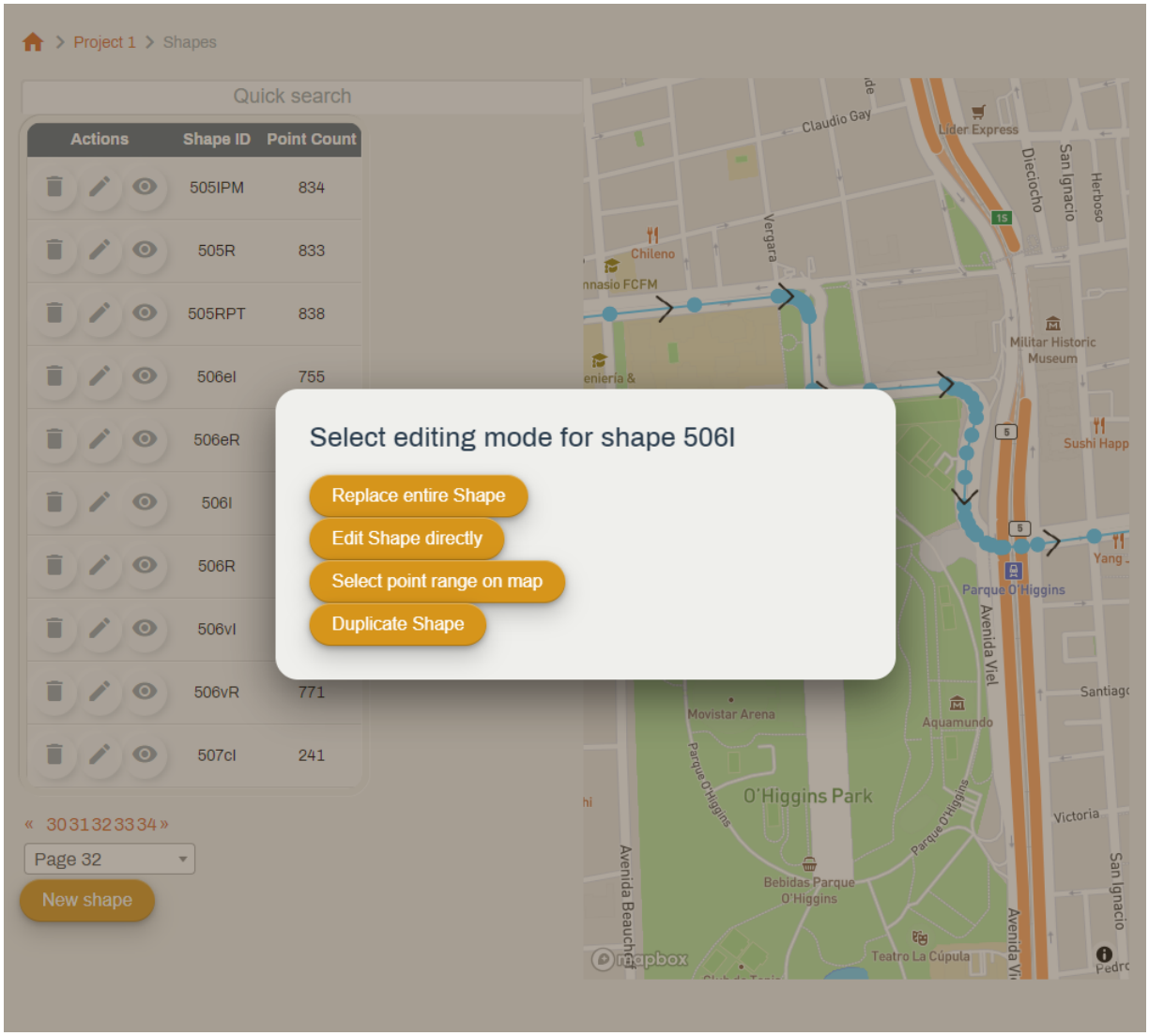


Figura 4.6: Modos de edición

del mapa. El botón de *Invert Shape* toma todos los puntos que estamos editando e invierte su orden, esto es útil cuando el usuario desea crear el shape describiendo el viaje de vuelta de un Shape pre-existente. El botón de *Save and exit* guarda los cambios, creando o actualizando el Shape dependiendo del modo de edición. Al apretar este botón se muestra un error si hay algún problema y volviendo a la vista general de Shapes si ha tenido éxito. El botón de *Exit* permite al usuario salir sin guardar los cambios, requiriendo confirmación del usuario.

Como los datos de un GTFS comunmente representan rutas en una calle, se incorporó la herramienta de Map Matching de Mapbox, la cual traduce puntos geográficos a un recorrido utilizando las calles. Para utilizar esta herramienta le envío un request a la API de la herramienta con un arreglo de puntos y algunos parámetros y me devuelve un Shape que describe una ruta que pasa por todos esos puntos en orden pasando por las calles de Mapbox en las direcciones correctas. En la figura 4.7 hay un ejemplo de esto. El usuario está editando un rango de puntos del Shape 506I, utilizado por los viajes de ida del recorrido 506. Tenemos 4 puntos verdes que corresponden a los puntos ingresados por el usuario para generar el Shape,

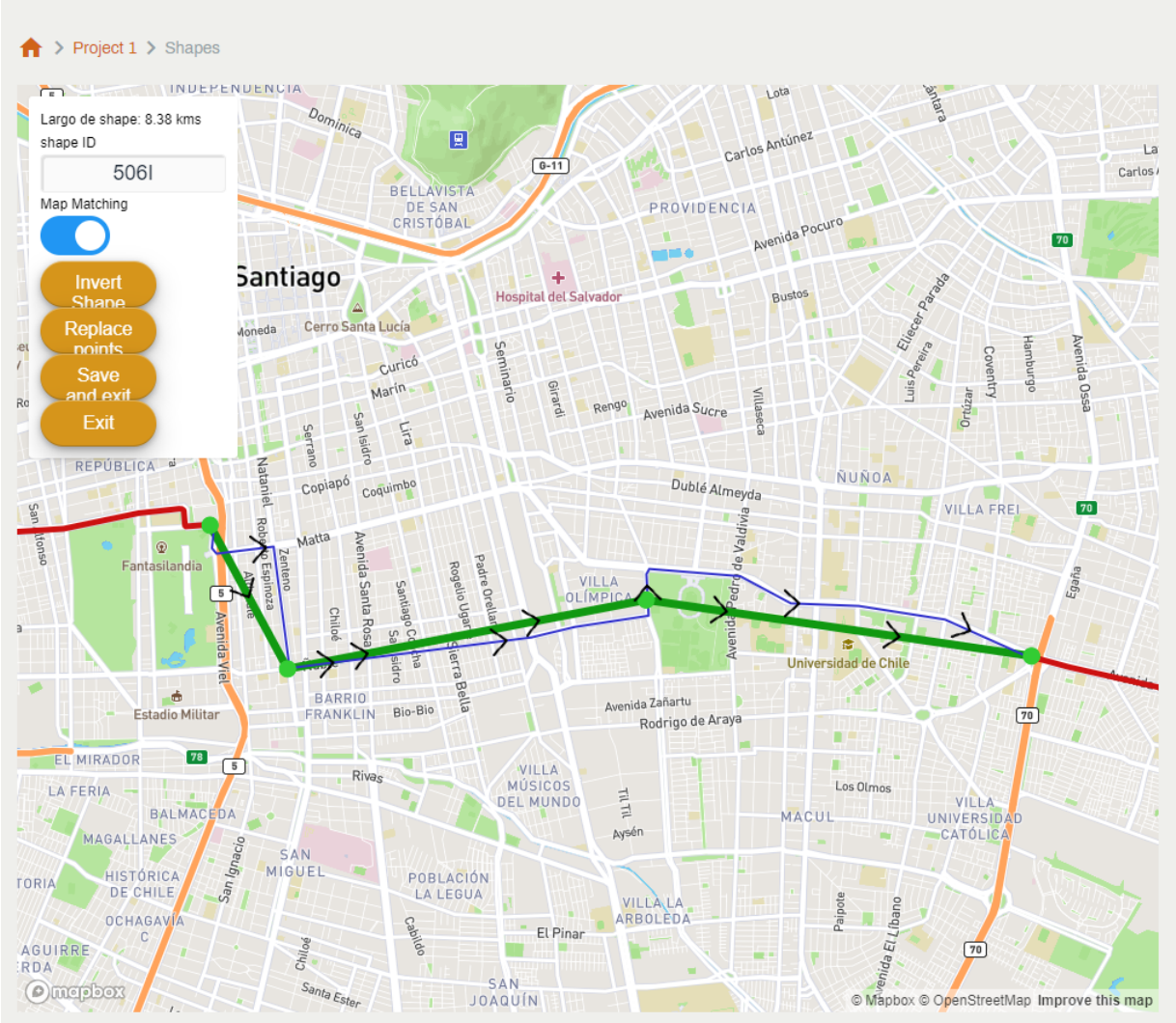


Figura 4.7: Mapmatching para Shape

una línea verde con flechas para indicar el orden de los puntos y una línea azul que muestra el resultado del Map Matching. Si el usuario presiona el botón de *Replace Points*, el editor desactiva el Map Matching y reemplaza los puntos verdes por los entregados por la API. La figura 4.8 contiene el resultado de apretar este botón en la vista de la figura 4.7.

4.3.3. StopTimes

Los StopTimes describen la secuencia de paraderos por la que pasa un viaje (Trip). Para que el usuario pueda realizar las acciones de la historia de usuario **H7** se ha diseñado una vista especial para StopTimes. Al abrir esta vista el usuario llega a la vista de la figura 4.9, donde puede visualizar los Shapes y paraderos asociados a los viajes. Desde esta vista el usuario puede abrir un selector de modos de edición con 3 opciones, el cual corresponde al modal de la figura 4.10. Los botones de *Edit StopTimes* y *Duplicate StopTimes* abren la vista de edición de StopTimes, con la diferencia de que en el caso de editar un StopTimes al guardar los cambios se actualizan, mientras que al duplicar se crea uno nuevo con un nuevo Trip asociado. El botón de *Duplicate StopTimes with headway* permite al usuario ingresar

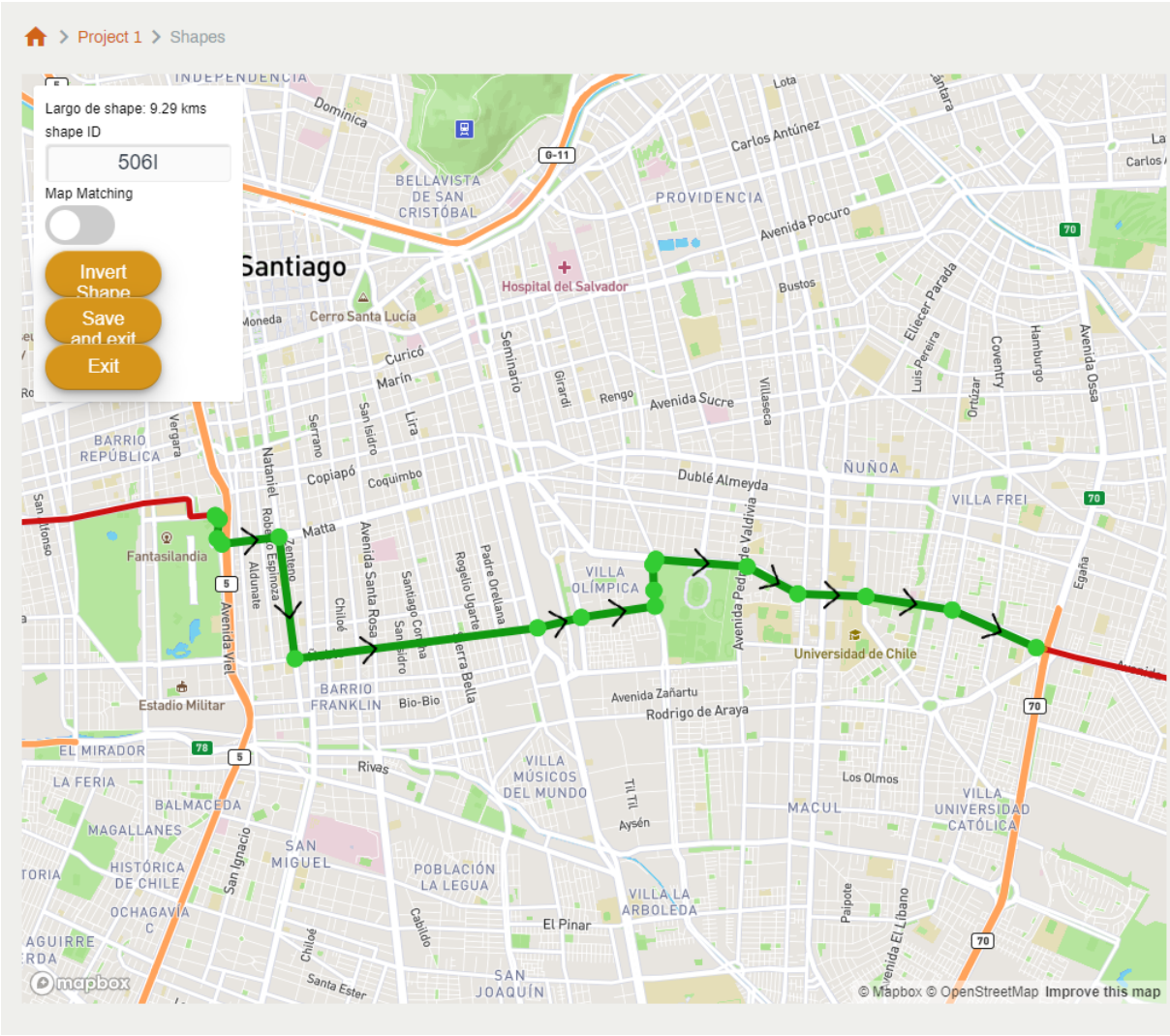


Figura 4.8: Shape tras Mapmatching

una diferencia de tiempo en segundos y un nuevo *Trip ID* y crea un nuevo Trip con el nuevo ID y la misma secuencia de paraderos pero con los respectivos `Arrival` y `Departure Time` desplazados en la cantidad de segundos ingresada. Por ejemplo, al ingresar 1.800 segundos, creamos un nuevo StopTimes con media hora de desfase. Cabe destacar que como no se puede crear un StopTimes sin un Trip asociado, no existe un botón que permita crear un StopTimes directamente.

El editor de StopTimes consiste en un mapa interactivo con una tabla, esta vista corresponde a la figura 4.11. Como los StopTimes describen la secuencia de paraderos por los que pasa un viaje, aproveché de utilizar el Shape asociado al viaje para agregar funcionalidades adicionales al editor. La función `nearestPointOnLine` de la librería `Turf.js` toma como argumentos un punto y una polilínea y realiza una proyección del punto a la polilínea, entregando como resultado un el punto de la polilínea más cercano al punto que le haya entregado como parámetro y la distancia en la polilínea, vale decir, la distancia que hay que recorrer para llegar a ese punto desde el origen de la polilínea siguiendo el trazado de esta.

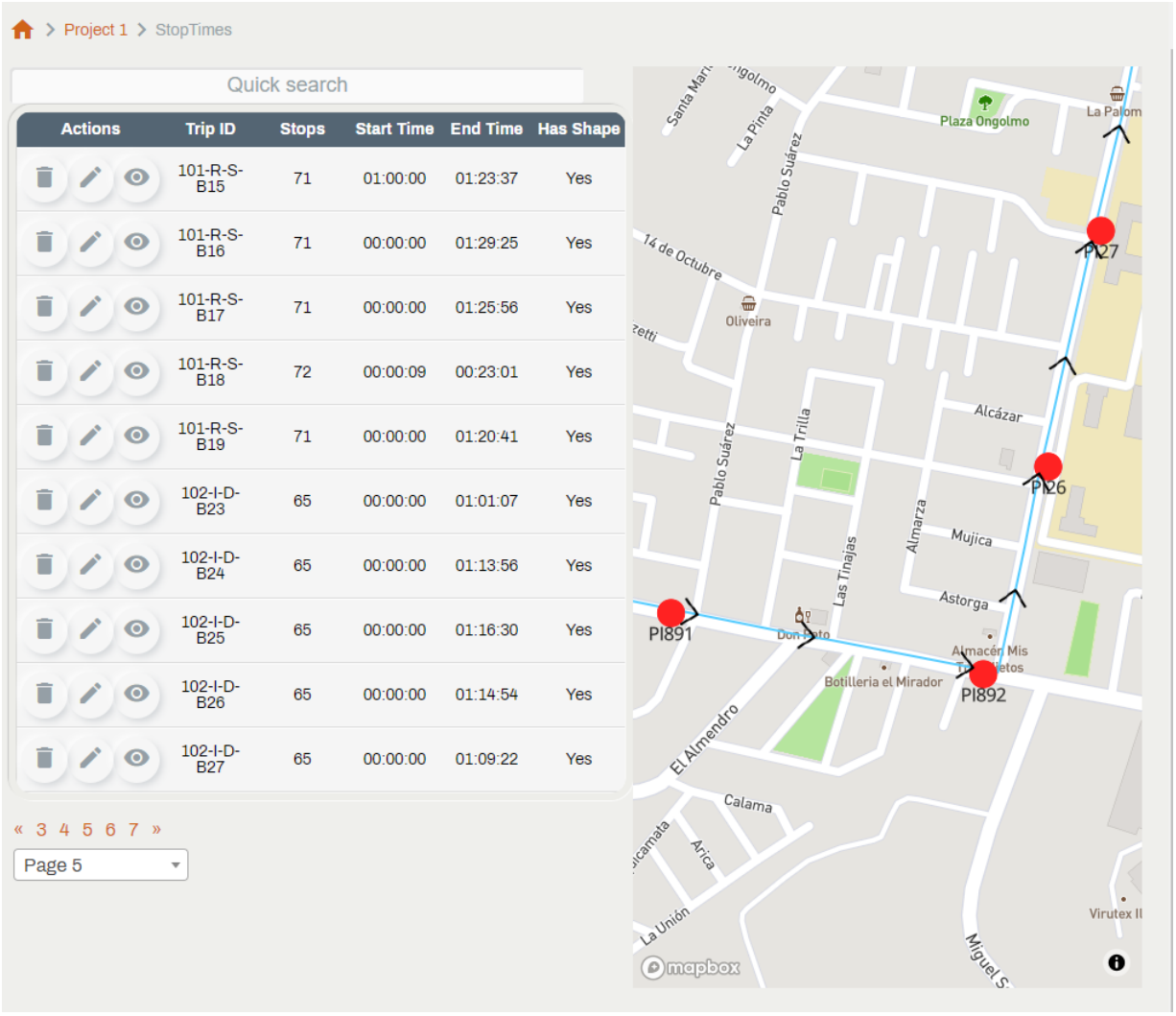


Figura 4.9: Vista general de StopTimes

En el snippet 4.23 tengo el método calculatePosition, el cual toma uno de los StopTimes y entrega cuál es la posición de este en el Shape. Con esta posición se puede fácilmente ordenar los puntos en un Shape, utilizando la posición como el valor a comparar. Desde el estado de la figura 4.11 se puede apretar el botón *Automatically order using shape*, esto abre un diálogo de confirmación y tras haber confirmado reordena los paraderos, llevando al usuario al estado de la figura 4.12.

Adicionalmente, se puede apretar el botón de *Automatically calculate times* para abrir el modal de la figura 4.13. De aquí se puede seleccionar un rango de paraderos y una velocidad promedio de viaje y, utilizando la posición en el shape y el `Arrival Time` del primero paradero, calcular los tiempos de subida y bajada en cada paradero, tal como podemos apreciar en la figura 4.14. Para esto se transformó el `Arrival Time` a segundos y calculo para cada paradero cuanto se demoraría el recorrido desde el paradero inicial al respectivo paradero en segundos, usando la distancia y la velocidad. Luego se transforma este tiempo en segundos a formato `[D hh:mm:ss]`.

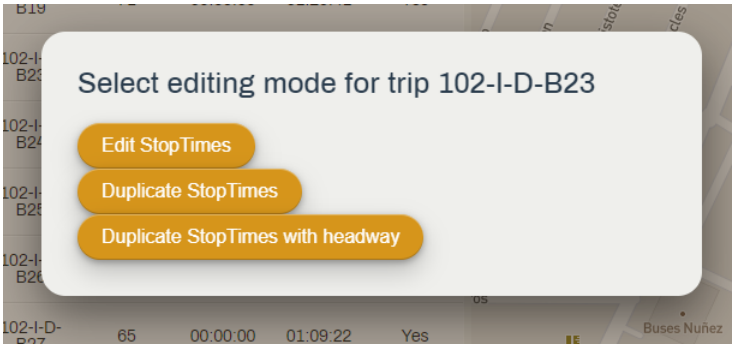


Figura 4.10: Modos de edición de StopTimes

Snippet 4.23: Método de cálculo de posición de un punto en el Shape

```
calculatePosition(st) {
    let stop = this.stop_map.get(st.stop);
    let point = turf.point([stop.stop_lon, stop.stop_lat]);
    let nearest = turf.nearestPointOnLine(this.turfShape, point);
    return nearest.properties.location;
}
```

Los StopTimes tienen bastantes parámetros opcionales, pero en los ejemplos con los que he trabajado solo se utilizan arrival y departure time. Como la tabla con todos los parámetros opcionales se ve como la de la figura 4.15, por esto se agreguó el toggle de *Show Optional Fields*, el cual agrega todas las columnas opcionales a la tabla.

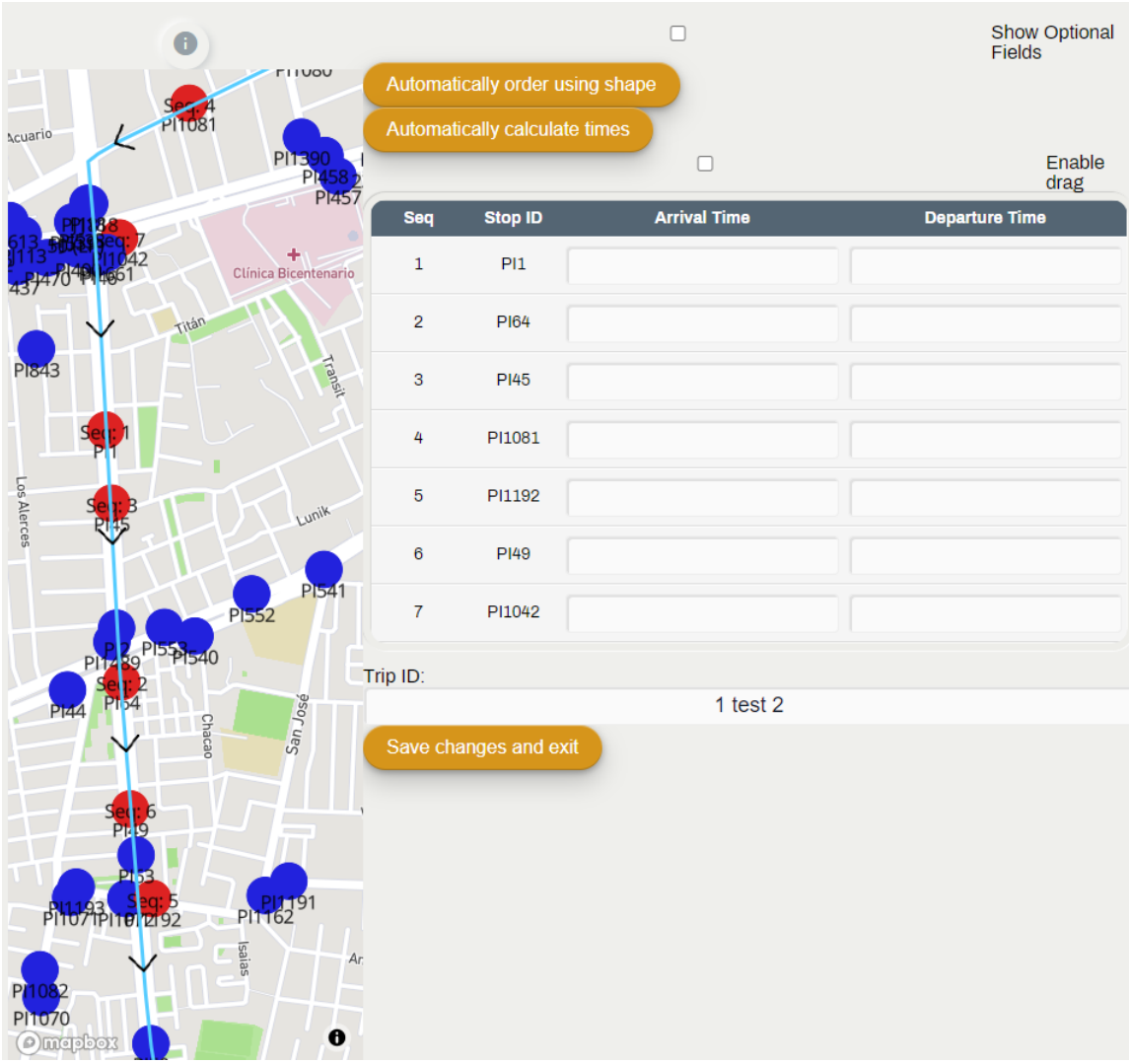


Figura 4.11: Editor de StopTimes

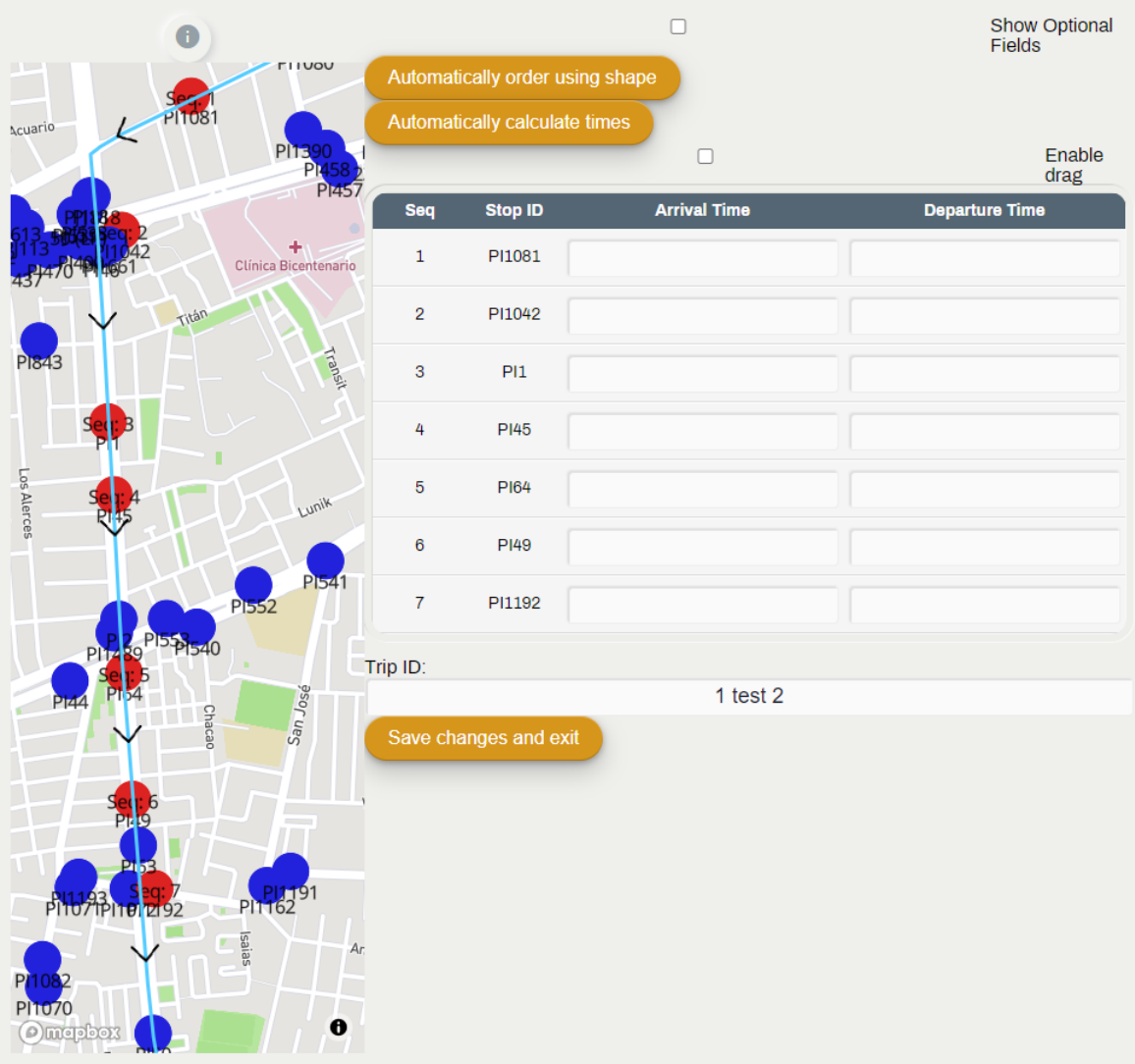


Figura 4.12: Editor de StopTimes tras haber ordenado los paraderos

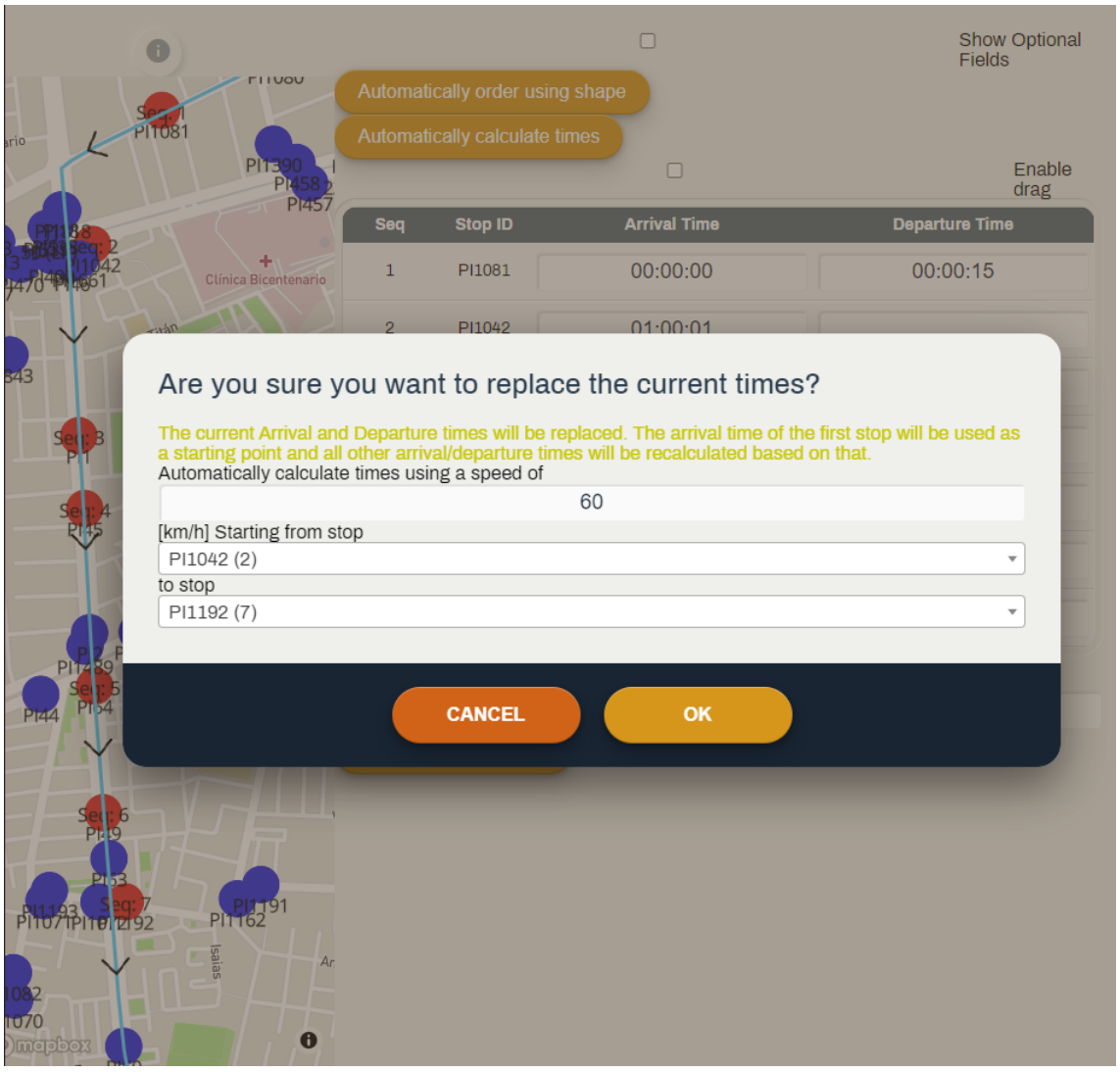


Figura 4.13: Modal de tiempos automáticos

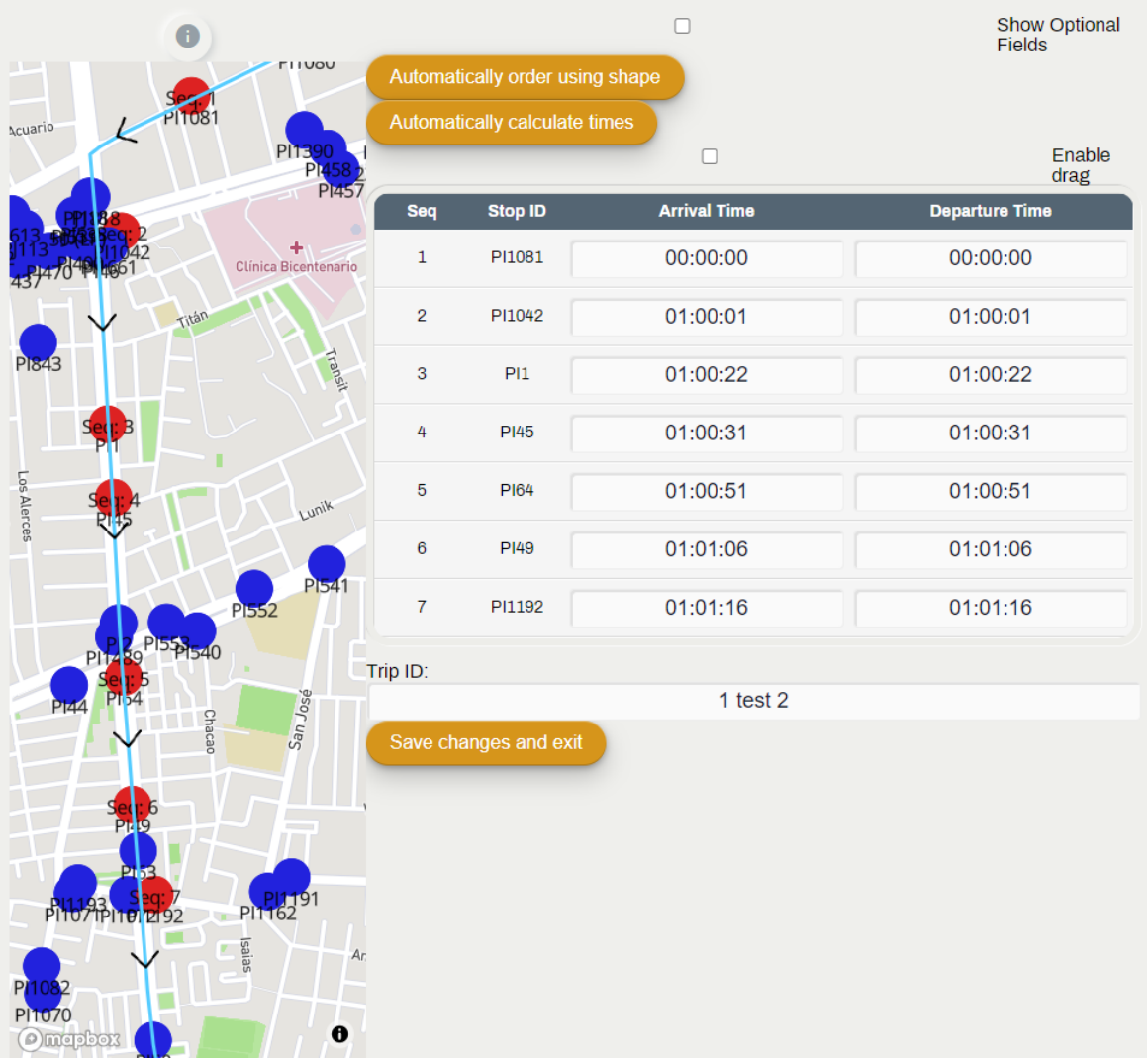


Figura 4.14: Editor de StopTimes usando tiempos automáticos

Seq	Stop ID	Arrival Time	Departure Time	Stop Headsign	Pickup Type	Drop-Off Type	Continuous Pickup	Continuous Drop-Off	Shape Distance Traveled	Timepoint
1	PI1081	00:00:0	00:00:0							
2	PI1042	01:00:0	01:00:0							
3	PI1	01:00:2	01:00:2							
4	PI45	01:00:3	01:00:3							
5	PI64	01:00:5	01:00:5							
6	PI49	01:01:0	01:01:0							
7	PI1192	01:01:1	01:01:1							

Figura 4.15: Tabla de StopTimes con campos opcionales habilitados

Capítulo 5

Validación

Como se empleó una metodología con reuniones diarias a lo largo del proyecto, se realizó validación de forma continua con los clientes. Además, los clientes han realizado pruebas de la aplicación y han quedado conformes con el producto entregado.

5.1. Estructura del proyecto

En la figura 5.1 se puede apreciar la estructura del proyecto que corresponde al backend de la aplicación. Los principales módulos son el módulo `gtfseditor`, que corresponde al módulo principal del proyecto y el módulo `rest_api`, el cual contiene todo el código asociado a la API REST.

La figura 5.2 muestra la estructura del proyecto correspondiente al frontend de la aplicación. La carpeta `api` contiene los métodos de JavaScript utilizados para llamar a la API del backend. La carpeta `components` contiene todos los componentes de Vue que he creado que no corresponden a vistas. La carpeta `mixins` incluye unos pocos mixins utilizados para no repetir funcionalidad entre vistas. `views` contiene a todas las vistas de la aplicación.

El backend de la aplicación cuenta con 235 tests, probando principalmente todas las operaciones de los endpoints. Estos tests logran una cobertura del 94 % de las líneas correspondientes al módulo de la API y 95 % del módulo principal (`gtfseditor`). Como los tests validan el formato de ingreso y salida de los datos, esto permite que al realizar algún cambio en el backend, los tests validen que el formato sea compatible con el frontend.

5.2. Validación de historias de usuario

A continuación veremos cómo la implementación logra resolver las necesidades del usuario en las historias definidas en la sección 3.1.

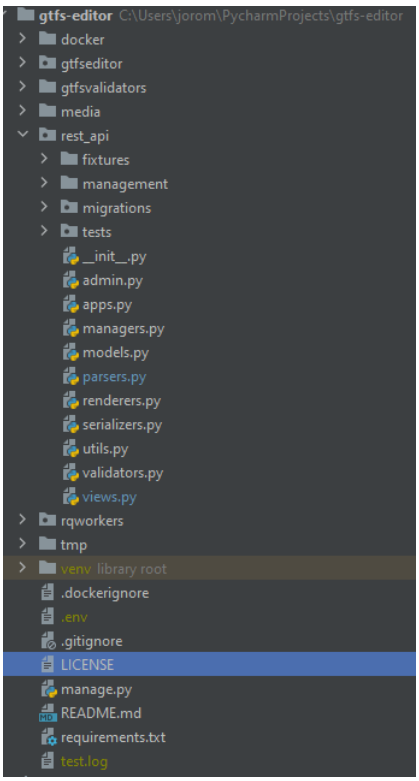


Figura 5.1: Estructura de proyecto del backend

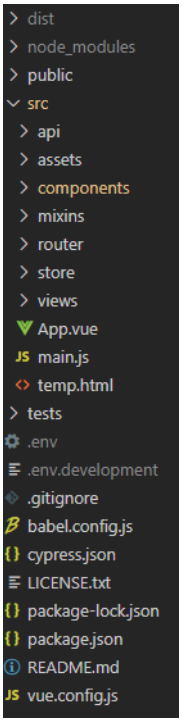


Figura 5.2: Estructura de proyecto del frontend

H1: Creación de GTFS

Como poseemos vistas de edición para todas las tablas de un GTFS, entonces el usuario puede crear un nuevo proyecto y agregar y editar los datos para crear un GTFS.

H2: Descarga como GTFS

El usuario puede simplemente crear un GTFS a partir de los datos y descargarlo.

H3: Edición de GTFS

El usuario es capaz de subir un GTFS con datos y operar de forma similar a lo descrito para la historia **H1**.

H4: Edición con CSV

La vista básica de tabla posee botones de subida y bajada de CSV, los cuales el usuario puede ocupar y el backend se encarga de realizar las operaciones correspondiente.

H5: Edición de paraderos

La vista de Stops entrega al usuario las herramientas requeridas por el usuario en esta historia.

H6: Edición de Shape

La vista de edición de Shapes satisface la necesidad del usuario en esta historia.

H7: Edición de StopTimes

El usuario de la historia puede simplemente acceder a la vista de edición de Stoptimes para lograr su objetivo.

Capítulo 6

Conclusión

El trabajo realizado permite a la gente de TranSapp crear y editar archivos GTFS y cumple con los objetivos planteados. Los modelos creados permiten manipular los datos que representan un GTFS, las vistas de tabla cumplen el objetivo de permitir la edición de datos a través de una API y los mapas interactivos ayudan al usuario con las tablas geográficas. La importación y exportación de GTFS está implementada y a través de las reuniones diarias, los clientes han podido validar el producto.

Django y Vue cumplieron bien su rol como frameworks, tienen una curva de aprendizaje media-alta pero una vez aprendidos simplifican bastante el diseño e implementación del backend y frontend. Los modelos de Django (ORM) son una característica sumamente útil, son un poco más limitados que SQL directo pero su uso es muchísimo más sencillo y evita tener que mezclar código en distintos lenguajes (SQL y Python).

Mapbox y Map-Matching cumplen bien la función para la que fueron empleadas, pero usarlas significa que las vistas que las ocupan son dependientes de que su disponibilidad. Si Mapbox decide, por ejemplo, cambiar su política de precios, o si actualiza su API y elimina el soporte para la versión actual, esto significa que la edición de 3 tablas importantes ya no es realizable con la aplicación. Existen alternativas pero implementarlas requeriría rediseñar completamente estas vistas. Pese a estos riesgos, utilizar las herramientas de Mapbox simplifica tanto un componente complejo que el beneficio que entrega supera bastante a los riesgos.

Las metodologías aplicadas dieron buen resultado para el desarrollo del proyecto, pues el proceso de validación continua permitió obtener retroalimentación inmediata de lo desarrollado, además de permitir establecer y priorizar tareas pequeñas. Una desventaja es que al tener reuniones diarias resultaba fácil perder de vista el objetivo principal y quedarse estancado en perfeccionar un componente del proyecto. Considerando todo, las metodologías empleadas eran adecuadas para el tipo de proyecto.

Un aprendizaje de esta memoria es acerca de las dinámicas de trabajo en equipo. Es difícil trabajar de forma solitaria, y realizar reuniones recurrentes de equipo ayudó a motivar el trabajo. La frecuencia de estas reuniones también es importante, dado que al pasar de

reuniones semanales a reuniones diarias, la velocidad de avance aumentó considerablemente. Además, desarrollar una memoria en pandemia también fue un desafío, esto pues se presentan más distracciones y es mucho más difícil establecer la línea entre el tiempo de trabajo y el tiempo de asueto.

6.1. Trabajo futuro

Existen mejoras de eficiencia que se podrían realizar. Algunas de las consultas son un poco lentas, sobre todo al utilizar la búsqueda en las tablas más grandes. Una búsqueda en trips puede hacer que la consulta se demore unos 5 segundos por página. Esto se podría mejorar agregando índices a las tablas.

Faltan algunas mejoras de interfaz. En una reunión con un diseñador de UX cerca del final del proyecto se levantaron varias sugerencias, pero con poco tiempo para implementarlas. Esto significó que en la fase final del proyecto quedaron varias mejoras pequeñas de interfaz pendientes.

Algunas de las tablas opcionales del estándar GTFS, vale decir, aquellas que no son requeridas para describir la red pero complementan la información, fueron dejadas de lado (levels, transfers, fareattributes y farerules), esto pues hoy en día no son usadas en TranSapp. Ocupan la vista de tabla pero si se empiezan a utilizar habría que analizar si requieren alguna funcionalidad o mejora de interfaz importante. Por ejemplo, los Stops tienen un atributo de zona, utilizado para definir tarifas al transportarse entre zonas utilizando la tabla de FareRules. Las zonas no son directamente una entidad del GTFS, pero una vista de zonas podría permitir manejar de mejor forma esta información.

Una guía de usuario resultaría bastante útil para enseñar a nuevos usuarios cómo usar la aplicación. Las interfaces más sencillas son intuitivas, pero las más complejas podrían resultar difíciles de entender.

Bibliografía

[1] A. E. Antoine Ortiz, "Caso práctico de estudio de incorporación de técnicas de gamificación en aplicación móvil, TranSapp," Memoria de Titulación. Universidad de Chile, 2018.

[2] TranSapp, "Sitio web de TranSapp," (último acceso: 23-04-2021). [Online]. Available: https://www.transapp.cl/

[3] DTPM, "Página de descarga del GTFS del DTPM," (último acceso: 23-04-2021). [Online]. Available: http://www.dtpm.gob.cl/index.php/gtfs-vigente

[4] Google, "Descripción de la especificación GTFS," (último acceso: 23-04-2021). [Online]. Available: https://developers.google.com/transit/gtfs

[5] Conveyal, "Documentación de Transit Data Tools," (último acceso: 23-04-2021). [Online]. Available: https://mtc-datatools.readthedocs.io/en/stable/

[6] M. van Laar, "Static-GTFS-manager," (último acceso: 23-04-2021). [Online]. Available: https://github.com/mvanlaar/static-GTFS-manager/

[7] Django Software Foundation, "Sitio web de Django," (último acceso: 23-04-2021). [Online]. Available: https://www.djangoproject.com/

[8] E. You, "Sitio web de Vue," (último acceso: 23-04-2021). [Online]. Available: https://vuejs.org/

[9] Django Software Foundation, "Referencia de modelos de Django," (último acceso: 23-04-2021). [Online]. Available: https://docs.djangoproject.com/en/3.1/topics/db/models/

[10] R. T. Fielding, *Architectural styles and the design of network-based software architectures*. University of California, Irvine Irvine, 2000, vol. 7.

[11] T. Christie, "Django REST framework," (último acceso: 23-04-2021). [Online]. Available: https://www.django-rest-framework.org/

[12] *ECMA-404 The JSON Data Interchange Format*, 1^{st} ed., ECMA, 2013.

[13] Amazon, "Sitio de AWS," (último acceso: 23-04-2021). [Online]. Available: https://aws.amazon.com/what-is-aws/

[14] Docker, “Manual de Docker,” (último acceso: 23-04-2021). [Online]. Available: https://docs.docker.com/engine/

[15] R. Wannapanop, “Documentación de Vuetable,” (último acceso: 23-04-2021). [Online]. Available: https://www.vuetable.com/#current-version/

[16] Select2, “Documentación de Select2,” (último acceso: 23-04-2021). [Online]. Available: https://select2.org/

[17] IETF, “Estándar GeoJSON,” (último acceso: 23-04-2021). [Online]. Available: https://tools.ietf.org/html/rfc7946

Apéndice A

Ejemplos de CSV

En este anexo disponemos de muestras de los principales archivos del GTFS del DTPM, omitiendo columnas poco importantes para que quepan en el documento.

stop_id	stop_name	stop_lat	stop_lon
PB1	PB1-Venezuela / Esq. Bolivia	-33.4045533117672	-70.6230946022859
PB2	PB2-Venezuela / Esq. H. De La Concepción	-33.4024533488634	-70.6266392527421
PB3	PB3-Reina De Chile / Esq. Avenida El Salto	-33.4012186488925	-70.6297343310634
PB4	PB4-Pedro Donoso / Esq. Santa Ana	-33.3992261697921	-70.6329785711632
PB5	PB5-Pedro Donoso / Esq. María Del Pilar	-33.3983332917554	-70.6368109638332

Tabla A.1: stops.txt

route_id	agency_id	route_short_name	route_long_name	
201	RM	201	Mall Plaza Norte - San Bernardo	3
203	RM	203	Huechuraba - La Pintana	3
204	RM	204	Alameda - Gabriela	3
205	RM	205	Santiago - Puente Alto	3
206	RM	206	Centro - La Pintana	3

Tabla A.2: routes.txt

shape_id	shape_pt_lat	shape_pt_lon	shape_pt_sequence
301c2R	-33.5860380005	-70.6555940001	1
301c2R	-33.5860410005	-70.6559680001	2
301c2R	-33.5860450005	-70.6565010001	3
301c2R	-33.5860380005	-70.6566750001	4
301c2R	-33.5860130005	-70.6568440001	5
106R	-33.5044860005	-70.565635	1
106R	-33.5046110005	-70.564978	2
106R	-33.5047070005	-70.564474	3
106R	-33.5047460005	-70.564269	4
106R	-33.5048910005	-70.563478	5

Tabla A.3: shapes.txt

route_id	service_id	trip_id	trip_headsign	direction_id	shape_id
101	L	101-I-L-B02	Cerrillos	0	101I
101	L	101-I-L-B03	Cerrillos	0	101I
101	L	101-I-L-B04	Cerrillos	0	101I
101	L	101-I-L-B05	Cerrillos	0	101I
101	L	101-I-L-B06	Cerrillos	0	101I

Tabla A.4: trips.txt

trip_id	arrival_time	departure_time	stop_id	stop_sequence
101-I-L-B02	00:00:00	00:00:00	PB1	1
101-I-L-B02	00:01:40	00:01:40	PB2	2
101-I-L-B02	00:02:59	00:02:59	PB3	3
101-I-L-B03	00:00:00	00:00:00	PB1	1
101-I-L-B03	00:01:57	00:01:57	PB2	2
101-I-L-B03	00:03:29	00:03:29	PB3	3

Tabla A.5: stoptimes.txt

trip_id	start_time	end_time	headway_secs	exact_times
101-I-D-B24	09:30:00	13:30:00	960	0
101-I-D-B24	21:20:00	09:30:00	960	0
101-I-D-B25	13:30:00	17:30:00	960	0
101-I-D-B25	00:40:00	13:30:00	800	1
101-I-D-B26	17:30:00	21:00:00	1400	0

Tabla A.6: frequencies.txt

Apéndice B

Mockups

En este anexo podemos ver los mockups creados para las vistas descritas por el flujo. Tomar en cuenta que fue necesario ajustar los mockups para que sean legibles en este documento. La figura B.1 representa a la vista de `Home Dashboard` de la figura 3.1. Hacerle click a un proyecto nos lleva a la vista de `Projct Overview`. Trasladándonos al flujo de la figura 3.2, la vista de `Project Overview` está representada por el mockup de la figura B.2. Las acciones de `Upload File`, `Publication Management` y `Validate` han sido implementadas con los botones de `Upload GTFS File`, `Publications` y `View Details` respectivamente. Hacer click a una tabla nos lleva a la tabla correspondiente con dos excepciones, la acción de Manage Shapes corresponde a hacerle click a la tabla de Shapes, llevándonos a la vista de `Shape Management View` y hacerle click a `stop-times` nos lleva a la vista representada por la figura B.13, la cual corresponde en el flujo a `Stop Times Table View`. Dentro de esta vista tenemos un botón para editar el stop-times. `Publication Manager` corresponde a la figura B.4, donde podemos ver la acción de publicar y la lista de tareas de publicación anteriores. La vista `Error View` se encuentra implementada en la figura B.5, donde tenemos la lista de errores y advertencias. `Shape Management` corresponde al mockup de la figura B.10.

Las vistas de tabla son descritas por el flujo de la figura 3.3. La figura B.6 representa `Basic Table View`, una vista de tabla básica empleada por la mayoría de las tablas de un GTFS, con distintas versiones según la tabla a editar. Como podemos apreciar contiene los botones para subir y bajar los datos como CSV, además de permitirnos filtrarlos datos y ordenarlos. `Route Tables View` está representada por la figura B.8, siendo básicamente la vista básica de tablas con un botón para acceder a los trips, el cual nos lleva a `Trips Table View`. Esta vista a su vez está implementada en la figura B.9, otra vista básica pero ahora con un botón para acceder al stop-times y un mapa para visualizar el `shape` y los paraderos, tal como se describe en el diagrama de flujo. Si accedemos a `stop-times` entonces llegaríamos a la vista de la figura B.14. La vista de frecuencias (figura B.15) corresponde a una vista básica pero se ha incluido en los mockups por su uso en la validación de estos. La vista `Stop Tables View` corresponde a la figura B.7. Como podemos ver, esta vista contiene el mapa interactivo descrito en el flujo. Al hacer click en un paradero este debiera destacarse en la tabla de paraderos, y arrastrar un paradero en el mapa debiera actualizar las coordenadas de este. Hacer click a edit en un `stop-times` en las vistas de las figuras B.9 y B.14 nos lleva a la vista de `Stop Times Table View`, la cual es representada por la figura B.14. La vista de edición

de `shapes` es accesible desde las vistas de las figuras B.9 (trips) y B.10 (gestión de shapes). Esta vista tiene dos versiones, la figura B.11 es la versión por defecto, donde uno define los puntos que componen el `shape` directamente y la figura B.12 utiliza una funcionalidad de MapBox/OpenStreetMaps para generar el shape a partir de puntos definidos por el usuario, utilizando la información de calles.

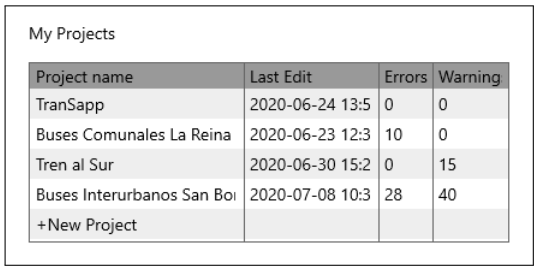


Figura B.1: Mockup de vista de proyectos

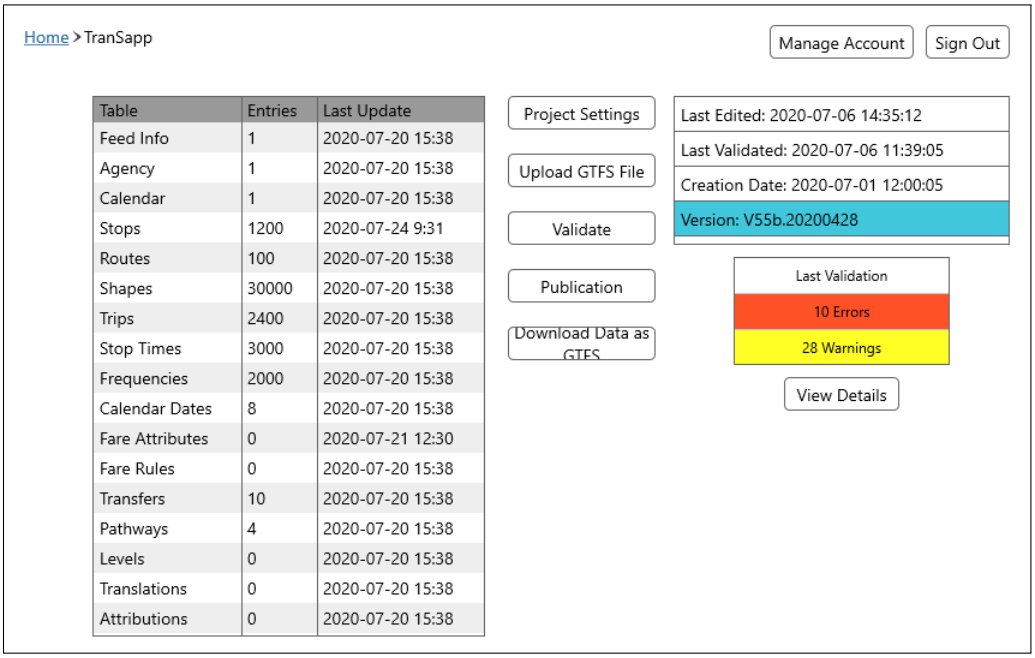


Figura B.2: Mockup de vista de proyecto

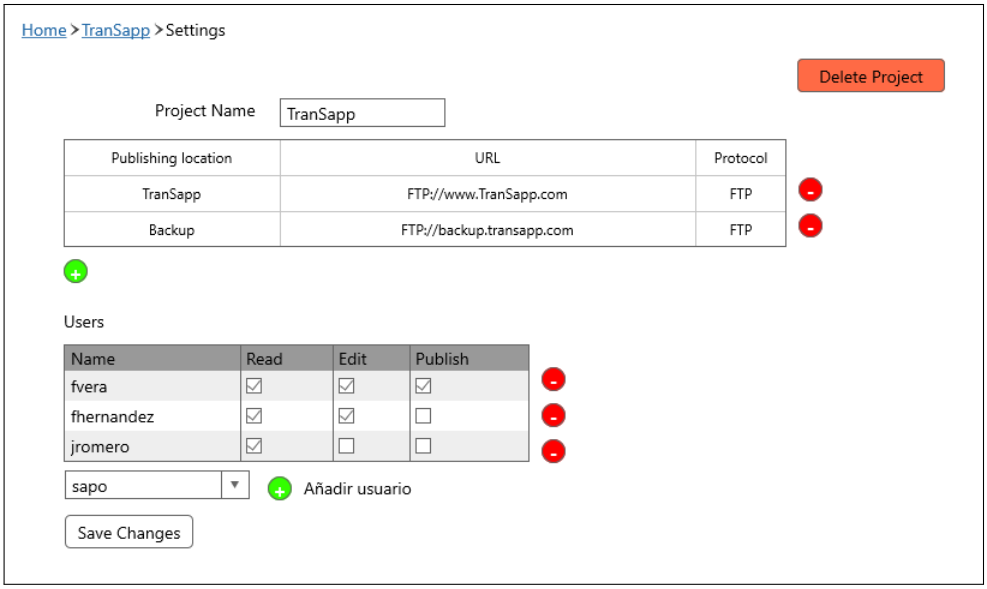


Figura B.3: Mockup de vista de configuración de proyecto

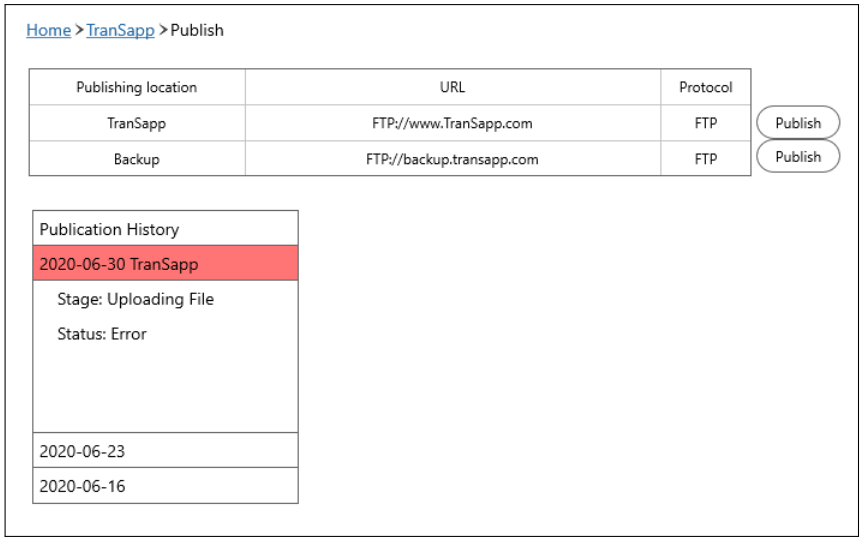


Figura B.4: Mockup de vista de publicación

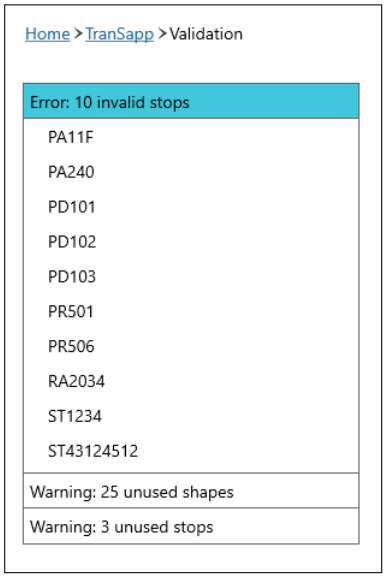


Figura B.5: Mockup de vista de validación

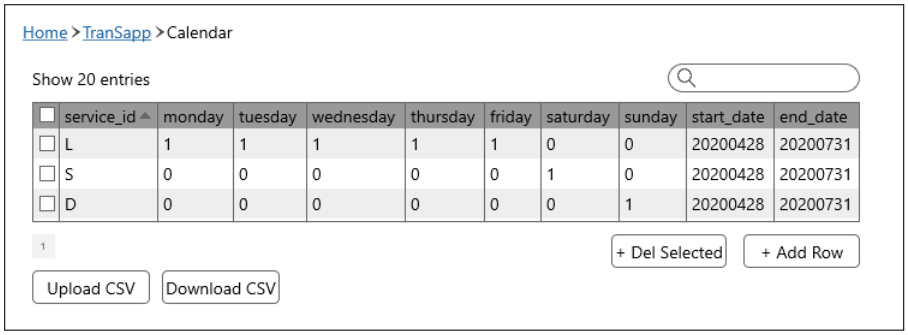


Figura B.6: Mockup de vista básica de edición de tabla

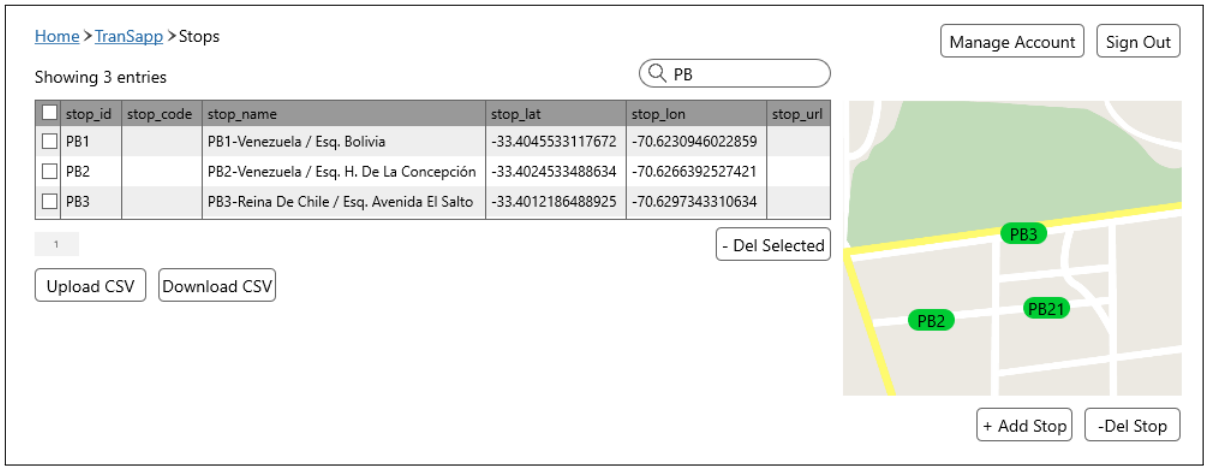


Figura B.7: Mockup de vista de edición de paraderos

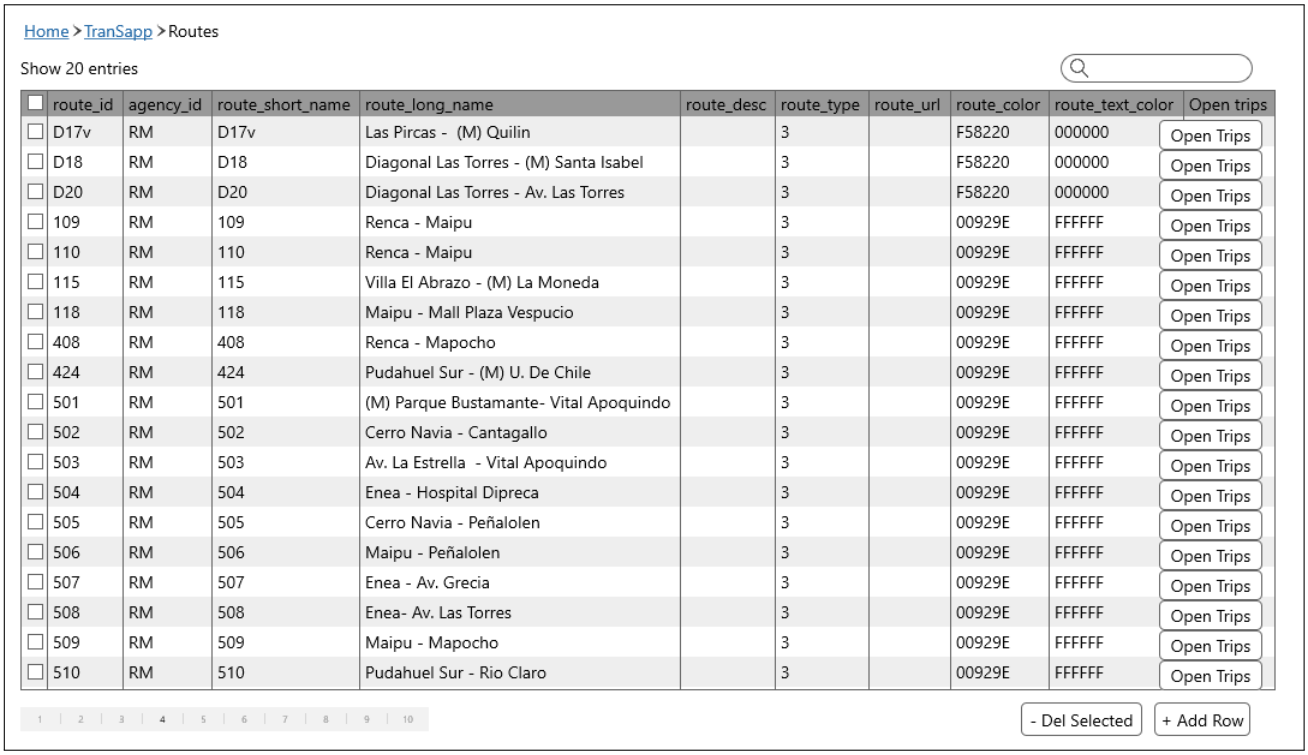


Figura B.8: Mockup de vista de edición de rutas

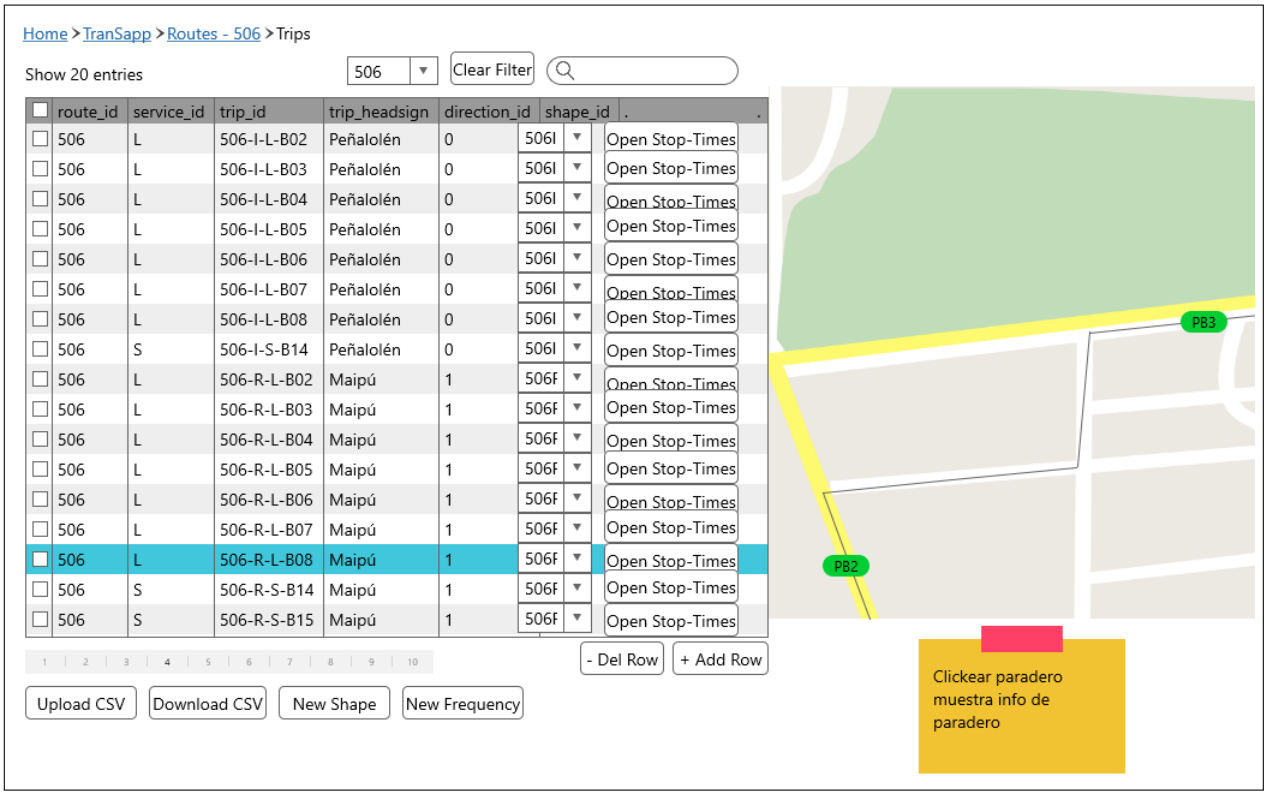


Figura B.9: Mockup de vista de edición de viajes

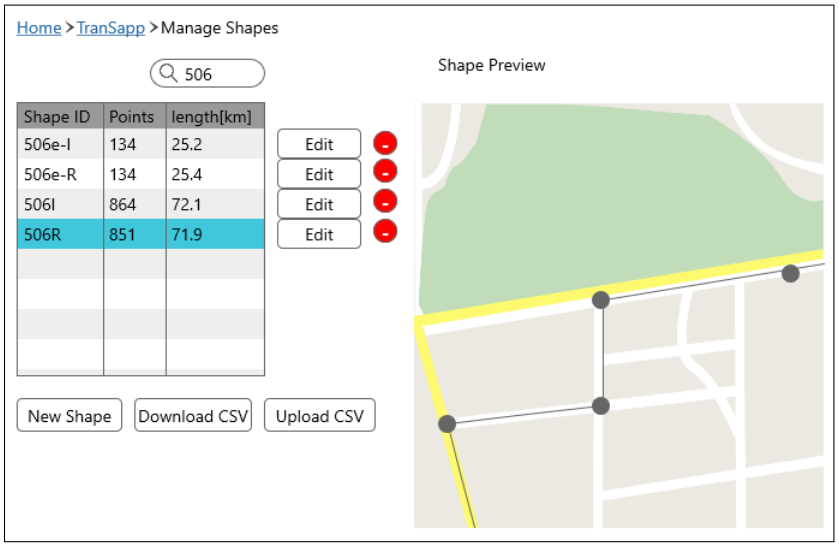


Figura B.10: Mockup de gestión de `shapes`

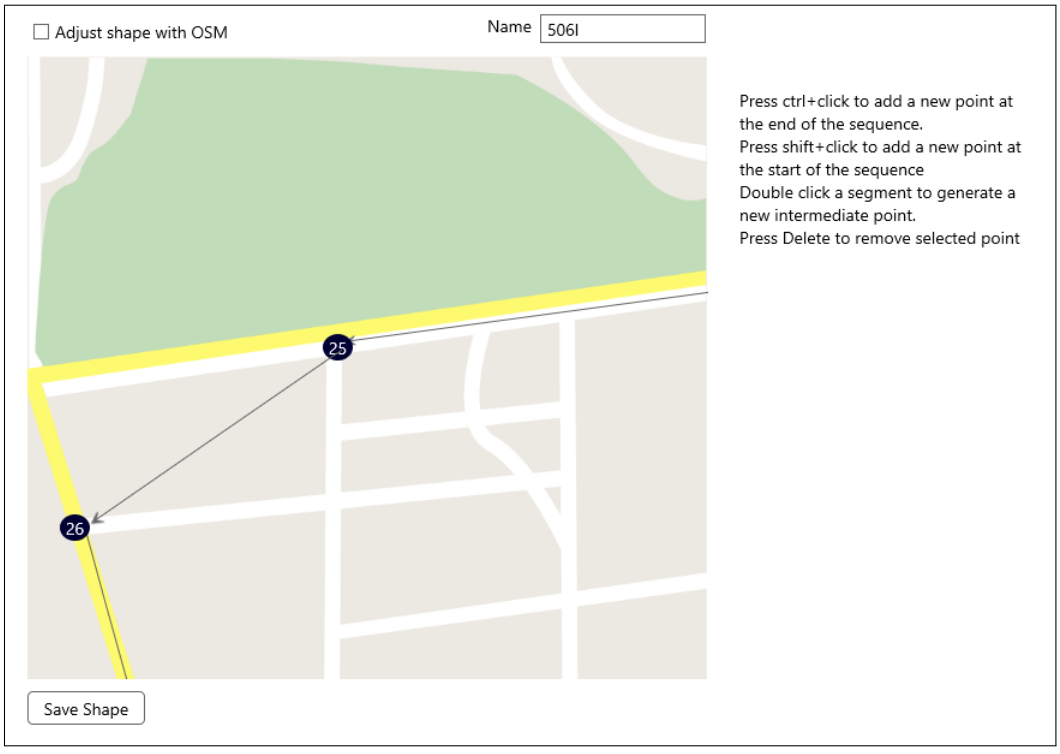


Figura B.11: Mockup de vista de edición de `shapes` con OSM

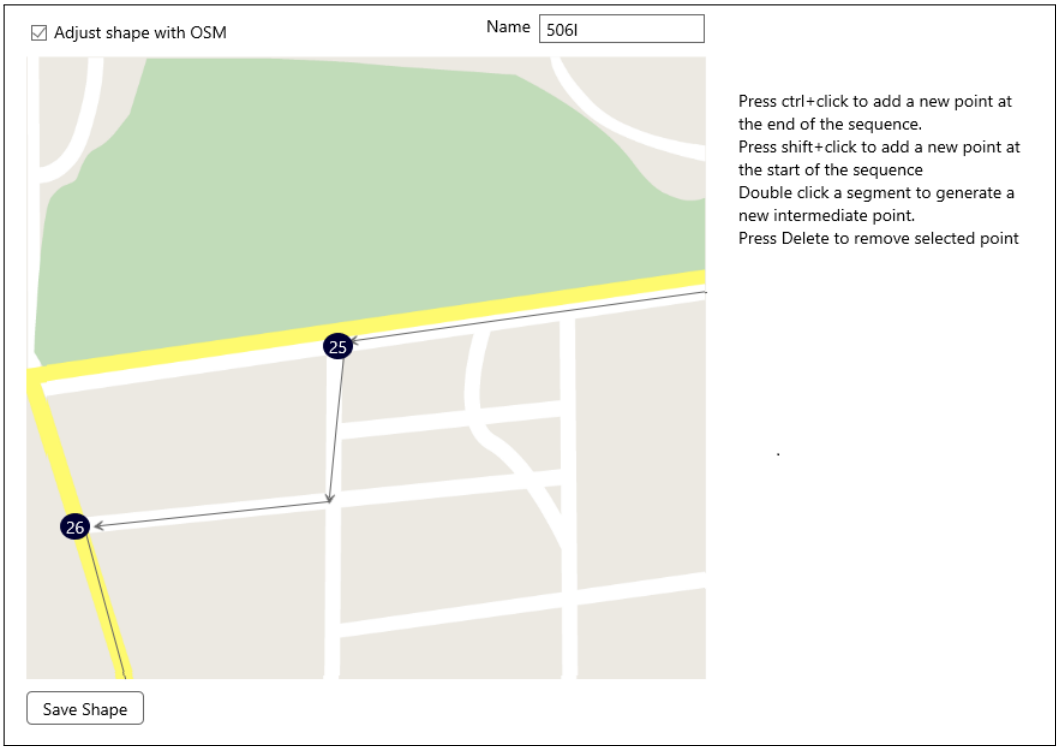


Figura B.12: Mockup de vista de edición de `shapes` sin OSM

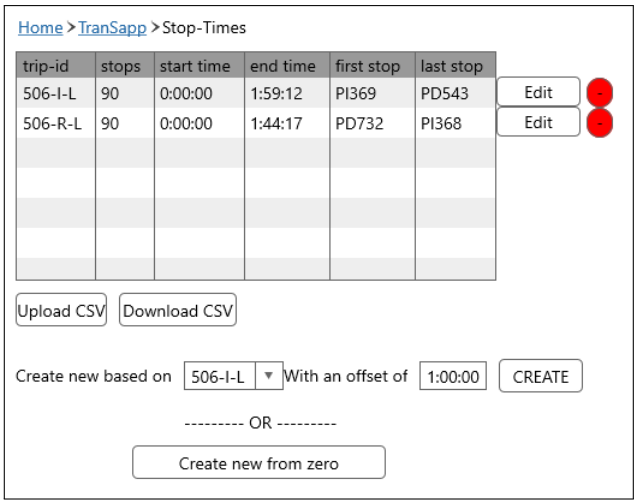


Figura B.13: Mockup de gestión de `stop-times`

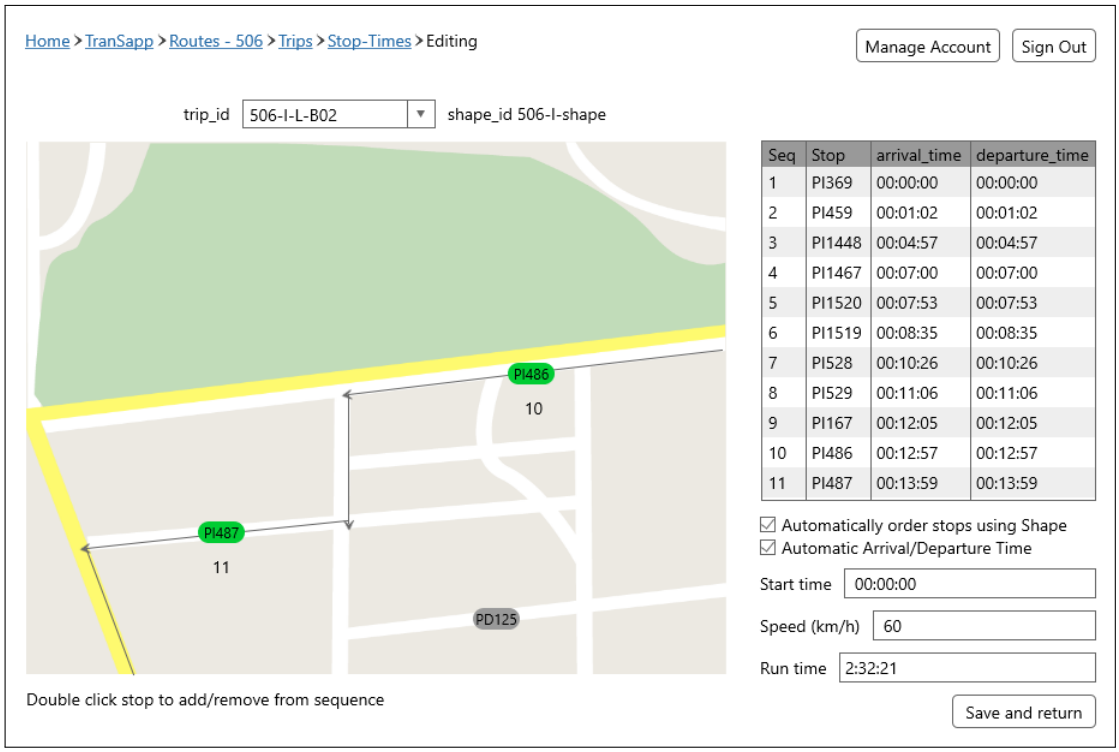


Figura B.14: Mockup de vista de edición de `stop-times`

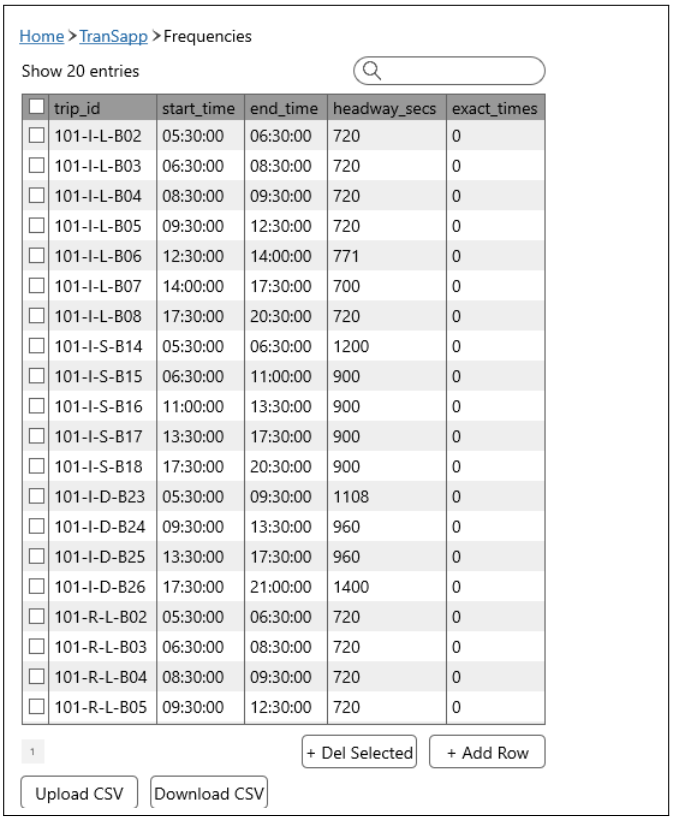


Figura B.15: Mockup de vista de edición de frecuencias