Proyecto Final de Carrera

Sistema de Información para el Seguimiento de la Actividad y Evolución de Personas Residentes en Centros de Atención Especializada

Autor: Joaquín Ocón Ojeda
Tutor: Agustín Salgado de la Nuez

Julio de 2016

A mi hermano Dani, mi fuerza

Agradecimientos

En primer lugar quiero dar las gracias a mi madre, por su amor y su entrega. Mamá, eres mi mayor orgullo.

Quiero dedicar un agradecimiento especial a Agustín Salgado. Te conocí como profesor, te elegí como tutor y ahora te nombro como amigo.

A mis padres Rita, Joaco, Tato, Sari por vuestro apoyo y cariño incondicional, por enseñarme y guiarme, mil gracias.

También quiero dar las gracias a mi amigo Javier, un ejemplo para mi como profesional, pero mucho más como persona.

A mi tío Antonio, por tratar siempre de inculcarme todo el conocimiento que soportan esos hombros, sin ti nada de esto sería posible.

Por último y de forma muy especial, quiero darte las gracias a ti, hermano, por ser mi fuerza cada vez que flaqueo. Te agradezco cada sonrisa, cada gruñido, cada momento que disfruté a tu lado.

Índice general

Página

1. Introducción **1**
1.1. Motivación Personal 2

2. Objetivos **3**
2.1. Objetivos Generales 3
2.2. Objetivos Personales 4

3. Estado del Arte **5**
3.1. Análisis de la Competencia 6

4. Metodología de Trabajo **9**
4.1. Modelo de Ciclo de Vida 9

5. Recursos Utilizados **11**
5.1. Recursos Software 11
5.2. Recursos Hardware 16
5.3. Recursos 'Cloud' 17

6. Planificación Temporal **19**

7. Definición del Proyecto **21**
7.1. Descripción y Alcance del Sistema 21
7.2. Requisitos Funcionales 24
7.3. Requisitos No Funcionales 24

8. Estudio Previo **26**
8.1. Android 26
8.2. XML y Android 31
8.3. Web-Services 31

9. Análisis **33**
9.1. Actores del Sistema 33
9.2. Diagramas de Casos de Uso 33

10. Diseño **38**
10.1. Arquitectura del Sistema 38
10.2. Diseño del Servidor 40

10.3. Diseño de CronCare . . . 41
10.4. Diseño de CronCare Centros . . . 45

11.Implementación **49**
11.1. Librerías Externas . . . 49
11.2. Implementación de la API . . . 50
11.3. Implementación de la App CronCare Centros . . . 60
11.4. Implementación de la App CronCare . . . 67

12.Plan de Negocio **73**
12.1. Resumen Ejecutivo . . . 73
12.2. Mercado Potencial . . . 73
12.3. Modelo de Negocio . . . 74
12.4. Estrategia Comercial . . . 75

13.Proyección Económica-Financiera **76**
13.1. Personal y Gastos . . . 76
13.2. Inversiones Fijas . . . 77
13.3. Créditos y Pagos . . . 77
13.4. Ventas Estimadas y Margen . . . 77
13.5. Proyección de Tesorería . . . 78

14.Pruebas **79**
14.1. Pruebas de Rendimiento . . . 79
14.2. Pruebas de Validación . . . 79
14.3. Pruebas de Usabilidad . . . 80

15.Conclusiones **81**
15.1. Conclusiones Académicas . . . 81
15.2. Conclusiones Personales . . . 82

16.Trabajo Futuro **83**

Apéndices **85**

A. Especificaciones del Análisis **86**
A.1. Modelo de Casos de Uso . . . 86

B. Clases Java-Android **106**
B.1. ActivityHTTP.java . . . 106
B.2. ThreadGET.java . . . 109
B.3. ThreadPOST.java . . . 112

C. Ficheros PHP **116**

D. Traza de Diferentes Llamadas a la API **122**

Bibliografía **126**

Índice de figuras

Página

1.1. Icono de CronCare 1

3.1. Uso de Internet Por Dispositivos. Fuente: OFCOM[20] 5
3.2. Icono de la App GerApp 6
3.3. Vista Previa de la App Gerapp 6
3.4. Icono de la App Wappa Senior 7
3.5. Vista Previa de la App Wappa Senior 7
3.6. Icono de la App Sanyres 8
3.7. Vista Previa de la App Sanyres 8

4.1. Modelo por Prototipado 9

5.1. Android Studio 12
5.2. Entorno de desarrollo Cloud9 12
5.3. Workflow de GIT 13
5.4. Bitbucket 13
5.5. Vista Previa del Software de Edición Sketch 14
5.6. Vista Previa del Software VioletUML 15
5.7. Vista Previa del Software LyX 15
5.8. Pantalla de Descarga del Software ARC 16
5.9. Listado de APIs Ofrecidas por Google Para Desarrolladores 17
5.10. Google Cloud Messaging 18

6.1. Gráfica de la Planificación Temporal 19

7.1. Esquema de la Comunicación del Sistema 21
7.2. Descripción de la API de CronCare 23

8.1. Diagrama de la Arquitectura de Android 27
8.2. Ciclo de Vida de una Actividad en Android 28
8.3. Estructura de un Proyecto en Android Studio 30

9.1. Diagrama de Casos de Uso nº1 - CronCare Para Familiares de Residentes . . . 34
9.2. Diagrama de Casos de Uso nº2 - CronCare Centros 36

10.1. Esquema del Diseño Arquitectónico del Sistema Basado en el Modelo Cliente-Servidor 39
10.2. Diagrama que Muestra el Procedimiento de una Llamada a la API 40

10.3. Diagrama Entidad-Relación de la Base de Datos 41
10.4. Vistas para Usuarios sin Sesión 42
10.5. Vistas para Usuarios sin Sesión 42
10.6. Vistas de la app para un usuario que ha accedido al sistema 43
10.7. Vista Principal de la App CronCare Centros 45
10.8. Ejemplo de Registrar Entrada de Residentes 46
10.9. Diagrama de Actividad para Seleccionar Residentes 47

11.1. Ejemplo de Gráfica Realizada con la Librería Highcharts 50
11.2. Fichero .htaccess Utilizado 51
11.3. Extracto del Fichero index.php 51
11.4. Final del Fichero index.php 52
11.5. Vista de los Ficheros de la API 53
11.6. Registro de un Familiar de Residente Mediante la API 53
11.7. Acceso de un Familiar de Residente mediante la API 54
11.8. Función validateVariable() del Fichero api_functions.php 55
11.9. Funciones send_response() y send_error() 56
11.10. Fichero config.php 57
11.11. Función db_select() dentro del Fichero db_functions.php 58
11.12. Extracto de la Función deliver_push_notifications() 59
11.13. Declaración del Fichero activity_main.xml 60
11.14. Diagrama de las clases ActivityLogin y ActivityMain 61
11.15. Función LoadPacients() dentro de ActivityMain 62
11.16. Diagrama de la Clase ActivityStatus 62
11.17. Interfaz de la Clase ActivityStatus y Sección de código 63
11.18. Declaración de la Clase ActivityLogin 64
11.19. Continuación de la Clase ActivityLogin 65
11.20. Diagrama de las Clases ThreadPOST y ThreadGET 65
11.21. Métodos onPreExecute() y onPostExecute() de ThreadPOST 66
11.22. Diferencias en los Métodos doInBackground() de ThreadPOST y ThreadGET . 67
11.23. Interfaz Gráfica de ActivityMain 68
11.24. Diagrama de la Clasa ActivityStats 69
11.25. Fragmento de la Clase ActivityStats 69
11.26. Diagrama de la Clase ActivityMain 70
11.27. Código de los Métodos para la Navegación Cronológica 71
11.28. Diagrama de la Clase ActivityHomeNote 72
11.29. Recepción de Respuesta de la API en ActivityHomeNote 72

14.1. Excepción en el Intento de Inicio de Sesión Múltiple 80

16.1. Millones de Usuarios por Smartphone en EE.UU. Fuente: eMarketer[8] 84

D.1. Intento de Acceso Fallido de un Familiar de Residente mediante la API 123
D.2. Llamada a la API para Acceder al Sistema 124
D.3. Llamada para Listar Residentes Vinculados a un Familiar mediante la API . . 125

Índice de tablas

Página

10.1. Llamadas a la API desde CronCare 44
10.2. Llamadas a la API desde CronCare Centros 48

13.1. Tabla de Personal y Gastos 76
13.2. Tabla de Inversiones Fijas 77
13.3. Tabla de Créditos y Pagos 77
13.4. Tabla de Ventas Estimadas y Margen 78
13.5. Proyección de Tesorería 78

A.1. Administrador - Tabla de Caso de Uso #01: Iniciar Sesión 88
A.2. Administrador - Tabla de Caso de Uso #02: Cerrar Sesión 89
A.3. Administrador - Tabla de Caso de Uso #03: Crear Perfil de Cuidador 89
A.4. Administrador - Tabla de Caso de Uso #04: Eliminar Perfil de Cuidador . . . 90
A.5. Administrador - Tabla de Caso de Uso #05: Crear Grupo de Residentes 91
A.6. Administrador - Tabla de Caso de Uso #06: Crear Perfil de Residente 92
A.7. Administrador - Tabla de Caso de Uso #07: Eliminar Perfil de Residente . . . 93
A.8. Administrador - Tabla de Caso de Uso #08: Crear Relación Familiar-Residente 94
A.9. Administrador - Tabla de Caso de Uso #09: Enviar Mensaje a Familias 95
A.10. Familiar- Tabla de Caso de Uso #10: Registrar Nuevo Usuario 96
A.11. Familiar- Tabla de Caso de Uso #11: Iniciar Sesión 97
A.12. Familiar - Tabla de Caso de Uso #12: Cerrar Sesión 97
A.13. Familiar- Tabla de Caso de Uso #13: Elegir Residente 98
A.14. Familiar- Tabla de Caso de Uso #14: Listar Mensajes 98
A.15. Familiar- Tabla de Caso de Uso #15: Listar Menú Semanal 99
A.16. Familiar- Tabla de Caso de Uso #16: Ver Datos del Centro 99
A.17. Familiar- Tabla de Caso de Uso #17: Cambiar Contraseña 100
A.18. Familiar- Tabla de Caso de Uso #18: Enviar Nota Corta 101
A.19. Familiar- Tabla de Caso de Uso #19: Ver Estadísticas de Evolución 101
A.20. Cuidador - Tabla de Caso de Uso #20: Iniciar Sesión 102
A.21. Cuidador - Tabla de Caso de Uso #21: Cerrar Sesión 103
A.22. Cuidador - Tabla de Caso de Uso #22: Listar Residentes 103
A.23. Cuidador -Tabla de Caso de Uso #23: Enviar Toma de Datos Sin Detalle . . . 104
A.24. Cuidador - Tabla de Caso de Uso #23: Enviar Toma de Datos Detallada . . . 105

Capítulo 1

Introducción

CronCare es una herramienta para la gestión de la comunicación entre Centros de Atención Especializada y los núcleos familiares de las personas residentes en estos. El sistema está compuesto de dos apps android: una para las familias y otra para los trabajadores del centro. Ambas apps se comunican mediante una API[1] con un servidor, el cual aloja también la base de datos del sistema.

Para conceptualizar este desarrollo se ha colaborado con algunas entidades privadas y con distintos del sector socio-sanitario. El feedback que este tipo de colaboradores aportaba en cada iteración del producto ha sido esencial a la hora de crear una herramienta preparada para su uso en el «mundo real», fuera del entorno estrictamente académico.

La decisión de implementar un sistema basado en apps tiene como motivación principal el ser partícipe del efecto que han causado las apps en nuestras vidas cotidianas, creando un producto tangible que pueda llegar al bolsillo de cualquier persona.

También se ha construido una API RESTFUL (ver página 31) que funciona como interfaz entre el servidor y las dos aplicaciones Android. Se sigue así el ejemplo de las principales empresas del sector, que utilizan APIs de manera interna o abriéndolas a su comunidad de desarrolladores para crecer.

Figura 1.1: Icono de CronCare

Aunque nace como un proyecto universitario, la facilidad de distribución que ofrecen las nuevas plataformas de apps permiten soñar con un futuro prometedor. La facilidad de traducir

[1] API es un acrónimo en inglés para «Application Programming Interface» (Interfaz de Programación de Aplicaciones).

todo el sistema mediante plantillas de idiomas XML también permitirían llegar a público de todo el mundo en un futuro deseable.

1.1. Motivación Personal

Tras la conceptualización de el sistema CronCare existe una serie de necesidades que el autor ha vivido en primera persona durante toda su vida, pues un familiar suyo residió muchos años en Centros de Atención Especializada. Es por ello que existía una predisposición natural ante la realización del proyecto por su parte en busca de esa pequeña satisfacción de saber que se trabaja en algo que puede llegar a significar mucho en la vida de sus potenciales usuarios. Es difícil imaginar lo que es vivir alejado de un ser querido y más aún cuando dependes de terceros para recibir cualquier información que le afecte.

Aunque en la actualidad existen opciones como el teléfono para llamar al centro o la posibilidad de desplazarse para hacer una visita, estas parecen insuficientes y optimizables (por ejemplo, las llamadas pueden saturar la centralita). A pesar de que el hecho de no convivir juntos parece insalvable, sí se puede hacer mucho por mejorar la conexión con el día a día del residente; Esta idea es la semilla de la que brota este proyecto.

Capítulo 2

Objetivos

Con el desarrollo del presente proyecto se pretende 'cerrar el círculo' de todas las fases alrededor de un producto de software. Desde la necesidad detectada, pasando por las distintas etapas definidas en las bases de la ingeniería del software para la conceptualización y ejecución del desarrollo, así como plantear una estrategia comercial demostrando la viabilidad y potencial rentabilidad de un sistema como CronCare.

2.1. Objetivos Generales

- Plantear un sistema que satisfaga la toma de requisitos confeccionada con la colaboración de varios Centros de Atención Especializada y personas del entorno del autor que son usuarios potenciales del producto.
- Implementar dos clientes en forma de apps Android:

    1. App para cuidadores o empleados del centro:

       Esta app permitirá una toma de datos eficaz pero precisa. La interfaz será optimizada para su ejecución en tablets, esto permitirá utilizar elementos gráficos simples y grandes que no den pie a errores. La pantalla, de mayor tamaño que un teléfono estándar, permite mostrar listados de residentes e interactuar con estos mediante 'acciones en lote'.

    2. App para familiares de residentes o personas autorizadas:

       Mediante esta app podrán acceder a datos referentes al día a día de la persona residente, así como datos de interés general del centro. Esta app recibirá notificaciones de tipo push cuando existan nuevos datos por consultar. Por ejemplo, podremos recibir un aviso cuando nuestro familiar se tome una medicina o termine una sesión de terapia, siempre que el cuidador tome el dato con su respectiva app.

- Desarrollar una API RESTFUL de manera que el acceso a los datos sea lo más sencillo, ágil y eficaz posible. El servidor será el responsable tanto de hospedar la base de datos como de gestionar las llamadas a esta API.

- Plantear, desde el punto de vista funcional, un panel de gestión al que accederán los administradores del centro y desde el cual podrán no solo modificar los datos del sistema (altas de cuidadores/residentes, menús de comida, horarios, etc.) sino también hacer consultas de tipo histórico y obtener información analítica.
- Desarrollar una estrategia comercial para CronCare, mediante el desarrollo de un modelo de negocio que encaje en el marco actual de las apps enfocadas a servicios.

2.2. Objetivos Personales

- Desarrollar un producto que realmente pueda mejorar la vida de algunas personas.
- Utilizar librerías de terceros distribuidas bajo licencias de software libre en repositorios online.
- Afianzar mis conocimientos sobre la Ingeniería del Software y en particular sobre los patrones de diseño más comunes entres las apps Android.
- Tratar de crear una interfaz gráfica simple pero funcional, cuyo principio básico es: «ninguna pantalla necesitará más de 3 'taps' para ser visualizada».

Capítulo 3

Estado del Arte

Las apps llevan unos diez años en el mercado y, desde hace siete, su crecimiento ha sido enorme. Este cambio de paradigma a la hora de acceder a la información ha llevado al los dispositivos móviles a liderar el tráfico en Internet. En efecto, tal y como se comprueba en la Figura 3.1, hoy en día se realizan más accesos a Internet desde dispositivos móviles que desde equipos de sobremesa o incluso ordenadores portátiles.

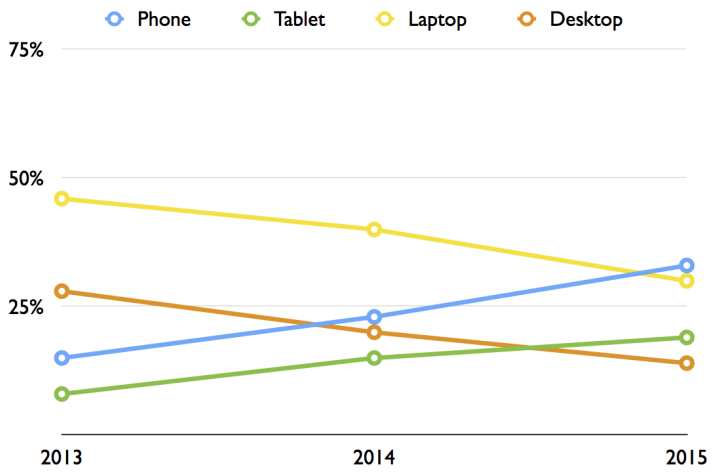


Figura 3.1: Uso de Internet Por Dispositivos. Fuente: OFCOM[20]

Gran parte del volumen de tráfico representado en este gráfico es debido al uso de apps. Las empresas de hoy en día son conocedoras de este 'boom' ocasionado tras la aparición del smartphone y cada día innovan más en dicho sector. Se puede decir que las apps son a los negocios lo mismo que fueron las webs hace 10 años: un escaparate imprescindible.

Otra funcionalidad de corta existencia, pero sin la cual no se podrían concebir las comunicaciones actuales, son las notificaciones push que nos llegan tras cada *whatsapp*[14]. Estas nos permiten ser informados sobre novedades relevantes en nuestras apps en el momento en que suceden. CronCare hace uso de este efecto «información inmediata en tu bolsillo» como un reclamo de su utilidad para las familias.

Las APIs, muchas veces en forma de servicios web, se han convertido en un elemento de comunicación básico entre los sistemas de todas las empresas del ámbito de Internet. Un sólo caso de éxito como *Twitter[13]* hace que millones de desarrolladores de todo el mundo creen sus propias apps nutridas de información de *Twitter* haciendo uso de su API abierta.

3.1. Análisis de la Competencia

Para estudiar el mercado que nos interesa en este proyecto se han analizado diferentes apps para la gestión de la comunicación entre residencias y familias. Durante el estudio de sistemas competidores se ha tratado de identificar las principales funcionalidades de cada uno de ellos, tanto las que están contempladas en CronCare como las que no.

Cabe destacar que no existen muchos negocios en torno a la digitalización de los centros de atención especializada, pero la existencia de sistemas ya en el mercado tiene una interpretación positiva al ver que otros comparten la visión de un futuro prometedor en este nicho de mercado.

3.1.1. GerApp

Figura 3.2: Icono de la App GerApp

Desarrollador: Smart Technologies Development[7]
Nº de descargas en Google Play: Entre 100 - 500
Página web oficial: www.gerapp.net

Figura 3.3: Vista Previa de la App Gerapp

Esta app desarrollada por un pequeño equipo catalán destaca por sus funcionalidades multimedia, permitiendo el envío de fotos desde el centro hacia las familias para acercar aún más la realidad del mismo. Sus vistas están bien diseñadas, dando prioridad a los datos frente al

atrezzo.

Como aspecto negativo se destaca la falta de personalización sobre los datos mostrados, pues leer "él/ella ha tenido una gran tarde de juegos de mesa" suena muy impersonal y nada cercano.

La gran diferencia entre Gerapp y el resto de sistemas evaluados es que sólo dispone de un cliente en forma de app. Es decir, han unificado en una app el uso que le puede dar un empleado de la residencia, un administrador o un familiar de un residente. Internamente el sistema permite acceder a según qué secciones en función del nivel permisos del tipo de usuario.

3.1.2. Wappa Senior

Figura 3.4: Icono de la App Wappa Senior

Desarrollador: Wappa Software Engineering, S.L[22]
Nº de descargas en Google Play: Entre 100 - 500
Página web oficial: www.wappa.net

Este producto no destaca por su relación de funcionalidades, pero su labor en posicionamiento es buena y han tenido alguna repercusión en prensa por lo que merecen una mención. Wappa incluye lo mínimo justo y necesario, su corta lista de funcionalidades hacen pensar que el producto está aún en desarrollo.

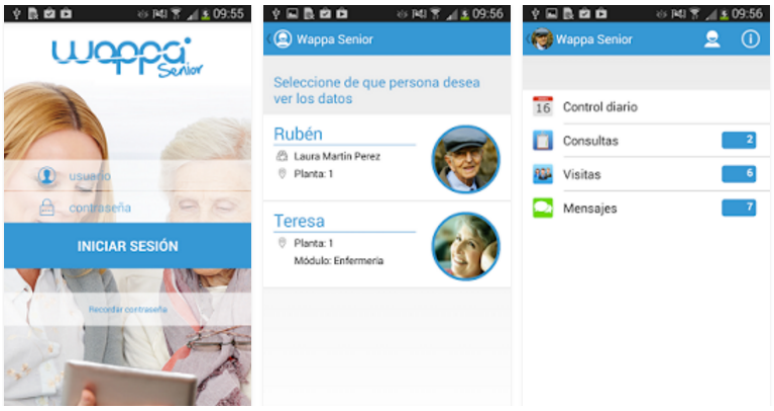


Figura 3.5: Vista Previa de la App Wappa Senior

Un punto a favor de este sistema es que registran y comparten con los usuarios vinculados a un mismo residente las visitas que este recibe en el centro. Esto facilita el reparto de franjas horarias para realizar visitas entre diferentes familiares y llevar un control cronológico de las mismas.

3.1.3. Sanyres

Figura 3.6: Icono de la App Sanyres

Desarrollador: Excelia SL[21]
Nº de descargas en Google Play: Entre 100 – 500
Página web oficial: www.excelia.com

Sanyres es una app desarrollada por una gran consultora tecnológica. Está enfocada como app de gestión integral para el trato con los clientes. No sólo informa de la actualidad del residente, también dispone de información genérica sobre la cadena de residencias. Como punto fuerte cabe destacar que permite consultar el histórico de facturas generadas hasta el momento.

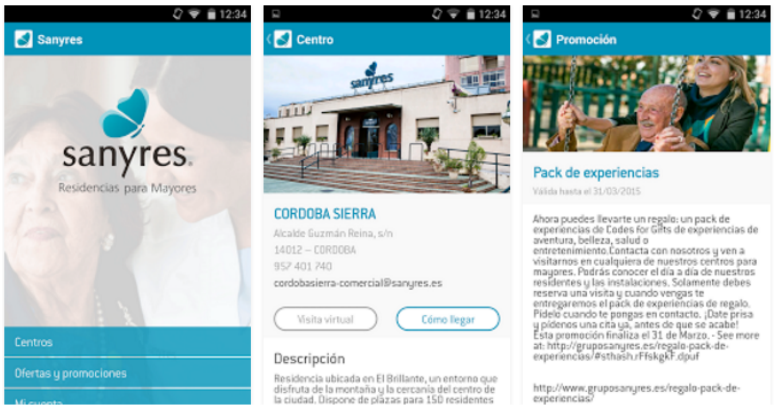


Figura 3.7: Vista Previa de la App Sanyres

Sanyres es un ejemplo de app corporativa, esto la aleja de CronCare que se presenta como una solución de «marca blanca». En el capítulo 13 (Modelo de Negocio) se explica en detalle las diferentes formas de comercializar una app como CronCare a diferentes empresas del mismo sector sin que estas tengan que renunciar a su imagen corporativa.

Capítulo 4

Metodología de Trabajo

4.1. Modelo de Ciclo de Vida

El modelo de ciclo de vida es un marco de referencia que contiene los procesos, las actividades y las tareas involucradas en el desarrollo del ciclo de vida del sistema desde la involucradas en el desarrollo, la vida del sistema desde la definición de los requisitos hasta la finalización de su uso.

Dado el aspecto académico de este proyecto, se ha tratado de hacer uso de los patrones de diseño, ciclos de vida y demás estándares aprendidos durante las diferentes asignaturas de la rama de Ingeniería del Software. Mediante estas técnicas se podrá lograr un sistema cuyo mantenimiento y futuras actualizaciones sean tareas relativamente sencillas para los desarrolladores y, a su vez, generar documentos de calidad que muestren la realidad del desarrollo.

Para el desarrollo de CronCare se hará uso del modelo por prototipado, este enfoque metodológico permite realizar evaluaciones sobre pequeñas funcionalidades y cambios introducidos al ritmo que se sacan nuevas versiones. Dado que necesariamente se generarán muchas versiones esto permite ir introduciendo elementos y funcionalidades cada vez más complejas de forma gradual.

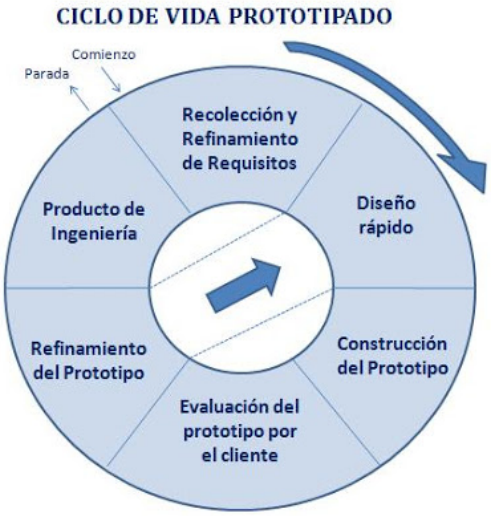


Figura 4.1: Modelo por Prototipado

El prototipado es eficaz para sacar actualizaciones inmediatas que mejoren errores detectados en muchas ocasiones por los propios usuarios del sistema. Lo cierto es que las dos mayores plataformas de distribución de apps, *Google Play[12]* y *AppStore*[10], han creado un entorno basado en este modelo: los desarrolladores pueden subir tantas actualizaciones de su aplicación como crean necesario y los usuarios pueden dar feedback inmediato sobre la experiencia con la nueva versión. Incluso es posible crear grupos de usuarios beta, con los que testear los cambios antes de añadirlos a la versión en producción.

Casi todas las empresas que compiten en el mercado de las apps han optado por este tipo de desarrollos. Basta fijarse en cuantas veces a la semana los smartphones piden permiso para descargar e instalar actualizaciones de una o varias apps que fueron cargadas en el sistema con anterioridad.

Durante las distintas etapas del desarrollo será necesario el uso de UML (Lenguaje Unificado de Modelado). Este lenguaje permite previsualizar las especificaciones, la construcción y la documentación del sistema.

En el entorno Android es necesario hacer uso de XML (Lenguaje de Marcas Extensibles) para definir cada una de las vistas de la interfaz gráfica, así como componentes individuales o arrays de datos.

El principio «no te repitas»[24] (en inglés Don't Repeat Yourself o DRY, también conocido como «una vez y sólo una») es una filosofía de definición de procesos que promueve la reducción de la duplicidad o redundancia. Durante el desarrollo de este proyecto se ha tratado de seguir esta filosofía.

Capítulo 5

Recursos Utilizados

En este capítulo se describen los diferentes recursos, tanto software como hardware, requeridos para el desarrollo del proyecto. Era un objetivo desde el principio, pero se hace hincapié en el uso que se hace de software firmado bajo licencias de libre distribución.

Gracias a diferentes plataformas para desarrolladores online comentadas a continuación, se ha podido ahorrar mucho tiempo al no tener que 'reinventar la rueda' en más de una ocasión. Esto permitió ceñirse aún más a las bases de la filosofía DRY, mencionada en el capítulo anterior.

5.1. Recursos Software

5.1.1. Herramientas de Desarrollo

Android Studio

Se trata de un IDE[1] distribuido de forma gratuita por *Google* que durante años estuvo en beta mientras muchos desarrolladores optaban por el ya establecido *Eclipse*. Hoy en día *Android Studio* es la herramienta favorita por la comunidad de desarrolladores de apps. Sus autocompletadores de código, junto a elementos predefinidos, rutinas y macros hacen de Android Studio una herramienta básica incluso para introducirnos en el desarrollo Android.

Sin lugar a duda, la gran ventaja de utilizar Android Studio frente al entorno Eclipse se encuentra en las etapas posteriores a la escritura del código en sí misma. El proceso de generación del fichero de producción final, listo para ser subido al Google Play, era tedioso e innecesariamente complicado. Android Studio supo agilizar y optimizar el proceso de firma y publicación, lo que llevó a la gran mayoría de los desarrolladores a migrar sus proyectos a este sistema.

[1] Un entorno de desarrollo integrado, en inglés Integrated Development Environment (IDE), es una aplicación informática que proporciona servicios integrales para facilitarle al desarrollador o programador el desarrollo de software.

Figura 5.1: Android Studio

Cloud9

Cloud9[4] es un IDE 100 % online que tan sólo requiere de un navegador web y una conexión a Internet para funcionar. El servicio ofrece gratuitamente generar una máquina virtual con el SO de nuestra elección y los módulos que se seleccionan de entre una larga lista (apache2, php5, SQL Server, etc.)

Una vez iniciada la máquina virtual se obtiene un entorno completo que integra un editor de texto, un navegador de archivos, un terminal de comandos, un cliente FTP y muchísimas más herramientas. Todo esto permite programar y testear contra un servidor 100 % online y funcional en cuestión de minutos.

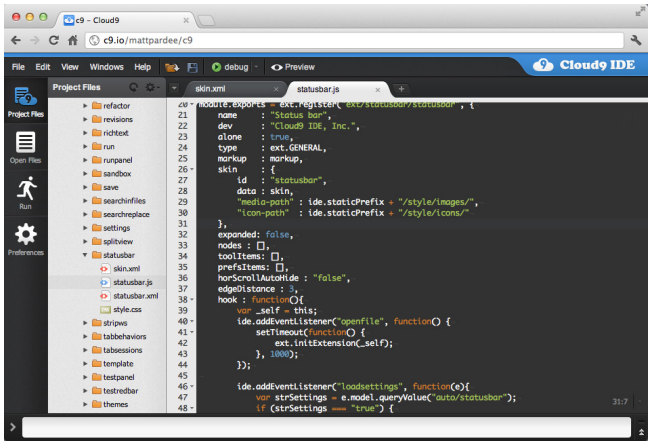


Figura 5.2: Entorno de desarrollo Cloud9

Durante la implementación de CronCare, para desarrollar muchas funcionalidades y testear nuevas versiones de la API, se ha utilizado una máquina virtual gratuita hospedada en Cloud9.

5.1.2. Sistema de Control de Versiones

El uso de un SCV ayuda, en caso de modificar un fichero, a visionar todas las etapas por las que ha pasado el mismo e incluso regresar a un estado anterior. Las modificaciones realizadas en los archivos se envían a un repositorio donde se almacenarán. Este repositorio puede ser accedido por más de un colaborador y también existen técnicas para la solución de conflictos sobre ficheros.

GIT

Según el libro Pro Git[3], Git fue diseñado para cumplir con los siguientes objetivos :

- Debe ser rápido.
- Debe tener un diseño sencillo.
- Debe tener un fuerte apoyo al desarrollo no-lineal (hasta miles de ramas paralelas).
- Debe ser totalmente distribuido.
- Git debe ser capaz de manejar grandes proyectos, como el núcleo de Linux, de manera eficiente.

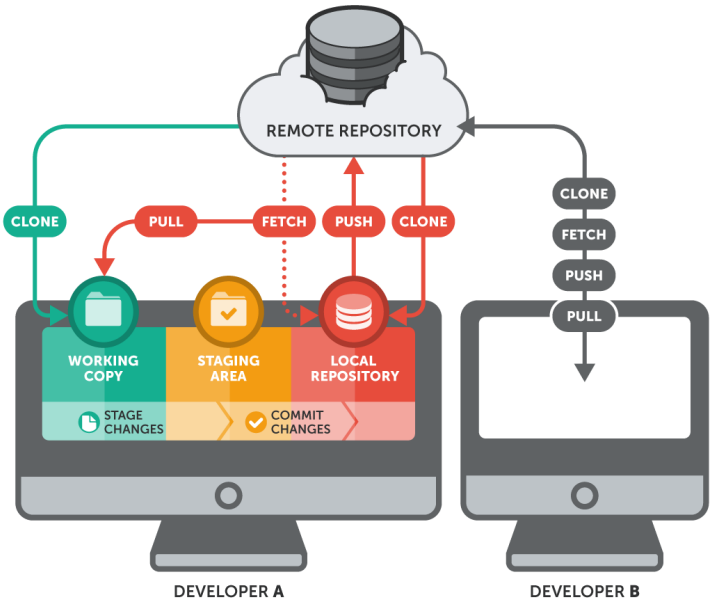


Figura 5.3: Workflow de GIT

BitBucket

Bitbucket es un servicio de alojamiento de repositorios desarrollado en Python e integrado en la web para proyectos que utilizan Mercurial o Git. La mayor ventaja que ofrece frente a otros servicios, es permitir un número ilimitado de repositorios privados (cada uno de los cuales pueden tener hasta cinco usuarios en el caso de las cuentas gratuitas).

Figura 5.4: Bitbucket

5.1.3. Software de Diseño

Sketch

Sketch es una herramienta de diseño vectorial totalmente enfocada hacia el diseño de interfaces de usuario. Su curva de aprendizaje es bastante equilibrada y tiene un precio mucho menor que su principal competidor Photoshop. Debido a su simplicidad, cualquier persona sin conocimiento previo sobre herramientas de diseño gráfico puede aprender tras un par de sesiones de trabajo.

El principal cambio que introducen aplicaciones como Sketch es el trabajo sobre elementos vectoriales. Esto quiere decir que en Sketch no se trabaja con imágenes formadas por píxeles, sino con formas geométricas que se escalan a necesidad sin perder nunca la resolución. Estas formas pueden sumarse o restarse y se pueden aplicar múltiples operaciones para generar formas complejas a partir de algunas más básicas.

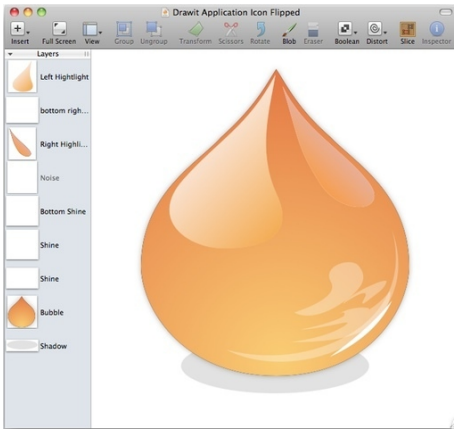


Figura 5.5: Vista Previa del Software de Edición Sketch

VioletUML

Esta es la lista que resume las principales características de VioletUML, un software para la generación de diagramas UML:

- Muy fácil de aprender y utilizar.
- Dibuja diagramas de aspecto agradable.
- Completamente libre.
- Multiplataforma.
- Enfocado a desarrolladores, estudiantes, maestros y el resto de autores que necesitan diagramas UML para producir.

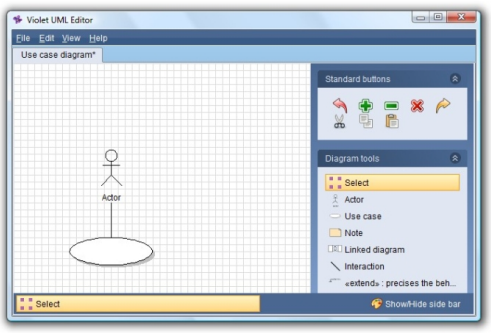


Figura 5.6: Vista Previa del Software VioletUML

5.1.4. Otros

LyX

LyX es un procesador de documentos que genera ficheros LaTeX[2]. LyX combina la potencia y flexibilidad de TeX/LaTeX con la facilidad de uso de una interfaz gráfica. En LyX tenemos una barra repleta de opciones que pudieran ser familiares de entornos como Microsoft Office. De esta forma, insertar imágenes, enlaces o generar la bibliografía son labores mucho más sencillas.

Así pues no se puede decir que LyX sea un simple editor de texto, pero tampoco se puede asegurar que siempre alcance el nivel de detalle que se puede lograr al escribir código LaTex directamente.

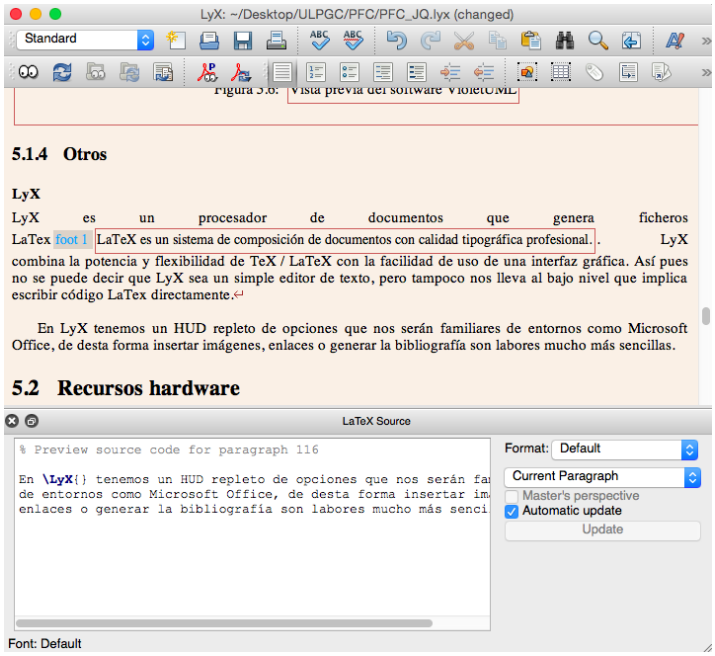


Figura 5.7: Vista Previa del Software LyX

[2]LaTeX es un sistema de composición de documentos con calidad tipográfica profesional.

Advanced Rest Client

ARC es una aplicación que corre sobre el navegador Google Chrome, está distribuida bajo licencia de código abierto Apache 2.0 y se puede contribuir al proyecto en github en: https://github.com/jarrodek/ChromeRestClient
Con esta aplicación se pueden configurar las llamadas HTTP que se quiere probar con cabeceras y de datos propios. Una herramientas de este tipo es imprescindible para desarrollar una API. A continuación se enumeran sus principales características:

- Editor visual de cabeceras y cuerpo de mensajes HTTP.
- Visor de HTML, JSON y XML.
- Autocompleta el texto de las cabeceras HTTP a modo de ayuda.
- Guardar peticiones como ficheros o en servicios de almacenamiento en la nube (Google Drive, Dropbox, etc.).
- Crear colecciones de llamadas para separar proyectos.
- Se integra con el historial de Google Chrome y autocompleta las URLs.
- Importación/Exportación de datos.

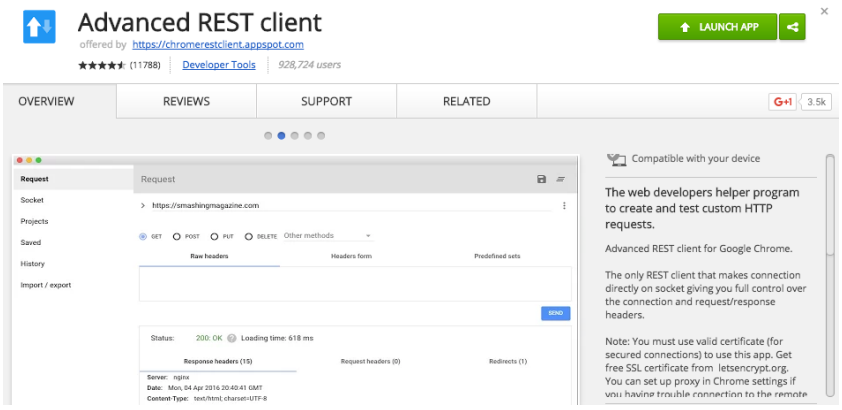


Figura 5.8: Pantalla de Descarga del Software ARC

5.2. Recursos Hardware

Ya que todo el proyecto se ha llevado a cabo haciendo uso de software disponible al gran público, los únicos recursos hardware utilizados para implementar CronCare han sido:

- Ordenador portátil - Apple Macbook Air 13":
  - Pantalla de 13.3 pulgadas a 1440x900.
  - 1.3GHz Intel Core i5-4250U.
  - 4,096MB DDR3 SDRAM 1,600MHz.

- 1,024MB Intel HD Graphics 5000.
- 128GB SSD.
- 802.11a/c wireless, Bluetooth 4.0 OS X Mountain Lion 10.8.4.

- Teléfono Móvil - BQ Aquaris E5 HD:

  - Pantalla IPS de 5 pulgadas con Dragon Trail Resolución 720 x 1280, 294 ppp.
  - Procesador MediaTek Quad Core Cortex-A7, @1.3 GHz.
  - Procesador gráfico Mali 400 MP2 RAM 2 GB.
  - Memoria 8/16 GB + microSD hasta 64 GB.
  - Versión Android 4.4.2 (KitKat).
  - Conectividad Wireless 802.11b/g/n Bluetooth 4.0 LE.
  - Puertos de expansión MicroUSB MicroSD (hasta 64 GB).
  - Cámaras 13 Mpx flash dual. 5 Mpx en la frontal.
  - Batería Batería Li-Po, 2.500 mAh.
  - Dual SIM.

5.3. Recursos 'Cloud'

5.3.1. Google APIs

Google pone a disposición de los desarrolladores un conjunto de APIs que permiten la comunicación e integración con sus principales servicios. Algunos de los más utilizados son: Gmail, Google Maps o Google Translate.

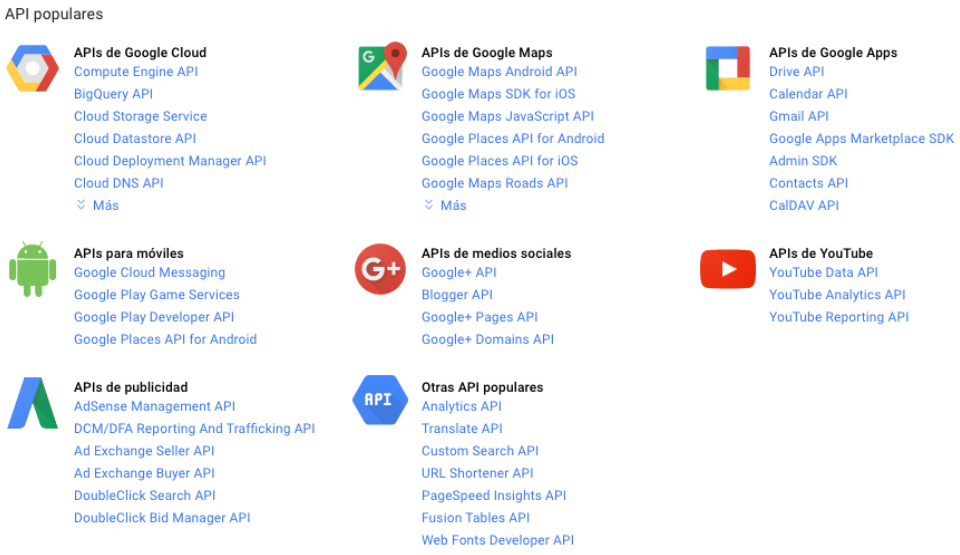


Figura 5.9: Listado de APIs Ofrecidas por Google Para Desarrolladores

Lo más importante que ofrecen estas APIs es el acceso a los datos del usuario cuando este ha otorgado los permisos correspondientes. De entre todas las APIs que Google pone a nuestra disposición, en CronCare tan solo se hace uso del servicio Google Cloud Messaging, el cual se explica en detalle en la siguiente sección.

Google Cloud Messaging

GCM es un servicio gratuito que permite a los desarrolladores enviar mensajes entre servidores y aplicaciones de cliente. El envío puede ser tanto de aplicación a servidor como de servidor a aplicación. Por ejemplo, un mensaje muy ligero podría informar a una aplicación cliente de que hay nuevos datos disponibles en el servidor, como en el caso de una notificación PUSH diciendo «nuevo correo sin leer».

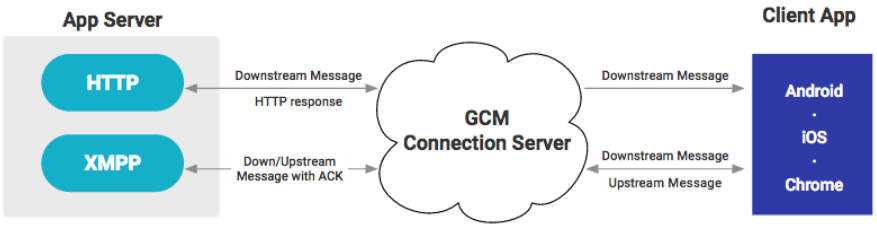


Figura 5.10: Google Cloud Messaging

Para los casos de uso como la mensajería instantánea, un mensaje GCM puede transferir hasta 4 kb de carga útil a la aplicación cliente, esto puede ser un mensaje corto entero, o bien, el principio de la cadena de texto con un símbolo '...' al final. El servicio GCM se encarga de todos los aspectos de gestión de colas de mensajes y entrega hacia y desde la aplicación cliente.

En el caso de CronCare se genera una notificación push en el servidor tras una acción de toma de datos por parte de algún cuidador del centro. Esa notificación será enviada a todos las apps de familiares o usuarios vinculados al residente.

Capítulo 6

Planificación Temporal

Tal y como se explica en el capítulo 4, en el ciclo de vida de este proyecto se hace uso del modelo por prototipado, esto entraña cierta dificultad a la hora de representar la carga de trabajo sobre un eje temporal. La solución pasa por crear un diagrama de tarta que muestre la fracción porcentual de tiempo que abarca cada una de las etapas.

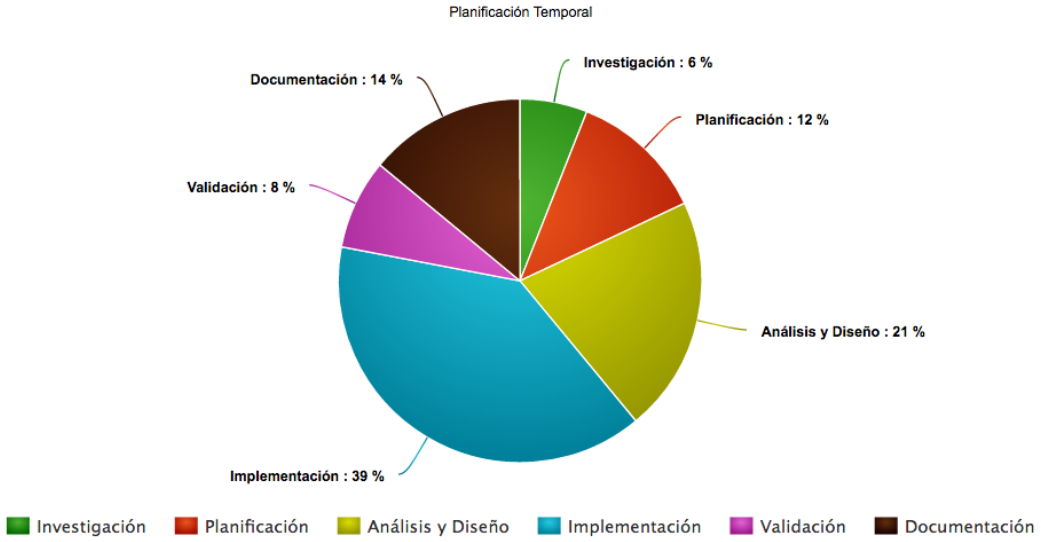


Figura 6.1: Gráfica de la Planificación Temporal

La duración total aproximada del proyecto es de unas 950 horas. A continuación se muestra el desglose de cada una de estas etapas en tareas menores.

- Investigación previa y toma de contacto con las tecnologías (6 %)
  - Estudio del estado del arte.
  - Investigación sobre desarrollo Android.
  - Estudio sobre el desarrollo de APIs RESTFUL.
- Planificación (12 %)
  - Definición de requisitos funcionales y no funcionales.

  - Primeras entrevistas a responsables de centros de atención especializada.
  - Establecer plazos con supervisión del tutor.

- Análisis y diseño (21 %)

  - Definición de los casos de uso.
  - Definición de clases del sistema.
  - Diseño de la base de datos.
  - Definición de las interfaces de las apps.

- Implementación (39 %)

  - Implementación de las interfaces.
  - Implementación de la API RESTFUL sobre un servidor Apache.
  - Integración de las llamadas a la API desde las apps.

- Validación (8 %)

  - Definición de pruebas.
  - Aplicación de pruebas.
  - Análisis de resultados.
  - Corrección de errores.

- Documentación (14 %)

  - Informes sobre pruebas.
  - Documentación final.

Capítulo 7

Definición del Proyecto

Una vez estudiados los problemas a resolver y tras varias reuniones con personal de Centros de Atención Especializada, así como con familiares de sus residentes, se concluye implementar dos apps android como dos proyectos paralelos: CronCare y CronCare Centros.

7.1. Descripción y Alcance del Sistema

Para lograr interfaces intuitivas y fáciles de usar se hace uso de los patrones de diseño recomendados, ya que los usuarios esperan ciertos comportamientos similares entre apps como pueden ser los menús básicos y la forma de mostrar ciertos tipos de datos.

CronCare y CronCare Centros se comunicarán con un servidor a través de una API RESTFUL. Esta API es capaz de procesar las llamadas de diferentes tipos de usuario para satisfacer las necesidades de ambas apps, la figura 7.1 muestra este esquema.

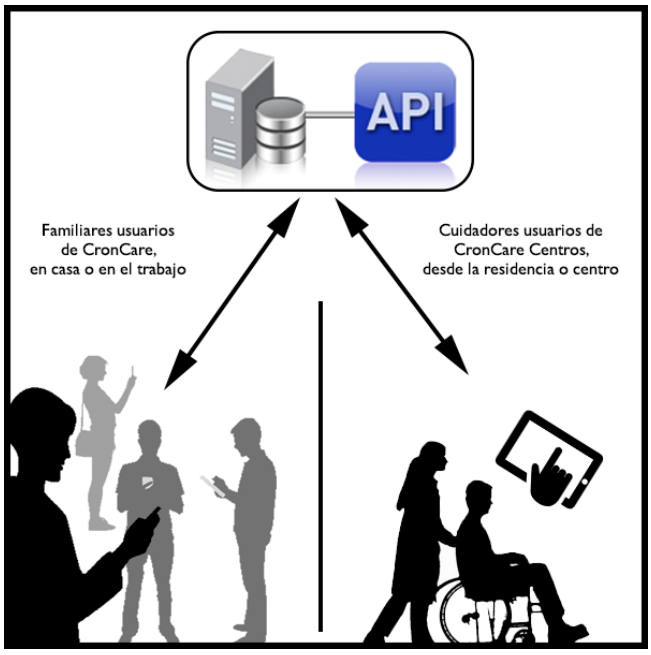


Figura 7.1: Esquema de la Comunicación del Sistema

7.1.1. CronCare

Se trata de la app para familiares o personas interesadas en recibir información por parte del centro sobre algún residente. Para ello tendrán que darse de alta en la app y darle al centro la dirección de e-mail que usaron. Esta cuenta de usuario, cuya dirección de e-mail es única, será vinculada al residente solicitado desde el centro. Desde ese momento tendrán acceso a toda la información que está en el sistema referente al residente, así como a las notificaciones del centro, el menú semanal de comida o las gráficas estadísticas de evolución de estado, sueño y comida.

CronCare está pensada como una app para teléfonos, aunque funciona sin problema en tablets también. Esto se debe a que el tipo de información que ofrece es 'importante' en cualquier momento. El teléfono móvil tiene más probabilidad de estar cerca del usuario de forma inmediata.

7.1.2. CronCare Centros

En el caso de CronCare Centros, la app está optimizada para su ejecución en tablets, ya que ofrecen mayor autonomía, espacio en pantalla, legibilidad y carece de accesorios hardware 'inútiles' para el fin de este proyecto, como podrían ser el teléfono, el gps, la radio, etc.

Esta app fue diseñada para ahorrar tiempo a quienes la usan. En ella se van a tomar nota de todos los eventos que ocurren en el centro para los residentes. Por ello hacen falta botones e iconos grandes y claros, fuente simple y nítida y capacidad de ejecutar acciones en lote para optimizar la toma de datos.

7.1.3. API RESTFUL

El sistema implementa una llamada a la API para cada acción que se ejecuta en ambas apps para interactuar con el sistema. Por norma general las llamadas que sólo necesitan leer datos del sistema responden a llamadas de tipo GET y aquellas que afectan al estado de los datos utilizan el tipo POST.

La API de CronCare está dividida en varios módulos:

- Módulo de Administradores
  - Este módulo procesa llamadas para crear los recursos del centro o residencia en el sistema. Sólo implementa llamadas de tipo POST y es desde dónde se crean, editan y eliminan perfiles de cuidadores y residentes, grupos para categorizar residentes, mensajes a familias y el menú semanal.
- Módulo de Cuidadores
  - Es el encargado de recibir las llamadas que se hacen desde la app CronCare Centros, utilizada desde las residencias por sus empleados. Consta de llamadas tipo GET y POST. Las primeras para recibir datos de residentes y poder interactuar con ellos dentro de la app, las llamadas tipo POST sirven para dar acceso al sistema y registrar los distintos eventos en la toma de datos (entrada, comedor, observación, etc.)

- Módulo de Familiares

  - La app CronCare, utilizada por los familiares de los residentes en cualquier lugar desde sus teléfonos, realiza sus llamadas a la API directamente contra este módulo. Existen algunas llamadas destinadas a enviar datos (tipo POST) como las de registro de nuevo usuario, login o envío de notas cortas. Sin embargo, la mayoría de las llamadas que implementa son de tipo GET, destinadas al consumo de información. La app de familiares solicita información sobre los residentes vinculados, mensajes del centro, menú semanal o estadísticas de evolución.

La figura 7.2 ofrece un esquema de la organización de la API.

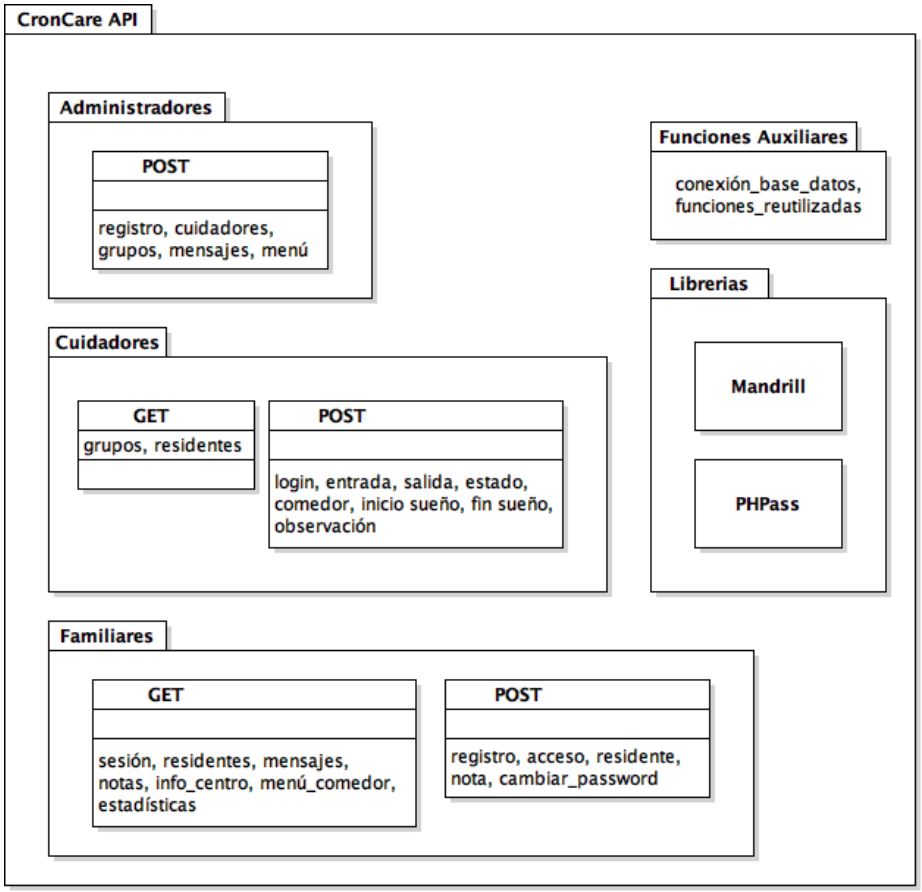


Figura 7.2: Descripción de la API de CronCare

Todas las llamadas a la API responden a URLs del tipo

`http://<ruta_API_croncare>/<tipo_usuario>/<accion>/<parámetros?>`

independientemente de que sean de tipo GET o POST, la diferencia radica en la forma de pasar los parámetros adicionales. GET pasa sus variables en el propia URL, al final, mientras que POST agrega un payload al paquete de datos dónde viajan los pares de valores.

De esta manera, se entiende que el acceso de un usuario a la app CronCare se realiza mediante una llamada del tipo:

```
POST http://<ruta_API_croncare>/familiar/session/
```

con el siguiente payload:

```
username=joaquin@demo.com&password=1p4ssS3gur0
```

Esta llamada a la API nos devolverá un token único para poder realizar llamadas futuras seguras al sistema. En la sección 11.2 se muestra a modo de ejemplo, una traza la de interacción con la API para registrar un usuario y luego acceder con él al sistema.

7.2. Requisitos Funcionales

Los requisitos funcionales definen el marco de funcionamiento del sistema, así como su alcance. La suma de estos requisitos representa «lo que el software debe hacer». Estos son los requisitos funcionales de CronCare:

- Los cuidadores accederán al sistema mediante códigos numéricos de 4 dígitos autogenerados por el sistema.
- Los familiares accederán a su app mediante un e-mail y una contraseña de su elección.
- Ambas apps deben poder mostrar el histórico de datos de los residentes a los que tengan acceso de forma cronológica.
- La app CronCare debe mostrar mediante gráficas de evolución los datos de los últimos 30 días sobre estado, ritmos de sueño y comedor.
- Ambas apps deben poder acceder a todas sus pantallas en menos de 3 'taps'.
- Los usuarios de CronCare deben poder cambiar su e-mail de acceso así como su contraseña.
- Los usuarios de CronCare deben poder enviar notas de longitud limitada al centro siempre y cuando el residente referido no se encuentre en el centro.
- Los cuidadores podrán tomar cualquiera de los datos tipificados en la app CronCare Centros de manera individualizada para un sólo residente o 'en lote' a más de uno con el mismo valor.
- Cada vez que un cuidador tome un dato mediante la app CronCare Centros, los usuarios vinculados a los residentes afectados que dispongan de la app CronCare recibirán una notificación PUSH en su dispositivo.
- El listado de residentes en la app CronCare Centros ayuda a evitar errores mediante filtros lógicos, por ejemplo, no muestra la opción de tomar datos de comedor a residentes que están dormidos.

7.3. Requisitos No Funcionales

A diferencia del punto anterior, los requisitos no funcionales definen otro aspecto del software, más relacionado con su aspecto visual, operatibilidad y mantenimiento. Para el presente proyecto se han definido los siguientes:

Disponibilidad

- El sistema estará disponible las 24 horas del día los 365 días del año y desde cualquier sitio con conexión a Internet.
- El sistema será compatible con dispositivos de tipo smartphone o tablet Android.
- La app de familiares no tiene un límite en el número de dispositivos que un usuario puede iniciar sesión.
- La app de cuidadores sólo permite a un trabajador iniciar sesión en un dispositivo a la vez.

Rendimiento

- Las operaciones que requieran llamadas al servidor desde las apps no bloquearán la UI y se realizarán de forma asíncrona.
- La latencia entre el servidor y los clientes no debe superar los límites normales ni interferir con la experiencia de uso.
- La app debe tener control sobre su ciclo de vida y prevenir eventos como recibir llamadas telefónicas durante su uso.

Capítulo 8

Estudio Previo

CronCare es un proyecto que bien podría llevar a cabo cualquier empresa de desarrollo software en la actualidad. El hecho de que exista competencia pero no demasiada indica que nos encontramos ante una idea con potencial pero innovadora. Es necesario saber como se llevan a cabo los proyectos del mundo profesional y que herramientas utilizan para el desarrollo de sistemas basados en apps Android. En este capítulo se hablará sobre el SO Android y su historia, pero también sobre otros elementos como los servicios web o el XML que, combinados nos proporcionan el ecosistema ideal para este proyecto.

8.1. Android

El sistema operativo Android es un sistema de código abierto que se utiliza principalmente en dispositivos móviles. Está escrito principalmente en Java y basado en el sistema operativo Linux. Fue desarrollado inicialmente por Android Inc. y finalmente adquirido por Google en 2005.

8.1.1. Historia y Futuro de Android

La empresa arranca en octubre de 2003 cuando Andy Rubin, Rich Miner ,Nick Sears y Chris White deciden fundar Android Inc. en Palo Alto, California. Dos años más tarde Android Inc. fue adquirida por Google por una cantidad que se ha mantenido en secreto y se convirtió en una empresa subsidiaria. Aún así los empleados clave se mantuvieron tras la adquisición.

En los dos años que siguieron el primer SDK Beta vio la luz para uso de los desarrolladores y fabricantes de teléfonos. Al estar basado en el núcleo de Linux resulta un sistema flexible y actualizable.

Google también ha tratado de coger su trozo de pastel en el mercado de las tabletas. Aunque se producen tabletas con Android 2.x desde 2010 (como la Samsung Galaxy Tab), el sistema operativo Android 3.0 Honeycomb fue lanzado a principios de 2011 y fue diseñado específicamente para tabletas.

La versión de Android de 2016 dista mucho de la primera en 2008, donde el Play Store aún no cubría un gran mercado y no había capacidades de reproducción de vídeo nativas ni soporte

multi-touch. En pocas palabras, era un SO con muchas carencias.

Sin embargo, Google hizo una gran adquisición y supo hacer de Android un éxito. Cabe esperar grandes cosas de la compañía de Palo Alto.

8.1.2. Arquitectura del Sistema Android

El siguiente diagrama muestra la arquitectura del sistema operativo Android:

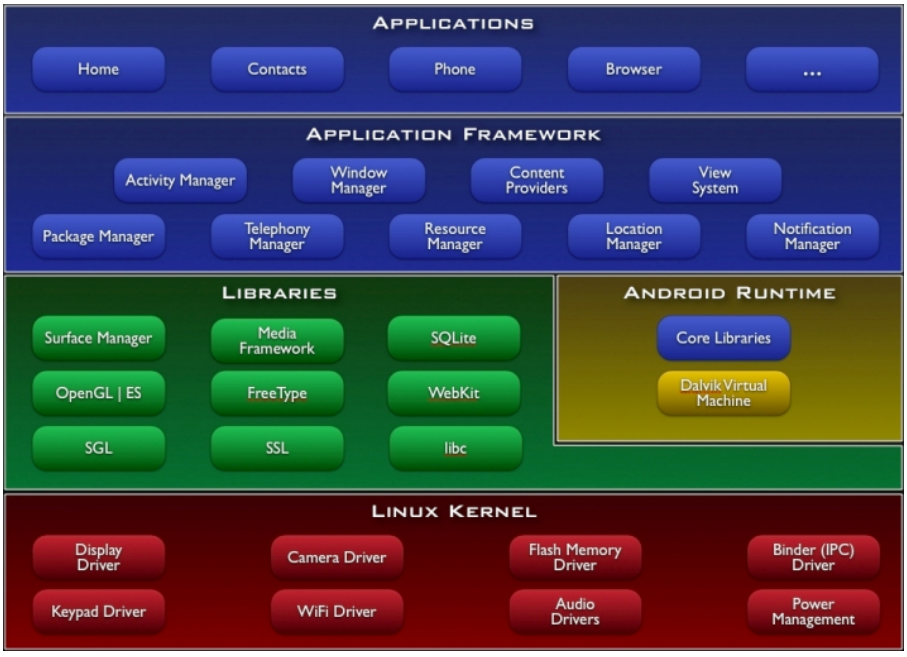


Figura 8.1: Diagrama de la Arquitectura de Android

Básicamente Android se compone de las siguientes capas:

- Aplicaciones (escritas en Java, ejecutadas en Dalvik)
- Framework y librerías
  - La mayoría de aplicaciones y frameworks se ejecutan en una máquina virtual.
- Librerías nativas, demonios y servicios (escritos en C y C++)
- Kernel Linux, que incluye:
  - Drivers para elementos hardware, protocolos de red, acceso al sistema de ficheros y comunicación entre procesos.

8.1.3. Ciclo de Vida de Una App Android

Las aplicaciones Android se basan en múltiples componentes, el más visible de ellos es la clase «Activity» que define una actividad. En este caso el término "visible" es literal, pues esta

clase se encarga de dibujar las vistas en la pantalla.

Siempre que una app se está ejecutando hay una actividad con el foco, por tanto el ciclo de vida de una app está directamente vinculado al ciclo de vida de las actividades.

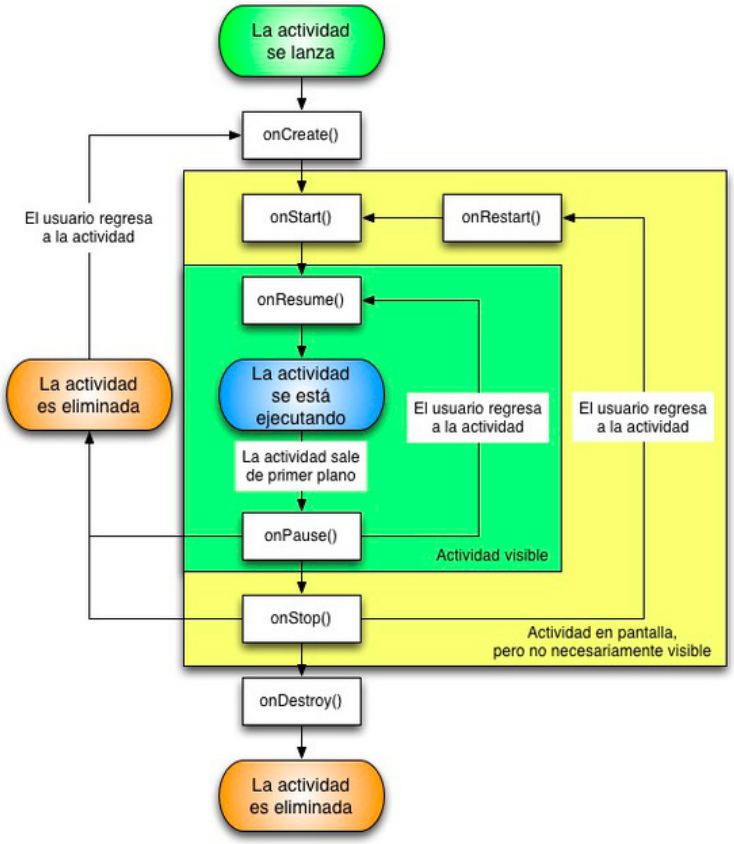


Figura 8.2: Ciclo de Vida de una Actividad en Android

El ciclo de vida de una aplicación Android es muy diferente al de una aplicación en otros SO, pues en Android el ciclo de vida está totalmente controlado por el propio SO.

- onCreate (Bundle):
  - Se llama en la instanciación de la clase Activity. Se utiliza para realizar todo tipo de inicializaciones en la clase, como pueden ser la creación de la interfaz de usuario o la inicialización de estructuras de datos. Puede recibir información de estado de otra actividad que la invoca (en una instancia de la clase Bundle).
- onStart():
  - Nos indica que la actividad está a punto de ser mostrada al usuario.
- onResume():
  - Se llama cuando la actividad va a comenzar a interactuar con el usuario.

- onPause():
  - Indica que la actividad está a punto de ser lanzada a segundo plano, normalmente porque otra actividad es lanzada. Es el lugar adecuado para almacenar los datos que estaban en edición.
- onStop():
  - La actividad ya no va a ser visible para el usuario. Pero si hay muy poca memoria disponible, es posible que la actividad se destruya sin llamar a este método.
- onRestart():
  - Indica que la actividad va a volver a ser representada después de haber pasado por onStop().
- onDestroy():
  - Se llama antes de que la actividad sea totalmente destruida. Pero, si hay muy poca memoria disponible, es posible que la actividad se destruya sin llamar a este método.

8.1.4. SDK Android

El SDK de Android (kit de desarrollo de software) es un conjunto de herramientas de desarrollo utilizadas para escribir aplicaciones destinadas a la plataforma Android. El SDK de Android incluye lo siguiente:

- Bibliotecas necesarias.
- Depurador.
- Un emulador.
- La documentación pertinente para las interfaces de programación de aplicaciones de Android.
- Código fuente de ejemplos y tutoriales para el sistema operativo Android.

Cada vez que Google lanza una nueva versión de Android, también da a conocer su nuevo SDK correspondiente. Para poder escribir programas con las últimas características, los desarrolladores deben descargar e instalar el SDK de cada versión para los nuevos dispositivos.

Los SDK de Android son compatibles con Windows (XP o posterior), Linux (cualquier distribución Linux reciente) y Mac OS X (10.4.9 o posterior). Los componentes del SDK se pueden descargar también por separado.

8.1.5. Estructura de un Proyecto en Android Studio

Aunque existen otros entornos de desarrollo, este documento se centra en el análisis de Android Studio pues es la herramienta con la que se han desarrollado CronCare y CronCare Centros.

Un proyecto en Android Studio contiene todo lo que define el espacio de trabajo de una aplicación, desde el código fuente y los recursos estáticos a módulos de test para probar el código y crear configuraciones. Al iniciar un nuevo proyecto, Android Studio crea la estructura necesaria para todos sus archivos y los hace visibles en la ventana «Proyecto».

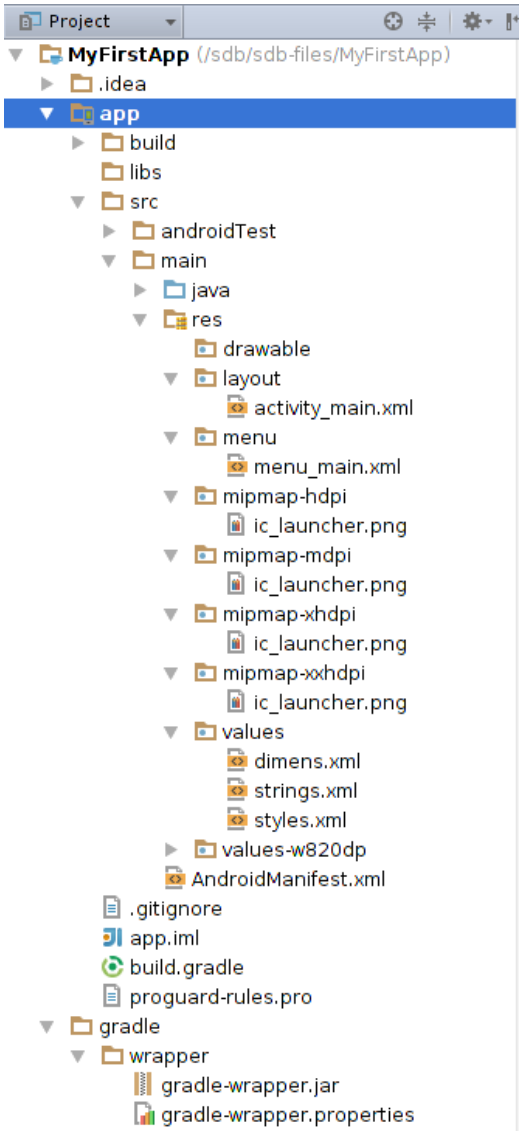


Figura 8.3: Estructura de un Proyecto en Android Studio

Por defecto, Android Studio muestra los archivos del proyecto en la vista de «Android». Este punto de vista no refleja la jerarquía de archivos real en el disco, pero está organizado por módulos y tipos de archivos para simplificar la navegación entre archivos fuente, el sistema

oculta ciertos archivos o directorios que no se usan comúnmente.

8.2. XML y Android

XML, siglas en inglés de eXtensible Markup Language (Lenguaje de Marcas Ampliable), es un metalenguaje extensible de etiquetas desarrollado por el World Wide Web Consortium (W3C). Es una simplificación y adaptación del SGML, que al igual que HTML, son lenguajes que nos permiten definir diferentes gramáticas para lenguajes específicos.

XML se extiende más allá de su uso en Internet y hoy en día es un estándar para el intercambio de información estructurada entre distintas plataformas. Se puede usar en bases de datos, editores de texto, hojas de cálculo y existen innumerables servicios online para tratar este tipo de ficheros.

Para Android, XML es esencial en muchos aspectos, por ejemplo a la hora de diseñar la interfaz gráfica, ya que se pueden estructurar los componentes de las distintas vistas declarándolos en ficheros XML. Si bien es posible definir una vista usando meramente código Java, esta es una praxis compleja y no recomendable para proyectos ágiles. Dentro de los ficheros de vistas XML que definimos en Android, podemos hacer uso de muchos componentes que el propio sistema nos ofrece y que iremos conociendo a los largo del documento.

8.3. Web-Services

8.3.1. APIs RESTUFL

API es un acrónimo en inglés para «Application Programming Interface» (Interfaz de Programación de Aplicaciones). Se entiende una API como el conjunto de subrutinas, funciones y procedimientos (o métodos, en la programación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.

Existen múltiples tipos de APIs, desde visuales hasta APIs para dar órdenes mediante voz humana a sistemas, pero este proyecto se centra en las APIs denominadas RESTFUL y no se puede definir el concepto completo sin explicar que es REST.

La Transferencia de Estado Representacional (Representational State Transfer) o REST es un estilo de arquitectura software para sistemas hipermedia distribuidos como la World Wide Web. El término se originó en el año 2000, en una tesis doctoral sobre la web escrita por Roy Fielding, uno de los principales autores de la especificación del protocolo HTTP y ha pasado a ser ampliamente utilizado por la comunidad de desarrollo.

Existen una serie de requisitos en los sistemas REST, y se dice que un sistema RESTFUL los cumple todos. A continuación se listan los requisitos de REST:

- Un protocolo cliente/servidor sin estado:

    - Cada mensaje HTTP contiene toda la información necesaria para comprender la petición. Como resultado, ni el cliente ni el servidor necesitan recordar ningún estado de las comunicaciones entre mensajes. Sin embargo, en la práctica, muchas aplicaciones basadas en HTTP utilizan cookies y otros mecanismos para mantener el estado de la sesión.

- Un conjunto de operaciones bien definidas que se aplican a todos los recursos de información:

    - HTTP en sí define un conjunto pequeño de operaciones, las más importantes son POST, GET, PUT y DELETE. Con frecuencia estas operaciones se equiparan a las operaciones CRUD en bases de datos (ABMC en castellano: crear, leer, actualizar, borrar) que se requieren para la persistencia de datos. En un sistema REST cada recurso es direccionable únicamente a través de su URI.

- El uso de hipermedios, tanto para la información de la aplicación como para las transiciones de estado de la aplicación:

    - La representación de este estado en un sistema REST son típicamente HTML o XML. Como resultado de esto, es posible navegar de un recurso REST a muchos otros, simplemente siguiendo enlaces sin requerir el uso de registros u otra infraestructura adicional.

Muchas empresas de software conocidas, como Google, Amazon, Twitter o LinkedIn hacen uso de APIs RESTFUL tanto de cara a sus usuarios como a sus respectivas comunidades de desarrolladores.

Capítulo 9

Análisis

El hecho de adoptar un modelo basado en prototipos implica que todas las fases del proyecto evolucionen, el análisis no es una excepción. Es por ello que en este capítulo se muestra el resultado de todas las fases aunado. Los actores, casos de uso y demás elementos son el fruto de todas las iteraciones del proyecto y representan al producto final con todas sus funcionalidades.

9.1. Actores del Sistema

- Administrador
    - Actúa con el sistema mediante llamadas a la API, tienes la capacidad de crear recursos para el centro o residencia en el sistema. Crea perfiles de cuidadores, administra a los residentes, envía mensajes generales, etc.
- Familiar o interesado en el seguimiento de un residente
    - Este actor hará uso de la app CronCare. Excepto por el envío de notas cortas en caso de ausencia del residente en el centro y el propio registro de usuario en sí, este actor no escribe datos en el sistema. Es un consumidor de información que accede a la mayoría de sus datos mediante llamadas tipo GET a la API.
- Cuidador o trabajador del centro:
    - Mediante la app CronCare Centros, este actor nutre al sistema de datos en el momento que los eventos ocurren. Su función es la de tipificar estos registros entre entrada/salida, inicio/fin de sueño, evento de comedor, registro del estado anímico/-físico y observación libre.

9.2. Diagramas de Casos de Uso

Dado que se van a implementar dos apps se ha tratado de plasmar en dos diagramas de casos de uso todas las funcionalidades a las que tienen acceso los usuarios para cada una de ellas:

9.2.1. Diagrama de Casos de Uso para CronCare:

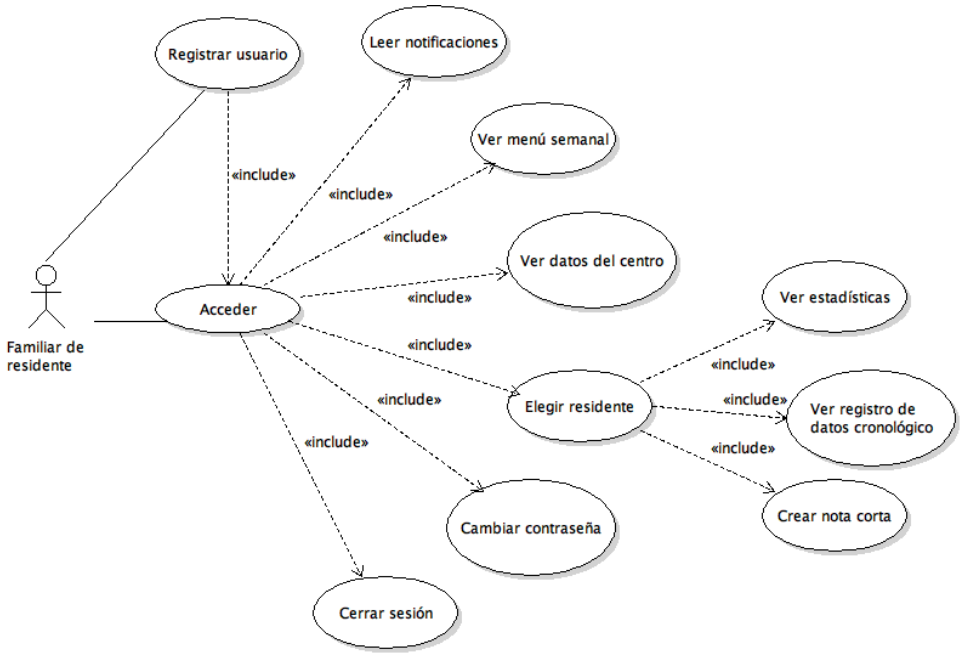


Figura 9.1: Diagrama de Casos de Uso nº1 - CronCare Para Familiares de Residentes

9.2.2. Resumen de los Principales Casos de Uso del Diagrama 9.1

A continuación se muestra un resumen de los casos de uso para facilitar la lectura de este documento. El listado completo se encuentra en el anexo A.1

- Registrar usuario
  - Permite registrar una nueva cuenta de usuario. Se solicitará al menos un e-mail válido y una contraseña.
- Acceder
  - Permite acceder al menú principal de la app, donde están disponibles el resto de funcionalidades del sistema.
- Leer notificaciones
  - Accede al listado de notificaciones recibidas desde el centro. Esta vista es similar a una bandeja de entrada de un gestor de correo electrónico.
- Ver menú semanal
  - Lista el menú de comedor establecido por el centro para la semana actual.
- Ver datos del centro

  - Permite visualizar datos como el teléfono, e-mail o dirección postal del centro. En caso de disponer de apps capaces de interpretar estos datos como parámetros de entrada se podrán ejecutar desde esta vista. Por ejemplo utilizar Google Maps[11] para llegar hasta la dirección mostrada por el centro.

- Elegir residente

  - En caso de tener más de un residente vinculado a la cuenta de usuario, el menú principal mostrará un botón para acceder a cada uno de ellos. Estos botones serán personalizables con la imagen deseada, preferiblemente una del residente en cuestión.

- Ver estadísticas

  - En esta pantalla se accederá a los registros tomados durante los últimos 30 días en forma de gráficas de evolución. Estarán disponibles las gráficas de estado anímico/-físico, calidad del sueño y comedor.

- Ver registro de datos cronológico

  - Esta es la vista por defecto de la app. Al lanzar la app se cargará el registro de eventos del último día disponible en el sistema, a partir de ahí se podrá navegar hacia delante o hacia atrás mediante flechas.

- Crear nota corta

  - Esta opción sólo está disponible cuando el residente figura como ausente del centro en el sistema. De esta forma el familiar tiene la opción de crear un mensaje de hasta 300 caracteres que será leído en el centro cuando el próximo acceso del residente sea registrado. Ejemplo: Crear una nota antes de llevar al familiar que diga «Por favor, durante el día de hoy no le den café. Muchas gracias.»

- Cambiar contraseña

  - Es indispensable proporcionar la contraseña anterior para poder actualizarla a una nueva.

- Cerrar sesión

  - Permite salir del sistema y dejar de recibir notificaciones en el dispositivo.

9.2.3. Diagrama de Casos de Uso para CronCare Centros:

Figura 9.2: Diagrama de Casos de Uso nº2 - CronCare Centros

9.2.4. Resumen de los Principales Casos de Uso del Diagrama 9.2:

- Acceso PIN
  - Esta será la única opción disponible en la primera ejecución de la app. Cada perfil de cuidador o empleado tiene asociado un código numérico de 4 dígitos único en el sistema autogenerado en el momento de su creación.
- Ver registro de datos cronológico de un residente
  - Al mantener pulsado el nombre de un residente se cargará el registro de eventos del último día disponible en el sistema, a partir de ahí se podrá navegar hacia delante o hacia atrás mediante flechas.
- Ver notas cortas sobre un residente
  - Al mantener pulsado el icono de nota corta, situado a la derecha del nombre de un residente, se cargará el registro de notas del último día disponible en el sistema, a partir de ahí se podrá navegar hacia delante o hacia atrás mediante flechas.
- Registrar evento en el sistema
  - Envía al sistema un registro del tipo indicado para el/los residente(s) seleccionado(s) con los valores introducidos.
- Seleccionar tipo de evento

  - Permite especificar si se va a enviar un evento de tipo entrada/salida, control de estado anímico/físico, comedor, inicio/fin de sueño u observación libre.

- Filtrar grupos de residentes

  - Todos los residentes pertenecen a un grupo que fue definido al ser dado de alta. En el listado de grupos se pueden activar múltiples opciones, eligiendo así cuales queremos visualizar en el listado de residentes.

- Seleccionar/Deseleccionar residentes

  - El listado de residentes también permite la selección múltiple de filas. Así se puede asignar un mismo valor para un registro a varios residentes sin duplicidad de operaciones.

- Añadir observación

  - Todos los tipos de eventos, a excepción de la observación libre, permiten añadir una observación de texto de manera opcional en la misma pantalla que se introduce el valor deseado para el registro.

Capítulo 10

Diseño

Según McGlaughlin[16], «hay tres características que sirven como parámetros generales para la evaluación de un buen diseño», se enumeran a continuación:

1. Un buen diseño debe implementar todos los requisitos explícitos obtenidos en la etapa de análisis.
2. Debe ser una guía que puedan leer y comprender los que construyen el código y los que prueban y mantienen el software.
3. Debe proporcionar una idea completa de lo que es el software.

10.1. Arquitectura del Sistema

El diseño de la arquitectura de software indica la estructura, el funcionamiento y la interacción entre los distintos módulos que componen el software. La definición de una arquitectura establece nuestro marco de trabajo, es por ello que debemos diseñar en base a objetivos y restricciones.

Dado que tratamos un proyecto basado en apps que interactúan mediante una API y un servidor intermedio, la arquitectura a implementar es el modelo cliente-servidor.

Existen cuatro características básicas en la arquitectura cliente-servidor, CronCare las cumple todas:

1. Los clientes tienen que conocer los nombres de los servidores que están disponibles.
2. Los servidores no requieren conocer la identidad de los clientes.
3. Los clientes acceden a través de llamadas a procedimientos remotos.
4. Cada servidor es responsable de las actividades de administración de datos.

La figura 10.1 representa el diseño arquitectónico de CronCare destacando sus principales componentes. Este esquema nos permite distinguir 3 grandes nodos del sistema que se evalúan en detalle en los siguientes puntos del capítulo:

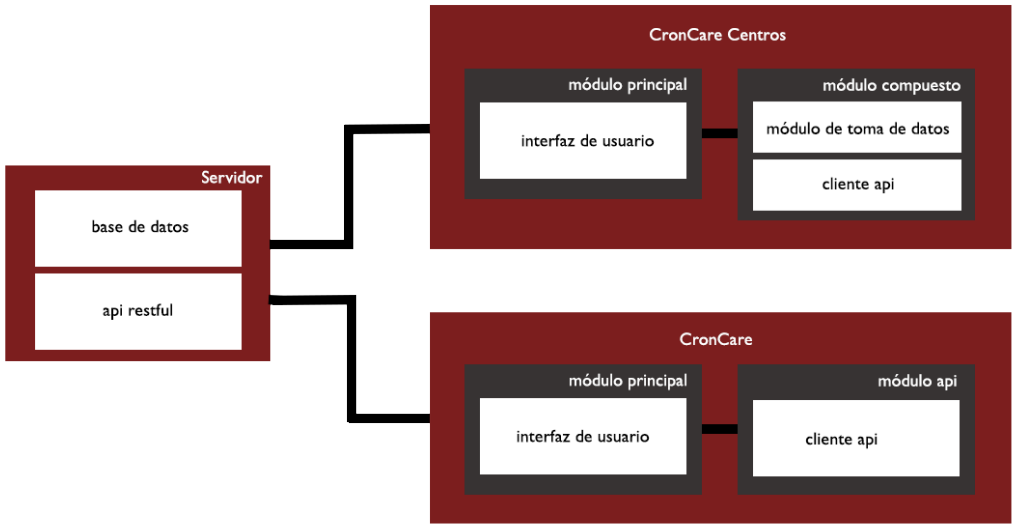


Figura 10.1: Esquema del Diseño Arquitectónico del Sistema Basado en el Modelo Cliente-Servidor

Servidor

El nodo del servidor es donde aparecen la base de datos y la API RESTFUL, la cual responderá a un conjunto de llamadas que se ciñen a la filosofía RESTFUL comentada en el punto 8.3.1.

Para el desarrollo del proyecto se ha utilizado un servidor apache local, el diseño del sistema nos permitirá cambiar el punto de entrada de la API tocando una única línea de un fichero. De esta forma se puede migrar el sistema entero a otra máquina remota y atacar a partir de eso momento a su dirección ip o dominio.

CronCare

Este es el nodo que representa la app android que utilizan como cliente los familiares interesados. Aquí se encuentra gran parte del trabajo de implementación, pues al ser el sistema que manejaran los usuarios externos debe estar cuidado al detalle a niveles de interfaz y rendimiento.

Es importante pues, que la interfaz gráfica sea independiente de las llamadas que realiza el cliente API al servidor. Las llamadas serán ejecutadas de forma asíncrona y se utilizarán métodos de comunicación entre clases para informar a la interfaz gráfica de la existencia de nuevos datos para que proceda a refrescar la vista.

CronCare Centros

Por último evaluamos el nodo que representa a la app de toma de datos que utilizarán los cuidadores y empleados del centro o residencia. Aunque a primera vista su interfaz pueda parecer demasiado simplista, su único objetivo ha sido ser clara y no incitar a errores. Por ello cuenta con grandes botones e iconos y la mayor parte visible de la pantalla está dedicada al listado de residentes con el que se trabaja continuamente.

Respecto al módulo de toma de datos cabe destacar su capacidad de filtrar la lista de residentes de forma 'inteligente', de manera que, por ejemplo, nunca se lista a residentes que no se encuentran en el centro para un registro de tipo 'comedor'. Todo esto se explica en puntos ulteriores de este mismo capítulo.

10.2. Diseño del Servidor

10.2.1. Diseño de la API RESTFUL

La API tiene un punto de entrada único para todas las llamadas. Posteriormente es dividida en distintos componentes que permiten enrutar un script de destino. El siguiente diagrama muestra el funcionamiento de una API RESTFUL:

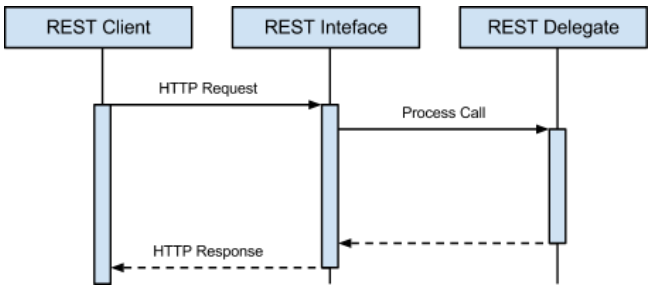


Figura 10.2: Diagrama que Muestra el Procedimiento de una Llamada a la API

Por simplicidad de mantenimiento, cada uno de los scripts php delegados de ejecutar las diferentes llamadas serán implementados de forma individual. Más adelante detallaremos como funciona la interfaz que enruta cada url con el recurso final.

10.2.2. Diseño de la Base de Datos

La base de datos es un sistema relacional MySQL que consta de múltiples tablas para almacenar todos los registros del sistema. Teniendo en cuenta que el diseño sigue la filosofía RESTFUL la base de datos sólo será accedida por el propio sistema cuando se encuentre ejecutando alguna instrucción derivada de una llamada a la API por parte de uno de los dos clientes implementados en forma de app. Al manejar un sistema altamente relacional la base de datos tiene configurados una serie de 'triggers' para ayudar a la consistencia de los datos. De esta forma tras eliminar a un usuario sus registros en la tabla de sesiones se eliminarán en cascada. En la figura 10.3 se muestra el esquema Entidad-Relación del sistema.

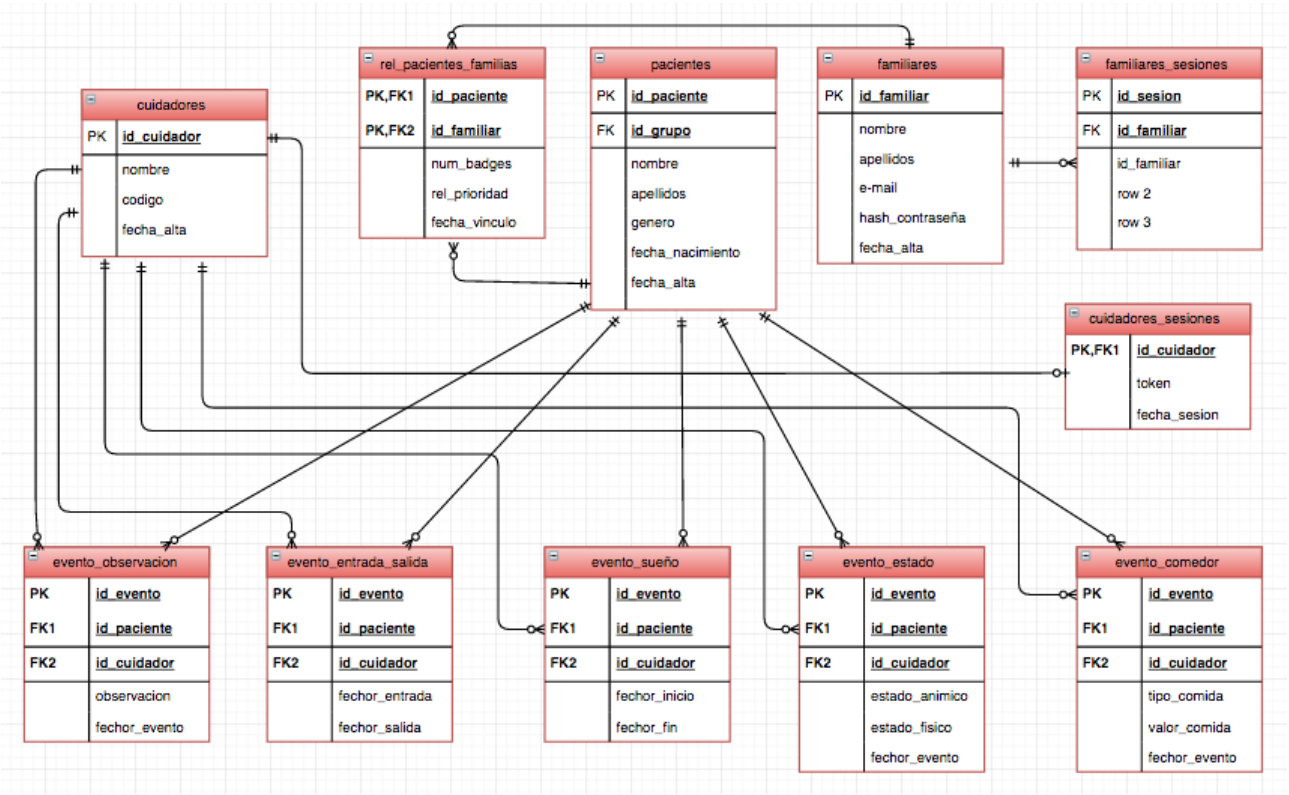


Figura 10.3: Diagrama Entidad-Relación de la Base de Datos

10.3. Diseño de CronCare

10.3.1. Interfaz de Usuario

Al tratarse de un desarrollo basado en prototipos, el propio diseño de la interfaz se define en el mismo entorno de desarrollo que la app. La modularidad del desarrollo permite programar los elementos de la interfaz visual mostrando datos estáticos para, una vez desarrollado el cliente API, conectar esas vistas a los datos que se obtienen mediante llamadas a la API. A continuación se definen una serie de requisitos mínimos a la hora de diseñar la interfaz:

- Introducción visual al sistema
  - Consiste en diseñar una serie de pantallas que expliquen para qué sirve la app antes de que el usuario realiza el registro. Este apartado sólo será visible mientras un usuario no se encuentre dentro de una sesión del sistema.
- Diseño atractivo y estándar
  - La app debe tener una apariencia comercial, con la calidad suficiente para ser distribuida al público general. Esto implica seguir los principales patrones de diseño y navegación entre vistas para no dar un experiencia de uso extraña.

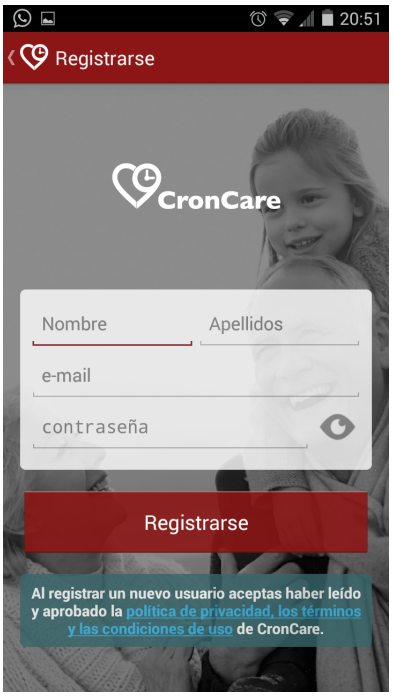
En los siguientes puntos se muestran las principales vistas de la interfaz de usuario de CronCare:

Vistas Para Usuarios que No Han Iniciado Sesión

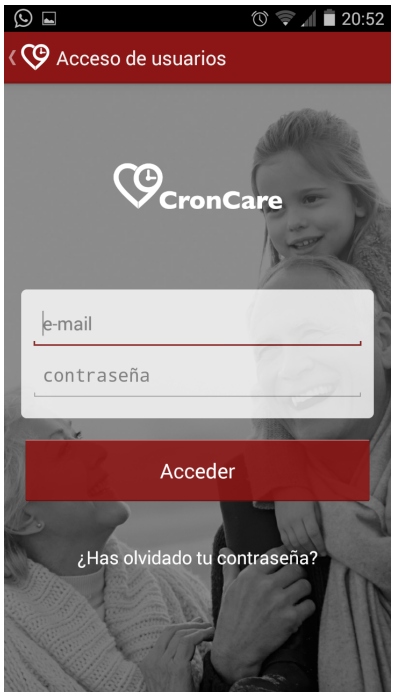
(a) Vista Inicial de CronCare

(b) Vista Inicial CronCare

Figura 10.4: Vistas para Usuarios sin Sesión



(a) Vista de Registro de Usuario



(b) Vista de Acceso de Usuarios

Figura 10.5: Vistas para Usuarios sin Sesión

En la figura 10.4 se ven las vistas disponibles en la app cuando el usuario no ha iniciado sesión. En la pantalla de registro de usuario, figura 10.5(a), se ha habilitado un botón que permite visualizar la contraseña parcial escrita en cualquier momento.

Principales Vistas de la App Para un Usuario que ha Accedido al Sistema

(a) Vista menú lateral (b) Vista registro diario de un residente

Figura 10.6: Vistas de la app para un usuario que ha accedido al sistema

- Menú lateral

  Es el menú de navegación principal de la app. Desde él se puede seleccionar el residente activo así como visitar diferentes secciones de la app como los datos del centro o el menú semanal. Para hacerlo aparecer basta con deslizar el dedo desde el lateral izquierdo de la pantalla, al ser un menú flotante está superpuesto al resto de capas visuales.

- Vista principal de un residente

  Esta es la vista más importante de la app y la más cuidada. La parte superior corresponde con una imagen que el usuario puede configurar en cualquier momento, así se personaliza la experiencia. Debajo de esta imagen encontramos la barra de navegación temporal, en el centro se encuentra la fecha de los datos que se ven debajo, las flechas laterales nos permiten avanzar o retroceder cronológicamente para encontrar otros días con actividad.

10.3.2. Cliente API

En el caso de CronCare el cliente API se encarga de realizar las llamadas al servidor de forma asíncrona y entregar los datos al controlador de la interfaz para mostrarlos en pantalla. A continuación se muestran las principales llamadas que se realizan desde el cliente API de la app CronCare:

Método	URL	Payload	Resultado
POST	/familiar/registro	nombre, apellidos, e-mail, contraseña	-
POST	/familiar/sesion	e-mail, contraseña	token único de acceso
GET	/familiar/residentes	-	[paciente1, paciente2, ...]
GET	/familiar/residentes/<id_residente>	-	Último registro disponible
GET	/familiar/residentes/<id_residente>/<fecha>	-	Registro del día <fecha>
GET	/familiar/residentes/<id_residente>/<fecha>/<prev | next>	-	Primer registro disponible antes o después de <fecha>
GET	/familiar/stats/<id_residente>/<estado | sueno | comedor>	-	Gráfica de evolución

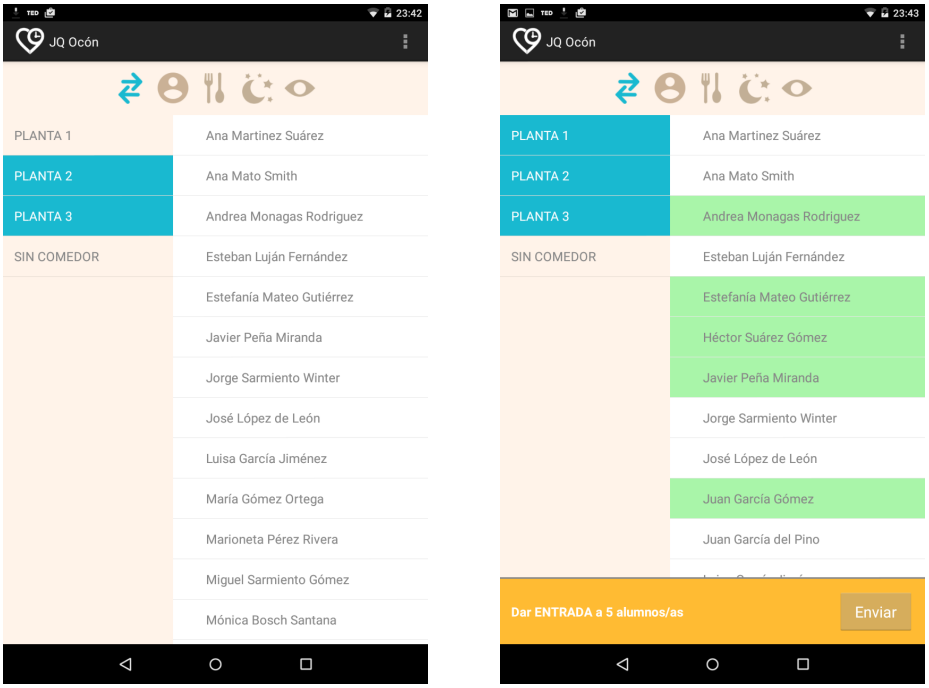
Tabla 10.1: Llamadas a la API desde CronCare

En la tabla anterior se ve la estructura con la que se forman las urls que conforman el conjunto de llamadas disponibles en la API. Cada llamada a la API lleva en su cabecera un token de seguridad generado tras la llamada a '/sesion', el resto de llamadas excepto '/registro' fallan ante la ausencia de este token o la presencia de uno inválido. A través de este token el sistema es capaz de reconocer y validar al usuario que ejecuta la llamada.

10.4. Diseño de CronCare Centros

10.4.1. Interfaz de Usuario

Esta es una app destinada a tabletas, en su diseño se ha valorado más la optimización en cuanto a la facilidad de uso que el aspecto puramente estético. Los cuidadores de una residencia necesitan invertir el menor tiempo posible al tomar un dato. Por ello la interfaz de CronCare Centros tiene una vista principal que consta de tres elementos que pasaremos a analizar a continuación.



(a) Vista Principal de la App de Toma de Datos

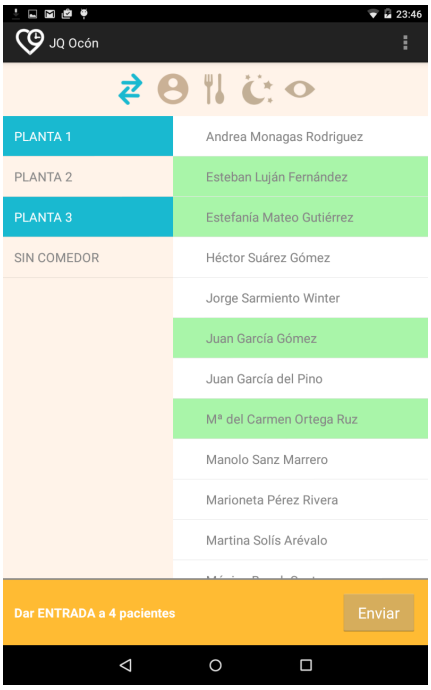
(b) Selección de Residentes para la Toma de Datos

Figura 10.7: Vista Principal de la App CronCare Centros

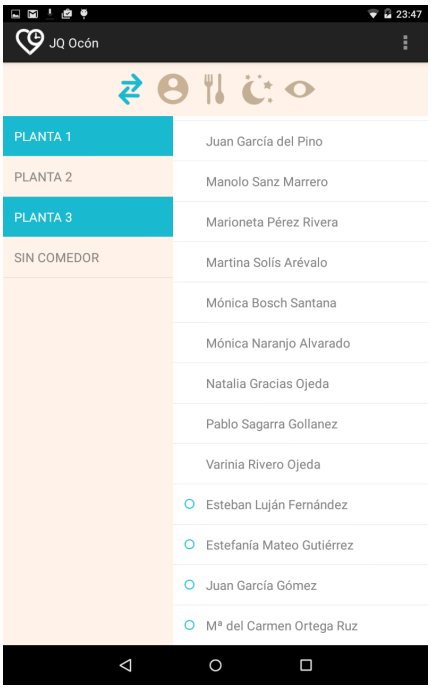
En la figura 10.7(a) se puede contemplar la vista por defecto de la app de cuidadores. La barra horizontal de iconos se corresponde al tipo de evento que se desea registrar, en el caso de la sub-figura a el cuidador se dispone a tomar registros en entrada/salida de residentes.

El icono activo siempre aparece de color azul destacando de los demás. La columna de la izquierda muestra todos los grupos de residentes existentes y el listado de la derecha muestra los residentes de los grupos seleccionados ordenados por orden alfabético.

En la figura 10.7(b) se ha comenzado a seleccionar residentes para marcar su entrada. Existen casos en los que esta lista de residentes se subdivide en dos listas a su vez, lo trataremos en los próximos puntos.



(a) Ejemplo de Registrar Entrada de Residentes a)



(b) Ejemplo de Registrar Entrada de Residentes b)

Figura 10.8: Ejemplo de Registrar Entrada de Residentes

En la figura 10.8 podemos apreciar el antes y el después de dar entrada a un grupo de residentes en el centro. En la figura 10.8(b) se aprecia que la lista de residentes para entrada/salida está divida en dos listas, el grupo inferior de residentes, que también se encuentra ordenado alfabéticamente, se encuentra presente en el centro y el circulo azul al comienzo de la fila facilita al usuario reconocer este dato. Esto sólo ocurre para dos tipos de eventos: entrada/salida y sueño.

En el caso de el listado de residentes para eventos de tipo sueño la lista diferencia mediante el círculo azul a los residentes que están dormidos de los que están despiertos.

A continuación se listan los filtros que se aplican automáticamente para cada tipo de evento en el listado de residentes, para todos los casos el listado de residentes base antes del filtro será el total de residentes pertenecientes a los grupos seleccionados en la columna izquierda.

- Entrada/Salida:
  - La lista siempre muestra a todos los residentes pero diferencia mediante marca a los presentes de los ausentes.
- Estado:
  - La lista sólo muestra a los residentes que estén presentes en el centro en este momento.
- Comedor:

- La lista sólo muestra a los residentes presentes en el centro.

- Sueño
  - La lista siempre muestra a todos los residentes pero diferencia mediante marca a los que están durmiendo de los que no lo están.

- Observación
  - La lista muestra siempre a todos los residentes sin aplicar ningún filtro.

Salta a la vista que existe un posible conflicto a la hora de registrar un evento de, por ejemplo, entrada/salida para más de un residente. Si estamos creando una selección de residentes ausentes para registrar un valor de entrada esa lista se irá llenando uno a uno hasta que pulsemos un residente que se encuentra presente (con un círculo azul en su celda), en este caso se comenzará una selección nueva para el valor de salida. Este ejemplo queda más claro tras visualizar el diagrama del siguiente apartado.

10.4.2. Módulo de Toma de Datos

En el siguiente diagrama de actividad se observa el procedimiento que se realiza al seleccionar residentes en lote antes de registrar un valor para un evento:

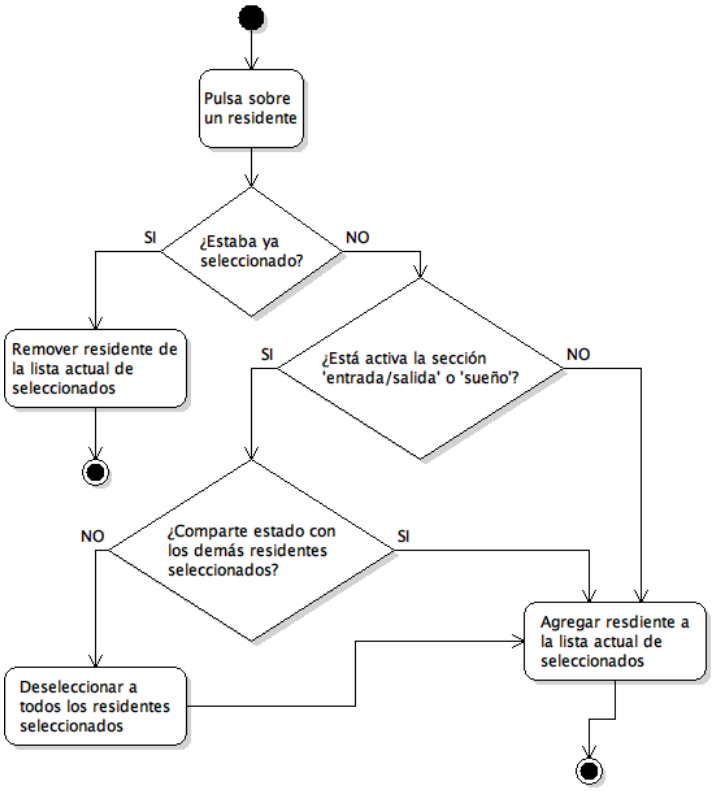


Figura 10.9: Diagrama de Actividad para Seleccionar Residentes

Método	URL	Payload	Resultado
POST	/cuidador/<entrada | salida>	[id_paciente1, id_paciente2, ...]	-
POST	/cuidador/<inicio_sueno | fin_sueno>	[id_paciente1, id_paciente2, ...]	-
POST	/cuidador/estado	[id_paciente1, id_paciente2, ...], estado_anímico, estado_físico	-
POST	/cuidador/observacion	[id_paciente1, id_paciente2, ...], observación	-

Tabla 10.2: Llamadas a la API desde CronCare Centros

10.4.3. Cliente API

El cliente API se encarga de realizar las llamadas al servidor de forma asíncrona y entregar los datos al controlador de la interfaz para mostrarlos en pantalla. A continuación se muestran las principales llamadas que se realizan desde el cliente API de la app CronCare Centros:

Cabe destacar que todas las llamadas que conllevan la escritura de registros para algún residente implican el envío de notificaciones push a las apps de los familiares vinculados.

Capítulo 11

Implementación

En esta sección se describen cada uno de los módulos que componen el sistema al más bajo nivel. En primer lugar se realizará un análisis sobre el servidor y sus scripts php para satisfacer las llamadas de la API. Dentro del estudio del servidor se explicará la configuración del servidor apache para funcionar como una API RESTFUL. Los siguientes puntos se enfocan en el desarrollo de las dos apps Android.

Aunque para poder explicar ciertos aspectos de un desarrollo software es imprescindible mostrar el código tal cual será compilado, pero por cuestiones de comodidad lectora se tratará en lo posible de explicar los mecanismos mediante esquemas, o incluso alguna captura parcial del código real, ya que las líneas de código fuente más significativas se pueden encontrar en el apéndice del capítulo A de este documento.

11.1. Librerías Externas

11.1.1. PHPass[18]

Requisito básico de seguridad es que ninguna aplicación almacena las contraseñas en texto claro. En su lugar, se hace uso de funciones criptográficas de resumen (hash), así las contraseñas se transforman mediante una función de este tipo y se almacenan los valores resumidos. Cuando un usuario quiere acceder con un nombre de usuario y una contraseña establecida previamente, la aplicación recoge la contraseña introducida, obtiene el resumen de esta contraseña y lo compara con el resumen almacenado en la base de datos para el usuario. Si los estos dos valores coinciden, la autenticación es correcta, de lo contrario se produce un error que no debe dar nunca más información de la necesaria.

La librería de código abierto PHPass permite desarrollar el módulo de cifrado y comparación de resúmenes e incluso añadir opciones de seguridad extra como 'la sal criptográfica'[1] a las contraseñas, son muchísimo más sencillas.

[1]La función principal de la sal es prevenir ataques de diccionario contra listas de resúmenes de contraseñas que se encuentran en tablas de arco-iris (rainbow). Una nueva sal se genera aleatoriamente para cada contraseña.

11.1.2. Mandrill[15]

Mandrill es un servicio de correo electrónico que permite el envío de e-mails transaccionales automatizados y personalizados. Este servicio permite enviar correos electrónicos a través de su potente API o a través del protocolo SMTP. También permite trackear a los usuarios que abren los correos o que hacen click en alguno de los enlaces. En croncare se ha preparado la infraestructura de envío de mensajes a través de Mandrill.

11.1.3. Highcharts[9]

Highcharts es una biblioteca para representar gráficas de datos escrita totalmente en JavaScript y no requiere plugins del lado del cliente como Flash o Java. Además, no es necesario instalar nada en el servidor, puede funcionar sin PHP ni ASP.NET. para usarla sólo se necesita incluir dos ficheros JavaScript. Se hace uso de Highcharts en la app de familiares a la hora de mostrar las gráficas de evolución de los residentes.

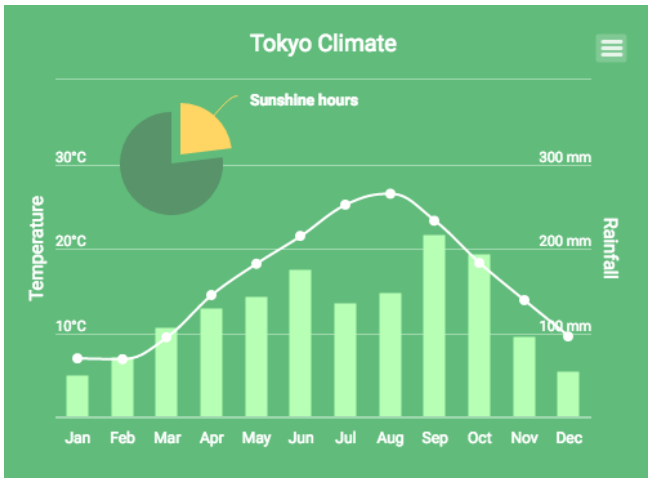


Figura 11.1: Ejemplo de Gráfica Realizada con la Librería Highcharts

11.2. Implementación de la API

11.2.1. Configuración de Apache en el Servidor

El servidor apache está escuchando en el puerto 80 a las peticiones HTTP (o en el 443 en caso de utilizar el protocolo cifrado), para que el servidor no responda como un servidor web cualquiera, se necesita crear un punto de entrada único que enrute todos los scripts de la API y devuelva cadenas de tipo JSON. El objetivo es definir un fichero, en esto caso index.php, que recoja cualquier petición al servidor que no sea respondida por un fichero existente, independientemente de la url solicitada por el cliente. Existen varias formas de lograr esto en apache, una de ellas sería tocar los ficheros de configuración y realizar cambios permanentes sobre el directorio de trabajo, en el caso que nos concierne se decidido utilizar un fichero '.htaccess' que defina las reglas que necesitamos.

Los ficheros .htaccess se pueden situar en cualquier directorio del árbol de ficheros y afectan a todos los ficheros y directorios a su mismo nivel. También se puede configurar un comportamiento recursivo en cascada para afectar al resto de ficheros. Son muy utilizados para reescribir direcciones web y que luzcan más elegantes escondiendo códigos de identificadores o nombres de ficheros que pocon importan al usuario. En nuestro caso lo utilizaremos en la raíz del servidor para que todas las urls que no apunten a un fichero determinado terminen invocando al fichero index.php.

```
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule ^(.*)$ %{ENV:BASE}index.php [QSA,L]
```

Figura 11.2: Fichero .htaccess Utilizado

RewriteCond básicamente significa «ejecutar la siguiente RewriteRule sólo si esto es cierto». El «!-f» es la condición de que la solicitud no sea para un fichero existente. El RewriteRule básicamente significa que si la petición cumple la expresión regular ^(. *)$ que coincide con cualquier url excepto la raíz del servidor, se debe reescribir como «index.php».

El flag QSA indica que añada como parámetros de la llamada el resto de la url original, quedando así «index.php?{parámetros_originales}». El flag L indica que si la regla se cumple no debe procesarse ninguna regla de las definidas más abajo en el fichero.

11.2.2. El Fichero index.php

Tras haber explicado como lograr que este index.php se encargue de procesar las llamadas a la API independientemente de su url, vamos a analizar como logra este fichero cargar el script correspondiente en cada ocasión.

Cuando el fichero index.php es cargado comienza a analizar la url que le ha invocado. Esto se logra dividiendo la cadena de texto que compone la url en función de los caracteres '/' y '?'[2]. El siguiente segmento de código realiza esa tarea:

```php
$headers = apache_request_headers();
$method = strtolower($_SERVER['REQUEST_METHOD']);

$exp = explode('?', $_SERVER['REQUEST_URI']);

if(count($exp)>1){
    $api_request = explode('/', $exp[0]);
} else{
    $api_request = explode('/', $_SERVER['REQUEST_URI']);
}

$api_request_size=count($api_request);

$client_db=$api_request[0];
$api_user=$api_request[1];
$api_call=$api_request[2];
```

Figura 11.3: Extracto del Fichero index.php

[2]En una url el carácter '?' indica el fin de la dirección de internet y el comienzo de la variables de tipo GET

Por ejemplo, para una llamada de tipo GET a «/sanlorenzo/familiar/residentes» tendríamos $client_db = sanlorenzo, $API_user = familiar y $API_call = 'residentes'. Esta llamada obtendría la lista de residentes vinculados al centro cuya base de datos se llama 'sanlorenzo' al usuario de la app de familiares que lo solicita. Si esa llamada fuera «/sanlorenzo/cuidadores/residentes» el sistema obtendría el valor 'cuidadores' para la variable $API_user y la respuesta no tendría nada que ver con la anterior, pues esa llamada devuelve todos los residentes del centro San Lorenzo a la app de cuidadores que la ejecuta.

Al más bajo nivel, diremos que $client_db define la base de datos sobre la que actuar, $API_user corresponde con el nombre de un directorio dentro de la API donde se encuentran todos los scripts disponibles para cada tipo de usuario y $API_call nos va a definir que fichero final cargar. La figura 11.4, muestra la parte final del fichero index.php, la cual logra enrutar el script adecuado para cada llamada:

```
if (file_exists("$api_user/{$api_call}_{$method}.php")) {

    connectToDB($client_db);

    if ($method=='post') {
        foreach ($_POST as $key => $input_arr) {
            $input_arr=trim($input_arr);

            if($input_arr=="" || strlen($input_arr)<1)
                unset($_POST[$key]);
            else
                $_POST[$key] = mysqli_real_escape_string($MySQLconnection,$input_arr);
        }
     }

    require_once("$api_user/{$api_call}_{$method}.php");
}

send_error();
exit;
```

Figura 11.4: Final del Fichero index.php

El bucle foreach se encarga de recorrer el vector de parámetros para las llamadas de tipo POST y filtrar su contenido por razones de seguridad. La línea clave de este fragmento es la instrucción 'require_once' donde estamos cargando un fichero directamente por su ruta en el sistema de ficheros.

11.2.3. Módulos PHP: Admin, Familiares y Cuidadores

En el punto anterior se muestra como el fichero index.php es capaz de cargar el script adecuado en función de la llamada que recibe. Cada una de las llamadas tiene un script independiente asociado situado en un directorio que coincide con el usuario que la ejecuta. Esto permite mejorar la trazabilidad, el mantenimiento y la estructuración modular del sistema.

En la figura 11.5 se puede visualizar la lista de ficheros y directorios que componen la API. Se ha desplegado el directorio «familiares» para mostrar como cada llamada tiene asociado un fichero php con el script a ejecutar.

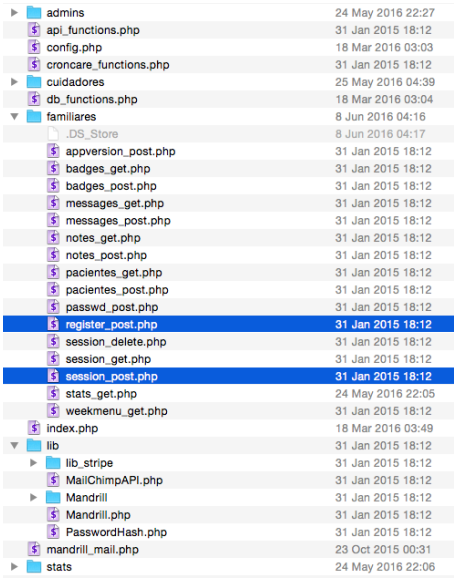
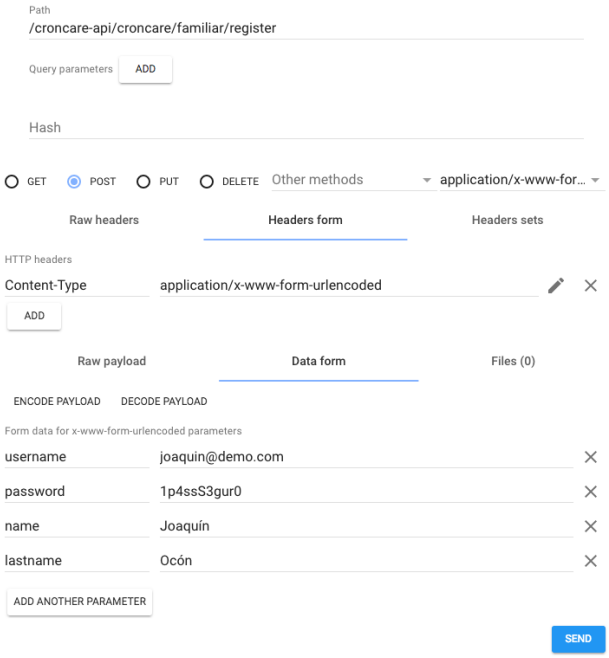


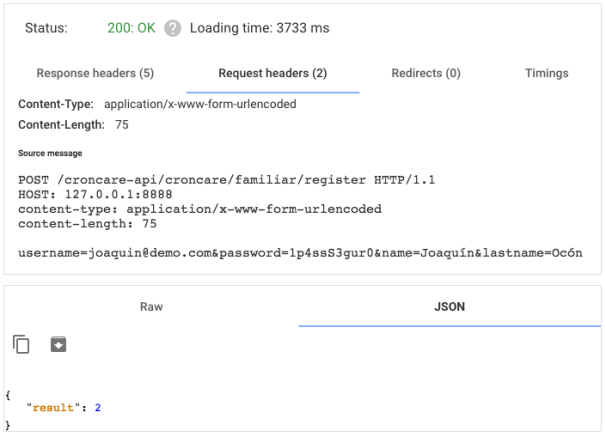
Figura 11.5: Vista de los Ficheros de la API

En la figura 11.5 se han destacado los fichero que se utilizarán a continuación al ejecutar las llamadas de registro de usuario e inicio de sesión con la herramienta ARC que nos ayuda a testear la API.

Las siguientes imágenes muestran un registro de usuario y un acceso al sistema realizando llamadas a la API al módulo de familiares:



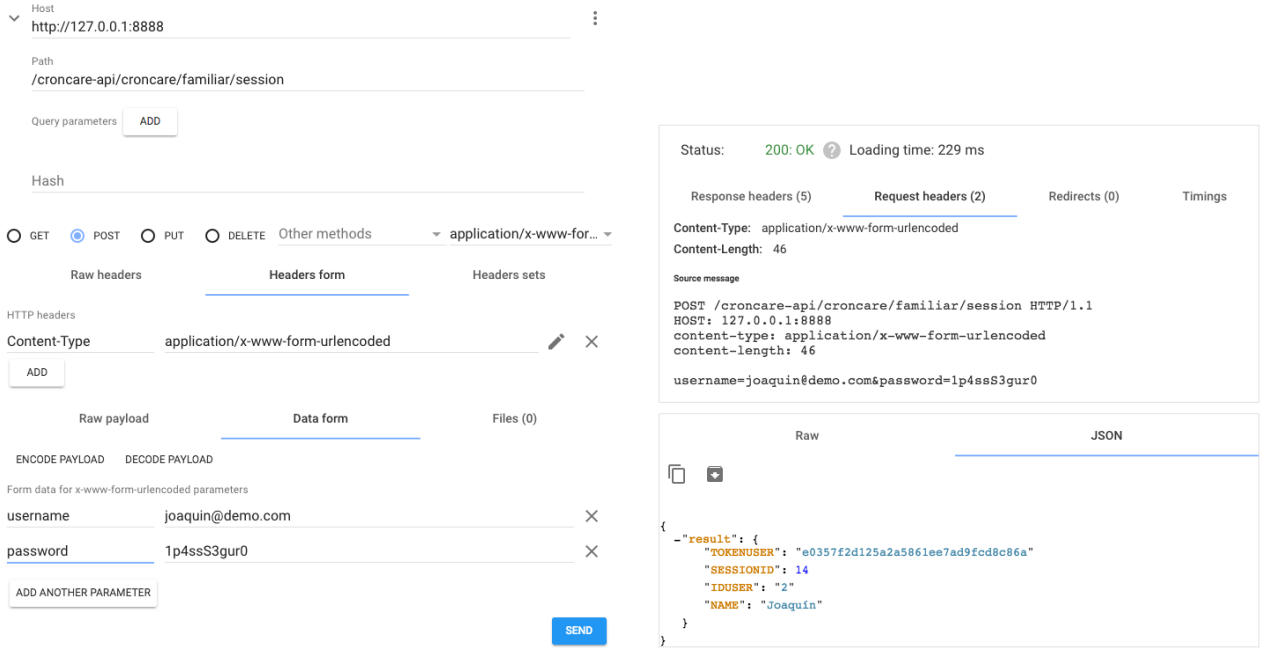
(a) Llamada a la API para Registrar un Usuario Familiar de Residente



(b) Respuesta de la API al Registro de Usuario

Figura 11.6: Registro de un Familiar de Residente Mediante la API

La figura 11.6 muestra el proceso de registrar un usuario. En la figura 11.6(a) se definen en el software ARC los datos que en un futuro serán enviados desde la app de familiares. A la derecha se puede ver que el servidor responde con el ID del usuario que ha creado, en este caso '2'.



(a) Llamada a la API para Acceder al Sistema como un Familiar

(b) Respuesta de la API al Registro de Usuario

Figura 11.7: Acceso de un Familiar de Residente mediante la API

Tras realizar la llamada que muestra la figura 11.7, el usuario recibe un token que utilizará en las llamadas posteriores al sistema. Se puede ver ese token en la sub-figura b. En el apéndice de este documento se podrán encontrar más trazas de diferentes llamadas a la API como estas.

11.2.4. Ficheros PHP y Librerías Auxiliares

La imagen 11.2 nos da una buena vista general de la estructura que tiene la API a nivel de ficheros. PHP es muy versátil a la hora de extraer código en ficheros externos o como librerías para luego ser importados mediante instrucciones de tipo «require(ruta_fichero)» o «require_once(ruta_fichero)[3]». Hasta ahora hemos visto como como la API carga cada uno de los ficheros contenidos en los directorios 'cuidadores', 'admins' o 'familiares', que contienen

[3]Esta instrucción asegura que posibles inclusiones redundantes de un mismo fichero no lleguen a ser cargadas en memoria

los scripts a ejecutar para cada llamada específica; A continuación se detallan la función de los otros ficheros y directorios que también se muestran en la figura 11.2.

- api_functions.php
  - En este fichero se encuentran algunas funciones auxiliares que se utilizan a la hora de validar las llamadas en cuanto a composición o número y tipo de variables. La función más usada de este fichero es «validateVariable($value,$type)», podemos ver su implementación a continuación:

```php
function validateVariable($value,$type){

    switch($type){
        case 'email':
            if(!filter_var($value, FILTER_VALIDATE_EMAIL))
                send_error(400,'email not valid');
            break;

        case 'password':
            if ( strlen(utf8_decode($value)) < 6 ){
                send_error(400,'password too short');
            }
            if ( strlen(utf8_decode($value)) > 50 ){
                send_error(400,'password too long');
            }
            if(!preg_match('/^[A-Za-z0-9!@#$%.?_]{6,50}$/', $value)){
                send_error(400, 'password invalid chars');
            }
            break;

        case 'numeric_id':
            if(!is_numeric((int)$value))
                send_error(409,'invalid number');
            break;

        case 'personName':
            if ( strlen(utf8_decode($value)) > 50 )
                send_error(400,'name too long');
            if ( strlen(utf8_decode($value)) < 2 )
                send_error(400,'name too short');
            break;

        case 'string_50':
            if ( strlen(utf8_decode($value)) > 50 )
                send_error(409,'string too long, 50 chars max.');
            break;

        case 'gender':
            if ( $value!="male" && $value!="female" )
                send_error(409,'invalid gender');
            break;

        default:
            break;
    }
    return;
}
```

Figura 11.8: Función validateVariable() del Fichero api_functions.php

validateVariable hace uso tanto de comparadores simples como longitud de cadenas como expresiones regulares para comprobar la validez de los datos. Vemos como en el caso de 'email' se hace uso de una expresión regular definida en el núcleo de php que es 'FILTER_VALIDATE_EMAIL'. Sin embargo, para poder establecer un criterio mínimo de fortaleza de las contraseñas que los usuarios introducen, se hace uso de una expresión regular definida específicamente para este caso:

`'/^[A-Za-z0-9!@#$%.?_]{6,50}$/'`

Esta expresión regular aceptará contraseñas con longitud entre 6 y 50 y que esté formada únicamente por caracteres alfanuméricos o alguno de los siguientes caracteres

especiales: '!', '@', '#', '$', '%', '.', '?' y '_'.

Se puede observar que siempre que alguna condición falla la API invoca a la función «send_error(error_code, error_message)». Existen dos funciones en las que puede terminar una llamada a la API, una de ellas es «send_error(error_code, error_message)» y la otra «send_response(response_data, response_code)». A continuación se muestran ambas funciones:

```php
function send_error($code=400,$detail=null){

    closeDBConnection();

    switch($code){
        case 400:
            $error_msg='Bad Request';
            break;
        case 401:
            header("WWW-Authenticate: OAuth realm='users'");
            $error_msg='Unauthorized';
            break;
        case 404:
            $error_msg='Not Found';
            break;
        case 409:
            $error_msg='Conflict';
            break;
        case 500:
            $error_msg='Internal Server Error';
            break;
        default:
            $error_msg='';
            break;
    }
    header("HTTP/1.1 $code $error_msg");
    header('Content-type: text/json; charset:utf-8');
    echo json_encode(array('error'=>$detail));
    exit;
}

function send_response($response=null,$code=200){

    if($code!=200 && $code!=201){
        send_error(500, 'Invalid response code');
    }

    closeDBConnection();

    if($code==201){
        header('HTTP/1.1 201 Created');
    }

    if($response!=null){
        header('Content-type: text/json; charset:utf-8');
        echo json_encode(array('result'=>$response));
    }

    exit;

}
```

Figura 11.9: Funciones send_response() y send_error()

Todo mensaje de respuesta contendrá un código de resultado, en el caso de llamadas que se resuelven con éxito este código siempre es el 200, excepto en el caso de haber creado un nuevo recurso en el sistema, en cuyo caso se devolverá un código «201 Created». Esto cumple con el estándar definido para al apis RESTFUL.

Los códigos de error pueden llegar a tener altos niveles de granularidad, pero depende del desarrollador el nivel de detalle que quiera aplicar al módulo de procesado de mensajes de error. En el caso de croncare se han contemplado las posibilidades más

comunes como fallos autenticación, urls mal formadas, etc.

- config.php
  - Este fichero es una simple extracción de ciertas macros del sistema que se utilizan a lo largo de todo el procedimiento, por tanto, todas las variables aquí definidas se encuentran en un entorno global al sistema.

```php
//System vars
$api_folder = 'croncare';

//DateTime
date_default_timezone_set('Atlantic/Canary');
$DateTimeNow=date_create();
$now=date_format($DateTimeNow,"Y-m-d H:i:s");
$today=date_format($DateTimeNow,"Y-m-d");

//DB vars
$hostname = 'localhost';
$username = 'root';
$password = 'root';
```

Figura 11.10: Fichero config.php

Las variables de acceso a la base de datos que se muestran aquí son las utilizadas en un entorno de desarrollo local, deberían ser modificados cuando el sistema se migre a un servidor de producción.

- croncare_functions.php
  - Este fichero contiene aquellas funciones que a lo largo del desarrollo se iban a utilizar en un punto concreto y, al ver que eran de nuevo necesarias en otro lugar, se han extraído a este fichero. Para visualizar las funciones que en él se encuentran es posible consultar los apéndicesA de este documento. A continuación se listan las funciones del fichero:
    - pacientHasData($id_pacient)
      - ◇ Función que comprueba si un residente tiene o no registros en el sistema.
    - getPacientIdCenter($id_pacient)
      - ◇ Dada una id de un residente, devuelve la id del centro al que pertenece.
    - getPacientTimeline($id_pacient, $in_date, $when)
      - ◇ Devuelve un vector con los registros del residente correspondiente en la fecha indicada. En caso de $when tener valor 'prev' o 'next' mostrará el día disponible anterior o posterior a la fecha $in_date correspondientemente.

- db_functions.php
  - Contiene funciones auxiliares para operaciones de escritura y lectura sobre la base de datos. Las tres funciones principales de este fichero son «db_select(), db_update() y db_insert()», ya que su nombre las autodefine pasemos a ver la ventaja de su uso. Tomemos como ejemplo la función «db_select()», una de las más utilizadas en casi todas las llamadas a la API.

```php
function db_select($table,$conditions='1',$fields='*',$params='',$multiple=true)
{
    global $MySQLconnection;

    $query="SELECT $fields FROM $table WHERE $conditions $params";

    $ret = array();
    $q = mysqli_query($MySQLconnection,$query);

    if (!$q || mysqli_affected_rows($MySQLconnection)<1) return $ret;

    while ($row = mysqli_fetch_assoc($q)) {
        $ret[]=$row;
    }
    mysqli_free_result($q);

    if($multiple)
        return $ret;
    else
        return $ret[0];
}
```

Figura 11.11: Función db_select() dentro del Fichero db_functions.php

Tras definir esta función podemos leer la tabla entera de residentes ejecutando simplemente:

```
$todos_los_residentes = db_select('pacientes')
```

La variable contendrá ahora un vector con todos los residentes del centro. A nivel interno, la función realizaría la siguiente operación SQL:

```
SELECT * FROM residentes WHERE 1
```

Con los parámetros de entrada se pueden lograr sentencia mucho más complejas. Veamos un ejemplo real del código de CronCare, como la llamada que ejecuta un usuario para recuperar su última sesión activa enviando si id y un token de acceso:

```
$last_session = db_select(
"users_sessions","ID_PACIENTE='$idpaciente'
AND TOKEN_PACIENTE='$token_paciente'","ID_SESSION","LIMIT 1",false
);
```

En este caso la variable $last_session únicamente contendrá un valor: el del campo 'ID_SESSION' de la fila encontrada.

- /lib
  - Únicamente contiene las librerías ya comentadas en la sección «librerías externas11.1».
- /stats
  - En este directorio ocurre algo diferente al resto de la API: dentro existe una pequeña web escrita en PHP, HTML y JavaScript cuya única función es cargar la librería Highcharts (vista en «librerías externas11.1») y mostrar la gráfica correspondiente en función de los parámetros solicitados. Esto quiere decir que /stats será cargado como una página web dentro de la app, gracias a una clase nativa de Android llamada «WebView».

11.2.5. Envío de Notificaciones PUSH

El módulo de envío de notificaciones PUSH se ejecuta siempre que un cuidador registra un evento en el sistema mediante la API. Dentro del directorio donde se encuentran los scripts de las llamadas de los cuidadores existe un fichero llamado «push_notifications.php» y todas las llamadas tipo POST que ejecutan los cuidadores requieren a este fichero para poder llamar a la única función que contiene: deliver_push_notifications(). A continuación se muestra el segmento de código más importante de esta función.

push_notifications.php — croncare

```php
// Cabecera
$GCMauthKey = get_gcmAuthKey($client_db);

$headers = array(
    "Authorization: key=$GCMauthKey",
    'Content-Type: application/json'
);

// Petición
$ch = curl_init();
curl_setopt( $ch, CURLOPT_HTTPHEADER, $headers );
curl_setopt( $ch, CURLOPT_URL, "https://android.googleapis.com/gcm/send" );
curl_setopt( $ch, CURLOPT_SSL_VERIFYHOST, 0 );
curl_setopt( $ch, CURLOPT_SSL_VERIFYPEER, 0 );
curl_setopt( $ch, CURLOPT_RETURNTRANSFER, true );
curl_setopt($ch, CURLOPT_IPRESOLVE, CURL_IPRESOLVE_V4 );

foreach($id_pacientes as $id_paciente) {
    $namePaciente = name_paciente_from_id($id_paciente);

    $payload = array(
                    "id_paciente"  => $id_paciente,
                    "namePaciente" => $namePaciente,
                    "msg"          => $msg,
                    "id"           =>  round(microtime(true))
                );

    $registrationIdsArray = get_gcm_tokens($id_paciente);

    if($registrationIdsArray!=null){

        $data = array(
           'data' => $payload,
           'registration_ids' => $registrationIdsArray
        );

        curl_setopt( $ch, CURLOPT_POSTFIELDS, json_encode($data));
        curl_exec($ch);
    }

}

// Cerramos conexión
curl_close($ch);
if($DBconnection!=null){
    mysql_close($DBconnection);
}
exit;
```

Figura 11.12: Extracto de la Función deliver_push_notifications()

Es interesante ver como se hace uso de la librería cURL, que es nativa de PHP, para hacer una llamada desde el servidor que aloja la API a los servidores que Google destina al enrutamiento de notificaciones PUSH. Dentro de los campos del payload que es enviado a Google existe un campo «id» que en nuestro caso se calcula con un round(microtime(true)), la única condición que pone google es cada identificador de una notificación PUSH sea un número entero y único en nuestro sistema. La función microtime en PHP devuelve un float que representa el tiempo pasado, en segundos, desde la fecha conocida como «UNIX Epoch»: 01/01/1970 00:00:00 GMT.

11.3. Implementación de la App CronCare Centros

Tal y como se observa en la figura del apartado10.1 la app CronCare Centros se divide en: Módulo principal, asociado al hilo de ejecución principal que contiene el interfaz de usuario; Módulo compuesto, que se encarga de realizar conexiones asíncronas con el servidor mediante el cliente API y de realizar funciones auxiliares en el módulo de toma de datos. A continuación se describen estos elementos:

11.3.1. Interfaz de Usuario

En esta sección se analiza la interfaz haciendo énfasis en los componentes propios de Android que facilitan el desarrollo de las diferentes vistas. Se analiza como ejemplo la definición de la vista activity_main.xml

Figura 11.13: Declaración del Fichero activity_main.xml

En la figura 11.18 se observa lo intuitivo que es el desarrollo de interfaces con el entorno Android Studio, dónde se puede previsualizar en tiempo real el efecto que tienen los cambios que se realizan sobre el fichero xml de la vista. La raíz de esta vista es un LinearLayout que permite disponer de bloques visuales en la orientación deseada, en este caso android:orientation=”vertical”. En el caso de los botones de selección de tipo de evento, se han dispuesto dentro de un HorizontalScrollView, esto es debido a que se puede asegurar que el ancho de todas las pantallas del mercado sea suficiente para mostrar todos los iconos con la resolución deseable. Los ScrollView y los componentes que heredan de esta clase tienen la capacidad de crear elementos más grandes que la pantalla física y desplazarlos con el dedo para mostrar el resto del contenido.

Por último se puede ver la declaración de los dos ListViews tratados anteriormente, lv_groups y lv_pacients, que a la derecha de la imagen aparecen rellenos de elementos ’dummy’ llamados «Ítem N». Cuando la actividad llama a LoadPacients() y la llamada retorna en la función responseGET en realidad se rellenan las listas List<> que se han definido como parámetros de la clase para luego alimentar estos ListViews. Todos los ListViews en Android llevan asociados un adaptador que vincula el ListView a un set de datos, normalmente de la clase List<> o Array<>.

11.3.2. Módulo de Toma de Datos

Este es el módulo alrededor del cual se ha desarrollado toda la app y su clase principal es ActivityMain. En esta actividad se presta especial atención a los método onCreate() y onResume(), ya que es importante que la tableta pase de un estado de inactividad a estar disponible para tomar datos en el menor tiempo posible. El método onCreate() se ejecutará sólo cuando el sistema crea la instancia de la clase y no volverá a ser llamado hasta que la instancia sea destruida y deje de existir en memoria. Es el propio sistema Android y su recolector de basura el que se encarga de terminar actividades si los recursos del sistema así lo requieren. Cada vez se regresa a ActivityMain, independientemente de si se ejecuta el método onCreate() o no, el método onResume() será ejecutado.

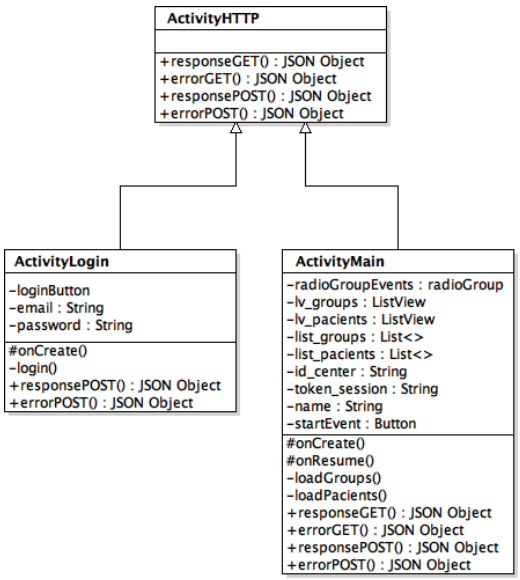


Figura 11.14: Diagrama de las clases ActivityLogin y ActivityMain

El método onCreate() es muy extenso y puede ser consultado en los apéndices del documento, por ello nos limitaremos a explicar su funcionamiento. Cuando se instancia la clase ActivityMain lo primero que hace es configurar la vista cargando su fichero activity_main.xml en el que se definen los elementos visibles en la pantalla. Seguidamente se conectan los controladores de eventos a estos elementos, se definen OnClickListener() en los listados de grupos y residentes y en los botones superiores de selección de categoría de eventos.

Básicamente se definen las acciones a realizar para cada uno de los elementos cuando se selecciona uno de sus elementos. Cuando se selecciona o deselecciona un grupo de la lista de la izquierda, se ejecuta una llamada de tipo GET a la API en la siguiente dirección:

```
/residentes/{id_group1, id_group2, ...}?st={session_token}
```

Donde el segundo parámetro es un listado con las ids de los grupos seleccionados separados por coma y la variable del token de session se pasa al final como una variable de tipo GET. Esta llamada es invocada desde la función LoadPacients() mostrada en la figura 11.15.

```
private void LoadPacients(){

    String id_groups="";
    for(int i=0; i< list_groups.size(); i++){
        if(list_groups.get(i).isSelected()){
            id_groups+= list_groups.get(i).getIdGroup()+",";
        }
    }

    if(!id_groups.isEmpty()){

        String event_type="";
        if(radioGroupEvents.getCheckedRadioButtonId()==entranceBtn.getId())
            event_type="entrance";
        else if(radioGroupEvents.getCheckedRadioButtonId()== statusBtn.getId())
            event_type="status";
        else if(radioGroupEvents.getCheckedRadioButtonId()==sleepBtn.getId())
            event_type="sleep";
        else if(radioGroupEvents.getCheckedRadioButtonId()==lunchBtn.getId())
            event_type="lunch";
        else
            event_type="obs";

        ThreadGET threadGetPacients = new ThreadGET(this,"load_pacients");
        threadGetPacients.execute("pacientes/" + id_center + "/" + event_type + "/" + id_groups + "?st=" + token_session);
    } else {

        lv_pacients.setVisibility(View.INVISIBLE);
        sendActionMenu.setVisibility(View.GONE);
    }

}
```

Figura 11.15: Función LoadPacients() dentro de ActivityMain

Es necesario enviar el tipo de evento para el que se realiza la petición (variable event_type) pues el servidor puede añadir campos necesarios para la app en caso de ser un tipo de evento determinado. Es el caso de el listado de residentes para eventos de tipo entrada/salida o registros de sueño, que como se indica en el capítulo 8.3.1 necesitan un campo que diferencie el estado de los residentes (ausentes, dormidos, etc.).

El método LoadPacients() también es invocado desde la función onResume() cada vez que la actividad recupera el foco, así se logra obtener un listado de residentes actualizado cada vez que se va a utilizar la app.

Para el envío de datos se hará uso de llamadas de tipo POST. A diferencia de los eventos de entrada/salida o sueño que se registran desde la vista principal, los eventos restantes requieren una vista propia para establecer los valores en cada situación. A continuación se muestra el ejemplo de los registros de estado anímico/físico del residente que suceden en la actividad ActivityStatus.

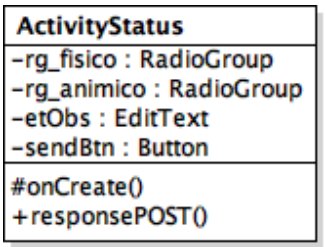
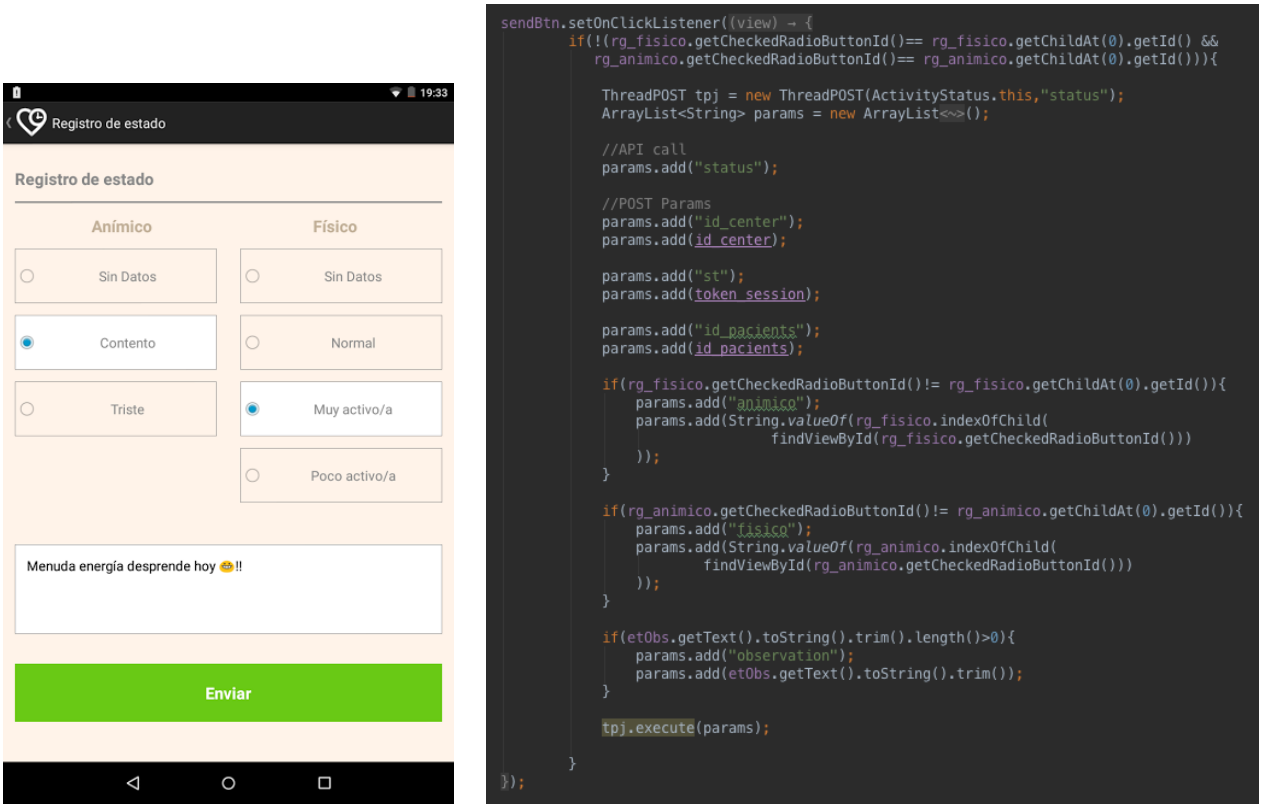


Figura 11.16: Diagrama de la Clase ActivityStatus



(a) Interfaz de la clase ActivityStatus (b) Sección de código de la clase ActivityStatus

Figura 11.17: Interfaz de la Clase ActivityStatus y Sección de código

En la figura 11.17(b) se muestra la sección de código más importante de la clase, que ocurre dentro de su método onCreate() cada vez que se instancia. El botón de envío se vincula a un escuchador que detecta el evento «click». En ese momento se recorren los elementos de selección de opciones y se crea dinámicamente la llamada a la API en función de los elementos seleccionados.

11.3.3. Cliente API

Tal y como vimos en el capítulo 8 8.1.3, se utiliza la clase Actividades para encapsular las vistas de la app con sus funcionalidades. Haciendo uso de la herencia entre clases, se ha implementado la clase *ActivityHTTP* que define métodos para recoger las respuestas de la llamadas a la API, así como sus posibles errores. Todas las actividades que vayan a utilizar las llamadas a la API serán una herencia de ActivityHTTP y tendrán que definir sus propias implementaciones de las funciones declaradas en la superclase. Tomemos como ejemplo la actividad que realiza el acceso de los usuarios.

```
public class ActivityLogin extends ActivityHTTP {
    private final String TAG = "croncare_log_ActivityLogin";

    private Button loginButton;
    private String password;
    private EditText input_pass;
    private TextView tv_version;
    private String id_center, namedb;

    private SharedPreferences preferences;
    private SharedPreferences.Editor preferencesEditor;

    @Override
    protected void onCreate(Bundle savedInstanceState) {...}

    private void login(){
        ThreadPOST tpj = new ThreadPOST(this,"login_cuidadores");
        ArrayList<String> params = new ArrayList<>();
        //API call
        params.add("session");

        //POST Params

        params.add("password");
        params.add(password);

        params.add("id_center");
        params.add(id_center);

        tpj.execute(params);
    }
```

Figura 11.18: Declaración de la Clase ActivityLogin

En la figura 11.11 se aprecia la declaración de la clase ActivityLogin, la cual hereda de ActivityHTTP. El único método propio que implementa esta clase es login(), el cual hará uso de una llamada tipo POST a la API para lograr el acceso de un usuario.

Para ejecutar la llamada hace uso de la clase ThreadPOST a la que añade por orden los parámetros de la llamada, primero la dirección de la url y a continuación los valores del payload que en este caso son la id del centro y el código pin de acceso de un cuidador. La última instrucción tpj.execute(params) se ejecutará de forma asíncrona e invocará al método responsePOST de la clase cuando obtenga respuesta desde el servidor.

```
@Override
public void responsePOST(JSONObject objresult, String action) {
    try {

        if(action.contentEquals("login_cuidadores")){

            JSONObject result = objresult.getJSONObject("result");

            String token = result.getString("TOKEN");
            String name = result.getString("NAME");

            preferences = getSharedPreferences("DataSession",MODE_PRIVATE);
            preferencesEditor = preferences.edit();
            preferencesEditor.putString("token", token);
            preferencesEditor.putString("name", name);
            preferencesEditor.commit();

            //Move to ActivityMain and clear the activities history
            startActivity(new Intent(getApplicationContext(), ActivityMain.class)
                    .setFlags(Intent.FLAG_ACTIVITY_NEW_TASK | Intent.FLAG_ACTIVITY_CLEAR_TASK));
            finish();
        }

    } catch (JSONException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}

@Override
public void errorPOST(String action, Integer errorcode, JSONObject errormsg) {
    //Login Unsuccessful
    if(action.contentEquals("login_cuidadores") && errorcode==401){
        showAlertMessage("Error", "Error de acceso. Por favor, comprueba tus datos y vuelv...", false);
    }
    else{
        super.errorPOST(action,errorcode,errormsg);
    }
}
```

Figura 11.19: Continuación de la Clase ActivityLogin

Se aprecia en la figura 11.19 como se hace uso de la cláusula @Override para declarar nuestra propia implementación del método responsePOST que se hereda de la superclase ActivityHTTP. En él se recoge un objeto JSON que contiene el nombre del cuidador que ha accedido y un token único para enviar como cabecera de todas las futuras llamadas y que están puedan ser satisfactorias. Tras guardar estos valores en la memoria local de la aplicación, se invoca una actividad que en este caso es la actividad principal de la app: MainActivity.

En este punto, se explicará el funcionamiento de las clases ThreadPOST y ThreadGET, ya que todas las actividades que hacen uso de llamadas a la API en la app siguen el mismo patrón.

ThreadGET y ThreadPOST heredan de la clase AsyncTask e implementan sus métodos asíncronos:

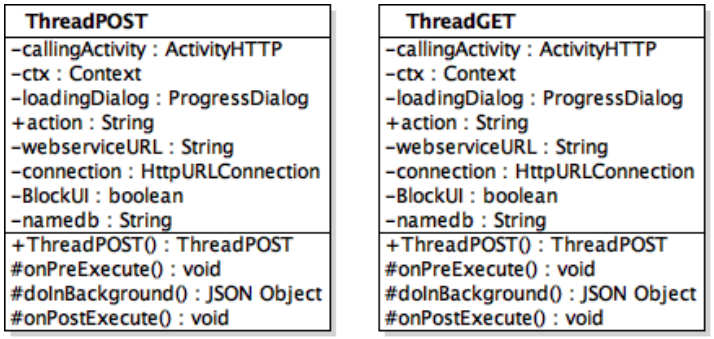


Figura 11.20: Diagrama de las Clases ThreadPOST y ThreadGET

Vemos que uno de los atributos de estas clases es la actividad vinculada a la misma, que

debe ser una ActivityHTTP. Por lo demás dispone de los elementos necesarios para mostrar un diálogo de carga de datos que informa al usuario de que se están recibiendo datos del servidor y tres métodos que nos ofrecen la capacidad de preparar la llamada, realizarla y recoger sus datos.

En ambos casos el método onPreExecute() simplemente mostrará en pantalla el diálogo de carga y onPostExecute invocará el método responseGET o responsePOST de la activityHTTP vinculada según corresponda. A continuación se muestra la implementación de ambos métodos para la clase ThreadPOST. En el case de la clase ThreadGET su método onPostExecute() terminaría llamando a la función responseGET o errorGET en lugar de responserPOST, por lo demás no difiere en nada de la siguiente:

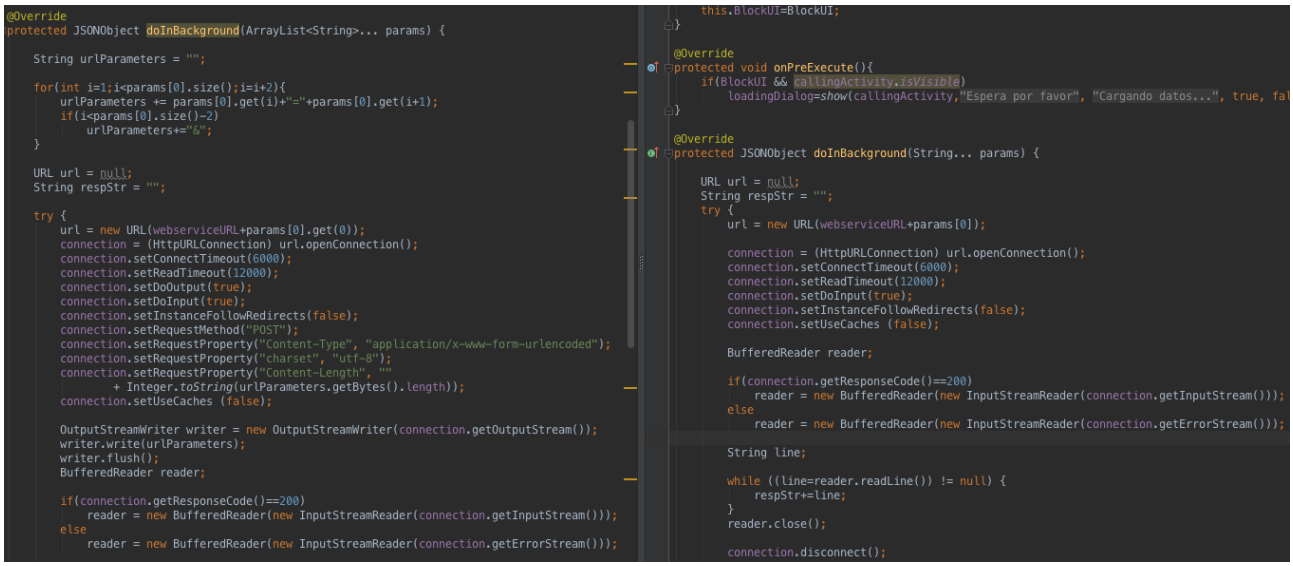
```
@Override
protected void onPreExecute(){
    if(BlockUI)
        loadingDialog=show(callingActivity,"Espera por favor", "Cargando datos...", true, false);
}
```

```
@Override
protected void onPostExecute(JSONObject result) {
    if(callingActivity!=null){
        if(callingActivity.isVisible){
            try {

                if(connection.getResponseCode()==200)
                    callingActivity.responsePOST(result, action);
                else
                    callingActivity.errorPOST(action, connection.getResponseCode(), result);

            } catch (IOException e) {
                e.printStackTrace();
                callingActivity.errorPOST(action, 0, result);
            }
        }

        if(BlockUI)
            loadingDialog.dismiss();
    }

    try {
        this.finalize();
    } catch (Throwable throwable) {
        throwable.printStackTrace();
    }
}
```

Figura 11.21: Métodos onPreExecute() y onPostExecute() de ThreadPOST

El método doInBackground básicamente se encarga de preparar el paquete de datos y enviarlo al servidor con los parámetros adecuados para que la API lo interprete como una llamada válida.

La gran diferencia entre los métodos POST y GET radica en las cabeceras de los paquetes y en que el método POST tiene un payload asociado con los pares de valores a enviar. Aunque los métodos completos se encuentran en el apéndice del documento, en esta imagen tomada con una división vertical de la pantalla del entorno podemos apreciar el comienzo de la implementación para ambas clases, a la izquierda ThreadPOST y a la derecha ThreadGET:



```
@Override
protected JSONObject doInBackground(ArrayList<String>... params) {

    String urlParameters = "";

    for(int i=1;i<params[0].size();i=i+2){
        urlParameters += params[0].get(i)+"="+params[0].get(i+1);
        if(i<params[0].size()-2)
            urlParameters+="&";
    }

    URL url = null;
    String respStr = "";

    try {
        url = new URL(webserviceURL+params[0].get(0));
        connection = (HttpURLConnection) url.openConnection();
        connection.setConnectTimeout(6000);
        connection.setReadTimeout(12000);
        connection.setDoOutput(true);
        connection.setDoInput(true);
        connection.setInstanceFollowRedirects(false);
        connection.setRequestMethod("POST");
        connection.setRequestProperty("Content-Type", "application/x-www-form-urlencoded");
        connection.setRequestProperty("charset", "utf-8");
        connection.setRequestProperty("Content-Length", ""
                + Integer.toString(urlParameters.getBytes().length));
        connection.setUseCaches (false);

        OutputStreamWriter writer = new OutputStreamWriter(connection.getOutputStream());
        writer.write(urlParameters);
        writer.flush();
        BufferedReader reader;

        if(connection.getResponseCode()==200)
            reader = new BufferedReader(new InputStreamReader(connection.getInputStream()));
        else
            reader = new BufferedReader(new InputStreamReader(connection.getErrorStream()));
```

```
        this.BlockUI=BlockUI;
    }

    @Override
    protected void onPreExecute(){
        if(BlockUI && callingActivity.isVisible)
            loadingDialog=show(callingActivity,"Espera por favor", "Cargando datos...", true, fa
    }

    @Override
    protected JSONObject doInBackground(String... params) {

        URL url = null;
        String respStr = "";
        try {
            url = new URL(webserviceURL+params[0]);

            connection = (HttpURLConnection) url.openConnection();
            connection.setConnectTimeout(6000);
            connection.setReadTimeout(12000);
            connection.setDoInput(true);
            connection.setInstanceFollowRedirects(false);
            connection.setUseCaches (false);

            BufferedReader reader;

            if(connection.getResponseCode()==200)
                reader = new BufferedReader(new InputStreamReader(connection.getInputStream()));
            else
                reader = new BufferedReader(new InputStreamReader(connection.getErrorStream()));

            String line;

            while ((line=reader.readLine()) != null) {
                respStr+=line;
            }
            reader.close();

            connection.disconnect();
```

Figura 11.22: Diferencias en los Métodos doInBackground() de ThreadPOST y ThreadGET

En la parte izquierda de la imagen se aprecia como al comienzo del código hay un bucle for para recorrer y preparar los valores que se desean enviar en el payload del paquete POST.

La variable connection permite definir parámetros para la llamada al servidor, es así como se pueden poner las cabeceras adecuadas para cada tipo de llamada. Se observa que la llamada POST envía sus datos codificados en formato utf-8, que es el mismo en el que se encuentra la base de datos y un estándar en internet.

En ambos casos se aprecia la comprobación del código de respuesta, un código 200 indica éxito y terminará enviando el objeto JSON de respuesta a onPostExceute que, a su vez, lo enviará a la función responsePOST o responseGET correspondiente de la actividad que ejecute la llamada.

11.4. Implementación de la App CronCare

11.4.1. Interfaz de Usuario

Vista Principal

El menú deslizante lateral hace uso de una clase proporcionada por el sistema Android: Fragments. Un Fragment tiene un ciclo de vida parecido al de una actividad, pero tiene aislada su funcionalidad. Esto permite crear zonas independientes entre ellas en la UI, una actividad puede contener uno o varios Fragments.

En el caso de la actividad principal ActivityMain, tendremos dos Fragments principales: el menú lateral izquierdo y la vista que queda justo debajo, donde se muestra la información referida a la sección elegida en el menú y que por defecto será el registro diario de la última fecha disponible del primer residente del listado del menú.

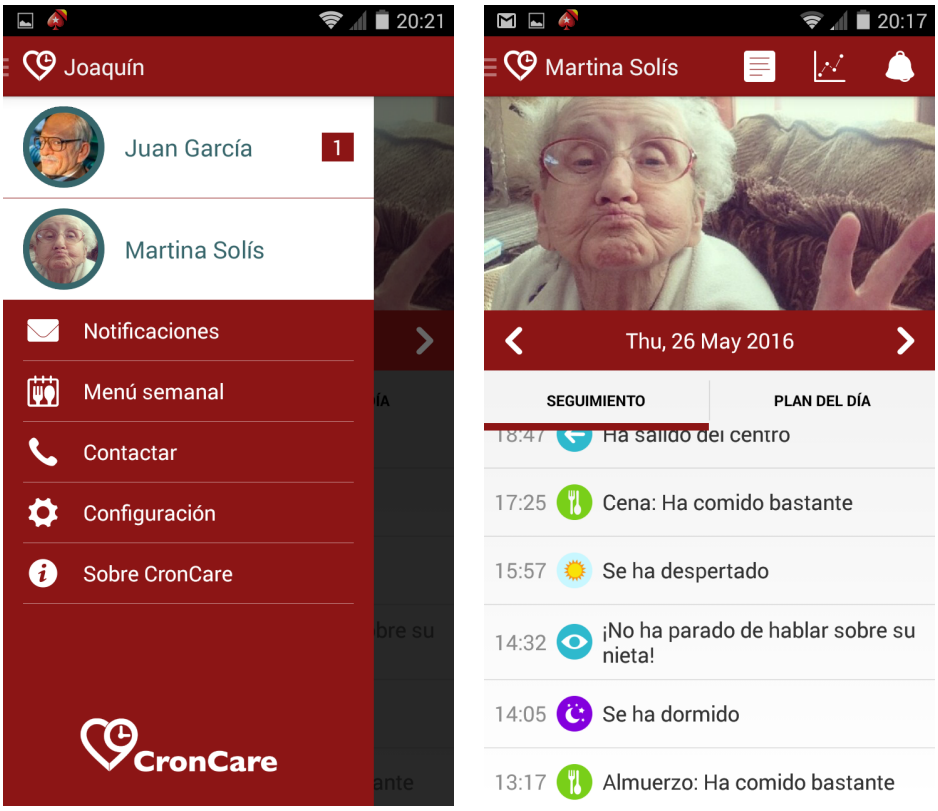


Figura 11.23: Interfaz Gráfica de ActivityMain

En la imagen de la izquierda se observa el menú desplegado cubriendo el fragment de los registros del residente que si es visible en la imagen de la derecha donde se ha ocultado el menú.

Para poder completar la lista de eventos se hace uso de un ListView cuyo adaptador se alimenta de la respuesta a la llamada /pacients/{id_residente} perteneciente al grupo descrito en el punto 10.3.2.

Pulsando en las flechas laterales se pueden llamar a la API con peticiones previas o posteriores a la fecha señalada.

Actividad de Estadísticas de Evolución: AvtivityStats

Este es un caso particular dentro de la app. Esta actividad hace uso de un elemento nativo de android que permite cargar páginas web dentro de la app: el WebView.

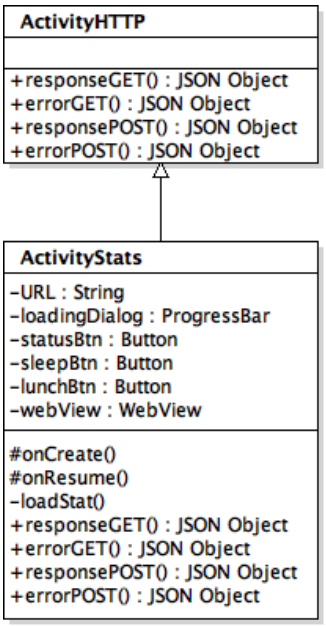


Figura 11.24: Diagrama de la Clasa ActivityStats

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_stats);
    ctx=this;

    preferences = getSharedPreferences("DataUser",MODE_PRIVATE);
    usertoken = preferences.getString("usertoken",null);
    URL="http://teamandtalent.com/croncare/croncare/users/"+"stats/"+getIntent().getStringExtra("id_pacient")+"/";

    myWebView = (WebView)findViewById(R.id.StatsWebView);

    statusBtn = (ImageButton)findViewById(R.id.wcButton);
    statusBtn.setOnClickListener((view) -> { loadStat("status"); });

    sleepBtn = (ImageButton)findViewById(R.id.sleepButton);
    sleepBtn.setOnClickListener((view) -> { loadStat("sleep"); });

    lunchBtn = (ImageButton)findViewById(R.id.lunchButton);
    lunchBtn.setOnClickListener((view) -> { loadStat("lunch"); });

    //Action bar and menu setup
    bar=getActionBar();
    bar.setDisplayHomeAsUpEnabled(true);
    bar.setHomeButtonEnabled(true);
    bar.setTitle("Evolución"+": "+getIntent().getStringExtra("id_pacient"));
    bar.setIcon(R.drawable.minilogo);
```

Figura 11.25: Fragmento de la Clase ActivityStats

11.4.2. Cliente API

Las clases ActivityHTTP, ThreadGET y ThreadPOST forman parte de un paquete que compone el módulo «cliente API», es por ello que su implementación y uso no varía en absoluto respecto a lo ya explicado en la sección anterior.

La app de familias reutiliza directamente el mismo módulo por lo que los patrones seguidos en el desarrollo de ambas apps son muy similares.

De manera análoga las actividades de esta app que requieran de llamadas a la API, serán una herencia de ActivityHTTP.

11.4.3. Clases Principales del Sistema

ActivityMain

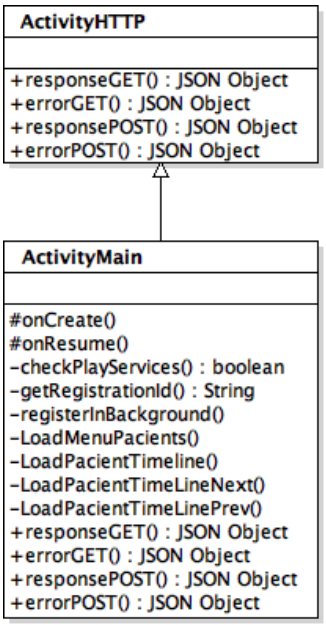


Figura 11.26: Diagrama de la Clase ActivityMain

Es una clase con muchos métodos, pero no realiza todo el trabajo en el hilo principal ya que eso sobrecargaría la UI y ralentizaría la experiencia de uso. Para que esto no ocurra, se implementan una serie de métodos asíncronos aparte de las llamadas a la API ya comentadas. A continuación se describe la función de cada uno de los métodos principales de esta clase.

- checkPlayServices()
    - Se encarga de comprobar que el sistema operativo dispone de una versión de Google Play Services lo suficientemente reciente para ser compatible con nuestra app. En caso de error mostrará al usuario un diálogo emergente con la opción de descargar o actualizar a la última versión disponible.
- getRegistrationId()
    - Realiza una comprobación para saber si la versión de la app ha variado. En caso de no ser la misma que la última vez que se ejecutó, normalmente debido a una actualización, se borrará el identificador actual y se obtendrá uno nuevo llamando a registerInBackground().
- registerInBackground()

- Hace uso de la clase AsyncTask para asegurar que no interfiere con el hilo principal de ejecución, se conecta al servidor de google y contiene un identificador único para el par (dispositivo, versión de la app). Este será el identificador que se use desde el servidor para lograr hacer el envío de notificaciones PUSH a los diferentes teléfonos de los usuarios.

- LoadMenuPacients()

  - Se encarga de utilizar un ThreadGET para solicitar un vector al servidor con la información básica sobre los residentes vinculados a la cuenta actual. En los datos de respuesta se encuentra el nombre del residente, su id y el número de eventos sin leer que existen para el usuario. Esa misma id será utilizada para cargar el timeline de cada residente al hacer click sobre su nombre.

- LoadPacientTimeline()

  - Llama a la API solicitando un vector con los registros de la última fecha disponible en el sistema para el residente. Tras recibir el JSON de respuesta se carga el Fragment que muestra el listado de eventos. Un adaptador se encarga de cargar los iconos y las ristras de texto que son elementos precargados como recursos estáticos del sistema.

- LoadMenuPacientsNext() y LoadMenuPacientsPrev()

  - Simplemente agrega el parámetro temporal a la llamada que se realiza a la API en el caso anterior. Con la siguiente imagen queda clara la diferencia entre los métodos para ofrecer la posibilidad de paginación cronológica:

```
private void LoadPacientTimeLine(String id_pacient){
    ThreadGET tgj = new ThreadGET(this,"load_pacient_tl");
    tgj.execute("pacients/"+id_pacient+"?st="+usertoken);
}

private void LoadPacientTimeLinePrev(String id_pacient, String date){
    ThreadGET tgj = new ThreadGET(this,"load_pacient_tl");
    tgj.execute("pacients/"+id_pacient+"/"+date+"/prev?st="+usertoken);
}

private void LoadPacientTimeLineNext(String id_pacient, String date){
    ThreadGET tgj = new ThreadGET(this,"load_pacient_tl");
    tgj.execute("pacients/"+id_pacient+"/"+date+"/next?st="+usertoken);
}
```

Figura 11.27: Código de los Métodos para la Navegación Cronológica

ActivityHomeNote

Esta actividad será invocada con el botón del bloc en la parte superior derecha de la pantalla. En ella un familiar puede rellenar un campo de texto de hasta 300 caracteres mientras el residente no se encuentre presente en el centro.

Este tipo de comunicación bidireccional regulada permite que el centro no se sature recibiendo mensajes a la vez que permite a las familias avisar de cosas realmente importantes. Cuando el residente vuelve a ingresar en el centro en la app de cuidadores aparece un icono indicando que existe una nota sin leer para el residente. Los cuidadores siempre pueden responder mediante el evento «observación libre».

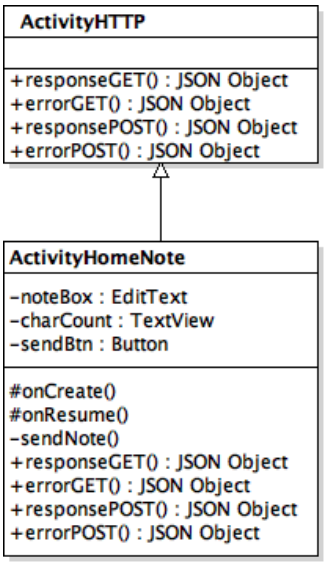


Figura 11.28: Diagrama de la Clase ActivityHomeNote

Cuando el usuario inicia esta actividad se ejecuta una llamada GET al servidor que puede devolver error en caso de que el residente se encuentre en el centro. Si no, cargará la última nota guardada en caso de haber alguna.

```
@Override
public void responseGET(JSONObject obj, String action) {
    if(action.contentEquals("getnote")){
        try {
            String note = obj.getJSONObject("result").getString("NOTE");
            noteBox.setText(note);
        } catch (JSONException e) {
            e.printStackTrace();
        }
    }
}

@Override
public void errorGET(String action, Integer errorcode, JSONObject errormsg) {
    if(action.contentEquals("getnote") && (errorcode==401 || errorcode==404)){
        switch(errorcode){
            case 404:
                break;
            case 401:
                showAlertMessage("Error","No es posible enviar notas mientras el paciente está en...",true);
                break;
        }
    } else {
        super.errorGET(action,errorcode,errormsg);
    }
}
```

Figura 11.29: Recepción de Respuesta de la API en ActivityHomeNote

Capítulo 12

Plan de Negocio

Un plan de negocio es una declaración formal de un conjunto de objetivos de una idea o iniciativa empresarial, que se constituye como una fase de proyección y evaluación.

12.1. Resumen Ejecutivo

CronCare es una plataforma que establece una nueva vía bidireccional de comunicación entre los Centros de Atención Especializada y los familiares de sus residentes a través del uso de la tecnología. Conscientes de la importancia de la implicación de la familia en la realidad de los centros, esta herramienta se ha diseñado para ser un canal seguro, ágil y eficiente.

La familia de cada residente recibe en su smartphone notificaciones en tiempo real de los eventos y las anotaciones que tienen lugar en el centro, además de poder consultar información relevante sobre actividades, menú semanal y notificaciones de la secretaría. El equipo de cuidadores dispondrá de una app para tabletas Android (también compatible con teléfonos) con la que podrá tomar los registros que alimentan al sistema y notificar a los familiares.

A la larga, esta herramienta será indispensable para ellos pues en ella estarán todos los datos que vayan registrando sobre sus residentes día a día. Así, cuando un cuidador tome el registro de comedor, sólo tendrá que seleccionar en la pantalla del sistema a los residentes para los que quiere guardar valores y enviarlos al sistema. En ese momento los familiares recibirán en sus smartphones un aviso mediante las notificaciones push.

12.2. Mercado Potencial

12.2.1. Clientes Directos

El planteamiento de cómo explotar el producto una vez desarrollado no es para nada sencillo. Pero el modelo sugerido en este caso es ofrecer sin coste alguno las app de familiares a través del principal canal de distribución de apps: Google Play Store.

El servicio en sí será prestado a los centros, que serán los clientes directos. El servicio se prestará a cambio de una cuota fija mensual que variará dependiendo del tipo de plan contratado. La localización de clientes potenciales es sencilla, ya que al pertenecer a un mercado regulado

como el sanitario, existen registros oficiales en los diferentes ámbitos: municipal, autonómico, estatal...

12.2.2. Clientes Indirectos

Cada persona que utiliza la app para consultar el estado de sus familiares ha proporcionado antes sus datos personales y e-mail. Incluso se le puede añadir automáticamente a una lista privada de correo gestionada por el equipo de CronCare. Estos usuarios son clave para el proyecto. Todo son ventajas para ellos, pues mejora la calidad y la eficiencia de la comunicación por parte del centro. Los usuarios están siempre al día de lo que ocurre allí donde se encuentra su familiar y no tienen que pagar por ello. Sin embargo, como clientes indirectos son muy valiosos, pues la base de conocimiento que se genera según crece la base de datos

12.3. Modelo de Negocio

12.3.1. Pruébalo Gratis

Aunque se entiende que a priori existen reticencias a la hora de 'regalar' el producto, son muchos los que confían en este sistema. Los centros y residencias podrán adquirir un plan gratuito y gestionar con el sistema un máximo de 20 residentes. Sólo podrán crear un grupo de residentes y asignar un cuidador al mismo.

Una vez que el centro comienza a usar el sistema de manera gratuita y los familiares de algunos residentes comienzan a recibir las notificaciones en tiempo real, es muy difícil quitarles después esta fuente de información. El movimiento lógico del centro será adquirir un plan de pago para satisfacer al resto de las familias.

Así pues el plan gratuito es en realidad un 'plan anzuelo'. Pero es de vital importancia causar una gran impresión durante el período de prueba. Las interfaces deben ser extremadamente sencillas y agradables. Aún existe resistencia al cambio tecnológico, sobre todo en personas de más de 50 años.

12.3.2. Opción Premium

Existen centros (sobre todo en zonas privilegiadas) que no se conformarán con usar el sistema CronCare. Este tipo de centros busca algo más, por su nivel de competencia, porque ellos nunca pueden ser menos que sus rivales directos, ellos buscan crear marca. Y eso es exactamente lo que este modelo les ofrece.

La opción premium es básicamente una adaptación para cambiar la piel al sistema y así generar una réplica en cuanto a funcionalidad pero con diferente apariencia. En este caso estará diseñado con los colores corporativos del centro en cuestión y se reemplazarán los logotipos de CronCare por los del centro. El centro dispondrá así con sendas apps en el Play Store y el nombre bajo el que aparezcan podrán elegirlo ellos mismos, haciendo crecer así su marca.

12.4. Estrategia Comercial

12.4.1. Redes Comerciales

En la actualidad existen muchas empresas que se dedican a comercializar los productos de terceros. Esto permite externalizar la red comercial y ofrecer a diferentes empresas la posibilidad de colaborar con CronCare a cambio de una comisión por venta y el compromiso de seguir premiando la permanencia de los clientes que ellos atraigan. De esta forma se hace uso del B2B[1] para acaparar más mercado y tener agentes que visiten los centros de las diferentes comunidades autónomas para mostrar el sistema y ofrecerles la prueba gratuita. Si el sistema funciona, se trataría de imitar el modelo en otro países. Comenzando por los núcleos de población y haciendo una expansión radial a las ciudad más extra periféricas de cada región.

12.4.2. Win-Win

Por otro lado cada centro o residencia es libre de hacer negocio a su vez gracias a CronCare. Aunque el plan de precio fuera una tarifa cerrada, ellos son libres de cobrar una cuota mensual, por ejemplo, a sus clientes por darles acceso al servicio. Otros preferirán añadirlo a su oferta tecnológica sin coste para los familiares y así dar una imagen más innovadora a través del valor añadido que ofrece un sistema como Croncare.

12.4.3. El salto a la Nube

Aunque el concepto general de «comercio online» está asimilado por la mayoría de las personas, el concepto de «negocio en la nube» no parece estar tan claro. No basta con tener una página web que ofrece un catálogo online y un carrito de la compra virtual. Un negocio en la nube está 100 % automatizado en cada uno de sus procesos, incluido el de adquisición de clientes. Algo así como comprar entradas para un concierto e imprimirlas.

Por ello y aunque los primeros clientes probablemente firmen contratos físicos y se tenga un trato más personal con ellos para conocer los entresijos de las primeras instalaciones del sistema, se plantea como estrategia a largo plazo crear un portal web capaz de registrar y gestionar la facturación recurrente de cada cliente. De esta forma cualquier persona del mundo podría acceder a CronCare con sólo rellenar una serie de datos en un formulario. Las limitaciones geográficas a este nivel se reducen al número de idiomas que sea posible traducir el sistema.

[1] Del inglés «Business To Business»: describe una operación comercial entre dos empresas.

Capítulo 13

Proyección Económica-Financiera

Para la realización de este capítulo se propone un caso ficticio en el que el sistema se trataría de llevar al mercado. Para ello se deben cubrir muchos requisitos en cuestión de personal e infraestructura que permitan llevar a cabo algo parecido al modelo de negocio planteado en el capítulo anterior.

13.1. Personal y Gastos

Se plantea un equipo de producción de software formado por 2 empleados durante el primer año y medio, momento en el que se incorporaría un tercero posiblemente para tratar de crecer hacia nuevas plataformas.

Respecto al equipo de marketing se ha considerado un único empleado, pues las campañas serán online y posiblemente se subcontraten servicios de publicidad de terceros.

Por último se cuenta con una secretaria que gestiona la comunicación de la empresa con el exterior y organiza el calendario de eventos y, por supuesto, con un director que lleve las riendas.

EQUIPO DE PRODUCCIÓN	Sem.1	Sem.2	Sem.3	Sem.4	Sem.5	Sem.6	**bruto / mes**	Sem.1	Sem.2	Sem.3	Sem.4	Sem.5	Sem.6
PRODUCCIÓN nº empleados	2	2	2	3	3	3	**1.500**	18000	18000	18000	27000	27000	27000
MARKETING nº empleados	1	1	1	1	1	1	**1.500**	9000	9000	9000	9000	9000	9000
TOTAL PROD. nº empleados	3	3	3	4	4	4	**3.000**	**27.000**	**27.000**	**27.000**	**36.000**	**36.000**	**36.000**
EQUIPO ADMINIST.													
DIRECTOR nº empleados	1	1	1	1	1	1	**2.300**	13800	13800	13800	13800	13800	13800
SECRETARÍA nº empleados	1	1	1	1	1	1	**1.200**	7200	7200	3600	7200	7200	7200
TOTAL ADMIN. nº empleados	**2**	**2**	**2**	**2**	**2**	**2**	**3.500**	**21.000**	**21.000**	**17.400**	**21.000**	**21.000**	**21.000**
TOTAL PERSONAL	5	5	5	6	6	6	**4.700**	**48.000**	**48.000**	**44.400**	**57.000**	**57.000**	**57.000**

Tabla 13.1: Tabla de Personal y Gastos

13.2. Inversiones Fijas

Las inversiones fijas son los gastos que realiza la empresa para adquirir todos los activos necesarios para el normal funcionamiento del negocio. Dentro de estos podemos considerar: Terrenos, inmuebles, maquinarias, equipos de oficina, muebles, vehículos, etc.
En este caso requerimos de material informático, muebles, licencias y gastos iniciales.

	1-2015	2-2015	1-2016	2-2016	1-2017	2-2017	TOTAL
Ordenadores y servidores	3.000	0	2.000	1.000	1.000	1.000	**8.000**
Muebles y equip	3.000	1.000	1.000	1.000	1.000	1.000	**8.000**
Licencias marcas software	2.000	0	0	0	0	0	**2.000**
Gastos arranque	5.000	0	0	0	0	0	**5.000**
	13.000	**1.000**	**3.000**	**2.000**	**2.000**	**2.000**	**23.000**

Tabla 13.2: Tabla de Inversiones Fijas

13.3. Créditos y Pagos

Aunque se podría escribir largo y tendido sobre los diferentes métodos de financiación que están revolucionando el mercado, como el Crowdfunding, nos limitaremos a suponer un préstamo bancario del monto suficiente para comenzar la empresa y brindarle algo de holgura financiera en sus primeras fases.

	1-2015	2-2015	1-2016	2-2016	1-2017	2-2017	TOTAL
Capital y Créditos recibidos	80.000	0	0	0	0	0	**80.000**
Devoluciones	0	0	-18.900	-19.600	-20.400	-21.100	**-80.000**
Saldo	80.000	80.000	61.100	41.500	21.100	0	0

Tabla 13.3: Tabla de Créditos y Pagos

13.4. Ventas Estimadas y Margen

En la tabla de ventas estimadas y margen de beneficio se muestra un valor denominado EBITDA. El EBITDA es un indicador financiero, acrónimo del inglés «Earnings Before Interest, Taxes, Depreciation, and Amortization» (beneficio antes de intereses, impuestos, depreciaciones y amortizaciones), es decir, el beneficio bruto de explotación calculado antes de la deducibilidad de los gastos financieros.[25]

	1-2015	2-2015	1-2016	2-2016	1-2017	2-2017	TOTAL
Ventas estimadas, euros	9.937	44.000	68.000	97.000	127.000	193.000	**538.937**
MENOS COMISIONES VENTA	-1.491	-6.600	-10.200	-14.550	-19.050	-28.950	**-80.841**
Margen comercial bruto	**8.446**	**37.400**	**57.800**	**82.450**	**107.950**	**164.050**	**458.096**
Gastos de producción	27.000	27.000	27.000	36.000	36.000	36.000	**189.000**
Gastos de personal de gestión	21.000	21.000	17.400	21.000	21.000	21.000	**122.400**
Otros gastos diversos	2.000	2.700	3.150	3.750	4.350	5.650	**21.600**
Total gastos	**50.000**	**50.700**	**47.550**	**60.750**	**61.350**	**62.650**	**333.000**
EBITDA	**-41.554**	**-13.300**	**10.250**	**21.700**	**46.600**	**101.400**	**125.096**
Amortizaciones	1.950	2.100	2.550	2.850	3.150	3.450	**16.050**
RESULTADO DE EXPLOTACIÓN	**-43.504**	**-15.400**	**7.700**	**18.850**	**43.450**	**97.950**	**109.046**
Gastos financieros	3.000	3.000	2.700	2.000	1.200	500	**12.400**
RESULTADO ANTES IMPTOS.	**-46.504**	**-15.400**	**7.700**	**18.850**	**43.450**	**97.950**	**106.046**

Tabla 13.4: Tabla de Ventas Estimadas y Margen

13.5. Proyección de Tesorería

Por último se muestra el cash-flow que esta empresa ficticia tendría a lo largo de tres ejercicios. Se puede observar que existe un hito en el primer semestre del último año (1-2017), pues se logra liquidar el préstamo bancario y las ganancias generarían beneficio tras deducir gastos en adelante.

	1-2015	2-2015	1-2016	2-2016	1-2017	2-2017
Entradas a tesorería	87.179,48	31.790,00	49.130,00	70.082,50	91.757,50	139.442,50
Salidas de Tesorería	66.000,00	54.700,00	72.150,00	84.350,00	84.950,00	86.250,00
Flujo de tesorería	**21.179,48**	**-22.910,00**	**-23.020,00**	**-14.267,50**	**6.807,50**	**53.192,50**

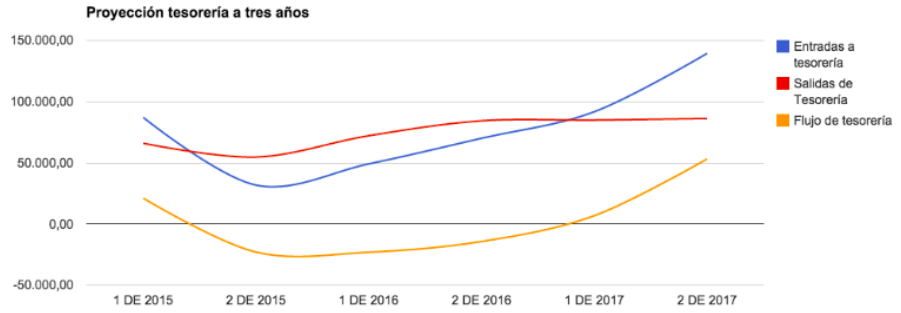


Tabla 13.5: Proyección de Tesorería

Capítulo 14

Pruebas

En este capítulo se describen las pruebas realizadas a lo largo del desarrollo del proyecto. La fase de prueba se lleva a cabo a lo largo de forma continua durante del ciclo de vida del proyecto: Al final de cada etapa de desarrollo se somete a evaluación la aplicación, pudiendo comprobar así que los requisitos definidos en la fase de análisis quedan satisfechos. Esto permite una temprana detección de errores y posibilita resolverlos antes de que el nivel de complejidad del proyecto crezca.

Citando al experto en pruebas de software Glen Myers, «El aumento de la competencia e interconexión entre todos los mercados ha obligado a las empresas a acortar su tiempo de salida al mercado sin dejar de ofrecer productos de alta calidad a sus clientes. Esto es particularmente cierto en una era en la que Internet hace posible la entrega de aplicaciones y servicios de software de manera casi instantánea.»[17]

Durante la implementación se realizaron diversos tipos de pruebas a continuación expondremos algunas de ellas.

14.1. Pruebas de Rendimiento

Principalmente se centraron en medir los posibles retrasos debido a la interacción con el servidor. Según se iba incrementando el abanico de llamadas API disponibles se utilizaban herramientas de depuración para medir la latencia de la conexión al servidor.

También es importante probar las apps en distintos modelos (modelos físicos preferibles a los emuladores) de teléfonos y tabletas con el objetivo de establecer la versión mínima de Android compatible.

14.2. Pruebas de Validación

Para la validación nos hemos centrado en los casos que iban saliendo a luz con las pruebas en entornos reales. Así se llego a controlar un gran número de excepciones con informe visual para el usuario. Un claro ejemplo es un perfil de educador que intenta despertar una tablet

donde tenía una sesión abierta tras haber iniciado sesión en otro dispositivo. En dicha situación su sesión anterior expirará automáticamente.

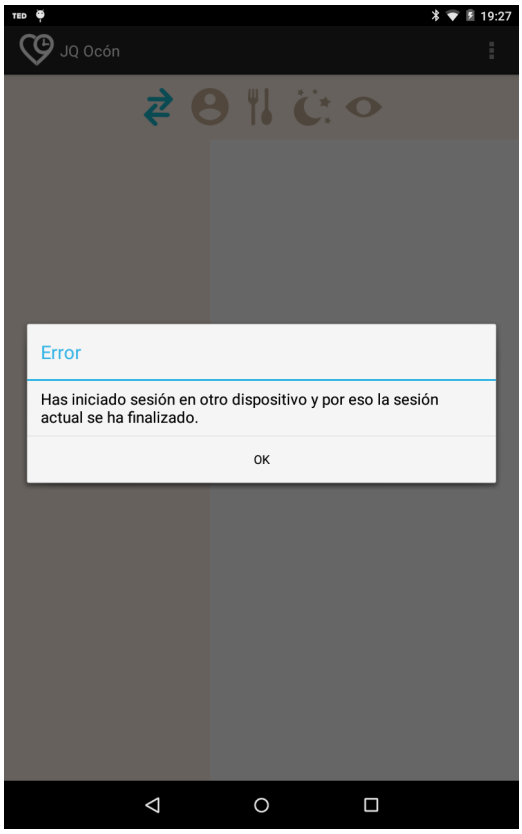


Figura 14.1: Excepción en el Intento de Inicio de Sesión Múltiple

14.3. Pruebas de Usabilidad

Siempre que se modificaba la interfaz de la app se le presentaba a un grupo de personas «no contaminadas» por el proyecto para que hicieran un uso de la app desde cero, sin noción alguna sobre su funcionamiento.

De este tipo de pruebas se extrajo información valiosa para reorganizar ciertos elementos dentro de la interfaz como pueden ser los botones del menú superior que inicialmente se encontraban ocultos tras el botón 'ajustes' del sistema.

Capítulo 15

Conclusiones

Sin duda hace falta llevar a cabo un proyecto de fin de carrera para valorar el conocimiento adquirido tras años de cursar asignaturas que tocan tantas y tan diversas ramas de la informática.

«Podría uno remontarse a los primeros cursos, donde se comienza a estudiar el desarrollo orientado a objetos y se aprende a evaluar el orden de ejecución de un algoritmo, y se daría cuenta de que gran parte de lo aprendido durante esas lecciones está reflejado en este trabajo.»

15.1. Conclusiones Académicas

El haber cursado asignaturas sobre la Ingeniería del Software es fundamental a la hora de abordar un proyecto, sin este conocimiento no se aplicarían las técnicas adecuadas en fases críticas como son la de análisis o la de diseño.

Cuando se cursan este tipo de asignaturas, habitualmente se estudian metodologías de desarrollo ágil y técnicas de gestión de proyecto. Esa base teórica se antoja insuficiente para la comprensión del impacto real que estas técnicas tienen sobre el tiempo de ejecución del proyecto y la calidad del producto resultante. Es por ello que el autor ha tratado de seguir estas metodologías adaptándolas a sus necesidades cuando era necesario.

Realizar un proyecto de esta envergadura, el cual engloba la implementación de dos apps Android y una API que las interconecta, permite al alumno mejorar y adquirir nuevas capacidades de cara al futuro profesional. Una de estas capacidades en las que cabe un gran margen de mejora es la estimación. Fase tras fase se ha tratado de mejorar en este aspecto, tratando de estimar la cantidad de tiempo y los recursos que serían necesarios para cada módulo del sistema. «El arte de la estimación consiste en perseguir un objetivo imposible a través de un camino asintótico».

El reto de diseñar e implementar una API para CronCare ha mejorado la comprensión del autor sobre las mismas y su capacidad para interactuar con servicios que ofrecen terceros.

Tanto alumno como tutor valoran muy positivamente la culminación de este proyecto que representa muchas horas de estudio, trabajo y dedicación.

15.2. Conclusiones Personales

Ante todo el autor quisiera manifestar la satisfacción que le produce el haber trabajado en un proyecto que, potencialmente puede mejorar, aunque sólo sea un pequeño aspecto de la vida de muchas familias. Por otra parte considera que se han puesto a prueba sus capacidades de investigación y comunicación en otros idiomas pues, en ocasiones, ha tenido que indagar en la red en busca de foros de programadores o similares para encontrar solución a algún problema.

Un aspecto que enorgullece personalmente al autor, aunque pudiera parecer nimio, es el diseño del icono y la elección del nombre, pues lo cierto es que no se considera muy capaz en labores creativas o artísticas.

No se puede cerrar este capítulo sin agradecer a las personas que han colaborado en calidad de «usuarios beta», fue gracias a todas las pruebas realizadas fuera del entorno académico que se pudo ver la utilidad real de un producto como CronCare, y esto fue un factor determinante en cuanto a motivación para su finalización.

Capítulo 16

Trabajo Futuro

Durante las fases de análisis, e incluso el desarrollo surgieron muchas ideas que tuvieron que quedar en el tintero. Otras mejoras simplemente son sugeridas por terceros que tienen acceso a probar el sistema. Muchas veces es la opinión de otro la que indica el camino adecuado, pues la implicación que uno tiene con su propio proyecto puede llegar a contaminar su juicio sobre el mismo. A continuación se listan varias mejoras que sin duda harían de CronCare un sistema mejor y más completo:

- Agregar capacidades multimedia a las apps:

  Es indudable que recibir información al instante es ya todo un avance respecto al antiguo método de llamar una vez al día a preguntar por un familiar. Pero recibir fotos y videos de las actividades o incluso simples momentos cotidianos que ocurren en los centros, eso sería un plus enorme para el producto.
  Es cierto que esta mejora implicaría ciertos aspectos legales a tener en cuenta en cuanto a la política de privacidad y uso de los datos.

- Nuevos idiomas:

  Vale la pena invertir en la traducción del fichero que contiene todas las ristras de texto que se muestran en una app. Cada idioma que se agrega al sistema es una puerta que se abre a un nuevo mercado.

- Página web comercial y panel de gestión on-line

  La visibilidad en la red es clave a la hora de comercializar una herramienta tecnológica y más si es una herramienta que utiliza internet como medio clave. Es por ello que una página web corporativa, descriptiva del producto y con opciones para apuntarse a una lista de correo es casi indispensable si esta idea saliera al mercado.
  Por otro lado, en esa misma dirección web, se debería habilitar un acceso para las cuentas de administradores de centros y residencias. Dicho acceso les llevaría a un panel de gestión que debería ofrecer una interfaz clara e intuitiva para realizar las operaciones de gestión de recursos, mensajería y personal dentro de CronCare.

- Apps para iPhone:

  Aunque aquí en España Android está en la gran mayoría de los dispositivos, esto no es así a lo largo de todo el globo. En EE.UU por ejemplo el mercado está muy dividido y

no se debe olvidar el detalle de que iPhone fue el primer smartphone con una plataforma propia de distribución de apps. Sería muy conveniente de cara al porvenir de este proyecto desarrollar una versión para iOS.

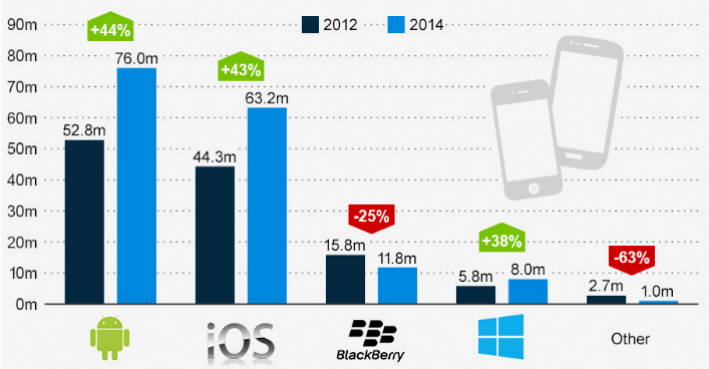


Figura 16.1: Millones de Usuarios por Smartphone en EE.UU. Fuente: eMarketer[8]

Apéndices

Apéndice A

Especificaciones del Análisis

Para facilitar la lectura del documento, se ha decidido listar en este apéndice todos los casos de uso que se definieron durante las distintas fases de análisis del proyecto.

A.1. Modelo de Casos de Uso

Cada caso de uso se representa mediante una ficha descriptiva compuesta por los siguientes campos:

- ID: valor numérico único y secuencial que identifica al caso de uso.
- Nombre: etiqueta asignada para definir de forma muy breve el caso de uso.
- Actores: menta a los actores involucrados en el caso de uso descrito.
- Descripción: explicación detallada del caso de uso.
- Precondiciones: define las condiciones del sistema que deben cumplirse para que se pueda satisfacer el caso de uso.
- Post-condiciones: define las condiciones del sistema que deben cumplirse una vez satisfecho el caso de uso.
- Flujo deseable: describe el camino que sigue el sistema cuando el caso de uso cumple su objetivo de forma exitosa.
- Flujo excepcional: describe los posibles caminos que sigue el sistema cuando una excepción no permite que se de el flujo deseable del caso de uso.

A.1.1. Actores del Sistema

- Administrador
    - Actúa con el sistema mediante llamadas a la API, tienes la capacidad de crear recursos para el centro o residencia en el sistema. Crea perfiles de cuidadores, administra a los residentes, envía mensajes generales, etc.

- Familiar o interesado en el seguimiento de un residente

  - Este actor hará uso de la app CronCare. Excepto por el envío de notas cortas en caso de ausencia del residente en el centro y el propio registro de usuario en sí, este actor no escribe datos en el sistema. Es un consumidor de información que accede a la mayoría de sus datos mediante llamadas tipo GET a la API.

- Cuidador o trabajador del centro:

  - Mediante la app CronCare Centros, este actor nutre al sistema de datos en el momento que los eventos ocurren. Su función es la de tipificar estos registros entre entrada/salida, inicio/fin de sueño, evento de comedor, registro del estado anímico/-físico y observación libre.

A.1.2. Casos de Uso del Administrador

ID	#01
Nombre	Inicio de sesión
Actores	Administrador
Descripción	Permite a un administrador acceder al sistema.
Precondiciones	1. El administrador deberá proporcionar un par e-mail/contraseña válido en el sistema.
Post-condiciones	1. Se escribe una nueva fila en la tabla «admins_sessions» conteniendo un token único que permitirá futuras llamadas a la API.
Flujo deseable	El cliente HTTP hace una llamada POST a la API, enviando su e-mail y contraseña en el payload, a la URL «/croncare-api/admins/session» y recibe una repuesta con código «200 OK» y una serie datos JSON que incluyen un token único de seguridad y un ID de sesión.
Flujo excepcional	En caso de faltar algún parámetro, no ser válido o cualquier otro error interno del sistema se obtendrá un código de respuesta que indique error y un mensaje corto informativo.

Tabla A.1: Administrador - Tabla de Caso de Uso #01: Iniciar Sesión

ID	#02
Nombre	Cerrar sesión
Actores	Administrador
Descripción	Permite a un administrador finalizar la sesión en el sistema.
Precondiciones	1. El administrador deberá haber iniciado previamente una sesión válida mediante #01. 2. La llamada debe incluir un token único de seguridad en su cabecera.
Post-condiciones	1. Se elimina la fila de la tabla «admins_sessions» cuyo token único coincide con el enviado por el usuario.
Flujo deseable	El cliente HTTP hace una llamada GET a la API a la URL «/croncare-api/admins/session/delete» y recibe una repuesta con código «200 OK».
Flujo excepcional	En caso de faltar algún parámetro, no ser válido o cualquier otro error interno del sistema se obtendrá un código «401 Unauthorized».

Tabla A.2: Administrador - Tabla de Caso de Uso #02: Cerrar Sesión

ID	#03
Nombre	Crear perfil de cuidador
Actores	Administrador
Descripción	Permite a un administrador añadir un nuevo perfil de cuidador al sistema.
Precondiciones	1. El administrador deberá haber iniciado previamente una sesión válida mediante #01. 2. La llamada debe incluir un token único de seguridad en su cabecera. 3. El administrador deberá proporcionar un nombre y al menos un apellido para el nuevo perfil.
Post-condiciones	1. Se crea una nueva fila en la tabla «cuidadores» que incluye un código PIN de 4 dígitos único y autogenerado por el sistema.
Flujo deseable	El cliente HTTP hace una llamada POST a la API, enviando nombre y apellidos en el payload, a la URL «/croncare-api/admins/cuidador» y recibe una repuesta con código «200 OK» y una serie datos JSON que incluyen el ID del nuevo perfil.
Flujo excepcional	En caso de faltar algún parámetro, no ser válido o cualquier otro error interno del sistema se obtendrá un código de respuesta que indique error y un mensaje corto informativo.

Tabla A.3: Administrador - Tabla de Caso de Uso #03: Crear Perfil de Cuidador

ID	#04
Nombre	Eliminar perfil de cuidador
Actores	Administrador
Descripción	Permite a un administrador eliminar un perfil de cuidador existente del sistema.
Precondiciones	1. El administrador deberá haber iniciado previamente una sesión válida mediante #01. 2. La llamada debe incluir un token único de seguridad en su cabecera. 3. El administrador deberá proporcionar un ID válido de un perfil de cuidador existente en el sistema.
Post-condiciones	1. Se establece a '0' el campo de tipo bit «activo» de la fila de la tabla «cuidadores» que contiene el ID recibido en la llamada. 2. Se eliminan en cascada todas las ocurrencias de la tabla «cuidadores_sessions» vinculadas por la clave ajena «ID_CUIDADOR» al perfil eliminado de la tabla «cuidadores».
Flujo deseable	El cliente HTTP hace una llamada GET a la API, enviando la ID de un cuidador en el payload, a la URL «/croncare-api/admins/cuidador/delete» y recibe una repuesta con código «200 OK».
Flujo excepcional	En caso de faltar algún parámetro, no ser válido o cualquier otro error interno del sistema se obtendrá un código de respuesta que indique error y un mensaje corto informativo.

Tabla A.4: Administrador - Tabla de Caso de Uso #04: Eliminar Perfil de Cuidador

ID	#05
Nombre	Crear grupo de residentes
Actores	Administrador
Descripción	Permite a un administrador crear un grupo con un nombre que permitirá categorizar residentes.
Precondiciones	1. El administrador deberá haber iniciado previamente una sesión válida mediante #01. 2. La llamada debe incluir un token único de seguridad en su cabecera. 3. El administrador deberá proporcionar un nombre válido para la posterior identificación del grupo.
Post-condiciones	1. Se crea una fila en la tabla «grupos_residentes» con el nombre indicado por el administrador.
Flujo deseable	El cliente HTTP hace una llamada POST a la API, enviando el nombre del nuevo grupo en el payload, a la URL «/croncare-api/admins/grupos» y recibe una repuesta con código «200 OK» y una serie datos JSON que incluyen el ID del nuevo grupo creado.
Flujo excepcional	En caso de faltar algún parámetro, no ser válido o cualquier otro error interno del sistema se obtendrá un código de respuesta que indique error y un mensaje corto informativo.

Tabla A.5: Administrador - Tabla de Caso de Uso #05: Crear Grupo de Residentes

ID	#06
Nombre	Crear perfil de residente
Actores	Administrador
Descripción	Permite a un administrador añadir un nuevo perfil de residente al sistema.
Precondiciones	1. El administrador deberá haber iniciado previamente una sesión válida mediante #01. 2. La llamada debe incluir un token único de seguridad en su cabecera. 3. El administrador deberá proporcionar un nombre, apellidos, género, fecha de nacimiento y un grupo (de entre los creados anteriormente con #05).
Post-condiciones	1. Se crea una nueva fila en la tabla «residentes» con los valores que se pasan en la llamada.
Flujo deseable	El cliente HTTP hace una llamada POST a la API, enviando nombre, apellidos, género, fecha de nacimiento y grupo en el payload, a la URL «/croncare-api/admins/residente» y recibe una repuesta con código «200 OK» y una serie datos JSON que incluyen el ID del nuevo perfil.
Flujo excepcional	En caso de faltar algún parámetro, no ser válido o cualquier otro error interno del sistema se obtendrá un código de respuesta que indique error y un mensaje corto informativo.

Tabla A.6: Administrador - Tabla de Caso de Uso #06: Crear Perfil de Residente

ID	#07
Nombre	Eliminar perfil de residente
Actores	Administrador
Descripción	Permite a un administrador eliminar un perfil de un residente del sistema.
Precondiciones	1. El administrador deberá haber iniciado previamente una sesión válida mediante #01. 2. La llamada debe incluir un token único de seguridad en su cabecera. 3. El administrador deberá proporcionar un ID válido de un residente existente en el sistema.
Post-condiciones	1. Se establece a '0' el campo de tipo bit «activo» de la fila de la tabla «residentes» que contiene el ID recibido en la llamada. 2. Se eliminan en cascada todas las ocurrencias de la tabla «relaciones_familiar_residente» cuyo valor en el campo «ID_RESIDENTE» coincide con el del perfil eliminado de la tabla «residentes».
Flujo deseable	El cliente HTTP hace una llamada GET a la API, enviando la ID de un residente en el payload, a la URL «/croncare-api/admins/residente/delete» y recibe una repuesta con código «200 OK».
Flujo excepcional	En caso de faltar algún parámetro, no ser válido o cualquier otro error interno del sistema se obtendrá un código de respuesta que indique error y un mensaje corto informativo.

Tabla A.7: Administrador - Tabla de Caso de Uso #07: Eliminar Perfil de Residente

ID	#08
Nombre	Crear relación familiar-residente
Actores	Administrador
Descripción	Permite a un administrador vincular una cuenta de familiar al perfil de un residente.
Precondiciones	1. El administrador deberá haber iniciado previamente una sesión válida mediante #01. 2. La llamada debe incluir un token único de seguridad en su cabecera. 3. El administrador deberá proporcionar un ID válido de un residente existente en el sistema y un e-mail válido de un familiar existente en el sistema.
Post-condiciones	1. Se crea una nueva fila en la tabla «relaciones_familiar_residente» compuesta con las IDs del familiar y del residente.
Flujo deseable	El cliente HTTP hace una llamada POST a la API, enviando la ID de un residente y el e-mail de un familiar en el payload, a la URL «/croncare-api/admins/relacion» y recibe una repuesta con código «200 OK».
Flujo excepcional	En caso de faltar algún parámetro, no ser válido o cualquier otro error interno del sistema se obtendrá un código de respuesta que indique error y un mensaje corto informativo.

Tabla A.8: Administrador - Tabla de Caso de Uso #08: Crear Relación Familiar-Residente

ID	#09
Nombre	Enviar mensaje a familias
Actores	Administrador
Descripción	Permite a un administrador enviar un mensaje que los familiares leerán en sus apps.
Precondiciones	1. El administrador deberá haber iniciado previamente una sesión válida mediante #01. 2. La llamada debe incluir un token único de seguridad en su cabecera. 3. El administrador deberá proporcionar un listado de IDs válidos de grupos de residentes existentes en el sistema, un asunto y un mensaje de texto.
Post-condiciones	1. Se crea una nueva fila en la tabla «mensajes». 2. Se crea una nueva fila por cada familiar vinculado a cada residente perteneciente a los grupos destinatarios, compuesta con las IDs del familiar y del mensaje.
Flujo deseable	El cliente HTTP hace una llamada POST a la API, enviando el listado de IDs de grupos de residentes, un asunto y un mensaje de texto, a la URL «/croncare-api/admins/message» y recibe una repuesta con código «200 OK».
Flujo excepcional	En caso de faltar algún parámetro, no ser válido o cualquier otro error interno del sistema se obtendrá un código de respuesta que indique error y un mensaje corto informativo.

Tabla A.9: Administrador - Tabla de Caso de Uso #09: Enviar Mensaje a Familias

A.1.3. Casos de Uso del Familiar (usuario de la app CronCare)

ID	#10
Nombre	Registrar nuevo usuario
Actores	Familiar
Descripción	Permite a un familiar registrar una nueva cuenta en el sistema.
Precondiciones	1. El usuario debe encontrarse en la pantalla de registro de usuario. 2. El usuario debe completar los campos requeridos con un nombre, apellidos y un par e-mail/contraseña válido. 3. El usuario debe pulsar sobre el botón «Registrarse».
Post-condiciones	1. Se escribe una nueva fila en la tabla «users» con el perfil del nuevo usuario. 2. El usuario se encuentra en la vista de iniciar sesión.
Flujo deseable	El usuario introduce los datos y pulsa el botón, el servidor procesa el registro y la app avanza a la vista de acceso de usuario.
Flujo excepcional	Una mensaje emergente muestra un error referente al registro de usuario.

Tabla A.10: Familiar- Tabla de Caso de Uso #10: Registrar Nuevo Usuario

ID	#11
Nombre	Inicio de sesión
Actores	Familiar
Descripción	Permite a un familiar acceder al sistema.
Precondiciones	1. El usuario debe encontrarse en la pantalla de inicio de sesión. 2. El usuario debe completar los campos requeridos con un par e-mail/contraseña válido. 3. El usuario debe pulsar sobre el botón «Acceder».
Post-condiciones	1. Se escribe una nueva fila en la tabla «users_sessions» conteniendo un token único que permitirá futuras llamadas a la API. 2. El usuario se encuentra en la vista principal de la app.
Flujo deseable	El usuario introduce los datos y pulsa el botón, el servidor procesa el acceso y devuelve un token que ha escrito en la base de datos.
Flujo excepcional	Una mensaje emergente muestra un error referente al inicio de sesión.

Tabla A.11: Familiar- Tabla de Caso de Uso #11: Iniciar Sesión

ID	#12
Nombre	Cerrar sesión
Actores	Familiar
Descripción	Permite a un familiar finalizar la sesión desde la app.
Precondiciones	1. El usuario debe haber iniciado sesión mediante #11. 2. El usuario debe encontrarse en la vista «Configuración». 3. El usuario debe pulsar el botón «Cerrar sesión».
Post-condiciones	1. Se ha eliminado de la tabla «users_sessions» la fila correspondiente a la sesión finalizada. 2. El usuario se encuentra fuera del sistema en la vista «Presentación» de la app.
Flujo deseable	El usuario entra en la vista de configuración y pulsa el botón de cerrar sesión. Seguidamente la app muestra la vista inicial que ve un usuario según la instala.
Flujo excepcional	En caso de fallar la conexión a la red u ocurrir un error en el servidor, el usuario verá un aviso emergente indicando la excepción.

Tabla A.12: Familiar - Tabla de Caso de Uso #12: Cerrar Sesión

ID	#13
Nombre	Elegir residente
Actores	Familiar
Descripción	Permite a un familiar acceder a los registros cronológicos en detalle de un residente.
Precondiciones	1. El usuario debe haber iniciado sesión mediante #11. 2. El menú lateral debe estar desplegado. 3. El usuario debe hacer click sobre el nombre de un familiar en el menú lateral.
Post-condiciones	1. La vista principal muestra los datos de la última fecha disponible en el sistema para el residente seleccionado.
Flujo deseable	El usuario hace click sobre el nombre de su familiar, el menú desplegable se oculta y la vista principal muestra los datos de la última fecha disponible en el sistema para el residente seleccionado.
Flujo excepcional	En caso de fallar la conexión a la red u ocurrir un error en el servidor, el usuario verá un aviso emergente indicando la excepción.

Tabla A.13: Familiar- Tabla de Caso de Uso #13: Elegir Residente

ID	#14
Nombre	Listar mensajes
Actores	Familiar
Descripción	Permite a un familiar acceder a la bandeja de entrada de mensajes del centro.
Precondiciones	1. El usuario debe haber iniciado sesión mediante #11. 2. El menú lateral debe estar desplegado. 3. El usuario debe hacer click sobre el botón «notificaciones».
Post-condiciones	1. La app navega a una vista que lista los mensajes del centro con sus cabeceras y fecha de recepción. Los mensajes están ordenados cronológicamente en orden descendente y se diferencian leídos de no-leídos.
Flujo deseable	El usuario hace click sobre el botón «notificaciones», el menú desplegable se oculta y la app navega a una vista que lista los mensajes del centro.
Flujo excepcional	En caso de fallar la conexión a la red u ocurrir un error en el servidor, el usuario verá un aviso emergente indicando la excepción.

Tabla A.14: Familiar- Tabla de Caso de Uso #14: Listar Mensajes

ID	#15
Nombre	Listar menú semanal
Actores	Familiar
Descripción	Permite a un familiar acceder al menú semanal en el comedor del centro.
Precondiciones	1. El usuario debe haber iniciado sesión mediante #11. 2. El menú lateral debe estar desplegado. 3. El usuario debe hacer click sobre el botón «Menú semanal».
Post-condiciones	1. La app navega a una vista que lista los primeros platos, segundos platos y postres para cada día de la semana actual.
Flujo deseable	El usuario hace click sobre el botón «Menú semanal», el menú desplegable se oculta y la app navega a una vista que lista el menú de cada día para la semana actual.
Flujo excepcional	En caso de fallar la conexión a la red u ocurrir un error en el servidor, el usuario verá un aviso emergente indicando la excepción.

Tabla A.15: Familiar- Tabla de Caso de Uso #15: Listar Menú Semanal

ID	#16
Nombre	Ver datos del centro
Actores	Familiar
Descripción	Permite a un familiar acceder a información general sobre el centro.
Precondiciones	1. El usuario debe haber iniciado sesión mediante #11. 2. El menú lateral debe estar desplegado. 3. El usuario debe hacer click sobre el botón «Contactar».
Post-condiciones	1. La app navega a una vista que lista los datos del centro: teléfonos, e-mails y dirección postal. 2. En caso de tener el usuario apps capaces de reconocer estos datos como parámetros de entrada podrá ejecutarlas desde la app CronCare. Por ejemplo visitar la dirección postal con Google Maps o utilizar la app «teléfono» para llamar al número del centro.
Flujo deseable	El usuario hace click sobre el botón Contactar», el menú desplegable se oculta y la app navega a una vista que muestra los teléfonos, e-mails y dirección postal del centro.
Flujo excepcional	En caso de fallar la conexión a la red u ocurrir un error en el servidor, el usuario verá un aviso emergente indicando la excepción.

Tabla A.16: Familiar- Tabla de Caso de Uso #16: Ver Datos del Centro

ID	#17
Nombre	Cambiar contraseña
Actores	Familiar
Descripción	Permite a un familiar cambiar la contraseña vinculada a su cuenta.
Precondiciones	1. El usuario debe haber iniciado sesión mediante #11. 2. El usuario debe encontrarse en la vista «Configuración». 3. El usuario debe completar los campos «contraseña actual» y «nueva contraseña» con valores válidos. 3. El usuario debe hacer click sobre el botón «Restablecer».
Post-condiciones	1. La tabla «users» se ha visto modificada en el campo que almacena el hash de la contraseña del usuario. 2. El usuario ve una venta emergente que confirma el cambio de contraseña.
Flujo deseable	El usuario rellena los campos «contraseña actual» y «nueva contraseña» con valores válidos, pulsa el botón «Restablecer» y la app muestra una venta emergente que confirma el cambio de contraseña.
Flujo excepcional	En caso de fallar la conexión a la red u ocurrir un error en el servidor, el usuario verá un aviso emergente indicando la excepción.

Tabla A.17: Familiar- Tabla de Caso de Uso #17: Cambiar Contraseña

ID	#18
Nombre	Enviar nota corta
Actores	Familiar
Descripción	Permite a un redactar una nota mientras el residente no está presente en el centro.
Precondiciones	1. El usuario debe haber iniciado sesión mediante #11. 2. El residente seleccionado no debe encontrarse presente en el centro. 3. El usuario debe encontrarse en la vista «redacción de nota corta». 4. El usuario debe escribir un texto de hasta 300 caracteres. 5. El usuario debe pulsar el botón «guardar».
Post-condiciones	1. Se ha escrito un nuevo registro en la tabla «notas». 2. El usuario está de vuelta en la vista principal del sistema.
Flujo deseable	El usuario entra en la vista de redactar nota y recupera la nota actual que podrá seguir editando mientras el residente no acceda al centro.
Flujo excepcional	En caso de encontrarse el familiar en el centro el familiar no podrá acceder a la vista de redactar nota y será avisado con una ventana emergente. En caso de fallar la conexión a la red u ocurrir un error en el servidor, el usuario verá un aviso emergente indicando la excepción.

Tabla A.18: Familiar- Tabla de Caso de Uso #18: Enviar Nota Corta

ID	#19
Nombre	Ver estadísticas de evolución de un residente
Actores	Familiar
Descripción	Permite a un familiar ver estadísticas de evolución del residente actual
Precondiciones	1. El usuario debe haber iniciado sesión mediante #11. 2. El usuario debe encontrarse en la vista principal del sistema 3. El usuario debe hacer click sobre el botón con el icono de gráfica bidimensional en la parte superior derecha.
Post-condiciones	1. El usuario está en una vista que muestra las gráficas de evolución para los últimos 30 días.
Flujo deseable	El usuario hace click sobre el botón con el icono de gráfica y accede a la vista de gráficas de evolución.
Flujo excepcional	En caso de fallar la conexión a la red u ocurrir un error en el servidor, el usuario verá un aviso emergente indicando la excepción y no avanzará a la siguiente vista.

Tabla A.19: Familiar- Tabla de Caso de Uso #19: Ver Estadísticas de Evolución

A.1.4. Casos de Uso del Cuidador (usuario de la app CronCare Centros)

ID	#20
Nombre	Inicio de sesión
Actores	Cuidador
Descripción	Permite a un cuidador acceder al sistema.
Precondiciones	1. El usuario debe encontrarse en la pantalla de inicio de sesión. 2. El usuario debe completar el campo «código PIN» con un código válido.
Post-condiciones	1. Se escribe una nueva fila en la tabla «cuidadores_sessions» conteniendo un token único que permitirá futuras llamadas a la API. 2. El usuario se encuentra en la vista principal de la app.
Flujo deseable	El usuario introduce su código PIN único y pulsa el botón, el servidor procesa el acceso y devuelve un token que ha escrito en la base de datos.
Flujo excepcional	Una mensaje emergente muestra un error referente al inicio de sesión.

Tabla A.20: Cuidador - Tabla de Caso de Uso #20: Iniciar Sesión

ID	#21
Nombre	Cerrar sesión
Actores	Cuidador
Descripción	Permite a un cuidador finalizar la sesión desde la app.
Precondiciones	1. El cuidador debe haber iniciado sesión mediante #20. 2. El cuidador debe pulsar el botón «Cerrar sesión».
Post-condiciones	1. Se ha eliminado de la tabla «cuidadores_sessions» la fila correspondiente a la sesión finalizada. 2. El cuidador se encuentra fuera del sistema en la vista «Iniciar sesión» de la app.
Flujo deseable	El cuidador pulsa el botón de cerrar sesión en la esquina superior derecha de la pantalla. Seguidamente la app muestra la vista de acceso mediante código PIN.
Flujo excepcional	En caso de fallar la conexión a la red u ocurrir un error en el servidor, el usuario verá un aviso emergente indicando la excepción.

Tabla A.21: Cuidador - Tabla de Caso de Uso #21: Cerrar Sesión

ID	#22
Nombre	Listar residentes
Actores	Cuidador
Descripción	Permite a un cuidador recibir un listado de residentes sobre los que tomar datos.
Precondiciones	1. El cuidador debe haber iniciado sesión mediante #20. 2. Debe haber al menos un grupo de residentes seleccionado en la columna izquierda de la pantalla.
Post-condiciones	1. La app muestra un listado de residentes basada en el evento actual seleccionado en el menú superior y los grupos seleccionados en la columna lateral izquierda.
Flujo deseable	El cuidador selecciona uno o más grupos de residentes de la columna izquierda, el sistema devuelve el listado de residentes adecuado para el tipo de evento seleccionado en el menú superior.
Flujo excepcional	En caso de fallar la conexión a la red u ocurrir un error en el servidor, el usuario verá un aviso emergente indicando la excepción.

Tabla A.22: Cuidador - Tabla de Caso de Uso #22: Listar Residentes

ID	#23
Nombre	Enviar toma de datos tipo entrada/salida o sueño
Actores	Cuidador
Descripción	Permite a un cuidador enviar datos de tipo entrada/salida o sueño sobre uno o más residentes al sistema.
Precondiciones	1. El cuidador debe haber iniciado sesión mediante #20. 2. Debe estar seleccionada la opción «entrada/salida» o «sueño» en el menú superior. 3. Debe haber al menos un residente seleccionado en el listado principal.
Post-condiciones	1. El residente o los residentes seleccionados invierten su estado actual. (Ejemplo: estaba dormido y ahora está despierto.)
Flujo deseable	El cuidador selecciona uno o más residentes del listado principal y pulsa en botón de enviar toma de datos, el residente o los residentes seleccionados invierten su estado actual. «Ausentes» que pasan a «presentes», o «despiertos» que pasan a «dormidos».
Flujo excepcional	En caso de fallar la conexión a la red u ocurrir un error en el servidor, el usuario verá un aviso emergente indicando la excepción.

Tabla A.23: Cuidador -Tabla de Caso de Uso #23: Enviar Toma de Datos Sin Detalle

ID	#24
Nombre	Enviar toma de datos detallada
Actores	Cuidador
Descripción	Permite a un cuidador enviar datos de tipo «estado anímico/físico», «comedor» u «observación libre» sobre uno o más residentes al sistema.
Precondiciones	1. El cuidador debe haber iniciado sesión mediante #20. 2. Debe estar seleccionada una opción de tipo «estado anímico/físico», «comedor» y «observación libre» en el menú superior. 3. Debe haber al menos un residente seleccionado en el listado principal. 4. Deben completarse los datos específicos del evento.
Post-condiciones	1. El sistema ha registrado los nuevos eventos. 2. La app del cuidador vuelve a la vista principal.
Flujo deseable	El cuidador selecciona uno o más residentes del listado principal y pulsa en botón de continuar toma de datos, el sistema registra los nuevos eventos y la app retorna a la vista principal con el listado de residentes.
Flujo excepcional	En caso de fallar la conexión a la red u ocurrir un error en el servidor, el usuario verá un aviso emergente indicando la excepción.

Tabla A.24: Cuidador - Tabla de Caso de Uso #23: Enviar Toma de Datos Detallada

Apéndice B

Clases Java-Android

B.1. ActivityHTTP.java

```
package com.croncare.croncare;

import android.app.Activity;
import android.app.AlertDialog;
import android.content.DialogInterface;
import android.content.Intent;
import android.os.Bundle;

import org.json.JSONObject;

public class ActivityHTTP extends Activity {

    public static boolean isVisible=false;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        this.isVisible=true;
    }

    @Override
    protected void onResume() {
        super.onResume();
        this.isVisible=true;
    }

    @Override
    protected void onPause() {
        super.onPause();
        this.isVisible=false;
    }
```

```
//Responses for each method that can be used to call the API
public void responseGET(JSONObject obj, String action){};
public void errorGET(String action, Integer errorcode,
   JSONObject errormsg){
    boolean killit = errorcode!=0;
    showErrorMessage(R.string.error_network, killit);
}

public void responseDELETE(JSONObject obj, String action){};
public void errorDELETE(String action, Integer errorcode,
   JSONObject errormsg){
    boolean killit = errorcode!=0;
    showErrorMessage(R.string.error_network, killit);
}

public void responsePOST(JSONObject obj, String action){};
public void errorPOST(String action, Integer errorcode,
   JSONObject errormsg){
    boolean killit = errorcode!=0;
    showErrorMessage(R.string.error_network, killit);
}

//Something went wrong...
public void showAlertMessage(Integer msgTitle, Integer msgError,
    Boolean finishActivity){
    CharSequence msg =getString(msgError);
    AlertDialog.Builder dlgAlert  = new AlertDialog.Builder(this
       );
    dlgAlert.setMessage(msg);
    dlgAlert.setTitle(getString(msgTitle));

    if(finishActivity){
        dlgAlert.setPositiveButton("OK", new DialogInterface.
           OnClickListener() {

            public void onClick(DialogInterface dialog, int
               which) {
                // TODO Auto-generated method stub
                //startActivity(new Intent(getApplicationContext
                   (),ActivityLoader.class));
                finish();
            }

        });
        dlgAlert.setCancelable(false);
```

```
        }
        else{
            dlgAlert.setPositiveButton("OK", new DialogInterface.
                OnClickListener() {

                public void onClick(DialogInterface dialog, int
                    which) {

                }

            });
            dlgAlert.setCancelable(true);
        }
        dlgAlert.create().show();
    };

    //Something went wrong...
    public void showErrorMessage(Integer codeError, Boolean
        finishActivity){
        CharSequence msg =getString(codeError);
        AlertDialog.Builder dlgAlert  = new AlertDialog.Builder(this
            );
        dlgAlert.setMessage(msg);
        dlgAlert.setTitle(R.string.error_title);

        if(finishActivity){
            dlgAlert.setPositiveButton("OK", new DialogInterface.
                OnClickListener() {

                public void onClick(DialogInterface dialog, int
                    which) {
                    // TODO Auto-generated method stub
                    startActivity(new Intent(getApplicationContext()
                        , ActivitySlider.class).setFlags(Intent.
                        FLAG_ACTIVITY_NEW_TASK | Intent.
                        FLAG_ACTIVITY_CLEAR_TASK));
                    finish();
                }

            });
            dlgAlert.setCancelable(false);
        }
        else{
            dlgAlert.setPositiveButton("OK", new DialogInterface.
                OnClickListener() {
```

```
                public void onClick(DialogInterface dialog, int
                    which) {

                }

            });
            dlgAlert.setCancelable(true);
        }
        dlgAlert.create().show();
    };
}
```

Listing B.1: ActivityHTTP.java

B.2. ThreadGET.java

```
package com.croncare.croncare;

import android.app.ProgressDialog;
import android.os.AsyncTask;

import org.json.JSONException;
import org.json.JSONObject;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.MalformedURLException;
import java.net.URL;

import static android.app.ProgressDialog.show;

public class ThreadGET extends AsyncTask<String, Void, JSONObject> {
    private final static String TAG = "croncare_log_ThreadGET";
    private final ActivityHTTP callingActivity;
    private ProgressDialog loadingDialog;
    public final String action;
    private final String webserviceURL;
    private HttpURLConnection connection;
    boolean BlockUI=true;

    public ThreadGET(ActivityHTTP callingActivity, String action){
        this.callingActivity = callingActivity;
        this.action=action;
```

```
        this.webserviceURL=callingActivity.getString(R.string.
            SERVER_DOMAIN);
    }

    public ThreadGET(ActivityHTTP callingActivity, String action,
        boolean BlockUI){
        this.callingActivity = callingActivity;
        this.action=action;
        this.webserviceURL=callingActivity.getString(R.string.
            SERVER_DOMAIN);
        this.BlockUI=BlockUI;
    }

    @Override
    protected void onPreExecute(){
        if(BlockUI && callingActivity.isVisible)
            loadingDialog=show(callingActivity,callingActivity.
                getString(R.string.wait_please), callingActivity.
                getString(R.string.loading), true, false);
    }

    @Override
    protected JSONObject doInBackground(String... params) {

        URL url = null;
        String respStr = "";
        try {
            url = new URL(webserviceURL+params[0]);

            connection = (HttpURLConnection) url.openConnection();
            connection.setConnectTimeout(25000);
            connection.setDoInput(true);
            connection.setInstanceFollowRedirects(false);
            connection.setUseCaches (false);

            BufferedReader reader;

            if(connection.getResponseCode()==200)
                reader = new BufferedReader(new InputStreamReader(
                    connection.getInputStream()));
            else
                reader = new BufferedReader(new InputStreamReader(
                    connection.getErrorStream()));

            String line;
```

```
            while ((line=reader.readLine()) != null) {
                respStr+=line;
            }
            reader.close();

            connection.disconnect();

        } catch (MalformedURLException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }

        if(respStr.length()>0){

            try {
                return new JSONObject(respStr);
            } catch (JSONException e) {
                e.printStackTrace();
            }
        }

        return new JSONObject();
    }

    @Override
    protected void onPostExecute(JSONObject result) {
        if(callingActivity.isVisible){
            try {

                if(connection.getResponseCode()==200)
                    callingActivity.responseGET(result, action);
                else
                    callingActivity.errorGET(action, connection.
                       getResponseCode(), result);

            } catch (IOException e) {
                e.printStackTrace();
                callingActivity.errorGET(action, 0, result);
            }

        }

        if(BlockUI)
            loadingDialog.dismiss();
```

```
        try {
            this.finalize();
        } catch (Throwable throwable) {
            throwable.printStackTrace();
        }
    }
}
```

Listing B.2: ThreadGET.java

B.3. ThreadPOST.java

```
package com.croncare.croncare;

import android.app.ProgressDialog;
import android.content.Context;
import android.os.AsyncTask;
import android.util.Log;

import org.json.JSONException;
import org.json.JSONObject;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.net.HttpURLConnection;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.ArrayList;

import static android.app.ProgressDialog.show;

public class ThreadPOST extends AsyncTask<ArrayList<String>, Void,
    JSONObject> {
    private final static String TAG = "croncare_log_ThreadPOST";
    private ActivityHTTP callingActivity=null;
    private Context ctx=null;
    private ProgressDialog loadingDialog;
    public String action=null;
    private final String webserviceURL;
    private HttpURLConnection connection;
    boolean BlockUI=true;

    public ThreadPOST(ActivityHTTP callingActivity, String action){
```

```
        this.callingActivity = callingActivity;
        this.action=action;
        this.webserviceURL=callingActivity.getString(R.string.
            SERVER_DOMAIN);
    }

    public ThreadPOST(ActivityHTTP callingActivity, String action,
        boolean BlockUI){
        this.callingActivity = callingActivity;
        this.action=action;
        this.webserviceURL=callingActivity.getString(R.string.
            SERVER_DOMAIN);
        this.BlockUI=BlockUI;
    }

    public ThreadPOST(Context context){
        this.ctx = context;
        this.webserviceURL=ctx.getString(R.string.SERVER_DOMAIN);
        this.BlockUI=false;
    }

    @Override
    protected void onPreExecute(){
        if(BlockUI)
            loadingDialog=show(callingActivity,callingActivity.
                getString(R.string.wait_please), callingActivity.
                getString(R.string.loading), true, false);
    }

    @Override
    protected JSONObject doInBackground(ArrayList<String>... params)
        {

        String urlParameters = "";

        for(int i=1;i<params[0].size();i=i+2){
            urlParameters += params[0].get(i)+"="+params[0].get(i+1)
                ;
            if(i<params[0].size()-2)
                urlParameters+="&";

        }

        URL url = null;
        String respStr = "";
```

```
try {
    url = new URL(webserviceURL+params[0].get(0));
    Log.d(TAG, "send POST to: " + url);
    connection = (HttpURLConnection) url.openConnection();
    connection.setConnectTimeout(25000);
    connection.setDoOutput(true);
    connection.setDoInput(true);
    connection.setInstanceFollowRedirects(false);
    connection.setRequestMethod("POST");
    connection.setRequestProperty("Content-Type", "
       application/x-www-form-urlencoded");
    connection.setRequestProperty("charset", "utf-8");
    connection.setRequestProperty("Content-Length", "" +
       Integer.toString(urlParameters.getBytes().length));
    connection.setUseCaches (false);

    OutputStreamWriter writer = new OutputStreamWriter(
       connection.getOutputStream());
    writer.write(urlParameters);
    writer.flush();

    BufferedReader reader;

    if(connection.getResponseCode()==200)
        reader = new BufferedReader(new InputStreamReader(
           connection.getInputStream()));
    else
        reader = new BufferedReader(new InputStreamReader(
           connection.getErrorStream()));

    String line;

    while ((line=reader.readLine()) != null) {
        respStr+=line;
    }
    writer.close();
    reader.close();

    connection.disconnect();

} catch (MalformedURLException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
```

```
        if(respStr.length()>0){

            try {
                return new JSONObject(respStr);
            } catch (JSONException e) {
                e.printStackTrace();
            }
        }

        return new JSONObject();
    }

    @Override
    protected void onPostExecute(JSONObject result) {
        Log.d(TAG, result.toString());
        if(callingActivity!=null){
            if(callingActivity.isVisible){
                try {

                    if(connection.getResponseCode()==200)
                        callingActivity.responsePOST(result, action)
                            ;
                    else
                        callingActivity.errorPOST(action, connection
                            .getResponseCode(), result);

                } catch (IOException e) {
                    e.printStackTrace();
                    callingActivity.errorPOST(action, 0, result);
                }
            }

            if(BlockUI)
                loadingDialog.dismiss();
        }

        try {
            this.finalize();
        } catch (Throwable throwable) {
            throwable.printStackTrace();
        }
    }
}
```

Listing B.3: ThreadPOST.java

Apéndice C

Ficheros PHP

Index.php

```
<?php
        require_once('config.php');
        require_once('db_functions.php');
        require_once('api_functions.php');
        require_once('croncare_functions.php');

        require_once('langs.php');

        $headers = apache_request_headers();
        $method = strtolower($_SERVER['REQUEST_METHOD']);

        //Check for GET params
        $exp = explode('?', $_SERVER['REQUEST_URI']);

        if(count($exp)>1){
                //Splits the request by '/' and stores it in a
                   sorted array
                $request = explode('/', $exp[0]);
        }
        else{
                //Splits the request by '/' and stores it in a
                   sorted array
                $request = explode('/', $_SERVER['REQUEST_URI']);
        }

        $call_root = array_search($api_folder, $request);
        $request_size=count($request)-$call_root;

        if($request_size<4)
                send_error();
```

```
$api_request=array();
$j=0;
for($i=$call_root+1;$i<count($request);$i++){
        $api_request[$j++]=$request[$i];
}

$api_request_size=count($api_request);

//Grab destination DB
$client_db=$api_request[0];
//Define root collection to access: [ users, edus or admins
   ]
$api_user=$api_request[1];
//The real api call requested
$api_call=$api_request[2];

if($api_request_size>3){
        $api_call_tail=array_slice($api_request, 3);
} else {
        $api_call_tail=array();
}

$stringApiCall=$api_call;
foreach($api_call_tail as $tail){
        $stringApiCall="$stringApiCall/$tail";
}

if(file_exists("$api_user/{$api_call}_{$method}.php")){
        connectToDB($client_db);
        if($method=='post'){
                //grabs the $_POST variables and adds
                   slashes to escape dangerous chars
                foreach ($_POST as $key => $input_arr) {
                        $input_arr=trim($input_arr);
                        //delete empty params
                        if($input_arr=="" || strlen(
                           $input_arr)<1)
                                unset($_POST[$key]);
                        else
                                $_POST[$key] =
                                   mysqli_real_escape_string
                                   ($MySQLconnection,
                                   $input_arr);
                }
        }
```

```
                if(file_exists("$api_user/aa_{$api_user}_functions.
                    php"))
                        include_once("$api_user/aa_{$api_user}
                            _functions.php");
                if(file_exists("$api_user/aab_{$api_user}_{$method}.
                    php"))
                        include_once("$api_user/aab_{$api_user}_{
                            $method}.php");

                require_once("$api_user/{$api_call}_{$method}.php");
        }

        send_error();
        exit;
?>
```

Listing C.1: index.php

Api_functions.php

```
<?php

function checkCallParams($received,$required_vars,$optional_vars=
    array()) {
        if( count($received) < count($required_vars) ||
                count($received) > (count($required_vars) + count(
                    $optional_vars)) )
                send_error();

        //Check required vars
        foreach($required_vars as $key => $type) {
                if(!array_key_exists($key,$received))
                        send_error();
        }

        foreach($received as $key => $value) {
                if(!array_key_exists($key,$required_vars)) {
                        if(!array_key_exists($key,$optional_vars))
                                send_error();
                        else
                                validateVariable($value,
                                    $optional_vars[$key]);
                } else
                        validateVariable($value,$required_vars[$key
                            ]);
```

```
        }

        return;
}

function validateVariable($value,$type) {
        switch($type) {
                case 'email':
                        if(!filter_var($value, FILTER_VALIDATE_EMAIL
                           ))
                                send_error(400,'email no valid');
                        break;

                case 'password':
                        if ( strlen(utf8_decode($value)) < 6 )
                                send_error(400,'password too short')
                                   ;

                        if ( strlen(utf8_decode($value)) > 50 )
                                send_error(400,'password too long');

                        if(!preg_match('/^[A-Za-z0-9!@#$%.?_]{6,50}$
                           /', $value))
                                send_error(400, 'password invalid
                                   chars');
                        break;

                case 'new_password':
                        if ( strlen(utf8_decode($value)) < 6 )
                                send_error(400,'new password too
                                   short');

                        if ( strlen(utf8_decode($value)) > 50 )
                                send_error(400,'new password too
                                   long');

                        if(!preg_match('/^[A-Za-z0-9!@#$%.?_]{6,50}$
                           /', $value))
                                send_error(400, 'new password
                                   invalid chars');
                        break;

                case 'numeric_id':
                        if(!is_numeric((int)$value))
                                send_error(409,'invalid number');
                        break;
```

```
case 'fullPersonName':
        if ( strlen(utf8_decode($value)) > 150 )
                send_error(400,'name too long');
        if ( strlen(utf8_decode($value)) < 2 )
                send_error(400,'name too short');
        break;

case 'personName':
        if ( strlen(utf8_decode($value)) > 50 )
                send_error(400,'name too long');
        if ( strlen(utf8_decode($value)) < 2 )
                send_error(400,'name too short');
        break;

case 'personLastName':
        if ( strlen(utf8_decode($value)) > 100 )
                send_error(400,'lastname too long');
        if ( strlen(utf8_decode($value)) < 1 )
                send_error(400,'lastname too short')
                    ;
        break;

case 'string_50':
        if ( strlen(utf8_decode($value)) > 50 )
                send_error(409,'string too long, 50
                    chars max.');
        break;

case 'gender':
        if ( $value!="male" && $value!="female" )
                send_error(409,'invalid gender');
        break;

case 'sql_date':
        $exp=explode('-',$value);

        if(count($exp)!=3)
                send_error(409,'invalid date format'
                    );

        if(!checkdate($exp[1],$exp[2],$exp[0]))
                send_error(409,'invalid date value')
                    ;
        break;
```

```
                default:
                        break;
        }
        return;
}

?>
```

Listing C.2: api_functions.php

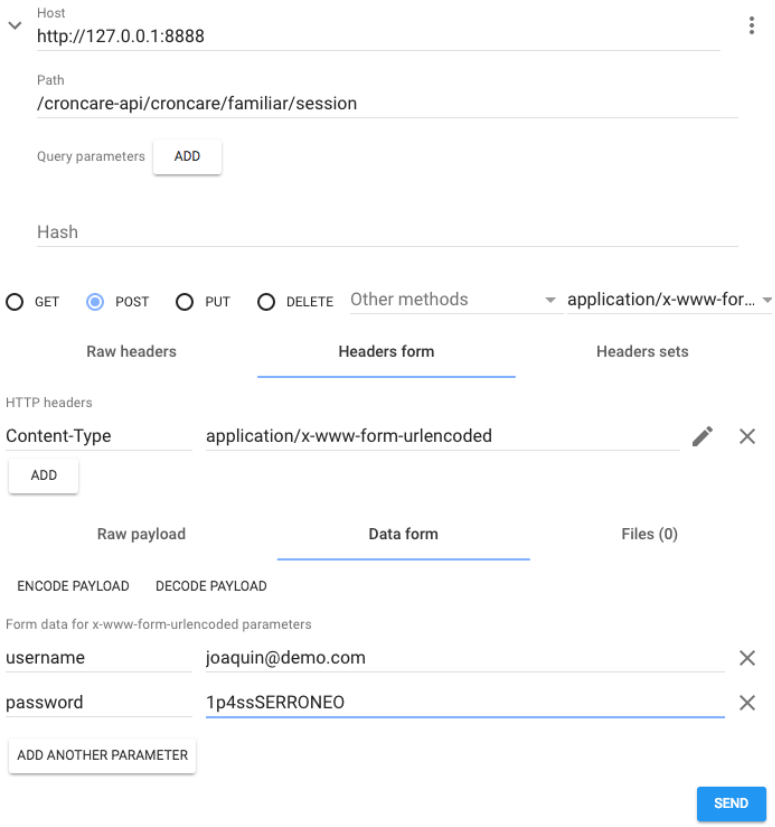
Session_post.php

```
<?php
        //CENTERDB/cuidadores/session
        $required_vars=array('password'=>'numeric_id');
        checkCallParams($_POST, $required_vars);
        $pincode=$_POST["password"];
        $cuidador=db_select("cuidadores","IDCENTER='$idcenter' AND
            ACTIVE='1' AND YEKSSAP='$pincode'","IDCUIDADOR,NAME,
            LASTNAME","LIMIT 1",false);
        if(count($cuidador)<1)
                send_error(401,"bad login");

        //Login correcto
        $idcuidador=$cuidador["IDCUIDADOR"];

        db_delete("cuidadores_sessions","IDCUIDADOR='$idcuidador'");

        $newtoken=bin2hex(openssl_random_pseudo_bytes(16));

        $newsession_array=array(
                                                "IDCUIDADOR"=>
                                                    $idcuidador,
                                                "TOKENSESSION"=>
                                                    $newtoken,
                                                "TS_CREATION"=>$now)
                                                    ;

        $newsession_id=db_insert("cuidadores_sessions",
            $newsession_array);

        $newsession_array["NAME"]=$edu["NAME"]." ".$edu["LASTNAME"];
        send_response($newsession_array);
?>
```

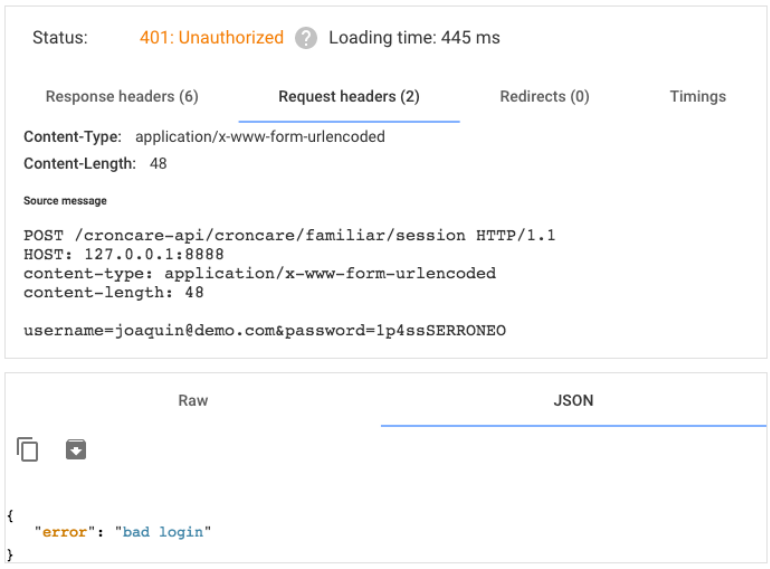
Listing C.3: session_post.php

Apéndice D

Traza de Diferentes Llamadas a la API

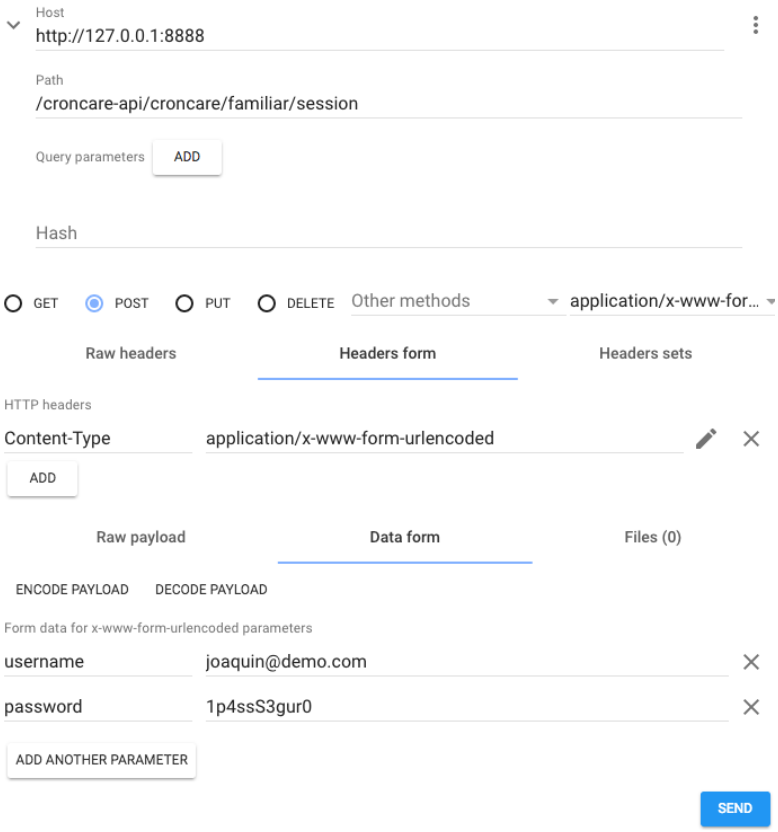


(a) Llamada a la API para Forzar un Error de Acceso

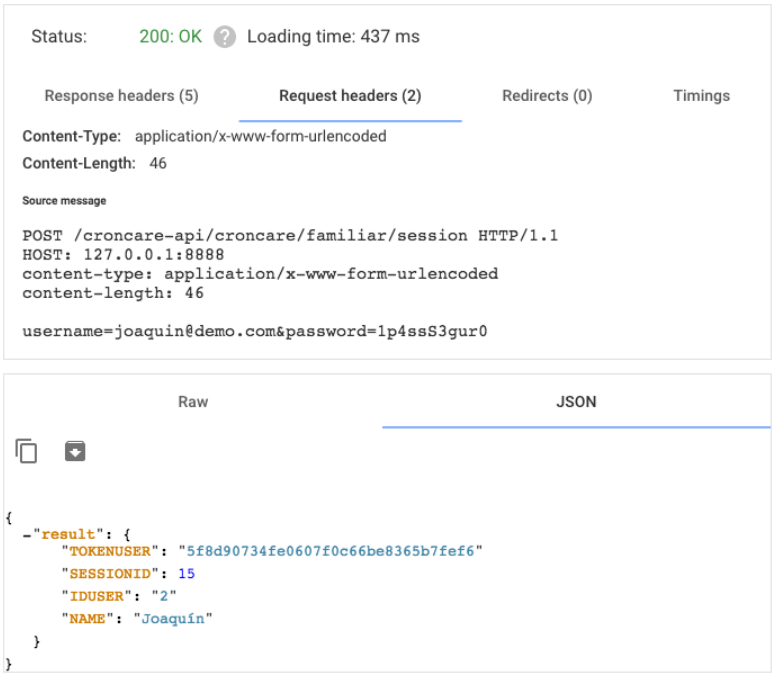


(b) Respuesta de la API ante un Acceso Fallido

Figura D.1: Intento de Acceso Fallido de un Familiar de Residente mediante la API

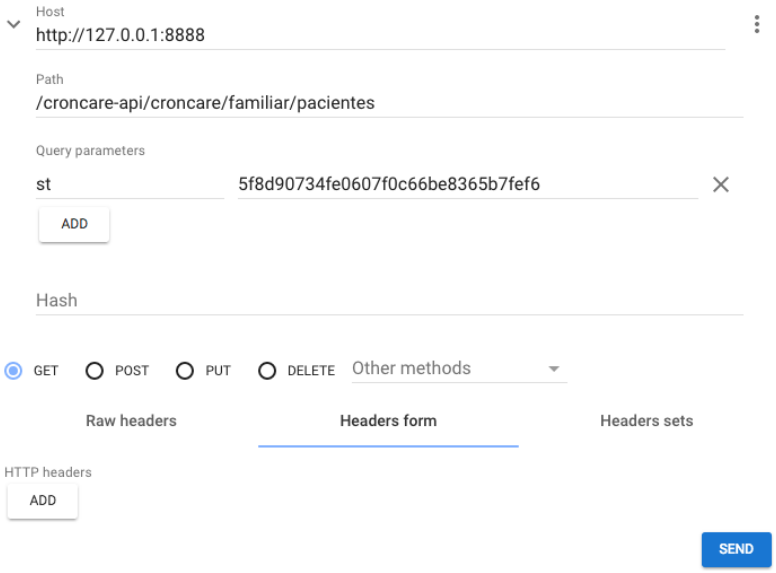


(a) Llamada válida para Acceder al Sistema

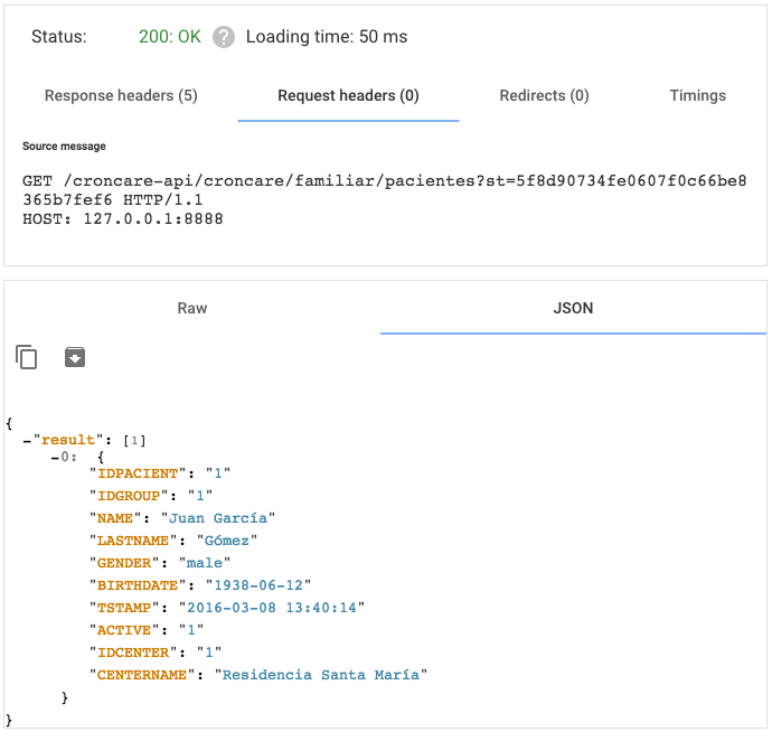


(b) Respuesta de la API ante un Acceso Fallido

Figura D.2: Llamada a la API para Acceder al Sistema



(a) Llamada a la API para Listar Residentes Vinculados a un Familiar



(b) Respuesta de la API para listar residentes vinculados a un familiar

Figura D.3: Llamada para Listar Residentes Vinculados a un Familiar mediante la API

Bibliografía

[1] Grady Booch, James Rumbaugh, and Ivar Jacobson. *UML: el lenguaje unificado de modelado*. Editorial Addison Wesley, ISBN: 84-7829-028-1.

[2] Andrés Cabrera and Enrique Lluch. *Economía y organización de empresas*. Editorial SM, ISBN: 84-348-9450-5.

[3] Scott Chacon and Ben Straub. Pro git. https://progit.org/.

[4] Inc Cloud9 IDE. Cloud9. https://c9.io/.

[5] Larman Craig. *UML y patrones: introducción al análisis y diseño orientado a objetos*. Editorial Prentice Hall, ISBN: 970-17-0261-1.

[6] Rafael de las Heras del Dedo, Carmen Lasa Gómez, and Alonso Álvarez García. *Métodos ágiles y Scrum*. Editorial Anaya Multimedia, ISBN: 9788441531048.

[7] Smart Technologies Development. Gerapp. http://smarttechnologiesdevelopment.com/.

[8] eMarketer Inc. emarketer. https://www.emarketer.com/.

[9] HighCharts. Highcharts. http://www.highcharts.com/.

[10] Apple Inc. App store. http://www.apple.com/itunes/.

[11] Google Inc. Google maps. https://www.google.es/maps.

[12] Google Inc. Google play. https://play.google.com/.

[13] Twitter Inc. Twitter. https://www.twitter.com/.

[14] Whatsapp Inc. Whatsapp. https://www.whatsapp.com/.

[15] MailChimp. Mandrill. https://mandrill.zendesk.com.

[16] R. McGlaughlin. *Some Notes on Program Design, Software Engieneering Notes, vol. 16*. págs 53-54.

[17] Glenford Myers. *The Art of Software Testing*. 2012.

[18] OpenWall. Phpass. http://www.openwall.com/phpass/.

[19] R.S. Pressman. *Ingeniería del Software: Un enfoque práctico*. (6ª edición), Editorial McGraw-Hill, 2006.

[20] Independent regulator and competition authority for the UK communications industries. Internet use by device. http://stakeholders.ofcom.org.uk/.

[21] Excelia SL. Sanyres. www.excelia.com.

[22] Wappa Software Engineering SL. Wappa. https://www.wappa.net/.

[23] Sommerville. *I. Ingeniería de software*. Editorial Addison Wesley, 2002.

[24] Wikipedia. Don't repeat yourself. https://en.wikipedia.org/.

[25] Wikipedia. Ebitda (artículo). https://es.wikipedia.org/wiki/Ebitda.