



TRABALLO FIN DE MÁSTER  
MÁSTER UNIVERSITARIO EN ENXEÑARÍA INFORMÁTICA

**Línea de producto software para  
aplicaciones móviles nativas en el ámbito  
de la gestión del trabajo en movilidad**

**Estudiante:** Raquel Díaz Otero  
**Dirección:** Miguel Ángel Rodríguez Luaces  
**Dirección:** Alejandro Cortiñas Álvarez

A Coruña, xuño de 2021.



*A mi familia.*



### **Agradecimientos**

A mi familia, por su apoyo incondicional durante todo este tiempo y por haberme permitido llegar hasta aquí.

A mis tutores, Alejandro y Miguel, por la ayuda, la atención y la motivación que me han proporcionado en todo momento.

Y a todas las personas que me han acompañado durante esta etapa de mi vida.



## **Resumen**

El objetivo de este trabajo de fin de máster es desarrollar una aplicación móvil nativa para la gestión del trabajo en movilidad que se integre dentro de la línea de producto software del proyecto GEMA, basada en *scaffolding* y en el desarrollo dirigido por modelos, con el fin de proporcionar un resultado que permita a las empresas de este ámbito obtener productos adaptados a sus necesidades de negocio.

Para alcanzar este objetivo fue necesario en primer lugar realizar un análisis del proyecto con el fin de definir los requisitos y determinar la variabilidad de la LPS. Posteriormente, se llevó a cabo el análisis, diseño, implementación y prueba de todas las funcionalidades de la aplicación y su integración dentro de la línea de producto software.

En el desarrollo del producto se empleó spl-js-engine como motor de derivación para la generación de productos y Flutter para la implementación de la aplicación nativa. También se hizo uso del servidor proporcionado por la plataforma GEMA y del sistema de gestión de bases de datos PostgreSQL.

El trabajo de fin de máster se gestionó siguiendo una metodología iterativa e incremental, por lo que el proyecto se dividió en varias iteraciones en cada una de las cuales se llevó a cabo el desarrollo de un conjunto de funcionalidades.

## **Abstract**

The objective of this end-of-master project is to develop a native mobile application for mobility work management to be integrated into the software product line of the GEMA project, based on *scaffolding* and on model-driven development (MDD), in order to provide a final result that allows companies related with this field of work to obtain products adapted to their business needs.

In order to achieve this goal, it was necessary first of all to carry out an analysis of the project in order to define the requirements and to determine the variability of the SPL. Subsequently, the analysis, design, implementation and testing of the functionalities defined for the application was carried out, as well as its integration within the software product line.

In the development process, spl-js-engine was used as a derivation engine for product generation and Flutter was used for the implementation of the native application. The server provided by the GEMA platform and the PostgreSQL database management system were also used.

---

The project was managed following an iterative and incremental methodology, so the development process was divided into several iterations in which was performed the development of a set of functionalities.

**Palabras clave:**

- Generación de código
- Línea de producto software
- Desarrollo dirigido por modelos
- Gestión de la movilidad
- Geolocalización
- Aplicación nativa
- Multiplataforma
- Flutter
- Dart
- GitLab

**Keywords:**

- Code generation
- Software product line
- Model Driven Development
- Mobility management
- Geolocation
- Native application
- Cross-platform
- Flutter
- Dart
- GitLab



# Índice general

---

<b>1</b>	<b>Introducción</b>	<b>1</b>
1.1	Motivación . . . . .	1
1.2	Objetivos . . . . .	2
<b>2</b>	<b>Fundamentos tecnológicos</b>	<b>3</b>
2.1	Estado del arte . . . . .	3
2.2	Tecnologías utilizadas . . . . .	13
<b>3</b>	<b>Metodología y planificación</b>	<b>15</b>
3.1	Metodología de desarrollo . . . . .	15
3.1.1	Equipo Scrum . . . . .	16
3.1.2	Eventos Scrum . . . . .	16
3.1.3	Artefactos Scrum . . . . .	17
3.1.4	Herramientas de soporte a la metodología . . . . .	18
3.2	Metodología de desarrollo de la LPS . . . . .	20
3.3	Planificación y seguimiento . . . . .	22
3.3.1	Recursos . . . . .	22
3.3.2	Planificación . . . . .	23
3.3.3	Seguimiento . . . . .	25
<b>4</b>	<b>Análisis de la línea de producto software</b>	<b>29</b>
4.1	Funcionamiento de la LPS . . . . .	29
4.2	Modelos de datos genérico . . . . .	31
4.3	Modelado de la variabilidad de la LPS . . . . .	32
<b>5</b>	<b>Análisis</b>	<b>35</b>
5.1	Actores . . . . .	35
5.2	Requisitos . . . . .	36

5.2.1	Requisitos funcionales . . . . .	36
5.2.2	Requisitos no funcionales . . . . .	36
5.2.3	Pila del producto . . . . .	37
5.3	Arquitectura del sistema . . . . .	39
5.4	Interfaz de usuario . . . . .	41
5.5	Modelo conceptual de datos . . . . .	47
<b>6</b>	<b>Diseño</b>	<b>49</b>
6.1	Arquitectura tecnológica del sistema . . . . .	49
6.1.1	Aplicación nativa . . . . .	49
6.1.2	Base de datos y servidor . . . . .	51
6.2	Diseño de la aplicación . . . . .	51
6.2.1	Entidades . . . . .	54
6.2.2	Repositorios . . . . .	54
6.2.3	Controladores . . . . .	55
6.2.4	Vistas . . . . .	55
6.2.5	Estructura de clases . . . . .	55
<b>7</b>	<b>Implementación y pruebas</b>	<b>59</b>
7.1	Implementación . . . . .	59
7.1.1	Entidades . . . . .	59
7.1.2	Widgets . . . . .	65
7.1.3	Mapa . . . . .	75
7.2	Pruebas . . . . .	81
<b>8</b>	<b>Construcción de la línea de producto software</b>	<b>83</b>
8.1	Arquitectura . . . . .	83
8.1.1	Motor de derivación . . . . .	84
8.2	Implementación . . . . .	90
8.3	Integración de la aplicación . . . . .	93
8.3.1	Pruebas . . . . .	93
<b>9</b>	<b>Solución desarrollada</b>	<b>95</b>
9.1	Aplicación móvil . . . . .	95
9.2	Aplicación multiplataforma . . . . .	103
<b>10</b>	<b>Conclusiones y trabajo futuro</b>	<b>107</b>
<b>A</b>	<b>Diagrama del modelo de características de GEMA</b>	<b>113</b>

## **ÍNDICE GENERAL**

---

<b>B</b>	<b>Prototipos de pantallas</b>	<b>115</b>
<b>C</b>	<b>Manual de instalación</b>	<b>129</b>
C.1	Base de datos y servidor . . . . .	129
C.2	Aplicación nativa . . . . .	130
<b>D</b>	<b>Glosario de acrónimos</b>	<b>131</b>
<b>E</b>	<b>Glosario de términos</b>	<b>133</b>
	<b>Bibliografía</b>	<b>135</b>



# Índice de figuras

---

2.1	Pantallas de la aplicación GOME [1] . . . . .	4
2.2	Pantallas de la aplicación Worker App [2] . . . . .	5
2.3	Pantallas de la aplicación Worker App [2] . . . . .	6
2.4	Pantallas de la aplicación Krama GOT [3] . . . . .	6
2.5	Ejemplo de composición de árboles de características con FeatureHOUSE [4] . . . . .	7
2.6	Ejemplo de composición con AHEAD [5] . . . . .	8
2.7	Ejemplo de código anotado con CIDE [6] . . . . .	8
2.8	Interfaz de comandos de Yeoman [7] . . . . .	9
2.9	Ejemplo de modelado con Rational Software Architect en Eclipse . . . . .	10
2.10	Ejemplo de generación de código con Rational Software Architect en Eclipse . . . . .	11
2.11	Ejemplo de definición de un modelo con Acceleo . . . . .	12
3.1	Tablero de Gitlab . . . . .	19
3.2	Registro de información en una <i>issue</i> . . . . .	20
3.3	Metodología seguida para el desarrollo de la LPS . . . . .	21
3.4	Diagrama de Gantt de planificación . . . . .	24
3.5	Diagrama de Gantt de seguimiento . . . . .	28
4.1	Árbol de características de GEMA . . . . .	30
4.2	Operaciones de GEMA . . . . .	30
4.3	Editor del modelo de datos de GEMA . . . . .	31
4.4	Modelo genérico de datos . . . . .	32
4.5	Modelo de características de la LPS . . . . .	34
5.1	Actores . . . . .	35
5.2	Arquitectura general del sistema . . . . .	40
5.3	Menú de la aplicación sin usuario autenticado y con usuario autenticado . . . . .	41
5.4	Pantallas de iniciar sesión y de recuperar la contraseña . . . . .	42

5.5	Pantallas de lista con búsqueda y de lista con filtros y <i>popup</i> . . . . .	43
5.6	Detalle y formulario de una entidad . . . . .	44
5.7	Pantallas de mapa con <i>popup</i> y de mapa editando . . . . .	45
5.8	Pantallas de calendario de visitas y de mapa con información de una visita . .	46
5.9	Modelo conceptual de datos . . . . .	48
6.1	Arquitectura tecnológica del sistema . . . . .	50
6.2	Estructura de paquetes de la aplicación . . . . .	52
6.3	Diagrama de clases de una entidad . . . . .	56
6.4	Diagrama de clases de visita . . . . .	56
6.5	Diagrama de clases de usuario . . . . .	57
6.6	Diagrama de clases de cliente y empleado . . . . .	57
8.1	Arquitectura de la línea de producto software . . . . .	84
8.2	Esquema del motor de derivación <i>spl-js-engine</i> . . . . .	85
8.3	Estructura del fichero de especificación de un producto . . . . .	86
8.4	Ejemplo de la propiedad <i>features</i> en el fichero de especificación de un producto	87
8.5	Ejemplo de la propiedad <i>basicData</i> en el fichero de especificación de un producto	87
8.6	Ejemplo de la propiedad <i>enums</i> en el fichero de especificación de un producto	88
8.7	Ejemplo de la propiedad <i>entities</i> en el fichero de especificación de un producto	88
8.8	Ejemplo de la propiedad <i>forms</i> en el fichero de especificación de un producto .	89
8.9	Ejemplo de la propiedad <i>lists</i> en el fichero de especificación de un producto .	90
8.10	Configuración de los delimitadores de las anotaciones . . . . .	91
9.1	Pantalla inicial y menú sin usuario autenticado . . . . .	96
9.2	Pantallas de inicio de sesión y de recuperación de contraseña . . . . .	96
9.3	Menú con usuario autenticado . . . . .	97
9.4	Pantallas de listados y filtros . . . . .	97
9.5	Pantallas de listado con <i>popup</i> y de formulario de creación . . . . .	98
9.6	Pantalla inicial y menú sin usuario autenticado . . . . .	99
9.7	Formularios de detalle y de edición, y alerta de confirmación de borrado . .	100
9.8	Vistas del mapa de edición . . . . .	101
9.9	Pantallas de lista, calendario y detalle de las visitas . . . . .	101
9.10	Vistas del mapa de visitas . . . . .	102
9.11	Formulario de detalle de una entidad . . . . .	104
9.12	Formulario de edición de una entidad . . . . .	104
9.13	Mapa de edición . . . . .	105
9.14	Lista de visitas . . . . .	105

## ÍNDICE DE FIGURAS

---

9.15 Mapa de visitas . . . . .	106
A.1 Modelo de características de la LPS de GEMA . . . . .	114
B.1 Pantalla inicial . . . . .	116
B.2 Menú lateral sin usuario autenticado . . . . .	116
B.3 Pantalla de iniciar de sesión . . . . .	117
B.4 Pantalla de recuperar la contraseña . . . . .	117
B.5 Menú lateral con usuario autenticado . . . . .	118
B.6 Pantalla de lista con búsqueda simple y ordenación . . . . .	118
B.7 Pantalla de lista con filtros y ordenación . . . . .	119
B.8 Pantalla de lista con <i>popup</i> en un elemento . . . . .	119
B.9 Alerta de confirmación de borrado . . . . .	120
B.10 Pantalla de filtros de una entidad . . . . .	120
B.11 Pantalla de formulario de una entidad . . . . .	121
B.12 Pantalla de detalles de una entidad . . . . .	121
B.13 Pantalla de ordenación . . . . .	122
B.14 Pantalla de mapa . . . . .	122
B.15 Pantalla de información de un elemento en el mapa . . . . .	123
B.16 Pantalla de editar en el mapa . . . . .	123
B.17 Pantalla de mapa editando . . . . .	124
B.18 Pantalla de lista de visitas . . . . .	124
B.19 Pantalla de calendario de visitas . . . . .	125
B.20 Pantalla de modificación del tipo de vista en el calendario de visitas . . . . .	125
B.21 Pantalla de filtros en el calendario de visitas . . . . .	126
B.22 Pantalla de detalle de una visita . . . . .	126
B.23 Pantalla de formulario de una visita . . . . .	127
B.24 Pantalla de mapa de visitas . . . . .	127
B.25 Pantalla de información de una visita en el mapa . . . . .	128
B.26 Pantalla de filtros en el mapa de visitas . . . . .	128
C.1 Configuración de la IP del servidor en la aplicación . . . . .	130



# Índice de cuadros

---

3.1	Salarios por hora de cada rol . . . . .	24
3.2	Costes estimados para el proyecto . . . . .	25
3.3	Costes reales del proyecto . . . . .	28



# Capítulo 1

# Introducción

---

En este capítulo se detalla la motivación que ha conducido a la realización de este proyecto y los objetivos principales definidos para este.

## 1.1 Motivación

En el ámbito de las empresas con trabajadores en movilidad (empresas que prestan servicios fuera de sus instalaciones, cuyos trabajadores se desplazan para atender a los clientes) han surgido un conjunto de aplicaciones conocidas como aplicaciones de Gestión del Trabajo en Movilidad (GTM). Estas aplicaciones normalmente proporcionan un conjunto de funcionalidades similares, centradas en la gestión de tareas de los trabajadores en movilidad, pero cubriendo aspectos como la gestión de horarios, rutas y eventos, o la detección de actividades de cada trabajador. Del lado del trabajador se hace necesaria una aplicación que pueda usar en un dispositivo móvil, y que permita a cada empleado ver la lista de sus tareas planificadas, ver la posición geográfica donde se realizan, ver los datos de los clientes asociados a dichas tareas, utilizar funcionalidades de cálculo de rutas y/o navegación para llegar al destino, poder registrar una tarea como realizada (u otro estado), etc.

En este entorno, y teniendo en cuenta que las funcionalidades de este tipo de aplicaciones son similares, el Laboratorio de Bases de Datos de la UDC ha diseñado una plataforma de generación automática de este tipo de aplicaciones basada en el paradigma de Líneas de Producto Software (LPS): el proyecto GEMA [8]. Actualmente, esta plataforma parte de la especificación de un producto concreto, que incluye una selección de funcionalidades y la definición de un modelo de datos, y genera el código fuente de un servidor REST y de un cliente web, diseñado siguiendo el paradigma Progressive Web Application (PWA). Puesto que no siempre es eficiente el uso de tecnologías web en móviles, es importante que los productos generados incluyan también una aplicación móvil nativa que soporte el conjunto de funcionalidades de la plataforma.

El objetivo principal de este trabajo de fin de máster es desarrollar una aplicación móvil nativa que soporte funcionalidades de gestión de movilidad y que se integre dentro de la línea de producto software de GEMA. Con esto se pretende proporcionar un mecanismo que permita generar aplicaciones móviles nativas totalmente adaptadas al dominio y necesidades concretas de cada empresa, cubriendo de esta forma la necesidad de adaptación de los productos a cada caso particular, optimizando los recursos disponibles y disminuyendo la complejidad de las aplicaciones.

## 1.2 Objetivos

Para alcanzar el objetivo principal del proyecto, se deben alcanzar los siguientes objetivos específicos:

- **Definir las características** que formarán parte de las aplicaciones generadas seleccionando un subconjunto de las características que forman parte de la plataforma actual (GEMA) e incluyendo características propias de los entornos nativos relacionadas con la gestión del trabajo en movilidad. En términos generales, la aplicación deberá contar con: funcionalidades para la gestión de usuarios; formularios para la visualización, creación, modificación y borrado de las entidades del dominio; listados y búsquedas; y funcionalidades de geolocalización y mapas.
- **Desarrollar el producto**, esto es, una aplicación móvil nativa para la gestión del trabajo en movilidad que cuente con todas las características definidas.
- **Integrar la aplicación nativa en la línea de producto software (LPS)**, de forma que se permita la generación de aplicaciones a partir de un conjunto de características.
- **Generar el código fuente** de las aplicaciones nativas desarrolladas mediante el uso del motor de derivación de líneas de producto software de la plataforma.
- Proporcionar **productos generados de calidad**, que no tengan errores conocidos, para lo que deben realizarse pruebas de la aplicación.
- Proporcionar **productos generados seguros**, que permitan el acceso a los datos solo a los usuarios autorizados.
- Proporcionar **productos generados multiplataforma**, que sean compatibles con múltiples dispositivos y sistemas operativos (Android, IOS, Web, etc.).

## Capítulo 2

# Fundamentos tecnológicos

---

En este capítulo se describen los fundamentos tecnológicos sobre los que se ha basado el desarrollo del proyecto.

### 2.1 Estado del arte

En base a los objetivos definidos para el proyecto y teniendo en cuenta que el proyecto está compuesto por una aplicación para la gestión del trabajo en movilidad y por una línea de producto software, se ha realizado una búsqueda de herramientas existentes en el mercado que pudiesen cubrir las necesidades expresadas para cada uno de estos elementos.

Por una parte, se han buscado **aplicaciones para la gestión del trabajo en movilidad**, entre las cuales se han encontrado las siguientes alternativas:

- **GOME [1]**: herramienta software para la gestión y planificación de órdenes de trabajo. Esta herramienta está formada por dos elementos principales: una aplicación web destinada a la administración del trabajo y la flota, y una aplicación móvil destinada a los trabajadores de campo. Desde el punto de vista de la aplicación móvil, ofrece una serie de funcionalidades entre las que se pueden destacar las siguientes:
  - Permite la gestión de las órdenes de trabajo en tiempo real, recibiendo notificaciones de las nuevas actualizaciones. Proporciona mecanismos para la visualización de los trabajos en forma de lista y geolocalizados en el mapa.
  - Proporciona un mecanismo de monitorización en tiempo real de los servicios, permitiendo el registro del estado de los trabajos en todo momento.
  - Proporciona un mecanismo de localización GPS que permite conocer la posición de cada empleado en tiempo real.

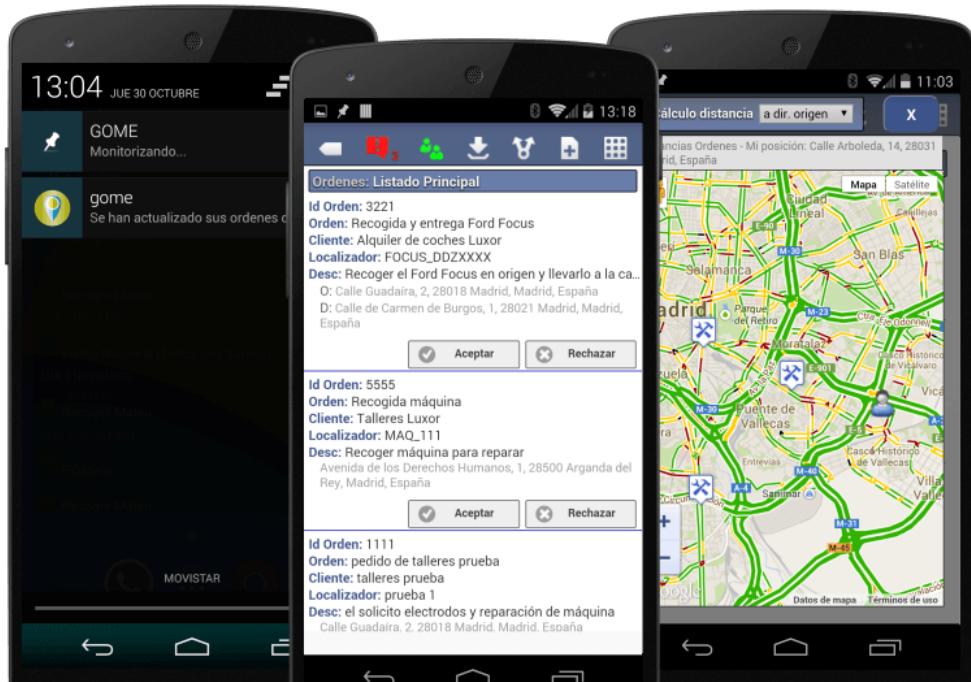


Figura 2.1: Pantallas de la aplicación GOME [1]

- Tiene soporte para datos digitales, firmas y notas, lo que permite a los empleados capturar fotos y recoger las firmas de los clientes. Estos datos son enviados en tiempo real.
- Soporta el funcionamiento *offline*, de forma que la aplicación mantiene los datos en el dispositivo móvil hasta que se recupere la conexión.

En la figura 2.1 se muestran algunas de las vistas de la aplicación.

- **Worker App** [2]: aplicación móvil para la gestión de partes y órdenes de trabajo. Esta aplicación proporciona las siguientes funcionalidades:

- Emisión de órdenes de trabajo y tareas.
- Creación de partes de trabajo.
- Geolocalización en tiempo real.
- Soporte para añadir descripciones, fotografías y firmas digitales de los clientes a las tareas.
- Gestión de horas trabajadas.
- Gestión de materiales y gastos.

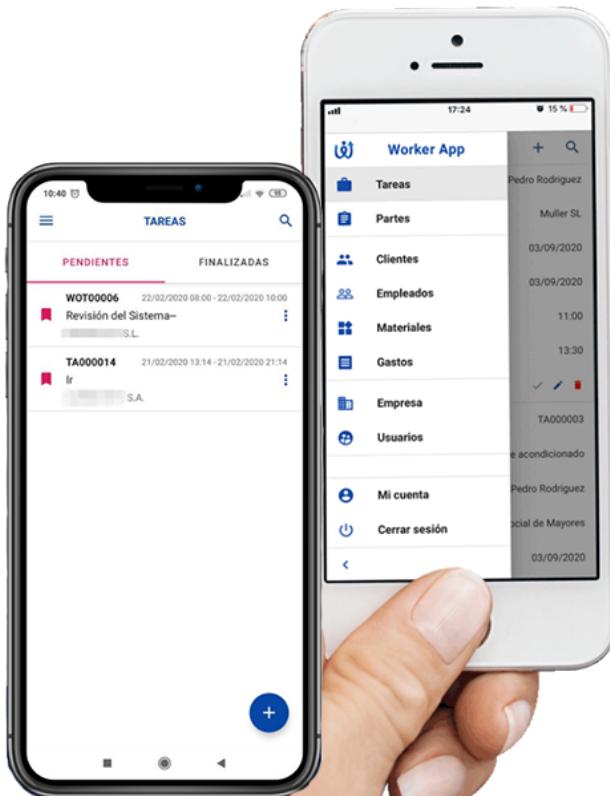


Figura 2.2: Pantallas de la aplicación Worker App [2]

- Extracción de informes por trabajador o cliente.
- Funcionamiento *offline*.

En las figuras 2.2 y 2.3 se muestran algunas de las vistas de esta aplicación.

- **Krama GOT** [3]: herramienta que permite gestionar el trabajo de empleados en movilidad como técnicos, repartidores, comerciales, etc., en multitud de tareas como instalaciones, mantenimiento, avisos urgentes, ventas o visitas comerciales, etc. Esta herramienta está compuesta de dos elementos principales: una aplicación web para la administración y asignación de tareas, y una aplicación móvil para la gestión de las tareas asignadas a los trabajadores. En cuanto a la aplicación móvil, el trabajador puede ver su agenda de tareas pendientes, localizar en un mapa su ubicación, utilizar el GPS para acudir a sus destinos, reportar la realización de cada tarea o dar de alta incidencias. Además, permite elaborar formularios personalizados y tiene notificaciones en tiempo real. En la figura 2.4 se muestran algunas de las vistas de la aplicación.

Por otra parte, en cuanto a la generación de productos mediante la línea de producto software, se ha realizado una búsqueda de **herramientas de derivación de líneas de producto**

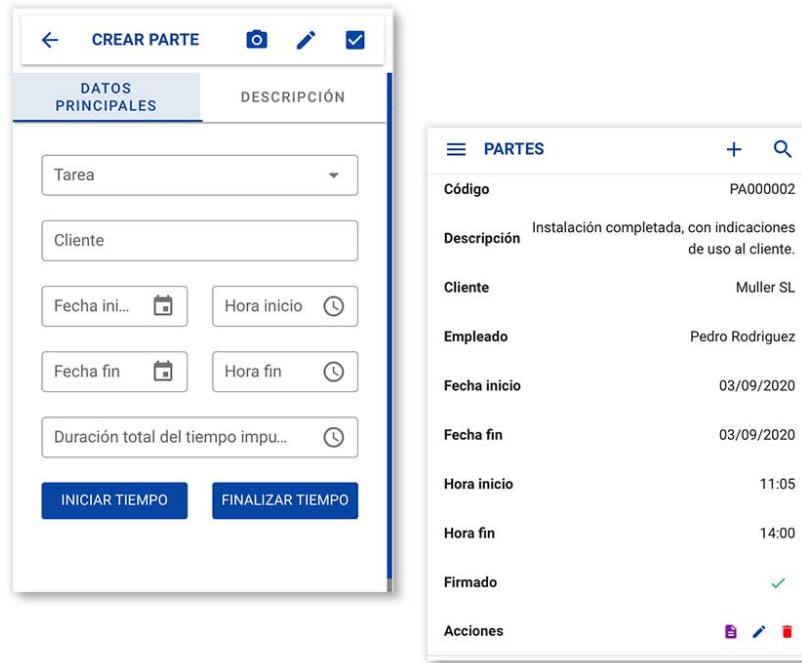


Figura 2.3: Pantallas de la aplicación Worker App [2]

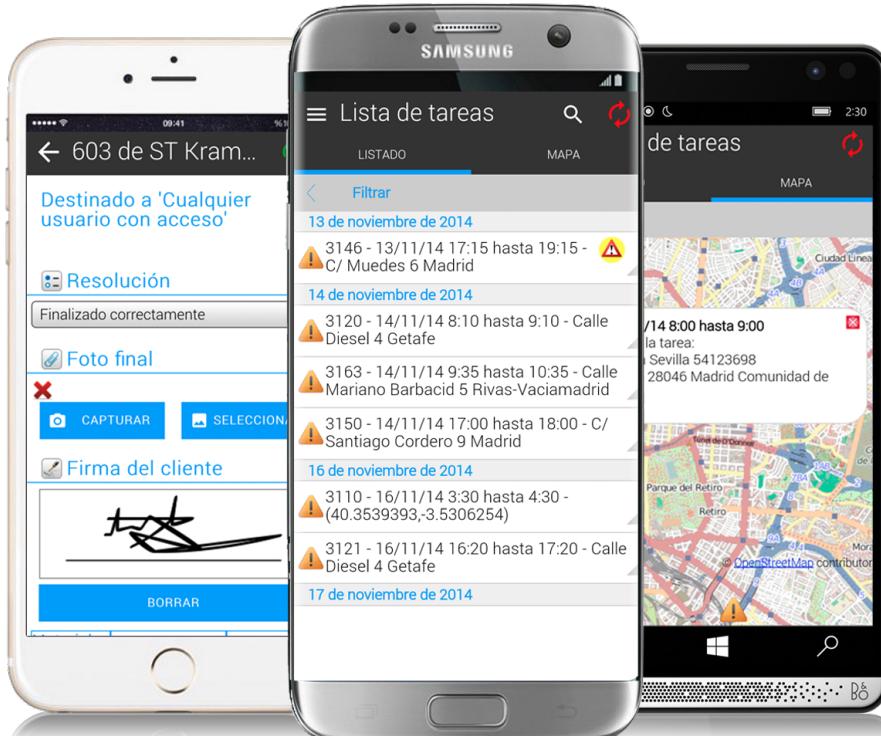


Figura 2.4: Pantallas de la aplicación Krama GOT [3]

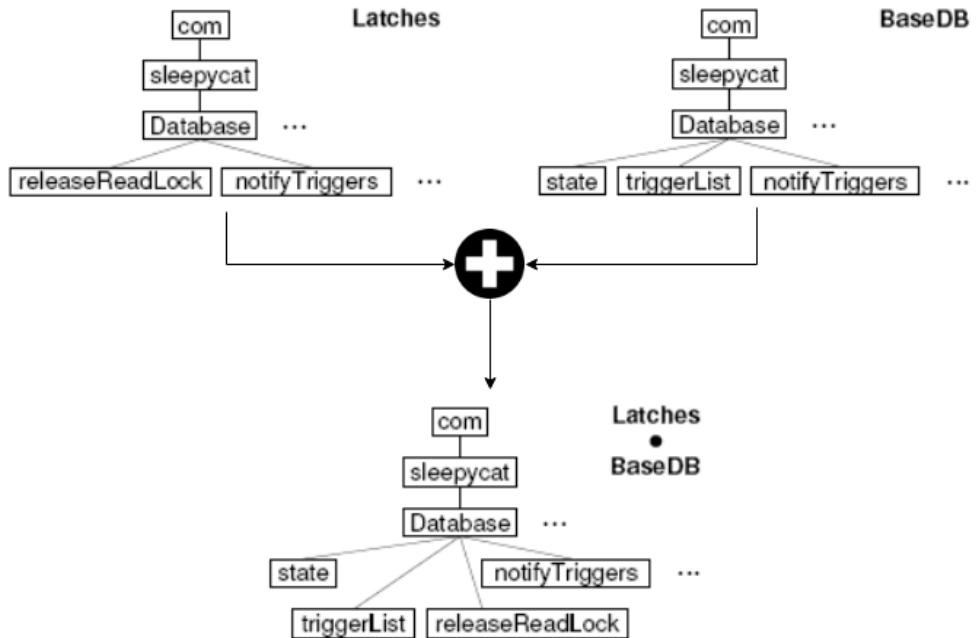


Figura 2.5: Ejemplo de composición de árboles de características con FeatureHOUSE [4]

**software (LPS) y de herramientas que apliquesen el desarrollo dirigido por modelos.**

Para el primer grupo, se encontraron las siguientes alternativas:

- **FeatureHOUSE** [4]: herramienta para la composición de artefactos software que es independiente del lenguaje, por lo que permite componer artefactos escritos en diferentes lenguajes de programación. En FeatureHOUSE los artefactos software se representan mediante árboles de estructura de características. La generación de los productos software se realiza mediante la combinación de los árboles de características de aquellas funcionalidades seleccionadas para el producto. En la figura 2.5 puede verse un ejemplo de composición para un producto a partir de la combinación de árboles de características.
- **AHEAD Tool Suite (ATS)** [5]: colección de herramientas basadas en Java que ofrecen soporte para la programación orientada a características (*Feature Oriented Programming, FOP*). Estas herramientas se basan en el concepto de AHEAD (*Algebraic Hierarchical Equations for Application Design*), un modelo arquitectónico que permite representar un número arbitrario de artefactos software como conjuntos anidados de ecuaciones. El elemento principal de AHEAD es el compositor, que recibe una ecuación como entrada y la expande recursivamente hasta crear un directorio de características compuestas. En la figura 2.6 puede verse un ejemplo de composición con AHEAD. Además, los archivos de código compuestos por herramientas AHEAD están escritos en un superconjunto de

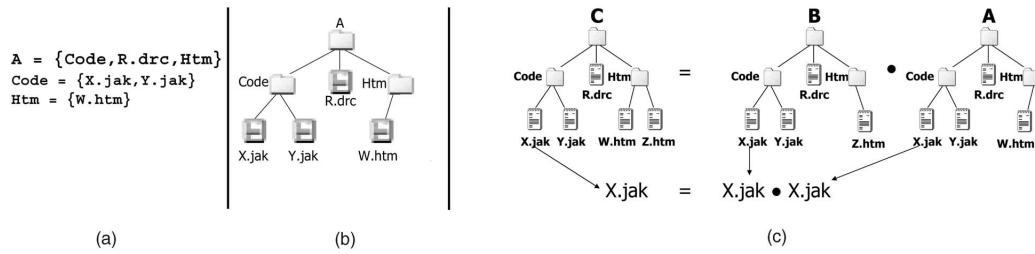


Figura 2.6: Ejemplo de composición con AHEAD [5]

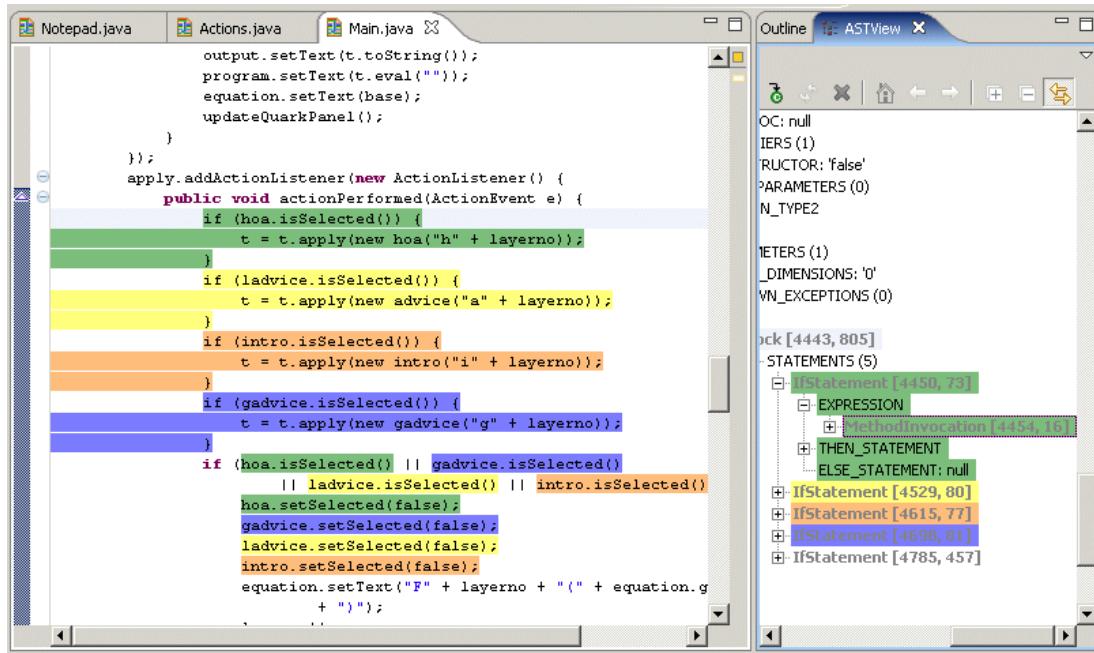


Figura 2.7: Ejemplo de código anotado con CIDE [6]

Java llamado Jak (Jakarta), esto es, Java extendido con lenguajes integrados de dominio específico, máquinas de estado y metaprogramación; lo que permite que las herramientas AHEAD puedan soportar muchos dialectos de Java.

- **CIDE (Colored Integrated Development Environment)** [6]: herramienta para el desarrollo de líneas de producto software. Sigue el paradigma de separación de intereses, es decir, para la generación de productos no se extrae físicamente el código de las características, sino que se anotan fragmentos de código dentro del código original y se emplean herramientas de soporte para las vistas y la navegación. Para la anotación se utilizan colores de fondo, de modo que los fragmentos de código que pertenecen a un componente se muestran con un color determinado. En la figura 2.7 puede verse un ejemplo de código anotado.

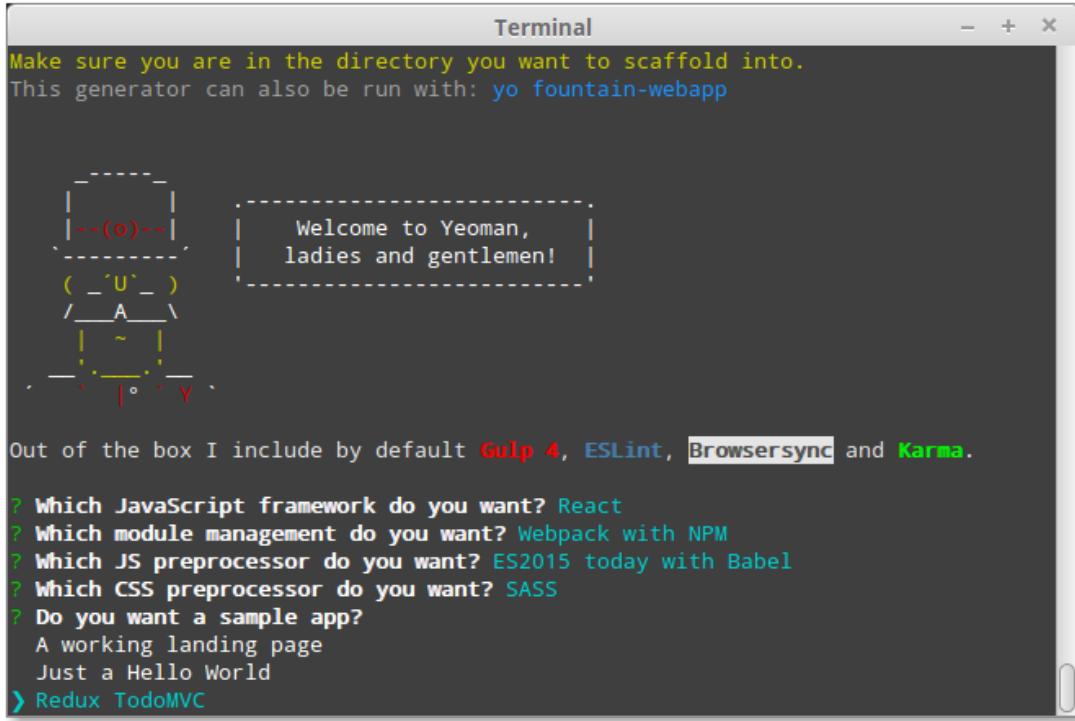


Figura 2.8: Interfaz de comandos de Yeoman [7]

- **Yeoman** [7]: sistema genérico de *scaffolding* que permite la creación de aplicaciones en cualquier lenguaje de programación. Yeoman permite implementar generadores de arquitecturas de proyectos de cualquier tipo, así como utilizar alguno de sus múltiples generadores disponibles para crear directamente el esqueleto de una aplicación. Dentro del entorno de Yeoman se encuentra *yo*, la herramienta que permite crear los proyectos utilizando plantillas de *scaffolding*, denominadas generadores. *yo* cuenta con una interfaz de línea de comandos para Node.js, como la mostrada en la figura 2.8, mediante la que se realiza la configuración y generación de los nuevos proyectos.

En cuanto al segundo grupo de herramientas, las alternativas encontradas fueron las siguientes:

- **Rational Software Architect Designer** [9]: conjunto de herramientas de diseño y desarrollo que permiten crear arquitecturas y diseños de software. Este software de IBM está basado e integrado en Eclipse y proporciona soporte para varios lenguajes de modelado como son: BPMN (Business Process Model and Notation), UML y extensiones UML específicas del dominio, como SoaML y UPIA. En la figura 2.9 puede verse un ejemplo de modelado con esta herramienta.

Una vez definidos los modelos, este software permite transformarlos a elementos de

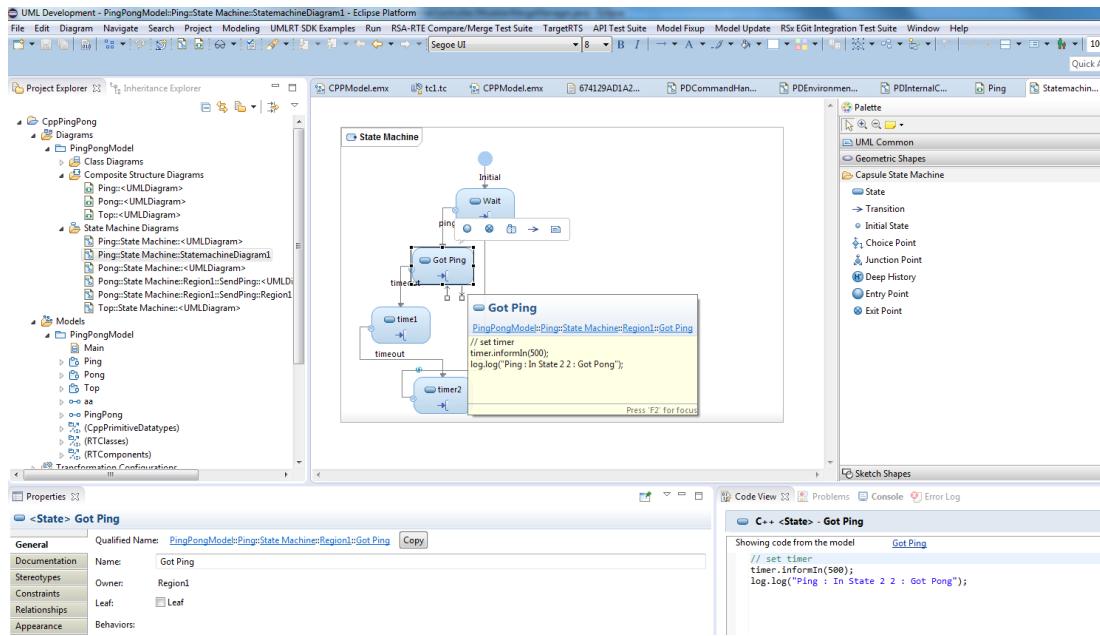


Figura 2.9: Ejemplo de modelado con Rational Software Architect en Eclipse

implementación, en concreto, se pueden transformar en código fuente Java o C++, en artefactos de tiempo de ejecución y en archivos de configuración; utilizando para ello las transformaciones que proporciona el software o las transformaciones personalizadas del usuario. En la figura 2.10 puede verse un ejemplo de generación de código a partir de modelos.

- **Acceleo [10]:** herramienta de generación de código para Eclipse. Esta herramienta es de código abierto, está basada en plantillas e implementa el estándar Model to Text Language (MTL). Este software permite generar automáticamente cualquier tipo de código fuente a partir de un modelo de datos definido en algún formato basado en EFM, como puede ser: UML, SysML, modelos específicos de dominio, etc. Para la definición de los modelos EFM la herramienta cuenta con un lenguaje propio, Acceleo Query Language (AQL). Además, su editor proporciona multitud de funcionalidades para facilitar el desarrollo (plantillas de código personalizables, enlaces de navegación a la declaración de elementos del modelo, refactorización, depurador, etc.). En la figura 2.11 se muestra un ejemplo de definición de un modelo con esta herramienta.
- **AndroMDA [11]:** framework de código abierto basado en el paradigma MDA. Esta herramienta permite generar componentes en cualquier lenguaje de programación (Java, .Net, HTML, etc.) a partir de un conjunto de modelos (generalmente definidos en UML y almacenados en formato XMI) y de un conjunto de *plugins* (bibliotecas de traducción)

## CAPÍTULO 2. FUNDAMENTOS TECNOLÓGICOS

---

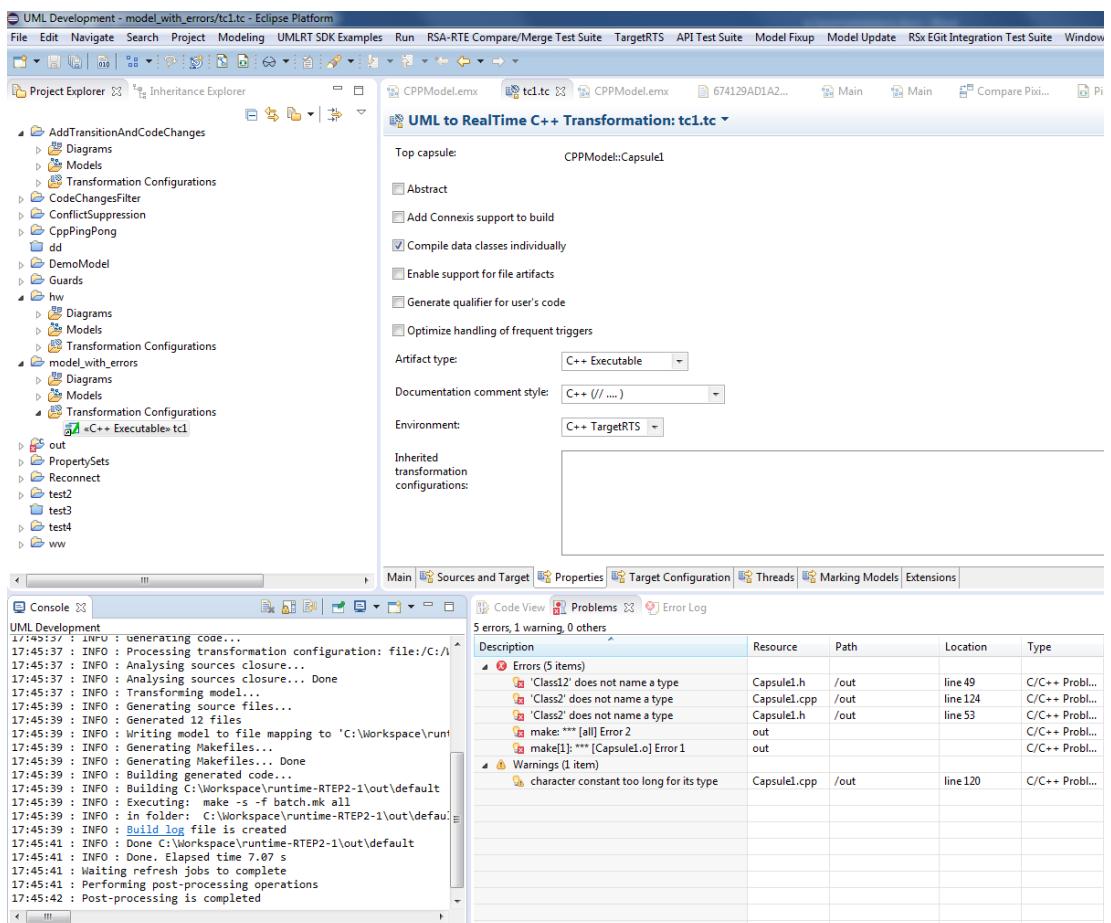


Figura 2.10: Ejemplo de generación de código con Rational Software Architect en Eclipse

```

[comment @main /]
[file (c.fullFilePath(), false, 'UTF-8')]
package [packageName()];

import java.util.List;

public class [javaName() {
    [for (att : Property | ownedAttribute) ]
        private [javaType()/] [javaName()];
    public [javaType()/] get[javaName().toU
        return [javaName()];
    }

    public void set[javaName().toUpperCaseFirst
        this.[javaName()] = [javaName()];
    }
}
[/for]
[/file]

```

Figura 2.11: Ejemplo de definición de un modelo con Acceleo

de AndroMDA. El proceso de generación se basa en refinar el modelo definido a uno más específico dependiente de la plataforma y a partir de este definir las plantillas en base a las que se generará el código. Esta transformación se lleva a cabo mediante el uso de los *plugins* de AndroMDA que se encargan de controlar la traducción y son totalmente personalizables, permitiendo así la generación de cualquier tipo de código a partir del modelo. El código generado se integra de forma automática dentro del proceso de construcción (*build*).

Una vez finalizado el estudio de todas las alternativas mencionadas, se ha llegado a varias conclusiones. Con respecto a las aplicaciones para la gestión del trabajo en movilidad, todas ellas cuentan con funcionalidades similares, centradas en la gestión de tareas y en la geolocalización, al igual que las características definidas en la plataforma GEMA. El análisis de estas aplicaciones ha servido como inspiración especialmente a la hora de realizar el diseño de la interfaz de usuario y a la hora de determinar características propias de los entornos móviles, como puede ser la utilización del GPS.

En cuanto a las herramientas de generación de código, las alternativas expuestas presentan algunos inconvenientes, como que son herramientas demasiado complejas, algunas generan código poco legible al aplicar sus mecanismos de composición (como AHEAD o FeatureHOUSE) y además, aunque algunas permiten la composición sobre varios lenguajes de programación (como FeatureHOUSE o CIDE), en la práctica sólo funcionan con determinadas versiones del lenguaje o es necesario implementar extensiones para cada uno [12].

Algo parecido ocurre con las herramientas de desarrollo dirigido por modelos, que también presentan algunas desventajas, como pueden ser que solo soportan la generación de código con un conjunto de lenguajes limitados o que necesitan elementos personalizados como *plugins* que tienen que ser desarrollados para llevar a cabo la generación de los productos.

Para solventar estos problemas, la línea de producto software de GEMA utiliza la herramienta de derivación **spl-js-engine** [13], basada en anotaciones y desarrollada por uno de los directores de este trabajo para su tesis doctoral [14].

## 2.2 Tecnologías utilizadas

En el desarrollo de este proyecto se ha hecho uso de diversas tecnologías, entre las que se pueden destacar:

- **spl-js-engine** [13]: herramienta JavaScript para la derivación de líneas de producto software.
- **Flutter** [15]: kit de desarrollo de software (SDK) de código abierto para la creación de aplicaciones nativas multiplataforma.
- **Dart** [16]: lenguaje de desarrollo de código abierto diseñado para la implementación del cliente de cualquier tipo de aplicaciones.
- **Material Design** [17]: sistema de diseño adaptable formado por un conjunto de guías, componentes y herramientas que permiten desarrollar interfaces de usuario basadas en las mejores prácticas de diseño.
- **PostgreSQL** [18]: sistema de gestión de bases de datos relacionales orientado a objetos.
- **flutter\_map** [19]: biblioteca que implementa una adaptación de la biblioteca de mapas Leaflet para Flutter.
- **user\_location** [20]: complemento para flutter\_map que añade el soporte para manejar y mostrar la ubicación actual del usuario.
- **flutter\_secure\_storage** [21]: biblioteca de Flutter que permite almacenar los datos de la aplicación de forma persistente, proporcionando seguridad mediante encriptación.
- **geojson\_v1** [22]: biblioteca de Dart que permite crear, leer, buscar, actualizar y eliminar datos geoespaciales (GIS), proporcionando un conjunto de tipos de datos específico.



## Capítulo 3

# Metodología y planificación

---

En este capítulo se detalla la metodología de desarrollo aplicada al proyecto, así como los mecanismos de planificación y seguimiento empleados. Además, se especifica la metodología seguida a la hora de llevar a cabo el desarrollo de la línea de producto software.

### 3.1 Metodología de desarrollo

Para la realización de este proyecto se ha optado por emplear una metodología de desarrollo iterativa e incremental, dirigida por las funcionalidades del sistema.

El desarrollo iterativo e incremental se basa en la división del ciclo de vida del software en varias iteraciones en secuencia, cada una de las cuales puede considerarse un pequeño proyecto autónomo compuesto por actividades de análisis, diseño, implementación y pruebas, en el que se desarrollan nuevas funcionalidades del sistema [23].

Este tipo de metodologías de desarrollo ágil se basan en la planificación adaptativa promoviendo la respuesta rápida y flexible al cambio [23], lo que proporciona una serie de ventajas frente a las metodologías de desarrollo clásicas. Las metodologías ágiles facilitan la planificación y estimación del proyecto, ya que esta se realiza para cada iteración en lugar de para el proyecto completo, y favorecen el seguimiento mediante las reuniones realizadas en cada iteración. Además, proporcionan entregas parciales del producto al cliente de forma periódica, haciéndolo partícipe del proceso de desarrollo y facilitando la detección temprana de cambios y errores. Esto, junto a la división del desarrollo en iteraciones, permite proporcionar una solución rápida a los problemas, minimizando los fallos y reduciendo los costes del producto.

La metodología de desarrollo aplicada a este proyecto está inspirada en Scrum, un marco de trabajo para el desarrollo y el mantenimiento de productos complejos, caracterizado por ser ligero y simple de entender. Scrum emplea un enfoque iterativo e incremental para optimizar la predictibilidad y el control del riesgo, y se basa en la teoría de control de procesos empírica o empirismo, que asegura que el conocimiento procede de la experiencia y permite tomar

decisiones basándose en lo conocido [24].

A continuación se describe la adaptación de Scrum aplicada al proyecto indicando qué técnicas y prácticas ha sido posible seguir teniendo en cuenta las características concretas del mismo. Para ello, se describirán los principales elementos de Scrum y su adaptación a este proyecto.

### 3.1.1 Equipo Scrum

Los equipos Scrum son autoorganizados y multifuncionales, y están compuestos por [24]:

- **Propietario del producto (*product owner*):** es la persona responsable de maximizar el valor del producto y el trabajo del equipo de desarrollo. Además, es el encargado de gestionar la pila del producto, definiendo y ordenando sus elementos.
- **Equipo de desarrollo (*development team*):** está compuesto por los profesionales encargados de crear un incremento de producto “terminado” al final de cada *sprint*.
- **Scrum Master:** es el responsable en asegurar que se entienda y se adopte Scrum.

En este proyecto el rol de propietario del producto ha sido desempeñado por los directores y la autora, ya que la definición y refinamiento de la pila del producto se ha realizado de forma conjunta entre todos los miembros del equipo, al igual que el rol de Scrum Master. El equipo de desarrollo ha estado constituido por un único miembro, la autora del proyecto.

Esta adaptación de la metodología implica varios cambios con respecto a la teoría de Scrum ya que el rol de propietario del producto ha sido desempeñado por más de una persona, al igual que el rol de Scrum Master, y el equipo de desarrollo ha contado con un único miembro en lugar de los siete que son habituales.

### 3.1.2 Eventos Scrum

En Scrum existen diferentes eventos predefinidos con el fin de crear regularidad y minimizar la necesidad de reuniones no definidas. Entre ellos se pueden destacar [24]:

- **Sprint:** periodo de tiempo durante el cual se crea un incremento de producto “terminado” y que constituye una iteración del ciclo de vida.
- **Scrum diario (*daily Scrum*):** reunión que se realiza diariamente en la que el equipo de desarrollo inspecciona el trabajo realizado desde el último Scrum diario y planifica el trabajo a realizar en las siguientes 24 horas.
- **Planificación del sprint (*sprint planning*):** reunión en la que se planifica el trabajo a realizar durante el *sprint*.

- **Revisión del sprint (*sprint review*)**: reunión que se lleva a cabo al finalizar un *sprint* con el fin de inspeccionar el incremento desarrollado y adaptar la pila del producto si fuese necesario.
- **Retrospectiva del sprint (*sprint retrospective*)**: reunión en la que se inspecciona el transcurso del *sprint* con el fin de identificar posibles mejoras sobre la forma en la que el equipo Scrum desempeña su trabajo.

En este proyecto, el ciclo de desarrollo se ha dividido en varios *sprints*, en cada uno de los cuales se llevaba a cabo el desarrollo de un conjunto de funcionalidades. Al finalizar cada *sprint* e inmediatamente antes de comenzar el siguiente se realizaba una reunión de revisión y planificación, equivalente a los eventos de planificación del *sprint* y revisión del *sprint* de Scrum. En estas reuniones, se efectuaba una revisión del trabajo realizado en el *sprint* terminado, tratando los errores y definiendo los futuros cambios a realizar. Asimismo, se llevaba a cabo el refinamiento de la pila del producto y se determinaba qué elementos de esta se desarrollarían en el *sprint* siguiente.

Puesto que el equipo de desarrollo ha estado formado por una única persona, no ha sido posible efectuar reuniones de Scrum diario, y tampoco se han realizado reuniones de retrospectiva del *sprint*.

### 3.1.3 Artefactos Scrum

Los artefactos de Scrum son elementos que representan el trabajo o el valor en diversas formas con el fin de proporcionar transparencia y asegurar que todos tengan el mismo entendimiento de la información clave. Los artefactos Scrum son [24]:

- **Pila del producto (*product backlog*)**: lista ordenada de todo lo que podría ser necesario en el producto. Es la única fuente de requisitos, normalmente expresados en forma de historias de usuario, y se va refinando y completando a medida que avanza el desarrollo del proyecto.
- **Pila del sprint (*sprint backlog*)**: conjunto de los elementos de la pila del producto seleccionados para llevarse a cabo en un *sprint*.
- **Incremento (increment)**: conjunto de los elementos de la pila del producto completados durante un *sprint* y de los incrementos anteriores. Al final del *sprint*, el incremento debe cumplir la definición de “terminado” y ser utilizable.

A lo largo del desarrollo de este proyecto, se han aplicado todos los artefactos definidos por Scrum. La pila del producto fue elaborada inicialmente en la fase de análisis preliminar y se fue refinando progresivamente en las reuniones de planificación y revisión realizadas.

En estas reuniones también se definió la pila del *sprint* para cada iteración que comenzaba y se realizó la revisión del incremento resultante del *sprint* finalizado. Para ello, fue necesario establecer una definición de “terminado” que asegurase el entendimiento entre los miembros del equipo a la hora de determinar que una historia de usuario había sido finalizada.

### 3.1.4 Herramientas de soporte a la metodología

En esta sección se mencionan las herramientas empleadas en el desarrollo del proyecto que han facilitado la aplicación de la metodología seleccionada.

#### Gitlab

A la hora de poner en práctica la adaptación de la metodología Scrum, la principal herramienta empleada ha sido una instancia de Gitlab desplegada y mantenida por el grupo de investigación al que pertenecen los directores de este trabajo, el Laboratorio de Bases de Datos. Gitlab es un repositorio de código gestionado con Git [25] que, además, cuenta con multitud de funcionalidades que facilitan la aplicación de metodologías ágiles como: un sistema de *issue tracking*, gestión de tareas e hitos, etiquetado de *issues* y tableros para su organización, gestión de documentación mediante wikis, mecanismos para mantener la trazabilidad, mecanismos para facilitar la colaboración, etc. [26].

Para el control de este trabajo se creó un nuevo proyecto en Gitlab y se asignó a un grupo.

Durante el desarrollo se creó un hito por cada iteración marcando la fecha de inicio y la fecha de fin del *sprint* correspondiente, y se representaron las historias de usuario definidas en la pila de producto en forma de *issues* dentro del proyecto, con un título y una descripción. Cada una de ellas se correspondía tanto con historias de usuario completas, como con subtareas contenidas dentro de una historia de usuario, y también se emplearon para representar las correcciones de los *bugs* detectados y las mejoras realizadas. Siguiendo la planificación, las *issues* se fueron asignando a los hitos correspondientes.

Para la gestión de las *issues* se empleó principalmente la vista de tablero, que facilitó su organización y planificación. Dicho tablero estaba compuesto de cuatro columnas que permitían diferenciar entre las tareas abiertas, las planificadas para su realización, las que se estaban desarrollando y aquellas que ya habían sido terminadas, como se puede ver en la figura 3.1. Asimismo, se emplearon etiquetas para clasificar las tareas y sus estados, y para diferenciar entre las correcciones de errores, las mejoras y las nuevas funcionalidades.

A la hora de mantener el registro del tiempo empleado en el desarrollo de cada tarea, se utilizó la funcionalidad de *time tracking* [27]. Este mecanismo permite imputar el tiempo dedicado a cada *issue* mediante el uso del comando */spend <time>*, que suma la cantidad indicada al total de tiempo asociado a la incidencia. Gracias a este sistema es posible conocer el tiempo que se ha dedicado a cada tarea, así como mantener un registro del tiempo empleado en el

## CAPÍTULO 3. METODOLOGÍA Y PLANIFICACIÓN

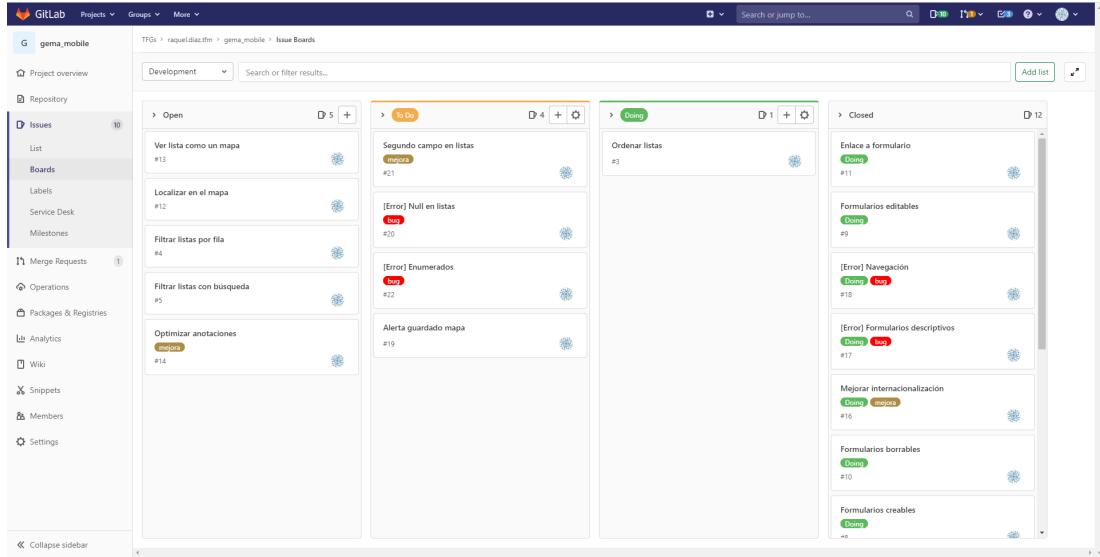


Figura 3.1: Tablero de Gitlab

desarrollo de cada *sprint*, ya que los hitos almacenan el total de los tiempos imputados en sus *issues* asociadas.

Esta información, así como todos los cambios realizados sobre las incidencias se registran tal y como puede verse en la figura 3.2.

En el momento de llevar a cabo la implementación de una *issue*, el procedimiento a seguir consistía en crear una *merge request* que creaba una nueva rama a partir de la rama *master*, en la cual se subían los cambios realizados para la implementación y prueba de la funcionalidad en cuestión. Una vez terminada la tarea, se añadían los cambios a la rama *master* mediante la ejecución de un *merge*. Este flujo de trabajo permite diferenciar qué *commits* pertenecen al desarrollo de cada funcionalidad, favoreciendo la localización de cambios y errores.

### Otras herramientas

Además de Gitlab, en la realización de las tareas del proyecto se emplearon las siguientes herramientas:

- **Android Studio [28]**: entorno de desarrollo integrado (IDE) empleado para el desarrollo de la aplicación nativa con Flutter.
- **Visual Studio Code [29]**: entorno de desarrollo integrado (IDE) empleado para la generación de productos y para el trabajo con el servidor.
- **Dbeaver [30]**: herramienta de administración de bases de datos empleada en el proyecto para facilitar el acceso a la información de la base de datos.

	Raquel Díaz Otero @raquel.diaz changed milestone to <a href="#">%Sprint 5</a> 3 months ago
	Raquel Díaz Otero @raquel.diaz added <a href="#">To Do</a> label 2 months ago
	Raquel Díaz Otero @raquel.diaz added <a href="#">Doing</a> label and removed <a href="#">To Do</a> label 2 months ago
	Raquel Díaz Otero @raquel.diaz created merge request <a href="#">!19 (merged)</a> to address this issue 2 months ago
	Raquel Díaz Otero @raquel.diaz mentioned in merge request <a href="#">!19 (merged)</a> 2 months ago
	Raquel Díaz Otero @raquel.diaz added 5h of time spent at 2021-04-10 1 month ago
	Raquel Díaz Otero @raquel.diaz added 1h 30m of time spent at 2021-04-10 1 month ago
	Raquel Díaz Otero @raquel.diaz closed via merge request <a href="#">!19 (merged)</a> 1 month ago
	Raquel Díaz Otero @raquel.diaz mentioned in commit <a href="#">6e65df1d</a> 1 month ago

Figura 3.2: Registro de información en una *issue*

- **Balsamiq Mockups** [31]: herramienta para elaborar maquetas de interfaces web y de aplicaciones móviles, empleada en el proyecto en la elaboración de los prototipos de la interfaz de usuario de la aplicación.
- **MagicDraw** [32] y **Draw.io** [33]: herramientas para la elaboración de diagramas.
- **Microsoft Excel** [34]: herramienta de análisis de datos, empleada para la elaboración de los diagramas de Gantt.

## 3.2 Metodología de desarrollo de la LPS

Las líneas de producto software se definen como un conjunto de sistemas software (productos), que comparten un conjunto común de características (*features*), las cuales satisfacen las necesidades específicas de un dominio o segmento particular de mercado, y que se desarrollan a partir de un sistema común de activos base (componentes reutilizables) de una manera preestablecida [35].

A partir de este concepto de generación de código, el Laboratorio de Bases de Datos de la UDC, al que pertenecen los dos directores de este trabajo, ha desarrollado como parte del proyecto GEMA [8] una herramienta para la generación de aplicaciones de gestión de trabajadores en movilidad utilizando líneas de producto software. Para ello, se han extraído componentes y funcionalidades de tres prototipos de aplicaciones web desarrolladas para los socios empresariales del proyecto, se han generalizado, se han modificado para adaptarlos a una

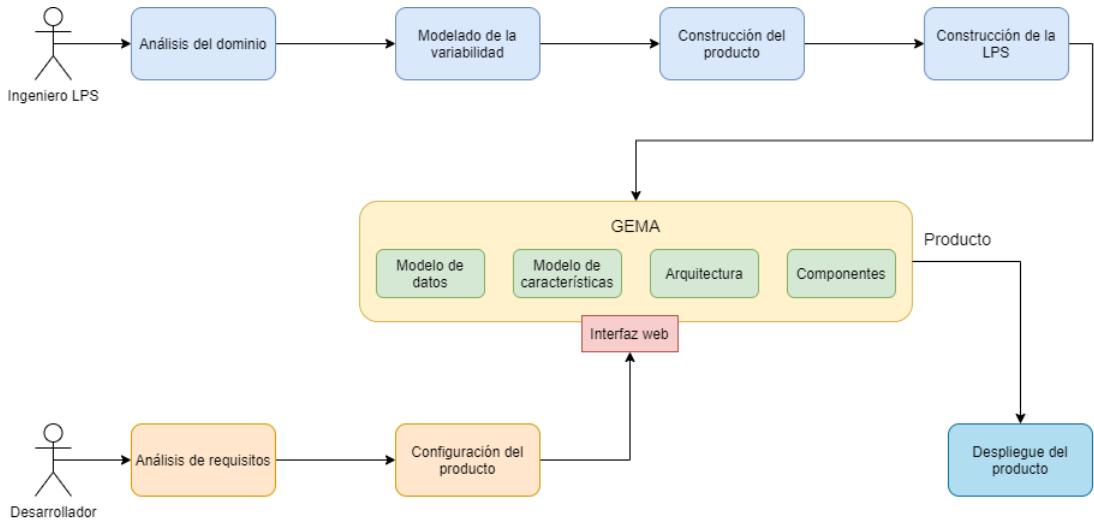


Figura 3.3: Metodología seguida para el desarrollo de la LPS

arquitectura común, y se ha desarrollado una plataforma que permite a un analista diseñar el modelo de datos y seleccionar las funcionalidades del producto que desea para obtener el código fuente de la aplicación web que se corresponde con su especificación.

Dado que el resultado actual de la plataforma GEMA es una aplicación web centrada en la gestión del trabajo por los empleados de administración de la empresa, y no incluye funcionalidad para los trabajadores en movilidad, el objetivo de este trabajo es extender la plataforma existente para añadir una aplicación nativa multiplataforma destinada al uso por parte de los empleados desplazados. Para ello, se ha seguido la metodología de desarrollo reflejada en la figura 3.3.

Como puede verse, a la hora de afrontar el desarrollo de la LPS, el primer paso es realizar un análisis del dominio para definir las características que formarán parte de los productos generados y, en base a ellas, definir los requisitos necesarios para la construcción del producto. En este caso, ya se partía del dominio definido para la plataforma GEMA, por lo que en esta fase del proceso se llevó a cabo un análisis de las características soportadas para seleccionar aquellas que serían implementadas en la aplicación nativa y adaptarlas al contexto de las aplicaciones móviles, así como para definir nuevas características o ampliaciones propias de este tipo de entornos.

En esta fase también es necesario definir el modelo de datos. Con respecto a esto, una de las principales características de esta plataforma para la generación de código es que permite definir el modelo de datos de cada producto generado, aunque al estar enfocada al ámbito de empresas de trabajo en movilidad, también cuenta con un modelo de datos definido para la gestión de algunos aspectos necesarios, como la gestión de las visitas de los trabajadores. En este punto, se estableció un modelo de datos para realizar las pruebas durante el desarrollo de

la aplicación.

Una vez seleccionadas y definidas las características que formarían parte de las aplicaciones nativas, se realizó el modelado de la variabilidad partiendo del ya existente para la plataforma implementada. En este proceso, detallado en la sección 4.3, se determinaron las características comunes a todos los productos de la LPS, así como aquellas que podían variar de unos productos a otros. El resultado de este análisis se representó mediante un modelo de características.

Tras finalizar el análisis se llevó a cabo el desarrollo del producto, esto es, el análisis, diseño, implementación y prueba de la aplicación nativa, así como su integración en la línea de producto software. Este proceso (detallado en los capítulos 5, 6, 7 y 8) estuvo guiado por el análisis previo de forma que la organización e implementación de los componentes buscó facilitar la labor de inclusión y exclusión de estos en los productos generados.

En este punto del proceso, ya se ha completado el desarrollo de la aplicación y su integración en la línea de producto software de GEMA. A partir de aquí, la plataforma podrá generar aplicaciones nativas partiendo de una configuración del producto, es decir, de un conjunto de características seleccionadas y de la definición de un modelo de datos.

### 3.3 Planificación y seguimiento

En esta sección se detallan los recursos necesarios para llevar a cabo el proyecto, así como la planificación y estimación realizadas inicialmente y el seguimiento real de la planificación, junto con las desviaciones en tiempo y coste.

#### 3.3.1 Recursos

A continuación se mencionan los recursos, tanto humanos como técnicos, necesarios para llevar a cabo este proyecto.

##### Recursos humanos

En la realización de este proyecto se ha contado con un equipo de tres personas formado por los directores y la autora del trabajo, que han asumido los roles de *product owner*, llevando a cabo la gestión de la pila del producto; de analista, realizando actividades de planificación y diseño; y de desarrollador, efectuando la implementación y prueba de las distintas funcionalidades.

### Recursos técnicos

Dentro de los recursos técnicos empleados en el desarrollo de este proyecto se pueden diferenciar dos grupos: recursos hardware y recursos software.

En cuanto a los **recursos hardware**, se ha empleado un ordenador portátil en el cual se ha llevado a cabo la implementación de todas las funcionalidades definidas para el sistema, así como la elaboración de los diagramas y la documentación del proyecto. Además, se ha hecho uso de un dispositivo móvil con sistema operativo Android para ejecutar la aplicación desarrollada y realizar las pruebas sobre esta.

Respecto a los **recursos software**, se incluyen los mencionados en la sección [3.1.4](#).

#### 3.3.2 Planificación

Al comienzo del proyecto se llevó a cabo un **análisis preliminar** con el fin de realizar un estudio de las tecnologías con las que se llevaría a cabo el desarrollo de la aplicación y de analizar las funcionalidades de la plataforma existente para comprender su funcionamiento. La fase de análisis preliminar tuvo una duración de cuatro semanas y en ella se llevaron a cabo las siguientes tareas:

- **Estudio de las tecnologías:** debido a la necesidad de que la aplicación desarrollada fuese nativa y multiplataforma, se decidió emplear la tecnología Flutter. Durante esta fase se llevó a cabo un proceso de formación principalmente orientado a la lectura de la documentación y a la realización de pruebas que permitiesen adquirir los conocimientos necesarios para llevar a cabo el desarrollo del producto.
- **Estudio de la plataforma actual:** se realizó un análisis del funcionamiento de la plataforma GEMA. Durante esta fase se analizaron las funcionalidades implementadas en la plataforma con el fin de comprender su funcionamiento y definir las adaptaciones necesarias para incluirlas en el producto desarrollado. Asimismo, se analizó el proceso de generación de productos y la definición de los modelos de datos, además del código del cliente y el servidor ya implementados.
- **Definición de los actores del sistema:** se realizó un análisis de los posibles roles y actores que podrían interactuar con el sistema, definiendo las funcionalidades a las que tendría acceso cada uno de ellos.
- **Elaboración de la pila del producto inicial:** se elaboró la versión inicial de la pila del producto, describiendo las funcionalidades del sistema en forma de historias de usuario.
- **Configuración del modelo de datos:** se definió un modelo de datos de prueba para llevar a cabo la implementación del producto en base a él.



Figura 3.4: Diagrama de Gantt de planificación

Rol	Salario/hora
Jefe de proyecto	28 €
Analista	24 €
Desarrollador	16 €

Cuadro 3.1: Salarios por hora de cada rol

- **Elaboración de los prototipos de pantallas:** se realizó el diseño de las pantallas de la aplicación móvil mediante la utilización de *mockups*.

Una vez finalizadas estas tareas se llevó a cabo la planificación del proyecto, cuyo resultado puede verse en el diagrama de Gantt de la figura 3.4. El proceso de desarrollo del software se dividió en *sprints* de cuatro semanas, dando lugar a seis iteraciones tras las cuales se planificó un último *sprint* para la realización de esta memoria. Se planificaron dos pausas en el desarrollo, correspondientes a los períodos de navidad y semana santa, y se tuvieron en cuenta los días festivos y puentes a la hora de establecer las fechas de inicio y fin de cada iteración, de forma que todos los *sprints* tuvieran una duración de 21 días laborables.

El trabajo a realizar en cada *sprint* se planificó de forma que la autora trabajaría 3 horas cada día, 5 días a la semana, en la realización de tareas de análisis, diseño, implementación y pruebas. Teniendo en cuenta la planificación realizada se estimó un total de 148 días laborables, dando lugar a **444 horas** de trabajo. A esta cantidad se le deben sumar las 36 horas invertidas en la fase de análisis preliminar, resultando en un total de **480 horas** de trabajo estimadas para el desarrollo del proyecto. Además, se estimó que la reunión de seguimiento y planificación correspondiente a cada *sprint* tendría una duración aproximada de una hora.

Una vez estimado el tiempo, se realizó la estimación del coste que supondría llevar a cabo el proyecto teniendo en cuenta los salarios por hora reflejados en el cuadro 3.1. Dicha estimación puede verse en el cuadro 3.2.

Miembro	Horas jefe de proyecto	Horas analista	Horas desarrollador	Coste
Miguel A. Rodríguez	3	5,5	-	216 €
Alejandro Cortiñas	3	5,5	-	216 €
Raquel Díaz	5	65	414	8 324 €
<b>Totales</b>	<b>11</b>	<b>76</b>	<b>414</b>	<b>8 756 €</b>

Cuadro 3.2: Costes estimados para el proyecto

### 3.3.3 Seguimiento

En esta sección se analiza el seguimiento del proyecto sobre la planificación inicial, detaillando el trabajo realizado en cada *sprint* con respecto a su planificación y mencionando las desviaciones en tiempo y coste sufridas junto a las causas de las mismas. Cabe destacar que la cantidad de historias asignadas a cada *sprint* varía dependiendo de su complejidad y del tiempo de desarrollo necesario para su implementación y prueba.

Hay que tener en cuenta que, siguiendo las directrices de la metodología empleada, en la reunión de revisión y planificación de cada *sprint* se realizó una revisión de las historias de usuario implementadas en la iteración finalizada, y se planificó la corrección e implementación de los errores y mejoras detectadas para el *sprint* siguiente.

#### Sprint 1

Las tareas planificadas para el primer *sprint* fueron:

- Crear y configurar el proyecto siguiendo las instrucciones definidas en la documentación de Flutter.
- Implementar el menú lateral (historia 1).
- Configurar e implementar los mecanismos necesarios para permitir la internacionalización de la aplicación (historia 2).
- Implementar las entidades de la aplicación.
- Desarrollar la funcionalidad de listado de entidades (historia 3).

Por motivos personales, durante el primer *sprint* se perdió una semana de trabajo, por lo que su duración se alargó una semana, terminando el día 13 de noviembre en lugar del día 6 de noviembre como estaba planificado. Sin embargo, gracias a la prolongación en la duración del *sprint* y el consecuente mantenimiento de los días de trabajo empleados, fue posible llevar a cabo el desarrollo de todas las funcionalidades planificadas. Por otra parte, en la reunión de revisión del *sprint* se detectó un fallo en el diseño de la arquitectura de la aplicación, que se planificó para solventar en la siguiente iteración.

## Sprint 2

Las tareas planificadas para el segundo *sprint* fueron:

- Implementar los formularios descriptivos (historia 4).
- Implementar los formularios creables (historia 5).
- Implementar los formularios editables (historia 6).

Este *sprint* también sufrió una prolongación de una semana en su duración, debido principalmente a la resolución del problema de arquitectura detectado en la iteración anterior, cuya solución consumió más tiempo del esperado, y a la preparación del anteproyecto, cuyo plazo de presentación coincidió temporalmente con esta iteración. De esta forma, el *sprint* terminó el día 18 de diciembre en lugar del día 9 de diciembre como estaba planificado. En este caso, no fue posible terminar la implementación de los formularios creables, por lo que la finalización de esta historia de usuario se planificó para el *sprint* siguiente.

## Sprint 3

Las tareas planificadas para el tercer *sprint* fueron:

- Implementar los formularios borrables (historia 7).
- Desarrollar la funcionalidad de enlace a formulario (historia 8).
- Implementar la búsqueda en las listas (historia 9).

En este *sprint* sí que fue posible llevar a cabo el desarrollo de todas las historias planificadas en el tiempo establecido además de finalizar la implementación de la funcionalidad que había quedado pendiente de la iteración anterior. Por lo tanto, ninguna tarea tuvo que ser aplazada.

## Sprint 4

Las tareas planificadas para el cuarto *sprint* fueron:

- Implementar el filtrado de listas por fila (historia 10).
- Implementar la ordenación en las listas (historia 11).
- Desarrollar la funcionalidad de localizar un elemento en el mapa (historia 12).
- Implementar la funcionalidad de ver la lista como un mapa (historia 13).

En este *sprint* se cumplió la planificación, llevando a cabo el desarrollo de todas las funcionalidades previstas en un periodo de cuatro semanas.

### Sprint 5

Las tareas planificadas para el quinto *sprint* fueron:

- Desarrollar las funcionalidades relacionadas con la gestión de usuarios, esto es:
  - Autenticar un usuario y ver su información asociada (historia 14).
  - Recuperar la contraseña de usuario (historia 15).
- Implementar parte de las funcionalidades relacionadas con la gestión de visitas, en concreto:
  - Visualizar la lista de visitas (historia 16).
  - Visualizar el detalle de una visita (historia 17).
  - Editar una visita (historia 18).

En el quinto *sprint* tampoco se produjeron retrasos en cuanto a la planificación y todas las funcionalidades planificadas se desarrollaron en el plazo previsto.

### Sprint 6

Las tareas planificadas para el sexto *sprint* fueron:

- Implementar el calendario de visitas (historia 19).
- Implementar el mapa de visitas (historia 20).
- Desarrollar la funcionalidad de cómo llegar a una localización (historia 21).
- Desarrollar la funcionalidad de ver el detalle de un cliente (historia 22).
- Desarrollar la funcionalidad de ver el detalle de un empleado (historia 23).

Este *sprint* fue el último destinado a la implementación y prueba de las funcionalidades de la aplicación y en él no se produjeron desviaciones en cuanto a la planificación. En la reunión de seguimiento se revisó la aplicación y se definieron los últimos retoques a realizar en la interfaz, que fueron planificados para la última iteración.

### Sprint 7

Las tareas planificadas para el último *sprint* fueron:

- Redactar la memoria.

En este *sprint* se implementaron los últimos retoques y correcciones de errores en la aplicación, y se redactó la memoria final.



Figura 3.5: Diagrama de Gantt de seguimiento

Miembro	Horas jefe de proyecto	Horas analista	Horas desarrollador	Coste
Miguel A. Rodríguez	3	4,5	-	192 €
Alejandro Cortiñas	3	5,5	-	216 €
Raquel Díaz	7	68	409	8 372 €
<b>Totales</b>	<b>13</b>	<b>78</b>	<b>409</b>	<b>8 780 €</b>

Cuadro 3.3: Costes reales del proyecto

#### Desviaciones en tiempo y coste

En la figura 3.5 se puede ver el diagrama de Gantt que refleja la progresión real del proyecto, con los cambios realizados sobre la planificación inicial y las desviaciones de tiempo sufridas. Asimismo, en el cuadro 3.3 se pueden observar los costes reales del proyecto calculados en base a los salarios del cuadro 3.1 y a las horas finalmente invertidas en la realización de cada actividad.

## Capítulo 4

# Análisis de la línea de producto software

---

En este capítulo se detalla el análisis llevado a cabo sobre la línea de producto software de GEMA para permitir el posterior desarrollo del producto y su integración dentro de esta, centrándose en el funcionamiento y en la estructura de la plataforma existente.

### 4.1 Funcionamiento de la LPS

La ingeniería de línea de productos de software es un paradigma para el desarrollo de aplicaciones software basado en el uso de plataformas y personalización. Este enfoque de desarrollo de aplicaciones utilizando plataformas implica construir piezas reutilizables y emplearlas en la construcción de productos. Además, generar aplicaciones personalizadas implica gestionar el concepto de variabilidad, es decir, los aspectos comunes y las diferencias entre los productos (en términos de requisitos, arquitectura, componentes y artefactos de prueba) de la línea de producto software tienen que modelarse de forma conjunta [36].

Siguiendo este paradigma de líneas de producto software se ha desarrollado el proyecto GEMA, que actualmente permite generar el código fuente de un servidor REST y un cliente web a partir de un conjunto de características seleccionadas y un modelo de datos definido, como veremos a continuación.

A la hora de configurar un producto, la plataforma cuenta con una interfaz web que permite definir todos los aspectos posibles. Por una parte, cuenta con un árbol de características con selectores que permite elegir qué características se incluirán en el producto, respetando las restricciones establecidas para la variabilidad de la LPS. Esta pantalla puede verse en la figura 4.1.

Por otra parte, se proporciona un editor que permite definir de forma gráfica el modelo de datos que empleará el producto generado. Este editor, que se puede ver en la figura 4.3,

#### 4.1. Funcionamiento de la LPS

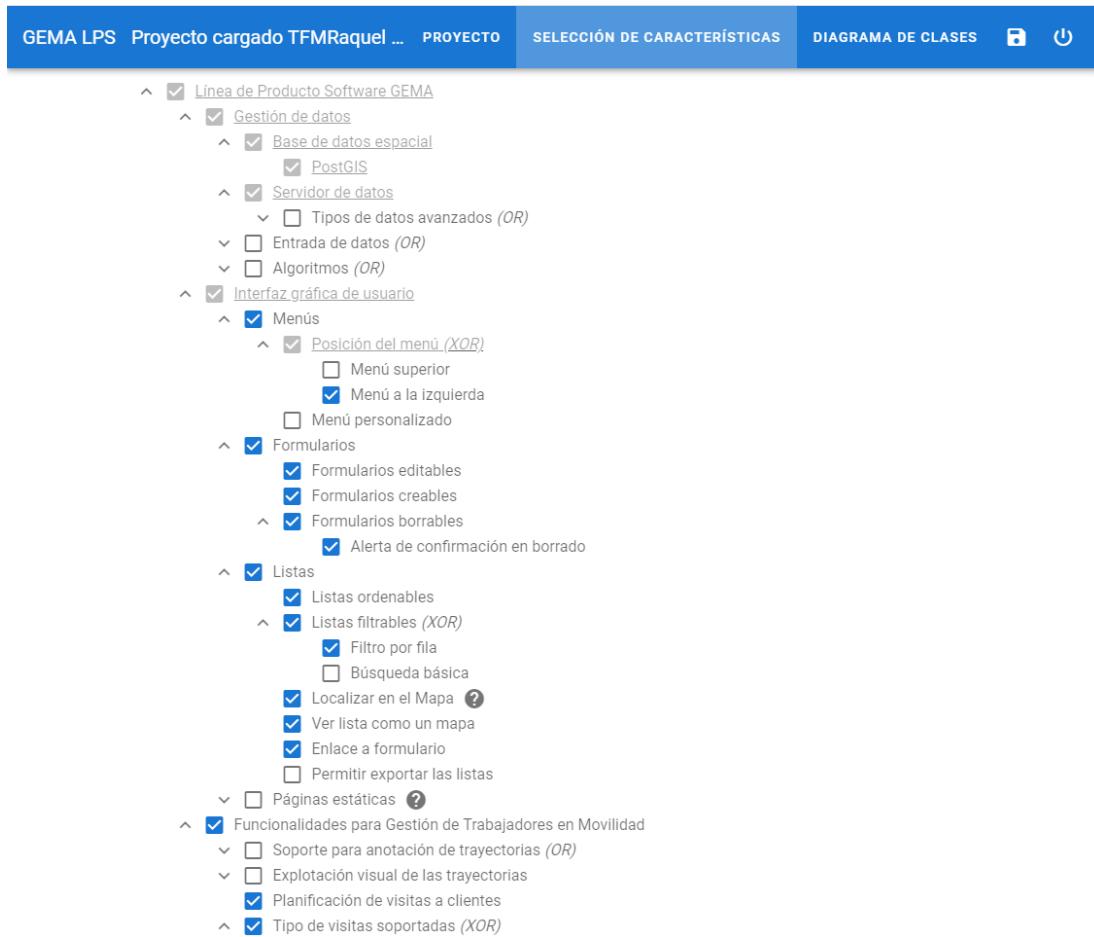


Figura 4.1: Árbol de características de GEMA



Figura 4.2: Operaciones de GEMA

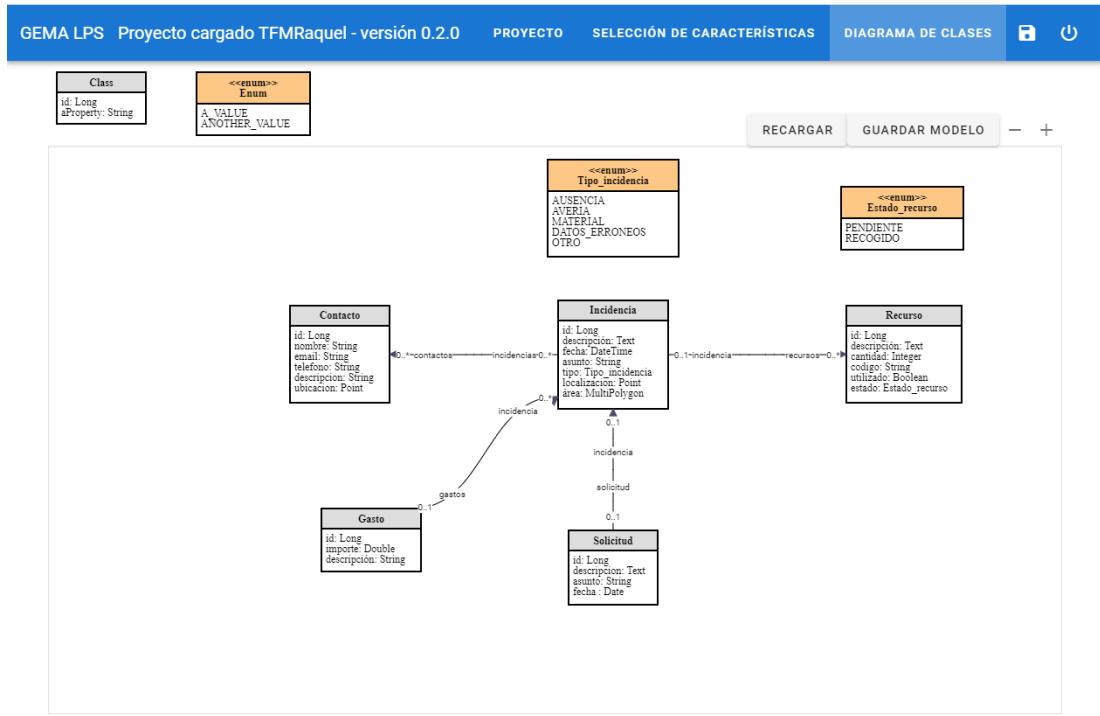


Figura 4.3: Editor del modelo de datos de GEMA

permite establecer las clases que formarán parte del sistema y las relaciones entre ellas, así como definir tipos de datos enumerados.

Una vez configurado el producto es posible generar el código fuente o exportar su especificación. Además, a través de la interfaz se ofrecen otra serie de operaciones como importar una especificación o configurar aspectos del producto como la conexión a la base de datos, pero no son relevantes para el ámbito de este trabajo. Las funcionalidades ofrecidas por la herramienta de generación se muestran en la figura 4.2.

El funcionamiento interno de la línea de producto software se explica en el capítulo 8.

## 4.2 Modelos de datos genérico

Como hemos visto, una de las características principales de la plataforma existente es que permite generar aplicaciones a partir de un modelo de datos específico para cada producto. Estos modelos de datos están sujetos a un conjunto de restricciones que establece la plataforma, que soporta un conjunto finito de tipos de datos. En la figura 4.4 se ha hecho una representación de este modelo de datos genérico abarcando todos los componentes que forman parte de las posibles configuraciones soportadas.

Como puede verse, el diagrama de clases genérico está compuesto por cuatro elementos

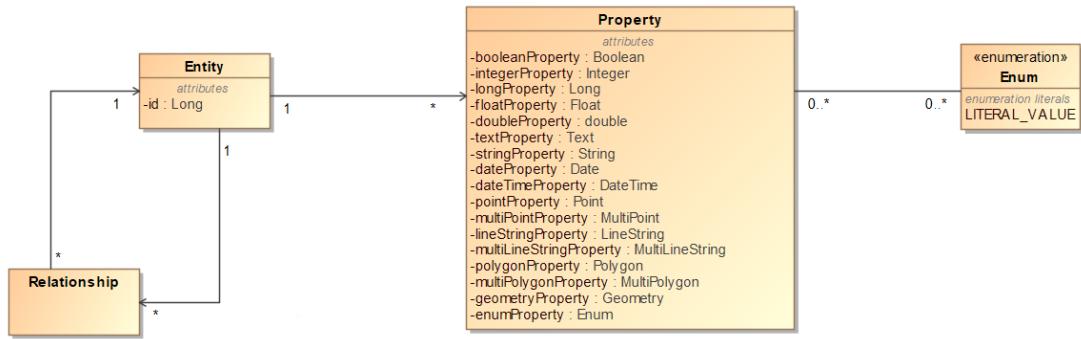


Figura 4.4: Modelo genérico de datos

principales, que son los siguientes:

- **Entity**: representa a una entidad del modelo de datos y tiene una propiedad obligatoria, que es su identificador. Puede tener múltiples propiedades de distintos tipos.
- **Relationship**: representa una relación entre dos entidades del modelo. Como se muestra en las cardinalidades de las relaciones entre *Entity* y *Relationship*, una entidad puede tener múltiples relaciones con otras entidades.
- **Property**: representa una propiedad de una clase. Las propiedades definidas para esta clase en el diagrama hacen referencia a los tipos de datos soportados. Cada propiedad pertenece a una única entidad, pero puede haber múltiples propiedades del mismo tipo asociadas a la misma o a distintas entidades.
- **Enum**: representa un tipo de datos enumerado que está compuesto por un conjunto de valores literales. Puede estar asociado a múltiples propiedades.

### 4.3 Modelado de la variabilidad de la LPS

El desarrollo de una línea de producto software implica gestionar la variabilidad de la misma. Para ello, es necesario definir los puntos de variación entre sus productos, determinando los elementos comunes (*commonalities*) y los elementos variables (*variabilities*). Las características comunes son aquellas que están presentes en todos los productos de la línea, mientras que las características variables pueden ser comunes a varios productos, pero no a todos [37].

A la hora de desarrollar este proyecto, puesto que se partía de una línea de producto software ya existente, no fue necesario analizar la variabilidad de esta desde cero. Una vez analizadas las funcionalidades de la plataforma, se seleccionaron aquellas que serían implementadas en la aplicación nativa y se estableció la variabilidad de la LPS para la aplicación

como un subconjunto de la variabilidad de la línea de producto software de GEMA. Esta variabilidad se ha representado empleando un modelo de características, mostrado en la figura 4.5. Este diagrama se ha diseñado para facilitar la visualización ya que, en la práctica, el modelo de características empleado en el desarrollo ha sido el de GEMA, que se puede ver en la figura A.1 con las ramas que no se han implementado en este trabajo cerradas.

Un modelo de características (*feature model*) es un árbol que estructura jerárquicamente el conjunto de funcionalidades del sistema. Dentro de esta estructura, cada característica se puede descomponer en varias sub-características que pueden ser obligatorias, opcionales o alternativas [36]. Las características obligatorias representan los elementos comunes a todos los productos (*commonalities*), mientras que las características opcionales representan los elementos variables (*variations*).

Como puede verse en la figura 4.5, existen diferentes tipos de relaciones entre las características definidas para el producto en base a sus requisitos. El operador *or* indica que se tiene que seleccionar obligatoriamente alguna de las características que forman parte de la relación, pudiendo seleccionar más de una; mientras que el operador *alternative* indica que se debe seleccionar una única característica de entre las que forman parte de dicha relación.

En cuanto a las funcionalidades seleccionadas para la aplicación, pueden verse tres bloques principales. El primero de ellos (*GraphicalUserInterface*) agrupa las características relacionadas con aspectos de la interfaz gráfica de la aplicación, como son el menú, los formularios y las listas. El segundo grupo (*MWMSupport*) hace referencia a los aspectos de la aplicación relacionados con la gestión de visitas, empleados y clientes. Por último, el tercer grupo (*UserManagement*) agrupa las características relacionadas con la gestión de usuarios, como la autenticación. Dentro de cada bloque se muestran las funcionalidades y grupos disponibles para cada uno, con los conectores correspondientes.

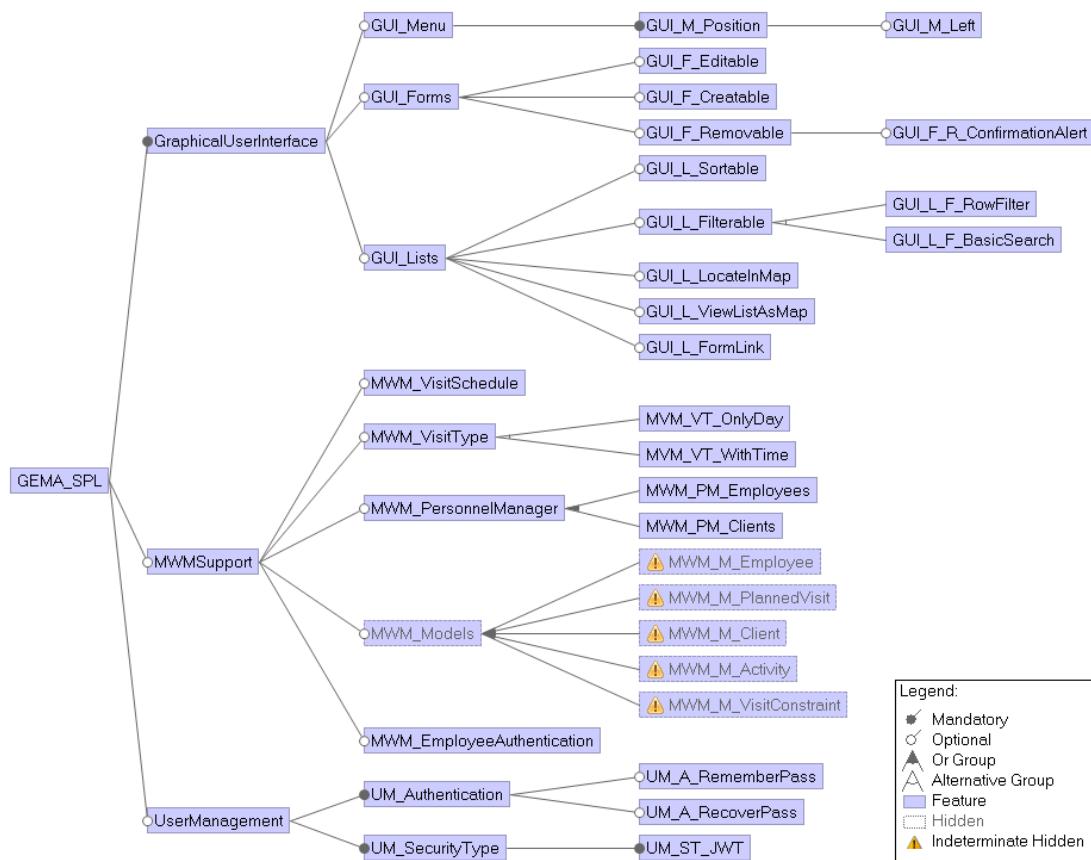


Figura 4.5: Modelo de características de la LPS

## Capítulo 5

# Análisis

---

En este capítulo se detalla el análisis llevado a cabo para el desarrollo de la aplicación. En este proceso se llevó a cabo la definición de los actores y de los requisitos del producto, el diseño de la arquitectura y de la interfaz de usuario, y la definición del modelo de datos.

### 5.1 Actores

En esta sección se detallan los actores definidos para el producto, cuya jerarquía se puede observar en la figura 5.1.

- **Usuario:** se trata de un actor abstracto que representa las funcionalidades comunes a todos los usuarios de la aplicación, prácticamente todas las soportadas.
- **Usuario anónimo:** puede iniciar sesión en la aplicación.
- **Usuario registrado:** puede ver la información asociada a su cuenta y solicitar la recuperación de su contraseña.

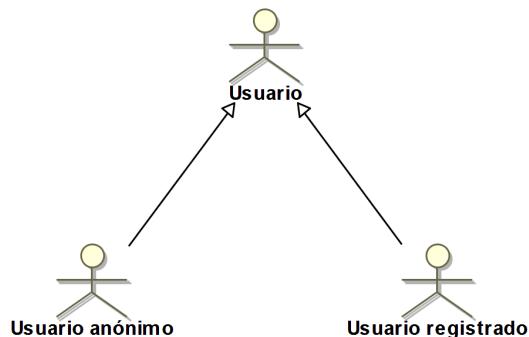


Figura 5.1: Actores

## 5.2 Requisitos

En esta sección se detallan los requisitos funcionales y no funcionales definidos para el producto. Inicialmente se enumeran los requisitos de forma general para posteriormente detallarlos en la pila del producto en forma de historias de usuario.

### 5.2.1 Requisitos funcionales

Los requisitos funcionales establecidos para la aplicación son los siguientes:

- Funcionalidades para la **gestión de usuarios**, permitiendo la autenticación de usuarios y la visualización de los datos asociados a su cuenta, así como la recuperación de la contraseña asociada.
- Funcionalidades para la **gestión de entidades**, permitiendo la visualización, creación, modificación y borrado de estas mediante el uso de formularios.
- Funcionalidades para la **gestión de listas**, proporcionando mecanismos de búsqueda, filtrado y ordenación de las mismas.
- Funcionalidades para la **gestión de visitas**, permitiendo su visualización en distintos formatos, su filtrado y su edición.
- Funcionalidades de **geolocalización y mapas**, proporcionando mecanismos para la gestión de ubicaciones y rutas.
- **Internacionalización** de los mensajes de la aplicación.

### 5.2.2 Requisitos no funcionales

Los requisitos no funcionales definidos para la aplicación son los siguientes:

- **Facilidad de uso**: la aplicación nativa debe ser fácil de usar, para lo que se debe desarrollar una interfaz intuitiva y sencilla que facilite la interacción y el trabajo con ella.
- **Facilidad de instalación**: el sistema debe ser fácil de instalar, por lo que se debe proporcionar documentación con las instrucciones adecuadas.
- **Seguridad**: la aplicación debe asegurar que los datos sean accesibles únicamente por los usuarios autorizados.
- **Multiplataforma**: la aplicación debe ser compatible con múltiples dispositivos.

### 5.2.3 Pila del producto

Siguiendo la metodología seleccionada, se elaboró una pila del producto que constituyó la fuente de requisitos durante el desarrollo de la aplicación. Los requisitos generales mencionados previamente se detallaron en forma de historias de usuario que, con el avance del desarrollo, se fueron refinando hasta obtener un nivel de detalle suficiente para su implementación. Asimismo, se fue reordenando la pila del producto en base a las prioridades de cada momento, hasta dar lugar a la versión final que se muestra a continuación.

1. Como usuario quiero visualizar el menú de la aplicación para poder acceder a las distintas funcionalidades. Dicho menú será un menú lateral desplegable desde el lateral izquierdo.
2. Como usuario quiero seleccionar el idioma de la aplicación para visualizar su contenido en dicho lenguaje. La selección podrá realizarse desde una entrada del menú de la aplicación.
3. Como usuario quiero visualizar en forma de lista las instancias de cada tipo de entidad de la aplicación. Cada entrada de la lista mostrará el campo de la entidad seleccionado como cadena a mostrar en la configuración del proyecto. La aplicación deberá cargar las entidades de forma dinámica a medida que se alcance el final de la lista.
4. Como usuario quiero acceder a los formularios descriptivos para visualizar los datos asociados a las entidades de la aplicación. Dichos formularios serán un conjunto de campos de distintos tipos asociados a las propiedades de las entidades.
5. Como usuario quiero acceder a los formularios creables para crear nuevas instancias de las entidades de la aplicación. Dichos formularios deberán contar con diferentes tipos de campos asociados a los distintos tipos de datos de la entidad.
6. Como usuario quiero acceder a los formularios editables para modificar los datos asociados a las instancias de las entidades de la aplicación. Dichos formularios deberán contar con diferentes tipos de campos dependiendo del tipo de dato a modificar.
7. Como usuario quiero tener un mecanismo de borrado para eliminar las instancias de las entidades de la aplicación. Dicho mecanismo será un botón en los formularios. La acción de borrado podrá mostrar una alerta de confirmación.
8. Como usuario quiero acceder a los formularios asociados a las entidades listadas de la aplicación a través de enlaces. En concreto, se deberán mostrar:
  - Un enlace en la entrada de la lista que permita acceder al formulario descriptivo que muestra la información de la entidad.

- Un botón que permita acceder al formulario editable de la entidad para modificar sus datos.
  - Un botón que permita eliminar la entidad.
9. Como usuario quiero filtrar el contenido de las listas para ver solo aquellos elementos que cumplan un determinado criterio. Dicho filtrado consistirá en una barra de búsqueda que permitirá introducir uno o varios términos, y cuyo resultado mostrará aquellas entidades de la lista que contengan en alguno de sus campos el término o términos introducidos.
10. Como usuario quiero filtrar el contenido de las listas para ver solo aquellos elementos que cumplan un determinado criterio. Dicho filtrado consistirá en un conjunto de campos que permitirán establecer el criterio que debe cumplir cada propiedad de la entidad de forma individual mediante el uso de distintos tipos de mecanismos de introducción de datos.
11. Como usuario quiero poder ordenar las listas de la aplicación, para ver las entradas clasificadas por un determinado criterio. Dicha ordenación podrá realizarse en base a cualquiera de los campos de las entidades listadas, pero únicamente en base a un campo, no se permitirán las ordenaciones combinadas.
12. Como usuario quiero acceder a la posición de un elemento geolocalizado en el mapa mediante un botón asociado a la entrada de la lista de dicha entidad.
13. Como usuario quiero visualizar los elementos de una lista en un mapa. Dicha visualización mostrará la posición de las propiedades geográficas de las entidades listadas, diferenciando las propiedades de cada entidad por colores. Además, al pulsar en un marcador se mostrará una ventana modal con la cadena a mostrar de la entidad correspondiente, junto a un botón que permitirá acceder al detalle de la entidad.
14. Como usuario anónimo quiero autenticarme en la aplicación mediante la introducción de mi usuario y mi contraseña.
15. Como usuario registrado quiero solicitar la recuperación de mi contraseña para recuperar el acceso a mi cuenta. Dicha solicitud se hará mediante la introducción del correo electrónico del usuario.
16. Como usuario quiero visualizar en forma de lista las visitas de la aplicación. Cada entrada de la lista mostrará la descripción de la visita, su fecha, su hora de inicio, su hora de fin y su estado. La aplicación deberá cargar las visitas de forma dinámica a medida que se alcance el final de la lista. En el caso de que exista un usuario autenticado, sólo se mostrarán las visitas a las que tenga acceso.

17. Como usuario quiero acceder al detalle de una visita para visualizar sus datos asociados. Para cada visita se mostrarán los siguientes datos: descripción, dirección, fecha, hora de inicio, hora de fin, estado, cliente, empleado y localización. Además, deberá mostrarse una casilla de verificación para modificar el estado de la visita, esto es, marcarla como realizada o como pendiente.
18. Como usuario quiero acceder al formulario de edición de una visita para modificar sus datos asociados. Dicho formulario mostrará distintos tipos de campos para modificar los siguientes campos: descripción, dirección, fecha, hora de inicio, hora de fin, estado, cliente, empleado y localización.
19. Como usuario quiero visualizar en un calendario las visitas de la aplicación para ver la organización temporal de las visitas. Dicho calendario permitirá ver las visitas planificadas para el día seleccionado y diferentes tipos de vistas (mensual, semanal, etc.). Además, se permitirá filtrar las visitas por descripción y estado, de forma que solo se muestren aquellas que cumplen los criterios especificados. En el caso de que exista un usuario autenticado, sólo se mostrarán las visitas a las que tenga acceso.
20. Como usuario quiero visualizar en un mapa las visitas de la aplicación para ver la posición geográfica de las visitas. El mapa deberá mostrar las localizaciones de las visitas con marcadores de diferentes colores. Además, al pulsar en un marcador se mostrará una ventana modal con los siguientes datos de la visita: descripción, dirección, fecha, hora de inicio, hora de fin, cliente y estado.
21. Como usuario quiero visualizar la ruta a una determinada localización en el mapa. Esta funcionalidad se añadirá en las ventanas modales que muestran los datos de un elemento del mapa (visita o entidad) en forma de un botón que abrirá la aplicación Google Maps y mostrará la ruta al elemento seleccionado.
22. Como usuario quiero acceder al detalle de un cliente para visualizar sus datos asociados. Para cada cliente se mostrarán los siguientes datos: nombre completo, correo electrónico, teléfono, dirección y localización.
23. Como usuario quiero acceder al detalle de un empleado para visualizar sus datos asociados. Para cada empleado se mostrarán los siguientes datos: nombre completo, correo electrónico, teléfono y localización.

### 5.3 Arquitectura del sistema

En este apartado se describe la arquitectura general de la aplicación, realizando una descomposición de alto nivel de los componentes que la conforman, así como su comunicación

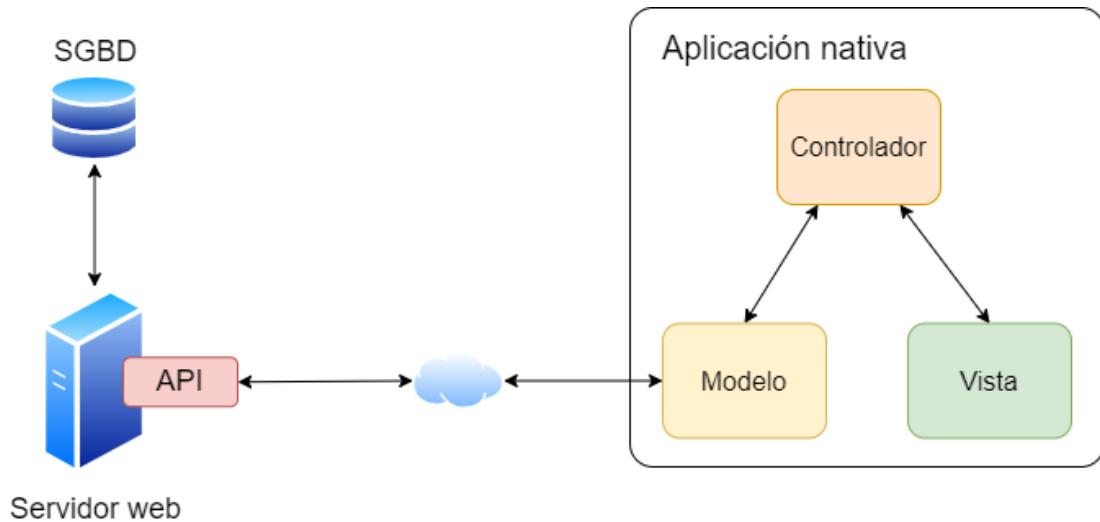


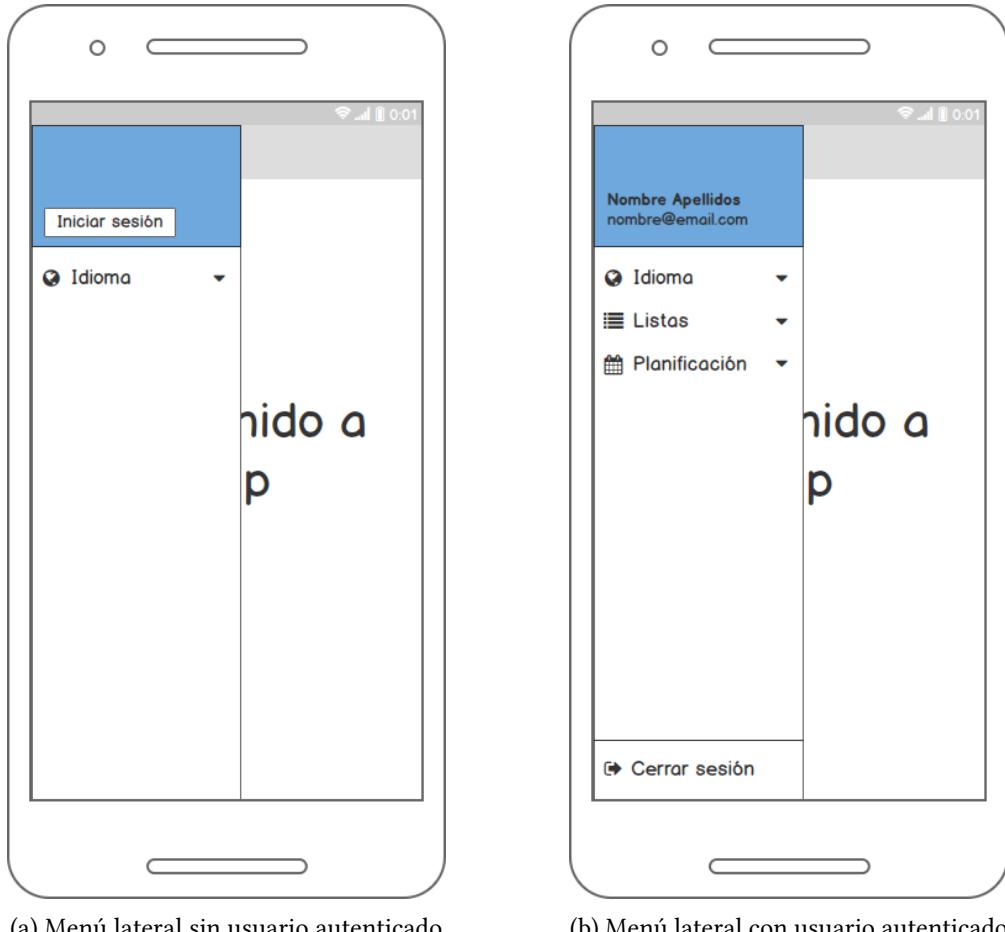
Figura 5.2: Arquitectura general del sistema

con el servidor web.

Como se puede ver en la figura 5.2, la estructura interna de la aplicación sigue el patrón de arquitectura Modelo-Vista-Controlador. Esta estructura separa la presentación e interacción de los datos del sistema y está compuesto por tres componentes lógicos, que son los siguientes [38]:

- **Modelo:** se encarga de la gestión de los datos del sistema y las operaciones asociadas a esos datos, como son la persistencia y recuperación de la información.
- **Vista:** define y gestiona cómo se presentan los datos al usuario, es decir, es la parte que se encarga de definir la interfaz del sistema y de recibir las interacciones del usuario.
- **Controlador:** se encarga de gestionar las interacciones del usuario recibidas a través de la vista y realiza las peticiones correspondientes al modelo. De esta forma, hace de intermediario entre los otros dos componentes, enviando los datos a la vista para que sean mostrados y solicitando las operaciones sobre los datos al modelo.

Si tenemos en cuenta la integración de la aplicación con el servidor web, el sistema sigue una arquitectura cliente-servidor, en la que la parte cliente es la aplicación desarrollada. Esta arquitectura favorece la separación e independencia entre ambos elementos, facilitando que sean intercambiados o modificados sin afectar a otras partes del sistema y, además, permite la existencia de diferentes clientes para un mismo servidor [38]. De esta forma, es posible que existan múltiples plataformas con diferentes instalaciones de la aplicación que se comuniquen con el mismo servidor.



(a) Menú lateral sin usuario autenticado

(b) Menú lateral con usuario autenticado

Figura 5.3: Menú de la aplicación sin usuario autenticado y con usuario autenticado

## 5.4 Interfaz de usuario

En esta sección se pretende realizar una descripción de alto nivel de la interfaz de usuario de la aplicación, reflejando la estructura general de las pantallas más importantes que la componen, así como la navegación entre ellas.

En la fase de análisis preliminar se llevó a cabo el diseño detallado de la interfaz de la aplicación mediante el uso de prototipos, con el fin de facilitar su futura implementación y favorecer el análisis de requisitos. En esta sección se muestran únicamente los prototipos más importantes, el diseño de todas las pantallas puede verse en el apéndice B.

En la figura B.1 se muestra la pantalla principal de la aplicación, en la que se muestra un mensaje de bienvenida. Pulsando en el ícono de la esquina superior izquierda o deslizando desde el lateral izquierdo se accede al menú de la aplicación.

Este menú muestra distinto contenido dependiendo de si hay algún usuario autenticado

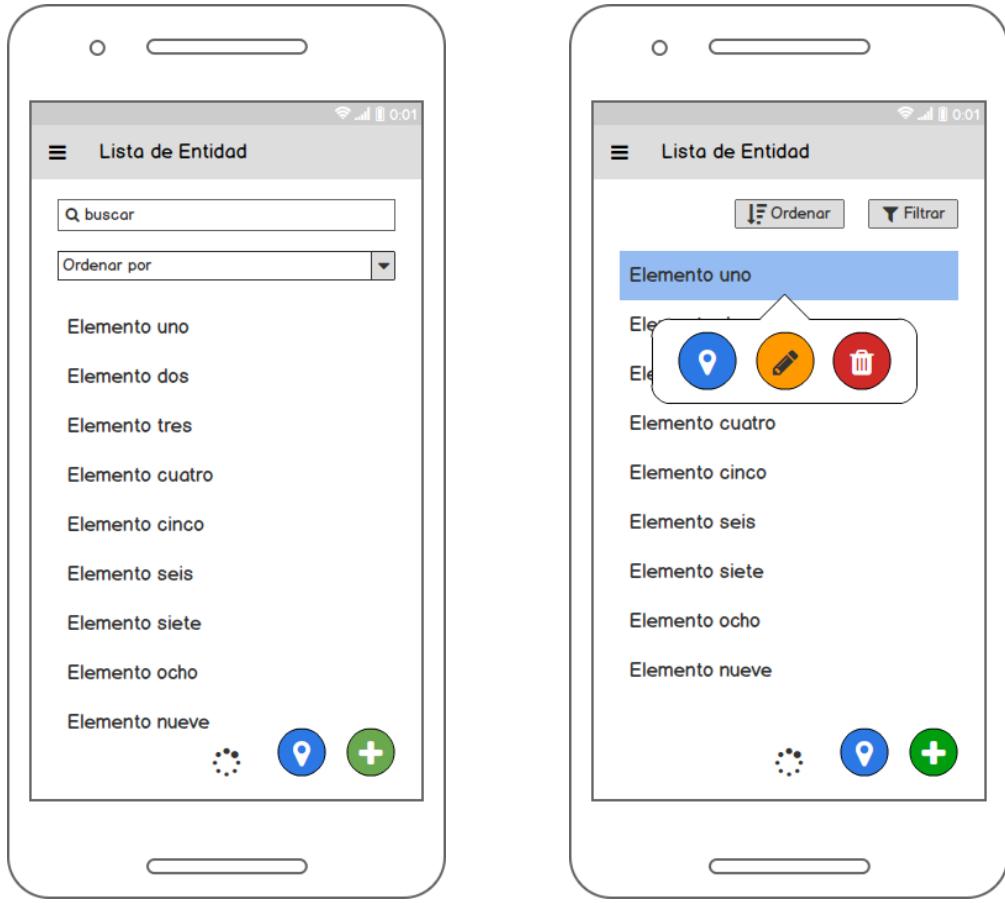


Figura 5.4: Pantallas de iniciar sesión y de recuperar la contraseña

o no. Para el caso de que no haya ningún usuario autenticado se muestra el contenido de la figura 5.3a, es decir, un botón en la cabecera que permite acceder a la página de inicio de sesión y un desplegable que muestra los idiomas disponibles para la aplicación.

Pulsando en el botón de iniciar sesión se redirige a la pantalla que se muestra en la figura 5.4a, en la que es necesario introducir el nombre de usuario y la contraseña para acceder. Desde esta pantalla se puede acceder a la vista de recuperación de contraseña (figura 5.4b) a través del enlace que pone “¿Olvidó su contraseña?”. En esta pantalla es necesario introducir una dirección de correo electrónico para que el servidor envíe un correo electrónico con las instrucciones de recuperación.

Una vez se ha autenticado un usuario en la aplicación, el menú muestra el contenido que aparece en la figura 5.3b. Como se puede ver, ahora se muestra el nombre y apellidos del usuario junto a su dirección de correo electrónico en la cabecera del menú, y tres desplegables en el cuerpo. El primero de ellos muestra los idiomas disponibles y permite seleccionar el lenguaje

Figura 5.5: Pantallas de lista con búsqueda y de lista con filtros y *popup*

de la aplicación; el segundo desplegable contiene los enlaces a los listados de las entidades del modelo de datos; y el último desplegable contiene las entradas al listado, calendario y mapa de visitas. Por último, en el pie del menú se muestra un botón que permite al usuario cerrar sesión.

Accediendo a alguno de los listados de las entidades se navega a una pantalla como alguna de las que se muestran en la figura 5.5. En esta vista existe una entrada por cada instancia del tipo de entidad listada y un conjunto de funcionalidades. Dependiendo de las características seleccionadas para el producto, dicha lista puede soportar una búsqueda simple (figura 5.5a) o un filtrado por los campos de las entidades (5.5b). En el primer caso se muestra un campo de texto que permite introducir los términos de búsqueda y en el segundo caso se incluye un botón que redirige a un formulario que permite introducir los valores deseados para cada propiedad (figura B.10). Además, también se permite la ordenación de las entradas, para lo que

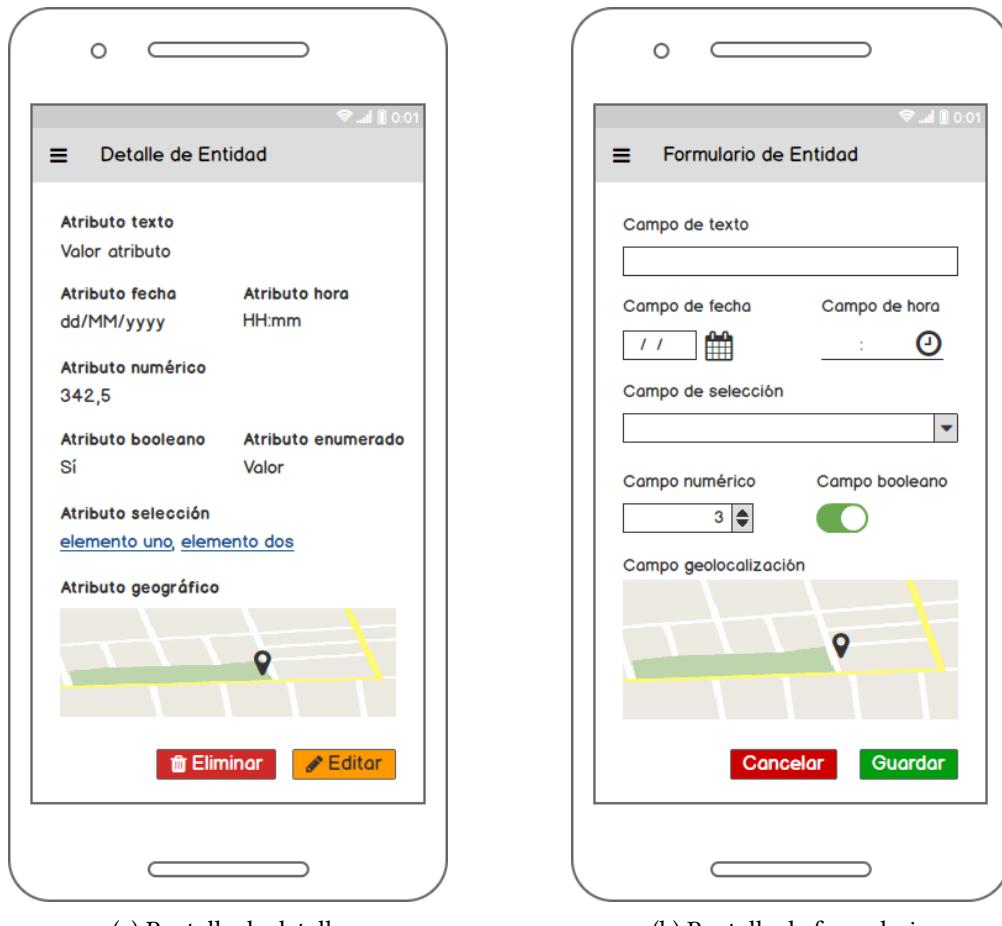
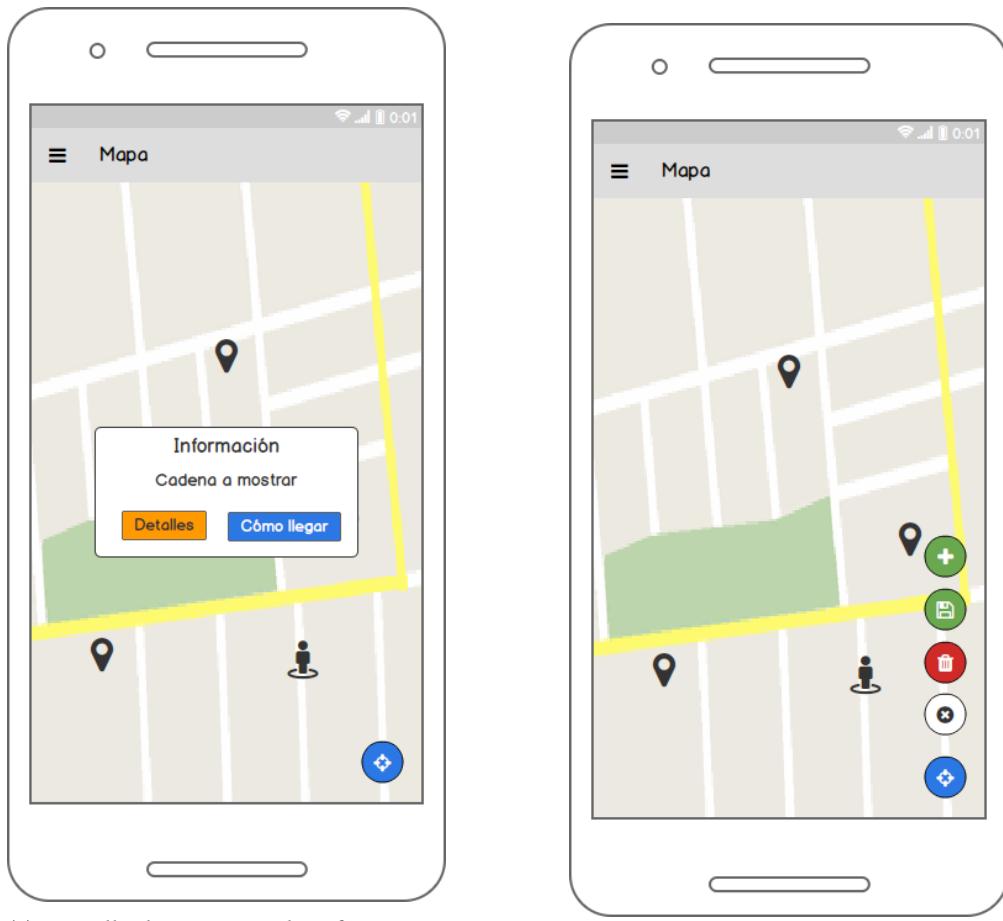


Figura 5.6: Detalle y formulario de una entidad

se muestra un desplegable que permite seleccionar la propiedad por la que se desea ordenar (figura B.13).

Desde esta vista también es posible crear una nueva entidad pulsando en el botón flotante verde que se muestra en la esquina inferior derecha, o ver la lista como un mapa pulsando en el botón flotante azul. Asimismo, pulsando en alguna de las entradas de la lista se accede a su vista de detalle y manteniendo una pulsación larga se muestra un menú *popup* (figura 5.5b) con tres entradas que permiten acceder a la ubicación de la entidad en el mapa, a su formulario de modificación y a su eliminación respectivamente.

La pantalla de detalle de una entidad puede verse en la figura 5.6a y muestra los valores de cada propiedad de la entidad en un formato adecuado al tipo de dato del campo. En el caso de las propiedades geográficas se muestra una miniatura de un mapa y al pulsar sobre él se accede a la vista de mapa, que se representa en la figura B.14 y en la que existe un botón para centrar la ubicación del usuario. Asimismo, las propiedades que hacen referencia a una

Figura 5.7: Pantallas de mapa con *popup* y de mapa editando

relación entre entidades se muestran como un enlace que permite navegar a la pantalla de detalle de la entidad correspondiente.

Desde esta vista de detalle es posible acceder al formulario de modificación de la entidad y también eliminarla. A la hora de eliminar una entidad desde cualquiera de las opciones disponibles, se muestra una ventana modal de confirmación como la que se puede ver en la figura B.9.

El formulario de modificación se muestra en la figura 5.6b y es el mismo que el utilizado a la hora de crear una entidad. Como puede verse, está formado por un conjunto de campos de entrada que permiten introducir el valor de las propiedades de la entidad, ajustándose al tipo de dato de cada una. En este caso, al pulsar sobre el campo de una propiedad geográfica se redirige a la pantalla que se muestra en la figura B.16. Esta pantalla muestra un mapa y un botón flotante que permite editar su contenido. Pulsando en ese botón se despliegan las



Figura 5.8: Pantallas de calendario de visitas y de mapa con información de una visita

funcionalidades de la figura 5.7b que permiten añadir un elemento (en el caso de propiedades múltiples) y guardar o eliminar los cambios.

Volviendo a los listados, pulsando en el botón flotante azul se navega a la pantalla que se muestra en la figura 5.7a. En esta pantalla se muestran las propiedades geográficas de las entradas de la lista diferenciadas por colores y al pulsar sobre un elemento del mapa se muestra una ventana modal con la cadena a mostrar de la entidad correspondiente. Esta ventana también permite acceder a la vista de detalle de la entidad u obtener la ruta al lugar seleccionado mediante la opción “Cómo llegar”.

Desde el menú también es posible acceder a los elementos de planificación a partir de su entrada correspondiente. El primero de ellos es un listado de las visitas, que se puede ver en la figura B.18 y cuyo diseño es igual al de los listados de entidades. Al pulsar sobre una de las entradas de la lista se navega a la pantalla de detalle de una visita (figura B.22) que, al igual

que las vistas de detalle de las entidades, muestra los valores de las propiedades de la visita en el formato correspondiente al tipo de dato. En este caso se incluye una casilla de verificación en la esquina superior derecha para marcar la visita como completada. Desde esta vista de detalle es posible acceder al formulario de modificación de la visita (figura B.23) que permite editar los valores de sus propiedades.

El segundo elemento de planificación es el calendario de visitas, cuyo diseño se muestra en la figura 5.8a. Como se puede ver, en esta pantalla se presenta un calendario en el que se indica para cada día el número de visitas asociadas y debajo de este, la lista de visitas asignadas al día seleccionado. Cada entrada de esta lista permite acceder a la pantalla de detalle de la visita, que se ha explicado previamente. Además, el calendario soporta distintos tipos de vistas (figura B.20) y filtrado de sus elementos (figura B.21).

El último elemento de planificación es el mapa de visitas, que se muestra en la figura B.24. En este mapa se representan las propiedades geográficas de las visitas en colores diferentes y al pulsar sobre un elemento del mapa se muestra una ventana modal con la información asociada a la visita y la opción de obtener la ruta hasta su ubicación, como se puede ver en la figura 5.8b. Este mapa también permite filtrado (figura B.26).

Es importante tener en cuenta que la variabilidad definida para la línea de producto software influyó en el diseño de la interfaz de usuario del producto de forma que una misma funcionalidad puede mostrarse de manera diferente dependiendo de la existencia o no de otras características. Por ejemplo, la funcionalidad de ordenación de las listas se muestra de forma diferente si se ha seleccionado la búsqueda simple o si se ha seleccionado el filtrado por campos.

Asimismo, hay que destacar que el diseño de las pantallas aquí expuestas se ha realizado teniendo en cuenta todas las posibles características de un producto de la LPS, pero a la hora de generar las aplicaciones, el diseño de su interfaz web variará en base a las características seleccionadas.

## 5.5 Modelo conceptual de datos

En este apartado se expone el modelo de datos de los componentes fijos de la aplicación, es decir, aquellos elementos que se han definido como necesarios para las aplicaciones de gestión del trabajo en movilidad y que son independientes del ámbito de aplicación de cada empresa, por lo que no están reflejados en el modelo de datos genérico definido en la sección 4.2.

Este modelo de datos puede verse en la figura 5.9 y de su diseño cabe destacar lo siguiente:

- **Visit:** representa una visita, es decir, una cita planificada que tiene el empleado para realizar un trabajo. Esta entidad almacena los datos de una visita y puede tener un

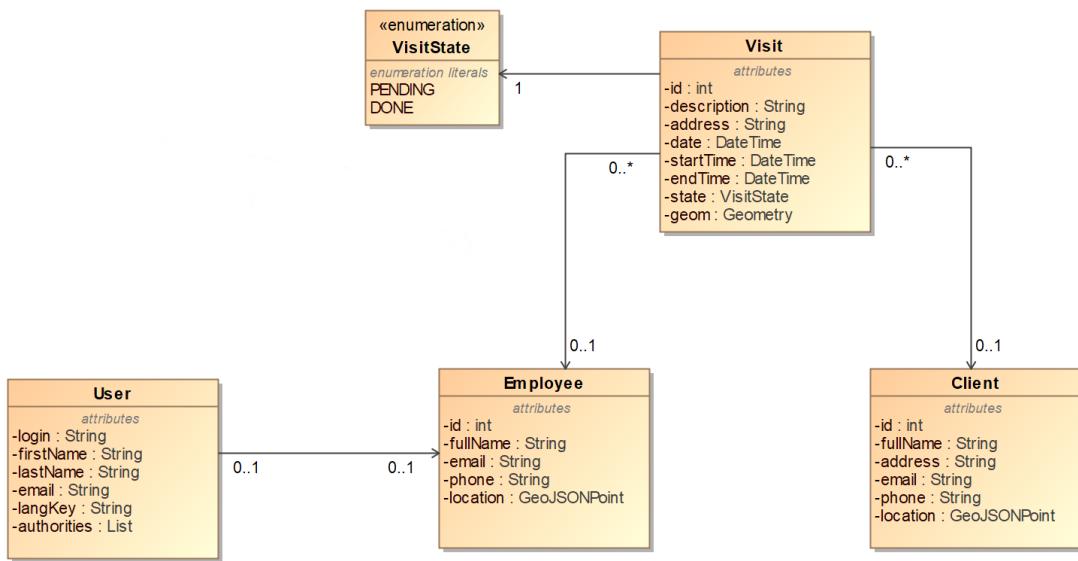


Figura 5.9: Modelo conceptual de datos

empleado asociado y/o un cliente asociado. El estado de las visitas se representa con el enumerado *VisitState*.

- **Client:** representa a un cliente de la empresa. Esta entidad almacena los datos personales del cliente y puede estar asociada a múltiples visitas.
- **Employee:** representa un empleado de la empresa. Esta entidad almacena los datos personales del empleado y puede estar asociada a múltiples visitas y/o a una cuenta de usuario.
- **User:** representa a un usuario de la aplicación. Esta entidad almacena los datos personales del usuario y puede estar asociada a un empleado.

Es importante resaltar que el modelo de datos aquí descrito está compuesto por todos los posibles elementos fijos que pueden formar parte de un producto de la línea de producto software, pero a la hora de generar una aplicación, su modelo de datos variará en función de las características que hayan sido seleccionadas para el mismo, pudiendo no incluir ninguno de estos elementos.

# Capítulo 6

# Diseño

---

En este capítulo se detallan los aspectos más relevantes del diseño de la aplicación.

## 6.1 Arquitectura tecnológica del sistema

En esta sección se pretende complementar el análisis de la arquitectura realizado en la sección 5.3 (figura 5.2), a través del estudio de las tecnologías empleadas en la aplicación. La figura 6.1 ilustra la arquitectura tecnológica.

### 6.1.1 Aplicación nativa

A la hora de llevar a cabo el desarrollo del producto, se estableció la necesidad de que la solución propuesta fuese multiplataforma. Es por esto por lo que se decidió implementar la aplicación haciendo uso de la tecnología Flutter. Flutter es un SDK (*Software Development Kit* o kit de desarrollo software) desarrollado por Google que permite construir aplicaciones compiladas de forma nativa para dispositivos móviles, web, de escritorio e integrados a partir de una única base de código [15]. De esta forma, es posible implementar y probar el código de la aplicación una única vez y obtener un producto que puede ser instalado en multitud de dispositivos con sistemas operativos diferentes sin necesidad de realizar varias implementaciones para asegurar la compatibilidad.

Flutter hace uso de Dart en la implementación, un lenguaje de desarrollo de código abierto orientado a objetos y con análisis de tipos estático [16]. Dart cuenta con un conjunto bastante extenso de herramientas integradas, tales como su propio gestor de paquetes, varios compiladores/transpiladores, un analizador y un formateador. Además, proporciona una máquina virtual y un mecanismo de compilación “Just-in-Time” que permite que los cambios realizados en el código se puedan ejecutar inmediatamente [39].

Entre las ventajas de Flutter se puede destacar que el código de una aplicación desarrollada con esta tecnología se compila a código nativo, consiguiendo así un rendimiento muy

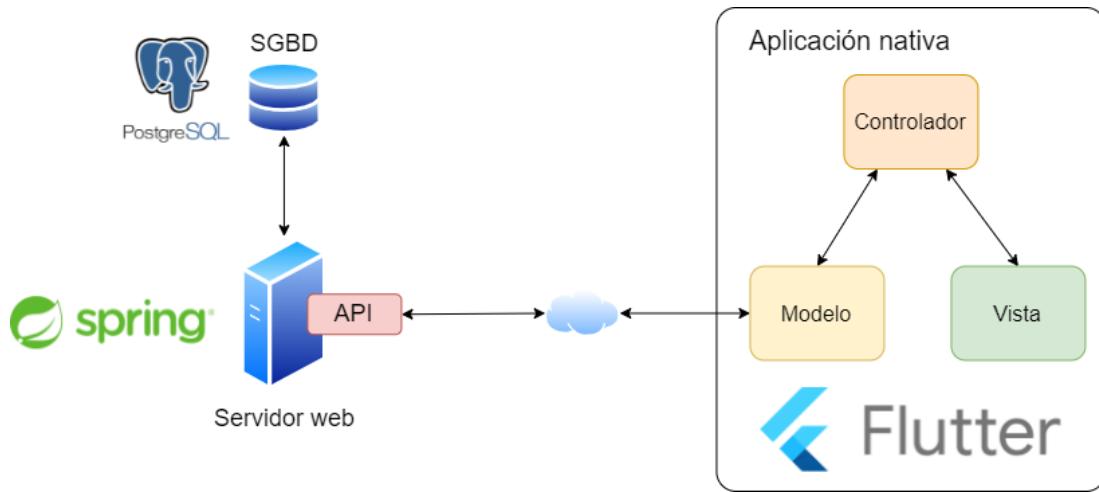


Figura 6.1: Arquitectura tecnológica del sistema

optimizado. Además, Flutter cuenta con sus propios componentes, llamados *widgets*, por lo que la misma aplicación se verá igual en cualquiera de los dispositivos que se instale, independientemente de su sistema operativo o versión, y permite también la construcción de *widgets* propios [39]. Asimismo, está integrado con Material Design, lo que facilita la creación de interfaces flexibles y la aplicación de buenas prácticas de diseño.

En base a la tecnología seleccionada para el desarrollo y a la arquitectura definida en la sección 5.3, la estructura interna de la aplicación móvil está compuesta por los siguientes elementos:

- En la parte **modelo** se han empleado **repositorios**, que son clases que se encargan de la comunicación con el servidor web, mediante la realización de peticiones HTTP. Estas clases solicitan al servidor la ejecución de las operaciones de recuperación y persistencia de la información a través de su API. La principal ventaja de los repositorios es que aíslan la capa de acceso a datos del resto de la aplicación favoreciendo que los cambios en ella, por ejemplo, en la firma de los métodos ofrecidos por la API, no afecten a otras partes del sistema. Los repositorios ofrecen sus servicios al componente controlador a través de un conjunto de métodos.
- En la parte **controlador** se han empleado **controladores**, que son clases encargadas de recibir las peticiones de la vista y enviarle los datos solicitados tras haber invocado a los métodos ofrecidos por el modelo.
- En la parte **vista** se han empleado **vistas**, que son clases que implementan los *widgets* de la aplicación. Estas clases son las encargadas de construir y mostrar la interfaz de usuario representando los datos y ofreciendo un conjunto de funcionalidades. Las

vistas gestionan las interacciones del usuario con la interfaz y realizan las peticiones correspondientes a los controladores.

Entre los paquetes de Dart utilizados en la implementación del producto cabe destacar los siguientes:

- **http** [40]: contiene un conjunto de funciones y clases que facilitan el consumo de recursos HTTP a través de peticiones.
- **global\_configuration** [41]: permite gestionar elementos de configuración de forma única a través de un *singleton*, haciéndolos accesibles desde toda la aplicación.
- **flutter\_secure\_storage** [21]: permite almacenar los datos de la aplicación de forma persistente, proporcionando seguridad mediante encriptación.
- **jwt\_decoder** [42]: permite recuperar los datos almacenados en un JWT, incluida su fecha de expiración.
- **geojson\_v1** [22]: permite crear, leer, buscar, actualizar y eliminar datos geoespaciales (GIS), proporcionando un conjunto de tipos de datos específico.
- **flutter\_map** [19]: es una adaptación de la biblioteca de mapas Leaflet para Flutter.
- **user\_location** [20]: complemento para flutter\_map que añade el soporte para manejar y mostrar la ubicación actual del usuario.

### 6.1.2 Base de datos y servidor

La tecnología empleada para gestionar el almacenamiento de la información por parte del servidor ha sido PostgreSQL [18], un sistema de gestión de bases de datos relacional de código abierto. Para el almacenamiento de los datos de la aplicación se creó una base de datos en la que fue necesario instalar la extensión PostGIS [43] para añadir soporte a los objetos geográficos.

En cuanto al servidor web, está implementado haciendo uso de Java [44], Spring [45] y Gradle [46].

## 6.2 Diseño de la aplicación

En esta sección se detalla la organización interna de los componentes del producto, cuya estructura de paquetes se muestra en la figura 6.2.

Los paquetes más importantes son los siguientes:

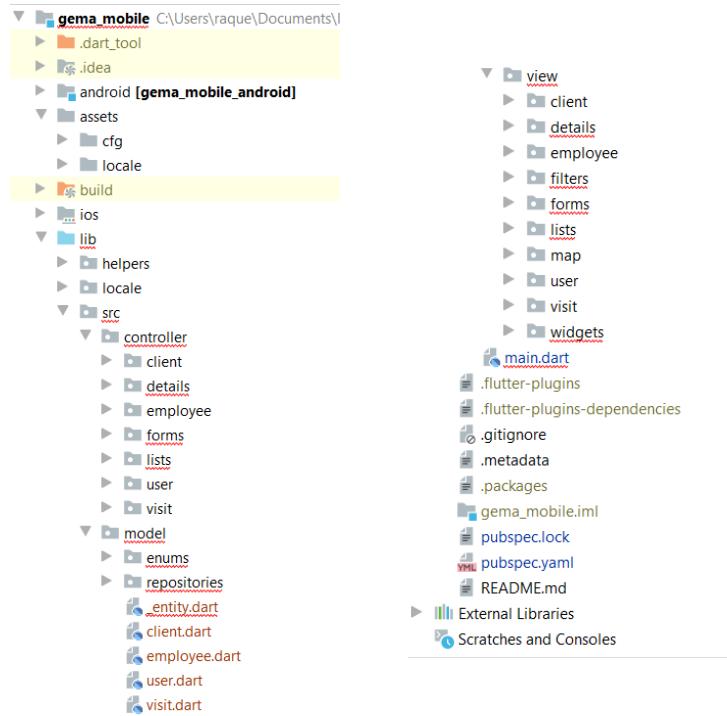


Figura 6.2: Estructura de paquetes de la aplicación

- **android**: contiene el proyecto de la aplicación nativa para Android.
- **assets**: contiene los recursos estáticos de la aplicación. Internamente está subdividido en dos paquetes:
  - **cfg**: contiene un fichero JSON en el que está configurada la IP del servidor web.
  - **locale**: contiene los ficheros JSON con los mensajes internacionalizados de la aplicación.
- **build**: contiene el código compilado de la aplicación nativa. Se genera automáticamente como resultado del proceso de compilación.
- **ios**: contiene el proyecto de la aplicación nativa para iOS.
- **lib**: contiene los archivos *.dart* que implementan el código de la aplicación. Internamente está estructurado de la siguiente forma:
  - **helpers**: contiene el archivo encargado de gestionar el acceso a los datos almacenados por la aplicación, esto es, un *singleton* que implementa FlutterSecureStorage.
  - **locale**: contiene los archivos que implementan los mecanismos para permitir la internacionalización de la aplicación.

- **src**: contiene el código principal de la aplicación. Internamente está estructurado según los componentes lógicos definidos para su arquitectura:
  - \* **controller**: contiene los controladores encargados de comunicar las vistas y los repositorios. Estos controladores se agrupan de la siguiente forma:
    - **client**: contiene los controladores encargados de gestionar las vistas relacionadas con los clientes.
    - **details**: contiene los archivos que generarán los controladores encargados de gestionar las vistas de detalle de las entidades del modelo de datos.
    - **employee**: contiene los controladores encargados de gestionar las vistas relacionadas con los empleados.
    - **forms**: contiene los archivos que generarán los controladores encargados de gestionar las vistas de formularios y filtros de las entidades del modelo de datos.
    - **lists**: contiene los archivos que generarán los controladores encargados de gestionar las vistas de lista de las entidades del modelo de datos.
    - **user**: contiene los controladores encargados de gestionar las vistas relacionadas con el usuario autenticado.
    - **visit**: contiene los controladores encargados de gestionar las vistas relacionadas con las visitas.
  - \* **model**: contiene los archivos encargados de la gestión de los datos. Internamente está estructurado del siguiente modo:
    - **enums**: contiene los archivos que generarán los enumerados definidos en el modelo de datos.
    - **repositories**: contiene los archivos que generarán los repositorios encargados de realizar las peticiones al servidor web.
    - **\_entity.dart**: es el archivo encargado de generar las entidades definidas en el modelo de datos.
    - **client.dart**: es el archivo que representa la entidad *Client*.
    - **employee.dart**: es el archivo que representa la entidad *Employee*.
    - **user.dart**: es el archivo que representa la entidad *User*, esto es, el usuario autenticado en la aplicación.
    - **visit.dart**: es el archivo que representa la entidad *Visit*.
  - \* **view**: contiene las vistas y los *widgets* de la aplicación. Estos elementos están agrupados de la siguiente forma:
    - **client**: contiene las vistas relacionadas con la gestión de los clientes.

- **details**: contiene los archivos que generarán las vistas que muestran los detalles de las entidades definidas en el modelo de datos.
  - **employee**: contiene las vistas relacionadas con la gestión de los empleados.
  - **filters**: contiene los archivos que generarán las vistas que muestran los filtros para los listados de las entidades definidas en el modelo de datos.
  - **forms**: contiene los archivos que generarán las vistas que muestran los formularios de las entidades definidas en el modelo de datos.
  - **lists**: contiene los archivos que generarán las vistas que muestran las listas de las entidades definidas en el modelo de datos.
  - **map**: contiene la vista del mapa.
  - **user**: contiene las vistas relacionadas con la gestión del usuario autenticado.
  - **visit**: contiene las vistas relacionadas con la gestión de las visitas.
  - **widgets**: contiene los *widgets* desarrollados que son comunes a toda la aplicación.
- **main.dart**: es el archivo principal de la aplicación, encargado de su ejecución.

A continuación se detallan los elementos de los paquetes más importantes y la estructura general de clases de la aplicación.

### 6.2.1 Entidades

Son las clases que representan los datos manejados por la aplicación. Representan tanto a las entidades definidas en el modelo de datos personalizado para el producto como a las entidades del modelo de datos estático (figura 5.9). Estas clases cuentan con una serie de atributos privados que representan las propiedades definidas para cada entidad y sus relaciones con otras entidades. Para permitir el acceso a estos atributos, cuentan con métodos de lectura y escritura (*getters* y *setters*). Además, tienen definidos un constructor con argumentos y otro vacío, así como métodos para codificar y decodificar las entidades en formato JSON.

### 6.2.2 Repositorios

Son los elementos encargados de llevar a cabo la comunicación con el servidor. Estas clases realizan las peticiones HTTP al servidor solicitando la ejecución de operaciones sobre los datos (lectura, creación, modificación y borrado) y procesan las respuestas para enviar la información a los controladores. Existe un único repositorio por cada entidad que agrupa las peticiones relacionadas con la gestión de sus datos y atiende las llamadas realizadas por todos los controladores que gestionan las vistas de esa entidad.

### 6.2.3 Controladores

Son las clases encargadas de comunicar las vistas y los repositorios. Estas clases gestionan las peticiones de la vista e invocan a los métodos ofrecidos por los repositorios, para posteriormente procesar los datos recibidos y enviarlos de vuelta a la vista. Existe un controlador por cada vista, excepto en algunos casos en los que las vistas gestionan los mismos aspectos de una entidad como es el caso de los formularios y los filtros.

### 6.2.4 Vistas

Son los elementos encargados de mostrar la interfaz de usuario. Las vistas construyen la interfaz mediante el uso de *widgets*, a través de los cuales muestran los datos a los usuarios y ofrecen las funcionalidades de la aplicación. Los *widgets* proporcionan los mecanismos de interacción con el usuario y las vistas se encargan de gestionar estas interacciones y enviar las peticiones correspondientes a los controladores para efectuar las operaciones solicitadas sobre los datos. En el diseño e implementación de las vistas se han empleado los *widgets* de Material Design y se han construido algunos *widgets* propios. Existen varias vistas para cada entidad, cada una de ellas asociada a un controlador.

### 6.2.5 Estructura de clases

En este apartado se explica a nivel general la estructura de clases de los componentes más importantes de la aplicación.

Por una parte, en la figura 6.3 se muestran las clases asociadas a cada entidad del modelo de datos personalizado. Como puede verse, existe una clase que representa la entidad en la aplicación y un repositorio para la gestión de sus datos. Cada entidad tiene tres controladores y cuatro vistas: un controlador y una vista para gestionar los detalles de la entidad; un controlador y una vista para gestionar la lista de instancias de la entidad; y un controlador y dos vistas para gestionar los formularios de la entidad.

Por otra parte, en la figura 6.4 se muestran las clases asociadas a la gestión de las visitas. Como puede verse, existe una clase que representa a una visita en la aplicación y un repositorio para gestionar sus datos. Cada visita tiene cinco controladores y seis vistas: un controlador y una vista para gestionar el calendario de visitas; un controlador y una vista para gestionar los detalles de la visita; un controlador y una vista para gestionar los formularios de las visitas; un controlador y una vista para gestionar la lista de visitas; y un controlador y una vista para gestionar el mapa de visitas. Esta última vista hace uso de dos *widgets*: una ventana modal que muestra los filtros sobre las visitas del mapa, y una ventana modal que muestra la información de una visita del mapa.

En cuanto al usuario autenticado, en la figura 6.5 se pueden ver sus clases asociadas, que

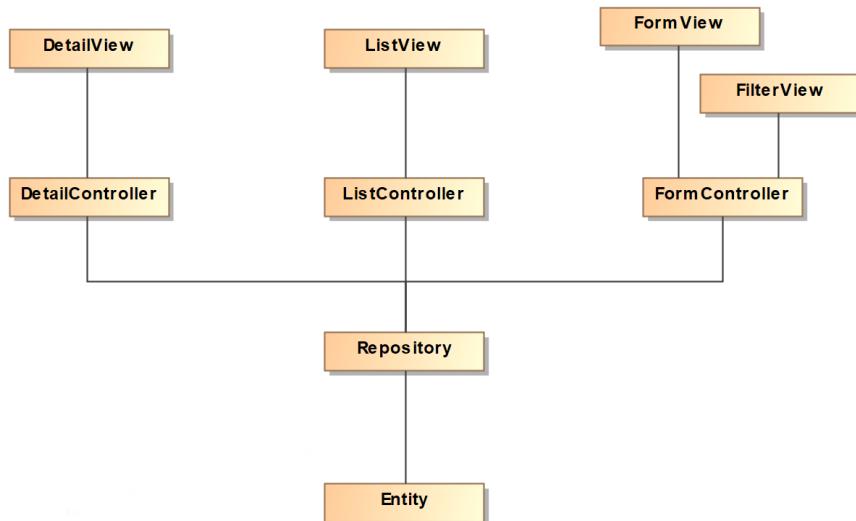


Figura 6.3: Diagrama de clases de una entidad

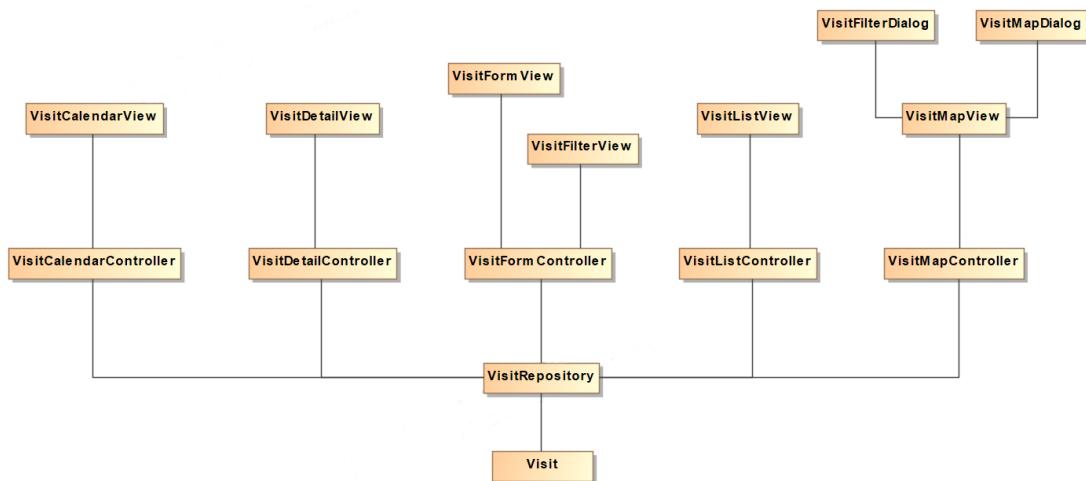


Figura 6.4: Diagrama de clases de visita

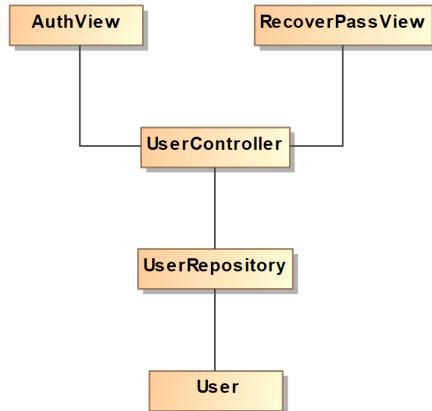


Figura 6.5: Diagrama de clases de usuario

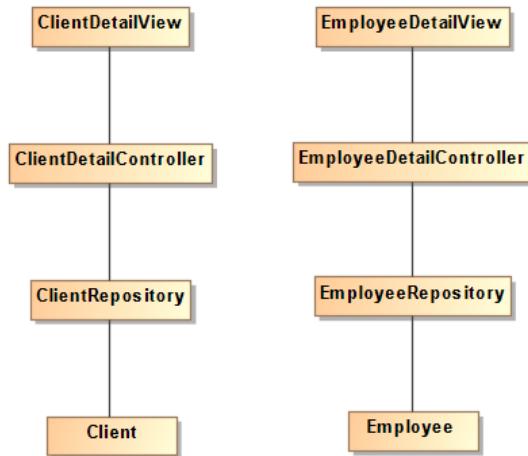


Figura 6.6: Diagrama de clases de cliente y empleado

son: una entidad que representa al usuario, un repositorio, un controlador y dos vistas (una para la autenticación y la otra para la recuperación de la contraseña).

Por último, la figura 6.6 muestra las clases asociadas a los clientes y empleados de la aplicación. Ambas entidades tienen las mismas clases: una clase que representa la entidad, un repositorio, y un controlador y una vista para gestionar sus detalles.



## Capítulo 7

# Implementación y pruebas

---

En este capítulo se detallan los aspectos de la implementación más complejos y otras cuestiones relevantes que complementan la información de las secciones anteriores. Para comprender los aspectos de la implementación del producto es necesario entender el funcionamiento de la línea de producto software por lo que previamente a la lectura de este capítulo es necesario revisar las secciones 8.1 y 8.2.

## 7.1 Implementación

### 7.1.1 Entidades

Uno de los aspectos más relevantes de la codificación fue la implementación de las entidades de la aplicación. Debido a la total dependencia de esta parte de la aplicación con el modelo de datos definido para el producto generado por la LPS, la mayor parte del código de esta clase está formado por anotaciones del derivador.

En el siguiente fragmento se muestra el código inicial del fichero a partir del que se generarán las entidades:

```
1 /*%@ return data.dataModel.entities.map(function(entity) {
2     return {
3         fileName: normalize(entity.name) + '.dart',
4         context: entity
5     };
6 }
7 %*/
8
9 /*% constructor_props = ""; %*/
10
11 import 'dart:convert';
12 import 'package:geojson_vi/geojson_vi.dart';
13 import 'package:json_annotation/json_annotation.dart';
```

```

14 import 'package:intl/intl.dart';
15 /*% context.properties.forEach(function(prop) {
16     var propertyClass = prop.class;
17     var propertyIsEntity =
18         data.dataModel.entities
19             .map(function(entity) { return entity.name; })
20             .indexOf(propertyClass) != -1;
21     var propertyIsEnum =
22         data.dataModel.enums
23             .map(function(entity) { return entity.name; })
24             .indexOf(propertyClass) != -1;
25     if(propertyIsEntity) { %*/
26 import 'package:gema_mobile/src/model/*%= normalize(propertyClass) %*/.dart';
27 /*% } %*/
28 /*% if(propertyIsEnum) { %*/
29 import 'package:gema_mobile/src/model/enums/*%= normalize(propertyClass) %*/.dart';
30 /*% } %*/
31 /*% ); %*/

```

Como se puede ver, en la parte superior se ha definido una anotación de generación de ficheros con la función que se encargará de generar los archivos de las entidades. Esta función recorre la lista de entidades definidas en la especificación y establece que los ficheros generados tendrán un nombre del tipo *entity.dart* y que su contenido (*context*) será la propia entidad.

A continuación se inicializa una variable JavaScript en la que se almacenarán las propiedades definidas para el constructor.

Seguidamente se definen las importaciones de la clase. En este caso, hay una serie de *imports* fijos de algunos paquetes de Dart y posteriormente una anotación con un bucle. En este bucle se recorren las propiedades definidas para la entidad del fichero generado, es decir, la entidad definida en el *context*, y sobre ellas se realizan algunas comprobaciones. En primer lugar, se crean tres variables en las que se almacenan: la clase de la propiedad (*propertyClass*), un indicador de si la propiedad es una entidad (*propertyIsEntity*), como sería el caso de las propiedades que representen relaciones entre entidades, y, por último, un indicador de si la propiedad es un enumerado. En base a estas variables se definen las comprobaciones necesarias para determinar los *imports*, añadiendo las rutas a los enumerados y a las entidades relacionadas.

A partir de aquí comienza la definición de la clase, cuyo primer tramo se puede ver a continuación:

```

1 @JsonSerializable(explicitToJson: true)
2 class /*%= normalize(context.name, true) %*/ {
3     /*% context.properties.forEach(function(prop) {
4         var propertyClass = prop.class;

```

```
5     propertyClass = propertyClass.split(' ')[0];
6     if (propertyClass == 'Boolean') propertyClass = 'bool';
7     if (propertyClass == 'Integer') propertyClass = 'int';
8     if (propertyClass == 'Long') propertyClass = 'int';
9     if (propertyClass == 'Float') propertyClass = 'double';
10    if (propertyClass == 'Double') propertyClass = 'double';
11    if (propertyClass == 'Text') propertyClass = 'String';
12    if (propertyClass == 'Date') propertyClass = 'DateTime';
13    if (propertyClass == 'DateTime') propertyClass = 'DateTime';
14    if (propertyClass == 'Point') propertyClass = 'GeoJSONPoint';
15    if (propertyClass == 'MultiPoint') propertyClass = 'GeoJSONMultiPoint';
16    if (propertyClass == 'LineString') propertyClass = 'GeoJSONLineString';
17    if (propertyClass == 'MultiLineString') propertyClass =
18      'GeoJSONMultiLineString';
19    if (propertyClass == 'Polygon') propertyClass = 'GeoJSONPolygon';
20    if (propertyClass == 'MultiPolygon') propertyClass = 'GeoJSONMultiPolygon';
21    if (propertyClass == 'Geometry') propertyClass = 'GeoJSONGeometryCollection';
22
23    var propertyIsEnum =
24      data.dataModel.enums
25        .map(function(en) { return en.name; })
26        .indexOf(propertyClass) != -1;
27    if (prop.multiple) {
28      propertyClass = 'List<' + normalize(propertyClass, true) + '>';
29    }
30    if (propertyIsEnum) {
31      propertyClass = normalize(propertyClass, true);
32    }
33
34    if (constructor_props.length == 0) {
35      constructor_props += 'this._' + normalize(prop.name);
36    } else {
37      constructor_props += ', this._' + normalize(prop.name);
38    } %*/
39 /*%= propertyClass %*/ /*%= '_' + normalize(prop.name) */;
40
41 /*%= propertyClass %*/ get /*%= normalize(prop.name) */ => /*%= '_'
42   normalize(prop.name) */;
43 set /*%= normalize(prop.name) *//*%*/ /*%= propertyClass %*/ /*%= normalize(prop.name)
44   */ {
45   /*%= '_' + normalize(prop.name) */ = /*%= normalize(prop.name) */;
46 } /*% */); /**/
```

Inicialmente se anota la clase con `@JsonSerializable` para permitir convertir el código de la clase a JSON y viceversa.

A continuación, se define el nombre de la clase con una anotación de interpolación en la que se llama a la función `normalize`. Esta función está definida en el fichero `extra.js` del derivador y se encarga de normalizar el texto que recibe como parámetro (eliminando caracteres raros, espacios, etc.). El segundo parámetro especifica si el texto debe empezar por letra mayúscula. El resultado de aplicar esta anotación de interpolación en el derivador hará que las clases generadas tengan nombres del estilo `Entity`.

Dentro del código de la clase, el primer paso fue definir las propiedades que forman parte de la entidad. Para ello se construyó el bucle que recorre la lista de propiedades definidas para la entidad y a continuación, realiza sobre ellas una serie de modificaciones y comprobaciones. En primer lugar, se ajustan los tipos de datos de las propiedades a los tipos soportados por Flutter, haciendo uso de la biblioteca `geojson_vi` para añadir soporte a los datos geográficos. En segundo lugar, se establecen los tipos de datos para las propiedades múltiples como una lista y para las propiedades de tipo enumerado. En tercer lugar, se añaden las propiedades a la variable que almacena las propiedades del constructor (definida previamente).

Dentro de este bucle se define el código (desde la línea 39 a la línea 45) para construir cada propiedad y sus métodos `getter` y `setter`. Este código está fuera de la anotación ya que es parte del código generado. El resultado para cada propiedad de la entidad es del estilo:

```

1 EntityClass _entityName;
2
3 EntityClass get entityName => _entityName;
4
5 set entityName(EntityClass entityName) {
6     _entityName = entityName;
7 }
```

Una vez definidas las propiedades, se establece el código para generar los constructores, que puede verse en el siguiente fragmento:

```

1 /*%=_ normalize(context.name, true) %*/.empty();
2
3 /*%=_ normalize(context.name, true) %*/(
4     /*%=_ constructor_props %*/
5 );
```

Para permitir la conversión de las entidades a formato JSON y viceversa fue preciso implementar dos métodos. El primero de ellos crea una instancia de la entidad a partir de un objeto JSON y su código se muestra a continuación:

```

1 /*%=_ normalize(context.name, true) %*/.fromJson(Map<String, dynamic> json):
2 /*% context.properties.forEach(function(prop, index, arr) {
```

## CAPÍTULO 7. IMPLEMENTACIÓN Y PRUEBAS

---

```
3  var propertyClass = prop.class.split(' ')[0];
4  var geographic = isGeographic(prop);
5  if (propertyClass == 'Boolean') propertyClass = 'bool';
6  if (propertyClass == 'Integer') propertyClass = 'int';
7  if (propertyClass == 'Long') propertyClass = 'int';
8  if (propertyClass == 'Float') propertyClass = 'double';
9  if (propertyClass == 'Double') propertyClass = 'double';
10 if (propertyClass == 'Text') propertyClass = 'String';
11 if (propertyClass == 'Date') propertyClass = 'DateTime';
12 if (propertyClass == 'DateTime') propertyClass = 'DateTime';
13 if (propertyClass == 'Point') propertyClass = 'GeoJSONPoint';
14 if (propertyClass == 'MultiPoint') propertyClass = 'GeoJSONMultiPoint';
15 if (propertyClass == 'LineString') propertyClass = 'GeoJSONLineString';
16 if (propertyClass == 'MultiLineString') propertyClass = 'GeoJSONMultiLineString';
17 if (propertyClass == 'Polygon') propertyClass = 'GeoJSONPolygon';
18 if (propertyClass == 'MultiPolygon') propertyClass = 'GeoJSONMultiPolygon';
19 if (propertyClass == 'Geometry') propertyClass = 'GeoJSONGeometryCollection';
20
21 var propertyIsEntity =
22   data.dataModel.entities
23     .map(function(entity) { return entity.name; })
24     .indexOf(propertyClass) != -1;
25 var propertyIsEnum =
26   data.dataModel.enums
27     .map(function(en) { return en.name; })
28     .indexOf(propertyClass) != -1;
29
30 if(arr[index + 1]) { var end = ","; } else { var end = ";" } /*/
31
32 /*% if (geographic) { */
33 /*%=_ + normalize(prop.name) */ = json[/*%= normalize(prop.name) */] == null ?
34   null : /*%=_ normalize(propertyClass, true) */.fromMap(json[/*%=
35   normalize(prop.name) */])/*%=_ end */
36 /*% } else if (propertyIsEntity) {
37   if (prop.multiple) { */
38 /*%=_ + normalize(prop.name) */ = json[/*%= normalize(prop.name) */] == null ?
39   null : /*%=_ normalize(propertyClass, true) */.decodeList(json[/*%=
40   normalize(prop.name) */])/*%=_ end */
41 /*% } else if (propertyIsEnum) { */
42 /*%=_ + normalize(prop.name) */ = /*%=_ normalize(propertyClass, true)
43   /*%*/Extension.getEnum(json[/*%= normalize(prop.name) */])/*%=_ end */
44 /*% }
```

```

42  /*% } else if (prop.multiple) { %*/
43 /*%=_ + normalize(prop.name) %*/ = json['/*=% normalize(prop.name) %*/'] == null ?
44     null : List<%= normalize(propertyClass, true) %>.from(json['/*=%
45     normalize(prop.name) %*/'])/*=% end %*/
46 /*% } else if (propertyClass == 'DateTime') { %*/
47 /*%=_ + normalize(prop.name) %*/ = _formatDateTime(json['/*=% normalize(prop.name)
48     %*/'])/*=% end %*/
49 /*% } else { %*/
50 /*%=_ + normalize(prop.name) %*/ = json['/*=% normalize(prop.name) %*/']/*=% end %*/
51 /*% } %*/
52 /*% } ); %*/

```

Para este método fue necesario crear una anotación con un bucle que recorre las propiedades de la entidad y hace una serie de comprobaciones, para posteriormente fijar las conversiones a JSON adecuadas a cada tipo de dato. En este bucle se realizan nuevamente las adaptaciones de los tipos de datos, las comprobaciones de si la propiedad es entidad o enumerado, y a mayores se comprueba si la propiedad es geográfica. En base a esto se establecen las traducciones a JSON de las propiedades (desde la línea 32 a la línea 48).

En el caso de que sea una propiedad geográfica se llama al método *fromMap* del tipo de dato correspondiente. En el caso de que sea una entidad se distingue entre si es múltiple o no, llamando en el primer caso al método *decodeList* y en el segundo al método *fromJson* del tipo de entidad correspondiente. En el caso de que sea un enumerado se llama al método *getEnum* declarado en su clase, que devuelve el texto asociado al valor literal. En el caso de que sea una propiedad múltiple de tipo básico se llama al método *from* del tipo de dato *List*, que genera una lista a partir de un objeto de texto. En el caso de que sea una propiedad de tipo *DateTime* se llama a una de las funciones auxiliares definidas para convertir fechas. Por último, en todos los demás casos se asigna directamente el valor de la propiedad JSON a la propiedad de la entidad.

El método que permite transformar la entidad a formato JSON se muestra en el siguiente fragmento:

```

1 Map<String, dynamic> toJson() =>
2 {
3     /*% context.properties.forEach(function(prop, index, arr) { %*/
4     /*% */
5         var propertyClass = prop.class.split(' ')[0];
6         var geographic = isGeographic(prop);
7         var propertyIsEntity =
8             data.dataModel.entities
9                 .map(function(entity) { return entity.name; })
10                .indexOf(propertyClass) != -1;
11         var propertyIsEnum =
12             data.dataModel.enums

```

```
13         .map(function(en) { return en.name; })
14         .indexOf(propertyClass) != -1; %*/
15
16     /*% if (geographic) { %*/
17     /*%=% normalize(prop.name) %*/: /*%=_ + normalize(prop.name) %*/ == null ?
18     null : /*%=_ + normalize(prop.name) %*/.toMap,
19     /*% } else if (propertyIsEntity) { %*/
20     /*%=% normalize(prop.name) %*/: /*%=_ + normalize(prop.name) %*/ == null ?
21     null : /*%=_ + normalize(prop.name) %*/,
22     /*% } else if (propertyIsEnum) { %*/
23     /*%=% normalize(prop.name) %*/: /*%=_ + normalize(prop.name) %*/ == null ?
24     null : /*%=_ + normalize(prop.name) %*/.value,
25     /*% } else if (prop.multiple) { %*/
26     /*%=% normalize(prop.name) %*/: /*%=_ + normalize(prop.name) %*/ == null ?
27     null : /*%=_ + normalize(prop.name) %*/,
28     /*% } else if (propertyClass == 'Date') { %*/
29     /*%=% normalize(prop.name) %*/: /*%=_ + normalize(prop.name) %*/ == null ?
30     null : _dateToJson(/*%=_ + normalize(prop.name) %*/),
31     /*% } else if (propertyClass == 'DateTime') { %*/
32     /*%=% normalize(prop.name) %*/: /*%=_ + normalize(prop.name) %*/ == null ?
33     null : _dateTimeToJson(/*%=_ + normalize(prop.name) %*/),
34     /*% } else { %*/
35     /*%=% normalize(prop.name) %*/: /*%=_ + normalize(prop.name) %*/,
36     /*% } %*/
37     /*% } ); %*/
38   };
```

Como puede verse, para este método se incluye nuevamente una anotación con un bucle que recorre las propiedades de la entidad y almacena si es una propiedad geográfica, una entidad o un enumerado. En base a esto se establecen las conversiones de forma similar a como se hizo en el método *fromJson* (desde la línea 16 a la línea 30).

Por último, se añadieron una serie de funciones auxiliares para facilitar la conversión de los datos a formato JSON y viceversa.

### 7.1.2 Widgets

En esta sección se detallan los aspectos más relevantes de la implementación de los *widgets* de la aplicación, que constituyen todas las vistas y elementos de la interfaz. Todos ellos están compuestos de dos clases principales con la siguiente estructura:

```
1 class className extends StatefulWidget {
2     type argument1;
3     type argument2;
4     ...
5 }
```

```

6   className([arguments]);
7
8   @override
9   _className createState() => _className();
10 }
11
12 class _className extends State<className> {
13   @override
14   void initState() {
15     [...]
16   }
17
18   @override
19   Widget build(BuildContext context) {
20     [...]
21   }
22 }
```

La primera clase representa un *widget* con estado mutable y constituye la interfaz pública del componente, ofrecida al resto de la aplicación. Dentro de ella se definen los argumentos públicos del *widget*, su constructor (mediante el que se asigna el valor a los argumentos), y la función *createState*, que crea un estado devolviendo una instancia de *\_className*.

La segunda clase representa un estado del *widget*, y se trata de una clase privada. En ella se definen las propiedades privadas del *widget*, que constituyen su estado, y se implementan los métodos necesarios para construir la interfaz del componente, que veremos a continuación.

Por una parte, está el método *initState*, que es el encargado de inicializar el estado del *widget*. En él se establecen los valores iniciales de las variables y se llama a las funciones necesarias para configurar el *widget*.

Por otra parte, está el método *build*, en el que se construye la interfaz del *widget* mediante un conjunto de componentes estructurados en forma de árbol. La estructura de este método en la mayor parte de los *widgets* de la aplicación es la siguiente:

```

1 @override
2 Widget build(BuildContext context) {
3   return Scaffold(
4     appBar: AppBar(
5       title: Text(AppLocalizations.of(context).translate('key')),
6       actions: <Widget>[
7         IconButton(
8           icon: const Icon(Icons.<icon>),
9           onPressed: () {
10             [...]
11           },
12         ),
13       ],
14     ),
15   );
16 }
```

```
13     ],
14   ),
15   body: _getBody(),
16 );
17 }
```

Este método recibe el contexto y devuelve el *Widget* construido. Como se puede ver, el elemento raíz de la estructura es la clase *Scaffold* que implementa la estructura básica de un *layout* de Material Design. Dentro de este componente se define la barra superior (*appBar*), en la que se establece el título de la página y, en algunos casos, una serie de herramientas (*actions*); y el cuerpo del *widget* (*body*).

A continuación explicaremos los elementos más importantes del método *\_getBody()* definido en el fichero de generación de los formularios editables de la aplicación.

El primer tramo de código definido es el siguiente:

```
1 Widget _getBody() {
2   if (_loading) {
3     return Center(
4       child: Padding(
5         padding: const EdgeInsets.all(8),
6         child: CircularProgressIndicator(),
7       )
8     );
9   } else if (_error) {
10    return Column(
11      mainAxisAlignment: MainAxisAlignment.center,
12      children: <Widget>[
13        Center(
14          child: Text(
15            AppLocalizations.of(context).translate('error.loading_form'),
16            style: TextStyle(fontSize: 18),
17            ),
18          ),
19        SizedBox(height: 12),
20        RaisedButton(
21          onPressed: () {
22            updateState("refresh", true);
23          },
24          child: Text(
25            AppLocalizations.of(context).translate('button.retry'),
26            style: TextStyle(fontSize: 18)
27            ),
28          ),
29        ],
30      );
```

31 } else {

Cómo se puede ver, inicialmente se hacen una serie de comprobaciones sobre los valores de las variables definidas en el estado del *widget*. En este caso se comprueba si el valor de la variable *\_loading* es verdadero, en cuyo caso se muestra un indicador de progreso circular centrado en la pantalla. En caso contrario, se comprueba si el valor de la variable *\_error* es verdadero y, de ser así, se muestra un mensaje de error y un botón que permite reintentar la búsqueda.

El método *AppLocalizations.of(context).translate('key')* se utiliza para obtener los textos internacionalizados de la aplicación mediante la clave definida en los ficheros JSON de mensajes.

Tras estas comprobaciones se define el cuerpo del *widget* en el caso de que no haya errores y ya se haya cargado la página. Para los formularios editables, el cuerpo está compuesto por el formulario que se puede ver a continuación:

```

1 return Container(
2   child: SingleChildScrollView(
3     physics: const AlwaysScrollableScrollPhysics(),
4     padding: const EdgeInsets.all(32),
5     child: Form(
6       key: _formKey,
7       child: Column(
8         crossAxisAlignment: CrossAxisAlignment.start,
9         children: [
10           /*% context.properties.filter(function(prop) {
11             return prop.editing;
12           }).forEach(function(prop) {
13             var theProperty = getProperty(entity, prop.property);
14             var entityProperty = getEntity(data, theProperty.class);
15             theProperty.class = theProperty.class.split(' ')[0];
16             var propertyIsEnum =
17               data.dataModel.enums
18                 .map(function(en) { return en.name; })
19                 .indexOf(theProperty.class) != -1;
20             if (theProperty.class != 'Geometry') { %*/
21             Row(
22               crossAxisAlignment: CrossAxisAlignment.start,
23               children: [
24                 Expanded(
25                   child: Column(
26                     crossAxisAlignment: CrossAxisAlignment.start,
27                     children: [
28                       Text(
29                         /*%= normalize(theProperty.name, true) %*/ + ': ',

```

```
30         style: TextStyle(
31             fontWeight: FontWeight.bold,
32             fontSize: 16
33         ),
34     ),
35     SizedBox(height: 5),
```

La estructura del árbol está formada por un contenedor y un componente que añade el soporte para el *scroll*. Como hijo de este componente se define el formulario, que contiene una clave y una serie de campos de entrada estructurados en filas dentro de una columna.

Para recorrer las propiedades se utiliza el bucle que se puede ver en el código, dentro del cual se establecen una serie de variables para cada propiedad que almacenan la propiedad, su entidad asociada, su clase y si es enumerado.

Dentro del bucle, para cada propiedad editable del formulario se define una fila que contiene el nombre de la propiedad, dentro de un texto y una columna; y a continuación, una fila con el campo de entrada del tipo de dato correspondiente.

Para las propiedades que representan relaciones entre entidades se utiliza el componente *SmartSelect*, que construye un selector. Para las relaciones “a uno” se utilizan selectores simples y para las relaciones “a muchos”, selectores múltiples. En el siguiente fragmento se muestra el código implementado para los selectores múltiples:

```
1 /*% if (entityProperty){
2     if (theProperty.owner){
3         if (theProperty.multiple){
4             var aForm = data.forms.find(function(form) { return form.entity ==
entityProperty.name; });
5             if (aForm) { %*/
6             TextFormField(
7                 builder: (FormFieldState<int> state) {
8                     return SmartSelect.multiple(
9                         title: /*%= normalize(theProperty.name, true) %*/,
10                        value: /*%= normalize(theProperty.name) %*/SelectedList,
11                        onChange: (state) => setState(() => /*%= normalize(theProperty.name)
%*/SelectedList = state.value),
12                        choiceItems: /*%= normalize(theProperty.name) %*/ChoiceItems,
13                        modalConfirm: true,
14                        modalFilter: true,
15                        placeholder:
AppLocalizations.of(context).translate('placeholder.multiple_select'),
16                    );
17                },
18                validator: (value) {
19                    /*% if (theProperty.required){ %*/
20                    if (_/*%= normalize(theProperty.name) %*/SelectedList.isEmpty) {
```

```

21     return AppLocalizations.of(context).translate('validator.required');
22 }
23 /*% } %*/
24 return null;
25 },
26 onSaveed: (int value) {
27     _entity./% normalize(theProperty.name) %/ = _/=%
28     normalize(theProperty.name) %/SelectedList != null
29     ? _formController.find/*% normalize(theProperty.name, true) %/(_/=%
30     normalize(theProperty.name) %/SelectedList)
31     : null;
32 },
33 /*% } %*/
34 /*% } else {

```

En este caso, se comprueba que la propiedad es entidad, que la entidad del formulario es el lado propietario en la relación y que es un formulario múltiple; y a continuación se define un campo del formulario (*FormField*) y en su constructor se configura el selector. Asimismo, se definen las validaciones sobre el valor de entrada (*validator*) y las acciones que se deben realizar cuando se guarde el contenido del formulario (*onSaved*), que consisten en asignar el valor de entrada a la propiedad de la entidad del formulario.

Para las propiedades de tipo *DateTime* se utilizan selectores de fecha y hora. El componente empleado ha sido desarrollado *ad-hoc* para la aplicación por la autora del proyecto. En el siguiente fragmento se puede ver el código para el campo de entrada este tipo de dato:

```

1 /*% if (theProperty.class == 'DateTime'){ %*/
2     TextFormField(
3         builder: (FormFieldState<int> state) {
4             return DateTimePicker(
5                 type: DateTimePickerType.dateTimeSeparate,
6                 dateFormat: 'dd/MM/yyyy',
7                 firstDate: DateTime(1900),
8                 lastDate: DateTime(2100),
9                 initialValue: _entity./% normalize(theProperty.name) %/,
10                dateLabelText: AppLocalizations.of(context).translate('label.date'),
11                timeLabelText: AppLocalizations.of(context).translate('label.time'),
12                onChanged: (value) => {
13                    if (_checkFormatDate(value) == "DATE"){
14                        _dateTime = DateFormat("dd/MM/yyyy").parse(value),
15                        setState(() => {
16                            _entity./% normalize(theProperty.name) %/ = _entity./%=
17                            normalize(theProperty.name) %/ != null
18                            ? new DateTime(_dateTime.year, _dateTime.month, _dateTime.day,
19                            _entity./% normalize(theProperty.name) %/.hour, _entity./%=

```

```

normalize(theProperty.name) %*/.minute)
    : _dateTime
})
} else if (_checkFormatDate(value) == "TIME") {
    _dateTime = DateFormat("HH:mm").parse(value),
    setState(() => {
        _entity./**= normalize(theProperty.name) %*/ = _entity./**=
normalize(theProperty.name) %*/ != null
            ? new DateTime(_entity./**= normalize(theProperty.name) %*/.year,
            _entity./**= normalize(theProperty.name) %*/.month, _entity./**=
normalize(theProperty.name) %*/.day, _dateTime.hour, _dateTime.minute)
            : _dateTime
})
} else {
    setState(() => _entity./**= normalize(theProperty.name) %*/ = null),
}
},
),
),
),
/*% } %*/

```

Como se puede ver, se configuran los parámetros del componente y las acciones a ejecutar cuando se modifica el valor de la entrada. Para los datos de tipo *Date* se emplea el mismo componente, indicando que el tipo es solo de fecha.

Para los enumerados se usan desplegables, cuyo código es el siguiente:

```

/*% if (propertyIsEnum){ %*/
Row(children: [
    Expanded(
        flex: 9,
        child: DropdownButtonFormField(
            items: _/**= normalize(theProperty.name) %*/ListItems,
            value: _entity./**= normalize(theProperty.name) %*/,
            onChanged: (value) {
                setState(() => _entity./**= normalize(theProperty.name) %*/ = value);
            },
            onSaved: (value) {
                setState(() => _entity./**= normalize(theProperty.name) %*/ = value);
            },
            decoration: InputDecoration(
                hintText: '/**= normalize(theProperty.name, true) %*/',
            ),
        ),
    ),
    Expanded(

```

```

20         flex: 1,
21         child: IconButton(
22             icon: Icon(Icons.clear),
23             onPressed: () {
24                 setState(() => this._entity/*=% normalize(theProperty.name) %*/ = null);
25             },
26         ),
27     ),
28 ],
29 /*% } %*/

```

En el caso de los tipos de datos booleanos, se utilizan componentes *switch* cuyo valor se asocia a la propiedad de la entidad, como se puede ver en el siguiente fragmento de código:

```

1 /*% if (theProperty.class == 'Boolean') { %*/
2     Switch(
3         value: (_entity != null && _entity/*=% normalize(theProperty.name) %*/ != null)
4             ? _entity/*=% normalize(theProperty.name) %*/ : false,
5         onChanged: (value) {
6             setState(() {
7                 _entity/*=% normalize(theProperty.name) %*/ = value;
8             });
9         },
10         activeTrackColor: Colors.lightGreenAccent,
11         activeColor: Colors.green,
12     ),
13 /*% } %*/

```

En cuanto a las propiedades textuales se emplean campos de texto cuya implementación se muestra a continuación:

```

1 /*% if (theProperty.class == 'String' || theProperty.class == 'Text') { %*/
2     TextFormField(
3         initialValue: _entity/*=% normalize(theProperty.name) %*/ != null ?
4             _entity/*=% normalize(theProperty.name) %*/.toString() : '',
5         validator: (value) {
6             /*% if (theProperty.required){ %*/
7                 if (value.isEmpty) {
8                     return AppLocalizations.of(context).translate('validator.required');
9                 }
10            /*% } else if (theProperty.min != null) { %*/
11                if (value.isNotEmpty && value.length < /*=% theProperty.min %*/ ) {
12                    return AppLocalizations.of(context).translate('validator.longer') + ' /*=%
13                         theProperty.min %*/';
14                }
15            /*% } else if (theProperty.max != null) { %*/
16                if (value.isNotEmpty && value.length > /*=% theProperty.max %*/ ) {

```

## CAPÍTULO 7. IMPLEMENTACIÓN Y PRUEBAS

---

```
15         return AppLocalizations.of(context).translate('validator.shorter') + ' /*= theProperty.max */';
16     }
17 /*% } %*/
18 /*% if (theProperty.patternType == 'emailPattern'){ */
19     if (value.isNotEmpty && !value.isEmail()){
20         return AppLocalizations.of(context).translate('validator.format.email');
21     }
22 /*% } %*/
23 /*% if (theProperty.patternType == 'urlPattern'){ */
24     if (value.isNotEmpty && !value.isUrl()){
25         return AppLocalizations.of(context).translate('validator.format.url');
26     }
27 /*% } %*/
28 /*% if (theProperty.patternType == 'ipPattern'){ */
29     if (value.isNotEmpty && !value.isIPV4()){
30         return AppLocalizations.of(context).translate('validator.format.ip');
31     }
32 /*% } %*/
33 /*% if (theProperty.patternType == 'customPattern'){ */
34     if (value.isNotEmpty){
35         RegExp exp = new RegExp(r'/*= theProperty.pattern */');
36         if (exp.hasMatch(value)){
37             return AppLocalizations.of(context).translate('validator.format.custom');
38         }
39     }
40 /*% } %*/
41     return null;
42 },
43 /*% if (theProperty.patternType == 'emailPattern'){ */
44     keyboardType: TextInputType.emailAddress,
45 /*% } %*/
46 /*% if (theProperty.patternType == 'urlPattern'){ */
47     keyboardType: TextInputType.url,
48 /*% } %*/
49 /*% if (theProperty.patternType == 'ipPattern'){ */
50     keyboardType: TextInputType.number,
51 /*% } %*/
52 /*% if (theProperty.class == 'Text'){ */
53     keyboardType: TextInputType.multiline,
54 /*% } %*/
55     onSaved: (String value){
56         _entity./*= normalize(theProperty.name) */ = value.isEmpty ? null : value;
57     },
58 ),
59 /*% } %*/
```

Como se puede ver en el fragmento, se establece el valor del campo de entrada como el de la propiedad correspondiente de la entidad y en el validador se establecen todas las restricciones posibles. Para este tipo de datos, existen restricciones de obligatoriedad, de longitud mínima y máxima, y de ajuste a un patrón. Además, dependiendo de estas restricciones, se establece el tipo de teclado que se abrirá al poner el foco en el campo de entrada.

Para los campos de tipo numérico se emplean también campos de entrada de texto, pero en su caso, el teclado es siempre numérico y se establecen distintas restricciones, comprobando la obligatoriedad, los valores mínimos y máximos, y si el formato es de número entero o decimal.

La última parte del formulario es una fila que contiene los botones de cancelación y guardado de los datos. El código de esta fila puede verse en el siguiente fragmento:

```

1 Row(
2   mainAxisAlignment: MainAxisAlignment.start,
3   mainAxisSize: MainAxisSize.end,
4   children: [
5     RaisedButton(
6       onPressed: () {
7         Navigator.pop(context);
8       },
9       child: Text(
10         AppLocalizations.of(context).translate('button.cancel'),
11         style: TextStyle(fontSize: 18)
12       ),
13       color: Colors.red,
14       textColor: Colors.white,
15       shape: RoundedRectangleBorder(
16         borderRadius: BorderRadius.circular(6.0),
17       ),
18     ),
19     SizedBox(width: 15),
20     RaisedButton(
21       onPressed: () {
22         if (_formKey.currentState.validate()) {
23           _formKey.currentState.save();
24           _saveEntity();
25         }
26       },
27       child: Text(
28         AppLocalizations.of(context).translate('button.save'),
29         style: TextStyle(fontSize: 18)
30       ),
31       color: Colors.green,
32       textColor: Colors.white,
33       shape: RoundedRectangleBorder(

```

```
34     borderRadius: BorderRadius.circular(6.0),  
35     ),  
36     ),  
37   ]  
38 )
```

El botón de cancelación navega a la página anterior. El botón de guardado inicialmente valida el contenido de los campos del formulario, a continuación, guarda sus datos (ejecutando el método *onSaved* de cada campo de entrada) y, por último, ejecuta el método *\_saveEntity* que llama a la función del controlador encargada de guardar la entidad.

Dentro del fichero de generación de los formularios editables se definen también un conjunto de funciones encargadas de comunicarse con el controlador y de actualizar el estado del *widget*.

### 7.1.3 Mapa

En esta sección se detallan los aspectos más relevantes de la implementación del mapa de la aplicación. Para este elemento, se ha desarrollado un *widget* personalizado haciendo uso de la biblioteca “*flutter\_map*”.

Este *widget* recibe una serie de parámetros, y a partir de ellos dibuja los elementos correspondientes en el mapa si aplica.

El estado (*state*) del *widget* está compuesto por un conjunto de variables entre las que se pueden destacar: *\_defaultBounds* que contiene las coordenadas de los límites por defecto en los que se enfoca el mapa; *\_mapController* que es el controlador del mapa; *\_userLocationOptions* que es un componente para configurar y soportar la geolocalización del usuario; las listas *\_markers*, *\_lines* y *\_polygons* que contienen los marcadores, las líneas y los polígonos del mapa respectivamente; y la variable *\_bounds* que almacena los límites de enfoque del mapa con todos los elementos.

En el método *initState()* de este *widget* se inicializan algunas variables del estado, así como los componentes del mapa mediante la ejecución del método *\_initMap()*, cuyo código se muestra a continuación:

```
1 void _initMap(){  
2   /*% if (feature.GUI_L_ViewListAsMap) { %*/  
3   if (widget._multiple && widget._geometries != null){  
4     int i = 0;  
5     List<LatLng> _pointsList = new List<LatLng>();  
6     widget._geometries.asMap().forEach((index, geometry) => {  
7       _buildGeoJsonCollection(geometry, _pointsList, _colors.elementAt(i)/% if  
(feature.GUI_L_ViewListAsMap) { %/, index /*% } %/),  
8       if (i < _colors.length){  
9         i++,
```

```

10         } else {
11             i = 0,
12         }
13     });
14 } else {
15 /* } */
16 if (widget._coordinates != null) {
17     _buildGeoJsonObject(
18         widget._type, widget._coordinates, new List<LatLng>(), null/* if
(feature.GUI_L_ViewListAsMap) { /*/, null, null /* } */);
19 } else if (widget._geometries != null) {
20     _buildGeoJsonCollection(widget._geometries, new List<LatLng>(), null/* if
(feature.GUI_L_ViewListAsMap) { /*/, null /* } */);
21 }
22 /* if (feature.GUI_L_ViewListAsMap) { /*/
23 }
24 /* } */
25 }

```

Como se puede ver, dentro del método se diferencian los casos en los que se deben representar elementos de varias entidades y los que no. Para dibujar los elementos en el mapa se ejecutan los métodos `_buildGeoJsonCollection` para las colecciones de elementos y `_buildGeoJsonObject` para los elementos simples. A continuación veremos la implementación de ambos métodos.

El siguiente fragmento muestra la implementación del método `_buildGeoJsonCollection`:

```

1 void _buildGeoJsonCollection(List<dynamic> geometries, List<LatLng> pointsList, Color
color/*% if (feature.GUI_L_ViewListAsMap) { /*/, int collectionIndex /* } */)
async {
2     if (widget._type != null && geometries != null && geometries.isNotEmpty) {
3         geometries.asMap().forEach((index, element) =>
4             _buildGeoJsonObject(element.type, element.coordinates, pointsList, color/*%
if (feature.GUI_L_ViewListAsMap) { /*/, collectionIndex, index /* } */));
5     }
6 }

```

Como se puede ver, a la hora de dibujar las colecciones de elementos geográficos, lo único que se hace es llamar al método `_buildGeoJsonObject` para cada elemento de la colección. La implementación de este método es la siguiente:

```

1 void _buildGeoJsonObject(GeometryType type, List<dynamic> coordinates, List<LatLng>
pointsList,
2     Color color/*% if (feature.GUI_L_ViewListAsMap) { /*/, int collectionIndex, int
index /* } */) async {
3     if (type != null && coordinates != null) {
4         switch (type) {

```

```
5      case GeometryType.point:
6          _markers.add(
7              _buildMarker(coordinates, pointsList, color != null ? color :
Colors.black/*% if (feature.GUI_L_ViewListAsMap) { /*/, collectionIndex, index
/*% } /*/),
8          );
9          break;
10     case GeometryType.multiPoint:
11         (coordinates)
12             .forEach((pair) => _markers.add(_buildMarker(pair, pointsList, color != null ? color : Colors.black/*% if (feature.GUI_L_ViewListAsMap) { /*/, collectionIndex, index /*% } /*/)));
13         break;
14     case GeometryType.lineString:
15         _lines.add(_buildLine(coordinates, pointsList, color != null ? color :
Colors.red/*% if (feature.GUI_L_ViewListAsMap) { /*/, collectionIndex, index /*% } /*/));
16         break;
17     case GeometryType.multiLineString:
18         (coordinates)
19             .forEach((line) => _lines.add(_buildLine(line, pointsList, color != null
? color : Colors.red/*% if (feature.GUI_L_ViewListAsMap) { /*/, collectionIndex,
index /*% } /*/)));
20         break;
21     case GeometryType.polygon:
22         _polygons.add(_buildPolygon(coordinates, pointsList, color != null ? color :
Colors.red/*% if (feature.GUI_L_ViewListAsMap) { /*/, collectionIndex, index /*% } /*/));
23         break;
24     case GeometryType.multiPolygon:
25         (coordinates).forEach(
26             (polygon) => _polygons.add(_buildPolygon(polygon, pointsList, color
!= null ? color : Colors.red/*% if (feature.GUI_L_ViewListAsMap) { /*/, collectionIndex,
index /*% } /*/)));
27         break;
28     }
29     _bounds = LatLngBounds.fromPoints(pointsList);
30 }
31 }
```

Esta función recibe una variable que indica el tipo del elemento geográfico, una lista de coordenadas (que puede ser una lista anidada en el caso de elementos como líneas o polígonos), una lista de puntos que contiene todos los puntos de los elementos representados en el mapa y que se utiliza para definir los límites de enfoque, el color y unos índices en el caso de representar varias entidades.

Dentro de ella se distingue entre los posibles tipos de datos geográficos y para cada uno de ellos se ejecuta el método correspondiente. En el caso de los tipos de datos múltiples (*multiPoint*, *multiLineString* y *multiPolygon*) se ejecuta el método correspondiente sobre cada elemento de la lista. Los resultados obtenidos de la ejecución de estos métodos se añaden a las listas de marcadores, líneas y polígonos del mapa según corresponda. Además, se va añadiendo a la variable *\_bounds* los nuevos límites teniendo en cuenta los elementos añadidos.

Para dibujar los marcadores se ha implementado el método *\_buildMarker*, cuyo código es el que se muestra a continuación:

```

1 Marker _buildMarker(List<double> coordinates, List<LatLng> pointsList, Color color/*%
2     if (feature.GUI_L_ViewListAsMap) { /*/, collectionIndex, index /*% } /*%/) {
3         Marker marker = new Marker(
4             width: 80.0,
5             height: 80.0,
6             point: new LatLng(coordinates.last, coordinates.first),
7             /*% if (feature.GUI_L_ViewListAsMap) { /*/
8             builder: (ctx) => new GestureDetector(
9                 child: Icon(
10                     Icons.location_on,
11                     size: 35,
12                     color: color,
13                     ),
14                     onTap: () {
15                         if (widget._multiple) {
16                             widget._showDetails(widget._geometries[collectionIndex][index],
17                             GeoJSONPoint(coordinates));
18                         }
19                     },
20                     builder: (ctx) => new Icon(
21                         Icons.location_on,
22                         size: 35,
23                         color: color,
24                         ),
25                     /*% } /**/
26 );
27         pointsList.add(new LatLng(coordinates.last, coordinates.first));
28         return marker;
29 }
```

Este método recibe las coordenadas del punto, la lista global de puntos del mapa, el color y los índices si aplica. Dentro de él se configura un objeto *Marker* que representa un marcador del mapa, estableciendo su ícono, color, coordenadas y, en el caso de representar múltiples en-

tidades (como en la funcionalidad de ver la lista como un mapa), el comportamiento cuando se pulsa sobre él, que consiste en mostrar una ventana modal con la información correspondiente. Las coordenadas del punto se añaden a la lista de puntos global y se devuelve el marcador.

Para dibujar las líneas se ha implementado el método `_buildLine`, cuyo código se puede ver a continuación:

```
1 Polyl ine _buildLine(List<dynamic> coordinates, List<LatLng> pointsList, Color color/*%
2     if (feature.GUI_L_ViewListAsMap) { /*/, collectionIndex, index /*% } /*% /) {
3         List<LatLng> linePoints = new List<LatLng>();
4         coordinates
5             .forEach((pair) => linePoints.add(new LatLng(pair.last, pair.first)));
6         pointsList.addAll(linePoints);
7         /*% if (feature.GUI_L_ViewListAsMap) { /*/
8         if (widget._multiple && linePoints.length > 1) {
9             LatLng point = linePoints[linePoints.length ~/ 2];
10            _markers.add(
11                _buildMarker([point.longitude, point.latitude], pointsList, color != null ?
12                    color : Colors.black/*% if (feature.GUI_L_ViewListAsMap) { /*/, collectionIndex,
13                    index /*% } /*% /));
14            }
15        /*% } /*/
16        return new Polyl ine(points: linePoints, color: color, strokeWidth: 4.0);
17    }
```

Esta función recibe la lista de coordenadas, la lista global de puntos del mapa, el color y los índices si aplica. Dentro de ella se define una lista a la que se van añadiendo todos los pares de coordenadas que representan un punto y posteriormente se construye un objeto *Polyline* a partir de esa lista de puntos. Además, en el caso de que se representen múltiples entidades, se añade un punto aproximadamente en la mitad de la línea, que se utiliza para mostrar la ventana modal de información al pulsar sobre él, ya que las líneas no soportan detección de gestos.

Para dibujar los polígonos se ha implementado el método `_buildPolygon`, cuyo código se puede ver a continuación:

```
1 Polygon _buildPolygon(List<dynamic> coordinates, List<LatLng> pointsList, Color
2     color/*% if (feature.GUI_L_ViewListAsMap) { /*/, collectionIndex, index /*% } /*% /) {
3         List<LatLng> polygonPoints = new List<LatLng>();
4         coordinates.forEach((line) => line.forEach(
5             (pair) => polygonPoints.add(new LatLng(pair.last, pair.first))));
6         pointsList.addAll(polygonPoints);
7         Polygon polygon = new Polygon(
8             points: polygonPoints,
9             color: color.withOpacity(0.2),
10            borderColor: color,
```

```

10    borderStrokeWidth: 4.0,
11  );
12 /*% if (feature.GUI_L_ViewListAsMap) { %*/
13 if (widget._multiple) {
14   double latitudes = 0.0;
15   double longitudes = 0.0;
16   polygon.points.forEach((point) {
17     latitudes += point.latitude;
18     longitudes += point.longitude;
19   });
20   LatLng point = new LatLng(latitudes / polygon.points.length,
21     longitudes / polygon.points.length);
22   _markers.add(
23     _buildMarker([point.longitude, point.latitude], pointsList,
24       color != null ? color : Colors
25         .black /*% if (feature.GUI_L_ViewListAsMap) { %*/,
26         collectionIndex, index /*% } %*/));
27 }
28 /*% } %*/
29 return polygon;
30 }

```

Este método recibe la lista de coordenadas, la lista global de puntos del mapa, el color y los índices si aplica. Dentro de él se define una lista a la que se van añadiendo todas las líneas y posteriormente, se construye un objeto *Polygon* a partir de esa lista de puntos. Además, en el caso de que se representen múltiples entidades, se añade un punto aproximadamente en el centro del polígono, que se utiliza para mostrar la ventana modal de información al pulsar sobre él, ya que los polígonos no soportan detección de gestos.

Por último, en cuanto a la construcción de la interfaz del *widget* en el método *build*, las partes más destacadas del código se muestran en el siguiente fragmento:

```

1 Widget build(BuildContext context) {
2   _userLocationOptions = UserLocationOptions(
3     context: context,
4     mapController: _mapController,
5     markers: _markers,
6     zoomToCurrentLocationOnLoad: (_markers.isEmpty && _lines.isEmpty &&
7       _polygons.isEmpty),
8     updateMapLocationOnPositionChange: (_markers.isEmpty &&
9       _lines.isEmpty &&
10      _polygons.isEmpty &&
11      !widget._interactive),
12     showMoveToCurrentLocationFloatingActionButton: widget._interactive,
13   );
14   return Scaffold(

```

```
14     body: SafeArea(
15         child: Stack(children: <Widget>[
16             FlutterMap(
17                 mapController: _mapController,
18                 options: MapOptions(
19                     bounds: _bounds,
20                     boundsOptions: FitBoundsOptions(padding: EdgeInsets.all(40.0)),
21                     interactive: widget._interactive,
22                     plugins: [
23                         UserLocationPlugin(),
24                     ],
25                     onTap: (LatLng point) {
26                         [...]
27                     },
28                     layers: [
29                         new TileLayerOptions(
30                             urlTemplate:
31                             "https://s.tile.openstreetmap.org/{z}/{x}/{y}.png",
32                             subdomains: ['a', 'b', 'c']),
33                         _userLocationOptions,
34                         new PolylineLayerOptions(polylines: _lines),
35                         new PolygonLayerOptions(polygons: _polygons),
36                         new MarkerLayerOptions(markers: _markers)
37                     ],
38                 ),
39             ),
40         ],
41     ),
42 
```

Como se puede ver, inicialmente se configura el componente *UserLocationOptions* indicando, por ejemplo, cuando se debe mostrar el botón flotante para centrar la vista del mapa en la posición del usuario o cuando se debe centrar dicha vista de forma automática. A continuación, se define la estructura del *widget*, en la que el elemento principal es el componente *FlutterMap*, para el que se configuran tres elementos principales: el controlador del mapa, las opciones y las capas. Dentro de las opciones se establecen los límites de enfoque, si es un mapa interactivo, los *plugins* y las acciones a realizar cuando se pulsa en alguna parte del mapa. En cuanto a las capas, se define una lista formada por: la capa que muestra el fondo del mapa a partir de la plantilla indicada, la capa que soporta la ubicación del usuario, la capa de las líneas, la capa de los polígonos y la capa de los marcadores.

## 7.2 Pruebas

En esta sección se detallan las pruebas realizadas sobre la aplicación para verificar su correcto funcionamiento.

Para la realización de las pruebas se asumió que las funcionalidades ofrecidas por la API

del servidor funcionaban correctamente y que sus resultados eran correctos.

Las pruebas realizadas consistieron en pruebas manuales de integración y aceptación, para asegurar la correcta comunicación de la aplicación con el servidor web y para verificar la adecuación de la implementación a los requisitos definidos para el producto.

Durante la ejecución de este tipo de pruebas se ejercitaron todas las posibles interacciones de un usuario con la interfaz de la aplicación y se comprobó que su comportamiento era el adecuado.

Por una parte, se verificó que las peticiones realizadas al servidor y recibidas de este eran correctas y que los datos obtenidos se traducían correctamente a las entidades de la aplicación mediante la depuración del código.

Por otra parte, se forzaron errores para comprobar que se gestionaban correctamente y que se proporcionaba el *feedback* adecuado al usuario mediante la utilización de alertas, mensajes temporales, etc.

Asimismo, se verificó que toda la información de la interfaz se mostrase correctamente formateada, que los campos de entrada se ajustasen a los tipos de datos de las propiedades (con selectores de fechas y horas, desplegables, campos de texto, campos numéricos, casillas de verificación, etc.) y que se realizarasen todas las validaciones necesarias sobre la información de entrada, mostrando los mensajes correspondientes en el caso de que los datos no se ajustasen a las restricciones.

En cuanto a los mapas, se comprobó que los elementos geográficos se presentasen correctamente y que al modificar este tipo de propiedades se dibujasen de forma adecuada. También se verificó el funcionamiento del sistema de geolocalización del usuario y la correcta redirección a Google Maps con la información geográfica de los puntos a la hora de obtener una ruta.

Por último, con las pruebas se aseguró que la navegación entre las pantallas fuese correcta y que los datos enviados a través de las rutas se recibiesen correctamente. Asimismo, se verificó que la información del usuario autenticado se almacenase correctamente en el dispositivo y que se gestionase de forma adecuada en la aplicación. También se comprobó la correcta internacionalización de los mensajes de la aplicación mediante la prueba de esta en los dos idiomas soportados.

## Capítulo 8

# Construcción de la línea de producto software

---

En este capítulo se detallan los aspectos más relevantes de la arquitectura e implementación de la línea de producto software, así como la integración de la aplicación desarrollada en ella.

## 8.1 Arquitectura

En esta sección se detalla la arquitectura de la línea de producto software mediante el estudio de los componentes que la conforman. Esta arquitectura está reflejada en la figura 8.1.

Como se puede ver, la línea de producto software está compuesta de cuatro componentes principales, que son los siguientes:

- **Interfaz web:** es el componente que permite al usuario seleccionar las características y definir el modelo de datos del producto que se generará. Proporciona además otras funcionalidades como importar o exportar la especificación de un producto, modificar aspectos del proyecto asociado, parametrizar el proyecto, etc. Una breve explicación del funcionamiento de la interfaz se muestra en la sección 4.1.
- **Especificación del producto:** es el elemento que contiene la configuración del producto a generar. En este fichero se definen las características seleccionadas para el producto y los componentes que forman parte del modelo de datos. En el apartado 8.1.1 se detalla la estructura de este fichero.
- **spl-js-engine:** es el motor de derivación encargado de generar el código de los productos a partir de los componentes y de la especificación definida para el producto. En el apartado 8.1.1 se detalla el funcionamiento de este componente.

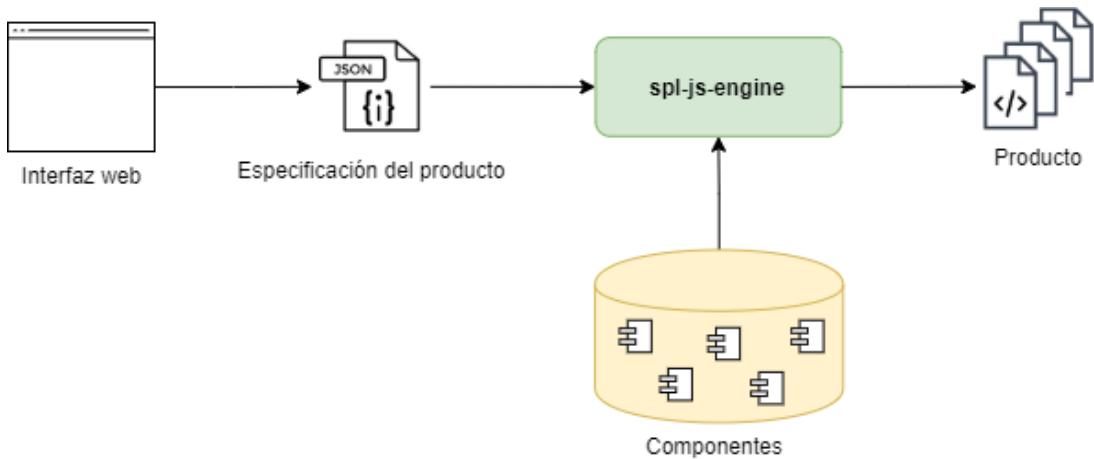


Figura 8.1: Arquitectura de la línea de producto software

- **Componentes:** son los elementos reutilizables que implementan las distintas funcionalidades y características definidas para el producto. Los aspectos más relevantes de la implementación de estos componentes pueden verse en el capítulo 7.

### 8.1.1 Motor de derivación

Para la generación del código fuente de los productos de la LPS, la plataforma GEMA utiliza la herramienta *spl-js-engine*, un motor de derivación JavaScript basado en *scaffolding*. Esta herramienta hace uso de anotaciones sobre el código original del producto (plantillas) para definir los fragmentos que pertenecen a cada componente, en lugar de separar físicamente el código de cada uno. Esto supone una ventaja sobre los enfoques compositivos ya que simplifica el proceso de desarrollo de la LPS. Las anotaciones en las plantillas se indican mediante comentarios del lenguaje de programación de la plantilla y su contenido es cualquier código JavaScript. Estos comentarios son eliminados del código fuente generado [12].

En la figura 8.2 puede verse un esquema de los elementos de entrada que recibe el motor de derivación para generar un producto, y que se describen a continuación.

- **model.xml:** fichero en formato XML que contiene el modelo de características de la línea de producto software. Para este proyecto se utilizó el modelo de características definido para la plataforma GEMA. Para la creación y modificación de este archivo se puede emplear FeatureIDE [47], un *plugin* de Eclipse que permite definir modelos de características en formato XML y generar su representación gráfica (figura 4.5).
- **config.json:** fichero en formato JSON que contiene la configuración del motor de derivación. En él se definen los delimitadores de las anotaciones en función de la extensión

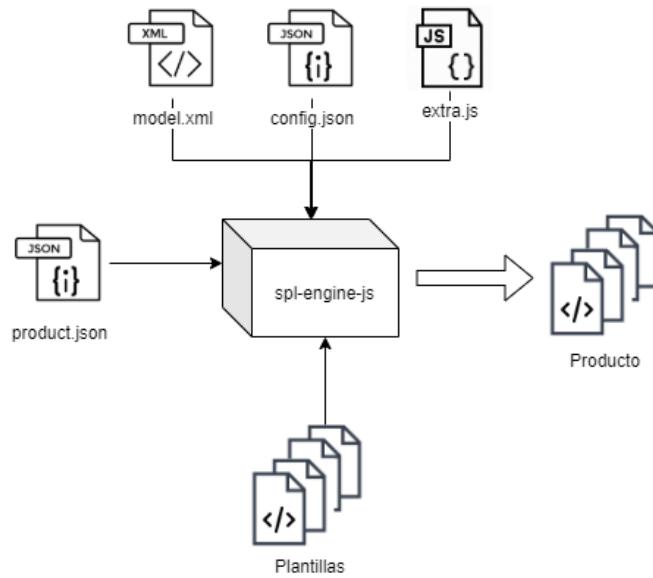


Figura 8.2: Esquema del motor de derivación *spl-js-engine*

de los archivos, así como las carpetas que se deben ignorar a la hora de generar los productos.

- **extra.js**: fichero que contiene código JavaScript que se puede emplear al derivar un producto. En él se definen funciones que se pueden incluir en el código JavaScript de las anotaciones y que se ejecutan la hora de generar productos.
- **product.json**: fichero que contiene la especificación de un producto en formato JSON. En el subapartado 8.1.1 se detalla la estructura de un fichero de especificación para un producto.
- **Plantillas**: código fuente anotado, es decir, el código fuente desarrollado para el producto con las anotaciones correspondientes (capítulo 7).

### Product.json

En la figura 8.3 se muestra la estructura del fichero de especificación de un producto. Esta estructura está formada por las siguientes propiedades:

- **features**: lista con las funcionalidades seleccionadas para el producto. En la figura 8.4 se puede ver un ejemplo del contenido de esta propiedad.
- **data**: propiedad con los datos del producto, incluida la información del modelo de datos y la información de configuración. Esta propiedad está subdividida en varias propiedades que se describen a continuación.

```

    <!
    > "features": [...],
    <!
    > "data": {
    <!
    >   "basicData": {...},
    <!
    >   "dataModel": {
    <!
    >     "enums": [...],
    <!
    >     "entities": [...]
    <!
    >   },
    <!
    >   "forms": [...],
    <!
    >   "gui": {...},
    <!
    >   "lists": [...],
    <!
    >   "menus": [...],
    <!
    >   "maps": [],
    <!
    >   "statics": []
    <!
    > }
  }
}

```

Figura 8.3: Estructura del fichero de especificación de un producto

- **basicData**: contiene datos de configuración, como el nombre del proyecto. En la figura 8.5 se puede ver un ejemplo del contenido de esta propiedad.
- **dataModel**: contiene la información de los enumerados y las entidades definidas en el modelo de datos.
  - \* **enums**: contiene la lista de los enumerados definidos para el producto. En la figura 8.6 se muestra un ejemplo de la definición de un enumerado. Como se puede ver, para cada enumerado se define un nombre y una lista de valores.
  - \* **entities**: contiene la lista de entidades definidas en el modelo de datos del producto. En la figura 8.7 se muestra un ejemplo de la definición de una entidad. Como se puede ver, para cada entidad se define un nombre, una cadena a mostrar (*displayString*) y una lista de propiedades. Para cada propiedad se establece un nombre, una clase (que representa el tipo de dato de la propiedad) y una serie de restricciones si aplica, por ejemplo, si es clave primaria, si es una propiedad obligatoria, si es única, si tiene restricciones de valores mínimos y máximos o si se ajusta a algún patrón. Además, las propiedades que representan relaciones entre entidades almacenan una serie de datos de la relación, en concreto: si es la entidad propietaria, si la relación es bidireccional y la multiplicidad.

```

"features": [
    "GEMA_SPL",
    "DataManagement",
    "GraphicalUserInterface",
    "MapView",
    "Tools",
    "DM_SpatialDatabase",
    "DM_DataServer",
    "MV_MapServer",
    "MV_Tools",
    "MV_MapCenter",
    "MV_MapManagement",
    "DM_SD_PostGIS",
    "MV_T_Pan",
    "MV_T_Zoom",
    "GUI_Menu",
    "GUI_M_Left",
    "GUI_Lists",
    "GUI_Forms",
    "GUI_F_Creatable",
    "GUI_F_Removable",
    "GUI_F_Editable",
    "GUI_F_R_ConfirmationAlert",
    "GUI_L_Sortable",
    "GUI_L_LocateInMap",
    "GUI_L_FormLink",
    "GUI_M_Position",
    "GUI_L_ViewListAsMap",
    "MV_MM_UniqueMapView",
    "MV_MC_BBox",
    "GUI_L_Filterable",
    "GUI_L_F_RowFilter",
    "MV_GeoJSON",
    "MWMSSupport",
    "MWM_VisitSchedule",
    "MWM_M_VisitConstraint",
    "MWM_M_Employee",
    "MWM_M_PlannedVisit",
    "MWM_M_Client",
    "MWM_VisitType",
    "MVM_VT_WithTime",
    "MWM_M_Activity",
    "MWM_PersonnelManager",
    "MWM_PM_Employees",
    "MWM_PM_Clients",
    "T_GIS",
    "MWM_Models"
],

```

Figura 8.4: Ejemplo de la propiedad *features* en el fichero de especificación de un producto

```

"basicData": {
    "index": {
        "component": "STATIC",
        "view": "welcome"
    },
    "languages": [
        "es",
        "en"
    ],
    "name": "TFMRaquel",
    "packageInfo": {
        "artifact": "tfmraquel",
        "group": "es.udc.lbd.gisspl"
    },
    "database": {},
    "extra": {},
    "SRID": "4326",
    "version": "0.2.0"
},

```

Figura 8.5: Ejemplo de la propiedad *basicData* en el fichero de especificación de un producto

```

"enums": [
  {
    "name": "Tipo_incidencia",
    "values": [
      "AUSENCIA",
      "AVERIA",
      "MATERIAL",
      "DATOS_ERRONEOS",
      "OTRO"
    ]
  },

```

Figura 8.6: Ejemplo de la propiedad *enums* en el fichero de especificación de un producto

```

"entities": [
  {
    "name": "Incidencia",
    "properties": [
      {
        "name": "id",
        "class": "Long (autoinc)",
        "pk": true,
        "required": true,
        "unique": true
      },
      {
        "name": "descripción",
        "class": "String"
      },
      {
        "name": "fecha",
        "class": "DateTime",
        "patternType": null,
        "pattern": null,
        "min": null,
        "max": null
      },
      {
        "name": "asunto",
        "class": "String"
      },
      {
        "name": "tipo",
        "class": "Tipo_incidencia",
        "patternType": null,
        "pattern": null,
        "min": null,
        "max": null
      },
      {
        "name": "localizacion",
        "class": "Point",
        "patternType": null,
        "pattern": null,
        "min": null,
        "max": null
      }
    ],
    "relationships": [
      {
        "name": "área",
        "class": "MultiPolygon",
        "patternType": null,
        "pattern": null,
        "min": null,
        "max": null
      },
      {
        "name": "solicitudes",
        "class": "Solicitud",
        "owner": false,
        "bidirectional": "incidencia",
        "multiple": true,
        "required": false
      },
      {
        "name": "contactos",
        "class": "Contacto",
        "owner": true,
        "bidirectional": "incidencias",
        "multiple": true,
        "required": false
      },
      {
        "name": "recursos",
        "class": "Recurso",
        "owner": true,
        "bidirectional": "incidencia",
        "multiple": false,
        "required": false
      }
    ],
    "displayString": "$id"
  }
],

```

Figura 8.7: Ejemplo de la propiedad *entities* en el fichero de especificación de un producto

```
"forms": [
  {
    "id": "Incidencia Form",
    "properties": [
      {
        "property": "id",
        "viewing": true,
        "editing": false
      },
      {
        "property": "descripción",
        "viewing": true,
        "editing": true
      },
      {
        "property": "fecha",
        "viewing": true,
        "editing": true
      },
      {
        "property": "asunto",
        "viewing": true,
        "editing": true
      },
      {
        "property": "tipo",
        "viewing": true,
        "editing": true
      }
    ],
    "entity": "Incidencia",
    "creatable": true,
    "editable": true,
    "removable": true,
    "confirmation": true
  }
],
```

Figura 8.8: Ejemplo de la propiedad *forms* en el fichero de especificación de un producto

- **forms:** contiene la lista de formularios definidos para el producto. En la figura 8.8 se muestra un ejemplo de la definición de un formulario. Como se puede ver, para cada formulario se almacena un identificador, la entidad a la que está asociado, si permite creación, si permite edición, si permite borrado, si muestra confirmación de borrado y una lista de propiedades. Estas propiedades se corresponden con las propiedades de la entidad asociada y para cada una de ellas se almacena el nombre de la propiedad, si es visible y si es editable.
- **gui:** contiene elementos de configuración del aspecto del cliente web como el tipo de fuente o el conjunto de colores. Esta parte de configuración no se ha empleado en este proyecto.
- **lists:** contiene el conjunto de listas definidas para el producto. En la figura 8.9 se muestra un ejemplo de la definición de una lista. Como se puede ver, para cada lista se almacena un identificador, la entidad a la que está asociado, el formulario al que está asociada, si permite borrado, si permite ordenación, si permite búsqueda, si permite filtrado por fila y un conjunto de propiedades. Estas propiedades se corresponden con las propiedades de la entidad asociada y para cada una de ellas

```

"lists": [
{
  "id": "Incidencia List",
  "properties": [
    {
      "property": "id"
    },
    {
      "property": "descripción"
    },
    {
      "property": "fecha"
    },
    {
      "property": "asunto"
    },
    {
      "property": "tipo"
    },
    {
      "property": "contactos",
      "form": "Contacto Form"
    },
    {
      "property": "materiales",
      "form": "Recurso Form"
    }
  ],
  "entity": "Incidencia",
  "form": "Incidencia Form",
  "removeLink": true,
  "sorting": true,
  "searching": false,
  "filtering": true
},

```

Figura 8.9: Ejemplo de la propiedad *lists* en el fichero de especificación de un producto

se almacena el nombre de la propiedad.

- **menus:** contiene la lista de menús definidos para la aplicación con sus entradas.  
Esta parte de la configuración no se ha empleado en este proyecto.
- **maps:** contiene elementos de la configuración de los mapas del producto. Esta parte de la configuración no se ha empleado en este proyecto.
- **statics:** contiene elementos de configuración de las páginas estáticas del producto.  
Esta parte de la configuración no se ha empleado en este proyecto.

## 8.2 Implementación

El aspecto más relevante en la implementación de la línea de producto software es la anotación del código desarrollado para el producto.

```
    "delimiters": [
      {
        "extension": ["html", "jsp", "xml"],
        "start": "<!--%",
        "end": "%-->"
      },
      {
        "extension": [
          "css",
          "js",
          "java",
          "vue",
          "g4",
          "tokens",
          "gradle",
          "md",
          "conf",
          "gitignore",
          "properties",
          "sql",
          "dart"
        ],
        "start": "/*%",
        "end": "%*/"
      },
      {
        "extension": ["py"],
        "start": "#%",
        "end": "%#"
      },
      {
        "extension": ["json"],
        "start": "###<",
        "end": ">###"
      }
    ]
```

Figura 8.10: Configuración de los delimitadores de las anotaciones

Como se ha explicado en la sección 8.1, las anotaciones empleadas por el motor de derivación se incluyen en el código fuente del producto como comentarios del lenguaje de programación de cada plantilla y su contenido es código JavaScript.

A la hora de anotar el código fuente del producto, es necesario definir los delimitadores de los comentarios para cada tipo de plantilla en el archivo de configuración (*config.json*). Dicha configuración puede verse en la figura 8.10.

Esta configuración permite establecer para cada extensión de archivo un delimitador diferente, posibilitando de esta forma que las anotaciones se integren como comentarios del lenguaje empleado y no afecten a la hora de compilar el código. En el caso concreto de este proyecto, debido a la estructura del lenguaje .dart y a la dependencia del modelo de datos definido, no es posible añadir las anotaciones sin causar errores en el código de las plantillas que impiden su compilación. Además, también se generan errores de compilación en ficheros que no admiten comentarios, como los archivos JSON que contienen los mensajes internacionalizados.

El derivador de código soporta distintos tipos de anotaciones que se explican a continua-

ción.

Por una parte, están las **anotaciones normales**, que simplemente incluyen código JavaScript. Este tipo de anotaciones se utilizan para incluir comprobaciones, bucles, llamadas a funciones, etc. Algunos ejemplos de este tipo de anotaciones son los siguientes:

```
1 /*% if (feature.GUI_F_Editable){ %*/
2
3     /*% context.properties.filter(function(prop) {
4         return prop.editing;
5     }).forEach(function(prop) {
6         var theProperty = getProperty(entity, prop.property);
7         var entityProperty = getEntity(data, theProperty.class);
8         theProperty.class = theProperty.class.split(' ')[0];
9         var propertyIsEnum =
10             data.dataModel.enums
11                 .map(function(en) { return en.name; })
12                 .indexOf(theProperty.class) != -1;
13
14     %*/
```

Por otra parte, están las **anotaciones de interpolación**. Este tipo de anotaciones permite incluir el valor de una variable del código JavaScript en el código fuente del producto generado, pudiendo incluir llamadas a funciones. Se diferencian de las otras anotaciones por el carácter “=”. Un ejemplo de este tipo de anotaciones es el siguiente:

```
1 /*%= normalize(entity.name, true) %*/ entity;
```

Por último, están las **anotaciones de generación de ficheros**, que sirven para generar nuevos archivos a partir de la especificación. A diferencia de los otros dos tipos, estas anotaciones tienen que estar ubicadas siempre al principio de los ficheros y tienen que contener la implementación de una función que reciba una especificación (*data* y *feature*) y devuelva un array de objetos con dos propiedades: *fileName* y *context*. Estas anotaciones se diferencian de las demás por el uso del carácter “@”. Un ejemplo de este tipo de anotaciones es el siguiente:

```
1 /*@ return data.forms.map(function(form) {  
2   return {  
3     fileName: normalize(form.entity) + '.dart',  
4     context: form  
5   };  
6 }) */
```

El resultado de aplicar la anotación del ejemplo es un fichero por cada formulario del array `data.forms`. Cada fichero generado tendrá el nombre (`fileName`) de la entidad asociada al formulario junto a la extensión de archivo “.dart”, y el contenido (`context`) será el formulario.

El *context* definido en este tipo de anotaciones puede ser empleado en el resto de las anotaciones normales empleadas en el fichero, pudiendo acceder a sus propiedades, por ejemplo:

```
1 /*=% context.name %*/;
```

## 8.3 Integración de la aplicación

El proceso de integración de la aplicación desarrollada en la línea de producto software se basó principalmente en la introducción de las anotaciones en el código fuente para definir las plantillas.

Una vez desarrollado y anotado el código, sólo fue necesario introducir la carpeta del proyecto en el directorio correspondiente de la herramienta de generación para poder generar los productos.

### 8.3.1 Pruebas

A la hora de probar la correcta integración de la aplicación desarrollada en la línea de producto software, se realizaron numerosas pruebas manuales.

Con el fin de verificar que las anotaciones definidas para las plantillas eran correctas, se generaron múltiples productos, pero, debido al gran número de elementos del modelo de características diseñado, no fue factible probar todas las combinaciones posibles de características. Sin embargo, se intentó probar aquellas más relevantes de forma que se verificase la corrección de todas las anotaciones incluidas en el código.

Asimismo, para comprobar la correcta adecuación del producto a los modelos de datos definidos en la LPS, se generaron múltiples productos con diferentes modelos de datos, probando todas las posibles relaciones entre entidades y todos los tipos de datos soportados.



## Capítulo 9

# Solución desarrollada

---

En este capítulo se muestran las características más relevantes de la aplicación desarrollada a modo de visita guiada por las principales funcionalidades. La aplicación de ejemplo aquí expuesta muestra todas las características posibles de un producto de la LPS.

### 9.1 Aplicación móvil

La pantalla principal de la aplicación se puede ver en la figura 9.1a y en ella se muestra el mensaje de bienvenida con el nombre del proyecto. Desplegando el menú, en el caso de que no haya ningún usuario autenticado, se muestra el contenido de la figura 9.1b. Como se puede ver, en el encabezado se sitúa el botón de inicio de sesión, y en el cuerpo del menú aparece un único desplegable que permite seleccionar el idioma de la aplicación.

Pulsando en el botón de inicio de sesión se accede a la pantalla que se muestra en la figura 9.2a, en la que existen dos campos de texto que permiten introducir el nombre de usuario y la contraseña de la cuenta. En el caso de querer recuperar la contraseña, es necesario pulsar en el enlace “¿Olvidó su contraseña?” que redirige a la pantalla de la figura 9.2b, en la que simplemente se introduce la dirección de correo electrónico.

Una vez que existe un usuario autenticado en la aplicación, el contenido del menú es el que se muestra en la figura 9.3. Para cada entidad del modelo de datos definido existe una entrada que permite navegar a su listado, además de las tres entradas que permiten acceder a los elementos de planificación.

Al acceder a uno de los listados, si el producto generado ha incluido la característica de búsqueda simple se mostrará la pantalla que se puede ver en la figura 9.4a. En el caso de que se haya seleccionado la funcionalidad de filtrado por campos se accederá a la pantalla de la figura 9.4b, cuya opción de filtrar redirige a la vista que se muestra en la figura 9.4c. En esta pantalla existe un campo de entrada por cada propiedad de la entidad que permite filtrado, ajustando el formato al tipo de dato correspondiente. Al pulsar en el botón de filtrar, se regresa

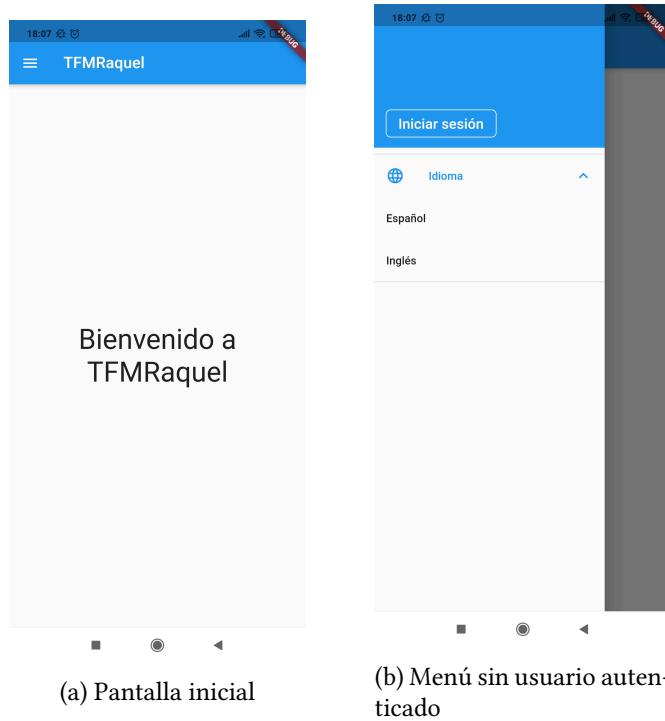


Figura 9.1: Pantalla inicial y menú sin usuario autenticado

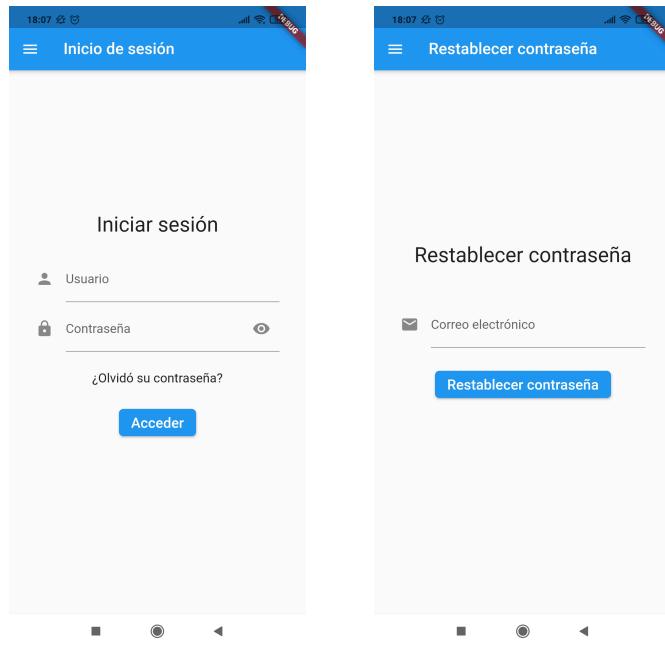


Figura 9.2: Pantallas de inicio de sesión y de recuperación de contraseña

## CAPÍTULO 9. SOLUCIÓN DESARROLLADA

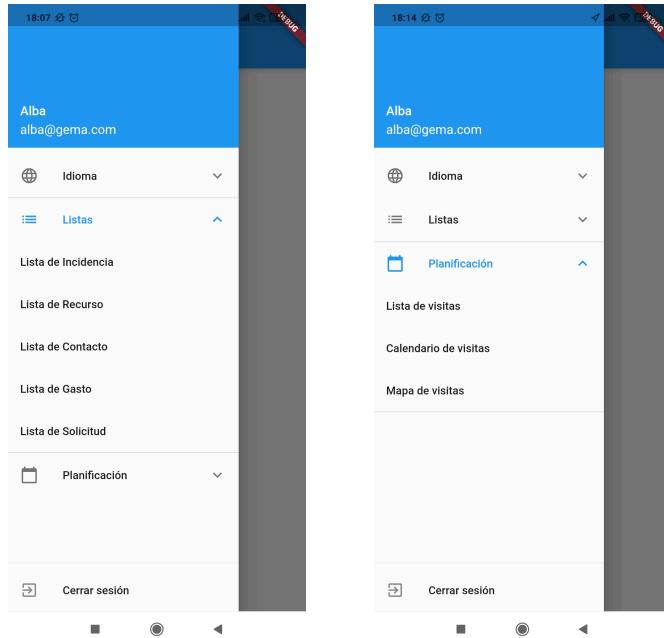
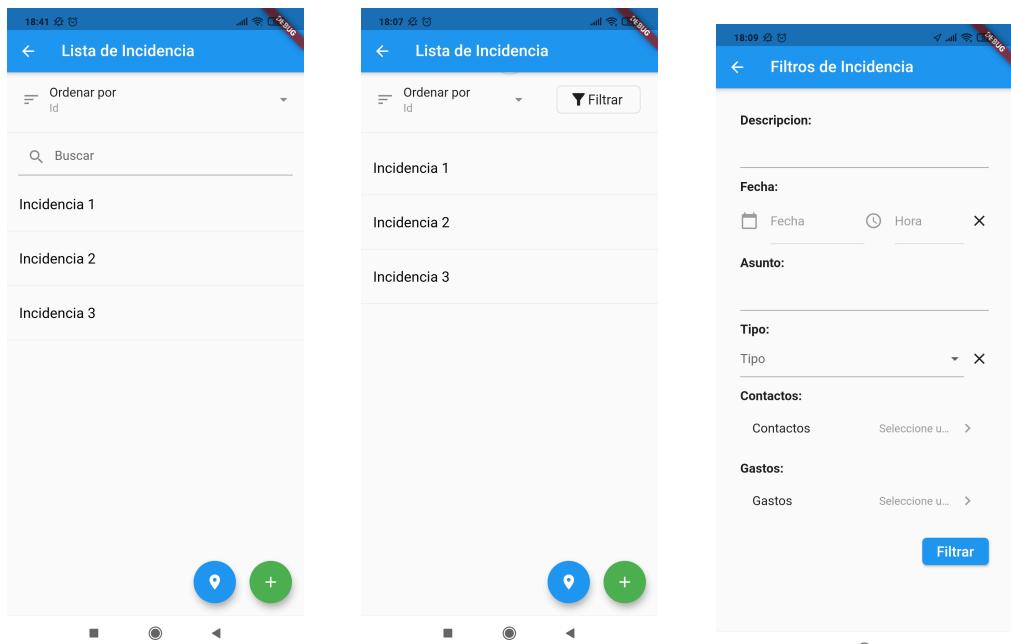


Figura 9.3: Menú con usuario autenticado

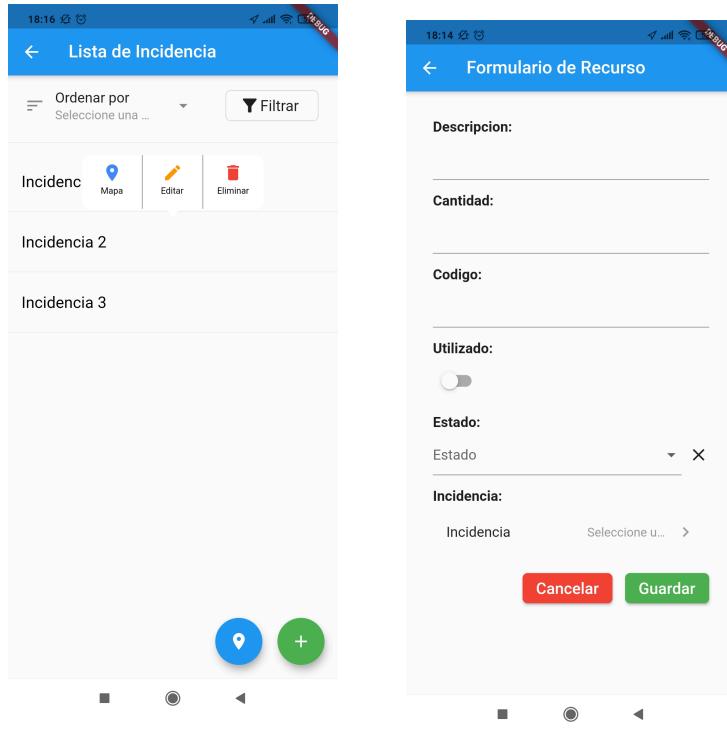


(a) Listado con búsqueda simple

(b) Listado con filtros por propiedades

(c) Filtros de una lista

Figura 9.4: Pantallas de listados y filtros



(a) Listado con *popup* en un elemento

(b) Formulario de creación

Figura 9.5: Pantallas de listado con *popup* y de formulario de creación

a la pantalla de la lista, que ya muestra los resultados obtenidos tras haber aplicado los filtros.

Manteniendo una pulsación larga sobre una entrada de la lista, se muestra un menú *popup* con una serie de atajos. Este menú se puede ver en la figura 9.5a y consta de tres entradas. La primera permite acceder a la posición del elemento en el mapa, la segunda permite acceder a la modificación del elemento y la última permite eliminar la instancia. Explicaremos un poco más adelante todas estas vistas.

Siguiendo en la vista de la lista, existen dos botones flotantes en la parte inferior derecha. El botón coloreado en verde permite acceder al formulario de creación de una nueva entidad. En la figura 9.5b se muestra un ejemplo de este tipo de formularios, que están compuestos por una serie de campos de entrada que se corresponden con las propiedades de la entidad. Cada campo de entrada se ajusta al tipo de dato de la propiedad, proporcionando así entradas de texto, selectores de fechas y horas, desplegables, mapas, etc. Además, sobre todos los valores se establecen las restricciones definidas en el modelo de datos del producto, por ejemplo: obligatoriedad, tamaño mínimo y máximo, restricciones de patrones, etc.

El segundo botón, coloreado en azul, permite ver los elementos de la lista reflejados en un mapa, como se muestra en la figura 9.6a, en la que las propiedades geográficas de cada entidad aparecen señaladas con un color diferente. Al pulsar en alguno de estos elementos

## CAPÍTULO 9. SOLUCIÓN DESARROLLADA

---

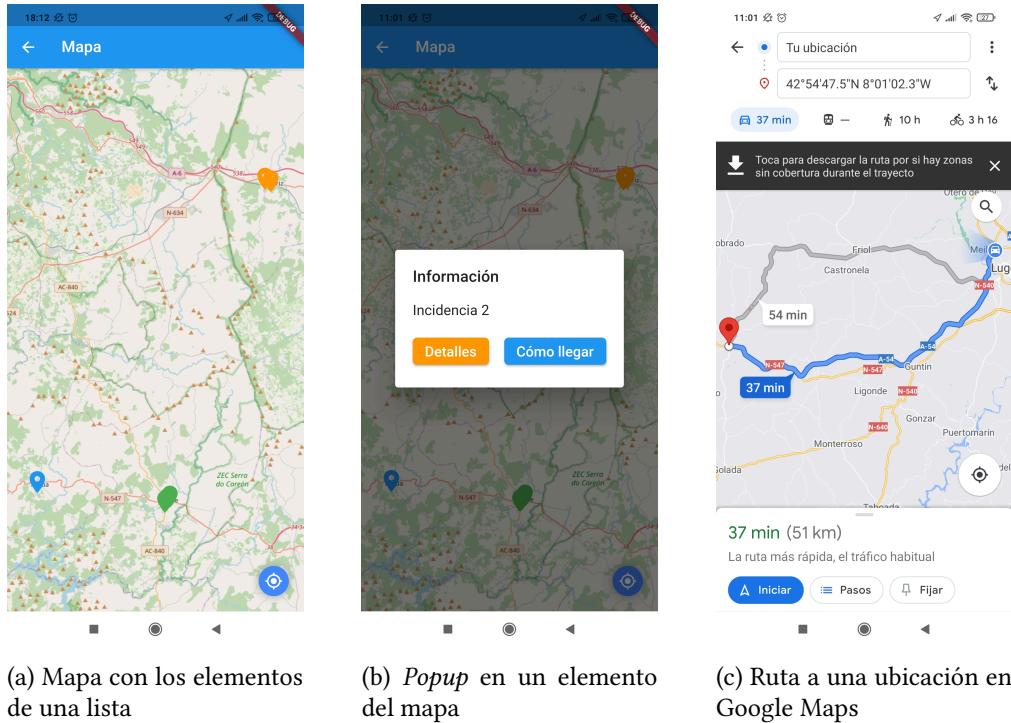


Figura 9.6: Pantalla inicial y menú sin usuario autenticado

aparece una ventana modal con la entrada de la lista a la que está asociada y la posibilidad de acceder a sus detalles o de obtener la ruta hasta su ubicación (figura 9.6b). La opción de obtener la ruta redirige a la aplicación de Google Maps con los datos de la ubicación, como se muestra en la figura 9.6c. En todos los mapas de la aplicación se incluye el botón flotante que permite centrar la posición del usuario.

Desde el listado también es posible acceder al detalle de cada entidad pulsando sobre su entrada en la lista, acción que redirige a la pantalla de la figura 9.7a. Como se puede ver, esta vista está formada por una serie de campos que se corresponden con las propiedades de la entidad asociada, y para cada uno de ellos se muestra su valor en el formato adecuado. En el caso de las propiedades que representan una relación entre entidades, se muestra un enlace a la entidad correspondiente, y en el caso de las propiedades geográficas se muestra una miniatura de un mapa que se puede ampliar pulsando sobre él. Más adelante veremos la vista de mapa.

La pantalla de detalle permite eliminar la entidad, al igual que el atajo del menú *popup* de las listas, mostrando previamente la alerta de confirmación de borrado que se muestra en la figura 9.7b.

Asimismo, permite acceder al formulario de edición de la entidad, al igual que el atajo del menú *popup* de las listas. Dicha pantalla de modificación puede verse en la figura 9.7c.

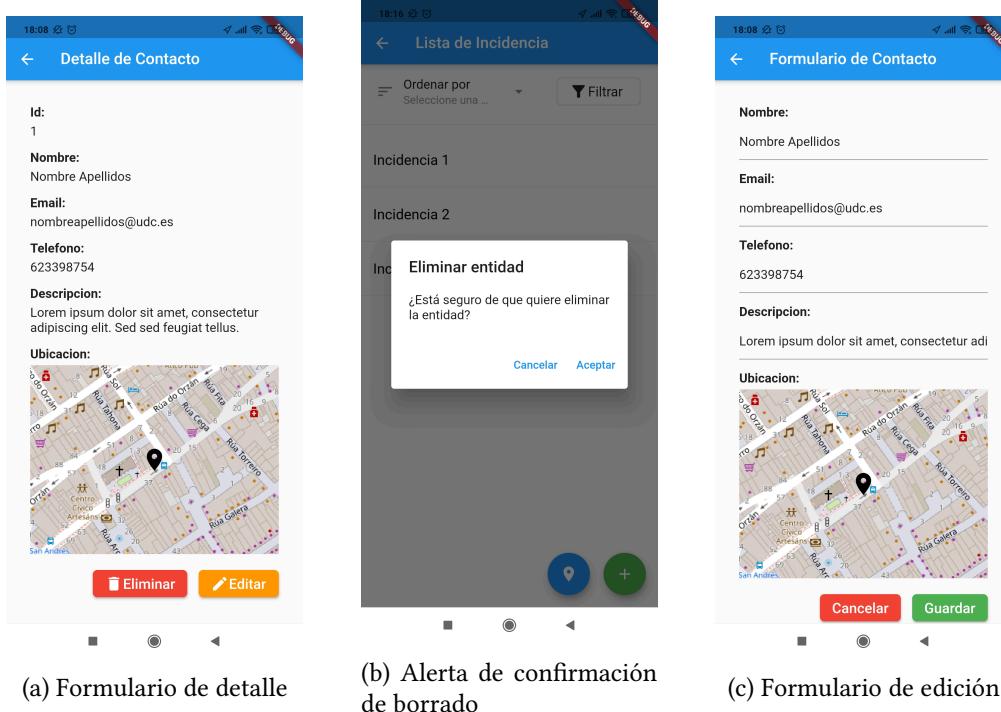


Figura 9.7: Formularios de detalle y de edición, y alerta de confirmación de borrado

Esta vista consiste en un formulario formado por un conjunto de campos asociados a las propiedades de la entidad, igual que el formulario de creación explicado previamente, pero en este caso mostrando los valores guardados para cada propiedad.

Las propiedades geográficas se editan en un mapa como el que se puede ver en la figura 9.8a. Este mapa es el mismo que el que se utiliza a la hora de mostrar una propiedad geográfica, pero en el caso de que sea posible la edición se incluye el botón flotante de color naranja que se puede ver en la esquina inferior derecha. Al pulsar en él se despliegan los botones que aparecen en la figura 9.8b para el caso de propiedades simples, y los botones que se muestran en la figura 9.8c para el caso de las propiedades múltiples. En ambos casos se muestra un botón que permite guardar los cambios, un botón que permite eliminar los cambios y un botón que permite salir del modo de edición. A mayores, en las propiedades múltiples se añade un botón, el que se sitúa en la parte superior, que permite añadir un nuevo elemento. En el caso de intentar finalizar la edición sin haber guardado los cambios se muestra una alerta.

En cuanto a los elementos de planificación de la aplicación, como se ha mencionado previamente, existen tres entradas en el menú que permiten acceder a cada uno de ellos. El primer elemento es el listado de visitas, que se puede ver en la figura 9.9a y cuya estructura es muy similar a la de los listados de entidades. Sin embargo, en este caso, para cada entrada de la lista se muestran varias propiedades de la visita, en concreto, su descripción, su fecha y horas de

## CAPÍTULO 9. SOLUCIÓN DESARROLLADA

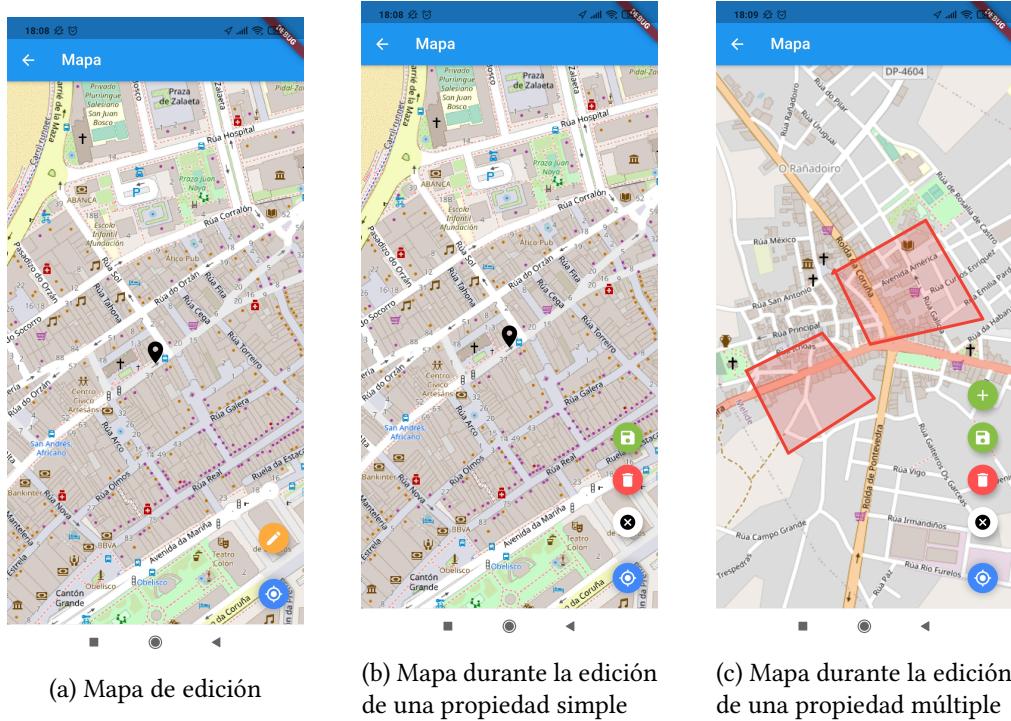


Figura 9.8: Vistas del mapa de edición

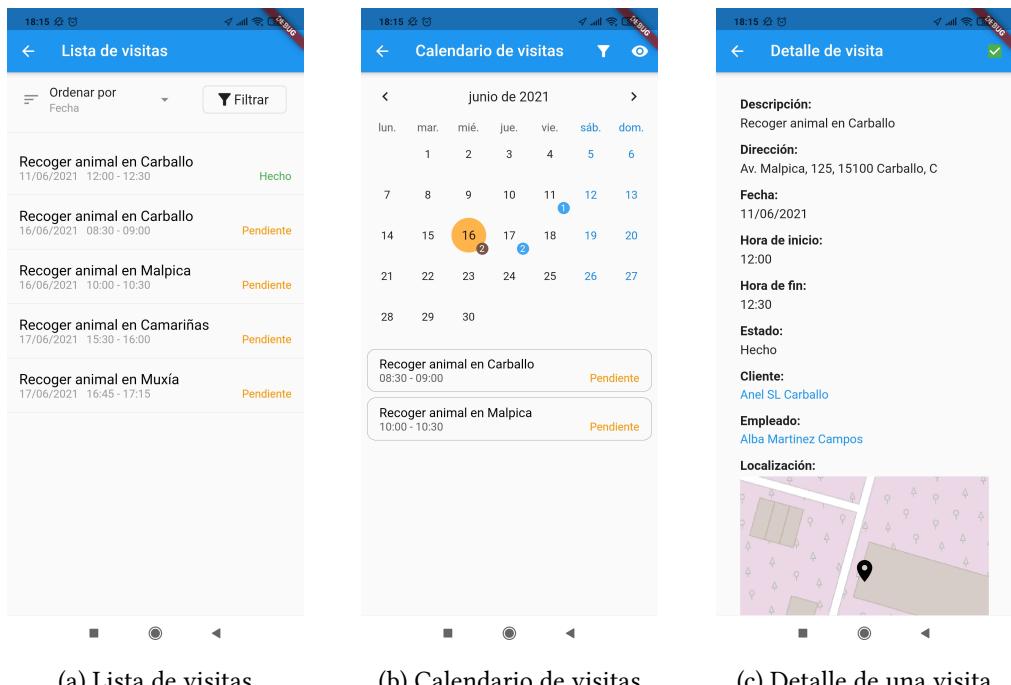


Figura 9.9: Pantallas de lista, calendario y detalle de las visitas

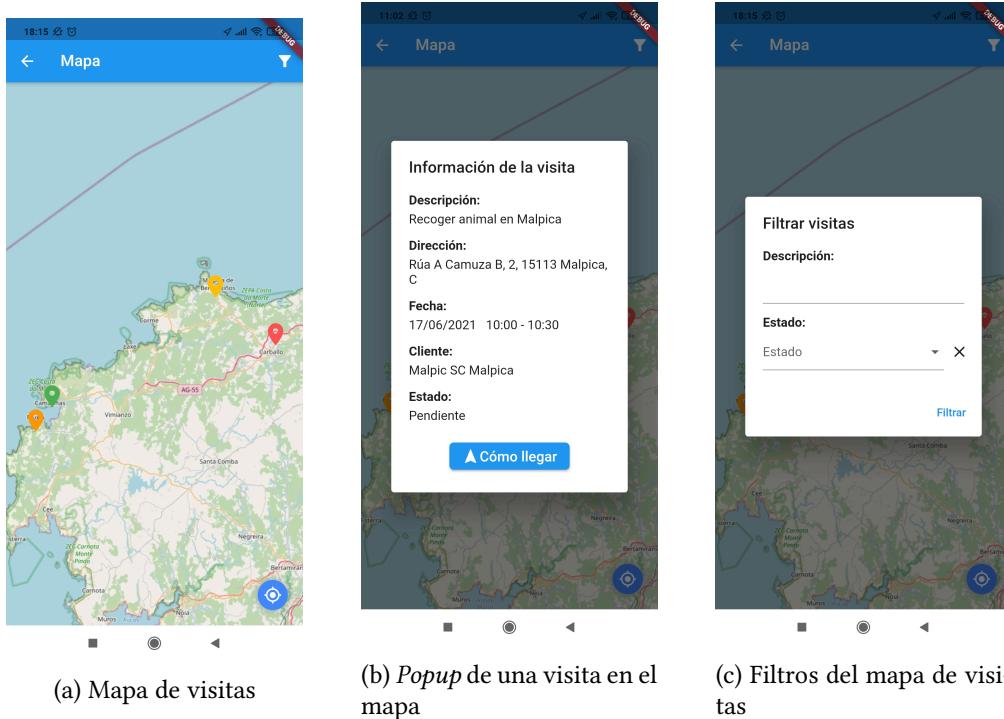


Figura 9.10: Vistas del mapa de visitas

inicio y fin, y su estado. Este listado también permite ordenar las entradas por una propiedad y filtrar sus elementos. El campo de ordenación se selecciona con un selector y la pantalla de filtros tiene la misma estructura que las mostradas previamente para las listas de entidades.

La segunda vista de planificación es el calendario de visitas, cuya pantalla se puede ver en la figura 9.9b. En esta vista se muestra un calendario en el que se marca para cada día el número de visitas que tiene asociadas. Al seleccionar un día en el calendario, se muestra debajo de él la lista de visitas asociadas a ese día, indicando para cada una de ellas su descripción, su fecha y horas de inicio y fin, y su estado. Esta pantalla incluye además una funcionalidad que permite cambiar la vista del calendario (en la esquina superior derecha), soportando la vista mensual, semanal y de dos semanas; y una funcionalidad de filtrado que permite filtrar las visitas por descripción y estado.

Tanto las entradas de la lista de visitas como las entradas de la lista del calendario permiten acceder al detalle de la visita correspondiente al pulsar en ellas. La pantalla de detalle de una visita se puede ver en la figura 9.9c y su estructura es prácticamente igual a la vista de detalle de una entidad. La única diferencia es que se incluye una casilla de verificación en la esquina superior derecha que permite marcar la visita como realizada sin necesidad de acceder a su formulario de edición, al que también se puede navegar desde esta pantalla y cuya estructura es igual a la de los formularios de las entidades.

Desde el detalle de la visita es posible navegar a las pantallas de detalle del cliente y del empleado a través de los enlaces correspondientes.

El último elemento de planificación es el mapa de visitas, cuya pantalla se muestra en la figura 9.10a, y en el que se representa cada visita con elementos de colores diferentes. Al pulsar sobre alguno de los elementos del mapa se muestra una ventana modal (figura 9.10b) con la información más relevante de la visita y un botón que permite obtener la ruta hasta su ubicación redirigiendo a Google Maps, igual que en el caso de las propiedades geográficas de las entidades. La vista de mapa también permite aplicar filtros sobre las visitas que se muestran, en concreto, filtrando por su descripción y por su estado, como se puede ver en la figura 9.10c.

## 9.2 Aplicación multiplataforma

Como se ha establecido al principio de este trabajo, uno de los objetivos para el proyecto era que la aplicación desarrollada fuese multiplataforma. Esta fue una de las principales razones por las que se seleccionó la tecnología Flutter para su desarrollo, ya que permite crear aplicaciones compatibles con plataformas móviles, web, de escritorio y embebidas [15].

Debido a las dificultades que ofrecen algunas de estas plataformas a la hora de probar las aplicaciones, principalmente relacionadas con la necesidad de disponer de dispositivos concretos (como los sistemas embebidos o plataformas móviles de Apple), el producto desarrollado se ha probado para plataformas web, además de en la plataforma móvil Android empleada durante el desarrollo.

Partiendo del proyecto de Flutter desarrollado, solo es necesario ejecutar los siguientes comandos para desplegar la aplicación en un entorno web:

```
1 flutter create .
2 flutter run -d chrome
```

En las figuras 9.11, 9.12, 9.13, 9.14 y 9.15 se muestran algunas capturas de pantalla del producto desplegado en el navegador Google Chrome.

Durante esta prueba, se detectaron unos pocos errores debido a que algunos de los componentes utilizados en la aplicación aún no tienen soporte para aplicaciones web y el desarrollo del producto se hizo orientado a plataformas móviles.

## 9.2. Aplicación multiplataforma

[← Detalle de Incidencia](#)

**Id:**  
1

**Descripción:**  
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed sed feugiat tellus.

**Fecha:**  
08/04/2021 10:30:00

**Asunto:**  
Incidencia 1

**Tipo:**  
AUSENCIA

**Contactos:**

**Gastos:**

**Localización:**

**Área:**

[Eliminar](#) [Editar](#)

Figura 9.11: Formulario de detalle de una entidad

[← Formulario de Incidencia](#)

**Descripción:**  
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed sed feugiat tellus.

**Fecha:**  
 Fecha: 08/04/2021  Hora: 10:30 [X](#)

**Asunto:**  
Incidencia 1

**Tipo:**  
AUSENCIA

**Contactos:**  
Contactos [Seleccionar un...](#)

**Gastos:**  
Gastos [Seleccionar un...](#)

**Localización:**

**Área:**  
[Editar en el mapa](#)

Figura 9.12: Formulario de edición de una entidad

## CAPÍTULO 9. SOLUCIÓN DESARROLLADA

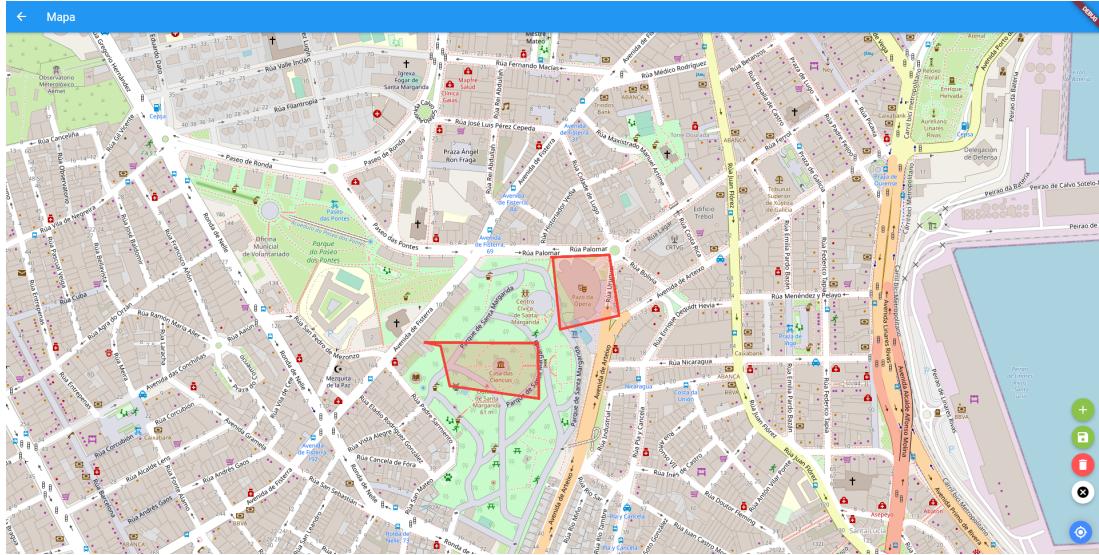


Figura 9.13: Mapa de edición

Lista de visitas	
<input type="button" value="Ordenar por"/>	Fecha
<input type="text" value="Filtrar"/>	<input type="button" value="Nuevo"/>
Atender a cliente Plaza de Lugo	Hecho
11/06/2021 07:30 - 09:30	
Atender a cliente Av. de Arteixo	Hecho
11/06/2021 10:10 - 11:30	
Recoger animal en Bertamiráns	Hecho
12/06/2021 08:00 - 08:30	
Recoger animal en Negreira	Hecho
12/06/2021 09:00 - 09:30	
Recoger animal en Carballo	Hecho
12/06/2021 12:00 - 12:30	
Atender a cliente Av. de Arteixo	Pendiente
12/06/2021 17:00 - 17:45	
Atender a cliente C/ Vicarrey Osorio	Pendiente
12/06/2021 18:15 - 19:30	
Atender a cliente Plaza de Lugo	Pendiente
13/06/2021 11:30 - 14:30	
Recoger animal en Malpica	Hecho
16/06/2021 11:00 - 11:30	
Atender a cliente C/ Vicarrey Osorio	Pendiente
17/06/2021 08:00 - 14:00	

Figura 9.14: Lista de visitas

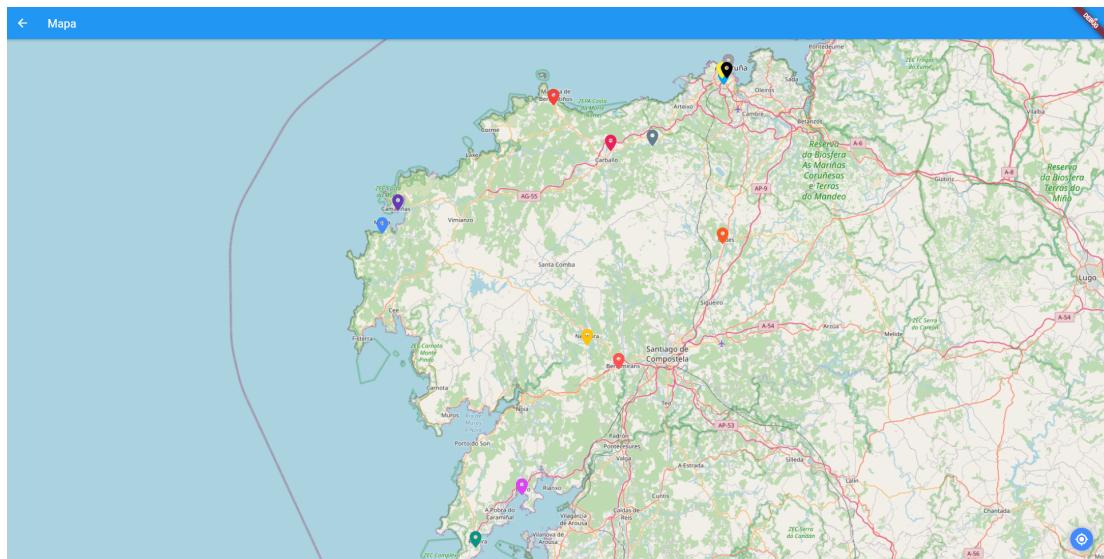


Figura 9.15: Mapa de visitas

## Capítulo 10

# Conclusiones y trabajo futuro

---

Al término de este proyecto, se puede afirmar que se han cumplido los objetivos establecidos para este:

- Se ha desarrollado una aplicación nativa multiplataforma orientada a la gestión del trabajo en movilidad con funcionalidades para la gestión de usuarios, formularios para la visualización, creación, modificación y borrado de entidades, listados y búsquedas, gestión de tareas, y funcionalidades de geolocalización y mapas.
- Se ha integrado la aplicación desarrollada en la línea de producto software del proyecto GEMA, permitiendo de este modo generar productos a partir de un conjunto de características seleccionadas y de un modelo de datos definido.
- Se proporcionan productos generados de calidad, seguros y fáciles de utilizar gracias al desarrollo de una interfaz simple e intuitiva y a la realización de múltiples pruebas.
- Se proporcionan productos generados multiplataforma gracias a la utilización de la tecnología Flutter en el desarrollo.
- Se proporcionan las instrucciones necesarias para la configuración y despliegue de los productos generados, facilitando su instalación.

Durante el proceso de desarrollo del proyecto, se han podido aplicar multitud de conocimientos adquiridos a lo largo de toda la titulación, entre los que se pueden mencionar:

- **Conocimientos técnicos** relacionados con las tecnologías y herramientas empleadas en este proyecto (Gitlab, Android Studio, etc.) que han permitido llevar a cabo el desarrollo de todas las tareas planificadas.
- Conocimientos sobre la aplicación práctica de **metodologías ágiles**, en particular Scrum, y sobre los mecanismos de planificación y estimación de las tareas de un proyecto, así como de asignación de recursos.

- 
- Conocimientos de **diseño de software** que han permitido definir la arquitectura y el diseño de la aplicación.

Además, gracias a la realización de este trabajo se han adquirido un gran número de nuevos conocimientos. Entre ellos se pueden destacar:

- Se ha profundizado en la implementación de líneas de producto software mediante el procedimiento seguido a la hora de trabajar con una LPS basada en la técnica de *scaffolding*.
- Se han comprendido los pilares básicos del desarrollo dirigido por modelos aplicado a la generación de código y se ha podido conocer el proceso a seguir a la hora de generar productos adaptados al dominio.
- Se han adquirido conocimientos en el empleo de Flutter, una tecnología con la que no se había trabajado previamente.
- Se han complementado los conocimientos ya existentes en el uso de algunas de las tecnologías empleadas y se ha profundizado en el desarrollo de aplicaciones móviles.
- Se ha experimentado de primera mano lo que supone llevar a cabo un proyecto de este alcance, resaltando la importancia de la planificación y el seguimiento constante del desarrollo.

El resultado obtenido tras la finalización de este proyecto tiene una aplicabilidad directa al mundo profesional ya que permite a las empresas del ámbito de la gestión del trabajo en movilidad contar con productos adaptados a sus necesidades y dominios concretos. La generación de aplicaciones móviles a medida proporciona a dichas compañías una solución eficiente que les permite llevar a cabo su actividad, ya que los productos generados cuentan con todas las funcionalidades necesarias para ello, a la vez que facilita las futuras adaptaciones que sean precisas si sus necesidades varían, todo ello haciendo uso de tecnologías completamente actuales y proporcionando productos multiplataforma, de calidad y fáciles de utilizar.

Pese a todo esto, la solución desarrollada podría mejorarse en algunos aspectos, como son:

- **Adaptación de la LPS** para incluir la configuración de aspectos exclusivos de entornos móviles. En este ámbito se podrían incluir características únicas de plataformas móviles, o permitir la configuración de elementos particulares.
- **Generación de productos empaquetados** en el formato de destino para el que se precisa el producto (APK, IPA, etc.) en lugar de generar el código fuente. Esto podría

realizarse estableciendo un parámetro de configuración en la LPS que permitiese seleccionar el formato del archivo e incluyendo la rutina de empaquetado como parte del proceso de generación.

- Proporcionar un **mecanismo de caché** que permitiese almacenar los datos en el dispositivo móvil en el caso de que no estuviese disponible la conexión con el servidor.
- **Gestión de notificaciones** para informar al usuario cuando se producen actualizaciones, por ejemplo, cuando se le asigna una nueva visita planificada. Esta mejora tendría que ir ligada a un mecanismo de **actualización en segundo plano** que solicitase los datos al servidor de forma periódica.



# **Apéndices**



**Apéndice A**

**Diagrama del modelo de  
características de GEMA**

---

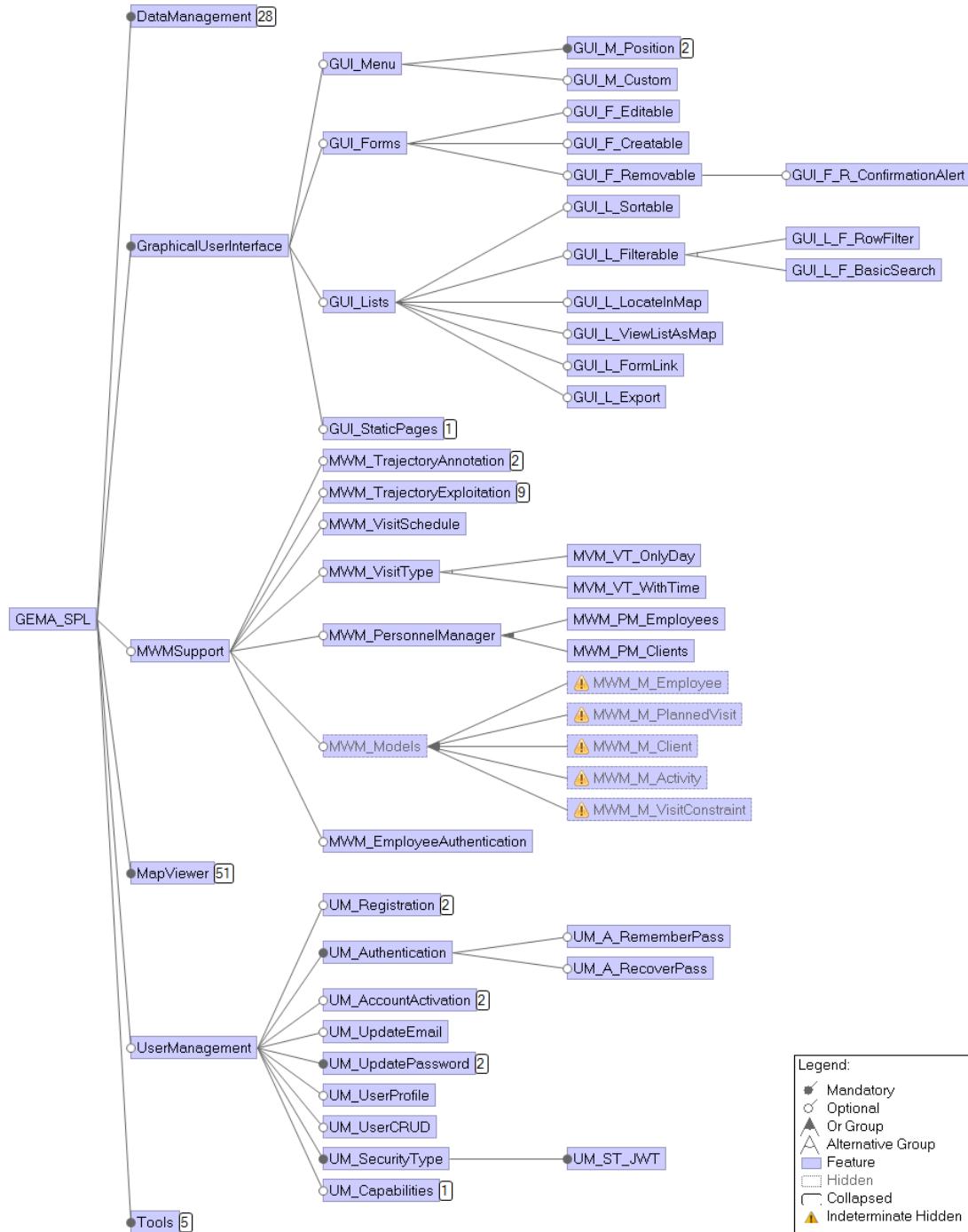


Figura A.1: Modelo de características de la LPS de GEMA

## **Apéndice B**

# **Prototipos de pantallas**

---



Figura B.1: Pantalla inicial

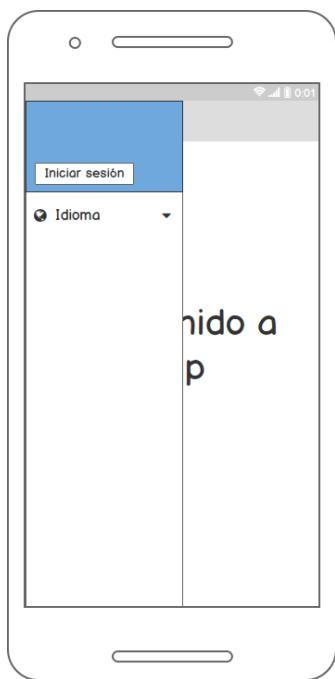


Figura B.2: Menú lateral sin usuario autenticado

## APÉNDICE B. PROTOTIPOS DE PANTALLAS

---



Figura B.3: Pantalla de iniciar de sesión



Figura B.4: Pantalla de recuperar la contraseña

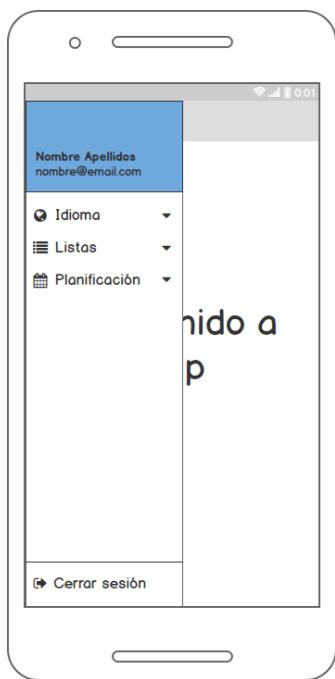


Figura B.5: Menú lateral con usuario autenticado

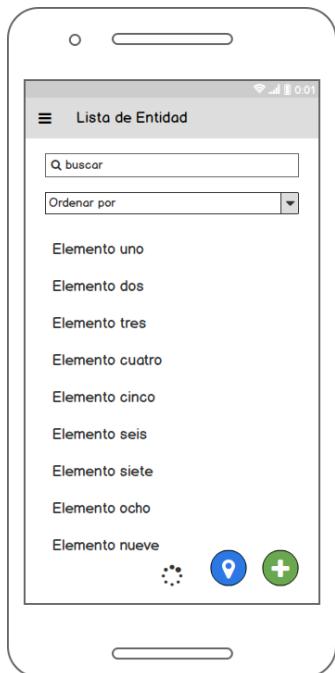


Figura B.6: Pantalla de lista con búsqueda simple y ordenación

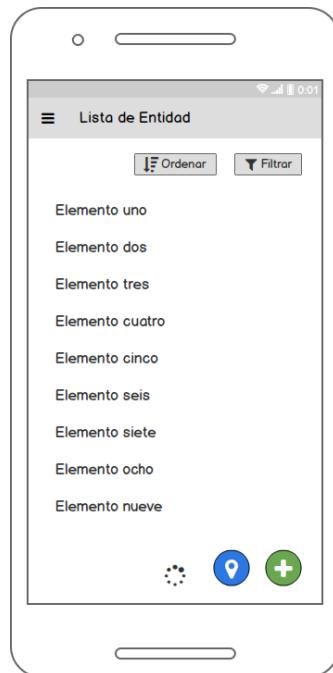


Figura B.7: Pantalla de lista con filtros y ordenación



Figura B.8: Pantalla de lista con *popup* en un elemento

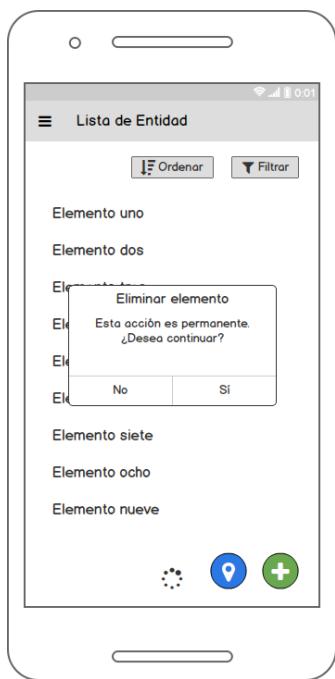


Figura B.9: Alerta de confirmación de borrado

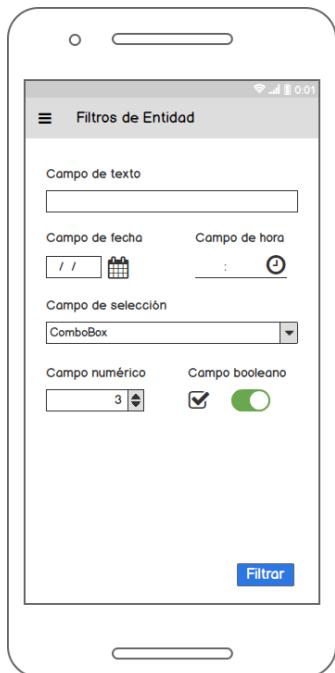


Figura B.10: Pantalla de filtros de una entidad

## APÉNDICE B. PROTOTIPOS DE PANTALLAS

---



Figura B.11: Pantalla de formulario de una entidad



Figura B.12: Pantalla de detalles de una entidad

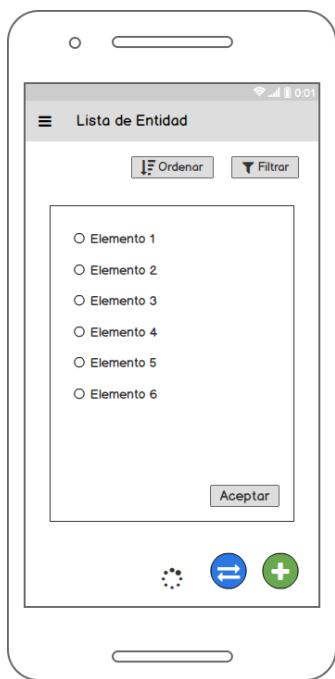


Figura B.13: Pantalla de ordenación



Figura B.14: Pantalla de mapa

APÉNDICE B. PROTOTIPOS DE PANTALLAS

---

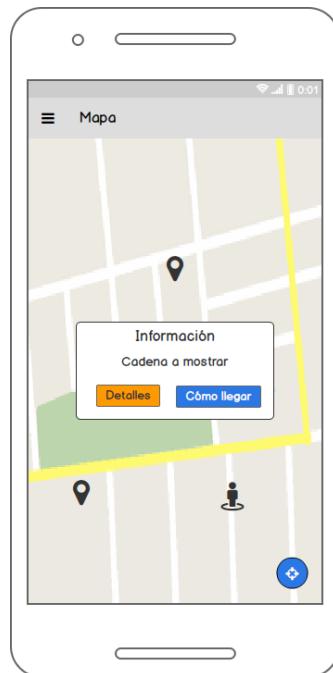


Figura B.15: Pantalla de información de un elemento en el mapa



Figura B.16: Pantalla de editar en el mapa

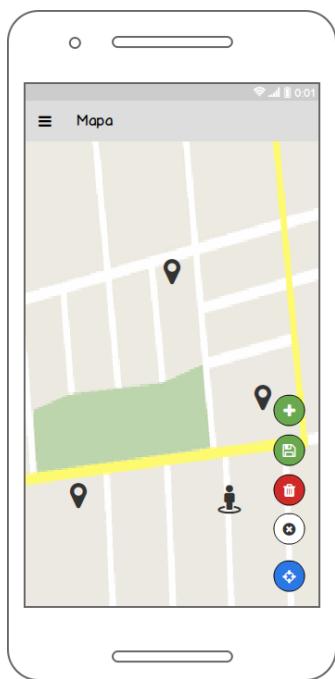


Figura B.17: Pantalla de mapa editando

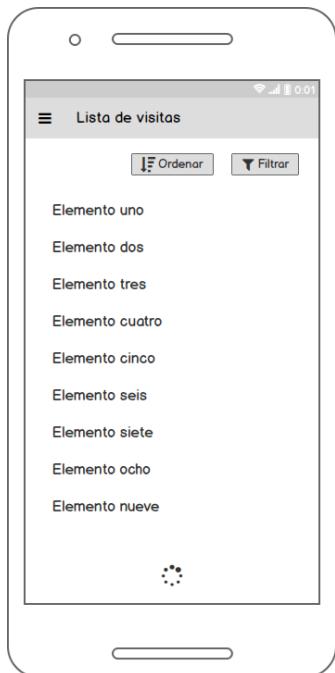


Figura B.18: Pantalla de lista de visitas

## APÉNDICE B. PROTOTIPOS DE PANTALLAS

---



Figura B.19: Pantalla de calendario de visitas



Figura B.20: Pantalla de modificación del tipo de vista en el calendario de visitas



Figura B.21: Pantalla de filtros en el calendario de visitas



Figura B.22: Pantalla de detalle de una visita



Figura B.23: Pantalla de formulario de una visita

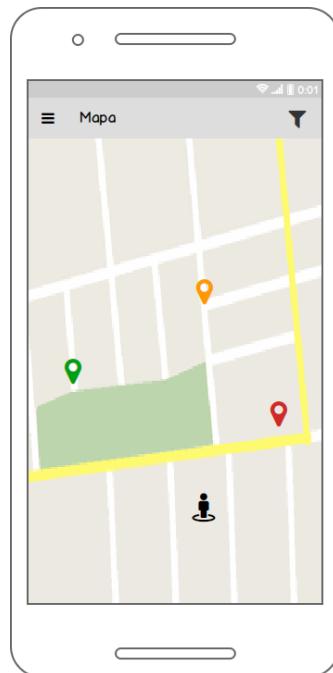


Figura B.24: Pantalla de mapa de visitas



Figura B.25: Pantalla de información de una visita en el mapa



Figura B.26: Pantalla de filtros en el mapa de visitas

## Apéndice C

# Manual de instalación

---

## C.1 Base de datos y servidor

Los **requisitos del sistema** para la instalación y despliegue del servidor son los siguientes:

- Java 8
- Gradle versión 5.4.1
- Servidor de PostgreSQL versión 11.3

Para la **configuración del entorno**, el primer paso es crear la base de datos y, a continuación, añadir la extensión PostGIS.

Para crear la base de datos se ejecuta el siguiente comando desde un terminal psql:

```
1 create database <database_name>;
```

Para añadir la extensión PostGIS es necesario ejecutar el siguiente comando desde un terminal conectado a la base de datos creada:

```
1 create extension postgis;
```

Una vez creada la base de datos, sus datos de conexión se establecen en el fichero de configuración: */src/main/resources/application.yml*.

Para el despliegue del servidor basta con ejecutar el siguiente comando en el directorio raíz del proyecto:

```
1 ./gradlew[ .bat ] bootRun
```

Una vez desplegado el servidor su API estará disponible a través de: <http://localhost:8080>

```
{  
    "apiUrl": "192.168.1.41:8080"  
}
```

Figura C.1: Configuración de la IP del servidor en la aplicación

## C.2 Aplicación nativa

Los **requisitos del sistema** para la instalación y despliegue de la aplicación son los siguientes:

- Flutter 2.0.6
- Dart 2.12.3
- Android 4.3

A la hora de desplegar la aplicación, el primer paso es **configurar** la dirección IP del servidor al que debe conectarse. Para ello, es preciso editar el fichero: *assets/cfg/app\_settings.json* como se muestra en la figura C.1.

Una vez configurada la IP, para lanzar la aplicación solo es necesario conectar el dispositivo móvil al ordenador y ejecutar el siguiente comando desde el directorio raíz del proyecto de Flutter:

```
1 flutter run
```

En el caso de querer desplegar la aplicación en una plataforma web los comandos a ejecutar son los siguientes:

```
1 flutter create .  
2 flutter run -d chrome
```

## Apéndice D

# Glosario de acrónimos

---

**API** *Application Program Interface.*

**EFM** *Eclipse Modeling Framework.*

**FOP** *Feature-Oriented Programming.*

**GIS** *Geographical Information System.*

**GPS** *Global Positioning System.*

**GTM** *Gestión del Trabajo en Movilidad.*

**HTML** *HyperText Markup Language.*

**HTTP** *Hypertext Transfer Protocol.*

**IDE** *Integrated Development Environment.*

**IP** *Internet Protocol.*

**JSON** *JavaScript Object Notation.*

**JWT** *JSON Web Token.*

**LPS** *Línea de Producto Software.*

**MDA** *Model Driven Architecture.*

**MDD** *Model Driven Development.*

**REST** *Representational State Transfer.*

**SDK** *Software Development Kit.*

---

**SGBD** *Sistema de Gestión de Bases de Datos.*

**SoaML** *Service oriented architecture Modeling Language.*

**SPL** *Software Product Line.*

**SysML** *Systems Modeling Language.*

**UML** *Unified Modeling Language.*

**UPIA** *UML Profile-Based Integrated Architecture.*

**XMI** *XML Metadata Interchange.*

**XML** *Extensible Markup Language.*

## Apéndice E

# Glosario de términos

---

**Bug** Error o problema en un programa informático.

**Desarrollo dirigido por modelos** Paradigma de desarrollo de software que se basa en la utilización de modelos para diseñar un sistema a distintos niveles de abstracción y, a partir de ellos, utilizar secuencias de transformación para generar el código final de las aplicaciones.

**Framework** Entorno software reusable que proporciona una funcionalidad particular para facilitar el desarrollo de una aplicación informática.

**Issue** Unidad de trabajo para completar una mejora en un sistema.

**Layout** Manera en la que están distribuidos los elementos y las formas dentro de un diseño.

**Línea de producto software** Conjunto de sistemas software, que comparten un conjunto común de características y que se desarrollan a partir de un sistema común de activos base (componentes reutilizables) de una manera preestablecida.

**Plugin** Componente software que agrega una característica específica a un programa informático.

**Programación orientada a características** Paradigma de programación para la construcción de sistemas software a gran escala que se basa en la descomposición del sistema en términos de las características que proporciona.

**Scaffolding** Técnica que consiste en la generación de código a partir de plantillas predefinidas y de una especificación proporcionada por el desarrollador.

**Singleton** Patrón de diseño que mantiene una única instancia de una clase.



# Bibliografía

---

- [1] “Página web de gome.” [En línea]. Disponible en: <https://www.gome.es/>
- [2] “Página web de worker app.” [En línea]. Disponible en: <https://overtel.com/movilidad/>
- [3] “Página web de krama got.” [En línea]. Disponible en: <https://www.kramagot.com/es/>
- [4] “Featurehouse: Language-independent, automated software composition.” [En línea]. Disponible en: <https://www.infosun.fim.uni-passau.de/spl/apel/fh/>
- [5] “Ahead tool suite.” [En línea]. Disponible en: <https://www.cs.utexas.edu/~schwartz/ATS/fopdocs/>
- [6] “Cide: Virtual separation of concerns.” [En línea]. Disponible en: <https://ckaestne.github.io/CIDE/>
- [7] “Página web de yeoman.” [En línea]. Disponible en: <https://yeoman.io/>
- [8] “Página web del proyecto gema.” [En línea]. Disponible en: <https://proxectogema.lbd.org.es/>
- [9] “Visión general del producto para rational software architect designer.” [En línea]. Disponible en: <https://www.ibm.com/docs/es/rational-soft-arch/9.5?topic=designer-rational-software-architect-product-overview>
- [10] “Página web de acceleo.” [En línea]. Disponible en: <https://www.eclipse.org/acceleo/>
- [11] “Página web de andromda.” [En línea]. Disponible en: <https://www.andromda.org/>
- [12] N. R. Brisaboa, A. Cortiñas, M. R. Luaces, and Óscar Pedreira, “Aplicando scaffolding en el desarrollo de líneas de producto software,” 2016.
- [13] A. Cortiñas, “Repositorio de github de spl-js-engine.” [En línea]. Disponible en: <https://github.com/AlexCortinas/spl-js-engine>

- [14] A. C. Álvarez, *Software product line for web-based geographic information systems*, 2017.
- [15] “Página web de flutter.” [En línea]. Disponible en: <https://flutter.dev/>
- [16] “Página web de dart.” [En línea]. Disponible en: <https://dart.dev/>
- [17] “Página web de material design.” [En línea]. Disponible en: <https://material.io/>
- [18] “Página web de postgresql.” [En línea]. Disponible en: <https://www.postgresql.org/>
- [19] “flutter\_map | flutter package.” [En línea]. Disponible en: [https://pub.dev/packages/flutter\\_map](https://pub.dev/packages/flutter_map)
- [20] “user\_location | flutter package.” [En línea]. Disponible en: [https://pub.dev/packages/user\\_location](https://pub.dev/packages/user_location)
- [21] “flutter\_secure\_storage | flutter package.” [En línea]. Disponible en: [https://pub.dev/packages/flutter\\_secure\\_storage](https://pub.dev/packages/flutter_secure_storage)
- [22] “geojson\_vi | dart package.” [En línea]. Disponible en: [https://pub.dev/packages/geojson\\_vi](https://pub.dev/packages/geojson_vi)
- [23] C. Larman, *Agile and Iterative Development: A Manager’s Guide*, 13th ed. Addison Wesley, 2004.
- [24] K. Schwaber and J. Sutherland, “The scrum guide™,” 2017. [En línea]. Disponible en: <https://www.scrumguides.org/scrum-guide.html>
- [25] “Página web de git.” [En línea]. Disponible en: <https://git-scm.com/>
- [26] “Gitlab agile delivery.” [En línea]. Disponible en: <https://about.gitlab.com/solutions/agile-delivery/>
- [27] “Página web de la documentación de gitlab | time tracking.” [En línea]. Disponible en: [https://docs.gitlab.com/ee/user/project/time\\_tracking.html](https://docs.gitlab.com/ee/user/project/time_tracking.html)
- [28] “Página web de android studio.” [En línea]. Disponible en: <https://developer.android.com/studio?hl=es>
- [29] “Página web de visual studio code.” [En línea]. Disponible en: <https://code.visualstudio.com/>
- [30] “Página web de dbeaver.” [En línea]. Disponible en: <https://dbeaver.io/>
- [31] “Página web de balsamiq.” [En línea]. Disponible en: <https://balsamiq.com/wireframes/>

## BIBLIOGRAFÍA

---

- [32] “Página web de magicdraw.” [En línea]. Disponible en: <https://www.nomagic.com/products/magicdraw>
- [33] “Página web de draw.io.” [En línea]. Disponible en: <https://www.draw.io/>
- [34] “Página web de microsoft excel.” [En línea]. Disponible en: <https://www.microsoft.com/es-es/microsoft-365/excel>
- [35] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.
- [36] K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.
- [37] F. J. van der Linden, K. Schmid, and E. Rommes, *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer, 2007.
- [38] I. Sommerville, *Ingeniería de Software*, 9th ed. Addison Wesley, 2011.
- [39] “¿qué es el lenguaje de programación dart? | inlab fib.” [En línea]. Disponible en: <https://inlab.fib.upc.edu/es/blog/que-es-el-lenguaje-de-programacion-dart>
- [40] “http | dart package.” [En línea]. Disponible en: <https://pub.dev/packages/http>
- [41] “global\_configuration | flutter package.” [En línea]. Disponible en: [https://pub.dev/packages/global\\_configuration](https://pub.dev/packages/global_configuration)
- [42] “jwt\_decoder | dart package.” [En línea]. Disponible en: [https://pub.dev/packages/jwt\\_decoder](https://pub.dev/packages/jwt_decoder)
- [43] “Página web de postgis.” [En línea]. Disponible en: <https://postgis.net/>
- [44] “Página web de java.” [En línea]. Disponible en: <https://www.java.com/es/>
- [45] “Página web de spring.” [En línea]. Disponible en: <https://spring.io/>
- [46] “Página web de gradle.” [En línea]. Disponible en: <https://gradle.org/>
- [47] “Página web de featureide.” [En línea]. Disponible en: <https://www.featureide.de/>

