



UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Proyecto de Fin de Grado en Ingeniería en Tecnologías de la Información

**Desarrollo de aplicaciones móviles multiplataforma:
Aplicación de información de enfermedades infecciosas.**

MIKEL GARCIA PIÑEIRO

Dirigido por: ISMAEL ABAD CARDIEL

Curso: Junio 2021



Desarrollo de aplicaciones móviles multiplataforma: Aplicación de información de enfermedades infecciosas.

Proyecto de Fin de Grado en Ingeniería en Tecnologías de la Información de modalidad específica

Realizado por: MIKEL GARCIA PIÑEIRO

Dirigido por: ISMAEL ABAD CARDIEL

Fecha de lectura y defensa: Junio 2021

Agradecimientos

Antes de comenzar me gustaría dar las gracias a las personas que me han animado a encarar estos estudios a distancia.

Me gustaría agradecer a mis padres todo el apoyo ofrecido para disponer del tan ansiado tiempo de estudio en una etapa en la que ha resultado complejo encontrar un equilibrio entre la vida laboral, personal y familiar.

Gracias a Itsaso por brindarme facilidades a la hora de realizar este proyecto y por su comprensión en momentos de estrés.

1. Resumen

El rastreo de contactos mediante dispositivos móviles está demostrando ser una herramienta eficaz para el control de enfermedades infecciosas. Debido a la actual situación mundial, muchos países han tratado de aportar distintas soluciones técnicas a este respecto sin contemplar las posibles consecuencias relativas a la privacidad individual. Estos sistemas se basan, esencialmente, en tres tipos de arquitecturas: centralizada, descentralizada e híbrida, que han sido analizadas para obtener una visión global sobre qué han aportado cada una de ellas y cuáles favorecen el anonimato del usuario enfermo.

El presente proyecto aborda la necesidad de desarrollar un sistema de trazabilidad de enfermedades infecciosas con anonimizado, al que llamamos SITEICA a partir de ahora, planteando una arquitectura descentralizada y apostando por la generación de identificadores únicos universales para reconocer de manera inequívoca tanto a los usuarios de la aplicación como a los encuentros cercanos.

Dentro de SITEICA se prevén 3 grandes bloques que permitan el seguimiento y la evaluación de las enfermedades: una aplicación móvil multiplataforma para el usuario enfermo, una aplicación móvil multiplataforma para el personal sanitario y, finalmente, una aplicación de servidor que permita la gestión y explotación de los datos.

En el presente PFG, se ha desarrollado la aplicación multiplataforma móvil para el usuario enfermo haciendo uso de Flutter, el SDK de Google que permite realizar aplicaciones para móvil, web y escritorio, compiladas de forma nativa. Esta aplicación recopila y transmite la información del movimiento del usuario y las interacciones con otras aplicaciones. También se encarga de mostrar la información respecto a la infección que existe en un determinado entorno y, por último, hace uso de la tecnología Bluetooth de baja energía (BLE) como una solución que ha sido considerada como óptima para el intercambio de información cercana entre dispositivos móviles.

2. Palabras clave

Rastreo de contactos

Enfermedades infecciosas

Aplicaciones multiplataforma

Anonimizado

Privacidad

Flutter

Material Design

Arquitectura descentralizada

Bluetooth de baja energía

Integración continua

Entrega continua

3. Abstract

Contact tracing using mobile devices is proving to be an effective infectious disease control tool. Due to the current global situation, many countries have tried to provide different technical solutions without considering the possible consequences regarding individual privacy. These systems are essentially based on three types of architectures: centralized, decentralized, and hybrid, which have been analyzed to obtain a global vision of what each has contributed and which ones favor the anonymity of the sick user.

This project addresses the need to develop an anonymized infectious disease traceability system, which we call SITEICA from now on, proposing a decentralized architecture and betting on the generation of universally unique identifiers to unequivocally recognize both users' app as well as close encounters.

Within SITEICA, three large blocks are foreseen that allow the monitoring and evaluation of diseases: a multiplatform mobile application for the sick user, a multiplatform mobile application for healthcare personnel, and, finally, a server application that allows management and operation of the data.

In this PFG, the multiplatform mobile application has been developed for the sick user making use of Flutter, Google's SDK that allows making applications for mobile, web, and desktop, compiled natively. This application collects and transmits the information of the user's movement and interactions with other applications. It is also responsible for displaying information regarding the infection that exists in a specific environment. Finally, it uses Bluetooth low energy technology (BLE) as a solution that has been considered optimal for exchanging close information between mobile devices.

4. Keywords

Contact tracing

Infectious diseases

Cross-platform applications

Anonymized

Privacy

Flutter

Material Design

Decentralized architecture

Bluetooth low energy

Continuous integration

Continuous delivery

Tabla de contenidos

1. Resumen.....	7
2. Palabras clave.....	9
3. Abstract.....	11
4. Keywords.....	13
5. Introducción.....	27
5.1. Marco del proyecto.....	27
5.2. Motivación.....	27
5.3. Objetivo.....	27
5.4. Estructura de la memoria.....	28
6. Estado del arte del problema.....	29
6.1. Situación actual.....	29
6.2. Rastreo con aplicaciones móviles.....	29
6.3. Arquitectura centralizada.....	31
6.4. Arquitectura descentralizada.....	33
6.5. Arquitectura híbrida.....	35
6.6. Conclusiones.....	36
7. Análisis del sistema.....	39
7.1. Requisitos funcionales.....	39
7.1.1. Alta en el sistema.....	39
7.1.2. Registro de encuentros.....	39
7.1.3. Notificación de usuario positivo.....	40
7.1.4. Carga de identificadores de encuentro.....	40
7.1.5. Análisis de riesgo.....	40
7.1.6. Mapa de infecciones.....	40
7.1.7. Información de evolución.....	41
7.2. Requisitos no funcionales.....	41
7.3. Casos de uso.....	41
7.3.1. Registro de usuario.....	42

7.3.2. Generación de identificadores de encuentros.....	42
7.3.3. Intercambio de identificadores de encuentros.....	43
7.3.4. Notificación de usuario positivo.....	43
7.3.5. Carga de semillas al servidor.....	44
7.3.6. Análisis de riesgo.....	44
7.3.7. Mapa de infección.....	45
7.3.8. Información de evolución.....	46
7.3.9. Diagrama de casos de uso.....	47
8. Tecnologías empleadas.....	49
8.1. SQLite.....	49
8.2. Bluetooth de baja energía.....	50
8.3. Conexión HTTPS con el servidor central.....	51
8.4. Flutter.....	52
8.5. Plugins de Flutter.....	54
8.5.1. Beacon Broadcast.....	54
8.5.2. Beacons Plugin.....	54
8.5.3. Flutter Map.....	54
8.5.4. Geolocator.....	55
8.5.5. Sqflite.....	55
8.5.6. Sqflite Migration Service.....	55
8.5.7. Injector.....	55
8.5.8. Json Serializable.....	55
8.5.9. UUID Enhanced.....	56
8.5.10. Intro Slider.....	56
8.5.11. Bluetooth Enable.....	56
8.5.12. Charts Flutter.....	56
8.6. Usabilidad y experiencia de usuario.....	56
8.6.1. Google Design.....	57
8.6.2. Material Design.....	57

8.7. Arquitectura tecnológica.....	58
9. Diseño del sistema.....	61
9.1. Funcionamiento global del sistema.....	61
9.1.1. Registro.....	61
9.1.2. Generación de identificadores de encuentros.....	63
9.1.3. Intercambio de identificadores de encuentros.....	63
9.1.4. Análisis de riesgo.....	64
9.1.5. Notificación de usuario positivo.....	65
9.1.6. Ciclo de trazabilidad de enfermedades infecciosas.....	66
9.1.7. Otras opciones del aplicativo.....	67
9.2. Prototipado.....	69
9.2.1. Bienvenida y registro.....	69
9.2.2. Inicio de la aplicación.....	70
9.2.3. Notificación de usuario positivo.....	71
9.2.4. Mapa de infecciones.....	73
9.2.5. Evolución.....	74
9.3. Diseño visual.....	75
9.4. Modelo entidad-relación.....	75
9.4.1. Tabla de usuario.....	76
9.4.2. Tabla de identificadores de encuentros.....	76
9.4.3. Tabla de encuentros.....	76
9.4.4. Tabla de notificaciones de positivo.....	77
9.4.5. Tabla de encuentros de riesgo.....	78
9.4.6. Tabla de análisis encuentros de riesgo.....	78
9.4.7. Tabla de información de evolución.....	79
9.4.8. Tabla de provincias.....	79
9.4.9. Tabla de evolución por provincias.....	79
10. Instalación y configuración del entorno.....	81
10.1. Aplicación de Flutter.....	81

10.2. Estructura básica del proyecto.....	81
10.3. Dependencias del proyecto.....	83
11. Implementación.....	85
11.1. Construyendo la interfaz de usuario.....	85
11.2. Inicialización de la base de datos	88
11.3. Bienvenida a la aplicación.....	89
11.4. Registro.....	92
11.5. Configurar la aplicación como emisor BLE.....	98
11.6. Configurar la aplicación como observador BLE.....	100
11.7. Inicio.....	104
11.7.1. Contactos de riesgo.....	105
11.7.2. Estado del sistema.....	107
11.7.3. Notificación de positivo.....	109
11.8. Mapa de infecciones.....	112
11.9. Evolución.....	113
12. Pruebas.....	117
12.1. CP-001. Permitir acceso a ubicación.....	117
12.2. CP-002. Registro.....	117
12.3. CP-003. Estado del sistema.....	118
12.4. CP-004. Generación de identificadores de encuentro.....	119
12.5. CP-005. Intercambio de identificadores de encuentro.....	119
12.6. CP-006. Notificación de usuario positivo.....	120
12.7. CP-007. Código de notificación ya utilizado.....	121
12.8. CP-008. Carga de identificadores de encuentros al servidor.....	122
12.9. CP-009. Análisis de riesgo.....	123
12.10. CP-010. Mapa de infección.....	123
12.11. CP-011. Datos de evolución.....	124
13. Conclusiones.....	127
13.1. Conclusiones.....	127

13.2. Líneas futuras.....	127
14. Bibliografía.....	129
15. Anexos.....	133
15.1. Anexo. Entorno tecnológico.....	133
15.1.1. IntelliJ IDEA.....	133
15.1.2. Sistema de control de versiones.....	133
15.2. Integración y entrega continua.....	134
15.3. Integración y entrega continua con GitHub Actions.....	135
15.3.1. Crear el proyecto en GitHub.....	135
15.3.2. Añadir el archivo de definición del flujo de trabajo.....	135
15.3.3. Comandos del flujo de trabajo.....	135
15.3.4. Crear un token de acceso personal en GitHub.....	137
15.4. Inyección de dependencias.....	137
15.5. Herramientas de documentación.....	138

Índice de figuras

Ilustración 1: Arquitectura centralizada.....	33
Ilustración 2: Arquitectura descentralizada.....	35
Ilustración 3: Arquitectura híbrida.....	36
Ilustración 4: Arquitectura adoptada en Siteica.....	37
Ilustración 5: Diagrama de casos de uso.....	47
Ilustración 6: Topología broadcasting en BLE.....	50
Ilustración 7: Arquitectura de Flutter. Fuente: https://flutter.dev/	53
Ilustración 8: Arquitectura tecnológica.....	59
Ilustración 9: DA Registro.....	62
Ilustración 10: DA Generación de identificadores de encuentros.....	63
Ilustración 11: DA Intercambio de semillas.....	64
Ilustración 12: DA Análisis de riesgo.....	65
Ilustración 13: DA Notificación de usuario positivo.....	66
Ilustración 14: PMN del ciclo de trazabilidad de enfermedades infecciosas.....	67
Ilustración 15: DA Mapa de infección.....	68
Ilustración 16: DA Evolución.....	69
Ilustración 17: Prototipo de Bienvenida y registro.....	70
Ilustración 18: Prototipo de Inicio.....	71
Ilustración 19: Prototipo de Notificación de usuario positivo.....	72
Ilustración 20: Prototipo de Mapa de infección.....	73
Ilustración 21: Prototipo de Evolución.....	74
Ilustración 22: Modelo entidad-relación.....	75
Ilustración 23: Resultado de ejecutar flutter doctor.....	81
Ilustración 24: Estructura del proyecto.....	82
Ilustración 25: Dependencias del proyecto.....	83
Ilustración 26: Método principal de la aplicación.....	85

Ilustración 27: Inicialización en main.dart del proyecto.....	86
Ilustración 28: Clase SiteicaApp, raíz del árbol de widgets.....	87
Ilustración 29: Archivos estáticos incluidos en el proyecto.....	88
Ilustración 30: Clase DatabaseService e inicialización de BD.....	88
Ilustración 31: Archivos de migración de base de datos.....	88
Ilustración 32: Lista de diapositivas de bienvenida.....	89
Ilustración 33: Constantes de estilo para textos de las diapositivas.....	90
Ilustración 34: Método build() de la vista WelcomePage.....	90
Ilustración 35: Widget que devuelve un objeto Icon.....	91
Ilustración 36: Navegar en la pila de navegación a RegisterPage.....	91
Ilustración 37: Pantalla de bienvenida de Siteica.....	91
Ilustración 38: Clase ProvinceService y método para obtener provincias.....	92
Ilustración 39: Obtener las provincias de la BD.....	93
Ilustración 40: Elementos de columna resaltados en rojo.....	94
Ilustración 41: Elementos de fila resaltados en azul.....	94
Ilustración 42: Método build de la clase RegisterPage.....	95
Ilustración 43: Elementos de UI para habilitar el Bluetooth.....	96
Ilustración 44: Método privado para habilitar el Bluetooth.....	96
Ilustración 45: Elementos de UI para la selección de provincia.....	97
Ilustración 46: Fila con los botones de la vista.....	97
Ilustración 47: Método privado encargado de añadir al usuario.....	98
Ilustración 48: Importar la librería beacon_broadcast.....	98
Ilustración 49: Singleton de EncounterSeedService y llamada a getEncounterSeed.....	99
Ilustración 50: Creación de una semilla de encuentro.....	99
Ilustración 51: Emisión de información a través de BLE.....	99
Ilustración 52: Temporizador que controla la caducidad de semillas.....	100
Ilustración 53: Manejo del servicio de observación BLE en segundo plano para Android.....	100
Ilustración 54: Autorización del servicio Location en iOS.....	101
Ilustración 55: Claves necesarias en Info.plist para iOS.....	101

Ilustración 56: Mensaje de permisos de localización para Android.....	102
Ilustración 57: Suscripción al stream de datos del observador BLE.....	103
Ilustración 58: Deserialización del frame de datos BLE.....	103
Ilustración 59: Almacenamiento del encuentro en BD local.....	103
Ilustración 60: Inicio de monitorización.....	104
Ilustración 61: Datos recibidos por un terminal en un encuentro.....	104
Ilustración 62: Lista de widgets que componen las secciones de inicio.....	104
Ilustración 63: Contenido de la vista de inicio.....	105
Ilustración 64: Obtención del último análisis de riesgo de la BD.....	105
Ilustración 65: Búsqueda de identificadores de riesgo.....	106
Ilustración 66: Análisis de riesgo.....	106
Ilustración 67: Widgets que componen la sección de contactos de riesgo.....	107
Ilustración 68: Método para la comprobación del Bluetooth.....	107
Ilustración 69: Widgets que componen la sección del estado del sistema.....	108
Ilustración 70: Botón para habilitar de forma manual el Bluetooth.....	108
Ilustración 71: Botón y su evento para la notificación de positivo.....	109
Ilustración 72: Elemento TextField para el código de diagnóstico.....	109
Ilustración 73: Apertura del calendario y selección de fecha.....	110
Ilustración 74: Método que comprueba si el OTP ya está usado.....	111
Ilustración 75: Acción de confirmación del envío de la notificación.....	111
Ilustración 76: Actualización de registros de encuentro transmitidos.....	112
Ilustración 77: Configuración de marcadores del mapa.....	112
Ilustración 78: Interfaz de usuario de la vista de "Mapa".....	113
Ilustración 79: Consultas para obtener datos de evolución.....	114
Ilustración 80: Obtención de totales del mes y de la semana anterior.....	114
Ilustración 81: Configuración de la serie de datos de la gráfica de tiempo.....	115
Ilustración 82: Elementos de la interfaz de la sección "Casos diarios".....	116
Ilustración 83: Solicitud de permisos de ubicación.....	117
Ilustración 84: Comprobación y activación del Bluetooth.....	118

Ilustración 85: Registro de usuario en la BD local.....	118
Ilustración 86: Estado del sistema y activación del Bluetooth.....	118
Ilustración 87: Identificadores de encuentro en la BD local cada 15 minutos.....	119
Ilustración 88: Registros de encuentros en la base de datos local.....	120
Ilustración 89: Notificación de usuario positivo.....	121
Ilustración 90: Código ya utilizado en notificación de positivo.....	122
Ilustración 91: Registros de encuentros transmitidos.....	122
Ilustración 92: Notificación de contacto de riesgo.....	123
Ilustración 93: Mapa de infección.....	124
Ilustración 94: Datos de evolución.....	125
Ilustración 95: CI / CD. Fuente: https://www.redhat.com/en/topics/devops/what-is-ci-cd	134
Ilustración 96: Archivo YAML con la definición del flujo de trabajo.....	136
Ilustración 97: Configurar el repositorio de GitHub.....	137

Índice de tablas

Tabla 1: RF-001. Alta en el sistema.....	39
Tabla 2: RF-002. Registro de encuentros.....	39
Tabla 3: RF-003. Notificación de usuario positivo.....	40
Tabla 4: RF-004. Carga de identificadores de encuentro.....	40
Tabla 5: RF-005. Análisis de riesgo.....	40
Tabla 6: RF-006. Mapa de infecciones.....	40
Tabla 7: RF-007. Información de evolución.....	41
Tabla 8: CU-001. Registro de usuario.....	42
Tabla 9: CU-002. Generación de identificadores de encuentros.....	43
Tabla 10: CU-003. Intercambio de identificadores de encuentros.....	43
Tabla 11: CU-004. Notificación de usuario positivo.....	44
Tabla 12: CU-005. Carga de identificadores de encuentros al servidor.....	44
Tabla 13: CU-006. Análisis de riesgo.....	45
Tabla 14: CU-007. Mapa de infección.....	45
Tabla 15: CU-008. Información de evolución.....	46
Tabla 16: CP-001. Solicitud de permisos de ubicación válida.....	117
Tabla 17: CP-002. Registro válido.....	118
Tabla 18: CP-003. Comprobación del estado del sistema válido.....	118
Tabla 19: CP-004. Generación de identificadores de encuentro válido.....	119
Tabla 20: CP-005. Intercambio de identificadores de encuentro válido.....	120
Tabla 21: CP-006. Notificación de usuario positivo válido.....	121
Tabla 22: CP-007. Código de notificación ya utilizado.....	122
Tabla 23: CP-008. Carga de identificadores de encuentros al servidor.....	122
Tabla 24: CP-009. Análisis de riesgo válido.....	123
Tabla 25: CP-010. Mapa de infección válido.....	124
Tabla 26: CP-011. Datos de evolución válidos.....	125

5. Introducción

5.1. Marco del proyecto

El presente Proyecto de Fin de Grado (PFG) consiste en el análisis y diseño completo de una aplicación móvil multiplataforma para el rastreo de enfermedades infecciosas como pieza fundamental, orientada al usuario enfermo, del ecosistema de SITEICA.

El sistema será una herramienta capaz de comunicarse de máquina a máquina (M2M) a través de la tecnología BLE y recoger, de forma anónima, la información oportuna de los encuentros realizados entre ellas. Esta información permitirá ser capaces de detectar e informar sobre posibles encuentros de riesgo, así como ofrecer información sobre el estado de la enfermedad.

5.2. Motivación

El proyecto surge tras los numerosos debates sobre privacidad y derechos fundamentales que emergen debido a la aparición, de forma apresurada, de múltiples aplicaciones de rastreo de contactos como respuesta a la crisis mundial del coronavirus. Algunas de estas soluciones han recogido información personal del usuario en la fase de registro, como el número de teléfono o la edad, que ha sido almacenada en un servidor central donde se corre el riesgo de que se termine identificando a la persona a través de la combinación de fuentes de datos. Además, muchas de las aplicaciones existentes no han sido transparentes a la hora de informar sobre los sensores e información a la que acceden.

Debido a la necesidad de buscar una solución que priorice la privacidad por encima de todo, se ha tratado de crear un sistema donde la información permanezca anónima y descentralizada, sin la necesidad de recoger ni transmitir datos personales del usuario de la aplicación. De forma complementaria, se ha perseguido el objetivo de crear una aplicación móvil multiplataforma con la finalidad de enfocar los esfuerzos de desarrollo en un único código fuente, permitiendo así el ahorro en tiempo y recursos.

Finalmente, gracias a la información recogida por SITEICA, se ha buscado proporcionar al usuario información relevante y fiable sobre la propagación de la infección, así como revelar datos sobre el entorno en el que se encuentran en relación con la enfermedad.

5.3. Objetivo

El principal objetivo es el análisis, diseño y desarrollo de una aplicación móvil multiplataforma para el rastreo de enfermedades infecciosas que no recoja datos personales del usuario, respetando así su anonimato.

De forma general, el sistema deberá ser capaz de dar de alta a los usuarios de forma totalmente anónima y deberá poder comunicarse con otros dispositivos cercanos con el objetivo de recopilar información sobre el encuentro sin transmitir información sensible.

De esta forma, los objetivos que persigue el proyecto son los siguientes:

1. Análisis sobre las distintas arquitecturas desarrolladas en torno a las aplicaciones de rastreo y su impacto en la privacidad individual.
2. Realización del diseño de la aplicación en relación a la arquitectura escogida en la que prevalezca el anonimato.
3. Análisis de la tecnología BLE para realizar una comunicación M2M con el objetivo de recopilar información sobre los encuentros entre dispositivos.
4. Mantener el anonimato tanto en el registro como en el intercambio de datos.
5. Implementación del sistema en Flutter capaz de intercambiar datos con otras aplicaciones e informar sobre el estado de la propagación de la enfermedad, así como de posibles encuentros de riesgo realizados por el usuario.

5.4. Estructura de la memoria

El proyecto se inicia con una descripción del Estado del Arte en el que se analiza la situación actual de las aplicaciones de rastreo de contactos y se realiza un recorrido a través de las arquitecturas más utilizadas, descubriendo así qué ventajas aporta cada una a la privacidad del usuario final. Este capítulo finaliza realizando unas conclusiones finales acerca de la solución adoptada en SITEICA.

El siguiente capítulo se centra en realizar un análisis del sistema que se ha desarrollado a través de la descripción de los requerimientos de los servicios que ha de proveer la aplicación, así como la exposición de sus funciones en forma de casos de uso.

La memoria continúa con la exposición completa de las tecnologías que se han utilizado para resolver con éxito las necesidades expuestas en los requerimientos y casos de uso, además de exponer las decisiones adoptadas en relación a usabilidad y experiencia de usuario, algo primordial en una aplicación móvil. El capítulo se cierra realizando una exposición completa de la arquitectura tecnológica del proyecto, detallando dónde se han utilizado las distintas tecnologías y qué requerimientos han satisfecho.

Posteriormente se detalla el diseño completo del sistema. En este capítulo se detalla el funcionamiento global de la aplicación del usuario enfermo, se realiza una aproximación visual a través de su prototipado y, finalmente, se analiza el modelado de la base de datos local y en qué funcionalidades afecta.

El capítulo posterior se dedica a la descripción el proceso de instalación y configuración del entorno necesario para la realización y ejecución del proyecto.

Finalmente, los dos últimos capítulos se centran en la implementación de la solución en Flutter en base al análisis previo realizado y en la realización de los casos de prueba necesarios para la comprobación del cumplimiento de los requisitos, respectivamente.

Como anexo se ha incluido un capítulo dedicado al entorno tecnológico específico del proyecto, donde se mencionan ciertas configuraciones con el objetivo implantar prácticas DevOps.

6. Estado del arte del problema

6.1. Situación actual

El 11 de marzo de 2020, la Organización Mundial de la Salud (OMS) declara brote de enfermedad por coronavirus (COVID-19) como pandemia a nivel mundial. En esos momentos la enfermedad se propaga a una velocidad alarmante, habiéndose multiplicado por 13 los casos fuera de China, lugar que se considera el epicentro del virus. Algunos estudios [1] han situado el número de reproducción, también conocido como R_0 , entre 2.24 y 3.58.

Debido a la rápida expansión del virus, se ha considerado de vital importancia realizar rastreos de forma temprana y poner en cuarentena a los contactos cercanos con el objetivo de romper la cadena de transmisión, aplanando así la curva de infectados. De esta forma, se ha comprobado [2] que la estrategia basada en los datos biomédicos de la persona contagiada es insuficiente para rastrear la enfermedad y poner en cuarentena a los posibles afectados, a causa de la rapidez con la que se desplaza el individuo en las sociedades modernas.

Otras estrategias de contención se han basado en la contratación de rastreadores de Atención Primaria, personas encargadas de detectar enfermos y rastrear sus contactos. En Estados Unidos, el estado de Massachusetts asignó un presupuesto de 44 millones de dólares estadounidenses para la contratación de 1.000 rastreadores. Este plan, además de una gran inversión económica, requiere de tiempo para realizar la contratación del personal y su posterior instrucción, algo que juega en contra de la contención de la enfermedad. De hecho, se estima [3] que el número de nuevos rastreadores que necesita Estados Unidos asciende a 200.000 a fin de satisfacer la demanda actual.

Muchos países han optado por limitar la movilidad y cerrar ciudades para evitar una mayor propagación, poniendo de manifiesto que los enfermos asintomáticos podrían estar afectando de forma aún más negativa a la tasa de contagios. De este modo, se ha demostrado [2] que es más económico cortar la cadena de transmisión y proteger al sector más vulnerable de la población que tratar a una gran cantidad de enfermos en los hospitales, añadiendo el riesgo de colapso del sistema sanitario. Por esta misma razón, muchos países han volcado sus esfuerzos en desarrollar una estrategia que les permita realizar un seguimiento individual capaz de ofrecer información sobre los contactos cercanos de forma rápida.

El objetivo principal de estas medidas es la disminución de la tasa de propagación del virus y conducir al llamado 'aplanamiento de la curva', ralentizando las infecciones por COVID-19 hasta que se desarrolle una vacuna aprobada. Por lo tanto, debido al alto coste económico y temporal de las estrategias basadas en la contratación de rastreadores [5], se ha optado por la introducción de aplicaciones de rastreo de contactos que ayuden a la identificación de las personas que tuvieron contacto cercano con el portador positivo.

6.2. Rastreo con aplicaciones móviles

Los análisis [4] de los datos más recientes han confirmado que el rastreo de contactos mediante teléfonos inteligentes, cuando se adopta durante un primer brote, solo puede ser

efectivo si se utiliza una tecnología de rastreo de contactos rápida y de alta precisión, y cuando una proporción significativa (más del 80%) de la población usa la aplicación en sus terminales. Estos estrictos requisitos hacen que sea poco probable que sea una solución viable por sí misma. Afortunadamente, para futuros brotes, y bajo la condición de que al menos el 20% de las personas obtengan la inmunización o que la tasa de reproducción se reduzca por otras medidas de distanciamiento social más indulgentes, el rastreo de contactos de teléfonos inteligentes puede ser muy eficaz, incluso cuando solo una parte de la población está dispuesta a utilizarlo (menos del 60%).

Dada la situación de emergencia a la que se han visto sometidos muchos países durante la pandemia, el desarrollo de soluciones técnicas para la implantación de aplicaciones de rastreo de contactos puede haberse realizado sin una comprensión completa de los efectos colaterales. Como consecuencia de esto, se han generado una gran cantidad de debates acerca de las bondades y los riesgos de la recogida de datos personales. De hecho, muchas de las implantaciones que se han hecho a nivel mundial han surgido antes de que las políticas de privacidad, los requisitos de anonimato y las políticas de retención de datos se hayan desarrollado completamente [6].

Las aplicaciones móviles de rastreo se utilizan para obtener la ubicación del usuario y los detalles de los contactos que se hayan realizado. De este modo, cuando las personas se infecten, se podrá usar su teléfono móvil para rastrear los contactos anteriores e identificar cualquier individuo que pueda ser un potencial contagio. Debido a la naturaleza de estas aplicaciones, se ha generado un amplio debate sobre su arquitectura, gestión de datos, eficacia, privacidad y seguridad.

Estas aplicaciones, que hacen la labor de rastreadores digitales, utilizan principalmente dos tecnologías en las que se apoyan para la obtención de datos referentes a ubicaciones y posibles contactos: Global Positioning System (GPS) y Bluetooth. Gracias a investigaciones [3], sabemos que la tecnología GPS puede ser inexacta y generar falsos positivos. Terminales que se encuentran en el mismo edificio pero en pisos diferentes, se registrarán en la misma localización. Del mismo modo, la ausencia de redes móviles o WiFi en zonas de baja o nula cobertura, como puede ser el metro, puede implicar la pérdida de mucha información valiosa. La tecnología Bluetooth ha demostrado ser más precisa a la hora de recoger información acerca de contactos cercanos.

Otras soluciones [5] han optado por el uso de redes móviles y redes Wi-Fi para determinar cuándo se realiza un contacto. No obstante, estas soluciones ofrecen resultados poco precisos. Las redes móviles, divididas en áreas delimitadas llamadas celdas, pueden cubrir cientos de miles de metros. Por otro lado, las redes Wi-Fi, aunque también pueden darnos datos de poca precisión, pueden utilizar la fuerza de la señal recibida (RSSI) para hacer una estimación de la distancia.

Es posible evaluar los diferentes mecanismos que intervienen en la obtención de una red de contactos utilizando los datos recopilados por la Universidad de NCCU. Este análisis se realizó utilizando un total de 115 dispositivos Android de estudiantes que asisten a la Universidad Nacional de Chengchi, Taiwán. El rastreo fue monitorizado durante un período de 15 días, donde se recogieron datos GPS, puntos de acceso Wi-Fi y contactos cercanos realizados mediante Bluetooth. El tiempo se especifica con una resolución de un segundo y la

información de posición se redondea a metros. Los análisis [4] arrojan un resultado de 7.66 como valor medio de número de contactos por día y persona, mientras que los resultados de cada una de estas soluciones nos ofrecen una clasificación promedio de la exactitud con la que se obtiene una red de contactos: a través de las redes móviles se obtuvo un resultado de 89.31, con Wi-Fi un 69.18, GPS un 41.00 y Bluetooth, en cambio, un 27.48. Esto nos demuestra que el rango es decisivo cuando se trata de estimar el número de contactos.

Otro aspecto importante de cara al desarrollo de una solución para la recolección de datos en las aplicaciones móviles de rastreo es el tipo de arquitectura, pieza fundamental para encarar los problemas de seguridad y privacidad a los que se tiene que enfrentar una aplicación de estas características. De esta manera, se han analizado [5] tres de las arquitecturas más comunes sobre las que han trabajado algunas aplicaciones de rastreo COVID-19: centralizadas, descentralizadas e híbridas, esta última una combinación de las dos anteriores. El análisis se ha centrado en cómo se usa al servidor y qué datos debe almacenar.

6.3. Arquitectura centralizada

Dentro de las soluciones de arquitecturas centralizadas se encuentra BlueTrace [8], un protocolo de código abierto que facilita el rastreo digital de contactos con el firme objetivo de detener la propagación de la pandemia. Desarrollado inicialmente por el gobierno de Singapur, BlueTrace adopta, como uno de sus principios fundamentales, la preservación de la privacidad y la cooperación de las autoridades sanitarias y sirve de cimiento para la aplicación TraceTogether. Algunos países, como Nueva Zelanda, ya están considerando la adopción de BlueTrace. Otros, como Australia, ya usan este protocolo como solución para el rastreo de contactos.

Para lograr el objetivo de la preservación de la privacidad, la información personal se recopila solo una vez en el momento en el que el usuario realiza su registro y solo se usa para contactar a pacientes potencialmente infectados. Además, los usuarios pueden cancelar su participación en cualquier momento, borrando así toda información personal e impidiendo que los datos registrados se puedan rastrear. El rastreo de contactos se realiza de forma totalmente local en el dispositivo cliente mediante BLE, almacenando todos los encuentros en un registro histórico de contactos que registra los encuentros de los últimos 21 días. Los usuarios en el registro de contactos se identifican mediante "identificaciones temporales" anónimos emitidos por la autoridad sanitaria. Esto significa que la identidad de un usuario no puede ser comprobada por nadie excepto por la autoridad sanitaria con la que está registrado. Además, dado que las identificaciones temporales cambian con regularidad, los terceros malintencionados no pueden rastrear a los usuarios mediante la observación de las entradas del registro a lo largo del tiempo.

La arquitectura centralizada [5], [8] consta de las siguientes fases:

1. Instalación y registro: un usuario descarga la aplicación y registra detalles como nombre, número de teléfono, franja de edad y código postal, datos que se transmiten al servidor. El servidor verifica el número de móvil enviando una contraseña de un solo uso (OTP) por SMS. Tras la verificación, el servidor calcula una identificación temporal (TempID) que únicamente será válida durante un breve período de tiempo (es

recomendable que no supere los 15 minutos). Así, el TempID y el tiempo de caducidad se transmiten a la aplicación del usuario.

2. Registro de encuentros: tan pronto como dos usuarios de la aplicación entran en contacto, se lleva a cabo el intercambio de un "mensaje de encuentro" por Bluetooth. Un mensaje de encuentro está compuesto por el TempID, el modelo del teléfono y la potencia de transmisión con la que se ha hecho el intercambio (TxPower). Cada dispositivo también almacena el indicador de intensidad de la señal recibida (RSSI) y la marca temporal de la entrega del mensaje. Teniendo en cuenta que los TempID son generados y encriptados por el servidor, éstos no revelarán ninguna información personal del usuario de la aplicación. Por lo tanto, ambos usuarios de la aplicación tienen un registro simétrico del encuentro que se guarda en el almacenamiento local de sus respectivos teléfonos. El protocolo utiliza una lista negra temporal para evitar que un usuario registre contactos duplicados. Por lo tanto, una vez que un usuario recibe un mensaje de encuentro, la aplicación incluye automáticamente al remitente en la lista negra durante un breve período de tiempo.
3. Carga de encuentros: todos los registros de encuentros se almacenan localmente y no se cargan automáticamente en el servidor. Cuando se confirma que un paciente está infectado, las autoridades sanitarias preguntan si tienen instalada la aplicación, en cuyo caso se les pide que carguen su historial de encuentros a la autoridad sanitaria. Para proteger a los usuarios y al sistema de cargas fraudulentas, la autoridad de salud proporciona un código de autorización que se ingresa a través de la aplicación para obtener un token válido que se utiliza para transmitir los registros.
4. Procesamiento de la carga de encuentros: el servidor es el encargado de recorrer la lista de mensajes de encuentro, descifrando cada TempID con su clave secreta y, de esta manera, pudiendo identificar al usuario. El servidor utiliza los valores de TxPower y RSSI para determinar la distancia a la que se ha producido el encuentro. Estos datos de proximidad, junto con las marcas de tiempo, se utilizan para determinar el perfil de riesgo (cercanía y duración) del encuentro.

En definitiva, en la arquitectura centralizada el servidor central juega un papel clave en el desempeño de funcionalidades básicas como el almacenamiento de información personal de identificación (PII) encriptada, la generación de TempID anónimos, análisis de riesgo y notificaciones para contactos cercanos. Esta acumulación de responsabilidades ha generado numerosos debates sobre la privacidad. Se supone que el servidor es de confianza en esta arquitectura, y algunos países introducen estrictas normas de protección de la privacidad para salvaguardar el uso y el ciclo de vida de los datos recopilados.

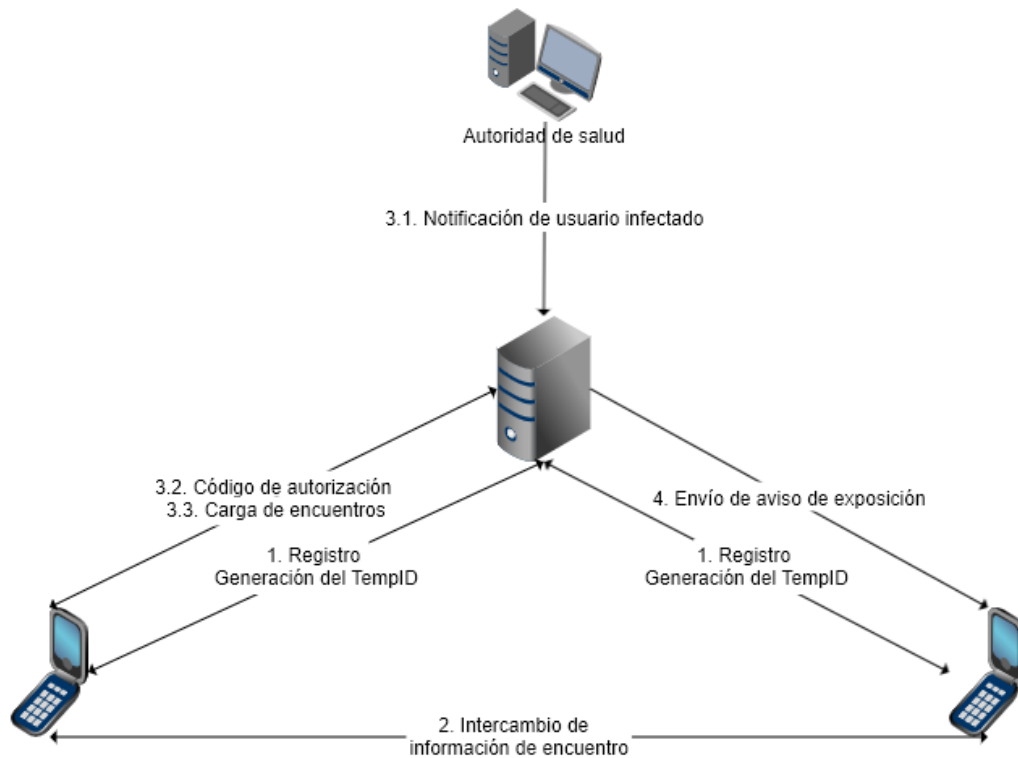


Ilustración 1. Arquitectura centralizada.

6.4. Arquitectura descentralizada

El enfoque de la arquitectura descentralizada [5], [7], analizada en base al protocolo Private Automated Contact Tracing (PACT) [9], propone que tanto los datos de contacto como los de ubicación se recopilen exclusivamente en los dispositivos de cada usuario, dejando al servidor con una participación mínima en el proceso de rastreo de contactos. Esto permite que los datos de contactos puedan ser compartidos por separado y de forma selectiva y voluntaria, solo cuando el ciudadano ha dado positivo en la prueba de COVID-19 y con un nivel de detalle que preserva la privacidad.

La idea de este enfoque es mejorar la privacidad del usuario mediante la generación de identificadores anónimos en los dispositivos del usuario (manteniendo en secreto las identidades reales del usuario de los otros usuarios, así como del servidor) y procesando las notificaciones de exposición en dispositivos individuales en lugar del servidor centralizado. A su vez, esto permite tanto el rastreo de contactos como la detección temprana de puntos críticos de brotes en una escala geográfica con una granularidad más fina. Esta arquitectura también es escalable a grandes poblaciones, ya que solo los datos de pacientes positivos deben manejarse a nivel central.

El enfoque descentralizado no requiere que los usuarios de la aplicación realicen un registro previo, evitando así el almacenamiento de cualquier PII en el servidor. Los dispositivos generan sus semillas de forma aleatoria, que se utilizan en combinación con la marca de tiempo para generar pseudónimos que preservan la privacidad con una vida útil muy corta de aproximadamente 1 minuto. Estas piezas de información se intercambian periódicamente con otros dispositivos que entran en lo que se considera un contacto cercano. Una vez que un

usuario recibe un diagnóstico positivo de COVID-19, puede ofrecerse como voluntario para cargar sus semillas y los registros de tiempo relevante en un servidor central. Esto contrasta con la arquitectura centralizada donde se carga la lista completa de mensajes de encuentros.

El servidor central solo actúa como un punto de encuentro, sometido a la tarea de transmitir las semillas de los usuarios infectados. Otros usuarios de la aplicación pueden descargar estas semillas para reconstruir la traza de contactos (usando las marcas temporales o registros de tiempo) que hayan sido enviados por los usuarios infectados. El servidor, así como otros usuarios, no pueden extraer ningún detalle de identificación con solo conocer las semillas. Solo los usuarios de la aplicación pueden realizar un análisis de riesgo para comprobar si han estado expuestos durante un tiempo suficiente. Esta búsqueda unidireccional restringe la funcionalidad del servidor y alivia algunos de los riesgos de privacidad.

Las fases de una solución mediante arquitectura descentralizada son:

1. Instalación: las aplicaciones que adoptan una arquitectura descentralizada no requieren, de forma obligatoria, que el usuario se registre una vez instale la aplicación. De modo que la aplicación se instala desde la tienda de aplicaciones correspondiente, desplegando un algoritmo de generación de semillas aleatorias que no se vincula de ninguna forma al teléfono móvil del usuario.
2. Generación de semillas e intercambio: tras la instalación de la aplicación, el algoritmo genera una semilla aleatoria de 256 bits cada hora que solo conoce el teléfono donde se ha creado. Esta semilla, junto con la hora actual, se utilizan posteriormente en una función pseudoaleatoria que genera los metadatos que se utilizan para el intercambio. Los metadatos no están vinculados a una persona ni a su teléfono, por lo que, en principio, son anónimos. La aplicación genera nuevos metadatos cada minuto y se transmiten cada pocos segundos a través del Bluetooth, de modo que la aplicación que escucha sea capaz de estimar el alcance de la exposición. Los receptores almacenan automáticamente estos metadatos junto con la marca temporal y el valor máximo de RSSI. A diferencia de la arquitectura centralizada donde los identificadores temporales se crean en el servidor, aquí las semillas y los metadatos se generan siempre en el dispositivo.
3. Carga de datos de encuentros: si a un usuario se le diagnostica como positivo, se le otorga un número de permiso único para autorizar la carga de todas las semillas usadas que están almacenadas localmente en su teléfono, así como la fecha de creación y de caducidad de estas semillas.
4. Proceso de rastreo de contactos: en el caso de la arquitectura descentralizada, el proceso de seguimiento de contactos lo realiza localmente el usuario de la aplicación en su dispositivo en lugar del servidor central. Para saber si ha estado en contacto con alguna persona diagnosticada, el usuario descarga la base de datos de exposición (o los fragmentos que se consideren relevantes) y verifica si algún metadato en sus registros de contactos "coincide" con alguna entrada en la base de datos. En caso de que eso suceda, esto indica que ha habido un contacto con una persona diagnosticada.

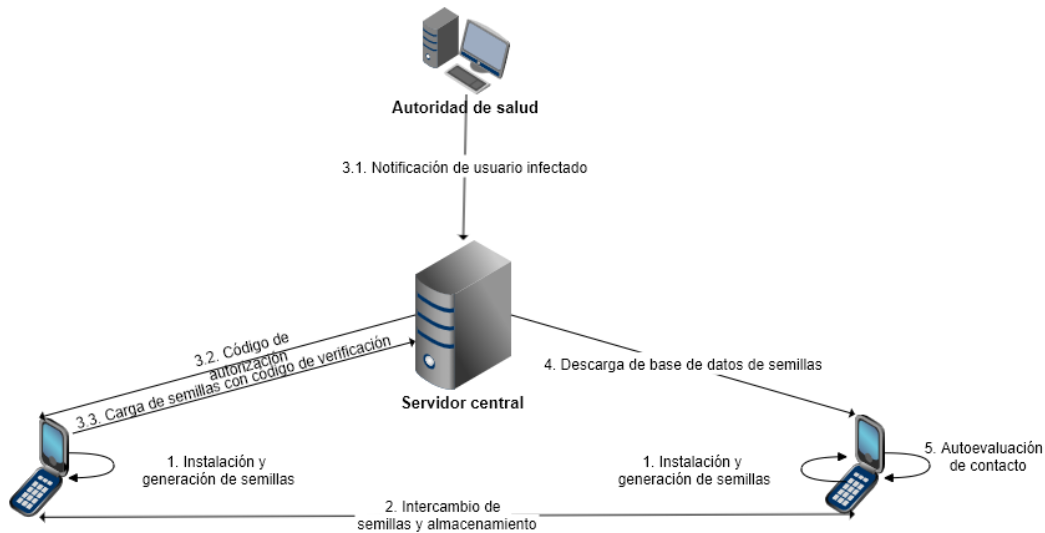


Ilustración 2: Arquitectura descentralizada.

6.5. Arquitectura híbrida

La arquitectura híbrida, adoptada en el protocolo Desire [10], propone que las tareas necesarias para realizar el rastreo y la notificación se dividan entre el servidor central y los dispositivos. Concretamente, plantea que la generación y la gestión de los TempID se gestione en los dispositivos para garantizar la privacidad (descentralizado), mientras que el análisis de riesgos y las notificaciones son responsabilidad del servidor centralizado. Hay tres razones principales por las que resulta interesante realizar el proceso de rastreo en el servidor, tal y como se menciona en [5]:

- i) En la arquitectura descentralizada, el servidor no tiene información sobre el número de usuarios en riesgo de contagio, puesto que son los dispositivos los que se encargan de realizar el análisis de riesgo sin tener en cuenta al servidor. De esta manera, se pierde la capacidad de obtener información estadística en el servidor, así como la posibilidad de ejecutar análisis de datos para identificar grupos de exposición.
- ii) El análisis de riesgo y las notificaciones son procesos sensibles que deberían manejarse por parte de las autoridades, teniendo en cuenta los recursos de infraestructura existentes y el estado de la pandemia.
- iii) La información de encuentros cargada de los usuarios infectados no se pone a disposición de los otros usuarios, sino que se conserva únicamente en el servidor. Esto es útil para evitar posibles ataques de anonimización de usuarios en la arquitectura descentralizada.

En este entorno, las aplicaciones interactúan con el sistema de la siguiente forma:

1. Inicialización: el usuario instala la aplicación desde las tiendas oficiales. Después, la aplicación se registra en el servidor generando un identificador permanente que almacenará junto con el resto de los identificadores generados. Esta información almacenada no está asociada, de ninguna manera, a una identidad particular.

2. Generación e intercambio de identificadores efímeros: cada terminal genera un identificador efímero (EphID) mediante el protocolo criptográfico Diffie-Hellman que se refresca, de forma genérica, cada 15 minutos. El dispositivo comienza a transmitir el EphID a través de Bluetooth hasta que el emisor lo haya recibido. En el proceso de intercambio, la aplicación genera dos tokens de encuentro privados (PET) que almacena para representar el encuentro.
3. Carga de datos de encuentros: una vez el usuario de positivo, y tras su consentimiento y autorización explícita, se realiza una carga en el servidor de los PET que se hayan generado en el terminal.
4. Solicitud del estado de exposición: la aplicación consulta el estado de exposición del usuario cargando en el servidor su lista de PET. El servidor realiza un análisis de riesgo haciendo coincidir los PET de la tabla de consulta con los PET cargados por el usuario infectado, evaluando si está en riesgo a través de los valores de tiempo y duración. El sistema se encarga de enviar una notificación al dispositivo de cualquier usuario que se encuentre en riesgo para que se comunique con la autoridad sanitaria.

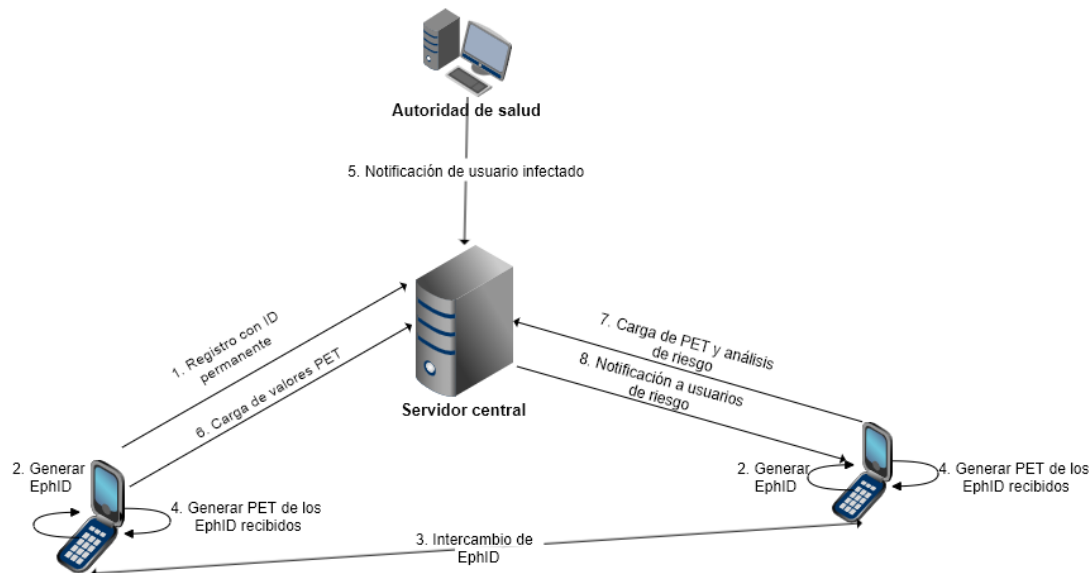


Ilustración 3: Arquitectura híbrida.

6.6. Conclusiones

Como recoge el artículo de PR Newswire [18], el 19 de abril de 2020 se hizo pública una carta firmada por casi 300 académicos advirtiendo que los sistemas centralizados pueden arriesgar la privacidad y sugirió que Apple y Google deberían considerar desarrollar una que use un sistema descentralizado y de confirmación única (opt-in).

De este modo, siendo el objetivo principal del proyecto la creación de una aplicación móvil multiplataforma que nos permita recopilar datos de forma anónima sobre las interacciones con otras aplicaciones, se ha optado por hacer uso de los beneficios que nos aporta la arquitectura descentralizada de cara a proteger la privacidad del usuario.

Con la adopción de una arquitectura descentralizada buscamos trasladar las funcionalidades centrales a los dispositivos móviles, dejando al servidor con una participación mínima en el proceso de rastreo de contactos. Como menciona la referencia [5], la idea es mejorar la privacidad manteniendo oculta la identidad real del usuario, generando identificadores únicos que sirvan para registrar los encuentros entre dispositivos, y procesar las notificaciones de exposición en cada terminal móvil en lugar del servidor centralizado.

Dado que el intercambio de información es el punto más sensible del sistema, se ha optado por no recoger ningún dato personal del usuario del terminal. Además de esto, la aplicación móvil generará identificadores únicos en periodos de 15 minutos que se utilizan para identificar los encuentros. Cada dispositivo actuará como emisor y receptor BLE y, en el momento en el que dos dispositivos entren en contacto, intercambiarán dichos identificadores únicos además de valores relativos al encuentro: distancia, RSSI, ubicación geográfica y marca temporal del encuentro. De este modo, los datos que se intercambian entre dos usuarios no revelan ninguna información personal por sí solos.

Una vez realizado un diagnóstico positivo por la autoridad de salud pertinente, el usuario recibe una OTP que deberá introducir en la aplicación para comunicar su positivo. En este instante se procede a una carga de los identificadores de encuentros que se encuentran en su base de datos local. Esta comunicación con el servidor central se realiza a través de una conexión HTTP mediante el protocolo cifrado TLS, haciendo uso del algoritmo criptográfico de Diffie-Hellman para Curvas elípticas que, como menciona la referencia [11], es una buena opción para sistemas integrados móviles al ser más rápido y requerir claves más cortas que el campo finito Diffie-Hellman.

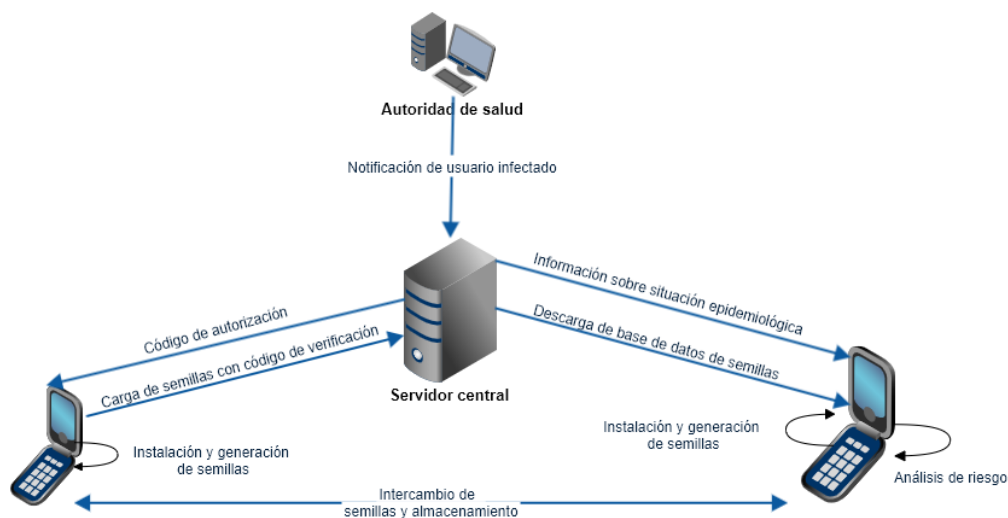


Ilustración 4: Arquitectura adoptada en Siteica.

Como ya se ha mencionado, cada dispositivo personal será el encargado de realizar el análisis de riesgo. Para tal objetivo, se descarga la lista de encuentros de riesgo del servidor y el dispositivo del usuario realiza el análisis de riesgo comparando los valores contra su lista de encuentros. Esta comunicación, de igual manera, se establece mediante HTTPS para asegurar la confidencialidad.

Otro de los objetivos que se persiguen es la posibilidad de proporcionar información a los usuarios sobre el entorno en el que se encuentran en relación a la infección, así como datos sobre la evolución epidemiológica. Para este hito se ha elegido el servidor central como elemento que provee información general, dado que es el punto de la arquitectura donde se almacena toda la información relativa a positivos. De igual manera que en el resto de comunicaciones entre los dispositivos y el servidor central, las conexiones se realizan a través de HTTP y protocolo cifrado TLS.

Para el objetivo de la creación de una solución que se despliegue en distintos dispositivos, nos apoyamos en Flutter como framework sobre el que realizar el desarrollo. Gracias al uso de este kit de desarrollo, podemos ejecutar nuestro código fuente de una manera ágil y sin dificultades tanto en dispositivos conectados vía USB o en los simuladores que nos ofrece Android Studio.

7. Análisis del sistema

En el presente apartado se recogen los requisitos del sistema a través de sus casos de uso que surgen tras las conclusiones del estado del arte.

7.1. Requisitos funcionales

Como parte del ciclo de vida, es indispensable realizar un análisis de los requisitos funcionales que describan lo que el sistema debe hacer, de forma que se satisfagan las necesidades de negocio. De este modo, en el siguiente apartado, se ha recogido una especificación de requisitos que nuestro software debe cumplir.

7.1.1. Alta en el sistema

RF-001	Alta
Descripción	Los usuarios deben poder darse de alta en el sistema para hacer uso de él.
Dependencias	-
Observaciones	Nuestro sistema hace especial hincapié en el anonimato, por lo que en el alta del usuario no se requerirán datos personales. Únicamente se solicita la provincia para que el servidor central pueda realizar análisis estadísticos.

Tabla 1: RF-001. Alta en el sistema.

7.1.2. Registro de encuentros

RF-002	Registro de encuentros
Descripción	El sistema debe recopilar datos de los posibles encuentros entre los usuarios de la aplicación.
Dependencias	RF-001
Observaciones	Para garantizar el anonimato, el sistema no intercambiará datos personales en el momento en el que se produzca un encuentro.

Tabla 2: RF-002. Registro de encuentros.

7.1.3. Notificación de usuario positivo

RF-003	Notificación de usuario positivo
Descripción	El sistema debe ser capaz de notificar al servidor central que un usuario ha dado positivo en la enfermedad infecciosa.
Dependencias	-
Observaciones	Serán solo los profesionales médicos los encargados de asignar un caso positivo y, de esta manera, informar al usuario de la aplicación.

Tabla 3: RF-003. Notificación de usuario positivo.

7.1.4. Carga de identificadores de encuentro

RF-004	Carga de identificadores de encuentro al servidor
Descripción	El sistema debe recopilar los encuentros de los últimos 14 días de la persona infectada y notificarlos al servidor central.
Dependencias	RF-003
Observaciones	La notificación de los encuentros se realiza tras el aviso de un positivo por parte del usuario de la aplicación.

Tabla 4: RF-004. Carga de identificadores de encuentro.

7.1.5. Análisis de riesgo

RF-005	Análisis de riesgo
Descripción	El sistema debe realizar un análisis de riesgo del usuario periódicamente en base a su base de datos de encuentros.
Dependencias	-
Observaciones	El análisis de riesgo se realiza sin intervención del usuario.

Tabla 5: RF-005. Análisis de riesgo.

7.1.6. Mapa de infecciones

RF-006	Mapa de infecciones
Descripción	El sistema debe mostrar un mapa que proporcione información sobre el entorno en el que se encuentran en relación a la infección.
Dependencias	-
Observaciones	-

Tabla 6: RF-006. Mapa de infecciones.

7.1.7. Información de evolución

RF-007	Información de evolución
Descripción	El sistema debe mostrar información actualizada sobre el estado de la evolución de la enfermedad.
Dependencias	-
Observaciones	-

Tabla 7: RF-007. Información de evolución.

7.2. Requisitos no funcionales

A continuación se recogen los atributos de calidad que la aplicación debe cumplir y que especifican las características del sistema.

RNF-001. El desarrollo multiplataforma se realiza con Flutter.

RNF-002. Uso del sistema de diseño Material Design con el objetivo de lograr un tiempo de aprendizaje corto de la aplicación.

RNF-003. Se requiere permisos para acceder a la ubicación del dispositivo.

RNF-004. Se requiere habilitar el Bluetooth para el uso completo de la aplicación.

RNF-005. La aplicación se desarrolla para Android e iOS.

RNF-006. Se utiliza inyección de dependencias como parte de la inversión de control.

RNF-007. Las bases de datos serán de tipo SQLite.

RNF-008. Se implementa un sistema de gestión de cambios de bases de datos, dotando al proyecto de una administración de cambios incrementales y reversibles y al control de versiones de esquemas de bases de datos relacionales.

RNF-009. No se recoge ningún dato personal del usuario con el objetivo de mantener el anonimato.

RNF-010. La aplicación necesita conectividad a la red para transmitir y recoger datos del servidor central.

RNF-011. Se establecen conexiones HTTP a través de TLS (algoritmo ECDH) para garantizar una conexión segura.

7.3. Casos de uso

En el siguiente apartado se describen los distintos casos de uso. Cada uno de ellos proporciona una descripción de cómo los usuarios realizan tareas en la aplicación. Describe, desde el punto de vista del usuario, el comportamiento de un sistema cuando responde a una solicitud. Cada caso de uso se representa como una secuencia de pasos simples, que comienzan con el objetivo del usuario y finalizan cuando se cumple ese objetivo.

7.3.1. Registro de usuario

CU-001	Registro de usuario
Versión	1.0 (25/02/2021)
Dependencias	-
Precondición	El usuario accede a la tienda de aplicaciones oficial, descarga la aplicación y realiza su instalación.
Descripción	El sistema genera un identificador único universal (UUID) por usuario y se notifica al servidor.
Secuencia normal	<ol style="list-style-type: none"> 1. El usuario descarga la aplicación de la tienda oficial de aplicaciones desde su dispositivo. 2. El usuario instala la aplicación. 3. El usuario acepta las condiciones de uso y privacidad. 4. El usuario selecciona su provincia. 5. El sistema genera un UUID para el usuario. 6. El UUID se almacena en la base de datos local. 7. El sistema notifica el UUID al servidor.
Postcondición	El usuario queda activado en el sistema.
Excepciones	<ol style="list-style-type: none"> 3. Si el usuario no acepta las condiciones de uso y privacidad, <ol style="list-style-type: none"> 3.1. El sistema deniega el uso de la aplicación al usuario.
Comentarios	Es fundamental informar sobre la política que se aplica al servicio.

Tabla 8: CU-001. Registro de usuario.

7.3.2. Generación de identificadores de encuentros

CU-002	Generación de identificadores de encuentros
Versión	1.0 (25/02/2021)
Dependencias	CU-001
Precondición	El usuario se ha dado de alta correctamente.
Descripción	El sistema genera, cada 15 minutos, un UUID utilizando la fecha, hora y dirección MAC del dispositivo.
Secuencia normal	<ol style="list-style-type: none"> 1. Usando la fecha, hora y dirección MAC del dispositivo, el sistema genera un UUID que se utilizará como identificador durante el intercambio en un encuentro. 2. Cada 15 minutos, el sistema genera un nuevo UUID de encuentro. 3. El sistema almacena el UUID en la BD local. 4. El sistema borra el UUID anterior.
Postcondición	Se genera un UUID que identifica al usuario en un momento específico para su posterior intercambio en el momento del encuentro.

Excepciones	-
Comentarios	Los UUID de encuentro ya caducados no es necesario almacenarlos puesto que no nos aportan información útil. Todos los identificadores útiles residen en la tabla donde se almacenan los encuentros entre dispositivos.

Tabla 9: CU-002. Generación de identificadores de encuentros.

7.3.3. Intercambio de identificadores de encuentros

CU-003	Intercambio de identificadores de encuentros
Versión	1.0 (25/02/2021)
Dependencias	CU-002
Precondición	El usuario ha activado el Bluetooth y los dispositivos que realizan el intercambio se encuentran a 2 metros o menos de distancia durante, al menos, 15 minutos. Ambos actúan como emisores y receptores al mismo tiempo.
Descripción	El sistema, una vez descubierto otro dispositivo cercano, hará un intercambio de UUID de encuentros entre ambos dispositivos y almacena una lista de los encuentros de los últimos 14 días.
Secuencia normal	<ol style="list-style-type: none"> 1. El dispositivo comienza con la emisión (broadcasting) y recepción a través de BLE. 2. El sistema registra ambos UUID en su base de datos local y almacena datos del encuentro (tiempo, distancia y geolocalización).
Postcondición	Se generan los registros correspondientes al encuentro en la base de datos local del usuario.
Excepciones	-
Comentarios	-

Tabla 10: CU-003. Intercambio de identificadores de encuentros.

7.3.4. Notificación de usuario positivo

CU-004	Notificación de usuario positivo
Versión	1.0 (25/02/2021)
Dependencias	
Precondición	El usuario enfermo da positivo y el usuario sanitario genera una contraseña de un solo uso (OTP) tras diagnosticar un positivo.
Descripción	El usuario recibe su OTP vía SMS y lo introduce para notificar que es un caso positivo.
Secuencia normal	<ol style="list-style-type: none"> 1. El usuario recibe su OTP en su terminal.

	<ol style="list-style-type: none"> 2. El usuario introduce su OTP en la aplicación. 3. El usuario recibe información sobre los datos que van a compartirse y consiente la carga de semillas al servidor.
Postcondición	El OTP queda deshabilitado para su posible posterior uso.
Excepciones	<ol style="list-style-type: none"> 3. Si el usuario no consiente la carga de semillas, <ol style="list-style-type: none"> 3.1. La aplicación muestra un aviso de que no se podrán notificar los posibles encuentros de riesgo.
Comentarios	Es importante en todo momento informar al usuario de lo que se transmite para evitar desconfianza frente a los posibles datos que se están tratando.

Tabla 11: CU-004. Notificación de usuario positivo.

7.3.5. Carga de semillas al servidor

CU-005	Carga de identificadores de encuentros al servidor
Versión	1.0 (25/02/2021)
Dependencias	CU-004
Precondición	El usuario introduce su OTP para notificar que es un caso positivo y admite la carga de UUID de encuentros al servidor.
Descripción	El sistema carga en el servidor la lista UUID de encuentros realizados en los últimos 14 días.
Secuencia normal	<ol style="list-style-type: none"> 1. El sistema establece una conexión HTTP a través del protocolo cifrado TLS. 2. A través de API REST, el sistema envía los registros de la tabla de encuentros de los últimos 14 días que estén pendientes de transmitir.
Postcondición	La carga finaliza solamente cuando se han transmitido todos los registros pendientes.
Excepciones	-
Comentarios	La información transmitida no contiene datos personales de ningún tipo. Solo se transfieren identificadores únicos de encuentros (UUIDs generados a través de la fecha, hora y MAC del dispositivo) y la ubicación de los encuentros.

Tabla 12: CU-005. Carga de identificadores de encuentros al servidor.

7.3.6. Análisis de riesgo

CU-006	Análisis de riesgo
Versión	1.0 (25/02/2021)
Dependencias	CU-005

Precondición	El proceso se ejecuta 24 horas después del último análisis.
Descripción	Se descarga la lista de encuentros de riesgo del servidor y el terminal realiza el análisis de riesgo comparando los valores contra su lista de encuentros.
Secuencia normal	<ol style="list-style-type: none"> 1. El dispositivo, una vez al día, descarga la lista de encuentros positivos del servidor. 2. El dispositivo compara la lista descargada con la lista de UUID de encuentros de su base de datos local. 3. Si se encuentra una correspondencia, <ol style="list-style-type: none"> 3.1. Se emite un aviso de posible exposición a la enfermedad.
Postcondición	El sistema refleja la posibilidad de exposición al virus y muestra una serie de recomendaciones.
Excepciones	-
Comentarios	-

Tabla 13: CU-006. Análisis de riesgo.

7.3.7. Mapa de infección

CU-007	Mapa de infección
Versión	1.0 (25/02/2021)
Dependencias	CU-006
Precondición	El usuario navega a la opción del menú indicada como “Mapa”.
Descripción	Puesto que el sistema solicita la lista de identificadores de encuentros de riesgo al servidor de manera periódica (CU-006), el sistema hace uso de estos datos para mostrar un mapa de infecciones con datos cercanos.
Secuencia normal	<ol style="list-style-type: none"> 1. El usuario accede al mapa de infección. 2. El sistema extrae la lista de encuentros de riesgo, cercanos a su posición, de la base de datos local. 3. La aplicación muestra un mapa de infección con los encuentros de riesgo cercanos.
Postcondición	-
Excepciones	<ol style="list-style-type: none"> 3. Si el terminal no es capaz de conectarse a Internet, <ol style="list-style-type: none"> 3.1. Se muestran los datos almacenados en la BD local.
Comentarios	-

Tabla 14: CU-007. Mapa de infección.

7.3.8. Información de evolución

CU-008	Información de evolución
Versión	1.0 (25/02/2021)
Dependencias	-
Precondición	El usuario navega a la opción del menú indicada como “Evolución”.
Descripción	El sistema muestra la evolución de las infecciones, mostrando número de positivos en los últimos 14 días en una gráfica temporal. También se muestran los datos globales que el servidor central se encarga de mantener.
Secuencia normal	<ol style="list-style-type: none"> 1. El usuario accede a la información sobre la evolución. 2. El sistema solicita los datos de evolución que se transmiten desde el servidor vía conexión HTTP a través del protocolo cifrado TLS. 3. Se almacenan los datos actualizados sobre la evolución en la base de datos local. 4. La aplicación muestra una gráfica temporal y datos globales.
Postcondición	-
Excepciones	<ol style="list-style-type: none"> 2. Si los registros de la BD local han sido actualizados en las últimas 24 horas, <ol style="list-style-type: none"> 2.1. No se solicitan datos al servidor. Así evitamos la descarga constante de datos. 3. Si el terminal no es capaz de conectarse a Internet, <ol style="list-style-type: none"> 3.1. Se muestran los datos almacenados en la BD local, indicando hora de la última actualización.
Comentarios	-

Tabla 15: CU-008. Información de evolución.

7.3.9. Diagrama de casos de uso

A continuación se representa, mediante un diagrama de casos de uso, la relación entre los actores y los requisitos del sistema.

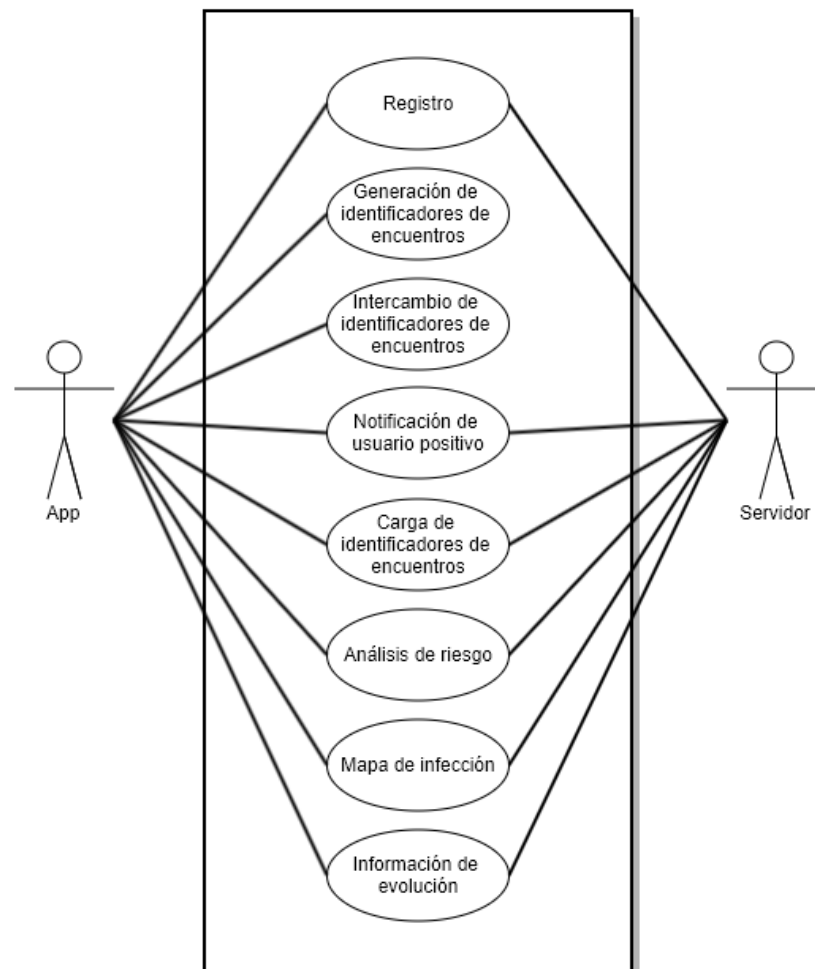


Ilustración 5: Diagrama de casos de uso.

8. Tecnologías empleadas

Una vez completo el análisis de requisitos y casos de uso, se han examinado las distintas tecnologías para alcanzar los objetivos que se persiguen. De este modo, en el siguiente apartado se detallan y describen las tecnologías principales, además de otros recursos, que se han empleado para la implementación del presente proyecto.

8.1. SQLite

El siguiente objetivo es la elección de un motor de base de datos ligero y óptimo para el manejo desde un terminal móvil, por lo que se ha optado por el uso de SQLite.

SQLite es una biblioteca de dominio público desarrollada en C que implementa un motor de base de datos SQL pequeño, rápido, autónomo y de alta confiabilidad que implementa todas las funciones de un motor de base de datos transaccional, siendo totalmente compatible con ACID. SQLite es el motor de base de datos más utilizado del mundo. La gran ventaja de SQLite es que ya está integrado en todos los teléfonos móviles, con los beneficios que eso conlleva.

SQLite es una biblioteca que implementa un motor de base de datos transaccional SQL autónomo, sin servidor y sin necesidad de configuración. El código de SQLite es de dominio público y, por lo tanto, es gratuito para cualquier propósito, comercial o privado. SQLite es, de hecho, la base de datos más implementada en el mundo en proyectos de todo tipo.

A diferencia de la mayoría de bases de datos SQL, SQLite no tiene un proceso de servidor separado, sino que es un motor de base de datos SQL embebido. SQLite lee y escribe directamente en archivos de disco. De hecho, una base de datos SQL completa con múltiples tablas, índices, disparadores y vistas está contenida en un solo archivo físico. El formato de archivo de la base de datos es multiplataforma, por lo que una base de datos puede copiarse entre sistemas de 32 y 64 bits o entre arquitecturas big-endian y little-endian.

Otras de sus grandes ventajas es que, aún con todas las funciones habilitadas, el tamaño de la biblioteca puede ser inferior a 600 KB, según la plataforma de destino y la configuración de optimización del compilador. Y pese a que SQLite generalmente se ejecuta más rápido cuanto más memoria le da, el rendimiento suele ser bastante bueno incluso en entornos con poca memoria. Esta característica hace que sea un motor ideal para dispositivos móviles.

SQLite también es conocido por tratarse de un sistema altamente fiable debido a que se realizan pruebas exhaustivas antes de cada lanzamiento. La mayor parte del código fuente de la librería está dedicado exclusivamente a procesos de prueba y verificación, alcanzando una cobertura de un 100%.

Debido a estas características, se ha elegido SQLite como motor de base de datos en el que almacenar los datos relativos a encuentros, así como para la información de encuentros en los que un usuario haya dado positivo. Como se explica más adelante, se ha optado por el uso del componente Sqflite de Flutter para el manejo y creación de bases de datos relacionales.

8.2. Bluetooth de baja energía

Una vez escogido el motor de base de datos donde almacenar los datos relativos a la situación y evolución de la enfermedad infecciosa, así como los datos de los encuentros cercanos, es turno de analizar qué sistema es el idóneo para transmitir información entre dispositivos móviles cercanos.

La tecnología conocida como Bluetooth Low Energy (BLE) es una variante del estándar inalámbrico Bluetooth diseñado para un bajo consumo de energía. Fue introducido por el Bluetooth Special Interest Group (Bluetooth SIG) en diciembre de 2009 como parte de la especificación Bluetooth 4.0, con la intención de dar solución a aquellas aplicaciones que no requieren una señal fuerte y, de este modo, obtener un ahorro de energía considerable. Algunos ejemplos donde la tecnología encaja a la perfección son, tal y como menciona la referencia [13]: los ya conocidos rastreadores de actividad física, algunos electrodomésticos inteligentes o sensores de proximidad.

La referencia [12] explica cómo los llamados sensores de proximidad, conocidos como Beacon en la plataforma de Google o iBeacon en Apple, solo necesitan realizar una transferencia periódica de pequeñas cantidades de datos de corto alcance, tarea para la que la tecnología BLE está especialmente diseñada.

De este modo, la tecnología BLE nos ofrece una solución óptima para realizar los intercambios de semillas tan pronto se realice un encuentro. Para este cometido, convertimos cada dispositivo en un emisor BLE y, a su vez, en un cliente.

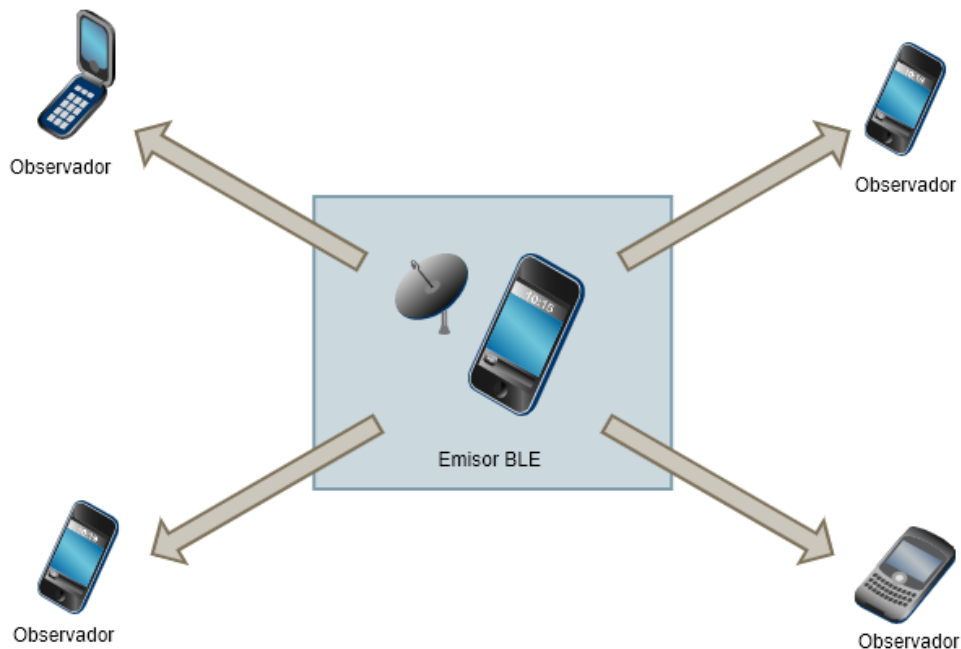


Ilustración 6: Topología broadcasting en BLE.

Un dispositivo BLE puede comunicarse con el resto de terminales de dos formas: por transmisión (broadcasting) o por conexiones. Cada mecanismo tiene sus propias ventajas y limitaciones, y ambos están sujetos a las pautas establecidas por el Perfil de Acceso Genérico

(GAP). Haciendo uso del modo de transmisión conocido como broadcasting, es posible enviar datos a cualquier dispositivo configurado en modo de escaneo o receptor en un rango de escucha. Como se muestra en la Ilustración 6, este mecanismo permite enviar datos en un solo sentido a cualquier mecanismo que sea capaz de recoger los datos transmitidos.

Esta topología define dos roles separados:

- Emisor (broadcaster): envía periódicamente paquetes no conectables a cualquier dispositivo que desee recibirlos.
- Observador (observer): escanea repetidamente las frecuencias preestablecidas para recibir cualquier paquete no conectable que se esté emitiendo actualmente.

Es importante recordar que, en lo que se refiera a la comunicación a través de BLE, esta estructura es la única que nos permite que un dispositivo transmita datos a más de un dispositivo a la vez [14]. Esto nos garantiza, en el proyecto actual, poder realizar una comunicación conjunta entre los distintos terminales que vayan realizando encuentros sin necesidad de emparejamientos.

El paquete de datos estándar permitido para el emisor contiene una carga útil de 31 bytes que se utilizan para incluir datos que describen al broadcaster y sus capacidades, además de admitir la inclusión de cualquier información personalizada que se desee transmitir a los observadores. Si esta carga útil estándar no es suficientemente, BLE también admite una carga secundaria opcional (conocida como Respuesta de Escaneo o Scan Response) que permite a los dispositivos que detectan a un broadcaster solicitar un segundo marco de datos con otra carga útil de 31 bytes, llegando a un total de 62 bytes.

Así, se ha alcanzado la configuración de esta topología BLE mediante el paquete beacon_broadcast con el que convertir el terminal en un emisor BLE, mientras que con el plugin beacons_plugin conseguimos que cada terminal sea capaz de escanear emisores y así leer los marcos de datos que transmiten.

8.3. Conexión HTTPS con el servidor central

Pese a que el desarrollo de los servicios que provee el servidor central no pertenecen al alcance del presente proyecto, sí es conveniente tener en cuenta cómo han de realizarse las comunicaciones entre servidor y aplicación móvil.

Como se ha mencionado anteriormente, existe una comunicación directa entre los servidores centrales y las aplicaciones móviles de cara a realizar la carga de identificadores (o semillas) de los encuentros que hayan dado positivo, así como para obtener información general sobre el estado actual de la infección.

Para este objetivo se plantea una conexión HTTP a través del protocolo SSL, que nos asegura el establecimiento de conexiones seguras a través de Internet mediante el uso de certificados digitales.

En [11] se explica cómo el Intercambio de Claves Diffie-Hellman para Curvas elípticas (ECDH) es una buena opción para sistemas integrados móviles porque es más rápido y requiere claves más cortas que el campo finito Diffie-Hellman. Por este motivo y como TLS nos ofrece

soporte para distintos algoritmos de intercambio de claves, se ha optado por el intercambio de claves ECDH como parte de la conexión segura de SITEICA.

El protocolo de Intercambio de Claves Diffie-Hellman para Curvas elípticas (ECDH) es un esquema de establecimiento de claves anónimo que permite a dos participantes, cada uno de ellos poseedor de un par de claves pública-privada de curvas elípticas, acordar un secreto compartido a través de un canal inseguro. ECDH es muy similar al algoritmo clásico de intercambio de claves de Diffie-Hellman (DHKE), con la diferencia de que usa la multiplicación de puntos de **criptografía de curva elíptica** (ECC) en lugar de exponenciaciones modulares. ECDH se basa en la siguiente propiedad de los puntos de las curvas elípticas:

$$(a * G) * b = (b * G) * a$$

Si tenemos dos números secretos **a** y **b** (dos claves privadas, pertenecientes a Alice y Bob) y una curva elíptica ECC con punto generador G, podemos intercambiar sobre un canal inseguro los valores $(a * G)$ y $(b * G)$ (las claves públicas de Alice y Bob) y luego podemos derivar un secreto compartido: $\text{secreto} = (a * G) * b = (b * G) * a$. La ecuación anterior se traduce en:

Clave Pública Alice * Clave Privada Bob = Clave Pública Bob * Clave Privada Alice = secreto compartido

El algoritmo lleva a cabo las siguientes operaciones:

1. Alice genera un par de claves ECC aleatorias: {clavePrivAlice, clavePubAlice = clavePrivAlice * G}
2. Bob genera un par de claves ECC aleatorias: {clavePrivBob, clavePubBob = clavePrivBob * G}
3. Alice y Bob intercambian sus claves públicas a través del canal inseguro
4. Alice calcula la clave compartida = clavePubBob * clavePrivAlice
5. Bob calcula la clave compartida = clavePubAlice * clavePrivBob
6. Ahora tanto Alice como Bob tienen la misma clave compartida == clavePubBob * clavePrivAlice == clavePubAlice * clavePrivBob

8.4. Flutter

Flutter, desarrollado por Google, es un conjunto de herramientas de interfaz de usuario orientadas al desarrollo de aplicaciones para dispositivos móviles, web y escritorio. El soporte para dispositivos móviles permite compilar el código fuente de Flutter a aplicaciones nativas de iOS y Android, así como en aplicaciones nativas de Windows, macOS o Linux en su soporte para escritorio.

Una de las grandes diferencias de Flutter con respecto a otros frameworks para construir aplicaciones móviles es que Flutter no usa componentes del fabricante original, es decir, no hace de puente. En cambio, Flutter usa su propio motor de renderizado de alto rendimiento para dibujar los distintos componentes de la interfaz de usuario., aportando un alto rendimiento a las aplicaciones.

Además, Flutter es diferente porque solo tiene una pequeña capa de código C/C++. Flutter implementa la mayor parte de su sistema (gestos, animaciones, framework, widgets, etc.) en Dart (un lenguaje moderno, de sintaxis breve y orientado a objetos) al cual los desarrolladores pueden acercarse fácilmente para leer, cambiar, reemplazar o suprimir. Esto da a los desarrolladores un enorme control sobre el sistema, así como también reduce significativamente la curva de aprendizaje y la accesibilidad para la mayoría del sistema.

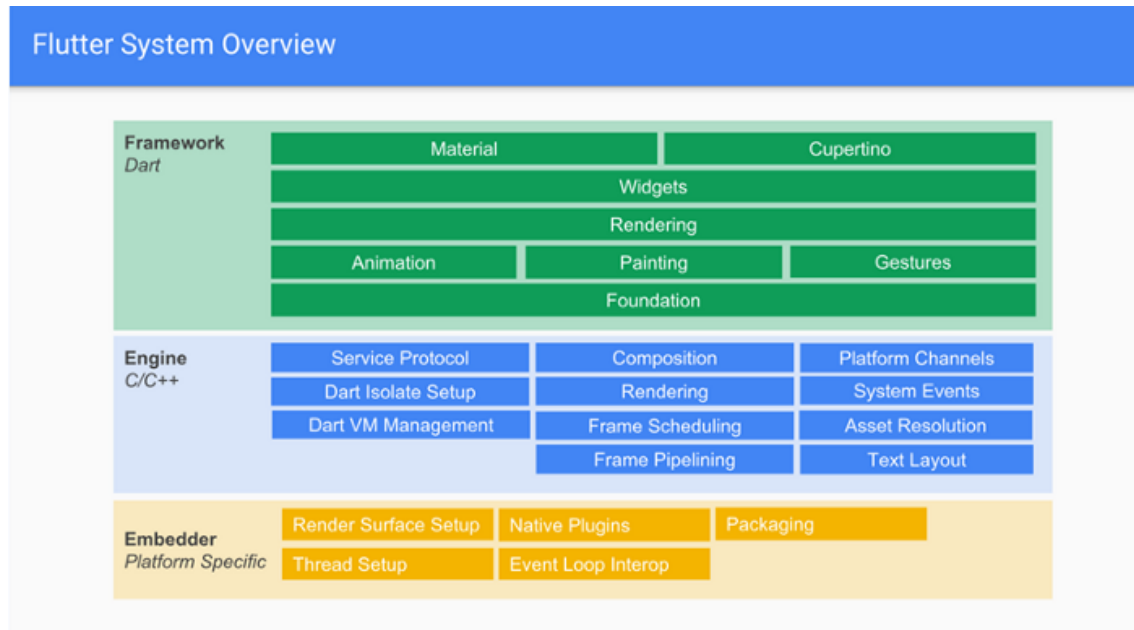


Ilustración 7: Arquitectura de Flutter. Fuente: <https://flutter.dev/>.

Entre los beneficios que nos aporta el uso de Flutter como framework de desarrollo móvil se encuentran los siguientes:

- **Multiplataforma:** Flutter nos permite que se pueda utilizar el mismo código fuente para crear una aplicación en iOS y Android, convirtiéndolo así en un SDK multiplataforma.
- **Desarrollo ágil:** además de la facilidad de aprendizaje de su sintaxis, Flutter incorpora la funcionalidad conocida como hot reload que permite experimentar de forma rápida, construir interfaces de usuario, arreglar errores y añadir funcionalidades. Hot reload trabaja inyectando ficheros de código fuente actualizados en la Máquina Virtual (VM) Dart en ejecución [20].
- **Rendimiento:** el SDK de Google ofrece un rendimiento excepcional por dos razones. Primero, como se ha mencionado, gran parte de su sistema usa Dart que se compila en código nativo. En segundo lugar, Flutter tiene sus propios widgets, por lo que no es necesario acceder a los componentes del fabricante original (OEM). Como resultado, hay menos comunicación entre la aplicación y la plataforma. Estas dos características de Flutter garantizan tiempos de inicio rápidos de la aplicación y menos problemas de rendimiento en general.
- **Compatibilidad:** puesto que Flutter implementa sus propios widgets, los problemas de compatibilidad son poco frecuentes. Esto también permite que el comportamiento y el

aspecto de las aplicaciones se mantengan en las distintas versiones de Sistemas Operativos (SO). Del mismo modo, esto permite asegurar que funcionen en versiones futuras de los SO.

Debido a estas características se ha optado por Flutter como SDK para el desarrollo de la aplicación móvil multiplataforma de SITEICA. Esta decisión nos aportará la posibilidad de realizar un único desarrollo ágil que podrá ser compilado tanto para iOS como Android, así como una experiencia de usuario de alta calidad y un gran rendimiento.

8.5. Plugins de Flutter

Flutter tiene soporte para el uso de paquetes compartidos, publicados en el repositorio Pub. Estos paquetes hacen posibles múltiples casos de uso, como veremos a continuación.

8.5.1. Beacon Broadcast

Se ha utilizado el complemento Beacon Broadcast [22] para convertir cada terminal en un emisor BLE y conseguir la topología broadcasting, tal y como se ha analizado en la sección 7.4.1., de forma que todos los dispositivos emiten información y la reciben. Este plugin nos permite configurar la emisión, además de ajustar datos como el UUID con el que transmitiremos o incluso la potencia de transmisión en caso de querer acotarla.

De este modo, se ha configurado la aplicación para funcionar como un emisor que transmite paquetes de datos de forma periódica. Más concretamente, se emite el identificador único que utilizamos para gestionar y almacenar los encuentros de manera inequívoca.

8.5.2. Beacons Plugin

El plugin Beacons [27], junto con el anteriormente comentado, ha sido utilizado para conseguir en conjunto la topología BLE requerida para realizar de manera efectiva el intercambio de identificadores cada vez que ocurre un encuentro entre usuarios.

Con este plugin se ha logrado configurar cada terminal como un observador BLE, encargado de escanear repetidamente las frecuencias preestablecidas para recibir cualquier paquete de datos no conectable que se esté emitiendo.

8.5.3. Flutter Map

Este paquete Flutter Map [26] es una implementación de la librería Leaflet en Dart para su utilización en aplicación de Flutter. Este plugin nos permite configurar el proveedor de mapa, pudiendo escoger entre Open Street Map o Azure Maps. En el actual proyecto se ha optado por la utilización de Open Street Map como proveedor de uso libre bajo licencia abierta.

La librería también nos ofrece la posibilidad de ubicar la cuadrícula del mapa en un punto geográfico concreto, añadir sobre él marcadores con información -que se han utilizado para incluir información relativa a la situación de la enfermedad sobre el mapa- y personalizar los sistemas de coordenadas.

8.5.4. Geolocator

Un complemento de geolocalización de Flutter, llamado Geolocator [24], que ofrece acceso a servicios de ubicación específicos de la plataforma (FusedLocationProviderClient o, si no está disponible, LocationManager en Android y CLLocationManager en iOS).

Gracias a este paquete podemos conocer la ubicación del usuario para situarlo en el mapa o para registrar la geolocalización (latitud y longitud) del momento en el que se realiza un encuentro entre usuarios.

8.5.5. Sqflite

El complemento Sqflite [23] proporciona clases y funciones para la interacción con bases de datos SQLite. Incluye soporte de transacciones y lotes, gestión automática de versiones durante la apertura y la capacidad para operar sobre la base de datos en un hilo en segundo plano tanto en iOS como en Android.

8.5.6. Sqflite Migration Service

Usado en conjunto con Sqflite, este complemento [28] nos aporta la capacidad de gestionar las migraciones de la base de datos de una forma ágil y cómoda. La idea detrás de cualquier herramienta de migración de bases de datos es la de mejorar la administración de éstas a través de los distintos entornos de distribución.

Para lograr dicho objetivo, Sqflite Migration Service propone la creación de archivos de migración con extensión .sql que cumplan una convención de nombre específica (“[schema_version_number]_[migration_name].sql”) y, tras esto, inicializar el sistema de migraciones en el punto de entrada de la aplicación. De esta forma conseguimos que en cada arranque del sistema se comprueben qué migraciones han sido ejecutadas y cuáles no, en cuyo caso se ejecutarán sin intervención humana.

Esto nos garantiza que siempre tengamos bajo control el estado de la base de datos en la máquina actual, asegurar que los scripts se hayan aplicado, que todo el equipo de desarrollo trabaje bajo un mismo marco y, en definitiva, mantener el sistema sincronizado.

8.5.7. Injector

Injector [25] es una librería de inyección de dependencias (DI) para Dart. Internamente, el inyector es un singleton que almacena instancias y builders en una estructura de tipo Map. Se ha optado por un paquete que nos ofrezca un mecanismo sencillo de DI con el objetivo de conseguir beneficios tan importantes como implementaciones desacopladas, clases mucho más reutilizables y, sobre todo, un código fuente más organizado.

8.5.8. Json Serializable

Con el objetivo de almacenar y obtener fácilmente datos estructurados de la base de datos local se ha optado por el uso de la librería json_serializable [29]. Este complemento nos permite generar, de forma automática, los métodos necesarios para serializar y deserializar estructuras de datos.

Para generar los métodos correspondientes de serialización y deserialización basta con incluir la anotación `@JsonSerializable` en la clase correspondiente. A continuación se ejecuta el siguiente comando en la consola:

```
> pub run build_runner build
```

Dicho comando nos genera, por cada clase, un archivo equivalente con la extensión “.g.dart” donde los métodos correspondientes para realizar la conversión de datos.

8.5.9. UUID Enhanced

El complemento `uuid_enhanced` [30] nos ofrece una librería de clases sencilla para la creación de identificadores únicos universales (UUID), un número de 128 bits usado para identificar objetos de manera única, que nos permite su creación en las siguientes versiones:

- Versión 1: `Uuid.fromTime()` nos permite generar un UUID a partir de la marca temporal.
- Versión 4: nos permite generar un UUID de forma aleatoria con `Uuid.randomUUID()`.
- Versión 5: esta versión se basa en la realización de un hasing SHA-1 de un espacio de nombres y un nombre específico con el método `Uuid.fromName('www.uned.es', namespace: Uuid.NAMESPACE_URL)`.

Esta librería se ha utilizado para la generación de identificadores únicos de los encuentros, de modo que el sistema intercambie información que no revele ningún dato personal, así como para identificar de manera única al usuario de la aplicación.

8.5.10. Intro Slider

La librería `intro_slider` [33] nos facilita la creación de un carrusel de diapositivas. Este apoyo visual se ha utilizado como parte de la introducción que recibe el usuario a la aplicación la primera vez que hace uso de ella, aportando y atractivo en su uso.

8.5.11. Bluetooth Enable

Es interesante poder ofrecer la posibilidad al usuario de activar el Bluetooth directamente desde el código al ser un requisito indispensable. Para ello, el plugin `bluetooth_enable` [34] nos permite solicitar la activación dentro del propio flujo de la aplicación, dando la posibilidad de personalizar el diálogo de alerta que le muestra al usuario.

8.5.12. Charts Flutter

Flutter no dispone de ninguna clase que permita dibujar gráficas en pantalla. Por esta razón, se ha optado por el uso de la librería `Charts Flutter` [35] que permite representar gráficas de líneas, de barras y circulares.

8.6. Usabilidad y experiencia de usuario

Como parte de la Ingeniería de Factores Humanos, se ha decidido basar los diseños de la aplicación en un conjunto de principios y directrices específicas. Esto ayuda a asegurar una

mejor usabilidad a través del uso de guías de estilo mediante la coherencia y estabilidad que estas imponen.

8.6.1. Google Design

Se han utilizado las guías de estilo y diseño de Google recogidas en lo que se conoce como Google Design. A lo largo de una serie de artículos presentados a modo de casos de estudio y guías prácticas en la plataforma Web de Google Design, se presentan las bases con las que desarrollar una experiencia de usuario acorde a los estilos de Google, independientemente de la plataforma y el tamaño del dispositivo.

Este portal recoge una gran cantidad de artículos sobre cómo diseñar para lograr una accesibilidad global, recomendaciones sobre cómo realizar análisis de diseño con la intención de obtener retroalimentación constante y, en definitiva, guías con las que lograr que nuestras aplicaciones tengan un *look & feel* común sea la plataforma que sea.

Google, además, ofrece un portal con información sobre las investigaciones relacionadas en el ámbito de la disciplina de Interacción Persona-Ordenador (IPO). Aquí se recogen todas las publicaciones realizadas sobre esta disciplina, demostrando cómo sus investigaciones han contribuido a los diseños de sus aplicaciones más utilizadas como Gmail, Docs, YouTube o Maps, entre otras. El resultado más claro de sus investigaciones se puede ver de forma completa en Material Design, como veremos a continuación.

8.6.2. Material Design

Material Design es el sistema de diseño open-source de Google orientado a ayudar a los equipos a desarrollar aplicaciones con una experiencia de usuario de calidad en Android, iOS, Flutter y Web sobre el que se ha trabajado. Se presentó por primera vez en 2014 y comenzó a aplicarse, en los años siguientes, en todos los productos de Google, consiguiendo así una experiencia homogénea en todas las plataformas.

Material Design se apoya en tres principios:

1. El diseño debería ser una metáfora del mundo real. Esto quiere decir que elementos como los botones deberían verse como se verían en la vida real, utilizando sombras y animaciones que nos ayuden a identificar el objeto en pantalla y qué interacción puede esperar el usuario final.
2. Material Design se guía por los métodos de diseño de impresión. El uso de las cuadrículas, la tipografía, el espacio, el color o las imágenes nos ayudan a crear una jerarquía clara en pantalla.
3. El uso del movimiento y las animaciones para proporcionar retroalimentación a las acciones del usuario.

La guía de estilo de Google nos ofrece, además de las reglas fundamentales de disposición de elementos, navegación, colores o tipografía, una biblioteca completa de componentes. Estos componentes son bloques con los que construir interfaces de usuario que ya incluyen un sistema de estados con los que comunicar acciones como selección, arrastrar y soltar, errores o deshabilitar. Estos componentes están disponibles para Android, iOS, Flutter y Web, contando con una amplia colección de ejemplos.

Por otra parte, Material pone en valor todos los principios de accesibilidad y usabilidad recogidos en los distintos casos de estudio y artículos de Google Design, apoyándose en los siguientes principios:

- Claridad: debemos ayudar al usuario a navegar diseñando interfaces claras.
- Robustez: el diseño debe acomodarse a una gran cantidad de tipos de usuario.
- Específico: admitir distintas tecnologías de asistencia.

8.7. Arquitectura tecnológica

A continuación se han desgranado los distintos casos de uso y las tecnologías que han intervenido en cada uno de ellos para poder llevarse a cabo.

Con el objetivo de satisfacer el requisito **RNF-001** se ha escogido Flutter como framework de desarrollo de aplicaciones móviles puesto que, como se ha mencionado, es el conjunto de herramientas que ha permitido que la aplicación pueda ser instalada tanto en iOS como en Android. Además, los complementos de Flutter han permitido extender sus funcionalidades y cumplir así las necesidades de distintos casos de uso, como veremos a continuación.

El proceso comienza con la instalación de la aplicación, el registro del usuario y la notificación al servidor central tal y como se describe en el **CU-001**. La primera vez que el usuario hace uso de la aplicación recibirá información relacionada con el sistema, para lo cual se ha hecho uso de la librería `intro_slider` que nos ha simplificado la creación de un carrusel de diapositivas intuitivo. El siguiente paso ha sido identificar de manera única al usuario a través del plugin `uuid_enhanced`, apoyándonos en los métodos que ofrece la librería para la creación de identificadores únicos universales. Finalmente, tras la selección de una provincia, el usuario se almacena en la base de datos haciendo uso de las clases y funciones que nos provee el complemento `Sqflite` y se notifica al servidor central realizando una conexión HTTP a través del protocolo SSL, tal y como se menciona en el apartado 7.5.

Para el **CU-002**, donde se realiza la generación de identificadores de encuentros, se ha realizado un proceso que, cada 15 minutos, genera un nuevo identificador de encuentro haciendo uso de `uuid_enhanced` y lo almacena en la base de datos del usuario gracias a `Sqflite`.

La tecnología BLE y la topología por transmisión ha permitido el intercambio de identificadores de encuentros, tal y como se determina en el **CU-003**. Para alcanzar dicha topología se han utilizado dos complementos de Flutter: `Beacon Broadcast` y `Beacons Plugin`. El primero de ellos ha permitido que el terminal actúe como un emisor, enviando de forma periódica su identificador de encuentro actual generado en el **CU-002**. El segundo complemento se ha utilizado para configurar el dispositivo como un observador BLE de forma que sea capaz de recoger los UUID de encuentro de otros dispositivos cercanos y almacenar su información en la BD.

En el momento en el que se produce un encuentro, se ha empleado el complemento `Geolocator` para obtener la geolocalización en dicho instante y así almacenar dicha información.

En el **CU-004**, donde se realiza la notificación de un usuario positivo, la autoridad de salud hace llegar un OTP que el usuario debe usar en la aplicación. En este proceso se también solicita la fecha del diagnóstico y el consentimiento informado para realizar la carga de los encuentros que se hayan realizado en dicho dispositivo y que se recuperan mediante consultas Sqflite. La carga de identificadores de encuentros al servidor central, **CU-005**, se realiza siempre a través de una conexión cifrada HTTPS.

La aplicación de Flutter realiza, también de forma periódica, un análisis de riesgo siguiendo los criterios del **CU-006**. Se establece una conexión HTTPS al servidor central y este responde con la lista de encuentros positivos de los últimos 14 días que se almacenan en la BD local mediante sentencias Sqflite. Entonces, el dispositivo realiza una búsqueda para encontrar alguna correspondencia entre sus UUID de encuentro y aquellos registros recién descargados que ya se consideran de riesgo. En caso de encontrar un identificador de encuentro propio dentro de la lista de riesgo, se le notifica a través de la aplicación los posibles riesgos.

A partir de los datos de riesgo y su geolocalización, se ha realizado un mapa de infección como se describe en el **CU-007**. Se ha utilizado el paquete Flutter Map para mostrar el mapa donde se encuentra el usuario y añadir marcadores sobre él mostrando los encuentros de riesgo cercanos a su posición.

En cuanto a la información sobre la evolución de la enfermedad descrita en el **CU-008**, se ha utilizado la librería FL Chart para dibujar una gráfica temporal que nos permita visualizar el número de positivos en los últimos 14 días.

A través del siguiente diagrama se muestra, sobre la arquitectura escogida para el desarrollo del proyecto, en qué puntos intervienen algunas de estas tecnologías.

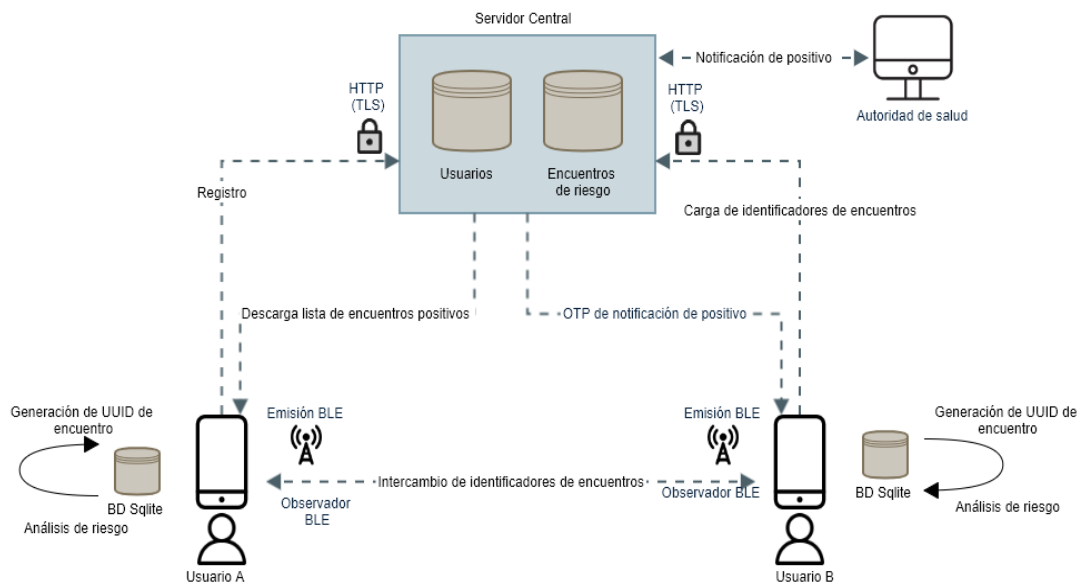


Ilustración 8: Arquitectura tecnológica.

9. Diseño del sistema

Tras especificar los requisitos y los distintos casos de uso del sistema, se va a proceder a la descripción detallada del funcionamiento global del sistema que enlaza de forma directa con los casos de uso anteriormente descritos.

Cada caso de uso, de forma análoga, se describe en forma de Diagrama de Actividad (DA). Los DA se utilizan para exponer la sucesión de acciones y actividades, de modo que se visualice el flujo de trabajo desde un nodo inicial a un nodo final.

9.1. Funcionamiento global del sistema

Como se ha mencionado anteriormente en las conclusiones del estado del arte, se busca la creación de un sistema el registro de contactos cercanos entre dispositivos móviles de forma anónima. Para la consecución de tal objetivo se ha optado por la adopción de una arquitectura descentralizada dados los claros beneficios que aporta a las necesidades que se persiguen. Paralelamente, se elige Flutter como SDK para el desarrollo de una solución multiplataforma de forma ágil y eficaz.

9.1.1. Registro

CU-001. El usuario de la aplicación, una vez descargada la App de la tienda oficial, tendrá acceso a una pantalla donde se le informará sobre la política de privacidad. En este punto, el usuario debe aceptar las condiciones para poder empezar a hacer uso del aplicativo, sin necesidad de realizar un registro. Tras esto, la aplicación solicita permisos para habilitar el uso del Bluetooth, así como la provincia del usuario y genera, una única vez, un identificador único universal (UUID) con el objetivo de identificar de manera exclusiva a la persona que usa la aplicación.

Se ha decidido por solicitar la provincia al usuario debido a que es una información útil para que el servidor central pueda realizar análisis estadísticos y, al no disponer de ningún dato más del usuario, resulta imposible determinar la identidad del mismo. De esta manera, el anonimato del usuario está asegurado.

Finalmente, ambos datos se almacenan en la base de datos local del dispositivo y se notifican al servidor para su posterior almacenamiento.

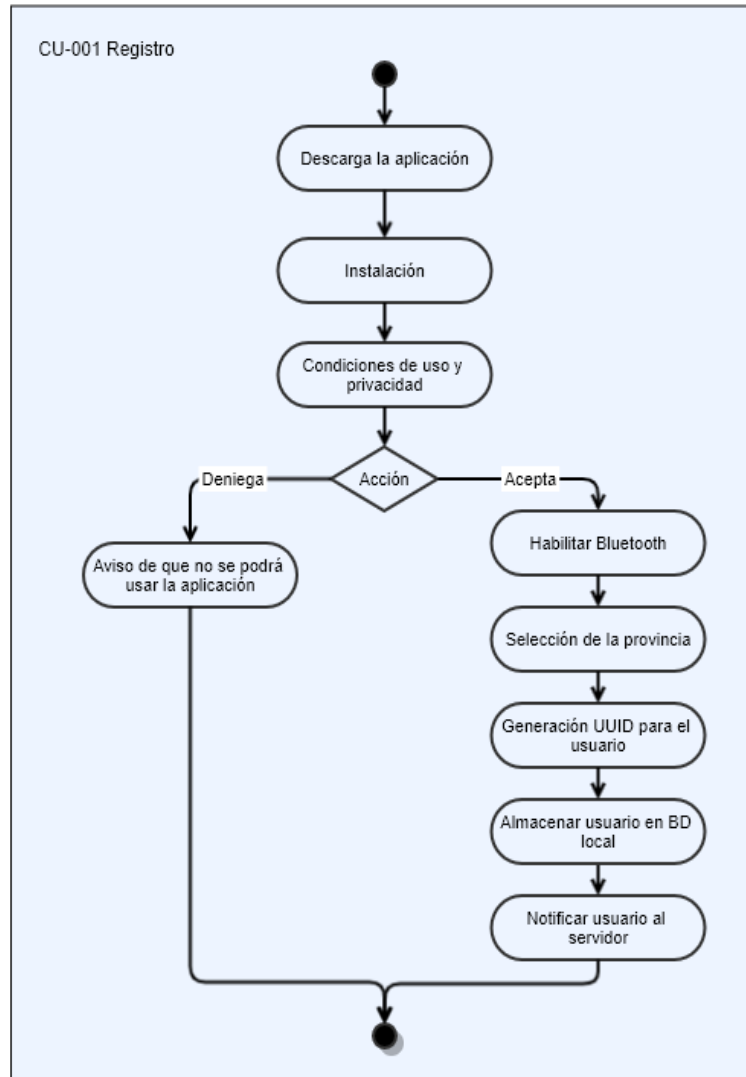


Ilustración 9: DA Registro.

A partir de aquí la aplicación lanza varios procesos, algunos de ellos sin intervención del usuario, que completan el ciclo de trazabilidad de enfermedades infecciosas con anonimizado.

9.1.2. Generación de identificadores de encuentros

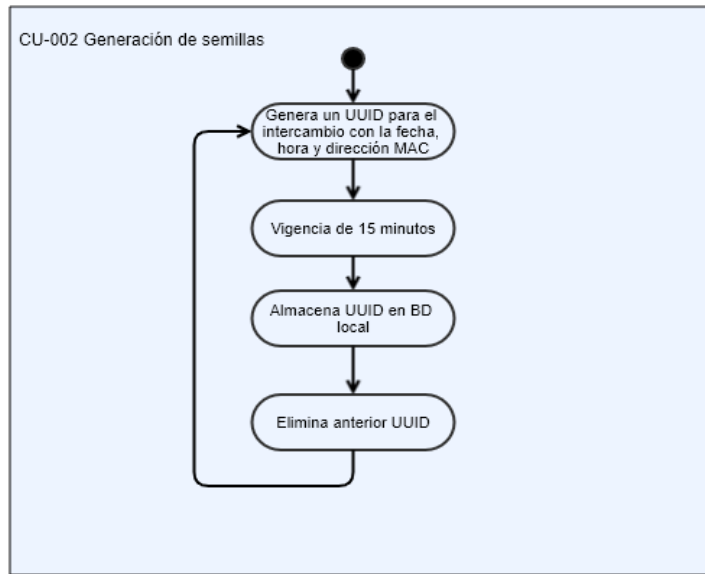


Ilustración 10: DA Generación de identificadores de encuentros.

CU-002. Uno de los objetivos primordiales que el sistema debe cumplir es el registro de encuentros entre dispositivos, de modo que podamos obtener y analizar estos datos para determinar el riesgo de contagio. Para ello, cada 15 minutos, el sistema genera un UUID a través de la fecha, hora y MAC del dispositivo. Estas semillas se usan como identificadores de los encuentros. Como no contienen información personal, su intercambio a través de BLE no revelará información sensible.

Cada vez que el sistema genera una de estas semillas, se almacenan en la base de datos del usuario y la semilla anterior queda descartada a través de un borrado lógico.

9.1.3. Intercambio de identificadores de encuentros

CU-003. Para el registro de encuentros se ha optado por el uso de la tecnología BLE. La razón principal para tal decisión es que algunos estudios [15] han demostrado que las mediciones de RSSI pueden ser suficientes para proporcionar contribuciones útiles a la evaluación de riesgos epidemiológicos.

Tan pronto como el sistema genere UUID de encuentro, ya está capacitado para el intercambio. De este modo, cada dispositivo se configura como un emisor BLE (broadcasting) y, a su vez, como un receptor. En [14] se explica la importancia de comprender el broadcasting, ya que es la única forma en que un dispositivo puede transmitir datos a más de un terminal a la vez, siendo otra de las razones por las que optamos por esta tecnología. Mediante esta configuración, en la que cada aplicación actúa como transmisor de información y como receptor, conseguimos una comunicación completa entre entidades que realicen un encuentro.

En el instante en el que el dispositivo recibe información de un emisor BLE, se realiza una estimación de la proximidad a través del valor RSSI. Como explica T. S. Rappaport [16], el valor RSSI medio (en dBm) a una distancia d del emisor viene definido por la ecuación:

$$RSSI(dBm) = RSSI_{d_0}(dBm) - 10n \log_{10}(d/d_0)$$

En dicha ecuación, $RSSI_{d_0}$ se corresponde con el RSSI promedio (en dBm) a una distancia de referencia de 1 metro, y n es el exponente de pérdida de trayectoria. El receptor puede realizar una estimación de la distancia a la que se está realizando la transmisión a través de su RSSI.

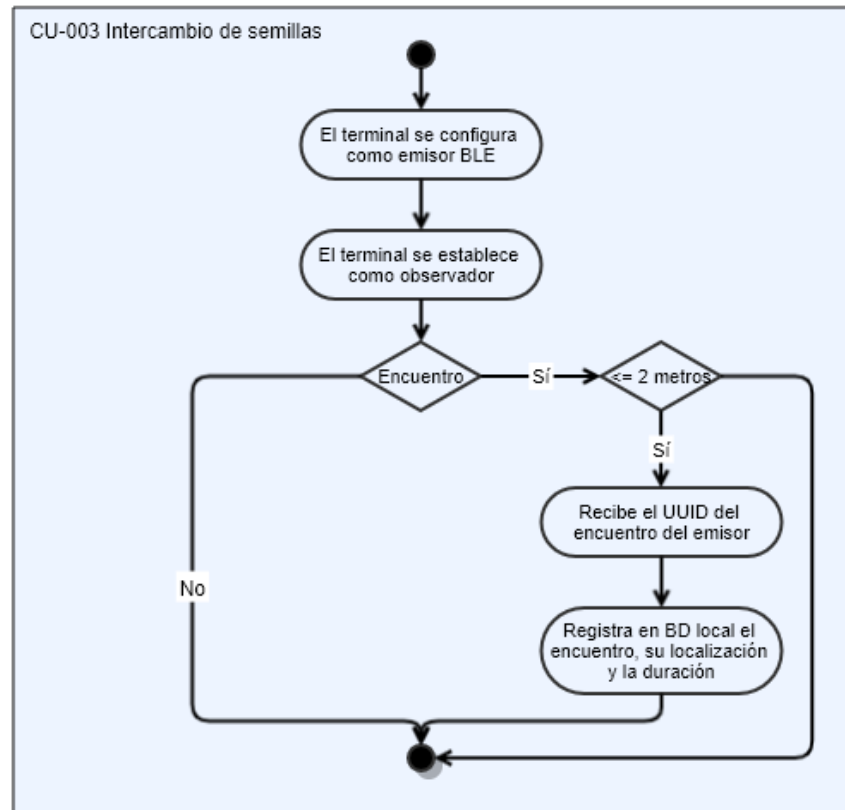


Ilustración 11: DA Intercambio de semillas.

Cuando los dispositivos que realizan el intercambio se encuentran a 2 metros o menos de distancia durante, al menos, 15 minutos, el sistema almacena en su base de datos local los siguientes datos: su UUID de encuentro actual, el UUID del dispositivo con el que ha realizado el encuentro, datos sobre el tiempo de exposición y su geolocalización, con el objetivo de proveer datos sobre el entorno en posteriores procesos. Estos datos de ubicación solo los conoce el terminal que almacena el encuentro, de este modo solo transmitimos identificadores únicos a través de BLE, evitando la exposición de datos personales.

9.1.4. Análisis de riesgo

CU-006. Puesto que este enfoque pretende mejorar la privacidad, el proceso de análisis de riesgo lo realiza localmente el usuario de la aplicación en su dispositivo en lugar del servidor central. En caso de que la persona haya estado en contacto con otro usuario que haya dado positivo, el sistema realiza una descarga de la base de datos encuentros de los últimos 14 días desde el servidor, siempre mediante una conexión HTTP con cifrado TLS. Una vez informado de

los encuentros, la aplicación comprueba si algún UUID en su base de datos de encuentros coincide con alguno de los que se consideran riesgo. En caso de que esto ocurra, señala que se ha dado un contacto con una persona diagnosticada.

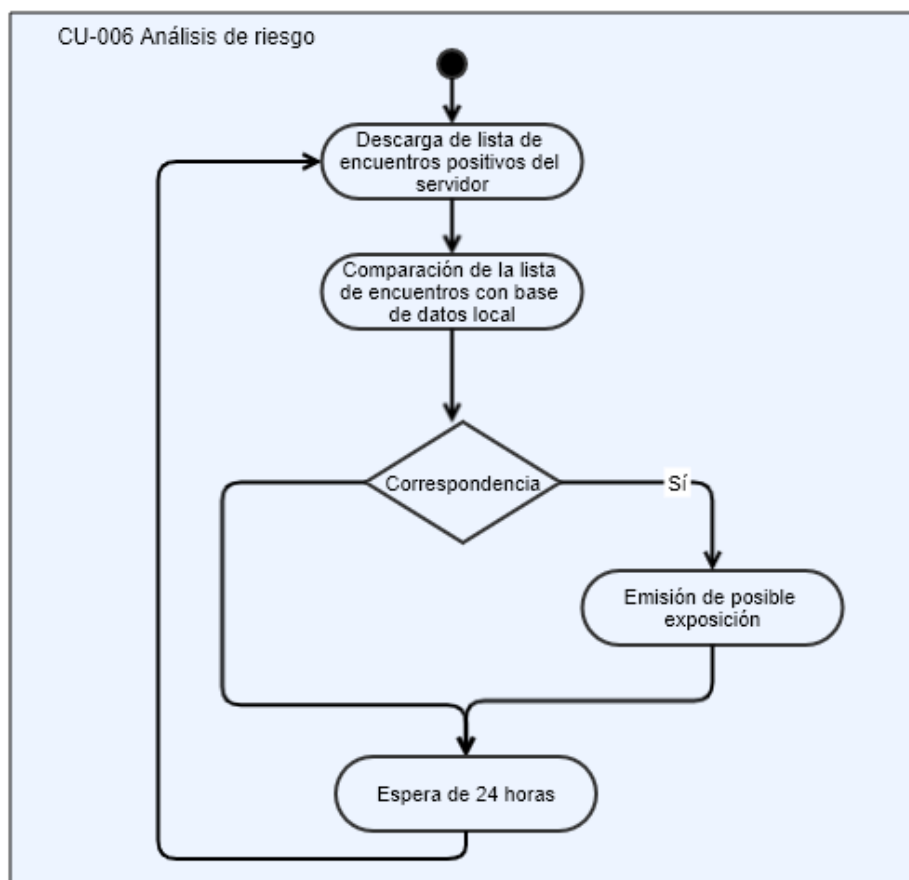


Ilustración 12: DA Análisis de riesgo.

Este proceso se realiza cada 24 horas sin intervención del usuario. El sistema comprobará el riesgo al que ha estado expuesto el dispositivo y, en ese caso de detectar algún peligro, alertará de una posible transmisión a través de notificaciones en la aplicación.

9.1.5. Notificación de usuario positivo

CU-004. Dentro del ciclo completo relativo a los procesos de trazabilidad de la enfermedad, la notificación de un positivo es el único procedimiento con intervención del usuario. El organismo de salud pertinente, tras un análisis previo, genera una contraseña de un solo uso (OTP) si el resultado ha resultado positivo. Una vez el usuario recibe su OTP a través de SMS, éste debe introducirla en la App para notificar su positivo.

Una vez utilizada la contraseña y aceptado el consentimiento, la aplicación comienza la transmisión de la base de datos de encuentros del propio terminal al servidor.

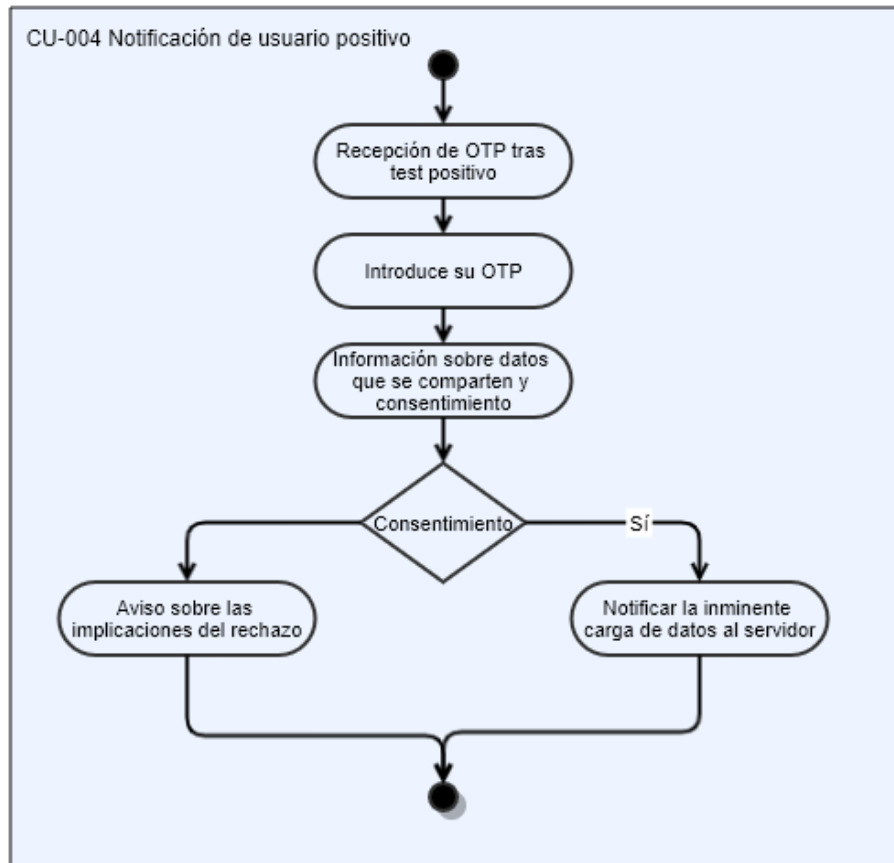


Ilustración 13: DA Notificación de usuario positivo.

9.1.6. Ciclo de trazabilidad de enfermedades infecciosas

Se ha modelado, en notación BPMN, el ciclo de trazabilidad completo para poder tener una imagen global de todo el proceso al completo. El funcionamiento de la aplicación está basado en dicho proceso repetido y formalizado con esta notación, siendo además el objetivo principal de la herramienta.

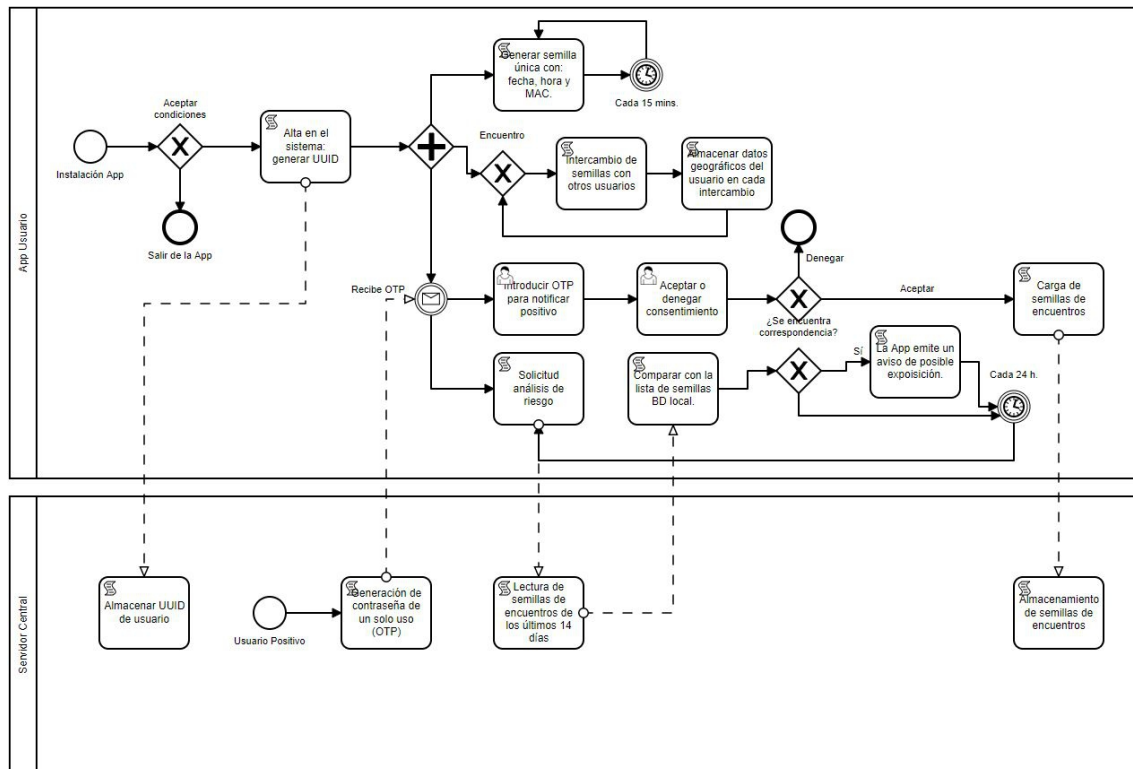


Ilustración 14: PMN del ciclo de trazabilidad de enfermedades infecciosas.

9.1.7. Otras opciones del aplicativo

La aplicación permite al usuario, a través de las opciones del menú inferior, navegar entre las distintas pantallas que emplean los datos recopilados para informar sobre la infección en la zona geográfica del usuario y sobre la evolución en términos absolutos.

9.1.7.1 Mapa de infecciones

CU-007. Cuando el usuario selecciona la opción “Mapa” del menú inferior, el sistema muestra un mapa de infecciones en la zona donde se encuentra el usuario, proporcionando información sobre el entorno en el que se encuentra en relación con la infección.

Para lograr tal objetivo, la aplicación recupera dicha información de la base de datos local, donde ya se almacenan los datos de los encuentros de riesgo (CU-006). En caso de que la aplicación no haya recibido aún datos relativos a encuentros de riesgo, se realiza una petición al servidor para obtener los datos actuales.

Con el objetivo de no cargar toda la información de la base de datos, se filtra la lista de identificadores de encuentros de riesgo que se encuentren en un radio de 25 km y que hayan sido registrados en los últimos 14 días.

Sobre el mapa se muestran círculos de colores que varían en función de si el usuario se ha visto expuesto o no. Un círculo en rojo sobre el mapa nos indica que el usuario ha estado en contacto con un positivo, mientras que los iconos en naranja nos muestran dónde se han realizado encuentros de riesgo entre otros usuarios de la aplicación, con el objetivo de obtener una imagen de las zonas más expuestas.

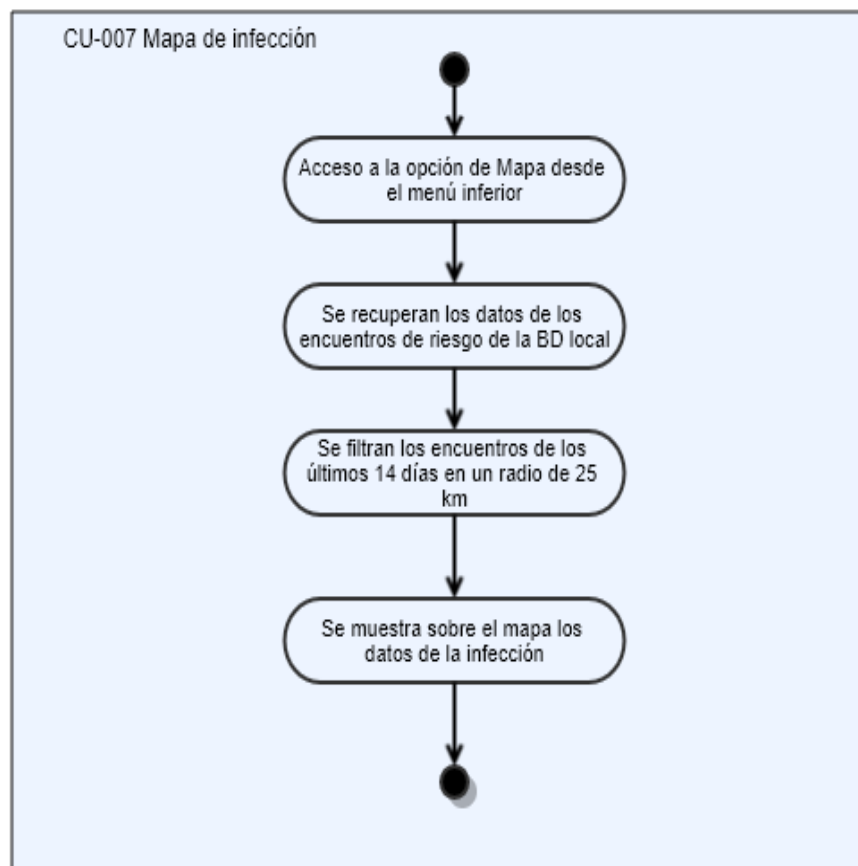


Ilustración 15: DA Mapa de infección.

9.1.7.2 Evolución

CU-008. El usuario puede consultar información sobre la evolución de la enfermedad al pulsar sobre la opción “Evolución” del menú inferior. A continuación se muestra una gráfica donde visualizar los nuevos casos confirmados para cada día y, a modo de resumen más general, datos específicos sobre el total global de infectados.

Dicha información, que es una generalización de todos los datos que recoge el sistema, se consulta al servidor central a través de una petición HTTPS. El servidor central, a través de una API Rest, devuelve los datos actualizados de la evolución de la enfermedad. La aplicación almacena en su base de datos local dicha información a modo de histórico, de modo que puedan consultarse días anteriores sin necesidad de realizar numerosas peticiones al servidor.

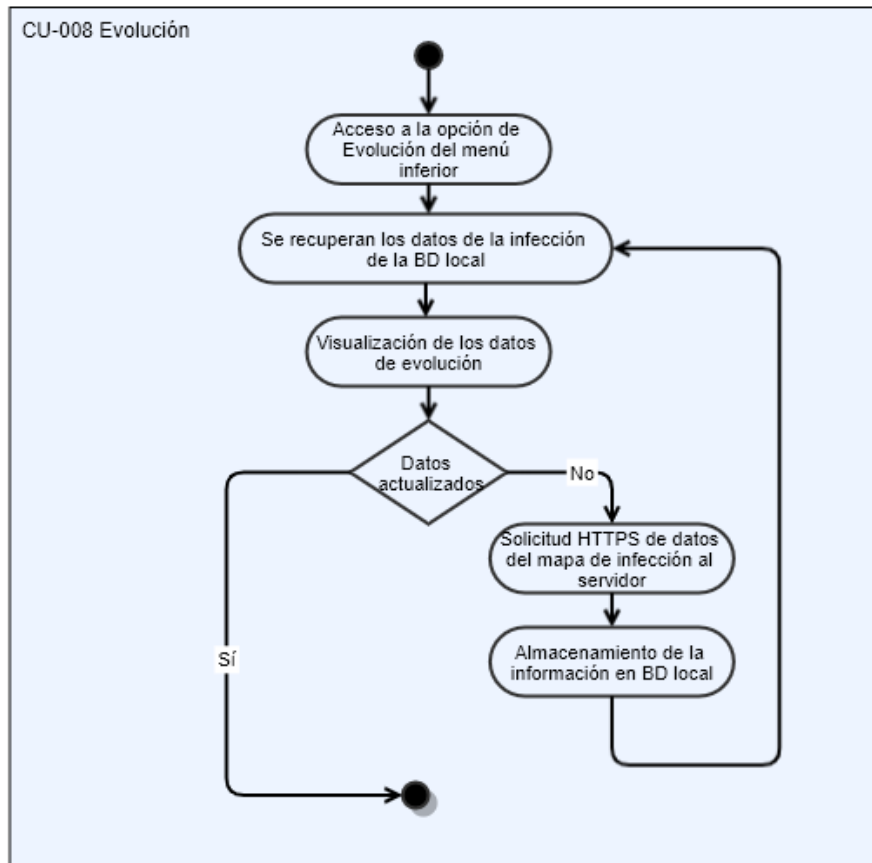


Ilustración 16: DA Evolución.

9.2. Prototipado

Para finalizar el modelado, se ha realizado un prototipado mediante mockups de las pantallas que intervienen en los distintos casos de uso. A través de los Diagramas de Actividad, es posible obtener una idea general de la composición de las pantallas y la forma en la que se visualizarán los datos.

Es importante resaltar que, debido a la naturaleza de los prototipados, se verán sujetos a posibles cambios en su implementación.

9.2.1. Bienvenida y registro

La primera vez que el usuario haga uso de la aplicación, tal y como se refleja en el DA de Registro, se muestra un mensaje de bienvenida y se informa sobre la política de privacidad.

Tras esto, y solo si se aceptan las condiciones, se solicita que aporte el dato de la provincia donde reside. Como ya hemos mencionado, este dato por sí solo nunca nos permitirá determinar su identidad, pero puede aportar información valiosa a la hora de realizar un análisis por zonas geográficas.



Ilustración 17: Prototipo de Bienvenida y registro.

9.2.2. Inicio de la aplicación

En el siguiente prototipo se ha definido la pantalla principal una vez el usuario ha instalado la aplicación, aceptado las condiciones y activado el Bluetooth para el correcto funcionamiento del sistema. Esta pantalla nos ofrece información general .

El primer bloque de información nos comunica si ha habido contactos de riesgo. Además de esto, si el usuario ha deshabilitado el Bluetooth, se muestra un mensaje avisando de que el sistema está inactivo y se informa sobre cómo activar dicha opción.

Desde esta pantalla, además, tenemos un botón que permite al usuario notificar su positivo una vez haya recibido su OTP tras haberse sometido al test específico, dando solución a la interacción del usuario para la notificación dentro del caso de uso **CU-004**.

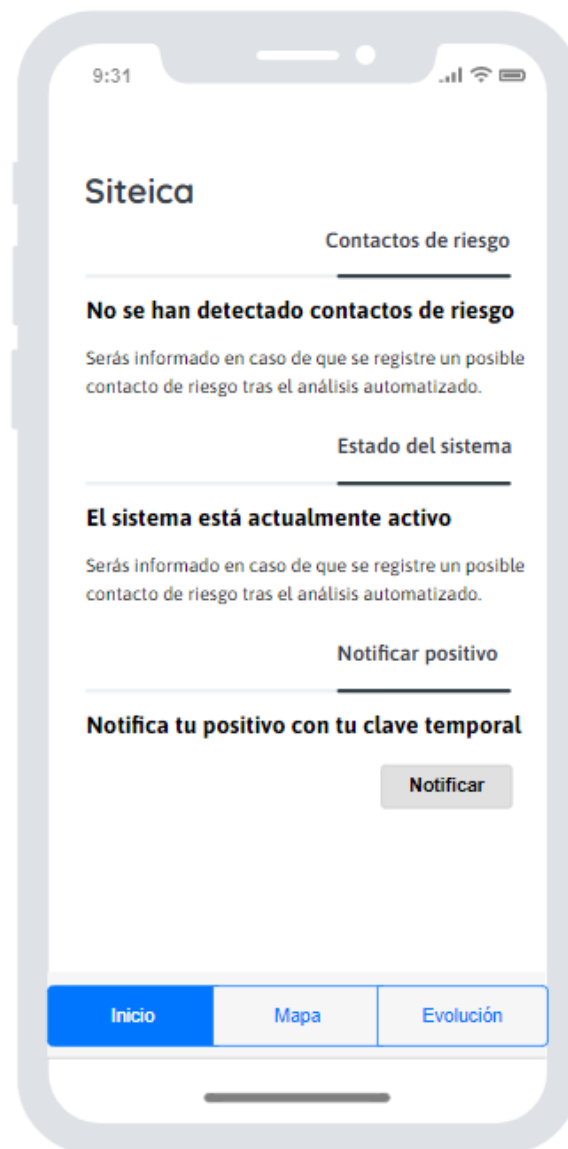


Ilustración 18: Prototipo de Inicio.

9.2.3. Notificación de usuario positivo

Tal y como se refleja en el **CU-004**, el flujo comienza cuando el usuario recibe su OTP tras realizar un test de infección. Desde la pantalla de inicio, el usuario pulsa sobre el botón “Notificar” para comenzar con el proceso.

The image displays two mobile application screens for reporting a positive diagnosis. Both screens have a status bar at the top showing the time 9:31 and signal/battery icons. A red back arrow is in the top left corner of each screen.

Screen 1 (Left): The title is "Notificar positivo". Below it is a section header "Código de diagnóstico". There is a text input field with the placeholder "Introduce tu código de diagnóstico". Below the field is an example: "Ejemplo: 012-3456-789". The next section header is "Fecha del diagnóstico". Below it is a text input field with the date "04/12/2021" and a calendar icon. A paragraph of text reads: "Si recuerdas el día en el que comenzaron los síntomas, introdúcelo a continuación. En caso contrario, introduce la fecha de diagnóstico." At the bottom are two buttons: "< Cancelar" and "Continuar >".

Screen 2 (Right): The title is "Notificar positivo". Below it is a section header "Datos de diagnóstico". It shows the filled-in information: "Tu código es: 012-3456-789" and "Tu fecha del diagnóstico es: 04/12/2021". The next section header is "Información adicional". Below it is a paragraph of text: "SITEICA no recopila información personal de ningún tipo. Si pulsar en el botón 'Notificar', se compartirán los datos anónimos sobre tus encuentros para notificar a otros usuarios sobre el posible riesgo. En caso de no estar de acuerdo, pulsa en el botón 'Cancelar'." At the bottom are two buttons: "< Cancelar" and "Notificar >".

Ilustración 19: Prototipo de Notificación de usuario positivo.

9.2.4. Mapa de infecciones

Como parte de las necesidades del caso de uso **CU-007** se encuentra la visualización del mapa de infecciones. Cuando el usuario consulta el mapa de infecciones, el sistema posiciona al terminal de forma automática sobre el mapa con el objetivo de consultar información cercana. El cliente puede mover el mapa para consultar otras zonas.

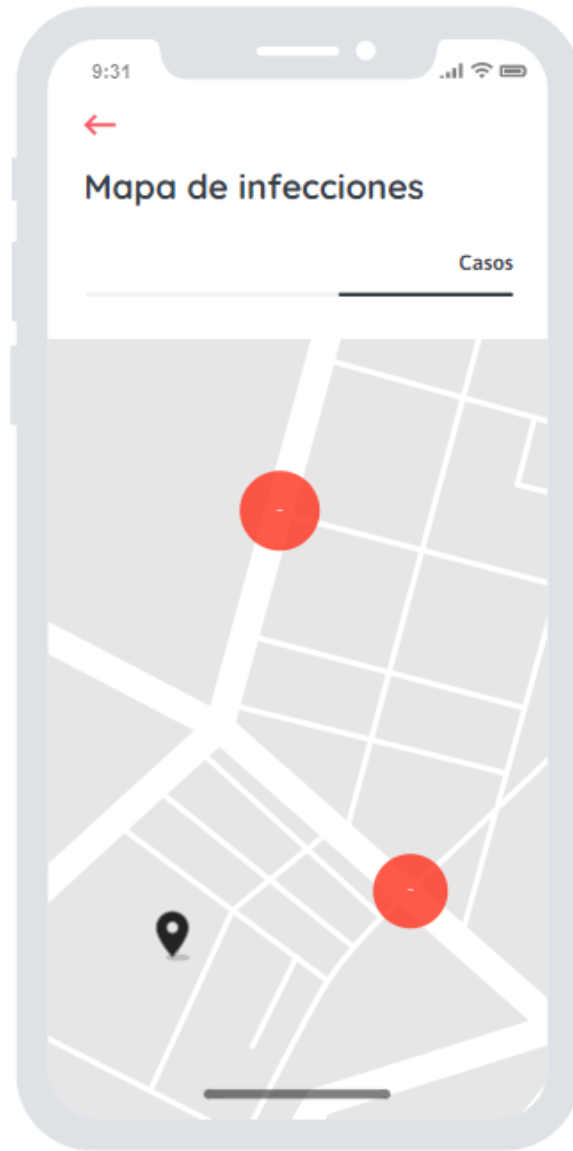


Ilustración 20: Prototipo de Mapa de infección.

Como se ha mencionado anteriormente, el mapa nos informará de los contactos de riesgo a los que el usuario se ha visto expuesta a través de un círculo en color rojo sobre el mapa. El resto de encuentros de riesgo se mostrarán en naranja, indicando que ha habido riesgo en la zona.

9.2.5. Evolución

El usuario puede consultar los datos sobre la evolución de la infección a través del botón del menú inferior “Evolución”, tal y como puede verse reflejado en el caso de uso **CU-008**. En este prototipo podemos comprobar que el usuario tiene la posibilidad de consultar los datos actualizados sobre la enfermedad en lo que se ha nombrado como “Resumen”, así como los “Casos diarios” en forma de gráfica de tiempo y los datos segmentados por provincia.



Ilustración 21: Prototipo de Evolución.

9.3. Diseño visual

Tras el análisis y definición del prototipado, donde se ha esbozado la disposición visual de la interfaz de usuario, se han aplicado las características específicas de diseño que dotarán a la aplicación de identidad, concretando así su presentación e interacción (“look and feel”).

Se ha utilizado Material Design como normativa de diseño, cumpliendo así las características mencionadas anteriormente en la sección 7.8.2. y en el requisito **RNF-002**, logrando así una experiencia de usuario ya conocida debido a que las aplicaciones oficiales de Google se basan en dichos principios.

Para la aplicación de Material Design, Flutter provee de un catálogo completo de componentes UI que implementan estas guías de diseño.

9.4. Modelo entidad-relación

A continuación se exponen, de manera conjunta, las entidades que se han diseñado para que el sistema sea capaz de dar soporte a todas las necesidades de los casos de uso. Para entender las responsabilidades de cada tabla, se han definido cada una de ellas de manera individual en los siguientes subapartados, donde además se analizan a qué acciones de los DA dan soporte dichas entidades.

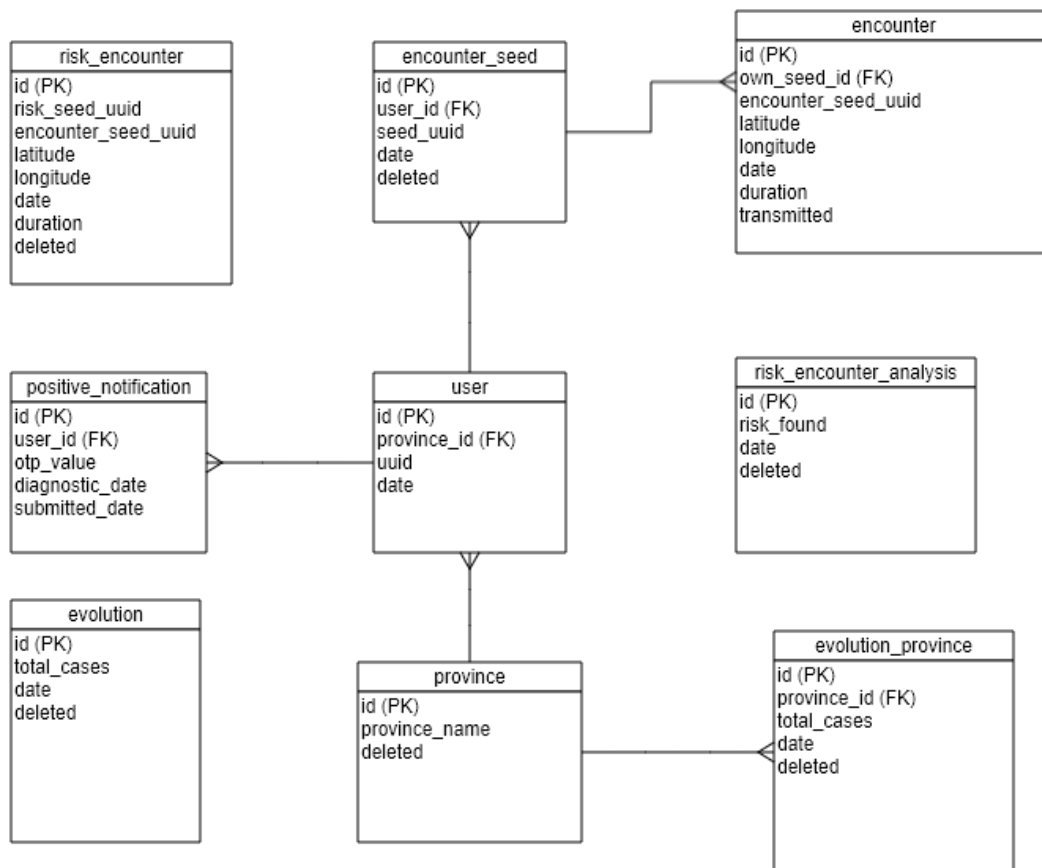


Ilustración 22: Modelo entidad-relación.

9.4.1. Tabla de usuario

Esta tabla, llamada 'user', es utilizada para almacenar datos anónimos del usuario de la aplicación. Nada más se arranque la aplicación por primera vez, el sistema genera un registro en la tabla de usuario, creando un nuevo UUID que nos sirva para determinar de manera única al usuario y anotando la marca temporal del momento del registro. Esta entidad resuelve las necesidades del caso de uso **CU-001**.

Sus campos son:

- id (PK): identificador único autonumérico y clave primaria de la entidad.
- province_id (FK): clave externa que referencia la provincia selecciona en el registro.
- uuid: identificador único del usuario de la aplicación.
- date: valor numérico que representa la marca temporal de la creación del usuario.

9.4.2. Tabla de identificadores de encuentros

El objetivo de esta tabla, a la que se ha llamado 'encounter_seed', es almacenar los identificadores de encuentros. Estos UUID relativos a encuentros, tienen una vigencia de 15 minutos, como se explica en el caso de uso **CU-002**.

Para lograr tales objetivos, la siguiente entidad almacena una relación al usuario de la aplicación (entidad 'user'), así como el UUID del encuentro y su valor temporal del momento en el que se ha creado. De esta forma el sistema es capaz de marcar como eliminado el identificador y generar uno nuevo tras el periodo de vigencia mencionado.

Sus campos son:

- id (PK): identificador único autonumérico y clave primaria de la entidad.
- user_id (FK): clave externa al usuario actual.
- seed_uuid: identificador único que se utiliza como identificador de encuentro del dispositivo en un momento específico.
- date: valor numérico que representa la marca temporal del momento de creación de la semilla. Así el sistema controla el tiempo de vigencia de una semilla (15 minutos).
- deleted: campo de tipo *boolean* que almacena si un registro ha sido eliminado (borrado lógico).

9.4.3. Tabla de encuentros

La entidad 'encounter' es la encargada de almacenar los encuentros cercanos entre dos dispositivos. El sistema genera un registro en la tabla actual cuando dos terminales entran en contacto a una distancia menor o igual a 2 metros, almacenando los UUID actuales del encuentro que cada teléfono móvil emite a través de la tecnología BLE. En caso de que el UUID recibido ya existe en la base de datos local, el sistema no genera un nuevo registro si no que actualiza la duración del encuentro. Este dato se contabiliza en segundos ya que el complemento beacons_plugin realiza una lectura en intervalos de un segundo.

A la hora de realizar el registro, se almacenan también datos relativos a la ubicación del encuentro y el tiempo total que se hayan mantenido en contacto. Resuelve así las necesidades del caso de uso **CU-003**.

Sus campos son:

- **id (PK):** identificador único autonumérico y clave primaria de la entidad.
- **own_seed_id (FK):** clave externa a la tabla `encounter_seed` con la semilla actual del dispositivo en el instante del encuentro.
- **encounter_seed_uuid:** identificador único de la semilla del dispositivo con el que se realiza el encuentro en dicho instante.
- **latitude:** valor numérico de la latitud de la coordenada geográfica en el instante del encuentro.
- **longitude:** valor numérico de la longitud de la coordenada geográfica en el instante del encuentro.
- **date:** valor numérico que representa la marca temporal del encuentro.
- **duration:** duración total del encuentro. El receptor BLE realiza una lectura de las emisiones cada segundo, con lo que la duración se contabiliza en segundos.
- **transmitted:** campo de tipo *boolean* que almacena si un registro ha sido transmitido al servidor.

9.4.4. Tabla de notificaciones de positivo

En la entidad que se ha nombrado como 'positive_notification' se almacenan el histórico de notificaciones de positivo que el usuario ha realizado al servidor central. Estos avisos al servidor central se dan cuando una persona realiza el test en su centro de salud y, tras dar un positivo, recibe una clave de un solo uso que permite comunicar al servidor que está infectado. El proceso de envío de dicha información puede verse en el prototipo de notificación de usuario positivo.

Cuando el usuario sigue dicho proceso y consiente el envío de su información, el sistema almacena en la tabla actual un registro identificativo del valor de la OTP que se ha usado y la fecha del diagnóstico, así como el momento en el que se ha realizado la notificación, cubriendo así las necesidades del **CU-004**.

Sus campos son:

- **id (PK):** identificador único autonumérico y clave primaria de la entidad.
- **user_id (FK):** clave externa al usuario actual.
- **otp_value:** valor de la clave de un solo uso que se ha utilizado para realizar la notificación.
- **diagnostic_date:** fecha del diagnóstico que el usuario ha introducido en la pantalla de notificación.
- **submitted_date:** fecha del envío de la notificación al servidor central.

9.4.5. Tabla de encuentros de riesgo

La tabla 'risk_encounter' es utilizada para almacenar los encuentros de riesgo que el dispositivo descarga del servidor.

En el DA de Análisis de riesgo hemos visto la necesidad de obtener, del servidor central perteneciente a la autoridad sanitaria, la lista de encuentros de riesgo de los últimos 14 días para que el sistema pueda llevar a cabo el análisis de riesgo e informar sobre la situación geográfica de la enfermedad. Para llevar a cabo dichos objetivos, la aplicación solicita dicha información al servidor a través de una petición HTTPS y la almacena en su base de datos local. De esta manera, dicha entidad resuelve las necesidades del caso de uso **CU-006** y **CU-007**.

La información útil de 'risk_encounter' reside en los UUID de encuentros que almacenamos en 'risk_seed_uuid' y 'encounter_seed_uuid', que se utilizan para que el usuario local pueda comparar si alguno de sus identificadores, en su registro de encuentros 'encounter', ha estado en contacto con algún contacto de riesgo.

Esta tabla no deja de ser un registro global de los encuentros que han realizado usuarios que han dado positivo. El usuario local solo ha de buscar si alguno de sus identificadores de encuentro, almacenados en 'encounter_seed', se encuentran en algún registro de esta tabla. Concretamente en la columna 'encounter_seed_uuid', puesto que indicaría que un usuario positivo se ha encontrado con el usuario local en algún momento.

Sus campos son :

- id (PK): identificador único autonumérico y clave primaria de la entidad.
- risk_seed_uuid: identificador único de la semilla del dispositivo infectado.
- encounter_seed_uuid: identificador único de la semilla del dispositivo con el que se realiza el encuentro en dicho instante.
- latitude: valor numérico de la latitud de la coordenada geográfica en el instante del encuentro.
- longitude: valor numérico de la longitud de la coordenada geográfica en el instante del encuentro.
- date: valor numérico que representa la marca temporal del encuentro de riesgo.
- duration: el valor de duración recogido en el momento del encuentro y que también se vuelca al servidor central.
- deleted: campo de tipo *boolean* que determina el borrado lógico del registro.

9.4.6. Tabla de análisis encuentros de riesgo

Esta tabla, definida como 'risk_encounter_analysis', almacena los análisis que la aplicación ha realizado a lo largo del tiempo. Estos datos almacenan la fecha de la última vez que se ha realizado el análisis de riesgo, lo que permite que la aplicación no vuelva a lanzar el proceso de nuevo hasta que hayan pasado 24 horas, y si el usuario ha resultado expuesto a un posible riesgo.

Sus campos son :

- **id (PK):** identificador único autonumérico y clave primaria de la entidad.
- **risk_found:** campo de tipo *boolean* que determina si durante el análisis se ha detectado una posible exposición de riesgo.
- **date:** valor numérico que representa la marca temporal del último análisis de riesgo realizado. El sistema utiliza este dato para que la siguiente comprobación no se haga hasta haber superado las 24 horas desde el último análisis.
- **deleted:** campo de tipo *boolean* que determina el borrado lógico del registro, de modo que el usuario puede borrar el aviso si así lo desea y dejar de visualizarlo en la aplicación.

9.4.7. Tabla de información de evolución

La siguiente tabla, 'evolution', es utilizada para almacenar los datos generales de la evolución de la enfermedad infecciosa. La información sobre la evolución se recoge desde el servidor central puesto que es el encargado de recibir todos los datos sobre positivos y encuentros de riesgo. El servidor, como parte de su responsabilidad, deberá mantener un registro de los casos totales por días. Por su parte, la aplicación utiliza dicha información para mostrar una gráfica de tiempo con la que informar sobre los casos diarios y su progreso. Resuelve las necesidades del caso de uso **CU-008**.

Sus campos son:

- **id (PK):** identificador único autonumérico y clave primaria de la entidad.
- **total_cases:** valor numérico del total de casos.
- **date:** valor numérico que representa la marca temporal de los datos sobre la evolución. Cada registro almacena los datos de un día.
- **deleted:** campo de tipo *boolean* que determina el borrado lógico del registro.

9.4.8. Tabla de provincias

La entidad 'province' es una tabla maestra de provincias. Contiene todas las provincias que el usuario puede seleccionar en el momento del registro y, además, resuelve las necesidades del caso de uso **CU-008**, en referencia a los datos de evolución segmentados por provincias.

Sus campos son:

- **id (PK):** identificador único autonumérico y clave primaria de la entidad.
- **province_name:** nombre de la provincia.
- **deleted:** campo de tipo *boolean* que determina el borrado lógico del registro.

9.4.9. Tabla de evolución por provincias

Esta tabla llamada 'evolution_province' es utilizada para almacenar los datos generales de la evolución en una provincia. Resuelve las necesidades del caso de uso **CU-008**, en referencia a los datos de evolución segmentados por provincias.

El servidor es el encargado de completar los datos de evolución dado que es el punto central de la arquitectura y el que dispone de toda la información sobre los contagios, además de más capacidad de computación para realizar tareas pesadas como el cálculo de totales. Esta petición se solicita al servidor a través de una petición HTTPS y se almacena en la BD local.

Sus campos son:

- `id` (PK): identificador único autonumérico y clave primaria de la entidad.
- `province_id` (FK): clave externa al registro de provincia (`province`).
- `total_cases`: valor numérico del total de casos.
- `date`: valor numérico que representa la marca temporal de los datos sobre la evolución.
- `deleted`: campo de tipo *boolean* que determina el borrado lógico del registro.

10. Instalación y configuración del entorno

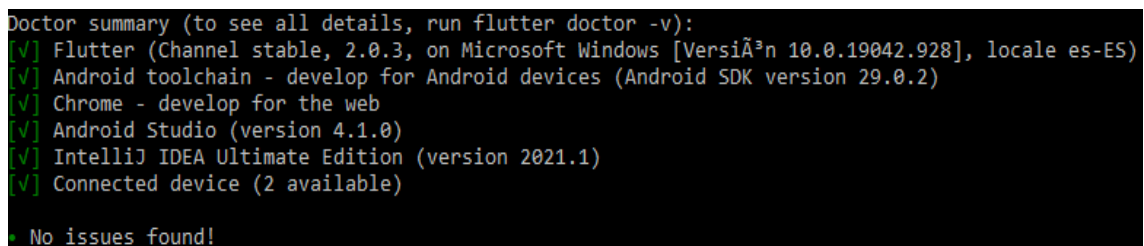
En el siguiente apartado se describe el proceso de instalación y configuración del entorno necesario para la realización y ejecución del proyecto.

10.1. Aplicación de Flutter

El primer paso para poder realizar el desarrollo de una aplicación móvil en Flutter es obtener el SDK siguiendo las especificaciones de la web oficial. Los pasos seguidos han sido, tal y como se explica paso a paso en la documentación oficial [31], descargar la última versión estable, descomprimir el archivo y agregar dicho directorio al PATH de Windows, SO escogido para el desarrollo del proyecto.

A partir de ahora, la herramienta de línea de comandos de Flutter nos permite comunicarnos con el SDK mediante diversos comandos. Antes de nada es importante cerciorarse del estado de la instalación con el objetivo de conocer si tenemos todas las dependencias instaladas. Para ello Flutter dispone del siguiente comando:

```
> flutter doctor
```



```
Doctor summary (to see all details, run flutter doctor -v):
[✓] Flutter (Channel stable, 2.0.3, on Microsoft Windows [VersiÃ³n 10.0.19042.928], locale es-ES)
[✓] Android toolchain - develop for Android devices (Android SDK version 29.0.2)
[✓] Chrome - develop for the web
[✓] Android Studio (version 4.1.0)
[✓] IntelliJ IDEA Ultimate Edition (version 2021.1)
[✓] Connected device (2 available)

• No issues found!
```

Ilustración 23: Resultado de ejecutar flutter doctor.

Flutter evalúa la instalación y nos muestra por pantalla su estado, notificando si nos falta algún componente o si ha encontrado algún problema. A través de este contacto también es posible conocer las versiones instaladas en nuestro equipo, tanto del propio SDK como de sus dependencias.

Tras la configuración del entorno de trabajo, Flutter nos permite crear un proyecto nuevo con una estructura básica a través del siguiente comando:

```
> flutter create siteica-user
```

10.2. Estructura básica del proyecto

Dentro de la estructura completa de un proyecto de Flutter encontramos lib/, directorio principal del proyecto donde se encuentra todo el código fuente de nuestra aplicación de Flutter. Para alcanzar un mayor nivel organizativo se ha distribuido esta carpeta en las siguientes subcarpetas:

- app/: este directorio se ha creado con la finalidad de recoger la funcionalidad del proyecto que se ejecuta nada más se lanza la aplicación, como registrar las dependencias en el contenedor (Injector) para su posterior uso como singletons.
- models/: aquí se han ubicado las clases relativas a modelos de datos. Estos modelos se corresponden con las entidades de la base de datos, con lo que se han utilizado para manejar colecciones de estos datos y ser utilizados en las vistas para visualizarlos.
- services/: servicios encargados de la lógica de negocio. Estas clases se registran en el contenedor de dependencias, de modo que son transversales a toda la aplicación.
- ui/: las clases encargadas de manejar la interfaz de usuario.
- utils/: en esta carpeta se han ubicado elementos como constantes globales y métodos estáticos que proveen funcionalidades o utilidades reutilizables.
- main.dart el punto de entrada de la aplicación.

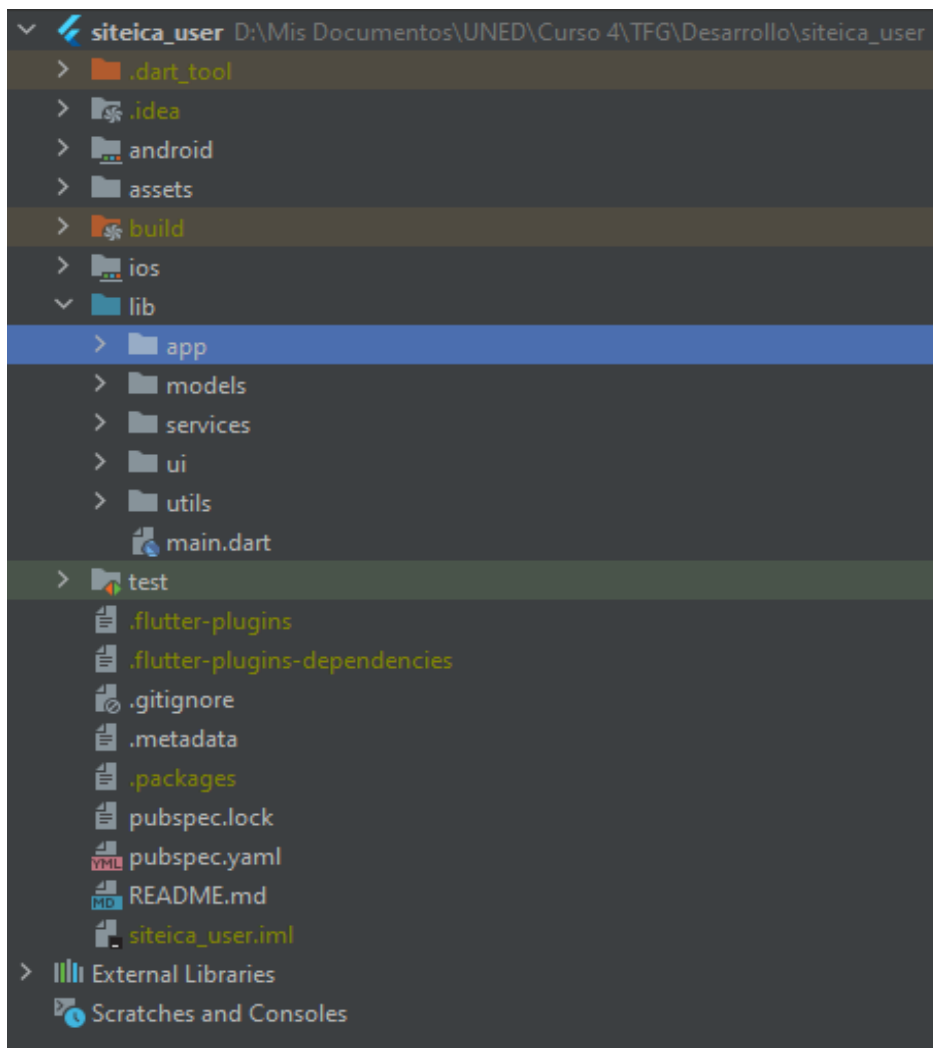


Ilustración 24: Estructura del proyecto.

Además del código fuente de la propia aplicación, encontramos otros archivos y directorios importantes:

- `android/`: aquí se encuentran los archivos específicos para la compilación en Android. Los cambios que se quieran aplicar a nivel de este sistema se realizan en este directorio.
- `assets/`: la carpeta donde se ubican los archivos estáticos. Además de iconos e imágenes, aquí se han ubicado los archivos SQL encargados de lanzar las migraciones de la base de datos.
- `build/`: este directorio se genera de forma automática como parte del proceso de compilación de Flutter.
- `ios/`: de un modo análogo a la carpeta de Android, aquí se ubican los cambios específicos para la compilación en iOS.
- `test/`: es un directorio que contiene código orientado a la ejecución de test automatizados.
- `pubspec.yaml`: contiene propiedades generales del proyecto como el nombre o su descripción, así como la lista de paquetes que son necesarios para ejecutar la aplicación y las rutas de los distintos archivos estáticos que van a empaquetarse.

10.3. Dependencias del proyecto

Un proyecto de Flutter utiliza Pub como administrador de paquetes, herramienta que está presente como parte del ecosistema de Dart. Pub nos simplifica la administración de software compartido, como bibliotecas de clases o herramientas específicas, gestionando incluso las dependencias de versiones.

Como hemos mencionado, el archivo `pubspec.yaml` es el encargado de manejar las dependencias del proyecto, mencionadas en la sección 7.7., y sus versiones correspondientes.

```
dependencies:  
  beacon_broadcast: ^0.2.3  
  beacons_plugin: ^1.0.20  
  flutter:  
    sdk: flutter  
  flutter_map: ^0.12.0  
  geolocator: ^7.0.1  
  json_serializable: ^3.5.1  
  injector: ^2.0.0  
  sqflite: ^2.0.0+3  
  sqflite_migration_service: ^1.0.6  
  uuid_enhanced: ^3.0.2
```

Ilustración 25: Dependencias del proyecto.

Para obtener las dependencias Flutter nos ofrece el siguiente comando:

```
> flutter pub get
```

Este comando es el encargado de determinar de qué paquetes depende la aplicación y, en el caso de los paquetes publicados, pub descarga el complemento desde el repositorio pub.dev, así como sus dependencias asociadas.

11. Implementación

Una vez se ha configurado el entorno es turno de construir la solución en base al análisis realizado previamente.

11.1. Construyendo la interfaz de usuario

La idea principal sobre la que se basa Flutter, al igual que otras librerías como React, es que la interfaz de usuario se construye de widgets al completo. Los widgets, que son el componente mínimo de la UI, describen cómo es una vista, dada su configuración y estado actual.

Los widgets forman una jerarquía en forma de árbol basada en la composición. Cada widget se anida dentro de su padre y puede recibir su contexto. Esta estructura llega hasta el widget raíz, el que ya conocemos como `main.dart`, punto de partida de una aplicación de Flutter.

Del mismo modo que ocurre en Java, se debe definir una función `main()` para poder ejecutar cualquier programa escrito en Dart. Como puede observarse en la siguiente ilustración, se ejecutan dos sentencias cuando arranca la aplicación. La primera de ellas se corresponde con la configuración del inyector de dependencias, punto donde se registran los distintos servicios como singleton. La segunda, común a cualquier otra aplicación de Flutter, establece el widget raíz dentro del árbol.



```
>> void main() {  
    setupDependencyInjector();  
    runApp(SiteicaApp());  
}
```

Ilustración 26: Método principal de la aplicación.

La raíz, en este caso, se corresponde con la clase `SiteicaApp` que extiende de `StatefulWidget`. De forma general, la forma de crear nuevos widgets es a través de la creación de una clase que extienda de `StatelessWidget` o `StatefulWidget`, dependiendo de si el widget gestiona algún estado o debe permanecer inmutable. Como parte de su funcionalidad básica, estas nuevas clases deben implementar el método `build` en el que devolver el widget que se introducirá en el siguiente nivel del árbol.

En el proyecto de SITEICA se ha optado por inicializar una serie de procesos asíncronos en la clase que hace de raíz del árbol de widgets, de ahí que se haya optado porque `SiteicaApp` será un widget con estado. Para lograr tal objetivo se ha extendido el método `initState()`, función que se ejecuta cuando el objeto se inserta en el árbol, donde se llama a la función asíncrona `initDatabaseAndGetUser()`.

Dicho método realiza dos operaciones. La primera de ellas se encarga de ejecutar la lógica del servicio de migraciones, proceso que se lanza cada vez que la aplicación arranca y

```
initDatabaseAndGetUser() async {  
  await _databaseService.initialise();  
  _user = await _userService.getUser();  
  setState(() {  
    _isLoading = false;  
  });  
}  
  
@override  
void initState() {  
  super.initState();  
  initDatabaseAndGetUser();  
}
```

Ilustración 27: Inicialización en main.dart del proyecto.

que se ha descrito en la siguiente sección. La segunda sentencia, a través del servicio correspondiente que implementa la lógica de negocio relativa a usuarios, busca en la base de datos local si existe un usuario.

Ambas operaciones son asíncronas, es decir, devuelven objetos de tipo Future que permiten que el programa complete su trabajo mientras espera a que otras instrucciones finalicen. En este caso concreto, puesto que sin haber inicializado la base de datos será imposible acceder a ninguna tabla, usamos la instrucción await que permite obtener el resultado completado de una expresión asíncrona.

```

class SiteicaApp extends StatefulWidget {
  @override
  _SiteicaAppState createState() => _SiteicaAppState();
}

class _SiteicaAppState extends State<SiteicaApp> {
  final _userService = Injector.appInstance.get<UserService>();
  final _databaseService = Injector.appInstance.get<DatabaseService>();
  bool _isLoading = true;
  User _user;

  initDatabaseAndGetUser() async {...}

  @override
  void initState() {...}

  @override
  Widget build(BuildContext context) {
    return _isLoading ? CommonProgressIndicator() : MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.lightBlue,
        visualDensity: VisualDensity.adaptivePlatformDensity,
      ), // ThemeData
      home: _user == null ? WelcomePage() : StartPage(),
    ); // MaterialApp
  }
}

```

Ilustración 28: Clase SiteicaApp, raíz del árbol de widgets.

Podemos observar que, al final del método, se ha llamado a `setState()`. Esta instrucción permite notificar al framework que el estado del widget ha cambiado y que debe volver a dibujarlo (es decir, volver a ejecutar el método `build()` del widget). En este caso resulta interesante para mantener en la variable booleana `_isLoading` si las instrucciones anteriores han finalizado. Así, mientras las operaciones bloqueantes no hayan finalizado, se visualiza un icono a modo de indicador de progreso.

Finalmente, como resultado del método `build()`, mostramos un icono de progreso si `_isLoading` es cierto y, una vez se haya llamado a `setState()` y `_isLoading` pase a falso, se devuelve un objeto de tipo `MaterialApp`, una clase del SDK que incluye una serie de widgets que se requieren para aplicaciones que usen los principios de Material Design.

11.2. Inicialización de la base de datos

Como ya se ha mencionado en la sección 7.7.6, se ha optado por incluir un sistema ligero de migraciones de bases de datos que nos permita simplificar tanto la creación de las entidades como la gestión de los cambios a través del plugin Sqlflite Migration Service.

Como paso inicial, se ha configurado el proyecto para incluir una ruta de archivos estáticos al proyecto. En dicho directorio se alojan los ficheros SQL que contienen las sentencias necesarias para la creación o actualización de entidades, así como la inserción de registros. Así, se ha editado el archivo pubspec.yaml para incluir, dentro del nodo “assets”, la carpeta /sql.

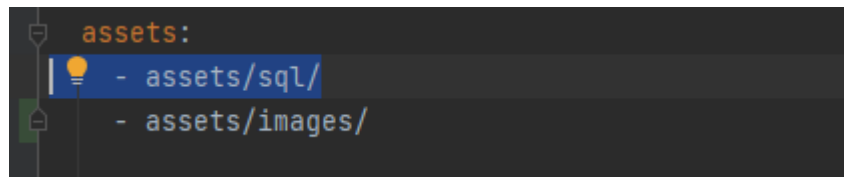


Ilustración 29: Archivos estáticos incluidos en el proyecto.

Una vez incluida la ruta, se deben alojar los archivos de migración en dicho directorio con la siguiente nomenclatura: (“[schema_version_number]_[migration_name].sql”). El número de versión del esquema es el que sirve para comparar contra la versión de la base de datos actual y ejecutar las versiones que falten.

```
class DatabaseService {
  final _migrationService =
    Injector.appInstance.get<DatabaseMigrationService>();
  Database _database;

  Future initialise() async {
    _database = await openDatabase(DB_NAME, version: 1);

    await _migrationService.runMigration(
      _database,
      migrationFiles: [
        '1_create_schema.sql',
        '2_create_data_set.sql',
      ],
      verbose: true,
    );
  }
}
```

Ilustración 30: Clase DatabaseService e inicialización de BD.

A continuación se ha configurado el servicio de migraciones dentro del servicio DatabaseService. La clase DatabaseService contiene un método initialise() que se encarga de abrir la base de datos en modo lectura y escritura y de lanzar el proceso de migraciones a

través del método `runMigration()` de la clase `DatabaseMigrationService`. Este método espera dos argumentos: la base de datos donde se ejecutarán las migraciones y los archivos de migración.

Como podemos observar, obtenemos una instancia de la clase `DatabaseMigrationService` a través de `Injector`. Esto es así porque el paquete de migraciones solo contiene dicha clase, por lo que se ha optado por registrarla en el inyector de dependencias para poder obtenerla como singleton.

11.3. Bienvenida a la aplicación

La primera vez que el usuario hace uso de la aplicación, como parte del **CU-001**, accede a una vista donde recibe información sobre SITEICA y, tras la aceptación de las condiciones, selecciona su provincia como parte de su registro en el sistema.

Como hemos visto en la clase `SiteicaApp`, raíz del árbol de widgets, se muestra la vista `WelcomePage()` si no existe usuario en la base de datos local. Este widget es el encargado de dar la bienvenida al usuario aportando información de interés de una manera muy visual. Para ello, se ha utilizado como apoyo el complemento `Intro Slider` de Flutter.

El paquete nos ofrece una clase llamada `IntroSlider`, un widget que nos permite mostrar el carrusel de diapositivas y configurar su aspecto. Esta clase espera un array de objetos de tipo `Slide`, de forma que cada uno de ellos será un elemento del carrusel que se muestra al usuario.

Se han definido las diapositivas, junto con su contenido y aspecto, en el archivo `welcome_slides.dart`. La clase `Slide` nos permite definir un título y una descripción, así como una imagen que represente el contenido de lo que se trata de informar. También es posible cambiar su aspecto modificando el color de fondo con la propiedad `'backgroundColor'`, así como los estilos de los textos con `'styleTitle'` y `'styleDescription'`.

```
import 'package:flutter/material.dart';
import 'package:intro_slider/slide_object.dart';

List<Slide> welcomeSlides = [
  new Slide(
    title: "Bienvenido/a a SITEICA",
    description: "Gracias por descargar la aplicación de rastreo SITEICA."
      "Con su uso estás ayudando a mantener controlada la enfermedad infecciosa.",
    pathImage: "assets/images/thanks.png",
    backgroundColor: Colors.white,
    styleTitle: _welcomeTitleStyle,
    styleDescription: _welcomeDescriptionStyle,
  ),
  new Slide(...),
  new Slide(...),
  new Slide(...), // Slide
];
```

Ilustración 32: Lista de diapositivas de bienvenida.

Como los estilos de los títulos y las descripciones son comunes a todas las diapositivas, se han definido por separado como constantes. En Flutter los estilos de texto se definen a través de la clase `TextStyle` y sus propiedades, donde podemos asignar tanto la fuente como el tamaño, así como otras características del texto.

```
const TextStyle _welcomeTitleStyle = TextStyle(
  color: Color(0xff000000),
  fontSize: 30.0,
  fontWeight: FontWeight.bold,
  fontFamily: 'RobotoMono'); // TextStyle
const TextStyle _welcomeDescriptionStyle = TextStyle(
  color: Color(0xff000000),
  fontSize: 18.0,
  fontWeight: FontWeight.normal,
  fontFamily: 'RobotoMono'); // TextStyle
```

Ilustración 33: Constantes de estilo para textos de las diapositivas.

Una vez definido el contenido de las diapositivas, es turno de configurar la clase `IntroSlider` en nuestra vista llamada `WelcomePage`. Para ello, solo necesitamos asignar la lista `welcomeSlides` a la propiedad 'slides' y configurar el aspecto con el resto de propiedades.

```
@override
Widget build(BuildContext context) {
  return new IntroSlider(
    slides: this.slides,
    onDonePress: this.onDonePress,
    renderSkipBtn: renderSkipBtn(),
    colorSkipBtn: Color(0x33000000),
    highlightColorSkipBtn: Color(0xff000000),
    renderNextBtn: renderNextBtn(),
    renderDoneBtn: renderDoneBtn(),
    colorDoneBtn: Color(0x33000000),
    highlightColorDoneBtn: Color(0xff000000),
    colorDot: Color(0x336C63FF),
    colorActiveDot: Color(0xff6C63FF),
    sizeDot: 13.0,
    hideStatusBar: true,
    backgroundColorAllSlides: Colors.grey,
  ); // IntroSlider
}
```

Ilustración 34: Método `build()` de la vista `WelcomePage`.

Entre ellas se encuentran 'renderSkipBtn', 'renderNextBtn' y 'renderDoneBtn', que nos permiten definir un widget para la visualización de los botones de navegación entre diapositivas. Estos widgets se han definido como iconos mediante la clase Icon que nos permite seleccionar iconos de Material Design a través de sus identificadores así como asignarles un color específico.

```
Widget renderDoneBtn() {
  return Icon(
    Icons.done,
    color: Color(0xff6C63FF),
  ); // Icon
}
```

Ilustración 35: Widget que devuelve un objeto Icon.

También cabe destacar la propiedad 'onDonePress' que espera una función que defina el comportamiento de la pulsación del último botón de la presentación. En este caso, se ha definido una función onDonePress() que se encarga de navegar a la pantalla de registro. En Flutter, la navegación se realiza a través de la Navigator, un widget que maneja un conjunto de widgets hijo a través de un comportamiento de pila.

```
void onDonePress() {
  Navigator.pushReplacement(
    context,
    MaterialPageRoute(builder: (context) => RegisterPage()),
  );
}
```

Ilustración 36: Navegar en la pila de navegación a RegisterPage.

A través de las configuraciones del carrusel y el contenido que se ha elegido, el aspecto final de la vista de bienvenida puede verse en la siguiente ilustración.

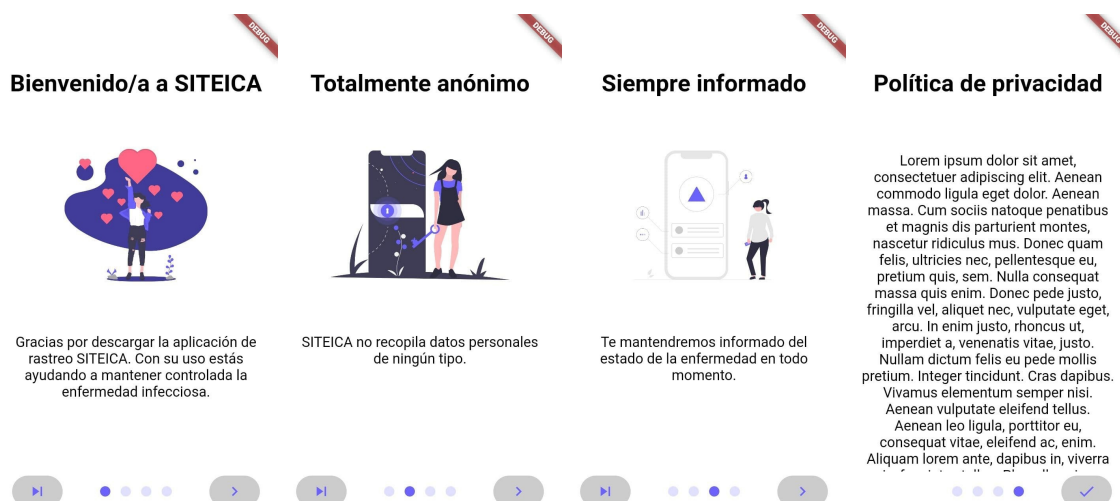


Ilustración 37: Pantalla de bienvenida de Siteica.

11.4. Registro

El **CU-001** continua con el registro de usuario, paso que se completa en la vista RegisterPage. Lo primero que este widget ha de obtener es la lista de provincias de la base de datos local para insertarlas en el selector que permita al usuario escoger una.

```
const String provinceTableName = 'province';

class ProvinceService {
  Future<List<Province>> getProvinces() async {
    Database _database = await openDatabase(DB_NAME, version: 1);
    List<Map> results = await _database.query(
      provinceTableName,
      where: 'deleted = ?',
      whereArgs: [0],
    );

    return results.map((e) => Province.fromJson(e)).toList();
  }
}
```

Ilustración 38: Clase ProvinceService y método para obtener provincias.

Las operaciones relativas a la entidad 'province' se encuentran en el archivo province_service.dart. El método getProvinces() es el que nos devuelve una lista de provincias, abriendo una conexión a la base de datos y ejecutando una consulta sobre la tabla asociada mediante el paquete Sqflite.

Esta clase es un singleton registrado en el Injector, por lo que lo primero que se ha hecho en RegisterPage es obtener la instancia de dicho servicio. Se han obtenido las provincias desde un método asíncrono de modo que, al finalizar, se notifique al framework que el estado ha cambiado con setState(). De esta forma es posible mostrar un icono que representa que hay un proceso en espera y, en el momento en el que se vuelva a dibujar la vista, ya podemos mostrar el selector con sus opciones.

```
final _provinceService = Injector.appInstance.get<ProvinceService>();
final _userService = Injector.appInstance.get<UserService>();
List<Province> _provinces;
Province _selectedProvince;
bool _continue = false;

getProvinces() async {
  _provinces = await _provinceService.getProvinces();
  setState(() {});
}

@override
void initState() {
  super.initState();
  getProvinces();
}
```

Ilustración 39: Obtener las provincias de la BD.

En el método build() comprobamos el contenido de la variable _provinces. En caso de estar aún a nulo, el método devuelve el widget CommonProgressIndicator() para transmitir que la aplicación está trabajando. Si ya tenemos contenido, se construye la interfaz de usuario para la vista del registro.

De forma general, las interfaces se crean a través de columnas y filas como elementos primordiales, por lo que resulta muy útil identificar cuáles de estos elementos necesita nuestra vista con el objetivo de organizar los elementos. En el caso del registro, como se muestra en la siguiente ilustración, se han definido una serie de elementos dispuestos en una única columna.

El siguiente paso es analizar cada fila. Vemos que, en general, son elementos apilados uno sobre otro en la columna. Para ello basta con incluir los widgets deseados, como textos con la clase Text(), como elementos hijo de la columna principal en su propiedad 'children'. La excepción la encontramos en los botones, donde encontramos varios elementos en una misma fila.

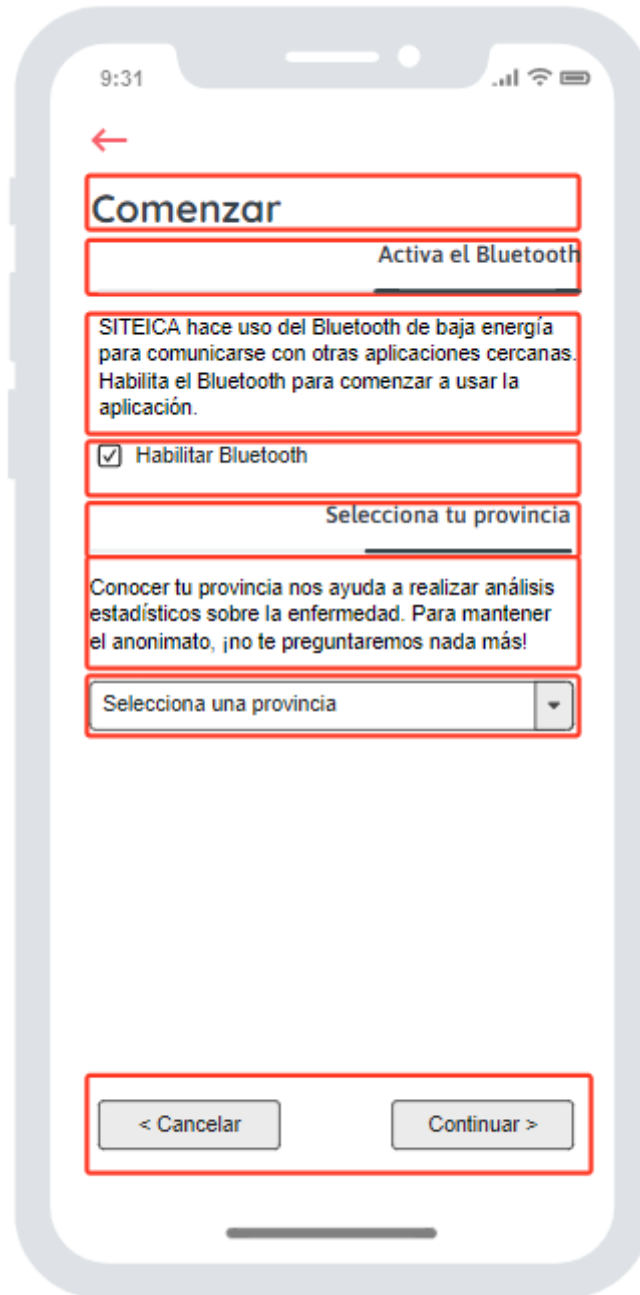


Ilustración 40: Elementos de columna resaltados en rojo.

En azul se han resaltado los elementos de una misma fila que pueden representar en Flutter usando la clase `Row()` y su propiedad `'children'`. En este caso los elementos que compondrán la fila serán widgets de tipo botón. Para este caso se ha usado la clase `ElevatedButton()`, un tipo concreto de botón con elevación de Material Design.

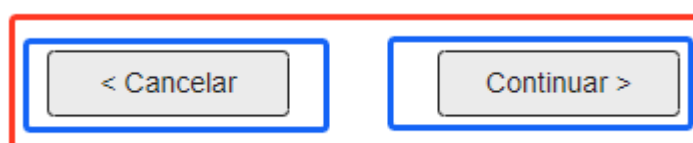


Ilustración 41: Elementos de fila resaltados en azul.

Como podemos observar, se ha utilizado la clase `Scaffold()` como raíz de la vista puesto que implementa la estructura visual básica de Material Design, incluyendo la barra navegación superior en su propiedad `'appBar'`. En el cuerpo de la interfaz se ha optado por el uso de `SingleChildScrollView()` puesto que, en resoluciones más bajas, implementa de forma automática el desplazamiento vertical para la visualización del contenido al completo. `Padding()`, por su parte, ofrece la capacidad de añadir márgenes.

```
@override
Widget build(BuildContext context) {
  if (_provinces == null) {
    return CommonProgressIndicator();
  }

  _selectedProvince ??= _provinces.first;

  return Scaffold(
    appBar: AppBar(
      title: Text("Comenzar"),
    ), // AppBar
    body: SingleChildScrollView(
      child: Padding(
        padding: const EdgeInsets.all(8.0),
        child: Column(...), // Column
      ), // Padding
    ), // SingleChildScrollView
  ); // Scaffold
}
```

Ilustración 42: Método build de la clase RegisterPage.

En la columna, como hemos visto, se configuran los elementos que se han mostrado en el prototipo. Primero, se ha configurado la sección relativa a la activación del Bluetooth a través del widget `CommonTitle()`, creado específicamente para mostrar títulos con un separador inferior de 1 pixel de alto, seguido de un texto explicativo con `Text()` y un botón que nos permita que el usuario habilite el Bluetooth.

```

CommonTitle(title: "Activa el Bluetooth"),
Text(
  "SITEICA hace uso del Bluetooth de baja energía para comunicarse"
  " con otras aplicaciones cercanas. Habilita el Bluetooth para"
  " comenzar a usar la aplicación.",
), // Text
Padding(
  padding: const EdgeInsets.all(8.0),
  child: ElevatedButton(
    style: raisedButtonStyle,
    onPressed: _enableBluetooth,
    child: Text('Habilitar Bluetooth'),
  ), // ElevatedButton
), // Padding

```

Ilustración 43: Elementos de UI para habilitar el Bluetooth.

La pulsación del botón responde al evento `_enableBluetooth`, función privada que habilita el Bluetooth del terminal y, al realizarse la activación, habilita el botón que nos permite continuar con el registro.

```

Future<void> _enableBluetooth() async {
  BluetoothEnable.enableBluetooth.then((value) {
    if (value == "true") {
      setState(() {
        _continue = true;
      });
    }
  });
}

```

Ilustración 44: Método privado para habilitar el Bluetooth.

La columna continua con los elementos de UI relativos a la selección de la provincia. Para el selector se ha utilizado el widget `DropDownButton()`, elemento que permite elegir al usuario un elemento de una lista dada. Esta clase, en su propiedad `'onChanged'`, espera una función que se ejecuta cuando el valor del elemento ha cambiado. En dicha función se establece, entonces, el valor de la variable `_selectedProvince` para almacenar el valor seleccionado por el usuario.


```
CommonTitle(title: "Selecciona tu provincia"),
Text(
  "Conocer tu provincia nos ayuda a realizar análisis estadísticos sobre la enfermedad."
  "Para mantener el anonimato, ¡no te preguntaremos nada más!",
), // Text
DropDownButton<Province>(
  isExpanded: true,
  value: _selectedProvince,
  onChanged: (Province newValue) {
    setState(() {
      _selectedProvince = newValue;
    });
  },
  items: _provinces
    .map<DropDownMenuItem<Province>>((Province value) {
      return DropDownMenuItem<Province>(
        value: value,
        child: Text(value.provinceName),
      ); // DropDownMenuItem
    }).toList(),
), // DropDownButton
```

Ilustración 45: Elementos de UI para la selección de provincia.

Y, finalmente, se ha configurado la fila de botones. Como hemos mencionado anteriormente, se han utilizado los widgets `ElevatedButton()` como botones. En el caso del botón de cancelar, en su evento 'onPressed', se navega hacia atrás en la pila de navegación.

```
Padding(
  padding: const EdgeInsets.only(top: 18.0),
  child: Row(
    mainAxisAlignment: MainAxisAlignment.spaceAround,
    children: [
      ElevatedButton(
        style: raisedButtonStyle,
        onPressed: () {
          Navigator.pop(context);
        },
        child: Text('Cancelar'),
      ), // ElevatedButton
      ElevatedButton(
        style: raisedButtonStyle,
        onPressed: _continue ? _addUser : null,
        child: Text('Continuar'),
      ), // ElevatedButton
    ],
  ), // Row
), // Padding
```

Ilustración 46: Fila con los botones de la vista.

El botón de continuar se activa cuando el Bluetooth se ha habilitado a través del botón asociado, momento en el que la variable `_continue` pasa a tener el valor de verdadero. Una vez

habilitado, el usuario pulsa en dicho botón y se activa el evento `_addUser` que es el encargado de registrar a dicho usuario con la provincia seleccionada y asignándole un UUID.

```
_addUser() async {
  await _userService.addUser(provinceId: _selectedProvince.id);
  Navigator.pushReplacement(
    context,
    MaterialPageRoute(builder: (context) => StartPage()),
  );
}
```

Ilustración 47: Método privado encargado de añadir al usuario.

El método `addUser()` del servicio `UserService()` es el encargado de almacenar el registro en la tabla de usuarios. Internamente crea un UUID nuevo, de modo que no es necesario pasárselo vía parámetro.

Una vez registrado al usuario, se reemplaza la pila de navegación con la vista `StartPage()`, así se evita que se pueda navegar hacia rutas anteriores.

11.5. Configurar la aplicación como emisor BLE

Como se ha mencionado en diferentes ocasiones, el momento del encuentro entre dos aplicaciones es uno de los puntos cruciales de la aplicación. Lo primero que ha de conseguirse es convertir la aplicación en un emisor BLE que se encargue de distribuir su UUID de encuentro en dicho instante, funcionalidad que se ha conseguido a través del plugin de Flutter llamado Beacon Broadcast.

Dicha funcionalidad, al igual que todo lo que esté relacionado con la lógica de negocio relativa a BLE, se ha definido en el archivo `ble_service.dart`. Aquí se centralizan las operaciones de emitir y recibir paquetes de modo que, si en un futuro se decidiera optar por otras librerías para el manejo de los frames de datos a través de Bluetooth de baja energía, bastaría con modificar los métodos de este servicio.

```
import 'package:beacon_broadcast/beacon_broadcast.dart';
```

Ilustración 48: Importar la librería beacon_broadcast.

Una vez incorporada la dependencia al proyecto del paquete `beacon_broadcast`, el primer paso para usar dicha librería es importar la siguiente referencia:

Y, a partir de este punto, ya es posible emitir mensajes a través de BLE. Para ello, en la capa de servicio de BLE, se ha definido una función llamada `startBroadcast(User _user)` que se encarga del proceso completo visto en el DA Generación de identificadores de encuentros, además de iniciar la emisión del UUID del terminal.

Este método, que espera el usuario actual de la aplicación, comienza obteniendo la instancia del servicio `EncounterSeedService` desde el Injector. Este servicio se encarga de la lógica de negocio relacionada con los identificadores de encuentro, ofreciendo operaciones de búsqueda e inserción en la base de datos local. De esta forma, se realiza una búsqueda del

UUID de intercambio para el usuario actual. Dicho método, `getEncounterSeed(User _user)`, devuelve un nulo si no se ha encontrado ningún registro.

```
/// Gestión de semillas de intercambio
final _encounterSeedService =
    Injector.appInstance.get<EncounterSeedService>();
EncounterSeed _encounterSeed = await _encounterSeedService.getEncounterSeed(
    _user,
);
```

Ilustración 49: Singleton de EncounterSeedService y llamada a getEncounterSeed.

En caso de que aún no tenga un UUID de encuentro o si el UUID ha superado los 15 minutos de vigencia, se crea un nuevo UUID para dicho usuario y se almacena en su base de datos, en la entidad 'encounter_seed'.

```
/// Si es null o está caducada (15 mins en constants)
/// Creamos nueva semilla y la almacenamos
if (_encounterSeed == null ||
    timeExceeded(_encounterSeed.date, ENCOUNTER_SEED_EXPIRATION)) {
    _encounterSeed = await _encounterSeedService.addEncounterSeed(_user);
}
```

Ilustración 50: Creación de una semilla de encuentro.

Una vez obtenido el UUID del encuentro en el momento actual, es posible comenzar la emisión BLE. A través de la clase `BeaconBroadcast()`, configuramos el dispositivo como emisor asignando dicho UUID como identificador del "beacon" y se inicia la transmisión con el método `start()`.

Como podemos observar en la siguiente ilustración, la configuración del emisor BLE incluye la asignación de los valores mayor y menor. El valor mayor permite identificar un subgrupo de beacons dentro de un grupo más amplio, mientras que el valor menor se utiliza como número de identificación específico. En el caso del actual proyecto es suficiente con dotar de valores únicos a los UUID, aunque la librería obliga a incluir los valores mayor y menor para poder emitir vía BLE.

```
/// Variables para la emisión vía BLE
BeaconBroadcast beaconBroadcast = BeaconBroadcast();
beaconBroadcast
    .setUUID(_encounterSeed.seedUuid)
    .setMajorId(1)
    .setMinorId(100)
    .start();
```

Ilustración 51: Emisión de información a través de BLE.

Finalmente se crea un proceso periódico que, cada 60 segundos, comprueba si el UUID ha expirado. En cuyo caso, crea un nuevo UUID de encuentro en la base de datos local y vuelve a configurar la emisión BLE con el nuevo dato.

```
Timer.periodic(Duration(seconds: 60), (timer) async {
  if (timeExceeded(_encounterSeed.date, ENCOUNTER_SEED_EXPIRATION)) {
    _encounterSeed = await _encounterSeedService.addEncounterSeed(_user);
    beaconBroadcast
      .setUUID(_encounterSeed.seedUuid)
      .setMajorId(1)
      .setMinorId(100)
      .start();
    print("New seed ${_encounterSeed.seedUuid}");
  }
}); // Timer.periodic
```

Ilustración 52: Temporizador que controla la caducidad de semillas.

11.6. Configurar la aplicación como observador BLE

El complemento Beacons Plugin de Flutter permite habilitar el escaneo de paquetes a través de BLE, así como leer valores de proximidad.

Previo al uso del plugin, se han realizado configuraciones específicas para Android e iOS de cara a habilitar la funcionalidad de escanear como un proceso en segundo plano.

```
import com.umair.beacons_plugin.BeaconsPlugin
import io.flutter.embedding.android.FlutterActivity

class MainActivity: FlutterActivity() {
  override fun onPause() {
    super.onPause()
    //Habilitar proceso en segundo plano para escanear BLE
    BeaconsPlugin.startBackgroundService(this)
  }

  override fun onResume() {
    super.onResume()
    //Finalizar proceso en segundo plano para escanear BLE
    BeaconsPlugin.stopBackgroundService(this)
  }
}
```

Ilustración 53: Manejo del servicio de observación BLE en segundo plano para Android.

Para Android, se ha modificado el archivo MainActivity.kt bajo la carpeta android del proyecto de Flutter, en concreto en la ruta “android/app/src/main/kotlin/es/uned/siteica_user/”. La clase Activity ofrece una serie de métodos de tipo callback que permiten

conocer si el estado de la App ha cambiado. Los métodos del ciclo de vida que se han utilizado han sido `onPause` y `onResume`. El primero de ellos ocurre cuando la actividad ya no está en primer plano, momento en el que se habilita el servicio en segundo plano del plugin. De un modo análogo, el evento `onResume` ocurre cuando se reanuda la actividad y es entonces cuando finalizamos el proceso en segundo plano.

En el caso de iOS, debemos habilitar el uso de `CoreLocation`, framework que está incluido dentro del SDK de iOS y que proporciona servicios que determinan la ubicación geográfica del dispositivo, así como la capacidad de detectar emisores BLE cercanos. Para ello, en el archivo `AppDelegate.swift`, debemos solicitar autorización para el uso de dichos servicios importando la librería, instanciando un objeto de la clase `CLLocationManager` y solicitando autorización a través del método `requestAlwaysAuthorization()`.

```
import UIKit
import Flutter
import CoreLocation

@UIApplicationMain
@objc class AppDelegate: FlutterAppDelegate {

    let locationManager = CLLocationManager()

    override func application(
        _ application: UIApplication,
        didFinishLaunchingWithOptions launchOptions: [UIApplication.LaunchOptionsKey: Any]?
    ) -> Bool {

        locationManager.requestAlwaysAuthorization()
        GeneratedPluginRegistrant.register(with: self)

        return super.application(application, didFinishLaunchingWithOptions: launchOptions)
    }
}
```

Ilustración 54: Autorización del servicio Location en iOS.

Para llamar a `requestAlwaysAuthorization()`, el SDK obliga a tener las siguientes claves en el archivo `Info.plist`:

- `NSLocationAlwaysAndWhenInUseUsageDescription`.
- `NSLocationWhenInUseUsageDescription`.
- `NSLocationAlwaysUsageDescription`.

Además de las claves, debemos aportar el mensaje que se muestra al usuario cuando se le notifica por qué la aplicación solicita dicho permiso.

```
<key>NSLocationAlwaysAndWhenInUseUsageDescription</key>
<string>SITEICA necesita permisos de localización para escanear dispositivos cercanos.</string>
<key>NSLocationWhenInUseUsageDescription</key>
<string>SITEICA necesita permisos de localización para escanear dispositivos cercanos.</string>
<key>NSLocationAlwaysUsageDescription</key>
<string>SITEICA necesita permisos de localización para escanear dispositivos cercanos.</string>
```

Ilustración 55: Claves necesarias en Info.plist para iOS.

Una vez se ha realizado la configuración del plugin, ya es posible poner al dispositivo en modo observador y escanear los paquetes de datos BLE cercanos. Para ello solo debemos suscribirnos al stream de datos que nos provee el método `listenToBeacons(StreamController controller)` de la clase `BeaconsPlugin` y, una vez suscritos, leer los datos recibidos a través de BLE cada vez que el stream de datos recibe información.

El proceso relativo al observador BLE se encuentra en el método `startObserver()` que se encuentra en el archivo `ble_service.dart`. Este procedimiento comienza comprobando los permisos relativos a localización puesto que, a partir de Android 10, las aplicaciones que hacen uso de los servicios de ubicación requieren solicitar dichos permisos y establecer un mensaje asociado al motivo de su uso [32]. Estos servicios, en el proyecto actual, son necesarios para escanear dispositivos cercanos. El método encargado de llevar a cabo la comprobación es la función privada `_showDisclosureMessage()`.

```
_showDisclosureMessage() async {
  if (Platform.isAndroid) {
    await BeaconsPlugin.setDisclosureDialogMessage(
      title: "Se requieren permisos de localización",
      message:
        "Esta aplicación recopila información de los encuentros cercanos.");
    await BeaconsPlugin.clearDisclosureDialogShowFlag(false);
  }
}
```

Ilustración 56: Mensaje de permisos de localización para Android.

Después, el método `startObserver()` asigna el controlador del stream de datos para poder realizar la escucha y realiza una suscripción al stream de datos del controlador donde se reciben los paquetes de datos recibidos a través de BLE. En el momento en el que se recibe un paquete se extraen sus propiedades y, en caso de estar a 2 metros o menos de distancia, se almacena el encuentro en la base de datos local.

El método privado `_addEncounter` es el encargado de almacenar el encuentro, revisando que el frame de datos recibido contenga información. Una vez se ha comprobado que se ha recibido algún dato a través de BLE, se obtiene la posición geográfica del dispositivo, su semilla actual y se deserializa el frame recibido a través de BLE a un objeto de tipo `Beacon` que contiene todas sus propiedades.

```

/// Asignar el controlador del stream de datos
BeaconsPlugin.listenToBeacons(_streamController);

/// Suscripción al stream
_streamController.stream.listen(
  (data) async {
    print("Beacon $data");
    _addEncounter(data, _user, _encounterService, _encounterSeedService);
  },
  onDone: () {},
  onError: (error) {
    print("Ha ocurrido un error: $error");
  },
);

```

Ilustración 57: Suscripción al stream de datos del observador BLE.

Las propiedades del objeto Beacon nos proveen toda la información necesaria para identificar dicho encuentro. Entre ellas se encuentra la distancia a la que se han encontrado ambos teléfonos, dato que se ha usado para comprobar si el encuentro se produce a dos metros o menos. De ser así, se almacena el encuentro a través del método correspondiente del servicio EncounterService.

```

if (_distance <= ENCOUNTER_MIN_DISTANCE) {
  _encounterService.addEncounter(
    ownSeedId: _encounterSeed.id,
    encounterSeedUuid: _beacon.uuid,
    latitude: _position.latitude,
    longitude: _position.longitude,
    date: DateTime.now().millisecondsSinceEpoch,
    distance: _distance,
  );
}

```

Ilustración 59: Almacenamiento del encuentro en BD local.

Finalmente se habilita la ejecución en segundo plano mediante la instrucción `BeaconsPlugin.runInBackground(true)` y se inicia la monitorización comprobando, solo en Android, que el escáner está configurado y listo para usarse.

Con el objetivo de realizar una prueba previa, se ha lanzado la aplicación en dos dispositivos diferentes, un móvil Xiaomi Redmi 4X y una Energy Tablet Pro 4, y se ha comprobado que ambos dispositivos emiten y reciben los paquetes de datos en cuanto se encuentran. Para ello se ha usado el método `print()` que permite escribir en consola y, en la suscripción al stream de datos, se han recogido los paquetes y se han impreso por pantalla para comprobar su contenido.

```

I/flutter (19363): Beacons DataReceived: {
I/flutter (19363):   "name": "null",
I/flutter (19363):   "uuid": "f27fb3e7-7409-4c77-b698-1660f4824944",
I/flutter (19363):   "macAddress": "5F:09:11:06:B9:82",
I/flutter (19363):   "major": "1",
I/flutter (19363):   "minor": "100",
I/flutter (19363):   "distance": "1.37",
I/flutter (19363):   "proximity": "Near",
I/flutter (19363):   "scanTime": "04 May 2021 12:07:50 PM",
I/flutter (19363):   "rssi": "-67",
I/flutter (19363):   "txPower": "-59"
I/flutter (19363): }

```

Ilustración 61: Datos recibidos por un terminal en un encuentro.

11.7. Inicio

Esta vista se compone de una columna con una serie de elementos individuales que se han organizado en distintos métodos privados, uno por sección, con el objetivo de organizar al código de una manera eficiente.

Estos métodos privados hacen referencia a cada una de las secciones de la vista: `_riskEncounterData()` compone el apartado relativo al análisis de contactos de riesgo, `_systemStatus()` refleja el estado del sistema y `_positiveNotification()` es la sección encargada de iniciar la notificación de un positivo.

```

List<Widget> _content = <Widget>[];
_content.addAll(_riskEncounterData());
_content.addAll(_systemStatus());
_content.addAll(_positiveNotification());

```

Ilustración 62: Lista de widgets que componen las secciones de inicio.

Todos ellos devuelven una lista de widgets, de modo que se ha creado una lista de elementos en la que se añaden todas las secciones y se asigna dicha lista a la propiedad 'children' de la columna. Estas secciones se detallan en los siguientes subapartados.


```
return Scaffold(
  appBar: AppBar(
    title: Text("Inicio"),
  ), // AppBar
  body: SingleChildScrollView(
    child: Padding(
      padding: const EdgeInsets.all(8.0),
      child: Column(
        crossAxisAlignment: CrossAxisAlignment.start,
        mainAxisAlignment: MainAxisAlignment.start,
        children: _content,
      ), // Column
    ), // Padding
  ), // SingleChildScrollView
); // Scaffold
```

Ilustración 63: Contenido de la vista de inicio.

11.7.1. Contactos de riesgo

La función `_checkSelfRisk()` realiza el análisis de riesgo del usuario de la aplicación siempre que el último análisis se haya hecho hace más de 24 horas. Para ello se obtiene de la base de datos local el último registro de la tabla 'risk_encounter_analysis' a través del método `getLastRiskEncounterAnalysis()` del servicio que gestiona las operaciones sobre dicha tabla, `RiskEncounterAnalysisService`.

```
Future<RiskEncounterAnalysis> getLastRiskEncounterAnalysis() async {
  Database _database = await openDatabase(DB_NAME, version: 1);

  List<Map> _results = await _database.rawQuery(
    'SELECT * FROM $riskEncounterAnalysisTableName ORDER BY date DESC LIMIT 1',
  );
  List<RiskEncounterAnalysis> _analysis =
    _results.map((e) => RiskEncounterAnalysis.fromJson(e)).toList();
  return _analysis.isEmpty ? null : _analysis.first;
}
```

Ilustración 64: Obtención del último análisis de riesgo de la BD.

Un valor nulo devuelto por la función nos indica que aún no se han realizado análisis en el dispositivo. Si ya existe un análisis previo, se comprueba si ha excedido las 24 horas desde que se produjo. En ambos casos, se realiza el análisis de riesgo consultando los identificadores de encuentro del propio usuario de los últimos 14 días a través del método `getEncounterSeeds()`. Con la lista de identificadores del usuario, lo siguiente es obtener los identificadores de encuentros de riesgo que se proveen desde el servidor central y comprobar si alguno de los identificadores de encuentro del usuario se encuentra entre alguno de los encuentros obtenidos desde el servidor.

```

_checkSelfRisk(User _user) async {
  RiskEncounterAnalysis _analysis =
    await _riskEncounterAnalysisService.getLastRiskEncounterAnalysis();

  if (_analysis == null ||
    timeExceeded(_analysis.date, RISK_ANALYSIS_TIME_PERIOD)) {
    List<EncounterSeed> _encounterSeeds =
      await _encounterSeedService.getEncounterSeeds(_user);
    List<RiskEncounter> _myRiskEncounters = await _riskEncounterService
      .getSelfEncounterRisk(_user, _encounterSeeds);
    _riskFound = _myRiskEncounters
      .any((element) => element.duration >= RISK_DURATION_INTERVAL);
    _riskEncounterAnalysisService.addRiskEncounterAnalysis(_riskFound);
  } else {
    _riskFound = _analysis.riskFound == 1;
  }
  setState(() {});
}

```

Ilustración 66: Análisis de riesgo.

Esta búsqueda nos retorna una lista de elementos de tipo RiskEncounter, que son aquellos registros de encuentros de riesgo sobre los que se ha encontrado una coincidencia con los UUID del usuario. Una vez obtenidos dichos registros, se comprueba si la duración de alguno de ellos supera el límite establecido en la constante RISK_ANALYSIS_TIME_PERIOD, fijada en 15 minutos como se establece en el **CU-002**, en cuyo caso se establece el valor de la variable booleana _riskFound y se almacena su resultado como un nuevo registro sobre la tabla 'risk_encounter_analysis'.

Con el valor de la variable _riskFound se compone la interfaz de esta sección. En caso de que se haya obtenido un valor verdadero se notifica al usuario con un mensaje sobre su posible exposición y se informa a través de un color rojo con el objetivo de que sea más identificable.

```

List<Widget> _riskEncounterData() {
  return [
    CommonTitle(title: "Contactos de riesgo"),
    InformationTitle(
      title: _riskFound
        ? "Se ha detectado contacto de riesgo."
        : "No se han detectado contactos de riesgo.",
      warning: _riskFound,
    ),
    Text(
      _riskFound
        ? "Te recomendamos que contactes con tu centro de salud y sigas "
          "las recomendaciones que te ofrezcan."
        : "Serás informado en caso de que se registre un posible contacto "
          "de riesgo tras el análisis automatizado.",
    ),
    Padding(
      padding: const EdgeInsets.only(bottom: 21.0),
    ) // Padding
  ];
}

```

Ilustración 67: Widgets que componen la sección de contactos de riesgo.

11.7.2. Estado del sistema

Como parte de la vista de inicio de la aplicación se ha planteado informar al usuario el estado del sistema, indicando si actualmente está activo. Este control se ha realizado a través de la comprobación del Bluetooth, que es el servicio principal con el cual es posible transmitir y enviar identificadores de encuentro.

Se ha definido una función privada en la que, gracias de nuevo al complemento `bluetooth_enable`, se realiza una comprobación del estado del Bluetooth. Si está ya activado, nuestra pantalla informará de que el sistema está activo. En caso contrario se muestra un mensaje de advertencia y se solicita de forma inmediata la activación.

```

_checkBluetoothAndEnable() async {
  BluetoothEnable.enableBluetooth.then((value) {
    if (value == "true") {
      setState(() {
        _bluetoothEnabled = true;
      });
    }
  });
}

```

Ilustración 68: Método para la comprobación del Bluetooth.

El método que se encarga de realizar la comprobación se ejecuta en `initState()` de modo que sea lo primero que se analiza. La variable `_bluetoothEnabled` se ha utilizado para mostrar un mensaje de advertencia en caso de estar a falso.

```
List<Widget> _systemStatus() {
  List<Widget> _widgetList = <Widget>[
    CommonTitle(title: "Estado del sistema"),
    InformationTitle(
      title: _bluetoothEnabled
        ? "El sistema está actualmente activo"
        : "El sistema no está activo.",
      warning: !_bluetoothEnabled,
    ), // InformationTitle
    Text(
      "Siteica hace uso del Bluetooth para comunicarse con dispositivos "
      "cerca y mantenerse protegido.",
    ) // Text
  ]; // <Widget>[]

  if (!_bluetoothEnabled) {...}

  _widgetList.add(Padding(...)); // Padding

  return _widgetList;
}
```

Ilustración 69: Widgets que componen la sección del estado del sistema.

Como apoyo al usuario, se ha incluido un botón que le permita habilitar el Bluetooth desde la propia aplicación. Este botón solo se muestra si la variable `_bluetoothEnabled` está a falso.

```
if (!_bluetoothEnabled) {
  _widgetList.add(
    Padding(
      padding: const EdgeInsets.all(8.0),
      child: ElevatedButton(
        style: raisedButtonStyle,
        onPressed: _checkBluetoothAndEnable,
        child: Text('Habilitar Bluetooth'),
      ), // ElevatedButton
    ), // Padding
  );
}
```

Ilustración 70: Botón para habilitar de forma manual el Bluetooth.

11.7.3. Notificación de positivo

Desde la vista de inicio de la aplicación también es posible acceder al flujo de la notificación de un caso positivo. Esta sección es fija y ofrece iniciar el proceso a través de un botón habilitado para ello.

```
ElevatedButton(  
  style: raisedButtonStyle,  
  onPressed: () {  
    Navigator.push(  
      context,  
      MaterialPageRoute(builder: (context) => NotificationPage()),  
    );  
  },  
  child: Text('Notificar'),  
) // ElevatedButton
```

Ilustración 71: Botón y su evento para la notificación de positivo.

La clase NotificationPage() contiene dos elementos de entrada de datos. El primero de ellos, habilitado para el código de diagnóstico, es un widget de tipo TextField() que permite al usuario introducir texto así como personalizar su comportamiento y aspecto a través de sus distintas propiedades o responder a ciertos eventos.

```
TextField(  
  controller: inputController,  
  keyboardType: TextInputType.number,  
  decoration: InputDecoration(  
    border: OutlineInputBorder(),  
    hintText: 'Introduce tu código de diagnóstico',  
  ), // InputDecoration  
  inputFormatters: [  
    FilteringTextInputFormatter.digitsOnly,  
    NotificationNumberFormatter(),  
  ],  
  onChanged: (text) {  
    setState(() {  
      _otpCorrect = inputController.text.length >= 12;  
    });  
  },  
) // TextField
```

Ilustración 72: Elemento TextField para el código de diagnóstico.

Esta clase nos permite establecer el tipo de teclado que se mostrará cuando el foco entre en el control mediante su propiedad 'keyboardType', lo que ha permitido establecer el teclado a numérico de cara a que resulte más intuitivo introducir el OTP. Se ha combinado esta

funcionalidad con la propiedad 'inputFormatters' que permite realizar una validación del contenido además de permitir sobrescribir su formato. El valor `FilteringTextInputFormatter.digitsOnly` solo acepta valores numéricos y `NotificationNumberFormatter` es una clase que extiende de `TextInputFormatter` y que sobrescribe el contenido para agregar guiones tras el tercer y séptimo dígito.

El segundo de los elementos de entrada es un selector de fechas para el que se ha utilizado la función `showDatePicker`. Este método muestra un diálogo que contiene el selector de fecha con los estilos de Material Design y el valor que devuelve es un objeto `DateTime` con la fecha seleccionada por el usuario.

Se ha configurado la apertura del diálogo desde una función privada llamada `_selectDate` donde se llama al método `showDatePicker` y se espera por su resultado con la instrucción `await`. El parámetro 'initialDate' permite establecer una fecha inicial desde la que se inicializa el calendario y al que se le ha pasado la fecha actual del sistema. Los parámetros `firstDate` y `lastDate` permiten configurar las fechas de inicio y fin del calendario.

```
_selectDate() async {
  final DateTime picked = await showDatePicker(
    context: context,
    initialDate: _selectedDate,
    firstDate: DateTime(2015, 8),
    lastDate: DateTime(2101),
  );

  if (picked != null && picked != _selectedDate)
    setState(() {
      _selectedDate = picked;
    });
}
```

Ilustración 73: Apertura del calendario y selección de fecha.

Una vez se ha introducido el valor del código de diagnóstico y la fecha, el botón para continuar con el proceso se habilita. Al pulsar sobre él se comprueba si el valor del OTP ya ha sido utilizado realizando una búsqueda sobre la base de datos. Si se encuentra un registro con el mismo valor, la aplicación muestra un aviso en forma de alerta. En caso de que el valor del código sea válido, accedemos a un resumen de la notificación que está a punto de enviarse, concretamente a la vista `notify_confirmation.dart`, momento en el que el usuario puede aceptar o rechazar el proceso.

```

_continue(DateTime _selectedDate, String _otp) async {
  User _user = await _userService.getUser();
  PrivateNotification _notification = await _privateNotificationService
    .searchPrivateNotificationByOtpValue(_user, _otp);

  if (_notification == null) {
    Navigator.push(
      context,
      MaterialPageRoute(
        builder: (context) => NotifyConfirmationPage(
          selectedDate: _selectedDate,
          diagnosticCode: _otp,
        ), // NotifyConfirmationPage
      ), // MaterialPageRoute
    );
  } else {
    _showOtpUsedDialog();
  }
}

```

Ilustración 74: Método que comprueba si el OTP ya está usado.

Si el usuario decide continuar, se ejecuta el método `_continue` donde se genera un registro sobre la tabla 'private_notification' a modo de histórico y se realiza una notificación vía POST al servidor central con el código recién utilizado y los encuentros de los últimos 14 días. En el presente proyecto el método hace una llamada a modo de demostración a una API REST gratuita llamada JSONPlaceholder.

```

_continue(String otpValue, int diagnosticDate) async {
  User _user = await _userService.getUser();
  await _privateNotificationService.addPrivateNotification(
    _user, otpValue, diagnosticDate);
  _notifyToServer(otpValue, diagnosticDate);
  _showDialog();
}

_notifyToServer(String otpValue, int diagnosticDate) async {
  await _apiService.createNotification(otpValue, diagnosticDate);
  List<Encounter> _encounters = await _encounterService.getEncounters();
  await _apiService.uploadEncounters(_encounters);
  _updateEncounters(_encounters);
}

```

Ilustración 75: Acción de confirmación del envío de la notificación.

Finalmente, tras el envío de los encuentros al servidor, se procesa a marcar todos los registros como transmitidos en la base de datos local a fin de evitar que se envíen una segunda vez. El método encargado de llevar dicha actualización a cabo se llama `_updateEncounters()`.

```
_updateEncounters(List<Encounter> _encounters) async {
  _encounters.forEach((element) {
    _encounterService.updateEncounterTransmitted(element);
  });
}
```

Ilustración 76: Actualización de registros de encuentro transmitidos.

11.8. Mapa de infecciones

A través de la opción “Mapa” del menú inferior el usuario puede acceder al mapa de infecciones y visualizar qué encuentros de riesgo han ocurrido cerca de su posición. El archivo map.dart es el encargado de mostrar esta información por pantalla.

El método privado `_getRiskEncounters()` tiene la responsabilidad de cargar los encuentros de riesgo de la base de datos local a través del método `getAllRiskEncounters` de la clase `RiskEncounterService` y crear una lista de marcadores por cada uno de ellos que sirven para poder ser visualizados sobre el mapa. Estos marcadores se crean a partir de la clase `Marker` perteneciente a la librería `flutter_map`. Estos objetos esperan una posición geográfica que asignamos a través de los valores de latitud y longitud de los encuentros de riesgo y una representación del marcador a través de su propiedad 'builder'. En dicha propiedad podemos devolver cualquier objeto de tipo widget y, en este caso, se ha optado por el uso de la clase `Icon` para representar, sobre el mapa, un icono de advertencia en color rojo.

```
_getRiskEncounters() async {
  List<RiskEncounter> _riskEncounters =
    await _riskEncounterService.getAllRiskEncounters();
  _riskEncounters.forEach((element) {
    _riskMarkers.add(Marker(
      width: 80.0,
      height: 80.0,
      point: LatLng(element.latitude, element.longitude),
      builder: (ctx) => Container(
        child: Icon(Icons.warning_amber_outlined,
          color: Colors.red,
          size: 24.0,
          semanticLabel: 'Contacto de riesgo')), // Icon, Container
    )); // Marker
  });
}
```

Ilustración 77: Configuración de marcadores del mapa.

Esta función se ejecuta en cuanto `MapPage` entra en el árbol de widgets, en la sobrecarga del método `initState()`, de modo que comienza a obtener la lista de marcadores antes de que se haya construido la vista.

Además, como puede observarse al final del método, se obtiene la posición del usuario en el momento actual a través de `determinePosition()` perteneciente a la librería `geolocator`. Esto nos permite centrar el mapa sobre las coordenadas del usuario.

Finalmente, la clase `MapPage` retorna en su método `build()` un objeto de tipo `FlutterMap`, perteneciente también a la librería `flutter_map`. A través de su propiedad `'options'` se ha configurado la vista por defecto del mapa, centrándolo sobre las coordenadas anteriormente obtenidas y estableciendo un zoom por defecto. Así mismo, `'layers'` espera una lista de opciones de capas donde se ha configurado el proveedor de mapas `OpenStreetMap` mediante la clase `TileLayerOptions` y los marcadores que se visualizan sobre el mapa con `MarkerLayerOptions`.

```
body: _isLoading
  ? CommonProgressIndicator()
  : Center(
    child: FlutterMap(
      options: MapOptions(
        center: LatLng(_position.latitude, _position.longitude),
        zoom: 16.0,
      ), // MapOptions
      layers: [
        TileLayerOptions(
          urlTemplate:
            "https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png",
          subdomains: ['a', 'b', 'c']), // TileLayerOptions
        MarkerLayerOptions(
          markers: _riskMarkers,
        ), // MarkerLayerOptions
      ],
    ), // FlutterMap
  ), // Center
```

Ilustración 78: Interfaz de usuario de la vista de "Mapa".

11.9. Evolución

La opción “Evolución” del menú inferior muestra los datos generales recogidos sobre la enfermedad, divididos en tres secciones.

La primera de ellas, llamada “Resumen”, informa sobre el total de casos confirmados durante la semana actual en el momento de la consulta, el número total del mes anterior y el dato sobre el acumulado de positivos recogidos por SITEICA. Para la obtención de esta información se han creado dos métodos en `EvolutionService` que realizan la consulta sobre la base de datos. El primero de ellos, `getTotal()`, realiza una suma total de la columna `'totalCases'` en la tabla `'evolution'`, mientras que la segunda función `getTotalBetweenDates()` realiza una suma entre dos fechas dadas por parámetros.

En la clase `EvolutionPage`, donde se encuentra la interfaz de usuario de dicha sección, se hacen uso de estas consultas para recoger los distintos totales que se muestran al usuario. En el caso del total de casos del mes anterior, se ha hecho uso de la clase `DateTime` para obtener la fecha del primer día del mes anterior y el último dentro del método `_getLastMonthTotal()`.

```

Future<int> getTotal() async {
  Database _database = await openDatabase(DB_NAME, version: 1);
  return Sqflite.firstIntValue(await _database
    .rawQuery('SELECT SUM(totalCases) FROM $tableName '));
}

Future<int> getTotalBetweenDates(int _startTime, int _finishTime) async {
  Database _database = await openDatabase(DB_NAME, version: 1);
  return Sqflite.firstIntValue(await _database.rawQuery(
    'SELECT SUM(totalCases) FROM $tableName WHERE date >= ? AND date <= ?',
    [_startTime, _finishTime]
  ));
}

```

Ilustración 79: Consultas para obtener datos de evolución.

De la misma manera, el método `_getLastWeekTotal()` obtiene el primer día de la semana actual para realizar la consulta sobre la base de datos.

En la segunda sección, “Casos diarios”, se ha utilizado la clase `TimeSeriesChart` del complemento `charts_flutter` para la visualización de los casos de los últimos 14 días en forma de gráfica de tiempo. Para construir la serie de datos de la gráfica se ha un objeto de tipo `Series` que provee el complemento. Este objeto espera dos clases que la gráfica usa para su representación. El primero de ellos, que se la llamado `LinearEvolution`, es el tipo de dato que muestra la gráfica. El segundo sirve como tipo de dato que se maneja en el eje X, en esta caso un objeto de tipo `DateTime` al ser una gráfica temporal.

```

_getLastMonthTotal(DateTime now) async {
  var lastDayPrevMonth = new DateTime(now.year, now.month, 0);
  var firstDayPrevMonth = new DateTime(now.year, now.month - 1, 1);
  _lastMonthTotal = await _evolutionService.getTotalBetweenDates(
    firstDayPrevMonth.millisecondsSinceEpoch,
    lastDayPrevMonth.millisecondsSinceEpoch,
  );
}

_getLastWeekTotal(DateTime now) async {
  var firstDayPrevWeek = DateTime(now.year, now.month, now.day)
    .subtract(Duration(days: now.weekday - 1));
  _lastWeekTotal = await _evolutionService.getTotalBetweenDates(
    firstDayPrevWeek.millisecondsSinceEpoch,
    now.millisecondsSinceEpoch,
  );
}

```

Ilustración 80: Obtención de totales del mes y de la semana anterior.

De este modo, se obtienen los registros de los casos totales de los últimos 14 días a través del método `getEvolution()`, se transforman a una lista de objetos de tipo `LinearEvolution` haciendo una conversión de marca de tiempo a `DateTime` y, finalmente, cargamos en el array `_chartData` la única serie de datos que se muestra al usuario, dotándola de un nombre, un color y qué propiedades de `LinearEvolution` son del dominio actual.

```
_getEvolutionData() async {
  List<Evolution> _evolution = await _evolutionService.getEvolution();
  List<LinearEvolution> _linearEvolution = [];
  _evolution.forEach((element) {
    var date = new DateTime.fromMicrosecondsSinceEpoch(element.date * 1000);
    _linearEvolution.add(LinearEvolution(date, element.totalCases));
  });

  _chartData = [
    new charts.Series<LinearEvolution, DateTime>(
      id: 'Casos diarios',
      colorFn: (_, __) => charts.MaterialPalette.blue.shadeDefault,
      domainFn: (LinearEvolution evolution, _) => evolution.date,
      measureFn: (LinearEvolution evolution, _) => evolution.total,
      data: _linearEvolution,
    )
  ];
}
```

Ilustración 81: Configuración de la serie de datos de la gráfica de tiempo.

Una vez obtenidas las series de datos de la gráfica solo queda su visualización. Para ello, dentro del método `build()` de `EvolutionPage`, definimos la sección. Como en ocasiones previas, se ha utilizado el `widgetCommonTitle` para mostrar el título de la sección y, después, se ha dibujado la gráfica utilizando el widget `TimeSeriesChart` al que se le ha asignado `_chartData` como parámetro de entrada. El parámetro `'animate'` nos permite que los datos sobre la gráfica se dibujen con una animación, en vez de salir de golpe, mientras que `'dateTimeFactory'` permite establecer una factoría para la conversión de los objetos de tipo fecha y hora en el eje X de coordenadas.

```
CommonTitle(title: "Casos diarios"),
Padding(
  padding: const EdgeInsets.only(bottom: 12.0),
  child: Container(
    decoration: BoxDecoration(
      color: Colors.black12,
    ), // BoxDecoration
    height: 300,
    padding: const EdgeInsets.symmetric(vertical: 12.0),
    child: charts.TimeSeriesChart(
      _chartData,
      animate: true,
      dateTimeFactory: const charts.LocalDateTimeFactory(),
    ), // charts.TimeSeriesChart
  ), // Container
), // Padding
```

Ilustración 82: Elementos de la interfaz de la sección "Casos diarios".

12. Pruebas

Como parte del ciclo del desarrollo de la solución, se ha realizado los casos de prueba necesarios con el objetivo de comprobar tanto el funcionamiento adecuado del sistema como el cumplimiento de los distintos requisitos recogidos con anterioridad.

12.1. CP-001. Permitir acceso a ubicación.

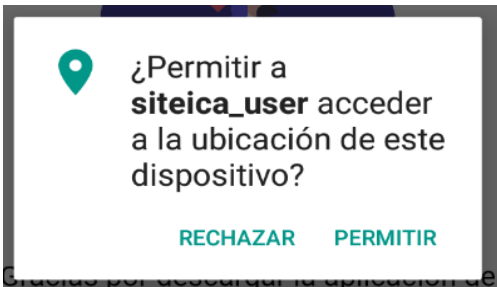
CP-001	Solicitud de permisos de ubicación válida
Caso de uso	RNF-003
Objetivo	Validar el correcto funcionamiento de la solicitud de permisos de ubicación al ejecutar por primera vez la aplicación.
Precondición	Primer uso de la App.
Pasos	1. Abrir la aplicación.
Resultado esperado	Se solicita acceso a la ubicación del dispositivo para la aplicación de SITEICA.
Resultado obtenido	 <p><i>Ilustración 83: Solicitud de permisos de ubicación.</i></p>

Tabla 16: CP-001. Solicitud de permisos de ubicación válida.

12.2. CP-002. Registro.

CP-002	Registro válido
Caso de uso	CU-001
Objetivo	Validar la creación del usuario en el primer uso de la aplicación tras aceptar las condiciones de uso y privacidad y seleccionar una provincia.
Precondición	No existe un usuario anterior.
Pasos	<ol style="list-style-type: none"> 1. Confirmar las condiciones de uso y privacidad. 2. Habilitar del Bluetooth. 3. Seleccionar una provincia. 4. Pulsar en “Continuar” para formalizar el registro.
Resultado esperado	El usuario queda activado en el sistema y accede a la vista de “Inicio” de la aplicación.

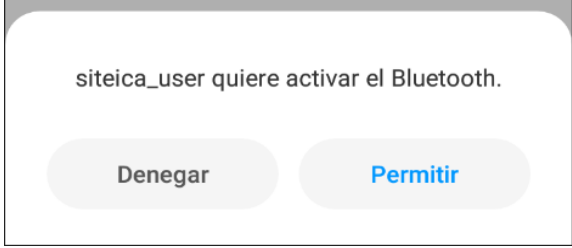
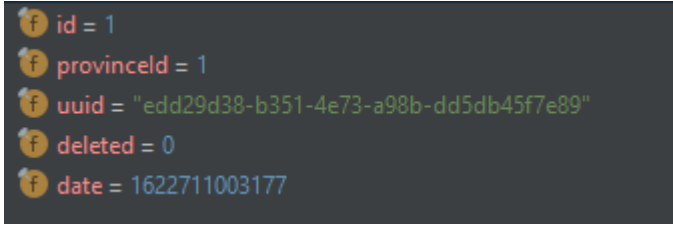
Resultado obtenido	 <p><i>Ilustración 84: Comprobación y activación del Bluetooth.</i></p>  <p><i>Ilustración 85: Registro de usuario en la BD local.</i></p>
--------------------	--

Tabla 17: CP-002. Registro válido.

12.3. CP-003. Estado del sistema.

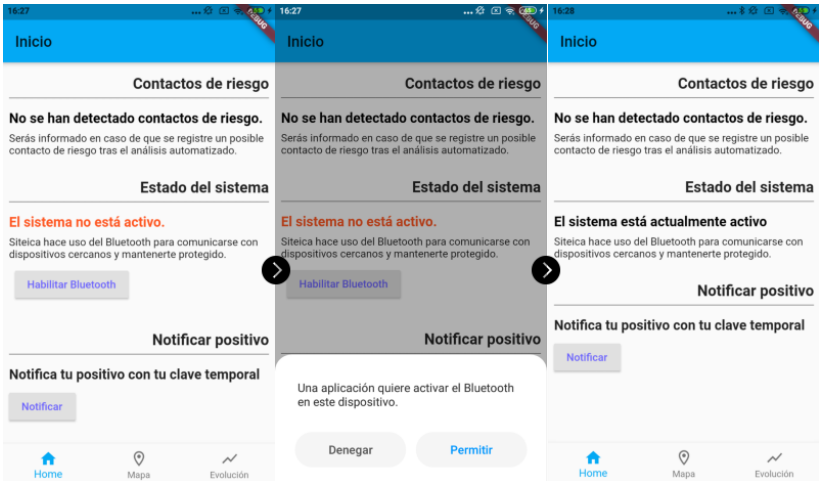
CP-003	Comprobación del estado del sistema válido
Caso de uso	RNF-004
Objetivo	Validar el correcto funcionamiento del estado del sistema verificando el Bluetooth.
Precondición	El usuario entra en la vista de “Inicio”.
Pasos	1. Navegar a la opción “Inicio” sin el Bluetooth activado.
Resultado esperado	Se notifica que el sistema no está operativo y se ofrece la posibilidad de realizar una activación manual.
Resultado obtenido	 <p><i>Ilustración 86: Estado del sistema y activación del Bluetooth.</i></p>

Tabla 18: CP-003. Comprobación del estado del sistema válido.

12.4. CP-004. Generación de identificadores de encuentro.

CP-004	Generación de identificadores de encuentro válido
Caso de uso	CU-002
Objetivo	Validar la creación de identificadores de encuentro cada 15 minutos.
Precondición	El usuario ha finalizado el registro.
Pasos	1. Abrir la aplicación.
Resultado esperado	Se crean identificadores de encuentro en la base de datos local.
Resultado obtenido	 <p><i>Ilustración 87: Identificadores de encuentro en la BD local cada 15 minutos.</i></p>

Tabla 19: CP-004. Generación de identificadores de encuentro válido.

12.5. CP-005. Intercambio de identificadores de encuentro.

CP-005	Intercambio de identificadores de encuentro válido
Caso de uso	CU-003
Objetivo	Validar el correcto intercambio de identificadores de encuentro entre dos dispositivos.
Precondición	El usuario ha finalizado el registro y tiene el Bluetooth activado.
Pasos	<ol style="list-style-type: none"> 1. Abrir la aplicación. 2. El terminal se encuentra con otro dispositivo con la aplicación de SITEICA.

Resultado esperado	El dispositivo recibe el UUID de encuentro del dispositivo cercano y se almacenan los datos del encuentro en la BD local.
Resultado obtenido	 <p><i>Ilustración 88: Registros de encuentros en la base de datos local.</i></p>

Tabla 20: CP-005. Intercambio de identificadores de encuentro válido.

12.6. CP-006. Notificación de usuario positivo.

CP-006	Notificación de usuario positivo válido
Caso de uso	CU-004
Objetivo	Validar el proceso de notificación de un usuario positivo.
Precondición	El usuario tiene un OTP válido que ha recibido de la autoridad de salud.
Pasos	<ol style="list-style-type: none"> 1. Pulsar el botón “Notificar” en la vista de “Inicio”. 2. Introducir un OTP válido. 3. Introducir la fecha del diagnóstico. 4. Pulsar el botón “Notificar”. 5. Revisar la información que va a enviarse. 6. Pulsar el botón “Confirmar”.
Resultado esperado	El OTP queda inutilizado y se notifica al usuario que sus datos han sido enviados. Los registros de encuentro de la base de datos quedan marcados como transmitidos.

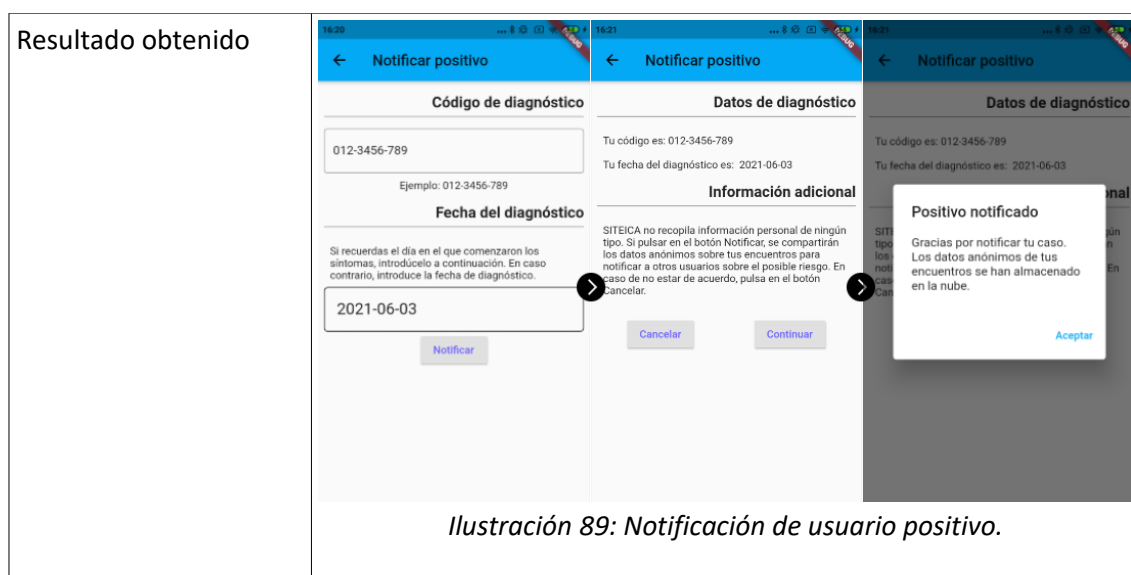


Tabla 21: CP-006. Notificación de usuario positivo válido.

12.7. CP-007. Código de notificación ya utilizado.

CP-007	Código de notificación ya utilizado
Caso de uso	CU-004
Objetivo	Validar un código de notificación ya utilizado.
Precondición	El OTP ya ha sido utilizado con anterioridad.
Pasos	<ol style="list-style-type: none"> 1. Pulsar el botón “Notificar” en la vista de “Inicio”. 2. Introducir un OTP ya utilizado. 3. Pulsar el botón “Notificar”.
Resultado esperado	Se muestra un mensaje de que el OTP ya ha sido utilizado y no se permite continuar con el proceso.

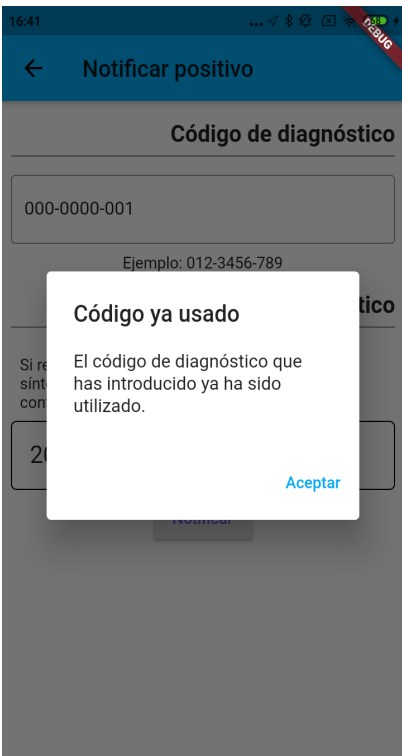
Resultado obtenido	 <p><i>Ilustración 90: Código ya utilizado en notificación de positivo.</i></p>
--------------------	---

Tabla 22: CP-007. Código de notificación ya utilizado.

12.8. CP-008. Carga de identificadores de encuentros al servidor.

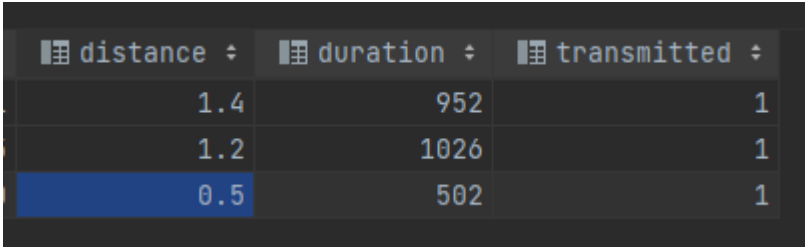
CP-008	Carga de identificadores de encuentros al servidor
Caso de uso	CU-005
Objetivo	Validar la carga de identificadores de encuentro.
Precondición	Introducir un código de diagnóstico válido y confirmar el envío.
Pasos	1. Pulsar el botón “Confirmar” en el último paso de la notificación.
Resultado esperado	Los registros quedan marcados en la base de datos como transmitidos.
Resultado obtenido	 <p><i>Ilustración 91: Registros de encuentros transmitidos.</i></p>

Tabla 23: CP-008. Carga de identificadores de encuentros al servidor.

12.9. CP-009. Análisis de riesgo.

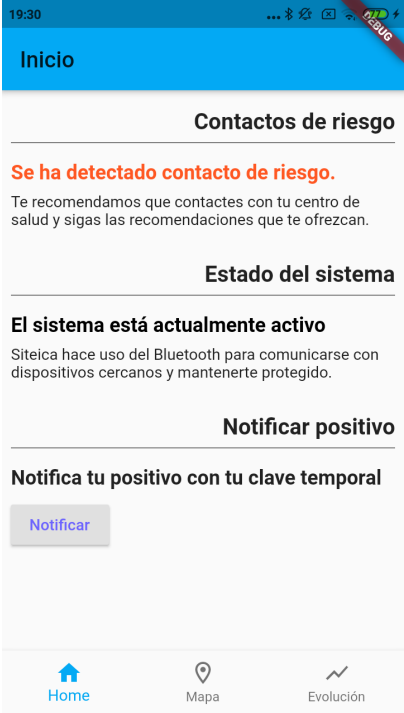
CP-009	Análisis de riesgo válido
Caso de uso	CU-006
Objetivo	Validar el análisis de riesgo cuando se ha descargado un encuentro de riesgo que coincide con una de los identificadores del usuario.
Precondición	Se ha descargado un encuentro de riesgo que coincide con una de las semillas del usuario local.
Pasos	1. Abrir la aplicación.
Resultado esperado	El sistema realiza una comprobación contra sus semillas de encuentro y notifica al usuario que se ha encontrado una coincidencia.
Resultado obtenido	 <p><i>Ilustración 92: Notificación de contacto de riesgo.</i></p>

Tabla 24: CP-009. Análisis de riesgo válido.

12.10. CP-010. Mapa de infección.

CP-010	Mapa de infección válido
Caso de uso	CU-007
Objetivo	Validar que se ubica al usuario sobre el mapa y se visualizan los contactos de riesgo.
Precondición	Se han descargado encuentros de riesgo cercanos a la posición del terminal.
Pasos	1. Acceder a la opción “Mapa” del menú inferior.

Resultado esperado	Se visualizan los contactos de riesgo sobre el mapa con un icono de precaución en color rojo y la ubicación actual del usuario con un icono de localización en negro.
Resultado obtenido	 <p><i>Ilustración 93: Mapa de infección.</i></p>

Tabla 25: CP-010. Mapa de infección válido.

12.11. CP-011. Datos de evolución.

CP-011	Datos de evolución válidos
Caso de uso	CU-008
Objetivo	Validar que se se muestran la información completa sobre la evolución de la enfermedad infecciosa.
Precondición	-
Pasos	1. Acceder a la opción “Evolución” del menú inferior.
Resultado esperado	Se visualiza el resumen de los casos positivos recogidos por SITEICA, se muestra una gráfica de tiempo con los casos de los últimos 14 días y se informa sobre los totales por provincias.



Tabla 26: CP-011. Datos de evolución válidos.

13. Conclusiones

13.1. Conclusiones

El principal objetivo del proyecto consistía en la creación de un sistema de trazabilidad de enfermedades infecciosas donde recoger información sobre contactos cercanos de forma anónima. Este objetivo se ha cumplido a través de la adopción de una solución descentralizada así como la ausencia de recopilación y transmisión de datos personales del usuario. El sistema genera UUIDs tanto para identificar al usuario como para los encuentros con lo que, incluso en el intercambio entre terminales, un ataque de intermediario solo sería capaz de obtener identificadores únicos y aleatorios que no presentan vinculación con datos del usuario.

En referencia a los encuentros entre usuarios, se había fijado el claro objetivo de lograr una comunicación M2M con otras aplicaciones. Gracias a los análisis previos realizados para la búsqueda de la tecnología adecuada, se ha logrado desarrollar un sistema capaz de actuar como emisor y receptor BLE y lograr así dicho objetivo con éxito. El BLE además proporciona un mecanismo sencillo de transmisión de datos entre terminales con un consumo energético realmente bajo.

Otro de los objetivos perseguidos era la creación de una aplicación multiplataforma móvil. Con la adopción de Flutter como SDK de desarrollo se ha logrado codificar una aplicación que puede ejecutarse tanto en iOS como en Android de forma nativa. No obstante, se han tenido que realizar algunas configuraciones específicas para cada entorno que entran dentro de lo esperado en este tipo de aplicaciones.

Finalmente, se han utilizado los datos recopilados por las aplicaciones para mostrar información relevante sobre la enfermedad a los usuarios de las mismas. De esta manera el sistema es capaz de ofrecer una idea general sobre el estado de propagación de la enfermedad.

13.2. Líneas futuras

Como parte de la naturaleza implícita del proyecto existen distintos evolutivos que deberían llevarse a cabo para un funcionamiento completo del sistema: una aplicación de servidor que permita la gestión y explotación de los datos y una aplicación móvil multiplataforma para el personal sanitario.

La aplicación de servidor que permita la gestión y explotación de los datos, perteneciente a la autoridad de salud, está dentro de lo que hemos catalogado en varias ocasiones como “servidor central”. Es el punto de origen y destino de los datos recopilados por las aplicaciones móviles que, de cara a poder realizar una comunicación con estas, debería exponer una API Rest para poder realizar operaciones que en el presente proyecto se han hecho de manera ficticia.

Por otra parte, existe una limitación en la emisión BLE recogida en la documentación oficial que informa de que, una vez la aplicación actúa como baliza, esta debe continuar ejecutándose en primer plano para transmitir las señales BLE. Esto conlleva que el sistema

pierda eficacia en cuanto el usuario cierra la aplicación, por lo que podría ser interesante buscar alternativas para los dispositivos de Apple que permitan continuar con su ejecución como una tarea en segundo plano.

Finalmente, la introducción de técnicas de Inteligencia Artificial como Machine Learning podría ayudar a aumentar la precisión de la predicción de este tipo de sistemas. Además de esto, la base de conocimiento que se puede crear a partir de los datos recogidos, en conjunto con técnicas predictivas, podría ser eficaz a la hora de detectar próximos brotes de la enfermedad.

14. Bibliografía

- [1] S. Zhao *et al.*, “Preliminary estimation of the basic reproduction number of novel coronavirus (2019-nCoV) in China, from 2019 to 2020: A data-driven analysis in the early phase of the outbreak,” vol. 92, pp. 214–217, 2020, doi: <https://doi.org/10.1016/j.ijid.2020.01.050>.
- [2] S. Wang, S. Ding, and L. Xiong, “A New System for Surveillance and Digital Contact Tracing for COVID-19: Spatiotemporal Reporting Over Network and GPS,” vol. 8, no. 6, p. e19457, Jun. 2020, doi: 10.2196/19457.
- [3] J. Berglund, “Tracking COVID-19: There’s an App for That,” vol. 11, no. 4, pp. 14–17, 2020, doi: 10.1109/MPULS.2020.3008356.
- [4] E. Hernandez-Orallo, P. Manzoni, C. Tavares Calafate, and J.-C. Cano, “Evaluating How Smartphone Contact Tracing Technology Can Reduce the Spread of Infectious Diseases: The Case of COVID-19,” vol. 8, pp. 99083–99097, 2020, doi: 10.1109/ACCESS.2020.2998042.
- [5] N. Ahmed *et al.*, “A Survey of COVID-19 Contact Tracing Apps,” vol. 8, pp. 134577–134601, 2020, doi: 10.1109/ACCESS.2020.3010226.
- [6] A. Urbaczewski and Y. J. Lee, “Information Technology and the pandemic: a preliminary multinational analysis of the impact of mobile tracking technology on the COVID-19 contagion control,” vol. 29, no. 4, pp. 405–414, Jul. 2020, doi: 10.1080/0960085X.2020.1802358.
- [7] M. Nanni *et al.*, “Give more data, awareness and control to individual citizens, and they will help COVID-19 containment,” vol. 13, no. 1, pp. 61–66, Apr. 2020.
- [8] J. Bay *et al.*, “BlueTrace: A privacy-preserving protocol for community-driven contact tracing across borders,” 2020, [Online]. Available: <https://bit.ly/2NiOJ0p> . [Accesed: Feb 12, 2021]
- [9] R. L. Rivest, J. Callas, and R. Canetti, “The PACT protocol specification,” Apr. 08, 2020, [Online]. Available: <https://bit.ly/30IJPNh>. [Accesed: Feb 18, 2021]
- [10] C. Castelluccia *et al.*, “DESIRE: A Third Way for a European Exposure Notification System Leveraging the best of centralized and decentralized systems,” Aug. 2020, [Online]. Available: <https://arxiv.org/abs/2008.01621>. [Accesed: Feb 25, 2021]
- [11] D. Antonioli, N. O. Tippenhauer, and K. Rasmussen, “Nearby Threats: Reversing, Analyzing, and Attacking Google’s Nearby Connections’ on Android,” 2019.
- [12] “What is Bluetooth Low Energy?,” May 23, 2019, [Online]. Available: <https://elainnovation.com/what-is-ble.html>. [Accesed: Mar 2, 2021]

- [13] P. Christensson, "BLE Definition," Mar. 08, 2019, [Online]. Available: <https://techterms.com/definition/ble>. [Accessed: Mar 2, 2021]
- [14] K. Townsend, C. Cufí, Akiba, and R. Davidson, *Getting Started with Bluetooth Low Energy*. Sebastopol: O'Reilly Media, Incorporated, 2014.
- [15] F. Sattler et al., "Risk estimation of SARS-CoV-2 transmission from bluetooth low energy measurements," vol. 3, no. 1, p. 129, Jan. 2020, doi: 10.1038/s41746-020-00340-0.
- [16] T. S. Rappaport, *Wireless communications: principles and practice*. Prentice Hall PTR, 2002.
- [17] P. Lapolla and R. Lee, "Privacy versus safety in contact-tracing apps for coronavirus disease 2019," vol. 6, pp. 205520762094167–2055207620941673, Jul. 2020, doi: 10.1177/2055207620941673
- [18] "Privacy guardians issue joint statement on COVID-19 contact tracing applications," PR Newswire Association LLC, New York, May 07, 2020.
- [19] "What is CI/CD? ," Jan. 31, 2018, [Online]. Available: <https://www.redhat.com/en/topics/devops/what-is-ci-cd>.
- [20] "Flutter." [Online]. Available: <https://flutter.dev/>. [Accessed: Mar 24, 2021]
- [21] "Git," [Online]. Available: <https://git-scm.com/>. Accessed: Apr 17, 2021]
- [22] "Beacon Broadcast." https://pub.dev/packages/beacon_broadcast. [Accessed: Apr 2, 2021]
- [23] "Sqlite." <https://pub.dev/packages/sqlite>. [Accessed: Mar 25, 2021]
- [24] "Geolocator." <https://pub.dev/packages/geolocator>. [Accessed: Apr 5, 2021]
- [25] "Injector." <https://pub.dev/packages/injector>. [Accessed: Apr 15, 2021]
- [26] "Flutter Map." https://pub.dev/packages/flutter_map. [Accessed: Apr 6, 2021]
- [27] "Beacons Plugin." https://pub.dev/packages/beacons_plugin. [Accessed: Apr 2, 2021]
- [28] "Sqlite Migration Service." https://pub.dev/packages/sqlite_migration_service. [Accessed: Apr 15, 2021]
- [29] "json_serializable | Dart Package" https://pub.dev/packages/json_serializable. [Accessed: Apr 17, 2021]
- [30] "uuid_enhanced | Dart Package" https://pub.dev/packages/uuid_enhanced. [Accessed: Apr 17, 2021]
- [31] "Windows install – Flutter" <https://esflutter.dev/docs/get-started/install/windows>. [Accessed: Apr 27, 2021]

- [32] <https://developer.android.com/training/location/permissions>. [Accesed: May 14, 2021]
- [33] “intro_slider” https://pub.dev/packages/intro_slider. [Accesed: May 16, 2021]
- [34] “bluetooth_enable” https://pub.dev/packages/bluetooth_enable. [Accesed: May 17, 2021]
- [35] “charts_flutter” https://pub.dev/packages/charts_flutter. [Accesed: May 29, 2021]

15. Anexos

15.1. Anexo. Entorno tecnológico.

En el presente anexo se presentan las herramientas específicas que se han utilizado para el desarrollo de SITEICA y sus distintas configuraciones con el objetivo implantar prácticas DevOps y así obtener un entorno tecnológico estable que nos permita acortar el ciclo de vida del desarrollo de dicho software.

15.1.1. IntelliJ IDEA

IntelliJ IDEA es un entorno de desarrollo integrado (IDE) multiplataforma diseñado para lenguajes JVM, aunque sus funcionalidades pueden extenderse a través del uso de plugins para conseguir una experiencia con soporte para multitud de lenguajes de programación. Entre las versiones disponibles de IntelliJ IDEA se encuentran la Community Edition, edición gratuita orientada al desarrollo en JVM y Android, y la versión Ultimate, edición de pago que incluye algunas características exclusivas orientadas al desarrollo de aplicaciones web y empresariales.

Para el desarrollo de la aplicación móvil multiplataforma de SITEICA se ha utilizado la versión IntelliJ IDEA Community Edition en conjunto con los plugins de Dart, que ofrece soporte para el lenguaje, y Flutter, que habilita el desarrollo de aplicaciones con el SDK de Google.

15.1.2. Sistema de control de versiones

Como primer objetivo se ha marcado la necesidad de elegir un sistema de control de versiones (VCS). Los VCS son un conjunto de herramientas orientadas al registro de cambios realizados en los archivos mediante un seguimiento de las modificaciones realizadas en el código a lo largo del tiempo.

Este tipo de herramientas se han vuelto imprescindibles en el ámbito del desarrollo de aplicaciones. Como sabemos, un producto software se desarrolla de forma conjunta por un número variable de programadores que, además, pueden estar ubicados en lugares diferentes, contribuyendo cada uno de ellos en características o funcionalidades específicas. Con esta composición, es habitual que varias personas hagan cambios sobre el mismo archivo en el mismo espacio de tiempo. Los VCS nos ayudan a simplificar estos flujos de trabajo realizando un seguimiento exhaustivo de todos los cambios individuales de cada colaborador del proyecto, contribuyendo a evitar que el trabajo concurrente entre en conflicto.

Entre los beneficios que nos aportan los VCS se encuentran los siguientes:

- Brindan mecanismos de colaboración eficientes, agilizando el desarrollo del proyecto en equipo.
- Minimiza las posibilidades de errores y conflictos en el código gracias a la trazabilidad de cada cambio, por pequeño que sea.

- Las personas que pertenecen al equipo pueden aportar cambios independientemente del lugar en el que se encuentren.
- Las modificaciones de cada desarrollador se encuentran siempre en su repositorio local, lo que evita comprometer el estado del proyecto. En flujos de trabajo estándar, los cambios se incorporan una vez se han validado por el equipo.
- Gracias a que estos sistemas guardan un historial completo de cambios, nos brindan la posibilidad de recuperación en casos de errores críticos.
- Nos ofrecen información extensa sobre quién hizo un cambio, cuándo y por qué.

Estas bondades de los VCS son razón suficiente para su uso en cualquier tipo de proyecto y equipo de desarrollo. Por ello, se ha elegido git como herramienta de control de versiones y GitHub como repositorio remoto privado.

Git, tal y como lo definen sus autores [21], es un sistema de control de versiones distribuido de código abierto y gratuito diseñado para manejar todo, desde proyectos pequeños a muy grandes, con velocidad y eficiencia. GitHub, por su parte, es una plataforma de alojamiento de proyectos que utiliza el sistema de control de versiones Git.

15.2. Integración y entrega continua

La integración continua (CI) y la entrega continua (CD), conocida en conjunto como CI/CD, incorporan una cultura, un conjunto de principios operativos y una colección de prácticas que permiten a los equipos de desarrollo de aplicaciones entregar cambios de código con mayor frecuencia y confiabilidad.

La CI/CD es un conjunto de prácticas que pueden implementar los equipos de devops y que dan solución a los problemas asociados a la integración de código nuevo a los equipos de desarrollo y operaciones. Estas técnicas agregan la automatización continua y el control permanente en todo el ciclo de vida de las aplicaciones, desde las fases de integración y prueba hasta las de distribución e implementación.

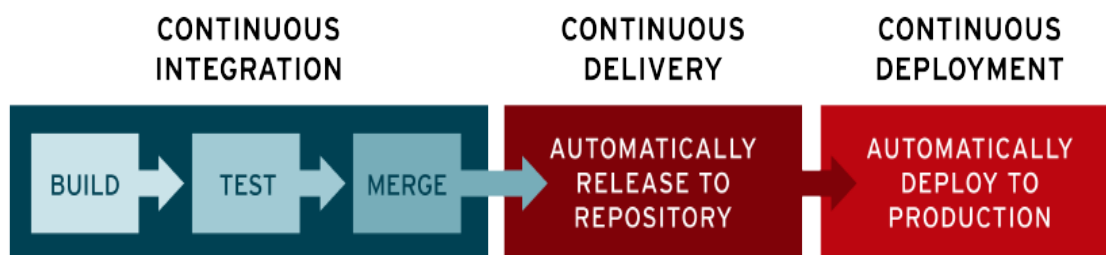


Ilustración 95: CI / CD. Fuente: <https://www.redhat.com/en/topics/devops/what-is-ci-cd>

También es una práctica altamente recomendada de metodología ágil, ya que permite a los equipos de desarrollo de software concentrarse en cumplir con los requisitos comerciales, la calidad del código y la seguridad porque los pasos de implementación están automatizados.

Como parte de esta estrategia, y dado que se ha optado por el uso de GitHub como alojamiento remoto para el control de versiones, se ha optado por el uso de GitHub Actions.

15.3. Integración y entrega continua con GitHub Actions

Se ha mencionado en la sección 4.1. la elección de GitHub como alojamiento remoto para el presente proyecto, bajo el sistema de control de versiones conocido como git. Como parte de las utilidades que ofrece GitHub se encuentra GitHub Actions, un conjunto de herramientas de automatización que nos permiten implementar el flujo de trabajo de CI/CD.

A continuación se describen los pasos necesarios para configurar dicho flujo de trabajo en GitHub.

15.3.1. Crear el proyecto en GitHub

Se ha creado un nuevo repositorio en GitHub con el nombre de siteica-user, haciendo referencia a la aplicación móvil específica del usuario general. Para la creación de este nuevo proyecto se ha utilizado la interfaz Web de la plataforma, asignando un nombre, descripción y configurándose como privado.

Tras este primer paso, es hora de subir el proyecto de Flutter desde la terminal, haciendo uso de los siguientes comandos de git en la carpeta raíz del proyecto:

```
git init
git add .
git commit -m "first commit"
git remote add origin https://github.com/Mikaulin/siteica-user.git
git push -u origin master
```

15.3.2. Añadir el archivo de definición del flujo de trabajo

En la carpeta raíz de nuestro proyecto Flutter se ha creado una carpeta con el nombre `.github` y, dentro de esta, una carpeta llamada `workflows`. Aquí se situarán todos los archivos YAML referentes a flujos de trabajo, donde podemos tener flujos diferenciados para realizar una build, release, etc. De esta forma, se ha creado el archivo `main.yml` en el directorio `workflows` recién creado.

15.3.3. Comandos del flujo de trabajo

El siguiente paso es definir el evento de GitHub en el cual queremos que se ejecute el flujo de trabajo. En el presente proyecto se ha optado por el evento push.

```

1 on:
2   push:
3     branches:
4       - master
5 name: Test, Build and Release apk
6 jobs:
7   build:
8     name: Building APK
9     runs-on: ubuntu-latest
10    steps:
11      - uses: actions/checkout@v1
12      - uses: actions/setup-java@v1
13        with:
14          java-version: '12.x'
15      - uses: subosito/flutter-action@v1
16        with:
17          flutter-version: '2.0.3'
18      - run: flutter pub get
19      - run: flutter test
20      - run: flutter build apk --debug --split-per-abi
21      - name: Create a Release APK
22        uses: ncipollo/release-action@v1
23        with:
24          artifacts: "build/app/outputs/apk/debug/*.apk"
25          token: ${ secrets.TOKEN }

```

Ilustración 96: Archivo YAML con la definición del flujo de trabajo.

A continuación se describen las líneas más importantes del flujo:

- **Líneas 1-4:** nos permite que el flujo de trabajo se ejecute cuando se haga un push sobre la rama `master` de nuestro repositorio, evitando así que las subidas de trabajo en desarrollo desencadenen la ejecución de las acciones.
- **Líneas 6-9:** se define una tarea (job) que se ejecuta bajo `ubuntu-latest` a la que hemos llamado `Building APK`. Cada tarea se ejecuta en una instancia de una máquina virtual, en nuestro caso la última versión de Ubuntu, y puede contener uno o más pasos.
- **Línea 11:** el primer paso de la tarea desde el que ejecutamos la acción de GitHub conocida como `checkout`, con la que estaremos descargando el contenido actualizado del repositorio.
- **Líneas 12-14:** en el segundo paso de la tarea se establece el entorno de Java necesario para compilar una aplicación de Flutter.
- **Líneas 15-17:** se hace uso de la acción `flutter-action` que permite establecer el entorno de Flutter para usarlo dentro de las acciones de GitHub.
- **Línea 18:** mediante este comando de Flutter se obtienen las dependencias del proyecto.

- **Línea 19:** ejecutamos los tests dentro de la automatización, asegurando el funcionamiento de la app.
- **Línea 20:** generamos la APK en modo debug. El parámetro `--split-per-abi` permite que se generen diferentes APK según arquitecturas, a fin de evitar un solo APK de gran tamaño.
- **Línea 21:** esta acción crea un lanzamiento en GitHub de la APK recién compilada. Para realizar la publicación con éxito debemos crear un token de acceso personal que de acceso al repositorio.

15.3.4. Crear un token de acceso personal en GitHub

Para crear el token de acceso personal debemos se han realizado los siguientes pasos:

- En nuestra cuenta de GitHub vamos a la opción “Settings”, “Developer settings” y “Personal access token”.
- Pulsamos sobre el botón “Generate new token”, establecemos los permisos para el control de los repositorios privados y pulsamos sobre el botón “Generate token”. Entonces vemos la clave recién creada que copiamos al portapapeles.
- Vamos al repositorio del proyecto y en la pestaña de “Settings” accedemos a la opción “Secrets”. Se ha establecido como valor la clave que se ha creado en el paso anterior.

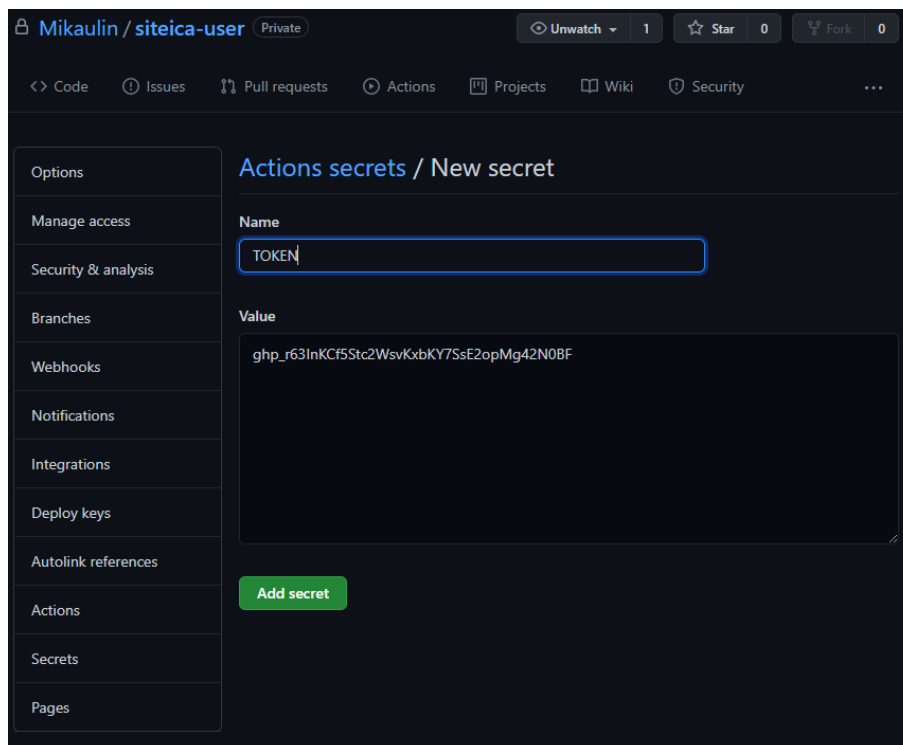


Ilustración 97: Configurar el repositorio de GitHub.

15.4. Inyección de dependencias

La inyección de dependencia (DI) es un patrón de diseño de software que se utiliza para implementar las características de la inversión de control (IoC). Permite la creación de objetos

dependientes fuera de una clase y proporciona esos objetos a una clase de diferentes formas. Usando DI, la creación de objetos se delega a un punto central de la arquitectura para luego ser suministrados en forma de objetos a la clase que los necesite. Esto aporta un mayor nivel de flexibilidad, desacoplamiento y pruebas más sencillas.

El paquete Injector de Flutter nos aporta una librería de clases para el manejo de la Inyección de Dependencias.

15.5. Herramientas de documentación

Además de todas las herramientas y tecnologías ya mencionadas, se han utilizado diversas herramientas como apoyo para documentar el presente proyecto:

- OpenOffice Writer: para la creación de la memoria del proyecto se ha optado por el uso de Apache OpenOffice Writer como procesador de texto. Este editor forma parte de la familia Apache OpenOffice, suite ofimática libre y de código abierto.
- Moqups: esta herramienta online para la realización de prototipados nos permite crear y colaborar en tiempo real. Se ha utilizado una cuenta gratuita que nos da soporte para la realización de un proyecto sin ningún tipo de coste.
- Gliffy Diagrams: es un software basado en la nube orientado a la creación de diagramas de distintos tipos. Se ha optado por instalar Gliffy como aplicación de Chrome, lo que simplifica su uso, y se han desarrollado los distintos diagramas de flujo del proyecto, así como el diagrama UML o la estructura de tablas de SQLite.
- BPMN.io: herramienta Web para el modelado de procesos de negocio con notación BPMN que nos permite guardar y cargar un diagrama desde el equipo local, así como exportar a imagen.
- Freelcons.io: base de datos de iconos gratuitos que se ha utilizado para algunas secciones de la aplicación móvil con el objetivo de crear una interfaz de usuario más atractiva y sencilla de interpretar a través de dichos iconos.