Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

TrackYourTrails: aplicación web para la geolocalización de recorridos

TrackYourTrails: web application for route geolocation

Paula Elena Expósito Estévez

La Laguna, 10 de junio de 2021

D. **Vicente José Blanco Pérez**, con N.I.F. 42.171.808-C profesor Titular de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor

C E R T I F I C A

Que la presente memoria titulada:

"*TrackYourTrails: aplicación web para la geolocalización de recorridos*"

ha sido realizada bajo su dirección por Dña. **Paula Elena Expósito Estévez**, con N.I.F. 43.382.565-B.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 10 de junio de 2021

Agradecimientos

Quiero agradecer a todas las personas que me han acompañado y apoyado durante estos años.

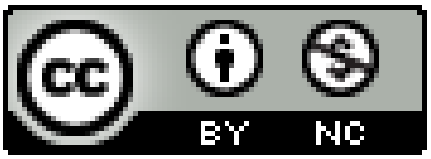
En especial a Grandma, que siempre está pendiente de mí, e incluso se conoce las fechas de entrega mejor que yo.

A mi madre, a mi padre y a mi hermana, que han sufrido mi mal humor cada vez que una práctica no salía o había época de exámenes y tenían que llevarme a coger aire.

A los amigos y conocidos que he hecho en la carrera, que la han hecho más amena y que me han enseñado muchas cosas. Son ellos los que han generado los mejores recuerdos de mi paso por la universidad.

A mi tutor de este trabajo y el resto de profesores que se han esforzado en enseñarme y de quienes he aprendido mucho.

Licencia



Resumen

Debido al uso generalizado de aplicaciones móviles nativas para el registro de tracking *GPS, en este trabajo se desarrolla una aplicación web que realiza estas funciones. El interés reside en la dificultad de las aplicaciones web para trabajar en segundo plano, por lo que, para solventar este problema, se utilizan* Service Workers*, una tecnología que permite programar aplicaciones de tipo* frontend *desacopladas de la visualización.*

La aplicación desarrollada está conformada por un sistema distribuido con tres servicios. Una base de datos no relacional en la nube, MongoDB Atlas. Un servidor NodeJS con una API desarrollada con Koa mediante una arquitectura de tres capas. Y una aplicación cliente de VueJS desarrollada con el framework *Quasar. Los dos últimos servicios se han construido como contenedores de Docker que se despliegan automáticamente en Heroku.*

Para el desarrollo del proyecto se ha hecho uso de las facilidades y herramientas que ofrece GitHub. Tanto para el desarrollo y seguimiento del trabajo como para la integración y el despliegue continuo.

Palabras clave: NodeJS, API, VueJS, Service Worker, CI/CD, tracking GPS

Abstract

There are many mobile native applications for registering GPS tracking, but not so many implemented as a web application. In this project, a web application with these capabilities has been developed. The main interest in this project resides in the difficulty to implement background processes in web applications while the devices are locked (no frontend running). A web technology called Service Workers are used to solve this problem. This technology allows you to develop frontend applications disconnected from visualization functionalities.

The developed application consists of a distributed system with three services: a non-relational cloud database, MongoDB Atlas; a NodeJS server with a Rest API implemented with Koa and a three-tier architecture; and a VueJS client application which has been developed in the Quasar framework. The last two services have been built as Docker containers that are automatically deployed to Heroku.

GitHub facilities and related tools have been used to manage the project. Code versioning, project management and issues as well as continuous integration and continuous deployment were managed with these tools.

Keywords: NodeJS, API, VueJS, Service Worker, CI/CD, tracking GPS

Índice general

Índice de Figuras

Índice de Tablas

Índice de Listados

Capítulo 1

Introducción

En este capítulo se presenta el propósito del proyecto, los objetivos y la metodología utilizada. Así como los antecedentes y el estado actual del tema.

1.1. Motivación y propósito del trabajo

El verano pasado empecé a caminar a menudo por diferentes partes de la isla. Tras haberlo hecho un par de semanas empecé a querer tener un registro de las caminatas que había hecho. Conseguirlo no fue difícil ya que hay muchas aplicaciones que registran *tracking*.

Al buscar qué aplicación utilizar me pareció curioso que, a pesar de tener una versión web, ninguna me dejaba registrar el *tracking* desde ahí. Todas me pedían que las instalase en el móvil.

Por ello, el propósito de este proyecto es desarrollar una aplicación web que permita guardar el *tracking* realizado por una persona al hacer senderismo. De forma que, además de registrar el recorrido, el usuario tenga a su disposición todas las rutas que ya ha hecho.

1.2. Antecedentes y estado actual del tema

Aunque caminar es algo que los seres humanos han hecho siempre, según se fueron adoptando nuevas formas de transporte las personas fueron alejándose de esta actividad. No fue hasta el siglo XVIII que se empezó a tomar el caminar como una actividad recreativa. A partir de entonces se fue popularizando, hasta tal punto que en 1947 Francia empezó a señalizar caminos con el fin de crear una red de senderismo que recorriese todo el país.

La medida francesa fue rápidamente adoptada por el resto de países, consi-

guiendo que practicar senderismo fuese seguro para todos. En la actualidad, el senderismo está muy relacionado con el turismo, cuestión que contribuye a la mejora y mantenimiento de los caminos. Caminar por espacios naturales es una forma muy buena de conocer nuevos lugares, estar más sano y relacionarse con otras personas.

Por otra parte, se trata de una actividad que cualquier persona puede realizar. Tanto niños como personas mayores pueden practicarlo adaptando los recorridos a sus necesidades. Al no haber restricciones de tiempo ni de dificultad existen miles de recorridos donde elegir, con diferentes dificultades, duraciones y desniveles. Además, es algo que se puede hacer en casi cualquier sitio y en cualquier momento, no hay que ir muy lejos para encontrar un buen lugar para caminar un rato por la naturaleza.

Debido a la popularidad del senderismo y el gran avance de las tecnologías, en los últimos años ambos se han unido. Registrar *tracking*, grabar recorridos, saber con qué velocidad se ha caminado en cada momento, conocer el desnivel realizado, calcular estadísticas que resuman las actividades hechas, controlar las calorías quemadas,... son sólo algunos ejemplos de lo que las aplicaciones pueden hacer.

Disponer de todos estos datos interesa mucho a los caminantes. Es un campo muy amplio, donde además de existir muchas aplicaciones para móviles que ofrecen todas estas características, hay un mercado entero dedicado a relojes deportivos que son muy precisos y que están dotados de una gran variedad de funciones y sensores especializados.

A pesar de que al buscar en Internet encontremos muchas aplicaciones dedicadas al registro de *tracking*, no encontraremos ninguna, o casi ninguna, que permita registrarlo sin descargar una aplicación, a pesar de disponer de una versión web. Esto se debe a que se trata de aplicaciones nativas para sistemas operativos móviles como Android o iOS.

El motivo de que no haya aplicaciones web que registren *tracking* es que este tipo de aplicaciones no funcionan bien en segundo plano. Hasta hace relativamente poco no había una tecnología lo suficientemente potente para darle soporte durante largos períodos de tiempo.

Los *Service Workers* [1] son unos *scripts* que permiten realizar estas tareas en *background*, e incluso sin conexión a Internet. Con estos *scripts* sí que es posible que haya aplicaciones web que registren *tracking*. Disponiendo de esta tecnología y no habiéndose explotado en este campo es interesante estudiar cómo se comporta y qué tanto se diferencia su comportamiento en comparación a las aplicaciones nativas ya existentes.

Las aplicaciones de referencia en este campo son las siguientes:

- **AllTrails [2]:** esta aplicación está especializada en ofrecer rutas por entornos naturales. Destaca por el tratamiento que da a las rutas fuera de carretera, mostrando de forma muy clara todas las pistas que existen. En otras aplicaciones similares la comprensión del mapa es difícil.
- **Wikiloc [3]:** es una aplicación española muy popular que destaca por su trabajo con mapas. Tiene muchísimas rutas disponibles y se centra principalmente en las versiones para móviles.
- **Strava [4]:** esta aplicación está más orientada a un ámbito de competición deportiva pero contiene las principales características de registro de *tracking* y gestión de mapas. A diferencia de las anteriores esta aplicación permite interactuar con otros usuarios.
- **Endomondo [5]:** aunque esta es una aplicación deportiva que ha cerrado recientemente destacaba porque permitía registrar a otros usuarios como amigos y les mostraba actualizaciones sobre las actividades de estos, facilitando la interacción y motivando a los usuarios a seguir haciendo ejercicio.

1.3. Objetivos

El objetivo de este proyecto es desarrollar una aplicación web capaz de registrar *tracking*. Para ello se utilizan tecnologías modernas que facilitan el desarrollo, la escalabilidad y la seguridad de la aplicación.

A parte de registrar y visualizar el *tracking* mientras se graba, con la aplicación se podrá acceder a las rutas y a los datos que ya se han almacenado con anterioridad. Además, las rutas grabadas se añadirán a un banco de rutas general que otros usuarios podrán ver. Por último, se podrán crear eventos. Esto consiste en un espacio donde los usuarios pueden planificar una excursión, especificando qué ruta se va a seguir, la fecha en la que se va a realizar, el punto de encuentro y otros datos que puedan ser de interés. Una vez definidos otros usuarios se podrán unir.

1.4. Metodología

Al inicio del proyecto se plantearon las siguientes tareas a realizar:

- Setup de la infraestructura
- Prototipado de la aplicación
- Diseño y desarrollo de la API
- Desarrollo del Backend

· Setup del Frontend

· Desarrollo del Frontend

Aunque en un principio se planeó realizar estas tareas de forma secuencial, en la práctica se fueron haciendo en paralelo, según qué funcionalidad se estaba desarrollando. Además, se iban actualizando los componentes terminados cuando se añadían otros nuevos.

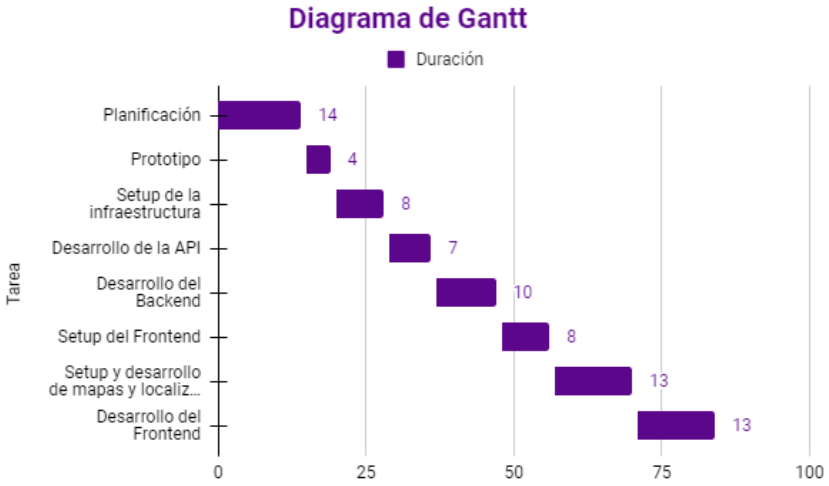


Figura 1.1: Planificación inicial de las tareas a realizar.

Capítulo 2

Tecnologías y herramientas

En este capítulo se describen las tecnologías y las herramientas utilizadas para el desarrollo del proyecto.

2.1. Entorno de desarrollo

A continuación se incluyen las herramientas que, aún no participando del código fuente de la aplicación, sí que han hecho posible y facilitado su desarrollo. Contiene el editor y el sistema utilizado, las herramientas que posibilitan el despliegue y la integración continua y el *linter* utilizado.

2.1.1. Visual Studio Code

Visual Studio Code [6] es un editor de código desarrollado por Microsoft que permite el desarrollo colaborativo. Se puede utilizar en diferentes sistemas operativos como Windows y Linux. En este caso se ha utilizado en un subsistema Linux, que se comenta con más detalle en el siguiente apartado. Además, VSC está optimizado para construir y depurar aplicaciones web y en la nube.

Se ha elegido porque tiene muchas extensiones que permiten personalizarlo y adaptarlo a las necesidades de cada proyecto, siendo muy fácil de utilizar. En concreto, para el proyecto se utilizan las extensiones ”Docker” y ”Vetur”.

2.1.2. WSL. Windows Subsystem Linux

WSL [7] es, como indica su nombre, un subsistema Linux en un sistema Windows. Con esta herramienta se puede desarrollar desde Windows con todas las ventajas de un entorno Linux, sin tener que utilizar máquinas virtuales ni hacer particiones en el disco duro. En este caso se utiliza un Ubuntu 20.04 LTS.

2.1.3. Docker

Docker [8] es un proyecto de código abierto que automatiza el despliegue de aplicaciones mediante la creación de contenedores. La contenerización, o *dockerización*, consiste en empaquetar una aplicación en diferentes módulos que se distribuyen por separado pero que trabajan conjuntamente. Por ello, para esta aplicación se han creado dos contendores diferentes, uno para la aplicación de NodeJS [9] del *backend* y otro para la aplicación de VueJS [10] del *frontend*.

Al desarrollarse la aplicación en Windows, los requisos de Docker obligan a utilizar WSL 2.

2.1.4. GitHub

GitHub [11] es una plataforma colaborativa donde alojar proyectos que utilizan control de versiones en la nube. A parte de almacenar y administrar los proyectos, GitHub tiene muchas otras funcionalidades, de las cuales algunas se han utilizado en el proyecto. Son las siguientes:

GitHub Projects

GitHub Projects [12] es una herramienta de administración de proyectos. Consiste en un tablero con diferentes columnas donde se colocan tarjetas que representan cada tarea del proyecto. Para esta aplicación se utiliza un tablero *kanban* automático que consta de cuatro columnas y múltiples tarjetas. Las tarjetas se mueven de una columna a otra automáticamente a través de los *issues*.

GitHub Actions

GitHub Actions [13] es una herramienta que automatiza flujos de trabajo como la ejecución de los tests, la publicación de la cobertura y la integración y el despliegue continuo.

GitHub Packages

GitHub Packages [14] es un repositorio en la nube donde se pueden publicar paquetes. Por ejemplo, almacena contenedores de Docker que luego pueden descargarse y ejecutarse en cualquier máquina. Los paquetes deben registrarse en el GitHub Package Container Register para poder ser publicados.

2.1.5. Heroku

Heroku [15] es una plataforma como servicio (PaaS) para desplegar, ejecutar y administrar aplicaciones en la nube. Se ha elegido porque es una herramienta fácil de integrar con GitHub Actions, lo que permite automatizar el despliegue de la aplicación.

2.1.6. Codecov

Codecov [16] es una herramienta que muestra el estado de la cobertura del código, otorgando a los desarrolladores conocimiento sobre la calidad de este. En el proyecto, como se utilizan dos aplicaciones diferentes, una para el *backend* y otra para el *frontend*, se ha hecho uso de los Flags [17] para poder obtener la cobertura total.

2.1.7. ESLint

ESLint [18] es un *linter* de JavaScript, se utiliza para arreglar errores en la sintaxis del código automáticamente. Se compone de reglas personalizadas que se aplican con un comando. Suelen utilizarse para seguir buenas prácticas y guías de estilo, consiguiendo que todo el código tenga la misma apariencia.

2.2. Backend

En este apartado se explican las tecnologías utilizadas en el *backend* de la aplicación. Incluye el entorno y el *framework* utilizado, las herramientas para testing, el estándar para la gestión de la sesión y la base de datos elegida.

2.2.1. NodeJS

NodeJS es un entorno de ejecución multiplataforma orientado a eventos asíncronos. Está basado en el motor V8 de Google, lo que permite ejecutar en diferentes navegadores aplicaciones JavaScript del lado del servidor. Con NodeJS se pueden desarrollar sistemas escalables debido a que no tiene muchas funciones bloqueantes de la E/S, lo que consigue que el proceso nunca se bloquee.

2.2.2. Koa

Koa [19] es un *framework* de NodeJS optimizado para aplicaciones web y APIs. Se eligió este *framework* frente a Express [20] porque está específicamente diseñado para ser más robusto y ligero que este. Y se eligió sobre NestJS [21] porque

en el proyecto sólo se necesitaba desarrollar una API, lo que hizo que un *framework* tan grande y que se diferenciaba tanto de las tecnologías ya conocidas fuese innecesario. Koa es un *framework* que facilita el manejo de errores y que, al ser muy ligero, no incluye *middlewares* en su núcleo.

2.2.3. Postman y Newman CLI

Postman [22] es una plataforma para el desarrollo colaborativo de APIs. En este proyecto se ha utilizado para realizar tests End-To-End (E2E) a la API mediante la creación de colecciones que se ejecutan en la línea de comandos con Newman CLI [23].

```
> trackyourtrails@1.0.0 test:api /home/pauluchi/TFG-TrackYourTrails/backend
> newman run test/postman/auth.collection.json -e test/postman/auth.environment.json
ostman_collection.json

newman

Auth Collection

→ Create user Amaia
  POST localhost:3000/api/signup [201 Created, 530B, 239ms]
  ✓  User created successfully with a 200 code
  ✓  Check that it returns an object
  ✓  Confirm response message
  ✓  Check that token is generated

→ Create user Amaia Again
  POST localhost:3000/api/signup [409 Conflict, 218B, 73ms]
  ✓  User couldn't be created with 409 code
  ✓  Check that it returns an object
  ✓  User already exists response message

→ Create user (with not enough data)
  POST localhost:3000/api/signup [409 Conflict, 227B, 4ms]
  ✓  User couldn't be created with 409 code
  ✓  Check that it returns an object
  ✓  Missing data in request message
```

Figura 2.1: Ejecución de la colección de autenticación.

2.2.4. Jest

Jest [24] es un *framework* de testing de JavaScript que se utiliza para los tests unitarios del servidor. Está diseñado para ser simple y rápido. Además, cuando los tests acaban de ejecutarse muestran un resumen de la cobertura.

2.2.5. JWT. JSON Web Tokens

JSON Web Token [25] es un estándar para la generación de *tokens* de acceso para los usuarios. Utilizándolos se lleva a cabo la gestión de usuarios, garantizando

que un usuario es quien dice ser y permitiéndole el acceso a ciertas partes de la aplicación. En el Capítulo 3 se explican con más detalle.

2.2.6. Mongo Atlas

MongoDB [26] es una base de datos no relacional de código abierto. Se trata de una base de datos documental que almacena los datos en objetos JSON, aportándoles legibilidad y naturalidad. Además, cuenta con un lenguaje de consultas muy potente.

MongoDB Atlas [27] es la base de datos como servicio (DBaaS) de MongoDB. Se trata de una base de datos en la nube automática, segura y fácil de administrar. Es la versión que se ha utilizado en este proyecto.

2.3. Frontend

Este apartado se centra en las tecnologías utilizadas en el *frontend* de la aplicación. Los *frameworks* utilizados, la conexión con el *backend*, los *Service Workers* y las herramientas para la integración de los mapas.

2.3.1. Vue

VueJS es un *framework* progresivo de JavaScript. Tiene una curva de aprendizaje sencilla y se caracteriza porque contiene en un mismo archivo código HTML, JavaScript y CSS. Principalmente se utiliza para crear proyectos de tipo Single Page Application (SPA).

Vuex

Vuex [28] es una librería de Vue que implementa el patrón Flux. Este patrón reemplaza el Modelo-Vista-Controlador creando un *store* único en la aplicación al que todos los componentes pueden acceder. Al centralizarse, la comunicación entre componentes se simplifica mucho, ya que cuando cambia un valor del *store* se renderizan directamente los componentes que hacen uso de este dato.

VueRouter

VueRouter [29] es la herramienta encargada de enrutar las SPA de Vue. Otorga a cada vista una URL virtual que sólo existe dentro de la aplicación de Vue. Dependiendo de qué vista se muestre al usuario se renderizan unos componentes u otros.

2.3.2. Quasar

Quasar [30] es un *framework* de Vue para desarrollar aplicaciones del lado del cliente. El interés de este *framework* reside en que con un mismo código se puede compilar a SPA (Single Page Application), PWA (Progressive Web App), BEX (Browser Extension), SSR (Server Side Rendered App), Hybrid Mobile App y Multiplatform Desktop App.

Se ha elegido porque tiene una documentación muy completa y clara. Además de que contiene muchos componentes y extensiones que se pueden integrar fácilmente en los proyectos, y que facilitan el desarrollo de aplicaciones responsivas.

Quasar CLI

Quasar CLI [31] es la herramienta de la línea de comandos para ejecutar comandos de Quasar.

Quasar Testing

Quasar Testing [32] es una extensión para ejecutar tests unitarios y tests E2E. Dependiendo el tipo de testing que se quiera utilizar se pueden utilizar diferentes extensiones, por ejemplo, para tests unitarios se pueden utilizar Jest o AVA y para tests E2E, Cypress o WebDriver.io. En este caso se utiliza Jest, al igual que en el *backend*.

Notify

Notify [33] es un *plugin* de Quasar para mostrar mensajes a los usuarios en forma de notificaciones. Pueden personalizarse y desencadenar acciones.

2.3.3. Axios

Axios [34] es una librería que realiza peticiones HTTP basándose en promesas. Con ella podemos hacer peticiones al servidor y recibir respuestas de forma más sencilla que con un *fetch* tradicional.

2.3.4. Service Workers

Los Service Workers son *scripts* de JavaScript que se ejecutan aún cuando la aplicación cliente no está activa. Vienen a solucionar la pérdida de conectividad que sufren las aplicaciones web. Con ellos las webs pueden realizar tareas en segundo plano y sin conexión a internet.

Debido a que la actividad principal de la aplicación es el registro de *tracking* GPS y que se quiere simular el comportamiento de una aplicación nativa que realiza esta función, el uso de los *service workers* es crucial para registrar las coordenadas mientras la aplicación está inactiva. Al ser tan importantes, en el capítulo 3 se explican detalladamente.

2.3.5. Mapbox

Mapbox [35] es un proveedor de mapas de código abierto. Cuenta con muchos productos y servicios para la creación de mapas y el desarrollo de aplicaciones en torno a ellos.

2.3.6. Leaflet

Leaflet [36] es una librería de código abierto de JavaScript, se utiliza para la creación y gestión de mapas interactivos. Se ha elegido debido a su simplicidad y a que se integra muy bien con Mapbox.

Capítulo 3

Desarrollo de la aplicación

En este capítulo se detalla el desarrollo del proyecto. Se comentan las configuraciones utilizadas para el entorno de desarrollo y el despliegue continuo, las implementaciones de las aplicaciones de *backend* y *frontend* y el modelo de datos definido.

3.1. Configuración del entorno

Antes de empezar a codificar fue necesario preparar el entorno de desarrollo. Para ello, se creó un repositorio de GitHub [37] donde alojar el proyecto en la nube. Una vez creado, se le añadió un tablero *kanban* automático que facilitase la gestión y la planificación de las tareas. A continuación, se incluye un extracto de cuando todavía se estaba desarrollando la aplicación:

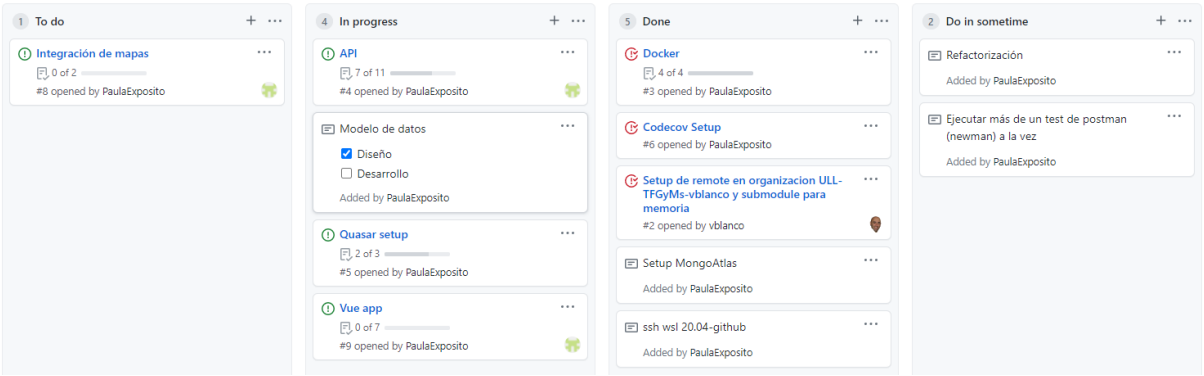


Figura 3.1: Extracto del tablero *kanban* para la gestión del proyecto.

La automatización de las tareas viene dada gracias a los Issues de GitHub. A cada *issue* se le asigna una tarjeta del tablero *kanban*. Entonces, cada vez que un *issue* se abre, cierra o reabre, la tarjeta asignada se mueve de una columna a otra automáticamente. En la Figura 3.2 se incluyen algunos de los *issues* utilizados.

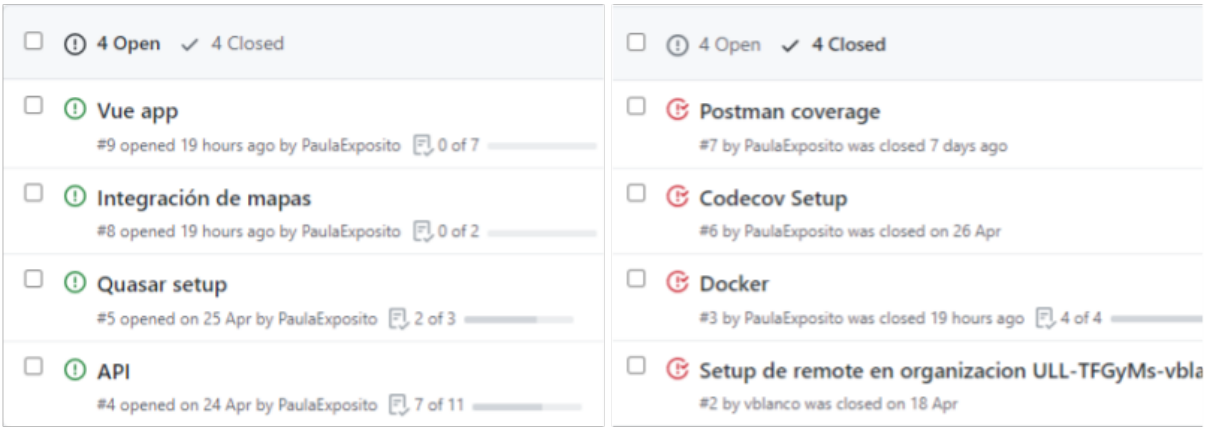


Figura 3.2: Issues del proyecto.

Durante esta etapa inicial fue también cuando se creó el clúster en Mongo Atlas para almacenar los datos generados por la aplicación. Para ello se eligió un servidor AWS situado en Irlanda.

Además, se habilitó Codecov para el repositorio. Debido a que el proyecto está compuesto por dos aplicaciones hubo que crear dos *flags* para poder subir a la misma instancia los datos recolectados por el *backend* y por el *frontend*. Codecov calcula el porcentaje total de la cobertura conseguida teniendo en cuenta el tamaño de cada aplicación.

Para actualizar estos datos se hace uso de la integración continua mediante un flujo de trabajo de GitHub Actions. Después de ejecutar los tests, ya sean los del *backend* o los del *frontend*, se envía la cobertura generada a Codecov. La acción utilizada se encuentra en el Listado 3.1.

3.2. Modelo de datos

También antes de empezar a programar, fue necesario pensar en el diseño del modelo de datos. Teniendo en cuenta las funcionalidades que se quería que tuviese la aplicación, se determinó que habría tres modelos que serían accesibles desde la aplicación cliente mediante una API CRUD.

Los modelos definidos fueron “User” para los datos referidos a los usuarios, “Trail” para los datos de las rutas y “Event” para los eventos. En la Figura 3.3 se encuentra el diseño preliminar de los *endpoints* para estos modelos.

```yaml
name: Frontend tests Coverage to Codecov
on:
  push:
    branches: [ main ]
defaults:
  run:
    working-directory: frontend
jobs:
  build:
    steps:
        [...]
      - name: Install dependencies
        run: |
          npm install -g @quasar/cli
          npm ci
      - name: Collect coverage report
        run: npm run test:unit:coverage
      - name: Upload coverage
        run: bash <(curl -s https://codecov.io/bash) -t ${{ secrets.
           CODECOV_TOKEN }} -c -F frontend -n frontend
```

Listado 3.1: Acción para la actualización de datos en Codecov.

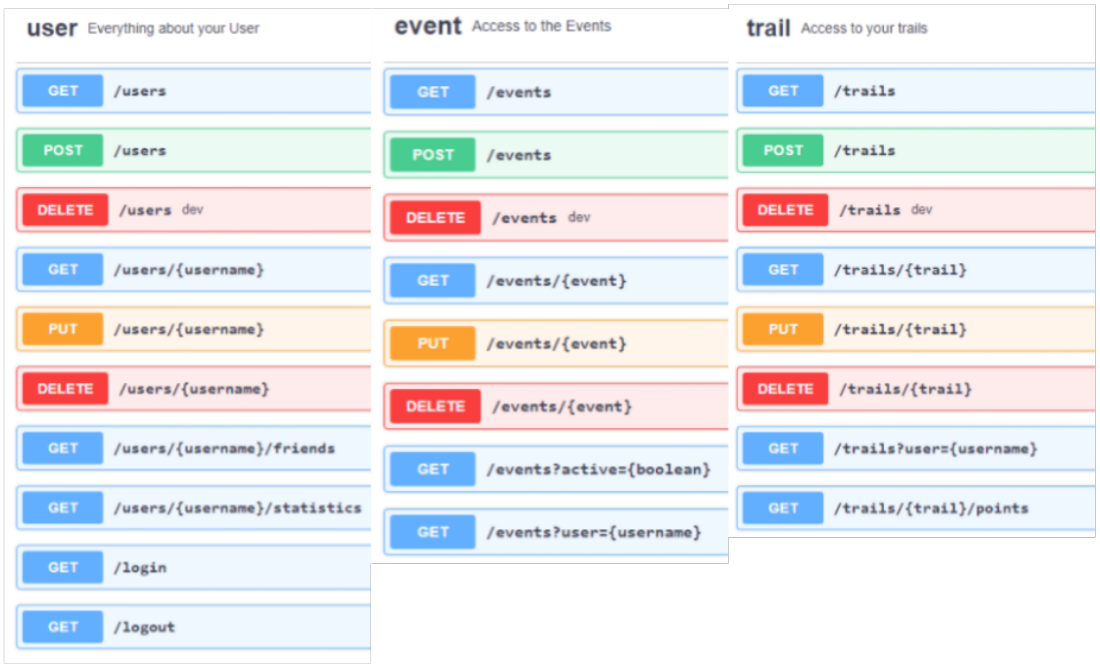


Figura 3.3: Diseño del modelo de datos.

3.3. Servidor

Tras haber definido el modelo de datos se empezó el desarrollo del servidor. Para tener una buena estructura que facilitase el desarrollo y la comprensión del código, se decidió utilizar un diseño basado en la arquitectura de 3 capas. En los siguientes apartados se detalla cómo esto se llevó a cabo, cómo se desarrolló y testeó la API y cómo se realiza la gestión de la sesión mediante los JWT.

3.3.1. Arquitectura

La arquitectura utilizada se caracteriza porque está formada por tres capas con una responsabilidad única y, además, cada una sólo puede comunicarse con las capas que están inmediatamente al lado suyo. Puede verse en la Figura 3.4.

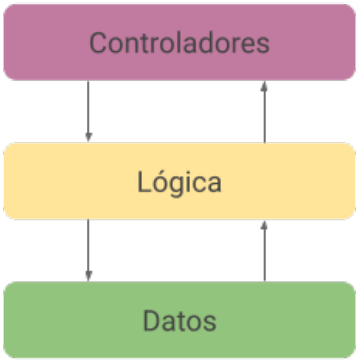


Figura 3.4: Arquitectura en 3 capas.

La primera es la capa de los controladores, también llamada capa de presentación. Es la única con la que pueden comunicarse los usuarios del servidor. En este caso está compuesta por los *endpoints* de la API, se incluye el código correspondiente al *endpoint* de registro de usuario en el Listado 3.2. Como puede verse, el controlador se limita a recibir la petición del usuario, mandarle al servicio de autenticación los datos y, dependiendo del código de la respuesta, responderle al usuario si pudo crearse la cuenta o no.

La siguiente es la capa de servicios, en esta capa se encuentra todo el código que constituye la lógica de negocios de la aplicación. No debe contener objetos *req* o *res* que corresponden al usuario. Por ejemplo, el servicio correspondiente al registro de usuarios contiene la lógica necesaria para crear un usuario; comprueba si los datos recibidos son correctos, hace una consulta a la base de datos para saber si el usuario ya existe, si no existe crea un *token* que devuelve a la capa de controladores, y si sí existe se devuelve un código de error. Se trata de un método con una funcionalidad específica que puede llamarse desde cualquier lugar de la aplicación.

Por último, la capa de datos es la que contiene la definición del esquema de

```
router
    .post('/signup', async (ctx, next) => {
        const res = await authService.signup(ctx.request.body);
        if (res == -1) {
            ctx.response.body = { msg: "Must pass user and password"
               };
            ctx.response.status = 409;
        }
        else if (res == -2) {
            ctx.response.body = { msg: "User already exists" };
            ctx.response.status = 409;
        }
        else {
            ctx.response.body = res;
            ctx.response.status = 201;
        }
        await next();
    })
```

Listado 3.2: Extracto de código de la capa de controladores.

datos utilizado. Se utiliza la librería Mongoose [38] para conectar el esquema a la instancia de MongoDB. En el Listado 3.4 se incluye el esquema definido para el modelo "User".

Tras aplicar los conceptos de la arquitectura de 3 capas, el *scaffolding* del *backend* quedó como muestra el Listado 3.5. Donde el directorio `/api` contiene los *endpoints* y los *middlewares* correspondientes a la capa de controladores. `/service` contiene el código correspondiente a la lógica de la aplicación. Y `/models` contiene las deficiones de los esquemas de datos. El resto de directorios corresponden a otras prácticas utilizadas que se explican en siguiente apartado.

3.3.2. Koa API y Testing

Como se comentó en el Capítulo 2, se eligió Koa como *framework* para el desarrollo de la API. Para ponerlo en marcha se decidió encapsularlo en un cargador, al igual que Mongoose, de esta forma el proceso de inicio de la aplicación se divide en módulos testeables. Esto corresponde al directorio `/loaders` del Listado 3.6.

En el archivo `koa.mjs` se incluyen los *middlewares* utilizados como `@koa/cors` y `koa-bodyparser`, que no se incluyen en el núcleo del *framework*. Se utilizan cuatro tipos de rutas diferentes que son: las rutas de autenticación, las rutas de usuarios (con el prefijo `/users`), las rutas de recorridos (`/trails`) y las rutas de eventos (`/events`). Estas rutas permiten crear, actualizar, leer y eliminar datos a partir de

```
async function signup(userDTO) {
    // Comprobar que se pasan el usuario y la contraseña en el DTO
    if (userDTO.username == null || userDTO.password == null)
        return -1;

    // Comprobar que el usuario no existe
    const userExists = await User.findOne({ "username": userDTO.
       username });
    if (userExists != null)
        return -2;

    const user = new User(userDTO);

    // Cifrar la contraseña
    user.password = await user.encryptPass(user.password);

    // Crear token
    const payload = { "_id": user._id };
    const token = jwt.sign(payload, secret.jwt, { expiresIn: 60 * 60
       * 24 });

    await user.save();
    return { user, token };
}
```

Listado 3.3: Extracto de código de la capa de lógica.

```
const User = new mongoose.Schema({
    username: { required: true, type: String },
    firstName: String,
    lastName: String,
    email: String,
    password: { required: true, type: String },
    phone: String,
    statistics: {
        distance: Number,
        gradient: Number,
        time: Number,
        numberOfRegisters: Number
    },
    friends: Array,
    token: String,
});
```

Listado 3.4: Extracto de código de la capa de datos.

```
.
|-- Dockerfile
|-- Procfile
|-- api
|   |-- middlewares
|-- app.mjs
|-- config
|-- loaders
|-- models
|-- package-lock.json
|-- package.json
|-- run.sh
|-- services
|-- test
    |-- jest
    |-- postman
```

Listado 3.5: Backend scaffolding.

```
.
|-- loaders
    |-- index.mjs
    |-- koa.mjs
    |-- mongoose.mjs
```

Listado 3.6: Directorio `/loaders`.

```
var data = JSON.parse(responseBody);
pm.environment.set("token", data.token);

pm.test("User created successfully with a 200 code", () => {
    pm.response.to.have.status(201);
});

pm.test("Check that it returns an object", () => {
    let jsonData = pm.response.json();
    pm.expect(jsonData).to.be.an("object");
});

pm.test("Confirm response message", () => {
    let jsonData = pm.response.json();
    pm.expect(jsonData.user.username).to.eql("amaia");
});

pm.test("Check that token is generated", () => {
    let jsonData = pm.response.json();
    pm.expect(jsonData.token).not.to.be.eq(null);
});
```

Listado 3.7: Extracto de tests en Postman.

una serie de *endpoints*.

La API se ha testeado haciendo uso de tests End-To-End (E2E) con Postman, mediante la creación de colecciones que se ejecutan desde la línea de comandos con la herramienta Newman CLI. Se incluye un extracto, Listado 3.7, del *testing* realizado al *endpoint* de registro de usuario cuando la respuesta es satisfactoria.

También se han realizado tests unitarios con Jest. Estos tests son los que contribuyen a la cobertura que recoge Codecov. En el Listado 3.8 se incluye un ejemplo de los realizados a la lógica de *logout*. Todos los tests se encuentran en el directorio `/test`.

3.3.3. JWT

Los JSON Web Tokens se utilizan para llevar a cabo el control de la sesión de la aplicación. Su funcionamiento se basa en un *token* creado en el servidor cuando el usuario se identifica. A partir de ese momento, cada vez que el usuario quiera acceder a información que requiera cierto nivel de privilegios tiene que enviar el *token* en la cabecera de la petición.

Estos *tokens* se caracterizan porque están formados por tres componentes. Una

```
const rightTestDTO = { username: "tester", password: "tester123" };
const wrongTestDTO_pass = { password: "tester123" };

test('Logout successfully', async () => {
    const result = await auth.logout(rightTestDTO).then(data => {
       return data; });
    expect(auth.logout(rightTestDTO)).toBeInstanceOf(Object);
    expect(result.token).toBe(null);
});

test('Logout fails', async () => {
    const result = await auth.logout(wrongTestDTO_pass).then(data =>
       { return data; });
    expect(result).toBe(-1);
});
```

Listado 3.8: Extracto de tests con Jest.

cabecera que contiene el algoritmo de *hash* y el tipo de *token* utilizados, el *payload* que contiene la información que se transmite, y la firma, que está encriptada y depende de una clave secreta. Puede verse en la Figura 3.5.

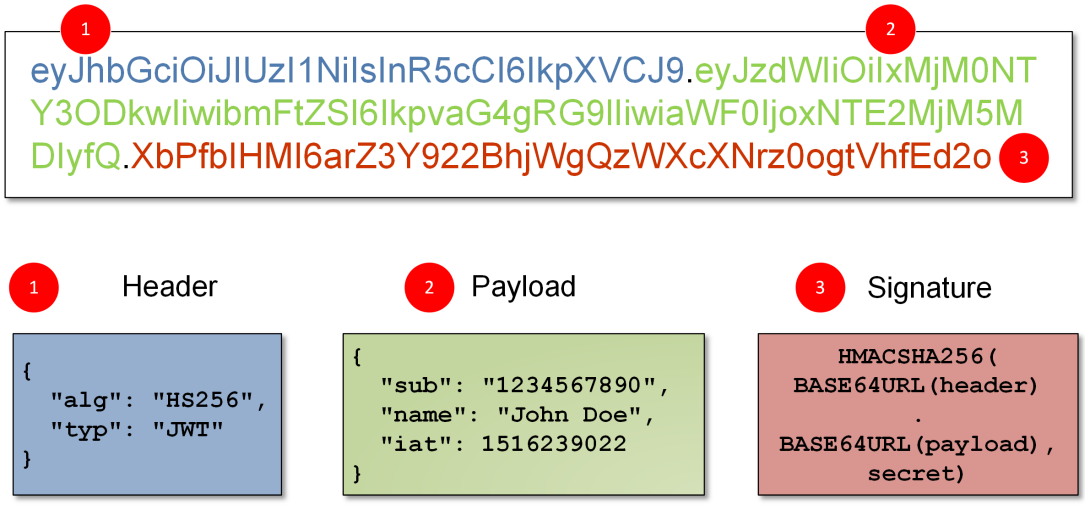


Figura 3.5: Estructura de un *token* JWT.

En la aplicación se ha incluido un *middleware* para comprobar que el *token* es válido. Se ha programado para que el *payload* dependa del id del usuario, ya que es único para cada uno.

3.4. Aplicación cliente

La aplicación cliente se ha desarrollado en Vue con el *framework* Quasar. En este apartado se describe el proceso que se ha seguido para ello. Se comentará la estructura de la aplicación, explicando los conceptos principales como los *service workers* y la integración de los mapas, además de mostrar los *mockups* iniciales.

3.4.1. Prototipo

Antes de empezar a codificar las interfaces se realizaron una serie de *mockups* para tener claro qué es lo que se iba a desarrollar. Aunque existen muchas herramientas para realizar prototipos, para este proyecto se hicieron en papel, al no necesitar la colaboración de otras personas. Los principales se muestran en la Figura 3.6.

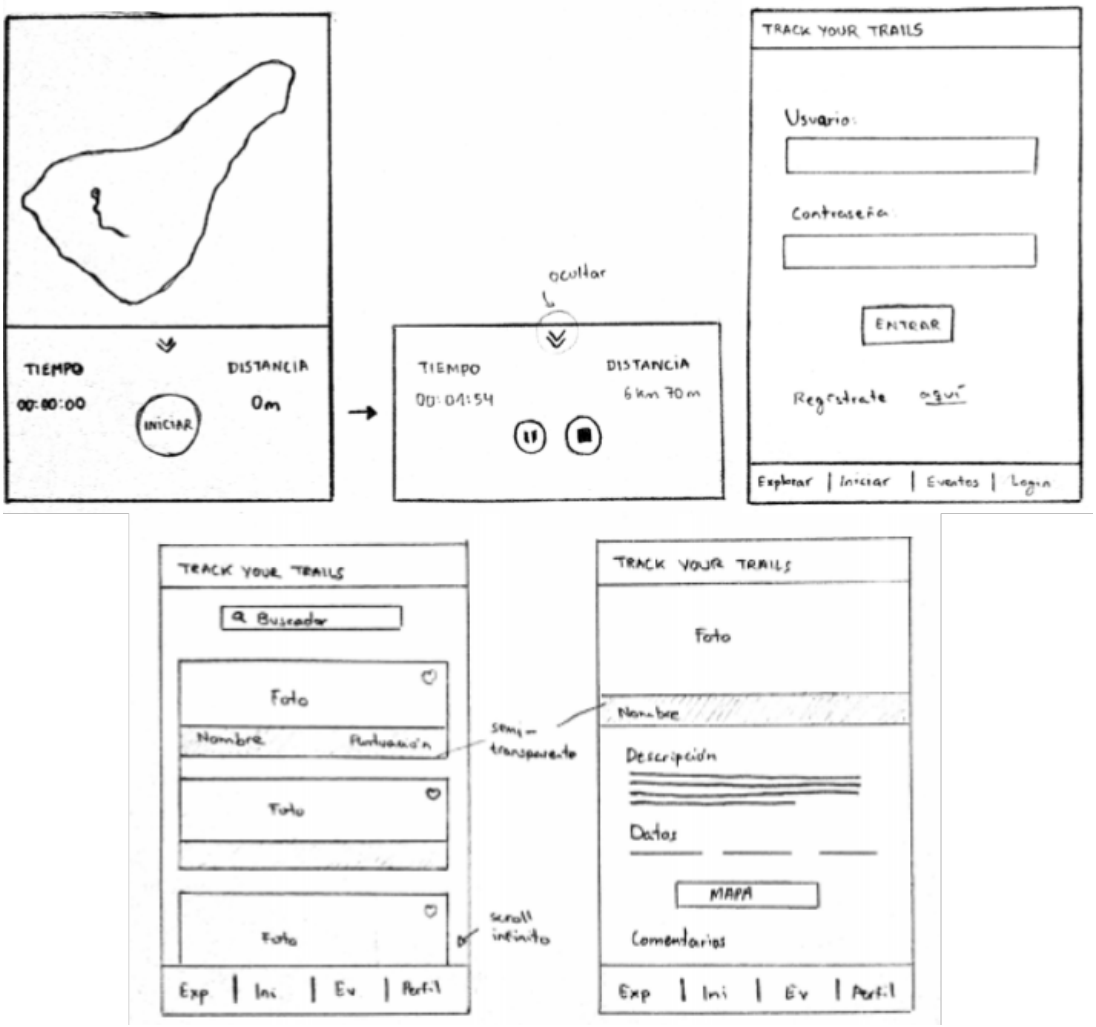


Figura 3.6: Mockups.

3.4.2. Quasar

Para facilitar la creación y gestión de las dependencias se utilizó Quasar CLI, por lo que la estructura del proyecto se creó de forma automática. El *scaffolding* del *frontend* es el que se muestra en el Listado 3.9.

Debido a la gran cantidad de archivos y directorios generados sólo se explican los principales, que están más relacionados con la implementación.

- `/src/boot/axios.js`: este archivo crea una instancia de Axios con la URL base del servidor para hacer las peticiones a la API.
- `/src/components`: en este directorio se encuentran las componentes de Vue, que son las unidades de código independiente que pueden reutilizarse.
- `/src/pages`: este directorio corresponde al de las vistas de las diferentes pantallas que se utilizan en una Single Page Application (SPA). Suelen estar formadas por varias componentes.
- `/src/router`: en este directorio se encuentra el código relacionado con el Vue Router, como las rutas definidas para la SPA.
- `/src/store`: aquí se encuentra el código del *store* de Vuex, es decir, los estados, las acciones, las mutaciones y los *getters*. Por ejemplo, se ha creado un módulo de autenticación donde se almacenan los datos relacionados con los usuarios, como el *token* de acceso. En el Listado 3.10 se ve la estructura de este directorio en detalle.
- `/test`: aquí se incluyen los tests realizados a la aplicación de Vue. Se ha intentado seguir la filosofía TDD, *Test-Driven-Development*, para el desarrollo de las pruebas. Esta práctica consiste en implementar el test antes de crear el código. Así, la primera vez que se ejecuta el programa el test debe fallar (*red stage*), luego se debe conseguir que el test pase con la mínima cantidad de código posible (*green stage*). Y por último, se refactoriza el código (*refactor stage*). Aunque finalmente, debido a dificultades a la hora de la realización de los tests la mayoría se realizaron a posteriori, acercándose más al *unit testing*.

3.4.3. Service Workers

Los *Service Workers* son fundamentales para el correcto funcionamiento de la aplicación, debido a que gracias a ellos las aplicaciones web pueden trabajar en segundo plano, requisito necesario para conseguir que la aplicación pueda registrar *tracking* correctamente. Por ejemplo, cuando se utiliza la aplicación en un móvil que está bloqueado, interesa que se sigan registrando las coordenadas GPS, pero,

```
.
|-- Dockerfile
|-- README.md
|-- babel.config.js
|-- dist
|-- jest.config.js
|-- jsconfig.json
|-- package-lock.json
|-- package.json
|-- public
|-- quasar.conf.js
|-- quasar.extensions.json
|-- quasar.testing.json
|-- run.sh
|-- src
|   |-- App.vue
|   |-- assets
|   |-- boot
|   |   |-- axios.js
|   |   |-- utils.js
|   |-- components
|   |-- css
|   |-- index.template.html
|   |-- layouts
|   |-- pages
|   |-- router
|   |   |-- index.js
|   |   |-- routes.js
|   |--  store
|       |-- index.js
|       |-- module-auth
|           |-- actions.js
|           |-- getters.js
|           |-- index.js
|           |-- mutations.js
|           |-- state.js
|-- static.json
|-- test
```

Listado 3.9: Frontend scaffolding.

```
.
|-- store
    |-- index.html
    |-- index.js.html
    |-- module-auth
        |-- actions.js
        |-- getters.js
        |-- index.js.
        |-- mutations.js.
        |-- state.js.
```

Listado 3.10: Vue store scaffolding.

al estar bloqueado la aplicación está inactiva. Por ese motivo en este apartado se detalla su implementación.

El ciclo de vida de un *service worker*, [39], está formado por el registro, la instalación y la activación del mismo. Para ponerlo en marcha se utilizó de nuevo Quasar CLI, que genera automáticamente la estructura de uno al añadir el modo Progressive Web App (PWA) a la aplicación.

Los *service workers* funcionan como un *proxy* entre la aplicación y el servidor, por lo que, al estar activo, intercepta todas las peticiones que se hacen al servidor, pudiendo además modificar las respuestas de este. En la Figura 3.7 se muestra un esquema de este flujo. Esta es la característica que se aprovecha en la aplicación, la idea es que cuando se detecte que no se pudo guardar una coordenada en el servidor porque no se pudo conectar con él, el *service worker* guarde los datos en la caché.

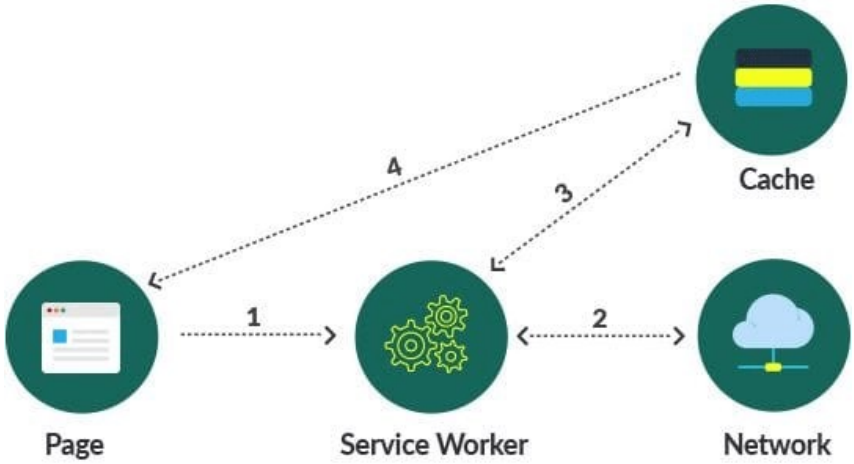


Figura 3.7: Funcionamiento de una aplicación con un *Service Worker.*

Pero esta tecnología presenta un problema, únicamente puede almacenar en la caché las peticiones *GET*, y para la aplicación las que más interesan son las

peticiones *POST*, que son las que envían los datos de las coordenadas al servidor. Por tanto, hubo que buscar una solución alternativa. Esta solución se centra en guardar de forma local las peticiones hasta que el usuario vuelva a tener conexión. Los pasos seguidos fueron los siguientes:

1. Cuando se detecta que una petición *POST* falla debido a la falta de conexión, se manda un aviso al cliente.
2. Una vez hecho esto, se guardan los datos de la petición en el *local storage* temporalmente. Se utiliza el *local storage* debido a que es más fácil de utilizar que IndexedDB o CouchDB.
3. Tras guardar los datos sólo hay que esperar a volver a tener conexión. Esto se detecta con *Background Sync*, un servicio que envía un evento al *service worker* cuando vuelve.
4. Por último, cuando ya ha vuelto la conexión se vuelve a realizar el *POST* y se elimina la entrada de los datos del *local storage*.

3.4.4. Integración de los mapas

Para la integración de los mapas se ha utilizado la librería Leaflet de JavaScript, que permite crear mapas interactivos. Gracias a ella, por ejemplo, puede hacerse zoom en los mapas de la aplicación y modificarse dinámicamente los puntos que se marcan.

El mapa que se utiliza proviene de Mapbox, en este caso se eligió uno predefinido, el *Mapbox Outdoors [40]*. Este mapa se ha diseñado específicamente para mostrar rutas y caminos fuera de las carreteras, el desnivel del terreno y puntos de interés, por lo que cubre con creces las necesidades de la aplicación.

3.5. Despliegue continuo

Para el despliegue de la aplicación se utilizan contenedores de Docker. Se han creado dos imágenes Node Alpine que son más ligeras que las distribuciones basadas en Node, una para el servidor y otra para la aplicación de Vue. En los Listados 3.11 y 3.12 se incluyen los *dockerfiles* del *backend* y del *frontend*, respectivamente.

Para facilitar el despliegue en local se hizo uso de Docker Compose [41]. Mientras que para el despliegue en producción se utiliza Heroku Container Register, aunque los contenedores también se han registrado en GitHub Packages.

De nuevo, se hace uso de los flujos de GitHub Actions, en este caso para la automatización de los despliegues. Estos se realizan cada vez que se hace una

```
FROM node:15.13.0-alpine
STOPSIGNAL SIGINT
EXPOSE 3000
WORKDIR /app

COPY package.json .
RUN npm install
COPY . .

CMD ["node", "app.mjs"]
```

Listado 3.11: Backend Dockerfile.

```
FROM node:14.16.0-alpine
STOPSIGNAL SIGINT
EXPOSE 8080
WORKDIR /app

RUN npm install -g @quasar/cli
COPY package.json .
RUN npm install
COPY . .

RUN quasar build
CMD ["quasar", "dev"]
```

Listado 3.12: Frontend Dockerfile.

```yaml
name: Deploy backend to Heroku
on:
  push:
    branches: [release]
defaults:
  run:
    working-directory: backend
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v1
    - name: Login to Heroku Container registry
      env:
        HEROKU_API_KEY: ${{ secrets.HEROKU_API_KEY }}
      run: heroku container:login
    - name: Build and push
      env:
        HEROKU_API_KEY: ${{ secrets.HEROKU_API_KEY }}
      run: heroku container:push -a ${{ secrets.
         HEROKU_APP_BACKEND_NAME }} web
    - name: Release
      env:
        HEROKU_API_KEY: ${{ secrets.HEROKU_API_KEY }}
      run: heroku container:release -a ${{ secrets.
         HEROKU_APP_BACKEND_NAME }} web
```

Listado 3.13: Deploy backend container in Heroku Action.

release. La acción definida para el despliegue del *backend* se encuentra en el Listado 3.13.

Mientras que para el despliegue del contenedor de *frontend*, al tratarse de una aplicación estática, hizo falta utilizar dos *buildpacks* de Heroku, además de una acción similar a la del *backend*. Se utilizaron los *buildpacks* que se incluyen en el Listado 3.14.

Los contenedores de *backend* y *frontend* están alojados, respectivamente, en:

```
1 heroku buildpacks:add heroku/nodejs -a trackyourtrails
2 heroku buildpacks:add https://github.com/heroku/heroku-buildpack-
    static -a trackyourtrails
```

Listado 3.14: Buildpacks utilizados para desplegar frontend app.

· https://tytbackend.herokuapp.com

· https://trackyourtrails.herokuapp.com

Capítulo 4

Descripción de la aplicación

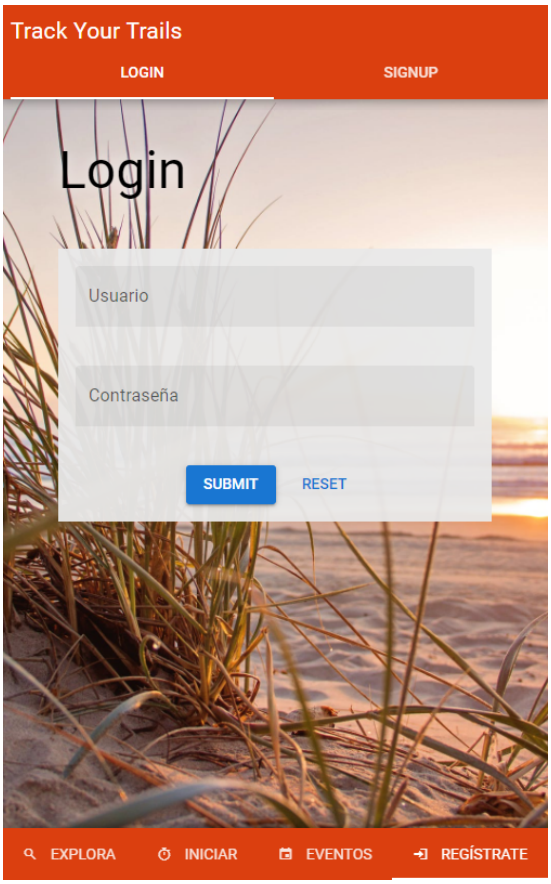
En este capítulo se incluye la descripción y la interfaz de la aplicación desarrollada, además de una explicación sobre el funcionamiento de la misma.

Las aplicaciones web, al ejecutarse desde los navegadores, pueden utilizarse en muchos dispositivos diferentes. Por ello, deben diseñarse de forma que resulten usables en todos. En este caso se ha utilizado la técnica *#mobilefirst* que consiste, como indica su nombre, en realizar primero el diseño para móviles y luego adaptarlo a tablets y sistemas de escritorio. Se ha elegido este diseño debido a que *TrackYourTrails*, la aplicación desarrollada, es una aplicación centrada en la captura GPS, por lo que preferentemente se utilizará desde dispositivos móviles.

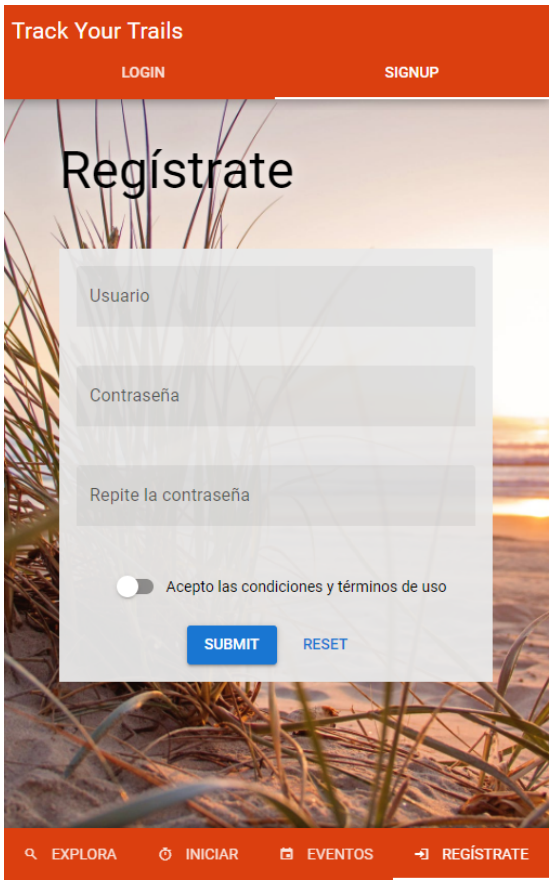
En los siguientes apartados se explica el funcionamiento de las principales vistas de la aplicación. Como se puede observar en la Figura 4.1, las pantallas entre las que se puede mover el usuario contienen una cabecera con el nombre de la aplicación y un menú inferior con cuatro opciones “Explora”, “Iniciar”, “Eventos” y “Regístrate”. El menú se ha puesto en la parte baja para simular el comportamiento de una aplicación móvil, aunque en las webs no sea habitual esta estructura.

4.1. Regístrate

En este apartado se incluyen las interfaces con las que un usuario puede registrarse en el sistema, y una vez registrado, iniciar sesión para poder guardar sus rutas. En la parte superior de la vista hay un submenú para poder elegir si se quiere realizar el inicio de sesión (*login*), Figura 4.1(a), o el registro (*signup*), Figura 4.1(b). Cuando un usuario se ha autenticado satisfactoriamente, la opción “Regístrate” del menú pasa a ser “Perfil”, donde el usuario puede ver y modificar sus datos.



(a) Vista Login.



(b) Vista SignUp.

Figura 4.1: Vistas de Track Your Trails.

4.2. Explora

En esta sección se muestran las rutas que los usuarios de la aplicación han guardado, también es la pantalla que se ve al abrir la aplicación. Tiene una vista donde se muestra una carta representativa de cada ruta con una imagen, el nombre y algunos datos de interés, Figura 4.2(a). Cuando se pulsa sobre una de estas tarjetas se expande la información de la ruta, mostrando una descripción, los datos más relevantes como la duración y la longitud, la ruta registrada en un mapa y una sección de comentarios, Figura 4.2(b).

4.3. Inicia

Esta es la vista principal de la aplicación. Es aquí desde donde se realiza el registro de *tracking*. Puede verse en la Figura 4.3(a). Consiste en un mapa donde se muestra el recorrido y la ubicación actual del usuario. Además, también hay un panel de control donde se muestra la distancia que se ha recorrido, el tiempo empleado y los botones para iniciar y terminar el recorrido.

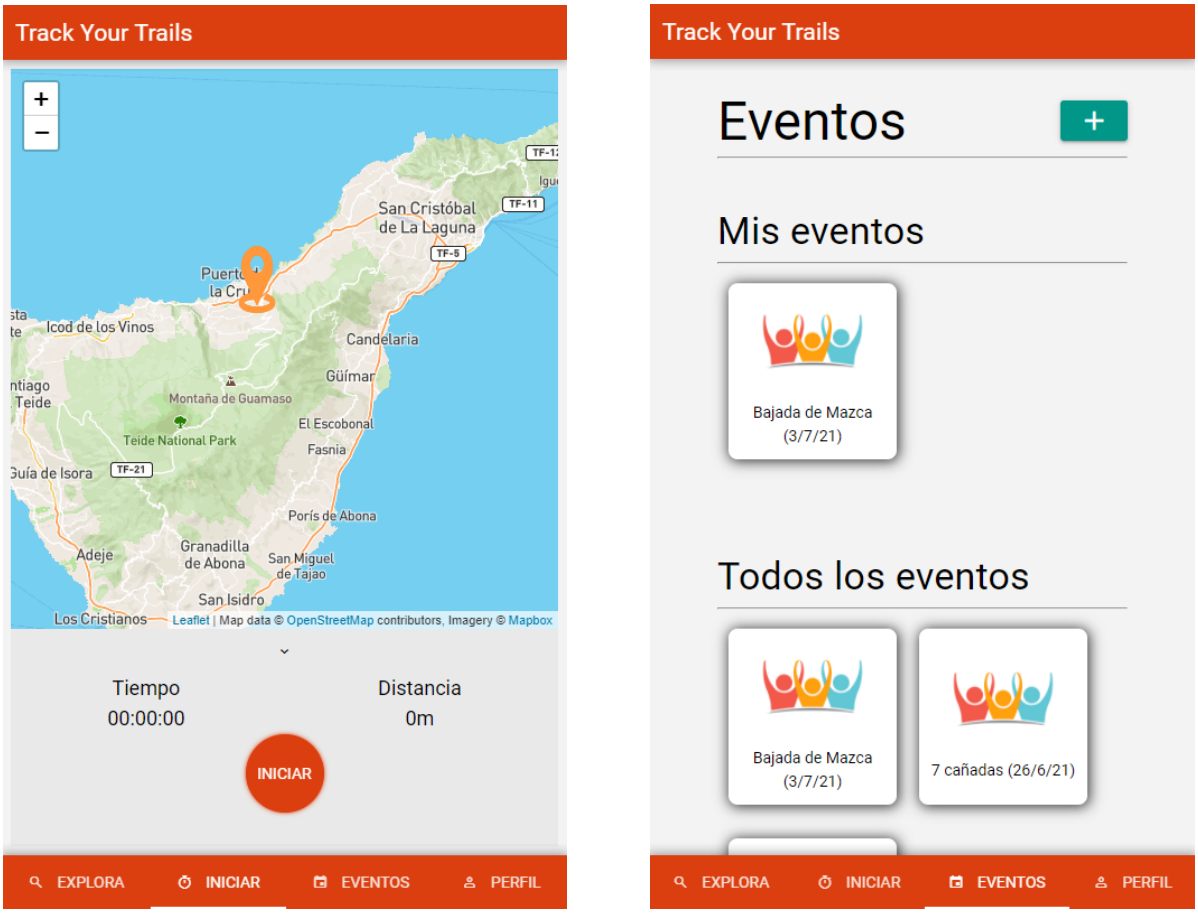
(a) Vista Explore.

(b) Vista Trail.

Figura 4.2: Vistas de Track Your Trails.

4.4. Eventos

La funcionalidad de eventos consiste en un espacio donde los usuarios pueden crear ítems de caminatas futuras que van a realizar, a los que otros usuarios pueden suscribirse. En esta vista se ven los diferentes eventos que hay disponibles. Si el usuario está logueado, en primer lugar le aparecerán los eventos a los que se ha suscrito y luego todos los existentes. Además, desde aquí también se puede crear un nuevo evento mediante un formulario. Al igual que en la sección “Explora”, al pulsar sobre las tarjetas la información de los eventos se expande. Esta pantalla se incluye en la Figura 4.3(b).



(a) Vista Tracker.

(b) Vista Events.

Figura 4.3: Vistas de Track Your Trails.

Capítulo 5

Conclusiones y líneas futuras

En este capítulo se exponen las conclusiones a las que se ha llegado durante el desarrollo del proyecto, así como las líneas de trabajo futuras que podrían seguirse.

5.1. Conclusiones

Tras finalizar la parte práctica del proyecto, se ha obtenido una aplicación web capaz de obtener la ubicación de un usuario, almacenarla y mostrarla en un mapa, igualando las funciones de una aplicación nativa en cuanto a *tracking*. Al tratarse de una aplicación web se facilita el despliegue y la portabilidad entre dispositivos. Puede utilizarse tanto en teléfonos móviles, y, de hecho, se ha diseñado para ello, como en sistemas de escritorio y en *tablets*. Además, el usuario no tiene que realizar actualizaciones, a diferencia de con las aplicaciones nativas. La aplicación también incluye las funcionalidades de buscar nuevas rutas y la de crear y suscribirse a eventos.

Para la realización del proyecto se ha hecho uso de muchas tecnologías y herramientas complejas que cuentan con una gran cantidad de funcionalidades, las cuales han tenido que combinarse para poder lograr el resultado final. Debido a esto y a que muchas de las tecnologías se utilizaban por primera vez, el proyecto ha resultado más una actividad académica en la que se ha primado la formación en relación a estas tecnologías y a estudiar cómo combinarlas, que al desarrollo de una aplicación preparada para ser utilizada por usuarios reales.

Aunque es cierto que muchas de las herramientas utilizadas, como Docker y GitHub, no se han exprimido al máximo, sí que se ha llegado a un punto donde se consiguió que funcionasen correctamente mientras se hacía uso de algunas buenas prácticas recomendadas, como por ejemplo, la automatización de los despliegues.

En relación a la implementación, por parte del *backend*, basado en NodeJS

y Koa, se consiguió desarrollar una API CRUD completa con manejo de errores, probada con tests unitarios y tests funcionales. Al haberse desarrollado utilizando los principios de la arquitectura de tres capas, finalmente se ha obtenido un código fácil de escalar, organizado y limpio. Mientras que en el *frontend*, que se ha implementado utilizando VueJS, se han tenido más dificultades. El código se estructuró según el convenio de Quasar, separando las vistas y componentes visuales en pequeños módulos con funcionalidades específicas, logrando un código legible y modular. El principal problema que hubo fue la realización de los tests, debido a que la librería de *testing* de Vue es bastante nueva y no hay mucha documentación sobre ella.

Aún así, y sabiendo que hay mucho margen de mejora, se considera que el proyecto ha llegado a un punto satisfactorio y que ha servido para ampliar conocimientos en el ámbito de las tecnologías web, además de como introducción al desarrollo de aplicaciones escalables y con despliegues automatizados.

5.2. Líneas futuras

A pesar de haber obtenido una aplicación de registro de *tracking* funcional, hay varias mejoras que pueden incluirse de seguir trabajando en la aplicación. En primer lugar, sería interesante estudiar en profundidad el rendimiento de los *Service Workers* para el registro de *tracking* en comparación con el de una aplicación nativa. Relacionado también con los *Service Workers*, podría seguirse la filosofía *#offlinefirst*, que de forma similar al *#mobilefirst*, se centra en el desarrollo de aplicaciones que funcionen sin conexión a internet, una buena forma de introducirlo sería utilizando CouchDB como base de datos para almacenar los datos que actualmente se guardan en el *local storage*.

Por otro lado, si se llegase a considerar un uso real por parte de los usuarios, habría que mejorar la usabilidad y la accesibilidad. Además de que podrían seguirse líneas más especializadas relacionadas con la salud o el deporte, como trabajar con el desnivel recorrido, el cálculo de calorías, la velocidad y demás conceptos que suelen incluirse en aplicaciones de este tipo. Otra funcionalidad importante a incluir es el desarrollo de un sistema para la socialización privada entre usuarios, debido a que a estos les gusta estar conectados con sus conocidos y no es algo que las aplicaciones de este tipo suelan incluir. También, sería conveniente incluir la compatibilidad con otros estándares de datos GPS, junto con la funcionalidad de descarga y subida de archivos contenedores de recorridos.

Capítulo 6

Summary and Conclusions

This chapter includes the main conclusions of the project and the future work that can be done.

6.1. Conclusions

After finishing the practical part of the project, a web application has been developed which captures, saves and shows the user's location on a map. The implemented functionalities are similar to those available in native tracking applications. The fact we have a web application makes the deployment and portability easier to manage. The developed app has been designed for mobile phones, but it can be used in tablets and desktop systems too. Moreover, users do not have to do updates, in contrast with native applications. The application also includes other functionalities such as searching for new routes and creating and subscribing to events.

Many complex technologies and tools, with lots of features, have been used for project development. They had to come together to get the final result. As it was the first time I used most of these tools, the project resulted in an academic activity where I learnt how to use and combine them. Therefore, the application is not ready to be used by real users.

Even if many of the used tools, like Docker or GitHub, have not been used to their full capabilities, a point has been reached where they worked successfully while some good practices were used. For example, the implementation of continuous deployment practices.

Related to the implementation, in the backend, which is based on NodeJS and Koa, a complete API CRUD with handling errors has been developed. It was tested with unit and functional tests. As it was developed using the three-tier architecture principles, the resulting code is clean, easy to scale and organize. On the

other hand, some difficulties have been found in the frontend, which has been developed with Vue. According to the Quasar convention, the code was organised separating the visual views and components into small units to obtain legible and modular code. The main problem was the implementation of the tests because Vue's testing library is quite new and there is not much information about it.

Knowing that there is a margin for improvement, it is considered that the project has reached at a satisfactory point. It has been useful to improve my knowledge in web technologies and as well as an introduction to the development of automatically deployed and scalable applications.

6.2. Future work

Even though a functional tracking registering app has been reached, some improvements can be included in the project. First of all, it would be interesting to study the efficiency of Service Workers compared to a native application. *#offlinefirst* philosophy can also be used. This concept is based on the development of applications that work without an internet connection. CouchDB is a good option for introducing *#offlinefirst*, for example, for saving the data in which up to now has been saved in the local storage.

Furthermore, if the application is to be used by real users, it would be necessary to improve usability and accessibility. Additionally, more specific lines such as sports and health ideas can be added. For example, working with trail gradients, calorie calculations, velocity and other concepts that are usually included in this type of applications. Another important functionality to be included is a system that allows private interactions between the users. This is interesting because only a few apps of tracking registering include it. It would also be convenient to add compatibility with other GPS standards, together with import and export trail files.

Capítulo 7

Presupuesto

En este capítulo se presenta el presupuesto estimado que conlleva el desarrollo del proyecto y el despliegue durante cuatro meses. El total incluye el coste de los recursos humanos y el de los servicios en la nube a los que es necesario suscribirse.

Los recursos humanos utilizados solamente incluyen un desarrollador FullStack. Mientras que los servicios en la nube necesarios son: una cuenta de GitHub Pro para tener 2GB de almacenamiento disponibles para los contenedores de Docker. Dos Heroku Dyno Hobby para el despliegue de los contenedores en Heroku, se necesitan para que la aplicación no se quede inactiva en ningún momento. Y un clúster M0 AWS de Mongo Atlas para el almacenamiento en la nube. Al tratarse de una aplicación pequeña y no comercial no es necesario contratar mucho almacenamiento ni capacidad de despliegue. Si se escalase habría que contratar planes superiores.

En la tabla 7.1. se incluye el coste de un mes de desarrollo y el coste total de los cuatro meses de duración de la asignatura Trabajo Fin de Grado.

Tipo	Descripción	Coste	Coste total
Recursos humanos	1 desarrollador FullStack	1200€	4800€
GitHub	Plan Pro de GitHub	3,28€	13,12€
Heroku	2x Heroku Dyno Hobby	11,46€	45,84€
Mongo Atlas	Clúster M0 en AWS	0€	0€
Total		1.214,74€	4.858,96€

Tabla 7.1: Estimación del presupuesto del proyecto.

Bibliografía

[1] "Service Workers." https://developer.mozilla.org/es/docs/Web/API/Service_Worker_API/.

[2] "AllTrails." https://www.alltrails.com/es/.

[3] "Wikiloc." https://es.wikiloc.com/.

[4] "Strava." https://www.strava.com/?hl=es.

[5] "Endomondo." https://www.endomondo.com/.

[6] "Visual Studio Code." https://code.visualstudio.com/.

[7] "Windows Subsystem Linux." https://ubuntu.com/wsl.

[8] "Docker." https://www.docker.com/.

[9] "NodeJS." https://nodejs.org/en/.

[10] "VueJS." https://vuejs.org/.

[11] "GitHub." https://github.com/.

[12] "GitHub Projects." https://github.com/features/project-management/.

[13] "GitHub Actions." https://github.com/features/actions/.

[14] "GitHub Packages." https://github.com/features/packages/.

[15] "Heroku." https://www.heroku.com/.

[16] "Codecov." https://about.codecov.io/.

[17] "Codecov Flags." https://docs.codecov.io/docs/flags/.

[18] "ESLint." https://eslint.org/.

[19] "Koa." https://koajs.com/.

[20] "Express." https://expressjs.com/es/.

[21] "NestJS." https://nestjs.com/.

[22] "Postman." https://www.postman.com/.

[23] "Newman." https://learning.postman.com/docs/running-collections/using-newman-cli/command-line-integration-with-newman/.

[24] "Jest." https://jestjs.io/.

[25] "JSON Web Tokens." https://jwt.io/.

[26] "MongoDB." https://www.mongodb.com/es.

[27] "MongoDB Atlas." https://www.mongodb.com/es/cloud/atlas/.

[28] "Vuex." https://vuex.vuejs.org/.

[29] "Vue Router." https://router.vuejs.org/.

[30] "Quasar." https://quasar.dev/.

[31] "Quasar CLI." https://quasar.dev/quasar-cli/.

[32] "Quasar testing." https://testing.quasar.dev/.

[33] "Quasar Notify." https://quasar.dev/quasar-plugins/notify/.

[34] "Axios." https://github.com/axios/axios/.

[35] "Mapbox." https://www.mapbox.com/.

[36] "Leaflet." https://leafletjs.com/.

[37] "Respositorio de TrackYourTrails." https://github.com/PaulaExposito/TFG-TrackYourTrails.

[38] "Mongoose." https://mongoosejs.com/.

[39] "PWA: introduction to service workers." https://techglimpse.com/pwa-essentials-introduction-service-workers/.

[40] "Mapbox outdoors." https://www.mapbox.com/maps/outdoors/.

[41] "Docker Compose." https://docs.docker.com/compose/.