TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

UNIVERSIDAD DE ALMERIA

ESCUELA SUPERIOR DE INGENIERÍA

Aplicación multiplataforma de viajes en Rijk Zwaan con Xamarin Forms y ASP.NET

Curso 2019/2020

Alumno/a:

Óscar López Montero

Director/es:

Yolanda Granados Carricondo

José Joaquín Cañadas Martínez

Índice

Índice de figuras

Resumen

Es innegable que el uso de tecnologías móviles se encuentra en pleno auge y que hoy día, estos dispositivos forman parte de la vida cotidiana de cada uno. Por ello, y debido a la variedad de sistemas operativos que existen en el mercado, el desarrollo de aplicaciones móviles multiplataforma se ha convertido en una de las tareas más importantes de la actualidad.

En Rijk Zwaan, las aplicaciones móviles son una manera importante de introducir información de manera sencilla y rápida, sin que el trabajador tenga que perder mucho tiempo en rellenar formularios a ordenador, y simplificando las labores del mismo.

En este caso, se ha propuesto gestionar de una manera más sencilla los viajes entre fincas de Rijk Zwaan Ibérica y el envío de material entre las mismas, de manera que se haga un uso común de los vehículos destinados a dicho uso, y suponiendo un menor gasto en servicios de transporte de mercancías.

Abstract

It’s undeniable that mobile technologies are currently booming and that those devices have become really important in our daily lives.
For that reason and due to the variety of operative systems in the market, cross-platform mobile development has become one of the most demanded jobs nowadays.

Mobile applications have a really important role at Rijk Zwaan since they are an easy and fast way to input information into their storage without losing time filling in big forms.

In this project, we want to manage trips and material shipping in between Rijk Zwaan Ibérica locations to provide a common usage of the vehicles, making the company to save money in delivery services.

1. Introducción

Hoy día, los dispositivos móviles se han convertido en una parte esencial de nuestra vida cotidiana con el fin de simplificar o facilitar nuestras tareas, es por ello que, la inclusión de los mismos tanto en el trabajo como en nuestro tiempo de ocio resulta en una importante reducción de la carga de trabajo que tenemos que manejar, pudiendo dedicar nuestro tiempo a tareas más productivas.

Según el estudio de Frost & Sullivan realizado por Samsung, los smartphones incrementan la productividad de los trabajadores en un 34%, ahorrando una media de 58 minutos de trabajo por empleado [1].

Este ahorro del trabajo ha motivado un incremento en el desarrollo de aplicaciones móviles, y la variedad de sistemas operativos que existen (iOS, Android, Windows Phone) exigen a las mismas su plena funcionalidad en el máximo número de dispositivos posible. Es por ello que hemos decidido realizar una aplicación móvil multiplataforma con interfaz web, con el fin de que la mayoría de los usuarios puedan usarla teniendo el menor número de inconvenientes posible.

1.1. Motivación

En Rijk Zwaan, tanto el transporte de materiales como los viajes entre fincas se han convertido en un elemento imprescindible del trabajo, llegando a suponer un gasto importante en servicios de transporte de mercancías y una escasez de vehículos destinados a la movilización de personal. Por ello, se ha decidido implantar una aplicación para organizar los viajes entre fincas y transportar materiales al mismo tiempo, aprovechando la totalidad del vehículo que se va a usar.

La necesidad de la aplicación surge principalmente del laboratorio de biología de la finca El Mamí de Rijk Zwaan debido a que se hacen una media de 5 envíos semanales, que además de ser costosos, se encuentran bastante limitados por el día y hora de llegada del material, suponiendo una pérdida importante de horas de trabajo.

El ahorro económico que puede suponer nuestra aplicación viene dado por la sumatoria del coste de una empresa de transporte de mercancías y el coste de combustible los vehículos usados individualmente por los trabajadores, además de aumentar la disponibilidad de vehículos de finca destinados a estos fines.

Respecto a la motivación personal por realizar este proyecto, siempre me ha resultado de especial interés el desarrollo de aplicaciones móviles, sobre todo de la interfaz o *front-end* de las mismas en Android. El no haber trabajado nunca con Android y Xamarin me ha

supuesto un pequeño reto, pero los conocimientos de programación orientada a objetos y C# que he adquirido en el grado me han ayudado bastante a aventurarme en este proyecto dado que Xamarin nos permite el desarrollo de aplicaciones multiplataforma con Visual Studio [2].

Por otro lado, me satisface el hecho de ayudar a los trabajadores a administrar, compartir y ahorrar en sus viajes facilitando su trabajo ya que puedo hacer que el trabajo de los demás sea más ameno y disfrutable, por ello, quiero que mis habilidades para desarrollar aplicaciones e interfaces crezcan hasta el punto de hacer que los usuarios disfruten usando mis aplicaciones haciendo uso de animaciones y transiciones lo más vistosas posibles.

Este proyecto supone una gran oportunidad para poner a prueba lo aprendido de C#, y a su vez aprender a desarrollar aplicaciones multiplataforma para dispositivos móviles y mejorar con ello mis habilidades en este ámbito.

1.2. Objetivos

El objetivo principal de nuestro Trabajo de Fin de Grado consiste en diseñar y desarrollar una aplicación multiplataforma en la que los usuarios puedan registrar los viajes que desean realizar entre las sedes de Rijk Zwaan Ibérica de manera sencilla y rápida, así como una parte web la cual, además de contener la funcionalidad de la aplicación móvil, permita la gestión a varios niveles de la aplicación por parte de usuarios con mayores privilegios. Para ello, se ha decidido implantar una estructura cliente-servidor con Xamarin Forms para la aplicación móvil y una página web y API desarrollada con ASP.NET Core para los trabajadores que acceden desde su ordenador.

La estructura de nuestro proyecto a desarrollar será la siguiente:

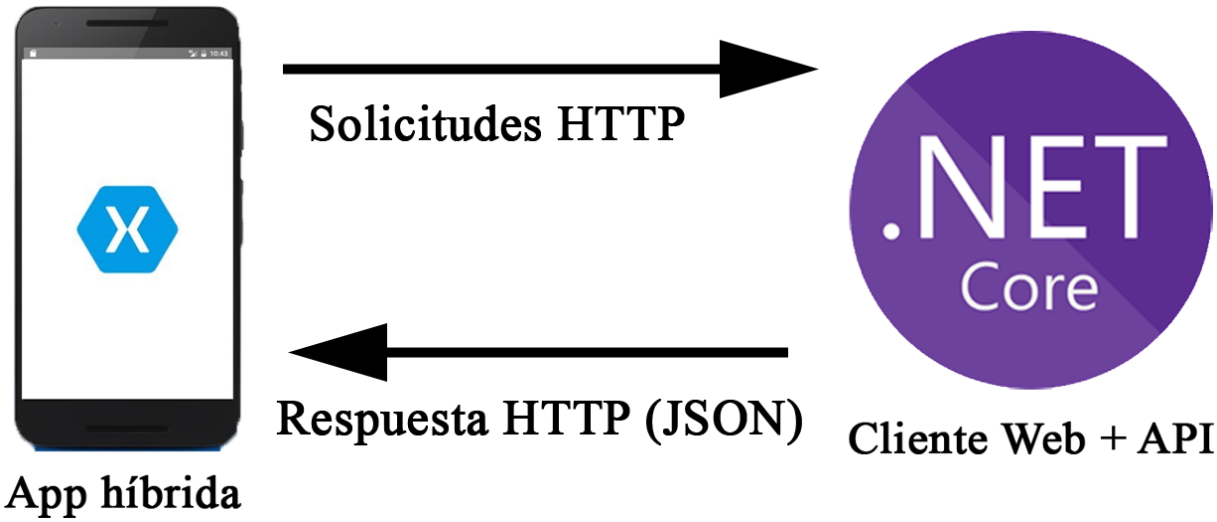

Figura 1. Estructura de aplicación de viajes

En la aplicación, de manera general, se debe permitir:

- Registro de viajes: El usuario debe informar del lugar al que desea viajar y del día de forma obligatoria, y de forma opcional, puede optar a informar del número de plazas, hora, descripción del viaje, …
- Consulta de viajes: El usuario debe poder consultar todos los viajes disponibles de los compañeros.
- Notificaciones: El usuario debe poder registrar ubicaciones de interés, y recibir una notificación a su correo electrónico en el caso de que se registre algún viaje entre alguna de estas ubicaciones.

Todas las funcionalidades de la aplicación serán explicadas más a fondo en forma de historias de usuario.

Como objetivo final de la aplicación, se espera conseguir, además del ahorro mencionado anteriormente, una gestión más eficiente de los viajes entre los departamentos de Rijk Zwaan Ibérica, así como una mayor independencia y eficacia a la hora de enviar material.

1.3. Planificación temporal

La planificación inicial que seguir durante el desarrollo del proyecto es la indicada en el cronograma:

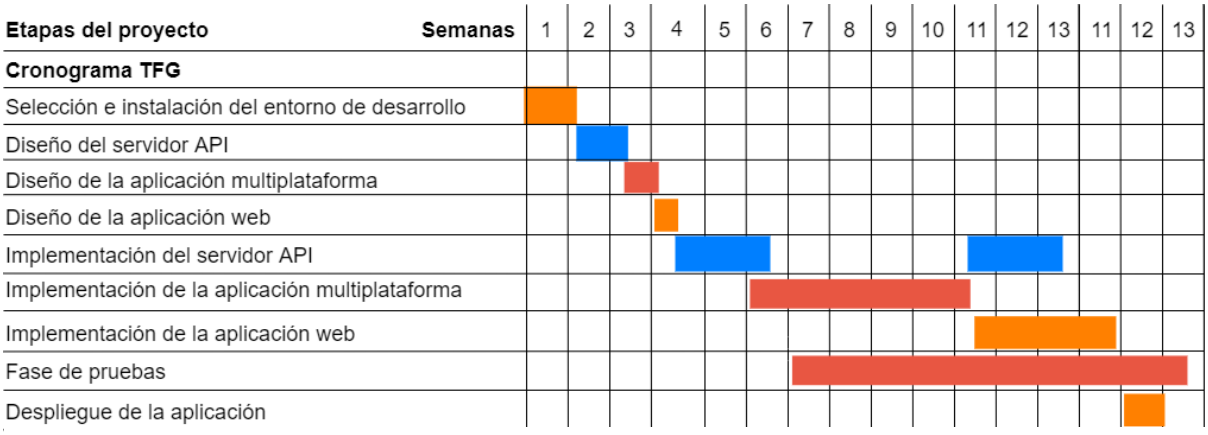

Figura 2. Planificación temporal

1- Selección e instalación del entorno de desarrollo: Selección, instalación y configuración del IDE. Duración estimada: (~10h)
2- Diseño del servidor API: Diseño del servidor API en un entorno de pruebas, así como nuestra base de datos, con el fin de proporcionar soporte API REST a nuestra aplicación. Duración estimada: (~5h)
3- Diseño de la aplicación multiplataforma: Diseño de la aplicación partiendo de diagramas de casos de uso, y prototipados de la interfaz. Duración estimada: (~20h)

4- Diseño de la aplicación web: Diseño de la aplicación partiendo de un diagrama de casos de uso y de la estructura de la API. Duración estimada: (~10h)
5- Implementación del servidor API: Implementación del servidor API en un entorno de pruebas, así como los métodos y *requests* con los que se trabajará posteriormente tanto en web como en la aplicación híbrida.
6- Implementación de la aplicación multiplataforma: Codificación de la aplicación multiplataforma usando Xamarin Forms. Esta parte incluye:
    a. - Codificación común usando Forms.
    b. - Codificación Android.
    c. - Codificación iOS.

Duración estimada: (~70h)

7- Implementación de la aplicación Web: Codificación de la aplicación web usando ASP.NET y *back-end*. Duración estimada: (~70h)
8- Fase de pruebas: Testeo del funcionamiento de la aplicación llevándola al mayor número de casos posibles. Duración estimada: (~40h)
9- Despliegue de la aplicación: Despliegue de la aplicación web en Plesk para el uso de los empleados. Duración estimada: (~10h).
10- Elaboración de la memoria TFG.

Esta planificación inicial se ha usado como estructura para comenzar el proyecto, aunque decidimos optar por seguir una metodología de desarrollo SCRUM, ya que de esta manera el cliente puede ir aportando *feedback* y nuevas funcionalidades a nuestra aplicación mediante reuniones con el equipo de desarrollo.

En este caso, Yolanda Granados hará el rol de *Product Owner* y nosotros desempeñaremos el rol de *Scrum Master* [7].

También cabe destacar, que el desarrollo de este proyecto ha sido interrumpido durante varios meses debido a la crisis del Covid19, llevando a tener varios meses de inactividad, y necesitando de varias reuniones adicionales para retomar el desarrollo del mismo.

Al seguir esta metodología de desarrollo, hemos tenido que dividir el proyecto en 5 diferentes *sprints*:

1- Toma de datos y diseño de la aplicación (13 Feb – 16 Feb):

   En la primera reunión con nuestro cliente, realizamos una toma de los requisitos que tendrá que cumplir nuestra aplicación, tras esta, realizamos el diseño de nuestra base de datos, así como de nuestra aplicación híbrida y web.

2- Implementación de la aplicación multiplataforma (16 Feb – 13 Mar):

En esta fase se ha realizado la implementación de la aplicación multiplataforma, así como de los requisitos posteriores que al cliente le han surgido tras ver los prototipados de cada reunión. Esta fase fue interrumpida por la crisis del Covid19, y, al no requerir más reuniones con el cliente, se procedió a completar después de la implementación web.

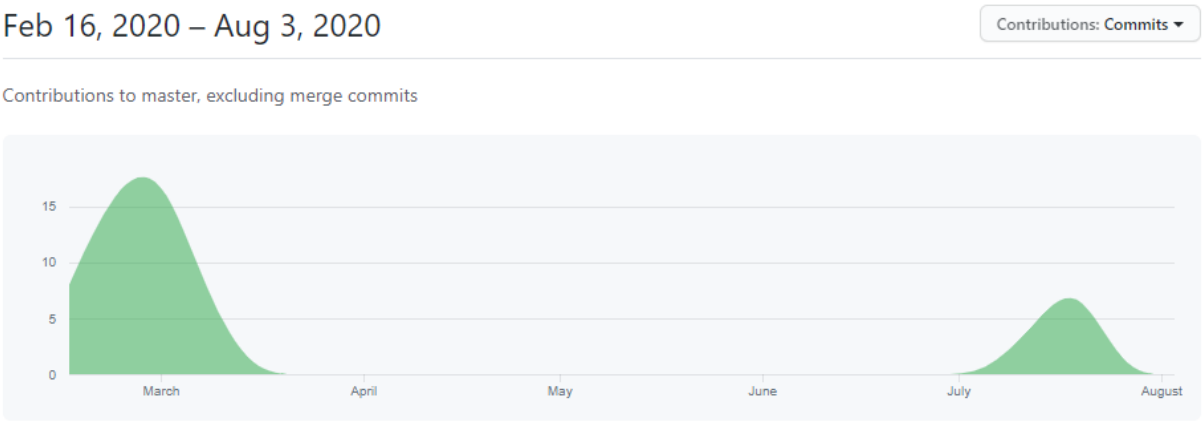

Figura 3. Commits en GitHub de la aplicación

3- Implementación de la aplicación web (20 Jun – 26 Jul):

En este sprint se realizaron varias reuniones para la reanudación del proyecto y se procedió a implementar la aplicación web del proyecto dedicada tanto a la gestión administrativa de la aplicación como a una alternativa a la aplicación móvil.

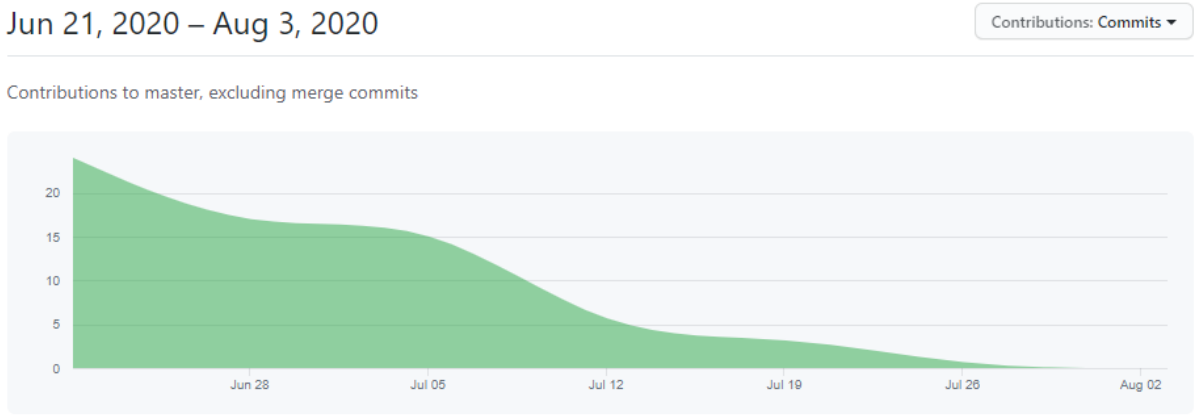

Figura 4. Commits en GitHub de la aplicación Web

4- Despliegue y puesta en marcha (1 Aug – 10 Aug):

En este sprint realizamos el despliegue de la aplicación web en Plesk y la puesta en marcha de la aplicación móvil en los dispositivos de los empleados.

5- Elaboración de la memoria (10 Aug- 31 Aug):

En este sprint se realizará la memoria del trabajo de fin de grado detallando las funcionalidades implementadas.

1.4. ¿Qué es Rijk Zwaan?

Rijk Zwaan es una empresa internacional dedicada a la venta e investigación de semillas con sede centralizada en Holanda. Actualmente se encuentra está posicionada entre las cinco principales compañías de semillas hortícolas a nivel mundial [3].

1.4.1. Visión

Rijk Zwaan desarrolla nuevas variedades hortícolas y comercializa las semillas producidas a partir de estas, lo que implica una gran variedad de procesos de negocio. La clave del éxito está en la óptima integración de estos.

En Rijk Zwaan se desarrollan nuevas variedades con mejoras, es decir, con nuevos rasgos agronómicos beneficiosos para toda la cadena agroalimentaria. Para lograrlo, cruzan distintas variedades y seleccionan sus descendientes basándose en unos criterios concretos. Este proceso resulta cada vez más eficiente gracias a la investigación de alta tecnología.

La empresa actualmente ocupa el cuarto puesto en el mercado de semillas hortícolas, siendo la empresa familiar independiente más grande del sector.

En Rijk Zwaan se sientan las bases de frutas y verduras saludables y atractivas con sus semillas. De este modo pueden contribuir activamente al consumo mundial de estos alimentos, y por consiguiente a la seguridad alimentaria. Juntos trabajan para conseguir su objetivo corporativo: «Compartir un futuro saludable».

1.4.2. Nuestra aplicación en Rijk Zwaan

MarathonRZ será desarrollada para su uso interno en la empresa, aunque no se restringe únicamente el acceso a los usuarios de la intranet, debido a que, nuestro principal acometido es que sean capaces de usarla con la mayor accesibilidad posible, ya estén en las inmediaciones de la empresa o no.

Las principales sedes de Rijk Zwaan Ibérica (Donde nos hemos centrado para la implantación de la aplicación) son las siguientes:

- Rijk Zwaan Ibérica, S.A. Oficinas Centrales (La Cañada | Almería)

- Rijk Zwaan Ibérica, S.A. Estación I+D El Ejido (El Ejido | Almería)
- Rijk Zwaan Ibérica, S.A. Estación I+D La Marina (Cartagena | Murcia)
- Rijk Zwaan Ibérica, S.A. (Valencia)
- Rijk Zwaan Ibérica, S.A. (Marruecos)

Organizar los viajes y envíos entre estas cinco localizaciones será el principal objetivo de nuestra aplicación.

1.5. Estructura del documento

Para la realización de la memoria del proyecto, se ha decidido dividir la misma en diferentes bloques:

Estado del arte

Se hablará de qué otras aplicaciones con una funcionalidad similar a la desarrollada existen actualmente o han existido en el mercado.

Tecnologías y herramientas

Lenguajes, aplicaciones y utilidades que se ha necesitado usar para el correcto análisis, diseño y desarrollo del proyecto.

MarathonRZ

En este apartado se define nuestro proyecto, su estructura, análisis de requisitos, diseño y desarrollo. Está dividido en dos partes:

- Aplicación móvil multiplataforma.

  Se detalla el diseño, la implementación y características de la aplicación móvil, así como el intercambio de datos con la API.

- Aplicación web y API.

  Se detalla el diseño, implementación, características, medidas de seguridad y funcionamiento de la aplicación web y de la API.

Conclusiones y trabajo futuro

Explicación de qué hemos conseguido con nuestro proyecto, así como de los conocimientos adquiridos en la realización de este.

2. Estado del Arte

Las aplicaciones de compartir coche para viajar surgieron con la popularidad de los dispositivos móviles y la facilidad de contactar con gente desconocida y ajena a nuestro círculo cerrado. El hecho de que socializar se haya trasladado a las redes hace que sea más fácil establecer contacto con otras personas de la misma provincia, haciendo que encontrar gente con la que compartir los gastos de viajes sea más sencillo.

Hoy en día, hay aplicaciones gratuitas bastante populares con el fin de compartir viajes, sin embargo, tanto la privacidad, como los costes de gestión o comisiones son factores a tener bastante en cuenta a la hora de usarlas:

2.1. BlaBlaCar

Figura 5. Logo de BlaBlaCar

BlaBlaCar es la aplicación líder para compartir viajes con otras personas. En ella se puede seleccionar cualquier localidad o lugar desde el que quieres viajar y encontrarás una lista de usuarios que, o viajan entre esas dos provincias o están de paso y pueden recogerte para llevarte.

En BlaBlaCar buscas una lista de posibles viajes mediante parámetros tales como la fecha, desde donde viajas, hacia donde viajes, etc. Y obtienes una lista de resultados que cuadran con los introducidos.

Sobre esta lista de viajes, puedes realizar una reserva pagando lo que el conductor (Usuario que ha inscrito el viaje) pide a cada acompañante con el fin de viajar en su coche.

Una vez has realizado tu reserva, la aplicación te deja ponerte en contacto con el conductor a través de un chat para especificar hora y lugar de recogida, así como cualquier otra información adicional.

2.2. Amovens

Figura 6. Logo de Amovens

Amovens es la versión española de la aplicación de BlaBlaCar, diferenciándose en un principio del desarrollo de esta en el cobro de comisiones en cada viaje.

Amovens era la alternativa española completamente gratuita a BlaBlaCar, manejando un modelo de economía completamente orientado hacia sus usuarios, y absteniéndose al uso de comisiones. Sin embargo, a partir de 2016, Amovens empezó a cobrar por gastos de gestión, haciéndose aún más similar a su principal competidor, BlaBlaCar.

Otra de las diferencias respecto a la app líder es que ofrece una gama de coches a seleccionar para realizar nuestros viajes, es decir, podemos alquilar un vehículo a la empresa en caso de necesitarlo.

Respecto al funcionamiento de la aplicación, es exactamente igual al de BlaBlaCar, filtramos los viajes, seleccionamos uno, lo reservamos y nos ponemos en contacto con el usuario una vez realizado el pago.

2.3. Conclusiones

Aunque existen aplicaciones públicas destinadas a compartir coche como las comentadas, el uso de estas para los desplazamientos en una compañía como Rijk Zwaan es inviable debido tanto a la privacidad de los trabajadores como del material.

El uso de una aplicación interna nos permite asegurarnos de que los trabajadores aprovechan al máximo los vehículos sin poner en riesgo su privacidad, seguridad y materiales además de ahorrar posibles gastos en terceros tales como los que imponen *Amovens* o *BlaBlaCar.*

Además, también se ha buscado la mayor simplicidad posible a la hora de registrar o consultar los viajes del resto de trabajadores, mientras que, en las aplicaciones existentes, para registrar un viaje tienes que dedicarle una mayor cantidad de tiempo.

3. Tecnologías y Herramientas

Para el desarrollo de nuestro proyecto hemos usado una amplia variedad de tecnologías y herramientas con el fin de facilitar nuestro desarrollo y asegurar la viabilidad, seguridad y fiabilidad del producto.

El uso de herramientas se ha vuelto imprescindible en el campo del desarrollo de software, algunas de las principales ventajas del uso de herramientas son:

1. Incremento de la calidad y de la eficiencia del desarrollo.
2. Mejora en los procesos seguidos durante el desarrollo.
3. Mayor facilidad de gestión del proyecto.
4. Automatización de tareas.
5. Incremento de la comunicación entre miembros del mismo equipo de desarrollo.

3.1. Tecnologías utilizadas

A continuación, vamos a explicar detalladamente las tecnologías de las que se ha hecho uso durante el desarrollo del proyecto.

3.1.1. C#

Figura 7. Logo corporativo de C#

C# (Pronunciado C-Sharp) es un lenguaje de programación multiparadigma orientado a objetos (OOP) desarrollado por Microsoft en el 2000 como parte de su iniciativa .NET. Es un lenguaje flexible y sencillo con una sintaxis bastante fácil de aprender y similar a Java.

Las aplicaciones desarrolladas en este lenguaje se ejecutan en .NET Framework, que está formado por un entorno en tiempo de ejecución (CLR), que es una adaptación de Microsoft

de una infraestructura de lenguaje común donde se describe un entorno virtual y un conjunto de bibliotecas de clases.

El código escrito en C# produce un código intermedio al ser compilado, a diferencia del código máquina que producen otros lenguajes como C. Este lenguaje intermedio es usado más tarde por el CLR para iniciar una compilación JIT (*Just-in-Time*) y transformarlo en código máquina. De esta manera nuestro CLR puede administrar el código con mucha más eficacia.

También es permitido, aunque no muy recomendable, el uso de código no administrado como punteros o buffers. Este código es denominado como no seguro y permite al usuario designar parte del código que no será administrado por el CLR [4].

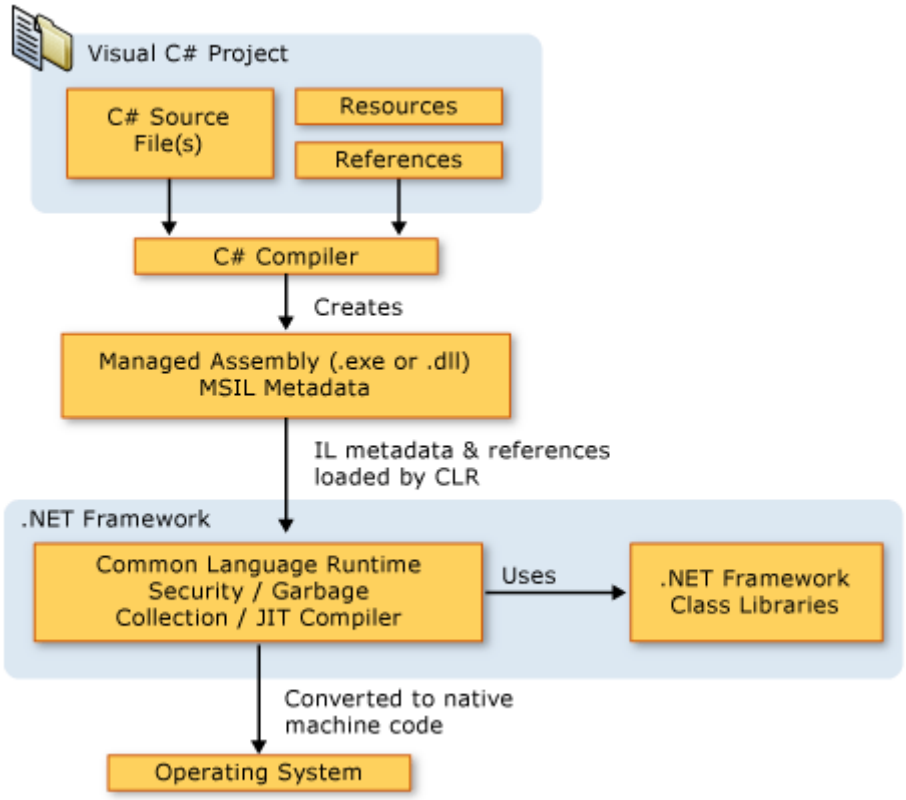

Figura 8. Arquitectura de .NET Framework [4]

3.1.2. Xamarin

Figura 9. Logo corporativo de Xamarin

Xamarin es la plataforma que hemos decidido utilizar para el desarrollo de nuestra aplicación híbrida, esta nos permite desarrollar aplicaciones para iOS, Android y Windows Phone usando únicamente C# y compartiendo más del 90% del código utilizado [2].

Se trata de una plataforma de código abierto anunciada el 16 de mayo del 2011 por Miguel de Icaza y fue posteriormente comprada por Microsoft el 24 de febrero de 2016.

La arquitectura que utiliza Xamarin es diferente dependiendo del sistema operativo en el que se ejecute.

En el caso de Android, se sigue un procedimiento similar al de cualquier aplicación compilada en C#, esta se compila en un lenguaje intermedio que es usado para realizar una compilación JIT obteniendo un ensamblado nativo como resultado.

En el caso de IOS, se usa una compilación AOT (*Ahead of Time*) o compilación anticipada del código en C# a código ARM (*Advanced RISC Machine*) nativo. Xamarin también hace uso de enlaces o *bindings* para permitir que nuestro código C# se comunique con código en Objective-C y viceversa.

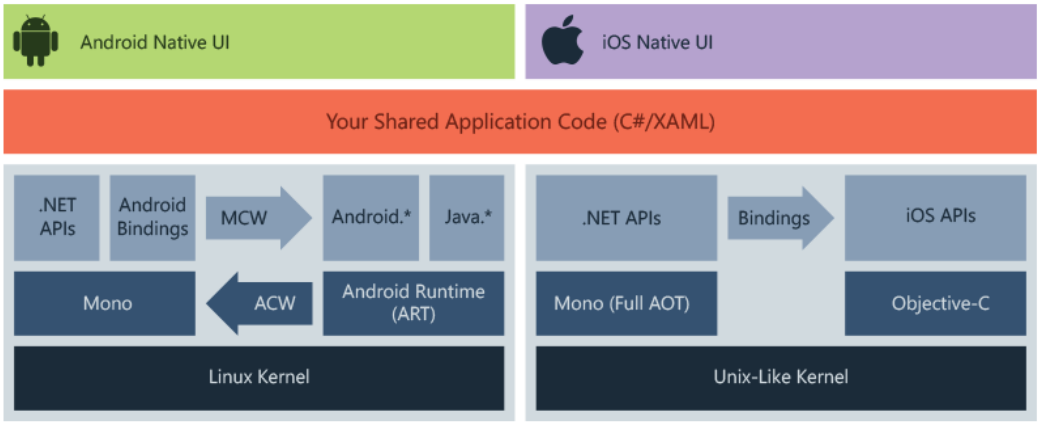

Figura 10. Arquitectura de las aplicaciones de Xamarin [2]

3.1.3. JSON

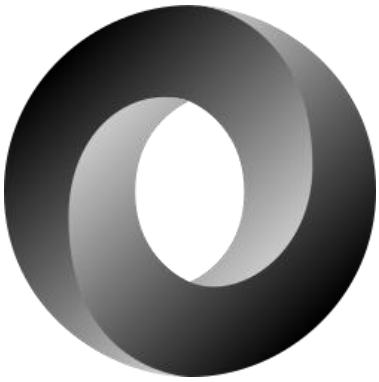

Figura 11. Logo de JSON

JSON es un formato de texto para encapsular e intercambiar datos. JSON es utilizado en nuestro proyecto para enviar/obtener objetos de la API. El intercambio de objetos es muy sencillo además de legible gracias a este lenguaje [5]. La sintaxis que se usa en JSON es la siguiente:

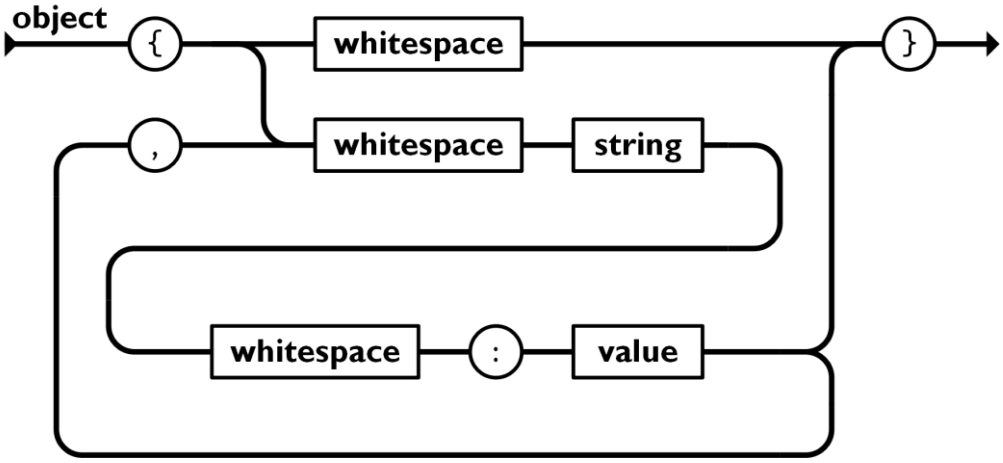

Figura 12. Estructura de elementos en un fichero JSON [6]

Un objeto viene encapsulado por los caracteres “{}” y sus atributos vienen definidos en pares “atributo: valor”, separados a su vez por comas [6].

En caso de ser una colección o un conjunto de elementos, se encapsularán de la misma manera, solo que entre los caracteres “[]”.

3.1.4. JavaScript

Figura 13. Logo de JavaScript en W3C

JavaScript es un lenguaje de programación interpretado, ligero y orientado a objetos que nos permite actualizar y modificar contenido del lado del cliente de manera dinámica.

Fue desarrollado por Brendan Eich en septiembre de 1995 bajo el nombre de Mocha.

En nuestro caso, hemos usado JavaScript en nuestro cliente web con el fin de proporcionar *feedback* al usuario y hacer la interfaz más interactiva, de esta manera nos aseguramos de que la experiencia del usuario con la aplicación sea lo más amigable posible [7].

El uso de JS en nuestro proyecto nos permite entre otras cosas:

-Actualizar los datos mostrados en nuestra página web sin necesidad de recargar la vista.

-Animar elementos o acciones de la interfaz.

-Aumentar la interactividad del usuario con el contenido.

-Validación de valores en formularios web antes de ser enviado, proporcionando *feedback* a través de avisos al usuario.

3.1.5. HTML5

Figura 14. Logo de HTML5 en W3C

HTML (*Hypertext Markup Language*) es un lenguaje de marcado para la creación de páginas web. Este define una estructura y código base para la definición del contenido de una página web. La estructura de la página viene determinada mediante marcadores o etiquetas proporcionando así una enorme adaptabilidad y estructuración del contenido de la web [8].

Fue anunciado por Tim Berners-Lee en 1993, después de mencionarlo como HTML Tags en un documento en 1991 y pasó a convertirse en el 2000 en un estándar internacional en la ISO/IEC 15445:2000.

HTML nos permite la flexibilidad de poder ser creado y editado con cualquier editor de textos, además de la existencia de editores que permiten ver el contenido que estamos diseñando a tiempo real a medida que se va desarrollando nuestra página.

Este lenguaje forma una parte importante de nuestro proyecto ya que gran parte de la interfaz de la página web está compuesta por HTML, que, junto a CSS nos permite tener una interfaz estructurada y bonita.

3.1.6. CSS

Figura 15. Logo de CSS en W3C

CSS u hoja de estilos en cascada es un lenguaje de estilo usado para describir como se presentan los elementos de un documento escrito en un lenguaje de marcado, en este caso, HTML. Este se ha convertido en un pilar fundamental del desarrollo web, permitiéndonos modificar cualquier propiedad de los elementos mostrados en la interfaz, incluyendo la fuente, el color, el tamaño, la separación, … CSS nos ofrece una flexibilidad increíble a la hora de diseñar la interfaz de una aplicación y nos permite ahorrarnos gran cantidad de código mediante la reutilización de las clases diseñadas en CSS [9].

En la actualidad, hay grandes hojas de estilo prediseñadas para dar formato a la interfaz, pero son fácilmente modificables si las importamos en nuestro proyecto. En este caso usaremos la hoja de estilos definida en bootstrap 3.3.7, pero sobrescribiremos gran parte de esta creando nuestra propia hoja de estilos para que la interfaz se muestre y reaccione como nosotros queremos.

El dominio de un lenguaje como CSS nos permite crear auténticas maravillas respecto a cómo reaccionan los componentes de la interfaz web al deslizar por encima de ellos, hacerles click o incluso al seleccionarlos de alguna otra manera.

3.1.7. ASP.Net Core

Figura 16. Logo de ASP.Net Core

ASP.Net Core es un *framework* multiplataforma de código abierto y alto rendimiento destinado a la construcción de aplicaciones web que nos permite, tanto compilar aplicaciones webs como *API's* y *back-ends* para dispositivos móviles [10].

Con ASP.NET Core ejecutamos la plataforma de código abierto .NET Core, la cual nos permite el desarrollo en la mayoría de los sistemas operativos que se usan actualmente (Linux, Windows, MacOS, …).

Las principales ventajas del uso de ASP.Net Core como *framework* para nuestra web y back-end son:

- Alto rendimiento: ASP.Net Core es con diferencia el *framework* que más requests por segundo es capaz de manejar, llegando a ser 7 veces superior al número de requests que soporta Node.JS.

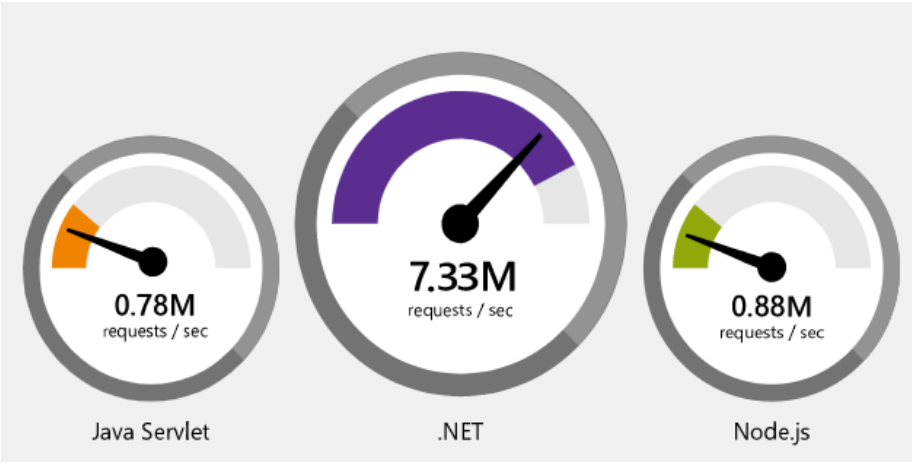

Figura 17. Rendimiento de ASP.Net Core frente a otros frameworks [17]

- De código abierto: Al ser de código abierto (open-source), cualquier persona puede aportar a mejorar el desarrollo, crecimiento y rendimiento de este.
- Apoyado por Microsoft: Microsoft es sin duda una empresa de confianza, y contar con el apoyo de grandes empresas hace que el *framework* no caiga en el olvido y reciba actualizaciones y mejoras de calidad constantemente.
- API y Modelo Vista-Controlador unificadas: En ASP.Net Core, nuestra API y web forman uno solo, es decir, podemos crear aplicaciones webs a la par que un servicio REST para dar soporte a solicitudes HTTPS con JSON o XML.

3.2. Herramientas utilizadas

3.2.1. Visual Studio

Figura 18. Logo de Visual Studio

Visual Studio es el IDE (Entorno de desarrollo integrado) por excelencia de Microsoft. En él se nos permite desarrollar en múltiples lenguajes de programación (C++, C#, F#, Java, Python, etc.) así como el uso de frameworks de desarrollo web, como en nuestro caso, ASP.Net Core y el desarrollo de aplicaciones híbridas con Xamarin [11].

Algunas de las utilidades que nos ofrece Visual Studio para el desarrollo de aplicaciones son:

- Permite escribir código con facilidad en su editor de texto gracias al autocompletado y resaltado de sintaxis que nos ofrece mediante el uso de IntelliSense.
- Facilita la depuración del código gracias a la herramienta de depuración que

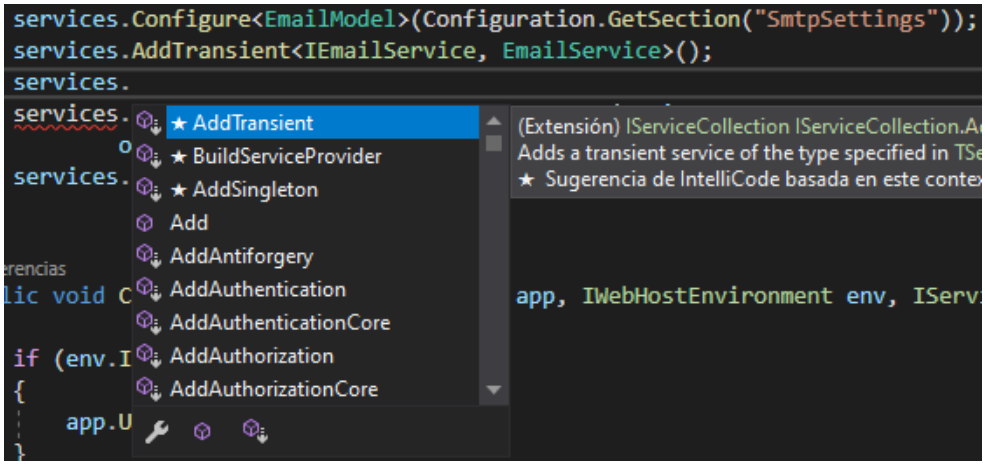

Figura 19. Autocompletado de Visual Studio

implementa, el cual, además de proporcionarnos una información útil sobre el rendimiento de la aplicación, permite un estado de inspección flexible y el rastreo de los procesos de la aplicación por separado.

- Integración con GitHub para el control de versiones.

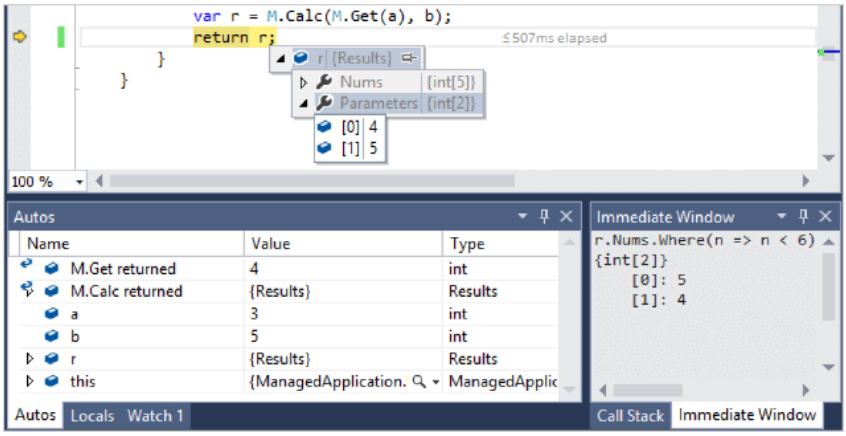

Figura 20. Depurador de Visual Studio

Nuestra aplicación ha sido desarrollada completamente usando Visual Studio, tanto la web, como la API y la aplicación móvil.

3.2.2. Microsoft SQL Server

Figura 21. Logo de SQL Server

Microsoft SQL Server es un sistema de gestión de base de datos relacional, desarrollado por la empresa Microsoft [18].

Para desarrollar una aplicación en un entorno de Microsoft, estamos casi obligado a usar Microsoft SQL Server, y no solo para esperar un mejor funcionamiento de nuestra aplicación, si no para facilitar el desarrollo de la aplicación a través de las características que tienen para implementarse entre ellas.

Visual Studio hace uso de Microsoft SQL Server Express durante el desarrollo de la aplicación, esta es una pequeña instancia local de una base de datos de SQL Server, por tanto, nos garantiza la funcionalidad de la aplicación en este tipo de bases de datos además de otorgarnos herramientas para trasladar estas tablas a una base de datos SQL Server con facilidad.

3.2.3. Postman

Figura 22. Logo de Postman

Postman es una herramienta utilizada para comprobar el correcto funcionamiento de la API, simplificando el desarrollo y fase de pruebas y con el fin de acelerar el desarrollo de la misma.

Es una herramienta muy útil durante el desarrollo ya que podemos controlar a la perfección tanto los headers como el contenido que le enviamos a nuestra API en nuestras solicitudes POST o que recibimos cuando realizamos un GET.

3.2.4. GitHub

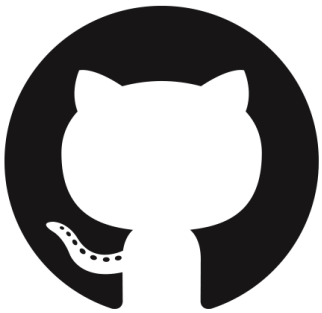

Figura 23. Logo de GitHub

GitHub es la herramienta líder de control de versiones, la cual nos permite guardar nuestro código en repositorios, tanto privados como públicos, seguros e independientes a nuestro equipo.

Además del control de versiones, GitHub ofrece una serie de características para ayudar a los desarrolladores a mantener un buen flujo de trabajo, tales como la inclusión de *bots* en el proyecto que avisan de vulnerabilidades de seguridad o la gestión de *tokens* de acceso de nuestra aplicación [12].

3.2.5. Visual Paradigm

Figura 24. Logo de Visual Paradigm

Visual Paradigm es una herramienta CASE UML para el diseño y modelado de software. Su diseño está centrado en torno a casos de uso y enfocado a la lógica de negocio, además,

permite la generación de código partiendo de los diagramas o al revés, haciendo uso de ingeniería inversa para construir diagramas a partir de código ya desarrollado.

4. MarathonRZ

MarathonRZ es una aplicación de viajes cuyo principal objetivo es facilitar el registro y gestión de los viajes entre sedes para los trabajadores de Rijk Zwaan así cómo mantenerse al día entre ellos cuando alguien registra un viaje de interés hacia un destino al cual un compañero envía material.

4.1. Análisis de requisitos

4.1.1. Historias de Usuario

Como primer paso al desarrollo de la aplicación, se realizó una reunión con el cliente para aclarar la funcionalidad de la misma, así como la estructura de los modelos que usaremos. Partiendo de la funcionalidad que nos ha especificado el cliente de la aplicación, realizamos las historias de usuario que nos servirán para el desarrollo de las distintas características funcionales y requisitos de la misma.

Como	Usuario sin autentificar
Quiero	Autentificarme en el sistema
Para	Poder ver y registrar viajes

Como	Usuario sin autentificar
Quiero	Recuperar mi contraseña
Para	Poder recuperar mi contraseña en caso de que se me olvide

Como	Usuario autentificado
Quiero	Ver los viajes de los demás
Para	Poder estar al día de los viajes de los otros trabajadores y contactar con ellos

Como	Usuario autentificado
Quiero	Recibir notificaciones de viajes
Para	Estar al día de cuando se registra un viaje de mi interés

Como	Usuario autentificado
Quiero	Marcar localizaciones de interés
Para	Poder ser notificado cuando se realicen viajes entre ellas

Como	Usuario autentificado
Quiero	Recibir notificaciones de viajes
Para	Estar al día de cuando se registra un viaje de mi interés

Como	Usuario autentificado
Quiero	Registrar un viaje
Para	Informar a los trabajadores interesados

Como	Usuario autentificado
Quiero	Modificar mis datos
Para	Modificar mis datos personales y de contacto en caso de que cambien

Como	Usuario autentificado
Quiero	Poder registrar viajes con rapidez y facilidad
Para	No perder demasiado tiempo

Como	Usuario autentificado
Quiero	Modificar mis viajes
Para	Cambiar la información en caso de error u otros problemas

Como	Usuario autentificado
Quiero	Eliminar mis viajes
Para	No seguir mostrándolo en caso de cancelación

Como	Usuario autentificado
Quiero	Filtrar los viajes
Para	Poder buscar viajes de cierta persona, fecha o lugar

Como	Usuario autentificado
Quiero	Ver un histórico de viajes
Para	Consultar los viajes que se han realizado

Como	Usuario autentificado
Quiero	Ser recordado
Para	No tener que iniciar sesión constantemente

Como	Usuario autentificado
Quiero	Poder contactar con facilidad con otros usuarios
Para	Informar al dueño del viaje de que estoy interesado

Como	Usuario privilegiado
Quiero	Registrar viajes a otros usuarios
Para	Registrar viajes a los usuarios que no disponen de los medios

Como	Usuario administrador
Quiero	Ver el listado de usuarios
Para	Poder salir de mi cuenta cuando lo desee

Como	Usuario administrador
Quiero	Añadir nuevos usuarios
Para	Registrar usuarios en el sistema

Como	Usuario administrador
Quiero	Modificar el rol de usuarios
Para	Modificar las funcionalidades que puede usar el usuario

Como	Usuario administrador
Quiero	Modificar los datos de usuarios
Para	Modificar los datos de otros usuarios en caso de que cambien

Como	Usuario administrador
Quiero	Modificar contraseña de otros usuarios
Para	Modificar la contraseña en caso de que un usuario esté teniendo problemas para iniciar sesión

Como	Usuario administrador
Quiero	Eliminar a otros usuarios de la aplicación
Para	Eliminar su cuenta en caso de que ya no pertenezcan a la empresa o estén usando otra cuenta

4.1.2. Requisitos funcionales

Partiendo de las historias de usuario definidas anteriormente, podemos dar paso a los requisitos funcionales de los que hará uso la aplicación.

Identificación del requisito	RF001
Nombre del requisito	Autentificación en el sistema
Descripción del requisito	El sistema debe ser capaz de gestionar la autentificación de usuarios.
Características	La autentificación debe ser simple, con una opción de recordad al usuario en caso de que él quiera, el usuario debe acceder con su correo electrónico y su contraseña.
Prioridad	Alta (X) Media ( ) Baja ( )

Identificación del requisito	RF002
Nombre del requisito	Recuperación de la contraseña
Descripción del requisito	El sistema debe ser capaz generar y gestionar peticiones de cambio de contraseña de los usuarios que la olviden y quieran recuperarla.
Características	Nuestra ventana de autentificación debe tener un apartado de recordar contraseña en el que, especificando el correo, el usuario puede acceder y cambiar su contraseña mediante el enlace que se le enviará.
Prioridad	Alta (X) Media ( ) Baja ( )

Identificación del requisito	RF003
Nombre del requisito	Gestión de viajes entre sedes
Descripción del requisito	El sistema debe ser capaz de mostrar una lista de viajes a los usuarios autentificados así como de gestionar el registro, modificación y eliminación de los mismos.
Características	El usuario, ya sea usuario estándar, usuario privilegiado o administrador, debe poder ver el listado de los viajes una vez se haya autentificado.
Prioridad	Alta (X) Media ( ) Baja ( )

Identificación del requisito	RF004
Nombre del requisito	Envío de notificaciones
Descripción del requisito	El sistema debe ser capaz de gestionar el envío de notificaciones por correo a los usuarios interesados.
Características	Cada vez que se registre un viaje, el sistema enviará un correo a los usuarios interesados avisándoles del viaje.
Prioridad	Alta (X) Media ( ) Baja ( )

Identificación del requisito	RF005
Nombre del requisito	Gestión de localizaciones de interés
Descripción del requisito	El sistema debe ser capaz de gestionar el registro y eliminación de localizaciones de interés
Características	Los usuarios administradores deben poder añadir nuevas localizaciones sobre las que registrar viajes.
Prioridad	Alta (X) Media ( ) Baja ( )

Identificación del requisito	RF006
Nombre del requisito	Gestión de usuarios
Descripción del requisito	El sistema debe ser capaz de gestionar el registro, modificación y eliminación de usuarios en la aplicación.
Características	Los usuarios administradores deben poder añadir nuevos usuarios, modificar sus datos y eliminarlos.
Prioridad	Alta (X) Media ( ) Baja ( )

Identificación del requisito	RF007
Nombre del requisito	Filtrado de viajes
Descripción del requisito	El sistema debe ser capaz de filtrar el listado de viajes.
Características	Los usuarios deben poder filtrar los viajes ya sea por nombre del propietario, su email, la localización del viaje o incluso fecha.
Prioridad	Alta ( ) Media ( ) Baja (X)

Identificación del requisito	RF008
Nombre del requisito	Mostrar histórico de viajes
Descripción del requisito	El sistema debe ser capaz de mostrar los viajes cuya fecha ha pasado, a modo de un histórico.
Características	La página principal tendrá la opción de filtrar entre los viajes ya finalizados o los viajes que aún están por realizarse.
Prioridad	Alta ( ) Media ( ) Baja (X)

Identificación del requisito	RF009
Nombre del requisito	Recordar al usuario
Descripción del requisito	El sistema debe ser capaz de recordar el acceso de un usuario para que no tenga que acceder muchas veces.
Características	El sistema debe generar cookies en el navegador del usuario con el fin de recordar las credenciales del mismo y autentificarle automáticamente.
Prioridad	Alta ( ) Media (X) Baja ( )

Identificación del requisito	RF010
Nombre del requisito	Gestión de roles
Descripción del requisito	El sistema debe ser capaz de gestionar los roles de los usuarios.
Características	El usuario puede tener tres roles, administrador, privilegiado o usuario. El usuario administrador debe ser capaz de modificar los roles de cualquier otro usuario, así como de especificarlo a la hora de registrarlo.
Prioridad	Alta (X) Media ( ) Baja ( )

Identificación del requisito	RF011
Nombre del requisito	Gestión de datos personales
Descripción del requisito	El sistema debe ser capaz de realizar cambios en los datos personales de los usuarios.
Características	Los usuarios deben ser capaces de modificar sus datos personales una vez autentificados, siempre y cuando los nuevos datos sean válidos.
Prioridad	Alta ( ) Media (X) Baja ( )

4.1.3. Requisitos no funcionales

Los requisitos no funcionales apuntan al correcto funcionamiento de la aplicación. Suelen denominarse atributos de calidad del sistema.

Nuestros requisitos no funcionales giran en torno a la velocidad de inscripción y de contacto con los empleados, de manera que, tanto registrar un viaje como consultarlos en una lista consuma el menor tiempo posible con una buena interfaz y rápida respuesta de la API.

Además, nuestra aplicación deberá funcionar en la versión de iOS y Android de los dispositivos que se usan en la empresa.

Identificación del requisito	RNF001
Nombre del requisito	Inscripción rápida y sencilla
Descripción del requisito	La respuesta del servidor al añadir nuevos viajes debe ser menor de 100ms.
Prioridad	Alta (X) Media ( ) Baja ( )

Identificación del requisito	RNF002
Nombre del requisito	Soporte para múltiples dispositivos móviles
Descripción del requisito	La aplicación debe funcionar correctamente en las versiones de Android 9 o superior e iOS 13.6 o superior.
Prioridad	Alta (X) Media ( ) Baja ( )

Identificación del requisito	RNF003
Nombre del requisito	Disponibilidad
Descripción del requisito	La aplicación debe estar disponible para su uso la mayor parte del tiempo posible.
Prioridad	Alta (X) Media ( ) Baja ( )

Identificación del requisito	RNF004
Nombre del requisito	Fiabilidad
Descripción del requisito	La aplicación debe ser estable y poder recuperarse frente a fallos sin problemas.
Prioridad	Alta (X) Media ( ) Baja ( )

Identificación del requisito	RNF005
Nombre del requisito	Seguridad
Descripción del requisito	La aplicación debe ser lo más segura posible para no exponer los datos personales de cada usuario.
Prioridad	Alta (X) Media ( ) Baja ( )

Identificación del requisito	RNF006
Nombre del requisito	Facilidad de uso
Descripción del requisito	La aplicación debe ser simple de usar, permitiendo realizar operaciones con el menor número de acciones posibles
Prioridad	Alta (X) Media ( ) Baja ( )

Identificación del requisito	RNF007
Nombre del requisito	Diseño web adaptable
Descripción del requisito	La página web debe adaptarse al tamaño de la pantalla, permitiendo su uso en cualquier tipo de dispositivo
Prioridad	Alta (X) Media ( ) Baja ( )

4.2. Diseño general

Tanto la aplicación web como la aplicación móvil tienen un diseño similar, sin embargo, difieren en algunos aspectos del mismo.

4.2.1. Uso de colores corporativos

Para el diseño de la interfaz, se ha buscado crear un entorno familiar y agradable para los trabajadores de Rijk Zwaan, por ello, es importante tanto la inclusión de elementos cotidianos en su trabajo diario como el uso de los colores corporativos. Por ello, el diseño se ha basado en la siguiente paleta de colores:

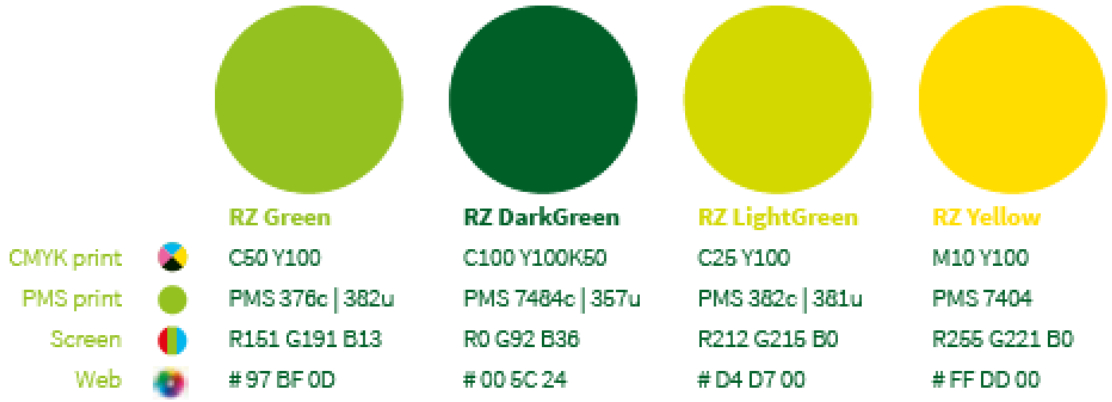

Figura 25. Colores corporativos de Rijk Zwaan

4.2.2. Diagrama de casos de uso

Nuestro diagrama de casos de uso está formado por cuatro actores, si contamos la funcionalidad que puede llegar a tener el usuario antes de identificarse en la aplicación:

- Usuario: Se trata del usuario sin autentificar, por defecto se le redirige a la ventana de autentificación y se le da la alternativa entre identificarse o recuperar su contraseña.
  Si decide identificarse, puede marcar opcionalmente si desea ser recordado por la aplicación evitando así tener que identificarse cada vez que accede a la web o aplicación móvil de la misma.
- Usuario autentificado: Es el usuario estándar de la aplicación, y por tanto tiene acceso a las características principales de la misma. Este puede registrar viajes, modificar y eliminar los suyos propios, así como consultar los viajes del resto con la ayuda de filtros y un histórico.

Además, también puede modificar sus propios datos personales (Correo electrónico, nombre, número de teléfono, etc.) y gestionar los lugares de los que desea recibir notificaciones.
Este rol es común a todos los autentificados, es decir, tanto el administrador como los usuarios privilegiados también tienen acceso a todas las funcionalidades del usuario autentificado.

- Usuario privilegiado: Este rol ha sido creado para los encargados de la gente de finca, ya que muchos de ellos no disponen de dispositivo móvil u ordenador corporativo, de manera que se les ha habilitado la posibilidad de especificar el nombre de la persona que será el propietario del viaje.
- Administrador: Es el encargado de la completa gestión de la aplicación, por tanto, tiene acceso y permisos sobre cualquier otro usuario, este puede gestionar completamente los viajes, los usuarios y las localizaciones entre las que puede especificar alguien que quiera realizar un viaje.

Tres de los actores (Autentificado, Privilegiado y Administrador), son usados como roles de nuestra aplicación, es decir, la funcionalidad de la aplicación dependerá del rol que desempeñemos en la misma, permitiendo el acceso a determinadas funcionalidades a determinados roles (Las cuales se pueden observar en el diagrama).

El actor de Usuario sin autentificar no es controlado en nuestra aplicación, ya que no se necesitan ningunas credenciales guardadas y sus funcionalidades pueden ser accedidas por cualquiera. Usamos la etiqueta *AllowAnonymous* para permitir que estos usuarios hagan ciertas tareas.

Tal y como se ha especificado en el apartado de herramientas, se ha usado Visual Paradigm para el diseño del diagrama de casos de uso, así como la metodología estudiada en las asignaturas Ingeniería del Software y Modelado y Diseño del Software de la especificación de Ingeniería del Software.

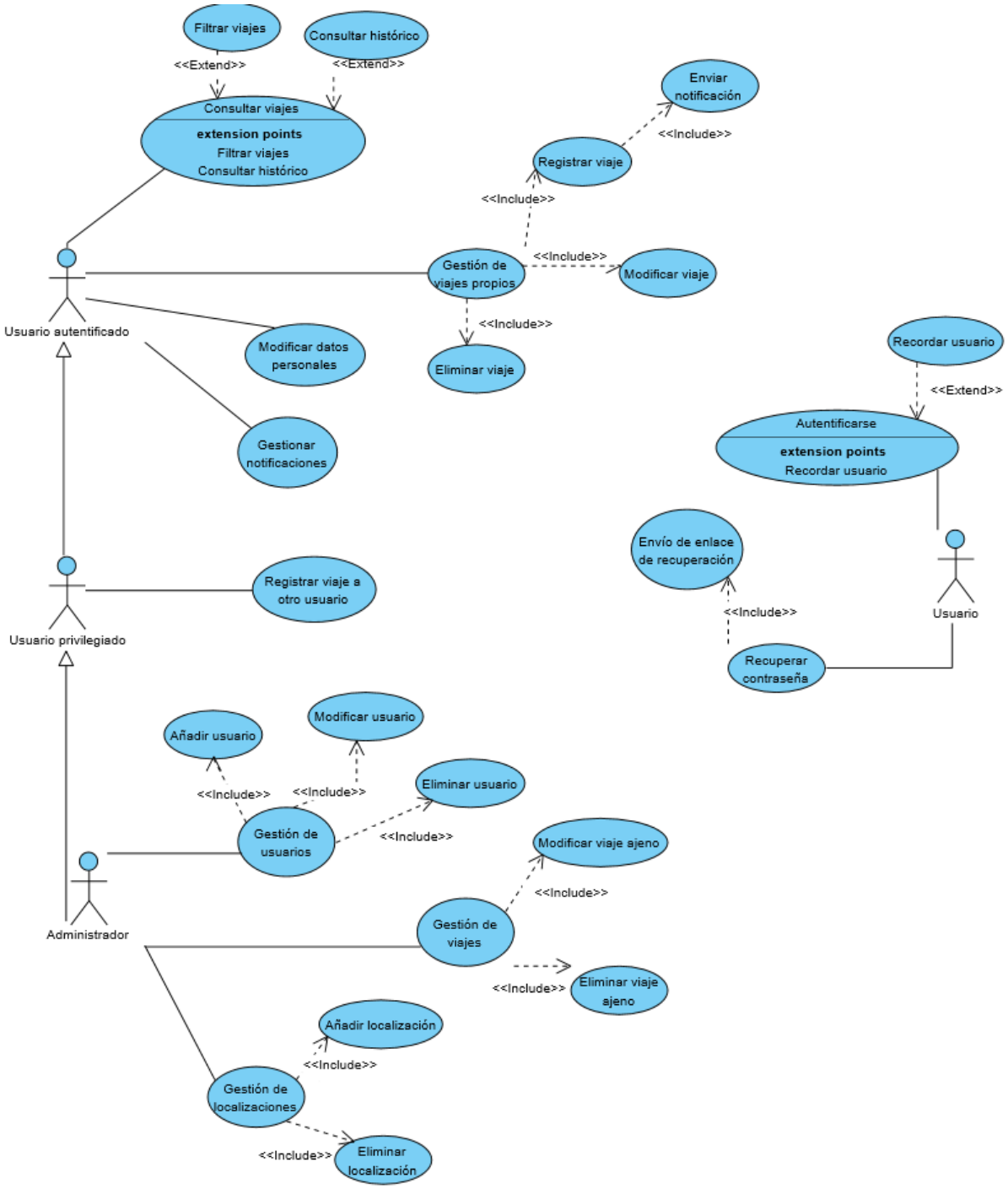

Figura 26. Diagrama de casos de uso de MarathonRZ

4.2.3. Modelo de datos: Diagrama de clases

Nuestro diagrama de clases será el encargado de determinar los modelos que se utilizarán en la aplicación y en nuestra base de datos con el fin de almacenar y manejar la información.

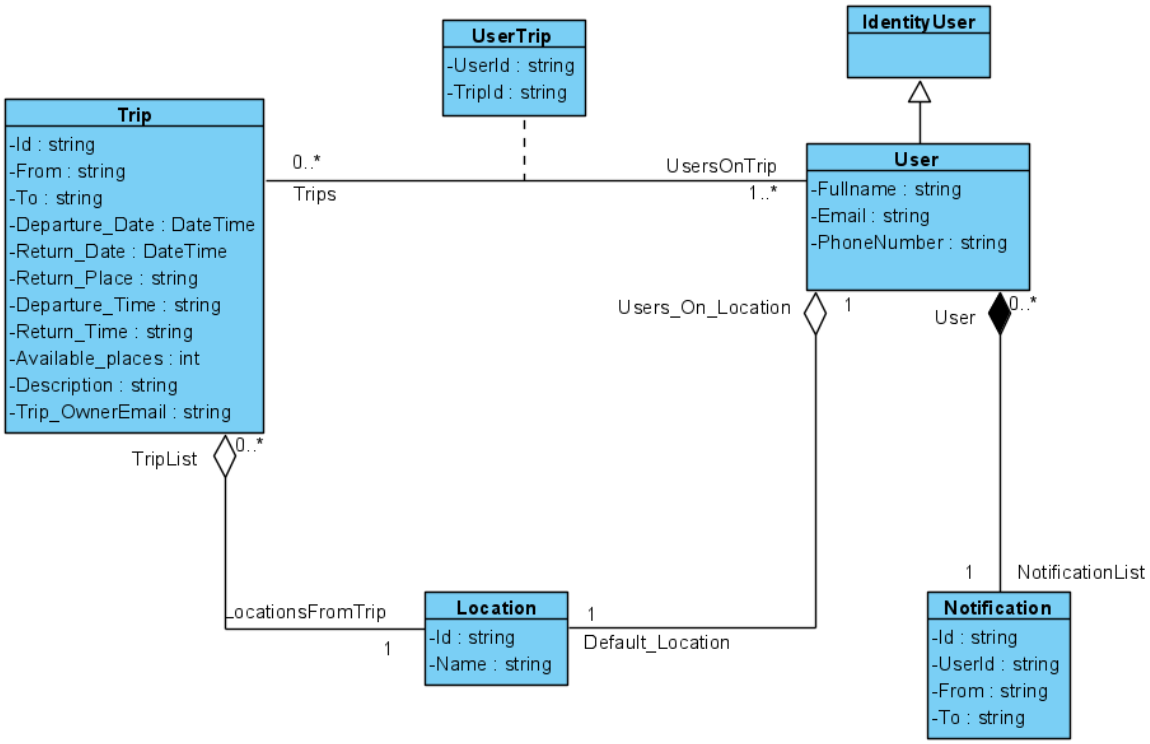

Figura 27. Diagrama de clases

En nuestro diagrama, se pueden ver reflejadas 5 clases:

- Trip: Clase encargada de guardar la información de cada viaje, su origen, destino, descripción, fechas, etc.
- User: Clase encargada de guardar la información relacionada con el usuario, esta clase hereda de IdentityUser. No mostramos los diagramas relacionados con IdentityUser debido a que esta API no es desarrollada por nosotros, sino que la utilizamos para una mejor gestión de los usuarios, siempre es más seguro el uso de sistemas fiables que inventar uno propio.
- UserTrip: Clase encargada de registrar la lista de viajes en los que está registrado cada usuario y viceversa.
- Location: Lugares de la aplicación sobre los que se puede viajar.
- Notification: Notificaciones de cada usuario, se usa una composición debido a que, si un usuario es eliminado, toda la lista de notificaciones también.

Pasamos a mostrar el diagrama de clases de Identity, del cual hereda el usuario de nuestra aplicación.

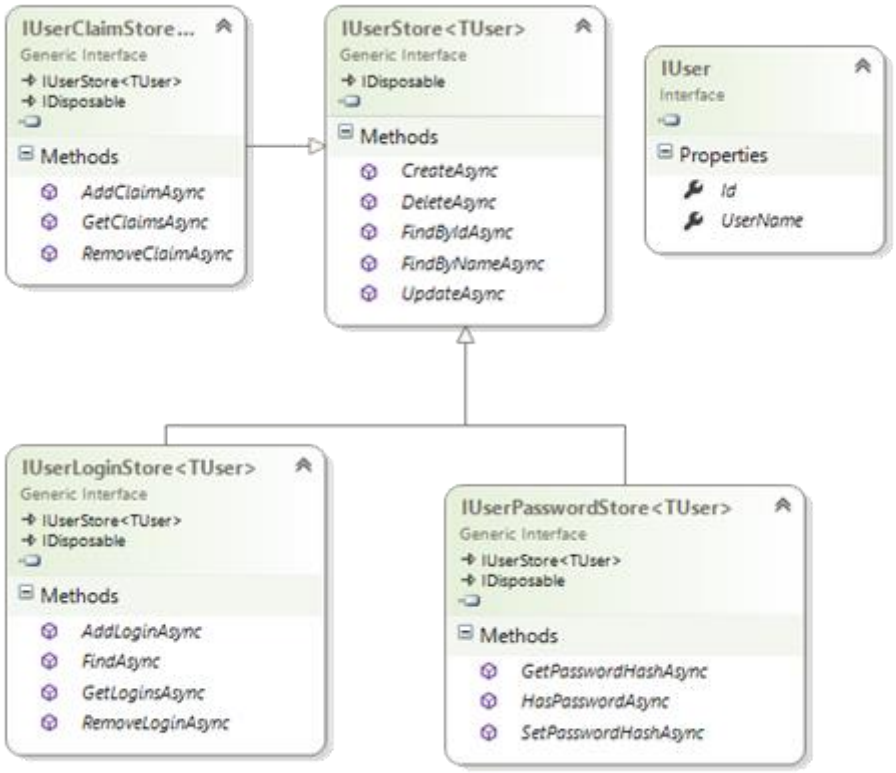

Figura 28. Diagrama de clases de Identity [19]

4.3. Arquitectura software

La arquitectura de nuestra aplicación cuenta con una aplicación web y API y una aplicación móvil multiplataforma.

La aplicación web ha sido desarrollada en ASP.Net Core, y trabaja con la información de la base de datos directamente a través de *EntityFramework* con el atributo *Context.*

A la hora de transmitir/recibir información con nuestra aplicación móvil, se utilizan HTTPRequests con objetos embebidos en JSON (Ver figura 1).

Esta información pasa por nuestro *middleware* antes de llegar a cada uno de nuestros extremos, hacemos uso de Middlewares con el fin de controlar las solicitudes y respuestas.

En ASP.Net Core se hace mucho uso de estos Middlewares con el fin de controlar autenticación, enrutamiento, redirección HTTPS, CORS (*Cross-Origin Resource Sharing*), etc.

Por tanto, siempre que se realiza una solicitud HTTP desde nuestra aplicación multiplataforma, esta pasa por una lista de middlewares para asegurarnos del correcto funcionamiento de nuestra aplicación y de devolver la respuesta correcta.

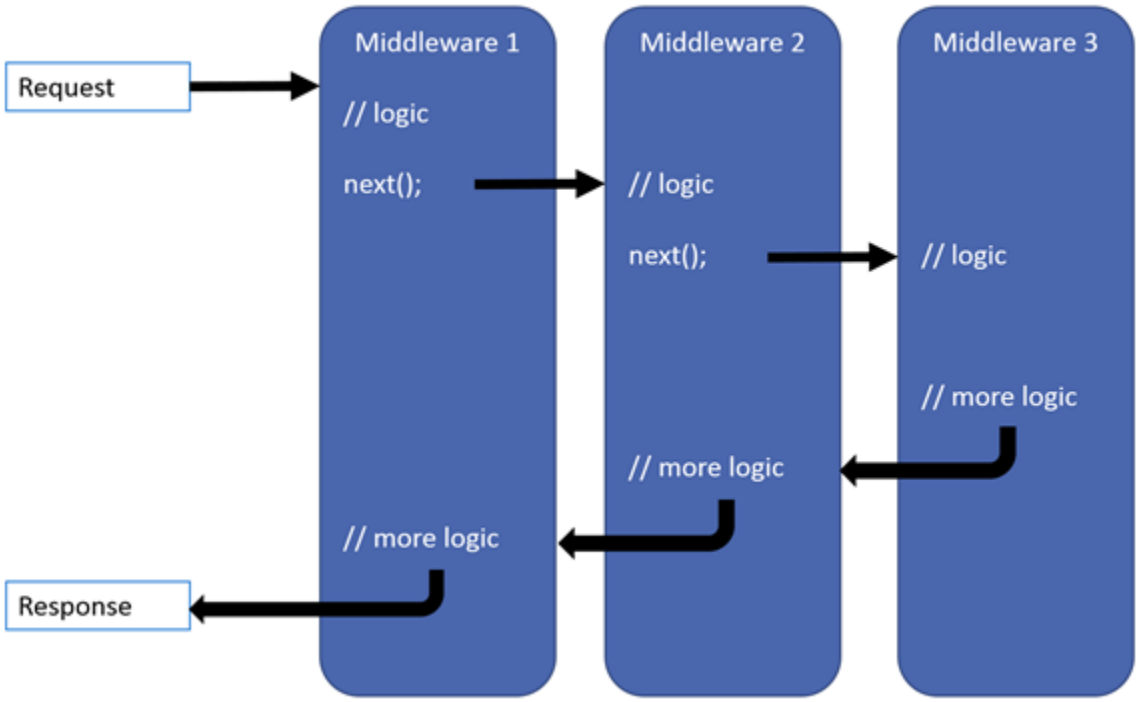

Figura 29. Uso de middlewares en ASP.Net Core [20]

4.4. Aplicación móvil multiplataforma

La aplicación multiplataforma de MarathonRZ ha sido desarrollada usando Xamarin Forms, el cual es un marco de trabajo de código abierto (*Open-source Framework*) de Microsoft para la construcción de aplicaciones multiplataforma, capaces de ejecutarse sin problemas tanto en iOS como en Android o Windows Phone.

4.4.1. Patrón de diseño

Para el desarrollo de la aplicación móvil, se ha utilizado el patrón de diseño MVVM (*Model* - View - *ViewModel*), el cual nos ayuda a separar la lógica de negocio de la interfaz o vista.

Este patrón nos permite, entre otras ventajas, crear pruebas unitarias para la vista y modelo por separado, rediseñar la interfaz sin tocar nuestro código y estructurar mejor el trabajo por equipos en una empresa [13].

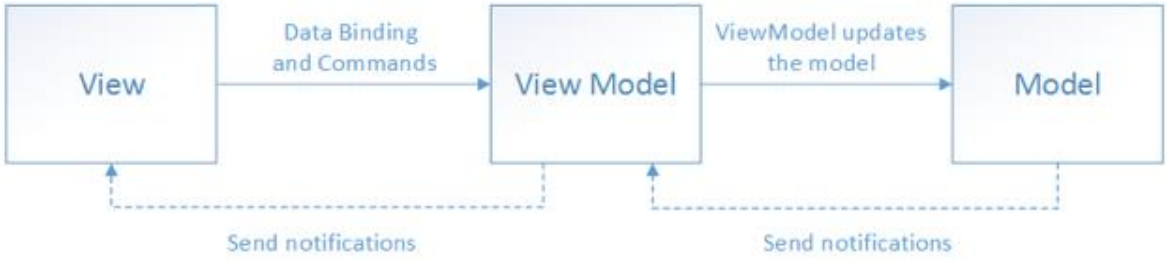

Figura 30. Patrón MVVM

En nuestro proyecto, tenemos la siguiente estructura de ficheros:

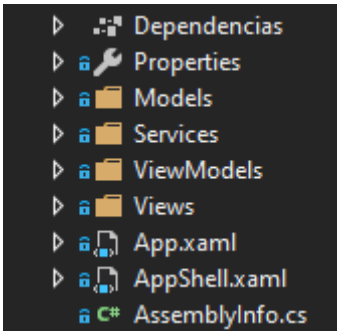

Figura 31. Estructura de carpetas en aplicación híbrida

- Modelos: Aquí almacenamos todos los modelos que usa nuestra aplicación, en este caso usamos un modelo para los viajes, otro para los usuarios y otro para las notificaciones. Además, para simplificar las operaciones de GET y POST con nuestra API, también tenemos dos modelos auxiliares llamados *DBUser* y *DBTrip*.

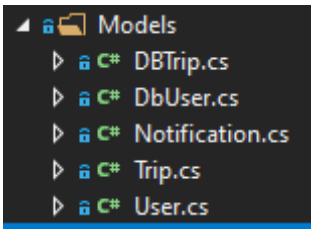

Figura 32. Modelos usados en la aplicación híbrida

- Servicios: Encargada de mantener todos los métodos relacionados con las solicitudes POST y GET que realiza nuestra aplicación con la API, además de una clase auxiliar para el manejo de alertas denominada "*IMessage*".

A estas clases las hemos denominado *DataStore* o almacenes de datos. La estructura que se ha seguido para esas clases es la siguiente:

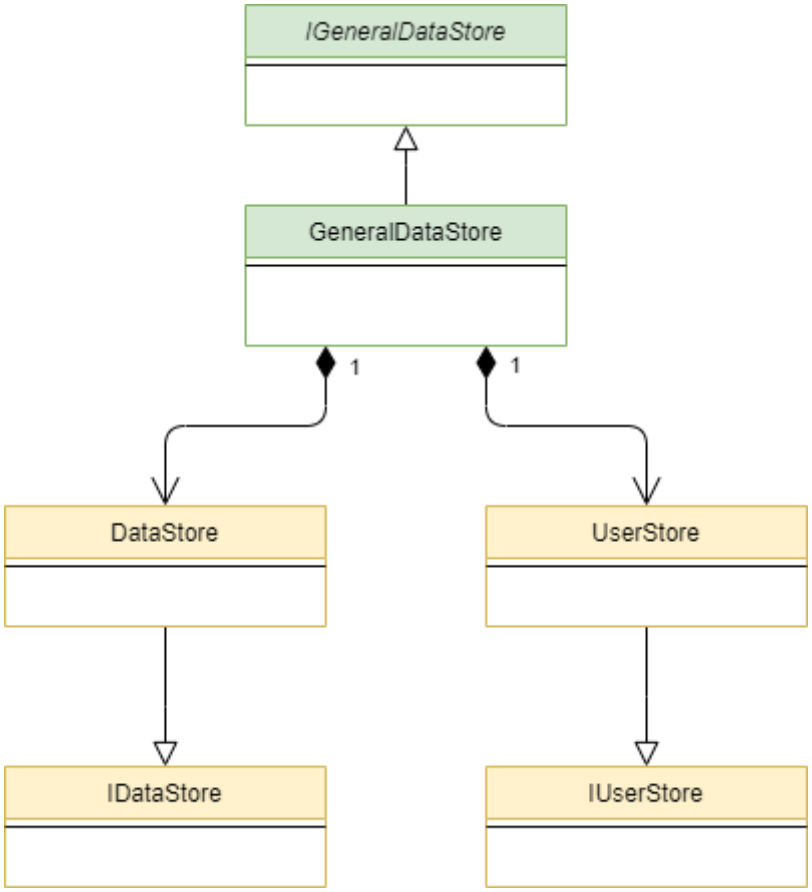

Figura 33. Estructura de clases para la conexión con el servicio REST

Esta estructura de clases fue estudiada en Modelado y Diseño de Software, y nos permite aislar los métodos dependiendo del modelo que se necesite en ese momento. De esta manera tenemos un código bastante más legible y organizado. En ella, la clase *UserStore* será la encargada de realizar las peticiones POST, GET, PUT y DELETE y de los usuarios, y la clase *DataStore* la de realizarlas para los viajes.

- *Viewmodels*: Aquí guardamos todos los encargados de preparar los datos que serán usados por la vista. Normalmente, usaremos un *Viewmodel* para controlar lo que muestra cada una de nuestras vistas, sin embargo, vamos a reutilizar *TripsViewModel* en tres de ellas gracias a que comparten una estructura similar, evitando así generar código repetitivo.

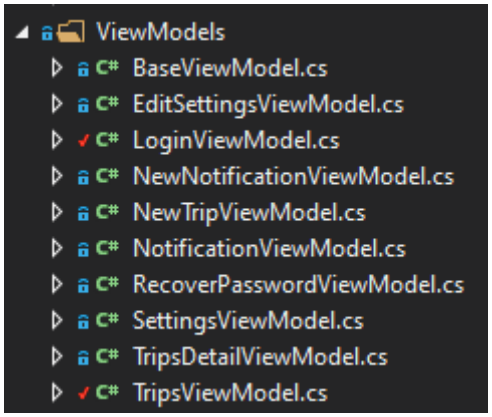

Figura 34. Viewmodels usados en la aplicación híbrida

- Vistas: En Xamarin, las vistas están formadas por dos definiciones parciales de la clase, una que contiene el lenguaje de marcado encargado de mostrar correctamente la interfaz, y otra que contiene código en C# encargado de conectar las variables de la interfaz con el *ViewModel* a través del atributo *BindingContext*.

  Nuestra aplicación híbrida está compuesta por un total de 11 vistas:

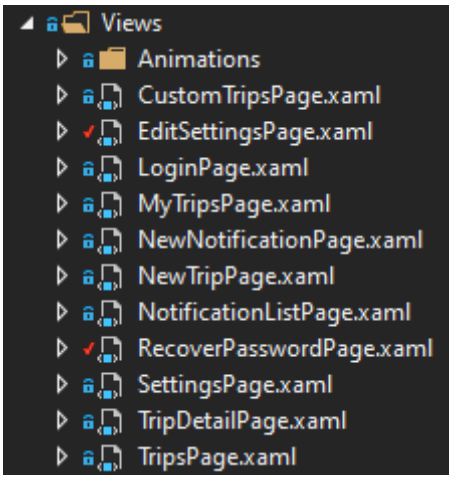

Figura 35. Vistas usadas en la aplicación híbrida

4.4.2. Diseño de la interfaz

La interfaz de usuario de la aplicación multiplataforma está compuesta por un *dashboard* con tres opciones principales y una de configuración:

- Viajes cercanos: Se muestran los viajes que parten desde y hacia la localización del usuario.
- Mis viajes: Se muestran todos los viajes de los que forma parte el usuario.
- Todos los viajes: Se muestran todos los viajes cuya fecha de inicio es igual y posterior a la actual

Estas tres opciones del dashboard comparten funcionalidades, tales como la de añadir o la de filtrado.

En el apartado de configuración podremos ajustar tanto las notificaciones que nos llegan al correo electrónico como los parámetros (Correo electrónico, nombre, teléfono y localización por defecto) de la cuenta del usuario.

También tendremos una opción para salir de la aplicación, siempre y cuando queramos entrar con otra cuenta.

4.4.2.1. Viajes disponibles

Vista en la que se muestran todos los viajes disponibles, cuya fecha inicial es igual o posterior a la actual.

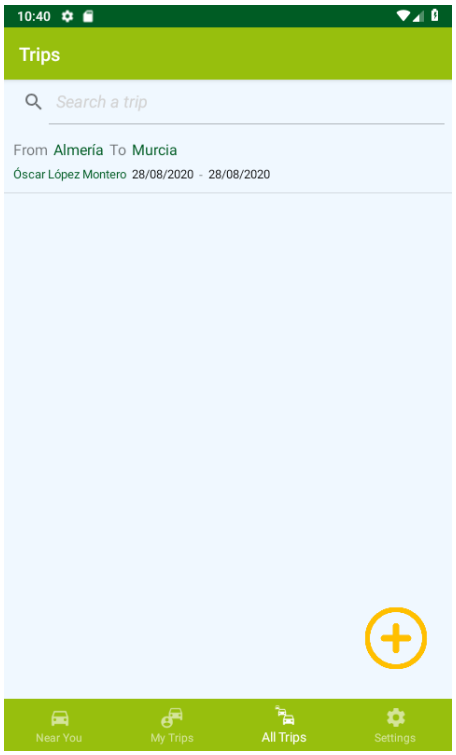

Figura 36.Vista de viajes disponibles

4.4.2.2. Viajes propios

Vista en la que se muestran tus propios viajes y lo que has marcado como favoritos.

Figura 37. Vista de mis viajes

4.4.2.3. Viajes recomendados

Vista en la que se muestra los viajes entre la localización por defecto del usuario.

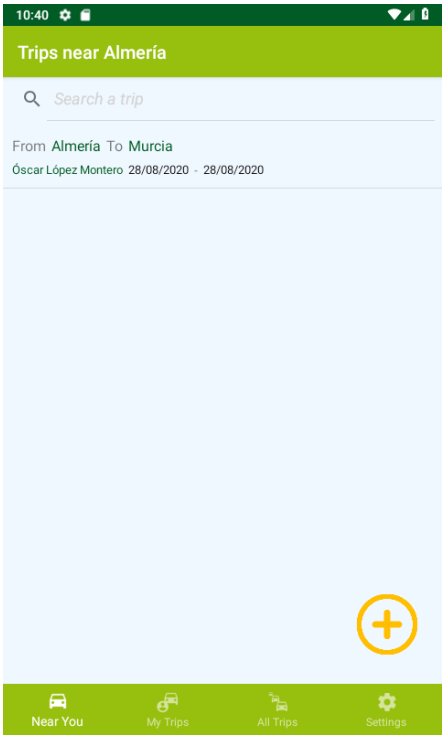

Figura 38. Vista de viajes cercanos

4.4.2.4. Nuevo viaje

Vista encargada del registro de nuevos viajes en la aplicación, es accesible desde cualquier vista de lista de viajes mediante el botón "+".

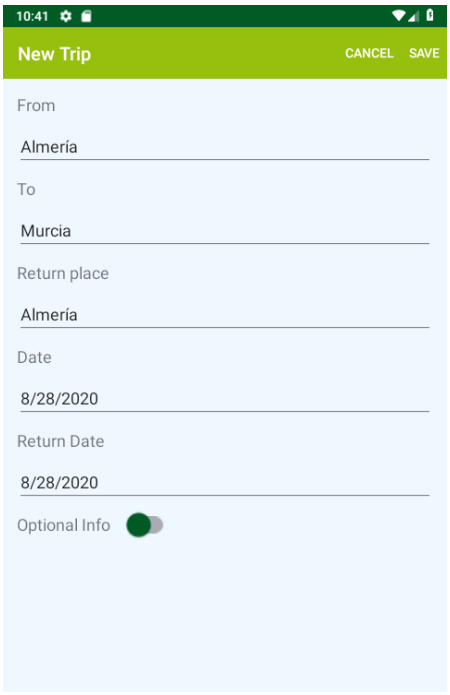

Figura 39. Vista de registro de nuevo viaje

4.4.2.5. Autenticación

Vista desde la que se accede a la aplicación. Contiene un login y una opción de recuperar contraseña.

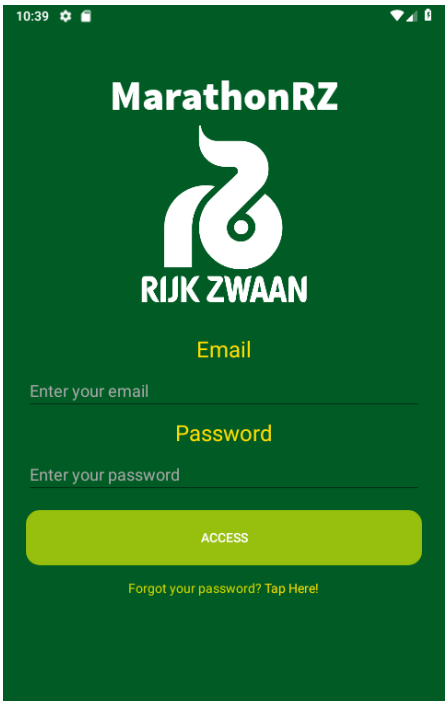

Figura 40. Vista del login de la aplicación

4.4.2.6. Recuperar contraseña

Vista que permite recuperar la contraseña del usuario introduciendo el correo electrónico.

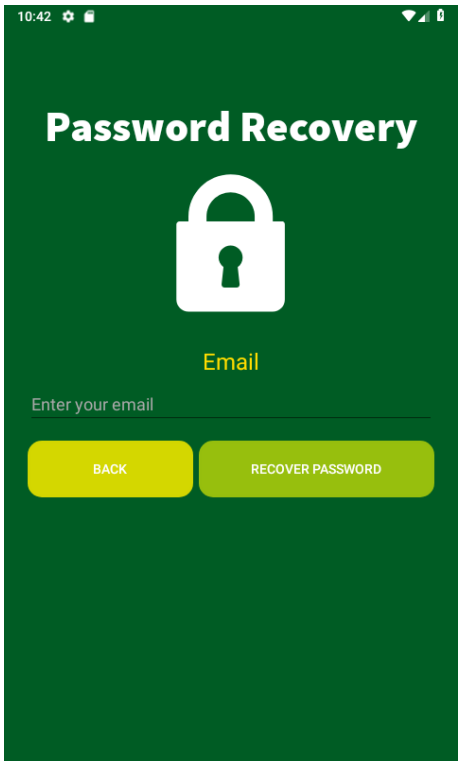

Figura 41. Vista de recuperación de contraseña

4.4.2.7. Detalles del viaje

Vista encargada de mostrar los datos de un viaje.

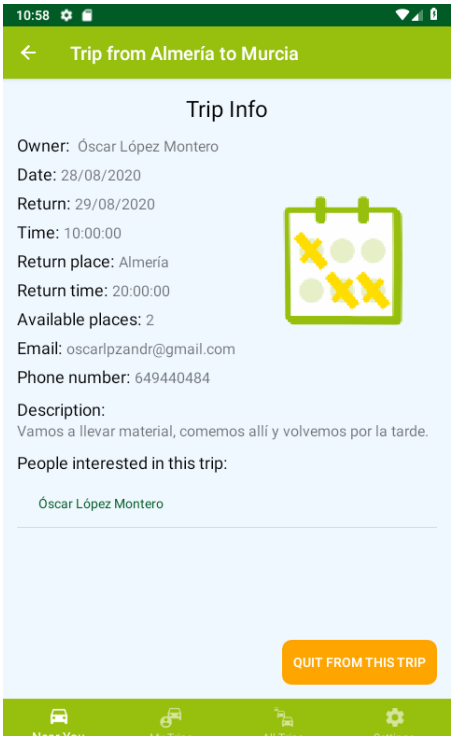

Figura 42. Vista de detalles de un viaje

4.4.2.8. Opciones y configuración

Vista en la que se muestra el menú de opciones de la aplicación.

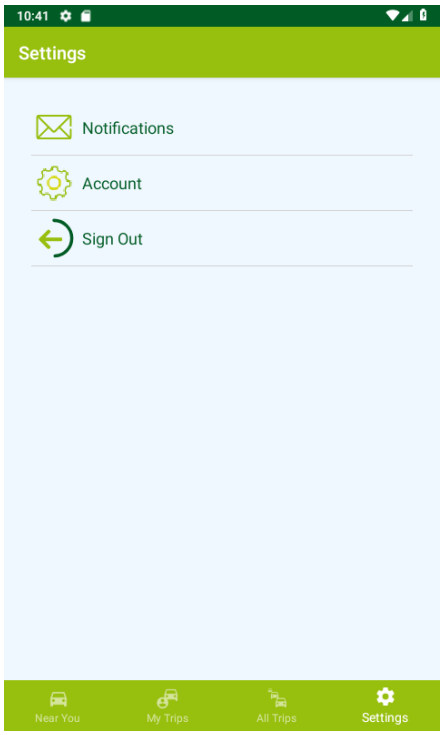

Figura 43. Vista de configuración

4.4.2.9. Editar configuración de usuario

Vista en la que se nos permite consultar y modificar los datos de usuario usados en la aplicación.

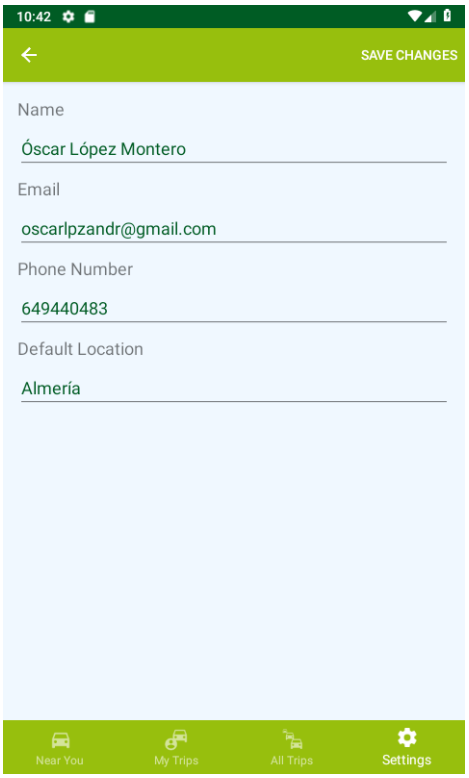

Figura 44. Vista de edición de datos de usuario

4.4.2.10. Lista de notificaciones

Vista en la que se muestra la lista de notificaciones activas que tiene el usuario.

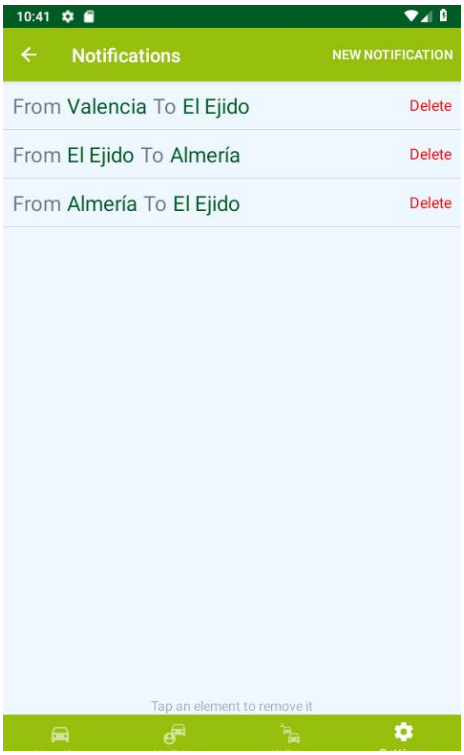

Figura 45. Vista de notificaciones

4.4.2.11. Nueva notificación

Vista encargada de añadir nuevas notificaciones al usuario.

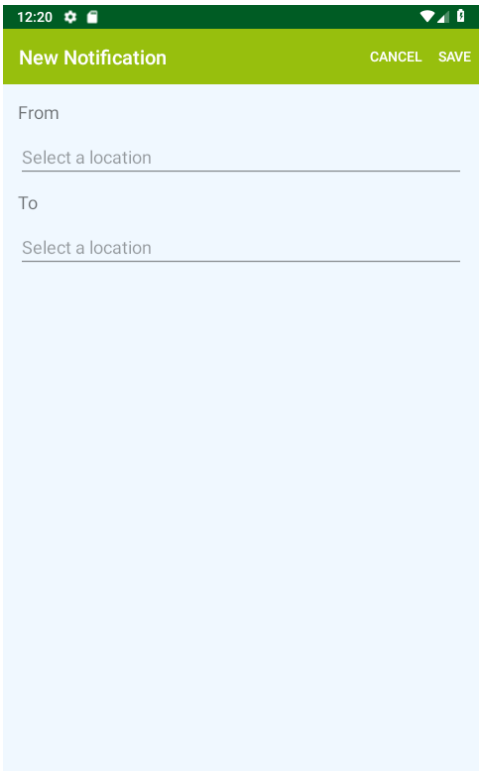

Figura 46. Vista de registro de notificaciones

4.4.3. Implementacion

En este apartado vamos a comentar los distintos detalles de implementación de la aplicación.

4.4.3.1. Intercambio de solicitudes HTTP

Para el envío y recepción de solicitudes HTTP entre nuestra aplicación y nuestro servidor web se ha utilizado una implementación de HttpClient.

HttpClient nos permite instanciar una clase para enviar y recibir solicitudes de un recurso identificado en una URI. Esta está diseñada para una vez creada la estancia, sea utilizada durante toda la sesión, con el fin de almacenar y manejar tanto las cabeceras como las Cookies de la sesión.

Para utilizar una única estancia de HttpClient y con el fin de que nuestro código sea lo más legible posible, hemos creado una clase que se encarga de inicializar sus parámetros, *RestService.cs.*

RestService se encarga de inicializar, tanto la estancia de HttpClient como del controlador de mensajes *HttpClientHandler* de la misma.

```
3 referencias
class RestService
{
    public HttpClient client;
    1 referencia
    public RestService()
    {
        client = new HttpClient(new HttpClientHandler())
        {
            Timeout = TimeSpan.FromSeconds(5)
        };
        client.DefaultRequestHeaders
              .Accept
              .Add(new MediaTypeWithQualityHeaderValue("application/json"));
    }
```

Figura 47. Implementación de HttpClient

Esta estancia es utilizada por la clase contenedora de todos los métodos encargados de enviar solicitudes a la API, *GeneralDataStore*.

Un ejemplo de POST a nuestra API web con *HttpClient* sería el siguiente:

```
4 referencias
public async Task<Trip> AddTripAsync(Trip item, HttpClient client)
{
    var content = new StringContent(
              JsonConvert.SerializeObject(new DBTrip(item)), Encoding.UTF8,
                                  "application/json");
    try
    {
        var response = await client.PostAsync(uri + "api/Trips", content).ConfigureAwait(false);
        if (response.IsSuccessStatusCode)
        {
            string responsecontent = await response.Content.ReadAsStringAsync();
            item = JsonConvert.DeserializeObject<Trip>(responsecontent);
        }
    }
    catch (Exception e)
    {
        Debug.WriteLine(e.StackTrace);
    }
    return item;

}
```

Figura 48. Método de publicación de nuevo viaje a API web

En él, realizamos los siguientes pasos:

1. Llamamos al método con un viaje (*Trip*), la estancia de *HttpClient* que se utiliza es la inicializada en la base de datos general.
2. Envolvemos el objeto en un JSON que contendrá las propiedades del viaje.
3. Llamamos al método *PostAsync* de *HttpClient*, indicando la dirección de la API y el contenido.
4. Esperamos la respuesta y devolvemos el ítem al modelo, esto es importante debido a que la API es la encargada de asignar el ID del viaje.

4.4.3.2. Uso de animaciones de Lottie

Lottie es formato de animaciones basadas en JSON que habilita a los diseñadores a lanzar sus animaciones en cualquier plataforma. Las animaciones de Lottie son escalables, open-*source*, y ligeras [14].

La inclusión de animaciones de Lottie en el proyecto es bastante sencillo, únicamente necesitamos importar el paquete de Lottie en nuestra aplicación, y empezar a usarlas haciendo uso de la etiqueta <*Lottie:AnimationView*> en nuestro xaml.

Las animaciones de Lottie deben ser almacenadas en la carpeta de *Assets* en el caso de Android y *Resources* en el caso de iOS.

Hacemos uso de Lottie tanto en las transiciones como para mostrar iconos animados en algunas ventanas, un ejemplo de uso de Lottie en una transición es en el Login:

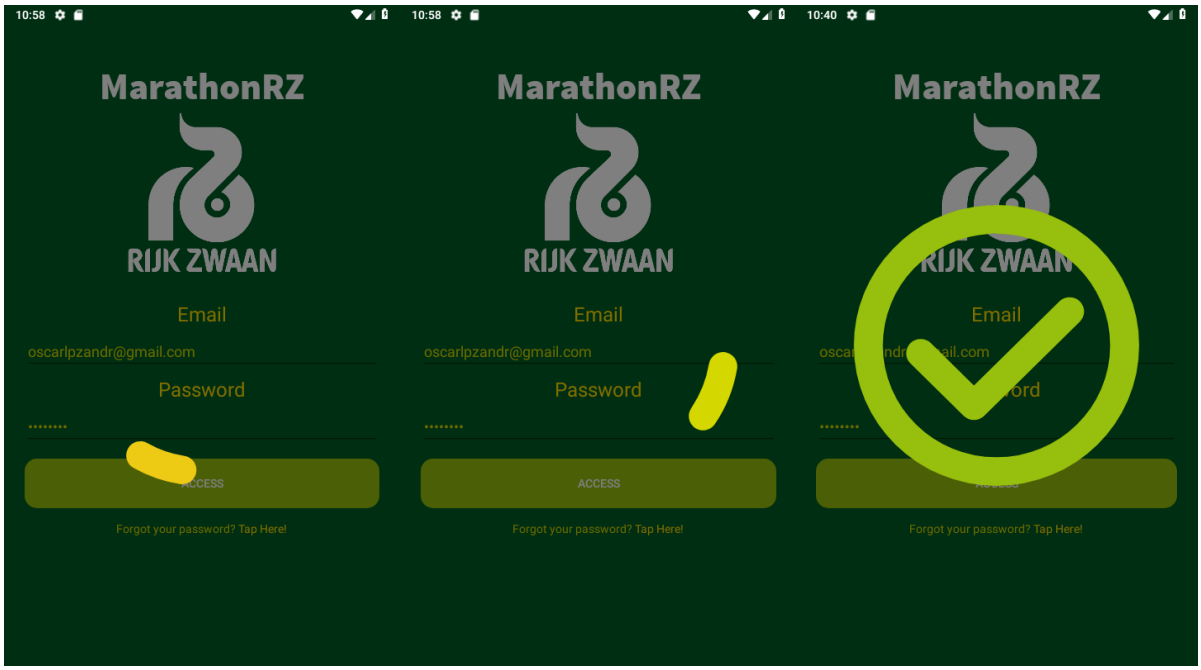

Figura 49. Animación de inicio de sesión

Esta animación es mostrada como un modal, con el fin de poder la ventana de fondo mientras está ocurriendo.

```
<ContentPage.Content>
    <lottie:AnimationView
x:Name="LottieView"
Animation="login.json"
Loop="False"
AutoPlay="True"
OnFinish="LottieView_OnFinish"/>
</ContentPage.Content>
```

Figura 50. Contenido de animación de Login

Otro ejemplo de uso de Lottie en nuestra aplicación es el caso de los iconos animados. En la vista de detalles, usamos un calendario animado para dar la impresión de un diseño más dinámico.

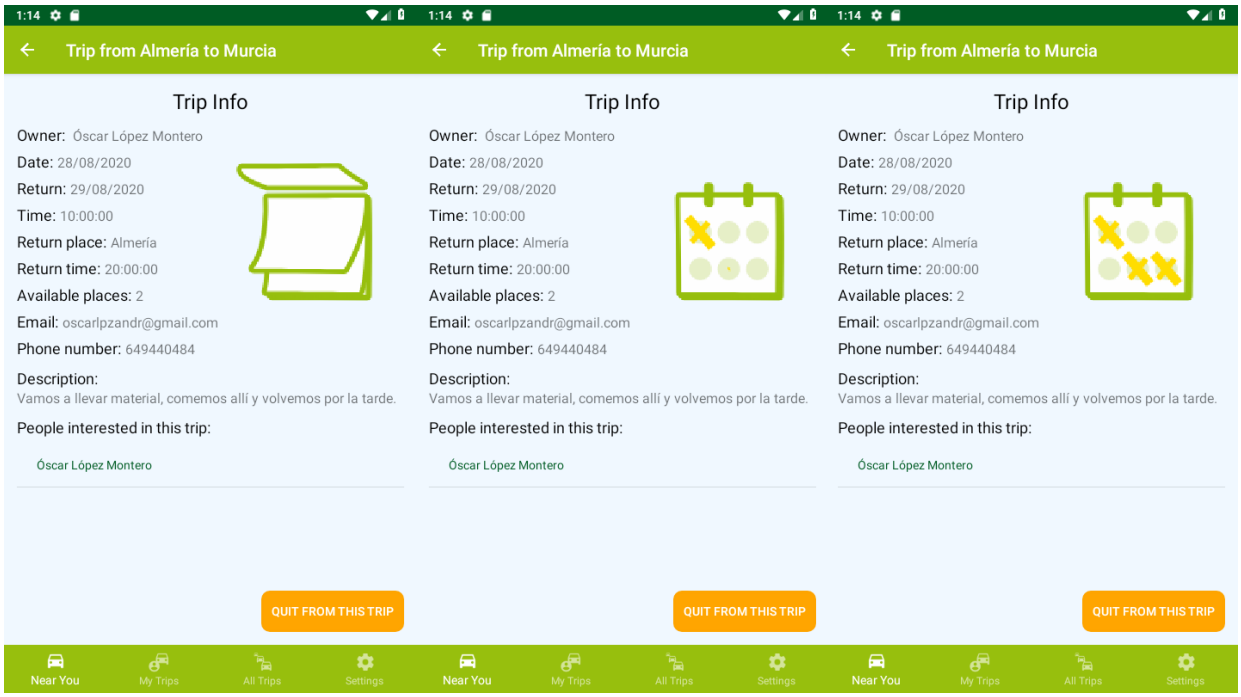

Figura 51. Animación de calendario en ventana de detalles

En este caso, en lugar de insertar la animación en una página nueva la cual es presentada como modal, introducimos la animación directamente en la posición de nuestra vista que queramos.

```
        <StackLayout Orientation="Horizontal" Spacing="5">
            <Label Text="Phone number:" FontSize="Medium" TextColor="Black"/>
            <Label Text="{Binding Owner.PhoneNumber}" VerticalOptions="End" FontSize="Body"></Label>
        </StackLayout>
    </StackLayout>
    <lottie:AnimationView
        x:Name="LottieView"
        Animation="calendar.json"
        Loop="False"
        AutoPlay="True"
        VerticalOptions="FillAndExpand"
        HorizontalOptions="FillAndExpand"
        Scale="1.5"
       HeightRequest="150"
       WidthRequest="150"/>
</StackLayout>
<StackLayout Orientation="Vertical" Spacing="0">
    <Label x:Name="TripDesc" Text="Description:" FontSize="Medium" TextColor="Black" />
    <Label Text="{Binding Trip.Description}" VerticalOptions="End" FontSize="Body" HorizontalOptions="Fill"></Label>
</StackLayout>
<StackLayout Orientation="Vertical">
```

Figura 52. Animación de calendario en el código

4.4.3.3. Uso de Xamarin.Essentials en la aplicación

Xamarin Essentials nos proporciona una API para el acceso a herramientas únicas del sistema operativo tales como acceso a la geolocalización, acelerómetro, batería, brújula, etc.

En nuestro caso, usamos Xamarin.Essentials para almacenar los datos de acceso del usuario en el almacenamiento seguro del teléfono con el fin de lograr un inicio de sesión automático. Esto es importante ya que no queremos que los trabajadores tengan que iniciar sesión cada vez que quieren registrar o revisar los viajes.

Esto lo logramos mediante la siguiente lógica:

1. Accedemos a la vista de inicio de sesión, desde ella se hace una llamada al método del modelo de vista que comprueba si existen datos de inicio de sesión en el *SecureStorage.*

```
1 referencia
private async void Init()
{
    if (await viewModel.SkipLogin())
    {
        await Navigation.PushPopupAsync(new LoginAnimation());
    }
}
```

Figura 53. Método Init de la vista del Login

2. El *ViewModel* comprueba si existe un usuario autentificado en ese móvil haciendo una llamada al almacenamiento seguro del dispositivo.

```
public async Task<bool> SkipLogin()
{
    var isLogged = await Xamarin.Essentials.SecureStorage.GetAsync("isLogged");
    if (isLogged == "1")
    {
        var user = await DataStore.Login(await Xamarin.Essentials.SecureStorage.GetAsync("Username"),
            await Xamarin.Essentials.SecureStorage.GetAsync("Password"));
        if (user != null)
        {
            return true;
        }
    }
    return false;
}
```

Figura 54. Método SkipLogin del modelo de vista

3. En caso afirmativo, se devuelve el usuario y se le autentifica automáticamente.

Los datos guardados en el almacenamiento seguro son eliminados al cerrar sesión en la cuenta desde la aplicación.

4.4.3.4. IMessage

Con el fin de aumentar el *feedback* que recibe el usuario, se ha implementado una clase que hará uso de la funcionalidad de iOS y Android para mostrar alertas tipo *Toast.*

Esta implementación es una de las pequeñas funcionalidades de nuestro código que necesita de ser programada por partes separadas debido a que funcionan de manera distinta, necesitando una implementación para iOS y otra para Android.

Ambas van a hacer uso de una interfaz común, con el fin de que, aun siendo dos implementaciones distintas, podamos incluirla en nuestro código como si se tratara de una sola. La diferencia entre *ShortAlert* y *LongAlert* es el espacio de tiempo durante el que se muestran.

```
23 referencias
public interface IMessage
{
    3 referencias
    void LongAlert(string message);
    22 referencias
    void ShortAlert(string message);
}
```

Figura 55. Interfaz de IMessage

La implementación de Android es la siguiente:

```
namespace ViajesRZ.Droid.Services
{
    1 referencia
    public class MessageAndroid : IMessage
    {
        3 referencias
        public void LongAlert(string message)
        {
            Toast.MakeText(Application.Context, message, ToastLength.Long).Show();
        }

        22 referencias
        public void ShortAlert(string message)
        {
            Toast.MakeText(Application.Context, message, ToastLength.Short).Show();
        }
    }
}
```

Figura 56. Implementación de Toast Text en Android

Nuestra implementación en iOS:

```
1 referencia
public class MessageIOS : IMessage
{
    const double LONG_DELAY = 3.5;
    const double SHORT_DELAY = 2.0;

    NSTimer alertDelay;
    UIAlertController alert;

    3 referencias
    public void LongAlert(string message)
    {
        ShowAlert(message, LONG_DELAY);
    }
    22 referencias
    public void ShortAlert(string message)
    {
        ShowAlert(message, SHORT_DELAY);
    }

    2 referencias
    void ShowAlert(string message, double seconds)
    {
        alertDelay = NSTimer.CreateScheduledTimer(seconds, (obj) =>
        {
            DismissMessage();
        });
        alert = UIAlertController.Create(null, message, UIAlertControllerStyle.Alert);
        UIApplication.SharedApplication.KeyWindow.RootViewController.PresentViewController(alert, true, null);
    }

    1 referencia
    void DismissMessage()
    {
        if (alert != null)
        {
            alert.DismissViewController(true, null);
        }
        if (alertDelay != null)
        {
            alertDelay.Dispose();
        }
    }
}
```

Figura 57. Implementación de Toast Text en iOS

4.4.3.5. Recogida, registro y visualización de datos

Nuestra está formada por dos funcionalidades principales, la recogida de datos mediante formularios con el fin de registrarlos en la API y la extracción de datos de la API con el fin de mostrarlos.

Para la primera, nuestra aplicación móvil tiene que realizar varios pasos:

1. Muestra en la vista una serie de formularios que el usuario rellena y envía.
2. La vista pasa los datos al *ViewModel* y este se encarga de llamar al servicio REST.
3. El servicio, mediante una estancia de HttpClient, realiza un POST a la API.

Vamos a ver un ejemplo de registro de datos de nuestra aplicación:

Primero, recogemos los datos desde la vista *NewTripPage*. Recordamos que nuestras vistas están compuestas por dos clases parciales, una encargada de la interfaz en xaml y otra encargada de controlar la lógica detrás de esa vista en C#.

En la vista, utilizamos la etiqueta Binding para referirnos a un atributo de nuestro *BindingContext*, este es declarado en la parte programada en C# de nuestra vista y suele apuntar a nuestro ViewModel.

```
<ContentPage.Content>
    <ScrollView x:Name="ScrollView">
        <StackLayout Spacing="10" Padding="15">
            <Label Text="From" FontSize="Medium"  />
            <Picker Title="Select a location" ItemsSource="{Binding DestinationList}" SelectedItem="{Binding Trip.From_Destination, Mode=TwoWay}" />
            <Label Text="To" FontSize="Medium" />
            <Picker Title="Select a location" ItemsSource="{Binding DestinationList}" SelectedItem="{Binding Trip.To_Destination, Mode=TwoWay}"/>
            <Label Text="Return place" FontSize="Medium" />
            <Picker Title="Select a location" ItemsSource="{Binding DestinationList}" SelectedItem="{Binding Trip.Return_Destination, Mode=TwoWay}"/>
            <Label Text="Date" FontSize="Medium" />
            <DatePicker MinimumDate="{Binding Today}"  Date="{Binding Trip.Date}"/>
            <Label Text="Return Date" FontSize="Medium" />
            <DatePicker MinimumDate="{Binding Today}"  Date="{Binding Trip.Return_Date}"/>
            <StackLayout Spacing="20" Padding="0" HorizontalOptions="Start" Orientation="Horizontal">
                <Label Text="Optional Info" FontSize="Medium" HorizontalOptions="Start" />
                <Switch Scale="1.3" HorizontalOptions="End" IsToggled="{Binding AdditionalInfo,Mode=TwoWay}" Toggled="Scroll"/>
            </StackLayout>
            <StackLayout Spacing="20" Padding="0" HorizontalOptions="FillAndExpand" IsVisible="{Binding AdditionalInfo, Mode=TwoWay}">
                <Label Text="Departure Time" FontSize="Medium" />
                <TimePicker Time="{Binding DepartureTime}" ></TimePicker>
                <Label Text="Return Time" FontSize="Medium" />
                <TimePicker Time="{Binding ReturnTime}"></TimePicker>
                <Label Text="Available places" FontSize="Medium" />
                <Entry Text="{Binding Trip.AvailablePlaces}"/>
                <Label Text="Observations" FontSize="Medium" />
                <Editor Text="{Binding Trip.Description,Mode=OneWayToSource}" d:Text="Item description" FontSize="Small" Margin="0" Placeholder="Observations (Optional)"/>
            </StackLayout>
        </StackLayout>
    </ScrollView>
</ContentPage.Content>
```

Figura 58. Contenido de la página de registro de viaje

Declaramos como contexto enlazado nuestro modelo de vista, y llamamos al registro de viajes del mismo, pasándole una estancia del viaje contenido en los formularios.

```
public partial class NewTripPage : ContentPage, INotifyPropertyChanged
{
    NewTripViewModel viewmodel;

    3 referencias
    public NewTripPage()
    {

        InitializeComponent();
        BindingContext = viewmodel = new NewTripViewModel();
        viewmodel.AdditionalInfo = false;
    }


    0 referencias
    async void Save_Clicked(object sender, EventArgs e)
    {
        if(!IsOnValidState()) return;
        await viewmodel.SaveTrip();
        await Navigation.PushModalAsync(new NewTripAnimation());
        DependencyService.Get<IMessage>().ShortAlert("The trip was created successfully");
    }
```

Figura 59. Llamada al ViewModel en NewTripPage

Desde el *ViewModel,* preparamos el objeto para realizar la llamada a la API e insertarlo.

```
public NewTripViewModel()
{
    User = DataStore.GetCurrentUser();
    Today = DateTime.Today;
    Trip = new Trip()
    {
        From_Destination = User.DefaultLocation,
        Return_Destination = User.DefaultLocation,
        Date = DateTime.Today,
        Return_Date = DateTime.Today,
        Trip_OwnerEmail = User.Email,
        Trip_OwnerName = User.Name,

};
}
1 referencia
public async Task SaveTrip()
{
    Trip.DepartureTime = DepartureTime.ToString();
    Trip.ReturnTime = ReturnTime.ToString();
    if(Trip.Description == null)
    {
        Trip.Description = "This trip doesn't include any description";
    }
    if (Trip.ReturnTime == "00:00:00" && Trip.DepartureTime == "00:00:00" && Trip.ReturnTime==Trip.DepartureTime) {
        Trip.ReturnTime = "Not Specified";
        Trip.DepartureTime = "Not Specified";
    }
    User.TripList.Add(Trip);
    MessagingCenter.Send(this, "AddTrip", Trip);
    return;
}
```

Figura 60. ViewModel de registro de viajes

Este realiza una llamada a MessagingCenter y suscribe el viaje como mensaje para ser recogido desde la vista destino, esto se realiza para evitar un mal uso de la aplicación y el registro de viajes duplicados en el hipotético caso de que se ejecuten varias estancias de la vista, además de reducir el acoplamiento entre clases.

```
MessagingCenter.Subscribe<NewTripViewModel, Trip>(this, "AddTrip", async (obj, item) =>
{
    var newItem = item as Trip;
    var trip = await DataStore.AddTripAsync(newItem);
    Trips.Add(trip);
});
```

Figura 61. Suscripción del mensaje y llamada a RestService

Finalmente, el RestService se encarga de realizar la solicitud POST a nuestra API (Ver Figura 45).

Para mostrar datos existentes de la API, se realiza lo mismo, pero en el orden contrario, primero traemos los datos de la API con un método HTTP GET desde nuestra clase *RestService*.

Una vez tenemos los datos, estos son cargados por el *ViewModel* correspondiente en uno de sus atributos. Una vez tengamos los datos a mostrar en nuestro modelo de vista nuestra vista será la encargada de acceder a ellos mediante su *BindingContext*.

4.4.3.6. Notificación de cambio de propiedad

Para aportar *feedback* a nuestros usuarios es importante actualizar lo que se muestra en la vista cada vez que ocurra una modificación, para ello, es necesario el uso de la notificación de cambio de propiedad *OnPropertyChanged*.

OnPropertyChanged nos permite que las propiedades de destino de enlace reflejen automáticamente los cambios dinámicos del origen de enlace.

Para implementarlo de la manera más simple, vamos a incluir el método en nuestro *BaseViewModel,* ya que el resto de modelos de vista heredan del mismo y podemos reutilizarlo en lugar de tener que realizar una implementación cada vez que queramos usarlo. En caso de necesitar usarlo fuera de los modelos de vista, se debería realizar una implementación aparte.

```
#region INotifyPropertyChanged
public event PropertyChangedEventHandler PropertyChanged;
8 referencias
protected void OnPropertyChanged([CallerMemberName] string propertyName = "")
{
    var changed = PropertyChanged;
    if (changed == null)
        return;

    changed.Invoke(this, new PropertyChangedEventArgs(propertyName));
}
#endregion
```

Figura 62. Implementación de OnPropertyChanged

Ahora, debemos modificar nuestra implementación de la clase que usa *OnPropertyChanged* añadiendo un atributo que será el mostrado, el cual notifica los cambios al atributo encargado de guardar los datos.

```
public ObservableCollection<Trip> _trips;
13 referencias
public ObservableCollection<Trip> Trips
{
    get
    {
        return _trips;
    }
    set
    {
        _trips = value;
        OnPropertyChanged("Trips");
    }
}
```

Figura 63. Uso de OnPropertyChanged en TripsViewModel

4.4.4. Despliegue

Para el despliegue de la aplicación multiplataforma a los dispositivos Android, se va a utilizar la funcionalidad de despliegue por correo electrónico de Visual Studio. Esto es debido a que no necesitamos que nadie ajeno a Rijk Zwaan instale nuestra aplicación.

Para realizar el despliegue por correo electrónico, primero generamos un paquete del ejecutable *.apk* para la instalación de la app en Android:

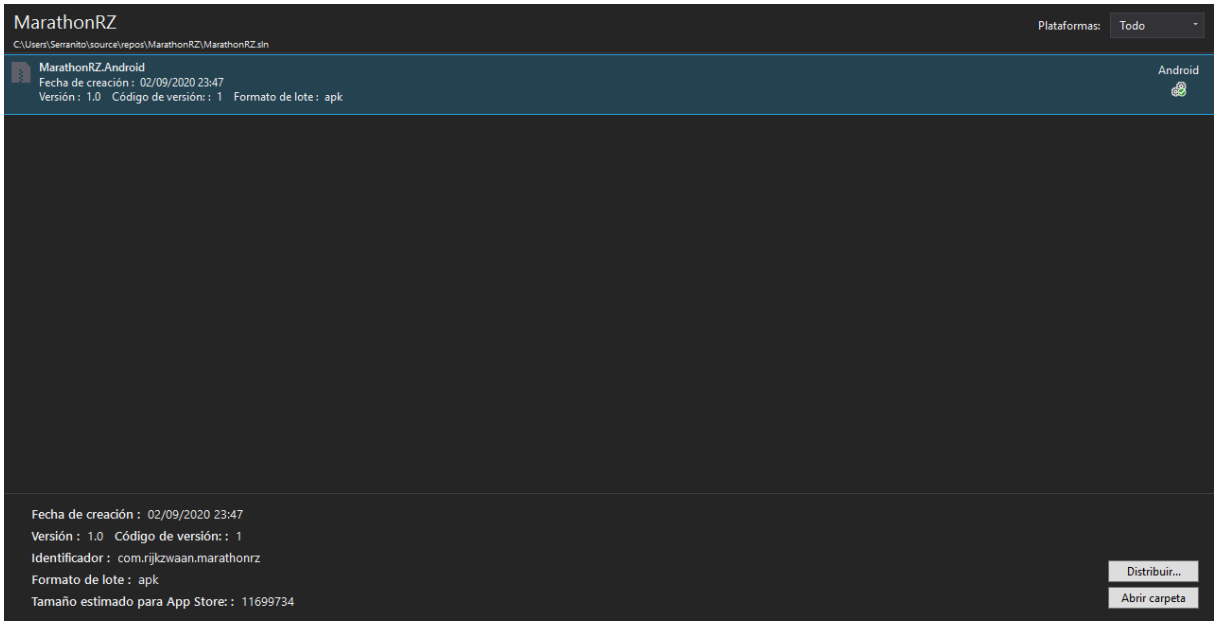

Figura 64. Paquete de instalación del .apk en Android

Una vez tenemos el paquete creado, solamente queda distribuirlo a nuestros clientes.

Como en este caso vamos a hacer la distribución por correo, accedemos al apartado de distribuir y seleccionamos la opción de Correo Electrónico.

Tras esto, los usuarios únicamente deberán abrir el fichero adjunto al correo para que este se instale automáticamente en el dispositivo.

4.5. Aplicación Web y API

4.5.1. Patrón de diseño

Para el diseño de la aplicación web, se han usado dos patrones, el modelo basado en páginas y la inyección de dependencias.

4.5.1.1. Modelo basado en páginas

El modelo basado en páginas busca facilitar la programación de escenarios basados en páginas buscando que sea más productiva que con controladores y vistas [15].

La estructura de este modelo es similar a la de un MVVM con la diferencia de que aquí no usamos ningún controlador, es decir, cada página tiene una vista y una clase heredada de la página que implementa la funcionalidad y la lógica de la misma (Como si se tratara de un modelo de vista).

Gracias a la etiqueta *@page*, se transforman los archivos en acciones de MVC, por tanto, no es necesario que las solicitudes de nuestra aplicación web pasen por un controlador.

4.5.1.2. Inyección de dependencias

Los controladores de nuestra aplicación solicitan dependencias a través de los constructores, facilitando las pruebas y el mantenimiento de la misma. Estos servicios son agregados como parámetros a los constructores de las clases en las que se insertan y son resueltos en tiempo de ejecución por el contenedor de servicios.

Antes de usar nuestras dependencias, debemos registrarlas para que nuestra aplicación esté al tanto de la existencia de las mismas. Estas se deben registrar como servicios en el método *ConfigureServices*. Un ejemplo de servicio que hemos registrado en nuestra aplicación para su uso como dependencia es *EmailService*, nuestro servicio encargado de enviar notificaciones por correo al usuario.

Para registrar un servicio, ASP.NET Core ofrece tres posibles soluciones:

AddSingleton, *AddTransient* y *AddScoped*.

AddSingleton reutiliza el servicio cada vez que es llamado, ahorrando memoria y tiempo. Esta es la mejor opción cuando no necesitamos nuevas estancias de nuestro servicio. Esta solución, aunque eficiente, es bastante arriesgada y suele ser más seguro el uso de AddScoped.

AddTransient siempre crea una nueva estancia cada vez que se requiere el servicio como dependencia. Esta opción es muy costosa y solo debe ser usada en casos en los que sea muy necesario inicializar una nueva estancia.

AddScoped es la opción más interesante, en ella se reutiliza la estancia del servicio mientras nos encontremos en la misma solicitud HTTP, una vez se realice una nueva solicitud, se creará otra estancia para ella.

En nuestro caso, vamos a hacer uso de *AddScoped.* Para registrar nuestro servicio llamamos al método y especificamos la clase e interfaz que queremos añadir a la colección.

```
services.AddScoped<IEmailService, EmailService>();
```

Figura 65. Registro de servicio para uso como dependencia

Para usar este servicio como dependencia, solamente tenemos que incluirlo como parámetro en el constructor de nuestra clase. En este caso, vamos a hacer uso del envío de notificaciones cuando se registre un nuevo viaje, así que vamos a necesitar este servicio en el controlador de viajes de la API.

4.5.2. Diseño de la interfaz

Para nuestra web, nuestra principal prioridad ha sido que nuestra interfaz sea *responsive*, es decir, que sus componentes se ajusten al tamaño de cada pantalla y/o dispositivo con

```
public class TripsController : ControllerBase
{
    private readonly SharingCarRZWebContext _context;
    private readonly UserManager<User> _userManager;
    private readonly IEmailService _emailService;

    0 referencias
    public TripsController(SharingCarRZWebContext context, UserManager<User> userManager, IEmailService emailService)
    {
        _context = context;
        _userManager = userManager;
        _emailService = emailService;
    }
```

Figura 66. Inyección de dependencias en TripsController

el fin de que sea visible y usable desde cualquier dispositivo de la empresa, incluso desde móvil en el caso de no tener la aplicación instalada.

Para asegurarnos de que nuestra aplicación web se muestra de forma correcta, hemos definido varios estilos para las clases que utilizamos en nuestro CSS, de forma que el tamaño de los componentes varíe dependiendo del tamaño de la pantalla en la que se están mostrando.

Además, nuestra interfaz también variará dependiendo del rol que seamos, debido a que los administradores pueden acceder a opciones no accesibles para usuarios normales ni privilegiados.

Para mostrar la interfaz de la aplicación web, vamos a mostrarla desde la vista del administrador, que es la que contiene todas las opciones disponibles, el resto de vistas son similares a esta, pero más limitadas.

En las imágenes a continuación, se pueden ver todas las funcionalidades desde la vista de un administrador, en el caso de un usuario, simplemente se les oculta el enlace en la barra de navegación (Además de no tener posible acceso).

4.5.2.1. Login

Página desde la cual se accede a la aplicación. Tiene una opción de recordar al usuario para ampliar la duración de la cookie y otra de recuperación de contraseña en caso de perder las credenciales de acceso.

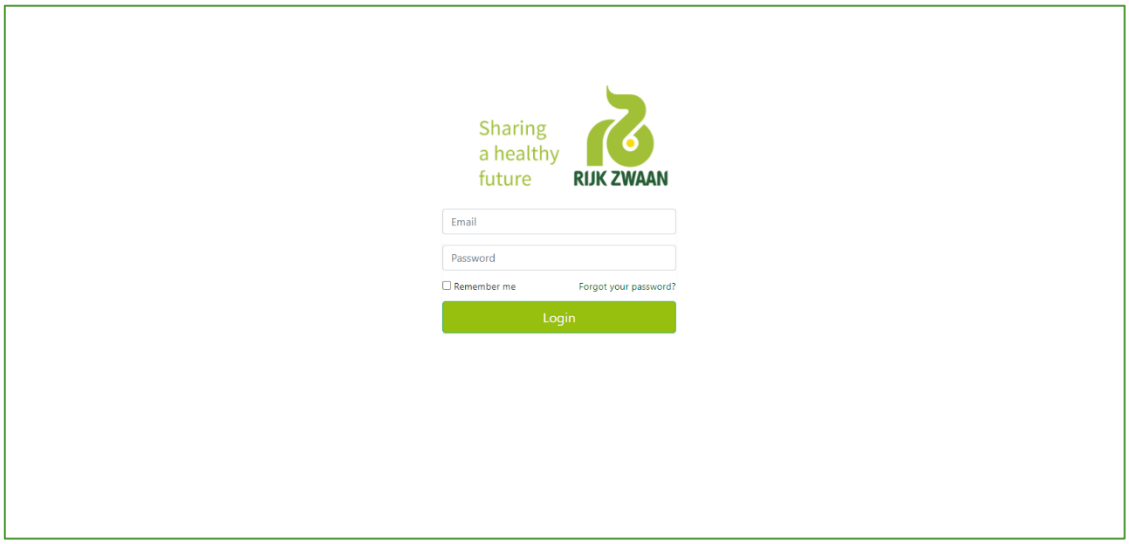

Figura 67. Autentificación en aplicación web

4.5.2.2. Recuperación de contraseña

Página que nos permite introducir nuestro correo electrónico para recuperar nuestra contraseña.

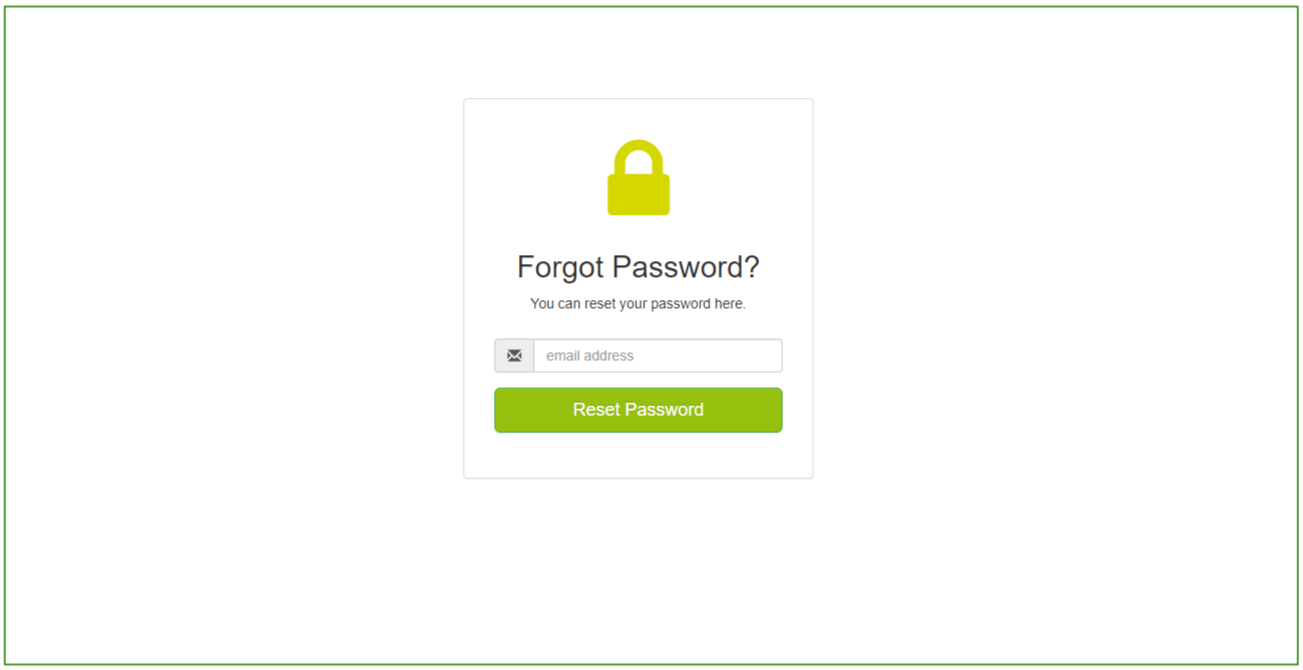

Figura 68. Recuperación de contraseña en aplicación web

4.5.2.3. Restablecer contraseña

Pantalla a la cual lleva el enlace de recuperación de contraseña enviado al correo.

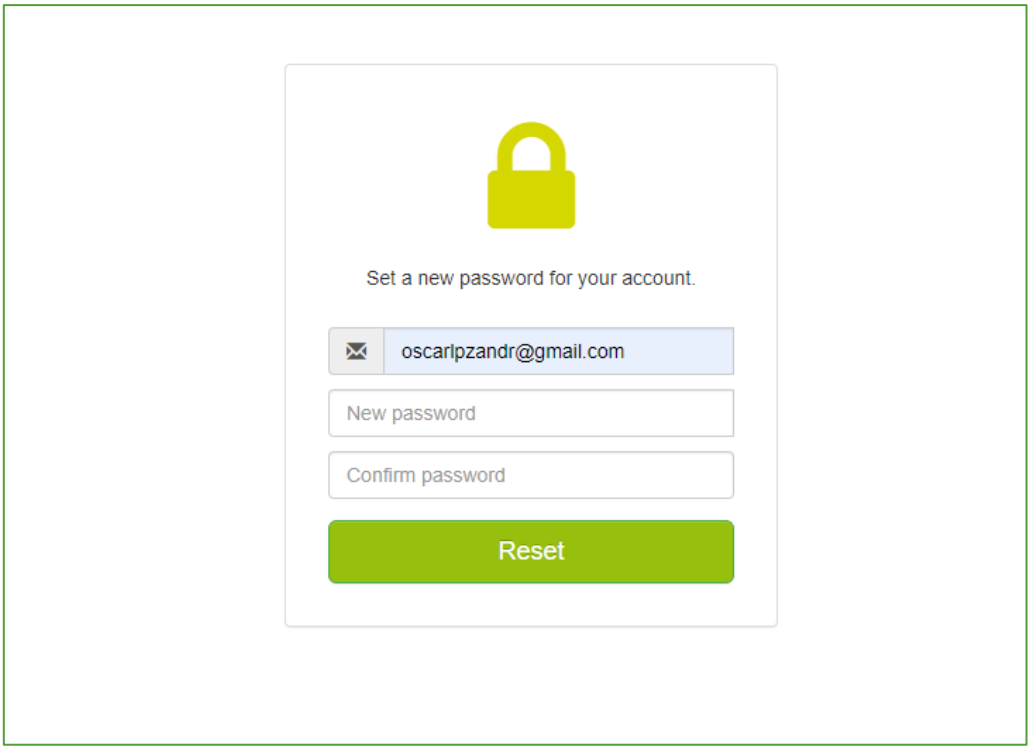

Figura 69. Restablecimiento de contraseña

4.5.2.4. Lista de viajes

Página en la que se muestran todos los viajes de la aplicación, se puede elegir entre mostrar los viajes pendientes o los ya finalizados, además de filtrarlos por cualquier parámetro.

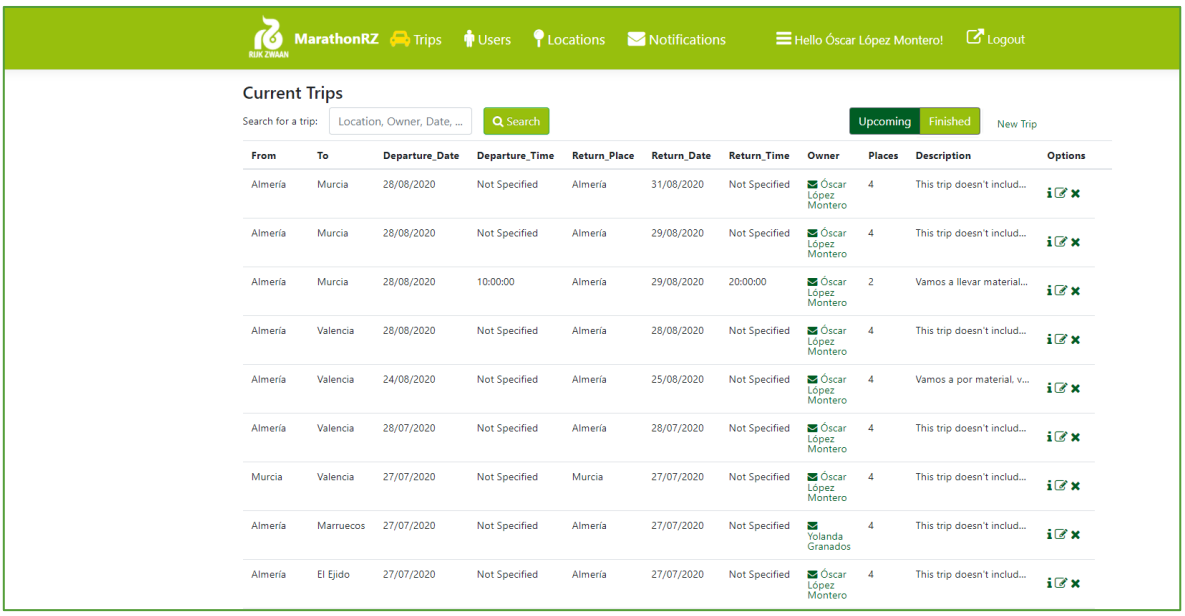

Figura 70. Lista de viajes en aplicación web

4.5.2.5. Lista de usuarios

Página en la que se muestran todos los usuarios de la aplicación. Esto es solo visible por un administrador.

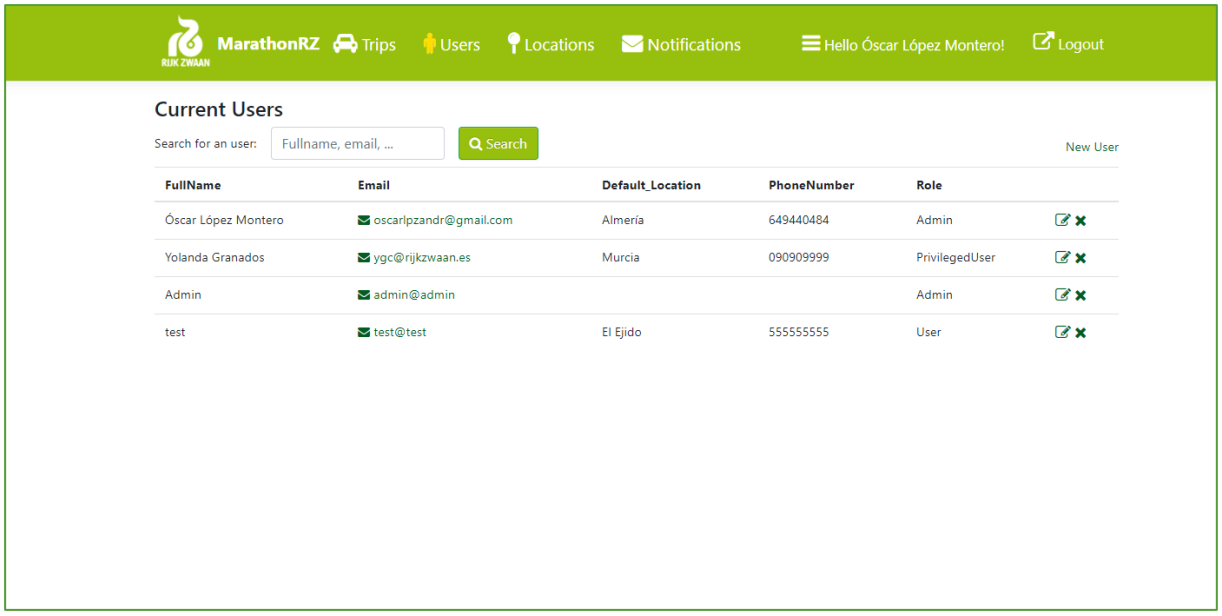

Figura 71. Lista de usuarios en aplicación web

4.5.2.6. Lista de localizaciones

Página en la que se muestran todas las localizaciones sobre las que se pueden registrar viajes en la aplicación. Esta ventana junto a la de usuarios son las dos únicas exclusivas del administrador.

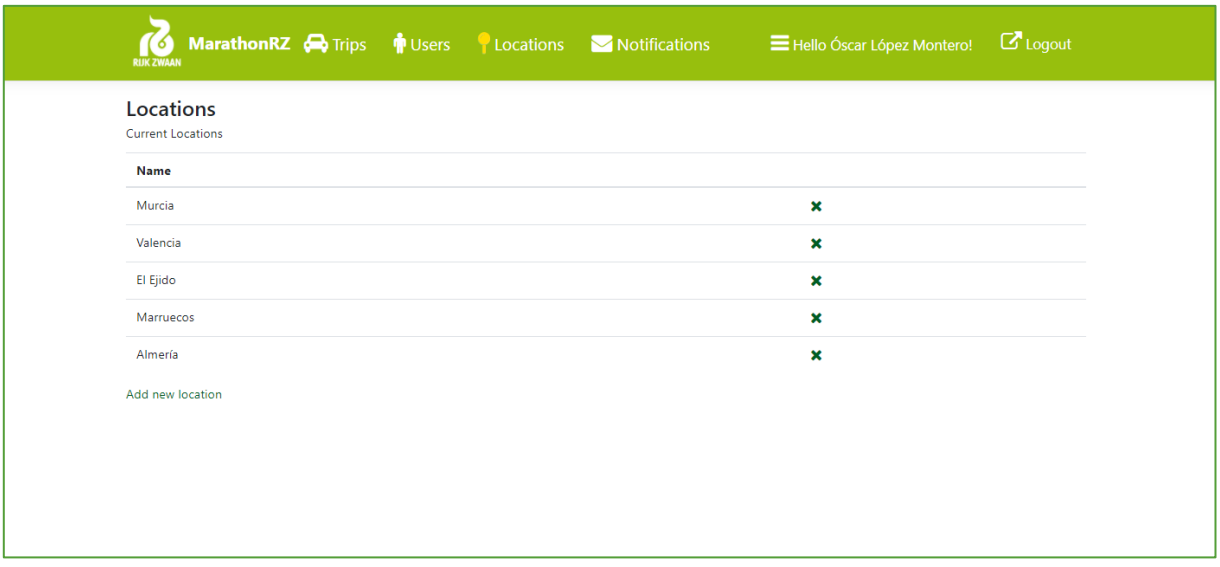

Figura 72. Lista de localizaciones en aplicación web

4.5.2.7. Lista de notificaciones

Página en la que podemos ver todas las notificaciones a las que estamos suscritos.

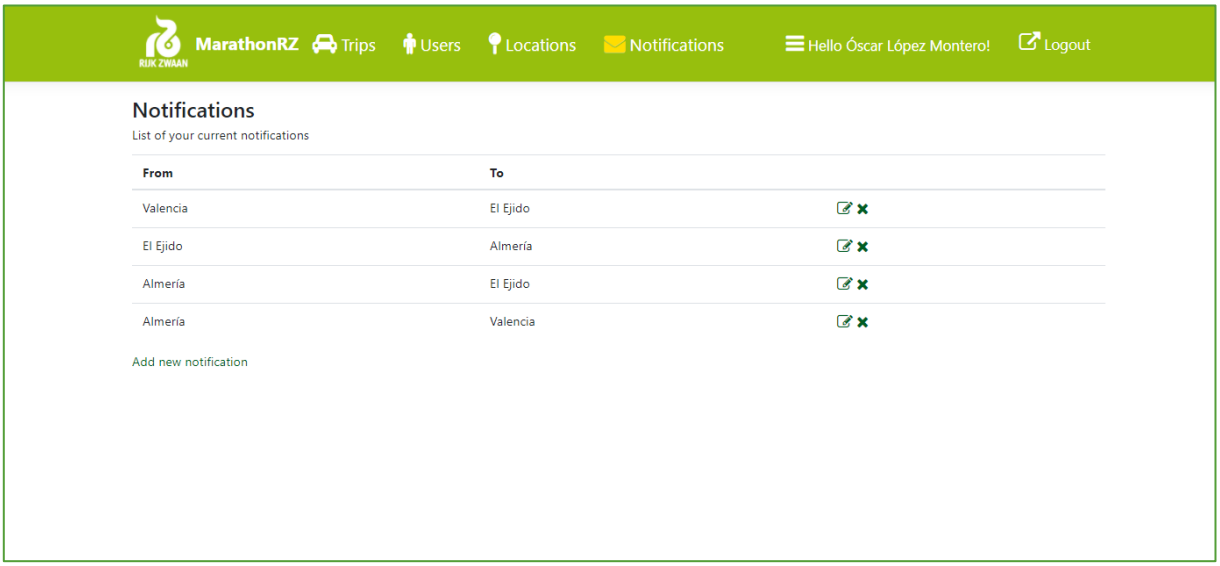

Figura 73. Lista de notificaciones en aplicación web

4.5.2.8. Configuración de usuario

Página que permite la modificación de los datos de usuario, así como opciones de configuración de la aplicación

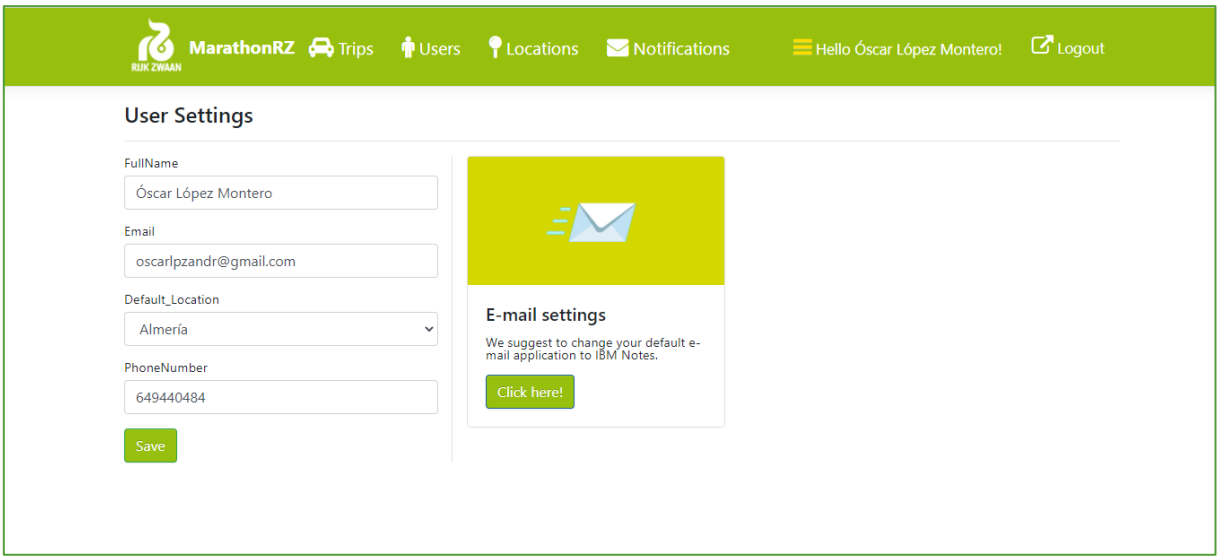

Figura 74. Página de configuración de usuario

4.5.2.9. CRUD

Para las CRUD (*Create, Read, Update and Delete*), formularios a través de los cuales vamos a actualizar nuestros valores de la base de datos, únicamente vamos a mostrar las de los viajes debido a que todas las CRUDS muestran una interfaz similar.

Añadir viaje

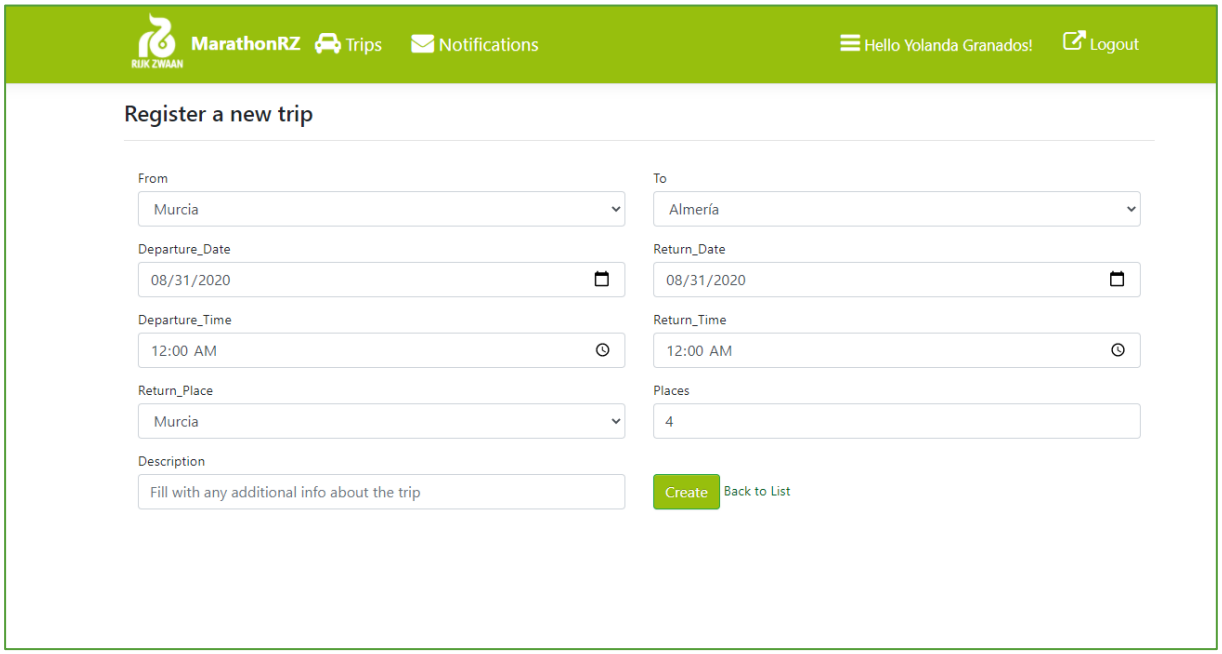

Figura 75. Registro de viajes en aplicación web

Editar viaje

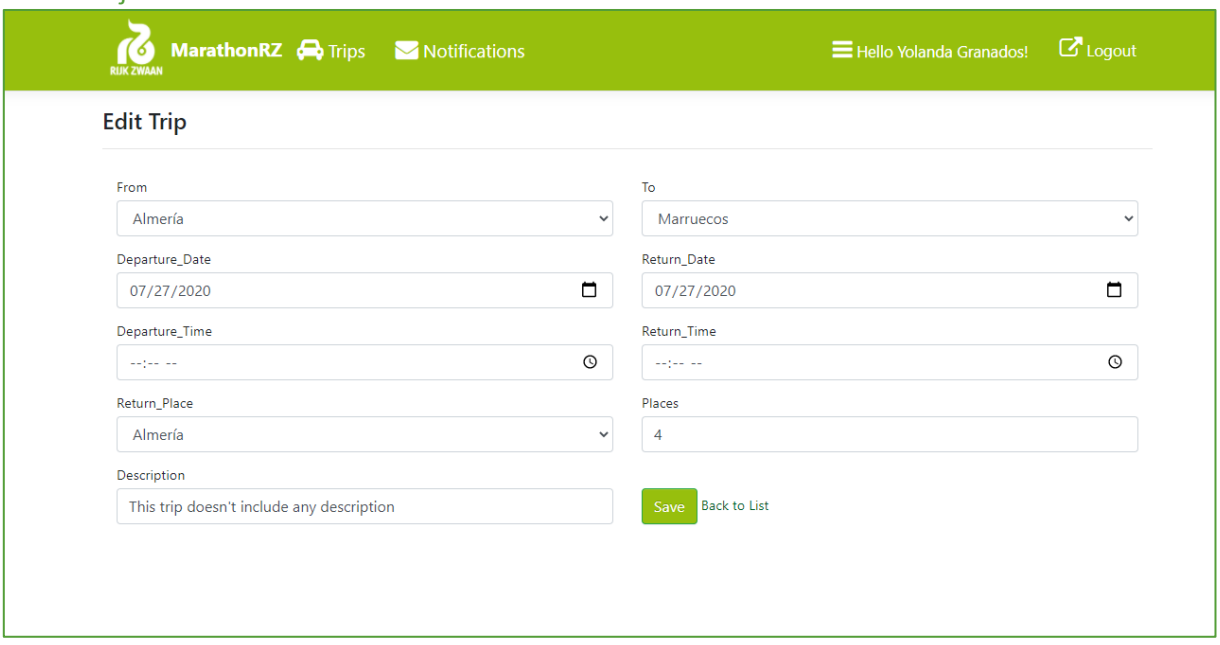

Figura 76. Edición de viajes en aplicación web

Eliminar viaje

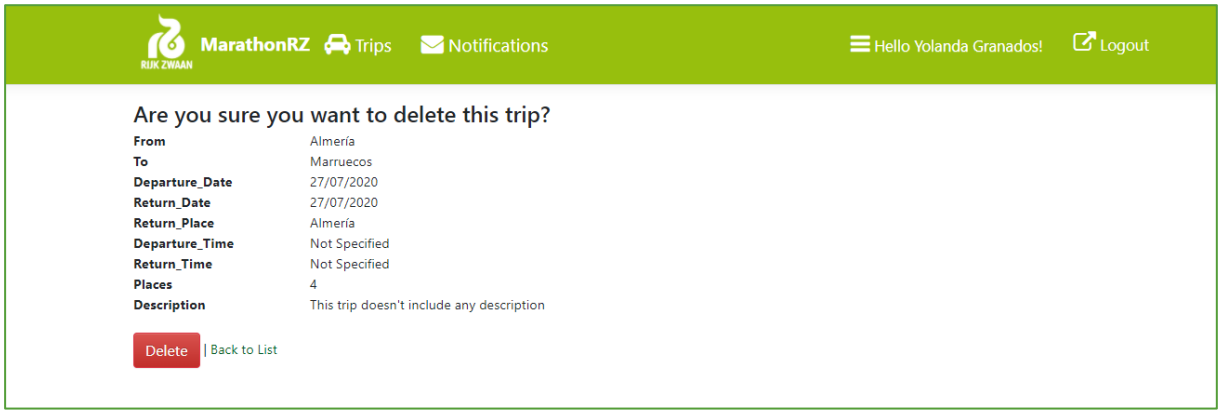

Figura 77. Eliminación de viaje en aplicación web

4.5.2.10. Notificación por correo

Las notificaciones por correo siguen un cuerpo predeterminado con el fin de mostrar un mensaje corporativo agradable y de confianza a los usuarios.

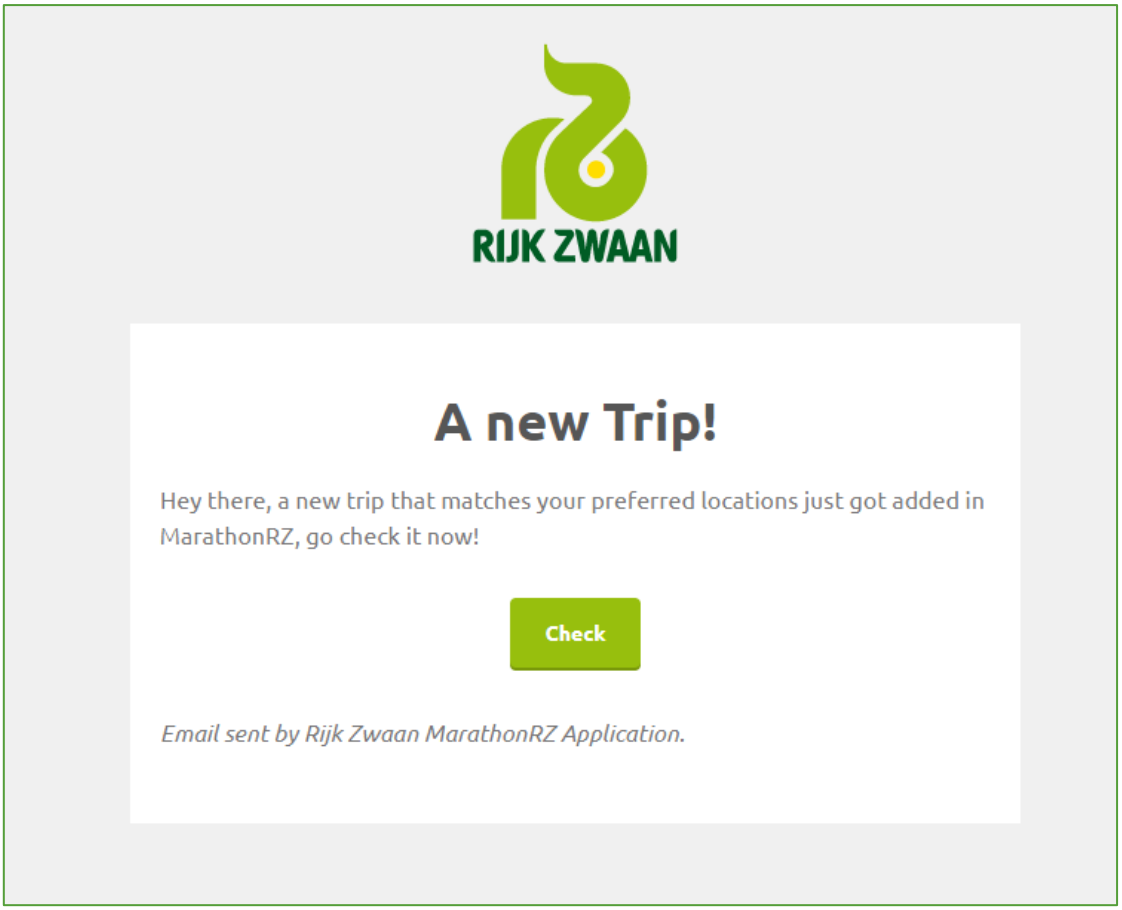

Figura 78. Notificación de nuevo viaje

4.5.3. Implementación

Ya que la web y la API forman uno en nuestro proyecto, vamos a hablar de la implementación, funcionalidad, características y detalles de calidad de ambas al mismo tiempo.

4.5.3.1. ASP.Net Core Identity

Identity es una API que implementa la funcionalidad de registro e inicio de sesión, administrando usuarios, contraseñas hasheadas, roles, tokens, etc. [16].

La principal ventaja de Identity es que nos aseguramos de que nuestro inicio de sesión sea lo más seguro posible al ser administrado por una API desarrollada por Microsoft. Además, esta permite ser editada de manera sencilla, permitiendo añadir, editar o eliminar atributos a nuestros usuarios.

Identity no guarda las contraseñas de los usuarios como tal. Cuando se registra un usuario en la aplicación, se le genera y guarda un hash como contraseña partiendo de la otorgada por el usuario, de esta manera nos aseguramos que nuestras contraseñas no sean reversibles.

Para comprobar si la contraseña es correcta, se genera un hash de la contraseña introducida en la autenticación y se compara con el almacenado en la base de datos, si son equivalentes se procede a la autenticación del usuario, en caso contrario.

4.5.3.2. Autenticación basada en Cookies

Para lograr la autenticación automatizada de los usuarios y que no tengan que introducir sus credenciales en la aplicación cada vez que deseen registrar un viaje vamos a usar la autenticación basada en cookies.

Las cookies son generadas para uso exclusivo de ese usuario cada vez que se autentifica y con una duración por defecto igual a la de la sesión del navegador, una vez se cierra este se elimina la cookie del navegador y de la web.

Además, se ofrece la posibilidad de guardar la cookie en el navegador y en la base de datos al marcar la opción de recordar al usuario en la autenticación. De esta manera el usuario no tiene que introducir constantemente sus credenciales y fomentamos el registro de viajes rápido y sencillo.

Estas cookies deben ir añadidas en las cabeceras de las solicitudes HTTP para probar que el usuario está autentificado en la aplicación, en el caso de no ser adjuntadas, se devolverá un error 401 al usuario.

Vamos a explicar el procedimiento por el que se generan y usan las cookies de autenticación en nuestra aplicación.

Antes que nada, debemos configurar nuestro proyecto para el uso de Cookies, añadiendo *UseAuthentication* a nuestro método *Configure* en nuestra clase *Startup*.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env, IServiceProvider serviceProvider)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }
    app.UseSession();

    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseRouting();
    app.UseCors(x => x
      .AllowAnyOrigin()
      .AllowAnyMethod()
      .AllowAnyHeader());

    app.UseAuthentication();
    app.UseAuthorization();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
        endpoints.MapControllers();
    });
    CreateRoles(serviceProvider);
}
```

Figura 79. Configuración de la aplicación web

UseAuthentication es el encargado de añadir el middleware de autenticación a nuestra aplicación que, por defecto, es mediante el uso de cookies, aunque se puede ajustar para el uso de *OAuth*, *JWT Bearer*, etc.

Tras esto, necesitamos especificar el comportamiento de nuestra aplicación en el caso de que la autenticación no sea exitosa.

```
services.ConfigureApplicationCookie(options =>
{
    options.Events.OnRedirectToAccessDenied =
    options.Events.OnRedirectToLogin = c =>
    {
        c.Response.StatusCode = StatusCodes.Status401Unauthorized;
        return Task.FromResult<object>(null);
    };
});
```

Figura 80. Configuración de Cookies

De esta manera nos aseguramos de que las solicitudes HTTP que lleguen a la API y no contengan la cookie con el token de autenticación sean rechazadas.

Las cookies son añadidas a las cabeceras de nuestro cliente HTTP tras autentificarse correctamente:

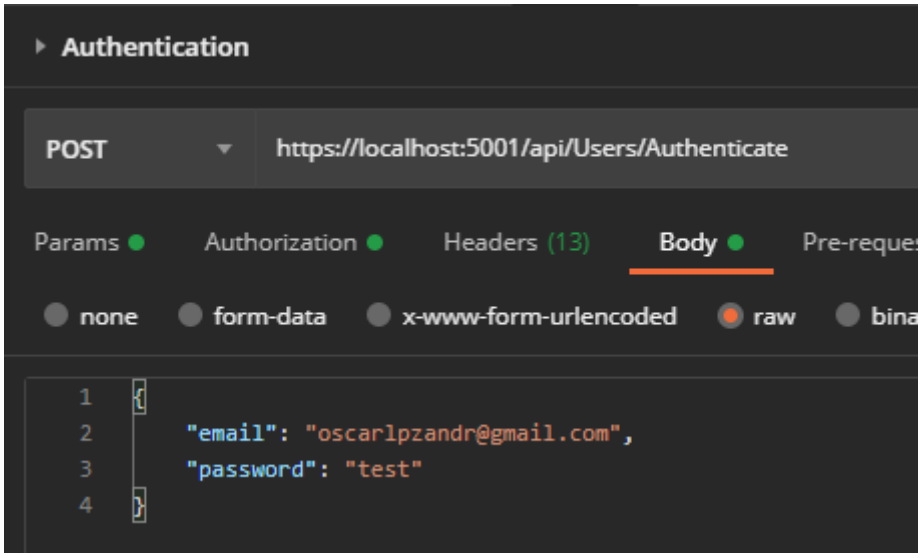

Figura 81. Autenticación en Postman

Este método nos devolverá el usuario en el que hemos autentificado (Para su uso en la aplicación híbrida) además de la cookie que nos permitirá acceder a la aplicación.

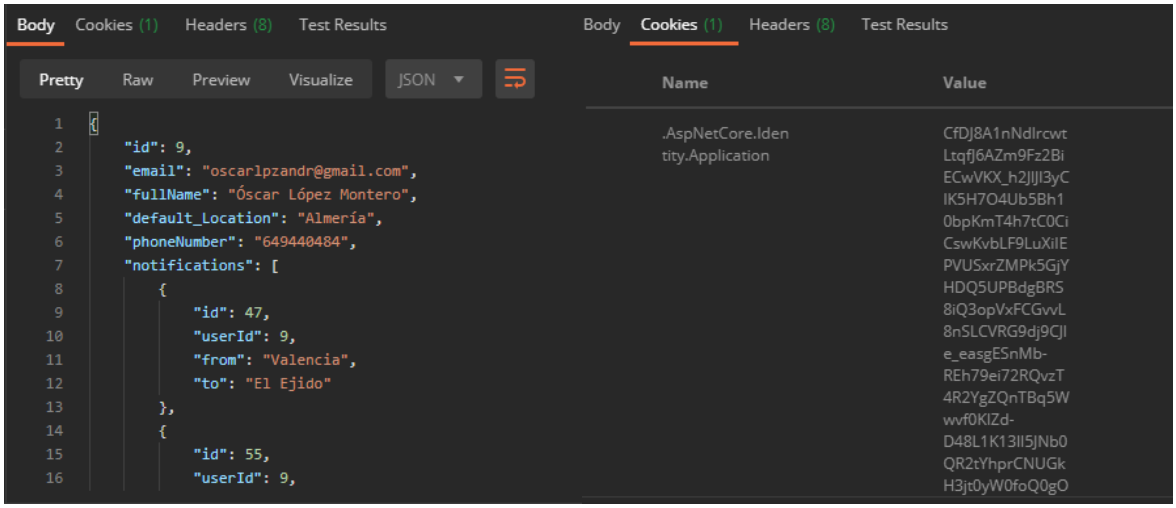

Figura 82. Respuesta 200 de la API en autenticación

Además, esta cookie también tendrá como atributo la duración de la misma y el dominio en el que está siendo usada.

El método usado en la API para la autenticación de los usuarios es el siguiente:

```
public async Task<ActionResult<User>> Authenticate([FromBody]UserLogin model)
{
    var user = await _signInManager.PasswordSignInAsync(model.Email, model.Password, false, false);
    if (user == null || !user.Succeeded)
        return BadRequest(new { message = "Username or password is incorrect" });
    var returnUser = await _userManager.FindByEmailAsync(model.Email);
    return Ok(returnUser);
}
```

Figura 83. Método de autenticación en la API

SignInManager y *UserManager* son dos servicios que añadimos a nuestros controladores mediante inyección de dependencias y que nos ayudan tanto a la autenticación como a la gestión de usuarios.

El primero es el encargado de autentificar a los usuarios en la aplicación, por defecto es compatible con Identity.

UserManager nos permite todo tipo de operaciones relacionadas con nuestro usuario de la base de datos, desde obtenerlo por email hasta generar un token de recuperación de contraseña.

4.5.3.3. Uso de Antiforgery Tokens

Cuando usamos cookies para autentificarnos estamos almacenando el método de acceso a nuestra aplicación en la memoria del navegador. Esto implica varias cosas:

- Es peligroso, debido a que pueden ser accesibles mediante ciberataques y producir robos de identidad.
- Cualquier página puede hacer uso de estas durante una sesión para realizar solicitudes imitando al usuario.

El uso de tokens de prevención de ataques de falsificación de solicitud entre sitios (*Cross-Site Request Forgery*) nos ayuda a evitar que esto suceda, de manera que no se pueda ejecutar código malicioso en nuestra aplicación web.

En ASP.Net Core, al usar *Razor Pages* estamos implementando automáticamente estos tokens, sin embargo, hay que habilitarlos en las páginas que deseemos usarlos. Para ello, usamos la etiqueta *ValidateAntiForgeryToken* en la clase o método que deseemos validarlo.

```
[ValidateAntiForgeryToken]
```

Figura 84. Etiqueta de validación de AntiForgeryToken

Podemos comprobar que funciona correctamente revisando las cookies de nuestro navegador.

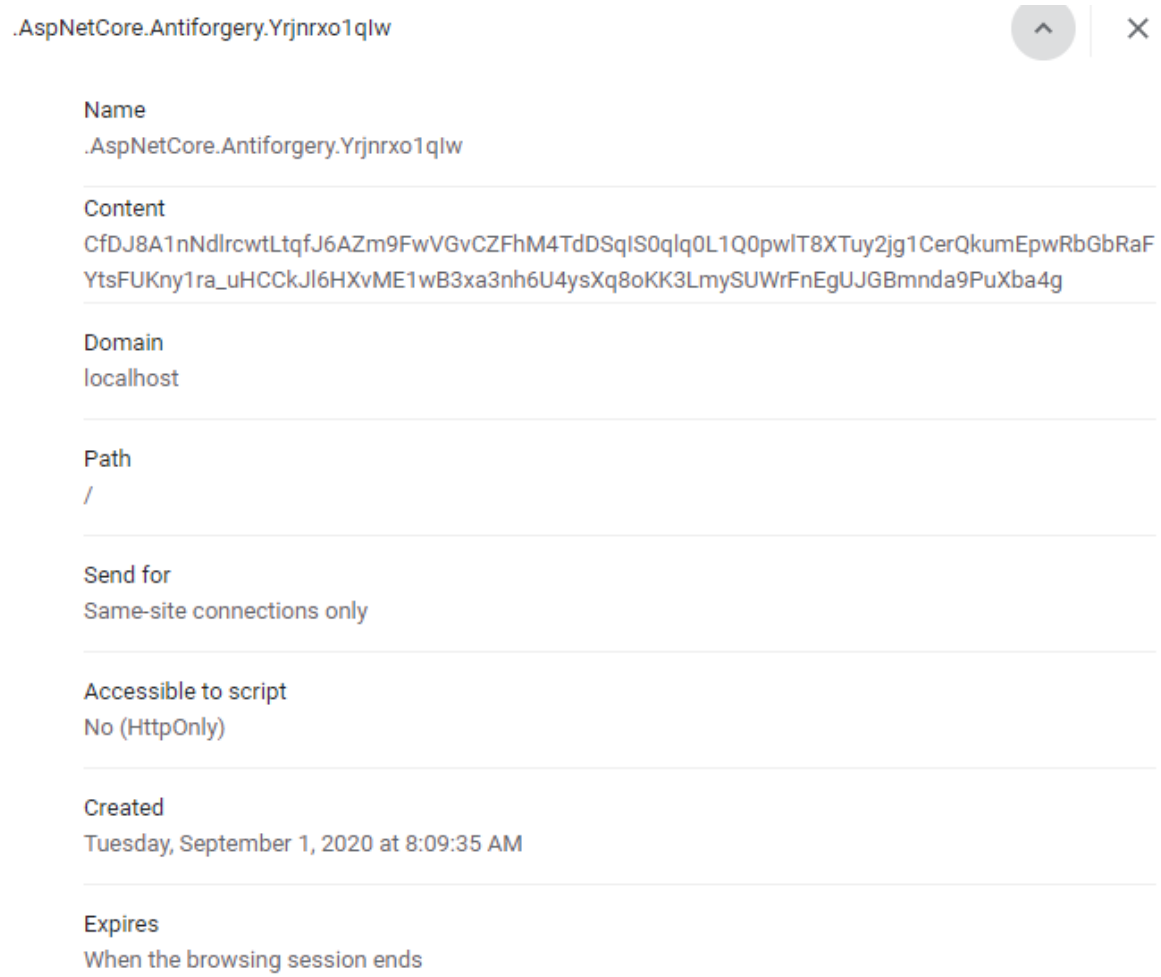

Figura 85. Antiforgery cookie en Google Chrome

4.5.3.4. Autorización basada en roles

Como ya sabemos, existen tres roles en nuestra aplicación, pero necesitamos manejar que ve, y a qué métodos de la API pueden acceder cada uno de esos usuarios.

Para la gestión de los roles de nuestra aplicación, usamos la clase *RoleManager* de la API de Identity. *RoleManager* nos permite asignar, gestionar y eliminar los roles de nuestra aplicación.

Primero, en nuestra clase de configuración *Startup*, vamos a comprobar si existe el rol que deseamos y en caso contrario lo añadimos. Esta comprobación la realizamos para que, en el hipotético caso de que nuestra aplicación se ejecute en un entorno en el que los roles aún no han sido inicializados, sea capaz de manejarlo e inicializarlo por ella misma.

El uso de wait no es recomendable en C# ya que podría bloquear nuestra aplicación, pero debido a que nos encontramos en un método sincrónico y que se ejecuta una única vez en el arranque de la aplicación, no supone ningún problema demasiado grave.

```
Task<bool> hasAdminRole = roleManager.RoleExistsAsync("Admin");
hasAdminRole.Wait();

if (!hasAdminRole.Result)
{
    roleResult = roleManager.CreateAsync(new AppRole()
    {
        Name = "Admin",
        NormalizedName = "ADMIN"
    });
    roleResult.Wait();
}
```

Figura 86. Inicialización del rol de Admin

Una vez inicializados ya podemos restringir el uso de ciertas funcionalidades dependiendo del rol del usuario, sin olvidar que el usuario tiene que tener el rol que deseamos asignado. En nuestra aplicación, los administradores pueden asignar y modificar el rol que deseen al usuario que registran a través de *UserManager*.

```
public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
    _ = returnUrl ?? Url.Content("~/");
    if (ModelState.IsValid)
    {
        var user = new User { FullName = Input.FullName, UserName = Input.Email, Email = Input.
        var result = await _userManager.CreateAsync(user, Input.Password);
        if (result.Succeeded)
        {
            await _userManager.AddToRoleAsync(user, UserRole);
            return RedirectToPage("./Index");
        }
        foreach (var error in result.Errors)
        {
            ModelState.AddModelError(string.Empty, error.Description);
        }

        return Page();
    }
```

Figura 87. Método de registro de usuario

Finalmente, usamos el atributo Authorize sobre la cabecera de la clase o método que queremos restringir a ciertos usuarios, junto al rol al que se le aplica. Si no especificamos ningún rol, el atributo Authorize comprobará únicamente si es un usuario autentificado, permitiéndole el acceso independientemente del rol que posea.

```
[Authorize(Roles = "Admin")]
6 referencias
public class CreateModel : PageModel
```

Figura 88. Autorización de rol de administrador en registro de nuevo usuario

Por otro lado, podemos permitir que los usuarios sin autentificarse usen ciertos métodos de nuestra API a través de la etiqueta *AllowAnonymous*, que usamos en métodos como el Login o la recuperación de contraseña.

```
[AllowAnonymous]
[HttpPost("RecoverPassword")]
0 referencias
public async Task<ActionResult<bool>> RecoverPassword([FromBody]string email)
{
    try
    {
        var user = await _userManager.FindByEmailAsync(email);
        if (user == null)
        {
            return NotFound();
        }
        var code = await _userManager.GeneratePasswordResetTokenAsync(user);
        code = WebEncoders.Base64UrlEncode(Encoding.UTF8.GetBytes(code));
        var callbackUrl = Url.Page(
            "/Account/ResetPassword",
            pageHandler: null,
            values: new { area = "Identity", code },
            protocol: Request.Scheme);

        await _emailService.SendRecoverMail(
            email,
            "Reset Password",
            $"Please reset your password by <a href='{HtmlEncoder.Default.Encode(callbackUrl)}'>clicking here</a>.");
        return Ok(email);
    }
    catch (Exception)
    {
        return BadRequest();
    }
}
```

Figura 89. Método de recuperación de contraseña en la API

4.5.3.5. Envío automático de notificaciones

Para implementar el envío de notificaciones de nuevos viajes por correo electrónico, hemos decidido implementar una clase encargada del envío como servicio (Ver figura 61).

Nuestro *EmailService*, tiene los métodos necesarios para el envío de correos electrónicos mediante SMTP (Simple Mail Transfer Protocol). Este se compone de 4 métodos:

SendSMTP

Encargado de enviar el correo de notificación de viaje a los interesados. Este método es llamado desde la API cuando se registra un nuevo viaje en la aplicación.

```
public async Task SendSMTP(Trip trip)
{
    MailMessage mail = new MailMessage();
    string FilePath = "wwwroot\\templates\\emailTemplate.htm";
    StreamReader str = new StreamReader(FilePath);
    string MailText = str.ReadToEnd();
    MailText = MailText.Replace("[HtmlPageHere]", string.Concat(_config.DetailsUrl, trip.Id));
    str.Close();
    mail.Subject = "A new trip has been registered in SharingCarRZ";
    mail.Body = MailText;
    foreach (var user in UsersInterestedIn(trip))
    {
        if (user.Email.Equals(trip.Trip_OwnerEmail)) continue;
        //Si es igual a el usuario que crea el viaje
        mail.To.Add(user.Email);
        //A cada user cambia email y user en el mensaje, tras esto envia un mail async.
    }
    await SendMailAsync(mail);
}
```

Figura 90. Implementación del método SendSMTP en EmailService

En él, se modifica la plantilla HTML que usamos para la notificación de nuevo viaje, añadiendo el viaje registrado como enlace en el botón. Además, se añaden todos los correos de los usuarios interesados al campo “To” del email.

UsersInterestedIn

Método que nos devuelve la lista de usuarios interesados en un viaje.

```
private List<User> UsersInterestedIn(Trip trip)
{
    var notifications = _context.Notification.Where(not => not.From == trip.From && not.To == trip.To);
    List<User> result = new List<User>();
    foreach (var not in notifications)
    {
        result.Add(_userManager.FindByIdAsync(not.UserId.ToString()).Result);
    }
    return result;
}
```

Figura 91. Implementación del método UsersInterestedIn de EmailService

SendRecoverMail

Método utilizado desde la funcionalidad de recuperar contraseña, simplemente ajusta el cuerpo del correo al del mensaje correspondiente. Este método está incluido aquí para aprovechar la instancia de SmtpClient.

```
public async Task SendRecoverMail(string email, string subject, string body)
{
    MailMessage mail = new MailMessage();
    mail.To.Add(email);
    mail.Subject = subject;
    mail.Body = body;
    await SendMailAsync(mail);
}
```

Figura 92. Implementación del método SendRecoverMail en EmailService

SendMailAsync

Método desde el cual se obtiene la información de la estancia de SmtpClient y se envía el correo pasado por parámetro. Este método es privado y es llamado por nuestros dos otros métodos.

```
private async Task SendMailAsync(MailMessage mail)
{
    try
    {
        mail.From = new MailAddress(_config.FromEmail);
        using var client = new SmtpClient("smtp@gmail.com")
        {
            Port = 587,
            EnableSsl = true,
            UseDefaultCredentials = false,
            Credentials = new System.Net.NetworkCredential(_config.FromEmail,
            _config.FromPassword)
        };
        await client.SendMailAsync(mail);
    }
    catch (Exception e)
    {
        Debug.WriteLine(e.StackTrace);
    }
}
```

Figura 93. Implementación del método SendMailAsync en EmailService

Las opciones de configuración del email las cogemos un fichero de configuración. Más adelante hablaremos de la implementación del mismo.

4.5.3.6. Scaffolding de modelos

Una de las principales ventajas del desarrollo web con ASP.Net Core es el Scaffolding.

Se trata de un framework de generación de código que nos permite inicializar la creación de las vistas y controladores partiendo de un modelo y un contexto de datos.

Para crear un elemento con scaffold, debemos ir a la opción de agregar como con cualquier otro elemento y seleccionar "Nuevo elemento con scaffold".

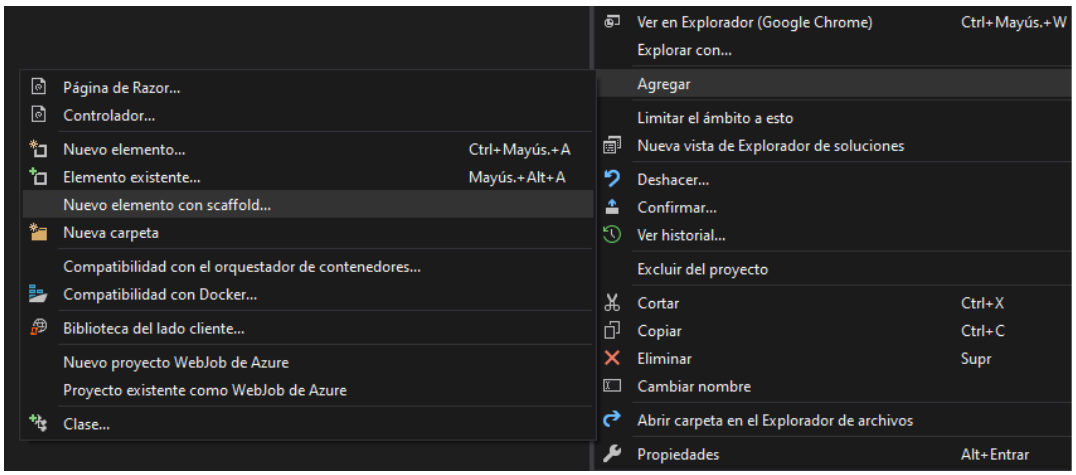

Figura 94. Opción de elemento con scaffold en Visual Studio

Una vez ahí, podemos seleccionar el scaffold que deseamos realizar. En nuestro caso, hemos usado Scaffolding para automatizar la creación de los métodos básicos de la API:

Get, Post, Delete y *Put.*

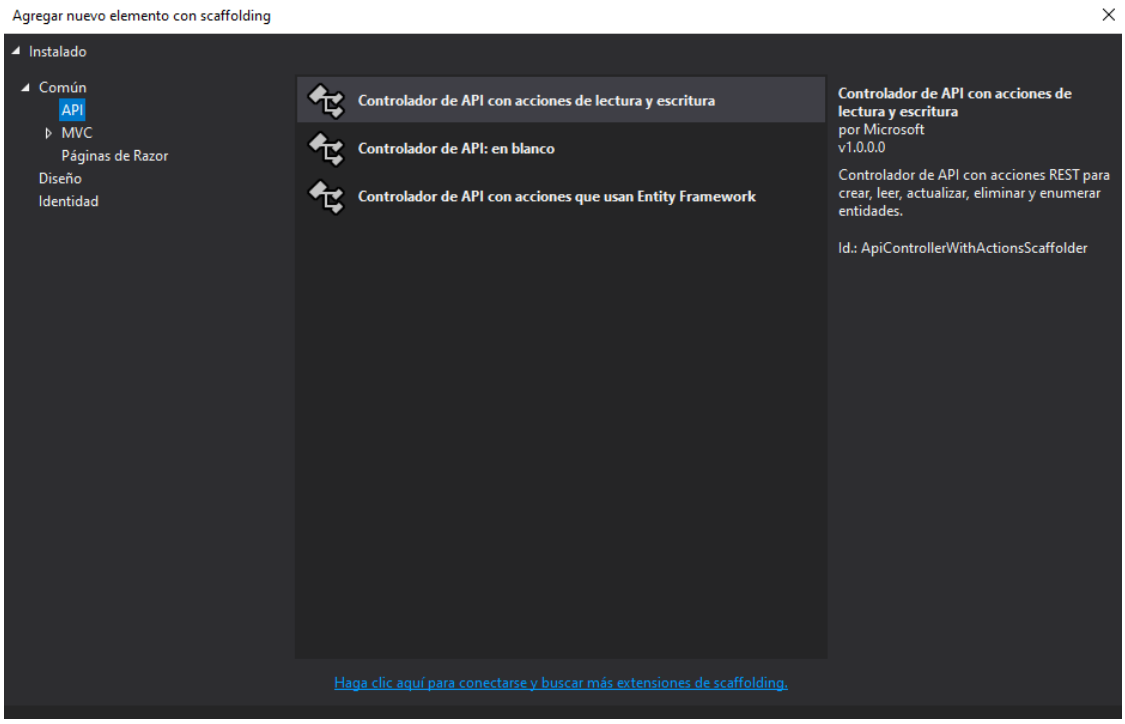

Figura 95. Creación de controlador con Scaffold

Un ejemplo de métodos creados mediante *Scaffold* son los *Gets* "básicos" que usamos en nuestra aplicación.

```
// GET: api/Trips
[HttpGet]
0 referencias
public async Task<ActionResult<IEnumerable<Trip>>> GetTrip()
{
    return await _context.Trip.ToListAsync();
}

// GET: api/Trips/5
[HttpGet("{id}")]
0 referencias
public async Task<ActionResult<Trip>> GetTrip(int id)
{
    var trip = await _context.Trip.FindAsync(id);

    if (trip == null)
    {
        return NotFound();
    }

    return trip;
}
```

Figura 96. Método Get generado con Scaffold

4.5.3.7. Enlace de secciones de configuración a servicios

Usar valores de configuración en ASP.Net Core no basta con referirnos a ellos mediante una clase, sino que, habría que inyectarlos como dependencia en la clase que vayamos a usarlos y referirnos a ellos con un Get para que nuestra aplicación busque la sección en el fichero. Este método, no solo es horrible e ineficiente, sino que puede acarrear tanto problemas de rendimiento como de calidad de código, ya que lo hace bastante menos legible.

Para solucionarlo, vamos a inicializar modelos que usaremos como configuración y los vamos a añadir a nuestra *ServiceCollection* para más tarde inyectarlos como dependencia.

Vamos a seguir el ejemplo de EmailService y sus parámetros de configuración.

Primero, tenemos que especificar nuestros parámetros en nuestro fichero de configuración JSON.

```
"SmtpSettings": {
  "DetailsUrl": "/Trips/Details?id=",
  "FromEmail": "TripReminderRZ@gmail.com",
  "FromPassword": "****************",
  "MailServerPort": "587"
}
```

Figura 97. Sección de configuración de SMTP en appsettings.json

Después, registramos esa configuración sobre nuestra clase modelo en el método *ConfigureServices.*

```
services.Configure<EmailModel>(Configuration.GetSection("SmtpSettings"));
```

Figura 98. Configuración de EmailModel

Una vez tengamos nuestro modelo registrado, podemos inyectarlo como dependencia para usar los parámetros de configuración accediendo a los atributos de la clase.

```
public class EmailService : IEmailService
{
    private readonly SharingCarRZWebContext _context;
    private readonly UserManager<User> _userManager;
    private readonly EmailModel _config;

    0 referencias
    public EmailService(SharingCarRZWebContext context, UserManager<User> userManager, IOptions<EmailModel> config)
    {
        _context = context;
        _userManager = userManager;
        _config = config.Value;
    }
```

Figura 99. Inyección de modelo de configuración como dependencia

4.5.3.8. Uso de tokens para recuperación de contraseña

A la hora de recuperar la contraseña de una cuenta, debemos seguir varias medidas de seguridad:

1. Que no pueda ser recuperada por un usuario cualquiera, es decir, tenemos que ligar esta funcionalidad al correo electrónico de los usuarios.
2. Que los usuarios no puedan reiniciar la contraseña de otro usuario, aun conociendo su dirección de correo electrónico.
3. Que el usuario pueda ajustar una contraseña a su gusto, y que cumpla los estándares de seguridad.
4. Asegurarnos de que se trata del usuario aportando el token enviado al correo.
5. No revelar en el correo de recuperación datos sensibles como el token que se usa en la verificación o la antigua contraseña.

Siguiendo estos pasos, vamos a explicar la implementación de la funcionalidad de recuperación de contraseña.

Primero, el usuario hace clic en el apartado de *¿Has olvidado tu contraseña?* y es redirigido a la ventana de recuperación de la misma, donde se le pide su correo electrónico (Ver figura 64).

Una vez en esta ventana, el usuario introduce su correo y es notificado como que se procederá a la recuperación de la contraseña, y se le ha enviado un correo con las instrucciones a seguir. Un paso importante en este apartado es que, en nuestra aplicación,

siempre informamos al usuario de que se le enviará el correo, incluyendo el caso en el que no exista, esto es con el fin de evitar que algunos usuarios soliciten cambios de correo a prueba y error para saber que correos están registrados en nuestra base de datos.

Una vez hace clic en enviar con un correo válido, nuestra API recibe una solicitud de recuperación de contraseña, comprueba el correo y en caso de que sea válido, genera una dirección de recuperación con un token embebido (Ver figura 85).

Es importante resaltar el uso de WebEncoders para introducir el token. En la imagen podemos ver como el código está embebido en la url.

Figura 100. Solicitud GET de la página de reseteo de contraseña desde Chrome

4.5.3.9. Enrutamiento

El enrutamiento es la tarea más simple en una aplicación de ASP.Net Core, esto es debido a que nuestro framework se encarga de mapear todas las vistas por el nombre de la clase que la contiene a través de la etiqueta *@page*.

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapRazorPages();
    endpoints.MapControllers();
});
```

Figura 101. Configuración de enrutamiento a acciones de vistas y controladores

En el caso de los controladores, para que .Net Core se encargue de mapearlos como a las vistas, hace falta añadirles una etiqueta sobre la clase.

```
[Route("api/[controller]")]
```

Figura 102. Etiqueta de enrutamiento a controlador

En este ejemplo la dirección para referirnos a ese controlador será: *https://marathonrz/api/nombreDelControlador*.

4.5.3.10. Uso de middleware de redirección HTTPS

Para asegurarnos que nuestras solicitudes HTTP son lo más seguras posibles, vamos a usar middleware de redirección HTTPS para obligar a que todas nuestras solicitudes viajen de forma más segura.

El framework de ASP.Net Core ofrece un método para forzar la redirección de las solicitudes, únicamente tenemos que añadirlo a nuestro método de configuración (Ver figura 71).

Debido a que usamos HTTPS, no es necesario encriptar de ninguna manera el contenido de nuestras solicitudes, ya que no tendría ningún sentido especial encriptar datos que van a volver a ser encriptados por el protocolo.

4.5.3.11. Uso de EntityFramework

EntityFramework es sin duda la mejor funcionalidad de ASP.Net Core, y la que mayor carga de trabajo nos ha ahorrado en este proyecto.

EntityFramework nos permite trabajar con nuestra base de datos librándonos tanto de diseñar las tablas de la base de datos, como de escribir las consultas que se realizan en SQL.

Funciona de manera simple, nuestra aplicación tiene un DBContext en el que almacenamos todo lo que vamos a usar.

```
27 referencias
public DbSet<Trip> Trip { get; set; }

21 referencias
public DbSet<Notification> Notification { get; set; }

12 referencias
public DbSet<User> User { get; set; }
15 referencias
public DbSet<UserTrip> UserTrip { get; set; }
20 referencias
public DbSet<Location> Location { get; set; }
```

Figura 103. Atributos de DBContext

Una vez registramos lo que queremos guardar en nuestro DBContext, cada vez que añadamos un modelo o registremos un cambio en los mismos, podemos modificar nuestra base de datos añadiendo una migración de la nueva *snapshot* con *AddMigration.*

Para actualizar la base de datos actual a la última migración, usaremos el comando *Update-Database.*

EntityFramework guarda todas las migraciones que se han realizado en la aplicación en caso de querer deshacerlas.

```
1 referencia
public partial class UserTripConId : Migration
{
    15 referencias
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "UserTrip",
            columns: table => new
            {
                UserId = table.Column<int>(nullable: false),
                TripId = table.Column<int>(nullable: false)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_UserTrip", x => new { x.TripId, x.UserId });
            });
    }

    15 referencias
    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(
            name: "UserTrip");
    }
}
```

Figura 104. Ejemplo de migración con EF

4.5.4. Despliegue

Para el despliegue de la web y la API vamos a usar una funcionalidad de Visual Studio tal y como hemos hecho con nuestra aplicación multiplataforma.

Visual Studio nos permite la publicación de aplicaciones web desde su aplicación mediante la opción de “Publicar”.

El despliegue va a ser realizado en un *hosting* de *Plesk Obsidian* interno destinado a la publicación de aplicaciones web.

A la hora de publicar la aplicación web, Visual Studio nos ofrece un amplio abanico de opciones sobre las que publicar nuestra aplicación:

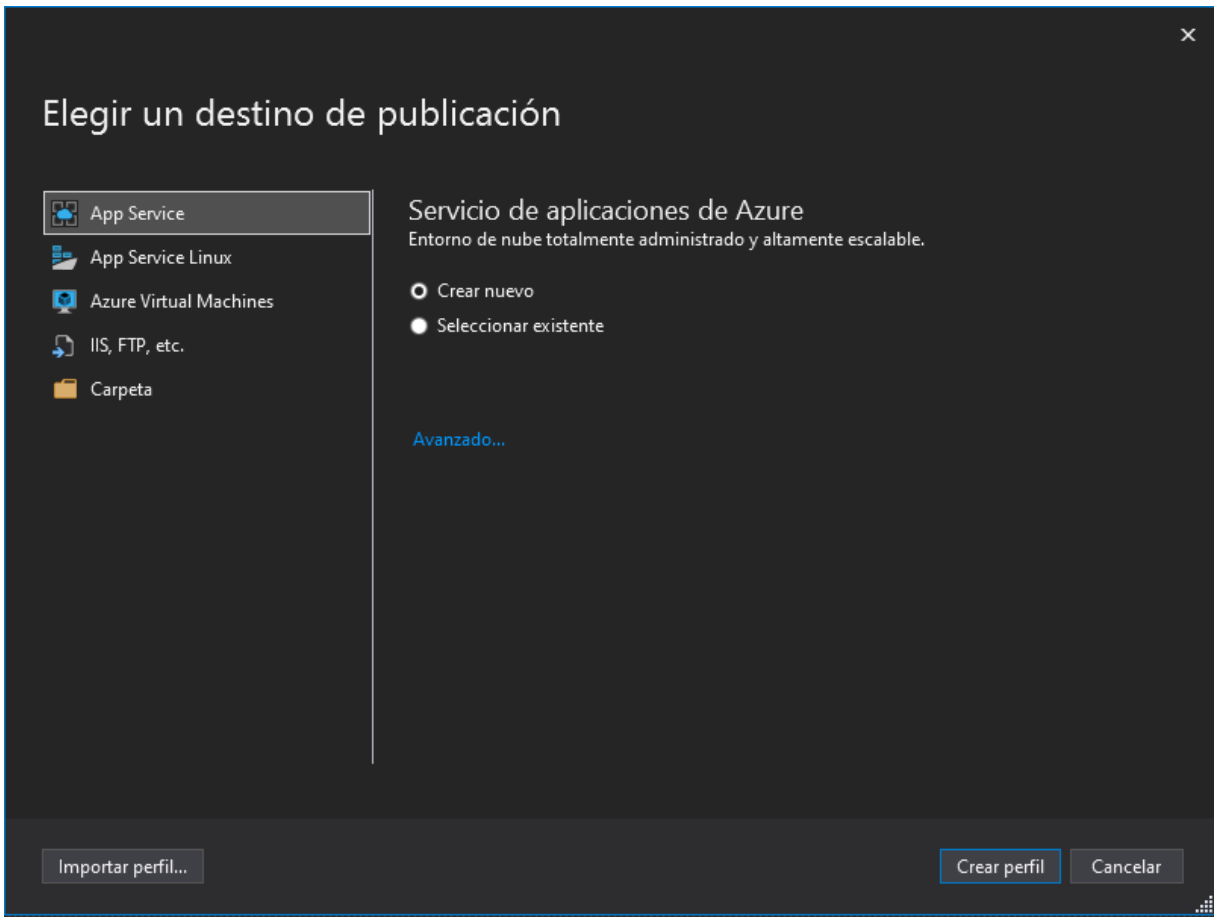

Figura 105. Opciones de despliegue web de Visual Studio

En nuestro caso, vamos a elegir la opción *de IIS, FTP o Web Deploy*.

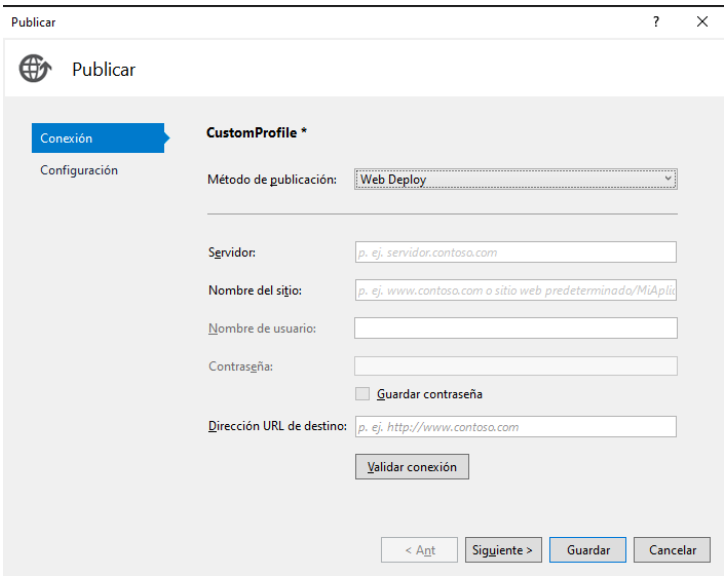

Figura 106. Parámetros de entrada para publicación en Web

Especificando los parámetros requeridos, Visual se encargará de la subida de la aplicación a la web. Para asegurarse del correcto funcionamiento, podemos validar la conexión, así como cambiar la configuración en el siguiente paso.

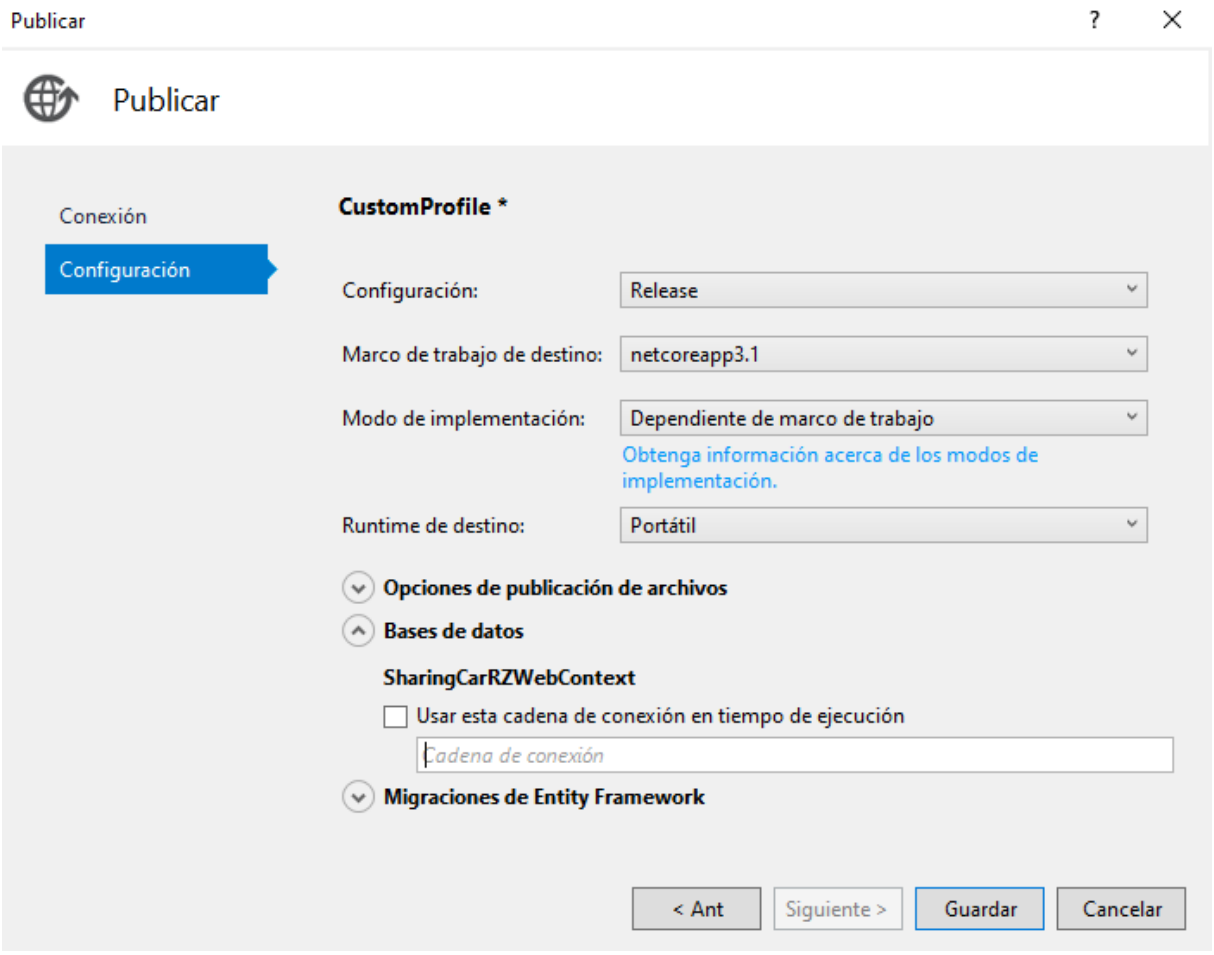

Figura 107. Configuración de la aplicación web a publicar

Tras especificar la configuración del despliegue, así como, en caso de que lo deseemos y no hayamos cambiado antes, la cadena de conexión con la base de datos. Nuestro Web Deploy subirá los ficheros a la URL especificada y dejará la aplicación lista para su uso.

5. Conclusiones y trabajo futuro

Hemos desarrollado una aplicación multiplataforma en Xamarin y una aplicación web junto a una API con ASP.Net Core cumpliendo con la funcionalidad esperada de organizar los viajes entre sedes.

Respecto a los conocimientos adquiridos en la realización de este trabajo de fin de grado, he aprendido varias cosas con las que nunca había trabajado o que no conocía:

- Lógica detrás del funcionamiento de una aplicación multiplataforma.
- Conocimientos adquiridos respecto a patrones de diseño en aplicaciones híbridas y web.
- Trabajar con solicitudes HTTP.
- Usar JSON para el intercambio de datos entre la aplicación y la API.
- Mejora en la percepción de la simplicidad en el diseño de la aplicación.
- Metodologías SCRUM en el desarrollo de una aplicación, así como la importancia de las reuniones con el cliente y la organización del proyecto.
- Importancia de implementar el máximo número de medidas de seguridad en la aplicación.
- Creación de estilos atractivos personalizando el CSS.

Estas son solo unas pocas cosas que he podido aprender durante el desarrollo de este trabajo de fin de grado.

Por otro lado, también hay varios apartados a mejorar en este TFG, así como apartados para tener en cuenta en el desarrollo de mis futuros proyectos tales como:

- Mejorar el manejo de excepciones, haciéndolo lo más específico posible, es decir, manejando el error de distinta manera dependiendo del tipo de este.
- Intentar minimizar el número de solicitudes a la API, pero sin cargarlas demasiado.
- Esconder y reducir el tiempo de respuesta de la aplicación mejorando el uso de animaciones.

Este proyecto me ha ayudado a orientar mi futuro laboral al desarrollo de aplicaciones móviles, no solo con Xamarin, sino con cualquier framework que pueda llegar a usar, ya que el patrón de diseño es similar además de poder reutilizar lo aprendido acerca de interfaces y animaciones, así como el uso de datos de una API o back-end.

6. Bibliografía

[1] Turek, M. (2016). *Samsung Insights*. Obtenido de Employees Say Smartphones Boost Productivity by 34 Percent: Frost & Sullivan Research. Recuperado el 23 de Julio de 2020 de https://insights.samsung.com/2016/08/03/employees-say-smartphones-boost-productivity-by-34-percent-frost-sullivan-research/

[2] Microsoft. (2020). *Microsoft.com*. Obtenido de What is Xamarin. Recuperado el 1 de Agosto de 2020 de https://docs.microsoft.com/es-es/xamarin/get-started/what-is-xamarin

[3] Zwaan, R. (s.f.). *RijkZwaan.es*. Obtenido de Sobre nosotros. Recuperado el 3 de Agosto de 2020 de https://www.rijkzwaan.es/sobre-nosotros

[4] Microsoft. (2015). *Microsoft.com*. Obtenido de Introduction to the C# language and .NET framework. Recuperado el 25 de Julio de 2020 de https://docs.microsoft.com/es-es/dotnet/csharp/getting-started/introduction-to-the-csharp-language-and-the-net-framework

[5] Marrs, T. (2017). *JSON at Work.* O'Reilly Media, Inc.

[6] Json.org. (s.f.). Obtenido de Introducción a JSON el 25 de Julio de 2020. Recuperado el 25 de Julio de 2020 de https://www.json.org/json-es.html

[7] Frisbie, M. (2019). *Professional JavaScript for Web Developers, 4th Edition.* Wrox.

[8] Kyrnin, J., & Meloni, J. C. (2019). *Sams Teach Yourself HTML, CSS, and JavaScript All in One, Third Edition.* Sams.

[9] Mora, S. L. (2001). *Programación en Internet: Clientes Web.* Editorial Club Universitario. R Obtenido de Aprenderaprogramar.es.

[10] Microsoft. (2020). *Introducción a ASP.NET Core.* Obtenido de Microsoft.com. Recuperado el 1 de Agosto de 2020 de https://docs.microsoft.com/es-es/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-3.1

[11] Microsoft. (s.f.). *Visual Studio Release Notes*. Obtenido de Microsoft.com. Recuperado el 28 de Julio de 2020 de https://docs.microsoft.com/en-us/visualstudio/releases/2019/release-notes#16.7.0

[12] GitHub. (s.f.). *GitHub Features*. Obtenido de https://github.com/features

[13] Microsoft. (2017). *Docs.microsoft.com*. Obtenido de Patrón Model-View-ViewModel. Recuperado el 30 de Julio de 2020 de https://docs.microsoft.com/es-es/xamarin/xamarin-forms/enterprise-application-patterns/mvvm

[14] Lottie. (2020). *What is Lottie*. Obtenido de LottieFiles. Recuperado el 29 de Julio de 2020 de https://lottiefiles.com/what-is-lottie

[15] Microsoft. (2020). *Microsoft Docs*. Obtenido de Introducción a Razor Pages en ASP.NET Core. Recuperado el 1 de Agosto de 2020 de https://docs.microsoft.com/es-es/aspnet/core/razor-pages/?view=aspnetcore-3.1&tabs=visual-studio

[16] Anderson, R. (2020). *Microsoft Docs*. Obtenido de Introducción a Identity. Recuperado el 12 de Agosto de 2020 de https://docs.microsoft.com/es-es/aspnet/core/security/authentication/identity?view=aspnetcore-3.1&tabs=visual-studio

[17] Microsoft.com. (2020). *What is Asp.NET-Core*. Obtenido de Dotnet.Microsoft.com. Recuperado el 12 de Agosto de 2020 de https://dotnet.microsoft.com/learn/aspnet/what-is-aspnet-core

[18] Microsoft.com. (2019). *SQL Server 2019*. Obtenido de Microsoft.com. Recuperado el 20 de Agosto de 2020 de https://www.microsoft.com/es-es/sql-server/sql-server-2019

[19] Allen, K. S. (2013). *Asp.NET-Core Identity*. Obtenido de OdetoCode. Recuperado el 20 de Agosto de 2020 de https://odetocode.com/blogs/scott/archive/2013/11/25/asp-net-core-identity.aspx

[20] Microsoft. (2020). *Asp.Net-Core Middleware*. Obtenido de Microsoft.com. Recuperado el 31 de Agosto de 2020 de https://docs.microsoft.com/es-es/aspnet/core/fundamentals/middleware/?view=aspnetcore-3.1

UNIVERSIDAD DE ALMERÍA

Resumen/Abstract

Es innegable que el uso de tecnologías móviles se encuentra en pleno auge y que hoy día, estos dispositivos forman parte de la vida cotidiana de cada uno. Por ello, y debido a la variedad de sistemas operativos que existen en el mercado, el desarrollo de aplicaciones móviles multiplataforma se ha convertido en una de las tareas más importantes de la actualidad.

En Rijk Zwaan, las aplicaciones móviles son una manera importante de introducir información de manera sencilla y rápida, sin que el trabajador tenga que perder mucho tiempo en rellenar formularios a ordenador, y simplificando las labores del mismo.

En este caso, se ha propuesto gestionar de una manera más sencilla los viajes entre fincas de Rijk Zwaan Ibérica y el envío de material entre las mismas, de manera que se haga un uso común de los vehículos destinados a dicho uso, y suponiendo un menor gasto en servicios de transporte de mercancías.

It's undeniable that mobile technologies are currently booming and that those devices have become really important in our daily lives.
For that reason and due to the variety of operative systems in the market, cross-platform mobile development has become one of the most demanded jobs nowadays.

Mobile applications have a really important role at Rijk Zwaan since they are an easy and fast way to input information into their storage without losing time filling in big forms.

In this project, we want to manage trips and material shipping in between Rijk Zwaan Iberica locations to provide a common usage of the vehicles, making the company to save money in delivery services.