Aplicación móvil del tiempo atmosférico usando la API de Openweathermap

Documento:

Memoria

Autor:

Daniel Muñoz Romero

Director:

Pau Fernández Durán

Titulación:

Grado en ingeniería de sistemas audiovisuales

Convocatoria:

Primavera, 2022.

TRABAJO FIN DE ESTUDIOS

I. Resumen

Desde la aparición de los smartphones el incremento de aplicaciones ha crecido exponencialmente, y una de las que no falta nunca son las aplicaciones del tiempo.

Si bien la mayoría son simples widgets que te redirigen a sus páginas webs, a medida que han surgido nuevas tecnologías para las creaciones de estas, han ido evolucionando. Una de estas tecnologías sería por ejemplo Flutter, que permite desarrollar aplicaciones para Android y IOS, los dos grandes sistemas operativos que ocupan el mercado. Es por esto que he dirigido mi proyecto a la creación de una aplicación para ambas plataformas que mezcle la simpleza de los widgets antiguos con los nuevos estilos de aplicaciones más dinámicos, con nuevas funciones y mapas interactivos.

A lo largo de esta memoria se describirá; el proceso de creación de la aplicación; el estudio de las API que proporcionan los datos y la creación de estos modelos de datos; el diseño de las pantallas que componen la aplicación, desde el boceto principal pasando por las aplicaciones a usar para ello hasta la creación final de estas dando una explicación detallada de cada elemento que lo compone y el porqué. Por último, se describirán las funcionalidades diseñadas, tanto su desarrollo como su implementación.

Una vez todas las funciones han sido implementadas, se ha pasado a la fase de testeo. La aplicación ha sido testeada en diferentes modelos, tanto en antigüedad como en tamaño. Por último, se ha pulido el diseño para que sea agradable y dinámico.

Finalmente se mostrarán las pantallas del resultado final tal y como vendría tras la descarga desde el store.

II. Abstract

Since the emergence of mobile phones the rising number of applications has been increasing exponentially, and one of the must have are the weather apps.

Even though most of them are just widgets that usually redirect to their web pages, as new technologies have appeared for the development of applications, these have been evolving. One of these would be for example Flutter, which allows you to develop applications for both Android and IOs, the two most important operating systems nowadays. And that reason is why I have led my project to the development of an application for both platforms that mixes the simplicity of the old widgets with the style from the new ones, more dynamics, with new functions and interactive maps.

Throughout this memory it will be described; the creation process of the application; the API study that provide us with all the data and the creation of their data models; the design of our mobile screens, from the first sketch and which applications have been used to the final design, explaining every element used and why. Lastly, all the functionalities designed will be descrived, both their development and implementation.

Once all the functionalities have been implemented, it was turn to the test phase. The application has been tested with different models, both antiquity and size to check if it fits well. Lastly, the design has been polished to make it more comfortable and dynamic.

Finally all the screens will be shown just as they will be after downloading it from the store.

III. Sumario

IV. Índice de tablas

Título y número de todas las tablas por orden de aparición en el texto.

V. Índice de figuras

Título y número de todos los gráficos por orden de aparición en el texto.

VI. Lista de abreviaturas/Glosario

AppBar: Widget situado en la parte superior de la pantalla, se suele utilizar como barra de herramientas colocando buttons y otros widgets.

BottomNavigationBar: Widget situado en la parte inferior de la pantalla, se compone de mínimo dos elementos, Icon y Text, este último obligatorio. Su uso principal es el de navegar por diferentes niveles de la aplicación.

Column: Widget que muestra a sus hijos en sentido vertical.

Container: Widget inicialmente vacío o de tamaño mínimo, que se adapta al tamaño de su hijo. Permite modificar su tamaño con sus propiedades height y weight.

CustomScrollView: ScrollView o widgets con la propiedad de deslizarse que utiliza Slivers para crear distintos efectos durante estos desplazamientos.

GestureDetector: Widget que proporciona la propiedad de detectar gestos a su hijo y ejecutar una acción en consecuencia.

GridView: Tipo de ScrollView que muestra sus elementos en forma de cuadrícula personalizada.

IconButton: Widget compuesto por un Icono o imagen, el cual ejecuta una acción al ser pulsado.

Listview: Tipo de ScrollView que muestra sus elementos linealmente, la dirección se puede controlar mediante la propiedad ScrollDirection.

MaterialApp: Widget de nivel superior que se encarga de buscar las rutas a las diferentes pantallas de una aplicación mediante una tabla de rutas o creando la ruta directamente.

Padding: Widget que añade espacio a su hijo, en la dirección indicada.

PageView: Tipo de ScrollView en que cada elemento tiene el mismo tamaño, por ejemplo pantallas enteras, la dirección de desplazamiento se puede controlar mediante la propiedad ScrollDirection.

Row: Widget que muestra a sus hijos en sentido horizontal.

Scaffold: Widget que implementa la estructura principal de diseño de Flutter. Se sitúa en el nivel más alto del árbol de widgets, e implementa otros widgets básicos como AppBar y BottomNavigationBar.

Stateful widget: Widget que dispone de estado, lo que le permite cambiar o redibujar mientras sus datos cambian.

SliverAppBar: AppBar envuelta en un Sliver, es el elemento principal de un CustomScrollView siendo este su primer hijo.

Stateless Widget: Widget sin estado, se usa principalmente en elementos fijos como pueden ser imágenes.

Text: Widget que muestra texto.

Widget: Elemento principal de Flutter, todo en Flutter son widgets, y estos pueden tener estado o no. Ej: Text, Container, Row, Padding.

1. Introducción

1.1 Objeto

El objetivo de este proyecto será el desarrollo de una aplicación móvil del tiempo totalmente funcional usando datos ofrecidos por una API gratuita y abierta.

La aplicación permitirá, aparte de consultar el tiempo actual, ver estadísticas por horas y días en gráficas, guardar tus favoritos, observar distintos mapas climáticos de la zona, búsqueda de zona mediante GoogleMaps o bien escrita, auto corrector y sugerencias a estas búsquedas y por último notificaciones pop cuando la aplicación está en segundo plano.

La principal diferencia entre esta aplicación y otras está en el hecho de que la mayoría de las aplicaciones de este ámbito te redirigen a las webs cuando buscas más información extra y no te permiten navegar por ellas mismas, mientras que en esta aplicación no hay enlaces externos y todo se hace mediante navegación en la app lo que la hace más cómoda.

Por último, la selección de ciudad o punto en el cual queremos recibir los datos a través de GoogleMaps hace más precisa nuestra selección.

1.2 Alcance

El proyecto incluye:

- El diseño de las pantallas del tiempo:
  - Se diseñarán todas las pantallas que compondrán la aplicación desde su totalidad.
- Gestión de los datos proporcionados por terceros:
  - Se utilizarán y gestionarán mediante gráficas o uso dentro de widgets en la aplicación para proporcionar la información al usuario.
- Gestión de imágenes y almacenamiento de estas:
  - Las diferentes imágenes relacionadas con el tiempo se almacenarán en los *assets* del programa para evitar el gasto de datos móviles cada vez que actualicemos pantallas. Un *asset* es un archivo o recurso que se incluye en la aplicación, pueden ser imágenes, videos, archivos o cualquier otro recurso que es accesible por la aplicación en tiempo de ejecución. En Flutter para que estos *assets* estén disponibles se han de guardar, comúnmente en una carpeta con el mismo nombre, e indicar a Flutter en su archivo pubspec.yaml el directorio de esta.
- Diseño de notificaciones y configuración:
  - Se diseñarán e implementarán notificaciones cuando la aplicación esté en segundo plano, informando del tiempo actual.
- Implementación de mapas del tiempo:
  - Se diseñarán mapas del tiempo a partir de plantillas suministradas por la API.
- Autoayuda en la búsqueda de localizaciones:
  - Se implementará una autoayuda en la búsqueda de localidades mediante una API para facilitar al usuario su uso.

El proyecto no incluye:

- Gestión de base de datos externas:
  - El proyecto no abarca el uso de base de datos propias ya que no hay necesidad de almacenar datos.
- Notificaciones de alertas de tiempo:
  - El proyecto incluye notificaciones del tiempo, mostrando la probabilidad de lluvia, la temperatura actual y la ciudad actual que muestra nuestra localización. Estas notificaciones se ejecutarán cuando la aplicación pase a segundo plano.

1.3 Requisitos

Flutter: es un framework que utiliza Dart como lenguaje de programación y que permite desarrollar aplicaciones tanto móviles, como web tanto para iOS como para Android

APIs: En esta aplicación utilizaremos diversas API para recibir los distintos datos necesarios, tanto para los datos climatológicos, como para transformar geolocalizaciones en localizaciones cada una de ellas requerirá su API key correspondiente, todas tienen versión gratuita y de pago, en este caso se utilizará la versión de estudiantes, que mezcla funcionalidades de ambas, proporcionando los datos necesarios.

- Openweathermap: Nos proporciona los datos climáticos, también permite el cambio de geolocalización a localización, pero es menos preciso que el utilizado posteriormente. Nos proporciona las capas de los mapas del tiempo, los cuales podremos modificar a nuestro gusto según los datos que añadamos en la llamada.
- Google cloud platform: Nos proporciona el cambio de geolocalización por localización aparte de proporcionarnos los mapas que se usan para seleccionar la localización.
- GeocodeAPI: Nos proporciona un autocompletado en la búsqueda de localizaciones, y nos proporciona información necesaria como geolocalización y país.

Como backend, los únicos datos a guardar necesarios serán las ciudades favoritas las cuales irán guardadas en las preferencias de usuario en formato json.

Webs y aplicaciones útiles:

- Postman: Permite hacer las llamadas a las API y observar los datos recibidos.
- app.quicktype.io/: Web de ayuda para creación de los modelos de datos.

1.4 Justificación

Las aplicaciones del tiempo actuales suelen ser o bien poco detalladas, datos de pocos días e información muy superficial o bien si contienen mucha información ésta está en la web por lo que la app es un simple enlace a ella.

El objetivo de la aplicación desarrollada en este proyecto es solucionar ambos problemas, tener una app con información muy detallada y que en ningún caso vaya a webs de terceros o haya que navegar por web para obtenerlos.

Por último, también para hacerla más llamativa y precisa, tendrá algunas mejoras con respecto a aplicaciones similares, como por ejemplo la búsqueda de localidad con GoogleMaps y las notificaciones cuando la aplicación esté en segundo plano y no tener que abrirla si simplemente queremos ver pequeños datos.

2 Antecedentes y/o revisión del estado de la cuestión

En este apartado se estudiará la evolución de los diferentes sistemas operativos para móviles, cómo se distribuye el mercado actual, cómo ha afectado a la creación de aplicaciones, qué lenguajes nativos se han desarrollado para ellos y cómo han ido apareciendo diferentes frameworks multiplataformas que han aligerado el tiempo de creación de aplicaciones.

Se analizará cómo han evolucionado las aplicaciones del tiempo en base a la sencillez de desarrollo actual.

2.1 Sistemas operativos

Si bien, los primeros smartphones datan de hace más de 20 años, no podemos hablar de smartphones tal y como los conocemos ahora hasta la aparición del primer IPhone en 2007 el cual supuso una innovación con la desaparición de las teclas y pantalla táctil y supuso un punto de partida para la creación de aplicaciones actuales, interactivas y dinámicas.

En aquel momento la distribución de los diferentes sistemas operativos estaba muy fraccionada: Symbian se llevaba la mitad del mercado y el resto se dividía en más de 10 sistemas operativos distintos.

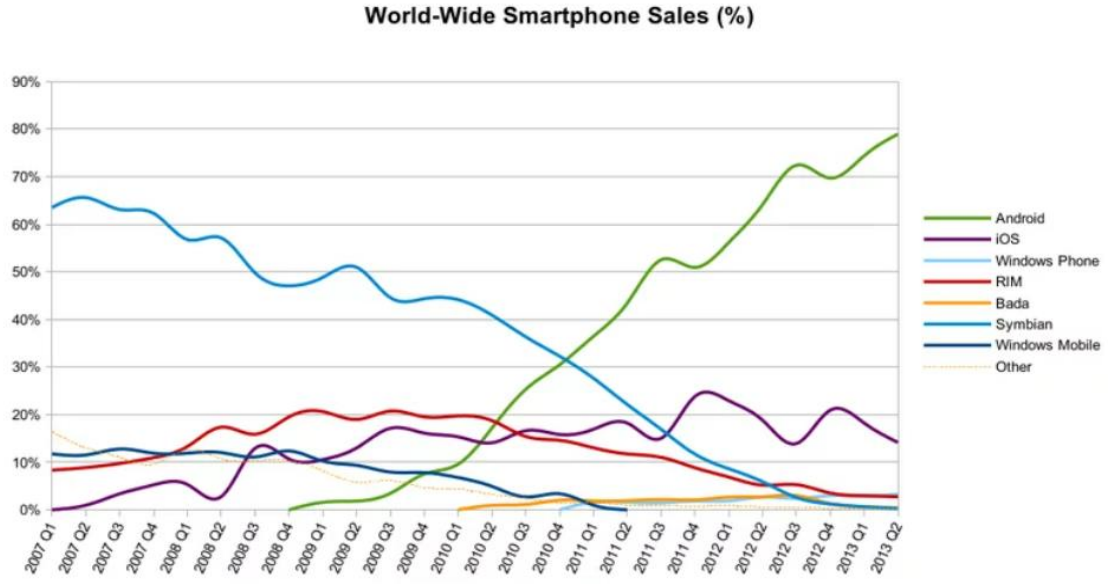


Figura 1. Crecimiento sistemas operativos 2007-2013 (Fuente: androidauthority.com)

Durante los años siguientes y con la aparición del Samsung Galaxy II y smartphones low-cost el mercado se fue dividiendo en 2 direcciones: iOS y Android, manteniendo este último más de un 80% del mercado.

2.2 Lenguajes nativos

Con relación a esta evolución el desarrollo de aplicaciones móviles se ha estandarizado en dos sistemas operativos cada uno con su lenguaje nativo propio: Kotlin para Android y Swift para iOS.

2.2.1 Android-Kotlin

Kotlin, desarrollado en 2011 por JetBrains es el lenguaje oficial de programación móvil para Android nativo desde 2019, desbancando a Java. En 2012 fue liberado bajo la licencia Apache 2. Utiliza la máquina virtual de java y todas sus librerías por lo que son interoperables, aunque es flexible en el tipado.

2.2.2 iOS-swift

Swift, desarrollado por Apple en 2014 y liberado bajo la licencia Apache 2 en 2015. Es el lenguaje oficial de iOS, desbancando a Objetive-C. Utiliza todas las librerías de este.

2.3 Frameworks multiplataforma

El desarrollo de frameworks multiplataforma surgió con la intención de reducir costes y tiempo a las empresas evitando el desarrollo de una misma aplicación en dos lenguajes distintos y unificando el código en uno solo. Muchas empresas no tienen los medios económicos o bien tiempo para desarrollar su aplicación en todos los sistemas operativos disponibles y con la consecuente pérdida de mercado que ello supone.

Si bien trabajar sobre nativo siempre es lo óptimo, ya que ofrece la posibilidad de modificar cualquier pequeño detalle, las últimas versiones de los frameworks más utilizados y los nuevos frameworks que van saliendo han solucionado gran parte de estos problemas.

Entre los más conocidos se encuentran: React Native, Ionic, Xamarin, y una última incorporación que ha ido ganando terreno, Flutter, el framework utilizado en este proyecto.

2.4 Flutter

Flutter es un SDK (Kit de Desarrollo de Software) gratuito para el desarrollo de aplicaciones móviles creado por Google en 2017. Su lenguaje de programación oficial es Dart desarrollado en 2011 para desarrollo web y reutilizado para Flutter.

Permite crear aplicaciones nativas multiplataforma (Android, iOS) para móvil, escritorio y web con un solo código lo que ahorra tiempo y recurso a las empresas.

Flutter está estructurado en capas, la más baja y su raíz está desarrollada en C++, es la capa que permite el renderizado gracias a la librería de Google Skia. Su segunda capa está desarrollada en su lenguaje principal Dart, es en sí el Framework, en ella, y gracias a sus librerías es donde se manejan las animaciones y la parte más importante de Flutter: los widgets.

En Flutter todo es un widget, y todos ellos están ligados de forma que el widget hijo está contenido en el padre. Una forma simple de verla sería poner como ejemplo una tarjeta, que sería el widget principal, dentro de ella habría un título, que sería otro widget, una imagen, y un pequeño texto. Todos los elementos serían widgets, y el widget padre la tarjeta siendo esta también un widget. Es esta distribución la que permite a Flutter y a nosotros mismos cambiar cada pequeño elemento de nuestra aplicación.

A esta distribución en forma de árbol se le llama árbol de widgets, y sus elementos o widgets principales són:

- MyApp: Es el primer widget de la aplicación y el que contendrá a los demás, el padre.
- MaterialApp: Widget de nivel superior que se encarga de buscar las rutas a las diferentes pantallas de una aplicación mediante una tabla de rutas o creando la ruta directamente.
- HomeScreen: Widget padre sobre el que se sustentan los widgets que la forman.
- Scaffold: Widget que implementa la estructura principal de diseño de Flutter. Se sitúa en el nivel más alto del árbol de widgets de la pantalla que lo contiene e implementa otros widgets básicos como AppBar y BottomNavigationBar.
- AppBar: Widget situado en la parte superior de la pantalla, se suele utilizar como barra de herramientas colocando botones y otros widgets.

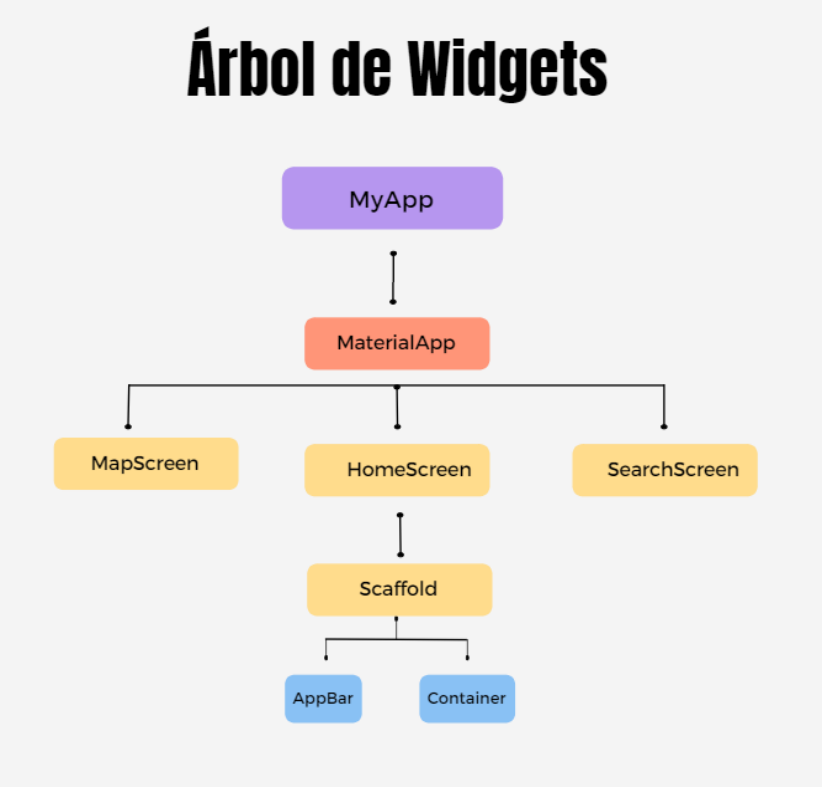


Figura 2. Árbol de widgets. (Fuente propia)

La gran diferencia con sus competencias como Framework principal es que Flutter permite un *hot reload*, que construye nuestra aplicación rápidamente y podemos ir viendo los cambios que hemos hecho y depurar errores al momento, esto facilita enormemente la creación de apps y lo hace muy cómodo. Otra de las grandes diferencias es como actúa con procesos nativos del móvil como puede ser la cámara o los sensores, mientras que otros *frameworks* necesitaban un paso entre medias, Flutter es capaz de comunicarse con procesos de Android nativo mediante *channels*.

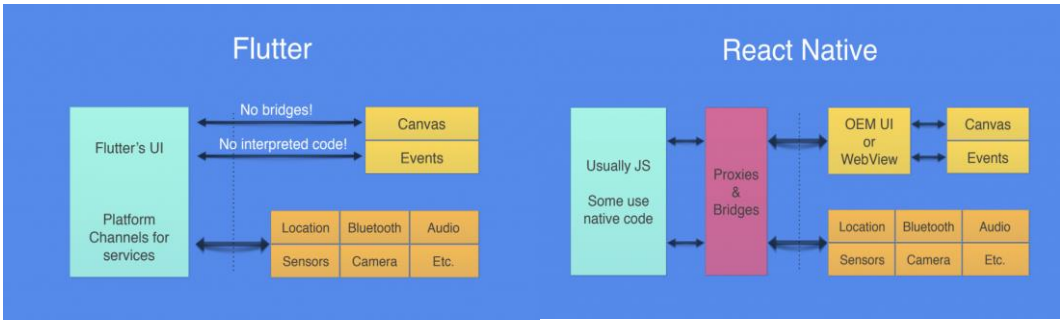


Figura 3. Nombre de la figura (Fuente: bbvanexttechnologies.com)

Esta y otras pequeñas ventajas son la causa del gran crecimiento que ha tenido este *Framework* los últimos años.

Desde su versión 2.9, Flutter implementa *null* Safety. En esta versión una variable no puede contener *null* si antes no lo hemos indicado explícitamente que puede contenerla, en el caso de Dart utilizando el signo de interrogación a continuación de la variable. ej int?.

En esta versión las variables no pueden ser *null* por defecto, lo que lleva al compilador a trabajar más rápido, ya que tiene por seguro cuando un valor puede ser *null* o no.

Y qué pasa con variables que sabemos que no son *null* pero no se inicializan al momento? Dart permite utilizar *late* para su inicialización, aunque esto no evita que estas variables no puedan ser *null* si no lo hemos especificado anteriormente.

2.5 Aplicaciones del tiempo

Las aplicaciones de móviles han ido evolucionando rápidamente a lo largo de los últimos años, por una parte, los procesadores de los móviles han ido creciendo y evolucionando tanto que pueden ser comparables a los de algunos ordenadores, y por otra parte las conexiones móviles de 3G, 4G y ahora 5G lo que permite que el intercambio de datos sea más rápido.

Eso ha llevado a los móviles a cambiar de usar simples widgets que mostraban la hora, o en el caso de widgets del tiempo a mostrar la temperatura, hora y lugar, a mostrar en aplicaciones, mapas en tiempo real, animaciones, gráficos y todo tipo de elementos que lo hacen a simple vista mucho más dinámicos.

Una de las malas costumbres que ha ido acompañando a este tipo de aplicaciones es la necesidad que tienen de redirigir al usuario a sus páginas web si este quiere recibir más información. En un primer momento esto era útil y lo óptimo si no querías gastar tus preciados datos móviles cuando las conexiones eran lentas, pero hoy en día ha dejado de tener sentido. Otra de las malas costumbres es la frecuencia con que estas aplicaciones y otras de diferente índole llenan la pantalla del usuario de banners y publicidad que hace la navegación por ellas incómoda y estresante. Por último, hay que destacar la falta de innovación en ellas, si bien una aplicación del tiempo lo que debe mostrar es exactamente eso, el tiempo y unos cuantos datos para dar información, siempre puede haber sitio para innovar en la forma en la que un usuario navega por ella, o la información es mostrada.

3 Metodología

En este apartado se detallarán todos los pasos seguidos para el desarrollo de la aplicación, explicación de las herramientas utilizadas, modelos de datos, la lógica detrás de las funciones de esta, librerías y todo lo necesario para poder replicar este modelo.

3.1 Funcionalidades principales

El primer paso para el desarrollo de una aplicación es tener claro el tema de ésta, de qué va a tratar nuestra aplicación, qué va a aportar al usuario y qué la va a hacer diferente o mejorar una ya existente. Como el tema ya está decidido de antemano el siguiente paso es decidir las funcionalidades de esta. Para ello se ha hecho un pequeño estudio observando y analizando aplicaciones ya existentes, empezando con la aplicación oficial de la API de la cual recibiremos los datos y alguna de las más populares del mercado.

Se ha revisado qué funciones son las más comunes, los comentarios y feedback que la gente publica en el store, lo cual es una buena fuente para observar los fallos de la aplicación o ver posibles mejoras que los usuarios ven útiles.

Los resultados han sido los siguientes:

Funcionalidades básicas que debería tener:

- Pantalla principal descriptiva, con la información necesaria (tiempo, temperatura, hora y ciudad)
- Foto descriptiva del tiempo y adaptativa
- Gráficas descriptivas de los datos
- Guardar tus ciudades favoritas
- Búsqueda con autoayuda para completar ciudades
- Tiempo por horas

Mejoras que no se han encontrado en la mayoría de las aplicaciones y sugerencias de comentarios:

- Idioma, la mayoría están en inglés
- Mapas interactivos
- Búsqueda de ciudad por mapa
- Aviso por notificaciones

Cosas a descartar:

- Anuncios y banners
- Funciones de pago
- Recogida de datos personales

3.2 Primer diseño de pantallas

Una vez decididas las funcionalidades de nuestra aplicación, el siguiente paso es hacer un pequeño boceto de las diferentes pantallas que esta tendrá, para hacerse una idea de cómo será la distribución de los elementos de la pantalla y dónde aplicaremos estas funciones. En este proyecto se utilizará Canva, una aplicación gratuita, que permite imitar

fácilmente una pantalla de móvil y crear perfectamente un croquis con sus elementos, aparte de editar en ellos, añadir imágenes y un largo etcétera.

3.2.1 Pantalla principal

La pantalla principal es la pantalla que se muestra una vez la aplicación está inicializada. Para que esta sea útil y clara deberá mostrar los datos más relevantes del tiempo actual, como son, la temperatura, la hora y la descripción del tiempo actual, que son la primera información que un usuario desea ver, junto con una imagen de fondo que vaya a juego con el tiempo. Además, incluirá la ciudad a la cual pertenecen estos datos y deberá indicar de forma clara si son los datos recibidos por la localización del móvil, evitando que el usuario añada por segunda vez la ciudad.

Desde la pantalla principal se ha de poder acceder a las diferentes funciones de la aplicación, por lo que tendrá iconos que dirigirán hacia las pantallas que incluyen estas funcionalidades como mapas o directamente tienen alguna funcionalidad como añadir favorito. Para ello se utilizara un widget muy utilizado en flutter, el IconButton, que hace precisamente eso, se trata de un widget que tiene como imagen un icono y que al detectar un cambio en el, una pulsación, este ejecuta la función de su propiedad OnTap(). En el caso que nos concierne dispondremos de tres IconButton: “mapa”, que dirigirá hacia la pantalla mapas, “buscar localización”, que nos dirigirá hacia la pantalla de búsqueda de ciudad y por último “favoritos”, que añadirá o borrará la ciudad de nuestra memoria. Estos iconos serán los hijos de un widget padre el cual será un Appbar, un widget que se usa principalmente como “barra de herramientas”, se coloca en la parte superior de la pantalla y suele mostrar, como en nuestro caso IconButton que ejecutan funciones o sirven para navegar por la aplicación.

Para hacer la aplicación más dinámica, la pantalla principal será desplazable. Esta y la pantalla de datos principal estarán unidas y serán navegables por ellas deslizando. Para ello Flutter dispone de varios widgets que hacen la función de *Scroll* o desplazamiento. Utilizaremos varios de ellos en esta aplicación, pero los más importantes son; el Listview, que muestra linearmente una lista de widgets hijos; el Gridview, que muestra widgets hijos en forma de cuadrícula, la cual podemos indicar la medida de esta; el SingleChildScrollView, que únicamente tiene un hijo el cual gana la propiedad de desplazarse; y los dos utilizados en esta pantalla, el customScrollView, que permite añadir varios efectos a nuestra AppBar entre ellos el de encogerse mientras hacemos scroll, el cual es el efecto que buscamos y el PageView, que nos permite tener varias “pantallas” iguales, lo cual nos permitirá construir y desplazarnos entre nuestras ciudades guardadas sin tener que ir navegando por la aplicación, de forma fluida y simple, ya que también permite añadir efectos de transición entre pantallas.

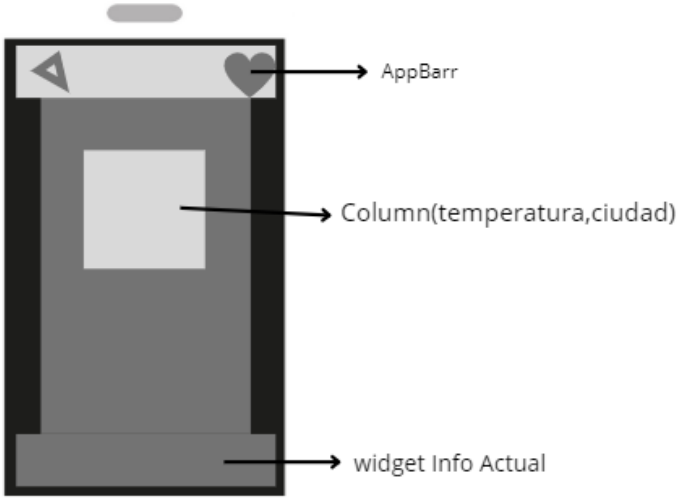


Figura 4. HomeScreen. (Fuente propia)

Como podemos ver en el boceto de la figura(nº4), El widget principal será un CustomScrollview, el cual tendrá hijos, en este caso los llamaremos slivers(, un AppBar o SliverAppBar con sus IconButtons, y a continuación en forma de pila o stack, la imagen y una column, qué es un widget que muestra a sus hijos en sentido vertical, lo que permitirá mostrar de forma ordenada toda la información del tiempo. El Stack o pila es un widget que permite disponer de otros widgets en diferentes niveles como si fuesen una pila, en este caso la imagen tendrá sobre ella toda la información.

Este Stack irá en el flexibleSpace de nuestro SliverAppBar que podemos ver en la figura(nº5) , el cual irá encogiendo a medida que hagamos scroll hasta convertirse en un AppBar, el cual mostrará únicamente los iconos y la ciudad y que irá fijo durante todo el desplazamiento.

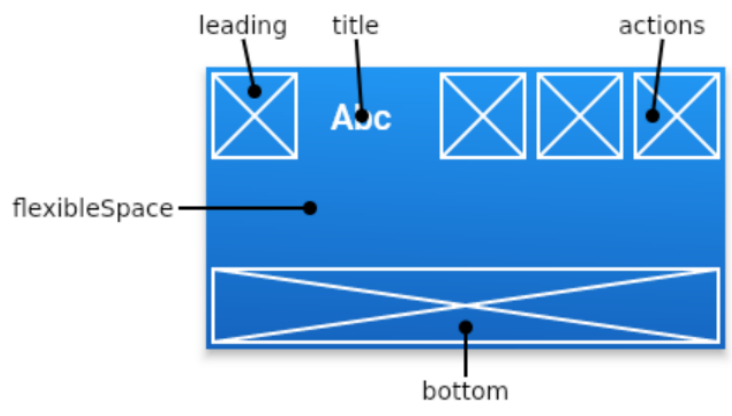


Figura 5.SliverAppBar(Fuente api.flutter.dev)

3.2.2 Pantalla de datos principal

Esta pantalla será la que contenga toda la información actual, datos de viento, lluvia, temperatura semanal y por horas. Irá en la misma pantalla que la principal por lo que será un hijo o Sliver del CustomScrollView.

Se engloban todos los Slivers en un SliverList que permite aplicar el efecto de Scroll en todos los elementos en conjunto, lo que hará que la animación no se interrumpa. En caso de usar Sliver independientes, puede que el elemento siguiente sea un Listview con desplazamiento vertical lo que haría que, dependiendo de donde se haga el scroll, el elemento que se desplace sea el Sliver o bien el ListView.

Esta SliverList estará dividida en widgets separados dependiendo de qué información proporciona, lo que hará que podamos prescindir de ellos si la situación lo requiere (Puede que no llueva ese día por lo que ese widget no aparecerá, a diferencia de los días de lluvia). También este método hace los widgets más reutilizables y evita la repetición de código, muy importante en cualquier programación pero que, en móviles y sobre todo en Flutter, es esencial, ya que utilizaremos los mismos widgets personalizados en diferentes pantallas o incluso otras aplicaciones.

Ambos widgets, tiempo hora y semana serán gestureDetectors que cambiarán a sus respectivas pantallas. GestureDectector es un widget que proporciona a su hijo la propiedad de detectar gestos, en otras palabras, el widget hijo será capaz de detectar un toque, una presión prolongada o bien un Scroll lateral, y ejecutar una acción en respuesta. En este caso la acción será la navegación por pantalla, o bien a la pantalla de información por horas o bien a la de días, ambas mediante un solo toque, “onTap”.

A continuación se mostrarán los widgets personalizados que mostraran la información principal, como la lluvia, la sensación térmica… Estos widgets son un conjunto de Container con padding y decoración para modificar su forma, como hijo una Column con los elementos (icono , texto, texto). todos estos widgets personalizados formarán parte de un Wrap, un widget que permite a sus hijos mostrarse a continuación del siguiente, y en caso de no tener suficiente espacio, adaptarse en la siguiente linea, perfecto en este caso ya que no todos los elemento se mostrarán siempre, así que el espacio no será fijo , se podrá ir adaptando a medida que los elementos van apareciendo.

La información que contiene porcentajes, como son nubes y humedad, se mostrarán con gráficas circulares y una pequeña animación al aparecer. Para ello se utilizara una librería externa percent_indicator que permite crearlos fácilmente, con el widget PercentCircularIndicator. Este widget permite crear tu propio gráfico circular con los datos que se le proporcionen, añadirle animación y modificar las propiedades de las mismas.

Por último, para mostrar las temperaturas semanales de forma llamativa se utilizará la librería fl_Chart, que permite crear todo tipo de gráficas, en este caso una gráfica de barras con la información de temperatura máxima y mínima diaria superpuesta.

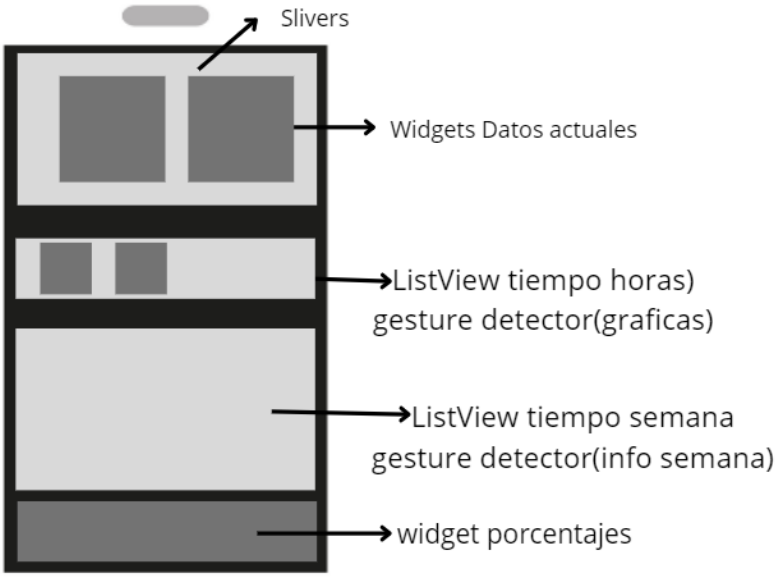


Figura 6.HomeScreen(Fuente propia)

3.2.3 Pantalla de gráficas datos por horas

Esta pantalla mostrará los datos por horas de lluvia, temperatura, sensación térmica y presión en forma de gráficas.

Column, que tendrá como hijos, cada una de las gráficas hechas con la librería fl_chart de Flutter, en este caso las gráficas lineales, e indicarán su valor al ser tocadas. También tendrá un ListView como índice para estas gráficas con las horas y un icono del tiempo actual.

Como Appbar simplemente tendrá el nombre de la ciudad a la que corresponde los datos y un IconButton de vuelta.

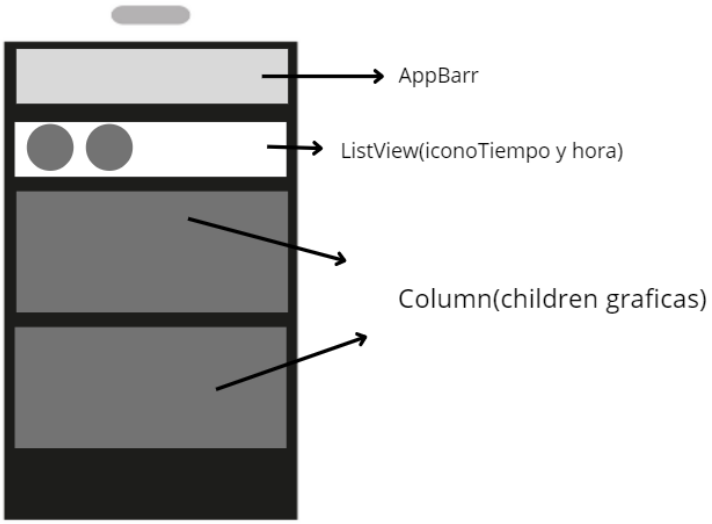


Figura 7.HorasInfoScreen(Fuente propia)

3.2.4 Pantalla de fichas con los datos diarios principales

Esta pantalla mostrará en forma de fichas el tiempo para cada día, el pronóstico será de 14 días. Los datos principales serán, la probabilidad de lluvia, nubes y la temperatura.

El widget principal será un Listview de Text como cabecera junto con un Widget personalizado el cual consta de un Container, como fondo una imagen en relación al tiempo general del día actual y como hijo un Row con dos Columns que tendrán los datos perfectamente alineados.

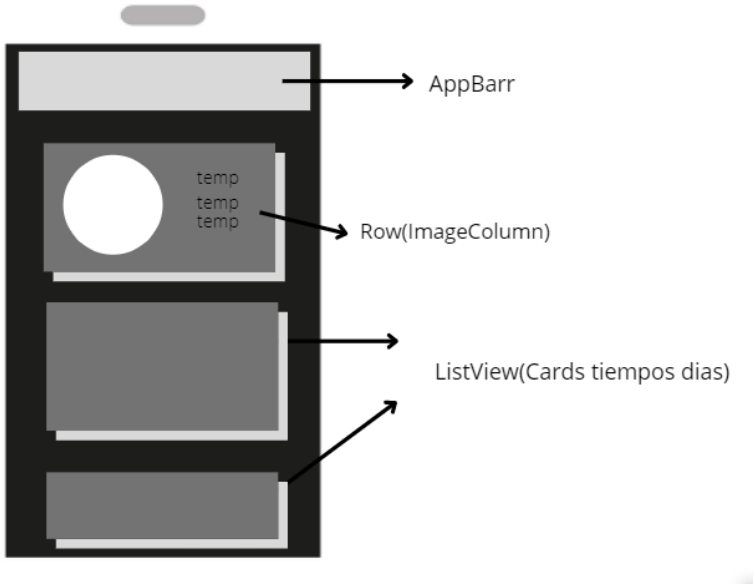


Figura 8. DiasInfoScreen(Fuente propia)

3.2.5 Pantalla de búsqueda

Es una de las pantallas que más funciones tiene, y estará dividida en dos.

La primera pantalla mostrará el mapa de GoogleMaps, en el cual podremos seleccionar la ciudad y añadirla a la aplicación, para ello utilizaremos la librería google_maps_flutter y un stateful widget, ya que es necesario repintar el mapa y otros elementos a medida que van variando los valores.

Esta pantalla no usará Appbar, en sustitución utilizará una Column con un SafeArea como widget padre, este widget evita que nuestro widget se vea tapado por el *notch* y la barra de notificaciones de los distintos móviles, cosa que un AppBar incluye implícitamente.

La Column estará formada por un Row con GestureDetector compuesto por un Icon y un Text. Como valor inicial tendrá la ciudad que marque nuestra localización. Una vez vayamos interactuando con el mapa, este irá cambiando de valor dependiendo de si la localización es exacta o es un punto sin datos. El GestureDetector activará un ShowSearch, una función que crea una nueva pantalla, el SerachDelegate, un Widget formado por un SearchField. Este StatelessWidget (su estado se maneja con un controller que comprueba el valor hay en ese momento, si se ha escrito) está formado por un container en el cual escribiremos la búsqueda deseada y gracias a su controller modificará los valores de un ListView con los resultados. El SearchField también constará de dos IconButton: “volver” y “borrar” .

Una vez seleccionada alguna de las sugerencias o ninguna, ShowSearch devuelve un resultado y a partir de este, el Text y mapa cambiará de valor.

El mapa de GoogleMaps será el segundo Widget de la Column. Como valor inicial mostrará la ubicación actual del usuario. El mapa será interactivo y cambiará el valor del Text anterior dependiendo de la posición que se toque. Si se escoge una posición nula, el texto indicará que es una ubicación imprecisa.

Por último, la pantalla contará con un TextButton para añadir la ciudad indicada a nuestra lista y ejecutar las funciones correspondientes asignadas como CurrentWeather y HourlyWeather. En caso de tener un valor nulo, el TextButton se mantendrá deshabilitado.

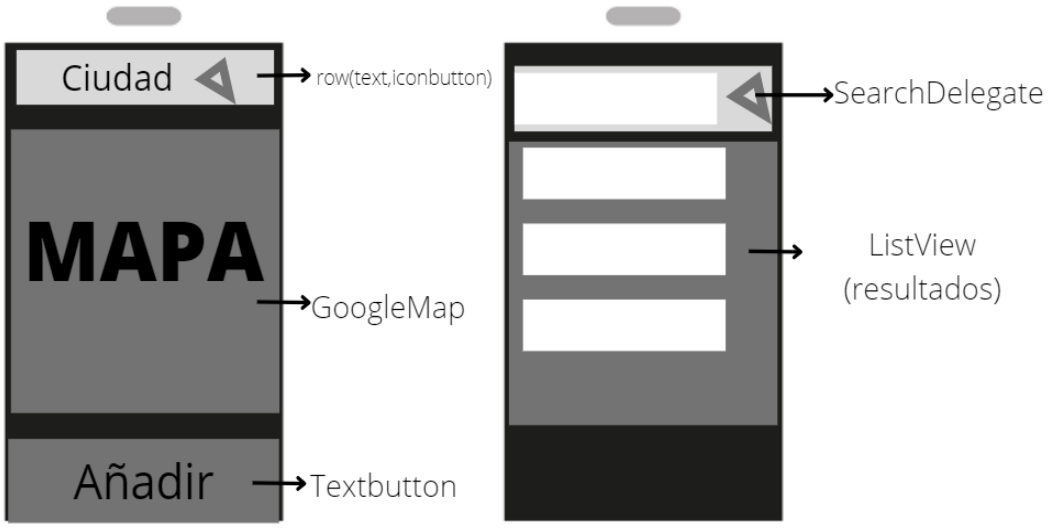


Figura 9. SearchScreen(Fuente propia)

3.2.6 Pantalla de mapas interactivos

Esta pantalla mostrará los mapas interactivos de temperatura, lluvias, nubes, viento y precipitación actuales.

Para la creación de este mapa se usará la librería Flutter_map. Este mapa, a diferencia del ofrecido por Google, permite añadir layers o capas personalizadas, que son exactamente los datos que ofrece la API.

La pantalla consta del mapa y una BottomNavigationBar que permitirá cambiar el layer del mapa que se desea visualizar. Este widget es parecido a la AppBar, ya que se suele utilizar en conjunto con el Scaffold de la aplicación, pero a diferencia de este se sitúa al inferior de la pantalla. Consiste en varios elementos, como mínimo dos, Icons y Text, que se distribuyen a lo largo del widget principal. Estos elementos son interactivos, como los IconButtons y permite navegar por diferentes pantallas o modificar widgets ya existentes, ya que como respuesta envía el valor o posición del elemento que se está apretando.

En este caso su función es cambiar el *layer* superior del mapa mediante un gestor de estado, Provider, que será explicado detalladamente más adelante. Simplificadamente, se controlará o escuchará la posición de la BottomNavigationBar mediante Provider y cada vez que haya un cambio, este informará y repintará la pantalla con el nuevo *layer*.

Sobre el mapa se pintara utilizando un CustomPainter, un widget que permite pintar sobre un canvas (el tapiz con el tamaño que indiquemos), la leyenda del *layer*, en forma de la

barra típica de colores y el título que identifique el mapa actual, el control del estado de este widget se controlará mediante el mismo Provider.

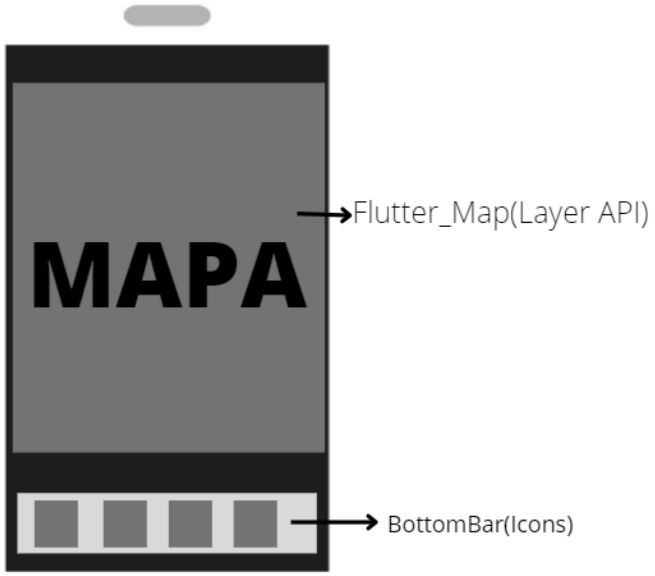


Figura 10. LayerMapScreen(Fuente propia)

3.3 Modelo de datos

Es la parte más importante de todo código. Se describe cada clase, el tipo de datos que contienen y cómo están relacionados entre ellos. A continuación, se detallarán los diferentes modelos y su obtención:

3.3.1 Modelos desde API

Antes de describir cada clase es importante hacer un estudio de la API que nos proporcionará estos datos. Hay que entender su funcionamiento, qué tipo de datos recibimos, en qué momento estos datos pueden ser nulos o no, como hacer las llamadas a esta API para así hacer el modelo lo más ajustado posible, evitar errores o saber de dónde provienen y por qué.

La API que se utilizará será la proporcionada por OpenWeather. Se trata de una API relativamente gratuita que requiere una API *key* para su uso.

En el caso de este proyecto utilizaremos la versión de estudiante que ofrece mayor número de llamadas, además del uso de *layers* para los mapas interactivos.

Distribución de datos y llamadas:

- Pantalla principal:

  Para la pantalla principal en un primer momento se barajó la posibilidad de usar la llamada a currentWeatherData, que ofrece el tiempo actual de la ciudad indicada

pero no ofrece el nombre de esta ni información horaria o semanal, por lo que se requeriría hacer varias llamadas extra. Un *parsing* de coordenadas a nombre de ciudad y llamadas a tiempo por horas y por días, lo que es un gasto excesivo de datos para el usuario y tiempo que haría poco fluida la aplicación.

La opción escogida será el uso de la OneCall API. Proporciona la información semanal y horaria simplificada; en la misma llamada incluye el *parsing* de coordenada a nombre de la ciudad y está incluida en el plan de estudiante.

One Call Model

Nombre	Descripción	Tipo
lat	Latitud	double
lon	Longitud	double
timezone	Zona horaria	String
timezoneOffset	Zona horaria offset	int
current	Tiempo actual	Class
hourly	Tiempo por horas simplificado	List<Class>
daily	Tiempo por dias simplificado	List<Class>

Tabla 1. Modelo datos OneCall (Fuente propia)

Current Model

Nombre	Descripción	Tipo
dt	Hora actual formato	String
temp	Temperatura actual	double
feelsLike	Sensación Térmica	double
pressure	Pressión	int
humidity	humedad	int
dewPoint	Punto de rocio	double
uvi	Indice UV	double
clouds	Nubes	int
visibility	Visibilidad	int
windSpeed	Velocidad del viento	double
weather	Datos del tiempo	Class o null
pop	probabilidad de lluvia	Class o null
rain	Datos lluvia	Class o null
snow	Datos nieve	Class o null

Tabla 2. Modelo datos Current (Fuente propia)

Daily Model

Nombre	Descripción	Tipo
dt	Hora actual formato	String
Temp	Datos Temperatura actual	Class
FeelsLike	Datos Sensación Térmica	Class
pressure	Pressión	int
humidity	humedad	int
dewPoint	Punto de rocio	double
uvi	Indice UV	double
clouds	Nubes	int
visibility	Visibilidad	int
windSpeed	Velocidad del viento	double
weather	Datos del tiempo	List<Class weather>
pop	probabilidad de lluvia	double
rain	Datos lluvia	num o null
snow	Datos nieve	double o null

Tabla 3. Modelo datos Daily (Fuente propia)

FeelsLike Model

Nombre	Descripción	Tipo
day	Sensación día	double
night	Sensación noche	double
eve	Sensación tarde	double
morn	Sensación Mañana	double

Tabla 4. Modelo datos FeelsLike (Fuente propia)

Temp Model

Nombre	Descripción	Tipo
day	Temperatura día	double
night	Temperatura noche	double
eve	Temperatura tarde	double
morn	Temperatura mañana	double
max	Temperatura máxima	double
min	Temperatura mínima	double

Tabla 5. Modelo datos Temp (Fuente propia)

Snow Model

Nombre	Descripción	Tipo
the1h	Nieve en la última hora	Double o null

Tabla 6. Modelo datos Snow (Fuente propia)

Rain Model

Nombre	Descripción	Tipo
the1h	Lluvia en la última hora	Double o null

Tabla 7. Modelo datos Rain (Fuente propia)

- Pantalla de gráficas datos por horas:

  Se utilizará la llamada Hourly Forecast 4 days.

Horas model

Nombre	Descripción	Tipo
cod	Codigo de la información	String
list	Lista de informacion horaria	List<class ListElement>
city	Datos de la ciudad	Class City

Tabla 8. Modelo datos HourlyForecast (Fuente propia)

City model

Nombre	Descripción	Tipo
id	Id de la ciudad	int
name	nombre de la ciudad	String
coord	Cordenadas latitud y longitud	Clase Coord
country	Pais de la ciudad	String
timezone	Zona horaria	int
sunrise	Amanecer	int
sunset	Puesta de sol	int

Tabla 9. Modelo datos City (Fuente propia)

Coord Model

Nombre	Descripción	Tipo
lat	latitud	double
lon	longitud	double

Tabla 10. Modelo datos Coord (Fuente propia)

ListElement model

Nombre	Descripción	Tipo
dt	Hora correspondiente form	int
main	datos principales del tiemp	Class Mainclass
weather	datos descriptivos del tiem	Class Weather
clouds	información de las nubes	Class Cloud
wind	información del viento	Class wind
visibility	visibilidad	int
pop	probabilidad de lluvia	double
rain	informacion de la lluvia	Class rain o null

Tabla 11. Modelo datos ListElement (Fuente propia)

MainClass model

Nombre	Descripción	Tipo
temp	temperatura	double
feelsLike	sensación térmica	double
tempMin	temperatura mínima	double
tempMax	temperatura máxima	double
pressure	pressión	int
humidity	Humedad	int

Tabla 12. Modelo datos MainClass (Fuente propia)

Clouds model

Nombre	Descripción	Tipo
all	porcentaje de nubes	int

Tabla 13. Modelo datos Clouds (Fuente propia)

Wind model

Nombre	Descripción	Tipo
speed	velocidad viento	double

Tabla 14. Modelo datos Wind (Fuente propia)

- Pantalla de pronóstico días:

  Se utilizará la llamada Daily Forecast 16 days.

SixteenDays Model

Nombre	Descripción	Tipo
City	información de la ciudad	Class cityModel
list	lista de información diaria	List< class ListElementSixteen

Tabla 15. Modelo datos DailyForecast (Fuente propia)

ListElementSixteen Model

Nombre	Descripción	Tipo
dt	Hora correspondiente formato	int
temp	información temperatura	Class temp
feelsLike	información sensación termica	Class feelsLike
pressure	pressión	int
humidity	humedad	int
weather	información tiempo	class Weather
speed	velocidad viento	double
clouds	porcentaje nubes	int
pop	probabilidad lluvia	double
rain	cantidad lluvia	double o null
snow	cantidad nieve	double o null

Tabla 16. Modelo datos ListElementSixteen (Fuente propia)

- Pantalla de mapas interactivos:

  Se utilizará la llamada Weather Maps 2.0. Esta llamada proporciona 15 *layers*, en este caso solo utilizaremos 5 (lluvia, viento, temperatura, nubes y lluvias acumuladas). Para este proyecto no será necesaria ninguna clase para guardar los datos recibidos ya que se recibe una imagen que se colocará encima de un mapa real.

Una vez decididas qué llamadas se van a hacer, es importante comprobar cómo funcionan, comprobar los datos recibidos y hacer varias pruebas con ellas. Un software muy útil es Postman, que permite hacer llamadas a distintos API, guardarlas, cambiar los parámetros que van en el cuerpo de la llamada y ver el archivo JSON recibido de una manera fácil e intuitiva.

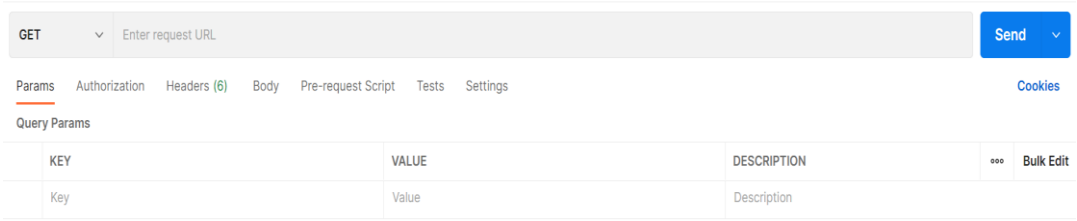


Figura 11. Postman Call Bar

El primer paso es introducir el cuerpo(*body*) de la llamada a la API dejando de lado los parámetros. A continuación, se rellenarán los parámetros obligatorios API key (appid) y las coordenadas y, por último, los opcionales como pueden ser el idioma y las unidades que por defecto de la API no es el sistema métrico internacional.

	KEY	VALUE
☑	lat	41.49064359025308
☑	lon	2.1356232423292703
☑	appid	8e00b4a7cb1f2ecb996a60a92f20f33b
☑	lang	es
☑	units	metric

Figura 12.Postman parámetros

Para finalizar la información será recibida en el formato que deseemos, en este caso Json, el cual será guardado y utilizado para la creación de las clases o modelos con Quicktype.

Quicktype es una aplicación gratuita, que permite crear modelos de datos en diferentes lenguajes de programación proporcionándole un Json. Si bien es de mucha ayuda, Dart en su versión 2.9 introdujo el *null* Safety. En esta versión una variable no puede contener *null* si antes no lo hemos indicado explícitamente que puede contenerla, en el caso de Dart utilizando en signo de interrogación a continuación de la variable int?. Esto evita errores con valores nulos por lo que en base al estudio de la API anterior habrá que modificar los datos de los modelos para evitar estos errores.

3.3.2 Modelos de datos extra

Además de los modelos mencionados anteriormente, la aplicación guardará las diferentes ciudades, como favoritos del usuario, por lo que será necesaria otra clase en particular.

En este caso un formato simple de Ciudad - Coordenadas que permite ser guardado en un Json y ser guardado y leído fácilmente desde las preferencias de usuario.

SharedPreferences Model

Nombre	Descripción	Tipo
name	nombre ciudad	String
geo	Coordenadas	List<String>

Tabla 17. Modelo datos SharedPreferences (Fuente propia)

Properties Model

Nombre	Descripción	Tipo
name	nombre busqueda	String
country	País	String o null
region	Región	String o null
locality	localidad	String o null
label	calle o punto de busqueda	String
countryA	Abreviatura País	String o null

Tabla 18. Modelo datos Properties (Fuente propia)

Por último, el modelo de autocomplete, ya que la API de ayuda envía modelos de datos de los resultados y se han de guardar los datos más relevantes de ellos.

AutoComplete Model

Nombre	Descripción	Tipo
features	lista con todas las ciuades	List< class Feature>

Tabla 19. Modelo datos AutoComplete (Fuente propia)

Feature Model

Nombre	Descripción	Tipo
geometry	Coordenadas	class Geometry
properties	Información ciudad	Class Properties

Tabla 20. Modelo datos Feature(Fuente propia)

Geometry Model

Nombre	Descripción	Tipo
coordinates	coordenadas	List<double>

Tabla 21. Modelo datos Geometry (Fuente propia)

Según avance el desarrollo puede que se vayan necesitando crear nuevas clases o modelos que vayan relacionados con los descritos anteriormente, como podría ser el caso del cambio de ciudad a coordenadas si hay estos datos se reciben externamente o algún modelo de datos que requerimos para pasar información internamente. La implementación de estos modelos se explicará a lo largo del desarrollo cuando estos elementos sean requeridos.

3.4 Gestor de estado

Cuando creamos una aplicación con Flutter la complejidad de esta puede hacer variar el modo en el que los datos se modifican en las diferentes pantallas. En aplicaciones simples, por ejemplo, una aplicación de compras en la que la pantalla principal muestra los artículos comprados y la segunda pantalla se seleccionan y compran estos productos, los datos de compra solo se reflejan en la pantalla principal por lo que la gestión del estado es simple; una vez pasas a la pantalla anterior esta se pinta con los nuevos datos. Por el contrario, en aplicaciones complejas puede que unos mismos datos estén reflejados en diferentes

pantallas a lo largo de la aplicación y la modificación de uno de estos datos se deba reflejar en todas las pantallas.

Una aplicación en Flutter puede tener muchos niveles, y un widget en el nivel inferior puede necesitar información del nivel superior. Una de las funciones de estos gestores de estados consiste en permitir a los diferentes niveles acceder a unos mismos datos en el momento necesario sin necesidad de pasarlos como parámetros, lo que implica que todos los widgets que estén escuchado a este gestor de estado y usando sus datos, si estos se modifican en cualquier parte o momento, el dato se modificará en todas partes. Gracias a estos gestores, podemos evitar usar StatefulWidgets innecesarios que repintan toda la pantalla, y únicamente volver a pintar los widgets independientes a los que el gestor proporciona datos, ya que Flutter es capaz de identificar qué widgets del árbol de widgets son los que están escuchando a este gestor.

En caso de que estos datos sean de gran tamaño o externos, ya vengan de documentos Json en *assets* o bien de peticiones http a API, entra en juego un concepto muy importante en programación para móviles y muchos otros tipos: la programación asíncrona. Cuando hacemos una petición http, puede que estos datos tarden en llegar y durante este proceso una aplicación tiene que estar en uso, no se puede permitir parar. Aquí interviene la programación asíncrona, continuando el programa y dejando en espera los datos, y una vez lleguen, hacer todo el proceso necesario con ellos.

En este momento es cuando el gestor de estado toma protagonismo, llamado a Flutter para que repinte los widgets que dependen de estos datos una vez los ha recibido y dejando el resto sin repintar.

Existen muchos gestores de estado en Flutter: Redux, Bloc, GetX, Provider, cada uno con un funcionamiento distinto y de complejidad variable. En el caso de este proyecto se usará Provider, un gestor de estado simple y potente con mucho soporte por parte de Flutter y con las máximas calificaciones en pub.dev, el repositorio de paquetes de Flutter.

4 Planteamiento y decisión sobre soluciones alternativas

En este apartado se detallarán las diferentes alternativas barajadas durante el desarrollo de este proyecto

4.1 Guardado de datos

En un primer momento se barajó la posibilidad de guardar los datos del usuario, las ciudades favoritas y sus coordenadas en un base de datos externa. Se pensó en Firebase, ya que es gratuito y permite la lectura y escritura en tiempo real además de tener un servicio de autenticación firebase auth muy útil para aplicaciones. Se desechó esta idea porque hacía que la velocidad de ejecución de la aplicación variará demasiado según los datos móviles de los que se disponen. Además, la aplicación no desea guardar datos personales del usuario como sería una cuenta, por lo que se buscó una alternativa con menos coste de tiempo.

La segunda alternativa fue guardar estos datos en SQLite, una base de datos multiplataforma perfecta para Flutter, pues tiene su librería Sqflite actualizada y, en comparación con Firebase, necesita menos recursos para el intercambio de datos. La desventaja fue que aun siendo menor la necesidad de recursos, los datos aún eran necesarios y dependían de la conexión.

Como alternativa final se eligió guardar los datos en las preferencias de usuario del móvil, que guarda para cada aplicación y se borran una vez eliminada la misma. Desaparecía la desventaja de necesitar datos móviles para recibir los datos y, al no ser datos excesivamente grandes, 20 o 30 ciudades con sus coordenadas, incluso si fuesen más, serían insignificantes y podrían guardase en un archivo Json, además de no recibir datos del usuario, un punto en el que suele ser reticentes.

4.2 Mapas

Una de las funciones más atractivas de la aplicación es el mapa interactivo con sus diferentes *layers* o capas, que muestra de forma vistosa el tiempo actual mundial, y que permite al usuario interactuar con él, cambiando la vista o bien moviéndose a lo largo del mapa y curioseando.

Con el estudio de la API, se vio que la información recibida es una capa compatible con diferentes mapas. En un primer momento se pensó en utilizar GoogleMaps para Flutter, ya que era utilizado en otra parte de la aplicación y la implementación sería más sencilla. Se vio que el mapa de Google no permite utilizar capas personalizadas sobre él y por lo consiguiente impide modificarlo con los datos recibidos, así que se decidió buscar librerías adaptadas a Flutter que si pudiesen modificar sus capas.

La solución final fué utilizar la librería flutter_maps, una implementación de Leaflet, una librería open-source de mapas interactivos en javaScript y con la cual la API se conecta perfectamente, permitiendo además de superponer la capa sobre el mapa, editar el rango de colores, la transparencia y otros aspectos.

4.3 Notificaciones

Tanto las notificaciones como el uso de la cámara de un móvil, sus sensores o acceder a sus archivos internos son acciones en el que el programador tiene que interactuar con la capa Android o IOs del móvil, lo que equivale a escribir lenguaje kotlin o bien swift para hacer uso de ellas.

Este proyecto no abarca el aprendizaje de estos lenguajes ya que las funciones para hacer uso de ellas son de un nivel bastante complejo por ello se buscó una solución intermedia para poder hacer uso de notificaciones, ya que es muy común que todo tipo de aplicaciones las tengan y den información al usuario. En el caso de este proyecto, puede ser muy útil el recibir la información del tiempo actual en forma de notificación si la aplicación está en segundo plano.

La solución pasa por usar un libreria de Flutter flutter_local_notifications que reduce el uso del lenguaje kotlin y swift, ya que la creación de *channels* (canales con los que la aplicación se comunica con el sistema) los hace la propia librería. Tanto la lógica de cuando estas notificaciones se activan y la configuración de unos pequeños archivos escritos en kotlin y swift para modificar algunos aspectos como la imagen de la notificación sí van a nuestro cargo.

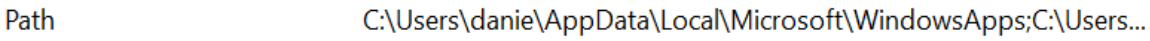
5 Desarrollo de la solución o soluciones escogidas

5.1 Instalación

El proyecto comienza con la instalación de los tres principales componentes, Dart, Flutter y el SDK de Android Studio. Para ello se seguirá la guía desarrollada en su página principal docs.flutter.dev. Los pasos a seguir serán:

1. Descarga e instalación del SDK de Flutter con el cual ya se incluye Dart. La instalación es sencilla, únicamente se han de seguir los pasos que se van indicando.

2. Una vez instalado se ha de editar las variables de sistema para poder utilizar Flutter desde el terminal. Para ello buscaremos "variables de entorno" en la barra de búsqueda de nuestro sistema. Seguidamente iremos a "opciones avanzadas/variables de entorno" y actualizaremos el *path* con el correspondiente a Flutter en nuestro sistema.

3. Instalaremos el SDK de Android Studio desde su página y Visual Studio Code.

4. Comprobaremos si está todo correctamente instalado con el comando flutter doctor. En caso de algún fallo habrá que repetir los pasos anteriores. Sí todo está bien y la única advertencia es la falta de un dispositivo conectado pasaremos al último paso.

5. Por último, hay 2 opciones, usar un móvil como dispositivo para ejecutar nuestro código o bien un emulador. En este caso se utilizará un dispositivo móvil, ya que consume menos recursos del ordenador y podemos interactuar más fácilmente con él. En caso de escoger un emulador, su creación se hará con Android Studio, "AVD manager/Create Virtual Device" y seguir los pasos.

5.2 Creación del proyecto

Desde nuestra consola, nos situaremos en el directorio en el que guardaremos todo nuestro proyecto y ejecutaremos el comando "flutter create nombre_de_la_app", al cabo de unos segundos, Flutter nos habrá creado todo nuestro proyecto para poder trabajar en él. Desde el proyecto creado ejecutaremos "code." para abrir nuestro proyecto en Visual Code Studio y empezar el desarrollo de nuestra aplicación.

El primer paso antes de empezar a escribir cualquier línea de código será la configuración de nuestro proyecto. Crearemos las carpetas necesarias para mantener una legibilidad del código. Estas carpetas deberían ser claras para identificar lo que contienen. Nuestro proyecto lo forman estas carpetas:

- Screens: Guardaremos todas las pantallas de nuestra aplicación
- Theme: Los temas como el estilo de letra, tamaño, y otros detalles que se repetirán a lo largo de la aplicación y queremos que sean iguales en todas nuestras pantallas.

- Widgets: Nuestros widgets personalizados que vayamos a usar en nuestras pantallas, con lo que el código de estas quedará mucho más limpio y además los hace reutilizables.
- Assets: Guardaremos imágenes o archivos que queramos utilizar durante la ejecución de la aplicación y queremos que sean accesibles en todo momento. Para su uso requerirán algunas modificaciones en otros archivos.
- Providers: Nuestros gestores de estados

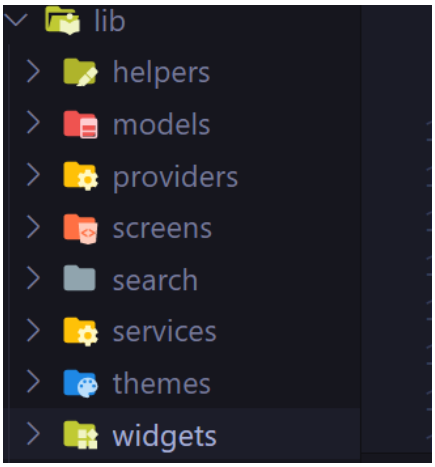


Figura 13. Carpetas proyecto(Fuente propia)

Como en todo código que use funciones, clases o cualquier librería que no esté incluida en ese archivo, estas deben ser importadas. Para evitar que nuestro código se llene de importaciones, es una buena práctica crear un archivo que haga de nexo y haga estas importaciones y exportaciones. Estos archivos se crearán con el nombre de cada carpeta y se hará un *import* y *export* de cada uno de los archivos que estén en esa carpeta. En el caso de *Screens*, este archivo se llamará screens.dart e incluirá la importación y exportación de cada una de las *screens*.

A continuación habrá que modificar el archivo main.dart. Este archivo contiene la función main que ejecuta nuestra aplicación (crea nuestro árbol de widgets) con runApp(). Primero se eliminará todo el código creado por defecto al crear el proyecto dejando únicamente la función runApp() y como parámetro, nuestro primer widget, al que comúnmente se le llama MyApp. MyApp como hijo tendrá un MaterialApp, un widget de nivel superior que se encarga de buscar las rutas a las diferentes pantallas de nuestra aplicación en caso de que creemos un navegador, en caso contrario para pasar de una pantalla a otra tendremos que crear las rutas manualmente cada vez que deseemos navegar.

En el caso de este proyecto crearemos el navegador en el MaterialApp para facilitar la lectura y el orden. Por el momento únicamente crearemos la ruta hacia la pantalla principal a la que llamaremos HomeScreen y se llamará *home*. Por norma, nuestro navegador deberá incluir una ruta por defecto al inicializarse, la cual será nuestro homeScreen. El código de nuestro widget MyApp quedará tal como podemos observar en la figura(nº14), también se puede observar cómo gracias a nuestro navegador, añadir nuevas pantallas o rutas es sencillo y fácilmente escalable.

```dart
class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      initialRoute: 'home',
      routes: {
        'home': (context) => const HomeScreen(),
      },
      theme: AppTheme.darkTheme,
    );
  }
}
```

Figura 14. Widget MyApp(Fuente propia)

Por último, se subirá el repositorio a Github utilizando git para mantener un control de versiones.

5.3 Pantalla principal, pantalla gráficas y pronóstico días

El desarrollo de las tres pantallas iniciales se estructurará en tres partes; creación del esqueleto con datos inventados para ver su distribución, lectura de datos reales y por último diseño final, el cual se puede ir modificando a medida que se creen las pantallas restantes para que sean homogéneas, lo cual ayudará la creación de un *theme* común.

La pantalla principal como se decidió en los bocetos iniciales estará formada por una pantalla con *scroll*, en la cual, la portada principal se minimizará mientras nos vamos desplazando hacia abajo quedando únicamente el appBar.

La pantalla de gráficas, sus widgets principales necesitan muchos datos por lo que se creará el archivo únicamente en el paso uno y se pasará directamente al paso dos.

La pantalla de pronóstico por días estará formada por widgets personalizados, en forma de tarjeta.

5.3.1 Esqueleto

El primer paso es crear el archivo que contendrá nuestra pantalla principal. Se llamará home_screen.dart y será en un primer momento un stateless widget, el cual no necesitará estado ya que se controlará con un provider. A medida que vayamos añadiendo funcionalidades a la aplicación, este tendrá que pasar a ser un widget con estado ya que el uso de notificaciones requiere de un stateful widget.

Este statelessWidget se llamará HomeScreen tal como es usado en nuestras rutas del main.dart. Haremos la importación y exportación en nuestro archivo nexo screens.dart y lo importamos en el Main. Ahora podemos ver la utilidad de este archivo ya que, a medida que añadamos pantallas, el único archivo a modificar será screens.dart.

El widget principal es un CustomScrollView en el cual su primer Sliver o hijo será nuestro widget SliverAppBar retráctil. Este widget lo llamaremos CustomAppBar, un nombre descriptivo para su funcionalidad y será nuestro primer widget propio en el cual crearemos su archivo en la carpeta Widgets. Como pasaba con las pantallas, usaremos su archivo nexo para ahorrarnos importaciones, con lo cual habrá que hacer su debida importación y exportación en widgets.dart.

5.3.1.1 CustomAppBar

Este widget será stateless y retornará el SliverAppBar. Primero modificaremos sus propiedades de diseño:

- floating: *False*, Quitamos el efecto de que sobresalga y la pequeña sombra.
- pinned: *True*, Con esto dispondremos de la AppBar y sus funciones durante todo el scroll.
- expandedHeight: A gusto, el espacio que ocupa la SliverAppBar completa.
- centerTitle: True, centramos el título, en este caso el nombre de la ciudad.
- shape: La forma que tendrá la SliverAppBar encogida. Usamos un RoundedRectangleBorder con las esquinas inferiores redondeadas.

Una vez elegida la forma que nuestra SliverAppBar tendrá cuando esté retraída se modificarán los elementos que aparecerán en ella. La estructura de una SliverAppBar es la mostrada anteriormente figura(nº15)

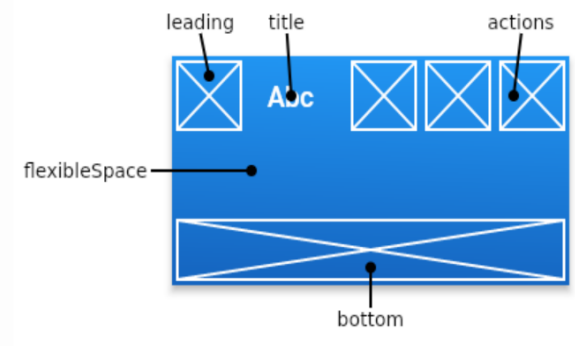


Figura 15. SliverAppBar (Fuente api.flutter.dev)

- Leading: IconButton de mapa, crearemos su función onPress para que nos vuelva a dirigir a esta pantalla y la modificaremos una vez la pantalla mapas esté creada.
- Actions: Esta propiedad tiene como hijo una lista de widgets, en este caso colocaremos dos IconButton, uno será una lupa que nos llevará a la pantalla de búsqueda, como en el caso del mapa la creación de esta aún no se ha llevado a cabo por lo que la implementamos para que nos redirija a la homeScreen. Y un corazón para favoritos, el cual deberemos aplicar una lógica.
    a. Aparecerá si la ciudad no coincide con tu ubicación actual.
    b. Aparecerá y estará pintado si está guardado en tus favoritos
    c. Aparecerá y no estará pintado si no está guardado en tus favoritos

  Como todavía no tenemos nuestro provider creado, la dejaremos como null

Por último faltaría completar el flexibleSpace, en el cual también está incluido el título. El título será un Text con el nombre de la ciudad y un maxLines de 2 para que no quede desconfigurado en nombre de ciudades excesivamente largos.

En el *background* o fondo del flexibleSpace usaremos un Stack para tener de fondo nuestra imagen descriptiva del fondo, la cual la tendremos en los *assets*. Por ello y para uso de futuros assets modificaremos el archivo pubspect.yaml quitando los comentarios en assets y añadiendo el nombre de la carpeta en la cual están teniendo muy en cuenta las tabulaciones.

```
  # To add assets to your application, add an assets section, like
this:
  assets:
    - assets/
```

Figura 16. pubspec.yaml Assets(Fuente propia)

Una vez configurado el archivo utilizaremos un widget para imágenes muy útil, el FadeInImage. Este widget hace un efecto de transición más suave una vez la imagen que deseamos está cargada, y tiene dos parámetros importantes, el placeholder en el cual pondremos el *path* de una imagen por defecto cuando no cargue y en el parámetro image la imagen deseada.

A continuación, en nuestro Stack irá toda la información superpuesta a esta imagen. Esta será una Column ya que los elementos estarán distribuidos verticalmente. Para evitar que el código sea muy grande ya no solo verticalmente, sino horizontalmente por niveles separaremos este widget como uno independiente pero al ser este uno poco reutilizable no lo añadiremos en un archivo aparte.

Crearemos una variable tipo *Size* que contendrá las medidas de nuestro móvil, la cual variará dependiendo del modelo lo que nos ayudará a mantener unas medidas que lucirán igual en todos los modelos. Para ello utilizaremos MediaQuery, un widget que nos proporciona información sobre el dispositivo por el cual está corriendo la aplicación.

La estructura de la Column será la siguiente:

- Text temperatura
- Row centrada (Text descripción tiempo y fadeInImage con icon tiempo)
- Text Hora actual
- Text Calle localización tiempo
- Icon localización si coincide con la localización actual

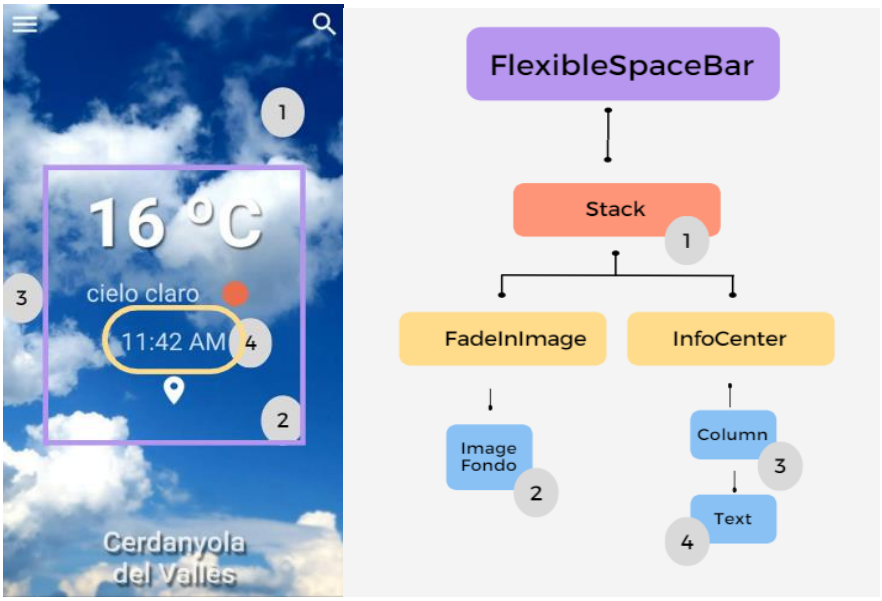


Figura 17. HomeScreen widget tree (Fuente propia)

5.3.1.2 Widget Información General

El último elemento de la CustomScrollView es la SliverList un widget que estará formado por cada uno de los Slivers independientes que formarán la pantalla principal.

El siguiente elemento que acompaña a la descripción del tiempo será un pequeño widget con la información general diaria el cual llamaremos MinMaxDescripcion y lo añadiremos a la carpeta de widgets para hacerlo reutilizable.

Este widget estará formado por una Column con dos Row que contendrán widgets Text con la temperatura máxima, mínima, descripción y fecha actual.

Estos dos widgets creados hasta ahora formarán la pantalla principal figura(nº18) cuando el SliverAppBar no está retraído mostrando lo que el usuario vería una vez inicializada la aplicación.

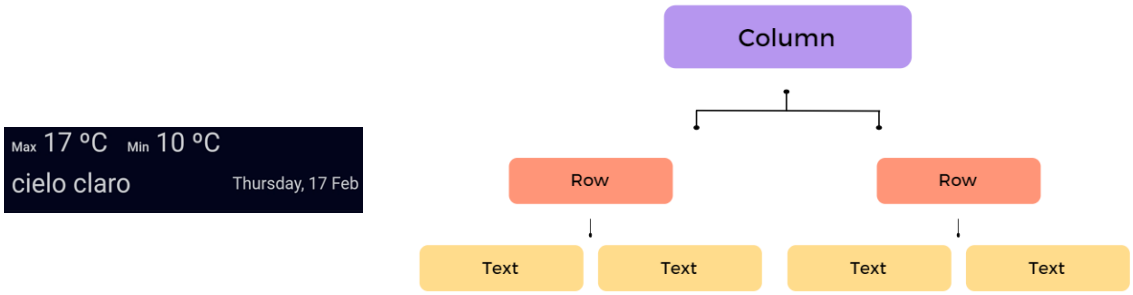


Figura 18. MinMaxDecrip widget tree(Fuente propia)

5.3.1.3 Horas Info Widget

Como anteriormente, creamos un nuevo documento llamado horas_info.dart y volvemos a importar y exportar en el archivo nexo.

Este widget mostrará la información del tiempo por horas de la localización lo cual es bastante larga con lo cual utilizaremos un widget *scrollable* descrito anteriormente, un Listview.builder con dirección horizontal, pero al usar información falsa en un primer

momento se utilizará un Listview con longitud fijada. Este tendrá como padre un container con altura fijada, para evitar el desbordamiento, ya que el Listview coge el tamaño de su padre, si este no existiera, las partes que aún no están construidas, partes no visibles tendrían tamaño infinito lo que provocaría un error.

Los elementos que muestra el listview serán widgets personalizados formados por una Column con:

- Text(hora)
- Image (Tiempo descriptivo)
- Text(temperatura)

Modificaremos el padre del Listview para añadirle un GestureDetector ya que este widget nos dirigirá en un futuro a la pantalla de gráficas con la información por horas. Por el momento podemos o bien dirigirla a la propia pantalla y modificarla posteriormente, dejarla null o bien crear el archivo que contendrá la pantalla únicamente mostrando un AppBar. Crearemos el archivo en la carpeta de screens y añadiremos la ruta en el MaterialApp donde antes habíamos creado la ruta home.

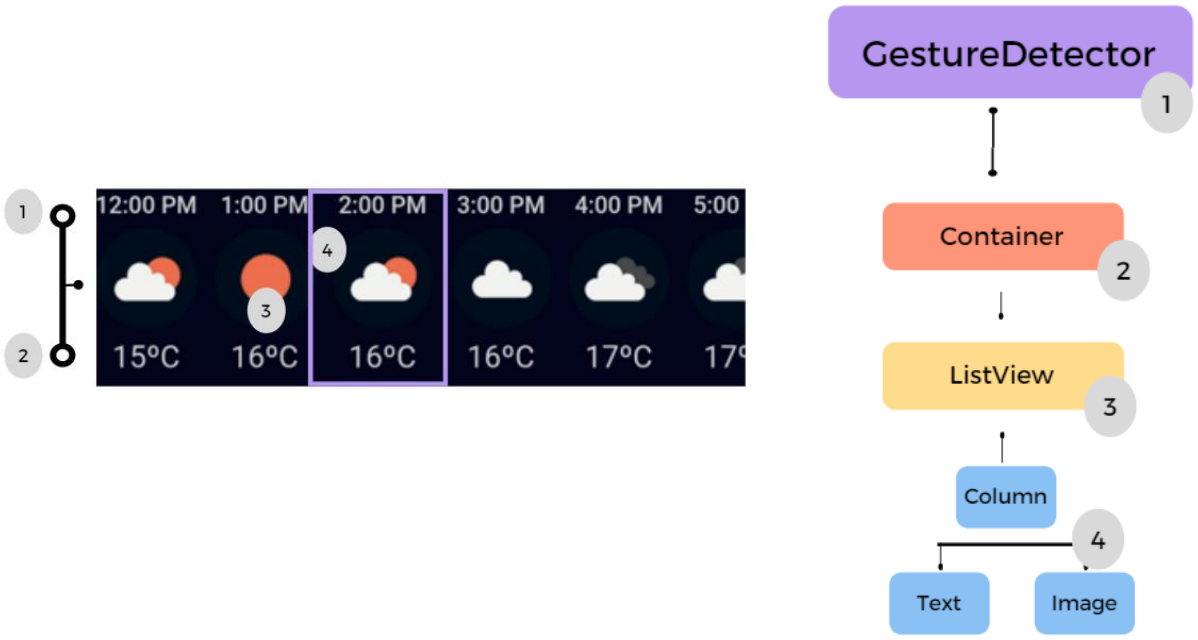


Figura 19. HorasInfo widget tree(Fuente propia)

5.3.1.4 Porcentajes widget

Los datos de humedad y nubes se reciben como porcentajes con lo cual, una buena manera de mostrarlos es usar gráficas circulares, que las hacen más visuales. Ya que la única diferencia entre ambas gráficas es el valor y el nombre descriptivo que muestra, sería algo ineficiente crear dos widgets diferentes. Este es el primer momento en el que vemos la utilidad de crear widgets independientes y reutilizables.

Comenzaremos creando el archivo para este widget en la carpeta de widgets, llamándolo Percent_circle.dart. Este widget recibirá dos parámetros que serán los que lo diferencian, el valor y la descripción y retornara un CircularPercentIndicator, un widget que nos permite crear una gráfica circular con los datos que le proporcionamos. Antes de su uso necesitaremos importar la librería que proporciona este widget, Percent_indicartor. Para

ello copiaremos la librería y su versión en el archivo pubspect.yaml en el apartado dependencies figura(nº20) y guardaremos. Este paso habrá que seguirlo siempre que queramos utilizar una librería exterior, cosa que ocurrirá a lo largo del proyecto.

```
dependencies:
  cupertino_icons: ^1.0.2
  fl_chart: ^0.45.0
  flutter:
    sdk: flutter
```

Figura 20. Pubspec.yaml depedencies(Fuente propia)

Una vez finalizada la instalación importamos la librería y hacemos uso del widget.

Estos widgets tendrán como padre un Row para mostrarlos horizontalmente.

Modificamos las propiedades siguientes del CircularPercentIndicator:

- radius: el tamaño que deseemos para la circunferencia.
- percent: el valor recibido como parámetro y que mostrará en la barra circular.
- animation: true para darle una animación.
- animationDuration: tiempo suficiente para apreciar la animación 2000 milisegundos.
- center: un widget que se muestra en el interior del círculo. Añadiremos un CircularAvatar, un widget que crea un container circular al cual añadiremos el valor en número y la descripción.

El resto de parámetros son opcionales y tienen relación con el *theme* de la aplicación ya que son el color de la barra, estilo, … Estos parámetros se editarán una vez finalizada la aplicación para hacerla lo más homogénea posible.

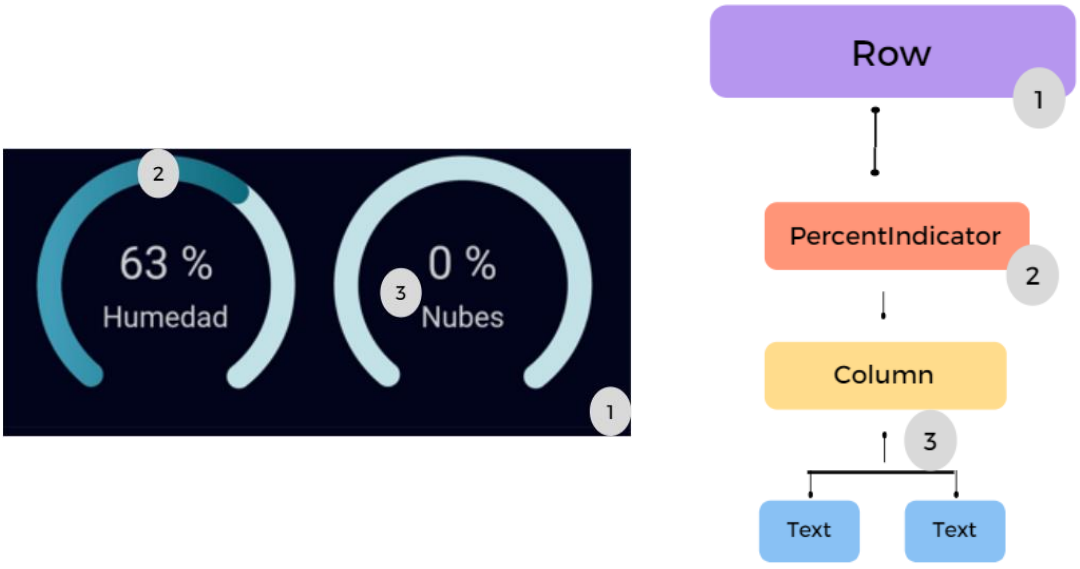


Figura 21. CircularPercentIndicator widget tree (Fuente propia)

5.3.1.5 Widgets Tiempo Actual

Este widget será nuevamente un stateless widget y estará formado por diversos mini widgets cada uno reflejando un dato de la información recibida. El primer paso nuevamente será crear el archivo correspondiente en la carpeta widgets y modificar el archivo nexo. Se llamará MiniInfoCard y estará formado por:

- Icon que refleje el dato.
- Text con la descripción de la información.
- Text con el valor.

Todos ellos teniendo como padre un Container para dar forma y una Column para la orientación

Estos datos son la diferencia entre los diferentes MiniInfoCards por lo que son las variables que recibamos por parámetros.

Como el número de MiniInfoCards puede ser variable, como bien podemos ver en los modelos de datos que estos pueden ser null usaremos un Wrap. Un Wrap es un widget que coloca widgets de forma continua y si este no tiene espacio suficiente lo recoloca en distintas filas, sumado a que el lenguaje Dart permite hacer listas de objetos con condiciones. Colocaremos pues como hijos una lista de widgets con condiciones para que aparezcan, estas condiciones serán; si estos datos son no nulos crearemos el MiniInfoCard y se recolocan, si en cambio es nulo no se creará y se reajustará como podemos ver en la figura(nº22).

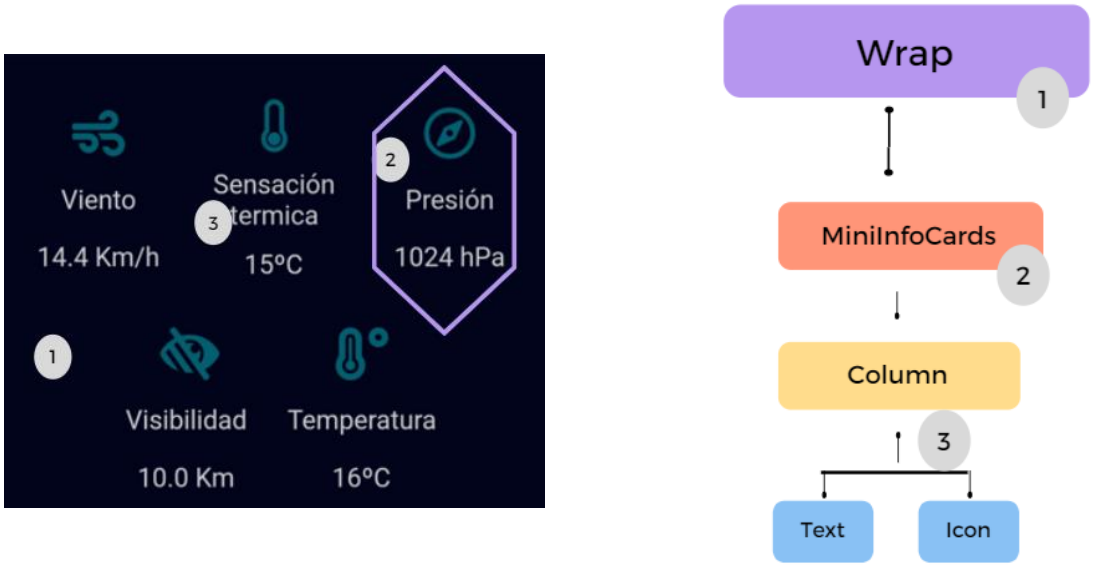


Figura 22. MiniInfoCard widget tree(Fuente propia)

5.3.1.6 Dias info Widget

Nuevamente empezaremos creando el archivo correspondiente en la carpeta de widgets y en el archivo nexo. Lo llamaremos DiasInfoWidget y cómo widget principal crearemos un Listview al que fijaremos una medida igual para cualquier dispositivo usando una vez más el MediaQuery. El uso de un Listview es útil para casos como un dispositivo con la pantalla pequeña que no pueda mostrar todos los elementos, esto permitirá desplazarte para mostrar los ocultos, además permite controlar errores en caso de no recibir algunos datos, ya que, al no tener tamaño fijo, mostrará únicamente los datos de los que si se dispone.

El Listview deplegará sus hijos en vertical y cada uno de ellos será un ListTile. Un ListTile es un widget de altura fija en forma de tarjeta que suelen usarse en Listview o Column y que suele estar formado por un leading, title y trailing, los cuales pueden ser cualquier widget.

El ListTile tendrá como heading un CircleAvatar con una NetworkImage que será redonda gracias a él y que recibiremos mediante una url que nos proporciona la API, en este caso como aún se está construyendo el esqueleto sin datos reales usaremos una por defecto. Cómo *body* irá un Row con dos Text, la fecha del día y su temperatura máxima y mínima y en el trailing o final de la ListTile un Icon.

Para finalizar encapsulamos todo el Listview en un GestureDetector en el cual su función onTap() nos dirigirá a la pantalla de pronóstico de 16 días, como aún no está implementado la podemos dejar en null o dirigirla hacia esta misma.

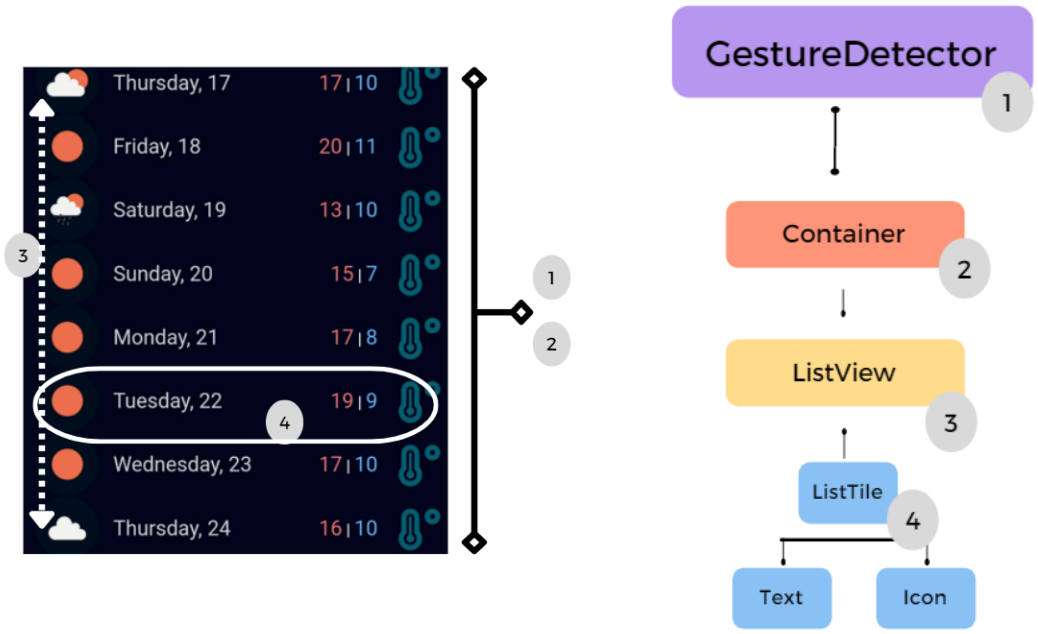


Figura 23. DiasInfo widget tree (Fuente propia)

5.3.1.7 Gráfica

El último Sliver del CustomScrollView será una gráfica que refleje las temperaturas máximas y mínimas de la semana.

En este Sliver únicamente haremos 3 pasos y lo dejaremos en standby:

1. Crearemos el archivo en la carpeta de widgets y en el nexo.
2. Crearemos un Container de cualquier color con el tamaño deseado para la gráfica o una imagen que lo refleje.
3. Importamos la libreria fl_chart para la construcción de gráficas en pubspect.yaml.

No avanzaremos hasta que no tengamos los datos reales ya que suponen una carga grande inventarse datos para simularlo y únicamente manteniendo el tamaño que se desea es suficiente para tener el esqueleto

5.3.1.8 Pantalla pronóstico días

Por último, crearemos el archivo para la pantalla del pronóstico a 16 días. Para ello volveremos a añadir la nueva ruta en nuestro MaterialApp a continuación de las siguientes y crearemos un ListView cuyos elementos serán las tarjetas con la información diaria.

Este widget lo llamaremos InfoWidget y volveremos a seguir los pasos de crear su propio archivo en la carpeta widgets y posteriormente las importaciones en su archivo nexo.

La estructura de este widget es la siguiente:

- Container: como *decoration* una imagen descriptiva
- Row y Column para ordenar la información

Una vez creado el esqueleto modificaremos el GestureDetector de DiasInfoWidget para que nos dirija a esta pantalla con la ruta creada anteriormente.

5.3.2 Datos desde API

Una vez tenemos el esqueleto de nuestra pantalla, el siguiente paso es sustituir los datos falsos por los recibidos de la API. Para ello seguiremos los siguientes pasos.

1. Creación de los modelos de datos
2. Creación del Provider.
3. Creación del Geolocator.
4. Pasar los datos a la pantalla principal utilizando Provider.

5.3.2.1 Modelos de datos

Como en el caso de las pantallas y los widgets crearemos nuestro archivo nexo que hará todos las importaciones y exportaciones de todos los modelos en un archivo.

Una vez creado el archivo iremos construyendo los modelos tal y como estaban planteados en el punto 3 de la memoria, para ello utilizaremos Postman y QuickType.

QuickType nos proporciona directamente el modelo completo con datos que no utilizaremos, por lo tanto eliminaremos los datos no necesarios tales como números de identificación internos o información sobre la llamada http para dejarlo lo más limpio posible.

Dividiremos el modelo de OneCallResponse en modelos más pequeños y más manejables, que también serán usados por otros modelos. Estos modelos serán:

- Current
- Daily
- Weather
- Rain
- Snow
- FeelsLike
- Temp

Una vez creado los diferentes archivos de los modelos crearemos el constructor de cada clase que la inicialice desde un archivo JSON correspondiente a los datos que recibimos desde la API. Este constructor se llamará NombreClase.fromJson y para su implementación importamos dart:convert que nos proporciona el método .decode. QuickType nos proporciona el constructor, pero habrá que adaptarlo al *null safety* de Flutter para evitar errores con valores nulos figura(nº24).

```
factory Temp.fromJson(String str) => Temp.fromMap(json.decode(str));
  String toJson() => json.encode(toMap());

  factory Temp.fromMap(Map<String, dynamic> json) => Temp(
        day: json["day"].toDouble(),
        min: json["min"].toDouble(),
        max: json["max"].toDouble(),
        night: json["night"].toDouble(),
        eve: json["eve"].toDouble(),
        morn: json["morn"].toDouble(),
);
```

Figura 24.ej constructor fromJson(Fuente propia)

Seguiremos los mismos pasos para HorasModel, el modelo de la información por horas y SixteenDaysModel para el pronóstico de 16 días.

5.3.2.2 Creación del Provider

Una vez creado los modelos, el siguiente paso es recibir los datos y rellenarlos. Estos datos al venir de llamadas a la API pueden tener cierto retraso por lo que no son instantáneos y la aplicación no puede pararse para esperarlos, con lo cual deberá mostrar algo que haga al usuario saber que se están cargando los datos y una vez recibidos mostrarlos.

Para la lógica de avisar del cambio de datos crearemos nuestro Provider, el cual llamaremos CurrentWeatherProvider y que extiende de la clase ChangeNofier, la cual nos proporciona la funcionalidad de ser escuchado para ver si hay algún cambio en ella. Este Provider se encargará de hacer las llamadas a la API y guardar los datos. Como el Provider se encarga de avisar a la pantalla principal, éste deberá estar colocado e inicializado en un nivel superior del árbol de widgets al que debe escuchar, por lo tanto, en nuestro caso irá antes del MyApp , en el nivel más alto, para que sea el primer widget que se inicialice y sus datos sean accesibles en toda la aplicación sin necesidad de ir pasándolos por los diferentes niveles y evitar declaraciones de variables como vemos en el diagrama de la figura(nº25).

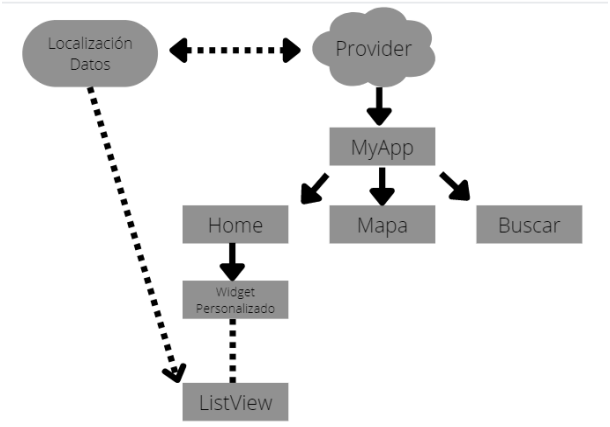


Figura 25. Diagrama Provider(Fuente propia)

El provider tendrá 4 variable importantes:

1. La localización actual del usuario: Esta localización será *null* hasta que nuestro geolocator nos actualice el dato. Esta variable será accesible por todas las pantallas gracias a Provider ya que es necesaria en muchas de ellas, para la inicialización de los mapas en la localización actual
2. Las 3 listas de tiempos (OneCall, diario, Hora): Se irán rellenando a medida que recibamos datos de la API. Al inicio estarán vacías y por lo consiguiente la pantalla mostrará que se están cargando los datos.

Se crearán estas 4 variables, inicializadas vacías. También crearemos variables Final, para almacenar datos que nunca cambiarán. Estos datos serán datos comunes para las llamadas a la API y son los siguientes:

- API key: Las credenciales para nuestra llamada. El valor de esta variable se guardará en un archivo especial comúnmente llamado .env en el cual el usuario no tendrá acceso y que evitaremos subir a nuestro repositorio de control de versiones. Para ello crearemos el archivo .env con la variable y en la función main() la llamaremos antes de correr la aplicación para tener uso de ella. Al ser una carga desde archivo, está función devuelve un Futuro, un valor que aún no disponemos de él, tal como pasa con las llamadas a la API, con lo cual modificaremos nuestro main para que la función sea asíncrona y espere el resultado antes de seguir.
- Idioma, unidades y la url de la API**:** estos datos sí pueden estar en el código ya que no suponen un riesgo de seguridad.

A continuación, crearemos la función que hará la llamada a la API para recibir el tiempo actual figura(nº26).

```
void getOneCallWeather(List<dynamic> geo) async {
    final jsonData =
        await _getJsonDataByGeo('data/2.5/onecall', geo[0], geo[1]);
    final OneCallResponse currentCall =
OneCallResponse.fromJson(jsonData);
    final localizacion = await getUbicacion(currentCall.lat,
currentCall.lon);

    currentCall.localizacion = localizacion;
    callsWeather.add(currentCall);

    if (localizacion.locality == _location) {
      weatherLocation = currentCall;
    }
    notifyListeners();
  }
```

Figura 26. Método getOneCallWeather(Fuente propia)

El funcionamiento de esta función es simple:

1. Recibe como parámetro las coordenadas de la localización a buscar.
2. Hace la llamada a la API correspondiente
3. Crea la clase con el Json que recibe gracias a la función .fromJson
4. Añade la clase a la lista de tiempos actuales
5. Avisa al widget que está escuchando los cambios para que repinte la pantalla con el método notifyListeners() de la clase ChangeNotifier.

Los datos que recibimos de esta llamada tienen un pequeño problema, el recibir la ubicación de la ciudad en coordenadas y no por nombre. Como solución, implementamos la función getUbicacion(). A partir de las coordenadas y usando un paquete de Flutter, “geolocator” permite recibir información sobre las coordenadas, entre ellas el nombre de la ciudad más cercana. Guardaremos este dato en una variable llamada Ubicación dentro de la clase OneCallResponse y con ello dispondremos de toda la información necesaria para rellenar el esqueleto de nuestra pantalla.

Haremos lo mismo con los datos diarios y por horas cambiando la llamada a la API, las funciones se llamarán getFourDayHourlyWeather y getSixteenDaysWeather.

Una vez tenemos creadas las funciones para las llamadas, crearemos el constructor del provider. La inicialización del provider seguirá esta lógica:

1. Buscará la localización actual y hará las llamadas correspondientes para recibir los datos usando las tres funciones creadas anteriormente. Una vez recibido los añadirá a las listas que las guardan.
2. Leerla nuestros favoritos que estarán guardadas en las preferencias de nuestro dispositivo. Este archivo contendrá las coordenadas de nuestros favoritos y con ellas y las funciones creadas anteriormente hará las llamadas correspondientes y recibirá los datos.

La primera acción que deberá hacer será buscar los datos de la localización actual ya que un usuario nuevo no tendrá favoritos guardados. Como el usuario puede tener la ubicación desactivada, debemos crear un aviso para que éste acepte los permisos y manejar los errores. Para ello crearemos una carpeta Services en la cual crearemos los archivos que manejan estas situaciones, ya que en un futuro con las notificaciones será necesario preguntar por permisos.

La documentación de este *Service* o servicio está en la documentación oficial del paquete en pub.dev. El funcionamiento es el siguiente:

Este servicio cuenta con dos variables importantes, un bool que indica si tenemos activado la localización, y una variable enum, que nos indica que tipo de permiso disponemos. Si la localización está desactivada, este enviará un mensaje de error y no dispondremos de la información, por el contrario, si este está activo pero no tenemos permisos, enviaremos una solicitud para recibirlos. Una vez dispongamos de ellos una función devuelve la posición. Con la posición recibida, provider hará las llamadas a la API y dispondremos de la información.

La segunda acción del provider será leer los favoritos del usuario. Para ello tendrá que leer los datos guardados en las shared preferences del dispositivo, estos datos se borrarán una vez eliminemos la aplicación. Estos datos serán guardados en un Json y estará formado por un Map<Ciudad,Coordenadas>

Necesitamos implementar cuatro métodos:

getDataPreferences: recibir datos desde las preferencias de usuario no es instantáneo por lo tanto esta función será asíncrona y devolverá un valor futuro. Lo primero que hará este método será crear una instancia de estas preferencias y posteriormente usar un método de las preferencias para recibir los datos. Si estos son nulos querrá decir que el usuario aún no ha añadido favoritos y no hará las llamadas. Por el contrario, si recibimos algún dato hará las posteriores llamadas a la API para recibir la información figura(nº27)

```
Future<void> getDataPreferences() async {
    final SharedPreferences prefs = await
SharedPreferences.getInstance();

    final datosJson = prefs.getString('datos');
    if (datosJson != null) {
      mapCities = Map<String,
dynamic>.from(jsonDecode(datosJson));

      mapCities.forEach((key, value) {
        getOneCallWeather(value);
        getFourDayHourlyWeather(value);
        getSixteenDaysWeather(value);
      });
    }
  }
```

Figura 27. Método getDataPrefereces(Fuente propia)

saveDataPreferences: Guarda los datos en las preferencias de usuario.

removeLocation: Recibe una ciudad y del Map el valor correspondiente a esa llave. Posteriormente llama a saveDataPreferences para actualizarlos datos internos.

addLocation: Añade al Map un nuevo valor <ciudad, coord> posteriormente llama a saveDataPreferences para actualizar.

La inicialización del provider quedará simplificada y se puede ver en la figura(nº28)

```
//init
 CurrentWeatherProvider() {
   getCurrentLocationWeather();
   getDataPreferences();
 }
```

Figura 28. Inicialización Provider(Fuente propia)

5.3.2 Datos reales

Una vez creado el Provider ya disponemos de todos los datos desde que iniciamos la aplicación. Los siguientes pasos serán, sustituir los datos falsos por datos reales en la pantalla principal y la pantalla de pronóstico 16 días; crear la gráfica en la pantalla principal y por último crear la pantalla de gráficas.

5.3.2.1 Pantalla principal

Para hacer uso de los datos guardados en el provider crearemos una variable CurrentWeatherProvider tal como la llamamos anteriormente. Cabe hacer hincapié en el lugar donde declaremos esta variable. Sí su declaración se hace dentro de un método, este Provider tiene que llevar la opción “listen” o escucha en false ya que daría error porque se volverá a pintar la pantalla continuamente.

Como ahora disponemos de datos reales y el usuario acostumbrará a tener más de una ciudad en su lista, añadiremos un widget padre a toda la pantalla principal. Este widget será un pageView, parecido a un ListView pero a diferencia de este, los elemento que muestra deben ser todos del mismo tamaño y acostumbran a ser pantallas enteras. En este caso cada elemento y una pantalla completa del tiempo actual por cada ciudad. El número de pantallas dependerá de la longitud de la lista de elementos guardados en la variable de nuestro provider por lo que al ser este asíncrono y tener un valor inicial nulo aplicaremos una condición mientras estos datos cargan. Si no hay datos se mostrará un widget de carga. Existen varios widgets de este tipo, podríamos poner desde una animación hasta una pantalla por defecto. En este caso y hasta el diseño final le añadiremos un CircularProgressIndicator, un widget que crea una animación de carga.

Se añadirán, ahora sí, imágenes sin derechos que reflejen el tiempo. Estas se guardarán en la carpeta de assets. Dentro de nuestro widget CustomAppBar crearemos un Map<String,String> con el código de identificación del tiempo recibido por la API y el path de la imagen correspondiente en los assets. Con esto se modificará la imagen por defecto por el correspondiente valor recibido por el Map, que siempre tendrá resultado.

Por otra parte, en estos momentos, una vez cargado los datos no hay manera de refrescarlos y actualizarlos. Para ello añadiremos otro widget encargado de esto, un RefresIndicator. Este widget será el padre de nuestro CustomListView, ya que tiene que ir acompañado de un widget scrollable y tiene una propiedad llamada onRefresh que ejecuta la acción que le indiquemos. En nuestro caso deberá refrescar los datos por si han habido cambios, por lo tanto tiene que hacer llamadas a la API. Ya que estos datos y los métodos que implica esta función están en el Provider y este es accesible desde cualquier parte, la implementaremos en él.

Esta función limpiara las variables donde están guardado los datos y llamara de nuevo a las funciones iniciales figura(nº29).

```
Future<void> refreshData() async {
  callsWeather.clear();
  infoPorDias.clear();
  infoPorHoras.clear();
  getCurrentLocationWeather();
  getDataPreferences();  }
```

Figura 29. Método refreshData(Fuente propia)

El siguiente paso será pasar los datos a cada uno de los widgets personalizados que creamos anteriormente, todos ellos ya están en el Provider.

Para finalizar con la pantalla principal, crearemos la gráfica con las temperaturas máximas y mínimas proporcionadas por OneCallResponse.

Para ello utilizaremos la librería fl_chart. En este caso utilizaremos un BarChart para crear la gráfica en forma de barras y ya que podemos suponer que la temperatura máxima y mínima diferirá en un par de grados, colocaremos ambas en la misma posición para que superpongan y creen un efecto vistoso al tener diferentes colores.

En los márgenes pondremos el día de la semana y sobre las columnas el valor.

Para poner la fecha real, utilizaremos la librería intl.dart con el cual podremos hacer un *parse* o cambio desde el valor que nos proporciona la API en la variable dt a día de la semana. El resultado final se puede ver en la figura(nº30).

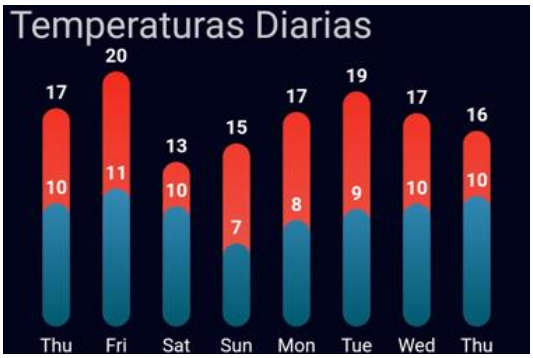


Figura 30. Gráfica temperaturas diarias(Fuente propia)

5.3.2.2 Pantalla pronóstico días

Esta pantalla deberá mostrar los datos correspondientes a la ciudad en la cual hemos hecho la navegación. En esta ocasión los datos correspondientes se recibirán en la pantalla anterior a través de Provider comprobando que coinciden las coordenadas geo ya que es el dato único que nos sirve de identificador. Este dato es el idóneo ya que, aunque hayan dos iguales, ambos deberán tener los mismos datos meteorológicos, en dos posiciones geográficas idénticas no pueden variar así se evitarán errores.

Una vez recibidos los datos en la pantalla de pronósticos, sustituiremos los datos falsos por ellos. Para las imágenes que decoran las tarjetas utilizaremos propias o sin derechos que permitan su uso. Estas las guardaremos en nuestra carpeta *assets* y crearemos un Map<código, path imagen>. Tal como vemos en nuestro modelo de datos Weather, este viene con un código correspondiente a cada tiempo el cual también se usa para recibir el icono correspondiente. Haremos uso de estos códigos para imitar esa funcionalidad con nuestro Map de imágenes.

5.3.2.3 Pantalla gráficas

Como en la pantalla de pronósticos, los datos serán enviados desde la pantalla anterior, evitando errores para identificar qué datos mostrar. En esta ocasión, el esqueleto de la pantalla no está construido ya que eran necesarios datos reales.

El primer paso será crear su AppBar para identificar la ciudad correspondiente a los datos. Está formada por un IconButton para volver a la pantalla anterior y como *title* el nombre de la localidad, barrio o ciudad.

A continuación, irán las gráficas, estas mostrarán la temperatura, sensación térmica, humedad y presión cada hora y también un índice con las horas correspondiente a los datos. Todos estos elementos tendrán la misma longitud ya que muestran el mismo número de horas por lo que colocaremos todos los elementos, el índice y las cuatro gráficas dentro de una Column para mostrarlos verticalmente un detrás del otro. Seguidamente colocaremos un SingleChildScrollView en la Column para darle la propiedad de scroll con horizontal y así poder desplazarnos por todas las horas. Poner este elemento en la Column y no en cada gráfica permitirá que todos los elementos se desplacen en sincronía.

Para ello lo primero que construiremos será el índice que indique a qué hora corresponde cada marca en las gráficas. Este índice será un ListView parecido al creado en HorasInfoWidget, el cual estará formado por un CircleAvartar que como imagen tendrá el icono del tiempo correspondiente y un Text con la hora. La imagen será un NetworkImage en la cual la *Url* siempre es la misma variando el valor Icon de nuestro modelo de datos Weather que recibimos de la API.

Los siguiente cuatro Widgets són iguales únicamente variando el título y los datos por lo que crearemos un widget personalizado que reciba estos datos. Llamaremos al widget HourlyLinearChart y para su construcción utilizaremos de nuevo la libreria fl_chart.

Utilizaremos el widget LineChart para crear nuestras gráficas lineales. Como leyenda únicamente mostraremos un rango de valores a la izquierda de la gráfica que refleje el máximo y mínimo y algunos valores medios. Para mostrar los valores correspondientes a cada hora, los valores solo se reflejan si tocamos la gráfica para evitar mostrar demasiada información en poco espacio.

5.3.3 Theme y modelo final

Una vez toda la lógica de las pantallas está aplicada, el último paso es crear el modelo final. Buscar un tema para la aplicación que sea homogéneo en todas las pantallas, reordenar widgets para que quede más vistoso, añadir animaciones etc.

Empezaremos con los estilos de los widgets como Text, Icons, AppBar. Para ello crearemos en nuestra carpeta Themes una clase llamada AppTheme con una variable estática de tipo ThemeData. En ella cambiamos el estilo de Text para diferentes momentos, encabezados, subtítulos, etc. Otros elementos a cambiar serían AppBarTheme ya que son usadas en muchas pantallas, así evitamos editarlas. IconTheme, listTileTheme.

Gracias a este archivo evitamos tener que cambiar elemento a elemento toda la aplicación, además nos permite tener varios temas y jugar con ellos.

Para el fondo o *background* de la aplicación crearemos una personalizada utilizando CustomPaint. Crearemos 3 fondos diferentes pero relacionados entre sí, todos ellos los guardaremos en la carpeta Themes.

Lo primero que haremos será crear un container con el tamaño deseado para nuestro fondo en este caso la medida completa de la pantalla. Este Container tendrá como hijo el CustomPaint que cogerá como tamaño el de su padre.

Para dibujar en nuestro canvas deberemos crear dos variables:

- Paint: El estilo que tendrá nuestro dibujo, tamaño de la línea, color…
- Path: La posición donde pintamos en nuestro canvas.

Mediante Path creamos el camino que seguirá nuestro pincel. Una vez creado todo el recorrido el método canvas.drawPath que recibe como parámetros el Path y el Paint, dibujara este camino con el estilo indicado.

Una vez creados los tres fondos, se aplicarán a las pantallas utilizando el Widget Stack para colocarlo como fondo y el resto de la pantalla pintado encima.

El resultado final de las primeras pantallas es el siguiente:

Figura 31. Pantallas actuales(Fuente propia)

5.4 Pantalla mapa layers

Esta pantalla mostrará los diferentes mapas meteorológicos que nos ofrece la API. Se llegará a través del IconButton mapa situado en la pantalla principal, no importa qué ciudad estamos situados ya que este mapa se iniciará con tu ubicación.

Dividiremos la pantalla en dos widgets:

- FlutterMap al que llamaremos CustomLayeredMap y será el responsable de mostrar los mapas y sus diferentes capas.
- BottomNavigationBar la cual implementará la lógica para el cambio de *layer*. Utilizaremos un pequeño provider para controlar este estado.

5.4.1 FlutterMap

Crearemos un nuevo widget en nuestra carpeta Widgets al que llamaremos CustomLayeredMap. Este widget retornará un FlutterMap y para su uso necesitaremos importar la librería correspondiente en nuestro pubspect.yaml.

Primero editaremos las opciones iniciales de FlutterMap:

- center: posición inicial, usaremos nuestro CurrentWeatherProvider para coger la variable currentLocation e inicializar el mapa con sus coordenadas.
- zoom: 5.0 suficiente para tener una vista amplia de la zona.
- maxZoom: 20.0 con este valor evitamos el límite de detalle que tiene los layers recibidos
- minZoom: 4.0 alejarnos demasiado desconfigurara el mapa
- screenSize: Gracias a MediaQuery cogeremos la medida de nuestro dispositivo

A continuación, crearemos las capas o layers de nuestro mapa. FlutterMap permite dos tipos de *layer,* unos que se pueden rotar y unos fijos. Después de varias pruebas, los *layers* rotables acababan desalineados respecto al mapa principal con lo cual se usarán los noRotatedLayers.

La opción noRotatedLayers consiste en una lista de widgets LayerOptions en lo cual se incluyen tanto layers (TileLayerOptions) como markers o marcadores (MarkerLayerOptions) que se dibujan en el mapa. En ella añadiremos dos TileLayerOptions.

El primero será la capa principal la cual recibiremos desde openstreetmap.

El segundo será nuestro *layer*, el cual recibiremos desde la API. Como habrá diferentes *layers* y lo único que cambiará será la llamada a la API, nos crearemos un método que devuelva el *layer* correspondiente. Como parámetros tendrá el valor que hay que cambiar en la Url para la llamada a la API y la paleta de colores, ya que el *layer* que recibimos es editable mediante la llamada a la API variando las opciones figura(nº32).

```
TileLayerOptions layerMap(String opt, String palette) {
  final String _apiKey = dotenv.env['API_KEY']!;
  return TileLayerOptions(
      urlTemplate:

"http://maps.openweathermap.org/maps/2.0/weather/$opt/{z}/{x}/{y}?app
id=$_apiKey",
      opacity: 0.85,
      additionalOptions: {'palette': palette});
}
```

Figura 32. Customable layer(Fuente propia)

La lógica de qué *layer* escoger y cuando se cambia se llevará a cabo en la BottomNavigationBar que explicaremos a continuación.

5.4.2 BottomNavigationBar

Se trata de un Widget que se utiliza para la navegación por pantallas en aplicaciones, en este caso su uso será el cambio de layer de nuestro mapa: Crearemos un pequeño provider que se ocupará de guardar y actualizar la posición en la que se encuentra nuestra BottomNavigationBar.

Empezaremos creando el archivo LayerProvider en la carpeta Providers y crearemos la clase extendiéndose como en el anterior provider de ChangeNotifier. Este pequeño provider solo tendrá una variable Int que almacenará la posición. Su valor inicial lo definiremos en 0 y crearemos un método que actualize el valor con un nuevo valor recibido. Un bottomNavigationBar, al ser pulsado uno de los elementos que lo componen devuelve la posición(int) que ocupa ese elemento, este será el valor que reciba este método figura(nº33).

```
class LayerProvider extends ChangeNotifier {
  int _layerPos = 0;

  int get layerPos => _layerPos;

  set layerPos(int pos) {
    _layerPos = pos;
    notifyListeners();
  }
}
```

Figura 33. LayerProvider(Fuente propia)

Una vez creado nuestro Provider proseguiremos construyendo el bottomNavigationBar. Editaremos las siguientes propiedades:

- currentIndex: añadiremos el valor de nuestro Provider, el cual cada vez que este varíe, Provider hará que vuelva a pintar con el nuevo valor.
- onTap: nuestro método que actualiza el valor actual con el valor recibido.
- Items: lista de BottomNavigationBarItem, tantas como layers tenemos, con su correspondiente Icon y descripción.

Por último crearemos una lista con las diferentes opciones para la llamada a la API, en el mismo orden que están colocados los elementos de nuestra navigationBar, gracias a la posición que recibimos de nuestro provider FlutterMap mostrará el layer correcto.

5.4.3 Leyenda

Por último, el layer recibido por la API, no dispone de leyenda que indique qué colores representan qué valores. Utilizaremos CustomPainter para crear nuestras propias leyendas. Para ello crearemos un Widget personalizado que reciba como parámetros las

unidades, el título y la paleta de colores, los únicos datos que cambian de uno a otro. La paleta de colores correspondiente la podemos encontrar en la documentación de la API. Para hacerla editable, crearemos una lista de paletas, con los valores por defecto, esta paleta será también utilizada en la llamada a la API con lo cual si modificamos los valores cambiarán en ambos lados.

Cada uno de estos widgets personalizados los volveremos a guardar en una lista con el orden anterior para controlar cual mostrar con nuestro Provider.

el resultado final es el mostrado en la figura(nº34).

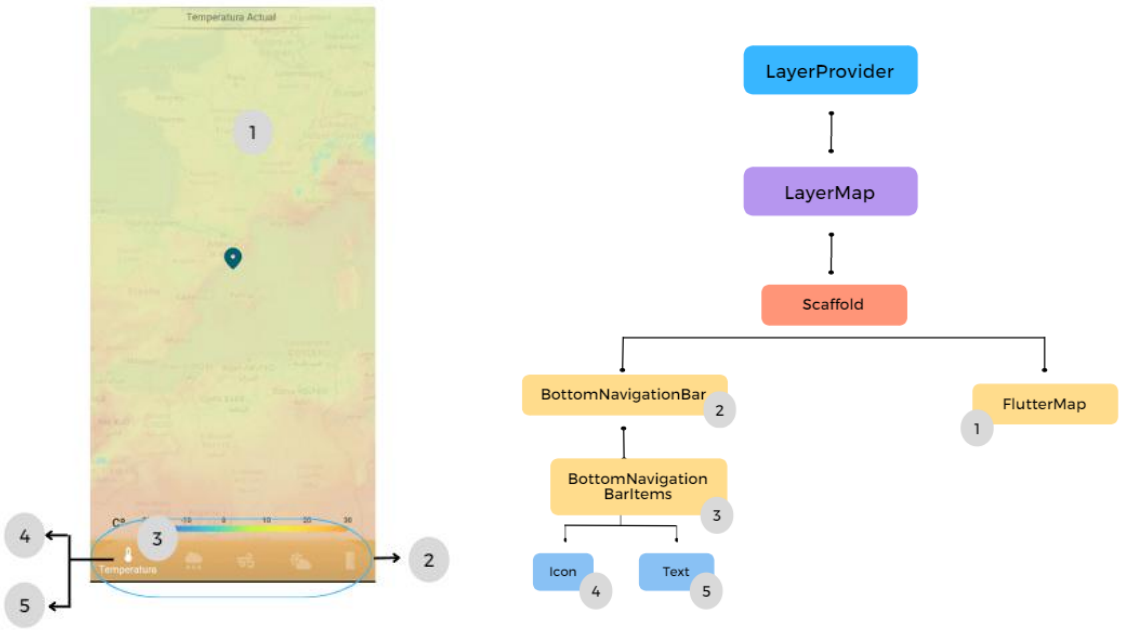


Figura 34. LayerMap widget tree(Fuente propia)

5.5 Pantalla de búsqueda

Empezaremos creando su archivo correspondiente en la carpeta screens y en el archivo nexo. Añadiremos también su ruta en nuestro MaterialApp a continuación de las otras rutas y modificaremos la función onPress del IconButton “lupa” de la homeScreen para que nos dirija a esta nueva pantalla.

Esta nueva pantalla se dividirá en 3 widgets:

- AppBar: Widget personalizado con GestureDectector que nos abrirá la pantalla de búsqueda por escritura.
- GoogleMap: Ocupará el centro de la pantalla y permitirá escoger la ciudad navegando por él.
- TextButton: Inactivo si no tenemos ciudad. Activo cuando encontremos ciudad por uno de los dos medios.

5.5.1 GoogleMap

Empezaremos creando nuestro GoogleMaps, haciendo uso de la librería google_maps_flutter. Además de editar el archivo pubspect.yaml nuevamente para su importación, GoogleMaps necesita una versión más alta de Android de la que flutter pone por defecto. Para cambiarlo iremos a Android>app>build.gradle y modificaremos el valor

de minSdkVersion a 21. Este paso es importante, ya que hay versiones anteriores que no soportan GoogleMaps u otras librerías que haremos uso más adelante.

El primer paso será cambiar nuestra pantalla de stateless widget a stateful ya que necesitaremos controlar el estado de algunos widgets y tendrán que ir actualizándose.

Crearemos una variable que almacene la ciudad escogida. Esta variable activará todos los widgets que componen nuestra pantalla.

Primero modificará el valor de nuestra AppBar y mostrará la nueva ciudad. Segundo, en el mapa se añadirá un marker a la posición de esa ciudad, y el mapa se desplazará a esa posición, y por último se activará el TextButton para añadirla.

Una segunda variable que cambiará de valor será una lista de markers, está se irá modificando con el nuevo valor que recibirá al interactuar por el mapa.

Por último necesitaremos un controller para nuestro mapa. Los controllers, en Flutter són utilizados para controlar el comportamiento de algunos widgets, como por ejemplo un TextFormfield, el cual queremos recibir el nuevo valor que el usuario escribe. Nuestro controller controlará tanto el desplazamiento por el mapa, como los markers que aparecen, el zoom o incluso el layer que muestra a medida que el usuario interactúe con él.

Una vez definidas las variables, crearemos nuestro widget GoogleMap el cual tendrá como padre un widget Expanded que le forzará a ocupar todo el espacio disponible entre la appBar y el textButton.

Modificaremos las siguientes propiedades:

- initialCameraPosition: modificaremos el zoom inicial y como posición inicial cogeremos la localización proporcionada por nuestro Provider.
- onMapCreated: Aquí inicializamos el controller de nuestro mapa, Flutter comprueba que no hay ninguna fallo asegurando que el controller devuelve resultado mediante controller.complete()
- onTap: Aquí desarrollaremos toda la lógica del mapa. La función se llamará changePlace y recibirá como parámetro las coordenadas pertenecientes a la posición que toquemos en el mapa. Una vez recibido, limpiara la lista de markers, y añadirá el marker de nuestra posición y el recibido. A continuación llamará a la función getUbicación de nuestro Provider para rellenar nuestro modelo de ciudad con estos datos. Estos datos estarán en "caché" hasta que decidamos añadir pulsando el TextButton o bien cancelemos. Por último antes de finalizar la función añadirá el nuevo nombre de la ciudad a la variable con estado creada al principio.

Con todo esto implementado, y grácias a la propiedad de los stateful widgets, todo la pantalla se irá actualizando a medida que interactuemos con el mapa.

5.5.2 TextButton

El siguiente paso será crear el TextButton y sus funciones. Este deberá añadir la nueva ciudad una vez seleccionada, mantenerse inactivo si esta ciudad es nula y enviar un SnackBar , un widget que muestra un pequeño mensaje al usuario por pantalla y que utilizaremos para informar al usuario de que la ciudad se ha añadido, este mensaje se mostrará en la pantalla home, por lo tanto, el TextButton una vez pulsado deberá dirigirnos hacia esa pantalla.

Empezaremos creando la función onPress. Esta añadirá una nueva ciudad a nuestra lista de ciudades favoritas utilizando la función de nuestro Provider addCity. Una vez añadida llamaremos a la función updateWeather para actualizar la lista de ciudades que se muestran y seguidamente se creará nuestro SnackBar mientras somos dirigidos a la pantalla principal figura(nº35).

```
      onPressed: () {
        final weatherProvider =
            Provider.of<CurrentWeatherProvider>(context,listen:
false);
        weatherProvider.addCity({
          city.name:                          [city.cood.lat.toString(),
city.cood.lon.toString()]
        });
        ScaffoldMessenger.of(context)
            .showSnackBar(SnackBar(content:  Text(city.name  +  '  '  +
'añadido')));
        weatherProvider.updateWeather(
            [city.cood.lat.toString(), city.cood.lon.toString()]);

        Navigator.pop(context);
      },
```

Figura 35. CreaciónSnackBar(Fuente propia)

Si por el contrario la variable con la ciudad está vacía la función onPress tomará el valor de *null* imposibilitando el uso de esta función y evitando errores.

5.5.3 AppBar

En este punto ya es posible añadir cualquier ciudad mediante la búsqueda por GoogleMaps, el último paso es permitir la búsqueda escrita. Para ello utilizaremos un searchDelegate, el cual será accionado mediante nuestra appBar que estará formada por un gestureDetector sobre un Row, que mostrará la ciudad actual, inicialmente la ciudad que indica nuestra localización, y un Icon para mostrar que es interactiva.

El primer paso será crear la función que modificará la cámara del mapa dirigiéndose hacia la posición buscada en nuestro searchDelegate. Esta función recibirá las nuevas coordenadas y el controller se encargará de situar la cámara en la posición mediante una animación suave figura(nº36).

```
 changePosition(CameraUpdate update) {
    _mapController.animateCamera(update);
    _mapController.moveCamera(update);
  }
```

Figura 36. Método changePosition(Fuente propia)

5.5.4 SearchDelegate

Para la creación de nuestro searchDelegate, crearemos el archivo nuevamente en la carpeta widgets y en el archivo nexo haremos las importaciones y exportaciones.

Llamaremos a la clase CountrySearchDelegate y extenderá de searchDelegate. En esta clase deberemos implementar cuatro métodos obligatoriamente los cuales juegan con una

propiedad de searchDelegate llamada query que es donde se almacenará el valor de búsqueda:

1. buildActions: devuelve una lista de widgets que suele ser IconButtons los cuales modifican el searchDelegate. Crearemos un IconButton con el método onPress para borrar la búsqueda o lo que es lo mismo poner el valor de query en blanco.
2. buildLeading: widget colocado antes de la barra de búsqueda. Utilizaremos un IconButton para volver a la pantalla anterior. El método onPress() llamará a un método de searchDelegate close() que cerrará esta pantalla y devolverá a la pantalla anterior, el valor que deseemos, en este caso null.
3. buildResults: el valor de la query con el cual estamos haciendo la búsqueda. Aparecerá en la barra de búsqueda.
4. buildSuggestions: Mostrará todos los resultados de la búsqueda que hagamos en un StreamBuilder, un widget que va recibiendo y construyendo los widgets a medida que recibe datos, con lo cual se actualiza automáticamente. Si es nula, mostrará un fondo por defecto.

5.5.4.1 Autocomplete

Para crear el buildSuggestions, primero necesitamos sugerencias, estas tendrán que ser recibidas desde el exterior cada vez que el usuario modifique el valor de búsqueda. Para ello utilizaremos la API geocodeapi la cual enviando la query como parámetro nos genera una lista de resultados que coinciden con ella.

Lo primero será crear el modelo de datos para albergar los datos recibidos. Lo llamaremos autocompleteModel y lo guardaremos en la carpeta Models. A continuación, en nuestro archivo provider del tiempo, crearemos un nuevo método que haga la llamada hacía la API. Este método tendrá como parámetro de entrada la *query* del searchDelegate y como salida una List<cityModel> con los resultados. Naturalmente este resultado será futuro ya que las llamadas http no son instantáneas.

En este momento esta función se llamaría cada vez que el usuario cambiase una simple letra lo cual es un gasto enorme en llamadas y datos. Para evitar llamar a la función tan seguido, crearemos un método muy utilizado en programación llamado debounce que hará esperar a la llamada un tiempo antes de realizarse. Este tiempo lo situaremos en 300 ms para no ser excesivamente largo y no dar sensación de retraso.

El funcionamiento es simple, el *debouncer* tiene un cronómetro con el tiempo fijado en la función, si este llega a 0 se llama a la función searchCity. En caso de que el usuario escriba, se llamará a la función debouncer y se reactivará el cronómetro que volverá a empezar la cuenta.

5.5.4.2 Streams y StreamBuilder

Un Stream, en Flutter, es una secuencia de eventos asíncronos llamados snapshots, que vamos recibiendo poco a poco y con ello podemos ir modificando nuestro widget. Pero, para tener constancia de estos cambios, primero es necesario escucharlos.

En nuestro searchDelegate utilizaremos un StreamBuilder para construir nuestra lista de sugerencias, este necesitará recibir un Stream que le informará de los datos que van cambiando y con ellos construirá un ListView que mostrará todos los resultados.

El primer paso será la construcción de este Stream. Para ello iremos nuevamente a nuestro provider y crearemos una variable privada de tipo StreamController, como ocurría en el caso de GoogleMaps este controller se utiliza para controlar el funcionamiento del Stream. A nuestro StreamController habrá que indicarle el tipo de datos que recibirá para escuchar, lo cual serán List<cityModels> correspondientes a los resultados.

Este StreamController será inicializado como broadcast para poder ser escuchado por más widgets en el caso de ser necesario.

A continuación, crearemos un método get que devolverá un Stream, el cual será el que utilicemos ahora si en el StreamController.

Por último, construiremos nuestro ListView con ListTile personalizadas que muestren, el nombre, y el país ya que suelen haber ciudades con el mismo nombre alrededor del mundo.

5.5.4.3 Actualización del mapa

Por último, solo queda actualizar el mapa con la nueva localización recibida, cambiar el resultado de la appBar y actualizar el estado del TextButton de añadir ciudad.

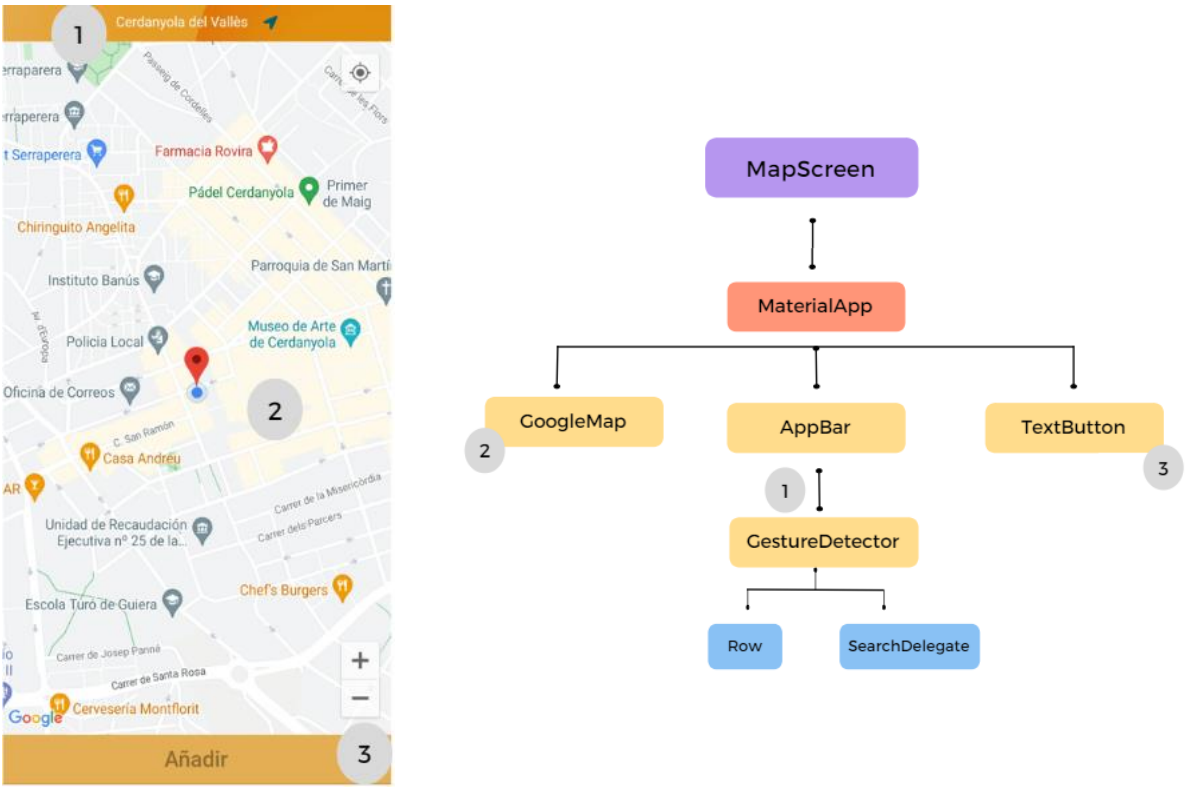


Figura 37. SearchScreen widget tree(Fuente propia)

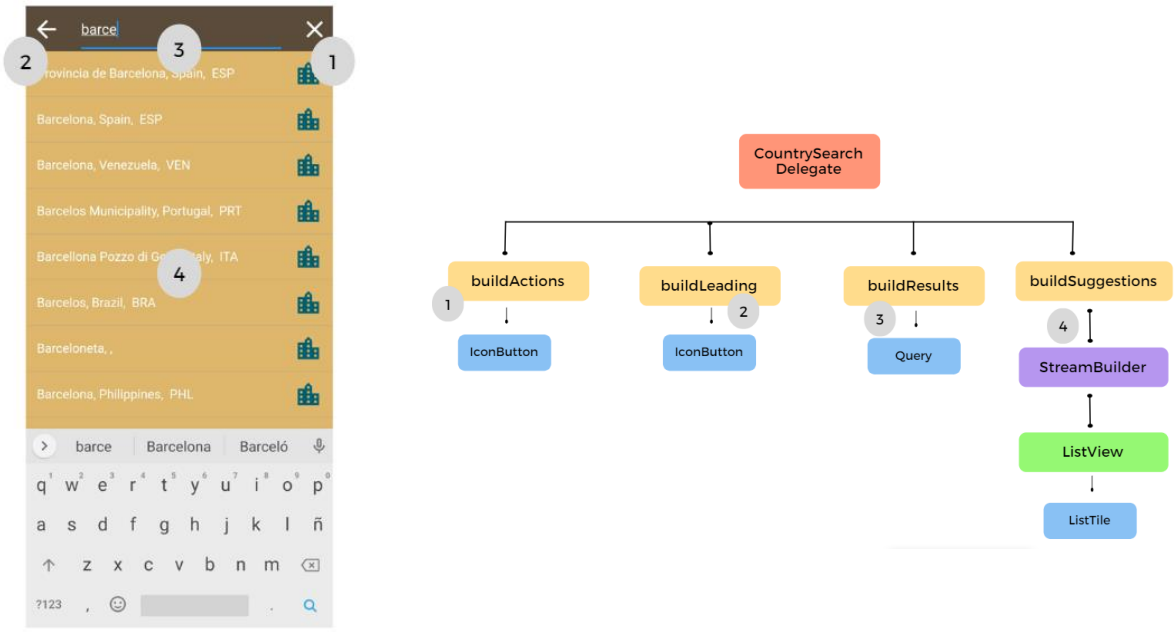


Figura 38. SearchDelegate widget tree(Fuente propia)

5.5.5 Borrado de favoritos

Solucionado el tema de añadir nuevas ciudades a nuestros favoritos, el último paso será implementar el método creado anteriormente en nuestro Provider que elimina una ciudad de nuestros favoritos.

Nos situaremos en la pantalla homeScreen y modificaremos el método onPress de nuestro IconButton favoritos. El método que añadimos proviene de nuestro Provider por lo tanto, recordamos que las llamadas a Provider en métodos se han de activar con la opción listen en false.

El borrado de datos de usuario en aplicaciones puede ser peligrosos, ya que puede ser difícil recuperar esa información o puede ser delicada. Por ello implementaremos un aviso a este método y la posibilidad de deshacer la acción. Para ello utilizaremos de nuevo un SnackBar, un widget muy útil para dar avisos, ya que los usuarios están acostumbrados a ellos y desaparecen rápidamente.

Un SnackBar puede contener cualquier tipo de widget dentro de él, en el caso del mapa únicamente colocamos un Text del aviso de la ciudad, pero en este caso aplicaremos el método de deshacer para lo cual utilizaremos un TextButton.

Este método llamará al método addLocation de nuestro Provider con las coordenadas del borrado anteriormente, dejando nuestros datos tal y como estaban.

5.6 Notificaciones

En este momento,nuestra aplicación es capaz de mostrar los mapas del tiempo, mostrar la información de una ciudad, tanto actual, como por horas y diarias. Además podemos añadir ciudades mediante búsqueda por mapa o escrita. El último paso en el desarrollo funcional de la aplicación es la creación de las notificaciones para dar aviso cuando la aplicación está en segundo plano.

Para ello debemos entender el ciclo de vida de una aplicación en Flutter. En Flutter una aplicación puede pasar por 4 estados:

- Detached: En este estado, nuestra aplicación está instalada en el dispositivo pero esta no está en uso ni pausada.
- Inactive: En este estado, la aplicación no recibe información del usuario, este estado es el correspondiente a cuando retrocedemos en la aplicación y aparecemos en otra aplicación que la teníamos en el background.
- Paused: La aplicación pasará a este estado cuando haya pasado cierto tiempo en Inactive. En este estado nuestra aplicación estará en background o segundo plano.
- Resumed: La aplicación está en uso y la tenemos en primer plano.

La idea de nuestras notificaciones es dar información al usuario cuando este pone la aplicación en segundo plano o paused, y que disponga de ella en su barra de notificaciones. En ningún momento avasallar al usuario con notificaciones cuando este tenga la aplicación cerrada por completo o esté haciendo uso de ella. Por lo consiguiente, trabajaremos con el estado Paused.

Empezaremos modificando la pantalla homeScreen, la cual necesitará heredar propiedades de la clase abstracta WidgetsBindingObserver. Las propiedades que nos otorgará esta clase serán la de controlar en qué estado se encuentra nuestra aplicación en todo momento.

Modificaremos el método didChangeAppLifecycleState para implementar toda nuestra lógica figura(nº39).

```
  void didChangeAppLifecycleState(AppLifecycleState state) {
    final CurrentWeatherProvider weatherProvider =
        Provider.of<CurrentWeatherProvider>(context, listen: false);

    super.didChangeAppLifecycleState(state);

    if (state == AppLifecycleState.detached ||
        state == AppLifecycleState.inactive) return;

    final isBackground = state == AppLifecycleState.paused;

    if (isBackground) {
      // aqui creare la notificación del tiempo


NotificationService().showNotifications(weatherProvider.weatherLocati
on!);
    }
  }
```

Figura 39. Método didChangeAppLifecycleState(Fuente propia)

- detached -> return
- inactive -> return
- paused -> salta notificación
- resumed -> nada

El último paso será crear las notificaciones, para ello utilizaremos la librería flutter_local_notifications. Modificaremos el archivo pubspec.yaml nuevamente para incluirla y comprobaremos en build.gradle tenemos la versión 21 que ya habíamos configurado anteriormente.

Como pasaba con el uso de la localización, el uso de notificaciones requiere permisos de usuario, con lo cual debemos informar a este para recibirlos.

Crearemos una nueva clase en nuestra carpeta de Services que implemente esa función, la cual llamaremos notification_service y será inicializada antes de la aplicación.

La implementación de esta clase requiere un conocimiento alto tanto de Flutter como de Android y la comunicación entre Flutter y el sistema operativo de nuestro dispositivo. Es por ello que la documentación de la librería incluye su implementación y nos genera los canales necesarios.

Nuestro trabajo será crear dentro de esta clase un método que cree estas notificaciones, el cual llamaremos showNotifications y será utilizada en la lógica anterior cuando nuestra aplicación esté en estado paused.

Este método recibe como parámetro de entrada la información del tiempo de nuestra localización y rellenará los datos que componen una notificación con ellos.

Estos datos serán:

- id: número de identificación de cada notificación. Siempre llevará el mismo número lo cual cada nueva notificación eliminará la anterior. En otras aplicaciones se suelen usar distintos números para cada tipo de notificaciones para evitar borrarlas o dar diferentes informaciones.
- body: El cuerpo de la notificación. En él mostraremos la ciudad y el tiempo actual.
- title: parte superior de la notificación. mostraremos la descripción del tiempo y la probabilidad de lluvia
- NotificacionsDetails: Dependiendo de la plataforma, se mostrará unos u otros. Estos están configurados en la documentación de la librería. Permite cambiar la importancia de la notificación, si suena o incluso si vibra. Dependiendo del grado de importancia la notificación saltará a la vista o se quedará en segundo plano con un Icon.
- payload: la información que queremos enviar a la pantalla que nos dirige la notificación, en nuestro caso ninguna ya que solo informa.

Una vez finalizada la creación de la notificación, el resultado al minimizar la aplicación será el de la figura(nº40).

Figura 40. Notificación WeatherApp(Fuente propia)

5.7 Pantalla de carga

Nuestra aplicación en este punto está lista para ser instalada, pero antes de ello puliremos algunos aspectos para hacerla más dinámica. Al iniciar la aplicación, en el punto actual se muestra una pantalla blanca que permanece un cierto tiempo dependiendo del procesador del dispositivo. Esta pantalla blanca la añade Flutter por defecto antes de inicializar la aplicación, para dar tiempo al material app de construir el widget tree. Para quitar este feo detalle, cambiaremos este fondo por uno que concuerde con el tema de la aplicación y crearemos una pequeña pantalla de inicio para dar la sensación de fluidez. Esta pantalla además ayudará a dar tiempo a Provider de cargar los datos.

El primer paso será cambiar el fondo por defecto. Este paso es delicado ya que hay que modificar archivos xml de Android. Iremos a Android>app>src>main>res>values y en el archivo styles.xml modificaremos android:windowBackground colocando el valor en hexadecimal de nuestro color. Una vez modificado este parámetro, la pantalla de inicio debería verse del color seleccionado.

Por último crearemos la pantalla que se mostrará a continuación. Crearemos el archivo nuevo y lo llamaremos loadScreen, también crearemos su *route* en nuestro materialApp.

La idea de esta pantalla es albergar una animación de entrada y una vez termine esta, dirigirnos a la homeScreen.

Crearemos dos tipos de animación. La primera mostrará el nombre de la aplicación empezando la animación con la primera letra. Para ello utilizaremos el widget AnimatedDefaultTextStyle, que permite hacer una animación del estilo de texto. Con este efecto cambiaremos el tamaño de nuestra letra para que vaya minimizandose y el resto del texto aparezca.

Crearemos una variable que nos indique si la letra está expandida o no, con está condición modificaremos que elemento se verá.

- Expanded vale *true*, se mostrará un container vacío a continuación de la letra expandida, el cual al tener tamaño mínimo no se verá.
- Expanded vale *false*, se mostrará el resto del text.

El cambio o transición de un estado a otro es la parte que se animará.

La segunda animación, será una animación creada por una librería exterior. La web Lottiefiles.com tiene una librería propia para Flutter que permite el uso de estos dentro de nuestra aplicación. Haremos uso de ella para controlar la duración y anular su repetición.

El último paso será crear un método que controle los tiempos de cada animación y una vez terminadas nos envíe hacia la siguiente pantalla figura(nº41).

```
    Future.delayed(const Duration(milliseconds: 300))
        .then((value) => setState(() => expanded = true))
        .then((value) => const Duration(milliseconds: 300))
        .then(
          (value)   =>   Future.delayed(const   Duration(milliseconds:
300)).then(
            (value) => _lottieAnimation.forward().then(
                  (value) => Navigator.of(context).pushAndRemoveUntil(
                      MaterialPageRoute(
                          builder: (context) => const HomeScreen()),
                      (route) => false),
                ),
          ),
        );
```

Figura 41. Animación LoadScreen(Fuente propia)

1. Espera 300 ms
2. Cambia el estado de expanded i activa a la animación
3. Espera 300 ms
4. Activa la animación Lottie
5. Acaba la animación y nos manda a la HomeScreen

el resultado final de la pantalla de carga será el de la figura(nº42)

Figura 42. LoadScreen(Fuente propia)

5.8 Test y bugs

El paso final del desarrollo de una aplicación son los test para comprobar fallos tanto en la funcionalidad como en el diseño. Primero comprobaremos fallos en el diseño y para ello utilizaremos dos modelos de móviles cuyas características están en la tabla(nº22).

	Xiomi redmi note 9 Pro	huawei P10 lite
Pantalla	6,53"	5,2"
Android	11.0.0	8.0.0
Procesador	MediaTek Helio G85 Octa-	Kirin 658 Octa Core 4×2.1 GHz y 4×1.7 GHz.
Memoria RAM	6GB	4GB

Tabla 22. Especificaciones modelos(Fuente propia)

Un diseño de móvil óptimo ha de ajustarse a cualquier modelo de móvil, por ello he escogido modelos con una diferencia amplia en el tamaño de la pantalla y la versión Android utilizada. Esto nos ayudará a comprobar si versiones antiguas y con menos procesador soporta los mapas implementados (van fluidos y cargan sin problema), si marcas con diferentes capas de Android manejan los permisos correctamente y si las gráficas, fotos y diversos widgets implementados se ajustan sin fallo alguno.

Se ha comprobado satisfactoriamente que tanto en la versión 7.0 de nuestro Huawei como en la versión 11 de Xiaomi Redmi Note 9, el utilizado durante todo el proceso, los mapas funciona correctamente y los permisos se ejecutan perfectamente, además la pantalla se ajusta a las 5.2 pulgadas de la pantalla de Huawei, si fallas en los gestureDetectors ni elementos fuera de pantalla. Además elementos como la barra de búsqueda y la muestra de sugerencias también se han adaptado bien y no se superponen en pantallas reducidas.

Para comprobar errores en las funcionalidades de la aplicación, estas se han centrado en la búsqueda tanto escrita como por mapa. Se han comprobado cientos de países para ver el comportamiento de nuestro autoComplete y el parse de coordenadas a ciudad.

Se ha visto que la búsqueda por texto funciona perfectamente con cualquier país, pero en búsqueda por mapa, ciudades de Inglaterra muestran pequeños errores en el modelo de cityModel, ya que no muestra el nombre de ciudad, en estos casos, se ha seleccionado que se muestre la calle y el distrito ya que este fallo solo se ha visto en el País Inglés.

El último pequeño error, sin influencia sobre la aplicación se ha encontrado en países de África, donde el modelo de datos OneCallweather no recibe datos por minuto ya que no disponen de ellos. Como es un dato que no utilizabamos en la aplicación y lo reservamos como datos extra ya que solo aporta información de 1h, se ha suprimido este dato del modelo.

Por último se ha comprobado la robustez de la API con diversas llamadas simultáneas en diferentes dispositivos la cuales se han recibido correctamente.

5.9. Resultado final

La figura(nº43) muestra el resultado final de la aplicación desarrollada a lo largo del proyecto. En ella podemos ver las distintas pantallas que esta ofrece.

Figura 43. Modelo final(Fuente propia)

El código fuente de la aplicación está disponible para su uso y modificación en este enlace:

https://github.com/Mecmecker/weatherapp

6 Resumen del presupuesto y/o estudio de viabilidad económica

El presupuesto de este proyecto está dividido en dos tipos:

- Humano: Salario del empleado y horas trabajadas.
- Material: Precio software y hardware, oficina si es necesaria, luz.

Para el cálculo del salario del empleado se cogerá como estándar el salario que la UPC considera como mínimo en el convenio de prácticas.

6.1 Gastos humanos

El salario se ha calculado basándose en el precio/hora propuesto por la UPC para becarios, ya que al no disponer de título, el salario no puede ajustarse al convenio de ingenieros.

El sueldo/hora propuesto por la UPC es de 8€. El número de horas mínimas para la creación del proyecto se sitúa en unas 600 horas. A estas hay que sumarle horas extras debido a bugs y/o complicaciones que suponen retrasos en las entregas.

El coste humano final se refleja en la gráfica siguiente:

Gastos humanos			
Tipo	cantidad	precio/h	horas
Becario	1	8	800
			total €
			6400

Tabla 23. Gastos humanos

6.2 Gastos materiales

En estos gastos entran tanto la compra del ordenador para llevar a cabo el proyecto como licencias, gastos de luz y oficinas.

Como hardware entrará la compra del ordenador portátil utilizado Asus Zenbook y el móvil para desarrollar la aplicación en tiempo real.

Gastos Hardware		
Tipo	cantidad	precio (€)
Portatil Asus	1	800
Movil Xiomi	1	200
		total
		1000

Tabla 24. Gastos Hardware

En el caso de este proyecto, todas las licencias son gratuitas, se ha utilizado software opensource, y las API, a pesar de requerir claves, estas han sido gratuitas, por lo que el gasto en software es nulo.

Gastos Software		
Tipo	Cantidad	Precio (€)
Flutter SDK	1	0
Android Studio	1	0

Keys		
Tipo	Cantidad	Precio (€)
API key openweather	1	0
API key geocodeapi	1	0

Tabla 25. Gastos herramientas

En el apartado de oficinas, no se han utilizado oficinas privadas, ni salas especiales. Todo se ha desarrollado en casa propia por lo que este gasto es nulo.

Gastos Infraestructura				
Tipo	Cantidad	Precio (€)	Meses	Total (€)
Oficina	1	0	6	0

Tabla 26. Gastos infraestructuras

En el apartado de luz, este año, el precio de esta ha crecido exponencialmente, aparte de contener franjas donde varía el precio, lo que afecta al consumo final. Por lo tanto, se ha tomado el precio de franja mas alto, donde se ha trabajado la mayoría del tiempo y el precio mas alto alcanzado de la luz.

Gastos Energeticos				
Tipo	Precio/Kwh	Kwh mes	meses	Total (€)
Luz	0,5€/Kwh	186	6	558

Tabla 27. Gastos energéticos

6.3 Gastos totales

El coste final de este proyecto se refleja en la tabla(). En ella vemos que el gasto total alcanza los 8000€, donde la gran mayoría del presupuesto se destina a gastos humanos.

Gastos Totales	
Tipo	Total (€)
Gastos humanos desarrollo	6400
Gastos material	1000
Gastos energeticos	558
Gastos totales realización proyecto	7958

Tabla 28. Gastos totales

7 Conclusiones

El desarrollo de este proyecto me ha ayudado a entender más profundamente Flutter. Por una parte, la gestión de estados, estudiando Provider y entendiendo su funcionamiento, y por la otra, la comunicación de Flutter con el sistema operativo de nuestro dispositivo, para el uso de funcionalidades tales como localización o notificaciones.

En cuanto a los objetivos planteados al inicio de este proyecto puedo concluir que se han logrado cada uno de ellos, cambiando el método en alguno, ya sea por costes o por sorpresas inesperadas, pero logrando el objetivo deseado.

- Se han diseñado cada una de las pantallas planteadas al inicio del proyecto.
- Se han plasmado los datos recibidos en gráficas o elementos visualmente atractivos.
- Se dispone de imágenes en función del tiempo
- Se ha logrado implementar notificaciones que informan al usuario cuando la aplicación está en segundo plano.
- Se han añadido mapas, tanto para reflejar el tiempo como para la búsqueda de ciudad.
- Se ha implementado un autocomplete como ayuda para la búsqueda de ciudades.

En cuanto a trabajo futuro, las mejoras girarán en torno a tres puntos:

- Creación de una versión IOs. Gracias a Flutter el código base es el mismo, el cambio vendrá en los archivos nativos y licencias necesarias, en caso de Apple, con coste económico.
- Nuevo diseño más profesional, creado por diseñadores.
- Nuevos tipos de notificaciones, riesgos y alertas.

Dependiendo de las nuevas funcionalidades de la API openweather.org también se podrían seguir otras rutas de mejora junto a propuestas realizadas por los usuarios.

La aplicación en este punto está preparada para ser subida en la PlayStore. La razón por la cual aún no ha sido lanzada radica en el hecho de que el proyecto aún no ha sido presentado, la versión IOS no ha sido creada y la licencia que permite su lanzamiento ha dejado de ser gratuita, quedando así abierta una línea de trabajo a desarrollar en el futuro.

8 Referencias

https://bibliotecnica.upc.edu/investigadors/citar-elaborar-bibliografia

https://es.statista.com/estadisticas/635543/cuota-de-mercado-mundial-de-sistemas-operativos-de-smartphones-desde-2009-hasta--por-trimestre/

https://www.androidauthority.com/why-developers-choose-android-285774/

https://www.bbvanexttechnologies.com/blogs/es-flutter-el-framework-del-futuro

https://stackoverflow.com/a/52922130/7834829

https://pub.dev/packages/flutter_local_notifications

https://blog.logrocket.com/implementing-local-notifications-in-flutter/

https://github.com/MaikuB/flutter_local_notifications/blob/master/flutter_local_notifications/example/android/app/src/main/AndroidManifest.xml

https://developer.android.com/guide/topics/ui/notifiers/notifications#Heads-up

https://openweathermap.org/api

https://pub.dev/packages/flutter_map

https://leafletjs.com/reference.html

https://pub.dev/packages/google_maps_flutter

https://www.youtube.com/c/FernandoHerreraCr