Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN COMPUTACIÓN

Aplicación móvil para visualización de carreras populares cercanas

Estudiante: Mario Carpente Varela
Dirección: Óscar Fresnedo Arias
José Pablo González Coma

A Coruña, xuño de 2020.

Dedicado a todos los aficionados del atletismo popular y a las personas que les habría gustado presenciar este momento.

Agradecimientos

Quiero agradecer especialmente a mis tutores José Pablo y Óscar toda la ayuda y ánimos recibidos durante el desarrollo de este trabajo, especialmente en los últimos días de entrega.

A toda mi familia y amigos por el apoyo incondicional que me han dado todos estos años.

Y por último y no menos importante, a todos los compañeros y profesores que he tenido durante el trascurso de este grado y en mi anterior formación académica. De todos ellos he aprendió cosas que me han ayudado a llegar hasta aquí.

Resumen

En este trabajo se ha desarrollado una aplicación móvil Android en la que se visualizan carreras populares que están próximas a la ubicación del dispositivo, permitiendo así al usuario poder informarse sobre ellas de manera simple y intuitiva, además de poder filtrar su visualización utilizando diferentes parámetros de interés.

Para completar el desarrollo de la aplicación, primero se ha centralizado la información de las distintas carreras populares disponible en diferentes paginas web, cotejando y complementando dicha información entre sí. Esa información se procesa y guarda en un servidor que, además, se encarga de proporcionar una API REST para su utilización por parte de los clientes, en este caso una aplicación Android, y que permite obtener la información procesada de las carreras populares.

Para el desarrollo del trabajo se ha utilizado el framework Scrapy como crawler para la centralización de la información y posterior envío al servidor, el framework Django para el montaje y configuración del servidor y el IDE Android Studio para el diseño e implementación de la aplicación móvil. También cabe destacar el uso de la API de Google Maps tanto en el servidor como en la aplicación.

Abstract

In this work has been developed an Android mobile application in which popular races that are close to the location of the device are displayed, thus allowing the user to be informed about them in a simple and intuitive way, in addition to being able to filter their visualization using different parameters of interest.

To reach the development of the application, the information of the different popular races has first been centralized in different web pages, comparing and complementing this information with each other on a server, which is in charge of saving it and providing a REST API for its use by customers, in this case an Android application, to obtain the processed information of the popular races.

For the development of the work the has been used Scrapy framework as a crawler for the centralization of the information and subsequent sending to the server, the Django framework for the assembly and configuration of the server and the Android Studio IDE for the design and implementation of the mobile application. Also it's notable the use of the Google Maps API both on the server and in the application.

Palabras clave:

- Carreras populares
- Android
- Django
- API REST
- Servidor
- Crawler
- Scrapy
- Google Maps
- Google Cloud Platform
- Python
- Java
- MVVM
- MongoDB

Keywords:

- Popular races
- Android
- Django
- API REST
- Server
- Crawler
- Scrapy
- Google Maps
- Google Cloud Platform
- Python
- Java
- MVVM
- MongoDB

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

La creciente tendencia a llevar una vida sana y el fomento del deporte han supuesto un incremento en la práctica de actividades como salir a correr, jugar al padel, andar en bici o hacer yoga. Uno de los deportes más practicados por la población es el atletismo popular, un deporte al alcance de cualquiera que no requiere ningún tipo de infraestructura ni de un caro equipamiento para su realización. La práctica de este deporte es individual, y no hay dependencia de otras personas para entrenar, esto permite escoger la hora y duración del entrenamiento que encaje mejor en tu tiempo libre.

Todas estas ventajas hacen que cada vez más gente se anime a salir a correr, mejorando así cualidades como la velocidad y la resistencia para batir sus marcas personales o medirse con otros corredores en competiciones como las carreras populares. Este aumento de corredores ha contribuido en el boom que han tenido en los últimos años las carreras populares. Actualmente se celebran dos o tres carreras cada fin de semana en un área relativamente cercana, mientras que hace diez años tan solo se disfrutaba de una cada semana en toda Galicia.

Por estos motivos este trabajo se centrará en facilitar la información de las carreras populares a sus participantes.

1.1 Motivación

La gran cantidad de carreras populares de diversos tipos y diferente número de participantes celebradas anualmente en Galicia, presentan un escenario amplio y usualmente incompleto cuando se buscan los eventos que se van a celebrar.

La información de estos eventos está muy desperdigada por diferentes recursos web como foros de correr, páginas de federaciones o páginas de los sistemas de cronometraje. Debido a esto surge la problemática de encontrar una carrera en una fuente de información y en otra ser inexistente.

Otro de los escenarios es el posible desconocimiento de los lugares donde se celebran las

carreras, que puede causar una menor participación al no saber la distancia desde tu ubicación y estimar una mayor a la real. La representación de la localización en un mapa es una buena solución a este problema pero su uso es casi inexistente en muchos de los sitios web y aplicaciones en los que puedes consultar la información de las carreras. Tampoco presentan facilidades para filtrar la búsqueda de acuerdo a diferentes criterios como la distancia del recorrido, el coste de la inscripción o el tipo de carrera.

Una aplicación móvil podría integrar todas estas funcionalidades de forma sencilla y cómoda, con la ventaja de que en la actualidad casi todo el mundo dispone de un *smartphone* y podría utilizar la aplicación.

Estos motivos conducen a la realización de este trabajo en el que se intentará centralizar la información de las carreras, sacando dicha información de distintas fuentes y añadiendo la geolocalización del lugar indicado en la carrera. Facilitando así esta información recopilada y procesada al usuario en una aplicación móvil, donde podrá visualizar las carreras por proximidad a su ubicación, fecha de celebración, rango de distancia, etc. Además de la posibilidad de mostrar la ubicación del dispositivo junto con la de las carreras en un mapa y poder visualizar directamente su localización.

1.2 Objetivos

Como se ha comentado en la sección anterior (sección 1.1), el principal objetivo de este trabajo es el diseño e implementación de una aplicación móvil para la visualización de información relacionada con futuras carreras populares. La aplicación tendrá en cuenta la ubicación del usuario para proporcionarle datos de interés y permitirá filtrar las carreras de acuerdo a diferentes criterios. Además del objetivo principal se han establecido otros objetivos para el buen desarrollo del trabajo que comentaremos.

En primer lugar, se requiere la búsqueda y obtención de los datos de las carreras. Extrayéndolos de distintas fuentes que contengan todos los datos necesarios sobre las carreras, o complementarlos entre sí para proporcionar una información detallada del evento. Los principales datos que se extraerán son: el nombre de la carrera, la fecha y lugar de celebración, la modalidad, la distancia del recorrido y el coste de inscripción. Para lograr esto, uno de los objetivos es la búsqueda y utilización de alguna herramienta o plataforma que facilite la extracción de datos y que sea escalable. Así se podrá ir aumentando la información cada vez que surja o se descubra un nuevo recurso que pueda ser de utilidad.

Una vez que se han recopilado todos los datos, es necesario procesarlos y almacenarlos de forma estructurada para poder ofrecérselos a la aplicación o a cualquier otro servicio. Una solución para afrontar este objetivo es implementar un servidor que ofrezca una API para obtener los datos, cumpliendo así otros de los objetivos que es la centralización de la información

y su almacenamiento en una base de datos por parte de un gestor.

Logrados los anteriores objetivos quedaría por realizar la implementación de la aplicación móvil. Donde se tratará de utilizar la ubicación del usuario, como hemos dicho anteriormente, para la visualización de las carreras y el cálculo de costes de desplazamiento. Además se añadirá un mapa con las localizaciones de cada carrera para una mayor claridad y simplicidad visual.

1.3 Estructura de la memoria

En este apartado se explicará brevemente los capítulos de la memoria. Esta compuesta por nueve capítulos que son:

- **Introducción.** Explica el contexto del proyecto y muestra la problemática que ha motivado a hacerlo y los objetivos que se quieren alcanzar.
- **Estado del arte.** Se exponen las aplicaciones similares que existen y se comentan sus funcionalidades. Termina con un apartado de conclusiones en el que se comentan las virtudes y carencias de las aplicaciones.
- **Fundamentos tecnológicos.** Se explican brevemente las herramientas y fuentes de información utilizadas en el proyecto, también se comentan los aspectos por los que se han escogido.
- **Metodología.** Se presenta la metodología que se va utilizar para desarrollar el proyecto, sus característica y su adaptación.
- **Planificación y seguimiento.** Presenta la planificación del proyecto con las tareas y recursos que fueron necesarios. Expone además la planificación inicial, el seguimiento realizado y el análisis económico.
- **Análisis.** Se muestra el resultado del análisis de requisitos del proyecto. Se comentan los actores, la arquitectura y los requisitos funcionales y no funcionales del sistema.
- **Diseño.** Se comentan las decisiones de diseño tomadas en cada una de las partes del sistema.
- **Implementación y pruebas.** En este capítulo, se explican los detalles más relevantes relacionados con la implementación de la aplicación basada en el diseño explicado en el capítulo anterior.
- **Conclusiones.** Se presentan las conclusiones finales que se pueden extraer tras el desarrollo del proyecto y se adelantan algunas líneas de trabajo que se podría considerar en el futuro.

Capítulo 2

Estado del arte

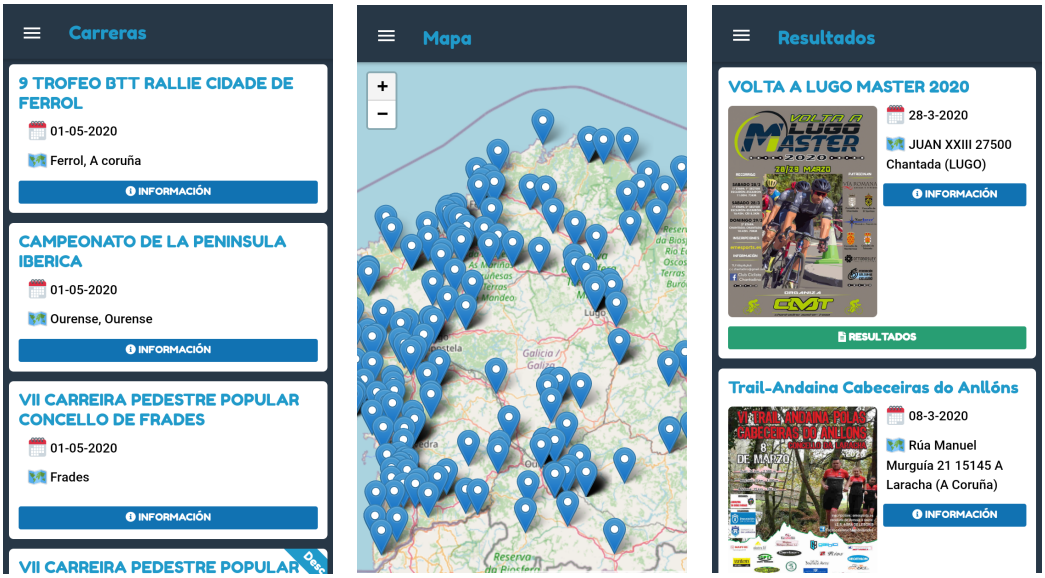
Para el comienzo del trabajo se ha realizado una búsqueda de diferentes aplicaciones con la misma temática, centrándose solo en las disponibles para el sistema Android. Se examinaran sus funcionalidades y la información que proporcionan sobre las carreras para tener en cuenta en la realización del trabajo.

2.1 GaliciaCorre

GaliciaCorre [4] es una aplicación con aspectos similares a los de este trabajo, se centra en la comunidad de Galicia y muestra tanto carreras como competiciones de ciclismo. Alberga un mapa en una de sus secciones donde se muestran las localizaciones de los eventos.

En la figura 2.1a se muestra un listado con las distintas competiciones en el que cada ítem muestra el nombre de la competición, la fecha de celebración, el lugar y, en algunos casos, una imagen del cartel anunciando la competición. Dispone de un botón para obtener más información y, para ciertos eventos, redirige a un sitio web ajeno a la aplicación donde formalizar la inscripción al evento. El mapa de la aplicación, como se puede ver en la figura 2.1b, muestra marcadores con las ubicaciones de los eventos y si pinchas en uno de ellos aparece una ventana con la información descrita anteriormente.

Otra de las secciones interesantes es la de resultados, mostrada en la figura 2.1c, en la que se listan las clasificaciones de las competiciones en el último mes. En esta sección, aparecen una serie de botones para acceder a las distintas clasificaciones y que, dependiendo del evento, redirige a una web o muestra un PDF. Para esta sección y la del listado inicial, la aplicación permite filtrar por mes o por el nombre de la localización.



(a) Lista de eventos (b) Mapa con localizaciones (c) Lista de resultados

Figura 2.1: Pantallas de GaliciaCorre

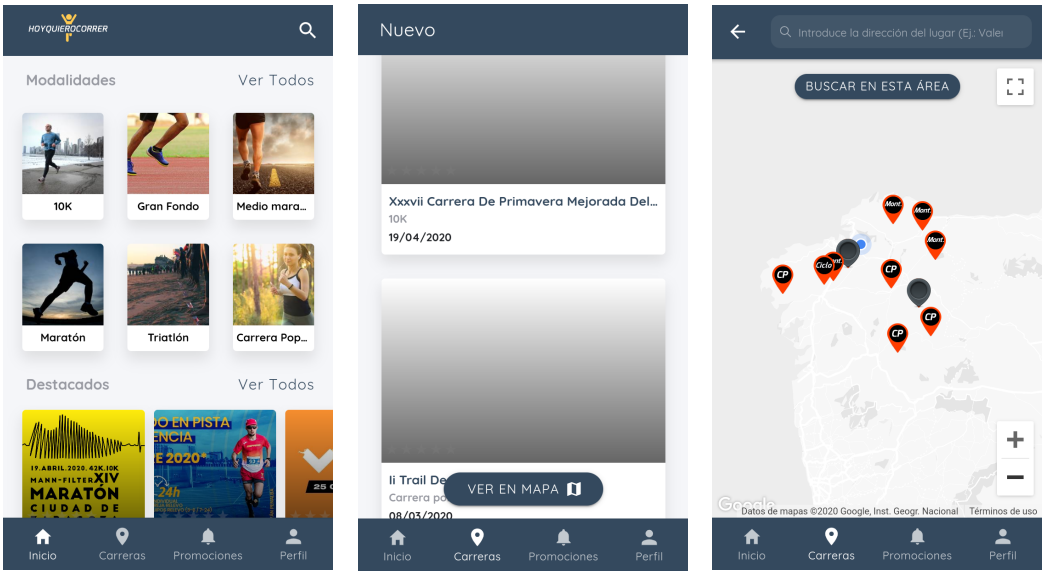
2.2 HoyQuieroCorrer

La aplicación HoyQuieroCorrer [5] abarca todo el territorio español, se centra en distintas modalidades como carreras, marchas, ciclismo, y deportes mixtos como el duatlón y triatlón. Permite que un usuario se registre en la aplicación para poder guardar competiciones como favoritas, compartir opiniones sobre ellas o darles una valoración positiva.

En la pantalla de inicio que se puede observar en la figura 2.2a, se pide el acceso a tu ubicación y se enseñan algunas de las modalidades disponibles. También se muestran diferentes listados que agrupan los eventos en: más destacados, nuevos y más cercanos a tu posición, este último solo si se ha concedido el permiso de ubicación y si está activada en el dispositivo.

En la figura 2.2b se puede observar la sección de carreras que muestra un listado con los eventos, algunos ya pasados de fecha, indicando su modalidad, nombre, fecha y en algún caso imagen. Además, desde aquí se tiene la opción de ver los eventos en el mapa como se muestra en la figura 2.2c y buscarlos en el área donde se ubica la posición de la cámara del mapa.

Por último, al seleccionar una de las competiciones la información mostrada es escasa, tan solo se observa el nombre, la fecha, la modalidad, un enlace web y opiniones de otros usuarios. Estas opiniones suelen ser inexistentes y no se guardan cuando hay un cambio de edición, esto es debido a que el cambio de edición de la misma carrera se trata como una nueva por lo que no se agregan las opiniones de ediciones anteriores.



(a) Pantalla de inicio (b) Lista de eventos (c) Mapa con localizaciones

Figura 2.2: Pantallas de HoyQuieroCorrer

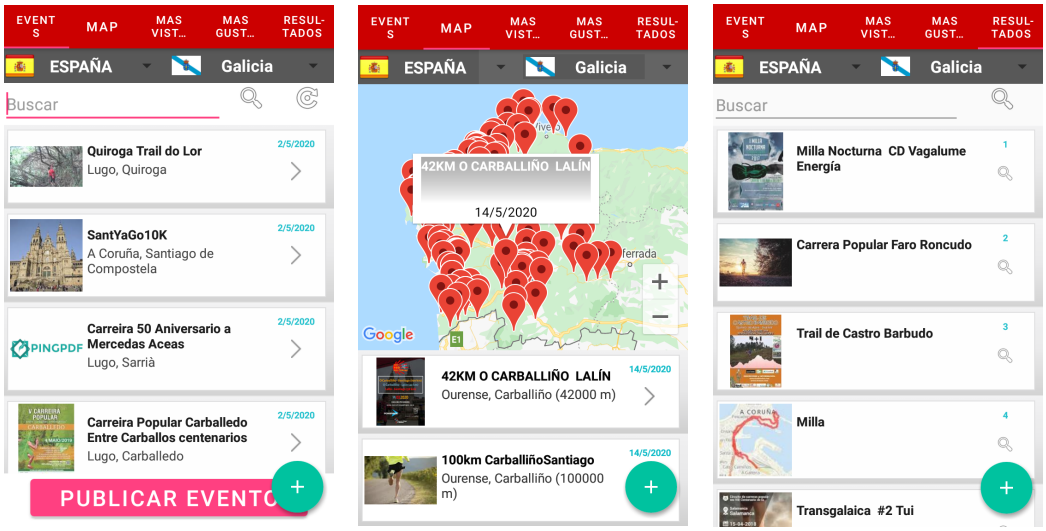
2.3 Carreras populares

La aplicación Carreras populares [6] ofrece el calendario de competiciones más completo, cuenta con el calendario de 6 países junto con sus comunidades o estados, uno de ellos es España y sus comunidades autónomas. También presenta un mapa con las localizaciones de los eventos que, al igual que los anteriores, no solo se centra en las carreras.

En la figura 2.3a se puede observar la pantalla inicial de esta aplicación, en la que se muestra un listado de eventos, ordenados por fecha de celebración y filtrados por país y comunidad, además permite al usuario buscar eventos o crearlos. La información que proporciona sobre los eventos cuenta con: el nombre, el lugar de celebración, la fecha y en algún caso la distancia y una imagen. Al seleccionar un evento permite añadir un comentario, guardarlo en favoritos, inscribirse redirigiendo a una página web y ver vídeos relacionados con el evento si hay alguno publicado.

En la sección del mapa que podemos ver en la figura 2.3b, aparecen las ubicaciones de los eventos señaladas con un marcador, al igual que en la pantalla inicial los eventos mostrados son filtrados por país y comunidad. Al seleccionar un marcador se despliega una lista con los eventos en esa ubicación y una pequeña ventana sobre el marcador con el nombre, la imagen y fecha de la primera competición de la lista.

En la figura 2.3c podemos ver una sección de resultados con un listado de eventos, al seleccionar uno redirige a vídeos o enlaces del evento, pero no tienen relación ninguna con la clasificación de los participantes.



(a) Lista de eventos (b) Mapa con localizaciones (c) Lista de resultados

Figura 2.3: Pantallas de Carreras populares

2.4 Conclusiones

Después de explorar alternativas similares a la aplicación que se va a desarrollar, se puede concluir que existen diversas aplicaciones para informarse sobre carreras populares y que todas ellas proporcionan funcionalidades similares. La mayoría de estas soluciones presentan un mapa con la localización de las competiciones que es uno de los objetivos del trabajo aunque, vista la saturación de los mapas, sería conveniente buscar una solución para una mejor visualización de los marcadores. La visualización en el mapa de más de un evento en la misma localización, es otro de los problemas a tener en cuenta y que se ha solucionado en la aplicación Carreras populares (sección 2.3). La única aplicación que tiene en cuenta la ubicación del dispositivo para mostrar las competiciones cercanas sería HoyQuieroCorrer (sección 2.2) por lo que es un dato muy poco explotado.

El uso de filtros en todas ellas es muy escaso y simple, por lo que se deben de estudiar datos de interés para los usuarios y así facilitar la búsqueda de los eventos. Se puede apreciar que existen datos básicos e importantes sobre las carreras como el nombre, lugar y fecha, comunes en todas ellas y útiles para mostrar en listados y sitios compactos.

La información proporcionada sobre las carreras es insuficiente para que un participante pueda tomar la decisión de inscribirse, faltan datos fundamentales como la distancia del recorrido o el coste de la inscripción. La planificación del viaje puede ser también un punto decisivo, se podría ofrecer la distancia, tiempo y coste del carburante de un viaje en coche hasta la localización de la carrera. Todo esto ayudaría al usuario a encontrar las carreras que entran dentro de su criterio y a llevar la cuenta de los gastos que le supone participar en ellas.

Capítulo 3

Fundamentos tecnológicos

Ahora se expondrán las fuentes de información, la tecnología y las herramientas utilizadas para el desarrollo del proyecto.

3.1 Fuentes de información

Las fuentes de información son los diferentes recursos web de los que se extraen los datos de las carreras.

3.1.1 Correr en Galicia

Correr en Galicia [7] es un foro de aficionados a las carreras populares que cuenta con casi 20.000 usuarios. Dispone del calendario más completo de carreras populares en Galicia, además de mostrar otros eventos como andainas, duatlón y triatlón que no pretendemos tener en cuenta en la aplicación. Podemos decir que es el medio principal para la extracción de datos obteniendo datos como el nombre de la carrera, el lugar, la fecha y hora, el tipo y algún dato más.

3.1.2 Carreiras galegas

Carreiras galegas [8] es la página web oficial de la federación gallega de atletismo. Dispone de un calendario donde se muestran las distintas pruebas en las que normalmente puedes participar si estás federado. La información proporcionada en este sitio web no es muy detallada, nos ofrece el nombre oficial de la carrera, el lugar y la fecha. De este modo, con el lugar y la fecha se puede identificar una carrera ya existente en los datos guardados y sustituir el nombre por el oficial.

3.1.3 Otras fuentes

Debido a la situación excepcional vivida en la actualidad por el COVID-19, las carreras actuales han sido suspendidas o aplazadas. Esto ha causado que muchas de las fuentes de información no dispongan datos de los eventos, por lo que se ha tenido que descartar su uso en el trabajo. En condiciones normales no habría ningún problema para la extracción de los datos como se ha hecho con las fuentes anteriormente mencionadas. Se han tenido en consideración las siguientes fuentes: *Emesports*, *RockTheSport*, *Champion Chip Norte*, *GaliTiming*,... y, en general, otros portales *web* para poder realizar la inscripción en las carreras.

3.2 Lenguajes de programación

3.2.1 Python

Python [9] es un lenguaje de programación de alto nivel creado a finales de los ochenta por Guido van Rossum. Es un lenguaje multiparadigma ya que permite realizar programación funcional, orientada a objetos e imperativa, dando una mayor libertad al programador para la elaboración de su código. Por otro lado, su tipado es dinámico ya que evalúa los tipos en la ejecución del programa y no en la compilación. Su diseño busca la transparencia y legibilidad del código ya que un mal sangrado produce errores de ejecución. Esto combinado con el tipado dinámico y su compilador interactivo capaz de ejecutar sentencias individuales hace que sea un lenguaje muy adecuado para la formación de nuevos programadores. Otra de sus características es que es un lenguaje interpretado, es decir, necesita un intérprete para la ejecución del código lo que lo convierte en multiplataforma ya que funcionará en todos los sistemas que tengan dicho intérprete.

Este lenguaje es necesario en el proyecto para el desarrollo en los *frameworks* de Scrapy (subsección 3.3.1) y Django (subsección 3.3.2), escritos en este mismo lenguaje, facilitando en gran medida la implementación con el tipado dinámico y el uso de librerías estándar que proporciona Python.

3.2.2 Java

Java [10] es otro de los lenguajes de alto nivel que se usarán en el desarrollo de este proyecto. Es un lenguaje concurrente, de propósito general y orientado a objetos, está diseñado para entornos de producción y para proporcionar un manejo simple que permita un uso fluido por parte de los desarrolladores. Una de sus mayores características es ser multiplataforma, por lo que el funcionamiento de los programas desarrollados no está limitado a un sistema operativo. Esto es posible gracias al *Java Runtime Environment* que se divide en una máquina virtual

donde se ejecuta el código Java y una biblioteca estándar que se encarga de proporcionar al desarrollador todas las funcionalidades básicas.

Actualmente es uno de los lenguajes oficiales para el desarrollo de aplicaciones Android y es el que utilizaremos para el desarrollo de nuestra aplicación móvil. Cuenta con una gran comunidad de usuarios que es de gran ayuda para resolver problemas que surgen en el código al usar las API más recientes proporcionadas por Google.

3.3 *Frameworks* y librerías

3.3.1 Scrapy

Una de las tareas principales de este proyecto es la búsqueda y obtención de la información de las carreras populares considerando diferentes recursos web. Para llevar a cabo esta tarea se ha considerado el uso de Scrapy [11], un *framework* de código abierto desarrollado en Python utilizado para tareas de *scraping* y *crawling*. El *scraping* consiste en la extracción de datos estructurados de una o varias web, teniendo conocimiento de los *links* y estructuras de las webs. El *crawling* es más genérico y engloba el *scraping*. Consiste en la utilización de un *bot*, o también llamado araña, que navega por la red utilizado los *links* de las páginas que visita, descargando e indexando su contenido. Esta técnica es normalmente utilizada por buscadores como Bing, Yahoo y Google.

Para iniciar un proyecto en Scrapy se ejecuta un comando que crea el directorio y los archivos base del proyecto, estos archivos contienen la configuración y comportamiento del proyecto. Una vez creada la base es necesario crear las arañas que extraerán los datos de los recursos web. En el caso de este proyecto se creará una araña por cada fuente de información que contenga datos relevantes sobre las carreras.

Al realizar la búsqueda de las diferentes tecnologías para la extracción de datos de las fuentes de información se han encontrado distintas herramientas software con interfaces gráficas como Octoparse [12]. Esta herramienta está diseñada para empresas y personas que no tengan conocimientos de programación. Lo que resta libertad de desarrollo al proyecto ya que limita su uso a las funcionalidades que ofrece. Impidiendo la comunicación con el servidor necesaria para el procesamiento de los datos. Otra de las opciones es la librería Beautiful Soup [13], disponible para Python, que analiza documentos *HyperText Markup Language* (HTML) de los que podemos obtener los datos. Sin embargo, Scrapy es una solución mucho más rápida, eficaz y potente. Dispone de funcionalidades para limpiar los datos extraídos, obtener el recurso web y moverse por los hipervínculos que se le indiquen. En cambio Beautiful Soup para extraer los datos necesita el documento HTML descargado dependiendo de otras librerías para la descarga.

3.3.2 Django

El procesamiento de los datos obtenidos de los recursos web y la centralización de ellos para su exposición a los clientes es una tarea fundamental del proyecto. Para el cometido de esta tarea se ha pensado en el uso de Django [14], un *framework* de desarrollo web de código abierto escrito en Python que sigue el patrón MTV. Permite crear un servidor web de manera rápida y ágil, al igual que otros *frameworks* similares, se encarga de gestionar posibles ataques al servicio de manera automática. Las peticiones a las bases de datos son gestionadas internamente, proporcionando al usuario los mismos métodos y funcionalidades sin tener que manejar explícitamente consultas *Structured Query Language* (SQL). En este proyecto, teniendo en cuenta que se hace uso de MongoDB (subsección 3.5.1), es además necesario la instalación de Djongo [15], un conector que transforma las consultas SQL en consultas apropiadas para MongoDB.

Para la creación de las API *Representational State Transfer* (REST) (subsección 3.6.2), Django se utiliza junto con otro *framework* complementario llamado Django REST [16]. Este *framework* contiene las herramientas necesarias para construir una API REST. Proporciona métodos de autenticación, serializadores para convertir los datos y poder pasárselos a los cliente y una interfaz para la gestión de datos por parte del administrador. Esta interfaz permite visualizar, crear, modificar y eliminar de manera sencilla y práctica los datos guardados.

Se ha escogido este *framework* principalmente por su facilidad de uso y su pequeña curva de aprendizaje para llevar a cabo todo lo necesario en este proyecto. Pudiendo ser innecesariamente más complejo si se utilizara por ejemplo Apache HTTP Server [17]. Otro importante factor de peso en esta decisión es el hecho de que está basando en Python. Evitando de esta forma el uso de lenguajes de programación adicionales en el proyecto como ocurriría en el caso de utilizar Ruby on Rails [18], que utiliza Ruby, o Laravel [19], que utiliza PHP.

3.3.3 GSON

La transformación de los datos *JavaScript Object Notation* (JSON) del servidor a clases Java en la aplicación móvil es necesaria para mostrar correctamente las carreras al usuario. Por lo que para facilitar esta tarea y hacerla más visual se ha decido utilizar Gson [20]. La herramienta Gson (Google Gson) es una librería de código abierto para Java que permite la serialización y deserialización de objetos Java a JSON (formato de texto sencillo para el intercambio de datos). Permite transformar directamente los datos JSON en objetos de una clase Java y viceversa.

Esta librería facilita la transformación de los datos proporcionados por la API en un array de una clase definida para tratar los datos en la aplicación y usarlos para diferentes funcionalidades de ella.

3.4 Herramientas de desarrollo

3.4.1 Visual Studio Code

La elección de un buen entorno de desarrollo agiliza y facilita las labores de programación. Para el desarrollo del código en Python de este proyecto se ha escogido Visual Studio Code [21]. Un editor de código fuente creado por Microsoft y no exclusivo para Windows, se puede instalar también en Linux y MacOS. Es válido para casi cualquier lenguaje de programación o formato de texto. Detecta automáticamente el formato del contenido y sugiere la instalación del complemento correspondiente. El complemento puede permitir funcionalidades como la depuración del código, el resaltado de la sintaxis, sugerencias y refactorización, paso de argumentos...Además, es gratuito y personalizable, permite cambiar el tema del editor, los atajos de teclado, opciones de depuración y otras preferencias.

Por todas estas características se ha escogido este editor para el desarrollo del *crawler* y del servidor, evitando así la instalación de un *Integrated Development Environment* (IDE) específico para el lenguaje y reutilizándolo para editar otros tipos de archivos si es necesario.

3.4.2 Android Studio

La visualización e implementación de los componentes gráficos de una aplicación es una herramienta muy importante para el diseño de la interfaz gráfica. Esta característica junto con la gestión de la estructura de una aplicación Android están presentes en su IDE oficial Android Studio [22]. Basado en el software IntelliJ IDEA de JetBrains, que ha sido modificado por Google y publicado de forma gratuita. Su diseñado esta orientado al desarrollo de aplicaciones Android y admite lenguajes como Java, Kotlin o C++. Dispone de un emulador para probar las aplicaciones desarrolladas en distintos dispositivos y versiones de Android, que se pueden descargar mediante el *Software Development Kit* (SDK) Manager que proporciona el IDE. Además indica si alguna de las dependencias o librerías utilizadas por la aplicación tiene una nueva versión pudiendo así mantener fácilmente actualizado el código.

Todas estas características hacen que Android Studio sea el IDE ideal para desarrollar la aplicación de este proyecto. Pudiendo diseñar rápidamente las distintas interfaces gráficas, crear soporte para más de un idioma y manejar recursos como imágenes de diferentes dimensiones.

3.4.3 Git

Tener un control de las versiones de un proyecto es una tarea importante que puede solucionar muchos de los errores que se producen en el avance del trabajo. Esta tarea puede ser fácilmente manejada por un software de control de versiones como Git [23], el cual se utilizará

en este trabajo. Git está pensado para mantener de forma eficiente y confiable, las distintas versiones del código fuente de aplicaciones con un gran número de archivos. Es muy útil para el registro de cambios y para coordinar el trabajo de varias personas en archivos compartidos de un repositorio *online*. Pudiendo ver los últimos cambios de algún compañero y decidir añadirlos o modificarlos en tu repositorio local.

Se ha escogido este software por ser gratuito, tener su uso familiarizado y permitir guardar una copia en un repositorio *online*. Se utilizará en todas las partes del proyecto para llevar un seguimiento de las versiones y para poder regresar a una anterior si es necesario.

3.5 Sistemas de Gestión de Bases de Datos

3.5.1 MongoDB

Almacenar los datos extraídos y procesados de las carreras es una tarea importante para obtener un rendimiento óptimo en el acceso a los datos. Para la realización de esta tarea se ha considerado el uso de MongoDB [24]. Un sistema de base de datos no relacional y NoSQL, orientado a objetos y de código abierto. Guarda estructuras de datos en formato *Binary JSON* (BSON), prácticamente igual al utilizado para mandar los datos a los clientes desde el servidor. A diferencia de las bases de datos relacionales que guardan los datos en tablas que se relacionan entre sí, este gestor de base de datos los guarda en documentos que no están relacionados ni tienen porque tener la misma estructura de datos. Esto permite añadir datos nuevos o cambiar su representación en la estructura, sin tener que modificar la base de datos ni los datos ya existentes. Además, MongoDB presenta una alta velocidad para el acceso a datos y posibilita la instalación de una interfaz de usuario que facilita el manejo de la base de datos.

La elección de una base de datos no relacional es debido a diversos factores como que los datos almacenados en la aplicación no guardan relación entre ellos, son renovados habitualmente, se necesita rapidez en las consultas, soportar un número elevado de peticiones y el uso de transacciones es mínimo. Además, teniendo en cuenta que en un futuro se puede llegar a manejar una gran cantidad de datos, nos ofrece también ventajas como una gran escalabilidad y un menor consumo de recursos, muy útiles para poder dividir la carga de peticiones entre distintas computadoras.

Por todas estas razones, el uso de MongoDB es preferible para el almacenamiento y manejo de datos basados en aplicaciones de *crawling*, como se demuestra en el artículo *Application of NoSQL Database in Web Crawling* [25].

3.6 Otros

3.6.1 Google Maps Platform

Mostrar las carreras en un mapa al usuario de la aplicación, obtener la distancia del usuario a la carrera y la localización geográfica de su lugar de celebración, son unos de los objetivos del proyecto. Para cumplir estos objetivos es necesario usar Google Maps Platform [26] que forma parte de la plataforma Google Clound, creada por Google para unificar todos los servicios que estaba ofreciendo por separado. Este servicio no es estrictamente gratuito, requiere de un registro donde hay que facilitar un método de facturación. Se cobraran los costes una vez gastados los 200 dólares mensuales no acumulables que nos regala la plataforma. Cada API establece un coste por petición que varía en función del número de peticiones hechas en un mes.

Para usar los servicios de la plataforma es necesario generar una API key, con ella se puede cargar el mapa en la aplicación y hacer peticiones a las API. En este proyecto se utilizan las API *Geocoding* y *Distance Matrix*. *Geocoding* proporciona la posición geográfica del nombre de un lugar si existe, con el resultado de esta petición podemos marca una carrera en el mapa. *Distance Matrix* permite obtener la distancia y el tiempo del viaje en coche de varios orígenes a varios destinos, con esta información calcularemos los costes de desplazamiento desde la posición del usuario a la de las carrera. Ambas API tienen limitado un número de peticiones por segundo y *Distance Matrix* solo permite veinticinco orígenes y destinos por petición.

3.6.2 REST

REST [27] es un estilo de arquitectura web creado por Roy Fielding en su tesis doctoral, y que es comúnmente utilizado para construir las API de los servicios web actuales. De modo que si una API cumple las reglas definidas por la arquitectura REST se denomina API REST. Seguir estas reglas ofrece una serie de ventajas como la separación entre la interfaz de usuario y el almacenamiento de datos, que a su vez proporciona visibilidad, fiabilidad y escalabilidad del producto.

Capítulo 4

Metodología

La metodología de desarrollo en ingeniería de software es un marco de trabajo usado para estructurar, planificar y controlar todo el proceso de desarrollo de un proyecto de software. En este capítulo se describirá la metodología escogida para la realización del trabajo, se expondrán sus características y las particularidades en su aplicación.

4.1 Iterativa e incremental

En este proyecto se ha utilizado la metodología de desarrollo iterativa e incremental [28]. Esta metodología define un proceso de desarrollo de software que consiste en dividir el desarrollo del proyecto en pequeñas iteraciones que cumplan objetivos o requisitos a corto plazo, lo que permite suplir las debilidades del modelo tradicional en cascada. Una iteración es la repetición de un proceso de desarrollo que genera un determinado resultado, el cual se utiliza como punto de partida de la siguiente iteración. Cada iteración se considera una porción del proyecto, que a su vez se denomina incremento. Por lo tanto, la construcción del proyecto consiste en realizar incrementos funcionales de él hasta satisfacer todos los requisitos y objetivos planteados.

El desarrollo de cada iteración sigue la estructura de un proceso típico en cualquier desarrollo software, que es el siguiente:

- **Análisis:** se establece una especificación completa de los requisitos y funcionalidades que debe abarcar la iteración.
- **Diseño:** define la arquitectura de la implementación que soluciona los requisitos y funcionalidades establecidos anteriormente.
- **Implementación:** se procede al desarrollo del sistema diseñado en la fase anterior.
- **Pruebas:** comprueba el correcto funcionamiento de la implementación utilizando distintos tipos de pruebas para conseguir encontrar el mayor número de errores.

La colaboración del cliente es fundamental en esta metodología para aportar su visión sobre los incrementos realizados. De esta forma, es posible corregir aspectos del programa que no le gustan al cliente o añadir nuevos requisitos en el siguiente incremento, sin esperar al final del proyecto. Con esto se evita un potencial rechazo del cliente tras la finalización del proyecto. Además permite ahorrar costes innecesarios debidos a la necesidad de rehacer el proyecto, que podría ocurrir si se usara un desarrollo en cascada. En la figura 4.1 se puede observar un ejemplo del desarrollo iterativo e incremental.

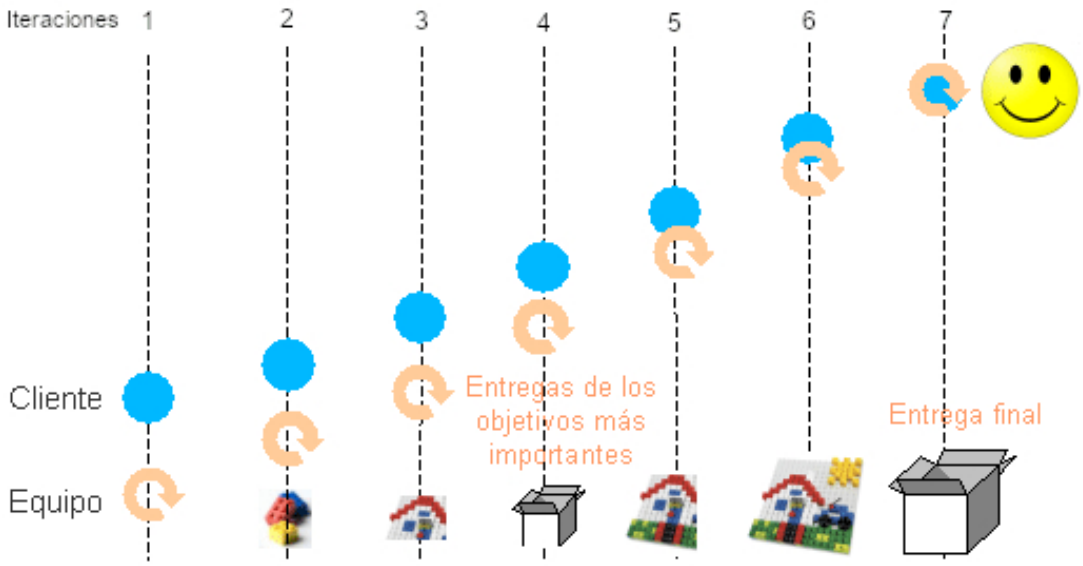


Figura 4.1: Ejemplo del desarrollo iterativo e incremental, imagen obtenida de la web *proyectosagiles* [1]

4.1.1 Características

Las características de una metodología son fundamentales para su elección. Indican las ventajas e inconvenientes que presenta su utilización, pudiendo así decidir si su uso es adecuado o no para un determinado proyecto.

Ventajas

Esta metodología ofrece las siguiente ventajas:

- Permite gestionar las expectativas del cliente de manera regular. Esto es interesante cuando el cliente no sabe exactamente lo que quiere o necesita cambios a corto plazo. Esto además favorece que el equipo de desarrollo pueda verificar si los requisitos que han extraído concuerdan con lo que espera el cliente.

- Se puede comenzar el proyecto con requisitos de alto nivel que se irán refinando con cada incremento. Así no es necesario una recolección completa y detallada de los requisitos al inicio del proyecto. Se recogerán con más detalle solo los de las primeras iteraciones.
- Desde las primeras iteraciones se pueden obtener resultados importantes y funcionales.
- Al finalizar una iteración se puede decidir cómo mejorar el trabajo con la experiencia obtenida en el proceso y las opiniones del cliente, gestionando así los cambios de una manera natural en la siguiente iteración.
- Permite conocer el progreso real del proyecto y mitigar desde el inicio sus riesgos.
- Minimiza el número de errores y aumenta la calidad del resultado. Esto se debe a que, en cada iteración, se proporciona un resultado parcial con unos requisitos terminados.

Inconvenientes

Los inconvenientes presentados por esta metodología son los siguientes:

- Se necesitan técnicas y herramientas que permitan hacer cambios fácilmente en el producto, de manera que pueda crecer en cada iteración sin suponer un gran esfuerzo adicional y manteniendo su calidad.
- Requiere de un cliente involucrado en todo el curso del proyecto y su relación debe ser más colaborativa (beneficiar al proyecto) que contractual (defender su propio beneficio).
- Cada iteración debe dar como resultado una versión funcional del producto que incluya los requisitos especificados, para así poder ser útil al cliente y no dejar tareas pendientes para futuras iteraciones. Esto requiere una planificación muy detallada y ajustada de cada iteración.
- La primeras entregas en iteraciones tempranas con poco valor para el cliente pueden resultar monótonas y producir un rechazo a la continuación del proyecto.

4.1.2 Conclusión

Este proyecto representa un reto importante para el autor ya que requiere familiarizarse y usar una variedad de tecnologías que no ha empleado hasta el momento. Por tanto, el uso de esta metodología permite adquirir el conocimiento necesario para afrontar los diferentes requisitos planteados para el proyecto. Como el desarrollo es iterativo, en el inicio de cada iteración, el desarrollador puede documentarse acerca de todo lo necesario para lograr cumplir

con los requisitos definidos para esa iteración. Además, durante el desarrollo se han tenido que introducir cambios y refinar muchos aspectos del proyecto que no se podrían hacer de manera natural con otras metodologías.

El cliente en este caso son los directores del proyecto. Estos aportan su opinión en cada incremento del proyecto, pudiendo implicar nuevos requisitos o modificaciones en el resultado que se añadirán en la siguiente iteración. Se puede contar con una alta implicación del cliente por lo que la mayoría de inconvenientes presentados anteriormente no afectan al desarrollo del proyecto.

Por todos estos motivos y debido al hecho de que el equipo de desarrollo está integrado por una única persona, se ha pensado en esta metodología como la más adecuada para el desarrollo del proyecto.

4.2 Adaptación al proyecto

Una vez escogida la metodología, se ha aplicado al desarrollo del proyecto. Se han realizado las adaptaciones necesarias y se ha perfilado una visión general de las iteraciones que es necesario realizar para cumplir con los objetivos propuestos.

Como se ha mencionado en la sección anterior (sección 4.1.2), los directores de este trabajo adoptarán el rol de cliente en esta metodología. Partiendo de esta filosofía, se ha aprovechado una parte de las reuniones de seguimiento del trabajo para realizar una sesión de demostración al cliente, en la que se extraen nuevos requisitos y se refinan los anteriores.

Al inicio del desarrollo se ha planteado la siguiente secuencia de iteraciones:

- **Iteración 1: Análisis general del proyecto.** Se extraen los requisitos del proyecto y se busca y estudia el abanico de tecnologías necesarias para su realización. Además se realiza una búsqueda inicial de fuentes de información que proporcionen los datos necesarios para el comienzo del proyecto.

- **Iteración 2: Extracción y centralización inicial de los datos.** Se configuran e implementan el *crawler* y el servidor. También se adapta el servidor para el uso de la base de datos y se implementa la visualización básica y el procesado de los datos extraídos.

- **Iteración 3: Implementación básica de la aplicación.** Se crea la aplicación y se desarrollan funcionalidades básicas como la obtención y visualización de las carreras. También se extrae la localización geográfica de las carreras y se muestran en un mapa.

- **Iteración 4: Inclusión de nuevas fuentes de información y nuevos campos.** Se agregan nuevas fuentes de información al *crawler* y se añaden más datos relevantes al modelo de la base de datos.

- **Iteración 5: Implementación de filtros y ajustes.** Se añade una pantalla de ajustes a la aplicación y se implementan los filtros que permiten obtener al usuario las carreras que busca. También se incluyen nuevos campos de interés como el coste de inscripción, los costes de desplazamiento, etc.

Capítulo 5

Planificación y seguimiento

Para llevar un control del progreso del proyecto se ha realizado una planificación con su posterior seguimiento y una estimación de los costes. En este capítulo se expondrán los detalles de esa planificación y su seguimiento, además de un análisis económico con los costes estimados y finales del proyecto.

5.1 Planificación

La planificación de un proyecto consiste en la ordenación de las tareas para completar los objetivos del proyecto, donde se expone lo que se necesita hacer, cómo debe llevarse a cabo y el tiempo necesario para realizarlo. Con esta planificación se puede detectar y resolver contratiempos en las tareas que pueden suponer un retraso en la fecha de finalización o un incremento de los costes.

En este apartado se comentarán los recursos necesarios, las tareas a realizar y la planificación inicial del proyecto.

5.1.1 Recursos

El uso de recursos es un elemento esencial en el desarrollo de un proyecto y requiere una correcta gestión para minimizar sus costes. Un recurso es todo aquel elemento que influye en la organización necesaria para realizar y materializar las tareas que sustentan un objetivo. Para realizar este proyecto han sido necesarios dos tipos de recursos: humanos y materiales.

Recursos humanos

Los recursos humanos son los empleados encargados de la realización de las tareas del proyecto. En este caso, se han simulado varios perfiles de recursos humanos que se asignarán a las tareas en las que se ha divide el trabajo. Los perfiles simulados son:

- **Jefe de proyecto:** es la persona encargada de planificar, ejecutar y monitorizar las acciones que forman parte de un proyecto. También se encarga de la coordinación y gestión de los recursos utilizados en él para un correcto desarrollo. Este rol lo realizan los directores del proyecto.

- **Analista:** es el encargado de identificar las necesidades del cliente y trasmitirlas al equipo de desarrollo en forma de requisitos. Este rol está asignado al autor del proyecto.

- **Diseñador:** se encarga de realizar un diseño del sistema que cumpla los requisitos proporcionados por el analista. Este rol lo desempeña al autor del proyecto.

- **Programador:** llevará a cabo la implementación del sistema de acuerdo a las directrices establecidas por el diseñador. Realizará también las pruebas necesarias para comprobar el correcto funcionamiento del sistema. Este rol es realizado por el autor.

Recursos materiales

Los recurso materiales son las propiedades tangibles como instalaciones, oficinas, maquinaria, equipos y herramientas. Son necesarios por parte de los recursos humanos en la realización de las tareas y suelen significar una inversión en el proyecto. En este proyecto se han necesitado los siguientes recursos:

- **Netway Shiva:** ordenador portátil con procesador Intel Core i5-7300HQ 2.50GHz, 16GB de memoria *Random Access Memory* (RAM) y tarjeta gráfica NVIDEA GTX-1050. Se ha utilizado este equipo para el desarrollo del proyecto y la instalación de las herramientas software necesarias.

- **BQ Aquaris U Plus:** *smartphone* con sistema operativo Android 7.1.1 y una pantalla de cinco pulgadas. Utilizado para la instalación y pruebas de la aplicación móvil.

5.1.2 Tareas

Las tareas son los diferentes trabajos que hay que realizar para obtener el producto final del proyecto. Para la planificación del proyecto se han dividido las iteraciones, mencionadas en el capítulo anterior (sección 4.2), en tareas que especifican la secuencia de acciones que hay que realizar en cada una de ellas. Las tareas planeadas para el desarrollo del proyecto son las siguientes:

- **Iteración 1.** Análisis general del proyecto.

  T1.1 Extraer de manera general los requisitos.

T1.2 Buscar las herramientas tecnológicas y las fuentes de información necesarias para el desarrollo.

T1.3 Estudiar y adaptar las tecnologías encontradas al proyecto.

- **Iteración 2.** Extracción y centralización inicial de los datos.

T2.1 Configurar el *crawler* e implementar la extracción de datos de la fuente principal.

T2.2 Configurar el servidor y adaptar su conexión con MongoDB.

T2.3 Conectar el *crawler* con el servidor.

T2.4 Crear el primer modelo de datos.

T2.5 Implementar el procesado y posterior almacenamiento de los datos.

- **Iteración 3.** Implementación básica de la aplicación.

T3.1 Implementar la estructura inicial de la aplicación usando el patrón MVVM

T3.2 Conectar la aplicación con el servidor.

T3.3 Crear vista de lista para las carreras.

T3.4 Obtener la localización geográfica de cada carrera en el servidor.

T3.5 Crear un mapa y posicionar las carreras.

T3.6 Hacer una vista con la información detallada de una carrera seleccionada.

T3.7 Segurizar el acceso a métodos que alteren los datos del servidor.

- **Iteración 4.** Inclusión de nuevas fuentes de información y nuevos campos.

T4.1 Buscar e implementar la extracción de datos de interés de nuevas fuentes.

T4.2 Implementar el procesado de los nuevos datos extraídos en el servidor.

T4.3 Modificar el modelo de la base de datos añadiendo los nuevos datos.

T4.4 Incluir los nuevos campos en la aplicación.

- **Iteración 5.** Implementación de filtros y ajustes.

T5.1 Calcular la distancia por carretera desde la posición del dispositivo a la de cada carrera.

T5.2 Obtener el coste de inscripción.

T5.3 Obtener los precios actuales de los combustibles.

T5.4 Calcular los gastos de desplazamiento.

T5.5 Modificar la vista detallada añadiendo información del viaje.

T5.6 Diseñar y desarrollar los filtros.

T5.7 Traducir la aplicación a más de un idioma.

T5.8 Crear e implementar una pantalla de ajustes.

T5.9 Refinar la vista del mapa agrupando los marcadores en *clusters*.

Al final de cada iteración se ha creado una tarea de seguimiento, no mencionada anteriormente, donde se realiza la reunión con los directores del proyecto para analizar y mejorar los aspectos del resultado obtenido. Además como última tarea se realizará la documentación del trabajo realizado.Para la planificación del proyecto se ha hecho una estimación de la duración de cada tarea que se expondrá en la siguiente sección.

5.1.3 Planificación inicial

Después de estimar los recursos necesarios para el proyecto y de definir las tareas que se van a realizar, se realiza la planificación inicial. En esta planificación, se asigna un tiempo estimado de trabajo a las tareas y los recursos que las van llevar a cabo.

El inicio de este proyecto se ha establecido el día 3 de Febrero de 2020 y la fecha estimada de finalización es el 26 de Mayo de 2020. En este intervalo de tiempo se han contemplado las festividades que se pueden observar en la figura 5.1, en las que no se ha trabajado en el proyecto. Aun así se prevé un duración de setenta y un días laborales con seiscientas treinta y dos horas de trabajo.

Excepciones | Semanas laborales

	Nombre	Comienzo	Fin
1	Carnavales	24/02/2020	25/02/2020
2	San José	19/03/2020	19/03/2020
3	Patrón Facultad	20/03/2020	20/03/2020
4	Semana Santa	06/04/2020	13/04/2020
5	Día del trabajo	01/05/2020	01/05/2020

Figura 5.1: Festividades contempladas en la planificación

Al realizar la planificación se han detectado riesgos que pueden afectar al trascurso del desarrollo. La falta de conocimiento del dominio por parte del autor en los temas de desarrollo y la poca experiencia en la planificación de proyectos son dos de los riesgos que se han tenido que asumir. Para evitar estos riesgos se debería contar con personal cualificado pero, en este proyecto, no es posible debido a que el trabajo tiene que ser desarrollado en su plenitud por el

autor con la colaboración de los directores. En cualquier caso, se han valorado estos riesgos a la hora de definir la planificación inicial.

Para mostrar de forma clara la planificación que se ha hecho, se han incluido en este documento dos figuras. La figura 5.2 muestra las tareas con la duración inicial estimada y los recursos asignados. En la figura 5.3 se muestra el diagrama de Gantt generado con la configuración de las tareas.

5.2 Seguimiento

En el seguimiento del proyecto han surgido diversas desviaciones que han afectado a la duración del proyecto. Algunas de ellas son leves retrasos de una hora o dos en algunas de las tareas.

Donde ha surgido un mayor retraso ha sido en la tercera iteración, debido a un desconocimiento del dominio que ha causado problemas en la implementación. Las tareas *Implementar la estructura inicial de la aplicación usando el patrón MVVM* y *Crear un mapa y posicionar las carreras* han sumado un total de 17 horas de retraso.

La situación actual causada por el COVID-19 ha reducido considerablemente la duración de la cuarta iteración. Las tareas *Buscar e implementar la extracción de datos de interés de nuevas fuentes* y *Implementar el procesado de los nuevos datos extraídos en el servidor* redujeron 30 horas de trabajo. Esto fue debido a que una parte de las fuentes de información ya no poseían datos de los eventos porque estos estaban cancelados o suspendidos.

En general la duración total del proyecto no se ha visto muy afectada, como podemos ver en la tabla 5.1, aunque de no ser por la situación excepcional actual podría haber experimentado un cierto retraso. En la figura 5.4 se puede ver una comparativa entre el diagrama de Gantt de la planificación inicial y el del seguimiento.

	Estimada	Real
Fecha de inicio	03/02/20	03/02/20
Fecha de finalización	17/06/20	18/06/20
Duración	86,34 días	87,46 días
Trabajo	740 horas	749 horas

Tabla 5.1: Comparativa de la duración estimada y real del proyecto

5.3 Análisis económico

El elevado aumento de los costes de un proyecto en su desarrollo puede obligar a cancelarlo y sufrir grandes perdidas. Por ello, en esta sección se analizarán los costes estimados

Id	Modo de tarea	Condicionada por el esfuerzo	Nombre de tarea	Trabajo	Duración	Predecesoras	Nombres de los recursos	Comienzo	Fin	Costo
1			**No Análisis general del proyecto**	**104 horas**	**13 días?**			**lun 03/02/20**	**mié 19/02/20**	**1.227,20 €**
2			Si Extraer de manera general los requisitos	8 horas	1 día?		Analista	lun 03/02/20	lun 03/02/20	94,40 €
3			Si Buscar las herramientas tecnológicas y las fuentes de información necesarias para el desarrollo	48 horas	12 días?	2	Analista[50%]	mar 04/02/20	mié 19/02/20	566,40 €
4			Si Estudiar y adaptar la tecnología encontrada al proyecto	48 horas	12 días?	2	Analista[50%]	mar 04/02/20	mié 19/02/20	566,40 €
5			No Reunión de seguimiento y revisión del proyecto	12 horas	4 horas	3;4	Jefe de Proyecto 1;Ju	jue 20/02/20	jue 20/02/20	217,60 €
6			**No Extracción y centralización inicial de los datos**	**135 horas**	**16,88 días?**			**jue 20/02/20**	**mié 18/03/20**	**1.218,50 €**
7			Si Configurar el crawler e implementar la extracción de datos de la fuente principal	40 horas	5 días?	5	Programador	jue 20/02/20	lun 02/03/20	360,00 €
8			Si Configurar el servidor y adaptar su conexión con MongoDB	60 horas	7,5 días?	7	Programador	lun 02/03/20	mié 11/03/20	540,00 €
9			Si Conectar el crawler con el servidor	6 horas	0,75 días?	8	Programador	jue 12/03/20	jue 12/03/20	54,00 €
10			Si Crear el primer modelo de datos	5 horas	0,63 días?	9	Programador[50%];D	jue 12/03/20	vie 13/03/20	48,50 €
11			Si Implementar el procesado y posterior guardado de los datos	24 horas	3 días?	10	Programador	vie 13/03/20	mié 18/03/20	216,00 €
12			No Reunión de seguimiento y revisión del proyecto	12 horas	4 horas	11	Jefe de Proyecto 1;Ju	mié 18/03/20	mié 18/03/20	217,60 €
13			**No Implementación básica de la aplicación**	**122 horas**	**15,46 días?**			**mié 18/03/20**	**mar 21/04/20**	**1.107,33 €**
14			Si Implementar la estructura inicial de la aplicación usando el patrón MVVM	40 horas	5,21 días?	12	Programador[80%];D	mié 18/03/20	lun 30/03/20	369,33 €
15			Si Conectar la aplicación con el servidor	8 horas	1 día?	14	Programador	lun 30/03/20	mar 31/03/20	72,00 €
16			Si Crear vista de lista para las carreras	8 horas	1 día?	15	Programador	mar 31/03/20	mié 01/04/20	72,00 €
17			Si Obtener la localización geográfica de cada carrera en el servidor	8 horas	1 día?	16	Programador	mié 01/04/20	jue 02/04/20	72,00 €
18			Si Crear un mapa y posicionar las carreras	40 horas	5 días?	17	Programador	jue 02/04/20	vie 17/04/20	360,00 €
19			Si Hacer una vista con la información detallada de una carrera seleccionada	10 horas	1,25 días?	18	Programador	vie 17/04/20	lun 20/04/20	90,00 €
20			Si Segurizar el acceso a métodos que alteren los datos del servidor	8 horas	1 día?	19	Programador	lun 20/04/20	mar 21/04/20	72,00 €
21			No Reunión de seguimiento y revisión del proyecto	15 horas	5 horas	20	Jefe de Proyecto 1;Ju	mar 21/04/20	mar 21/04/20	272,00 €
22			**No Inclusión de nuevas fuentes de información y nuevos campos**	**96 horas**	**10,83 días?**			**mar 21/04/20**	**jue 07/05/20**	**901,33 €**
23			Si Buscar e implementar la extracción de datos de interés de nuevas fuentes	24 horas	1,5 días?	21	Analista;Programado	mar 21/04/20	jue 23/04/20	249,60 €
24			Si Implementar el procesado de los nuevos datos extraídos en el servidor	56 horas	7 días?	23	Programador	jue 23/04/20	mar 05/05/20	504,00 €
25			Si Modificar el modelo de la base de datos añadiendo los nuevos	8 horas	1,33 días?	24	Programador[50%];D	mar 05/05/20	mié 06/05/20	75,73 €
26			Si Incluir los nuevos campos en la aplicación	8 horas	1 día?	25	Programador	mié 06/05/20	jue 07/05/20	72,00 €
27			No Reunión de seguimiento y revisión del proyecto	12 horas	4 horas	26	Analista;Jefe de Pro	jue 07/05/20	vie 08/05/20	217,60 €
28			**No Implementación de filtros y ajustes**	**97 horas**	**12,42 días?**			**vie 08/05/20**	**mar 26/05/20**	**880,75 €**
29			Si Calcular la distancia por carretera desde la posición del dispositivo a la de cada carrera	12 horas	1,5 días?	27	Programador	vie 08/05/20	lun 11/05/20	108,00 €
30			Si Obtener el coste de incripción	8 horas	1 día?	29	Programador	lun 11/05/20	mar 12/05/20	72,00 €
31			Si Obtener los precios actuales de los combustibles	8 horas	1 día?	30	Programador	mar 12/05/20	mié 13/05/20	72,00 €
32			Si Calcular los gastos de desplazamiento	4 horas	0,5 días?	31	Programador	mié 13/05/20	jue 14/05/20	36,00 €
33			Si Modificar la vista detallada añadiendo información del viaje	4 horas	0,5 días?	32	Programador	jue 14/05/20	jue 14/05/20	36,00 €
34			Si Diseñar y desarrollar los filtros	24 horas	3,3 días?	33	Programador[70%];D	jue 14/05/20	mié 20/05/20	223,75 €
35			Si Traducir la aplicación a más de un idioma	10 horas	1,25 días?	34	Programador	mié 20/05/20	jue 21/05/20	90,00 €
36			Si Crear e implementar una pantalla de ajustes	15 horas	1,88 días?	35	Programador	jue 21/05/20	lun 25/05/20	135,00 €
37			Si Refinar la vista del mapa agrupando los marcadores en clusters	12 horas	1,5 días?	36	Programador	lun 25/05/20	mar 26/05/20	108,00 €
38			No Reunión de seguimiento y revisión final del proyecto	15 horas	5 horas	37	Jefe de Proyecto 1;Ju	mar 26/05/20	mié 27/05/20	272,00 €
39			Si Elaboración de la documentación	120 horas	15 días?	38	Diseñador[50%];Ana	mié 27/05/20	mié 17/06/20	1.332,00 €

Figura 5.2: Planificación inicial de las tareas

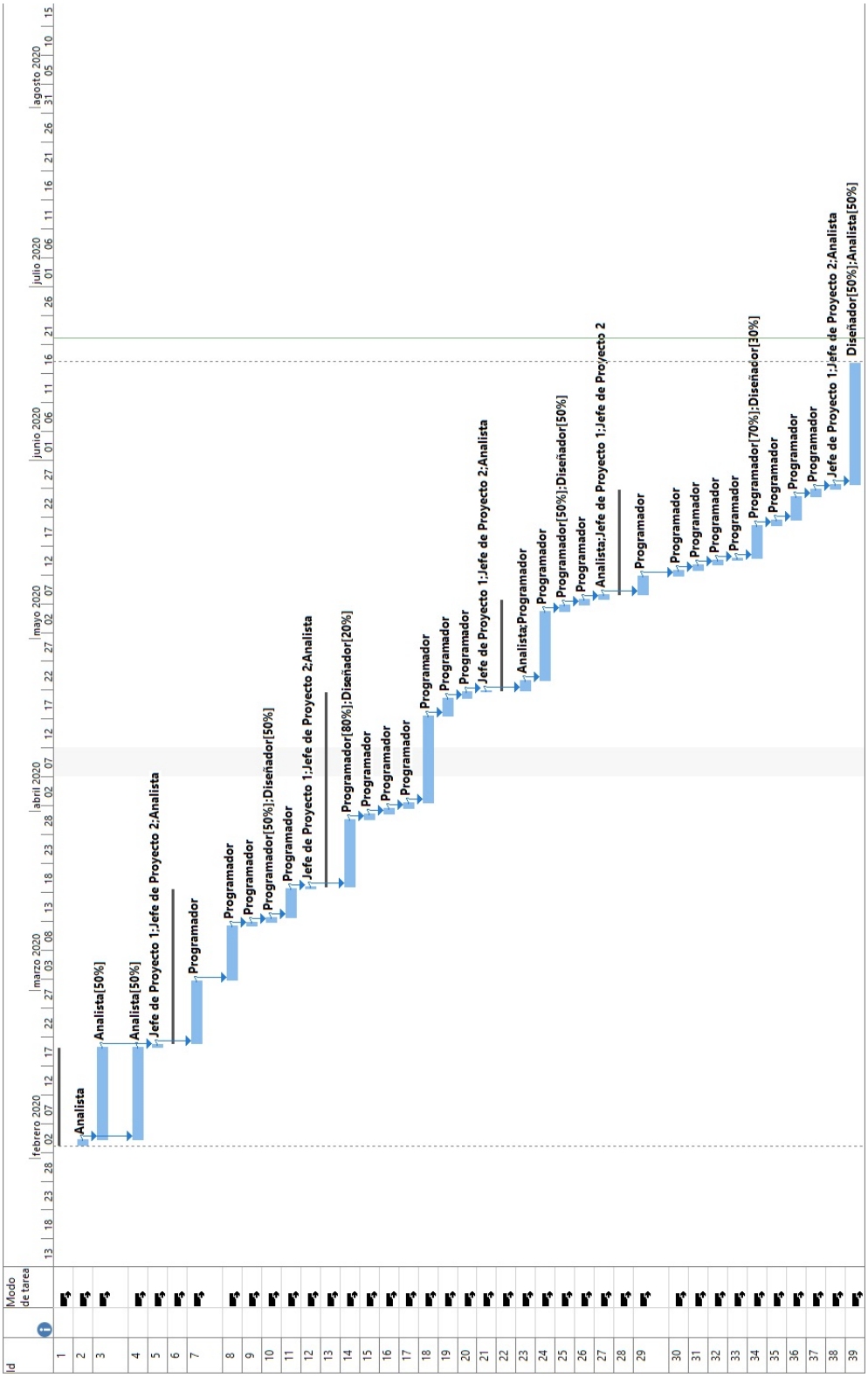


Figura 5.3: Diagrama de Gantt de la planificación inicial

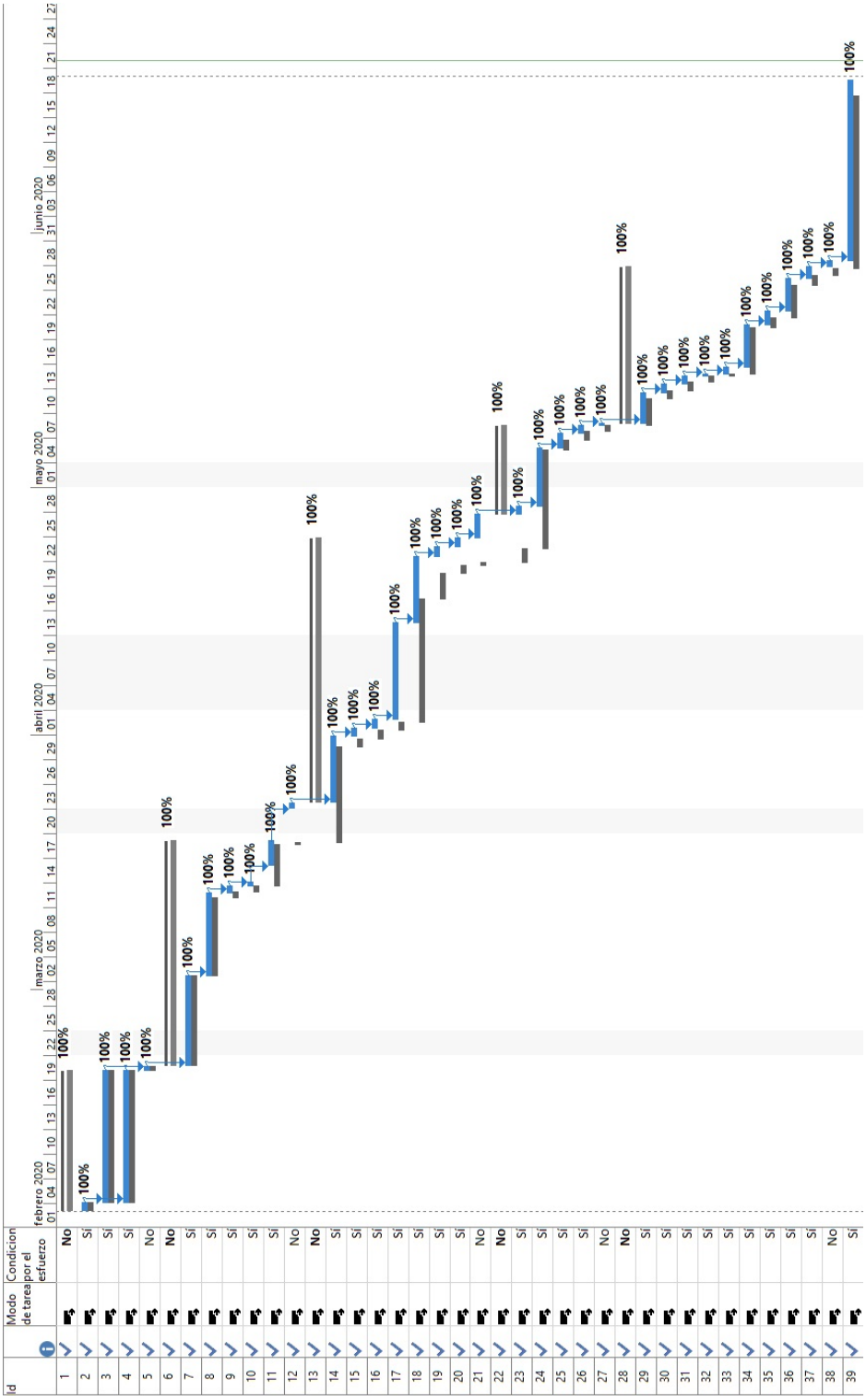


Figura 5.4: Comparativa diagrama de Gantt inicial y seguimiento

para la realización del proyecto y cómo se han visto afectados estos costes tras su finalización.

En primer lugar, se debe mencionar que los recursos materiales utilizados no han supuesto ningún coste adicional en el proyecto debido a que ya se disponía de ellos. En cambio, para los recursos humanos, se ha hecho el cálculo de los costes que han supuesto sus horas de trabajo. Para ello fue necesario especificar el salario por hora de cada perfil contratado, que se puede ver en la tabla 5.2.

Perfil	Salario hora	Salario anual
Jefe de proyecto	21,3€	45.000€
Analista	11,8€	25.000€
Diseñador	10,4€	22.000€
Programador	9€	19.000€

Tabla 5.2: Salario de los recursos humanos

El salario por hora ha sido calculado utilizando el salario anual proporcionado por un estudio salarial [29] realizado en Galicia. Para la realización del cálculo se ha contemplado una jornada de ocho horas diarias y un total de veintiún días laborables cada mes.

Para finalizar el análisis de los costes del proyecto, se incluye la tabla 5.3 en la que se puede apreciar las horas de trabajo y coste de cada recurso humano al inicio de la planificación y al final del seguimiento. Como se puede observar el coste total del proyecto no se ha visto muy afectado a pesar de los problemas surgidos en el desarrollo.

	Estimado		Real	
	Horas	Coste	Horas	Coste
Jefe de proyecto 1	22	468,6€	22	468,6€
Jefe de proyecto 2	22	468,6€	22	468,6€
Analista	198	2.336,4€	196	2.312,8€
Diseñador	77,38	804,74€	81,04	842,81€
Programador	420,62	3.785,58€	427,96	3.851,64€
TOTAL	740	7.863,92€	749	7.994,45€

Tabla 5.3: Horas y costes de los recursos humanos

Además de los costes del proyecto se ha realizado una estimación de los costes de las API de Google mencionadas en la sección Google Maps Platform (sección 3.6.1).

El coste por petición de cada API varia en función del número de peticiones realizadas en un mes. La tabla 5.4 muestra esta variación que ha sido extraída de la sección de precios [30] de la plataforma.

Suponiendo que cada usuario hace tres peticiones mensuales a *Geocoding* API y ciento veinte a *DistanceMatrix* API, y que el servidor realiza seis mil mensuales solo a *Geocoding* API. Se ha realizado el cálculo del coste mensual de las API para mil y diez mil usuarios, que

Rango de peticiones	*Geocoding*	*DistanceMatrix*
0 a 100.000	0.005 USD	0.005 USD
100.001 a 500.000	0.004 USD	0.004 USD
+500.000	Contactar ventas	Contactar ventas

Tabla 5.4: Costes por petición de las API utilizadas

se puede observar en la tabla 5.5.

Nº Usuarios	Peticiones *Geocoding*	Peticiones *DistanceMatrix*	Coste
1.000	9.000	120.000	325 USD
10.000	36.000	1.200.000	4.680 USD

Tabla 5.5: Estimación de los coste mensuales para mil y diez mil usuarios

Al resultado obtenido se le ha descontado los dos cientos USD mensuales, no acumulables, que regala la plataforma Google Maps Platform a todos los usuarios. Cabe destacar que el coste para diez mil usuarios sería más bajo debido a que las peticiones a *DistanceMatrix* API superan las quinientas mil mensuales. En este caso habría que ponerse en contacto con ventas para aplicar el precio por petición que le corresponde. A pesar de todo, el coste por usuario es bastante inferior a un USD mensual.

Capítulo 6

Análisis

El análisis de requisitos es fundamental en la elaboración de un proyecto software. En él se estudian las necesidades tecnológicas que tiene el cliente y se establecen los requisitos del sistema. Se deben concretar además los objetivos que debe cumplir la implementación para obtener un producto de calidad que satisfaga al cliente.

En este capítulo, se comentarán los actores, la arquitectura, los requisitos funcionales y no funcionales que presenta el sistema. Estos elementos se han extraído en el análisis realizado en la primera iteración del proyecto.

6.1 Actores

Un actor es una entidad que realiza algún tipo de interacción con el sistema, no tiene porque ser humana. En este sistema se han identificado los siguientes actores:

- **Usuario anónimo de la API:** es cualquier entidad que hace uso de la API sin estar identificado. Este usuario solo puede obtener o visualizar los datos de las carreras que ofrece el servidor en la API.
- ***Crawler*:** es la entidad que provee la información al servidor, y es necesario estar identificado con un usuario concreto en el servidor. Se le permite utilizar los métodos de la API que dan de alta, modifican, visualizan y borran las carreras.
- **Usuario administrador:** se trata del usuario identificado con las credenciales del administrador en el servidor. Este actor no tiene límites dentro del servidor, puede gestionar los usuarios, grupos y carreras del sistema directamente mediante una interfaz gráfica y utilizar todos los métodos de la API.
- **Usuario de la aplicación:** es toda persona que tenga instalada en su móvil la aplicación y haga uso de ella. No tiene ningún límite de uso dentro de la aplicación.

En la figura 6.1 podemos ver la jerarquía de los actores, que muestra los roles heredados de cada uno. El usuario anónimo de la API es el que tiene los roles más básicos de los que disponen todos por eso hereden dél. Como aclaración la API es el centro de la comunicación del sistema. Del *Crawler* hereda el usuario administrador porque el administrador incluye los roles del *Crawler* tanto propios como heredados. Si la aplicación no formara parte del sistema no existiría su actor, se vería como usuario anónimo de la API.

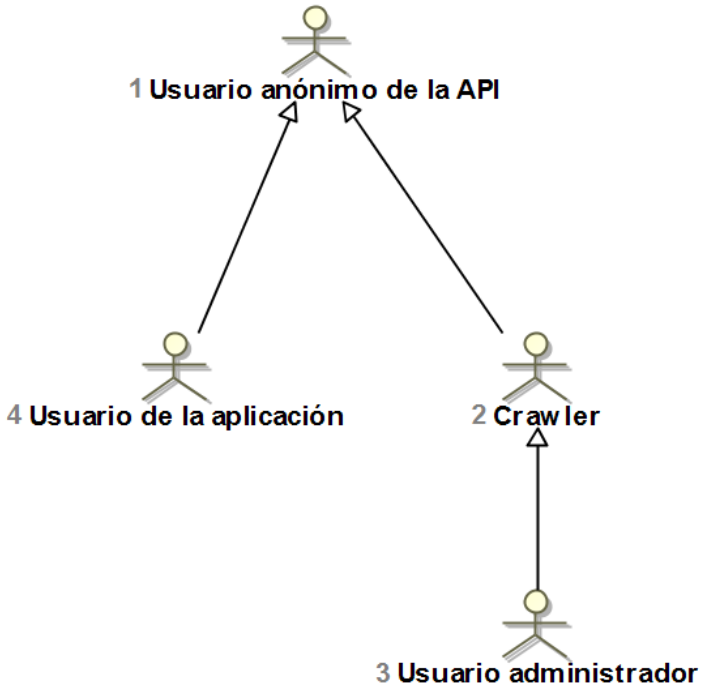


Figura 6.1: Jerarquía de generalización de los actores del sistema

6.2 Requisitos funcionales

Los requisitos funcionales describen las acciones que debe realizar el sistema software cuando un agente interactúa con él. En el análisis realizado se han extraído los siguientes requisitos funcionales:

Servidor

- **Autenticación.** El servidor tiene que proporcionar un método para identificar los usuarios. En concreto, es necesario proporcionar la autenticación al *crawler* y al administrador.

- **Procesar y guardar los datos.** El servidor debe procesar y guardar los datos de las carreras enviados por el *crawler* a la API. En este procesamiento está incluida la obtención de la geolocalización de los lugares en los que se celebran las carreras.

- **Proporcionar los datos guardados.** La API del servidor debe ofrecer un método para la obtención de los datos de las carreras guardadas. Podrá obtener estos datos cualquier actor del sistema.

Aplicación móvil

- **Listar las carreras.** La aplicación móvil debe mostrar al usuario un listado con las carreras obtenidas del servidor.

- **Mostrar las carreras en el mapa.** Se debe incluir un mapa en la aplicación móvil donde se mostraran los ubicaciones de las carreras obtenidas y la ubicación del usuario.

- **Ver los detalles de una carrera.** La aplicación móvil debe contener una sección donde mostrar la información detallada de cada carrera de manera individual.

- **Filtrar carreras.** Se debe de proporcionar algún método para filtrar, según ciertos criterios, las carreras en la aplicación móvil. Esto le facilita al usuario la búsqueda de las carreras que contengan características de su interés.

- **Obtener el nombre de la ubicación del usuario.** A partir del las coordenadas geográficas de la localización del usuario, poder obtener el nombre del lugar en el que se encuentra.

- **Obtener la distancia de la ubicación del usuario a las carreras.** Utilizando como origen la ubicación del usuario y destino la de las carreras, se debe obtener la distancia y tiempo del viaje por carretera para el usuario de la aplicación.

- **Obtener el precio del combustible.** Se debe obtener el precio actual del tipo de combustible indicado por el usuario para realizar cálculos en la aplicación.

- **Calcular los costes del viaje a cada carrera.** La aplicación debe mostrar al usuario los costes del viaje al lugar de celebración de la carrera. Para ello es necesarios calcular dichos costes con otros datos almacenados en la aplicación.

- **Ver y modificar ajustes.** Se debe proporcionar un apartado de ajustes en la aplicación, donde el usuario pueda ver y modificar opciones que cambien el funcionamiento de la aplicación.

Comunes

- **Manejo de errores.** Todas las partes del sistema tienen que informar y manejar los posibles errores que puedan surgir en la ejecución como, por ejemplo, un error en la obtención de los datos desde la aplicación, una malformación en los datos enviados por el *crawler* a la API, etc.

6.3 Requisitos no funcionales

Los requisitos no funcionales, al contrario que los anteriores, no hacen referencia directa a las funciones específicas del sistema si no que marcan las cualidades que debe tener. Los requisitos no funcionales de este sistema son:

- **Fácil de usar.** La aplicación móvil debe ser sencilla y intuitiva para que el usuario pueda usar todas las funcionalidades sin ninguna complicación.
- **Adaptarse al tamaño de pantalla.** La interfaz de usuario debe ser capaz de adaptarse a los distintos tamaños que pueda tener un móvil o *tablet* con sistema operativo Android.
- **Rápido.** Todas las partes del sistema deben ser rápidas para evitar esperas del usuario y que no suponga un tiempo excesivo el uso de sistemas ajenos.
- **Manejo de gran cantidad de datos.** El sistema debe contar con algún tipo de almacenamiento que permita guardar y acceder a una gran cantidad de datos de la manera más rápida y eficiente.
- **Soporte para peticiones simultaneas.** El servidor debe ser capaz de soportar múltiples peticiones simultaneas a la API, para poder responder a la demanda de datos.

6.4 Arquitectura

Después de la obtención de los requisitos y actores involucrados, se ha desarrollado la arquitectura completa del sistema. Para facilitar el mantenimiento del sistema se ha dividido en tres partes claramente diferenciadas. Cada una de ellas contiene una parte de las funcionalidades del sistema y se relaciona con las demás utilizando un mismo protocolo de comunicación. Con ello se consigue que el sistema sea modular, pudiendo cambiar la tecnología implicada en cada módulo sin afectar a los demás.

Las partes en las que se ha divido el sistema son:

- ***Crawler*:** es el encargado de la extracción de los datos de las fuentes de información utilizadas en el sistema. Conoce la estructura HTML de cada una de las fuentes y se encarga de proporcionar los datos extraídos al servidor.

- **Servidor:** proporciona una API REST para recibir los datos enviados por el *crawler* y compartirlos con otros módulos. Además, es el encargado de la autenticación de usuarios, el procesado de los datos y de la comunicación con la base de datos que, por simplicidad, se considerará parte del servidor.
- **Aplicación móvil:** se encarga de habilitar la visualización y filtrado de los datos así como de proporcionar ajustes al usuario. Contiene distintas representaciones de los datos y pone a disposición del usuario nuevos campos que utilizan su ubicación. Para la obtención de los datos se comunica con el servidor a través de la API correspondiente.

El uso de esta arquitectura permite replicar cualquiera de sus componentes, siempre y cuando se trabaje con los mismo datos, permitiendo así que el sistema sea escalable. Así se podría disponer de más servidores que puedan atender una mayor cantidad de peticiones y de *crawlers* que extraigan los datos de distintas fuentes de información simultáneamente en varias máquinas. Obviamente, la escalabilidad ya estaba presente en el uso de una aplicación móvil.

Para una visión más clara de la arquitectura del sistema se ha realizado un esquema que se puede observar en la figura 6.2. En este diagrama de bloques, se muestran las partes del sistema y las relaciones entre ellas y otros sistemas de terceros utilizados.

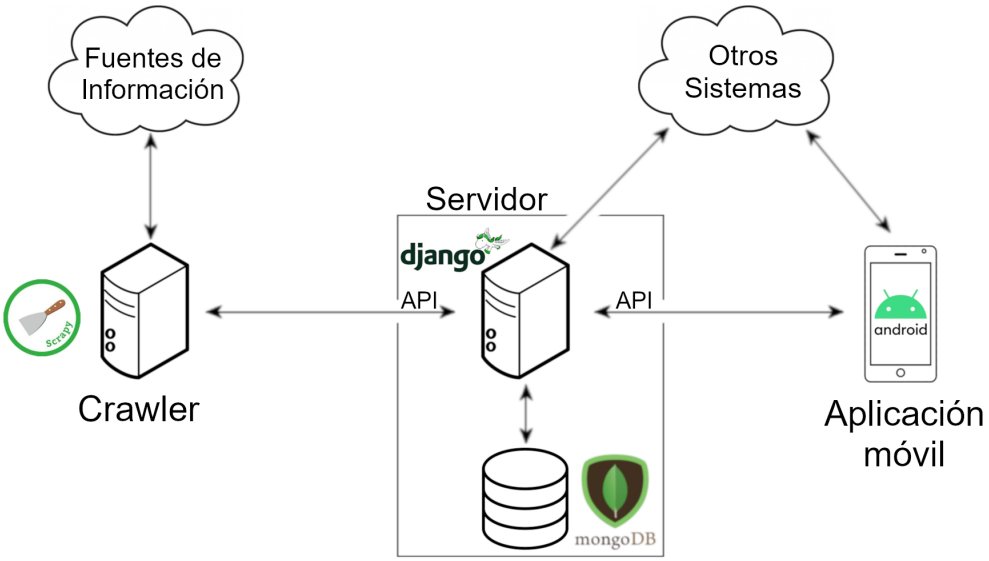


Figura 6.2: Esquema de la arquitectura general del sistema

Capítulo 7

Diseño

Una fase muy importante en el ciclo de vida de un proyecto es el diseño del producto *software*. Esta fase se encarga de diseñar un sistema capaz de cumplir con los requisitos funcionales y no funcionales establecidos en la fase de análisis. El diseño debe facilitar el manejo de la complejidad del sistema a los programadores, utilizando patrones de diseño y dividiendo la estructura en módulos. Así es posible introducir cambios de manera rápida sin afectar al resto de estructuras, facilitando un futuro mantenimiento del sistema.

En este capítulo se mostrarán las decisiones de diseño tomadas en cada una de las partes del sistema.

7.1 Diseño del *crawler*

En esta parte del sistema se ha utilizado el *framework* Scrapy (sección 3.3.1). Este *framework* dispone de una arquitectura propia para facilitar el desarrollo a sus usuarios, la cual ha sido suficiente para cubrir las necesidades de este proyecto.

Al crear un proyecto en Scrapy se genera una jerarquía de archivos, que contiene los ficheros Python en los que hay que establecer el comportamiento que se desea obtener del sistema. Además se genera el directorio *spiders* donde se guardan los *crawlers* (o arañas) creados en el proyecto, que son los encargados de moverse por los enlaces de las *webs* y extraer sus datos. El funcionamiento de los *crawlers* depende de la configuración de los siguientes ficheros generados:

- **items.py:** en este archivo se crean los modelos de los *items* que se van extraer utilizando *scraping* en el *crawler*.
- **middlewares.py:** en este fichero se definen los *middlewares*, necesarios si se requiere procesar la entrada o salida de datos de los *crawlers* y manejar los errores que se puedan producir.

- **pipelines.py:** aquí se definen los *pipelines*. Son en los que se establece que hacer con los *items* extraídos.

- **settings.py:** en este fichero se configura el comportamiento general del sistema, como el nombre del proyecto, los *pipelines* y *middlewares* que se utilizan, etc.

En la definición de cada *crawler* se especifica la *Uniform Resource Locator* (URL) inicial y los elementos de los que se extraen los datos. Por lo que la extracción de los datos del recurso *web* es dependiente a su estructura.

7.2 Diseño del servidor

Para el desarrollo de la parte del servidor se ha utilizado el *framework* Django (sección 3.3.2). Su arquitectura integra el patrón MTV, que es una variante del patrón *Model View Controler* (MVC) con cambios en las funciones de las capas vista y controlador. Las capas de este patrón son:

- **Modelo.** Es la capa de acceso a la base de datos, igual que en el MVC. En esta capa se especifica el modelo de los datos y sus campos, que se definirán en la base de datos sin necesidad de utilizar el lenguaje propio de la misma.

- ***Template.*** Es la capa de presentación, se encarga de la apariencia de los datos en la interfaz *web*. Sería equivalente a la vista del MVC pero contiene rasgos muy diferenciados. La vista en el MVC controla que datos se muestran y como, en cambio el *template* solo decide como mostrarlos.

- **Vista.** Se encarga de la lógica de negocio, como el controlador de el MVC. Procesa las peticiones enviadas al servidor y devuelve el *template* correspondiente.

En la figura 7.1, se puede observar la capas mencionadas anteriormente y sus conexiones. La vista recibe la petición del navegador y pide al modelo los datos, este accede a la base de datos y se los devuelve. Los datos recibidos por el modelo se procesan en la vista y devuelve el *template* que le corresponde al navegador.

Cuando se crea un proyecto en Django se genera automáticamente la estructura de ficheros que debe utilizar el desarrollador para establecer el comportamiento del servidor. Los ficheros con mayor impacto en el funcionamiento de este sistema son:

- **models.py:** en este archivo se definen los modelos de datos y sus campos, que luego se implantarán en la base de datos.

- **views.py:** se definen las vistas del servidor, que son las encargadas de la lógica de negocio. Procesan las peticiones enviadas al servidor y devuelven el resultado correspondiente a su destinatario.

- **urls.py:** aquí se establecen y asocian las rutas del servidor con las vistas creadas en el archivo *views.py*.

- **settings.py:** en este archivo se configura el comportamiento general del sistema. Se establece la base de datos que se va utilizar, las aplicaciones instaladas, la franja horaria del sistema, etc.

- **manage.py:** este fichero no es modificable. Se utiliza para la ejecución de comandos de Django, como el acceso al *shell*, la implantación de los modelos definidos en la base de datos, la creación de un usuario administrador, etc.

Además de estos archivos, el uso del *framework* Django REST, que complementa a Django para la creación de la API REST, proporciona el archivo *serializers.py*. En este archivo se declaran los serializadores de los modelos que se encargan de representar los datos como un JSON o un *eXtensible Markup Language* (XML). De esta forma, se posibilita el envío de datos complejos a otros elementos del sistema. Los serializadores también se utilizan para crear entidades de los modelos y validar si sus datos están bien formados para guardarlos en las base de datos.

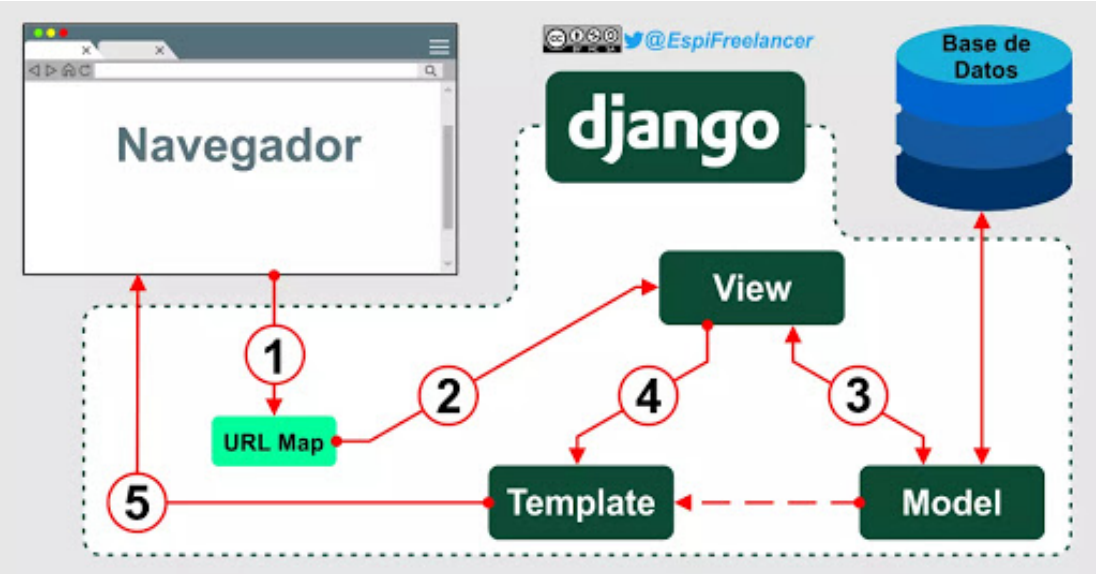


Figura 7.1: Capas del modelo MTV y sus conexiones, imagen obtenida de la *web arquitectura-virtual* [2]

7.3 Diseño de la aplicación móvil

El desarrollo de la aplicación móvil se ha centrado en los dispositivos que utilizan el sistema operativo Android. La estructura de un proyecto Android se distribuye en dos carpetas, una donde se almacenan los archivos de código fuente y otra en la que se guardan los recursos utilizados como imágenes, animaciones, *layouts*, etc.

Para el desarrollo de las aplicaciones, Android cuenta con un SDK con las librerías y herramientas necesaria para la implementación. Muchas de estas librerías se han tenido que tener en cuenta para tomar las decisiones de diseño, que se expondrán en esta sección.

7.3.1 Diseño principal

Para el desarrollo de la estructura principal de la aplicación se ha utilizado el patrón MVVM, recomendado por la documentación de Android. Este patrón divide el funcionamiento de la aplicación en tres capas:

- **Modelo.** Representa la capa de datos y contiene la información usada por el sistema. Se encarga de la obtención de los datos de bases de datos o repositorios web. Esta capa no tiene ningún tipo de dependencia con las demás ni conoce sus estructuras.
- **Modelo de vista.** Contiene la lógica de presentación de los datos. Esta capa interactúa con el modelo y es observada por la vista. Tiene que estar desacoplada de la vista por lo que no debe conocer sus componentes ni depender de ella.
- **Vista.** Su rol es el de observar los cambios en el modelo de vista y actualizar la interfaz de usuario en consecuencia.

Para facilitar la aplicación de este patrón, Android dispone de una clase llamada *ViewModel* con un ciclo de vida propio al que no le afectan las recreaciones de las vistas que se producen al rotar la pantalla. En esas recreaciones se vuelven a crear todas las instancias de la vista y producen errores cuando se utilizan esas instancias como persistentes.

Además para que la vista pueda observar al modelo de vista se utiliza la clase *LiveData*. Esta clase es observable y notifica cambios producidos en su valor, por lo que se utiliza en el modelo de vista para advertir de cambios en los datos a la vista.

En la figura 7.2 se puede observar un diagrama con las comunicaciones entre capas de un ejemplo de arquitectura que usa MVVM. La dependencia de los elementos siempre es hacia a bajo. También se ha incluido la figura 7.3 en la que se muestran las clases de la aplicación que siguen este patrón. En la capa *View* están las vistas que dependen de su modelo de vista situado en la capa *Viewmodel*. En cambio los modelos de vista dependen todos de *ModelFacade* situado en la capa *Model*.

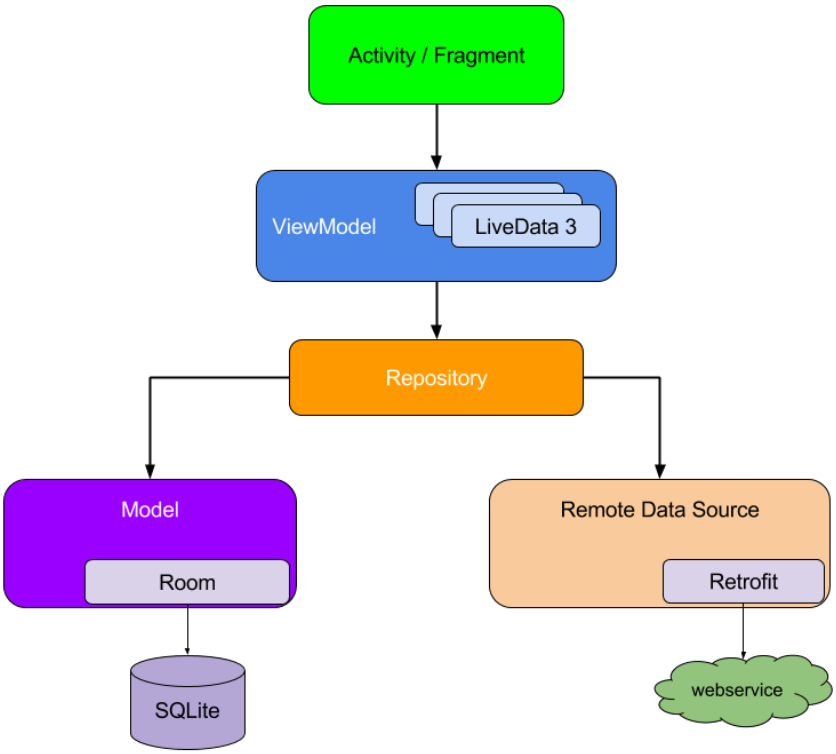


Figura 7.2: Ejemplo de arquitectura que sigue el patrón MVVM, extraída de la web de Android [3]

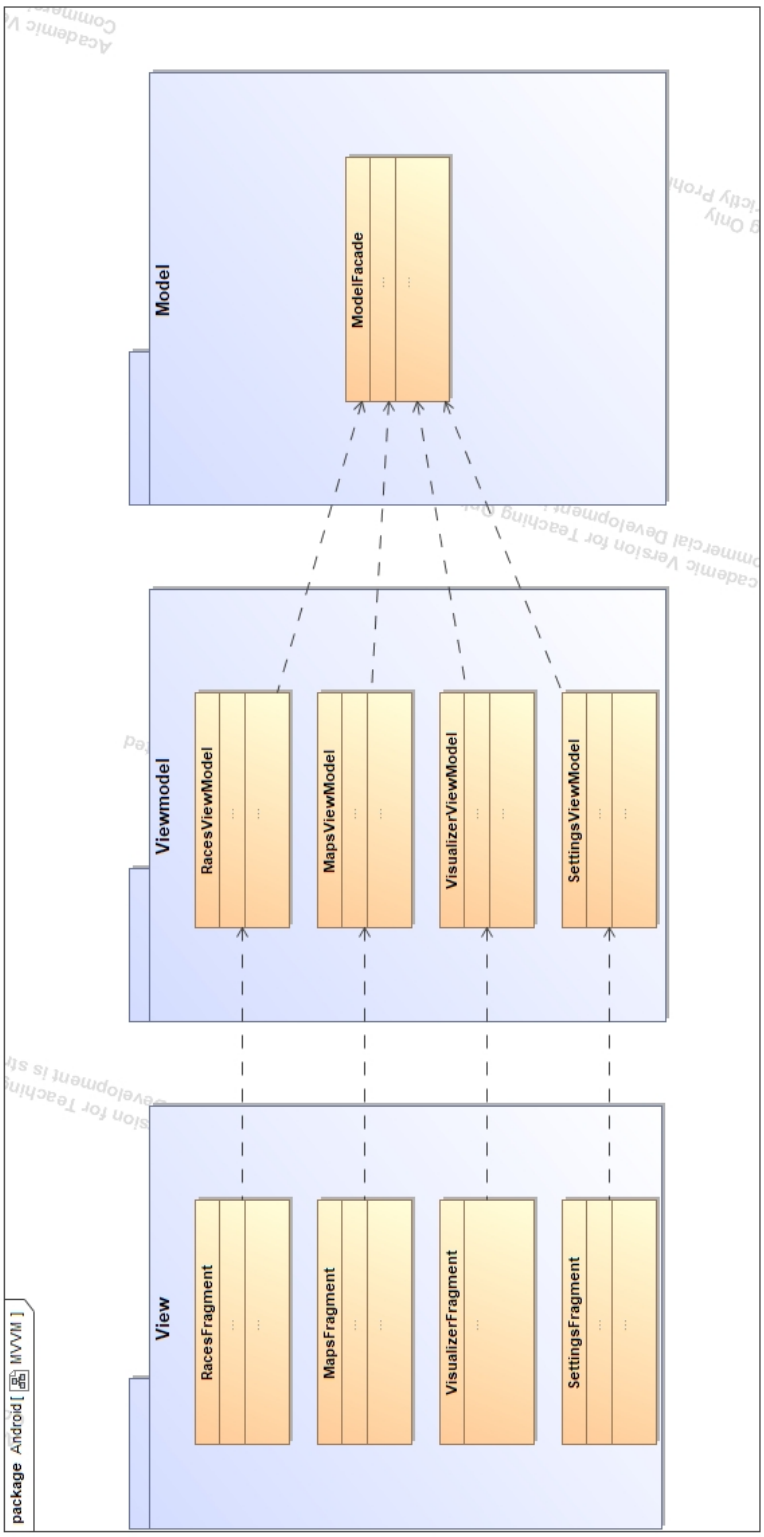


Figura 7.3: Diagrama de clases del patrón MVVM

7.3.2 Diseño del modelo

En la capa modelo de esta aplicación se ha contemplado la necesidad de obtener datos de distintas API. A parte de las carreras obtenidas del servidor es necesario la utilización de API ajenas al proyecto para desarrollar algunas funcionalidades como *Geocoding* API o *DistanceMatrix* API.

Para poder gestionar todas las llamadas a estas fuentes de datos se ha utilizado el patrón *Facade*. Este patrón se aplica ante la necesidad de proporcionar una interfaz simple para un subsistema complejo. Consiste en la utilización de una clase como fachada, en la que en sus métodos se realizan las llamadas a los diferentes subsistemas. Así es posible proporcionar una interfaz sencilla de obtención de datos al modelo de vista.

Para cada API utilizada se ha creado una clase encargada de realizar la petición. Para poder hacer pruebas en la aplicación sin necesidad de realizar una petición al servidor, se necesitaba la creación de un diseño que permitiera intercambiar dos clases. Una que realizara la petición al servidor y otra que imitara el comportamiento de la anterior sin realizar la petición de verdad.

Para resolver esta problemática se utilizo el patrón de diseño *Factory*. Primero se ha creado una jerarquía en la que la clase padre es una clase abstracta de la que heredan la clase que hace las peticiones al servidor y la que imita su comportamiento. Ahora se crea una factoría de la clase abstracta y se utiliza un campo guardado en las *properties* de la aplicación, para obtener el nombre de la clase que se debe instanciar y devolver. Así, cuando sea necesario cambiar la clase que devuelve la factoría, solo habría que modificar en las *properties* el nombre de la clase por el de una de las subclases de la clase abstracta.

Como los datos del modelo deben de estar disponibles para todos los modelos de vista, se ha utilizado el patrón de diseño *Singleton* en la clase fachada del modelo. Utilizando este patrón se asegura que todas las clases utilizan la misma instancia de la clase que lo implementa.

Para una visualización mas clara del diseño, en la figura 7.4 se puede ver el diagrama del modelo con todas las clases implicadas y sus relaciones

7.3.3 Diseño de los filtros

A la hora de añadir los filtros de la aplicación, se tuvo que diseñar una solución para notificar a los modelos de vista que había que realizar cambios en los datos mostrados. Esta notificación no puede ser realizada por la vista directamente sobre el modelo de vista ya que incumpliría el patrón MVVM. Pero la vista es la que gestiona los eventos producidos cuando el usuario interactúa con alguno de sus elementos.

Para resolver este problema se ha utilizado el patrón *Observer*, ya implementado en Java en las clases *Observer* y *Observable*. Este patrón consiste básicamente en que un objeto observa a

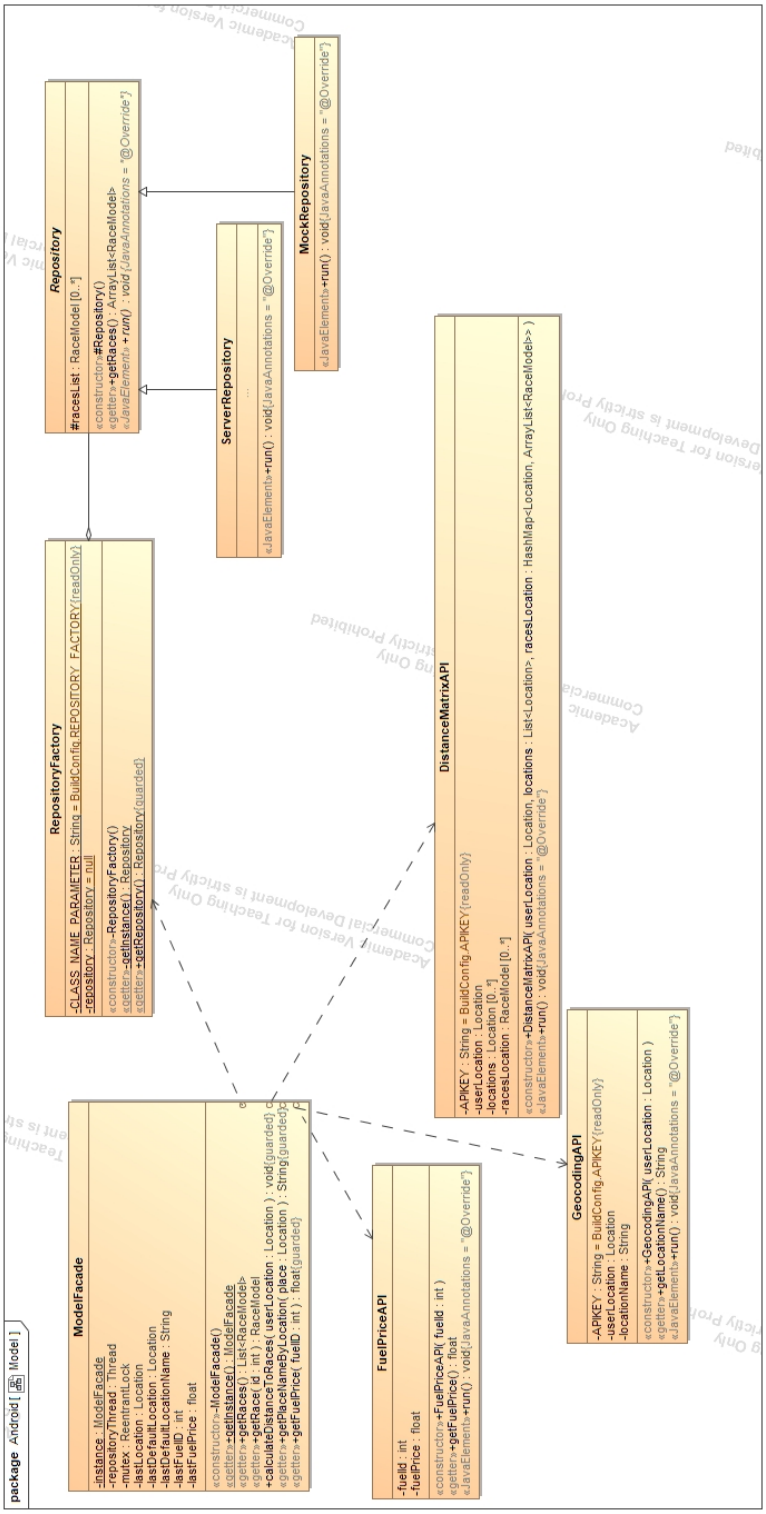


Figura 7.4: Diagrama de clases de la capa modelo

otro, el observador se suscribe al observado y, cuando se produce un cambio en el observado, éste notifica a sus observadores. En este caso, los objetos observados son los filtros y los observadores serían los modelos de vista. De este modo, cuando se produce un cambio en un filtro por la acción del usuario, se notifica a los modelos de vista para que filtren los datos y se produzca un cambio en la vista.

El resultado de aplicar este patrón sería que, en la capa vista, los filtros utilizados se suscriben al modelo de vista. De este modo, cuando el usuario interactúa para cambiar un filtro activa un evento de la vista en el que se cambia el estado del filtro seleccionado. El filtro notifica a sus observadores que son los modelos de vista y estos realizan el filtrado.

Los modelos de vista necesitan guardar los filtros a los que observa y aplicarlos todos si uno de ellos cambia, ya que no puede saber cual de ellos ha cambiado. Para guardarlos todos en una lista y aplicarlos de manera sencilla se ha aplicado el patrón *Strategy*, mediante la creación de una clase abstracta que extiende de *Observable* y de la que heredan todos los filtros. Esta clase tiene un método abstracto que tienen que definir sus descendientes y donde va el algoritmo de filtrado concreto de cada uno. Ese método es el que se utilizará en los modelos de vista para aplicar los filtros.

Como los filtros son comunes a varias partes de la aplicación se ha aplicado además el patrón *Singleton*. Así los cambios realizados en los filtros perduran en las distintas vistas de la aplicación.

Para ver de forma más clara el diseño de esta parte del sistema se ha realizado un diagrama de clases, que se puede ver en la figura 7.5. Además, en esta figura, también se incluye la aplicación del patrón *Strategy* en los modelos de vista que utilizan los filtros usados en otras clases del sistema que se explicarán más adelante. Tanto en la clase abstracta como en la interfaz se han generalizado el tipo utilizado por si fuera necesario en un futuro el filtrado de otras clases siguiendo otros criterios.

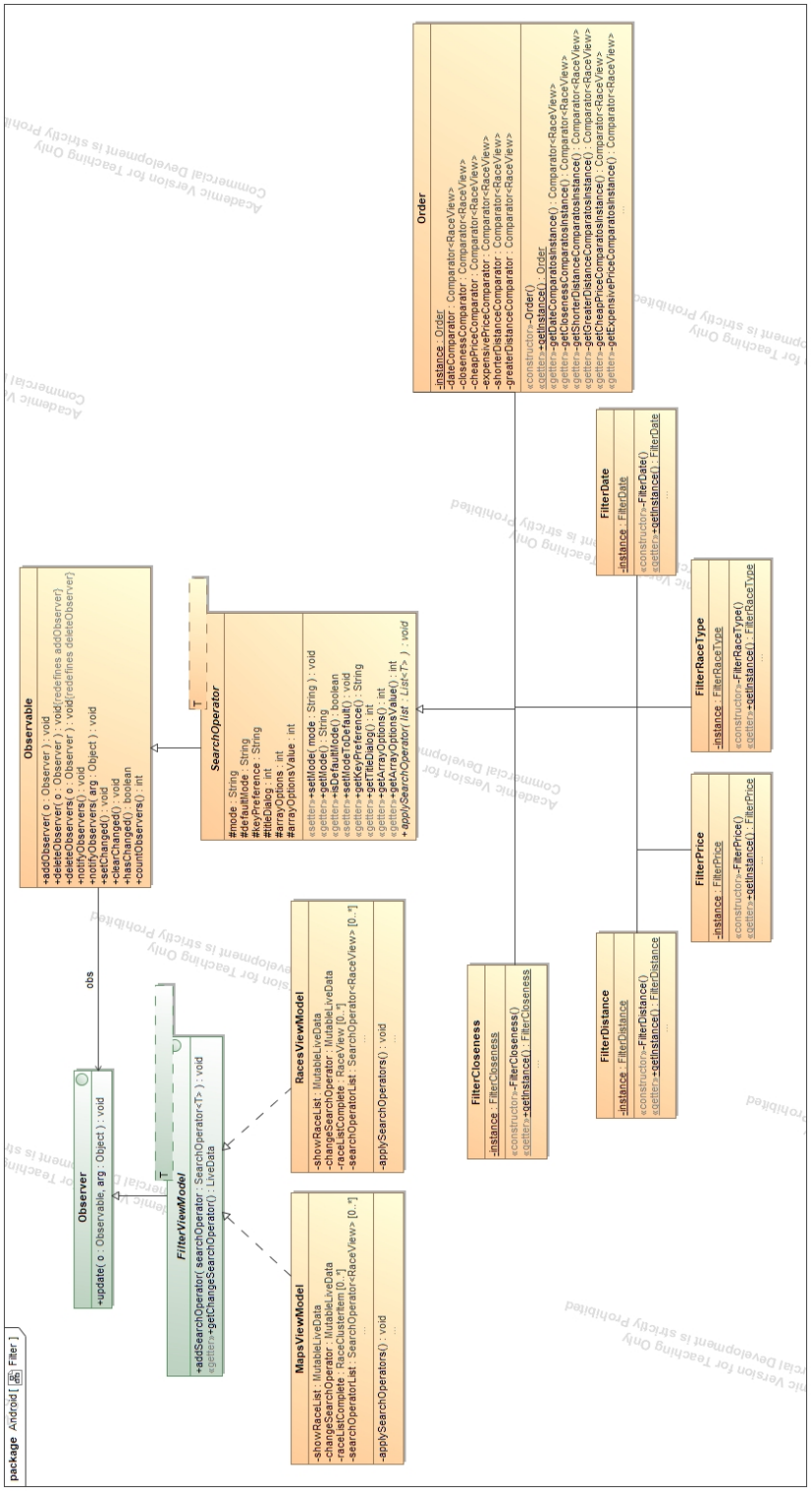


Figura 7.5: Diagrama de clases para el diseño de los filtros

Capítulo 8

Implementación y pruebas

Las fases de implementación y pruebas son las últimas del desarrollo. En la implementación se construye el sistema anteriormente diseñado, produciendo un resultado funcional. Las pruebas en cambio se realizan para poder detectar el mayor número de errores generados en las anteriores fases de desarrollo y así poder corregirlos antes de la entrega final del producto al cliente.

En este capítulo se comentará el trabajo realizado para la construcción del sistema y las pruebas que se le han realizado.

8.1 Implementación del *crawler*

En esta sección se comentará el desarrollo realizado en algunos de los ficheros comentados en el capítulo de diseño (sección 7.1).

8.1.1 *Crawlers*

Para extraer los datos de las fuentes de información se ha creado un *crawler*, por cada fuente utilizada, en la carpeta *spiders* del proyecto Scrapy. Además para no crear dependencias entre los *crawlers* y simplificar el procesado de datos en el servidor, se ha creado un *item* en el archivo *items.py* por cada uno de ellos.

El *framework* se encarga de descargar internamente el contenido de las URL indicadas por el *crawler*, empezando por la URL inicial definida en su fichero de configuración correspondiente. De esta forma, la parte clave consiste en analizar los datos descargados y seleccionar o extraer aquellos datos que son necesarios. Esta extracción se realiza utilizando la ruta XPath de los elementos HTML que se quieren recuperar de los recursos *web*. Estas rutas se han extraído a mano con la ayuda del navegador, analizando la estructura de los elementos HTML.

Correr en Galicia

El foro Correr en Galicia es la fuente de información principal del sistema. Cuenta con un calendario con la mayoría de carreras que se realizan en Galicia. Este calendario se puede observar en la figura 8.1, donde se pueden ver los distintos tipos de eventos que se contemplan en la *web* y las carreras que se celebran en cada día del mes actual. Al seleccionar una de ellas, se muestra la información detallada de ésta, de la que se extraerán los datos con ayuda del *crawler*. La figura 8.2 muestra como se representa la información detallada de una carrera. Como se puede observar, la información de las carreras se presenta de forma totalmente desestructurada e, incluso, el tipo de información y la forma de presentarla varía significativamente de una carrera a otra. Esto dificulta la extracción de los datos de interés de las carreras.

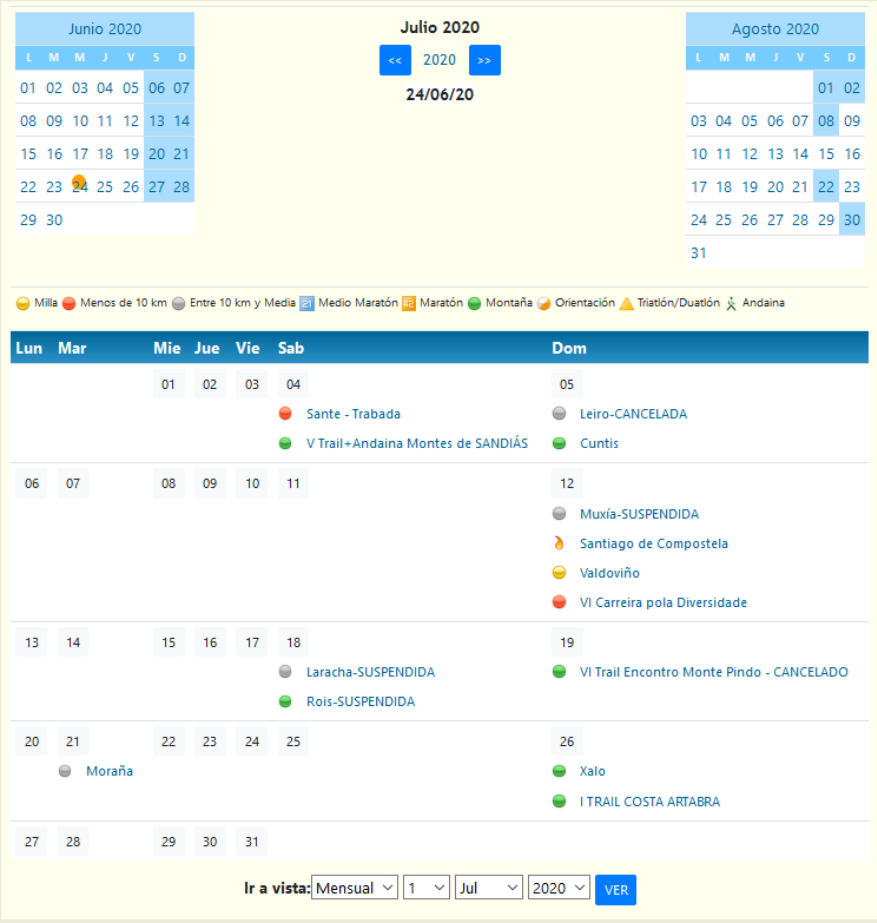


Figura 8.1: Calendario de carreras de Correr en Galicia

Se ha desarrollado un *crawler* llamado *correrengalicia*, que parte de la URL que devuelve el calendario con el mes actual. Esta URL es invariante, solo varía automáticamente su contenido cuando se cambia de mes de manera natural. Del contenido de ese calendario se extraen los

Figura 8.2: Vista detallada de una carrera en Correr en Galicia

enlaces que llevan a la información detallada de cada carrera y el enlace para pasar al siguiente mes.

De cada enlace extraído de una carrera se descarga su contenido y se obtienen los datos para guardarlos en el *item* creado para este *crawler*, en este caso llamado *CorrerEnGaliciaItem*. Este *item* guarda los siguientes campos obtenidos de la vista detallada:

- ***name*:** contiene el nombre de la carrera, normalmente es el nombre del lugar donde se celebra la carrera. Muchas veces el nombre del lugar esta acompañado del nombre de la carrera siguiendo este formato ”A Coruña - I Carrera popular”. En la figura 8.2 este dato sería ”Cuntis”.

- ***race_type*:** contiene la URL de la imagen que indica el tipo de la carrera. En la figura 8.2 se correspondería con la URL de la imagen del círculo verde.

- ***date*:** contiene la fecha de celebración de la carrera. En la figura 8.2 su valor sería ”05/07/2020”, que es el elemento subrayado en azul.

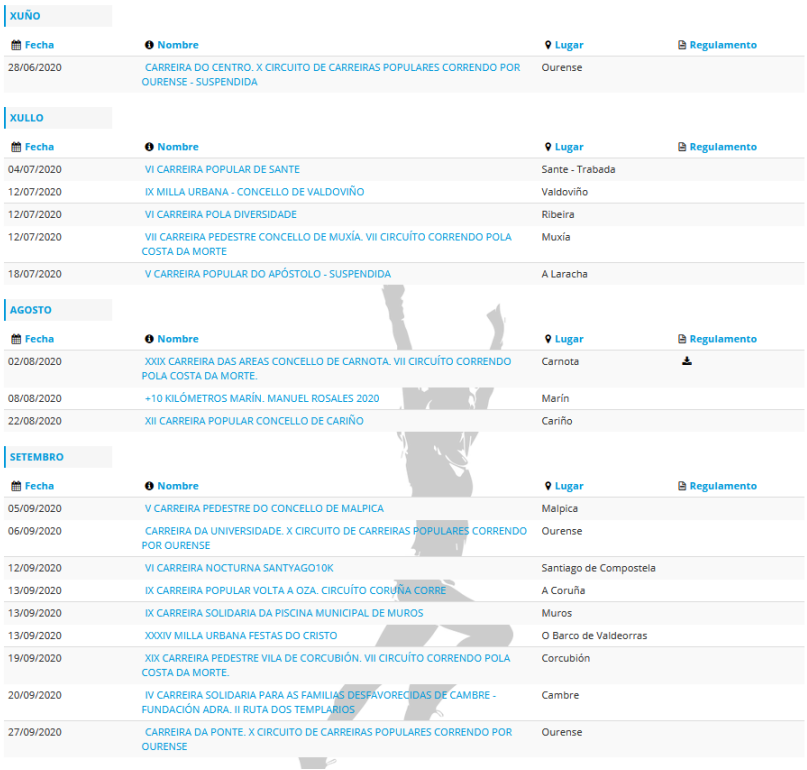
- ***time*:** contiene la hora de inicio y fin del evento separada mediante un guión. Este campo no se presenta en todas las carreras por lo que puede estar en blanco o contener texto no relevante. En la figura 8.2 su valor sería ”9:00 - 14:00”, que es el elemento subrayado en amarillo.

- ***description*:** contiene el texto que detalla aspectos de la carrera, como el reglamento, la distancia, el coste de inscripción, etc. Este campo puede estar en blanco en algunas de las carreras. En la figura 8.2 sería todo el bloque de texto que está por debajo del botón "Volver".

Una vez extraídos todos los enlaces de un mes se pasa al siguiente, utilizando la URL extraída del botón que usarían los usuarios para pasar al siguiente mes. Este proceso se repite hasta haber alcanzado el límite establecido en el código. En este caso, se ha decidido extraer las carreras que se celebrarán a lo largo del mes actual y los once meses posteriores.

Carreiras galegas

La página oficial de la federación gallega de atletismo, Carreiras galegas, dispone de un calendario con todas las carreras federativas celebradas en Galicia en el año actual. En la figura 8.3 se puede ver parte del calendario que se muestra en su página *web*.



XUÑO

Fecha	Nombre	Lugar	Regulamento
28/06/2020	CARREIRA DO CENTRO. X CIRCUITO DE CARREIRAS POPULARES CORRENDO POR OURENSE - SUSPENDIDA	Ourense	

XULLO

Fecha	Nombre	Lugar	Regulamento
04/07/2020	VI CARREIRA POPULAR DE SANTE	Sante - Trabada	
12/07/2020	IX MILLA URBANA - CONCELLO DE VALDOVIÑO	Valdoviño	
12/07/2020	VI CARREIRA POLA DIVERSIDADE	Ribeira	
12/07/2020	VII CARREIRA PEDESTRE CONCELLO DE MUXÍA. VII CIRCUÍTO CORRENDO POLA COSTA DA MORTE	Muxía	
18/07/2020	V CARREIRA POPULAR DO APÓSTOLO - SUSPENDIDA	A Laracha	

AGOSTO

Fecha	Nombre	Lugar	Regulamento
02/08/2020	XXIX CARREIRA DAS AREAS CONCELLO DE CARNOTA. VII CIRCUÍTO CORRENDO POLA COSTA DA MORTE.	Carnota	
08/08/2020	+10 KILÓMETROS MARÍN. MANUEL ROSALES 2020	Marín	
22/08/2020	XII CARREIRA POPULAR CONCELLO DE CARIÑO	Cariño	

SETEMBRO

Fecha	Nombre	Lugar	Regulamento
05/09/2020	V CARREIRA PEDESTRE DO CONCELLO DE MALPICA	Malpica	
06/09/2020	CARREIRA DA UNIVERSIDADE. X CIRCUITO DE CARREIRAS POPULARES CORRENDO POR OURENSE	Ourense	
12/09/2020	VI CARREIRA NOCTURNA SANTYAGO10K	Santiago de Compostela	
13/09/2020	IX CARREIRA POPULAR VOLTA A OZA. CIRCUÍTO CORUÑA CORRE	A Coruña	
13/09/2020	IX CARREIRA SOLIDARIA DA PISCINA MUNICIPAL DE MUROS	Muros	
13/09/2020	XXXIV MILLA URBANA FESTAS DO CRISTO	O Barco de Valdeorras	
19/09/2020	XIX CARREIRA PEDESTRE VILA DE CORCUBIÓN. VII CIRCUÍTO CORRENDO POLA COSTA DA MORTE.	Corcubión	
20/09/2020	IV CARREIRA SOLIDARIA PARA AS FAMILIAS DESFAVORECIDAS DE CAMBRE - FUNDACIÓN ADRA. II RUTA DOS TEMPLARIOS	Cambre	
27/09/2020	CARREIRA DA PONTE. X CIRCUITO DE CARREIRAS POPULARES CORRENDO POR OURENSE	Ourense	

Figura 8.3: Parte del calendario de carreras ofrecido por Carreiras galegas

Para la extracción de los datos de esta fuente se ha creado un *crawler* llamado *carreirasgalegas*, que utiliza la URL del calendario de esta *web* como enlace inicial. La extracción de datos se realiza sobre esa URL sin necesidad de moverse por más enlaces, ya que todo la información

que se va a extraer está en ella. Para guardar los datos extraídos en este *crawler* se ha creado un *item* llamado *CorrerEnGaliciaItem*, que contiene los siguientes campos:

- ***name*:** en el se guarda el nombre oficial de la carrera.
- ***date*:** contiene la fecha de celebración del evento.
- ***place*:** guarda el lugar donde se celebra la carrera.

Los datos en el HTML se presentan en forma de tabla, por lo que para extraerlos se ha ido recorriendo cada fila. Para cada una se crea un *CorrerEnGaliciaItem* y se rellenan sus campos.

8.1.2 *Pipelines*

En un *pipeline* se definen las acciones que se quieren realizar antes y después de ejecutar un *crawler* y cada vez que se procesa un *item*. Normalmente los *pipelines* se utilizan para guardar los datos extraídos en algún archivo en formatos como XML o JSON.

En este sistema se necesitaba una comunicación con el servidor, por lo que se definió un *pipeline* llamado *PostJsonPipeline*. Este se encarga de enviar los datos al servidor en formato JSON realizando una petición POST.

PostJsonPipeline, como todos los *pipeline* de Scrapy, dispone de tres métodos en los que se realiza la obtención de un listado de *items* y la comunicación con el servidor. Los métodos implementados son los siguientes:

- ***open_spider*:** se llama antes de ejecutar un *crawler*. En él se declara una lista vacía para guardar los *items* y se realiza una petición POST al *login* del servidor con las credenciales del usuario *Crawler*. El servidor devuelve en respuesta a esa petición un *token* y un identificador de sesión, que se guardan para enviar en el resto de peticiones. Sin estos datos el servidor rechazará la petición ya que es necesario estar identificado para realizarlas.
- ***process_item*:** este método se llama cada vez que se procesa un *item* en un *crawler*. Aquí solamente se añade el *item* a la lista declarada anteriormente.
- ***close_spider*:** al finalizar la ejecución del *crawler* se llama a este método. Aquí se convierte la lista en un texto con formato JSON y se envía al servidor con una petición POST. Una vez enviados los datos, hay que realizar otra petición POST pero, esta vez, al *logout* para que los datos de sesión ya no sean válidos. En ambas peticiones es necesario incluir el *token* en la cabecera de la petición como *X-CSRFToken* y enviarlo también en las *cookies* junto el identificador de sesión.

Para simplificar el procesado de datos en el servidor se habilitó una dirección exclusiva para el envío de datos de cada *crawler*. Así el servidor sabe que *crawler* le está enviando los datos y que campos son los que envían. Esta dirección acaba con el nombre que se le ha establecido al *crawler* que envía los datos. El nombre del *crawler* es un dato obligatorio en su definición por lo que el *pipeline* puede acceder a él. Entonces el *pipeline* dispone de parte de la dirección del servidor a la que le añade el nombre del *crawler* que se está utilizando. Así consigue enviar los datos a la dirección correcta sin tener que conocer variables internas de cada *crawler*.

8.2 Implementación del servidor

En esta sección se comentará el desarrollo realizado en algunos de los ficheros comentados en el capítulo de diseño (sección 7.2) y las clases propias que se han tenido que utilizar.

8.2.1 *Models*

Los *models* o modelos son los modelos de datos que implanta Django en la base de datos configurada en el servidor. Estos modelos se definen en el archivo *models.py* siguiendo la estructura del proyecto Django.

En este proyecto se ha decidido utilizar como base de datos MongoDB y para poder conectarla al servidor fue necesario utilizar Djongo. Este complemento dispone de las mismas clases para declarar modelos que Django, pero traduciendo las instrucciones a un lenguaje entendible por MongoDB.

El modelo principal creado es *Race*, donde se declaran todos los campos que se van a guardar de las carreras. Algunos de estos campos son compuestos, es decir, guardan otros modelos con sus propios campos. Esta práctica es habitual en formatos como JSON y es una representación válida en MongoDB, por lo que se ha decido utilizarla para facilitar el manejo de los datos y obtener una representación más clara.

En la figura 8.4 se puede ver el modelo de datos con todos los campos de *Race* y la representación de los campos compuestos utilizando relaciones de composición. De los campos mostrados, los únicos obligatorios son *id*, *name*, *date* y *race_type*. El campo *id* es autoincremental y es añadido por Django de manera automática.

Para conseguir declarar como parte del modelo los campos compuestos *price*, *distance* y *location* se han creado dos modelos abstractos llamados *Location* para el campo *location* y *FloatRange* para los campos *price* y *distance* que comparten la misma representación. Estos modelos son implementados de manera normal, declarando en el caso de *FloatRange* sus campos *floats* min y max y, en el caso de *Location*, los campos *latitude* y *longitude* que también son *floats*. Lo único que cambia con respecto a la implementación del modelo *Race* es que se

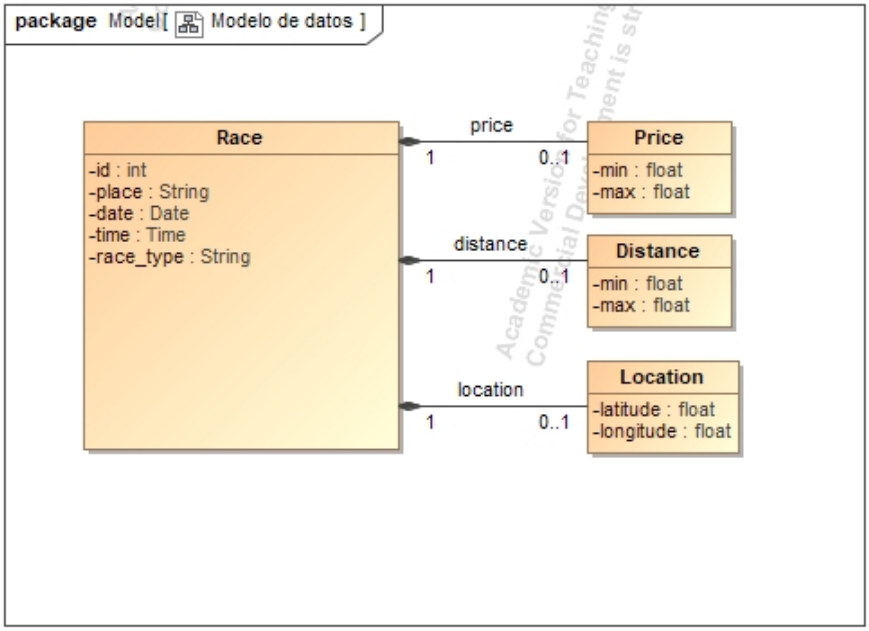


Figura 8.4: Diagrama del modelo de datos

debe de indicar explícitamente que son abstractos. Así se le indica a Django que no tienen que ser implantados en la base de datos como otras entidades. Además deben ser abstractos para poder añadirlos en el modelo *Race* como *EmbeddedField* (clase propia de Djongo) y así conseguir que se representen como campos compuestos.

En la figura 8.5 se puede ver un ejemplo de como sería la representación de todo los campos del modelo *Race* en la base de datos.

Como el servidor dispone de una interfaz web, proporcionada automáticamente por Django, que permite crear y modificar las carreras al usuario *admin*, se han creado en el *models.py* dos *ModelForm* para los campos compuestos. Si no se añadieran estos *ModelForm* como parámetros en los *EmbeddedField* del modelo *Race*, Django no entendería como representar los campos compuestos en la interfaz, por lo que no se podría visualizar correctamente estos campos y no se podrían rellenar ni modificar. Los *ModelForm* creados han sido *FloatRangeForm* y *LocationForm*, y estos elementos se han añadido a los campos que utilizan los modelos *FloatRange* y *Location*.

8.2.2 *Serializers*

Los *serializers* son una clase introducida en Django cuando se utiliza el *framework* Django REST y se declaran en el fichero *serializers.py*. Se utilizan en el proyecto para transformar los datos de las carreras a formato JSON y así poder enviarlos a los usuarios de la API. También sirven para crear nuevas entidades de los modelos en las vistas, con la restricción de tener que

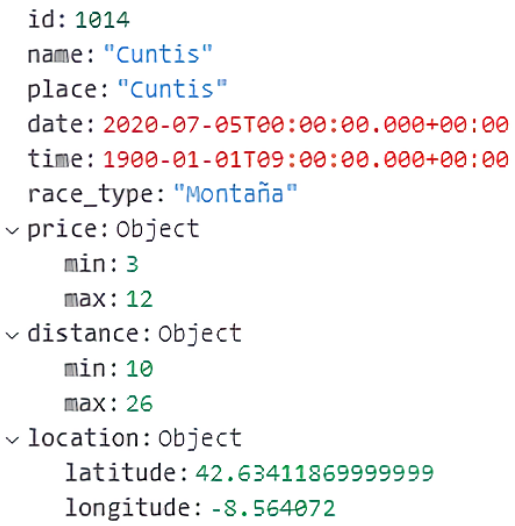


Figura 8.5: Ejemplo de la representación de una carrera en MongoDB

utilizar el método de validación de datos que proporcionan antes de poder guardar la nueva entidad.

Para enviar los datos de las carreras como JSON, se ha declarado un *serializer* llamado *RaceSerializer*. En él se indica el modelo de datos que debe transformar a JSON, en esta caso el modelo *Race* comentado en la sección anterior (sección 8.2.1).

Los campos compuestos del modelo se visualizarían como un *string* si no se añade nada más al *RaceSerializer*. Por este motivo, fue necesario declarar los *serializers LocationSerializer* y *FloatRangeSerializer*. En ellos se tuvo que declarar los campos de los que se componen, los mismos que sus modelos abstractos, y asociarlos en el *RaceSerializer* a los campos compuestos que les corresponden, es decir, *LocationSerializer* para el campo *location* y *FloatRangeSerializer* para los campos *price* y *distance*. Con eso se consigue que se representen de forma correcta, como campos que almacenan a su vez otros campos.

Como aclaración, en el *RaceSerializer* no es necesario declarar los campos porque estos ya se extraen del modelo indicado, en este caso *Race*. En cambio esto no es posible en los otros *serializers* porque sus modelos son abstractos.

8.2.3 *Views*

Las *views* o vistas son la parte del servidor que se ocupa del procesado de las peticiones, y se declaran en el archivo *view.py*.

Para crear una API REST hay que utilizar las clases vista que proporciona Django REST. Con ellas se pueden devolver explícitamente códigos de estado *Hypertext Transfer Protocol*

(HTTP) y procesar las peticiones POST, GET, PUT, DELETE, etc.

En la tabla 8.1 se puede ver la asociación de las direcciones del servidor con las vistas que componen la API REST, las cuáles se explicarán en detalle a continuación. Solo se mencionaran en las explicaciones los códigos de estado HTTP que se han utilizado explícitamente en la implementación, por lo que no se mencionarán los que devuelva Django de manera automática.

Dirección	**Nombre de la vista**
api/v1/carreras	RaceViewSet
api/v1/correrengalicia	CorrerEnGaliciaAPIV
api/v1/carreirasgalegas	CarreirasGalegasAPIV
accounts/login	LoginAPIV
accounts/logout	LogoutAPIV

Tabla 8.1: Direcciones asignadas a las vistas del servidor

RaceViewSet

Esta vista es la encargada de proporcionar los datos de las carrera almacenados cuando se realiza una petición GET. Para ello solo es necesario proporcionarle el *RaceSerializer* y el *queryset* que contiene todas las carreras. Este elemento se puede obtener llamando al método *all()* del modelo *Race*.

Esta vista está disponible para cualquiera, no es necesario estar identificado en el servidor. Además proporciona de manera automática un HTML donde se pueden ver los datos guardados para las diferentes carreras desde un navegador *web*. Para pedir los datos en formato JSON hay que añadir a la dirección asociada a la vista "/?format=json" o utilizar las cabeceras de una petición HTTP para indicar que se quiere obtener en JSON.

LoginAPIV y *LogoutAPIV*

Estas vistas han sido creadas para que el *crawler* pudiera abrir y cerrar sesión en el servidor. No disponen de interfaz gráfica ya que no están orientadas para el uso humano. Para poder gestionar los usuarios e identificarlos se ha tenido que habilitar esta funcionalidad en el *settings.py*.

Para identificarse en el servidor es necesario realizar una petición POST a *LoginAPIV*, pasando el usuario y contraseña en formato JSON como *username* y *password*. La vista se encarga de comprobar si el usuario es válido, devolviendo el código HTTP 200 si existe y en caso contrario el 404. Si el usuario existe, en la respuesta se incluye en las *cookies* el *token* y el identificador de sesión. Estos elementos son necesarios para acceder al servidor mediante los

métodos que requieren estar identificado. Obviamente para realizar peticiones a esta vista no es necesario estar identificado.

Cuando ya no es necesario seguir identificado en el servidor, se envía una petición POST a la vista *LogoutAPIV*. Esta vista se encarga de cerrar la sesión enviada, invalidando el *token* y el identificador de usuario. Devuelve el código HTTP 200 si se ha cerrado la sesión con éxito. Las peticiones a esta vista están restringidas para usuarios que no estén identificados, por lo que es necesario el envío de el *token* y del identificador de sesión.

CorrerEnGaliciaAPIV

En esta vista se reciben y procesan los datos extraídos de Correr en Galicia por el *crawler*. El envío de los datos se realiza con una petición POST y es necesario estar identificado. Los datos recibidos son los mencionados en la (sección 8.1.1). Esta vista devuelve como respuesta el código HTTP 201, que indica que se ha creado todo exitosamente, o el código 400 si ha habido algún error procesando los datos.

Antes de nada, es importante mencionar que si el procesamiento de los datos termina sin incidencias, se borran las carreras que estaban almacenadas en la base de datos antes del procesado. Se ha decidido hacer esto porque no se dispone de un identificador único para las carreras extraídas. Por lo tanto, si a la misma carrera le cambiaran en la fuente de información por ejemplo el nombre y/o la fecha, sería imposible identificarla en la base de datos y modificarle esos campos. De esta forma, se generarían un duplicado de la misma carrera y la anterior quedaría con datos incorrectos.

Otra solución para este problema sería dar de alta todas las carreras que no estén ya almacenadas y borrar las que no coincidan con ninguna de las extraídas. En esta solución se perdería mucho tiempo de procesamiento comparando las carreras extraídas con las ya almacenadas, por lo que se ha decidido que es mejor implementar la solución anterior.

Debido a que en Djongo no se han implementado las transacciones, se ha desarrollado una pequeña solución para que, si ocurre algún error durante el procesado, no se borren los datos actuales. Esta solución consiste en guardar en una lista al principio del procesamiento las carreras actuales y en otra las que se vayan procesando. Entonces si ocurre algún error durante el procesamiento se borran las carreras nuevas de la base de datos y si todo sale bien se borran las carreras que ya estaban al principio.

Al inicio del procesado se codifican los datos recibidos en la petición como un JSON y se van procesando uno a uno. Para empezar, se convierte el texto de la fecha recibido en un objeto *Date* y se comprueba que la fecha sea igual o posterior a la actual. En caso de que esta fecha anterior, se descarta y se pasa a la siguiente. Después se utiliza la clase *RaceType* para extraer el tipo de carrera a partir de la URL de la imagen que indica su tipo en la *web*. Esta clase se queda con el nombre de la imagen, eliminando la ruta y la extensión. Ese nombre se

inserta como clave en un diccionario en el que están almacenados los tipos de carreras que se quieren guardar en el sistema. Si el nombre no está registrado en el diccionario se produce un error y se descarta este elemento, ya que no es un tipo contemplado por el proyecto.

Los tipos de carrera que se han contemplado en este proyecto son: milla, menos de 10km, entre 10km y 21km, media maratón, maratón y montaña. Por comodidad, se han utilizado los mismos que en Correr en Galicia descartando los tipos que no implican única y exclusivamente la actividad de correr.

Una vez obtenido el tipo de la carrera se convierte la hora de inicio a un objeto *Time*, si la carrera dispone de ella. En caso contrario se estable como nula. Ahora se obtiene el nombre de la carrera y el lugar de celebración utilizando la clase *RaceNamePlace*. A esta clase se le envía el nombre extraído de la página de la carrera y contempla que ese *String* siga el formato ”Lugar - Nombre - Estado” o cualquiera de las variantes contempladas como ”Lugar”, ”Lugar - Nombre” y ”Lugar - Estado”. Utilizando una serie de reglas predefinidas, esta clase proporciona el nombre y el lugar de la carrera correspondientes. El estado es un indicativo de si la carrera esta aplazada, suspendida o cancelada y se tiene en cuenta para que esto no se devuelva como el nombre de una carrera. A veces, en primer lugar aparece el nombre de una organización como *CorreSan* que organiza un circuito de carreras en Santiago de Compostela. Esta casuística se ha contemplado en la definición de las reglas, sustituyendo este nombre por el lugar que le corresponde.

Ahora que se dispone del lugar de celebración de la carrera, se utiliza la clase *Geolocalice* para hacer la petición a *Geoconding* API y obtener las coordenadas del lugar de celebración. Para ahorrar peticiones a la API, se ha creado un diccionario en el que se utiliza como clave el nombre del lugar en minúsculas y se guarda como valor una entidad del modelo *Location*. Así, cada vez que se llama al método de la clase, se comprueba antes de realizar la petición si el lugar ya existe en el diccionario. Cada vez que se tiene que realizar la petición a la API se crea un *thread*, encargado de guardar el resultado en el lugar asignado en el diccionario.

Al finalizar el procesado de las carreras, se espera a que acaben todos los *threads* para recorrer el diccionario. Por cada clave del diccionario se hace una consulta en la base de datos para obtener las carreras cuyo lugar de celebración coincida con esa clave. Para aquellas carreras que coincidan con la búsqueda se guarda la localización asociada a la clave en el campo *location*. Si la localización obtenida es nula quiere decir que el lugar no existe, por lo que se establece el campo *place* a nulo.

Para que *Geoconding* API no devuelva resultados de otros países, se ha configurado para que los resultados obtenidos solo puedan ser dentro de España. Además, ante la devolución de varios resultados para un mismo lugar, tendrán preferencia los situados en Galicia.

Para finalizar el procesado de una carrera, se utiliza la clase *RacePriceDistance*. En ella se filtra el texto descriptivo de la carrera, utilizando expresiones regulares, para obtener los costes

de la inscripción y las distancias de la carrera. Como una carrera puede tener distintos costes de inscripción y distancias, se han guardado en el modelo *Race* como rangos. En concreto, se guarda el valor mínimo y máximo obtenidos en el filtrado.

En las expresiones regulares se ha contemplado el uso de números seguido de la palabra euro o el símbolo €, aplicando lo mismo para los kilómetros y lo metros. Si el filtrado por kilómetros no devuelve ningún dato se filtra con los metros y se pasa el resultado a kilómetros. En el filtrado para metros, no se consideran distancias menores a un kilómetro ya que, normalmente, se corresponde con actividades paralelas para niños pequeños.

Además, en el caso de la distancia, para tipos de carrera como la milla, la media maratón y la maratón, este campo se establece directamente sin realizar el filtrado ya que estas carreras tiene una distancia fija y conocida. En el caso de no obtener ninguna distancia se establece el rango definido para el tipo de carrera que es. Por ejemplo, para el tipo ”menos de 10km”, el mínimo sería un kilómetro y el máximo nueve, y lo mismo aplicaría al tipo ”entre 10km y 21km”. El tipo ”montaña” es al único al que no se le pude aplicar esto debido a que puede ser cualquier distancia, por lo que en este caso la distancia se establece como nula.

CarreirasGalegasAPIV

Esta vista recibe los datos extraídos por el *crawler* de Carreiras galegas. El envío de los datos se realiza con una petición POST y es necesario estar identificado. Los datos recibidos son los mencionados en la (sección 8.1.1).

Primero se codifican los datos recibidos como un JSON para poder recorrerlos uno a uno. En este caso los datos no necesitan ser procesados, exceptuando la fecha que se convierte en un objeto *Date*. Una vez la fecha ha sido convertida, se comprueba si dicha fecha es anterior al día actual y, si es así, se descarta la carrera y se procede con la siguiente. Si no se cumple lo anterior, se utiliza la fecha para realizar una consulta a la base de datos y obtener las carreras que se celebran en esa fecha.

Por cada una de las carreras obtenidas en la consulta se compara su lugar de celebración con el lugar de la carrera que se está procesando. Si son iguales, se modifica el nombre de la carrera guardada en la base de datos por el nombre oficial de la carrera. Considerando así que son la misma carrera, ya que es muy improbable celebrar más de una carrera al día en un mismo lugar. Con todo esto se consigue sustituir los nombres informales obtenidos de Correr en Galicia por los nombres oficiales de las carreras.

Si todo el proceso transcurre sin ningún incidente se devuelve el código HTTP 200.

8.2.4 *Admin*

El usuario administrador es una función disponible en todos los proyectos Django. Puede ser utilizada para modificar datos de los modelos y gestionar los permisos de los usuarios y

grupos en el servidor. Dispone de la dirección "/admin", que cuenta por defecto con un *login* para identificar al usuario administrador. Una vez identificado se accede a una interfaz *web* con todas las funcionalidades que dispone el administrador.

En este proyecto se ha utilizado el administrador para crear al usuario *crawler*. Así fue posible limitar el acceso a las funciones de la API que crean o modifican las carreras solo a él, puesto que en las vistas es necesario limitar el acceso a usuarios identificados y en este sistema no existen más. También se ha utilizado para crear algunas carreras necesarias para realizar pruebas.

En la figura 8.6 se puede observar la página principal de la interfaz *web* que proporciona Django al administrador. También, en la figura 8.7, se puede ver el formulario para crear una carrera, donde se pueden rellenar los campos compuestos gracias a los *ModelForm* comentados anteriormente en la sección donde se explicaban los *Models* (sección 8.2.1).

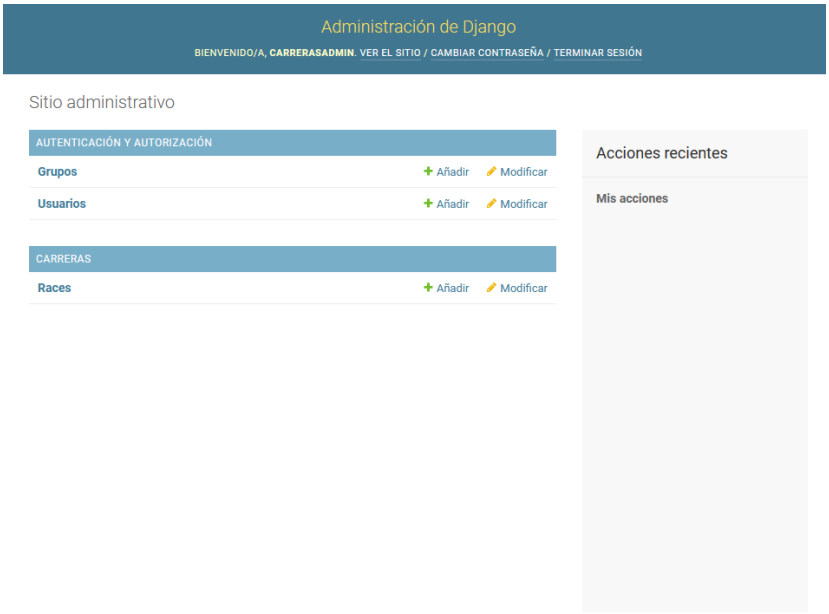


Figura 8.6: Pagina principal de la interfaz *web* del administrador

8.3 Implementación de la aplicación móvil

En esta sección se comentará el desarrollo de las partes de la aplicación Android. Se explicará en detalle la implementación de las decisiones de diseños mencionadas en el capítulo anterior (sección 7.3).

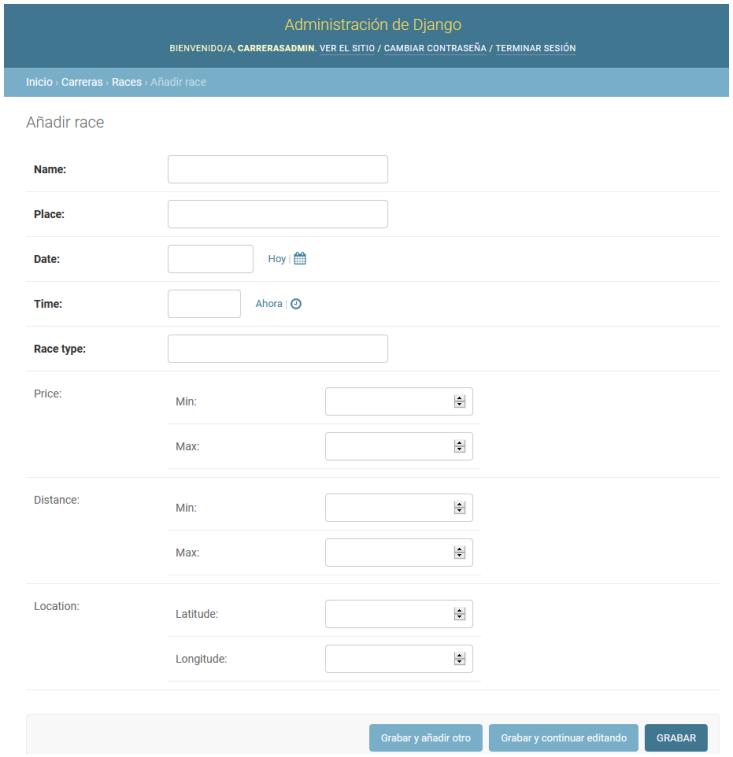


Figura 8.7: Formulario para crear una carrera de la interfaz *web* del administrador

8.3.1 Modelo

En el capítulo de diseño (sección 7.3.2), se ha expuesto la estructura de esta capa y las clases que la componen, las cuáles se explicarán en detalle a lo largo de esta sección. Para facilitar la explicación de la implementación se hará mención a las clases y métodos mostrados en el diagrama de clases de la figura 7.4.

En la capa modelo se han creado las clases *RaceModel* y *Location*, utilizadas para guardar los datos de las carreras. Estas clases fueron creadas para ser independientes del resto de la aplicación, por lo que solo sufren cambios si se ha modificado algo en la capa modelo. De esta forma, se guarda una representación fiel de los datos obtenidos, dejando las diferentes representaciones de estos datos al resto de capas de la aplicación. En la figura 8.8 se puede ver el diagrama de clase ampliado para estas dos entidades.

Una de las funcionalidades más importantes en la aplicación es poder listar las diferentes carreras que se van a celebrar. Para obtener la lista de *RaceModel* de la capa modelo hay que llamar al método *getRaces()* de la clase *ModelFacade*. Este método crea un *thread* con una instancia de la clase abstracta *Repository* que extiende de la clase *Thread*. Esta instancia es devuelta por *RepositoryFactory* y puede ser de *ServerRepository*, clase que obtiene las carreras del servidor, o de *MockRepository*, creada para imitar el comportamiento de una petición al

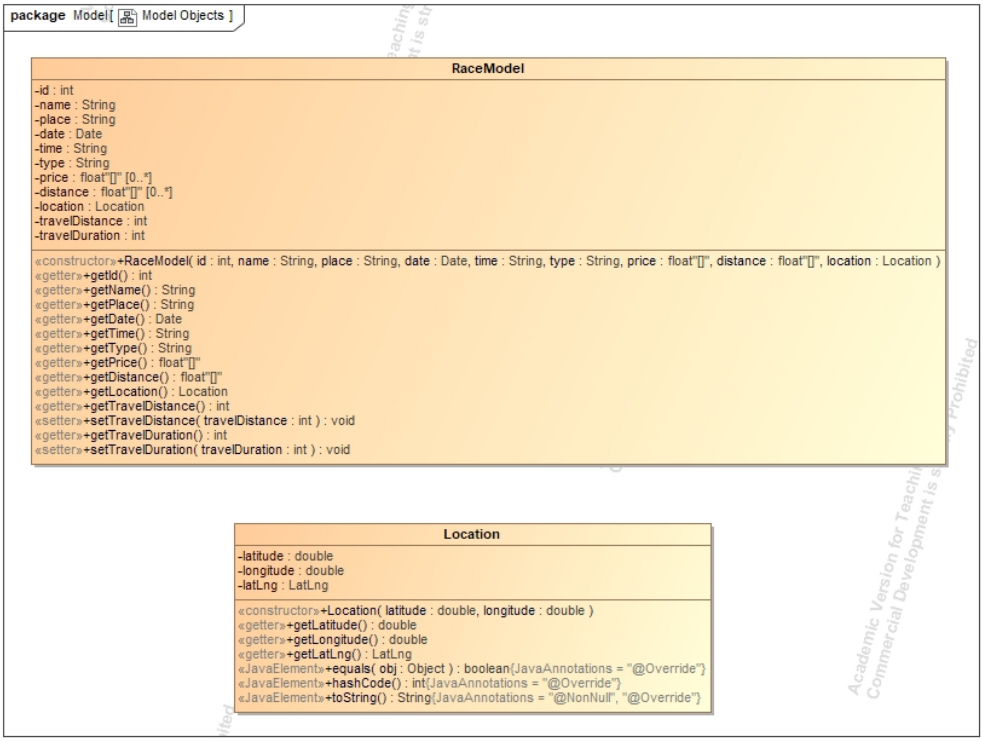


Figura 8.8: Diagrama de clases de *RaceModel* y *Location*

servidor y poder así hacer pruebas en la aplicación. Después de ejecutar el *thread* creado en este método y esperar a su finalización, se obtiene la lista de *RaceModel* de la instancia de *Repository* pasada al *thread*.

El método *getRaces()* de *ModelFacade* puede ser llamada por varios *threads* de la aplicación. Estos causan la creación de un *thread* por cada llamada al método, generando peticiones innecesarias, ya que con una bastaría, y tiempos de espera más largos para el usuario. Para solucionar esto se ha decidido guardar en *ModelFacade* la instancia del último *thread* creado en el método. Así se puede comprobar si el *thread* guardado sigue ejecutándose. De ser así, se pondría a los *threads* de la aplicación a esperar por su finalización. En caso contrario, si la lista devuelta está vacía, se crearía un nuevo *thread*. Esta comprobación tiene que ser *threadsafe* por lo que se ha utilizado un *mutex* para realizarla.

Está situación podría ocurrir si, por ejemplo, un usuario entrara en la aplicación y la primera pantalla hiciera una llamada a este método y, en ese mismo momento, el usuario cambiase a otra pantalla que también necesita llamar a este método. Si el *thread* tarda treinta segundos en finalizar y el usuario cambia de pantalla a los diez segundos, solo necesita esperar en la segunda pantalla veinte segundos por los datos. En cambio, si no se realizara esta mejora, el usuario tendría que espera por los datos cuarenta segundos desde que entra en la aplicación, los diez que tardo en cambiar de pantalla y los treinta del *thread* que creo la segunda pantalla.

Por lo que se tardaría más en disponer de los datos.

Cabe destacar que, en la clase *ServerRepository* mencionada anteriormente, se hace la petición GET a la API correspondiente de Django y se obtienen las carreras en formato JSON. Para transformar estos datos rápidamente en una lista de *RaceModel* se ha utilizado la librería GSON (sección 3.3.3). Para poder utilizar esta librería, se ha tenido que crear la clase *RaceModelDeserializer* que implementa *JsonDeserializer*. En esta clase se ha definido el método *deserialize* donde se realiza la transformación de los datos en una instancia de *RaceModel*. Pasándole la clase *RaceModelDeserializer* y el JSON a los métodos de la librería GSON, se consigue una lista de *RaceModel* de manera rápida y más visual.

Otro aspecto importante para el usuario de la aplicación es conocer la distancia desde su posición a cada una de las carreras de la lista. El método *calculateDistanceToRaces(Location userLocation)* de *ModelFacade*, se encarga de obtener la distancias y los tiempos del viaje por carretera que hay desde la localización que se le pasa como parámetro a cada una de las carreras. Este método no devuelve nada porque modifica directamente las instancias de la lista de *RaceModel* obtenida del servidor. Por lo que con tener una referencia a esa lista ya se obtienen los cambios realizados. El cálculo de las distancias se obtiene utilizando la clase *DistanceMatrixAPI* que realiza peticiones a la *DistanceMatrix* API. Las peticiones a esta API están limitadas a veinticinco destinatarios por petición. Por esa razón, se han tenido que crear *threads* de diferentes instancias de la clase *DistanceMatrixAPI*, pasándoles como máximo veinticinco carreras.

Para minimizar el número de peticiones se han tomado además dos decisiones. La primera de ellas es guardar la última localización calculada en el *ModelFacade*. De esta forma, si se vuelve a llamar a este método con la misma localización no se haría nada. La otra es utilizar un *HashMap* que tiene como clave una instancia de la clase *Location* y como valor una lista de *RaceModel*. Así, solo se procesará la localización de las carreras situadas en las mismas coordenadas una vez, y aplicando el resultado después a todos los *RaceModels* de la lista con esas coordenadas.

También se debe mencionar que en la clase *DistanceMatrixAPI* se convierten las coordenadas de las localizaciones en una *polyline*, utilizando la clase *PolyUtil* de Google, que es una representación de todas las localizaciones que ocupa menor número de caracteres en la petición. Esto se hace debido a que también está limitada la longitud de las peticiones.

Para obtener el nombre de una localización es necesario utilizar el método *getPlaceNameByLocation(Location place)* de la fachada. Este método crea un *thread* con una instancia de la clase *GeocodingAPI* que se encarga de obtener el nombre de la localización que se le pasa al método. La petición realizada por la clase *GeocodingAPI* está configurada para que devuelva un nombre del lugar aproximado y no muy extenso para una mejor visualización. De forma similar a lo que se ha comentado anteriormente, para ahorrar peticiones innecesarias se ha

guardado en la fachada la ultima localización utilizada en este método y su resultado. Por lo que si se vuelve a llamar con la misma localización se devuelve directamente el resultado.

Por último, otro punto importante es determinar los costes de desplazamiento del usuario hasta una determinada carrera. Para ello, en la clase *ModelFacade* se puede obtener el precio de un determinado tipo de carburante utilizando el método *getFuelPrice(int fuelID)*. En este método se lanza un *thread* con una instancia de la clase *FuelPriceAPI* que tiene el *id* del tipo de carburante. En la clase *FuelPriceAPI* se obtiene el precio del carburante indicando en todas las áreas de servicio de Galicia. Se ha pensado que lo más lógico era hacer la media de los precios y devolverla, que es lo que se ha implementado. Igual que en los casos anteriores, se guarda en la fachada el *id* y el precio del último carburante obtenido. En el caso de que se repita una petición con el mismo identificador que el guardado, se devuelve el resultado ya calculado.

La clase *ModelFacade* también dispone del método *getRace(int id)*, que simplemente recorre la lista de *RaceModels* y devuelve el *RaceModels* que coincida con el *id* que se pasa como parámetro. Este método resulta útil para acceder a la información detallada de una carrera en particular.

8.3.2 *MainActivity*

Es la clase principal de la aplicación, y en ella comienza la ejecución del código. Además, dispone del *layout* base de toda la aplicación. Un *layout* es el fichero donde se establecen los componentes gráficos de la interfaz de usuario. Este *layout* contiene un *BottomNavigationView* y un *FragmentContainerView*. Un *BottomNavigationView* es una barra de navegación situada en la parte inferior de la pantalla y un *FragmentContainerView* es un contenedor que permite mostrar e intercambiar distintos *fragments* en la vista de la aplicación.

El *BottomNavigationView* se utiliza para poder desplazarse por las tres vistas principales de la aplicación que son la lista de carreras, el mapa con marcadores y los ajustes. Para que el *fragment* mostrado en el *FragmentContainerView* cambie cuando se interactúa con el *BottomNavigationView*, se debe establecer la relación entre estos dos componentes junto con un gráfico de navegación en el *NavigationUI*. Estas operaciones se realizan en la creación de la clase *MainActivity*, estableciendo la navegación entre los distintos *fragments* de la aplicación y conectando el comportamiento entre los dos componentes mencionados.

La clase *MainActivity* también se encarga de pedir al usuario el permiso de ubicación, si no ha sido concedido. Este permiso es necesario para poder trabajar con la ubicación actual del dispositivo en varios *fragments* de la aplicación.

Además se encarga de establecer en los filtros el modo seleccionado en las preferencias por el usuario para que, al inicio de la aplicación, ya se apliquen los filtros deseados. También, la primera vez que se arranca la aplicación o cuando se borran sus datos, se tiene que establecer

los valores por defecto de alguna preferencias, comentadas más adelante en ajustes. Estos valores por defecto se guardan en el fichero *gradle.properties* y se acceden a ellos utilizando la clase *BuildConfig*. Esta clase se genera cada vez que se hace un *build* de la aplicación y guarda los valores de los campos de *gradle.properties*, que se indiquen en el *build.gradle*, como variables de clase.

8.3.3 Lista de carreras

En esta sección se comentará la implementación de la pantalla que muestra la lista de carreras. En esta implementación están implicadas las clases *RacesFragment* y *RacesViewModel* mostradas en la figura 7.3.

El *layout* de esta interfaz está constituido principalmente por un *RecyclerView*, una *ProgressBar* con un *TextView* y los botones de los filtros. El *RecyclerView* es donde se muestran las carreras como una lista. En la figura 8.9 se puede ver el diseño del *layout*.

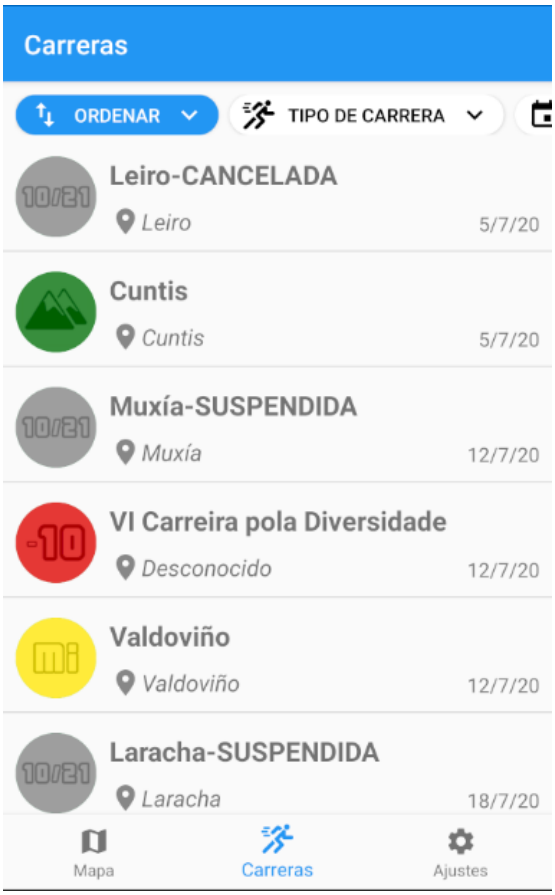


Figura 8.9: Lista de las carreras en la aplicación

En la parte de la vista se ha creado una clase llamada *RaceView* donde se guardan los

campos de las carreras con las representaciones utilizadas en varias partes de la vista. También se ha creado un enumerado llamado *RaceTypeEnum* donde están definidos los tipos de las carreras, y las imágenes que les corresponden. Estas entidades se pueden ver en el diagrama de clases de la figura 8.10.

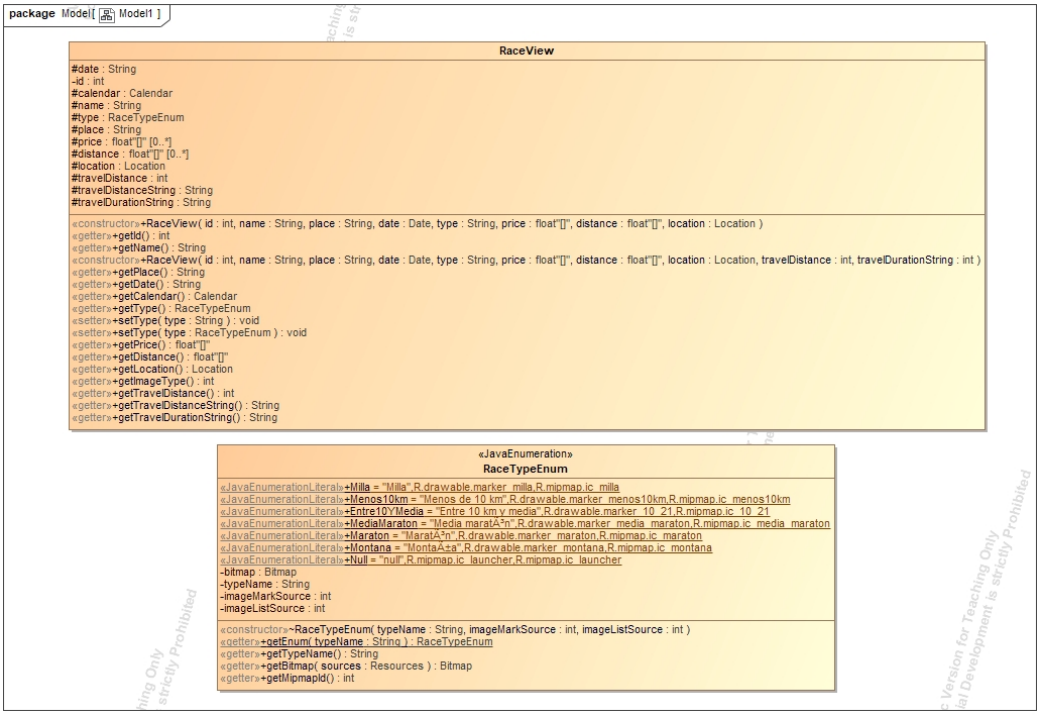


Figura 8.10: Diagrama de clases de *RaceView* y *RaceTypeEnum*

Al iniciar *RacesFragment* se configuran los diferentes elementos de la vista y se observan los *LiveData* que proporciona *RacesViewModel*. También se cargan las preferencias para saber si el usuario de la aplicación quiere utilizar la ubicación del dispositivo o una seleccionada por defecto. Si el usuario quiere utilizar su ubicación, se utiliza la clase estática *LocationManager* para poder obtenerla. Esta clase también se encarga de pedirle al usuario si desea activar la ubicación cuando está desactivada. En caso contrario, se guarda en las preferencias que el usuario no desea utilizar la posición de su dispositivo y se utiliza la ubicación guardada por defecto. El mismo comportamiento también se produce en la vista del mapa para poder mostrar al usuario en dicho mapa.

Una vez obtenida una ubicación, se manda a *RacesViewModel* para poder cargar los datos del modelo y obtener las distancias de la ubicación a las carreras. Así se pueden usar filtros que utilicen esos datos.

El componente más importante de esta vista es el *RecyclerView* donde se listan las carreras disponibles. Para poder listar las carreras en el *RecyclerView* fue necesario, en primer lugar,

crear un *layout* con los elementos que se van a representar para una carrera como *item* del *RecyclerView*. En este caso, el *layout* definido muestra la imagen asignada al tipo de carrera, el nombre de la carrera, el lugar y la fecha, como se puede apreciar en la figura 8.9. Las imágenes de los tipos de carrera se han elaborado de forma específica para esta aplicación.

Después de tener este *layout*, se define un *holder* llamado *RaceViewHolder* que extiende de *RecyclerView.ViewHolder*. *RaceViewHolder* contiene los elementos del *layout* definido para el *item* y se utiliza para crear un adaptador para el *RecyclerView*. Se ha creado un adaptador llamado *RaceAdapter* que extiende de *RecyclerView.Adapter<RaceViewHolder>*. En este adaptador se guarda la lista de carreras que se quiere representar. En él se implementan además unos métodos heredados para asignar la información de cada carrera a un *holder* que corresponde a un elemento de la lista. Aquí también se maneja la navegación a la vista en detalle de una carrera, mandando el *id* de la carrera seleccionada en las lista con el *NavController*.

Al arrancar la aplicación, en la vista de este *fragment*, solo se verá una *ProgressBar* y un *TextView* que indican que se están cargando los datos. Para actualizar el estado de la barra y el texto se observa un *LiveData* de un enumerado que indica el estado de la carga de datos. El estado inicial mantiene la barra como indeterminada, solo se mueve de un lado para el otro. Cuando se obtiene los datos del modelo, aún por procesar, el estado de la barra cambia a determinado y va aumentando según cambie otro *LiveData*, que contiene un entero con el número de procesados. En el momento en que ya se han cargado los datos la barra y el texto se ocultan y se muestra el *RecyclerView* y los filtros. Si se produce un error en la petición de los datos, el texto y la barra desaparecen y aparece una *SnackBar* con un mensaje de error y un botón para volver a realizar la petición.

La visualización de las carreras en el *RecyclerView* también depende de un *LiveData* de *RacesViewModel*. Este *LiveData* contiene una lista de *RaceView* y cambia solo cuando se aplican los filtros en *RacesViewModel*.

Cuando se cargan los datos con un *AsyncTask* en *RacesViewModel*, primero se llama al modelo para obtener la lista de *RaceModel* y llamar al método para calcular la distancia a la carreras. Después se crean en otra lista los *RaceView* equivalentes al los *RaceModel*. Esta lista se guarda en el *RacesViewModel* como la lista completa de datos y se aplican los filtros a una copia que es la guardada en el *LiveData*. El método para hacer el filtrado de las carreras que se visualizan se llama cada vez que un filtro cambia y se le aplican todos al clon de la lista completa.

8.3.4 Mapa con marcadores

En esta sección se comentará la implementación de la pantalla que muestra en un mapa las carreras. En esta implementación están implicadas las clases *MapsFragment* y *MapsViewModel* mostradas en la figura 7.3.

El *layout* de esta interfaz contiene una *ProgressBar*, un *RecyclerView*, un *FloatingActionButton*, un *FragmentContainerView* y los botones de filtrado. El elemento principal encargado de almacenar el mapa es el *FragmentContainerView*. En la figura 8.11 se puede ver como sería este *layout* con el mapa.

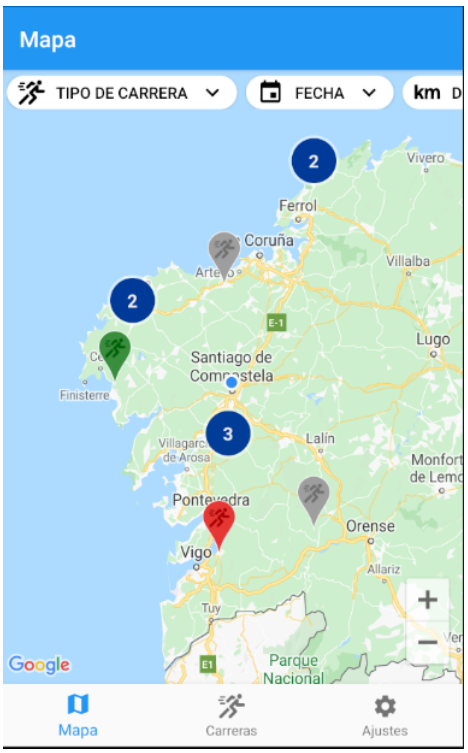


Figura 8.11: Mapa con los marcadores de las carreras

El inicio de *MapsFragment* es idéntico al de *RacesFragment*, en ambos se obtiene la localización para pasársela a la capa modelo para que cargue los datos. Se diferencian en que *MapsFragment* dispone de un mapa que hay que gestionar en el método *onMapReady*. Al igual que se ha comentado para *RacesFragment*, esta componente dispone de una *ProgressBar* para indicar que se están cargando los datos y de una *SnackBar* que permite reintentar la petición al modelo en caso de error.

Una vez comentadas las similitudes con lo explicado en la sección anterior, nos centraremos en explicar el funcionamiento del mapa.

Cuando el mapa está cargado se llama al método *onMapReady* en *MapsFragment*. Aquí se establece la configuración inicial del mapa, se activan los botones de zoom, se posiciona la cámara para que se muestre enfocando a Galicia, mientras no se ha obtenido la ubicación del usuario o la guardada por defecto. Si la localización del dispositivo está activada se puede habilitar para que muestre la ubicación en el mapa. Para imitar esta funcionalidad, cuando se utiliza una ubicación por defecto, se ha establecido un círculo azul como marcador en el

mapa.

Para mejorar la visualización de los marcadores en el mapa y no presenciar una aglomeración de ellos, se ha decidido utilizar la clase *ClusterManager*. Esta clase agrupa los marcadores que están muy juntos en *clusters*, separándolos o agrupándolos dependiendo del nivel de zoom de la cámara del mapa. Para utilizar esta clase se ha creado la clase *RaceClusterItem* que extiende a la clase *RaceView* y que implementa a las interfaces *ClusterItem* y *PointQuadTree.Item*. La primera interfaz es necesaria para poder añadir una instancia de *RaceClusterItem* a un *cluster*. En la figura 8.12 se puede ver el diagrama de clases de *RaceClusterItem*.

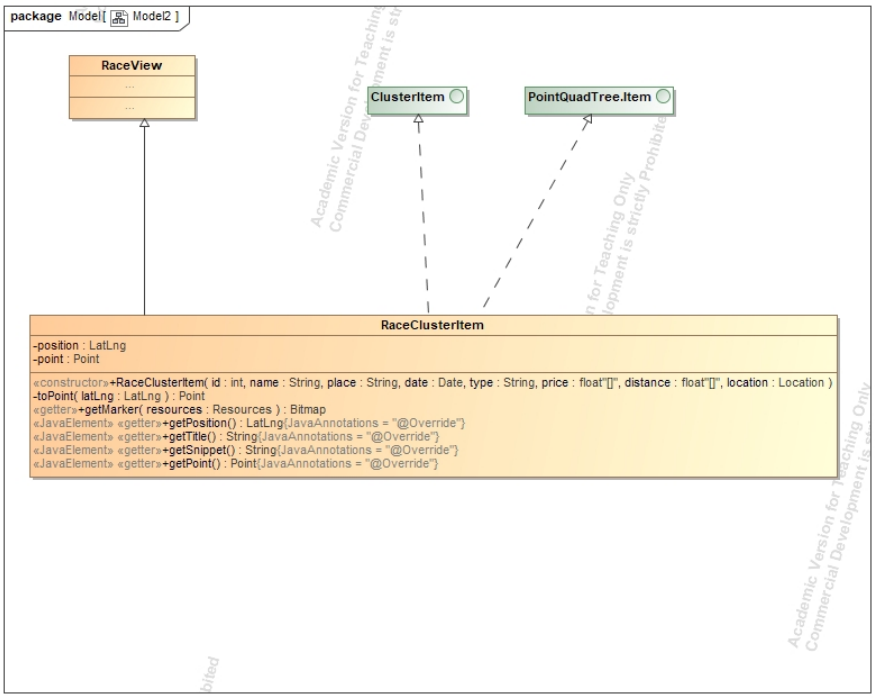


Figura 8.12: Diagrama de clase de *RaceClusterItem*

Además, para poder manejar los eventos de *click* sobre los marcadores, sobre las ventanas de los marcadores, etc., se tuvo que establecer explícitamente a *ClusterManager* como oyente de estos eventos en el mapa.

Al utilizar *ClusterManager* se pudo solucionar el problema presentado cuando hay más de una carrera en un mismo lugar. Utilizando marcadores solo se podía pinchar en el que quedaba por encima del resto, dejando inaccesibles los que quedaban por debajo. Para solucionar este problema, al seleccionar un *cluster* en el que todas las carreras tienen la misma localización, se despliega un *RecyclerView* con las carreras contenidas en el *cluster*, como se puede observar en la figura 8.13. Así es posible ver cuales son y acceder a la vista detallada al pinchar en una.

Además se han diferenciado este tipo de *cluster* del resto, cambiándoles el color a rojo con la ayuda de la clase *RaceClusterRenderer* que extiende de *DefaultClusterRenderer<RaceClusterItem>*.

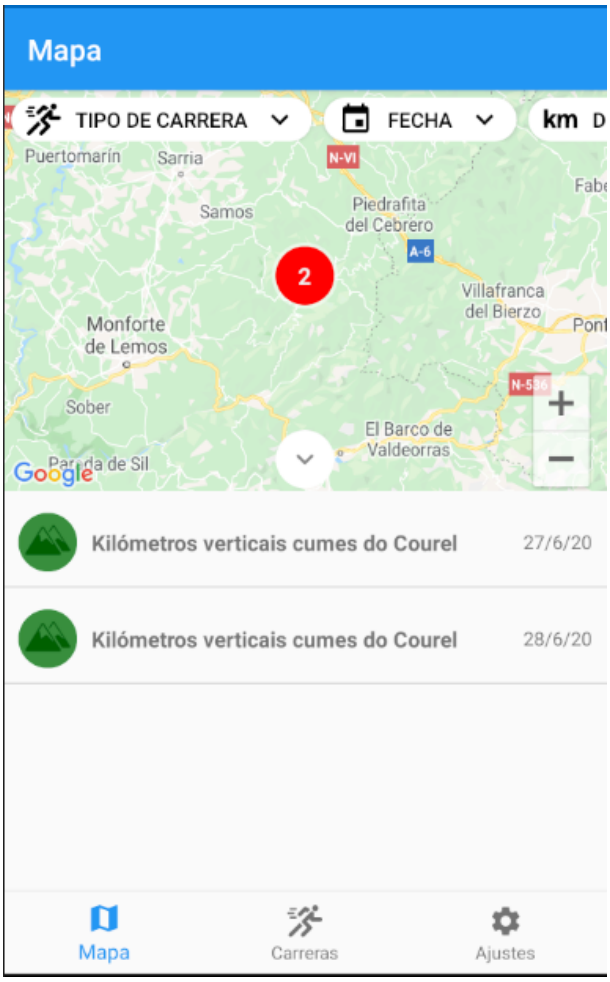


Figura 8.13: Lista de un *cluster* que tiene todos los elementos en la misma localizacion

Esta clase dispone de dos métodos: uno para personalizar los marcadores, que también se ha utilizado para crear unos marcadores originales para la aplicación, y otro para personalizar el icono de los *clusters*. En este método se comprueba si todas las carreras contenidas en el *cluster* que se va a pintar tienen la misma localización. De ser así se crea un icono en color rojo idéntico a los utilizados por defecto. Para conseguir obtener el mismo icono del resto de *cluster* con distinto color, ha sido necesario investigar en el código fuente para ver cómo se generaba el icono por defecto.

También se ha personalizado la ventana que sale al seleccionar un marcador para poder enseñar en ella el nombre de la carrera, el lugar y la fecha. Para este fin se ha utilizado la clase *RaceInfoViewAdapter* que implementa *GoogleMap.InfoWindowAdapter*. En esta clase se carga el *layout* personalizado y se rellena con los datos que se quieren mostrar. Además, el *click* en esa ventana lleva a la vista detallada de la carrera seleccionada.

Como funcionalidad extra se ha conseguido que al seleccionar un *cluster* normal se haga la cantidad de zoom exacta para que se separe en sus correspondientes marcadores. Para ello ha sido necesario que *RaceClusterItem* implementara *PointQuadTree.Item* y así poder utilizarlo en un *PointQuadTree*. Para hacer este cálculo se utiliza el zoom actual de la cámara del mapa y un *PointQuadTree* con los *RaceClusterItem* del *cluster* seleccionado. Esto se realiza en un método cuando se pulsa un *cluster*, por simplicidad no se contará el funcionamiento del método.

8.3.5 Vista detallada

En esta sección se comentará la implementación de la pantalla que muestra la información detallada de una carrera. En esta implementación están implicadas las clases *VisualizerFragment* y *VisualizerViewModel* mostradas en la figura 7.3.

El *layout* de esta interfaz esta constituido principalmente por dos partes, la información básica de la carrera y la información del viaje. En la información básica se muestra el nombre, la imagen del tipo de carrera, el lugar, la hora, la distancia y el coste de inscripción. En cambio, la información del viaje muestra la distancia hasta el lugar donde se celebra, el tiempo de viaje, los costes de desplazamiento a la carrera y el precio del combustible seleccionado. En la figura 8.14, se puede ver el aspecto de este *layout*.

Para llegar a esta pantalla se ha tenido que seleccionar una carrera en la lista de carreras o en el mapa. Al seleccionar la carrera se tiene que pasar el identificador mediante el *NavController* para poder cargar los datos correspondientes a esa carrera.

En el inicio de *VisualizerFragment* se recibe el identificador de la carrera que hay que cargar y se obtienen, a partir de las preferencias del usuario, el identificador del combustible y el consumo del vehículo. Estos tres datos se le pasan al *VisualizerViewModel* para que cargue la carrera del modelo, el precio del combustible y calcule los costes de desplazamiento.

Inicialmente se muestra una *ProgressBar* mientras se cargan los datos del modelo. En el

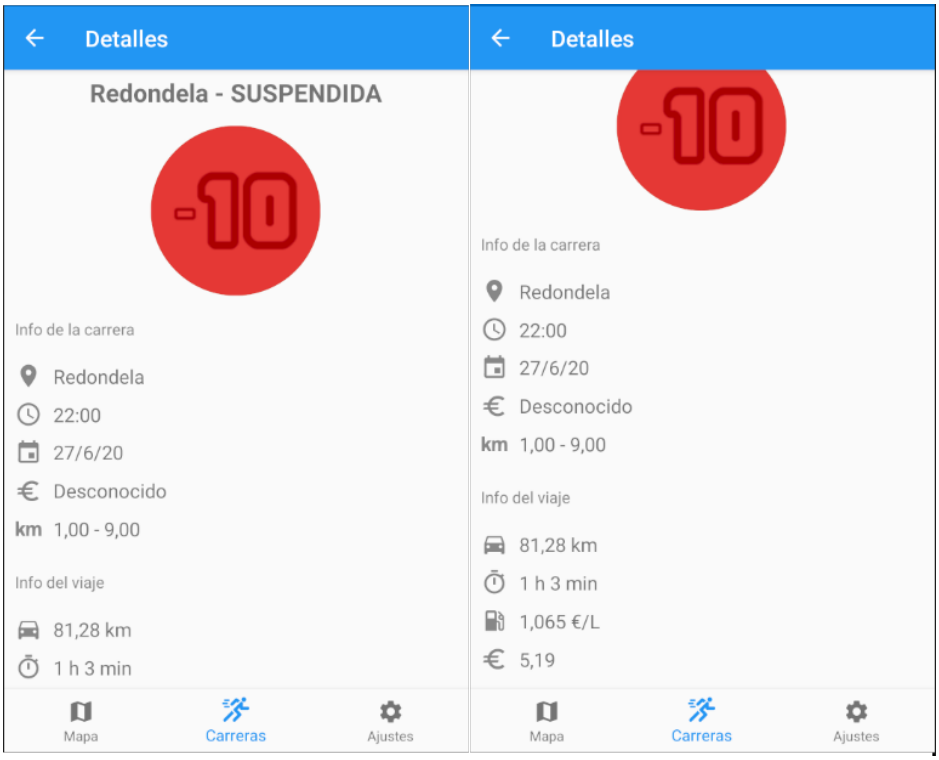


Figura 8.14: Vista detallada de las carreras en la aplicación

VisualizerViewModel hay un *LiveData* que contiene un *boolean*. Si el *boolean* se pone a *false* quiere decir que ha fallado la carga de los datos del modelo. En este caso, se oculta la *ProgressBar* y se muestra una *SnackBar* con un mensaje de error y un botón de reintentar. En cambio, si todo funciona correctamente, se oculta la *ProgressBar* y se cargan los datos de la carrera con los métodos del *VisualizerViewModel*. Estos métodos devuelven los datos formateados para ponerlos en la interfaz de usuario y también realizan los cálculos de los costes de desplazamiento.

Si el lugar de la carrera cargado es nulo, se oculta la parte de la vista con la información del viaje, ya que no se dispone de ella.

La petición de los datos desde el *VisualizerViewModel* al modelo se realiza con un *AyncTask*, donde se llama al método que devuelve el precio del combustible y al que devuelve el *RaceModel* asociado a un identificador.

8.3.6 Ajustes

En esta sección se comentará la implementación de la pantalla de ajustes de la aplicación. En esta implementación están implicadas las clases *SettingsFragment* y *SettingsViewModel* mostradas en la figura 7.3.

A diferencia de los demás *fragments*, el diseño de los ajustes no se realiza en un *layout* si no que se realiza en un archivo llamado *preferences.xml*. La estructura de este archivo se define con componentes *Preference* y sus correspondientes subtipos.

Las preferencias que se pueden establecer en los ajustes son:

- **Idioma:** se permite escoger entre inglés, gallego, castellano y el del sistema.
- **Localización:** se deja escoger si utilizar la localización del dispositivo o escoger una para usar por defecto.
- **Filtros:** se permite escoger para cada filtro un modo para que ya se muestren las carreras filtradas al abrir la aplicación.
- **Tipo de combustible:** deja escoger entre gasolina 95 y gasóleo A.
- **Consumo del vehículo:** permite escoger el consumo del vehículo en *L/100km*, en un rango de uno a cincuenta.

La figura 8.15 se puede ver la pantalla de ajustes con las preferencias mencionadas anteriormente.

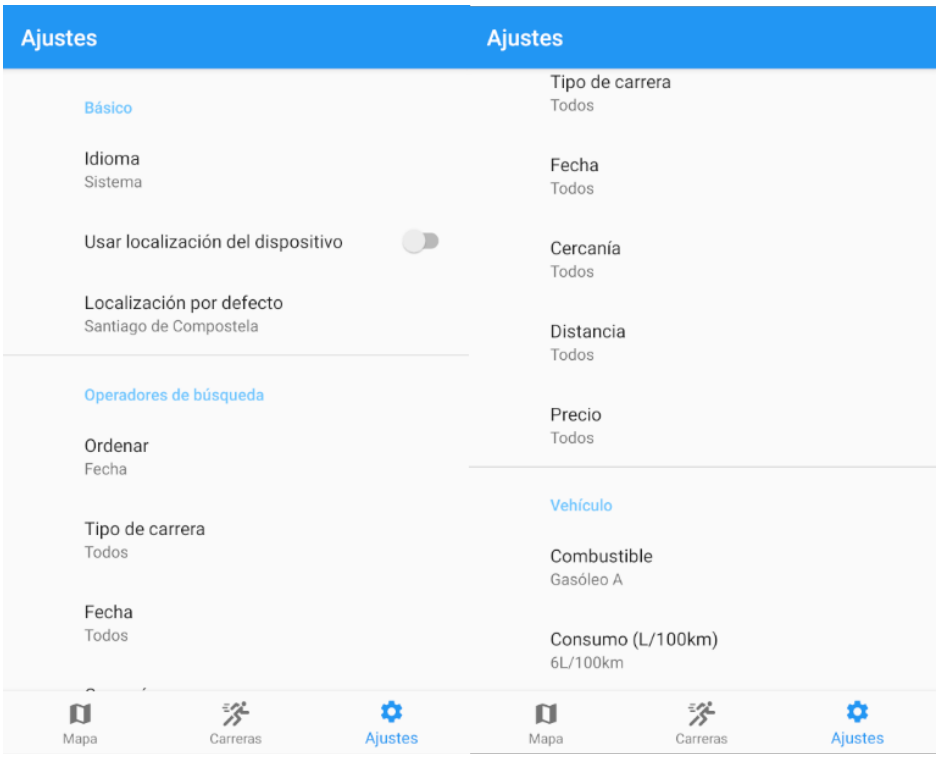


Figura 8.15: Vista de ajustes en la aplicación

En el método *onCreatePreferences* de *SettingsFragment* se configuran las acciones que se deben tomar cuando se modifican las selecciones de las preferencias, además de establecerse el *Summary* para saber la opción escogida en cada una.

Los cambios en la preferencias que afectan a los filtros se procesan en el *SettingsViewModel*, donde se establece el modo que ha seleccionado el usuario para cada instancia del filtro. Esto implica que en los lugares donde se utiliza el filtro quedaría ya activado con esa selección.

El cambio de idioma también se gestiona en el *SettingsViewModel*, utilizando la clase estática *LocaleManager*. Esta clase cambia la configuración del contexto, utilizado por la aplicación, para el idioma. Para que los cambios surtan efecto, es necesario recrear la actividad para que se generen de nuevo los textos. Se debe llamar a esta clase con el método *attachBaseContext* del *MainActivity* para proporcionar a la actividad principal el nuevo contexto con el idioma establecido. Esta llamada se realiza cuando se recrea la actividad pero, también, cuando se inicia normalmente la aplicación.

Al modificar la selección de una preferencia, esta se guarda automáticamente en las *SharedPreference*. En cambio, esto no ocurre para las preferencias declaradas como *Preference*. Este tipo de preferencia es la más básica, se utiliza para representar preferencias que no se contemplan como un tipo estándar. En este caso se han declarado como *Preference* el consumo, para poder mostrar un diálogo con un *NumberPicker* en el que escoger el valor, y la ubicación por defecto, que al pulsarla te lleva a otro *fragment* donde se puede escoger la ubicación en un mapa. Los cambios de valor de estas preferencias deben ser guardados manualmente en las *SharedPreference*, y además su valor por defecto debe ser establecido en el *MainActivity* la primera vez que se instala la aplicación para que este valor no sea nulo.

Cuando un usuario cambia el valor de su localización por defecto, se hace una petición al modelo para obtener el nombre del lugar de la nueva ubicación y poder mostrarlo. Esta llamada se realiza desde un *AsyncTask* en el *SettingsViewModel*.

8.3.7 Filtros

En esta sección se comentará la implementación del diseño expuesto en el capítulo de diseño (sección 7.3.3) para el caso concreto de las operaciones de filtrado de las carreras. Para facilitar la explicación de la implementación, se hará mención a las clases y métodos mostrados en el diagrama de clases de la figura 7.5.

En la implementación del diseño de los filtros, se han desarrollado las siguiente implementaciones de *SearchOperator*:

- ***Order.*** Esta clase sirve para ordenar una lista de *RaceView*. Permite ordenar por fecha, cercanía, menor distancia, mayor distancia, más barata y más cara. Se utiliza solo en la pantalla de la lista de carreras.

- ***FilterRaceType.*** Esta clase extrae de una lista de *RaceView* las carreras que sean tipo de carrera seleccionado.

- ***FilterDate.*** Esta clase extrae de una lista de *RaceView* las carreras cuya fecha esté dentro de un rango de fechas determinado.

- ***FilterCloseness.*** Esta clase extrae de una lista de *RaceView* las carreras que pertenezcan al rango de cercanía seleccionado, es decir, que están a una distancia de la localización del usuario que encaje en el rango seleccionado. Se utiliza solo en la pantalla de la lista de carreras.

- ***FilterPrice.*** Esta clase extrae de una lista de *RaceView* las carreras cuyo rango de coste de inscripción esté, total o parcialmente, incluido en el rango seleccionado.

- ***FilterDistance.*** Esta clase extrae de una lista de *RaceView* las carreras cuyo rango de distancia esté, total o parcialmente, incluido en el rango seleccionado. En este caso, la distancia hace referencia a los kilómetros de los que consta la carrera.

Además de este diseño, se ha creado una clase estática llamada *ButtonsSearchOperator* donde se configuran los botones de los filtros y su comportamiento para no tener que repetir código. Esta clase es utilizada en la pantalla de la lista de carreras y en la del mapa. Para que el mismo código funcionara en las dos pantallas se ha usado un patrón *Strategy* con sus modelos de vista y se han utilizado los mismos identificadores para los botones en ambos *layouts*.

En *ButtonsSearchOperator* se configuran además las acciones de los botones y los cambios de aspecto cuando se activan los filtros. Una pulsación larga en un botón establece el modo por defecto, mientras que una pulsación normal muestra una lista con las opciones para los diferentes filtros.

Para mostrar las opciones de los filtros se utiliza la clase *BottomDialogFragment*. Esta clase despliega un diálogo expansible en la parte baja de la pantalla y, en él, se muestran las opciones de cada filtro listadas con un *RecyclerView*. El *holder* del *RecyclerView* es el encargado de establecer el modo seleccionado por el usuario en la instancia del filtro. En la figura 8.16, se puede ver como sería el despliegue de las opciones para un determinado filtro.

8.4 Pruebas

En esta sección se exponen brevemente las distintas pruebas que se le han realizado para validar el correcto funcionamiento del sistema desarrollado.

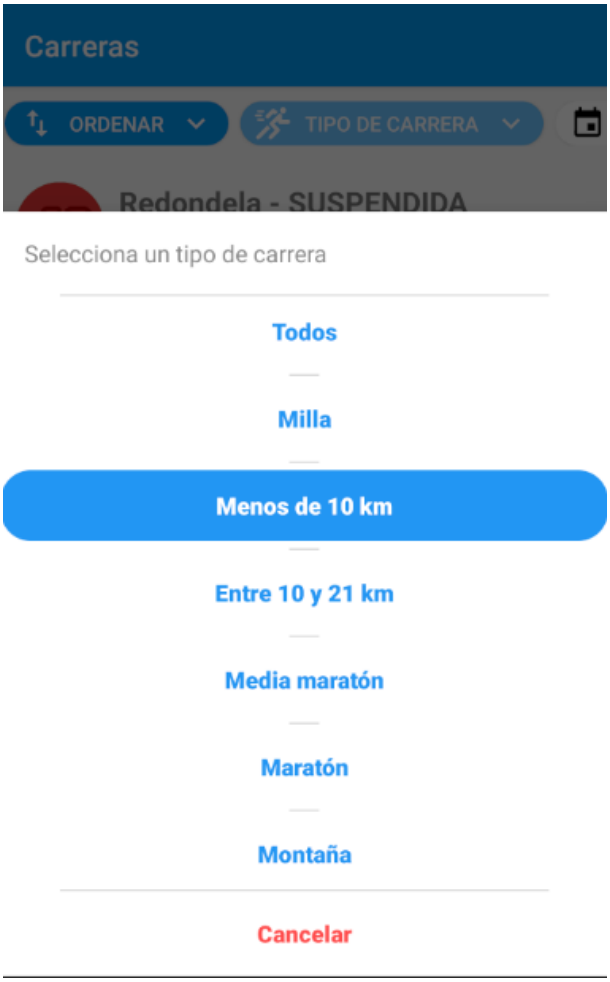


Figura 8.16: Vista de la lista de opciones que presenta el filtro *FilterRaceType*

8.4.1 Pruebas funcionales

Estas pruebas están dirigidas a comprobar el correcto funcionamiento de todas las funcionalidades que se han desarrollado en el proyecto. Para el desarrollo de estas pruebas se ha utilizado:

- **Un teléfono Android:** se ha utilizado para probar las funcionalidades de la aplicación. De esta forma, se ha comprobado que las distintas funcionalidades implementadas en la aplicación funcionen correctamente en un entorno real.
- **Un navegador *web*:** utilizado para probar funcionalidades del servidor y comprobar manualmente si el *crawler* extrae todos los datos disponibles.
- **Una herramienta para hacer peticiones HTTP:** se ha utilizado para comprobar el correcto funcionamiento de la API REST del servidor.

8.4.2 Pruebas de aceptación

Estas pruebas se realizan al final del desarrollo de un sistema, y sirven para comprobar si la aplicación resultante cumple con los requisitos establecidos.

Estas pruebas normalmente se realizan por el cliente, pero en este proyecto se realizaron por los directores del proyecto y el autor. Es importante destacar que, además, se le ha mostrado la versión final de la aplicación a varias personas muy aficionadas a correr y a acudir a carreras populares con el objetivo de verificar su utilidad y adecuación. El grado de satisfacción ha sido, en general, muy alto.

Capítulo 9

Conclusiones

Después de finalizar todo el desarrollo del proyecto, se ha podido analizar todo el trabajo realizado y extraer unas serie de conclusiones. En este capítulo se comentarán dichas conclusiones y el trabajo futuro.

9.1 Conclusiones

Una vez finalizado el proyecto se han podido extraer las siguiente conclusiones:

- El producto final satisface todas las funcionalidades inicialmente propuestas. Incluso se han añadido ciertos detalles para mejorar la experiencia del usuario.
- Se ha podido observar que existe una demanda real por la centralización de los datos, ya que muchos deportista se quejan de tener que buscar la información de las carreras en diversos páginas *web*.
- La aplicación móvil dispone de una interfaz muy agradable e intuitiva para los usuarios. Durante el desarrollo se ha podido comprobar dejándosela utilizar a varios aficionados.
- La extracción de los datos es dependiente de cambios en el diseño de los recursos *web*.

Por otro lado, el desarrollo de este proyecto ha permitido al autor profundizar en algunas tecnologías vistas a lo largo del grado pero, sobre todo, aprender y conocer los entresijos de un gran abanico de herramientas y *frameworks* necesarios para el desarrollo de una aplicación tan completa y con diferentes componentes como la que se ha presentado en esta memoria. Además, este proyecto le ha permitido enfrentarse a la realidad de desarrollar un producto *sotware* funcional desde el principio, siendo necesario establecer una metodología y una planificación adecuadas, así como realizar todo el análisis y diseño previos al desarrollo.

9.2 Trabajo futuro

Aunque se haya terminado el desarrollo del proyecto, la demanda que existe de este tipo de aplicaciones invita a seguir trabajando en él en los siguientes aspectos:

- **Publicar la aplicación.** Se podría adquirir un dominio para el servidor y publicar la aplicación el la *Play Store* de Android.
- **Expandir el área demográfica.** Buscar fuentes de información que ofrezcan datos sobre las carreras realizadas a nivel nacional y conseguir así el interés de más usuarios.
- **Ofrecer más datos de interés.** Se podría buscar la manera de calcular los costes de los peajes, obtener la hora de entrega de los dorsales, etc.
- **Utilizar API libres.** Para reducir los costes que supondría mantener la aplicación publicada, se podría buscar otras API de uso totalmente gratis en sustitución a las de Google.
- **Crear una interfaz *web*.** Aprovechando los datos que se han conseguido y el uso de Django como servidor, se podría crear una interfaz *web* que ofrezca funcionalidades similares a las de la aplicación móvil.

Apéndices

Lista de acrónimos

API *Application Programming Interface.*

HTML *HyperText Markup Language.*

SQL *Structured Query Language.*

REST *Representational State Transfer.*

MVVM *Model View Viewmodel.*

PDF *Portable Document Format.*

MTV *Model Template View.*

MVC *Model View Controler.*

IDE *Integrated Development Environment.*

JSON *JavaScript Object Notation.*

SDK *Software Development Kit.*

BSON *Binary JSON.*

RAM *Random Access Memory*

XML *eXtensible Markup Language*

URL *Uniform Resource Locator*

HTTP *Hypertext Transfer Protocol*

Glosario

API key Es un identificador único que se utiliza para autenticar las solicitudes asociadas a un proyecto registrado en la plataforma Google Clound, para fines de uso y facturación. 15

COVID-19 Es la enfermedad infecciosa causada por el coronavirus que se ha descubierto en diciembre de 2019. 10, 27

USD Es el código ISO 4217 para el dólar estadounidense. 32

Bibliografía

[1] Proyectosagiles, "Desarrollo iterativo e incremental," accedido por última vez el 16/06/2020. [En línea]. Disponible en: https://proyectosagiles.org/desarrollo-iterativo-incremental/

[2] I. Espindola, "Que es el patrón mtv (model template view)," accedido por última vez el 16/06/2020. [En línea]. Disponible en: http://arquitecturavirtual.net/mtv-django.html

[3] "Guía de arquitectura de apps," accedido por última vez el 16/06/2020. [En línea]. Disponible en: http://arquitecturavirtual.net/mtv-django.html

[4] "Página oficial de la app GaliciaCorre," accedido por última vez el 06/05/2020. [En línea]. Disponible en: https://galiciacorre.es

[5] "Página oficial de la app HoyQuieroCorrer," accedido por última vez el 06/05/2020. [En línea]. Disponible en: https://www.hoyquierocorrer.com/

[6] "Página oficial de la app Carreras populares," accedido por última vez el 06/05/2020. [En línea]. Disponible en: https://ramsoft.info/ramrun/

[7] "Dirección web del foro Correr en Galicia," accedido por última vez el 06/05/2020. [En línea]. Disponible en: https://www.correrengalicia.org

[8] "Página oficial de la federación gallega de atletismo," accedido por última vez el 06/05/2020. [En línea]. Disponible en: https://www.carreirasgalegas.com

[9] "Página oficial de Python," accedido por última vez el 06/05/2020. [En línea]. Disponible en: https://www.python.org/

[10] "Página oficial de Java," accedido por última vez el 06/05/2020. [En línea]. Disponible en: https://www.java.com

[11] "Página oficial del framework Scrapy," accedido por última vez el 06/05/2020. [En línea]. Disponible en: https://scrapy.org/

[12] "Página oficial del software de scraping Octoparse," accedido por última vez el 06/05/2020. [En línea]. Disponible en: https://www.octoparse.es/

[13] "Página oficial de la librería Beautiful Soup," accedido por última vez el 06/05/2020. [En línea]. Disponible en: https://www.crummy.com/software/BeautifulSoup/

[14] "Página oficial del framework Django," accedido por última vez el 06/05/2020. [En línea]. Disponible en: https://www.djangoproject.com/

[15] "Repositorio oficial de Djongo en GitHub," accedido por última vez el 06/05/2020. [En línea]. Disponible en: https://github.com/nesdis/djongo

[16] "Página oficial del framework Django REST," accedido por última vez el 06/05/2020. [En línea]. Disponible en: https://www.django-rest-framework.org/

[17] "Página oficial de Apache HTTP Server," accedido por última vez el 06/05/2020. [En línea]. Disponible en: https://httpd.apache.org/

[18] "Página oficial del framework Ruby on Rails," accedido por última vez el 06/05/2020. [En línea]. Disponible en: https://rubyonrails.org/

[19] "Página oficial de framework Laravel," accedido por última vez el 06/05/2020. [En línea]. Disponible en: https://laravel.com/

[20] "Repositorio oficial de Gson en GitHub," accedido por última vez el 06/05/2020. [En línea]. Disponible en: https://github.com/google/gson

[21] "Página oficial del editor Visual Studio Code," accedido por última vez el 06/05/2020. [En línea]. Disponible en: https://code.visualstudio.com/

[22] "Página oficial del IDE Android Studio," accedido por última vez el 06/05/2020. [En línea]. Disponible en: https://developer.android.com/studio

[23] "Página oficial de Git," accedido por última vez el 06/05/2020. [En línea]. Disponible en: https://git-scm.com/

[24] "Página oficial de la base de datos MongoDB," accedido por última vez el 06/05/2020. [En línea]. Disponible en: https://www.mongodb.com/

[25] Z. G. GU Yunhua, SHEN Shu, "Application of nosql database in web crawling," *International Journal of Digital Content Technology and its Applications*, vol. 5,

no. 6, 2011. [En línea]. Disponible en: https://pdfs.semanticscholar.org/ca23/da6ac61577c69c0cb1bcc90f0988685ebdd1.pdf

[26] "Página oficial de Google Maps Platform," accedido por última vez el 06/05/2020. [En línea]. Disponible en: https://cloud.google.com/maps-platform

[27] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, Universidad de California en Irvine, 2000. [En línea]. Disponible en: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf

[28] Tutorialspoint, "Sdlc - iterative incremental model," accedido por última vez el 16/06/2020. [En línea]. Disponible en: https://www.tutorialspoint.com/adaptive_software_development/sdlc_iterative_incremental_model.htm

[29] V. Consultores, "Guia salarial sector TI Galicia 2015-2016," accedido por última vez el 16/06/2020. [En línea]. Disponible en: https://vdocuments.site/guia-salarial-sector-ti-galicia-2015-2016.html

[30] G. C. Platform, "Precios de maps, routes y places," accedido por última vez el 16/06/2020. [En línea]. Disponible en: https://cloud.google.com/maps-platform/pricing/sheet?hl=es