ESCUELA UNIVERSITARIA DE INGENIERÍA TÉCNICA INDUSTRIAL DE BILBAO

GRADO EN INGENIERÍA INFORMÁTICA DE GESTIÓN Y SISTEMAS DE INFORMACIÓN

TRABAJO FIN DE GRADO

2014 / 2015

COOKING STARDUST

MEMORIA

DATOS DE LA ALUMNA O DEL ALUMNO	**DATOS DEL DIRECTOR O DE LA DIRECTORA**
NOMBRE: SARAH	NOMBRE: MIKEL
APELLIDOS: MARCIEL MARTÍN	APELLIDOS: VILLAMAÑE GIRONÉS
	DEPARTAMENTO: LENGUAJES Y SISTEMAS INFORMÁTICOS
FDO.:	FDO.:
FECHA:	FECHA:

RESUMEN

En este documento se recoge toda la documentación asociada al desarrollo de la aplicación para dispositivos móviles Android, Cooking Stardust. La experiencia ha sido satisfactoria, a pesar de las frustraciones pasadas, al conseguir cumplir con cada uno de los objetivos establecidos hace más de un año, y se ha obtenido como resultado una aplicación interesante, mejorable, una combinación entre una red social y una solución básica de recetas de cocina, en la que funcionalidades como el reconocimiento de voz y la búsqueda por localización la intentan diferencian de otras similares.

Asimismo, se ha logrado alcanzar unas habilidades en el desarrollo de aplicaciones móviles para Android, así como en herramientas tan innovadoras como lo son AppEngine y su Datastore, que me permitirán realizar otros proyectos futuros más allá de mejorar el presente.

A pesar de no haber cumplido con los plazos establecidos en la planificación principal, la sensación general es de satisfacción al haber conseguido por fin realizar un trabajo de semejante magnitud por cuenta propia.

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1. INTRODUCCIÓN

Algo tan rutinario como decidir qué cocinar día tras día puede llegar a ser desesperante, sobre todo si se quiere mantener una dieta más o menos equilibrada y sin excesos. Muchas veces la salida más fácil y la más cómoda es ayudarse de establecimientos de comida rápida o de paquetes pre-cocinados, pero en realidad, no es la mejor solución.

Debido a ello, se ha pensado en una aplicación para móviles Android. La elección de este sistema operativo por encima de otros sistemas, como iOS o Windows Phone, está basada en la facilidad de desarrollar para Android: las herramientas son gratuitas y fácilmente integrables y el lenguaje en el que se programa es Java, del cual ya se tienen unas nociones más que básicas, a diferencia de Objective -C o C++ en el caso de iOS y Windows Phone respectivamente. De esta manera, nace Cooking Stardust, una aplicación Android de recetas de cocina.

Asimismo, con sólo introducir algunos de los ingredientes que haya en la nevera, se abrirá todo un abanico de posibilidades culinarias, teniendo en cuenta el estado de salud (diabético, celíaco...) o simplemente los gustos a la hora de sentarse a la mesa. Una forma cómoda y sencilla de intentar facilitar una de las tareas diarias más difíciles y que tantos quebraderos de cabeza puede causar.

El trabajo fin de grado consistirá en una aplicación móvil que permita al usuario final elegir entre los diferentes platos de comida que se le ofrecen, habiendo introducido previamente unos ingredientes, o teniendo en cuenta simplemente, la localización del mismo, con el fin de enseñar y facilitar los platos típicos de una zona en concreto.

Una vez elegidos los ingredientes, también se tendrá la posibilidad de elegir entre diferentes dietas o gustos, como pueden ser dietas para diabéticos o celiacos, o simplemente elegir entre cocina tailandesa o española, entre varias. Además de mostrársele las diferentes recetas de comida que cumplan con lo introducido, así como la dificultad y tiempos de elaboración, también tendrá la opción de puntuar esa receta, ver las puntuaciones obtenidas por otros usuarios, calcular una lista de la compra para dicha receta, consultar las recetas cercanas a su ubicación, o incluso, activar el "modo cocina", mediante el cual el usuario podrá interactuar con la aplicación gracias al reconocimiento de voz para ejecutar cada uno de los pasos de la receta sin tener que utilizar las manos. Para acceder a todas estas funcionalidades, el usuario deberá registrarse en el sistema.

De este modo, el usuario será capaz de definir su perfil puntuando las recetas que vaya probando y el sistema será capaz de recomendarle recetas similares a sus gustos. También

se dispondrá la opción de poder seguir las recetas de otros usuarios, a pesar de que los gustos de ambos no coincidan.

2. PLANTEAMIENTO INICIAL

2.1 Objetivos:

El propósito del trabajo es crear una aplicación para móviles Android que facilite al usuario la elección de qué plato cocinar según lo que se disponga en la nevera. De este modo, los objetivos serán:

- Aprender a desarrollar aplicaciones para el sistema operativo Android.
- Aprender a utilizar la herramienta Appengine de Google, así como el lenguaje Python.
- Conseguir que la aplicación sea útil para el usuario final. Es decir, que pueda hacer uso de ella y que considere que las funcionalidades implementadas le son interesantes y adecuadas para una aplicación de recetas de comida.
- Que la aplicación permita al usuario el día a día a la hora de escoger las recetas más adecuadas.

2.2 Arquitectura

La aplicación tiene una arquitectura cliente/servidor, ya que la aplicación estará almacenada en el dispositivo móvil, como se detalla en la ilustración, pero la base de datos estará en un servidor, ya que no sólo va a acceder el usuario a su perfil, sino que además, tendrá la opción de que otros usuarios puedan acceder y ver sus recetas.

Como se muestra en la ilustración, y para este proyecto en concreto, se utilizará la herramienta App Engine que ofrece Google y que será el encargado de alojar la datastore. App Engine será el encargado de permitir la interacción entre la datastore y el cliente Android mediante una API backend, es decir, una API creada en el App Engine.

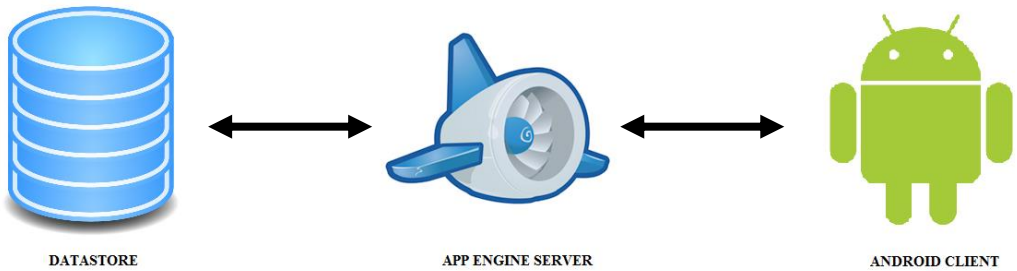


Ilustración 1: Arquitectura

2.3 Herramientas

Se utilizarán las siguientes herramientas para el desarrollo del trabajo:

2.3.1 Hardware:

- Portátil.
- Samsung Galaxy Mini III con sistema operativo Android 4.1.2.

2.3.2 Software:

- **Android Studio**: se trata de una herramienta de desarrollo íntegramente orientada a desarrollar aplicaciones Android. Ofrece infinidad de opciones configurables, que ayudan al desarrollo.
- **Gliffy**: se trata de una herramienta online para el desarrollo de los diagramas necesarios en el proyecto.
- **Gantt Project**: es un programa enfocado a la ayuda para la planificación de las diferentes tareas del proyecto a la hora de representarlas gráficamente.
- **Microsoft Office**: se trata de una herramienta de ofimática útil para la realización de la memoria y documentación.
- **Google App Engine**: es una plataforma para construir y ejecutar aplicaciones alojadas en la infraestructura de Google, gracias a las facilidades que brinda: facilidad de manejar y mantener, escalabilidad, almacenamiento…
- **Sublime Text 3**: se trata de un editor de textos y de código fuente en el cual se utilizado para implementar en Python toda la lógica del servidor.
- **GIMP**: es un programa gratuito de edición de imágenes y dibujos mediante el cual se han realizado todos los iconos o diseños utilizados en la aplicación.

2.4 Alcance

2.4.1 Fases del proyecto

- **Gestión**: es el primer paso a la hora de realizar un proyecto y aquí se definirá el proyecto en sí. Para ello, habrá que buscar información y definir las tareas que lo compondrán.

- **Análisis**: en este punto se definirán las funcionalidades del proyecto, así como las herramientas necesarias para su realización.

- **Diseño**: aquí se definirá la estructura de la aplicación y sus interfaces, cumpliendo con lo acordado en los puntos de gestión y análisis. Además, se diseñará el algoritmo de la aplicación para su posterior desarrollo.

- **Implementación**: se definirán los diferentes prototipos en los cuales se va a dividir el algoritmo y se realizará la implementación. Se dividirá en nueve prototipos:
    1. Guardar recetas nuevas.
    2. Buscar recetas (teniendo en cuenta todos los parámetros de búsqueda).
    3. Buscar recetas teniendo en cuenta la ubicación.
    4. Crear lista de la compra.
    5. Consultar recetas usando el reconocimiento de voz.
    6. Registrarse en el sistema (dar de alta un usuario, modificar perfil).
    7. Consultar perfiles de otros usuarios.
    8. Puntuar recetas.
    9. Recomendar recetas al usuario y a otros usuarios con gustos similares.

- **Plan de pruebas**: se realizará un plan de pruebas por cada prototipo implementado. De este modo, al terminar un prototipo, se harán las pruebas pertinentes para asegurar su correcto funcionamiento. Al final de toda la implementación, se harán las pruebas finales, con todos los módulos incluidos.

- **Documentación**: será el último paso a seguir, donde se reunirá toda la documentación y experiencias tenidas a lo largo de todo el proyecto.

2.4.2 Esquema de Descomposición del proyecto

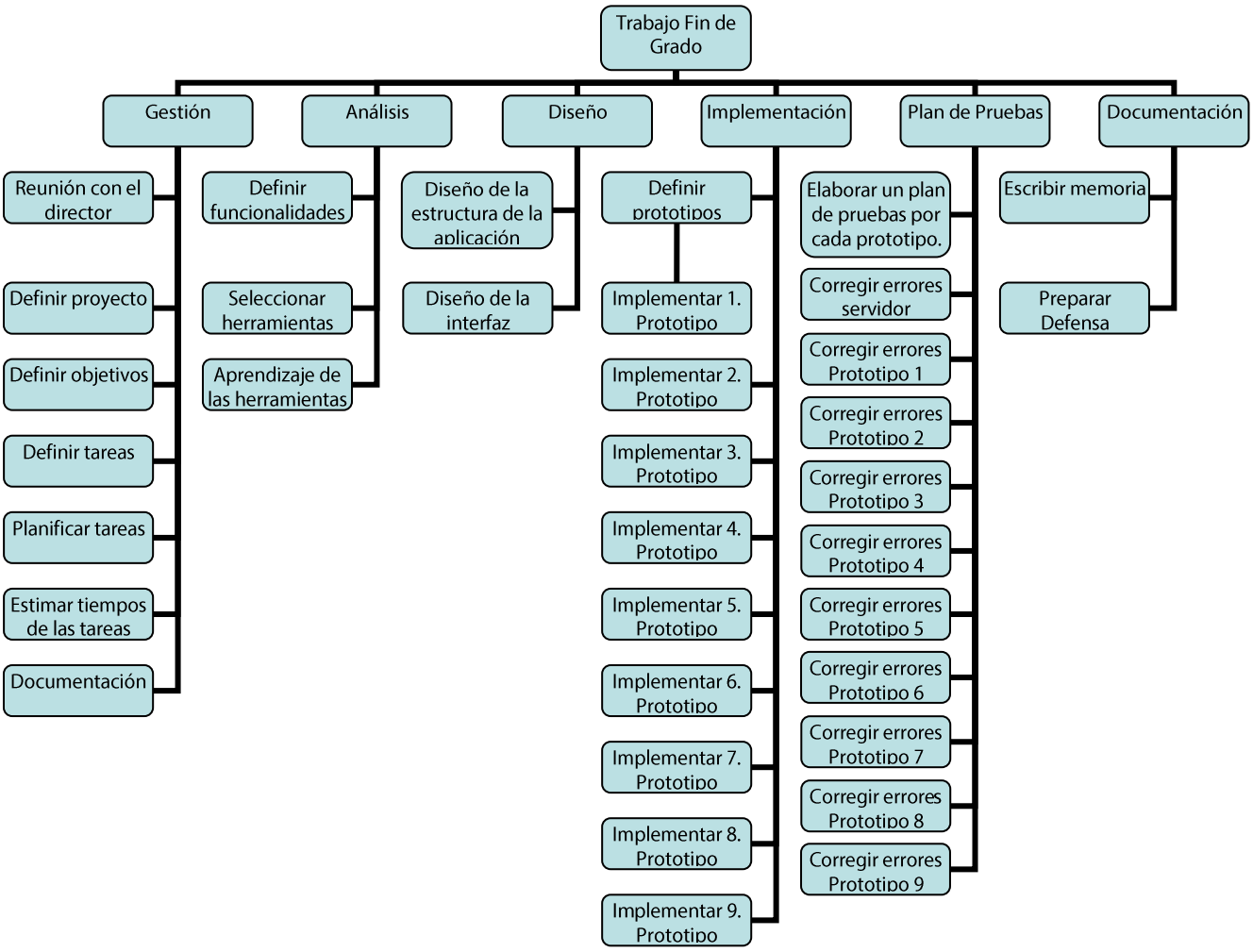


Ilustración 2: EDT

2.4.3 Tareas:

Se definirán más detalladamente las tareas especificadas en el EDT.

2.4.3.1 Gestión

2.4.3.1.1 Paquete de trabajo: [Reunión con el director]

Duración: [1 hora]

Descripción

Realizar una descripción clara de lo que va a ser el trabajo, qué funcionalidades se van a implementar…

Salidas/Entregables

La descripción formal del trabajo.

Recursos necesarios

Saber qué es lo que se quiere que haga la aplicación.

2.4.3.1.2 Paquete de trabajo: [Definir objetivos]

Duración: [2 horas]

Descripción

Establecer los objetivos que se van a llevar a cabo en el trabajo fin de grado.

Entrada

La descripción del trabajo.

Salidas/Entregables

Los objetivos del trabajo.

Recursos necesarios

Tener una idea clara de los objetivos que se quieren cumplir.

2.4.3.1.3 Paquete de trabajo: [Definir tareas]

Duración: [3 horas]

Descripción

Establecer las diferentes tareas que van a ser necesarias para la realización del trabajo.

Salidas/Entregables

La descripción de las tareas necesarias.

Recursos necesarios

Tener claras las diferentes fases del proyecto.

2.4.3.1.4 Paquete de trabajo: [Planificar tareas]

Duración: [2 horas]

Descripción

Establecer el orden en el que se van a llevar a cabo las tareas y en cuáles se van a subdividir.

Entrada

La descripción de las tareas a planificar.

Salidas/Entregables

La planificación de las tareas, sin los tiempos.

Recursos necesarios

Tener claras las tareas que se van a llevar a cabo.

Precedencias:

Definir tareas.

2.4.3.1.5 Paquete de trabajo: [Estimar tiempos de las tareas]

Duración: [4 horas]

Descripción

Realizar una estimación aproximada del tiempo que se va a necesitar emplear para la realización de las tareas.

Entrada

La planificación y descripción de las tareas.

Salidas/Entregables

La planificación de las tareas con los tiempos estimados.

Recursos necesarios

Tener claras las tareas que se van a llevar a cabo.

Precedencias:

Planificación de las tareas.

2.4.3.1.6 Paquete de trabajo: [Documentación]

Duración: [8 horas]

Descripción

Reunir toda la documentación que se haya creado en la fase de Gestión: las tareas, las estimación de los tiempos de las tareas, la viabilidad económica del proyecto, el Gantt, la descripción del proyecto…

Salidas/Entregables

El documento de objetivos del proyecto.

Recursos necesarios

Todo lo necesario para llevar a cabo el DOP.

Precedencias:

Todas las tareas anteriores de la fase Gestión.

2.4.3.2 Análisis

2.4.3.2.1 Paquete de trabajo: [Definir funcionalidades]

Duración: [15 horas]

Descripción

Definir las funcionalidades que se van a implementar en a aplicación.

Salidas/Entregables

La documentación relacionada a las funcionalidades

Recursos necesarios

Saber qué va a realizar el proyecto.

Precedencias:

Reunión con el director.

2.4.3.2.2 Paquete de trabajo: [Seleccionar herramientas]

Duración: [4 horas]

Descripción

Elegir las diferentes herramientas que se van a emplear para el desarrollo del proyecto, así como la búsqueda de tutoriales que faciliten el aprendizaje.

Recursos necesarios

Saber qué esperamos de las herramientas que queremos emplear.

2.4.3.2.3 Paquete de trabajo: [Aprendizaje de las herramientas]

Duración: [12 horas]

Descripción

Practicar con las herramientas seleccionadas con el fin de tener mayor soltura y de cara a la implementación (en el caso de las herramientas de desarrollo) o a la documentación (en el caso de las herramientas a emplear para realizar la memoria).

Salidas/Entregables

Diferentes tutoriales sobre las herramientas seleccionadas.

Recursos necesarios

Las herramientas.

Precedencias:

Seleccionar herramientas.

2.4.3.3 Diseño

2.4.3.3.1 Paquete de trabajo: [Diseño de la arquitectura de la aplicación]

Duración: [25 horas]

Descripción

Realizar un diseño de la arquitectura que va a disponer la aplicación, así como el diseño de un algoritmo que cumpla con todas las funcionalidades especificadas.

Salidas/Entregables

Un algoritmo general.

Recursos necesarios

Tener claras las funcionalidades a diseñar.

Precedencias:

Análisis de las funcionalidades.

2.4.3.3.2 Paquete de trabajo: [Diseño de las interfaces]

Duración: [15 horas]

Descripción

Realizar un diseño de las interfaces donde se muestre lo que va a realizar la aplicación.

Salidas/Entregables

El diseño de las interfaces.

Recursos necesarios

Tener claro qué es lo que se quiere que haga la aplicación.

Precedencias:

Análisis de las funcionalidades.

2.4.3.4 Implementación

2.4.3.4.1 Paquete de trabajo: [Definir prototipos]

Duración: [10 horas]

Descripción

La aplicación se dividirá en diferentes prototipos para facilitar su implementación, y cada uno de ellos irá añadiendo funcionalidades al anterior hasta construir toda la aplicación

Salidas/Entregables

Los algoritmos de los diferentes prototipos.

Recursos necesarios

Tener claras las funcionalidades.

Precedencias:

Análisis y diseño.

2.4.3.4.2 Paquete de trabajo: [Implementar Prototipo 1 – Guardar recetas]

Duración: [40 horas]

Descripción

Se implementará el primer prototipo "Guardar recetas nuevas".

Salidas/Entregables

El código implementado.

Recursos necesarios

El prototipo definido.

Precedencias:

Análisis, diseño.

2.4.3.4.3 Paquete de trabajo: [Implementar Prototipo 2 –Buscar recetas]

Duración: [30 horas]

Descripción

Se implementará el primer prototipo "Buscar recetas".

Salidas/Entregables

El código implementado.

Recursos necesarios

El prototipo definido.

Precedencias:

Análisis, diseño.

2.4.3.4.4 Paquete de trabajo: [Implementar Prototipo 3 – Buscar recetas por localización]

Duración: [20 horas]

Descripción

Se implementará el primer prototipo "Buscar recetas teniendo en cuenta la ubicación".

Salidas/Entregables

El código implementado.

Recursos necesarios

El prototipo definido.

Precedencias:

Análisis, diseño.

2.4.3.4.5 Paquete de trabajo: [Implementar Prototipo 4 – Lista de la compra]

Duración: [20 horas]

Descripción

Se implementará el primer prototipo “Crear lista de la compra”.

Salidas/Entregables

El código implementado.

Recursos necesarios

El prototipo definido.

Precedencias:

Análisis, diseño.

2.4.3.4.6 Paquete de trabajo: [Implementar Prototipo 5 – Consultar recetas usando el reconocimiento de voz]

Duración: [30 horas]

Descripción

Se implementará el primer prototipo “Consultar recetas usando el reconocimiento de voz”.

Salidas/Entregables

El código implementado.

Recursos necesarios

El prototipo definido.

Precedencias:

Análisis, diseño.

2.4.3.4.7 Paquete de trabajo: [Implementar Prototipo 6 – Registrarse en el sistema]

Duración: [25 horas]

Descripción

Se implementará el primer prototipo “Registrarse en el sistema”.

Salidas/Entregables

El código implementado.

Recursos necesarios

El prototipo definido.

Precedencias:

Análisis, diseño.

2.4.3.4.8 Paquete de trabajo: [Implementar Prototipo 7 – Consultar perfil]

Duración: [30 horas]

Descripción

Se implementará el primer prototipo "Consultar perfiles de otros usuarios".

Salidas/Entregables

El código implementado.

Recursos necesarios

El prototipo definido.

Precedencias:

Análisis, diseño.

2.4.3.4.9 Paquete de trabajo: [Implementar Prototipo 8 – Puntuar recetas]

Duración: [30 horas]

Descripción

Se implementará el primer prototipo "Puntuar recetas".

Salidas/Entregables

El código implementado.

Recursos necesarios

El prototipo definido.

Precedencias:

Análisis, diseño.

2.4.3.4.10 Paquete de trabajo: [Implementar Prototipo 9 – Recomendar recetas]

Duración: [50 horas]

Descripción

Se implementará el primer prototipo "Recomendar recetas".

Salidas/Entregables

El código implementado.

Recursos necesarios

El prototipo definido.

Precedencias:

Análisis, diseño.

2.4.3.5 Plan de pruebas

2.4.3.5.1 Paquete de trabajo: [Elaborar un plan de pruebas por cada prototipo]

Duración: [15 horas]

Descripción

Desarrollar un plan de pruebas que se ajuste a lo implementado en cada prototipo y cuyo objetivo sea cubrir todas las posibilidades de error.

Entrada

El código implementado.

Salidas/Entregables

Documentación sobre las pruebas realizadas.

Recursos necesarios

El código implementado y tener claro qué es lo que se necesita probar.

Precedencias:

Implementación.

2.4.3.5.2 Paquete de trabajo: [Corregir errores Prototipo 1]

Duración: [5 horas]

Descripción

Corregir todos los errores que se hayan detectado en el prototipo 1.

Salidas/Entregables

Documentación sobre cómo se han resuelto los errores.

Recursos necesarios

El código implementado.

Precedencias:

Implementación y el Plan de Pruebas.

2.4.3.5.3 Paquete de trabajo: [Corregir errores Prototipo 2]

Duración: [5 horas]

Descripción

Corregir todos los errores que se hayan detectado en el prototipo 2.

Salidas/Entregables

Documentación sobre cómo se han resuelto los errores.

Recursos necesarios

El código implementado.

Precedencias:

Implementación y el Plan de Pruebas.

2.4.3.5.4 Paquete de trabajo: [Corregir errores Prototipo 3]

Duración: [5 horas]

Descripción

Corregir todos los errores que se hayan detectado en el prototipo 3.

Salidas/Entregables

Documentación sobre cómo se han resuelto los errores.

Recursos necesarios

El código implementado.

Precedencias:

Implementación y el Plan de Pruebas.

2.4.3.5.5 Paquete de trabajo: [Corregir errores Prototipo 4]

Duración: [5 horas]

Descripción

Corregir todos los errores que se hayan detectado en el prototipo 4.

Salidas/Entregables

Documentación sobre cómo se han resuelto los errores.

Recursos necesarios

El código implementado.

Precedencias:

Implementación y el Plan de Pruebas.

2.4.3.5.6 Paquete de trabajo: [Corregir errores Prototipo 5]

Duración: [5 horas]

Descripción

Corregir todos los errores que se hayan detectado en el prototipo 5.

Salidas/Entregables

Documentación sobre cómo se han resuelto los errores.

Recursos necesarios

El código implementado.

Precedencias:

Implementación y el Plan de Pruebas.

2.4.3.5.7 Paquete de trabajo: [Corregir errores Prototipo 6]

Duración: [5 horas]

Descripción

Corregir todos los errores que se hayan detectado en el prototipo 6.

Salidas/Entregables

Documentación sobre cómo se han resuelto los errores.

Recursos necesarios

El código implementado.

Precedencias:

Implementación y el Plan de Pruebas.

2.4.3.5.8 Paquete de trabajo: [Corregir errores Prototipo 7]

Duración: [5 horas]

Descripción

Corregir todos los errores que se hayan detectado en el prototipo 7.

Salidas/Entregables

Documentación sobre cómo se han resuelto los errores.

Recursos necesarios

El código implementado.

Precedencias:

Implementación y el Plan de Pruebas.

2.4.3.5.9 Paquete de trabajo: [Corregir errores Prototipo 8]

Duración: [5 horas]

Descripción

Corregir todos los errores que se hayan detectado en el prototipo 8.

Salidas/Entregables

Documentación sobre cómo se han resuelto los errores.

Recursos necesarios

El código implementado.

Precedencias:

Implementación y el Plan de Pruebas.

2.4.3.5.10 Paquete de trabajo: [Corregir errores Prototipo 9]

Duración: [5 horas]

Descripción

Corregir todos los errores que se hayan detectado en el prototipo 9.

Salidas/Entregables

Documentación sobre cómo se han resuelto los errores.

Recursos necesarios

El código implementado.

Precedencias:

Implementación y el Plan de Pruebas.

2.4.3.6 Documentación

2.4.3.6.1 Paquete de trabajo: [Escribir memoria]

Duración: [20 horas]

Descripción

Escribir la memoria del proyecto, donde se incluirán toda la documentación reunida, así como todas las tareas, diseños, plan de pruebas y experiencias.

Salidas/Entregables

La memoria.

Recursos necesarios

El proyecto y la documentación recogida a lo largo de todo el proyecto.

Precedencias:

Todas las fases anteriores.

2.4.3.6.2 Paquete de trabajo: [Preparar defensa]

Duración: [10 horas]

Descripción

Preparar la defensa del proyecto.

Salidas/Entregables

La defensa.

Recursos necesarios

La memoria y el proyecto.

Precedencias:

Todas las fases anteriores.

2.4.4 Planificación temporal

Se establecen cuántas horas estimadas durará el proyecto y cuantas semanas serán necesarias para acabarlo. Además, se dispondrá de un gráfico Gantt para representar la planificación temporal a lo largo de las semanas.

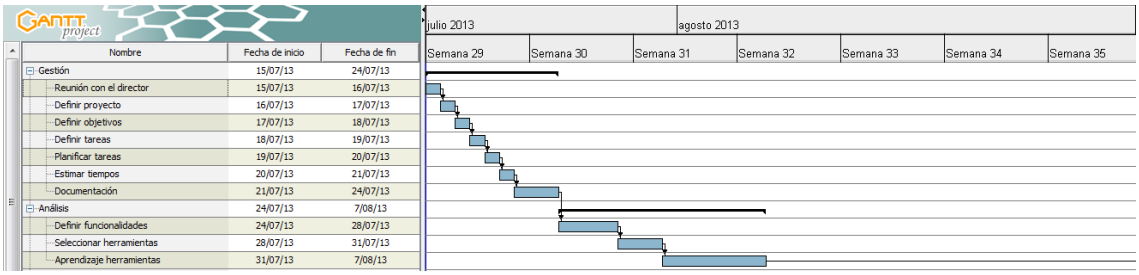
Tareas	Estimación de horas a dedicar
1. Gestión	**22 horas**
1.1 Reunión con el director	1 hora
1.2 Definir proyecto	2 horas
1.3 Definir objetivos	2 horas
1.4 Definir tareas	3 horas
1.6 Planificar tareas	2 horas
1.7 Estimar tiempos de tareas	4 horas
1.8 Documentación	8 horas
2. Análisis	**31 horas**
2.1 Definir funcionalidades	15 horas
2.2 Seleccionar herramientas	4 horas
2.3 Aprendizaje de herramientas	12 horas
3. Diseño	**40 horas**
3.1 Diseñar estructura de la aplicación	25 horas
3.2 Diseñar interfaces	15 horas
4. Implementación	**275 horas**
4.1 Definir prototipos	10 horas
4.2 Implementar prototipo 1	40 horas
4.3 Implementar prototipo 2	30 horas
4.4 Implementar prototipo 3	20 horas
4.5 Implementar prototipo 4	20 horas
4.6 Implementar prototipo 5	30 horas
4.7 Implementar prototipo 6	25 horas
4.8 Implementar prototipo 7	30 horas
4.9 Implementar prototipo 8	30 horas
4.10 Implementar prototipo 9	50 horas
5. Plan de pruebas	**25 horas**
5.1 Realizar un plan de pruebas	15 horas
5.2 Corregir errores servidor	10 horas
5.3 Prototipo 1	5 horas
5.4 Prototipo 2	5 horas
5.5 Prototipo 3	5 horas
5.6 Prototipo 4	5 horas
5.7 Prototipo 5	5 horas
5.8 Prototipo 6	5 horas
5.9 Prototipo 7	5 horas

5.10 Prototipo 8	5 horas
5.11 Prototipo 9	5 horas
6. Documentación	**30 horas**
6.1 Escribir memoria	20 horas
6.2 Preparar defensa	10 horas
Total	**468 horas**

Tabla 1: Planificación en horas

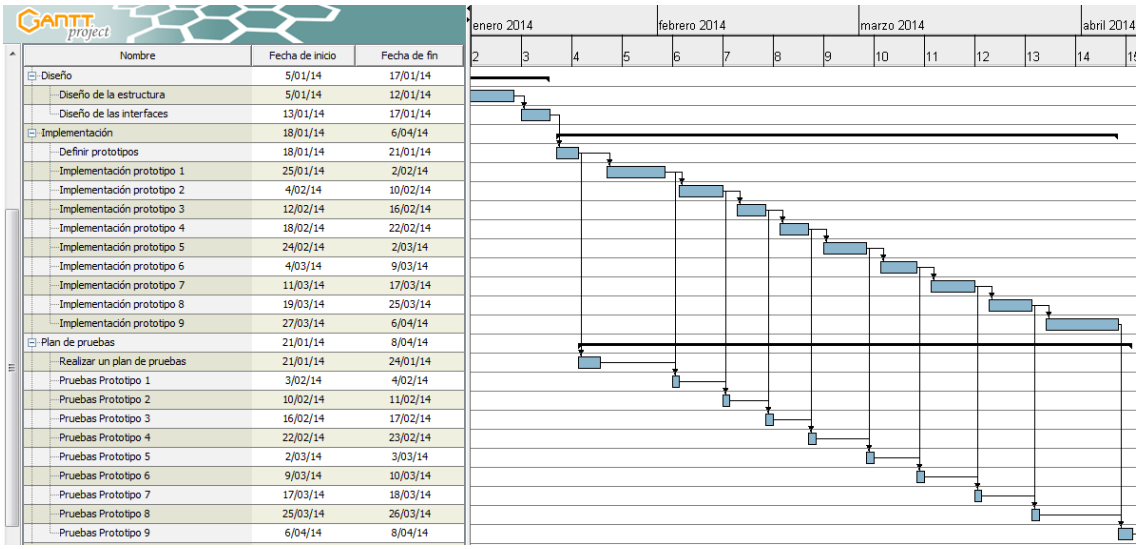
Estableciendo un promedio de 30 horas semanales dedicadas, el proyecto debería estar acabado en 16 semanas.

Teniendo en cuenta que el proyecto se realizará tanto sábados, domingos y otros días festivos, se ha realizado un Gantt en el que se muestra de manera detallada como quedaría la planificación organizada a través de las semanas, hasta la última semana de Abril.



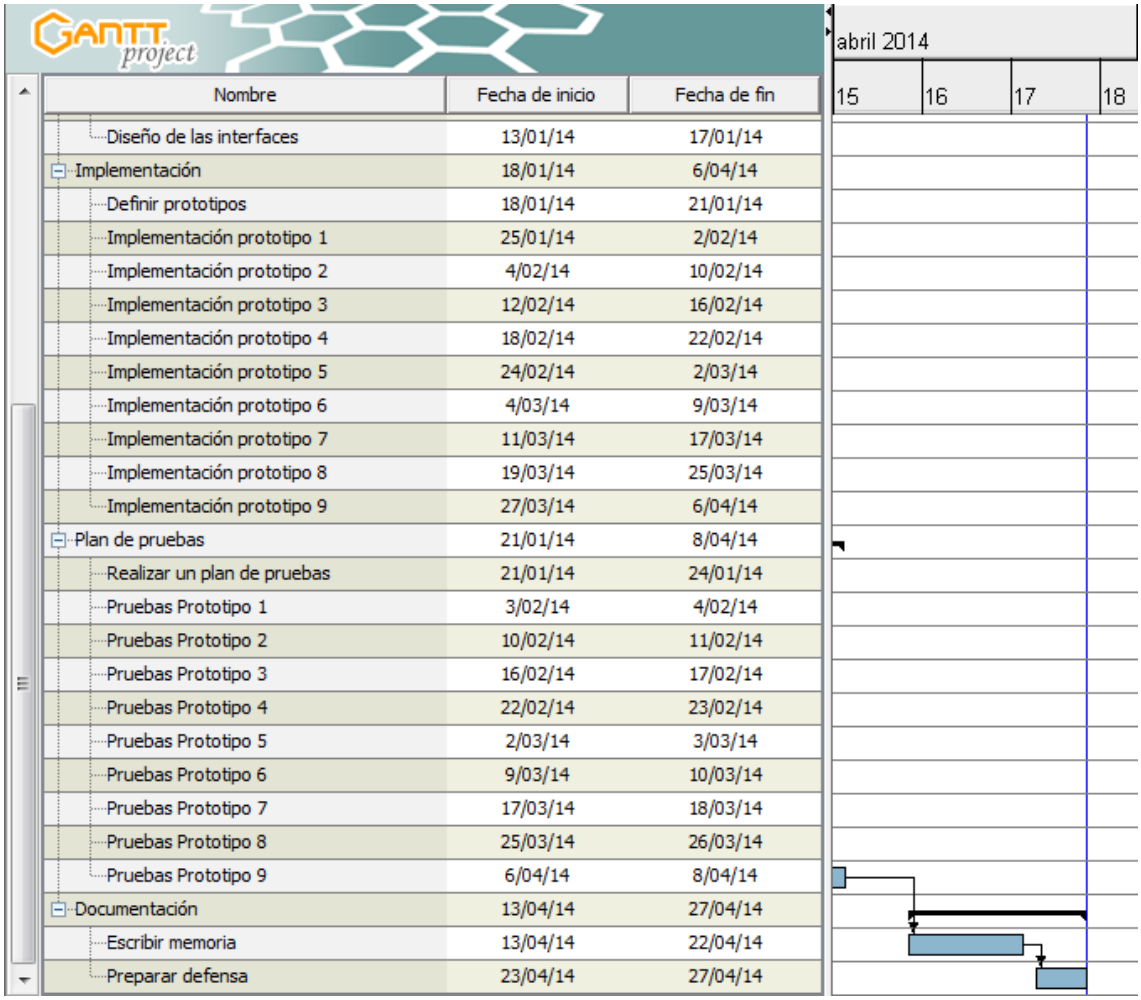
Nombre	Fecha de inicio	Fecha de fin
Gestión	15/07/13	24/07/13
Reunión con el director	15/07/13	16/07/13
Definir proyecto	16/07/13	17/07/13
Definir objetivos	17/07/13	18/07/13
Definir tareas	18/07/13	19/07/13
Planificar tareas	19/07/13	20/07/13
Estimar tiempos	20/07/13	21/07/13
Documentación	21/07/13	24/07/13
Análisis	24/07/13	7/08/13
Definir funcionalidades	24/07/13	28/07/13
Seleccionar herramientas	28/07/13	31/07/13
Aprendizaje herramientas	31/07/13	7/08/13

Ilustración 3: Gantt - 1



Nombre	Fecha de inicio	Fecha de fin
Diseño	5/01/14	17/01/14
Diseño de la estructura	5/01/14	12/01/14
Diseño de las interfaces	13/01/14	17/01/14
Implementación	18/01/14	6/04/14
Definir prototipos	18/01/14	21/01/14
Implementación prototipo 1	25/01/14	2/02/14
Implementación prototipo 2	4/02/14	10/02/14
Implementación prototipo 3	12/02/14	16/02/14
Implementación prototipo 4	18/02/14	22/02/14
Implementación prototipo 5	24/02/14	2/03/14
Implementación prototipo 6	4/03/14	9/03/14
Implementación prototipo 7	11/03/14	17/03/14
Implementación prototipo 8	19/03/14	25/03/14
Implementación prototipo 9	27/03/14	6/04/14
Plan de pruebas	21/01/14	8/04/14
Realizar un plan de pruebas	21/01/14	24/01/14
Pruebas Prototipo 1	3/02/14	4/02/14
Pruebas Prototipo 2	10/02/14	11/02/14
Pruebas Prototipo 3	16/02/14	17/02/14
Pruebas Prototipo 4	22/02/14	23/02/14
Pruebas Prototipo 5	2/03/14	3/03/14
Pruebas Prototipo 6	9/03/14	10/03/14
Pruebas Prototipo 7	17/03/14	18/03/14
Pruebas Prototipo 8	25/03/14	26/03/14
Pruebas Prototipo 9	6/04/14	8/04/14

Ilustración 4: Gantt - 2



Nombre	Fecha de inicio	Fecha de fin
Diseño de las interfaces	13/01/14	17/01/14
Implementación	18/01/14	6/04/14
Definir prototipos	18/01/14	21/01/14
Implementación prototipo 1	25/01/14	2/02/14
Implementación prototipo 2	4/02/14	10/02/14
Implementación prototipo 3	12/02/14	16/02/14
Implementación prototipo 4	18/02/14	22/02/14
Implementación prototipo 5	24/02/14	2/03/14
Implementación prototipo 6	4/03/14	9/03/14
Implementación prototipo 7	11/03/14	17/03/14
Implementación prototipo 8	19/03/14	25/03/14
Implementación prototipo 9	27/03/14	6/04/14
Plan de pruebas	21/01/14	8/04/14
Realizar un plan de pruebas	21/01/14	24/01/14
Pruebas Prototipo 1	3/02/14	4/02/14
Pruebas Prototipo 2	10/02/14	11/02/14
Pruebas Prototipo 3	16/02/14	17/02/14
Pruebas Prototipo 4	22/02/14	23/02/14
Pruebas Prototipo 5	2/03/14	3/03/14
Pruebas Prototipo 6	9/03/14	10/03/14
Pruebas Prototipo 7	17/03/14	18/03/14
Pruebas Prototipo 8	25/03/14	26/03/14
Pruebas Prototipo 9	6/04/14	8/04/14
Documentación	13/04/14	27/04/14
Escribir memoria	13/04/14	22/04/14
Preparar defensa	23/04/14	27/04/14

Ilustración 5: Gantt - 5

2.4.5 Gestión de riesgos

Todo proyecto tiene unos riesgos y conviene identificarlos para poder evitar las causas, o en caso de que ocurran, poder actuar con rapidez.

2.4.5.1 Problemas a la hora de la implementar el algoritmo.

- ***Probabilidad***: alta.
- ***Impacto***: medio.
- ***Consecuencia***: retraso en las tareas siguientes a la implementación e incluso la fecha de entrega del proyecto.
- ***Prevención***: dedicar el tiempo necesario a diseñar un algoritmo que sea inteligible y fácil de implementar, así como el aprendizaje previo de las herramientas y lenguajes a utilizar.
- ***Plan de contingencia***: dedicar más horas a la implementación y a las tareas siguientes con el fin de no retrasar demasiado la fecha del proyecto.

2.4.5.2 Problemas a la hora de la implementar el prototipo "Cocinar" por cambios en la API de reconocimiento de voz

- ***Probabilidad***: media.
- ***Impacto***: medio.
- ***Consecuencia***: retraso en las tareas siguientes a la implementación e incluso la fecha de entrega del proyecto.
- ***Prevención***: utilizar funciones básicas de la API, que tengan menos probabilidades de ser modificadas o eliminadas en una actualización de la misma.
- ***Plan de contingencia***: dedicar más horas a la comprensión y el funcionamiento de la API actualizada, en caso de que el prototipo implementado sea inservible debido a las actualizaciones. Además, dedicar más horas a la implementación del resto de prototipos, con el fin de no retrasar demasiado la fecha del proyecto.

2.4.5.3 Problemas a la hora de la implementar el prototipo "Buscar recetas por ubicación" por cambios en la API de geolocalización.

- ***Probabilidad***: media.
- ***Impacto***: medio.
- **Consecuencia**: retraso en las tareas siguientes a la implementación e incluso la fecha de entrega del proyecto.
- ***Prevención***: utilizar funciones básicas de la API, que tengan menos probabilidades de ser modificadas o eliminadas en una actualización de la misma.
- ***Plan de contingencia***: dedicar más horas a la comprensión y el funcionamiento de la API actualizada, en caso de que el prototipo implementado sea inservible debido a las actualizaciones. Además, dedicar más horas a la implementación del resto de prototipos, con el fin de no retrasar demasiado la fecha del proyecto.

2.4.5.4 Problemas a la hora de la implementar los prototipos por cambios en la API de Android.

- ***Probabilidad***: baja.
- Impacto: medio.
- ***Consecuencia***: retraso en las tareas siguientes a la implementación e incluso la fecha de entrega del proyecto.
- ***Prevención***: utilizar funciones básicas de la API seleccionada (en este caso la API 8), que tengan menos probabilidades de ser modificadas o eliminadas en una actualización de la misma.
- ***Plan de contingencia***: considerar si merece la pena modificar todos los prototipos implementados hasta la fecha para implementar una característica en cuestión. En caso de que la API modificada obligue a cambiar muchas de las funcionalidades a implementar, dedicar las horas necesarias para migrar de API y adecuar lo implementado a la nueva actualización. Además, dedicar más horas a la implementación del resto de prototipos, con el fin de no retrasar demasiado la fecha del proyecto.

2.4.5.5 Problemas a la hora de la implementar los prototipos por cambios en el App Engine o librerías de Python.

- ***Probabilidad***: media.
- ***Impacto***: medio.
- ***Consecuencia***: retraso en las tareas siguientes a la implementación e incluso la fecha de entrega del proyecto.
- ***Prevención***: investigar los cambios de las posibles actualizaciones del App Engine y de las librerías de Python, y en caso de ser obligatorias, actualizar ambas herramientas. Utilizar funciones lo más genéricas posibles en Python.
- ***Plan de contingencia***: en caso de que sea inevitable, adaptar los prototipos implementados a los nuevos cambios. En caso de que alguno de ellos deje de funcionar, buscar soluciones alternativas para su desarrollo e intentar dedicar más horas a realizar las tareas retrasadas.

2.4.5.6 Problemas personales

- ***Probabilidad***: baja.
- ***Impacto***: medio.
- ***Consecuencia***: retraso en las fechas establecidas para las tareas.
- ***Prevención***: no hay ninguna determinada dado que depende del tipo de problema que se tenga.
- ***Plan de contingencia***: intentar dedicar más horas a realizar las tareas retrasadas.

2.4.5.7 Problemas a la hora de diseñar las funcionalidades de la aplicación

- ***Probabilidad***: media.
- ***Impacto***: alto.
- ***Consecuencia***: retraso en las tareas siguientes y problemas a la hora de diseñar el algoritmo.
- ***Prevención***: dedicar las horas necesarias a tener claras y bien diseñadas las funcionalidades del sistema.
- ***Plan de contingencia***: dedicar las horas necesarias para cumplir con los plazos de las siguientes tareas, y en caso de ser necesario, volver a planificar.

2.4.5.8 Problemas de Hardware:

- ***Probabilidad***: baja.
- ***Impacto***: medio.
- ***Consecuencia***: retraso en las fechas establecidas para las tareas.
- ***Prevención***: realizar un mantenimiento ocasional de los componentes de hardware del ordenador, revisión de la memoria, el disco duro... Asimismo, realizar un backup diario de todo lo que se haga y guardarlo en una unidad aparte. Si fuera posible, considerar tener otro ordenador más o menos disponible con el fin de poder continuar parte del trabajo.
- ***Plan de contingencia***: intentar dedicar más horas a realizar las tareas retrasadas, recuperar el backup más reciente para poder avanzar desde esa última copia de seguridad.

2.4.5.9 Problemas de Software:

- ***Probabilidad***: media.
- ***Impacto***: medio.
- ***Consecuencia***: retraso en las fechas establecidas para las tareas.
- ***Prevención***: realizar un mantenimiento ocasional del sistema, así como análisis de antivirus, instalar el menos software nuevo posible... Asimismo, realizar un backup diario de todo lo que se haga y guardarlo en una unidad aparte.
- ***Plan de contingencia***: intentar dedicar más horas a realizar las tareas retrasadas.

2.4.6 Viabilidad económica

En este punto trataremos la viabilidad económica del proyecto, así como su coste total y la forma de recuperar la inversión realizada.

- **Personal**: teniendo en cuenta que un programador cobra 30euros/hora y que el proyecto dura 468 horas, el coste total del personal asciende a **14040 euros**.
- **Hardware**:
    - Ordenador portátil: valorado en 800 euros, con una vida media de 5 años y un uso de un 33% a lo largo del proyecto.
        - Amortización anual: 800/5 = 160 euros.
        - Gastos del portátil: ((160 x 13 semanas) x 0.33) / 52 semanas = **13,20 euros.**
    - Samsung Galaxy Mini III: valorado en 200 euros, con una vida media de 2 años y un uso del 20% a lo largo del proyecto.
        - Amortización anual: 200/2 años = 100 euros.
        - Gastos del móvil: ((100 x 13 semanas) x 0.2) / 52 semanas = **5 euros.**
    - **Gastos totales hardware: 18,20 euros.**

- **Software**:
    - Licencia Windows 7, incluida en el precio del portátil.
    - Licencia Microsoft Office, incluida en el paquete de Windows 7.
    - Eclipse: licencia gratuita.
    - Dia: licencia gratuita.
    - Gantt Project: licencia gratuita.
    - Gliffy: licencia trimestral: 21 euros.
    - Google App Engine: licencia gratuita (con limitaciones).
    - **Gastos totales software: 21 euros.**

- Otros: diversos gastos en fotocopias… En total, **30 euros.**

GASTOS	
Hardware	**18,20 euros**
Portátil	13,20 euros
Móvil	4,00 euros
Software	**21 euros**
Personal	**12690 euros**
Otros	**30 euros**
TOTAL	**12733,2 euros**

Tabla 2: Tabla de gastos

- **Recuperar la inversión**

En este apartado se tratarán las diferentes formas de recuperar la inversión realizada en el proyecto. Si se quiere vender la aplicación en el Android Market, habrá que tener en cuenta que la licencia de venta vale 20€ y que además, la empresa se queda con un 30% de los beneficios netos.
Por probabilidad, cuanto más cara, menos ventas, pero habrá que ver si la diferencia de precio compensa.

- Si el precio de la aplicación es de 1 euro , el número mínimo de compradores para obtener beneficios es el siguiente:
  - Beneficios ((x*1 euro) – (x*0.3)) – 12733,20 – 20 = 0) => **18219**.

- Si el precio de la aplicación asciende a 3 euros, , el número mínimo de compradores para obtener beneficios es el siguiente:
  - Beneficios ((x*3 euro) – (x*0.3)) – 12733,20 – 20 = 0) => **4724.**

La aplicación está orientada a cualquier persona interesada en consultar recetas de comida, por lo cual se podría considerar un público bastante amplio. Aun así, y teniendo en cuenta la cantidad de aplicaciones que se pueden conseguir de manera gratuita, establecer un precio superior a un euro sería demasiado. Por ende, a pesar de que el número de compras es tres veces mayor, el precio de un euro por la aplicación podría considerarse acertado.

3. ANTECEDENTES

Existen varias aplicaciones para dispositivos Android que son similares a la creada en este trabajo, de las cuales merece destacar las siguientes:

- **Canal Cocina**: la productora Chello Multicanal es la responsable de esta aplicación, que permite añadir tus propias recetas además de consultar el catálogo de Canal Cocina. Entre sus funcionalidades: puntuar recetas y ver vídeos del programa.

Ilustración 6: Canal Cocina

- **Recetario**: la aplicación más completa sobre recetas de cocina para plataforma Android, de la mano de Villy tiene ya más de 70000 usuarios. Además de las funcionalidades básicas, añade opciones sociales.

Ilustración 7: Recetario

- **Todas mis recetas**: esta aplicación de Cookpad Spain es la versión Android de la página web mis-recetas.org. Añade también una lista de la compra genérica, así como la posibilidad de compartir las recetas en varias redes sociales.

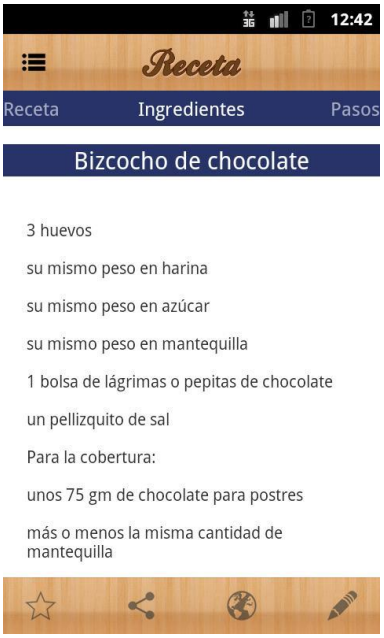


Ilustración 8: Todas mis recetas

- **Qué cocino hoy**: La funcionalidad a destacar de esta aplicación, además de las mencionadas en todas las aplicaciones anteriores y que esta también comparte, es la posibilidad de guardar las recetas en local, para poder consultarlas sin tener que disponer de red obligatoriamente.

Ilustración 9: Qué cocino hoy

- **Recetas Express**: se trata de un buscador de recetas, sin opción de añadir propias, y de la mano de El Corte Inglés y Unilever.

Ilustración 10: Recetas Express

- **Postres**: especializada en repostería y dulces tradicionales, contiene un variado catálogo de postres, así como la opción de poder consultar las recetas offline.

Ilustración 11: Postres

Añadir funcionalidades novedosas a una aplicación sobre recetas de cocina es en realidad, bastante difícil. A pesar de todo, Cooking Stardust proporcionará la opción de “Cocinar”, en un intento de añadir ideas nuevas a este tipo de aplicaciones.

Mediante esta funcionalidad, el usuario será capaz de desplazarse por los diferentes pasos de la elaboración de una receta haciendo uso de su voz, facilitando el llevar a cabo la receta sin tener que estar pendiente de tocar la pantalla táctil con las manos sucias.

Asimismo, y como comparte con otras aplicaciones mencionadas arriba, también tendrá una opción social, donde se dispondrá de un perfil completo de usuarios y listas de recetas, de las que podrá añadir a favoritos, puntuarlas y hasta ordenarlas por tipo de cocina, dieta, o tipo de plato o incluso acceder a recomendaciones personalizadas.

4. CAPTURA DE REQUISITOS

La aplicación final contará con un tipo de usuario, que podrá ser todo aquel que mediante la instalación y el registro en la aplicación podrá hacer uso de todas las funcionalidades definidas.

A continuación se ilustra el modelo de casos de uso asociado al usuario final, en el que se pueden apreciar los diferentes casos de uso y las relaciones entre ellos.

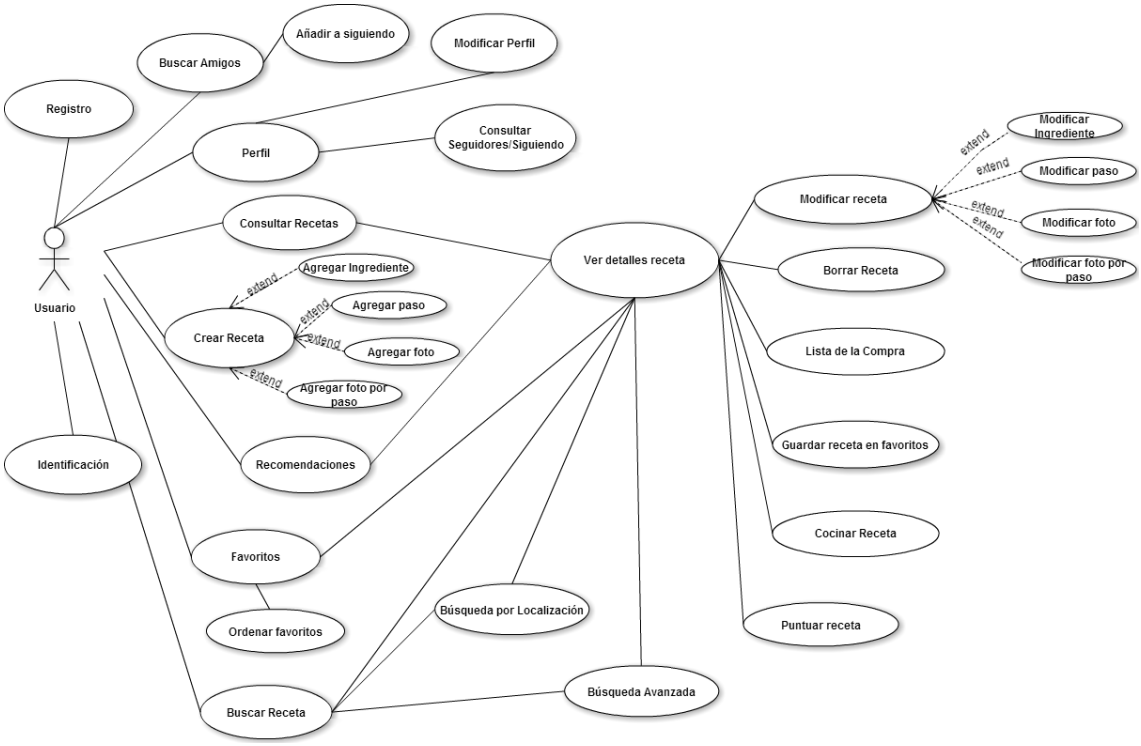


Ilustración 12: Modelo de Casos de Uso – Usuario

4.1 Casos de Uso:

4.1.1 Registro

Permite al usuario darse de alta en la aplicación, de tal modo que pueda acceder a las funcionalidades. Mediante la página de registro, el usuario podrá crear el usuario final con el que accederá a la aplicación.

4.1.2 Identificación

Permite al usuario, previamente registrado, conectarse a la aplicación para poder acceder a las distintas funcionalidades que se les brinda a los usuarios registrados.

4.1.3 Consultar recetas

Permite al usuario, una vez registrado y conectado, acceder a sus recetas propias y demás funcionalidades asociadas a ellas, como modificar los datos de la misma, borrarla o crear nuevas.

4.1.4 Ver detalles receta

Permite al usuario, una vez registrado y conectado, ver los detalles de una receta, con el fin de consultarla, guardarla en favoritos, hacer una lista de la compra asociada a dicha receta, o utilizar la funcionalidad de reconocimiento de voz para cocinarla.

4.1.5 Guardar receta

Permite al usuario, una vez registrado y conectado, crear una receta en su colección de recetas, añadiendo todos los detalles necesarios: ingredientes, elaboración, tipo de dieta… Tiene como subcasos de uso las opciones de Agregar Ingrediente, Agregar Paso, Agregar Foto y Agregar Foto por Paso.

4.1.6 Agregar Ingrediente

Permite al usuario agregar ingredientes a una receta. Es un subcaso que pertenece a Guardar Receta.

4.1.7 Agregar Paso

Permite al usuario agregar un paso en la elaboración de una receta. Es un subcaso que pertenece a Guardar Receta.

4.1.8 Agregar Foto

Permite al usuario guardar una foto de la receta. Es un subcaso que pertenece a Guardar Receta.

4.1.9 Agregar Foto por Paso

Permite al usuario guardar una foto por cada paso de la elaboración. Es un subcaso que pertenece a Guardar Receta.

4.1.10 Modificar receta

Permite al usuario, una vez registrado y conectado, modificar los detalles de una receta anteriormente creada, con lo que el usuario final considere necesario cambiar.

4.1.11 Modificar Ingrediente

Permite al usuario modificar ingredientes a una receta ya guardada. Es un subcaso que pertenece a Modificar Receta.

4.1.12 Modificar Paso

Permite al usuario modificar los pasos en la elaboración de una receta. Es un subcaso que pertenece a Modificar Receta.

4.1.13 Modificar Foto

Permite al usuario modficar una foto de la receta. Es un subcaso que pertenece a Modificar Receta.

4.1.14 Modificar Foto por Paso

Permite al usuario modificar la foto por cada paso de la elaboración. Es un subcaso que pertenece a Modificar Receta.

4.1.15 Borrar receta

Permite al usuario, una vez registrado y conectado, borrar una o varias recetas dentro de su colección de recetas, por los motivos que considere oportunos.

4.1.16 Buscar receta

Permite al usuario, una vez registrado y conectado, buscar una receta dentro de la base de datos, teniendo en cuenta el nombre.

4.1.17 Búsqueda Avanzada

Permite al usuario, una vez registrado y conectado, buscar una receta dentro de la base de datos, teniendo en cuenta el nombre, ingredientes, dieta, cocina y tipo de plato.

4.1.18 Búsqueda por Localización

Permite al usuario, una vez registrado y conectado, buscar una receta dentro de un radio de 50 km de la situación en la que se encuentre.

4.1.19 Guardar receta en favoritos

Permite al usuario, una vez registrado y conectado, buscar o consultar recetas y añadirlas a su apartado de favoritas.

4.1.20 Puntuar recetas

Permite al usuario, una vez registrado y conectado, puntuar recetas.

4.1.21 Lista de la compra

Permite al usuario, una vez registrado y conectado, y habiendo buscado y consultado una receta, generar la lista de la compra asociada a dicha receta para el número de comensales requerido por el usuario.

4.1.22 Perfil

Permite al usuario, una vez registrado y conectado, acceder a todos los datos de su perfil, salvo el nickname.

4.1.23 Modificar perfil

Permite al usuario, una vez registrado y conectado, modificar los datos de su perfil.

4.1.24 Consultar seguidores/siguiendo

Permite al usuario, una vez registrado y conectado, consultar el listado de sus amigos.

4.1.25 Buscar amigos

Permite al usuario, una vez registrado y conectado, buscar usuarios teniendo en cuenta el nickname o localización.

4.1.26 Añadir a siguiendo

Permite al usuario, una vez registrado y conectado, añadir uno o varios usuarios a su lista de amigos asociada a su perfil.

4.1.27 Favoritos

Permite al usuario, una vez registrado y conectado, consultar las recetas que tiene en su lista de Favoritas.

4.1.28 Ordenar favoritos

Permite al usuario, una vez registrado y conectado, ordenar su lista de recetas favoritas dependiendo del tipo de ingrediente, dieta o cocina.

4.1.29 Recomendaciones

Permite al usuario, una vez registrado y conectado, recibir recomendaciones sobre recetas por el sistema, teniendo en cuenta las puntuaciones dadas, las recetas guardadas en favoritos y las recetas consultadas.

4.1.30 Cocinar receta

Permite al usuario, una vez registrado y conectado, consultar la receta y los diferentes pasos dentro de la elaboración de la misma, sin tener que utilizar la pantalla táctil para ello y haciendo uso de la técnica de reconocimiento de voz.

4.2 Modelo de dominio

A continuación se muestra el modelo de dominio asociado a esta aplicación, donde se pueden ver qué tipos de datos se almacenan: usuarios, recetas, elaboración e ingredientes. Asimismo, se establece la relación entre estos cuatro tipos de elementos, donde las recetas tendrán uno o varios ingredientes asociados y donde el usuario tendrá una lista de recetas, de favoritos y de puntuaciones votadas, así como su propia lista de amigos y seguidores.

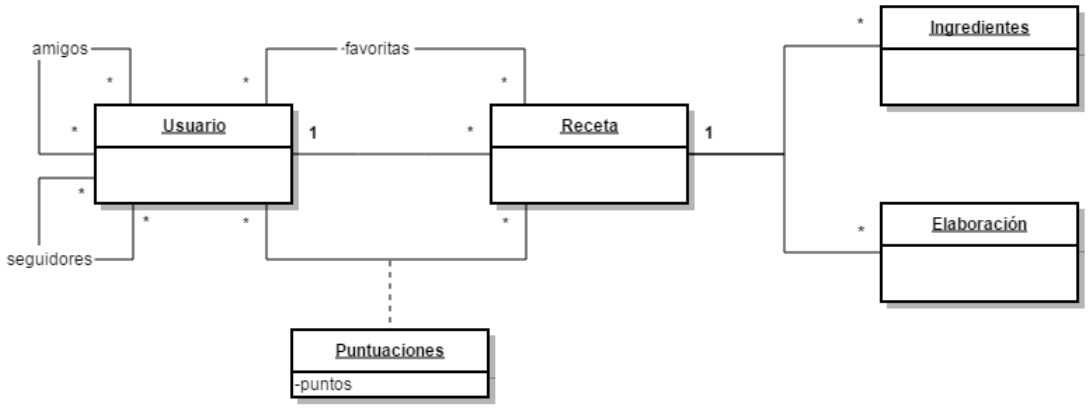


Ilustración 12: Modelo de dominio

5. ANÁLISIS Y DISEÑO

5.1 Introducción a Android y a Appengine

Como se ha comentado en apartados anteriores, el sistema operativo elegido como plataforma final para esta aplicación es Android. Dicho sistema está basado en el kernel de Linux e incluye un conjunto de bibliotecas base, escritas en lenguaje Java, además de su propia máquina virtual.

De este modo, las herramientas para desarrollar en Java una aplicación para cualquier versión de Android están disponibles en el paquete Android Software Development Kit (SDK) y son el único requisito indispensable para ello, además de ser fácilmente integrables en un IDE como Eclipse Kepler.

En este caso en concreto, la aplicación se ha desarrollado en la versión de Android 4.1.2. La selección de la versión va unida sobre todo a temas de diseño, así como a poder implementar las nuevas opciones y funcionalidades mejoradas que otorga Android en versiones superiores a la API 14: fragments, action bar, reconocimiento de voz, geolocalización y demás.

En este trabajo, y en general en cualquier proyecto desarrollado en Android, se utiliza el patrón de arquitectura llamado Modelo – Vista – Controlador. Android hace uso de Java para su desarrollo, pero se diferencia de cualquier aplicación de escritorio desarrollada en este mismo lenguaje. Para empezar, Android se divide en actividades. Una actividad es cualquier pantalla que tenga una interfaz de usuario y cada una de ellas tiene su propio fichero XML, en el que se definiría cualquier aspecto visual.

Asimismo, cada actividad define una clase (que heredará de la clase Activity), que será dónde se establezca la lógica necesaria para llevar a cabo las funcionalidades de la actividad en cuestión. Todas las clases comenzarán con el método onCreate() y además, se podrán definir todos los métodos que se crean oportunos. De esta manera, hay una división clara entre la interfaz y el controlador y la primera no interferirá en nada con la segunda.

Además de las actividades, también existen los Fragments. Los Fragments se desarrollaron con la aparición de las tablets para poder poner solución al tema de la adaptación de la interfaz gráfica a pantallas más grandes. De esta manera, se puede definir un fragment como un contenedor, o una parte, de la interfaz que puede añadirse o eliminarse de forma independiente al resto de elementos definidos en la actividad. En este proyecto, se han definido varios

fragments para poder hacer un mejor uso del espacio en las interfaces. Todas las actividades y fragments están definidos más abajo en el diagrama de clases correspondiente a este proyecto.

En cuanto a la parte del modelo, se ha escogido la plataforma Google App Engine para alojar la base de datos asociada a la aplicación. Mediante esta herramienta, Google facilita una infraestructura con las mismas tecnologías de velocidad y fiabilidad que utiliza cualquier sitio web de Google. Entre todas las opciones que abarca esta plataforma, la base de datos se alojará en la Datastore y será accesible mediante la API definida gracias al conjunto de herramientas y librerías de Google Cloud Endpoints.

Existe una versión Java y otra en Python para el desarrollo de la Datastore, y en este caso, se optó el lenguaje Python, debido a que dicho módulo está más desarrollado y existe más documentación accesible, además del incentivo de aprender a programar en un lenguaje nuevo.

La App Engine Datastore de Python es un almacén de datos no relacional (NoSQL), con todas las ventajas que ello conlleva:

- Mayor escalabilidad y de manera sencilla.
- Mayor velocidad de lectura y escritura.
- Evasión de cuellos de botella.
- Manejo de grandes cantidades de datos de forma fácil.
- Mayor consistencia de datos.

Por ende, no existen tablas ni campos en este tipo de base de datos, sino que tendremos entidades para describir un objeto. Cada entidad dispondrá de las propiedades asociadas a su condición, además de una clave única que lo identifique y serán instancias de la clase ndb.Model (la clase correspondiente a la estructura de las entidades en la Datastore). De esta manera, tendremos modelos. Y cada modelo corresponderá a la entidad y a todos los métodos asociados a la misma. Más abajo se podrán tomar como ejemplo y de manera explicada todos los modelos asociados a este trabajo.

Para hacer cualquier tipo de consultas en la Datastore se puede utilizar un lenguaje similar al SQL, llamado GQL (Google Query Language), o simplemente hacer uso del método query() asociado a cualquier modelo ndb y que asimismo, contiene un amplio abanico de métodos de filtrado y demás operaciones. Tanto para guardar como para modificar un dato existente, se utiliza el método put() asociado a la entidad del modelo que se quiera guardar o modificar. En el apartado de Desarrollo, se explicará en detalle cómo se realizan todas estas tareas y todos los métodos creados para desarrollar las funcionalidades de este proyecto.

Para poder acceder a los modelos de la Datastore, se han creado un número de llamadas o métodos mediante la librería Google Protocol RPC. Esta librería está formada por servicios basados en llamadas HTTP remotas, que ofrecen un conjunto de métodos y tipos de mensajes que proveen un acceso estructurado a las aplicaciones. De esta manera, mediante Python se pueden definir los mensajes y métodos necesarios para poder acceder a los datos de la Datastore.

Asimismo, Appengine ofrece un listado de scripts que facilitan la integración entre la parte servidor (Datastore + RPC) y la aplicación final. Para este caso en concreto, existe un script llamado endpointscfg.py que viene con la aplicación y mediante el cual se generan las librerías Java necesarias, en las que se incluyen todos los métodos definidos en Python, y las cuales serán accesibles simplemente importándolas en el proyecto final en el IDE escogido. Generar estas librerías es algo optativo, dado que una vez creado el servidor (Datastore + RPC), las llamadas a la API desde Android podrían hacerse simplemente con una llamada remota HTTP en un hilo asíncrono. Para este proyecto, se ha decidido generar las librerías con el fin de facilitar el acceso a la API desde Android. Una vez más, en el apartado 9.2 de este mismo capítulo se procederá a explicar las librerías generadas y sus funcionalidades.

Por otro lado, la Datastore no permite guardar ficheros que no sean definibles mediante modelos, por lo cual, para el alojamiento de las imágenes (o cualquier tipo de fichero de mayor tamaño) relacionadas con este proyecto, se utilizó otra de las herramientas asociadas a la AppEngine: Blobstore. A estos ficheros se los denomina *blobs* y el acceso a los mismos se añade perfectamente a las clases y métodos anteriormente creados con Python. Mediante la generación de una URL única, al guardar el blob, la llamada nos devolverá un identificador único, blob key, que será el necesario para luego poder rescatar el fichero. Para ello, se han definido dos clases específicas para guardar y devolver imágenes.

AppEngine es una herramienta bastante nueva, por lo cual todavía está en desarrollo. Por eso mismo, a pesar de que la Datastore sea capaz de almacenar coordenadas GPS, a fecha de octubre del 2014, no existe ninguna opción en GQL mediante la cual se puedan hacer búsquedas en función a las coordenadas y la distancia. Debido a ello, se han tenido que buscar otras alternativas para realizar la funcionalidad de buscar recetas cercanas al usuario.

La solución ha sido utilizar la API Search de Google, que facilita un modelo para indexar documentos. Un documento es un objeto que contiene un identificador único y una lista de campos. Entre todos los tipos de campos que se pueden guardar, está el campo Geopoint, que permite almacenar las coordenadas de latitud y longitud. Por otro lado, los índices guardan los

documentos y sirve para devolverlos de manera rápida y eficaz. Entre todas las opciones que existen en la API Search, para esta aplicación sólo se ha utilizado el método distance(), al cual dado dos puntos geográficos te calcula la distancia en metros entre ellos. Como se ha comentado anteriormente, en el apartado de Desarrollo se pasará a profundizar más en cómo se ha utilizado esta herramienta para desarrollar los métodos necesarios para llevar a cabo esta funcionalidad.

5.2 Diagrama de clases

A continuación se muestra el diagrama de clases correspondiente a la aplicación. Se ha dividido en tres imágenes para facilitar la compresión del mismo, debido al hecho del elevado número de clases que contiene. De esta manera, las cajas de color azul corresponden a las actividades, las de color verde a las clases que, junto con las actividades, forman la lógica de la aplicación y las rojas las clases correspondientes a la API y los modelos del Datastore.

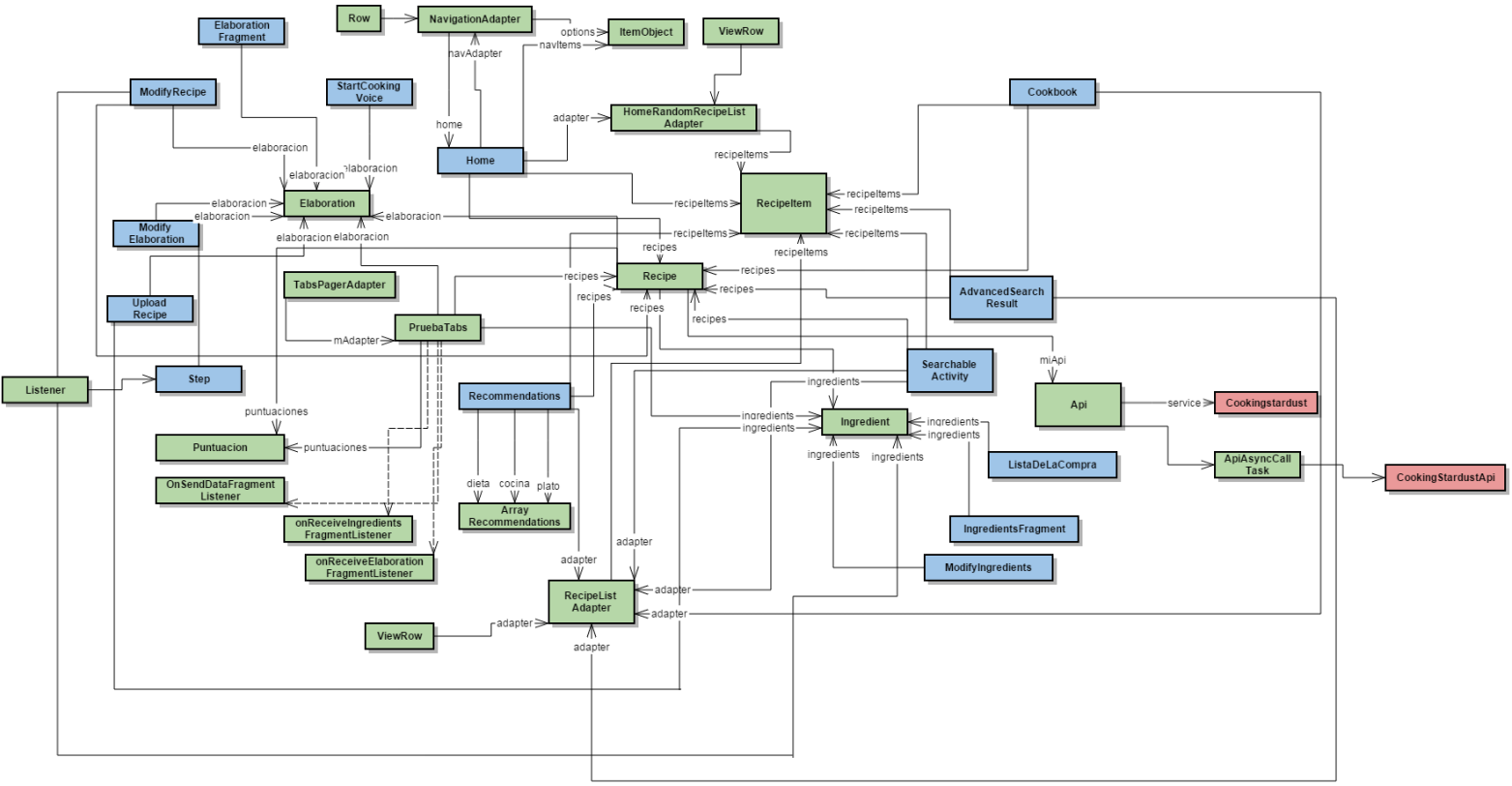


Ilustración 13: Diagrama de clases - 1

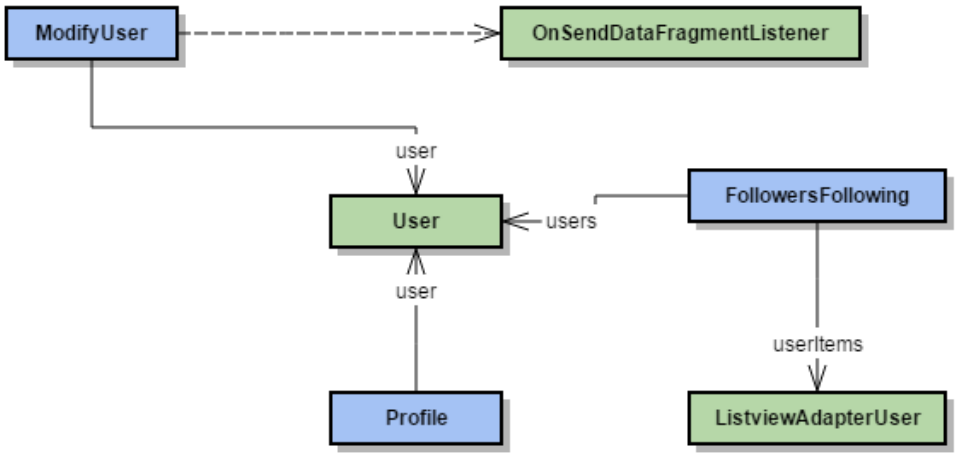


Ilustración 14: Diagrama de clases – 2

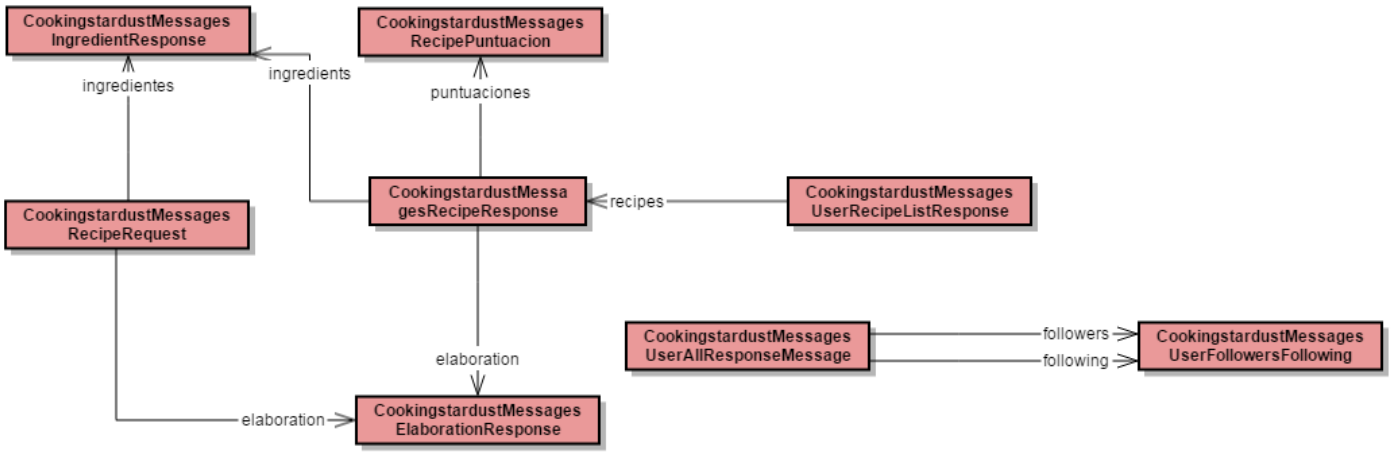


Ilustración 15: Diagrama de clases - 3

5.3 5.3 Detalles de las clases

A continuación se detallan las clases de los paquetes Android com.endpoints.cookingstardust y com.endpoints.cookingstardust.model, que son las que se han creado tanto mediante el script de generación de librerías Java del que dispone Google AppEngine, como otras clases necesarias para la perfecta comunicación entre el servidor y la aplicación cliente final. Estas clases son las encargadas de realizar las llamadas a la API y de recibir o enviar los datos de la manera correcta para guardarlos o extraerlos de la Datastore, de una manera asíncrona.

5.3.1 API

La clase API es la encargada de hacer las llamadas (GET o POST) a la Datastore, haciendo para ello uso de las demás clases creadas en el AppEngine.

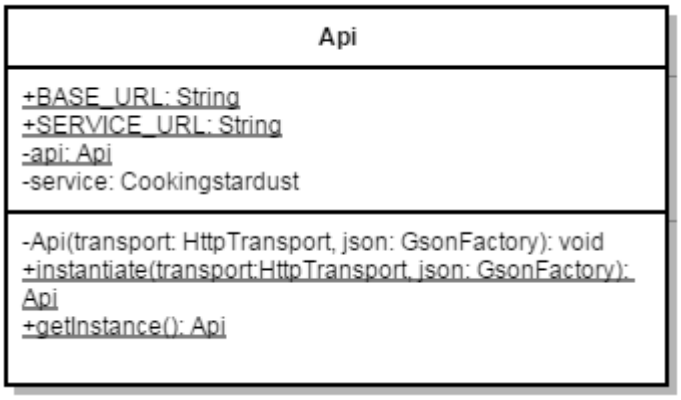


Ilustración 16: API

5.3.2 ApiCallAsyncTask

Esta clase es la encargada de hacer las llamadas que se hagan a la Datastore desde las clases User y Recipe de manera que se ejecuten en segundo plano, sin bloquear la aplicación. Se incluye también en la clase Api comentada más arriba.

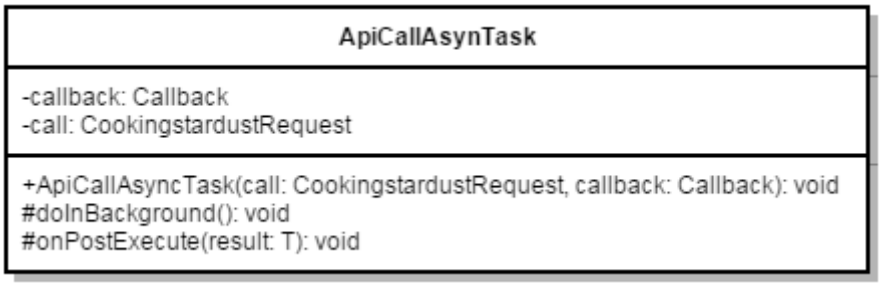


Ilustración 17: ApiAsyncCallTask

5.3.3 User

La clase User se incluye en la clase Api definida arriba, y se encarga de hacer todas las llamadas asociadas al modelo User de la Datastore.

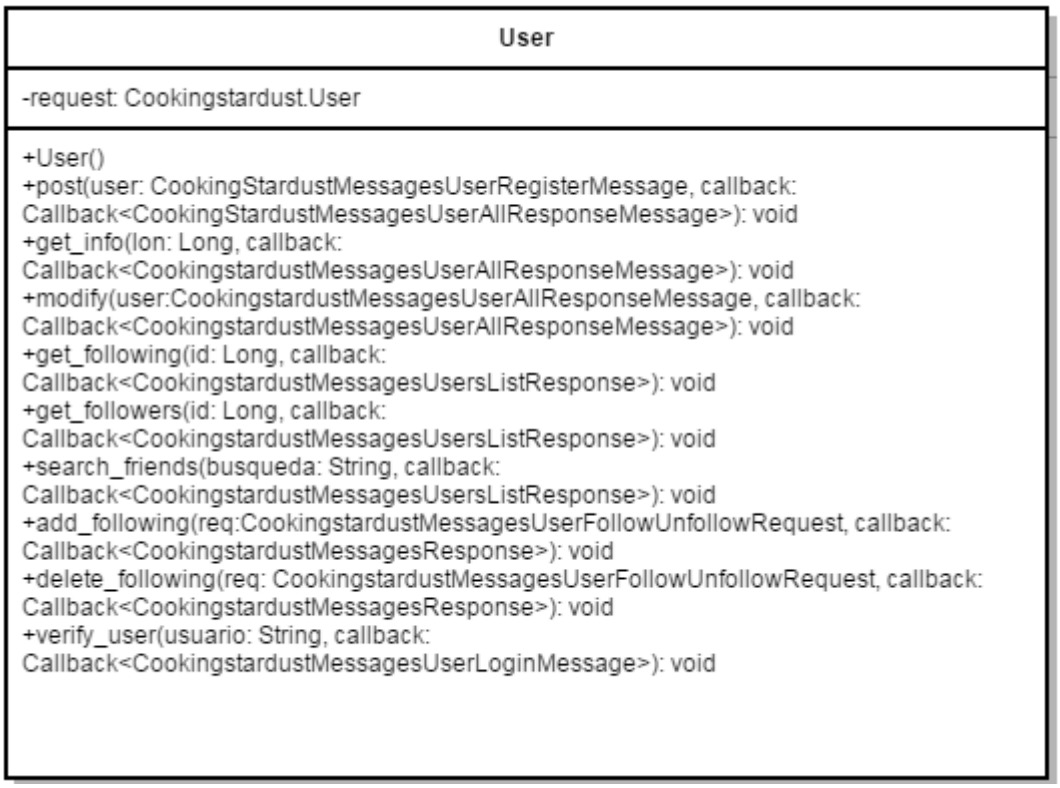


Ilustración 18: User

5.3.4 Callback

La interfaz callback es la encargada de devolver el resultado de los métodos que se ejecuten en segundo plano al hilo principal.

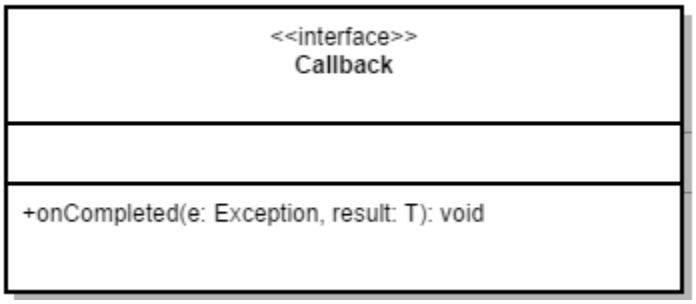


Ilustración 19: Callback

5.3.5 Recipe

La clase Recipe se incluye en la clase Api definida más arriba, y se encarga de hacer todas las llamadas asociadas al modelo Recipe de la Datastore.

Ilustración 20: Recipe

5.3.6 Cookingstardust

La clase Cookingstardust es la clase equivalente a la API de la parte servidor, por lo cual alberga todos los métodos necesarios para poder realizar las llamadas a la API de manera satisfactoria. Asimismo, todos sus métodos están generados de manera automática, por lo cual no se han realizado modificaciones en ellos.

5.3.7 CookingstardustRequest

Esta clase la clase encagada de hacer la llamada a la API, una vez se ha definido la llamada y el servicio gracias a la clase Cookingstardust comentada anteriormente. Establece toda la configuración necesaria para ello. Al igual que todas estas clases, se ha generado de manera automática.

5.3.8 Elaboration

Esta clase es la equivalente a la declarada en el modelo de la Datastore, y recoge la información asociada a la elaboración de una receta: paso y foto del mismo. Mediante el método parse, se parseará el resultado original de la Datastore (clase CookingstardustMessagesElaborationResponse) en un objeto entendible por Java.

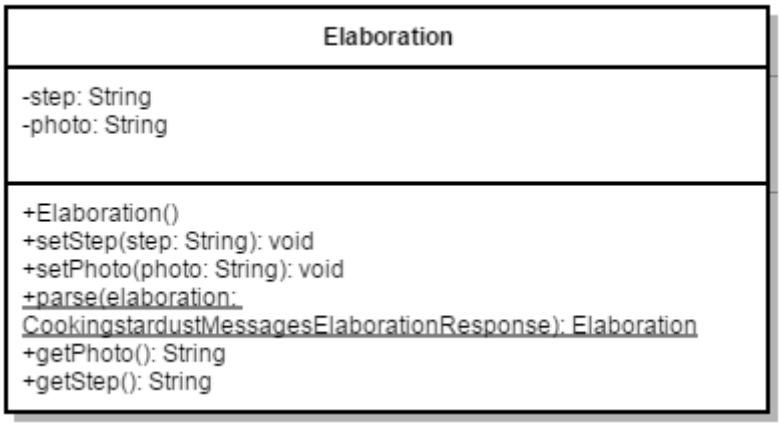


Ilustración 21: Elaboracion

5.3.9 Ingredient

Esta clase es la equivalente a la declarada en el modelo de la Datastore, y recoge la información asociada a un ingrediente: nombre y cantidad. Mediante el método parse, se parseará el resultado original de la Datastore (clase CookingstardustMessagesIngredientResponse) en un objeto entendible por Java.

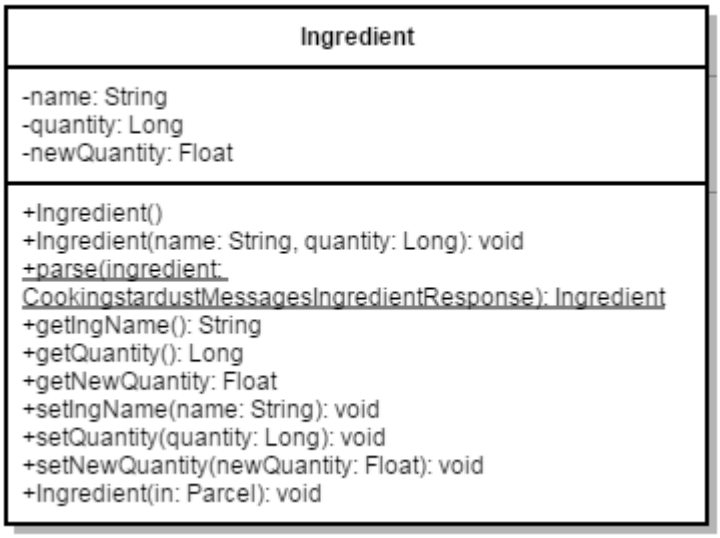


Ilustración 22: Ingredient

5.3.10 Puntuación

Esta clase es la equivalente a la declarada en el modelo de la Datastore, y recoge la información asociada a la puntuación de una receta: identificador del usuario y puntos. Mediante el método parse, se parseará el resultado original de la Datastore (clase CookingstardustMessagesRecipePuntuacion) en un objeto entendible por Android.

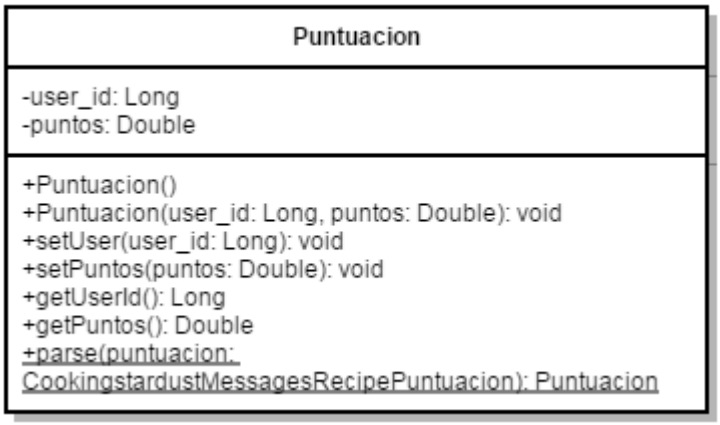


Ilustración 23: Puntuacion

5.3.11 CookingstardustMessagesUsersListResponse

Esta clase es una de las generadas por las librerías de la AppEngine, y mantiene la misma estructura que la clase UserRecipeListResponse de Python definida en el servidor y que sirve como clase intermedia entre el endpoint y la Datastore. En ella se define una lista de usuarios.

Ilustración 24: CookingstardustMessagesUserListResponse

5.3.12 Recipe

Esta clase es la equivalente a la declarada en el modelo de la Datastore, y recoge la información asociada a una receta, así como todos los métodos necesarios para poder acceder a los datos. Mediante el método parse, se parseará el resultado original de la Datastore (clase CookingstardustMessagesRecipeResponse) en un objeto entendible por Java.

Ilustración 25: Recipe

5.3.13 User

Esta clase es la equivalente a la declarada en el modelo de la Datastore, y recoge la información asociada a un usuario, así como todos los métodos necesarios para poder acceder a los datos. Mediante el método parse, se parseará el resultado original de la Datastore (clase CookingstardustMessagesUserAllResponseMessage) en un objeto entendible por Java.

Ilustración 26: User

5.3.14 CookingstardustMessagesUsersRegisterMessage

Esta clase es una de las generadas por las librerías de AppEngine, y mantiene la misma estructura que la clase UserRegisterMessage de Python definida en el servidor y que sirve como clase intermedia entre el endpoint y la Datastore. Es la clase encargada de guardar los datos de registro de un usuario con el fin de mandarlos a la Datastore mediante los métodos implementados en la clase Cookingstardust, por lo cual tiene los atributos y métodos necesarios para ello.

5.3.15 CookingstardustMessagesUserRecipeListResponse

Esta clase una de las generadas por las librerías de AppEngine, y mantiene la misma estructura que la clase UserRecipeListResponse de Python definida en el servidor y que sirve como clase intermedia entre el endpoint y la Datastore. Tiene como atributo una lista de recetas.

5.3.16 CookingstardustMessagesUserLoginMessage

Esta clase una de las generadas por las librerías de AppEngine, y mantiene la misma estructura que la clase UserLoginMessage de Python definida en el servidor y que sirve como clase intermedia entre el endpoint y la Datastore. En ella se definen como atributos los datos que se devuelven desde la API al loguearse un usuario y los métodos necesarios para acceder a ellos.

5.3.17 CookingstardustMessagesUserFollowUnfollowRequest

Esta clase una de las generadas por las librerías de AppEngine, y mantiene la misma estructura que la clase UserFollowUnfollowRequest de Python definida en el servidor y que sirve como clase intermedia entre el endpoint y la Datastore. En ella se definen como atributos el identificador tanto del seguidor como del siguiendo, que se necesitan para poder seguir o dejar de seguir a un usuario o amigo.

5.3.18 CookingstardustMessagesUserFollowersFollowing

Esta clase es una de las generadas por las librerías de AppEngine, y mantiene la misma estructura que la clase UserFollowersFollowing de Python definida en el servidor y que sirve como clase intermedia entre el endpoint y la Datastore. En ella se guarda como atributo el identificador de un usuario, y que servirá de base para otras clases a la hora de definir listas de seguidores y siguiendo.

5.3.19 CookingstardustMessagesUrlMessage

Esta clase es una de las generadas por las librerías de AppEngine, y mantiene la misma estructura que la clase UrlMessage de Python definida en el servidor y que sirve como clase intermedia entre el endpoint y la Datastore. En ella se define como atributo el campo url, y será utilizado para devolver la URL generada por la Blobstore para poder guardar una imagen en el servidor.

5.3.20 CookingstardustMessagesResponse

Esta clase es una de las generadas por las librerías de AppEngine, y mantiene la misma estructura que la clase Response de Python definida en el servidor y que sirve como clase intermedia entre el endpoint y la Datastore. En ella se define un atributo booleano there, que será devuelto en mediante los métodos implementados en la clase Cookingstardust que necesiten únicamente devolver un booleano.

5.3.21 CookingstardustMessagesUserAllResponseMessage

Esta clase es una de las generadas por las librerías de AppEngine, y mantiene la misma estructura que la clase UserAllResponseMessage de Python definida en el servidor y que sirve como clase intermedia entre el endpoint y la Datastore. En ella se definen todos los atributos necesarios en concordancia con los datos relacionados con un usuario que se guardan en la Datastore y a los cuales será necesario acceder para poder establecer el Perfil del usuario, por ejemplo, así como los métodos asociados para poder construir y devolver todos los campos.

5.3.22 CookingstardustMessagesRecipeResponse

Esta clase es una de las generadas por las librerías de AppEngine, y mantiene la misma estructura que la clase RecipeResponse de Python definida en el servidor y que sirve como clase intermedia entre el endpoint y la Datastore. En ella se definen todos los atributos necesarios en concordancia con los datos relacionados con una receta que se guardan en la Datastore y a los cuales será necesario acceder para poder mostrar la receta al completo, por ejemplo, así como los métodos asociados para poder establecer y devolver todos los campos.

5.3.23 CookingstardustMessagesRecipeRequest

Esta clase es una de las generadas por las librerías de AppEngine, y mantiene la misma estructura que la clase RecipeRequest de Python definida en el servidor y que sirve como clase intermedia entre el endpoint y la Datastore. Cada vez que se quiera guardar una receta, se creará una instancia de esta clase, con los atributos necesarios acordes a los datos de la receta, y se pasará como parámetro al método encargado de hacer la llamada a la API con el fin de guardar la receta en la Datastore.

5.3.24 CookingstardustMessagesRecipePuntuacion

Esta clase es una de las generadas por las librerías de AppEngine, y mantiene la misma estructura que la clase RecipePuntuacion de Python definida en el servidor y que sirve como clase intermedia entre el endpoint y la Datastore. La clase CookingstardustMessagesRecipeResponse tira de esta clase a la hora de definir el atributo Puntuaciones. Cada receta tendrá una lista de puntuaciones, y cada una de ellas guardará el identificador del usuario y la puntuación dada por este.

5.3.25 CookingstardustMessagesRecipePunctuationRequest

Esta clase es una de las generadas por las librerías de AppEngine, y mantiene la misma estructura que la clase RecipePunctuationRequest de Python definida en el servidor y que sirve como clase intermedia entre el endpoint y la Datastore. Será la encargada de guardar los datos necesarios para poder guardar una puntuación dada por un usuario concreto a una

receta concreta. Se pasará como parámetro al método encargado de hacer la llamada a la API para guardar los datos de la puntuación.

5.3.26 CookingstardustMessagesRecipeDeleteRequest

Esta clase es una de las generadas por las librerías de AppEngine, y mantiene la misma estructura que la clase RecipeDeleteRequest de Python definida en el servidor y que sirve como clase intermedia entre el endpoint y la Datastore. Tiene definidos como atributos los campos necesarios para poder eliminar una receta de un usuario y se pasará como parámetro al método encargado de hacer la llamada a la API.

5.3.27 CookingstardustMessagesIngredientResponse

Esta clase es una de las generadas por las librerías de AppEngine, y mantiene la misma estructura que la clase IngredientResponse de Python definida en el servidor y que sirve como clase intermedia entre el endpoint y la Datastore. . En ella se definen como atributos todos los datos relacionados con los ingredientes de una receta que se guardan en la Datastore y a los cuales será necesario acceder para poder mostrar la receta al completo, por ejemplo.

5.3.28 CookingstardustMessagesElaborationResponse

Esta clase es una de las generadas por las librerías de AppEngine, y mantiene la misma estructura que la clase ElaborationResponse de Python definida en el servidor y que sirve como clase intermedia entre el endpoint y la Datastore. En ella se definen como atributos todos los datos relacionados con la elaboración de una receta que se guardan en la Datastore y a los cuales será necesario acceder para poder mostrar la receta al completo, por ejemplo, así como definir la funcionalidad “Cocinar Receta” que va paso por paso.

A continuación se detallan las clases implementadas en el paquete com.example.cookingstardust, que son las que forman la aplicación en sí, y desde las cuáles se llaman a los métodos de los paquetes com.endpoints.Cookingstardust.model y com.endpoints.Cookingstardust con el fin de enviar u obtener los datos necesarios de la Datastore.

5.3.29 AdvancedSearch

La clase AdvanceSearch permite al usuario hacer una búsqueda más avanzada de las recetas, añadiendo a la búsqueda los campos nombre de ingrediente, dieta, cocina, dificultad y tipo de plato.

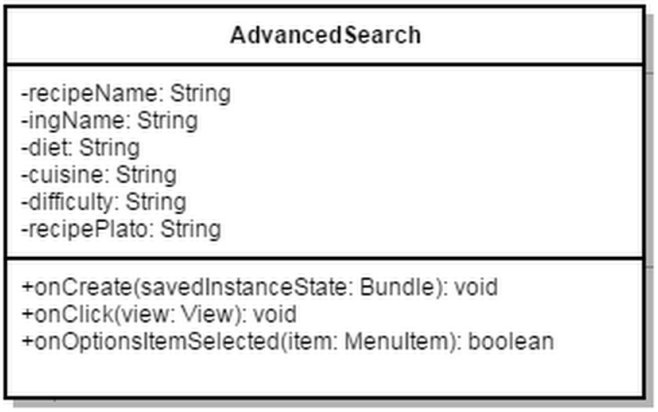


Ilustración 27: AdvancedSearch

5.3.30 AdvancedSearchResult

La clase AdvancedSearchResult muestra al usuario las recetas obtenidas después de haber hecho una búsqueda avanzada.

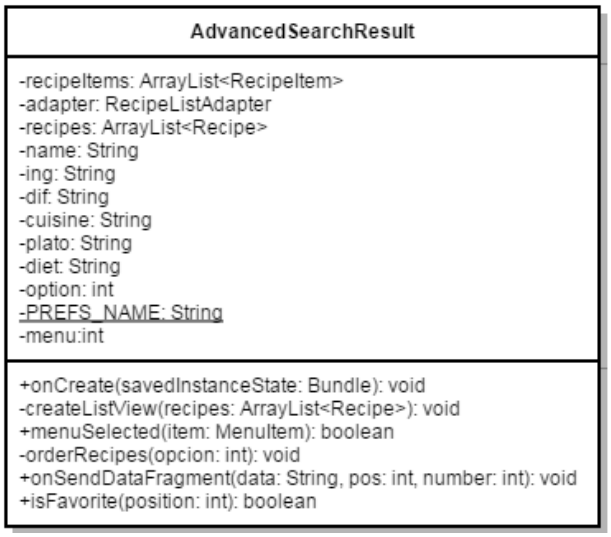


Ilustración 28: AdvancedSearchResult

5.3.31 AlfabeticoComparator

Esta clase implementa el método compare que te obliga la interfaz *Comparator* y se utiliza para ordenar las recetas de manera alfabética por su nombre.

5.3.32 CocinaComparator

Esta clase implementa el método compare que te obliga la interfaz Comparator y se utiliza para ordenar las recetas de manera alfabética por su tipo de cocina.

5.3.33 DietaComparator

Esta clase implementa el método compare que te obliga la interfaz Comparator y se utiliza para ordenar las recetas de manera alfabética por su tipo de dieta.

5.3.34 RecommendationsComparator

Esta clase implementa el método compare que te obliga la interfaz Comparator y se utiliza para ordenar las lista de nombres por puntuación.

5.3.35 Friends

La clase Friends permite al usuario hacer una búsqueda de usuarios, tanto por nombre como por nickname.

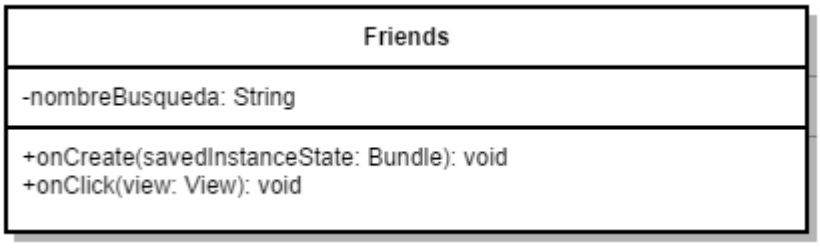


Ilustración 29: Friends

5.3.36 CustomUserAdapterList

Esta clase define un adaptador personalizado para mostrar el listado de usuarios en la clase *FollowersFollowing*. De esta manera, más allá de ser una simple lista, incluirá una foto del perfil del usuario a la izquierda del todo, así como su nombre y su nickname.

5.3.37 Cookbook

La clase Cookbook muestra al usuario un listado de sus recetas, teniendo la opción de seleccionar cualquiera de ellas para poder acceder a los detalles.

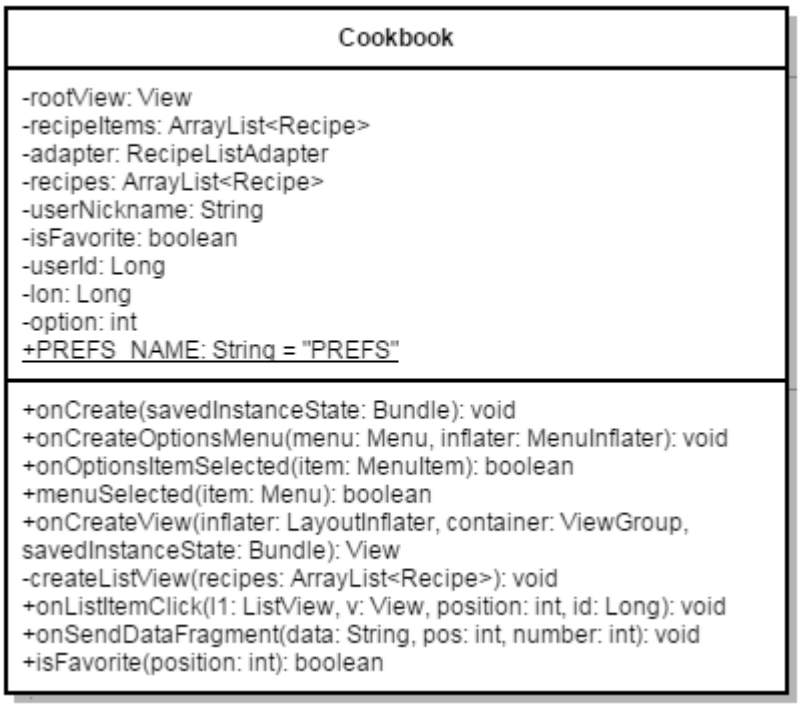


Ilustración 30: Cookbook

5.3.38 UploadRecipe

La clase *UploadRecipe* permite al usuario guardar una receta a su colección de recetas personales con todos los detalles que se le ofrecen, así como con la opción de adjuntar también una fotografía de la misma y de los pasos de la receta.

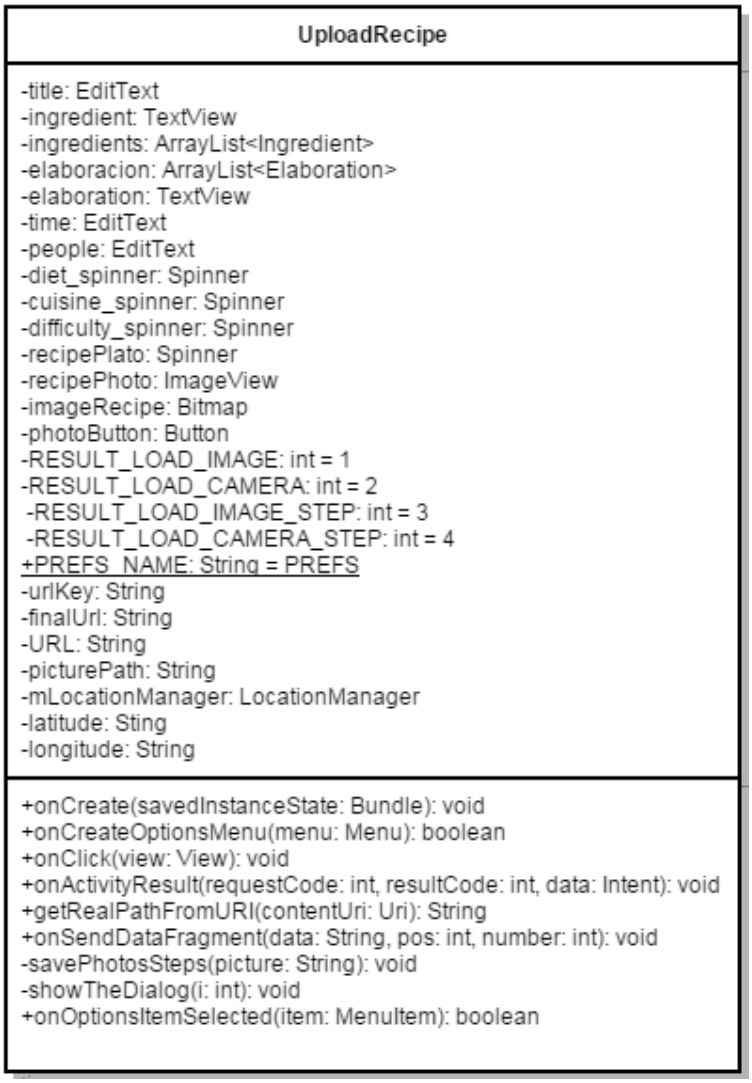


Ilustración 31: UploadRecipe

5.3.39 FollowersFollowing

La clase *FollowersFollowing* se utiliza para mostrarle al usuario un listado de usuarios, que bien pueden ser sus seguidores, sus siguiendo o la lista de usuarios que cuadren con la búsqueda que haya realizado desde la clase *Friends*. También tendrá la opción de seleccionar cualquiera de ellos para poder acceder a su perfil.

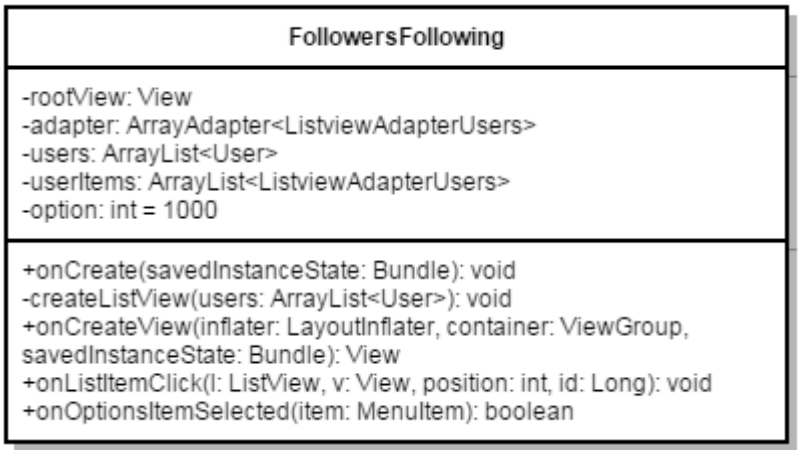


Ilustración 32: FollowingFollowers

5.3.40 IngredientsFragment

La clase *IngredientsFragment* implementa un fragment para mostrar los ingredientes de una receta. Será la segunda de las tres pestañas definidas en la clase *TabsPagerAdapter*, y se podrán pasar de una a otra deslizándolas con el dedo.

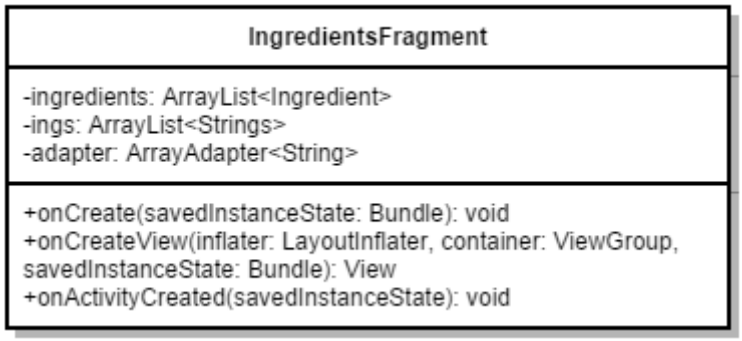


Ilustración 33: IngredientFragment

5.3.41 MainActivity

Esta será la principal, en la cual el usuario podrá identificarse o acceder a la opción de registrarse en el sistema.

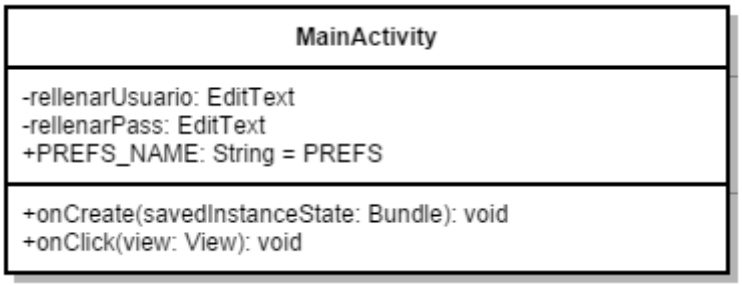


Ilustración 34: MainActivity

5.3.42 ItemObject

Esta clase se utiliza para crear los objetos (formados por el nombre y el icono correspondiente) del menú deslizable que está disponible en la pantalla *Home*.

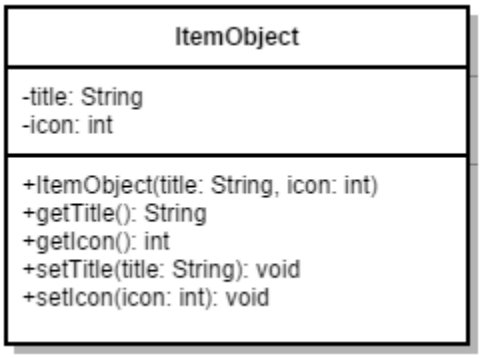


Ilustración 35: ItemObject

5.3.43 Home

Esta clase es a la que se accede una vez el usuario se ha identificado. Desde ella, y con el menú deslizable de la izquierda, podrá acceder al resto de funcionalidades de la aplicación.

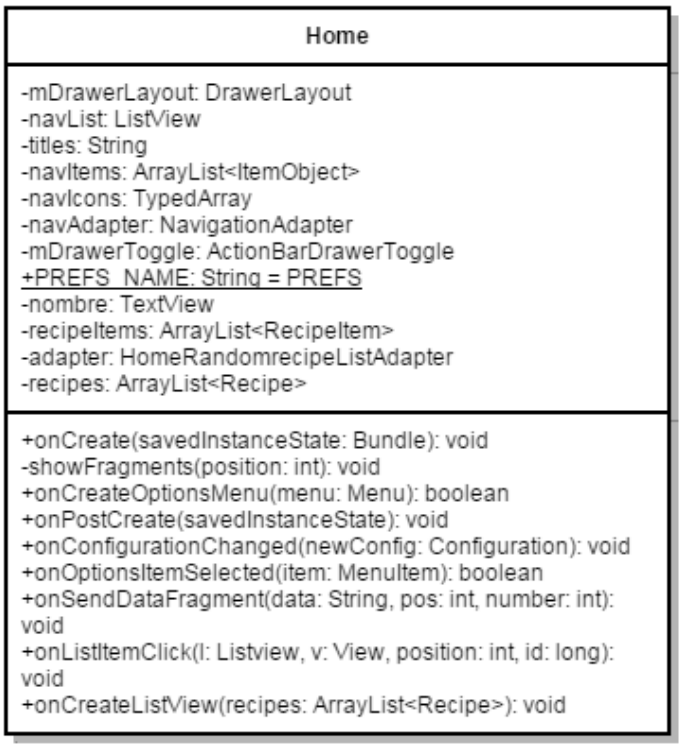


Ilustración 36: Home

5.3.44 ListaDeLaCompra

La clase *ListaDeLaCompra* muestra al usuario la cantidad de ingredientes que son necesarios para cocinar la receta seleccionada para el número de comensales que haya establecido.

Ilustración 37: ListaDeLaCompra

5.3.45 ListviewAdapterUsers

Esta clase se utiliza para crear los objetos (formados por el nombre, el nickname y el icono del avatar del usuario correspondiente) que utilizará la clase *CustomUserAdapterList* para crear el listado de usuarios personalizado.

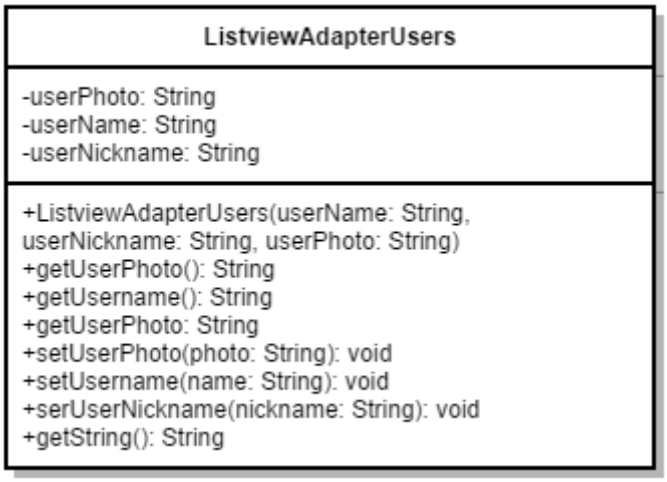


Ilustración 38: ListviewAdapterUsers

5.3.46 Location

La clase *Location* es la encargada de calcular la posición exacta del usuario, siempre y cuando tenga activada dicha opción en la pestaña de Configuración.

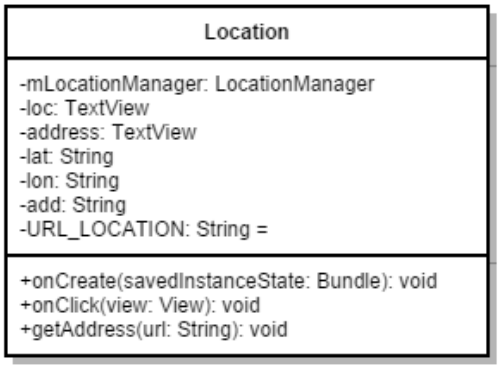


Ilustración 39: Location

5.3.47 ModifyElaboration

Esta clase es la encargada de recoger únicamente los pasos modificados de la elaboración de la receta que se quiera modificar.

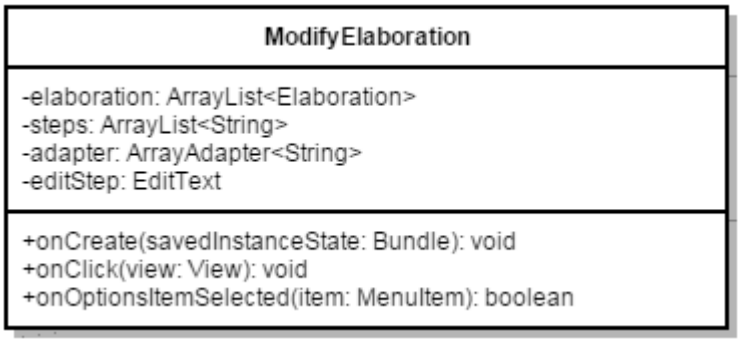


Ilustración 40: ModifyElaboration

5.3.48 ModifyIngredients

Esta clase es la encargada de recoger únicamente los ingredientes modificados de la receta, en caso de que se quieran modificar.

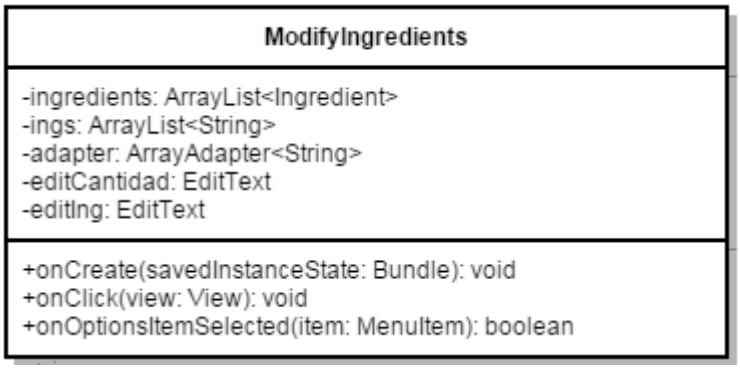


Ilustración 41: ModifyIngredients

5.3.49 NavigationAdapter

Esta clase define un adaptador personalizado para mostrar el listado de funcionalidades en el menú izquierdo de la clase *Home*. De esta manera, más allá de ser una simple lista, incluirá un icono de la funcionalidad.

5.3.50 ModifyUser

La clase ModifyUser permite al usuario modificar todos los datos de su perfil, así como la foto del mismo.

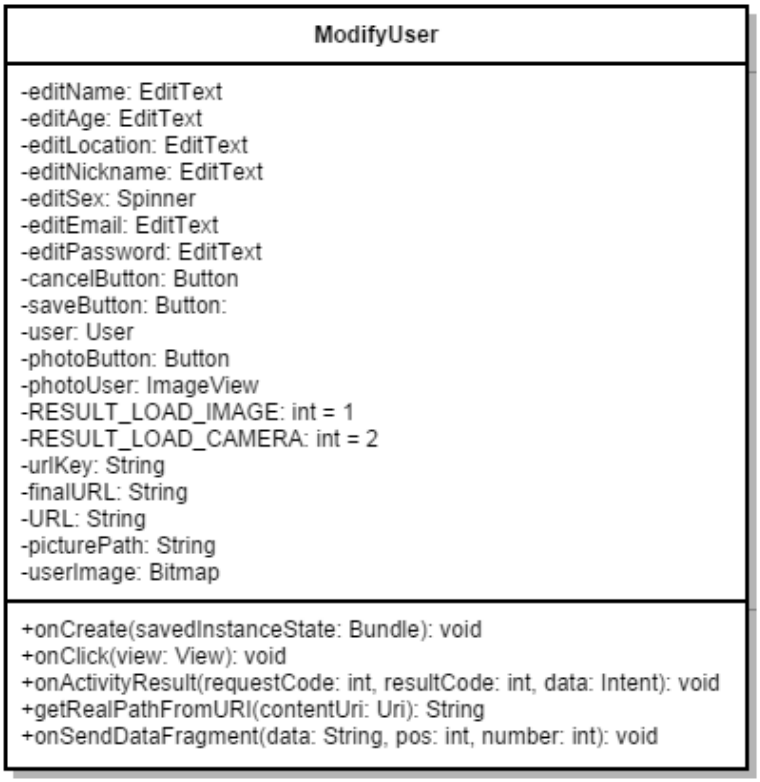


Ilustración 42: ModifyUser

5.3.51 StartCookingVoice

Esta clase permite al usuario comenzar con la funcionalidad *Cocinar*.

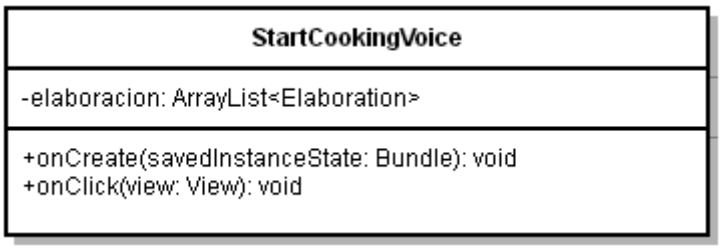


Ilustración 43: StartCookingVoice

5.3.52 TabsPagerAdapter

La clase *TabsPagerAdapter* define un adaptador de pestañas que se utiliza por la clase *PruebaTabs* para crear las tres pestañas deslizables que muestran la información de una receta.

5.3.53 ModifyRecipe

La clase *ModifyRecipe* permite al usuario modificar todos los datos de una receta suya, así como la foto de la misma.

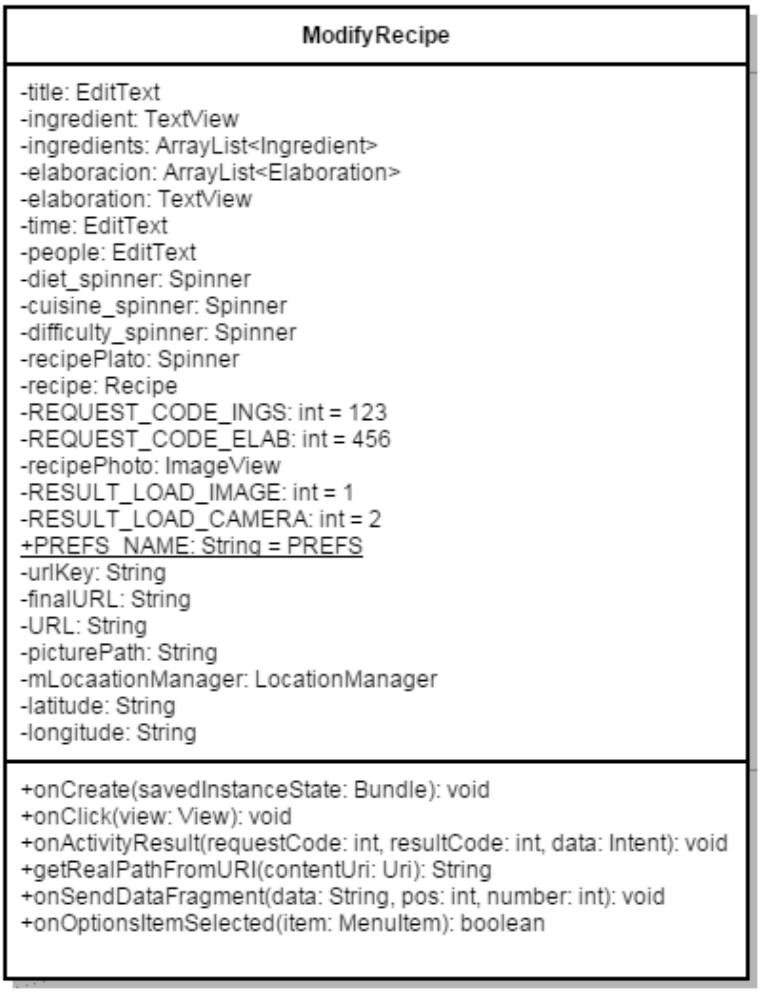


Ilustración 44: ModifyRecipe

5.3.54 Settings

Esta clase establece las opciones configurables de la aplicación de tal manera que el usuario pueda configurar su cuenta.

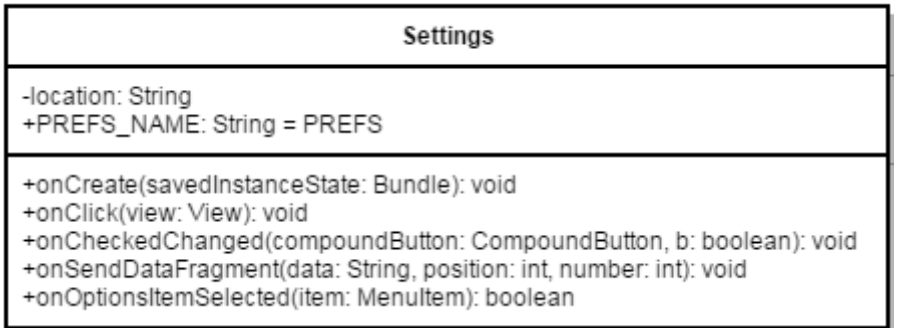


Ilustración 45: Settings

5.3.55 Profile

La clase *Profile* muestra el perfil del usuario, acorde con los datos que haya establecido.

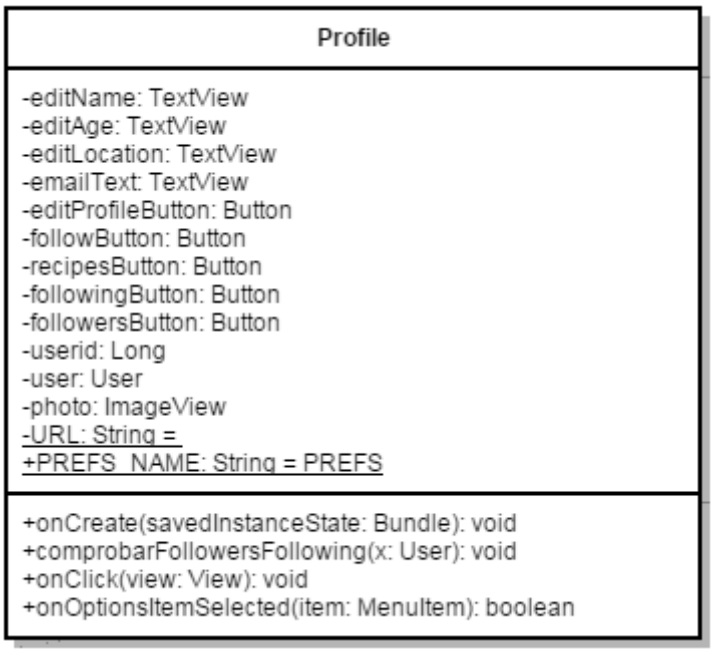


Ilustración 46: Profile

5.3.56 PruebaTabs

Esta clase es la encargada de generar las pestañas deslizables que muestran la información de una receta. Para ello hace uso de la clase *TabsPagerAdapter*.

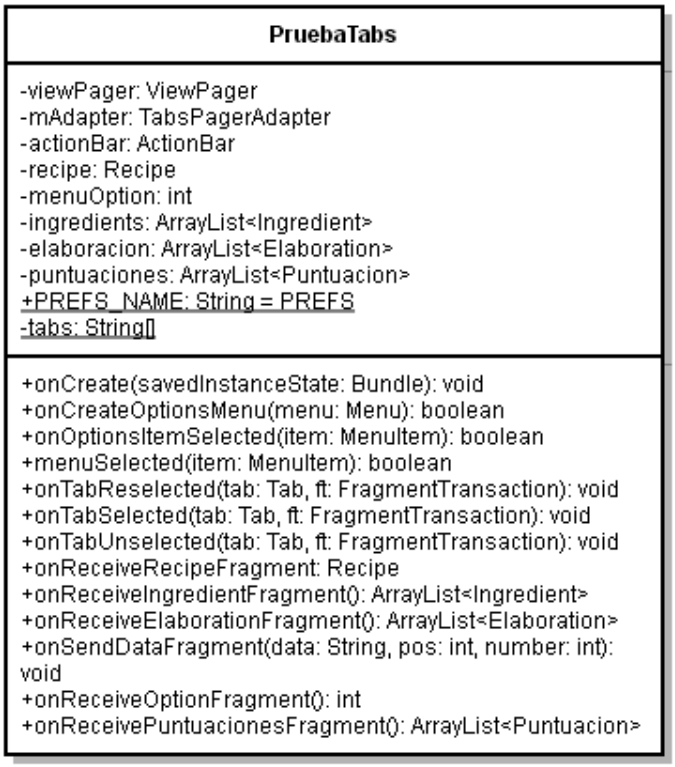


Ilustración 47: PruebaTabs

5.3.57 RecipeDialog

La clase *RecipeDialog* define varios fragments de diálogos personalizados para utilizarlos en varias funcionalidades diferentes de la aplicación.

5.3.58 RecipeItem

Esta clase se utiliza para crear los objetos (formados por el nombre de la receta y el icono de la foto de la misma) que utilizará la clase *RecipeListAdapter* para crear el listado de recetas personalizado. Tiene como parámetros el nombre, tipo de cocina y foto de la receta, así como los métodos necesarios para acceder a ellos.

5.3.59 RecipeListAdapter

Esta clase define un adaptador personalizado para mostrar el listado de recetas en la clase *Cooking*, por ejemplo. De esta manera, más allá de ser una simple lista, incluirá una foto de la receta a la izquierda del todo, así como su nombre y su tipo de cocina. Tiene como atributos la lista de recetas y el contexto de la actividad origen.

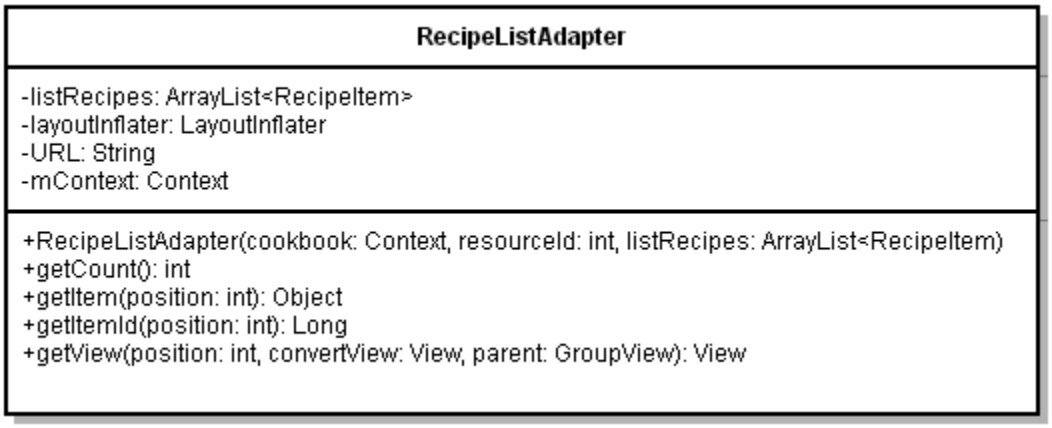


Ilustración 48: RecipeListAdapter

5.3.60 Registrar

La clase Registrar permite al usuario crear una cuenta en la aplicación para poder acceder a todas las funcionalidades que ofrece la misma.

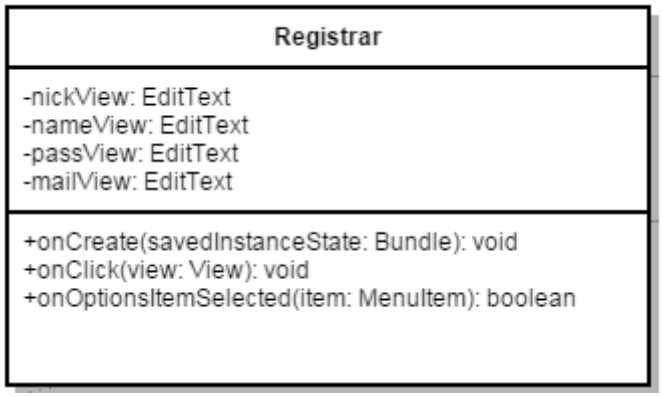


Ilustración 49: Registrar

5.3.61 SearchableActivity

Esta clase permite al usuario ver los resultados, en forma de lista, de una búsqueda ordinaria que haya podido hacer desde la barra de búsqueda en el *ActionBar*.

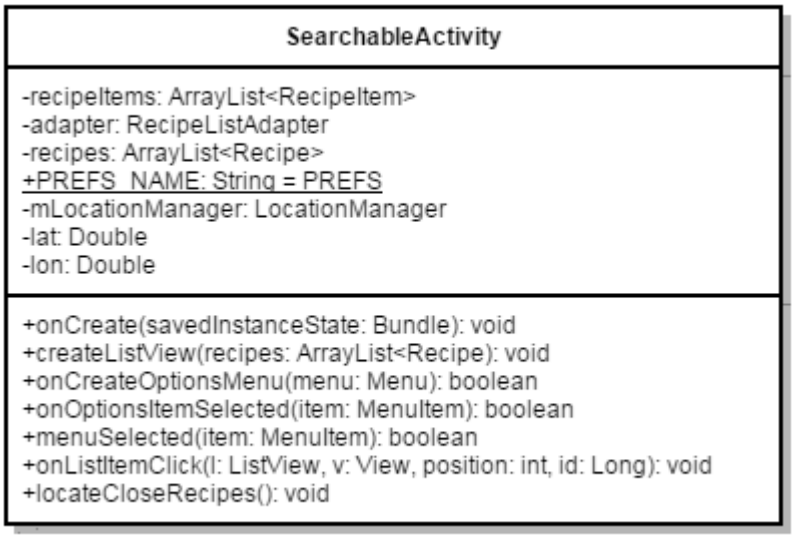


Ilustración 50: SearchableActivity

5.3.62ArrayRecommendations

La clase *ArrayRecommendations* implementa un elemento con un nombre y una puntuación y sirve para construir los ArrayList sobre los cuales se calcularán las recomendaciones basadas en la dieta, tipo de cocina y tipo de plato más comunes en un listado de recetas (favoritas o más puntuadas) de un usuario.

5.3.63Step

Esta clase es la encargada de mostrar los pasos de una receta uno por uno desde la funcionalidad Cocinar (clase *StartCookingVoice*). Asimismo, es la clase que implementa el servicio de reconocimiento de voz dentro de la la funcionalidad Cocinar.

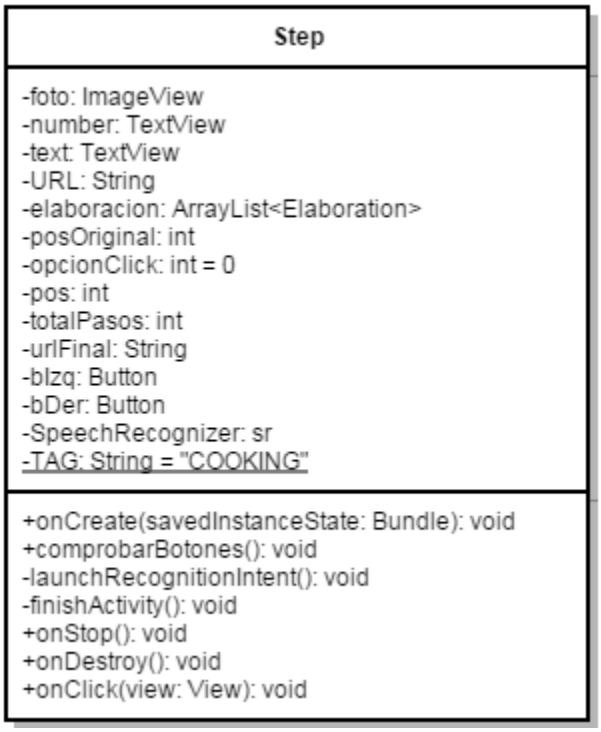


Ilustración 51: Step

5.3.64 SummaryFragment

La clase *SummaryFragment* implementa un fragment para mostrar los datos generales de una receta. Será la primera de las tres pestañas definidas en la clase *TabsPagerAdapter*, y se podrán pasar de una a otra deslizándolas con el dedo.

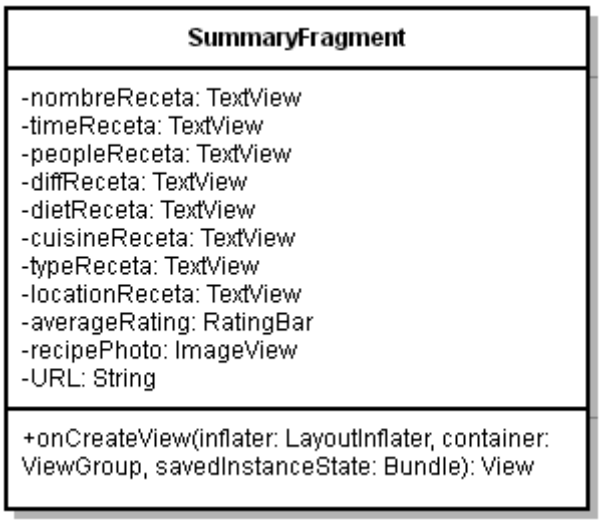


Ilustración 52: SummaryFragment

5.3.65 Recommendations

La clase *Recommendations* implementa la actividad que maneja la funcionalidad de Recomendaciones, basándose en las recetas favoritas o más puntuadas del usuario. En caso de que no sé de ninguna de estas casuísticas, se le mostrarán al usuario las veinte recetas mejor puntuadas del sistema.

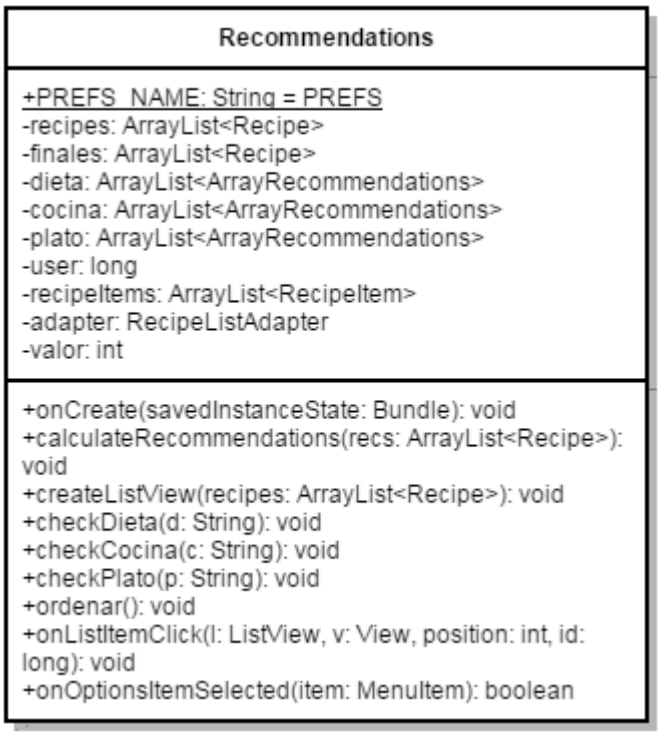


Ilustración 53: Recomendations

5.3.66 HomeRandomRecipeListAdapter

Esta clase define un adaptador personalizado para mostrar el listado de recetas aleatorias en la clase *Home*. De esta manera, más allá de ser una simple lista, incluirá una foto de la receta a que ocupará toda el ancho y alto de la fila, y el nombre de la misma irá en la parte inferior de la imagen.

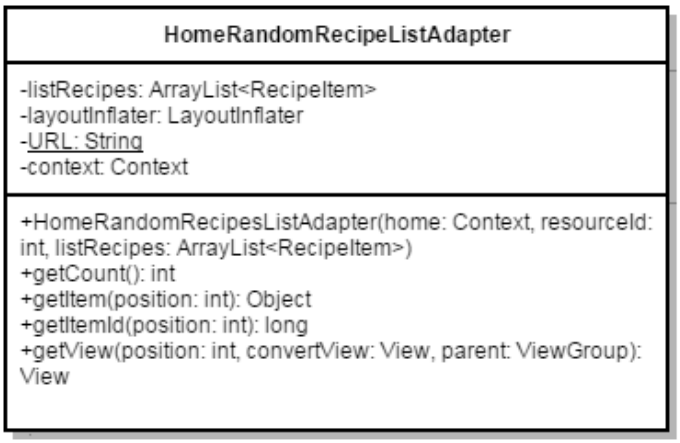


Ilustración 54: HomeRandomRecipeListAdapter

5.3.67 Listener

Esta clase está definida de manera interna en la clase Step e implementa la interfaz RecognitionListener, la encargada de recibir las notificaciones cuando ocurre algún evento de reconocimiento de voz. Tiene los métodos necesarios para tratar esos eventos, y recibir el resultado de los mismos.

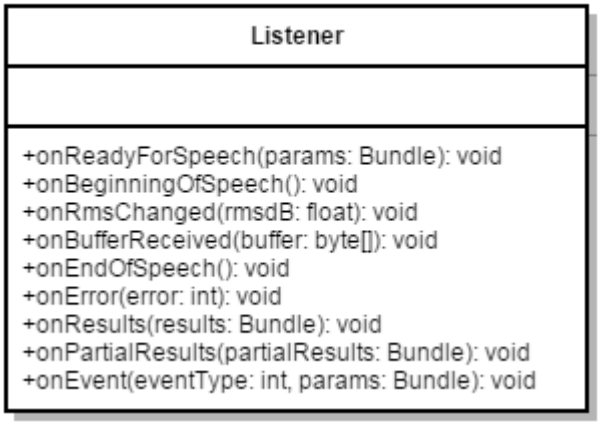


Ilustración 55: Listener

A continuación se detallan las clases creadas en la parte del servidor, entre las que se encuentran las clases de los modelos, los mensajes y la clase de la API con todos los métodos necesarios.

5.3.68 User

Es el modelo asociado a cualquier usuario, donde los atributos corresponden con los datos del mismo, y que contiene los métodos necesarios para hacer consultas, inserciones o borrados a la Datastore.

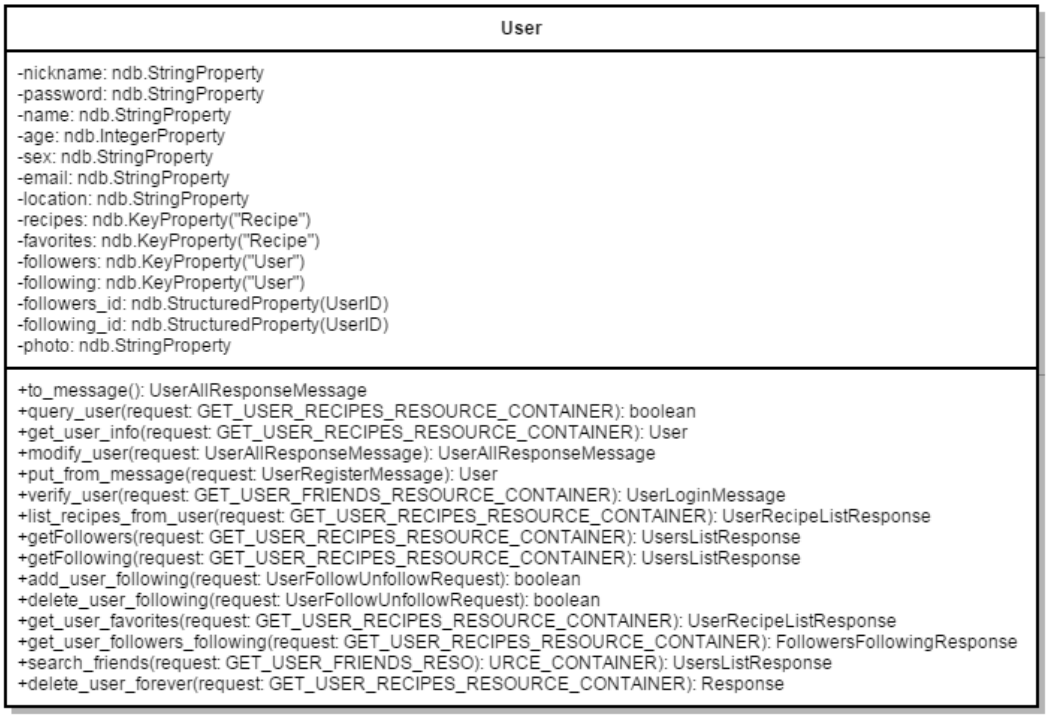


Ilustración 56: User

5.3.69 Ingredient

Es el modelo asociado a un ingrediente, con dos atributos definidos: nombre y cantidad. Este modelo carece de métodos dado que los ingredientes no se guardan en la Datastore, sino que se guardan instancias de ingredientes mediante el modelo correspondiente a la receta.

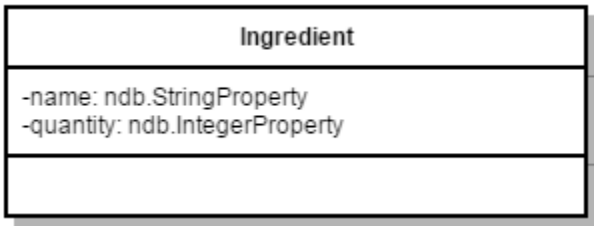


Ilustración 57: Ingredient

5.3.70 Elaboration

Es el modelo asociado a un paso de la elaboración, con dos atributos definidos: paso y foto. Este modelo carece de métodos dado que los pasos no se guardan en la Datastore directamente, sino que se guardan instancias de este modelo mediante el correspondiente a la receta.

5.3.71 Puntuación

Es el modelo asociado a una puntuación, con dos atributos definidos: identificador del usuario y los puntos. Este modelo carece de métodos dado que las puntuaciones no se guardan en la Datastore directamente, sino que se guardan instancias de este modelo mediante el correspondiente a la receta.

Ilustración 58: Puntuacion

5.3.72 UserID

Es el modelo asociado a un identificador, con un atributo definido: identificador del usuario. Este modelo carece de métodos dado que los identificadores no se guardan en la Datastore directamente, sino que se utiliza para crear listas de identificadores y guardarlas en instancias del modelo User.

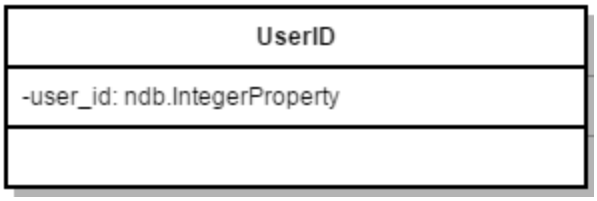


Ilustración 59: UserID

5.3.73 Recipe

Es el modelo asociado a cualquier receta, donde los atributos corresponden con los datos de la misma, y que contiene los métodos necesarios para hacer consultas, inserciones o borrados a la Datastore.

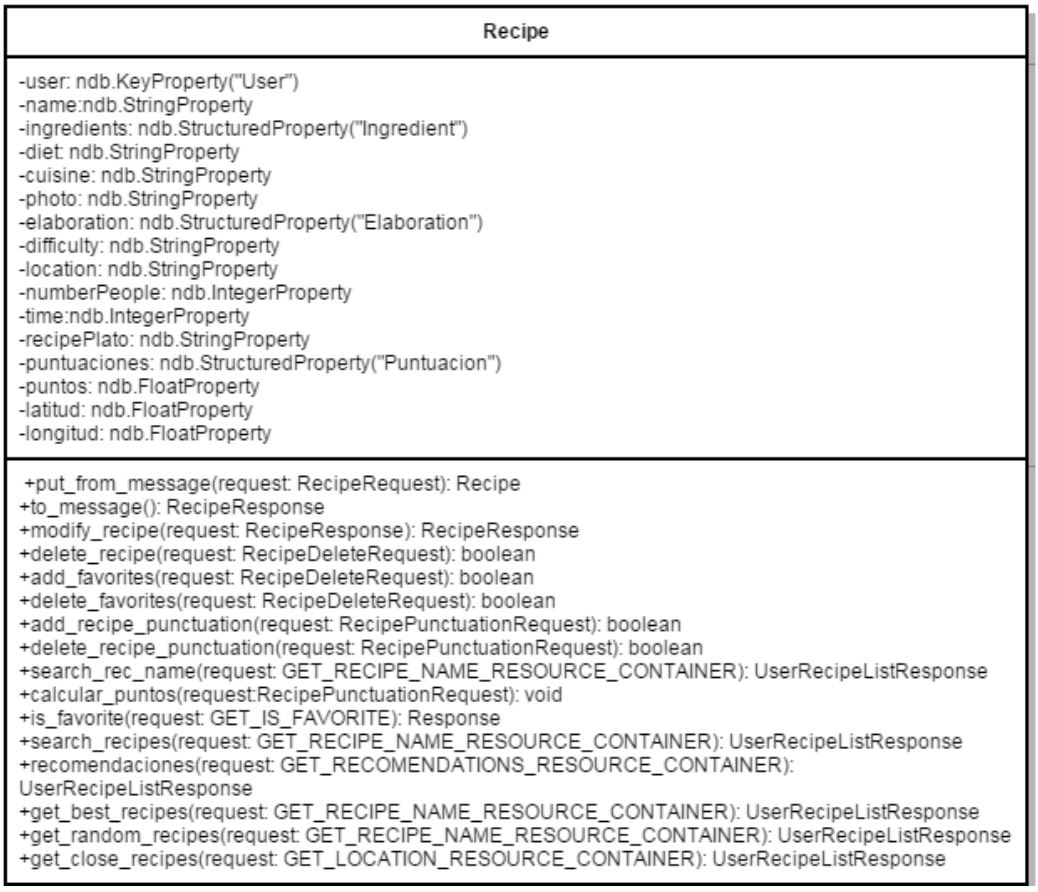


Ilustración 60: Recipe

5.3.74 FileUploadHandler

Es la clase asociada a guardar las imágenes en la Blobstore. Contiene un único método, post, que es el encargado de recibir la imagen, guardarla y devolver un identificador de la misma.

5.3.75 FileServeHandler

Es la clase asociada a descargar las imágenes en la Blobstore. Contiene un único método, get, que es el encargado de devolver la imagen del identificado facilitado.

5.3.76 ElaborationResponse

Es la clase de tipo Message correspondiente al modelo Elaboration, que se utiliza para conectar la Datastore y la parte cliente de Android. Contiene únicamente los atributos asociados a un paso: nombre y foto.

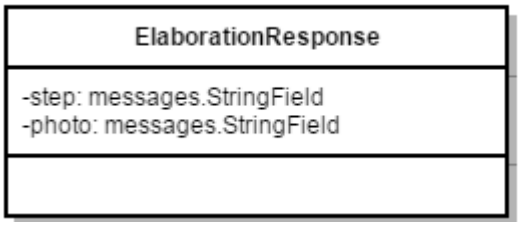


Ilustración 61: ElaborationResponse

5.3.77 IngredientResponse

Es la clase de tipo Message correspondiente al modelo Ingredient, que se utiliza para conectar la Datastore y la parte cliente de Android. Contiene únicamente los atributos asociados a un ingrediente: nombre y cantidad.

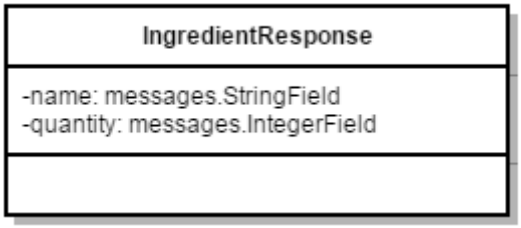


Ilustración 62: IngredientResponse

5.3.78 RecipeDeleteRequest

Es una clase de tipo Message que se utiliza para conectar la Datastore y la parte cliente de Android y realizar funciones como borrar una receta, o añadir una receta a favoritos. Contiene como atributos un identificador de un usuario y otro de una receta.

Ilustración 63: RecipeDeleteRequest

5.3.79 RecipePuntuationRequest

Es una clase de tipo Message que se utiliza para conectar la Datastore y la parte cliente de Android y realizar funciones como añadir una puntuación a una receta. Contiene como atributos un identificador de un usuario, otro de una receta y los puntos.

5.3.80 RecipeRequest

Es una clase de tipo Message que se utiliza para conectar la Datastore y la parte cliente de Android y pasarle al modelo correspondiente, en este caso el de Recipe, los datos necesarios para poder guardar una receta en la Datastore.

Ilustración 64: RecipeRequest

5.3.81 Response

Es una clase Message que simplemente tiene un campo de tipo booleano, y que sirve para devolver falso o verdadero a la parte cliente Android en consultas realizadas en la Datastore.

5.3.82 UrlMessage

Es la clase de tipo Message encargada de devolver la url correspondiente a la hora de guardar una foto en la Blobstore.

5.3.83 RecipeResponse

Es una clase de tipo Message que se utiliza para conectar la Datastore y la parte cliente de Android y devolver del modelo correspondiente, en este caso el de Recipe, los datos correspondientes a una receta.

Ilustración 65: RecipeResponse

5.3.84 UserFollowersFollowing

Es una clase de tipo Message que se únicamente tiene como atributo un identificador de usuario. Se utiliza para crear listas de identificadores de usuarios, correspondientes a los siguiendo o seguidores, de la clase UserAllResponseMessage.

5.3.85 UserFollowUnfollowRequest

Es una clase de tipo Message que simplemente conecta la parte cliente Android con la Datastore, y mediante la cual se envían dos identificadores: uno del usuario siguiendo y otro del usuario seguidor, con el fin de llevar a cabo funcionalidades como "Añadir usuario a seguidores" o "Dejar de seguir".

5.3.86 UserAllResponseMessage

Es la clase de tipo Message utilizada para devolver desde la Datastore a la parte cliente de Android todos los datos asociados a un usuario.

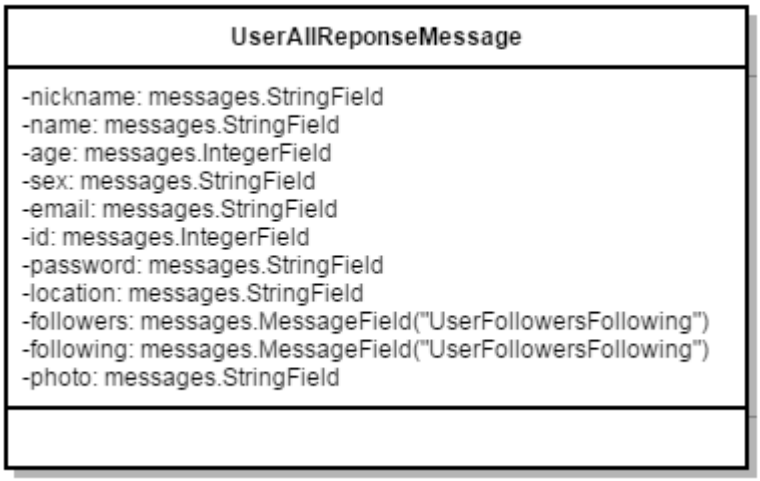


Ilustración 66: UserAllResponseMessage

5.3.87 UserRegisterMessage

Es la clase de tipo Message utilizada para enviar a la Datastore desde la parte cliente de Android todos los datos asociados al registro de un usuario, con el fin de guardarlo en la Datastore.

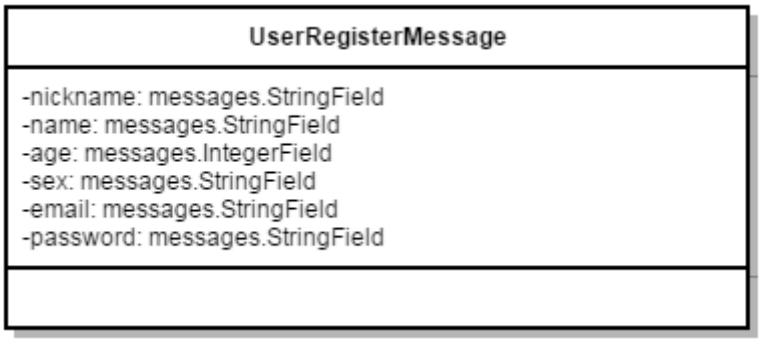


Ilustración 67: UserRegisterMessage

5.3.88 UserLoginMessage

Es la clase de tipo Message correspondiente a realizar la llamada de loguear un usuario. De esta manera, desde la parte cliente Android se devuelve los datos asociados al logueo: identificador del usuario, nickname y contraseña.

5.3.89 UserRecipeListResponse

Es la clase de tipo Message en la que se define una lista de recetas de tipo RecipeResponse. Instancias de esta clase son las que se mandan como resultado de cualquier consulta que devuelva un listado de recetas, por lo que únicamente tiene un atributo: la lista.

5.3.90 UsersListResponse

Es la clase de tipo Message en la que se define una lista de usuarios de tipo UserAllResponseMessage. Instancias de esta clase son las que se mandan como resultado de cualquier consulta que devuelva un listado de usuarios, por lo que únicamente tiene un atributo: la lista.

5.3.91 CookingStardustApi

Es la clase correspondiente a la API, donde están definidos todos los métodos a los que se accede mediante la parte cliente de Android, y mediante los cuales se mandan las instancias de las clases de tipo Message correspondientes a los métodos de los modelos, con el fin de hacer las consultas, inserciones o borrados pertinentes en la Datastore.

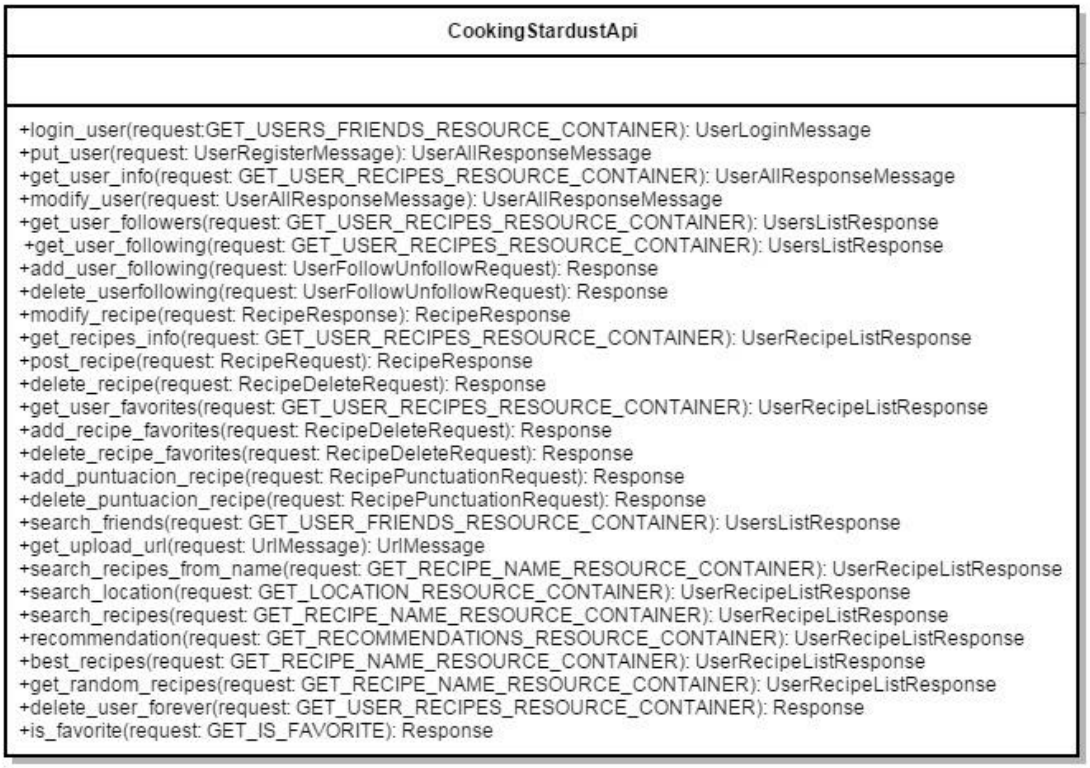


Ilustración 68: CookingStardustApi

6. DESARROLLO

En este apartado se va explicar el desarrollo que se ha realizado para llevar a cabo la aplicación Cooking Stardust. Se explicarán algunas de los problemas encontrados y las soluciones aplicadas a lo lago de proceso, tanto la parte cliente en Java, como la parte servidor en Python. El desarrollo de la aplicación se dividió en prototipos, con el fin de que la implementación fuera más llevadera y fácil a la hora de corregir errores.

6.1 App Engine, Google Endpoints y Datastore

Como se ha comentado en el apartado de Análisis, la base de datos implementada se trata de una base de datos no relacional, y además se hace uso de las herramientas AppEngine y Google Cloud Endpoints, así como de la Datastore. A continuación se explican los dos tipos de clases que se han creado para construir la parte servidor y base de datos de la aplicación.

6.1.1 Modelos: atributos y propiedades

Un modelo podría definirse como una clase que funciona como esquema de base de datos. Para este proyecto se han creado seis modelos diferentes (como se puede ver en el diagrama de clases), pero los más completos son Usuario y Receta.

```
class User(ndb.Model):
  nickname = ndb.StringProperty(required=True)
  password = ndb.StringProperty(required=True)
  name = ndb.StringProperty(required=True)
  age = ndb.IntegerProperty()
  sex = ndb.StringProperty()
  email = ndb.StringProperty(required=True)
  location = ndb.StringProperty()
  recipes = ndb.KeyProperty(kind='Recipe', repeated=True)
  favorites = ndb.KeyProperty(kind='Recipe', repeated=True)
  followers = ndb.KeyProperty(kind='User', repeated=True)
  following = ndb.KeyProperty(kind='User', repeated=True)
  followers_id = ndb.StructuredProperty(UserID, repeated=True)
  following_id = ndb.StructuredProperty(UserID, repeated=True)
  photo = ndb.StringProperty()
```

Ambas clases hacen uso la api de NDB y de toda su estructura. Existen varios tipos de propiedades para definir un modelo: ndb.StringProperty, ndb.IntegerProperty, FloatProperty, ndb.KeyProperty, ndb.StructuredProperty… Las dos primeras simplemente definen una

propiedad de tipo String y otra de tipo Integer respectivamente. Todas las propiedades tienen la opción *required*, que definirá si son obligatorias o no a la hora de crear una entidad de este tipo.

Asimismo, ndb.KeyProperty y ndb.StructuredProperty definen listas.

```
recipes = ndb.KeyProperty(kind='Recipe', repeated=True)
```

En este caso, cada usuario guarda una lista de identificadores únicos de sus recetas. La opción *kind*, define el tipo de objeto del cual se va a guardar los identificadores y el campo *repeated* define que será una lista en sí, dado que podrá haber varios.

```
following_id = ndb.StructuredProperty(UserID, repeated=True)
```

Ambas propiedades tienen la misma estructura y definen listas, pero en este caso, no existe campo *kind*, simplemente se pondrá el nombre de la clase de la que se quiera crear la lista.

Además de propiedades, las clases tienen métodos asociados a sí mismas, con el fin de realizar consultas, modificaciones, borrados y demás operaciones necesarias en la aplicación.

Para crear una entidad de cualquier modelo se puede utilizar bien el nombre del modelo o la opción *cls* en caso de que la entidad que queremos crear sea la que tiene el método asociado. Luego simplemente, habrá que poner una por una las propiedades con las que se reciben de una clase de tipo mensaje, que se explicarán más adelante.

```
@classmethod
def put_from_message(cls, message):
 user = cls(nickname=message.nickname, password=message.password,
name=message.name, age=message.age, sex=message.sex, email=message.email,
location="", recipes=[], favorites=[], followers=[], following=[],
followers_id=[], following_id=[], photo="nofoto")

if (User.query(User.nickname==user.nickname).count(limit=None))>0:
user = None
else:
user.put()
return user
```

Para hacer una inserción o una modificación de una entidad, se utiliza el método put(). Este método lo tienen asociado de manera interna todas las entidades que heredan de NDB.

Del mismo modo, para borrar una entidad de la Datastore, se utiliza el método key.delete(), que elimina la entidad a nivel de identificador.

```
@classmethod
def delete_user_forever(cls, message):
        user = ndb.Key('User', message.id).get()
        if (user != None):
                for recipe in user.recipes:
                        user.recipes.remove(recipe)
                        user.put()
                        recipe.key.delete()
                user.key.delete()
                return Response(there=True)
        else:
                return Response(there=False)
```

Las búsquedas son uno de los puntos más interesantes de la Datastore, que ofrece tanto la opción de crear queries propias mediante lenguaje GQL o simplemente, hacer uso de los métodos internos asociados a la NDB API.

```
@classmethod
def search_recipes(cls, message):
        busqueda = message.name.split("-")
        query = Recipe.query()
        if (busqueda[2] != ""):
                query = query.filter(Recipe.difficulty == busqueda[2])
        if (busqueda[3] != ""):
                query = query.filter(Recipe.cuisine == busqueda[3])
        if (busqueda[4] != ""):
                query = query.filter(Recipe.recipePlato == busqueda[4])
        if (busqueda[5] != ""):
                query = query.filter(Recipe.diet == busqueda[5])

        recipes = query.fetch()
```

Con el método query() se devuelve en forma de consulta todas las entidades pertenecientes a ese modelo, sin ningún tipo de filtro.
En este caso, como se quería filtrar por varios campos diferentes, se utilizó la opción filter() y se fue filtrando campo por campo de manera ordenada. De esta manera, se empieza con una consulta que tiene todas las entidades del modelo Recipe y sobre esa misma consulta se van filtrando los campos que se han definido: dificultad, tipo de cocina, tipo de plato y dieta, sin tener que hacer en ningún momento uso de lenguaje GQL.

Una vez se tiene la última consulta con todos los filtros aplicados, se utiliza el método fetch() para devolver los elementos en forma de lista.

Finalmente, también hay una opción order(), que sirve para ordenar elementos dentro una query por el campo que se le indique y que se utiliza en este proyecto para devolver ordenadas por puntuación las diez mejores recetas.

6.1.2 Google Cloud Endpoints

Los Endpoints son un grupo de herramientas y librerías que se utilizan para generar APIs y librerías cliente para facilitar el acceso a la información de la Datastore desde cualquier aplicación, en este caso desde Android. La Endpoint API implementada en este proyecto será la encargada de gestionar las llamadas y las respuestas desde Android hacia la Datastore y viceversa. Una de sus librerías es Google Protocol RPC y en este caso se ha utilizado para implementar tipos de mensajes y métodos remotos que facilitan la comunicación entre la Datastore y la parte Android.

Dichas clases serán el núcleo intermedio entre las llamadas que se reciben desde la parte cliente a través de la Endpoint Api y lo que se devuelve desde los métodos de los modelos. Es decir, no se devolverán (ni se enviarán) entidades directamente a Android, sino que se utilizarán clases que hereden de Message para parsear lo que se quiera devolver desde los métodos de los modelos definidos y serán instancias de estas clases las que se manden a la parte Android mediante los métodos implementados en la Endpoint Api.

```
class UserAllResponseMessage(messages.Message):
    nickname = messages.StringField(1)
    name = messages.StringField(2)
    age = messages.IntegerField(3)
    sex = messages.StringField(4)
    email = messages.StringField(5)
    id = messages.IntegerField(6)
    password = messages.StringField(7)
    location = messages.StringField(8)
    followers = messages.MessageField(UserFollowersFollowing, 9,
repeated=True)
    following = messages.MessageField(UserFollowersFollowing, 10,
repeated=True)
    photo = messages.StringField(11)
```

La estructura es parecida a la de cualquier modelo que herede de NDB, salvo que la terminología es diferente. En este caso, en vez de tener StringProperty o IntegerProperty, se tiene StringField o IntergerField, pero la función en realidad es la misma. En este caso, además, todos los atributos tienen el número que les identifica. En el caso de las listas, se utiliza el

atributo MessageField seguido de la clase de la que pertenecerán los objetos de la lista, así como la opción *repeated.*

La Endpoint Api está formada por un servicio RPC y uno o varios métodos endpoints. Estos métodos serán los encargados de acceder a los modelos de la Datastore y de recibir y/o enviar las clases Message explicadas justo anteriormente. La Api se implementa de la siguiente manera, definiendo una subclase de remote.Service:

```
@endpoints.api(name=cookingstardust, version='v1', description='Cooking
Stardust')
class CookingStardustApi(remote.Service):
```

Y a continuación, se implementan todos los métodos endpoints necesarios pertenecientes a esa clase. Los métodos pueden ser de dos tipos: GET (si sólo se van a devolver datos, por ejemplo, en una búsqueda) o POST (si se va a guardar o eliminar datos de la Datastore).

```
@endpoints.method(UserRegisterMessage, UserAllResponseMessage,
path='users', http_method='POST', name='user.post')
 def put_user(self,request):
        user = User.put_from_message(request)
        if (user == None):
               raise endpoints.BadRequestException('User already exists
in the      database!')
        else:
               return user.to_message()
```

Este sería el método correspondiente a registrar un usuario. Cuando se define el método, se definen varios opciones: la primera es el tipo de mensaje que se recibe, en este caso UserRegisterMessage y el segundo, el que se va a devolver: UserAllResponseMessage. La opción path define en qué ruta va a estar ese método en la URL, y el HTTP_METHOD el tipo de método que es. Como se va a guardar un usuario, es de tipo POST.

Una vez en el método, se llama al método put_from_message() del modelo User y se le pasa el mensaje recibido para guardarlo en la Datastore. Si es correcto, la API devolverá a la aplicación Android los datos del usuario en forma de instancia de la clase UserAllResponseMessage (que hereda de Message), pero en caso de que el nickname ya exista en la Datastore, se devolverá un error.

Por otro lado, en los métodos de tipo GET, lo que se recibe no es ninguna clase de tipo Message, sino que es un Resource Container. Se definen de la siguiente manera:

```
GET_USER_RECIPES_RESOURCE_CONTAINER =
endpoints.ResourceContainer(message_types.VoidMessage,
id=messages.IntegerField(1, variant=messages.Variant.INT32,required=True))
```

En este caso, se define que se recibirá un campo de tipo Integer como llamada. Un método de tipo GET sería el siguiente

```
@endpoints.method(GET_USER_RECIPES_RESOURCE_CONTAINER,
UserAllResponseMessage, path='profile/{id}', http_method='GET',
name='user.get_user')
 def get_user_info(self,request):
        user = User.get_user_info(request)
        if (user == None):
               raise endpoints.BadRequestException('User wrong')
        else:
               return user.to_message()
```

La estructura de un método endpoint de tipo GET es igual que la de un método endpoint de tipo POST, salvo porque se define el Resource Container anteriormente mencionado como parámetro de entrada.

Finalmente, cuando se tienen todos los métodos endpoints definidos, es necesario crear el servicio API para poder utilizarlos. En este caso, se ha creado de la siguiente manera, pasándole como parámetro el nombre de la clase que hereda de service.Remote:

```
APPLICATION = endpoints.api_server([CookingStardustApi],restricted=False)
```

Además de los ficheros que se hayan definido para implementar todas estas clases necesarias, para poder acceder a la API es necesario configurar el fichero app.yaml, que tiene la siguiente estructura.

```
application: prueba-keld
version: 1
runtime: python27
threadsafe: true
api_version: 1

handlers:
# Endpoints handler
- url: /_ah/spi/.*
  script: cookingstardust_api.APPLICATION
  secure: always
- url: .*
  script: prueba.app

libraries:
- name: pycrypto
  version: latest
- name: endpoints
  version: latest
```

En el apartado handlers de este fichero, se tiene que especificar dónde está creado el servicio de la API. En este caso está creado en el fichero Cookingstardust_api y en el apartado APPLICATION, como se ha podido ver un poco más arriba. Una vez configurado este fichero, habrá que incluirlo en la misma carpeta que los demás ficheros de Python para poder ejecutar la API desde la aplicación de App Engine.

6.1.3 Blobstore

Como se comentó en el apartado de Análisis, no se pueden guardar imágenes en la Datastore, y para ello se utiliza la Blobstore junto con webapp2 framework, que incluye las clases necesarias para manejar el formulario de la subida y la descarga de imágenes a través de la Blobstore. A nivel de cliente, se ha utilizado la librería Ion en Android para facilitar la subida y descarga de imágenes. El primer paso para subir imágenes es conseguir la URL (de la Blobstore) a la que subirlas. Para eso, existe un método predefinido que devuelve una URL totalmente aleatoria:

```
url = blobstore.create_upload_url('/upload')
```

Este método devolverá una URL muy similar a esta: http://0.0.0.0:9080/_ah/upload/ag9kZXZ-cHJ1ZWJhLWtlbGRyIgsSFV9fQmxvYlVwbG9hZFNlc3Npb25fXxiAgICAgMueCQw, que será contra la cual se hará la llamada para poder guardar la imagen en la Blobstore. La clase que maneja la subida de imágenes hereda de blobstore_handlers.BlobstoreUploadHandler y su

método post devuelve como parámetro el identificador de la imagen subida. En cuanto a la descarga de imágenes, la clase que la maneja hereda de blobstore_handlers.BlobstoreDownloadHandler.

Todas las imágenes se descargarán de la siguiente URL: http://localhost:9080/serve/ y el identificador que se haya guardado a la hora de subir la foto. Estas dos clases tienen su propio servicio webapp2 para ejecutarse, y no pasan por la Endpoints API definida anteriormente, por lo cual su funcionamiento es totalmente independiente.

```
MEDIA_ROUTES = [
       ('/upload', FileUploadHandler),
       ('/serve/' + '([^/]+)?' , FileServeHandler)]
app = webapp2.WSGIApplication(MEDIA_ROUTES, debug=True)
```

6.1.4 Documentos, índices y Geopoints

Como ya se comentó en el apartado de Análisis, para la búsqueda de recetas por localización es necesario utilizar documentos e índices. De este modo, existe un único índice llamado Localizaciones, que será dónde se guardan todos los documentos asociados a todas las recetas de las que se quiera guardar datos de localización. De este modo, cada vez que se guarde una receta con localización, hay que recuperar el índice:

```
index = search.Index(name=INDEX)
```

Entonces, creamos un elemento de tipo Geopoint con las coordenadas y una lista que incluya el identificador de la receta y el Geopoint creado.

```
geopoint = search.GeoPoint(latitude=message.latitude,
longitude=message.longitude)
fields = [search.TextField(name='key', value=str(recipe.key.id())),
search.GeoField(name='coordenadas', value=geopoint)]
```

Finalmente, creamos el document en sí y le agregamos los parámetros, para después añadir el documento a nuestro índice.

```
recipe_document = search.Document(fields=fields)
index.put(recipe_document)
```

De esta manera, cada vez que se quiera buscar recetas por localización, se tendrá que crear una query con el método *distance* que haga uso de la Search API y aplicarla a todos los documentos de nuestro índice.

```
dist_query = "distance(coordenadas, geopoint({}, {}))".format(latitud,
longitud)
query = dist_query + " < {}".format(distancia)
query_results = search.Query(query_string=query)
docs = search.Index(name=INDEX).search(query_results)
```

De esta manera, en la variable docs tendríamos los documentos que contienen los identificadores de receta que ha devuelto la búsqueda, y cuya distancia es menor a la establecida por el usuario.

Por otra parte, todos los métodos de la API son accesibles por navegador mediante el puerto configurado en el App Engine y la URL http://localhost:9080/_ah/api/explorer.

Una vez creados todos los métodos y clases necesarias para la API, se generaron las librerías java correspondientes para Android mediante el script endpointscfg.py facilitado por Appengine. Dichas librerías son las correspondientes a las clases que heredan de Message y a la Endpoint API en sí, y serán instancias de esas clases las que se envíen desde Android a la API y las que se reciban de igual manera. Asimismo, la clase java correspondiente a la Api, *Cookingstardust*, contiene todos los métodos y servicios necesarios para poder hacer las llamadas desde Android. Son clases generadas de manera automática, y por ende, un tanto complejas, por lo cual Google no recomienda modificarlas.

En los siguientes apartados se procederá a explicar cómo se realiza la conexión entre la API y la aplicación Android.

6.1.5 Seguridad y restricción de la aplicación

Google App Engine permite dirigir todo el tráfico de las llamadas de la API haciendo uso del SSL mediante HTTPS. De esta manera, una vez desplegada la aplicación, se nos facilita una URL única y global, mediante la cual se pueden realizar llamadas a la API desde la aplicación Android desde cualquier tipo de red. En este caso, la URL facilitada es la siguiente:
https://prueba-keld.appspot.com

Para garantizar una conexión segura mediante HTTPS, únicamente es necesario añadir el parámetro *secure* en el fichero de configuración app.yaml con el siguiente valor.

```
handlers:
# Endpoints handler
- url: /_ah/spi/.*
  script: pruebalogin_api.APPLICATION
  secure: always
```

De esta manera, nos aseguramos que todas las llamadas a la API se realizan mediante HTTPS, por lo cual tanto la información entrante como la saliente es encriptada para enviarla, y desencriptada después para su acceso.

Por otro lado, para el desarrollo de Cooking Stardust se había pensado en implementar OAuth 2.0. La primera versión que se consultó en octubre del 2013 permitía implementar esta funcionalidad, mediante la cual se restringe el acceso a la API, sin tener que loguearse de manera obligatoria con un usuario de una cuenta de Google. Actualmente, la API ha sido modificada, y el logueo con usuario de Google es obligatorio, por lo cual, finalmente, se descartó implementarlo. Aún así, y gracias al uso del SSL, toda la información será tratada de manera segura a lo largo de las llamadas desde la aplicación Android hacia la API y finalmente, la Datastore.

6.2 Conectar la aplicación con la API

Para conectar nuestra aplicación Android con la API lo primero que se hizo fue importar las clases generadas mediante el script predefinido del App Engine al proyecto y añadir la siguiente línea al manifiesto, que permite el acceso a Internet:

```
<uses-permission android:name="android.permission.INTERNET">
```

Una vez configurado el proyecto, todas las llamadas se ejecutan a través de un servicio que hace uso de la clase java de la API, llamada Cookingstardust. De esta manera, sólo es necesario construir el servicio y ejecutarlo, dado que toda la configuración de la API viene establecida de manera predefinida en las clases generadas. Con el fin de tener todas las llamadas centralizadas y evitar tener que crear el servicio cada vez que se quisiera hacer una llamada, se creó una clase java Api cuya funcionalidad es recoger todas las llamadas hacia la API. A continuación se muestra un gráfico de cómo están estructuradas las llamadas a la Endpoints API desde Android.

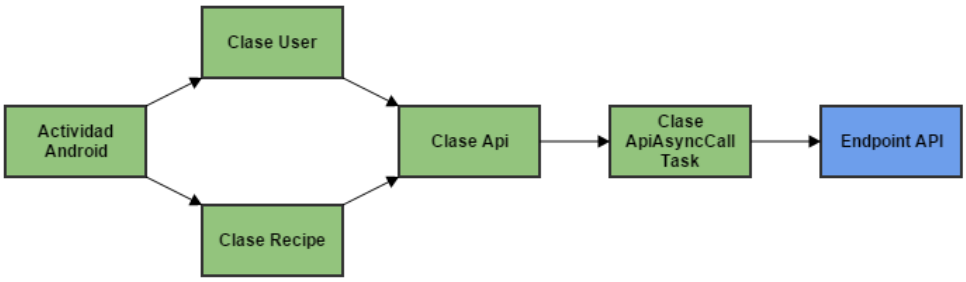


Ilustración 69: Estructura de las llamadas a la Endpoints API

De esta manera, desde cualquier actividad de la aplicación se llama a la clase User o a la clase Recipe (dependerá de la funcionalidad), y estas serán las encargadas de inicializar la api (construir el servicio) y mandar los datos necesarios para que desde la clase Api, el método correspondiente a la llamada que se quiera realizar sea ejecutado.

Como se ha comentado, la clase Api es la encargada de realizar todas las llamadas. Por eso mismo, su constructora implementa el servicio de la api Cookingstardust, que será el necesario para poder ejecutar las llamadas.

```
private Api(HttpTransport transport, GsonFactory json) {
Cookingstardust.Builder builder = new Cookingstardust.Builder(transport,
json, credentials);
builder.setRootUrl(SERVICE_URL);
service = builder.build();
}
```

Asimismo, esta clase Api contiene otras dos clases internas llamadas Api.User y Api.Recipe, que tienen un atributo llamado request. Este atributo es una instancia de las clases Cookingstardust.User y Cookingstardust.Recipe respectivamente y contienen todos los métodos asociados a su condición y equivalentes a los métodos endpoints definidos en la clase Pyhton en la parte servidor. Del mismo modo, el método Cookingstardust.user() o Cookingstardust.recipe() es la colección completa de todos esos métodos.

```
public class User {
       private Cookingstardust.User request;
       public User() {
         this.request = service.user();
     }

public class Recipe {
private Cookingstardust.Recipe request;

public Recipe() {
this.request = service.recipe();
}
```

Todas las llamadas se ejecutan haciendo uso de la clase AsyncTask, por lo cual se implementó otra clase, llamada ApiCallAsyncTask, que es la encargada de ejecutarlas en segundo plano y de devolver el resultado mediante la interfaz Callback.

```
public void list_recipes(Long lon,
Callback<CookingstardustMessagesUserRecipeListResponse> callback) {
try {
     Cookingstardust.Recipe.List call = request.list(lon);
     new ApiCallAsyncTask<CookingstardustMessagesUserRecipeListResponse>(
              call, callback).execute();
  } catch (IOException e) {
  // TODO Auto-generated catch block
     callback.onCompleted(e, null);
  }
}
```

Este método, por ejemplo, sería el encargado de crear la llamada correspondiente para listar las recetas de un usuario y se ejecutaría desde la clase Api.Recipe. El parámetro que va a ejecutar el servicio es call, y está definido de la siguiente manera: se le asigna el método correspondiente a listar recetas de la clase Cookingstardust.Recipe.List y se le pasa el parámetro correspondiente, en este caso, un identificador de tipo Long.

```
Cookingstardust.Recipe.List call = request.list(lon);
```

Y después se ejecutaría en segundo plano gracias a la clase ApiCallAsyncTask.

```
private class ApiCallAsyncTask<T> extends AsyncTask<Void, Void, T> {
private Callback<T> callback;
private CookingstardustRequest<T> call;
public ApiCallAsyncTask(CookingstardustRequest<T> call, Callback<T>
callback) {
        this.callback = callback;
        this.call = call;
        this.call.setDisableGZipContent(true);
 }
 @Override
 protected T doInBackground(Void... voids) {
        try {
                return call.execute();
        } catch (IOException e) {
                callback.onCompleted(e, null);
                return null;
                }
        }
 protected void onPostExecute(T result) {
        if (result != null) {
                callback.onCompleted(null, result);
        }
 }
```

A través de los callback.onCompleted(), se devolverían el resultado de la llamada a través de las clases, hasta llegar a User o Recipe, donde se pasaría a parsear el resultado y devolverlo a la actividad correspondiente. A continuación se explica en detalle cómo sería una llamada completa, desde la actividad hasta la propia Datastore, mediante la explicación de la funcionalidad Registrar Usuario.

6.2.1 Registrar Usuario

A la actividad Registrar se accede pulsando el botón Registrar de la clase principal. Se trata de un formulario de registro básico, en el que son necesarios introducir nickname, nombre, contraseña y correo electrónico.

Si el usuario pulsa Aceptar sin haber introducido estos campos, se le muestra un aviso. En caso contrario, se procede a guardar el usuario en la Datastore, de la siguiente manera:

```
User.create(nick, pass, name, email, new Callback<User>(){
 new Callback<User>(){
        @Override
        public void onCompleted(Exception e, User result){
               if (e!=null){
                  toast("El nickname ya existe en la base de datos");
               }else{
                  if (result!=null){
                      toast("User " + result.getName() + "creado!");
                  }
               }
        });
}
```

En este caso, Registrar Usuario es una funcionalidad propia de un usuario, por lo cual la llamada será dirigida a través de la clase User, encargada más tarde de parsear el resultado obtenido. Además, se utiliza la interfaz Callback para devolver el resultado, por lo que si el resultado devuelto no es nulo, se saca un mensaje de que el usuario ha sido guardado correctamente.

El siguiente paso es el método create(), en la clase User. Lo primero que se hace en este método es asignar la instancia de la Api en una variable, con el fin de inicializar el servicio de la api Cookingstardust. Acto seguido, se crea una instancia de la clase Cookingstardust MessagesUserRegisterMessage, que es la clase java equivalente a la clase Python UserRegisterMessage que se utiliza para comunicar la API con la datastore. Recordemos que estas clases eran las que se utilizaban como parámetros de entrada y salida en los endpoint métodos, por lo cual la variable user, a la que se le asignan todos los datos que se le han pasado desde la clase Registrar, será lo que se envíe a través del servicio a la API y de ahí a la Datastore.

```
public static void create(String nick, String pass, String name, String
email, final Callback<User> callback) {

Api api = Api.getInstance();
CookingstardustMessagesUserRegisterMessage user = new
CookingstardustMessagesUserRegisterMessage();
user.setNickname(nick);
user.setPassword(pass);
user.setName(name);
user.setEmail(email);

api.service().user().post(user, new
Callback<CookingstardustMessagesUserAllResponseMessage>() {
 @Override
 public void onCompleted(Exception e,
        CookingstardustMessagesUserAllResponseMessage result) {
        if (e != null){
               callback.onCompleted(e, null);
        }else{
               callback.onCompleted(null,
               User.parse(result));
               }
        });
 }
```

Finalmente, se realiza la llamada a la clase api, haciendo uso del servicio inicializado y el método interno user(), que devuelve la colección de métodos pertenecientes a Api.User. Entre ellos, se encuentra implementado el método post(), que será el encargado de la llamada para guardar un usuario. Del mismo modo que en la clase Registrar, se devolverá a la clase User el resultado de la API, en forma de instancia de CookingstardustMessagesUserAllResponseMessage (código JSON), y será en esta misma clase User donde se parsee el resultado, mediante el método parse(). La clase CookingstardustMessagesUserAllResponseMessage es la equivalente a la clase python UserAllResponseMessage, y es la que se devuelve del método endpoint correspondiente a guardar un usuario.

```
public static User parse(CookingstardustMessagesUserAllResponseMessage
json) {
       User user = new User();
       user.nickname = json.getNickname();
       user.name = json.getName();
       user.age = json.getAge();
       user.sex = json.getSex();
       user.email = json.getEmail();
       user.id = json.getId();
       user.password = json.getPassword();
       user.location = json.getLocation();
   user.photo = json.getPhoto();
              if (json.getFollowers() != null) {
                     ArrayList<Long> followers = new ArrayList<Long>();
                     for (int i = 0; i < json.getFollowers().size(); i++) {
                            Long id =
json.getFollowers().get(i).getFollId();
                            followers.add(id);
                     }
                     user.followers = followers;
              }
              if (json.getFollowing() != null) {
                     ArrayList<Long> following = new ArrayList<Long>();
                     for (int i = 0; i < json.getFollowing().size(); i++) {
                            Long id =
json.getFollowing().get(i).getFollId();
                            following.add(id);
                     }
                     user.following = following;
              }

              return user;
       }
```

Finalmente, en la clase Api está implementado el método post() asociado a esta funcionalidad.

```
public void post(CookingstardustMessagesUserRegisterMessage user,
Callback<CookingstardustMessagesUserAllResponseMessage> callback) {
try {
     Cookingstardust.User.Post call = request.post(user);
   new ApiCallAsyncTask<CookingstardustMessagesUserAllResponseMessage>(
      call, callback).execute();
          } catch (IOException e) {
              callback.onCompleted(e, null); }}
```

Una vez creada la llamada, call, correspondiente, se le pasa como parámetro a la clase encargada de ejecutarla en segundo plano, ApiCallAsyncTask y cuya funcionalidad ha sido explicada en el punto anterior.

Como se ha comentado, la clase Cookingstardust (la clase Java de la Endpoint API), tiene toda la configuración de la Endpoint API asignada, por lo cual al ejecutar la llamada, hace uso de esa configuración (donde se especifica URL, tipo de método…) y simplemente se accede al método endpoint de la API asociado a esa llamada, en este caso, put_user():

```
@endpoints.method(UserRegisterMessage, UserAllResponseMessage,
path='users',http_method='POST',name='user.post')
       def  put_user(self,request):
              user = User.put_from_message(request)
              if (user == None):
raise endpoints.BadRequestException('User already exists in the database!')
              else:
                     return user.to_message()
```

Desde aquí, se accedería al modelo User, donde se procedería a guardar el usuario, o a devolver un error en caso de que el nickname ya existiera en la Datastore:

```
def put_from_message(cls, message):
user = cls(nickname=message.nickname, password=message.password,
name=message.name, age=None, sex="", email=message.email, location="",
recipes=[], favorites=[], followers=[], following=[], followers_id=[],
following_id=[], photo="nofoto")
if (User.query(User.nickname==user.nickname).count(limit=None))>0:
              user = None
       else:
              user.put()
return user
```

Una vez guardado el usuario en la Datastore, se devuelve el resultado, por lo cual hay que parsear la instancia user del modelo User en la clase Message correspondiente, en este caso, UserAllResponseMessage, que será la que se envíe de vuelta. Para parsear se utiliza el método to_message(), que simplemente devuelve una instancia de la clase UserAllResponseMessage con los datos del usuario en cuestión.

Esta instancia será devuelta a la clase ApiAsyncCallTask, y de ahí a la clase Api, y de ahí a la clase User, mediante los métodos onCallback(). En esta clase se parseará el resultado con el método explicado un poco más arriba, y se procederá a enviar una instancia de la clase User

como respuesta a la actividad Registrar, en caso de que la llamada haya sido exitosa. Y con ese paso final, terminaría el ciclo de vida de una llamada a la Datastore.

Cabe destacar que todas las llamadas a la Datastore, tanto si son como para guardar datos como para simplemente devolverlos, se hacen de la misma manera. La única diferencia es que todas aquellas funcionalidades que pertenezcan más a ser una receta, las llamadas se harán desde la clase Recipe, donde también hay un método de parseo parecido al que hay en la clase User para tratar los datos JSON devueltos. Asimismo, la única diferencia es el nombre del método que realiza la llamada y los parámetros que se le pasan, dado que la estructura es la misma que la explicada con este ejemplo.

6.3 Elementos de navegación

En este apartado se van a explicar todos aquellos elementos que se han utilizado a la hora de implementar y mejorar la navegación por la aplicación, ya sean genéricos, o personalizados únicamente para las funcionalidades de Cooking Stardust.

6.3.1 Navigation Drawer

Dado que se quería diseñar una aplicación más o menos moderna, se pensó en utilizar un menú de este tipo con el fin de mejorar la usabilidad de la actividad principal. De este modo, la actividad Home, a la que accede el usuario según se identifica, no es simplemente un menú con todas las opciones de la aplicación, sino que también es una pantalla de presentación, desde la cual se le muestran recetas aleatorias, dejando el menú escondido en la parte izquierda de la pantalla, como se puede ver en la imagen de la derecha.

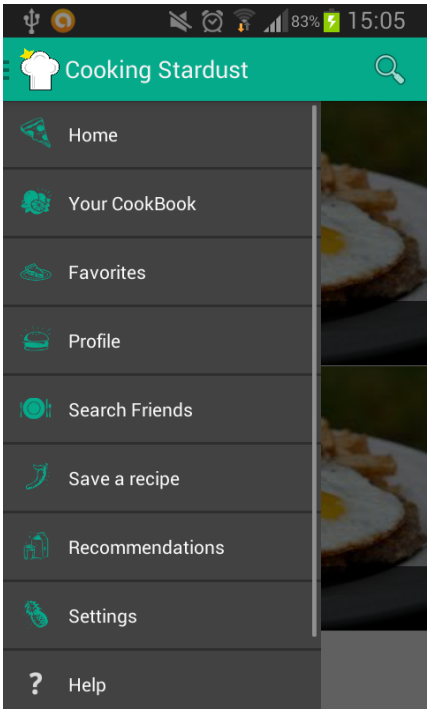


Ilustración 70: Navigation Drawer

Para definir este menú, basta con añadir el widget Drawer Layout al fichero XML de la interfaz de la actividad, y añadir un Listview para crear agregar el listado de opciones del menú:

```
<android.support.v4.widget.DrawerLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <ListView
        android:id="@+id/lista"
        android:layout_width="240dp"
        android:layout_height="fill_parent"
        android:layout_gravity="start"
        android:background="#424242"
        android:choiceMode="singleChoice"
        android:divider="#2E2E2E"
        android:dividerHeight="2dp"
        android:textColor="#424242" />

</android.support.v4.widget.DrawerLayout>
```

Una vez definido el menú en el layout, sólo queda agregarlo a la actividad. Para ello, basta con inicializar la instancia del menú Navigation Drawer con el listado de objetos que se quiera añadir, usando para ello un adaptador, de la siguiente manera:

```
NavAdapter = new NavigationAdapter(this, NavItems);
NavList.setAdapter(NavAdapter);
```

Un adaptador es un objeto que dado un listado de elementos, es capaz de crear una vista con cada uno de ellos. Generalmente se suele utilizar un adaptador con un array de strings, pero para este caso se creó un adaptador personalizado, llamado NavigationAdapter. La razón de esto es que se optó por personalizar las filas que forman el listview, mediante un layout personalizado con un icono y un texto. De este modo, el listado que se le pasa al adaptador, NavItems, es un listado de objetos de tipo ItemObject, que simplemente define un objeto con un icono y un nombre.

El uso y la creación de adaptadores personalizados se explicarán en el siguiente punto de Elementos de Navegación, dado que se han utilizado varios a lo largo del desarrollo de la aplicación.

Para que al seleccionar una opción de la lista se vaya a la funcionalidad correspondiente, se ha hecho que la el listview implemente la interfaz OnItemClickListener.

```
NavList.setOnItemClickListener(new AdapterView.OnItemClickListener() {
@Override
public void onItemClick(AdapterView<?> parent, View view, int position,
long id) {
               showFragments(position);
       }
});
```

De este modo, cada vez que se seleccione en alguna de las opciones, se llama al método showFragments pasándole la posición de la lista, y este método será el encargado de inicializar la actividad correspondiente a la posición. Por ejemplo:

```
private void showFragments(int position) {
ListFragment listFragment = null;
 Intent i;
       switch (position) {
   case 5:
               // Guardar receta
               i = new Intent(this, UploadRecipe.class);
               startActivity(i);
               break;
```

6.3.2 Adaptadores

Como se ha dicho anteriormente, un adaptador es un objeto que dado un listado de elementos, es capaz de crear una vista con cada uno de ellos. Para la realización de esta aplicación, se han utilizado adaptadores genéricos y personalizados. Los primeros se implementan simplemente inicializándolos con un array de strings, pero en este punto se explicarán los segundos, dado que son bastante más interesantes y conllevan más pasos a seguir.

Se han utilizado adaptadores en un varias funcionalidades, sobre todo, aquellas orientadas a mostrar un listado de objetos: un listado de recetas (ya sean del recetario propio del usuario, como sus favoritas o el resultado de una búsqueda, todos implementan el mismo adaptador), un listado de usuario, o simplemente implementar el listado de opciones del menú de Navigation Drawer de la pantalla principal, como el ejemplo mostrado en la Ilustración número 13.

Para crear un adaptador personalizado son necesarios tres elementos diferentes:

- **Layout**. Se necesita especificar un layout para definir de manera visual cómo se quiere diseñar cada fila del listview. Para este proyecto se han diseñado cuatro layouts diferentes, dependiendo de las necesidades de cada funcionalidad:
    - Mostrar listado de recetas: El layout está compuesto por un ImageView cuadrado a la izquierda y dos TextViews a la derecha, uno debajo de otro. De esta manera, cada elemento del listado de recetas mostrará una foto pequeña de la receta, su nombre, y el tipo de cocina. Se utiliza en varias funcionalidades: Ver Recetas, Ver Favoritas, Ver Resultados de Búsqueda, Ver Recomendaciones…
    - Mostrar recetas aleatorias: este layout sólo se utiliza en la actividad principal, Home, donde se muestra un listado de recetas aleatorias. Por cada una de ellas, el layout está definido de la siguiente manera: un ImageView con la foto de la receta que ocupe todo el ancho y alto de la fila, y en la parte de abajo, un TextView con el nombre de la receta.
    - Mostrar listado de usuarios: este layout se utiliza en las funcionalidades de Ver Siguiendo, Ver Seguidores y Buscar Amigos. El layout está compuesto por un ImageView cuadrado a la izquierda y un TextViews a la derecha, en el centro. De esta manera, en el listado de usuarios se podrá ver una pequeña foto de su perfil y su nickname en la aplicación.
    - Opciones del Navigation Drawer: se ha definido un layout con un icono (ImageView) en forma de comida a la izquierda y a la derecha el nombre de la opción, mediante un TextView.

Un ejemplo de cómo estaría definido en el XML el layout es el siguiente:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="48dp" >
    <ImageView
        android:id="@+id/icon"
        android:layout_width="25dp"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_marginLeft="12dp"
        android:layout_marginRight="12dp"
        android:layout_centerVertical="true" />
    <TextView
        android:id="@+id/title_item"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:layout_toRightOf="@id/icon"
```

```
        android:textColor="#FAFAFA"
        android:gravity="center_vertical"
        android:paddingTop="15dp"
        android:paddingRight="40dp"
        android:layout_centerVertical="true"
        android:paddingBottom="15dp" />

</RelativeLayout>
```

- **La clase que defina el objeto**: es necesario crear una clase equivalente al layout que se quiera implementar, de tal manera que luego se tenga una lista de objetos de esa clase que pasarle al adaptador. Dado que los layouts implementados en Cooking Stardust son bastante parecidos, en este caso se han reutilizado las clases de objetos, definiendo en cada funcionalidad los atributos que fueran necesarios. Por ejemplo, para crear el listado de recetas, se ha creado una clase java llamada RecipeItem que simplemente contiene tres atributos: nombre, cocina e icono. Dicha clase también se ha utilizado para implementar la lista de recetas aleatorias, sólo que en ese caso, el atributo cocina no se inicializa con ningún dato, dado que no aparece en el layout de recetas aleatorias.

- **La clase del adaptador**: es la clase que implementa el adaptador de por sí. Se han creado varias clases de adaptadores personalizados en este proyecto, pero todas tienen la misma estructura. La clase adaptador recibe como parámetro un ArrayList de la clase que defina el objeto y devuelve una vista. Por ejemplo:

```
convertView = inflator.inflate(R.layout.nav_drawer, null);
view.title_item.setText(itm.getTitle());
view.icon.setImageResource(itm.getIcon());
convertView.setTag(view);
```

  Por cada uno de los elementos de la lista, infla el layout correspondiente con los datos del elemento, en este caso el nombre y el icono.

Independientemente de estos tres elementos, también es necesario tener un listview. Generalmente, este listview se definirá en el layout de la interfaz de la actividad que esté implementando el adaptador, y será sobre este listview sobre el cual se inicialice el adaptador. De esta manera, una vez se tiene el listado de objetos, sólo hay que inicializar el adaptador con dicho listado, y establecerlo en el listview de la interfaz. Uno de los métodos más utilizados para esto en este proyecto es el siguiente.

```
private void createListView(ArrayList<Recipe> recipes) {
recipeItems.clear();
 for (int i = 0; i < recipes.size(); i++) {
        recipeItems.add(new RecipeItem(recipes.get(i).getName(), recipes
.get(i).getCuisine(), recipes.get(i).getPhoto()));
 }

 adapter = new (getActivity(), R.layout.recipe_listview, recipeItems);
 adapter.notifyDataSetChanged();
 getListView().setAdapter(adapter);

}
```

De esta manera, cada vez que de devuelve un listado de recetas que se quiera mostrar, por cada una de ellas se crea un objeto de tipo RecipeItem, en el cual se definen nombre, cocina y foto y se añade a un ArrayList, que será el que se envíe para inicializar el adaptador. Este devolvería la vista personalizada y mediante el método setAdapter() se establecería en la interfaz.

6.3.3 Action Bar

El Action Bar es la barra superior de la aplicación y generalmente su uso está orientado a proveer al usuario diferentes opciones y funcionalidades dependiendo de la pantalla en la que se encuentre. De este modo, muchas de las opciones de los menús que se han establecido a lo largo de la aplicación están accesibles desde la Action Bar.

Asimismo, se ha añadido el icono de la aplicación como forma de navegación entre las distintas pantallas, como se puede ver en la imagen de la derecha.

Esta navegación es diferente a la realizada con el botón de retroceso, dado que en este caso lo que se tiene en cuenta es la cronología de las pantallas y no la conexión entre ambas.

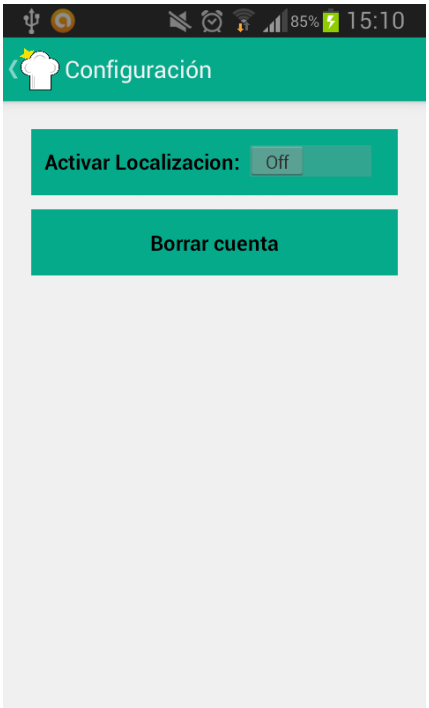


Ilustración 71: Action Bar

Para habilitar este tipo de navegación, se ha implementado lo siguiente en el método onCreate() decada una de las actividades:

```
actionBar.setDisplayHomeAsUpEnabled(true);
```

Además, es necesario establecer en el manifiesto la relación entre las actividades, para definir qué actividad precederá a otra a la hora de utilizar la navegación. De este modo, por cada actividad se ha definido cuál es su "parent activity", de la siguiente manera:

```
android:parentActivityName="com.example.Cookingstardust.MainActivity"
```

6.3.4 Menús

Como se ha comentado en el apartado en el que se trata la Action Bar, los menús van anclados en la Action Bar. Estos menús se implementan en el método onCreateOptionsMenu() al comenzar cualquier actividad y primero será necesario definirlos en un fichero XML. A lo largo de la aplicación se han construido diferentes menús, dependiendo de la pantalla a la que se esté accediendo. De este modo, cada vez que se muestran los datos de una receta existen diferentes menús, dependiendo de la receta:

- Si la receta es del usuario que la está consultando, el menú tendrá las siguientes opciones: Cocinar, Lista de la Compra, Modificar y Borrar.
- Si la receta se encuentra en la lista de favoritos del usuario, el menú tendrá las siguientes opciones: Cocinar, Lista de la Compra, Quitar de Favoritos y Puntuar.
- Si la receta no cumple ninguna de las dos condiciones anteriores, el menú tendrá las siguientes opciones: Cocinar, Lista de la Compra, Añadir a Favoritos y Puntuar.

Del mismo modo, otras pantallas incluyen otros menús. Cuando se busca una receta usando el widget de la Action Bar, también existe la opción de una búsqueda avanzada, o de una búsqueda por localización. Para definir cualquiera de estos menús, se ha hecho a nivel de interfaz en un fichero XML de la carpeta Menu.

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android" >

    <item
        android:id="@+id/action_shopping_cart"
        android:icon="@drawable/shoppingcart"
        android:showAsAction="always"
        android:title="@string/action_carrito"/>
    <item
        android:id="@+id/cocinar_option"
        android:icon="@drawable/fork"
        android:showAsAction="always"
        android:title="@string/action_cocinar"/>
```

La opción showAsAction está definida con la opción “siempre”, por lo cual esas opciones de menú aparecerán siempre en la Action Bar. Una vez creado el menú a este nivel, simplemente se han cargado en el método onCreateOptionsMenu:

```
public boolean onCreateOptionsMenu(Menu menu) {
 MenuInflater inflater = getMenuInflater();
 if (menuOption == 0) {
        inflater.inflate(R.menu.recipe_options, menu);
 } else {
        if (menuOption == 1) {
               inflater.inflate(R.menu.recipe_options_search, menu);
               } else {
                      if (menuOption == 2) {
inflater.inflate(R.menu.recipe_options_favs, menu);
                      }
        }
}
return true;
}
```

Este método es el que se ha utilizado para implementar el ejemplo anterior de las recetas. Dependiendo de la condición de la receta, se cargará un menú u otro. Luego mediante el método onOptionsItemSelected(), se establecen las acciones de cada opción del menú, por ejemplo:

```
private boolean onOptionsItemSelected(MenuItem item) {
DialogFragment dialogFragment = new RecipeDialog();
 Bundle bundle = new Bundle();
 switch (item.getItemId()) {
 case R.id.action_shopping_cart:
           bundle.putInt("valor", 13);
           dialogFragment.setArguments(bundle);
           dialogFragment.show(getFragmentManager(), "recipe_dialog");
     return true;
 case R.id.cocinar_option:
           bundle.putParcelableArrayList("pasos", elaboracion);
           Intent i = new Intent();
           i.setClass(this, StartCookingVoice.class);
           i.putExtras(bundle);
           startActivity(i);
        return true;
```

6.3.5 Search Widget

Para implementar el widget de búsqueda en el Action Bar de la clase principal, Home, se ha creado un menú específico llamado menu_search_bar, con la siguiente estructura:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android" >

    <item
        android:id="@+id/action_search"
        android:actionViewClass="android.widget.SearchView"
        android:icon="@android:drawable/ic_menu_search"
        android:showAsAction="collapseActionView|ifRoom"
        android:title="@string/action_search"/>

</menu>
```

De esta manera, se establece que el tipo de acción es un widget de búsqueda, que se mantendrá contraído en la Action Bar hasta que el usuario lo selecciona para usarlo. A la derecha se muestra una imagen del widget en cuestión.

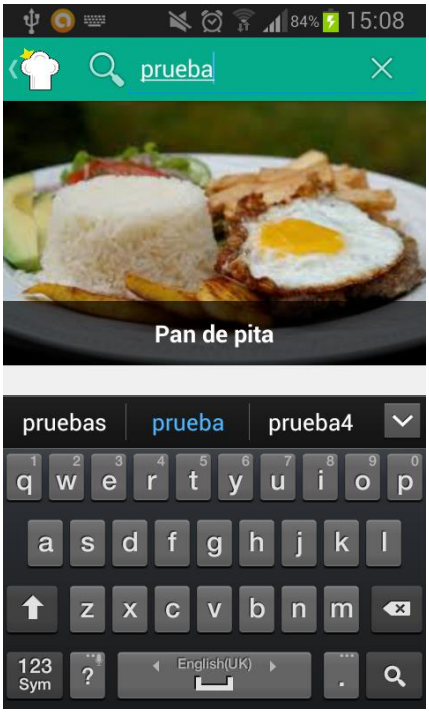


Ilustración 72: Search Widget

Para implementar este complemento de búsqueda, es necesario cargarlo en el método onCreateOptionsMenu de la clase Home. Para establecer la configuración de la búsqueda, se define una clase llamada SearchableActivity, cuya configuración en el Android Manifiest incluye la acción SEARCH.

```
<action android:name="android.intent.action.SEARCH" />
```

De esta manera, dado que el widget de búsqueda únicamente está en la actividad Home, será esta clase la que defina una configuración de búsqueda por defecto, también establecida en su definición en el manifiesto.

```
<meta-data
android:name="android.app.default_searchable"
android:value=".SearchableActivity">
</meta-data>
```

Asimismo, mediante la clase SearchManager, se establece que cada vez que el usuario pulse el botón de búsqueda y el contenido no sea vacío, la aplicación busca la actividad de búsqueda establecida por defecto en el manifiesto (SearchableActivity) y manda allí el resultado introducido en la barra. De este modo, desde esa clase se procederá a hacer la llamada a la Datastore.

```
SearchManager searchManager = (SearchManager)(Context.SEARCH_SERVICE);
mSearchView = (SearchView) menu.findItem(R.id.action_search).getActionView();
mSearchView.setSearchableInfo(searchManager.getSearchableInfo
(getComponentName()));
```

Para acceder al texto introducido se puede hacer de la siguiente manera, donde intent es el objeto que se recibe de la clase origen:

```
if (intent.ACTION_SEARCH.equals(intent.getAction())) {
query = intent.getStringExtra(SearchManager.QUERY);
}
```

Y una vez que se tiene una variable string con el texto a buscar, se pasaría a realizar la llamada a la Datastore del mismo modo en el que se ha explicado en el punto 10.2 del apartado de Desarrollo.

6.3.6 DialogFragments

A lo largo del desarrollo de la aplicación se han ido encontrado diferentes casuísticas en las cuales el uso de diálogos era necesario. Tanto para avisar al usuario de alguna acción que va a realizar, como para darle la opción de introducir datos o elegir entre varias opciones que se le ofrecen. De esta manera, el uso de los diálogos se extiende a muchas de las funcionalidades de la aplicación: se utiliza para añadir ingredientes o pasos al guardar una receta, para que el usuario escoja el filtro mediante el cual quiere ordenar sus recetas favoritas, para avisarle de que va a borrar o modificar una receta… como se puede ver en la imagen de la derecha.

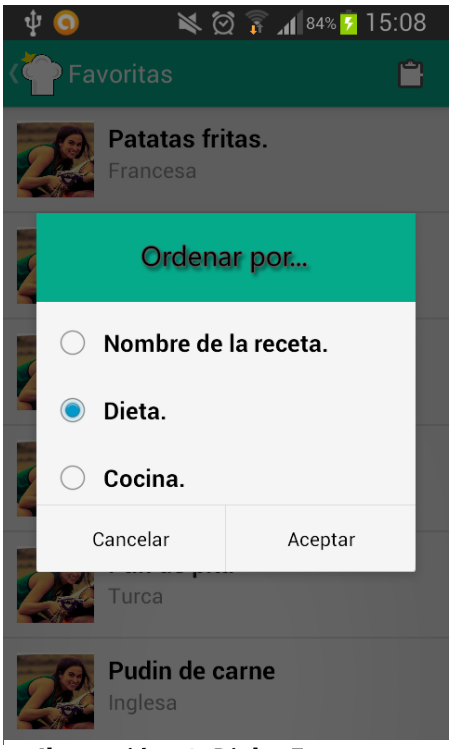


Ilustración 73: DialogFragment.

Para ello, se decidió crear una sola clase, que hereda de DialogFragment, que fuera la encargada de crear todos los diálogos y que en función de una variable mostrara uno u otro. Asimismo, dentro de los DialogFragments se decidió utilizar las AlertDialog, que ofrecen un diálogo con uno, dos o incluso tres botones. Para este proyecto se optó por personalizar los diálogos, y para ello fue necesario crear layouts personalizados para cada uno de ellos, que incluyeran los elementos que se quisieran mostrar.

Los diálogos se implementan en la actividad que quiere hacer uso de ellos de la siguiente manera:

```
DialogFragment dialogFragment = new RecipeDialog();
Bundle bundle = new Bundle();
bundle.putInt("valor", i);
dialogFragment.setArguments(bundle);
dialogFragment.show(getFragmentManager(), "recipe_dialog");
```

Luego la clase RecipeDialog es la encargada de crearlos, para ello primero creado una vista que haga uso del layout personalizado y añadiéndosela al constructor del diálogo. Asimismo, mediante la interfaz DialogInterface.OnClickListener se establece las acciones de los botones en caso de que sean pulsados.

```
AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
LayoutInflater inflater = getActivity().getLayoutInflater();
final View v = inflater.inflate(R.layout.i_dialog_layout, null);
builder.setView(v);
builder.setPositiveButton(R.string.ButtonIngs, new
DialogInterface.OnClickListener() {
@Override
  public void onClick(DialogInterface dialog, int id) {
         dialog.dismiss();
  }
});


builder.setNegativeButton(R.string.cancel_button, new
DialogInterface.OnClickListener() {
public void onClick(DialogInterface dialog, int id) {
         toast("pulsado cancelar");
         RecipeDialog.this.getDialog().cancel();
  }
});
```

6.3.7 Swipe Views, Tabs y Fragments

Cómo mostrar todos los datos de una receta siempre fue uno de los puntos más importantes de la aplicación, dado que Cooking Stardust se basa en eso, en recetas. Para ello, se estuvo pensando la idea de mostrar los datos en una única interfaz, pero el resultado era demasiado pobre. Después de una rápida investigación sobre el diseño de nuevas aplicaciones se decidió que los datos de las recetas estarían divididos en tres pestañas diferentes: una con la información general de la receta, otra con los ingredientes y una tercera con los pasos, como se puede apreciar en la imagen de la derecha. De esta manera, se optó por implementar swipe views, que básicamente consiste en poder pasar de una pestaña a otra mediante un gesto horizontal con el dedo.

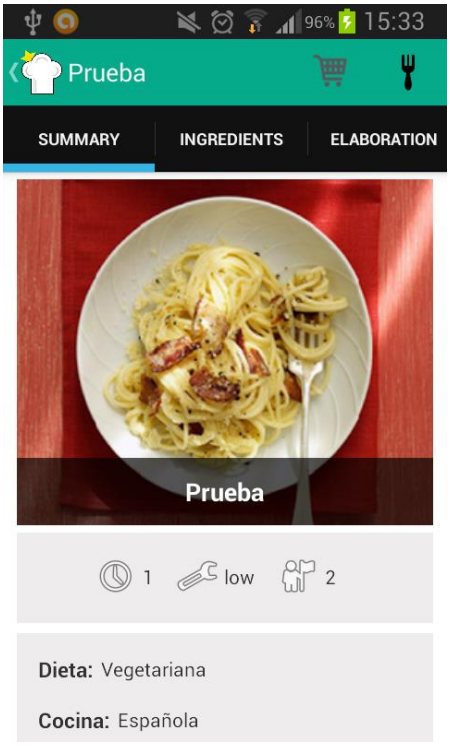


Ilustración 74: Swipe Tabs

Para crear esta funcionalidad se hizo uso de tres elementos diferentes.

- **ViewPager**: es un gestor de layouts que permite el desplazamiento entre pestañas de manera táctil. Por lo que la actividad que se utiliza para mostrar recetas debe definirlo en su layout:

```
<android.support.v4.view.ViewPager
    android:id="@+id/pager"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >
</android.support.v4.view.ViewPager>
```

- **Fragments**: son partes de la interfaz de una actividad que tienen un comportamiento propio, pero nunca totalmente independiente dado que dependen siempre de la actividad que los aloja. En este caso, se utilizarán Fragments para implementar el contenido de las tres pestañas, dado que de esta manera únicamente es necesario crear una sola actividad cuyo contenido irá variando dependiendo del Fragment que se cargue.

- **FragmentPagerAdapter**: es el adaptador necesario para poder añadir los Fragments que compondrán cada una de las pestañas. Para la aplicación se creó un adaptador personalizado que añadiera tres Fragments:

```
public class TabsPagerAdapter extends FragmentPagerAdapter {
	public TabsPagerAdapter(FragmentManager fm) {
		super(fm);
	}
	@Override
	public Fragment getItem(int index) {
		switch (index) {
		case 0:
			return new SummaryFragment();
		case 1:
			return new IngredientsFragment();
		case 2:
			return new ElaborationFragment();
		}
		return null;
	}
	@Override
	public int getCount() {
		return 3;
	}
}
```

De esta manera, el SummaryFragment será la pestaña en la cual se han definido los datos generales de la receta, mientras que el IngredientesFragment y el ElaborationFragment serán los encargados de mostrar los ingredientes y los pasos respectivamente.

Finalmente, una vez definidos todos estos elementos, sólo fue necesario inicializarlos y añadirlos en la actividad que aloja la funcionalidad de Mostrar Receta:

```
mAdapter = new TabsPagerAdapter(getSupportFragmentManager());
viewPager.setAdapter(mAdapter);
actionBar.setNavigationMode(ActionBar.NAVIGATION_MODE_TABS);
```

La opción NAVIGATION_MODE_TABS establece que las pestañas se implementarán en la Action Bar, por lo cual es necesario añadirlas:

```
for (String tab_name : tabs) {
actionBar.addTab(actionBar.newTab().setText(tab_name)
.setTabListener(this));
}
public void onTabSelected(Tab tab, FragmentTransaction ft) {
        viewPager.setCurrentItem(tab.getPosition());
}
```

Una vez añadidas, el paso final sería permitir la navegación entre pestañas haciendo uso del swipe views:

```
viewPager.setOnPageChangeListener(new ViewPager.OnPageChangeListener() {

public void onPageSelected(int position) {
        actionBar.setSelectedNavigationItem(position);
 }
 @Override
 public void onPageScrolled(int arg0, float arg1, int arg2) {
        // TODO Auto-generated method stub
 }
 @Override
 public void onPageScrollStateChanged(int arg0) {
        // TODO Auto-generated method stub
}
```

6.3.8 Localización

En Cooking Stardust existen tres formas diferentes de poder buscar recetas. Las dos primeras son simplemente búsquedas, la única diferencia que existe entre ambas es que la primera sólo busca por nombre (a través del widget de búsqueda en el Action Bar), mientras que la segunda implementa una búsqueda avanzada, con múltiples campos: nombre, ingredientes, tipo de cocina, dieta, tipo de plato…

Sin embargo, existe una tercera opción de búsqueda que se basa en la localización de las recetas guardadas. De este modo, si el usuario accede a esta opción se le listarán todas las recetas cercanas a él en un radio de cincuenta kilómetros. Para ello, es necesario que en la funcionalidad Guardar Receta, el usuario guarde la localización de la receta, dado que la búsqueda de recetas por cercanía se basará después en las coordenadas de las recetas guardadas.

Para configurar el acceso de la aplicación a la API de localización, ha sido necesario añadir las siguientes opciones en el manifiesto:

```
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" >
</uses-permission>
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION">
</uses-permission>
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION">
```

Asimismo, dado que no es legal ni moralmente ético el ubicar y/o guardar las coordenadas de un usuario sin su consentimiento, se ha establecido una configuración de localización. De este modo, el usuario tendrá que activar la opción Localización en el menú Configuración si quiere usar tanto la funcionalidad de búsqueda por cercanía como la opción de guardar sus recetas con localización.

De esta manera, siempre que el usuario haga uso de estas dos funcionalidades se comprobará que dicha opción está habilitada o deshabilitada. En caso de que no lo esté, se le advertirá al usuario de que debe activarla.

Para localizar las coordenadas de una aplicación en el momento de su uso, se ha utilizado una instancia de la clase LocationManager, que provee acceso a los servicios de localización del sistema, definidos previamente en el manifiesto.

```
mLocationManager = (LocationManager)
this.getSystemService(Context.LOCATION_SERVICE);
```

Junto al gestor de localizaciones, es necesario implementar la interfaz LocationListener, mediante la cual somos capaces de conseguir la latitud y la longitud.

```
LocationListener locationListener = new LocationListener() {
@Override
       public void onLocationChanged(android.location.Location location) {
latitude = Double.toString(location.getLatitude());
            longitude = Double.toString(location.getLongitude());
}

mLocationManager.requestLocationUpdates(LocationManager.NETWORK_PROVIDER, 0,
0, locationListener);
```

De esta manera, solo sería necesario hacer la llamada a la Datastore. En el caso de Guardar Receta, las variables latitude y longitude se añaden a la llamada de guardar una receta. En el caso de la búsqueda de recetas cercanas, la llamada a la Datastore se haría como se explicó en el punto 10.2, y la búsqueda de recetas en base a las coordenadas a nivel de Datastore se haría como se detalló en el punto 10.1.4 de este mismo apartado.

6.4 Otros Elementos

En este apartado se tratarán el resto de elementos que se han utilizado a la hora de hacer el proyecto pero que no se incluyen en los apartados anteriores.

6.4.1 Shared Preferences

La clase SharedPreferences se utiliza para almacenar y acceder cualquier dato de tipo primitivo: booleans, floats, ints, longs o strings de manera persistente. Es decir, los datos guardados mediante SharedPreferences no se perderán ni aunque el usuario cierre la aplicación. Debido a ello, se ha utilizado esta clase en dos funciones muy puntuales: para guardar el identificador del usuario cuando se loguea, de tal manera que aunque cierre la aplicación al abrirla se le redirija directamente a la actividad principal (salvo que se haya desconectado) y para mantener activada o desactivada la opción de localización en el menú de Configuración.

Cada vez que un usuario se loguea por primera vez se crea una instancia de SharedPreferences con el nombre que tenga la variable PREFS_NAME y se habilita su edición, añadiendo el identificador de usuario y guardándolo con el método commit().

```
SharedPreferences userID = getSharedPreferences(PREFS_NAME, 0);
SharedPreferences.Editor editor = userID.edit();
editor.putLong("userID", result.getId());
editor.commit();
```

De esta manera, cada vez que un usuario se loguea se comprueba que existe un valor para esa instancia de preferencias y en caso de que así sea, se le redirige directamente a la clase Home.

```
SharedPreferences userID = getSharedPreferences(PREFS_NAME, 0);
Long l = 4L;
final Long lon = userID.getLong("userID", l);
if (lon.longValue() != l.longValue()){
Intent i = new Intent();
       i.setClass(getApplicationContext(), Home.class);
       startActivity(i);
       this.finish();
}
```

Por otro lado, cada vez que un usuario se desconecta mediante la opción Logout del menú principal, se borran su identificador de las SharedPreferences, de tal modo que cuando acceda la siguiente vez a la aplicación, tenga que loguearse otra vez.

```
SharedPreferences userID = getSharedPreferences(PREFS_NAME, 0);
SharedPreferences.Editor editor = userID.edit();
editor.clear();
editor.commit();
```

Esta misma estructura se aplicaría a la función de activar y desactivar localización en el menú Configuración.

6.4.2 Interfaz Comparator

El usuario tiene la opción de ordenar sus recetas favoritas en base al nombre, la dieta y la cocina. Para ello, se ha utilizado la interfaz Comparator de Java y se han creado tres clases que implementan dicha interfaz.

```
public class AlfabeticoComparator implements Comparator<Recipe> {
 public int compare(Recipe r1, Recipe r2) {
        return r1.getName().compareTo(r2.getName());
 }
}
```

```
public class DietaComparator implements Comparator<Recipe> {
 @Override
 public int compare(Recipe r1, Recipe r2) {
        return r1.getDiet().compareTo(r2.getDiet());
 }
}
```

```
public class CocinaComparator implements Comparator<Recipe> {
 @Override
 public int compare(Recipe r1, Recipe r2) {
        return r1.getCuisine().compareTo(r2.getCuisine());
 }
}
```

De esta manera, únicamente es necesario pasarle la lista de recetas y la clase mediante cuyo criterio se quiere ordenar la lista y el método Collections.sort() se encarga de ordenarla.

```
Collections.sort(recs, new AlfabeticoComparator());
```

La interfaz Comparator también se utiliza en la funcionalidad de Recomendar Recetas para la misma función, ordenar listados de manera rápida y sencilla.

6.4.3 Lista de la Compra

La función de la Lista de la Compra es calcular una posible lista de la compra basándose en los ingredientes y el número de comensales para los cuales está definida la receta. Asimismo, el usuario deberá especificar el número de personas para las cuáles quiere realiza la lista.

Teniendo en cuenta estos datos, el cálculo es bastante sencillo. En caso de que el usuario elija el mismo número de personas que el que ya tiene definida la receta, se le mostrarán los ingredientes tal cual.

En caso contrario, se procederá a calcular las cantidades de los ingredientes acordes al nuevo número de comensales utilizando una regla de tres. En caso de que la cantidad original del ingrediente sea 1 y la calculada mayor que 1, se le añadirá una “s” del plural al nombre del ingrediente. En el caso contrario, en el que la cantidad calculada sea 1, se quitará la última “s” del nombre del ingrediente.

```
Long newQuantity1 = comensales*ingredients.get(i).getQuantity();
Float newQuantity2 = newQuantity1/(float)people;
Ingredient ing = new Ingredient();

if (ingredients.get(i).getQuantity() == 1 && Math.round(newQuantity2) !=
1.0){
String name = ingredients.get(i).getIngName() + "s";
       ing.setIngName(name);
}else{
if (ingredients.get(i).getQuantity() != 1 && Math.round(newQuantity2) ==
1.0){
        String name = ingredients.get(i).getIngName();
        name = name.substring(0, name.length()-1);
        ing.setIngName(name);
}else{
ing.setIngName(ingredients.get(i).getIngName());
 }
}

int can = Math.round(newQuantity2);
ing.setNewQuantity(Float.parseFloat(String.valueOf(can)));
newIngredientes.add(ing);
```

Por otro lado, se utiliza la función Math.round para que las cantidades sean exactas, sin ningún tipo de decimal, dado que las cantidades de los ingredientes siempre estarán en números enteros.

Una vez calculada la nueva lista de ingredientes, sólo es necesario añadirla al TextView que compone el layout de la actividad:

```
nota = (TextView) findViewById(R.id.listaDeLaCompraView);
for (int i=0; i<newIngredientes.size(); i++){
lista = lista+ "- "+ newIngredientes.get(i).getNewQuantity() + " " +
newIngredientes.get(i).getIngName() + "\n";
}
nota.setText(lista);
```

6.4.4 Recomendaciones

El algoritmo de recomendación de recetas se ha diseñado de la siguiente manera, teniendo en cuenta las recetas favoritas y las puntuaciones del usuario:

1. Si el usuario tiene recetas favoritas:
    a. Por cada receta de listado de favoritas se comprueban dieta, cocina y tipo de plato, y se construye un listado con esos tres campos, en los que cada uno de ellos tendrá la suma total de veces que aparecen en el listado de recetas favoritas del usuario.
    b. Se ordenan las tres listas y se cogen las siguientes posiciones:
        i. Los dos tipos de dieta con mayor puntuación.
        ii. Los tres tipos de plato con mayor puntuación.
        iii. Los tres tipos de cocina con mayor puntuación.
    c. Se hace la llamada a la Datastore con esos ocho datos:
        i. En la Datastore se filtran todas las recetas que cumplan con esos datos y que no se incluyan en el listado de recetas favoritas del usuario, ni en el listado propio de recetas del usuario.
        ii. Se ordenan las recetas recogidas por puntuación.
        iii. Se cogen las veinte primeras con la mejor puntuación y se devuelven.
2. En caso de que el usuario no tenga recetas favoritas, se coge el listado de sus recetas mejor puntuadas y se realiza lo mismo que en los pasos 1a, 1b y 1c.
3. En el caso de que el usuario no tenga recetas favoritas ni recetas puntuadas, se le devuelve un listado de las mejores 20 recetas del sistema.

Para implementar este algoritmo no se ha utilizado nada más que condiciones y llamadas a la Datastore, además del uso de la interfaz Comparator (explicada en el apartado 10.5.2).

6.4.5 Captura de fotos

Para la realización de este proyecto se consideró adecuado el permitirle al usuario adjuntar una foto de la receta, así como de los pasos a seguir para cocinarla. De esta manera, el usuario tendrá la opción de sacar una foto al momento con la cámara o adjuntar una foto que ya tuviera en la galería.

Para poder coger la foto directamente de la cámara, es necesario darle los permisos necesarios a la aplicación para ello, mediante el manifiesto.

```
<uses-feature android:name="android.hardware.camera"
android:required="true" />
```

Para realizar la acción de acceder a la foto de la cámara o la de la galería, Android utiliza los Intents, es decir, un objeto abstracto que va a permitir llevar a cabo la operación.

De esta manera, se define el Intent:

```
Intent i = new Intent(Intent.ACTION_PICK,
android.provider.MediaStore.Images.Media.EXTERNAL_CONTENT_URI);
```

Y se ejecuta mediante el método startActivityForResult(), que devolverá el resultado a la actividad mediante el método onActivityResult(), donde se le pasará la imagen.

```
startActivityForResult(i, RESULT_LOAD_IMAGE);
```

A este método se le pasa un código, en este caso asignado a la variable RESULT_LOAD_IMAGE, para que después, en el onActivityResult, se pueda confirmar que el resultado proviene de la operación deseada.

```
protected void onActivityResult(int requestCode, int resultCode, Intent data)
{
super.onActivityResult(requestCode, resultCode, data);
if (requestCode == RESULT_LOAD_IMAGE && resultCode == RESULT_OK && data !=
null) {
Uri selectedImage = data.getData();
String[] filePathColumn = {MediaStore.Images.Media.DATA};
Cursor cursor = getContentResolver().query(selectedImage,  filePathColumn,
null, null, null);
       cursor.moveToFirst();
       int columnIndex = cursor.getColumnIndex(filePathColumn[0]);
       picturePath = cursor.getString(columnIndex);
       cursor.close();
}
```

De esta manera, mediante el getData() del intent recibido como parámetro tendríamos el identificador de la foto, pero es necesario la ruta de la imagen. Para ello se define un cursor. Dado que una imagen contiene varias columnas de información, es necesario utilizar un cursor que dado el nombre de la columna, en este caso MediaStore.Images.Media.DATA, te devuelva primero la posición de dicha columna y después ya, la ruta de la misma. Será ese dato, el picturePath, el que se utilizará para guardar las fotos en la Blobstore.

Este ejemplo está orientado a sacar una foto de la galería, y la única diferencia que hay con respecto a la cámara es el Intent que se ejecuta. En el caso de la cámara sería el siguiente:

```
Intent takePictureIntent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
```

6.4.6 Reconocimiento de voz

Para desarrollar la funcionalidad Cocinar, se decició implementar un sistema de reconocimiento de voz, mediante el cual, utilizando algunas palabras clave, el usuario fuera capaz de avanzar en los pasos de la receta sin tener que utilizar las manos.

Debido a que no se encontró ninguna solución de terceros libre o fácil de implementar, finalmente se decidió hacer uso de la clase SpeechRecognizer propia de Android. A pesar de que en la documentación oficial no se recomienda la utilización de esta API como un servicio en segundo plano de reconocimiento de voz (debido al altísimo nivel de batería y ancho de banda que consume), se optó por esta solución al ser únicamente orientado a esta funcionalidad, y por tanto, solamente estaría activado cuando se accede a ella.

De esta manera, para poder implementar esta funcionalidad, es necesario añadir el siguiente permiso en el Android Manifiest, que permite grabar el audio.

```
<uses-permission android:name="android.permission.RECORD_AUDIO" >
```

Por otro lado, se ha creado una nueva clase llamada Listener que implementa la interfaz RecognitionListener, la encargada de manejar todos los eventos de reconocimiento de voz, así como recibir los resultados de la misma. Tiene un método *onResults()* responsable de recibir la información que el listener haya conseguido a través del reconocimiento de voz. Será en este método donde se definan las condiciones necesarias para avanzar o retroceder en la elaboración de la receta. Para ello, se hará uso de tres palabras: Siguiente, Anterior y Finalizar. Esta última es la palabra clave, tanto para terminar la actividad cuando se llegue al último paso de la elaboración, como para cerrarla en cualquier momento.

Por otro lado, las palabras “siguiente” y “anterior” serán mediante las cuales el usuario pueda avanzar o retroceder. Por esto mismo, siempre y dependiendo de en qué paso de la elaboración se encuentre el usuario, a pesar de que pueda decir “Siguiente” o “Anterior”, se realiza una comprobación de que en verdad es posible avanzar o retroceder.

La instancia de la clase SpeechRecognizer se crea de la siguiente manera.

```
SpeechRecognizer sr;
sr = SpeechRecognizer.createSpeechRecognizer(this);
sr.setRecognitionListener(new Listener());
```

Una vez establecida la clase Listener como clase encargada de manejar los eventos de reconocimiento de voz, sólo quedaría activar la función y comenzar a escuchar, mediante el siguiente código, que se ejecutaría al pulsar un botón específico:

```
private void launchRecognitionIntent() {
Intent intent = new Intent(RecognizerIntent.ACTION_RECOGNIZE_SPEECH);
intent.putExtra(RecognizerIntent.EXTRA_LANGUAGE_MODEL,RecognizerInten
t.LANGUAGE_MODEL_FREE_FORM);
intent.putExtra(RecognizerIntent.EXTRA_CALLING_PACKAGE,"com.example.p
ruebalogin");
intent.putExtra(RecognizerIntent.EXTRA_MAX_RESULTS,3);
sr.startListening(intent);
}
```

Se crea un Intent que sea la acción de reconocer voz, y se le pasan como parámetros el nombre del paquete y el número máximo de resultados que el listener será capaz de devolver. Huelga decir que está orientado para que se implemente en el idioma anglosajón, a pesar de que las palabras “Siguiente”, “Anterior” y “Finalizar” las entienda perfectamente.

De esta manera, cada vez que el listener recoja cualquier palabra que escuche, se tratarán en el método onResults() de la clase Listener:

```
public void onResults(Bundle results) {
 String str = new String();
 Log.d(TAG, "onResults " + results);
 ArrayList data =
results.getStringArrayList(SpeechRecognizer.RESULTS_RECOGNITION);
 for (int i = 0; i < data.size(); i++) {
       Log.d(TAG, "result " + data.get(i));
 }
 if (data.get(0).equals("finalizar")) {
       sr.stopListening();
       finishActivity();
 }
}
```

Por otro lado, debido a que esta clase no está preparada para su ejecución en segundo plano, no es posible quitar el sonido de aviso de que la aplicación está escuchando que se repite cada seis segundos. Por lo que, después de hacer pruebas, se llegó a la conclusión de que dicho sonido incluso servía de recordatorio al usuario de que la aplicación está escuchando y que espera a que interactúe con ella. Asimismo, cuando el listener está escuchando y se cumple el tiempo de los seis segundos sin que el usuario haya dicho nada, termina en error y deja de escuchar. Los errores se manejan en el método *onError()* de la clase Listener y para evitar que la aplicación dejara de escuchar en el momento en el que terminara como error, se implementó la siguiente solución:

```
public void onError(int error) {
 Log.d(TAG,  "error " +  error);
 launchRecognitionIntent();
}
```

El método launchRecognitionIntent() recoge el código necesario para volver a ejecutar el listener y que la aplicación continúe escuchando de manera indefinida. Por otro lado, también se han implementado los métodos onDestroy(), onStop(), onResume(), y onPause() propios de una actividad, y en los cuáles se para y se termina con la ejecución del listener cuando se cierra la actividad. Asimismo, durante la ejecución de esta actividad, se procederá al deshabilitar el bloqueo de la pantalla, con el fin de mejorar la experiencia al usuario. Una vez terminada la actividad, se devolverá el control del bloqueo de la pantalla al sistema operativo.

6.5 Software de terceros

Para la realización de este proyecto se ha hecho uso de la librería Ion (https://github.com/koush/ion) para la función de guardar y descargar las imágenes en el Blobstore a lo largo de toda la aplicación. Dicha librería implementa internamente la llamada HTTP al servidor de manera asíncrona, por lo cual se puede hacer de manera directa, sin afectar al hilo principal.

Para guardar la foto en el servidor, únicamente es necesario pasarle como URL la dirección en la que se va a guardar (explicado en el apartado 10.1.3).

```
Ion.with(getApplicationContext()).load(finalUrl).setMultipartParamete
r("file", "image/jpeg").setMultipartFile("file", new
File(picturePath)).asJsonObject().setCallback(new
FutureCallback<JsonObject>() {
public void onCompleted(Exception e, JsonObject json) {
        JsonElement element;
        element = json.get("file");
        urlKey = element.getAsString();
}
});
```

De esta manera, mediante el método setMultipartFile() sólo será necesario pasarle el URI de la foto sacada tanto de la galería como de la cámara y gracias a la interfaz Callback, se nos devuelverá el resultado de esa llamada, es decir, el identificador de la imagen que será necesario en el futuro en caso de querer descargar la imagen.

La dinámica de descargar una foto desde el Blobstore es parecida. Únicamente habrá que construir la URL definida mediante la llamada del Blobstore y el identificador de la imagen, dado que Ion permite descargar directamente una imagen en un ImageView, de la siguiente manera:

```
Ion.with(recipePhoto).placeholder(R.drawable.yo).error(R.drawable.yo)
.load(urlFinal);
```

La opción with permite establecer el ImageView en el que se va establecer la imagen, así como el método error() establece la imagen que se cargará en caso de error al devolver la imagen mediante el load() del servidor. En este caso ni siquiera es necesario el uso de la interfaz Callback.

Cabe destacar que el uso de esta librería no está únicamente orientado a guardar y descargar imágenes del servidor, sino que se puede realizar cualquier llamada POST/GET a cualquier servidor, parseando para ello el código JSON necesario. En este caso, dado que las llamadas las tenemos centralizadas a través de las clases generadas por el AppEngine y las clases Cookingstardust y Api, únicamente se ha utilizado la librería para la función de las fotos, dado que implementan su propia Api y no accede a la Datastore.

7. VERIFICACIÓN Y PRUEBAS

En este apartado se detallarán las pruebas realizadas a lo largo de todo el proceso de desarrollo, mediante las cuales se han podido ir identificando y solventando los errores encontrados por cada una de las funcionalidades implementadas. La mayoría del tiempo invertido en hacer pruebas fue a nivel de servidor, las cuales no están reflejadas en este documento dado que no eran respectivas a este proyecto sino al proceso de aprender a programar en Python y utilizar correctamente las herramientas AppEngine y Datastore.

7.1 Registrar un usuario

Código	Descripción	Resultado Esperado	Resultado Obtenido	Observaciones
1.	Registrar un usuario que ya existe en la Datastore.	Un aviso de que ya existe.	El aviso.	Correcto.
2.	Registrar un usuario que no existe en la Datastore.	Un mensaje indicando que se ha guardado correctamente.	El mensaje.	Correcto.

Tabla 2: Pruebas Registrar un usuario

7.2 Loguear un usuario

Código	Descripción	Resultado Esperado	Resultado Obtenido	Observaciones
1.	Loguear un usuario con la contraseña o el nickname incorrecto.	Un aviso de que la contraseña y el nickname no coinciden.	La llamada no se realizaba a la Datastore.	**Error.**
2.	Loguear un usuario con la contraseña y el nickname correcto.	La pantalla principal de la aplicación.	La llamada no se realizaba a la Datastore.	**Error.**

Tabla 3: Pruebas Loguear un usuario

El problema resultó ser la configuración de las llamadas a la Datastore. Después de realizar un registro exitoso de un usuario (es decir, un POST), se llegó a la conclusión de que hacer una consulta (un GET), tendría la misma estructura que el caso anterior.

El problema residía que en este caso (y en cualquiera en el que el tipo de llamada fuera una consulta) no se puede definir una clase de tipo Message, como era el caso, para realizar la llamada desde la parte cliente de Android a la Datastore. En cambio, hay que definir una variable de tipo ResourceContainer, explicadas en el apartado 10.1.2. Una vez definida esta variable, se le asignó al método correspondiente en la API y la comprobación funcionó correctamente.

7.3 Buscar una receta por nombre

Código	Descripción	Resultado Esperado	Resultado Obtenido	Observaciones
1.	Buscar una receta por nombre usando el Search Widget de la pantalla Home.	Un listado de recetas cuyo nombre coincidiera con lo instroducido.	Error y cierre de la aplicación.	**Error.**
2.	No introducir ninguna palabra en el Search Widget de la pantalla Home y buscar de todas formas.	Que el buscador no realizara la búsqueda.	No se realiza la búsqueda.	Correcto.

Tabla 4: Pruebas Buscar una receta por nombre

El funcionamiento del Search Widget es el siguiente: cuando un usuario introduce una palabra a buscar y pulsa el botón de búsqueda, automáticamente el widget redirige los datos introducidos a una actividad de tipo Searchable Actvity. El problema residía que dicha actividad existía, y estaba implementada para realizar la llamada a la Datastore, pero no estaba especificado en el Android Mainifiest, por lo cual, el widget era incapaz de saber a dónde debía redirigir los datos. Una vez configurado el manifiesto, la llamada resultaba ser correcta y se mostraba el listado de recetas que coincidían con el nombre introducido.

7.4 Buscar receta por localización

Código	Descripción	Resultado Esperado	Resultado Obtenido	Observaciones
1.	Buscar una receta por localización con la localización desactivada.	Un aviso de que la localización estaba desactivada.	El aviso.	Correcto.
2.	Buscar una receta por localización con la localización activada.	El listado de recetas cercanas.	El listado de recetas cercanas.	Correcto.

Tabla 5: Pruebas Buscar una receta por localización

7.5 Buscar receta avanzada

Código	Descripción	Resultado Esperado	Resultado Obtenido	Observaciones
1.	Buscar una receta de manera avanzada con todos los campos en blanco.	Un aviso de que es necesario introducir al menos un campo.	El aviso.	Correcto.
2.	Buscar una receta con diferentes campos.	El listado de recetas que coincidieran.	Un listado de recetas que no coincidían o un listado vacío.	**Error.**

Tabla 6: Pruebas Buscar receta avanzada

En este caso el problema era a nivel del modelo y del método correspondiente a la búsqueda de recetas avanzada. Se creaba una consulta sobre la que se iban realizando el resto de filtrados. El problema era que si el filtro era nulo o en blanco, se filtraban de todas formas todas las recetas que coincidieran, en este caso, ninguna. La solución fue, mediante condiciones, verificar que el filtro no era nulo o en blanco, antes de realizar la filtración.

7.6 Ver un listado de recetas

Código	Descripción	Resultado Esperado	Resultado Obtenido	Observaciones
1.	Mostrar un listado de recetas, sobre una lista de recetas vacía.	Un aviso de que no hay recetas que mostrar.	El aviso.	Correcto.
2.	Mostrar un listado de recetas sobre una lista de recetas.	El listado.	Error y cierre de la aplicación.	**Error.**

Tabla 7: Pruebas Ver un listado de recetas

El problema se generaba a la hora de crear el ListView con el adaptador personalizado correspondiente, no se había especificado ningún identificador del listview a nivel del XML, por lo cual era incapaz de generarlo. La solución es configurar el listview de la siguiente manera: android:id="@+id/android:list".

7.7 Ver receta al detalle

Código	Descripción	Resultado Esperado	Resultado Obtenido	Observaciones
1.	Mostrar los detalles de una receta propia de un usuario.	Los detalles de la receta y las opciones de menú correspondientes.	La receta.	Correcto.
2.	Mostrar los detalles de una receta ajena a un usuario.	Los detalles de la receta y las opciones de menú correspondientes.	La receta.	Correcto.
3.	Mostrar los detalles de una receta favorita de un usuario.	Los detalles de la receta y las opciones del menú correspondientes.	La receta, pero no aparecían las opciones correspondientes.	**Error.**

Tabla 8: Pruebas Ver Receta al detalle

El problema sólo ocurría cuando se accedía a una receta favorita pero no desde la funcionalidad Favoritas, sino por ejemplo, del resultado de una búsqueda. El problema era que a la hora de

establecer qué opción de menú era la correspondiente al realizar una búsqueda no se tenía en cuenta el factor "favoritas" del usuario. La solución fue realizar un método que comprobara si dicha receta pertenecía a las favoritas del usuario para poder mandar la opción correcta del menú.

7.8 Detalles de la elaboración

Código	Descripción	Resultado Esperado	Resultado Obtenido	Observaciones
1.	Mostrar una actividad con la imagen y el texto correspondiente a un paso en la elaboración.	Los datos del paso.	Error y cierre de la aplicación.	**Error.**

Tabla 9: Pruebas Detalles de la elaboración

El problema era a la hora de cargar la foto del paso, en el caso de que la hubiera. Si no la había, la misma condición donde se trataba que fuera diferente a null también trataba una igualdad de la imagen, por lo cual en el momento en el que la imagen era nula, la condición fallaba estrepitosamente, obligando a la aplicación a cerrarse. La solución fue utilizar condiciones anidadas para evitar el problema.

7.9 Lista de la compra

Código	Descripción	Resultado Esperado	Resultado Obtenido	Observaciones
1.	Mostrar una actividad con la lista de la compra asociada al número de comensales introducido.	La lista de la compra.	La lista de la compra.	**Mejora.**

Tabla 10: Pruebas Lista de la compra

A pesar de que la lista de la compra se generaba correctamente para los datos introducidos, las cantidades aparecían con decimales, por lo cual no era viable dejar de tal modo el resultado. Se optó por utilizar el método Math.round() de Java, para redondear hacia el número entero más cercano las cantidades de los ingredientes.

Del mismo modo, para mostrarlos por la interfaz, se optó por convertirlos en String y coger simplemente la parte entera del número de tipo Double, dado que siempre se mostraba un .0 después de la cantidad.

7.10 Ver listado de recetas favoritas

Código	Descripción	Resultado Esperado	Resultado Obtenido	Observaciones
1.	Mostrar el listado de recetas favoritas de un usuario que no tuviera.	Un aviso de que no hay recetas que mostrar.	El aviso.	Correcto.
2.	Mostrar el listado de recetas favoritas de un usuario.	El listado.	El listado.	Correcto.

Tabla 11: Pruebas Ver listado de recetas favoritas

7.11 Ver Perfil

Código	Descripción	Resultado Esperado	Resultado Obtenido	Observaciones
1.	Mostrar el perfil del propio usuario con la opción Modificar Perfil.	El perfil.	El perfil.	Correcto.
2.	Mostrar el perfil de un usuario amigo del propio, con la opción Dejar de Seguir.	El perfil.	El perfil.	Correcto.
3.	Mostrar el perfil de un usuario ajeno al propio con la opción Seguir.	El perfil.	El perfil.	Correcto.

Tabla 12: Pruebas Ver Perfil

7.12 Editar Perfil

Código	Descripción	Resultado Esperado	Resultado Obtenido	Observaciones
1.	Modificar el perfil dejando en blanco los campos de Nombre, Nickname o Contraseña.	Un aviso de que hay que completar esos campos.	El aviso.	Correcto.
2.	Modificar el perfil con todos los datos obligatorios (mínimo).	Un mensaje de que el perfil se ha actualizado.	El mensaje.	Correcto.

Tabla 13: Pruebas Editar Perfil

7.13 Ver Seguidores

Código	Descripción	Resultado Esperado	Resultado Obtenido	Observaciones
1.	Ver el listado de seguidores cuando el usuario no tiene ninguno.	Un aviso de que no tiene seguidores.	El aviso.	Correcto.
2.	Ver el listado de seguidores del usuario.	El listado.	El listado.	Correcto.

Tabla 14: Pruebas Ver Seguidores

7.14 Ver Siguiendo

Código	Descripción	Resultado Esperado	Resultado Obtenido	Observaciones
1.	Ver el listado de amigos cuando el usuario no tiene ninguno.	Un aviso de que no tiene amigos.	El aviso.	Correcto.
2.	Ver el listado de amigos del usuario.	El listado.	El listado.	Correcto.

Tabla 15: Pruebas Ver Siguiendo

7.15 Buscar Amigos

Código	Descripción	Resultado Esperado	Resultado Obtenido	Observaciones
1.	Buscar un usuario por un nombre o nickname que no existe (o no se incluye) en la Datastore.	Un aviso de que no existe dicho usuario.	El aviso.	Correcto.
2.	Buscar un usuario por un nombre o un nickname.	Un listado de usuarios que coincidan.	Un listado vacío.	**Error.**

Tabla 16: Pruebas Buscar Amigos

El problema en este caso se debía a la realización de una consulta errónea en la Datastore. Sólo se estaban teniendo en cuenta todos los usuarios en los que coincidiera plenamente el nombre o el nickname facilitado y no aquellos en los que únicamente se incluyera parte del nombre o del nickname. Se realizó un cambio en la consulta, añadiendo la opción IN, que verifica no solamente que el nombre coincida al 100%, sino que lo incluya.

7.16 Guardar una receta

Código	Descripción	Resultado Esperado	Resultado Obtenido	Observaciones
1.	Guardar una receta sin los campos obligatorios.	Un aviso de que dichos campos deben ser completados.	El aviso.	Correcto.
2.	Guardar una receta con al menos los campos obligatorios.	Un mensaje de que la receta se ha guardado correctamente.	El mensaje.	**Mejora.**

Tabla 17: Pruebas Guardar una receta

A pesar de que el resultado de dichas pruebas era correcto, se pensó en una mejora con respecto a los campos de Ingredientes y Elaboración. La primera opción fue guardar tanto los ingredientes como todos los pasos de la elaboración en un campo String. El problema era que de cara al futuro (y a otras funcionalidades como Cocinar o ver los pasos de la elaboración de

manera independiente) esta solución resultaba ser demasiado poble y complicada. Debido a ello, se implementaron los Dialog Fragments, mediante los cuales se podía controlar perfectamente la introducción de estos datos por parte del usuario para poder guardarlos en un ArrayList, de tal manera que después se pudiera acceder a los datos de ambos campos de manera ordenada.

7.17 Recomendaciones

Código	Descripción	Resultado Esperado	Resultado Obtenido	Observaciones
1.	Mostrar un listado de recetas recomendadas en base a sus favoritos y en caso de que no los tuviera, de sus recetas mejor puntuadas.	Un listado de recetas.	Un listado de recetas que no coincidía con lo esperado.	**Error.**
2.	Mostrar las veinte mejores recetas del sistema en caso de que el usuario no tuviera ni recetas favoritas ni puntuadas en las que basar la recomendación.	El listado de las mejores recetas del sistema.	El listado.	Correcto.

Tabla 18: Pruebas Recomendaciones

Como se ha explicado anteriormente en el apartado 10.5.4 en el que se explica el algoritmo generado para las Recomendaciones personalizadas, se cogen las primeras opciones en los campos de Dieta, Cocina y Tipo de Plato de los ArrayList generados en base a las recetas favoritas o mejor puntuadas del usuario. El problema residía que a la hora de ordenar el ArrayList se hacía de menor a mayor, por lo cual las primeras posiciones correspondían a las opciones menos puntuadas. De esta manera, las recomendaciones generadas eran en base a las opciones menos queridas por el usuario. Este problema se corrigió cogiendo las últimas tres posiciones de los ArrayList generados, es decir, las más puntuadas.

7.18 Activar/Desactivar Localización

Código	Descripción	Resultado Esperado	Resultado Obtenido	Observaciones
1.	Activar la localización.	Un mensaje de que se ha activado.	El mensaje.	Correcto.
2.	Desactivar la localización.	Un mensaje de que se ha desactivado.	El mensaje.	Correcto.

Tabla 19: Pruebas Activar/Desactivar Localización

7.19 Borrar Cuenta

Código	Descripción	Resultado Esperado	Resultado Obtenido	Observaciones
1.	Borrar la cuenta del usuario activo.	Un mensaje de que se ha borrado y cierre de la aplicación.	El mensaje y el cierre.	Correcto.

Tabla 20: Pruebas Borrar cuenta

7.20 Desconectarse de la aplicación

Código	Descripción	Resultado Esperado	Resultado Obtenido	Observaciones
1.	Desconectarse de la aplicación.	Un mensaje de que se va a desconectar y cierre de la aplicación.	El mensaje y el cierre.	Correcto.

Tabla 21: Pruebas Desconectarse de la Aplicación

7.21 Ver Acerca De.

Código	Descripción	Resultado Esperado	Resultado Obtenido	Observaciones
1.	Ver el mensaje de información de la aplicación.	El mensaje.	El mensaje.	Correcto.

Tabla 22: Pruebas Ver Acerca De.

7.22 Borrar una receta.

Código	Descripción	Resultado Esperado	Resultado Obtenido	Observaciones
1.	Borrar una receta propia del usuario.	Un mensaje de que se a borrar y el borrado.	El mensaje.	Correcto.

Tabla 23: Pruebas Borrar una receta

7.23 Modificar una receta

Código	Descripción	Resultado Esperado	Resultado Obtenido	Observaciones
1.	Modificar una receta sin los campos obligatorios.	Un aviso de que dichos campos deben ser completados.	El aviso.	Correcto.
2.	Modificar una receta con al menos los campos obligatorios.	Un mensaje de que la receta se ha modificado correctamente.	El mensaje.	**Mejora**.

Tabla 24: Pruebas Modificar una Receta

En este caso, en vez de utilizar Dialog Fragments como en el caso de Guardar Receta, se optó por implementar actividades nuevas para modificar los Ingredientes y la Elaboración, dado que primero se muestran los datos actuales de la receta, los cuales se pueden mantener, modificar o añadir sobre ellos. Se llegó a la conclusión de que la comunicación de tanta información era más fácil de llevar a través de actividades completas y no de diálogos.

7.24 Ordenar recetas favoritas

Código	Descripción	Resultado Esperado	Resultado Obtenido	Observaciones
1.	Ordenar recetas favoritas por el campo “Dieta”.	El listado de recetas ordenadas.	El listado.	Correcto.
2.	Ordenar recetas favoritas por el campo “Cocina”.	El listado de recetas ordenadas.	El listado.	Correcto.
3.	Ordenar recetas favoritas por el campo “Nombre”.	El listado de recetas ordenadas.	El listado.	Correcto.

Tabla 25: Pruebas Ordenar recetas favoritas

7.25 Puntuar una receta

Código	Descripción	Resultado Esperado	Resultado Obtenido	Observaciones
1.	Puntuar una receta con un número entero.	Un mensaje de que la puntuación ha sido añadida.	El mensaje.	Correcto.
2.	Puntuar una receta con un número de topo Float.	Un mensaje de que la puntuación ha sido añadida.	Error.	**Error.**

Tabla 26: Pruebas puntuar una receta

El problema se debía a que a nivel de la Datastore, las puntuaciones se habían definido como números enteros, por lo cual al meter un Float, daba error. La solución fue definir la puntuación como un número de tipo Float a nivel de servidor.

7.26 Reconocimiento de voz

Código	Descripción	Resultado Esperado	Resultado Obtenido	Observaciones
1.	Utilizar alguna de las palabras clave cuando está escuchando.	La tarea esperada por cada una de esas palabras.	La tarea.	Correcto.
2.	No decir nada cuando la aplicación está escuchando.	Que la aplicación escuche de manera indefinida.	La aplicación paraba de escuchar.	**Error.**
3.	Cerrar la actividad mediante el botón de navegación.	Que la actividad termine con el reconocimiento de voz.	Error y cierre de la aplicación.	**Error.**

Tabla 27: Pruebas Reconocimiento de Voz

El primer error se debía a que en el momento en el que el listener no escuchaba nada durante los seis segundos de ejecución, se paraba y terminaba en error. Para evitar esto lo que se hizo fue implementar el método de inicio del listener, de tal manera que cada vez que terminara de esta manera porque el usuario no había dicho nada, volviera a ejecutarse, y así, de manera indefinida.

El segundo error, en cambio, ocurría debido a que no se había especificado qué hacer en el momento de cierre de la actividad, por lo cual, el listener seguía escuchando, y al terminar en error, dado que el método ya no se encontraba accesible, provocaba el cierre de la aplicación. De esta manera, se pasaron a implementar los métodos *onDestroy()*, *onStop()*, *onResume()*, y *onPause()* propios de una actividad, y en los cuáles se para y se termina con la ejecución del listener cuando se cierra la actividad.

8. CONCLUSIONES Y TRABAJO FUTURO

En este apartado se tratarán las conclusiones del proyecto a todos los niveles implicados, tanto personales como las conclusiones finales del proyecto.

8.1 Conclusiones de gestión

En cuanto a la planificación temporal realizada al comienzo del desarrollo de esta aplicación, cabe destacar que el proyecto se ha retrasado completamente debido a diversos temas personales, entre los que se encuentra el hecho de compaginar un trabajo a tiempo parcial con el desarrollo de este proyecto y demás asuntos personales. En este caso, no había ningún plan de contigencia preparado, dado que la naturaleza de los problemas era imprevisible.

Asimismo, la fase de aprendizaje ha sido muchísimo más larga de lo esperado, dado que requería aprender a programar en un lenguaje totalmente desconocido como lo era Python, así como aprender a utilizar y trabajar con una herramienta tan innovadora como lo es Google AppEngine.

Por otro lado, la planificación de la fase de implementación también se ha retrasado, debido a la necesidad de primero controlar las herramientas que componían la parte servidor como se ha explicado en el párrafo anterior.

El desarrollo en Android ha resultado ser menos complejo de lo esperado, a pesar de que algunas de las funcionalidades implementadas hayan resultado ser bastante complicadas, como puede ser el caso del reconocimiento de voz, o la lógica similar a la de una red social que se ha construido entre los perfiles de los usuarios.

En este caso, este retraso sí que estaba entre los posibles riesgos, y la solución dada en su día fue dedicar más horas al aprendizaje tanto de Python como de Android. Solución totalmente satisfactoria, pero que conllevó bastantes más horas de desarrollo.

De esta manera, la estimación temporal del proyecto se había puesto como fecha de presentación la convocatoria de julio, retrasándolo hasta la convocatoria de noviembre del 2014. A continuación se muestra un gráfico y una tabla con las diferencias entre las horas estimadas y una aproximación de las horas invertidas en realizar el proyecto.

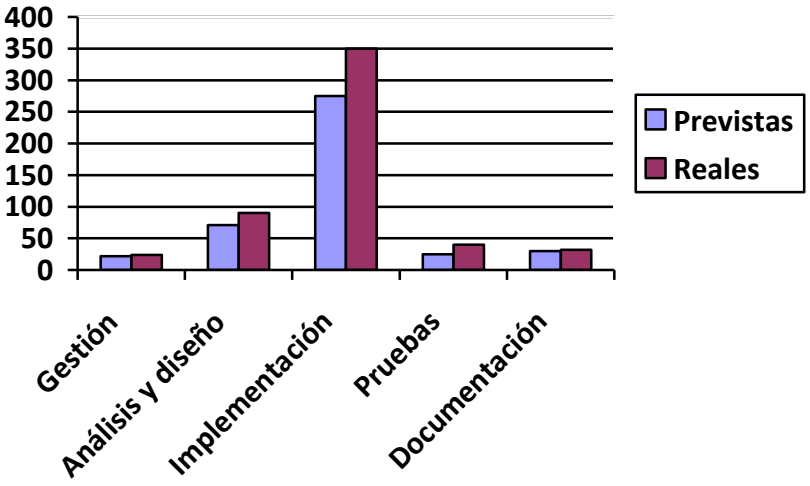


Ilustración 75: Gráfica de estimaciones

	Previstas	Reales
Gestión	**22**	**24**
Análisis y diseño	**71**	**90**
Implementación	**275**	**350**
Pruebas	**25**	**40**
Documentación	**30**	**32**
TOTAL	**423**	**536**

Tabla 28: Horas previstas y reales

De esta manera, en total se han dedicado **113** horas más al proyecto, sobre todo en la tarea de Implementación, donde la diferencia de horas previstas y reales es bastante amplia. Por ende, el coste total de la aplicación ha aumentado también notoriamente.

GASTOS	**Previstos**	**Reales**
Hardware	**18,20 euros**	**22,40 euos**
Portátil	13,20 euros	16,25 euros
Móvil	4,00 euros	6,15 euros
Software	**21 euros**	**21 euros**
Personal	**12690 euros**	**16080 euros**
Otros	**30 euros**	**30 euros**
TOTAL	**12733,2 euros**	**16153,4 euros**

Tabla 29: Gastos previstos y reales

Asimismo, la amortización tanto del portátil como del teléfono utilizado es mayor, dado que se han invertido 113 horas más en su uso. Además, esas horas de diferencia son 3390 euros más en cuanto a personal de programación, por lo cual el retraso en la planficación ha encarecido el desarrollo de la aplicación en 3420,2 euros. De esta manera, para recuperar la inversión:

- Si el precio de la aplicación es de 1 euro , el número mínimo de compradores para obtener beneficios es el siguiente:
  - Beneficios ((x*1 euro) – (x*0.3)) – 16153,40 – 20 = 0) => **23104. 18219**.

- Si el precio de la aplicación asciende a 3 euros, , el número mínimo de compradores para obtener beneficios es el siguiente:
  - Beneficios ((x*3 euro) – (x*0.3)) – 16153,40 – 20 = 0) => **5990. 4724.**

Por lo que, teniendo en cuenta los datos estimados al comienzo de la planificación, si la aplicación costara un euro sería necesario que 4885 personas más compraran la aplicación para recuperar la inversión. Y de tal manera, si la aplicación costara tres euros, serían necesarias 1266 personas más. Una cantidad de gente, que teniendo en cuenta el mercado actual, sería un poco difícil de conseguir.

8.2 Cumplimiento de objetivos

En cuanto a los objetivos establecidos al comienzo del desarrollo de esta aplicación, podría decirse que todos se han cumplido. Dos de los objetivos principales eran aprender a desarrollar aplicaciones en Android haciendo uso de la herramienta App Engine y el lenguaje Python y se han cumplido dado que se ha desarrollado una aplicación completa y funcional, así como se han

conseguido los conocimientos necesarios para poder afrontar el desarrollo de otros proyectos de temática similar en el futuro.

Los dos otros objetivos versaban sobre la utilidad de la aplicación de cara al usuario final. Como usuaria, podría decir que la aplicación cumple con mis expectativas, pero no sería totalmente objetiva. Para ver si estos dos objetivos se han cumplido todavía sería necesario esperar un tiempo y probar finalmente, con un número elevado de usuarios diferentes, si de verdad la aplicación aporta utilidad en el dia a día.

8.3 Conclusiones personales

A nivel personal la realización del proyecto ha resultado ser un gran paso a diferentes niveles. Se ha mejorado y aumentado la capacidad de concentración, la investigación y sobre todo el autoaprendizaje, puntos que se consideran bastante importantes hoy en día. Asimismo, se ha conseguido la motivación necesaria para no terminar este proyecto aquí, sino pensar en diferentes líneas futuras, así como en diferentes proyectos futuros en los que poder utilizar todas las habilidades y herramientas conseguidas en este. Pero sobre todo, la sensación de satisfacción personal obtenida cuando finalmente se ha conseguido, después de todas las frustraciones, finalizar este proyecto.

8.4 Líneas futuras

En cuanto a las líneas de futuro planteadas, se han considerado algunos aspectos en los que la aplicación todavía puede mejorar y aumentar:

- Más y mejores funciones sociales entre los usuarios, ya sea mediante mensajes o un chat, conseguir una mejor comunicación entre todos.
- Añadir la opción de agregar vídeos sobre las recetas en el apartado Elaboración, con el fin de mejorar la experiencia de la aplicación.
- Mejorar la funcionalidad de la Lista de la Compra, y quizás poder implementar la opción de crear un carrito de la compra asociado en alguna cadena de Supermercados.
- Conseguir evitar los pitidos en la funcionalidad “Cocinar”, que a pesar de no ser especialmente ruidosos, no son el resultado esperado.

Entre todas estas mejoras, las más probables de implementar de cara a un futuro son las primeras dos, dado que no se depende de librerías de terceros o modificaciones en las APIs

habidas para poder lograrlas. De esta manera, y como aproximación, ambas funcionalidades podrían estar listas en dos meses.

En cuanto a la Lista de la Compra, a pesar de ser una idea que incluso podría atraer ingresos, sería necesario hacer una búsqueda de qué supermercados permiten el acceso a sus datos o para los cuáles hay APIs disponibles e implementar una funcionalidad acorde, por lo cual, seguramente se necesitaría bastante tiempo. Lo mismo sucedería con la funcionalidad de "Cocinar". Dado que para esta aplicación no se ha encontrado una solución mejor, habría que esperar a que se actualizara la API de reconocimiento de voz, o a que alguien desarrollara una librería mediante la cual se podría impleentar esta funcionalidad, hecho que podría llevarnos meses.

9. BIBLIOGRAFÍA

Toda la bibliografía utilizada para el desarrollo de Cooking Stardust ha sido digital, sobre todo de la documentación oficial de Google tanto para la parte servidor como para la parte cliente.

http://developer.android.com/

https://cloud.google.com/appengine/docs/python/

http://www.vogella.com/tutorials/android.html

ANEXO I.

CASOS DE USO EXTENDIDOS

Los casos de uso extendidos explican de forma más detallada las diferentes funcionalidades que se quieren implementar.

1. Identificación

Nombre: Identificación

Descripción: Permite al usuario, previamente registrado, loguearse en la aplicación para poder acceder a las distintas funcionalidades que se les brinda a los usuarios registrados.

Actores: Usuario

Precondiciones: Ninguna

Requisitos funcionales: Ninguno

Flujo de eventos:

1. El usuario rellena los campos de “Usuario” y “Contraseña” y pulsará en “Login” (Ilustración 72).
2. El sistema comprobará si los datos son correctos. Para ello, accederá a la base de datos y buscará si dicho usuario y dicha contraseña coinciden.
    a. Si los datos son correctos, el usuario accederá a la interfaz de usuario logueado, o Inicio (Ilustración 73).
    b. Si los datos no son correctos, se elevará un error y se volverá a la página de login (Ilustración 74).

Poscondiciones: El usuario se habrá logueado en el sistema y por consiguiente, accederá a su página de inicio.

Interfaz gráfica:

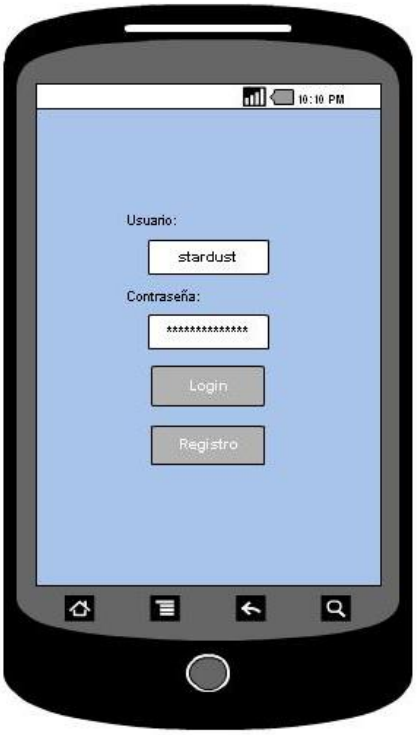


Ilustración 76: Identificación 1

Ilustración 77: Identificación 2

Ilustración 78: Identificación 3

2. Registro

Nombre: Registro

Descripción: Permite al usuario darse de alta en la aplicación, de tal modo que pueda acceder a las funcionalidades.

Actores: Usuario

Precondiciones: Ninguna

Requisitos funcionales: Ninguno

Flujo de eventos:

1. El usuario pulsará el botón de "Registrarse" y rellenará los campos establecidos: nickname, contraseña, cuenta de correo. Después, pulsará "Guardar" (Ilustración 75 y 76).
2. El sistema comprobará que dicho nickname no esté en la base de datos.
    c. Si el usuario es correcto, se continuará a la página inicial (Ilustración 77).
    d. Si el nickname existe, se elevará un error y se volverá a la página de registro (Ilustración 78).
3. Una vez realizada la comprobación, se enviará un correo informativo al usuario con su nickname y su contraseña.

Poscondiciones: El usuario se ha dado de alta y está logueado en su página de inicio.

Interfaz gráfica:

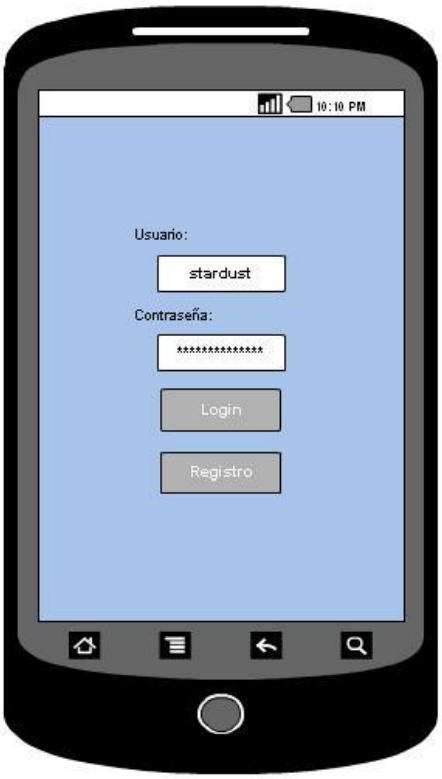


Ilustración 79: Registro 1

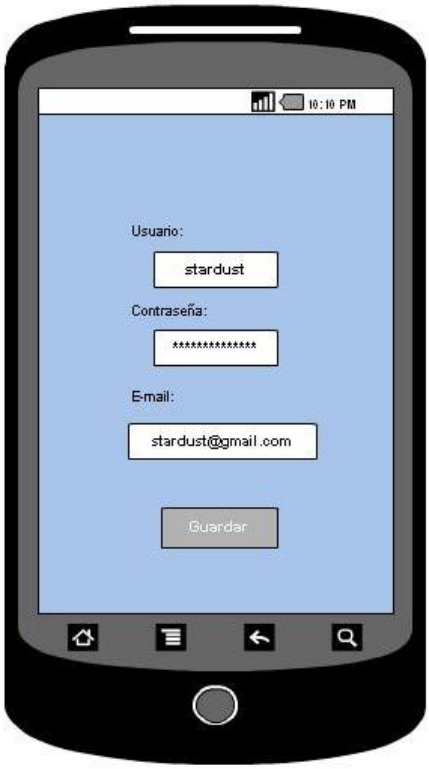


Ilustración 80: Registro 2

Ilustración 81: Registro 3

Ilustración 82: Registro 4

3. Consultar Recetas

Nombre: Consultar Recetas

Descripción: Permite al usuario, una vez registrado, acceder al listado de sus recetas y a funcionalidades como modificar, borrar y crear nuevas recetas.

Actores: Usuario

Precondiciones: Estar logueado

Requisitos funcionales: Ninguno

Flujo de eventos:

1. El usuario pulsará el botón de "Mi Recetario" (Ilustración 79).
2. El sistema le listará desde la base de datos las recetas que ése usuario ha creado.
3. Si pincha sobre una receta, aparecerán sus detalles.

Poscondiciones: El usuario ha accedido a sus recetas.

Interfaz gráfica:

Ilustración 83: Consultar Recetas

4. Ver detalles receta

Nombre: Ver detalles receta

Descripción: Permite al usuario, una vez registrado, ver los detalles de una receta, con el fin de consultarla, modificarla, borrarla, hacer una lista de la compra o cocinarla.

Actores: Usuario

Precondiciones: Estar logueado en el sistema registrado

Requisitos funcionales: Ninguno

Flujo de eventos:

1. El usuario accederá a esta funcionalidad, tanto mediante el caso de uso "Consultar Recetas", como por "Favoritos", "Recomendaciones" o mediante los diferentes métodos de búsqueda.
2. Aparecerán todos los datos correspondientes a la receta, así como nombre, dificultad ingredientes, elaboración… (Ilustración 80).

Poscondiciones: El usuario ha consultado los detalles de una receta.

Interfaz gráfica:

Ilustración 84: Ver detalle receta

5. Guardar Receta

Nombre: Guardar receta

Descripción: Permite al usuario, una vez registrado, guardar una receta, añadiendo todos los detalles necesarios: ingredientes, dificultad...

Actores: Usuario

Precondiciones: Estar logueado en el sistema registrado

Requisitos funcionales: Ninguno

Flujo de eventos:

1. El usuario accede a la página principal de la aplicación.
2. El usuario, mediante el menú de opciones de la izquierda, accede a "Guardar Receta".
3. Aparecerá una nueva interfaz en la que habrá diferentes campos (ingredientes, elaboración, dificultad...) que el usuario deberá rellenar (Ilustración 81).
4. El usuario pulsará "Guardar" para guardar la receta (Ilustración 82).

Poscondiciones: El usuario ha creado una nueva receta que se añade a su listado.

Interfaz gráfica:

Ilustración 85: Guardar Receta 1

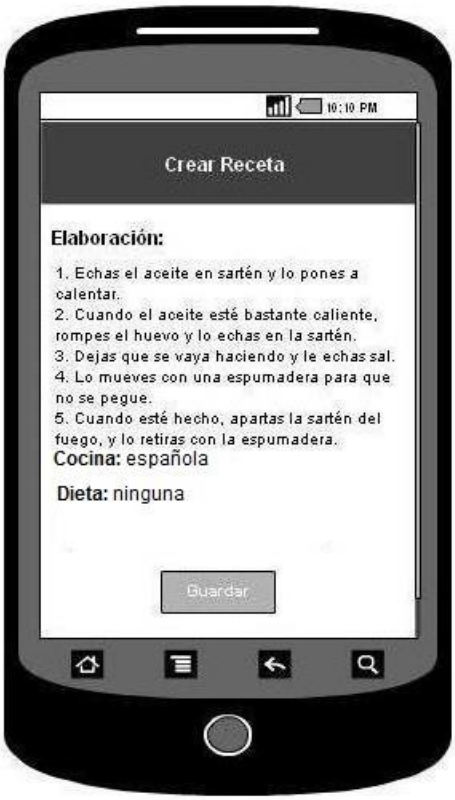


Ilustración 86: Guardar Receta 2

6. Agregar Ingrediente

Nombre: Agregar Ingrediente

Descripción: Permite al usuario, una vez ha accedido a la funcionalidad Guardar Receta, guardar uno o varios ingredientes.

Actores: Usuario

Precondiciones: Acceder a Guardar Receta.

Requisitos funcionales: Ninguno

Flujo de eventos:

1. El usuario pulsa en Agregar Ingrediente.
2. Se muestra un diálogo en el cual el usuario podrá agregar un ingrediente por cada vez, especificando cantidad y nombre.
3. El usuario pulsará "Guardar" para guardar el ingrediente (Ilustración 83).

Poscondiciones: El usuario ha guardado un ingrediente a la lista de ingredientes de la receta a guardar.

Interfaz gráfica:

Ilustración 87: Añadir Ingrediente

7. Agregar Paso

Nombre: Agregar Paso

Descripción: Permite al usuario, una vez ha accedido a la funcionalidad Guardar Receta, guardar uno o varios pasos en la elaboración.

Actores: Usuario

Precondiciones: Acceder a Guardar Receta.

Requisitos funcionales: Ninguno

Flujo de eventos:

1. El usuario pulsa en Agregar Paso.
2. Se muestra un diálogo en el cual el usuario podrá agregar un paso por cada vez.
3. El usuario pulsará "Guardar" para guardar el paso en la lista de elaboración. (Ilustración 84).

Poscondiciones: El usuario ha guardado un paso a la lista de elaboración de la receta a guardar.

Interfaz gráfica:

Ilustración 88: Añadir Paso

8. Agregar Foto

Nombre: Agregar Foto

Descripción: Permite al usuario, una vez ha accedido a la funcionalidad Guardar Receta, guardar una foto de la receta que se está creando.

Actores: Usuario

Precondiciones: Acceder a Guardar Receta.

Requisitos funcionales: Ninguno

Flujo de eventos:

1. El usuario pulsa en Agregar Foto.
2. Se muestra un diálogo en el cual el usuario podrá agregar una foto que tenga en la galería o sacar una foto desde la cámara.
3. El usuario pulsará "Guardar" para guardar la foto. (Ilustración 85).

Poscondiciones: El usuario ha guardado una foto de la receta a guardar.

Interfaz gráfica:

Ilustración 89: Añadir Foto

9. Agregar foto por paso

Nombre: Agregar Foto por Paso

Descripción: Permite al usuario, una vez ha accedido a la funcionalidad Guardar Receta, guardar una foto por cada paso de la elaboración.

Actores: Usuario

Precondiciones: Acceder a Guardar Receta.

Requisitos funcionales: Ninguno

Flujo de eventos:

1. El usuario pulsa en Agregar Foto por Paso.
2. Se muestra un diálogo en el cual el usuario podrá agregar una foto que tenga en la galería o sacar una foto desde la cámara.
3. El usuario pulsará "Guardar" para guardar la foto. (Ilustración 86).

Poscondiciones: El usuario ha guardado una foto delos pasos de la receta a guardar.

Interfaz gráfica:

Ilustración 90: Añadir foto por paso

10. Modificar Receta

Nombre: Modificar receta

Descripción: Permite al usuario, una vez registrado, modificar una receta ya creada en caso de que lo considere necesario.

Actores: Usuario

Precondiciones: que la receta exista.

Requisitos funcionales: Ninguno

Flujo de eventos:

1. El usuario accederá alguna de sus recetas de "Mi Recetario".
2. El usuario, mediante el botón de opciones del móvil, accede a "Modificar". (Ilustración 87)
3. Aparecerá una nueva pantalla que le permitirá modificar los campos necesarios (Ilustración 88).
4. El usuario pulsará "Guardar" para guardar la receta o "Cancelar" en caso contrario (Ilustración 89).

Poscondiciones: El usuario ha modificado una receta existente.

Interfaz gráfica:

Ilustración 91: Modificar Receta 1

Ilustración 92: Modificar Receta 2

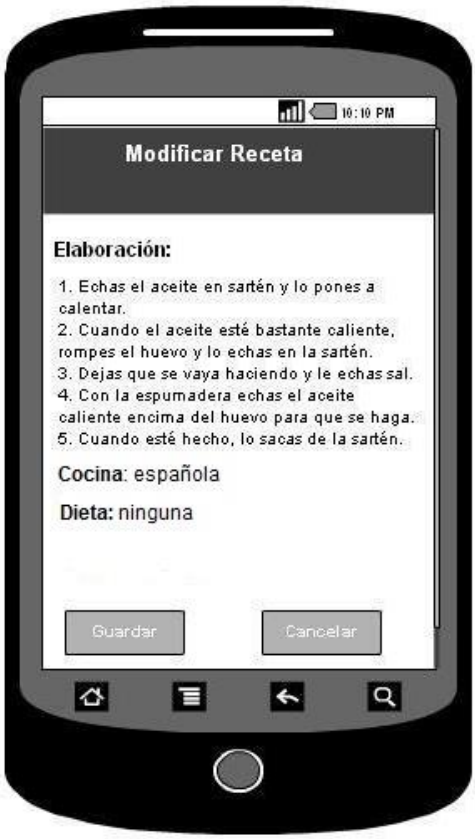


Ilustración 93: Modificar Receta 3

11. Modificar Ingredientes

Nombre: Modificar Ingredientes

Descripción: Permite al usuario, una vez ha accedido a la funcionalidad Modificar Receta, guardar uno o varios ingredientes.

Actores: Usuario

Precondiciones: Acceder a Modificar Receta.

Requisitos funcionales: Ninguno

Flujo de eventos:

1. El usuario pulsa en Modificar Ingrediente.
2. Se muestra un diálogo en el cual el usuario podrá modificar, borrar o añadir un ingrediente por cada vez, especificando cantidad y nombre.
3. El usuario pulsará "Guardar" para guardar el ingrediente (Ilustración 90).

Poscondiciones: El usuario ha modificado un ingrediente a la lista de ingredientes de la receta a modificar.

Interfaz gráfica:

Ilustración 94: Modificar Ingredientes

12. Modificar Elaboración

Nombre: Modificar Elaboración

Descripción: Permite al usuario, una vez ha accedido a la funcionalidad Modificar Receta, guardar uno o varios pasos de la elaboración.

Actores: Usuario

Precondiciones: Acceder a Modificar Receta.

Requisitos funcionales: Ninguno

Flujo de eventos:

1. El usuario pulsa en Modificar Elaboración.
2. Se muestra un diálogo en el cual el usuario podrá modificar, borrar o añadir un paso por cada vez.
3. El usuario pulsará "Guardar" para guardar el paso (Ilustración 91).

Poscondiciones: El usuario ha modificado un paso a la lista de elaboración de la receta a modificar.

Interfaz gráfica:

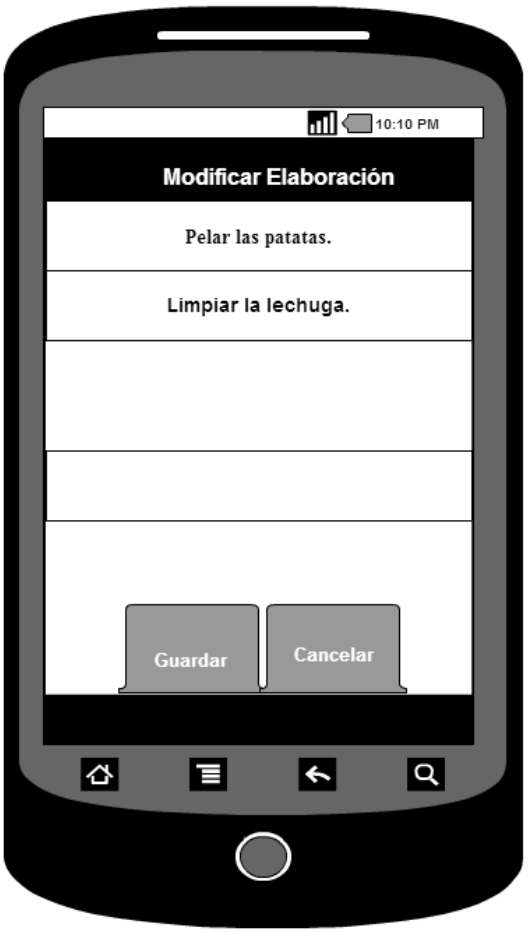


Ilustración 95: Modificar Elaboración

13. Modificar Foto

Nombre: Modificar Foto

Descripción: Permite al usuario, una vez ha accedido a la funcionalidad Modificar Receta, modificar la foto de la receta.

Actores: Usuario

Precondiciones: Acceder a Modificar Receta.

Requisitos funcionales: Ninguno

Flujo de eventos:

1. El usuario pulsa en Modificar Foto.
2. Se muestra un diálogo en el cual el usuario podrá modificar la foto actual, por una que quiera cargar tanto de la cámara como de la galería. (Ilustración 92)

Poscondiciones: El usuario ha modificado o borrado la foto de la receta a modificar.

Interfaz gráfica:

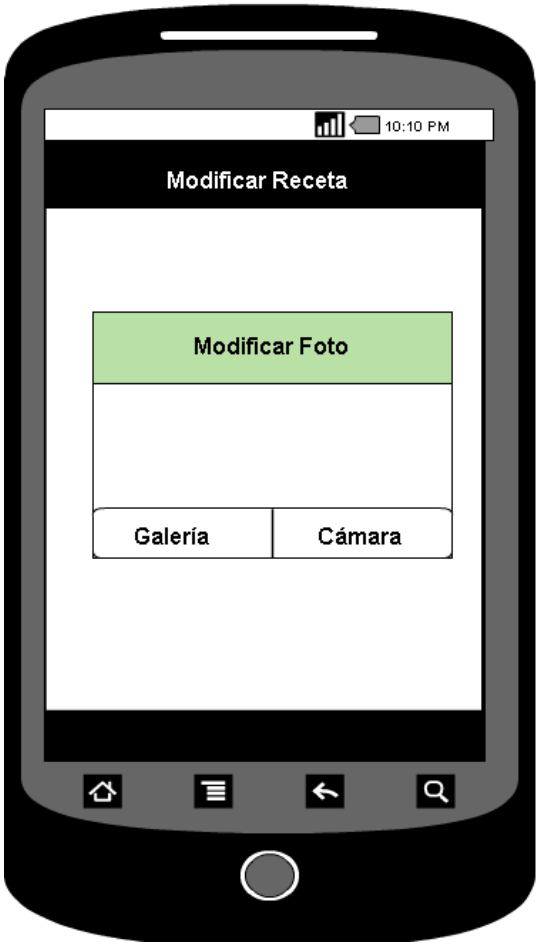


Ilustración 96: Modificar Foto

14. Modificar foto por paso

Nombre: Modificar Foto por Paso

Descripción: Permite al usuario, una vez ha accedido a la funcionalidad Modificar Receta, modificar la foto asociada a cada paso de la receta.

Actores: Usuario

Precondiciones: Acceder a Modificar Receta.

Requisitos funcionales: Ninguno

Flujo de eventos:

1. El usuario pulsa en Modificar Foto por Paso.
2. Se muestra un diálogo en el cual el usuario podrá modificar la foto actual, por una que quiera cargar tanto de la cámara como de la galería. (Ilustración 93)

Poscondiciones: El usuario ha modificado o borrado la foto del paso asociado de la receta a modificar.

Interfaz gráfica:

Ilustración 97: Modificar Foto por paso

15. Borrar Receta

Nombre: Borrar receta

Descripción: Permite al usuario, una vez registrado, borrar una receta o varias ya creadas.

Actores: Usuario

Precondiciones: que la receta exista.

Requisitos funcionales: Ninguno

Flujo de eventos:

1. El usuario accederá a "Mis Recetas".
2. El usuario, mediante el botón de opciones del móvil, accede a "Borrar Receta".
3. Le aparecerá un listado de las recetas a borrar, y el usuario pulsará una o varias (Ilustración 94).
4. El usuario pulsará en "Borrar" (Ilustración 95).

Poscondiciones: El usuario ha borrado una o varias recetas existentes de su listado de recetas, sin que ello afecte a los demás usuarios.

Interfaz gráfica:

Ilustración 98: Borrar Receta 1

Ilustración 99: Borrar Receta 2

16. Buscar Receta Avanzada

Nombre: Buscar receta Avanzada

Descripción: Permite al usuario, una vez registrado, buscar una receta teniendo en cuenta los ingredientes, tipos de cocina, dietas...

Actores: Usuario

Precondiciones: estar logueado en el sistema.

Requisitos funcionales: Ninguno

Flujo de eventos:

1. El usuario accederá a "Buscar Receta".
2. El usuario rellenará los campos que considere necesario (ingredientes, dieta, cocina...). (Ilustración 96).
3. Se buscará en la base de datos los datos introducidos.
    a. Si no se encuentra nada que coincida con todos los datos introducidos, saldrá una ventana: "No existen recetas" (Ilustración 97).
    b. Sino, saldrá el listado de recetas que concuerden. (Ilustración 98)

Poscondiciones: El usuario ha consultado una o varias recetas.

Interfaz gráfica:

Ilustración 100: Buscar Receta Avanzado 1

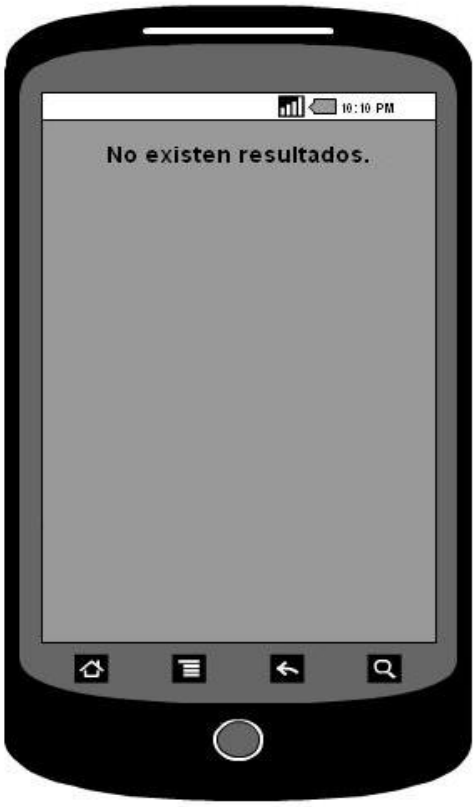


Ilustración 101: Buscar Receta Avanzado 2

Ilustración 102: Buscar Receta Avanzado 4

17. Buscar Receta

Nombre: Buscar receta

Descripción: Permite al usuario, una vez registrado, buscar una receta por nombre haciendo uso del widget de búsqueda en la Action Bar.

Actores: Usuario

Precondiciones: estar logueado en el sistema.

Requisitos funcionales: Ninguno

Flujo de eventos:

4. El usuario accederá la pantalla principal, en la cual se muestra el widget de búsqueda.
5. El usuario rellenará el campo con el nombre de la receta que quiera buscar.
6. Se buscará en la base de datos los datos introducidos. (Ilustración 99)
    c. Si no se encuentra nada que coincida con todos los datos introducidos, saldrá una ventana: "No existen recetas" (Ilustración 100).
    d. Sino, saldrá el listado de recetas que concuerden. (Ilustración 101)

Poscondiciones: El usuario ha buscado una o varias recetas.

Interfaz gráfica:

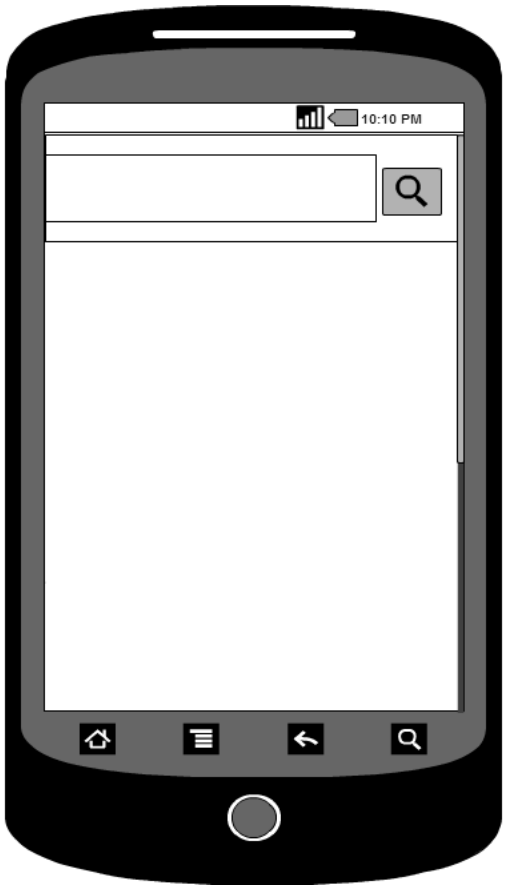


Ilustración 103: Buscar Receta 1

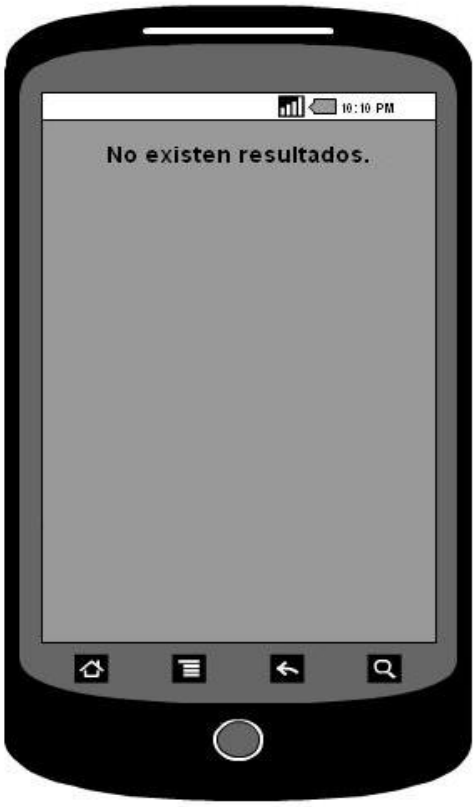


Ilustración 104: Buscar Receta 2

Ilustración 105: Buscar Receta 3

18. Buscar Receta por Localización

Nombre: Buscar receta por Localización

Descripción: Permite al usuario, una vez registrado, buscar las recetas cercanas a él en un radio de cincuenta kilómetros.

Actores: Usuario

Precondiciones: estar logueado en el sistema y tener la localización activada.

Requisitos funcionales: Ninguno

Flujo de eventos:

7. El usuario accederá a la pantalla de resultados de la búsqueda simple.
8. El usuario pinchará en el botón "Localización". (Ilustración 102).
9. Se buscará en la base de datos las recetas que tengan las coordenadas guardadas en un radio de cincuenta kilómetros.
    e. Si no se encuentra nada que coincida con todos los datos introducidos, saldrá una ventana: "No existen recetas cerca" (Ilustración 103).
    f. Sino, saldrá el listado de recetas que concuerden. (Ilustración 104)

Poscondiciones: El usuario ha buscado una o varias recetas por localización.

Interfaz gráfica:

Ilustración 106: Buscar Receta por Localización 1

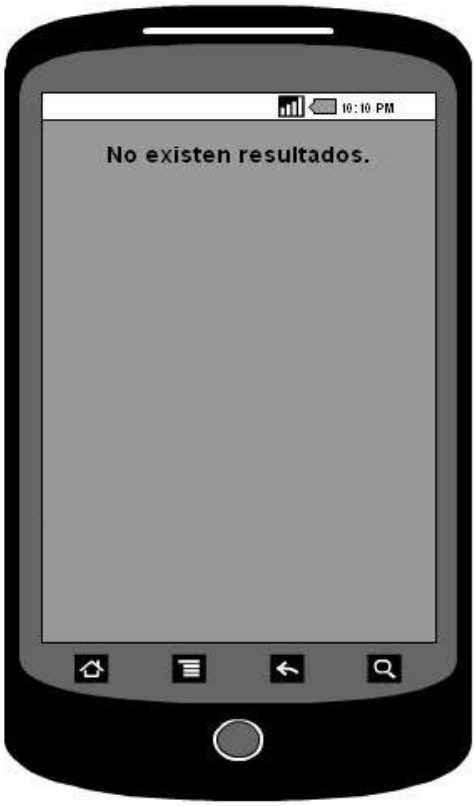


Ilustración 107: Buscar Receta por Localización 2

Ilustración 108: Buscar Receta por Localización 3

19. Guardar Receta en Favoritos

Nombre: Guardar receta en favoritos

Descripción: Permite al usuario, una vez registrado, buscar una receta teniendo en cuenta los ingredientes, tipos de cocina, dietas... y añadirla a favoritos.

Actores: Usuario

Precondiciones: estar logueado en el sistema.

Requisitos funcionales: Ninguno

Flujo de eventos:

1. El usuario buscará una receta (tanto en buscar recetas como en recomendaciones) y accederá a sus detalles (Ilustración 105).
2. Si considera que la receta le gusta podrá guardarla en sus favoritos. Para ello, deberá pulsar la opción "Guardar" mediante el botón de opciones del móvil (Ilustración 106).

Poscondiciones: El usuario ha guardado una o varias recetas en sus favoritos.

Interfaz gráfica:

Ilustración 109: Guardar Receta en Favoritos 1

Ilustración 110: Guardar Receta en Favoritos 2

20. Lista de la Compra

Nombre: Lista de la compra

Descripción: Permite al usuario, una vez registrado, buscar una receta teniendo en cuenta los ingredientes, tipos de cocina, dietas... y generar la lista asociada a esa receta y comensales y al número de comensales establecido.

Actores: Usuario

Precondiciones: estar logueado en el sistema.

Requisitos funcionales: Ninguno

Flujo de eventos:

1. El usuario accederá a una receta (tanto en buscar recetas como en recomendaciones) y accederá a sus detalles (Ilustración 107 y 108).
2. Si considera que la receta le gusta, podrá darle a "Lista de la Compra", establecer el número de comensales y el sistema le mostrará la lista de la compra asociada a esa receta. (Ilustración 109 y 110).

Poscondiciones: El usuario ha consultado la lista de la compra de la receta para los comensales que establezca.

Interfaz gráfica:

Ilustración 111: Lista de la compra 1

Ilustración 112: Lista de la compra 2

Ilustración 113: Lista de la compra 3

Ilustración 114: Lista de la compra 4

21. Perfil

Nombre: Perfil

Descripción: Permite al usuario, una vez registrado, acceder a su perfil.

Actores: Usuario

Precondiciones: estar logueado en el sistema.

Requisitos funcionales: Ninguno

Flujo de eventos:

1. El usuario accederá a "Perfil". (Ilustración 111).
2. El usuario podrá acceder a todos los datos asociados a su perfil, como a diferentes funcionalidades: Ver Recetas, Ver Seguidores, Ver Siguiendo y Modificar Perfil.

Poscondiciones: El usuario ha accedido a su perfil.

Interfaz gráfica:

Ilustración 115: Perfil

Modificar Perfil

Nombre: Modificar Perfil

Descripción: Permite al usuario, una vez registrado, modificar su perfil (salvo el nickname).

Actores: Usuario

Precondiciones: estar logueado en el sistema.

Requisitos funcionales: Ninguno

Flujo de eventos:

1. El usuario accederá a "Perfil".
2. El usuario pinchará en "Modificar Perfil".
3. El usuario podrá editar los campos que considere necesarios. (Ilustración 112).
4. Una vez haya terminado el usuario pulsará "Guardar".
    a. Si pulsa "Guardar", se guardarán los datos en la base de datos del sistema.
    b. Sino, no se almacenarán los cambios.

Poscondiciones: El usuario ha editado su perfil.

Interfaz gráfica:

Ilustración 116: Modificar Perfil

22. Consultar Seguidores/Siguiendo

Nombre: Consultar Seguidores/Siguiendo

Descripción: Permite al usuario, una vez registrado, consultar el listado de seguidores o siguiendo.

Actores: Usuario

Precondiciones: estar logueado en el sistema.

Requisitos funcionales: Ninguno

Flujo de eventos:

1. El usuario accederá a su perfil, desde el cual podrá pinchar en "Seguidores" o "Siguiendo".
2. El sistema le listará desde la base de datos las personas de las que es amigo el usuario (Ilustración 113).
3. El usuario podrá acceder a los detalles del amigo, sólo tendrá que elegir el usuario en cuestión. (Ilustración 114).

Poscondiciones: El usuario ha consultado los amigos que tiene.

Interfaz gráfica:

Ilustración 117: Ver Seguidores 1

Ilustración 118: Ver Seguidores 2

23. Buscar Amigos

Nombre: Buscar Amigos

Descripción: Permite al usuario, una vez registrado, buscar usuarios.

Actores: Usuario

Precondiciones: estar logueado en el sistema.

Requisitos funcionales: Ninguno

Flujo de eventos:

1. El usuario accederá a "Amigos".
2. El sistema le listará desde la base de datos las personas de las que es amigo el usuario.
3. El usuario pulsará el botón de opciones "Buscar".
4. Le saldrá una ventana para que rellene el campo de "nickname" (Ilustración 115).
5. El sistema buscará en la base de datos el nombre introducido.
    a. Si no lo encuentra, aparecerá un mensaje de error: "No existe el usuario" (Ilustración 116).
    b. Si lo encuentra, devolverá un listado con todos los nombres que concuerden. (Ilustración 117).

Interfaz:

Ilustración 119: Buscar Amigos 1

Ilustración 120: Buscar Amigos 2

Ilustración 121: Buscar Amigos 3

24. Añadir a Siguiendo

Nombre: Añadir a Siguiendo

Descripción: Permite al usuario, una vez registrado añadirlos un usuario buscado al listado de amigos.

Actores: Usuario

Precondiciones: estar logueado en el sistema.

Requisitos funcionales: Ninguno

Flujo de eventos:

1. El usuario accederá a buscar un usuario y se le mostrará su perfil.
2. Si el usuario quiere, puede añadir ese usuario a su listado de amigos.
3. El usuario pulsará el botón de "Seguir" (Ilustración 118).
4. El sistema procederá a guardar el usuario a su lista de amigos.
    c. Si ya se encuentra, el botón tendrá la opción "Dejar de Seguir.
    d. Si no, lo guardará.

Interfaz:

Ilustración 122: Añadir a Siguiendo

25. Favoritos

Nombre: Favoritos

Descripción: Permite al usuario, una vez registrado, consultar las recetas que tiene guardadas como favoritas.

Actores: Usuario

Precondiciones: estar logueado en el sistema.

Requisitos funcionales: Ninguno

Flujo de eventos:

1. El usuario accederá a "Favoritas".
2. El sistema le listará desde la base de datos las recetas que tiene guardadas (Ilustración 119).
3. Mediante el botón de opciones, podrá puntuar o modificar el orden de las recetas.

Poscondiciones: El usuario ha consultado sus recetas favoritas.

Interfaz gráfica:

Ilustración 123: Favoritas

26. Ordenar Favoritos

Nombre: Ordenar favoritos

Descripción: Permite al usuario, una vez registrado, ordenar las recetas que tiene guardadas como favoritas, dependiendo el tipo de ingrediente (pasta, legumbre, verduras...), el tipo de dieta (vegetariano, diabético) o el tipo de cocina (china, argentina, inglesa...).

Actores: Usuario

Precondiciones: estar logueado en el sistema.

Requisitos funcionales: Ninguno

Flujo de eventos:

1. El usuario accederá a "Favoritas".
2. Mediante el botón de opciones, el usuario accederá a "Ordenar Favoritas".
3. Se le abrirá un listado con las opciones por las que quiera ordenar (Ilustración 120).
4. El usuario seleccionará una de ellas y le pulsará "Aceptar".
5. El sistema le listará sus recetas favoritas según el criterio establecido (Ilustración 121).

Poscondiciones: El usuario ha consultado sus recetas favoritas según su criterio.

Interfaz gráfica:

Ilustración 124: Ordenar Favoritas 1

Ilustración 125: Ordenar Favoritas 2

27. Puntuar Recetas

Nombre: Puntuar recetas

Descripción: Permite al usuario, una vez registrado, puntuar las recetas.

Actores: Usuario

Precondiciones: estar logueado en el sistema.

Requisitos funcionales: Ninguno

Flujo de eventos:

6. El usuario accederá a cualquier receta.
7. Mediante el botón de opciones, el usuario accederá a "Puntuar Favoritas".
8. El sistema le listará el nombre de las recetas y a la izquierda un apartado desde una a cinco estrellas. (Ilustración 122).
9. El usuario seleccionará el número de estrellas que quiera otorgar a las recetas y pulsará "Aceptar".
    a. En caso que la receta esté puntuada con anterioridad, se tendrá la opción de modificar
10. El sistema guardará los cambios.

Poscondiciones: El usuario ha puntuado sus recetas favoritas.

Interfaz gráfica:

Ilustración 126: Puntuar

28. Recomendaciones

Nombre: Recomendaciones

Descripción: Permite al usuario, una vez registrado, recibir recomendaciones sobre recetas por el sistema, teniendo en cuenta las puntuaciones y las recetas guardadas o consultadas.

Actores: Usuario

Precondiciones: estar logueado en el sistema.

Requisitos funcionales: Ninguno

Flujo de eventos:

1. El usuario accederá a "Recomendaciones".
2. Se abrirá una nueva interfaz, donde se le mostrarán al usuario, recetas recomendadas (Ilustración 44).
3. Si pulsa sobre alguna, se le mostrarán los detalles de la misma.

Poscondiciones: ninguna.

Interfaz gráfica:

Ilustración 127: Recomendaciones

29. Cocinar Receta

Nombre: Cocinar Receta

Descripción: Permite al usuario, una vez registrado, consultar la receta y los diferentes pasos sin tener que utilizar las manos y gracias al sistema de reconocimiento de voz.

Actores: Usuario

Precondiciones: estar logueado en el sistema.

Requisitos funcionales: Ninguno

Flujo de eventos:

1. El usuario accederá a "Consultar Recetas" o "Buscar Recetas", desde donde accederá a los detalles de la misma. (Ilustración 124)
2. El usuario le dará a "Cocinar" y se le mostrará una pantalla por cada paso en la elaboración de la receta. El usuario podrá avanzar o retroceder diciendo en voz alta "Adelante" o "Atrás". También dispondrá de los botones asociados a las dos acciones. (Ilustración 125 y 126).

Poscondiciones: ninguna.

Interfaz gráfica:

Ilustración 128: Cocinar Receta 1

Ilustración 129: Cocinar Receta 2

Ilustración 130: Cocinar Receta 3

ANEXO II. DIAGRAMAS DE SECUENCIA

En todos los diagramas de secuencia que hay en este anexo, las clases se diferencian en dos colores: azul para todas aquellas clases pertenecientes a Android y rojo para todas aquellas que forman parte del lado del servidor, la API y la Datastore.

1. Registrar Usuario

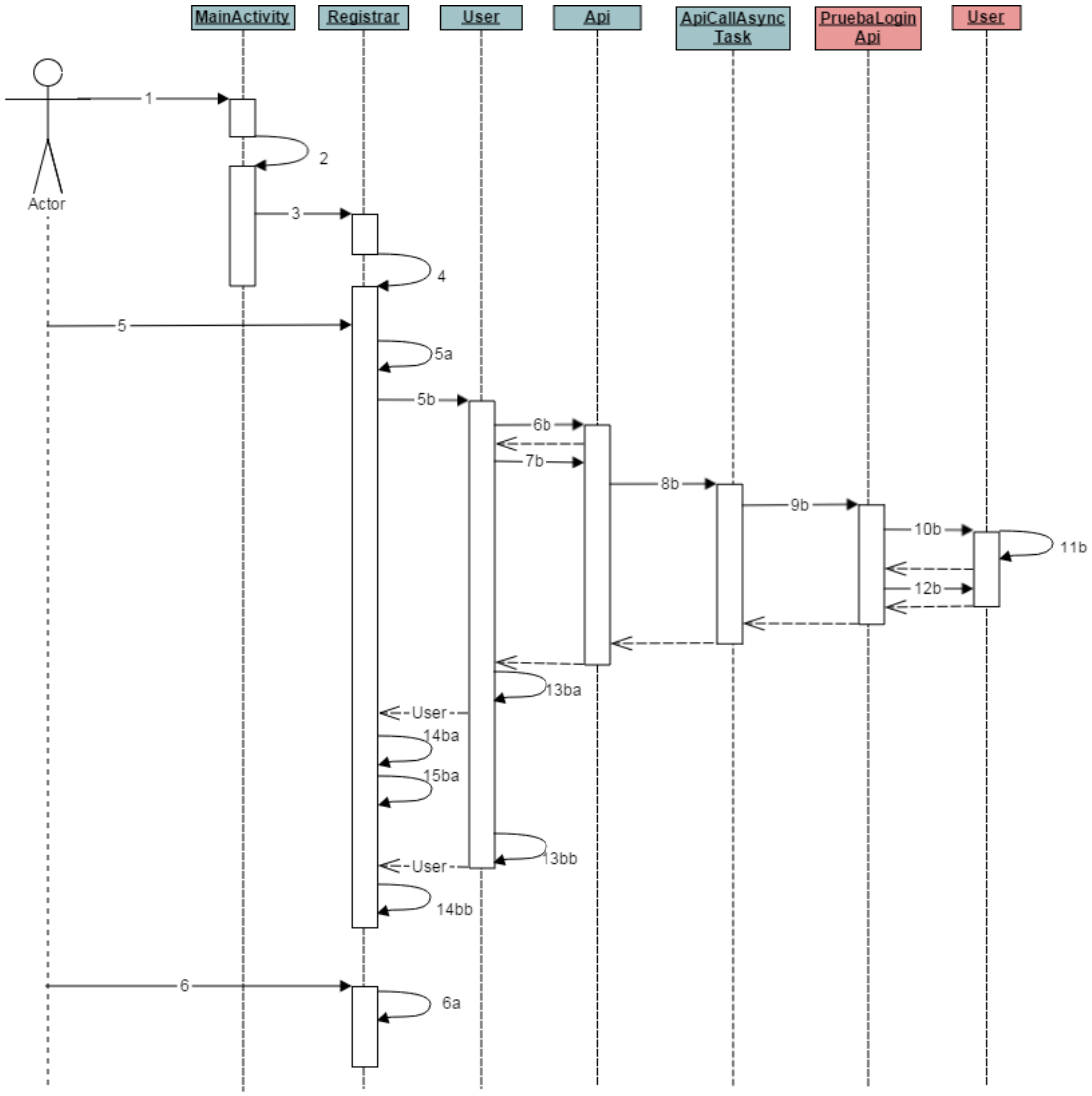


Ilustración 131: Diag. Sec. Registrar Usuario

1. El usuario inicia la aplicacion.

2. onCreate(): void

3. El usuario pincha en el botón Registrar - new Intent(this, Registrar.class)

4. onCreate(): void

5. El usuario pincha en Guardar:

    Si el nombre, el nickname, o la contraseña están vacíos:

    5a. toast("Username and password cannot be empty");

    Sino:

    5b. User.create(nick, pass, name, age, sex, email, new Callback<User>());

    6b. Api api = Api.getInstance();

7b. api.service.user.post(user, new Callback<CookingstardustMessagesUserAllResponseMessage>());

8b. new ApiCallAsyncTask<CookingstardustMessagesUserAllResponseMessage>(call, callback).execute();

9b. call.execute();

10b. User.put_from_message(request)

11b. user.put()

12b. user.to_message()

13b. Si el nickname se ha guardado correctamente:

13ba. onCompleted(null,User.parse(result));

14ba. toast("User " + result.getNickname() + " created. Check your e-mail");

15ba. this.finish()

Sino:

13bb. onCompleted(e, null);

14bb. toast("Error: User already exists in the database");

6. Si el usuario pincha en Cancelar:

6a. this.finish();

2. Loguear usuario

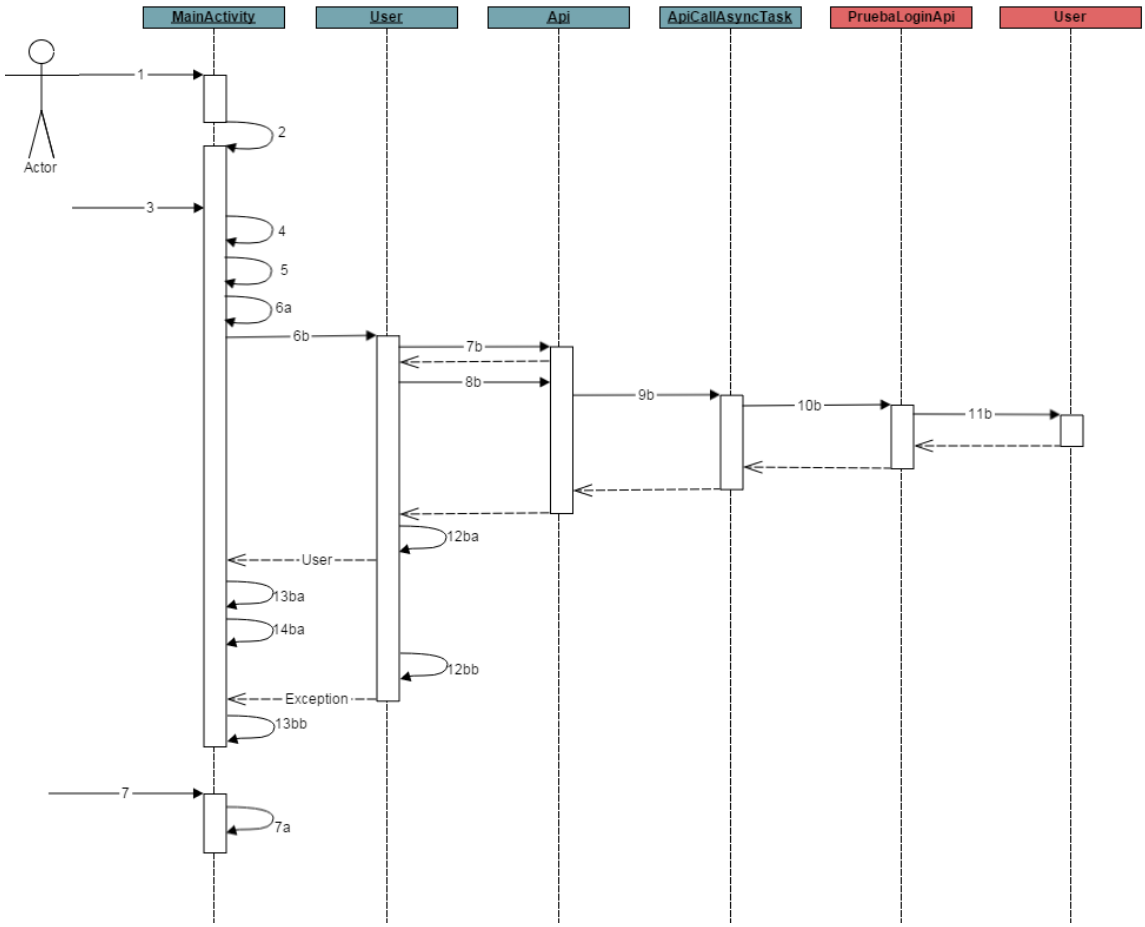


Ilustración 132: Diag. Sec. Loguear Usuario

1. El usuario inicia la aplicación.
2. onCreate(): void
3. El usuario pincha en Login.
4. String user = rellenarUsuario.getText().toString();
5. String pass = rellenarPass.getText().toString();
6. Si el nickname o la contraseña están vacíos:

    6a. toast("Introduce nick y contraseña!");

    Sino:

    6b. User.login(usuario, new Callback<User>());

    7b. Api api = Api.getInstance();

    8b. api.service.user.verify_user(usuario, new Callback<CookingstardustMessagesUserLoginMessage>());

    9b. new ApiCallAsyncTask<CookingstardustMessagesUserLoginMessage>(call, callback).execute();

    10b. call.execute();

    11b. User.verify_user(request)

12. Si el usuario y la contraseña son correctas:

12ba. onCompleted(null,User.parse(result));

13ba. toast("¡Bienvenido, " + result.getNickname() + "!");

14ba. new Intent(this, Home.class)

Sino:

12bb. onCompleted(e, null);

13bb. toast("Usuario o contraseña incorrectas!");

7. Si el usuario pincha en Cancelar:

7a. this.finish();

3. Desconectar usuario

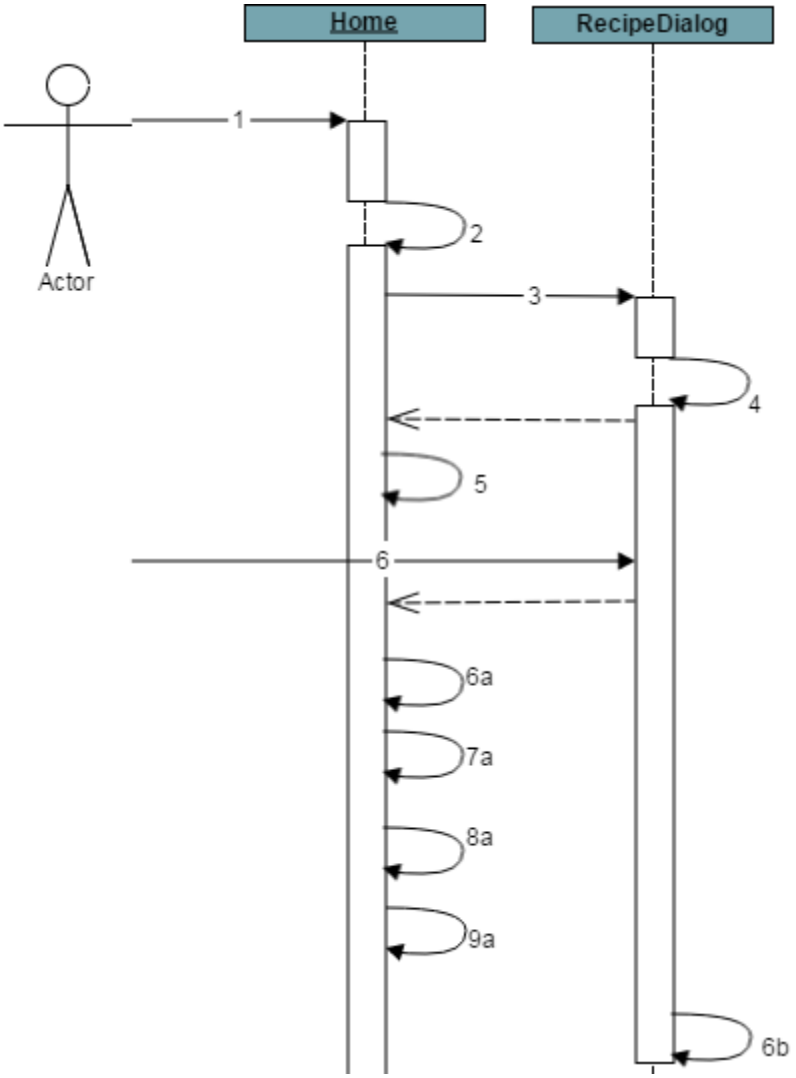


Ilustración 133: Diag. Sec. Desconectar Usuario

1. El usuario accede a la pantalla Home.
2. onCreate(): void
3. El usuario pincha en "Logout" en el menú deslizable de la izquierda. - new RecipeDialog();
4. onCreateDialog(): Dialog
5. dialogFragment.show(getFragmentManager(), "recipe_dialog");
6. Si el usuario ha pinchado "Aceptar"

    6a. SharedPreferences userID = getSharedPreferences(PREFS_NAME, 0);

    7a. SharedPreferences.Editor editor = userID.edit();

    8a. editor.clear().commit();

    9a. this.finish();

    Sino:

    6b. RecipeDialog.this.getDialog().cancel();

4. Ver listado de recetas

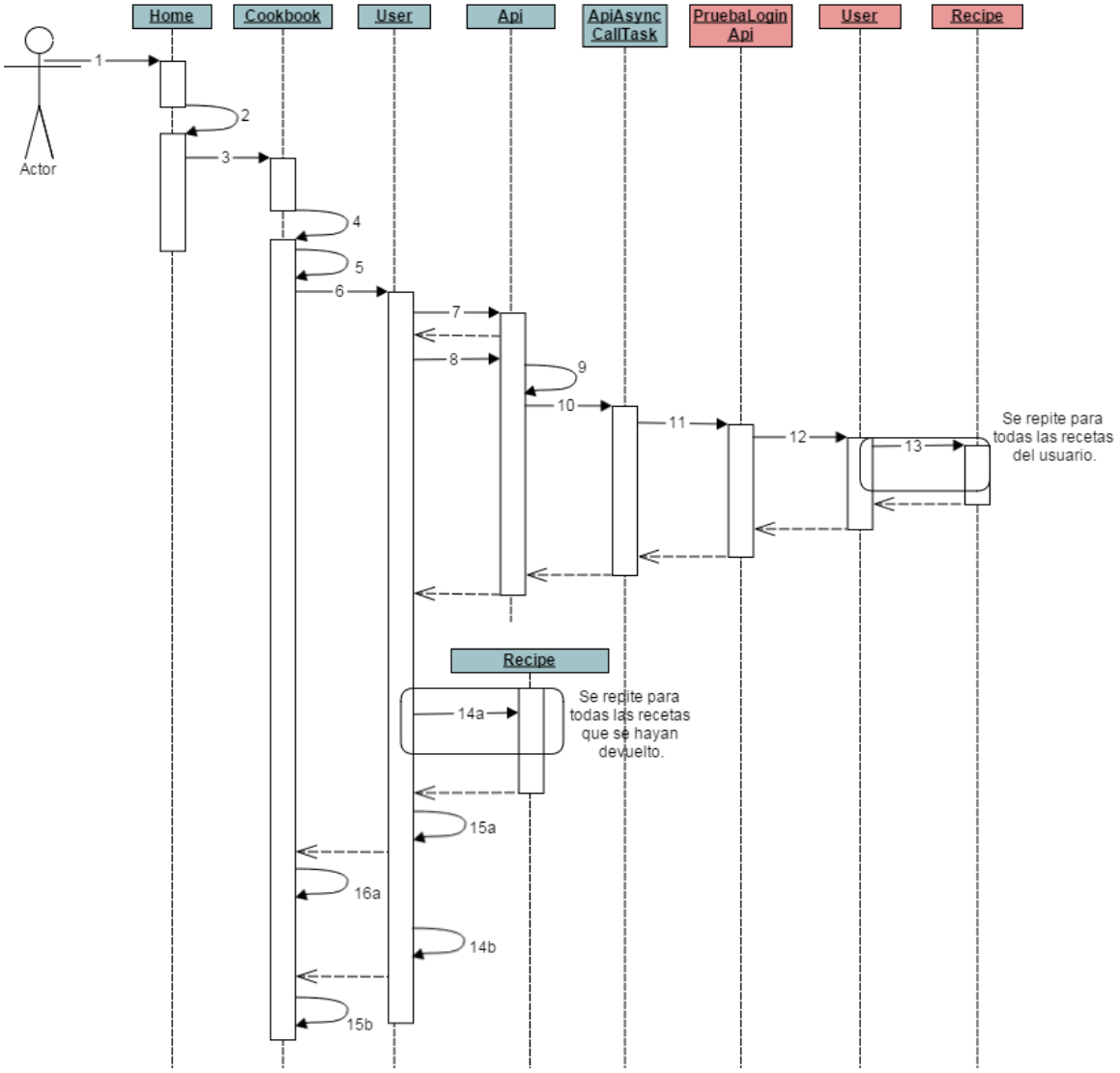


Ilustración 134: Diag. Sec. Ver Listado Recetas

1. El usuario accede a la pantalla Home.

2. onCreate(): void

3. El usuario pincha en "My Cookbook" en el menú deslizable de la izquierda. - listFragment = new Cookbook();

4. onCreate(): void

5. Long lon = getActivity().getSharedPreferences(PREFS_NAME, 0).getLong("userID", 4L);

6. list_recipes(lon, new Callback<ArrayList<Recipe>>();

7. Api api = Api.getInstance();

8. api.service().recipe()list_recipes(lon, new Callback<CookingstardustMessagesUserRecipeListResponse>();

9. Cookingstardust.Recipe.List call = request.list(lon);

10. new ApiCallAsyncTask<CookingstardustMessagesUserRecipeListResponse>(call, callback).execute();

11. call.execute();

12. User.list_recipes_from_user(request)

13. Recipe.to_message()

14. Si el usuario tenía recetas que listar (no es vacío):

    14a. Recipe.parse(item)

    15a. callback.onCompleted(null, response);

    16a. createListView(result);

Si no:

    14b. callback.onCompleted(e, null);

    15b. toast("Error: No hay recetas para este usuario!");

5. Ver detalle de una receta

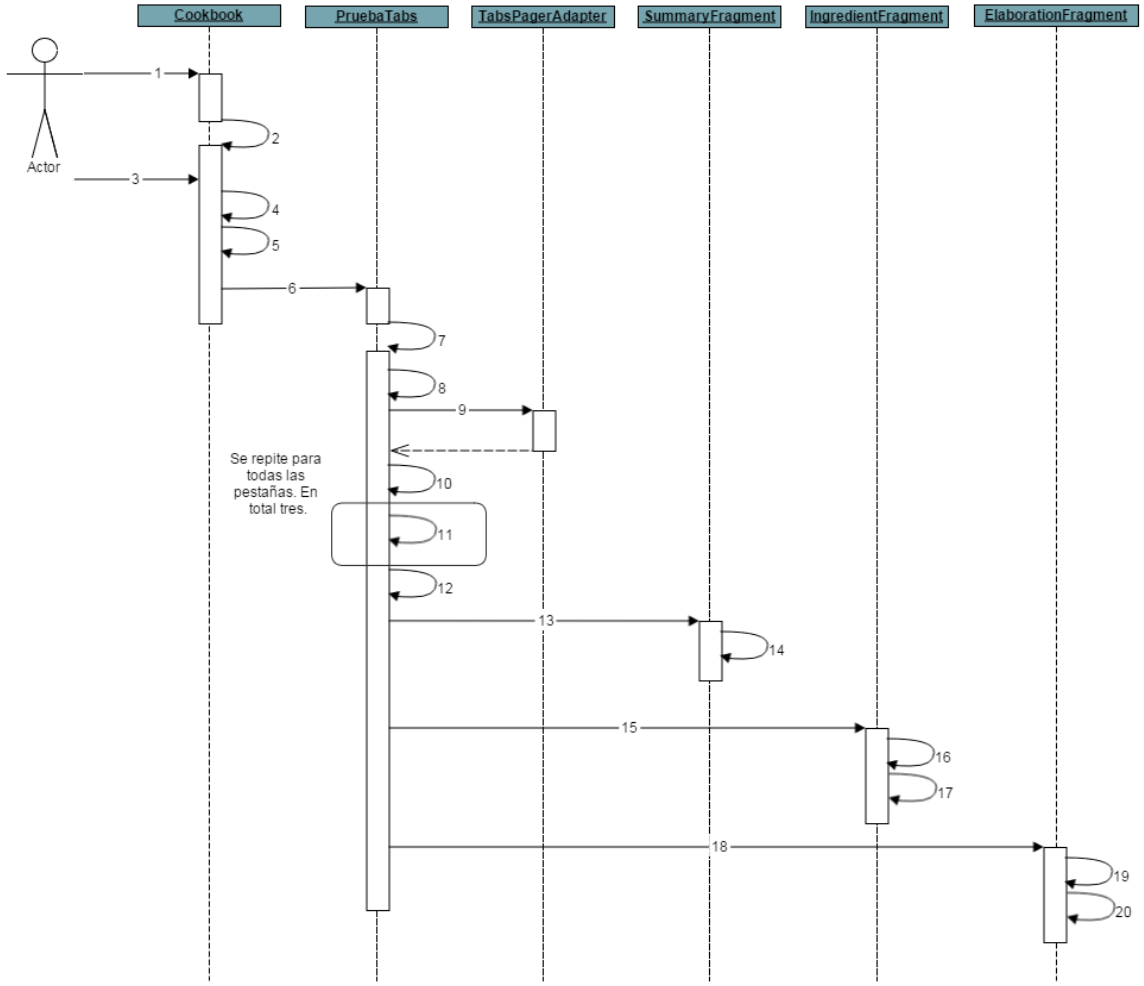


Ilustración 135: Diag. Sec. Ver Detalle Receta

1. El usuario accede a cualquier pantalla que tenga un listado de recetas (Mi Recetario, Recomendaciones, Favoritas, Resultado de búsqueda).
2. onCreate(): void
3. El usuario selecciona una de las recetas del listado.
4. Intent i = new Intent(this, PruebaTabs.class);
5. i.putExtras(extras);
6. startActivity(i);
7. onCreate(): void
8. actionBar = getActionBar();
9. mAdapter = new TabsPagerAdapter(getSupportFragmentManager());
10. viewPager.setAdapter(mAdapter);
11. actionBar.addTab(actionBar.newTab().setText(tab_name).setTabListener(this));
12. onCreateOptionsMenu(Menu menu): boolean

13. actionBar.setSelectedNavigationItem(position); //1

14. onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState): View

Dependiendo de para donde deslice el usuario la pantalla: a la izquierda:

15. actionBar.setSelectedNavigationItem(position); //2

16. onCreate(): void

17. onActivityCreated(Bundle savedInstanceState): void

18. actionBar.setSelectedNavigationItem(position); //3

19. onCreate(): void

20. onActivityCreated(Bundle savedInstanceState): void

6. Borrar una receta

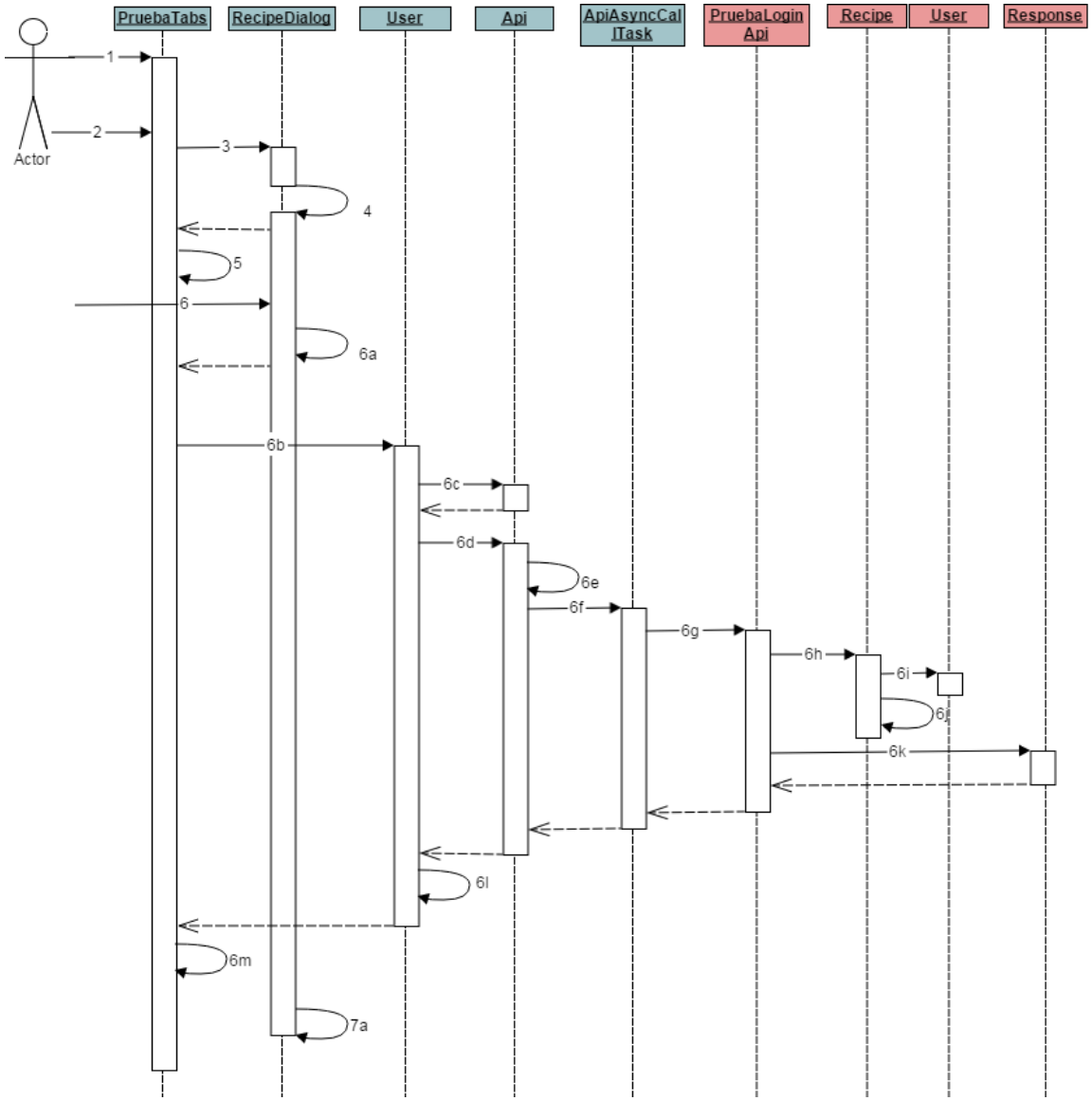


Ilustración 136: Diag. Sec. Borrar Receta

1. El usuario accede a cualquiera de sus recetas.

2. El usuario pulsa "Borrar Receta".

3. new RecipeDialog();

4. onCreateDialog(): Dialog

5. dialogFragment.show(getFragmentManager(), "recipe_dialog");

6. Si el usuario pincha en "Sí":

    6a. activity.onSendDataFragment("yes", 0, 0);

    6b. User.delete_recipe(lon, recipe.getRecipeId(), new Callback<Boolean>());

    6c. Api api = Api.getInstance();

    6d. api.service().recipe().delete_recipe(request, new Callback<CookingstardustMessagesResponse>();

    6e. Cookingstardust.Recipe.Delete call = request.delete(delete);

    6f. new ApiCallAsyncTask<CookingstardustMessagesResponse>(call, callback).execute();

    6g. call.execute();

    6h. Recipe.delete_recipe(request)

    6i. User.put()

    6j. recipe.key.delete()

    6k. return Response(there)

    6l. callback.onCompleted(null, resultado);

    6m. toast("Receta borrada");

7. Si el usuario pincha en no:

    7a. RecipeDialog.this.getDialog().cancel();

7. Puntuar una receta

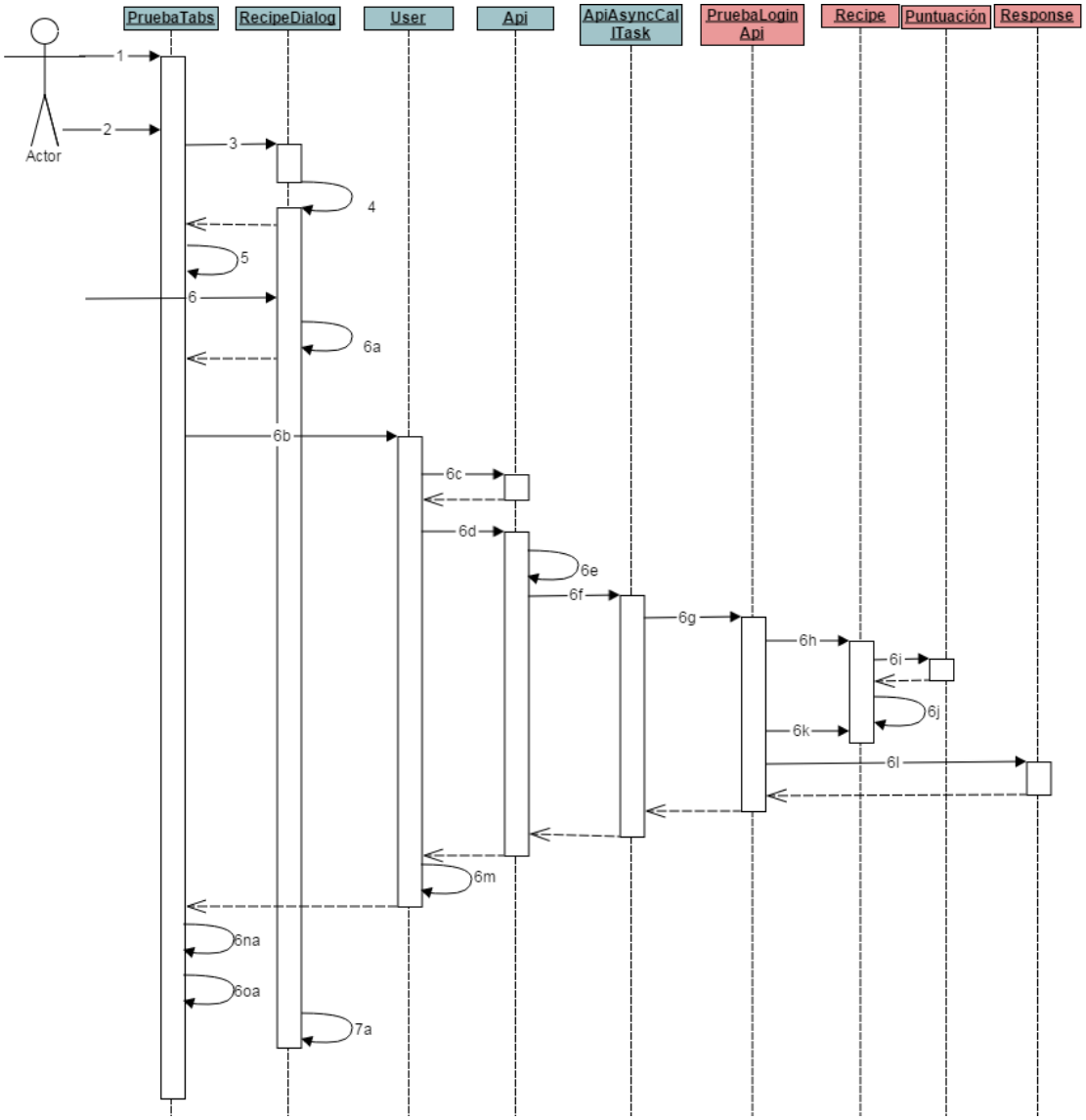


Ilustración 137: Diag. Sec. Puntuar Receta

1. El usuario accede a cualquier receta.
2. El usuario pulsa "Puntuar receta".
3. new RecipeDialog();
4. onCreateDialog(): Dialog
5. dialogFragment.show(getFragmentManager(), "recipe_dialog");
6. Si el usuario pincha en "Guardar":

    6a. activity.onSendDataFragment("puntuar", Math.round(puntuacion), 0);

    6b. User.addPuntuacion(lon, recipe.getRecipeId(), pos, new Callback<Boolean>());

6c. Api api = Api.getInstance();

6d. api.service().recipe().add_puntuacion(request, new Callback<CookingstardustMessagesResponse>();

6e. Cookingstardust.Recipe.AddPuntuacion call = request.addPuntuacion(puntos);

6f. new ApiCallAsyncTask<CookingstardustMessagesResponse>(call, callback).execute();

6g. call.execute();

6h. Recipe.add_recipe_punctuation(request)

6i. Puntuacion(user_id=message.user_id, puntos = message.puntos)

6j. recipe.put()

6k. Recipe.calcular_puntos(request)

6l. return Response(there)

6m. callback.onCompleted(null, resultado);

6n. Si el resultado es verdadero:

6na. toast("Puntuación de receta modificada");

6o. Si el resultado es falso:

6oa. toast("Puntuación añadida");

7. Si el usuario pincha en no:

7a. RecipeDialog.this.getDialog().cancel();

8. Añadir una receta a favoritos

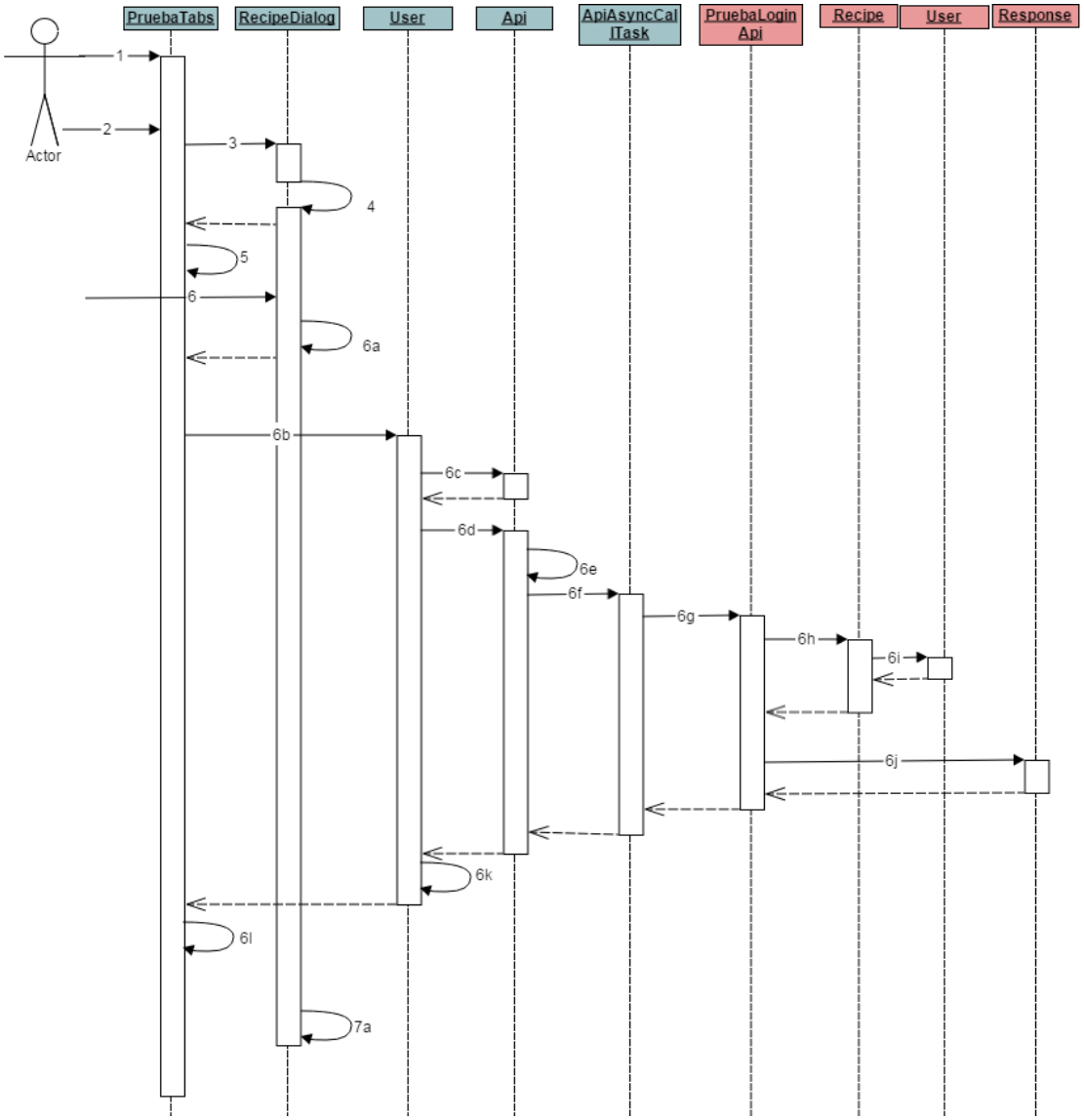


Ilustración 138: Diag. Sec. Añadir Receta a Favoritos.

1. El usuario accede a los detalles de cualquier receta (las que tenga ya en favoritos no tendrán esta opción):
2. El usuario pincha en "Favear Receta".
3. new RecipeDialog();
4. onCreateDialog(): Dialog
5. dialogFragment.show(getFragmentManager(), "recipe_dialog");
6. Si el usuario pincha en "Aceptar":

    6a. activity.onSendDataFragment("fav", 0, 0);

    6b. User.addFavorites(lon, recipe.getRecipeId(), new Callback<Boolean>());

    6c. Api api = Api.getInstance();

6d. api.service().recipe().fav_recipe(request, new Callback<CookingstardustMessagesResponse>());

6e. Cookingstardust.Recipe.AddFavorites call = request.addFavorites(fav);

6f. new ApiCallAsyncTask<CookingstardustMessagesResponse>(call, callback).execute();

6g. call.execute();

6h. Recipe.add_favorites(request)

6i. user.put()

6j. return Response(there)

6k. callback.onCompleted(null, resultado);

6l. toast("Receta añadida a favoritos");

7. Si el usuario pincha en "Cancelar":

7a. RecipeDialog.this.getDialog().cancel();

9. Quitar de favoritos una receta

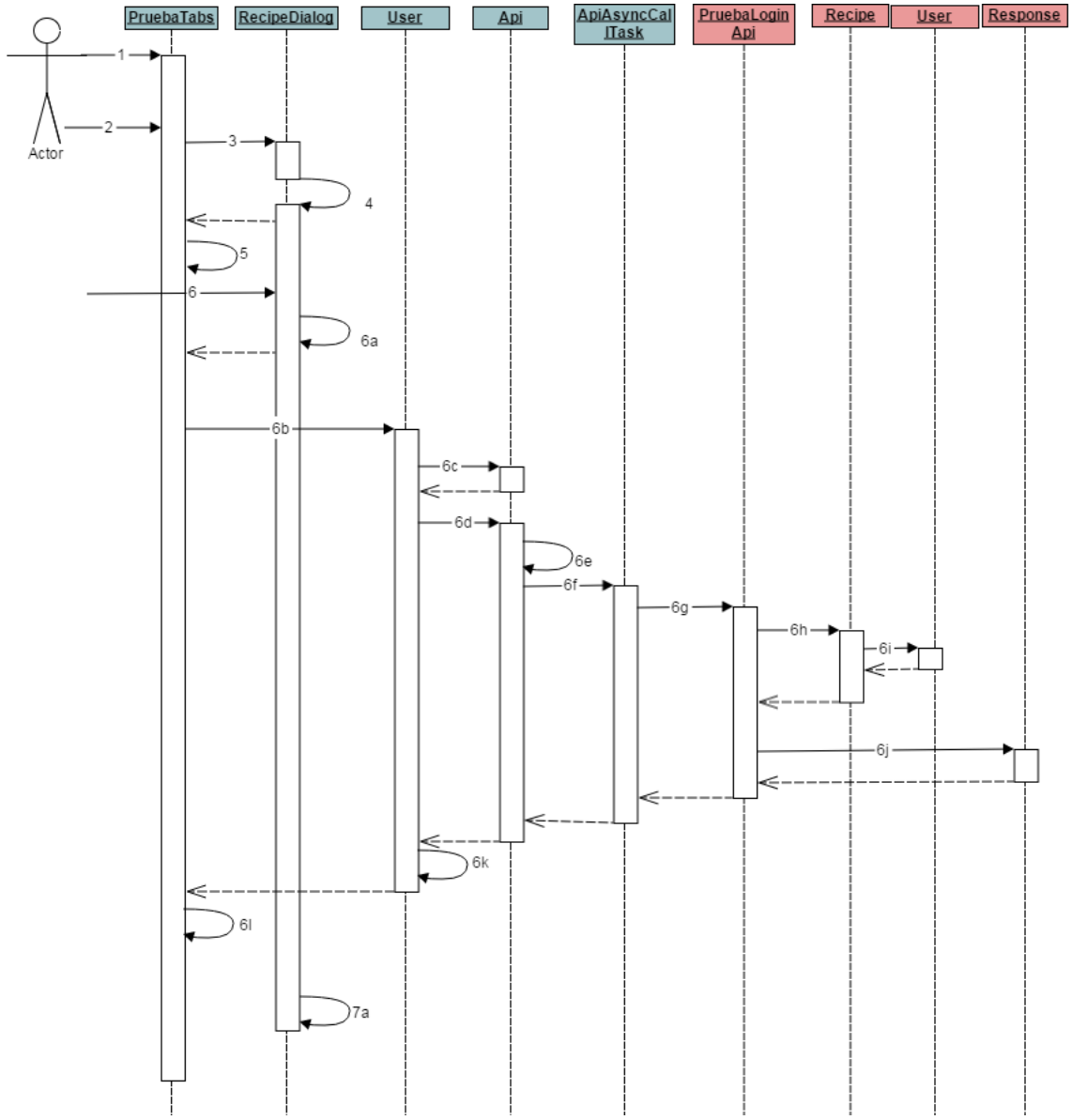


Ilustración 139: Diag. Sec. Quitar de favoritos una receta

1. El usuario accede a los detalles de cualquier receta suya que tenga en Favoritos:
2. El usuario pincha en "Desfavear Receta".
3. new RecipeDialog();
4. onCreateDialog(): Dialog
5. dialogFragment.show(getFragmentManager(), "recipe_dialog");
6. Si el usuario pincha en "Aceptar":

    6a. activity.onSendDataFragment("nofav", 0, 0);

    6b. User.deleteFavorites(lon, recipe.getRecipeId(), new Callback<Boolean>());

    6c. Api api = Api.getInstance();

    6d. api.service().recipe().delete_fav(request, new Callback<CookingstardustMessagesResponse>());

6e. Cookingstardust.Recipe.DeleteFavorites call = request.deleteFavorites(desfav);

6f. new ApiCallAsyncTask<CookingstardustMessagesResponse>(call, callback).execute();

6g. call.execute();

6h. Recipe.delete_favorites(request)

6i. user.put()

6j. return Response(there)

6k. callback.onCompleted(null, resultado);

6l. toast("Receta eliminada de favoritos");

7. Si el usuario pincha en "Cancelar":

7a. RecipeDialog.this.getDialog().cancel();

10.Ordenar recetas favoritas

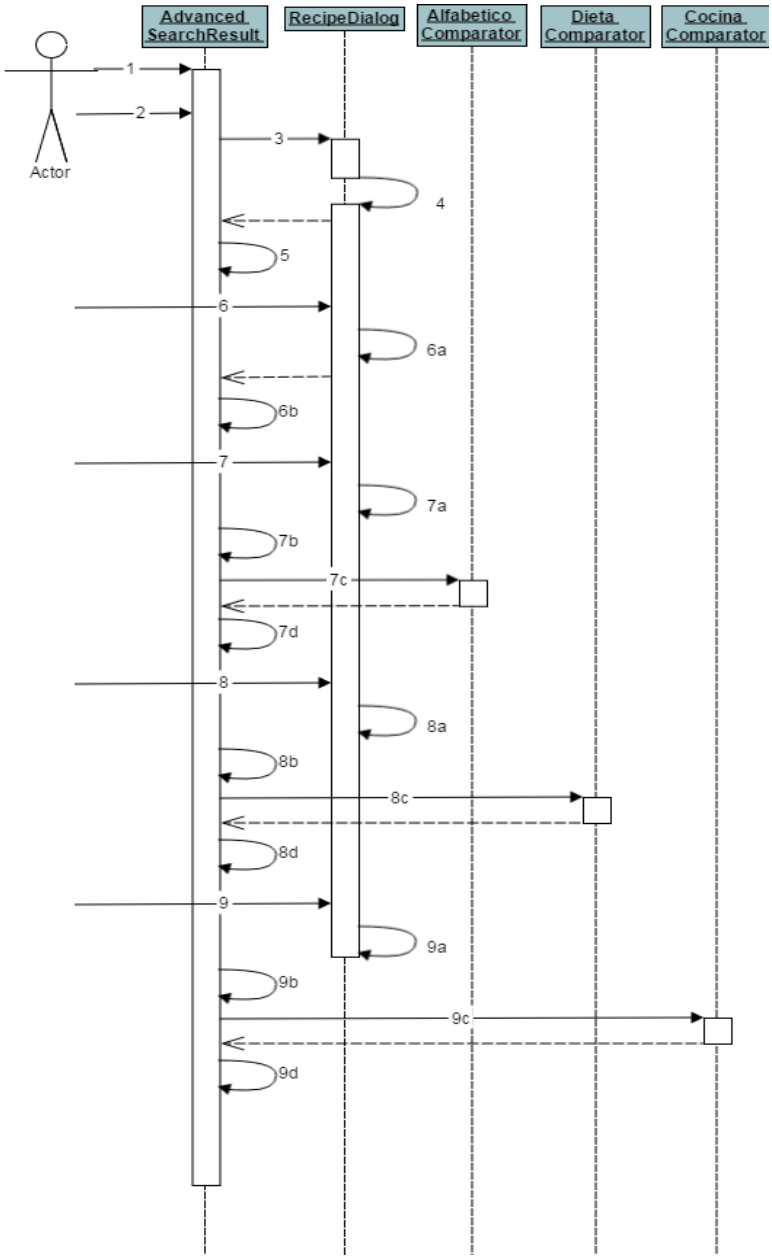


Ilustración 140: Diag. Sec. Ordenar Favoritas

1. El usuario accede a la pantalla de Recetas Favoritas.
2. El usuario pincha en "Ordenar".
3. new RecipeDialog();
4. onCreateDialog(): Dialog
5. dialogFragment.show(getFragmentManager(), "recipe_dialog");
6. Si el usuario no pincha en Ordenar sin haber escogido ninguna opción:
    6a. activity.onSendDataFragment("Ninguna", 1, 0);
    6b. toast("Tienes que elegir un orden!");
7. Si el usuario pincha en Ordenar y ha escogido "Nombre de la receta".
    7a. activity.onSendDataFragment("Nombre de la receta", 1, 0);

7b. orderRecipes(1);

7c. new AlfabeticoComparator();

7d. createListView(recs);

8. Si el usuario pincha en Ordenar y ha escogido "Dieta".

8a. activity.onSendDataFragment("Dieta", 1, 0);

8b. orderRecipes(2);

8c. new DietaComparator();

8d. createListView(recs);

9. Si el usuario pincha en Ordenar y ha escogido "Cocina".

9a. activity.onSendDataFragment("Cocina", 1, 0);

9b. orderRecipes(3);

9c. new CocinaComparator();

9d. createListView(recs);

11.Buscar Receta

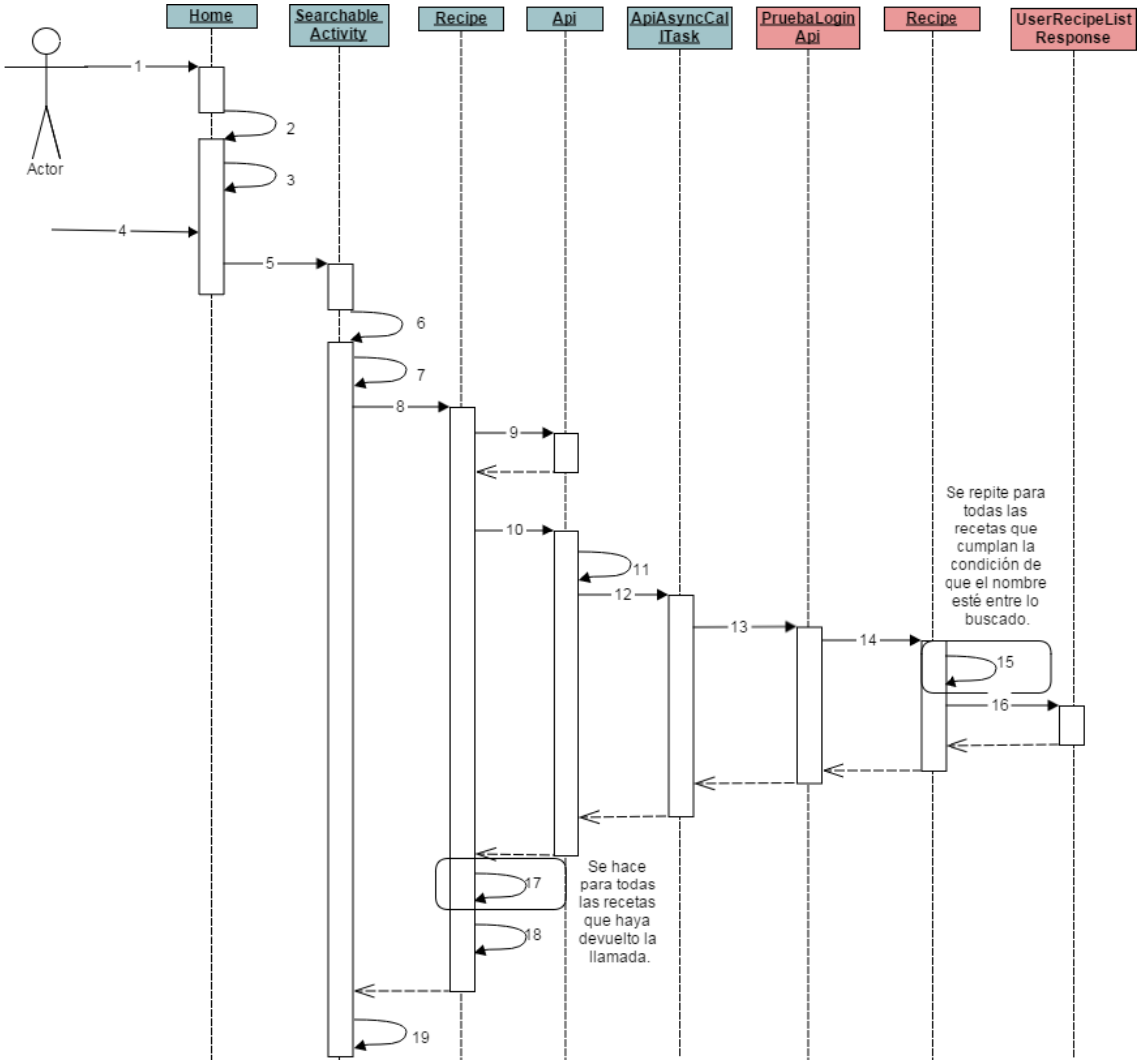


Ilustración 141: Diag. Sec. Buscar Recetas

1. El usuario accede a la pantalla Home.

2. onCreate(): void

3. onCreateOptionsMenu(): boolean

4. El usuario pincha en el icono de la lupa de la Action Bar, introduce la palabra que quiera busca y pincha el botón de búsqueda.

5. getSystemService(Context.SEARCH_SERVICE);

6. onCreate(): void

7. intent.getStringExtra(SearchManager.QUERY);

8. searchNameRecipe(query, new Callback<ArrayList<Recipe>>());

9. Api api = Api.getInstance();

10. api.service().recipe().searchNameRecipe(query, new Callback<CookingstardustMessagesUserRecipeListResponse>()

11. Cookingstardust.Recipe.SearchName call = request.searchName(query);

12. new ApiCallAsyncTask<CookingstardustMessagesUserRecipeListResponse>(call, callback).execute();

13. call.execute();

14. Recipe.search_rec_name(request)

15. recipe.to_message()

16. UserRecipeListResponse(recipes=rec)

17. Recipe.parse(item)

18. callback.onCompleted(null, response);

19. createListView(result);

12.Buscar receta modo avanzado

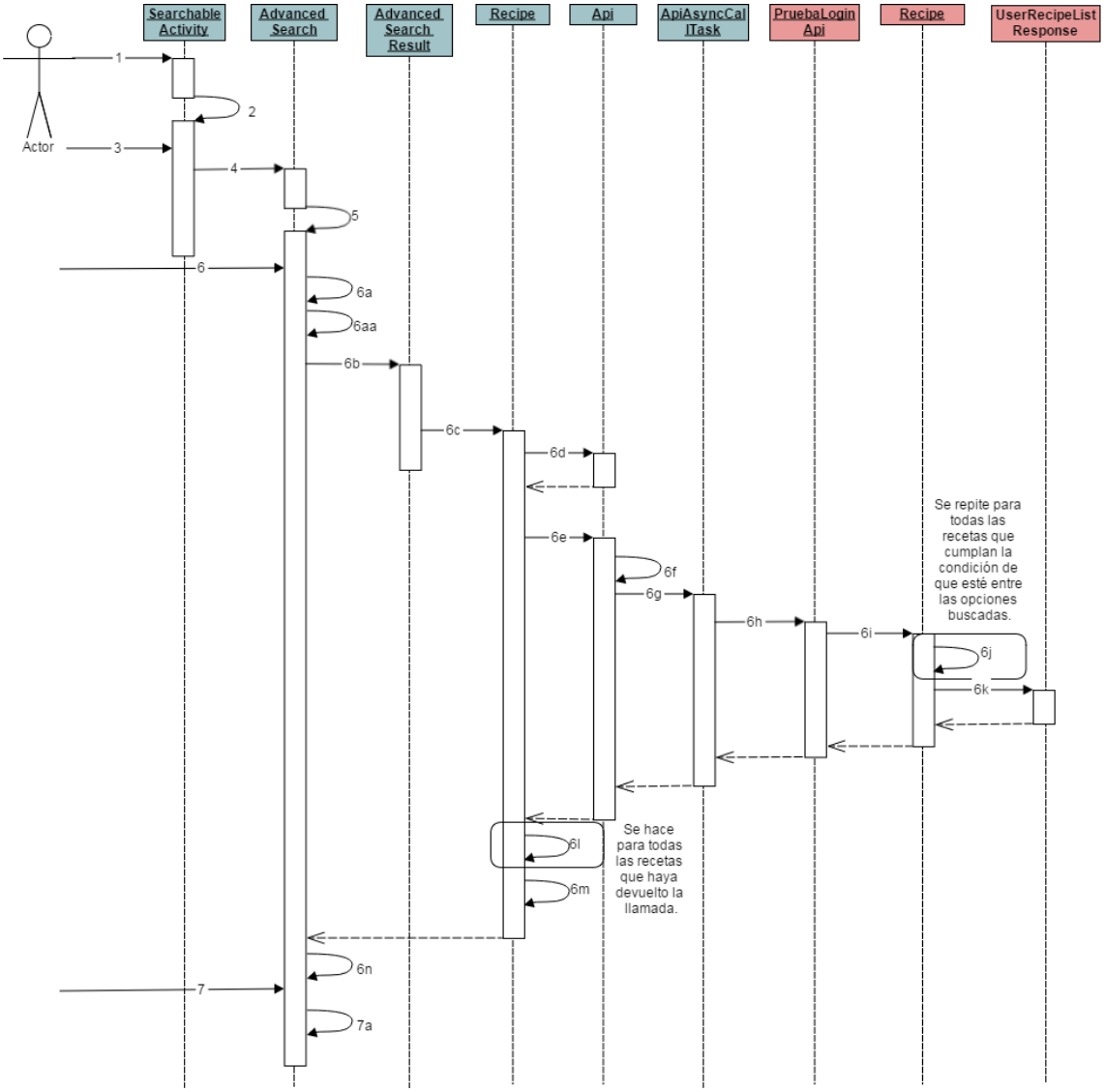


Ilustración 142: Diag. Sec. Buscar Recetas Avanzado

1. El usuario accede al listado de recetas que ha buscado desde el Action Bar.

2. onCreate(): void

3. El usuario pincha en el botón de una lupa con un "+". - Intent i = new Intent(this, AdvancedSearch.class);

4. startActivity(i);

5. onCreate(): void

6. El usuario pulsa "Buscar".

    6a. Si todos los campos están vacíos:

        6aa. toast("Introduce al menos un campo para buscar.");

    Sino:

    6b. startActivity(new Intent(this, AdvancedSearchResult.class));

    6c. searchAdvanceRecipe(query, new Callback<ArrayList<Recipe>>());

    6d. Api api = Api.getInstance();

    6e. api.service().recipe().searchAdvancedRecipe(query, new Callback<CookingstardustMessagesUserRecipeListResponse>()

    6f. Cookingstardust.Recipe.SearchVarious call = request.searchVarious(query);

    6g. new ApiCallAsyncTask<CookingstardustMessagesUserRecipeListResponse>(call, callback).execute();

    6h. call.execute();

    6i. Recipe.search_recipes(request)

    6j. recipe.to_message()

    6k. UserRecipeListResponse(recipes=rec)

    6j. Recipe.parse(item)

    6m. callback.onCompleted(null, response);

    6n. createListView(result);

7. El usuario pulsa "Cancelar".

    7a. this.finish();

13.Buscar receta por localización

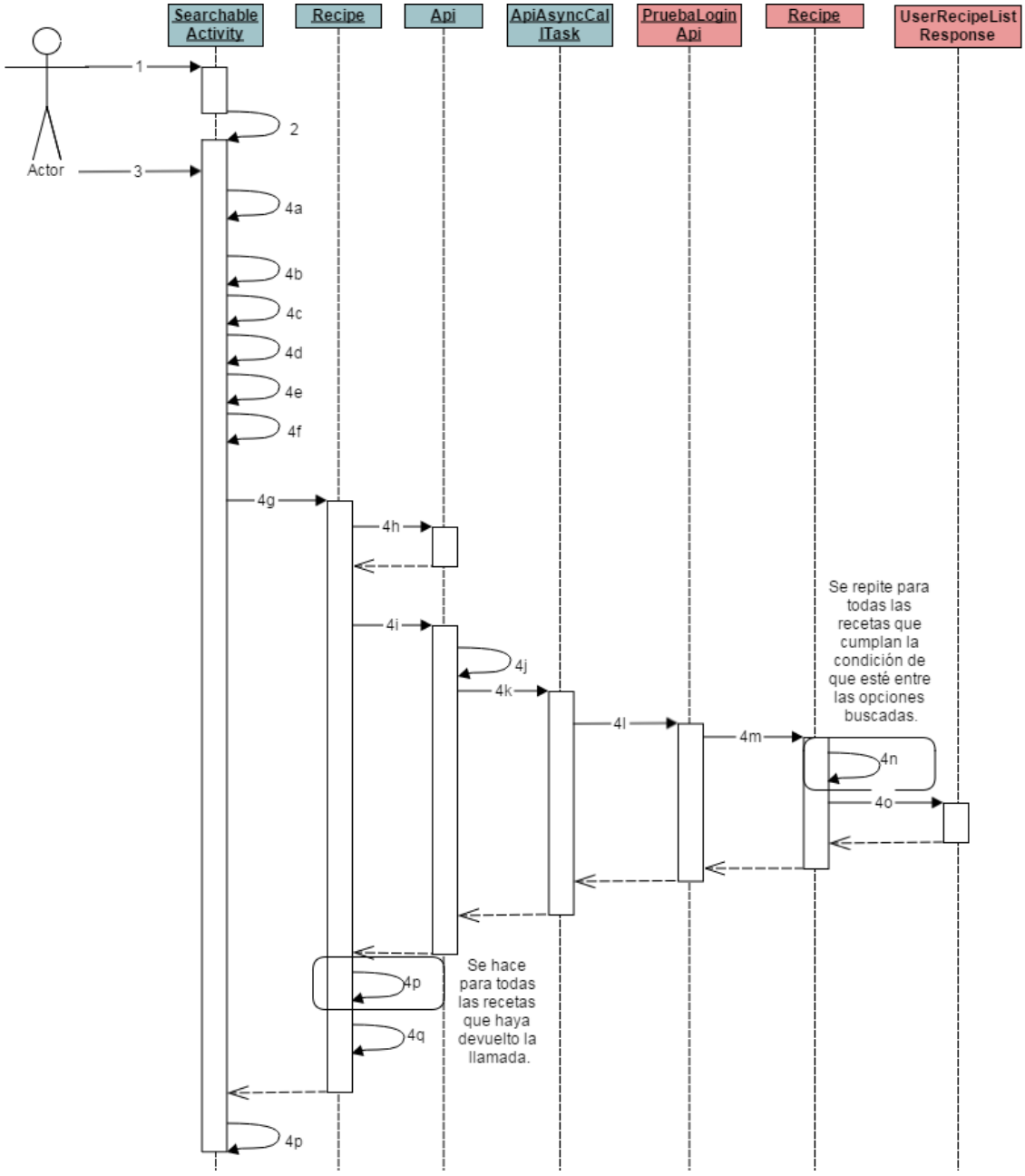


Ilustración 143: Diag. Sec. Buscar Recetas por Localización

1. El usuario accede al listado de recetas que ha buscado desde el Action Bar.

2. onCreate(): void

3. El usuario pincha en el botón "Localización".

4. Si la localización no está activada:

    4a. toast("Por favor, activa la localizacion en la pantalla de Configuracion.");

Sino:

    4b. locateCloseRecipes();

    4c. getSystemService(Context.LOCATION_SERVICE);

    4d. LocationListener locationListener = new LocationListener();

4e. latitude = Double.toString(location.getLatitude());

4f. longitude = Double.toString(location.getLongitude());

4g. findCloseRecipes(lat, lon, dist, new Callback<ArrayList<Recipe>>());

4h. Api api = Api.getInstance();

4i. api.service().recipe().getCloseRecipes(lat, lon, dist, new Callback<CookingstardustMessagesUserRecipeListResponse>());

4j. Cookingstardust.Recipe.Location.Close call = request.location().close(lon, lat, dist);

4k. new ApiCallAsyncTask<CookingstardustMessagesUserRecipeListResponse>(call, callback).execute();

4l. call.execute();

4m. Recipe.get_close_recipes(request)

4n. recipe.to_message()

4o. UserRecipeListResponse(recipes=rec)

4p. Recipe.parse(item)

4q. callback.onCompleted(null, response);

4p. createListView(result);

14.Ver Perfil

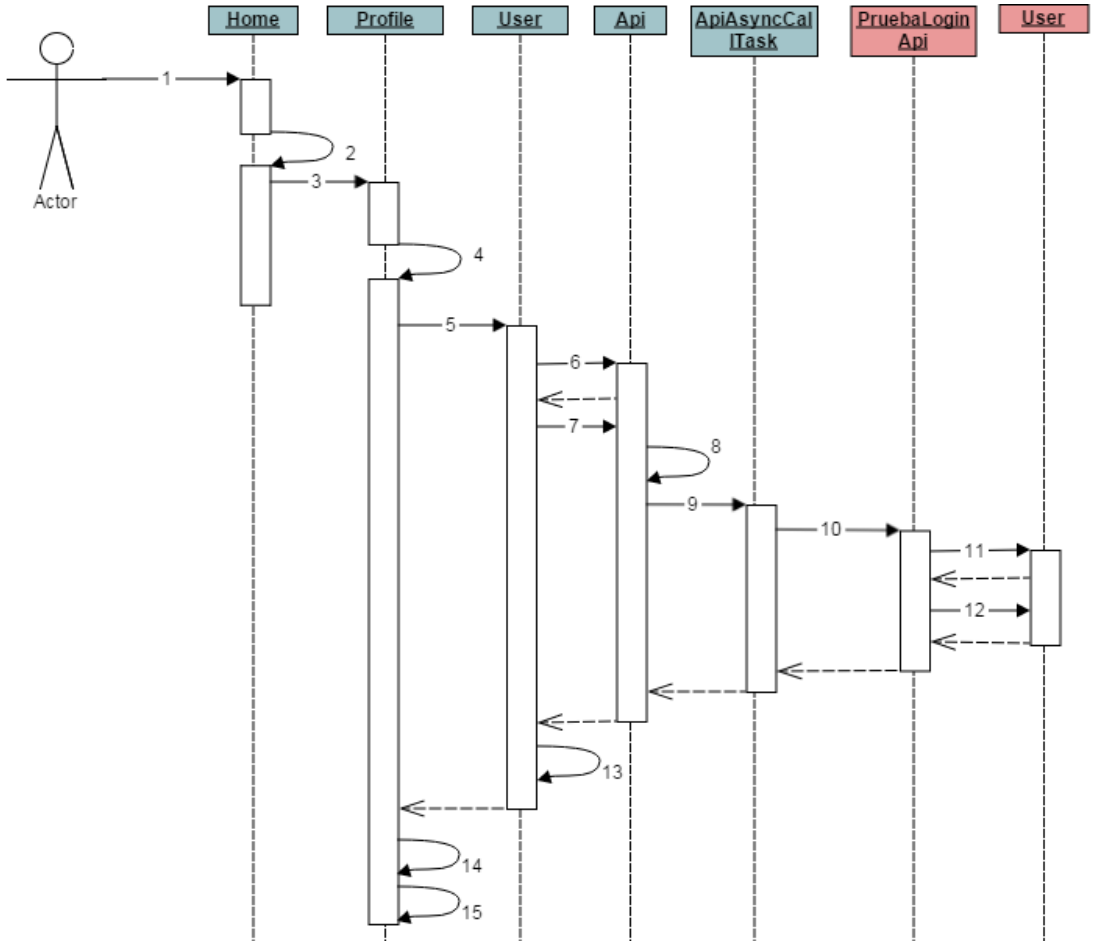


Ilustración 144: Diag. Sec. Ver Perfil

1. El usuario accede a la pantalla Home.

2. onCreate(): void

3. El usuario pincha en "Mi Perfil" en el menú deslizable de la izquierda. - startActivity(new Intent(this, Profile.class));

4. onCreate(): void

5. getUserInfo(lon, new Callback<User>());

6. Api api = Api.getInstance();

7. api.service().user().get_info(lon, new Callback<CookingstardustMessagesUserAllResponseMessage>());

8. Cookingstardust.User.GetUser call = request.getUser(lon);

9. new ApiCallAsyncTask<CookingstardustMessagesUserAllResponseMessage>(call, callback).execute();

10. call.execute();

11. User.get_user_info(request)

12. user.to_message()

13. callback.onCompleted(null, User.parse(result));

14. putInfo();

15. editProfileButton.setVisibility(View.VISIBLE);

15. Ver “Acerca De”

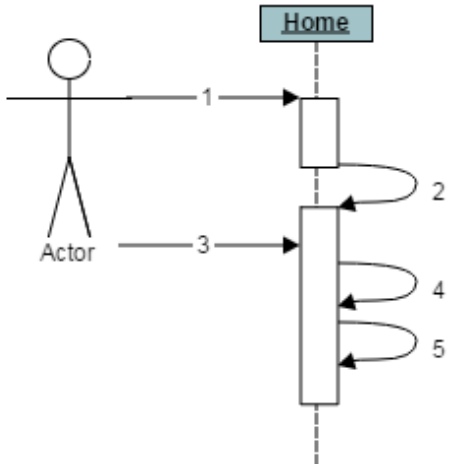


Ilustración 145: Diag. Sec. Ver Acerca De

1. El usuario accede a la pantalla Home.

2. onCreate(): void

3. El usuario pincha en "Acerca de" en el menú deslizable de la izquierda.

4. new AlertDialog.Builder(this).create();

5. alertDialog.show();

16.Borrar cuenta

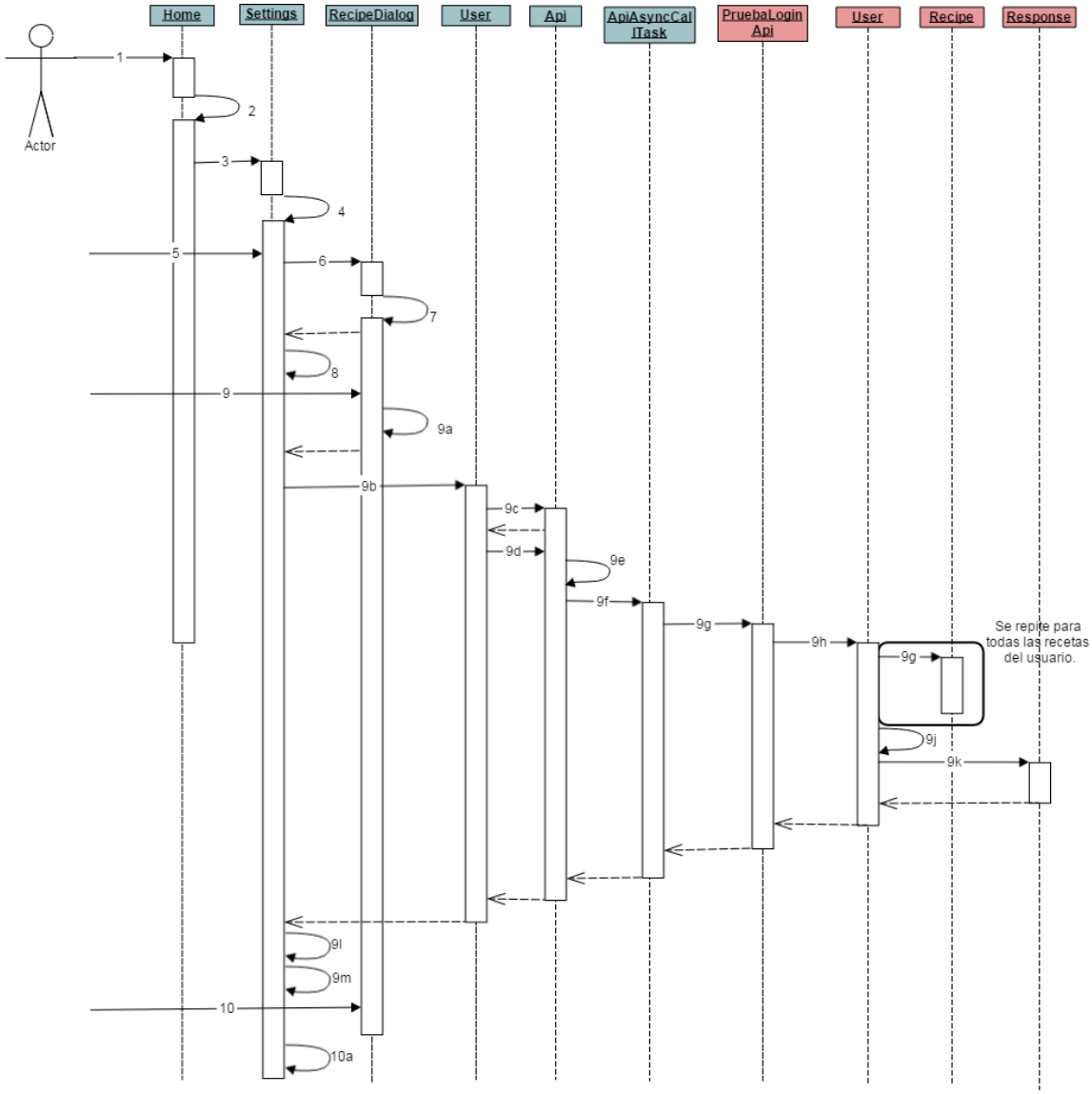


Ilustración 146: Diag. Sec. Borrar Cuenta

1. El usuario accede a la pantalla Home.
2. onCreate(): void
3. El usuario pincha en "Configuración" en el menú deslizable de la izquierda. - startActivity(new Intent(this, Configuration.class));
4. onCreate(): void
5. El usuario pincha en "Borrar Cuenta".
6. new RecipeDialog();
7. onCreateDialog(): Dialog
8. dialogFragment.show(getFragmentManager(), "recipe_dialog");
9. Si el usuario pincha "Aceptar":
    9a. activity.onSendDataFragment("borrar", 1, 0);

9b. remove_account(lon, new Callback<Boolean>()

9c. Api api = Api.getInstance();

9d. api.service().user().remove_user_forever(lon, new Callback<CookingstardustMessagesResponse>();

9e. Cookingstardust.User.DeleteForever call = request.deleteForever(lon);

9f. new ApiCallAsyncTask<CookingstardustMessagesResponse>(call, callback).execute();

9g. call.execute();

9h. User.delete_user_forever(request)

9i. recipe.delete()

9j. user.key.delete()

9k. Response(there=True)

9l. toast("Usuario borrado perfectamente!");

9m. this.finish()

10. Si el usuario pincha "Cancelar":

10a. this.finish();

17. Activar localización

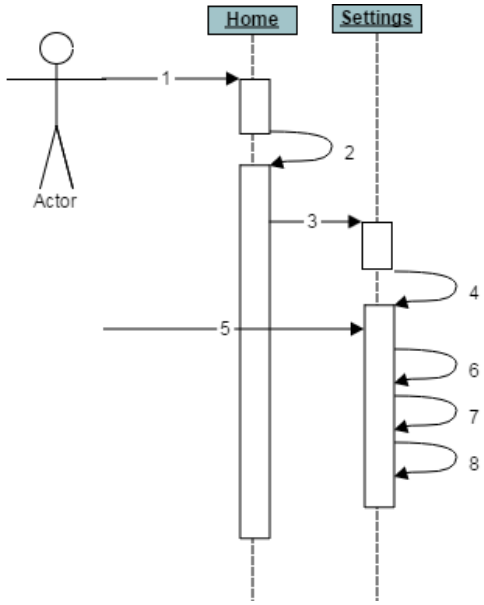


Ilustración 147: Diag. Sec. Activar Localización

1. El usuario accede a la pantalla Home.

2. onCreate(): void

3. El usuario pincha en "Configuración" en el menú deslizable de la izquierda. - startActivity(new Intent(this, Configuration.class));

4. onCreate(): void

5. El usuario pincha en el switch, para activar la localización.
6. SharedPreferences locationEnabled = getSharedPreferences(PREFS_NAME, 0);
7. SharedPreferences.Editor editor = locationEnabled.edit();
8. editor.putBoolean("location", true).commit();

18.Modificar Perfil

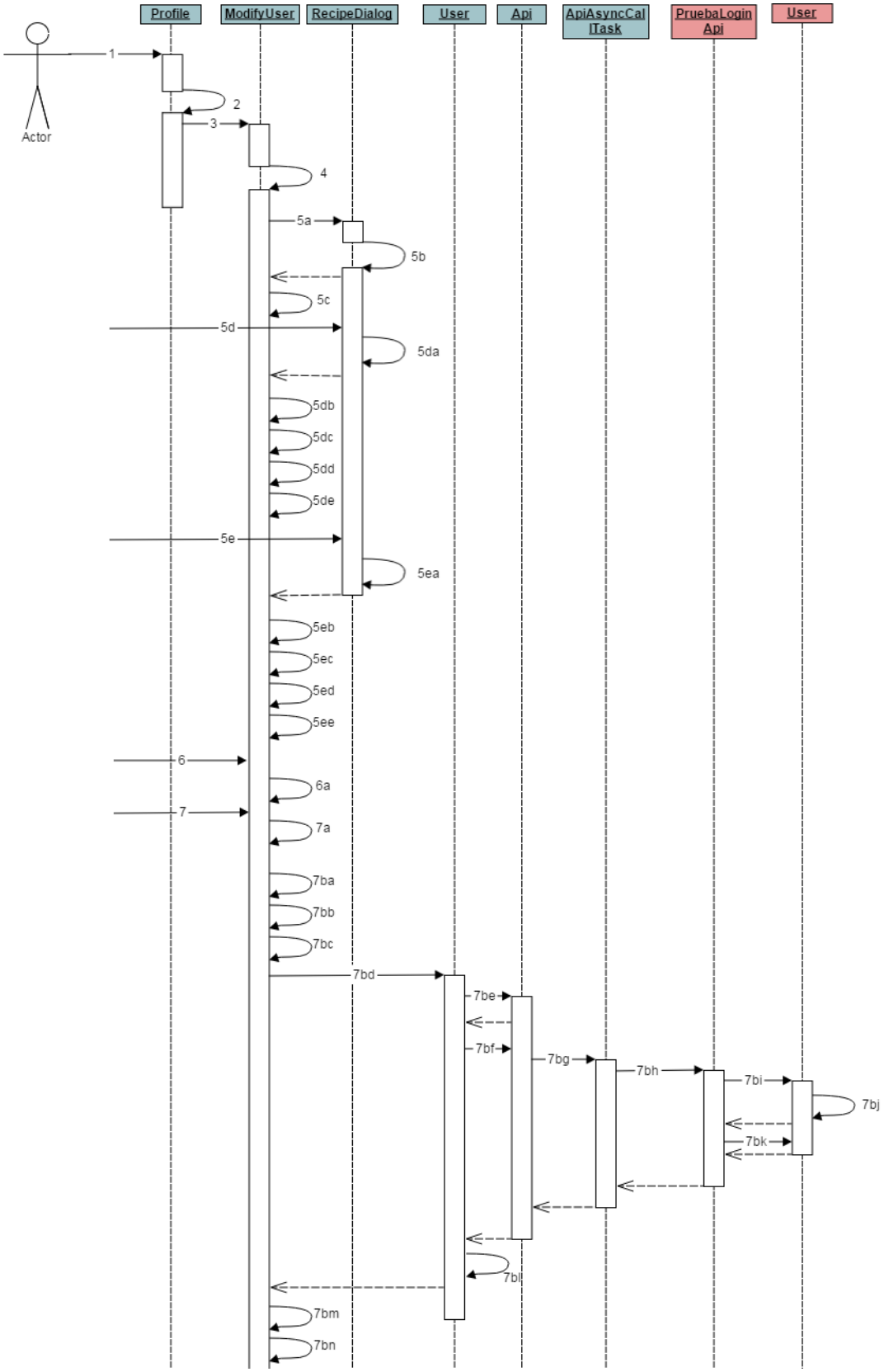


Ilustración 148: Diag. Sec. Modificar Perfil 1

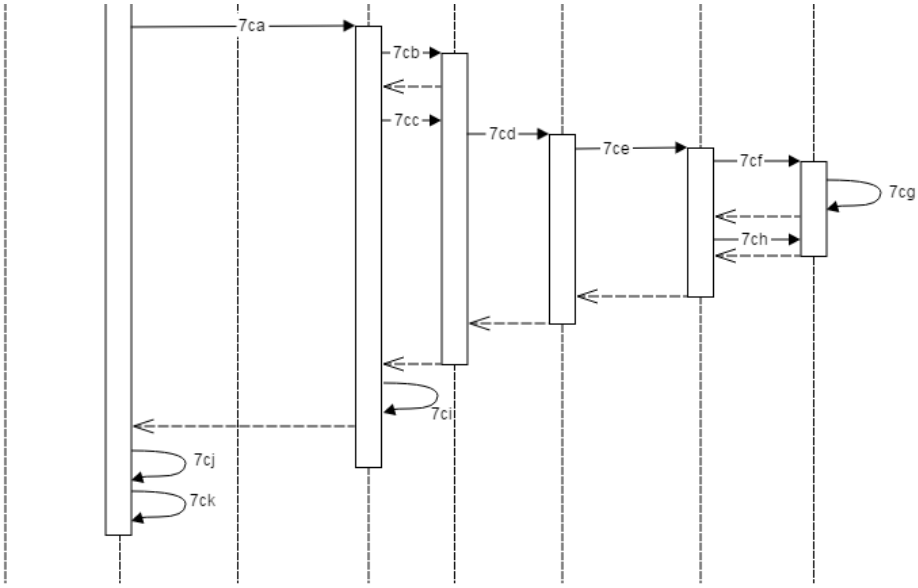


Ilustración 149: Diag. Sec. Modificar Perfil 2

1. El usuario accede a la pantalla Perfil.
2. onCreate(): void
3. El usuario pincha en "Modificar Perfil". - startActivity(new Intent(this, ModifyUser.class));
4. onCreate(): void
5. Si el usuario pincha en "Añadir Foto":
    5a. new RecipeDialog();
    5b. onCreateDialog(): Dialog
    5c. dialogFragment.show(getFragmentManager(), "recipe_dialog");
    5d. Si el usuario ha pinchado en "Galería":
        5da. activity.onSendDataFragment(tipo, 3, 0);
        5db. onSendDataFragment(String data, int pos, int number);
        5dc. new Intent(Intent.ACTION_PICK, android.provider.MediaStore.Images.Media.EXTERNAL_CONTENT_URI);
        5dd. startActivityForResult(i, RESULT_LOAD_IMAGE);
        5de. recipePhoto.setImageBitmap(BitmapFactory.decodeFile(picturePath));
    5e. Sino, si ha pinchado en "Cámara":
        5ea. activity.onSendDataFragment(tipo, 3, 0);
        5eb. onSendDataFragment(String data, int pos, int number);
        5ec. new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
        5ed. startActivityForResult(takePictureIntent, RESULT_LOAD_CAMERA);
        5ee. recipePhoto.setImageBitmap(imageBitmap);
6. Si el usuario pincha en "Cancelar":
    6a. this.finish();
7. Si el usuario pincha en "Guardar". Si el nickname, nombre o contraseña están vacíos:

7a. toast("The name, ingredients, elaboration, time or people fields cannot be empty!");

Sino:

7b. Si el usuario ha escogido una foto:

7ba. Ion.with(this).load(URL).asJsonObject().setCallback(new FutureCallback<JsonObject>(); //nos devuelve la url donde se guardará la foto

7bb. UrlMessage(url=url)

7bc. Ion.with(getApplicationContext()).load(finalUrl).setMultipartParameter("file", "image/jpeg").setMultipartFile("file", new File(picturePath)).asJsonObject().setCallback(new FutureCallback<JsonObject>();

7bd. User.modify_profile(nombre, nick, pass, email, age, id, sex, location, urlKey, new Callback<User>();

7be. Api api = Api.getInstance();

7bf. api.service().user().modify(user, new Callback<CookingstardustMessagesUserAllResponseMessage>());

7bg. new ApiCallAsyncTask<CookingstardustMessagesUserAllResponseMessage>(call, callback).execute();

7bh. call.execute();

7bi. User.modify_user(request)

7bj. user.put()

7bk. user.to_message()

7bl. callback.onCompleted(null, User.parse(result));

7bm. toast("Usuario " + result.getPhoto() + " modificado correctamente.");

7bn. this.finish();

7c. Si el usuario no ha escogido ninguna foto:

7ca. User.modify_profile(nombre, nick, pass, email, age, id, sex, location, urlKey, new Callback<User>();

7cb. Api api = Api.getInstance();

7cc. api.service().user().modify(user, new Callback<CookingstardustMessagesUserAllResponseMessage>());

7cd. new ApiCallAsyncTask<CookingstardustMessagesUserAllResponseMessage>(call, callback).execute();

7ce. call.execute();

7cf. User.modify_user(request)

7cg. user.put()

7ch. user.to_message()

7ci. callback.onCompleted(null, User.parse(result));

7cj. toast("Usuario " + result.getPhoto() + " modificado correctamente.");

7ck. this.finish();

19. Ver listado de amigos

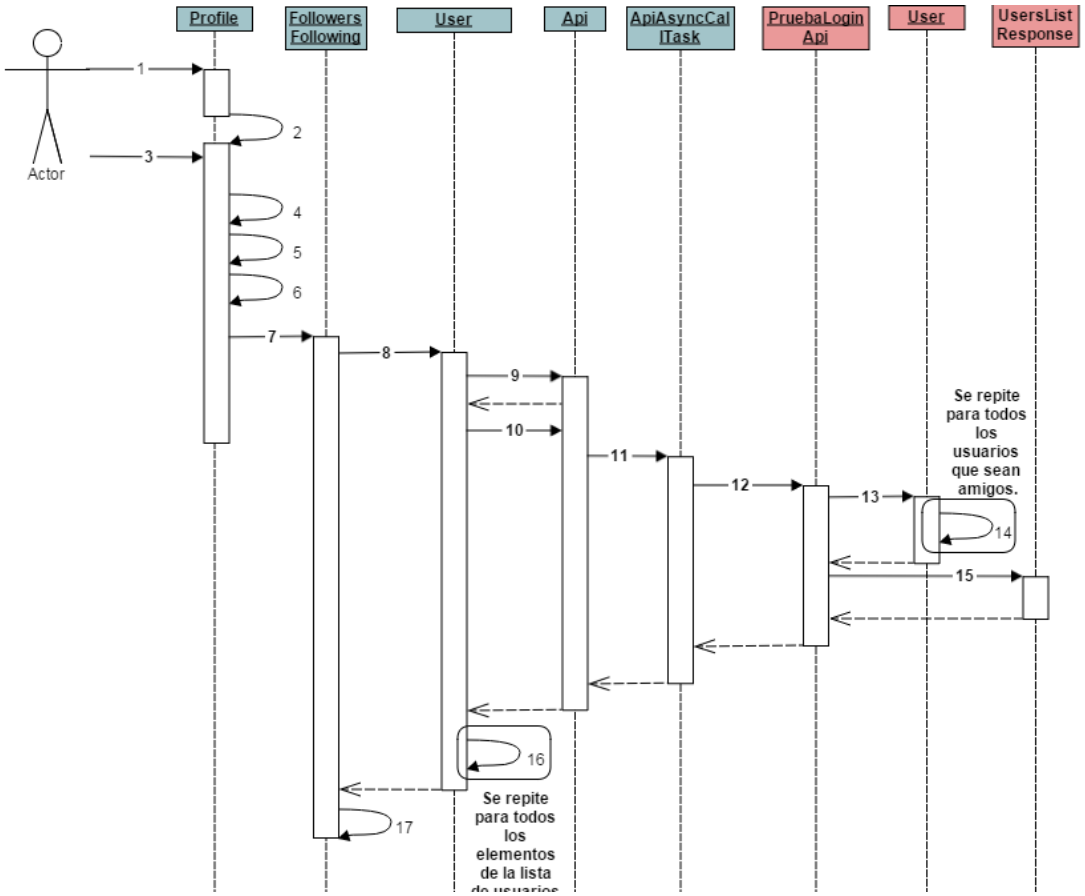


Ilustración 150: Diag. Sec. Ver Listado de Amigos

1. El usuario accede a la pantalla Perfil.
2. onCreate(): void
3. El usuario pincha en "Siguiendo".
4. FragmentManager fm = getFragmentManager();
5. FragmentTransaction f = fm.beginTransaction();
6. extras.putInt("opcion", 0);
7. f.replace(R.id.blank, new FollowersFollowing());
8. User.get_followers(id, new Callback<ArrayList<User>>());
9. Api api = Api.getInstance();
10. api.service().user().get_following(id, new Callback<CookingstardustMessagesUsersListResponse>());

11. new ApiCallAsyncTask<CookingstardustMessagesUsersListResponse>(call, callback).execute();

12. call.execute();

13. User.get_following(request)

14. users.append(us.to_message()) - bucle

15. UsersListResponse(users=users)

16. response.add(User.parse(item)); - bucle

17. callback.onCompleted(null, response);

18. createListView(result);

20.Buscar amigos

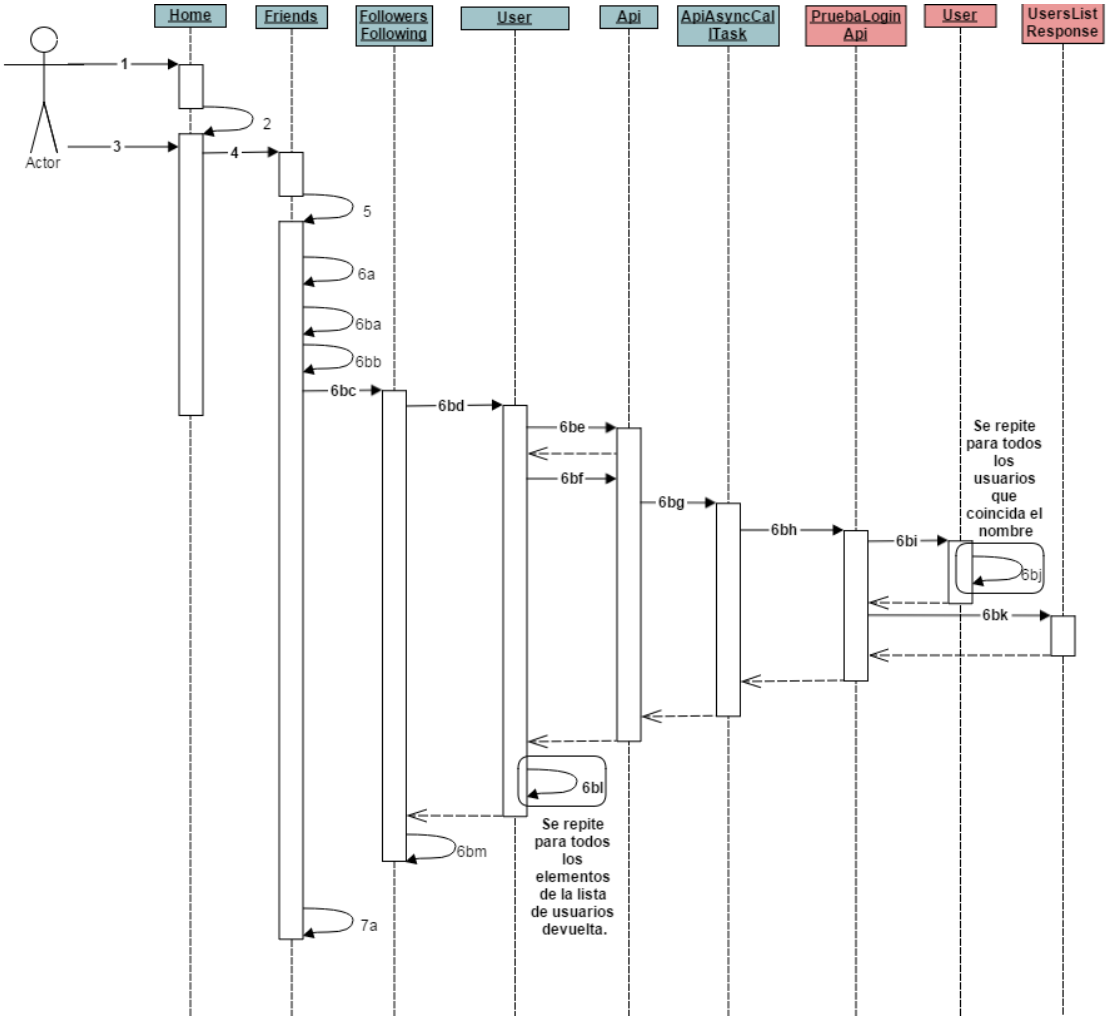


Ilustración 151: Diag. Sec. Buscar Amigos

1. El usuario accede a la pantalla principal, Home.

2. onCreate(): void

3. El usuario pincha la opción "Buscar Amigos", en el menú deslizable de la izquierda.

4. startActivity(this, Friends.class);

5. onCreate(): void

6. El usuario pincha "Guardar". Si el campo está vacío:

6a. toast("Introduce un nombre o nickname!");

6b. Sino:

6ba. FragmentTransaction f = fm.beginTransaction();

6bb. extras.putInt("opcion", 2);

6bc. f.replace(R.id.blank, new FollowersFollowing());

6bd. User.search_friends(busqueda, new Callback<ArrayList<User>>());

6be. Api api = Api.getInstance();

6bf. api.service().user().search_friends(busqueda, new Callback<CookingstardustMessagesUsersListResponse>());

6bg. new ApiCallAsyncTask<CookingstardustMessagesUsersListResponse>(call, callback).execute();

6bh. call.execute();

6bi. User.search_friends(request)

6bj. friends.append(u.to_message())

6bk. UsersListResponse(users=friends)

6bl. response.add(User.parse(item));

6bm. createListView(result);

7. El usuario pincha "Cancelar":

7a. this.finish();

21.Guardar Receta

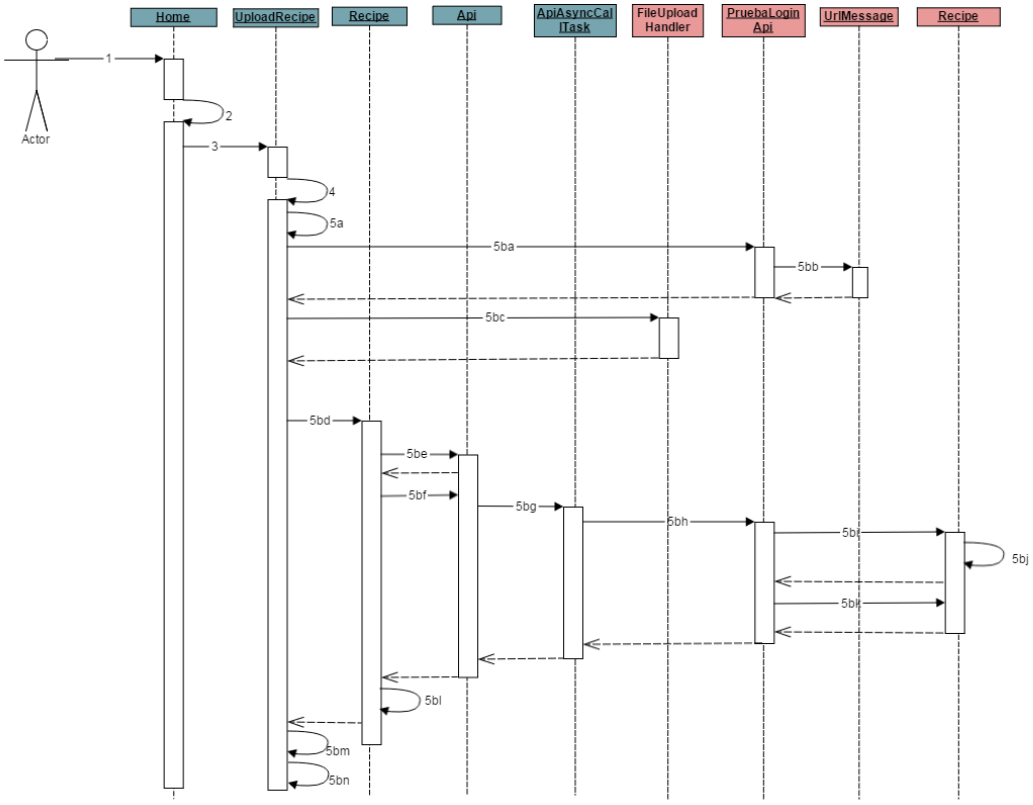


Ilustración 152: Diag. Sec. Guardar Receta 1

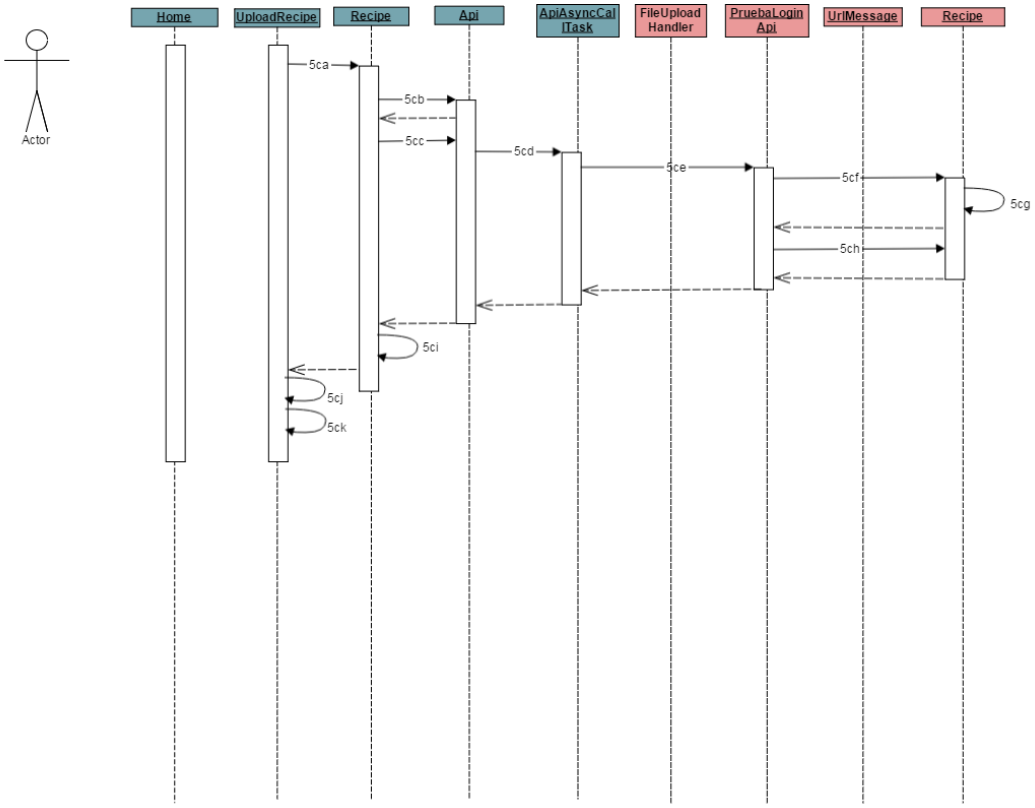


Ilustración 153: Diag. Sec. Guardar Receta 2

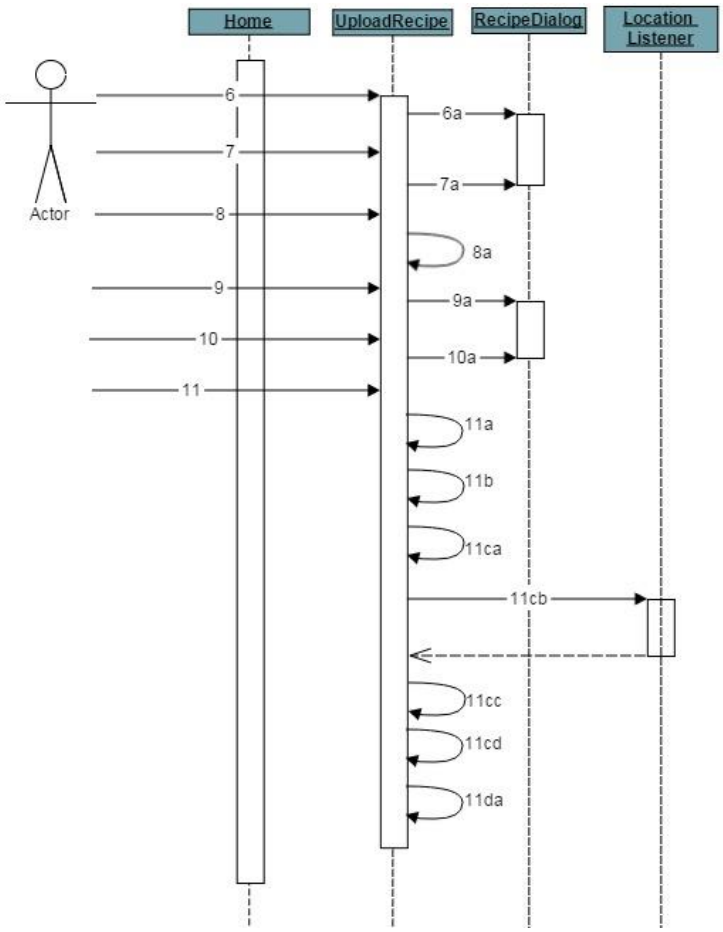


Ilustración 154: Diag. Sec. Guardar Receta 3

1. El usuario accede a la pantalla Home.

2. onCreate(): void

3. El usuario pincha en "Guardar una receta" en el menú deslizable de la izquierda. - new Intent(this, UploadRecipe.class);

4. onCreate(): void

5. El usuario pincha "Guardar". Si el nombre, la duración, el número de comensales, los ingredientes o la elaboración están vacíos:

5a. toast("The name, ingredients, elaboration, time or people fields cannot be empty!");

Sino:

5b. Si el usuario ha escogido una foto:

5ba. Ion.with(this).load(URL).asJsonObject().setCallback(new FutureCallback<JsonObject>(); //nos devuelve la url donde se guardará la foto

5bb. UrlMessage(url=url)

5bc. Ion.with(getApplicationContext()).load(finalUrl).setMultipartParameter("file", "image/jpeg").setMultipartFile("file", new File(picturePath)).asJsonObject().setCallback(new FutureCallback<JsonObject>();

5bd. Recipe.saveRecipe(titleView, ingredients, elaboracion, dietView, cuisineView, difficultyView, time_final, people_final, platoView, lon, urlKey, location, latitude, longitude, new Callback<Recipe>();

5be. Api api = Api.getInstance();

5bf. api.service().recipe().post(recipe, new Callback<CookingstardustMessagesRecipeResponse>();

5bg. new ApiCallAsyncTask<CookingstardustMessagesRecipeResponse>(call, callback).execute();

5bh. call.execute();

5bi. Recipe.put_from_message(request)

5bj. recipe.put()

5bk. recipe.to_message()

5bl. callback.onCompleted(null, Recipe.parse(result));

5bm. toast("Receta " + result.getTitle() + "guardada exitosamente!");

5bn. this.finish();

5c. Si el usuario no ha escogido ninguna foto:

5ca. Recipe.saveRecipe(titleView, ingredients, elaboracion, dietView, cuisineView, difficultyView, time_final, people_final, platoView, lon, urlKey, location, latitude, longitude, new Callback<Recipe>();

5cb. Api api = Api.getInstance();

5cc. api.service().recipe().post(recipe, new Callback<CookingstardustMessagesRecipeResponse>();

5cd. new ApiCallAsyncTask<CookingstardustMessagesRecipeResponse>(call, callback).execute();

5ce. call.execute();

5cf. Recipe.put_from_message(request)

5cg. recipe.put()

5ch. recipe.to_message()

5ci. callback.onCompleted(null, Recipe.parse(result));

5cj. toast("Receta " + result.getTitle() + "guardada exitosamente!");

5ck. this.finish();

6. El usuario pincha en "Añadir ingrediente":

6a. EXTEND CU "Añadir Ingrediente"

7. El usuario pincha en "Añadir paso":

7a. EXTEND CU "Añadir Paso"

8. El usuario pincha en "Cancelar"

8a. this.finish();

9. El usuario pincha en "Añadir foto"

9a. EXTEND CU "Añadir foto de receta"

10. El usuario pincha en "Añadir foto por paso"

10a. EXTEND CU "Añadir foto de receta por paso"

11. El usuario pulsa en "Guardar Localización"

11a. SharedPreferences location = getSharedPreferences(PREFS_NAME, 0);

11b. location.getBoolean("location", false);

11c. Si la localización está activada:

11ca. getSystemService(Context.LOCATION_SERVICE);

11cb. LocationListener locationListener = new LocationListener();

11cc. latitude = Double.toString(location.getLatitude());

11cd. longitude = Double.toString(location.getLongitude());

11d. Si no lo está:

11da. toast("Por favor, activa la localizacion en la pantalla de Configuracion.");

22.Guardar Ingrediente

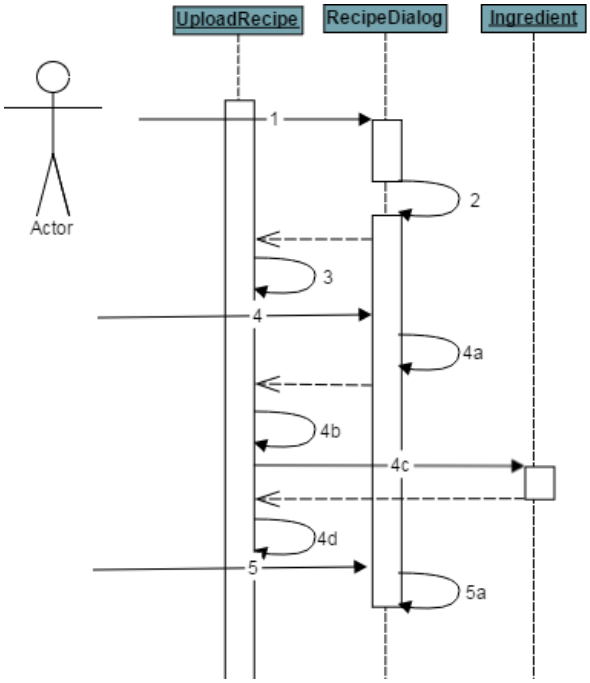


Ilustración 155: Diag. Sec. Guardar Ingrediente

1. new RecipeDialog();

2. onCreateDialog(): Dialog

3. dialogFragment.show(getFragmentManager(), "recipe_dialog");

4 Si el usuario ha pinchado en aceptar:

4a. activity.onSendDataFragment(nombre, 1, Integer.valueOf(cantidad));

4b. onSendDataFragment(String data, int pos, int number)

4c. Ingredient i = new Ingredient();

4d. ingredient.setText(ings);

Sino, si pulsa cancelar:

5a. RecipeDialog.this.getDialog().cancel();

23.Guardar Paso

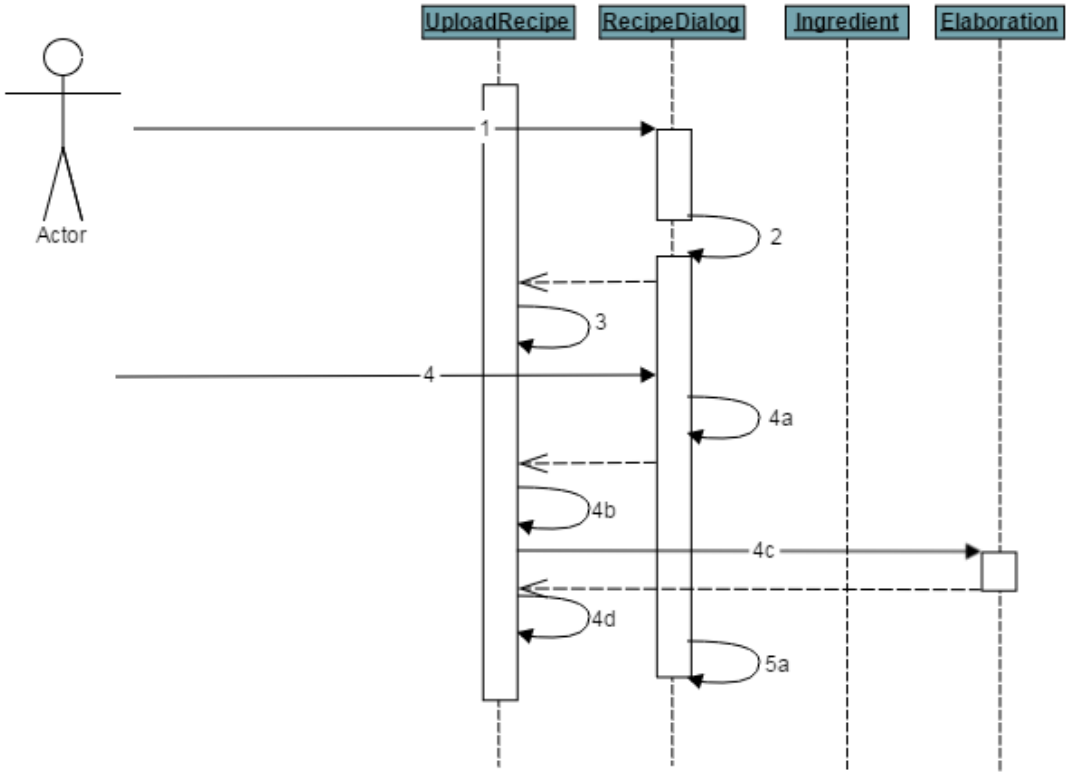


Ilustración 156: Diag. Sec. Guardar Paso

1. new RecipeDialog();
2. onCreateDialog(): Dialog
3. dialogFragment.show(getFragmentManager(), "recipe_dialog");
4. Si el usuario ha pinchado en aceptar:

    4a. activity.onSendDataFragment(nombre, 2, 0);

    4b. onSendDataFragment(String data, int pos, int number);

    4c. Elaboration p = new Elaboration();

    4d. elaboration.setText(pasos);

Sino, si pulsa cancelar:

    5a. RecipeDialog.this.getDialog().cancel();

24. Añadir Foto

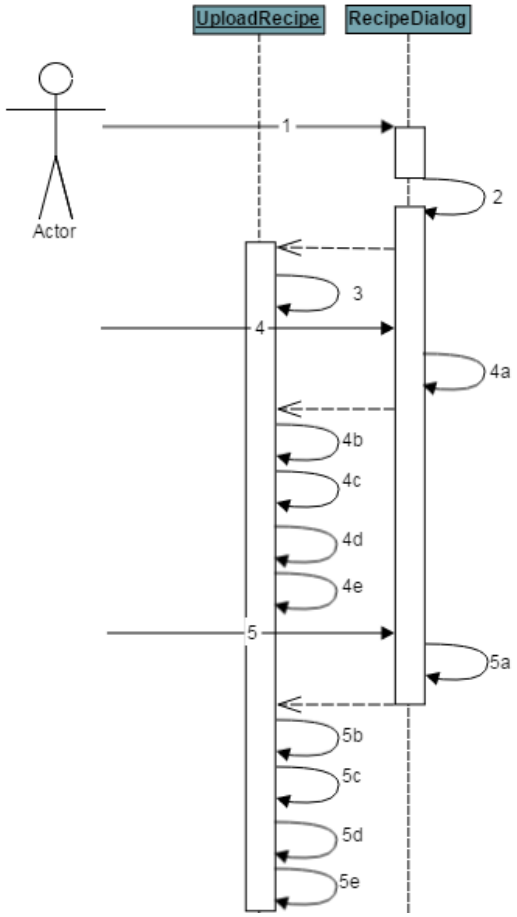


Ilustración 157: Diag. Sec. Añadir Foto

1. new RecipeDialog();
2. onCreateDialog(): Dialog
3. dialogFragment.show(getFragmentManager(), "recipe_dialog");
4. Si el usuario ha pinchado en "Galería":
    - 4a. activity.onSendDataFragment(tipo, 3, 0);
    - 4b. onSendDataFragment(String data, int pos, int number);
    - 4c. new Intent(Intent.ACTION_PICK, android.provider.MediaStore.Images.Media.EXTERNAL_CONTENT_URI);
    - 4d. startActivityForResult(i, RESULT_LOAD_IMAGE);
    - 4e. recipePhoto.setImageBitmap(BitmapFactory.decodeFile(picturePath));
5. Sino, si ha pinchado en "Cámara":
    - 5a. activity.onSendDataFragment(tipo, 3, 0);
    - 5b. onSendDataFragment(String data, int pos, int number);
    - 5c. new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
    - 5d. startActivityForResult(takePictureIntent, RESULT_LOAD_CAMERA);
    - 5e. recipePhoto.setImageBitmap(imageBitmap);

25. Añadir foto por paso

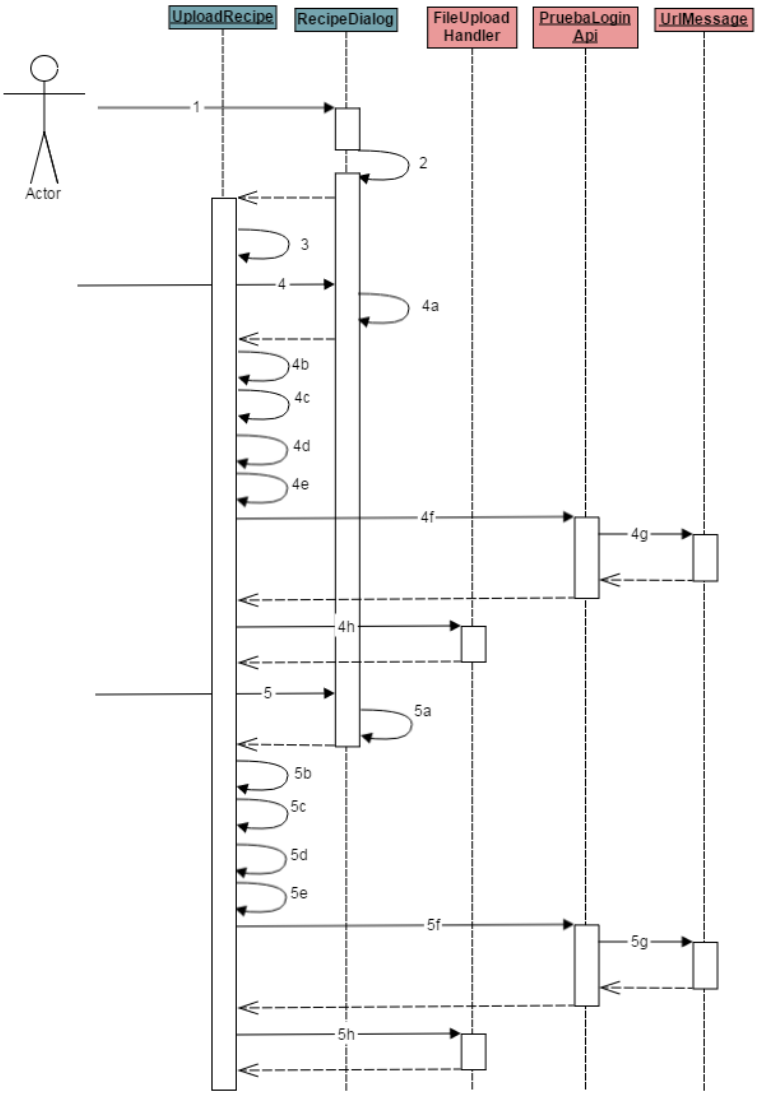


Ilustración 158: Diag. Sec. Añadir Foto por Paso

1. new RecipeDialog();

2. onCreateDialog(): Dialog

3 dialogFragment.show(getFragmentManager(), "recipe_dialog");

4. Si el usuario ha pinchado en "Galería":

4a. activity.onSendDataFragment(tipo, 4, 0);

4b. onSendDataFragment(String data, int pos, int number);

4c. new Intent(Intent.ACTION_PICK, android.provider.MediaStore.Images.Media.EXTERNAL_CONTENT_URI);

4d. startActivityForResult(i, RESULT_LOAD_IMAGE_STEP);

4e. recipePhoto.setImageBitmap(BitmapFactory.decodeFile(picturePath));

4f. Ion.with(this).load(URL).asJsonObject().setCallback(new FutureCallback<JsonObject>();

4g. UrlMessage(url=url)

4h. Ion.with(getApplicationContext()).load(finalUrl).setMultipartParameter("file", "image/jpeg").setMultipartFile("file", new File(picture)).asJsonObject().setCallback(new FutureCallback<JsonObject>();

5. Si el usuario pincha en "Cámara":

5a. activity.onSendDataFragment(tipo, 4, 0);

5b. onSendDataFragment(String data, int pos, int number);

5c. new Intent(MediaStore.ACTION_IMAGE_CAPTURE);

5d. startActivityForResult(takePictureIntent, RESULT_LOAD_CAMERA);

5e. recipePhoto.setImageBitmap(imageBitmap);

5f. Ion.with(this).load(URL).asJsonObject().setCallback(new FutureCallback<JsonObject>();

5g. UrlMessage(url=url)

5h. Ion.with(getApplicationContext()).load(finalUrl).setMultipartParameter("file", "image/jpeg").setMultipartFile("file", new File(picture)).asJsonObject().setCallback(new FutureCallback<JsonObject>();

26.Añadir Amigo

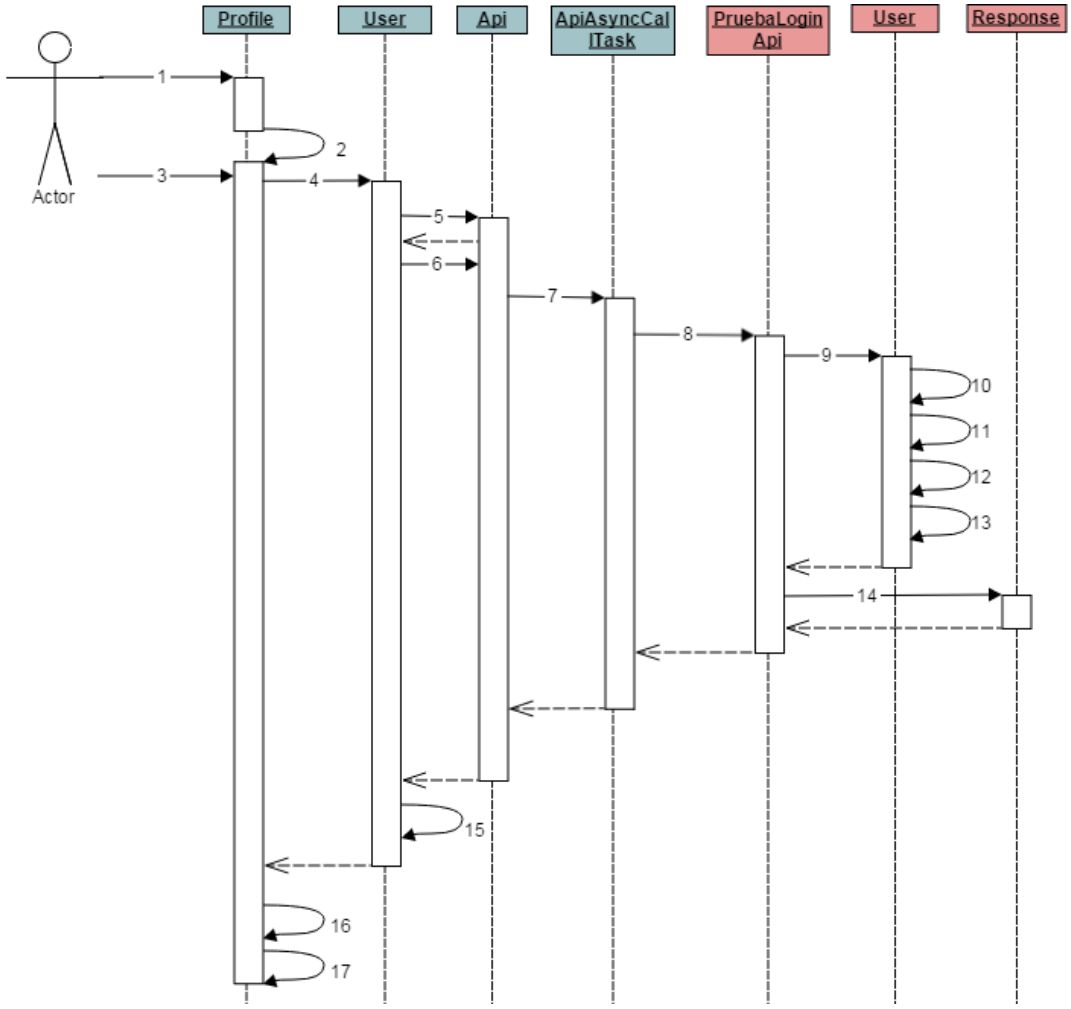


Ilustración 159: Diag. Sec. Añadir Amigo

1. El usuario accede al perfil del usuario que quiere agregar como amigo, Profile.class.
2. onCreate(): void
3. El usuario pincha en el botón "Seguir".
4. User.add_following(user.getId(), userid, new Callback<Boolean>();
5. Api api = Api.getInstance();
6. api.service().user().add_following(request, new Callback<CookingstardustMessagesResponse>();
7. new ApiCallAsyncTask<CookingstardustMessagesResponse>(call, callback).execute();
8. call.execute();
9. User.add_user_following(request)
10. me.following.append(follow.key)
11. me.put()
12. follow.followers.append(me.key)
13. follow.put()

14. Response(there=True)

15. callback.onCompleted(null, resultado);

16. toast("User añadido a tus following!");

17. this.finish();

27.Quitar amigo

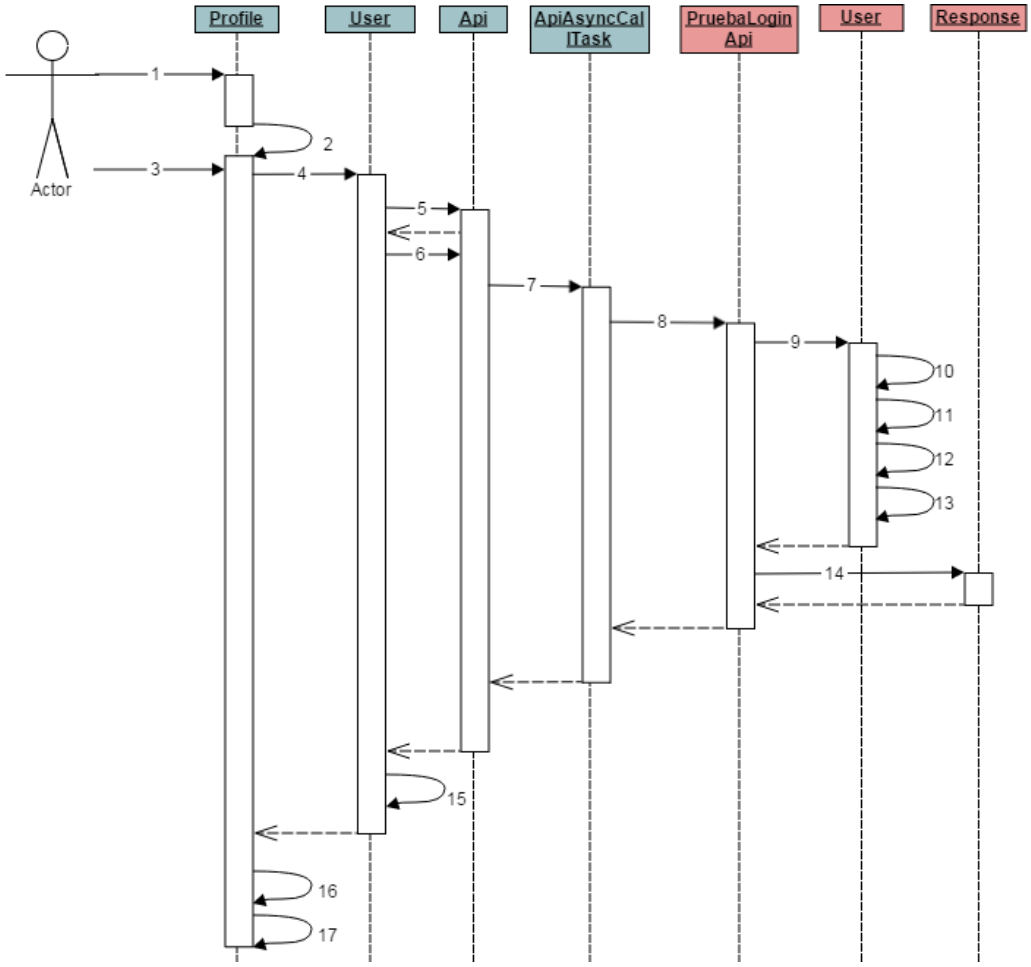


Ilustración 160: Diag. Sec. Quitar Amigo

1. El usuario accede al perfil del usuario que quiere quitar como amigo, Profile.class.

2. onCreate(): void

3. El usuario pincha en el botón "Dejar de seguir".

4. User.delete_following(user.getId(), userid, new Callback<Boolean>();

5. Api api = Api.getInstance();

6. api.service().user().delete_following(request, new Callback<CookingstardustMessagesResponse>();

7. new ApiCallAsyncTask<CookingstardustMessagesResponse>(call, callback).execute();

8. call.execute();

9. User.delete_user_following(request)

10. me.following.remove(unfollow.key)
11. me.put()
12. unfollow.followers_id.remove(user)
13. unfollow.put()
14. Response(there=True)
15. callback.onCompleted(null, resultado);
16. toast("User quitado de tus following!");
17. this.finish();

28. Ver listado de recetas favoritas

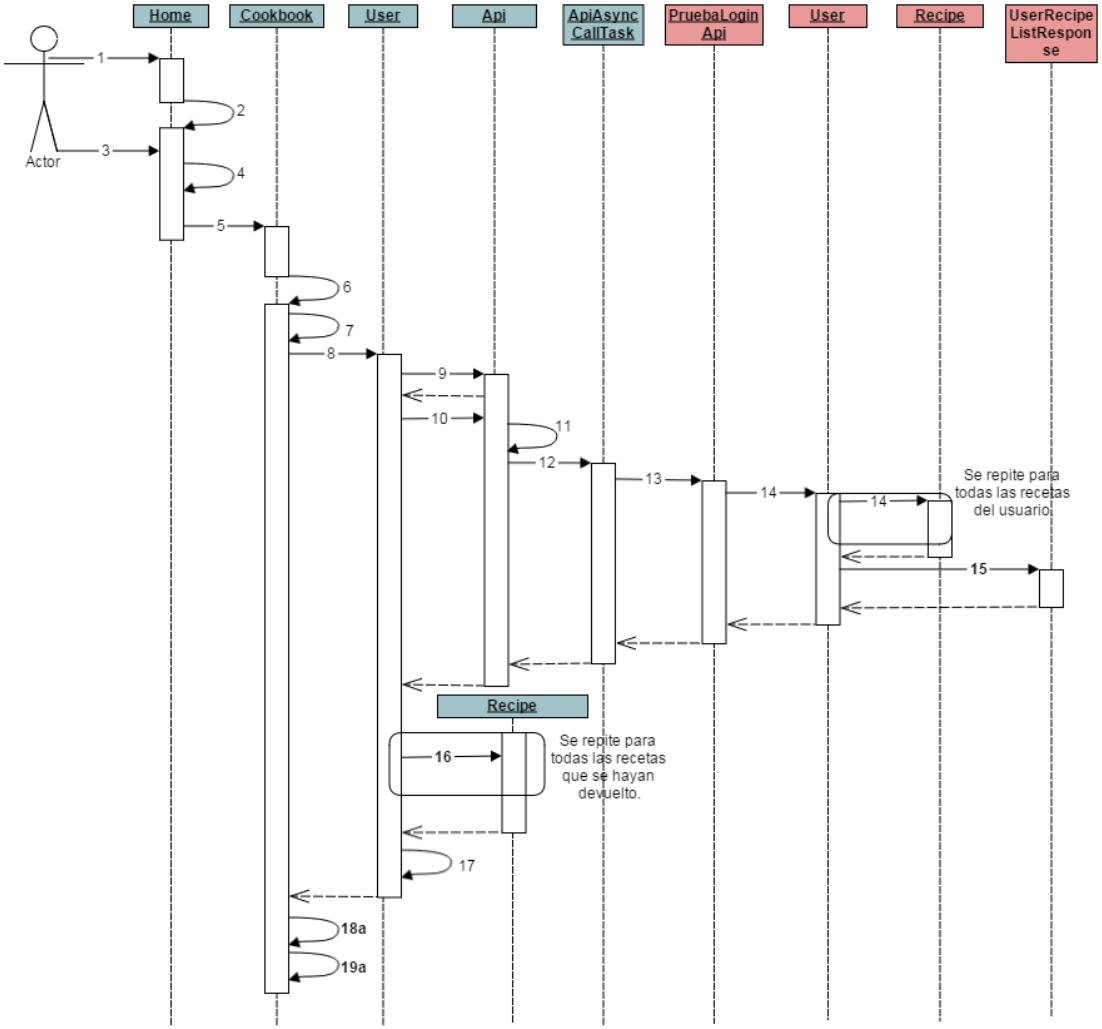


Ilustración 161: Diag. Sec. Ver Favoritas

1. El usuario accede a la pantalla Home.
2. onCreate(): void
3. El usuario pincha en "Favoritas" en el menú deslizable de la izquierda.
4. Bundle extras = new Bundle().putExtra(option, 1);
5. startActivity(AdvancedSearchResult.class);

6. onCreate(): void

7. Long lon = getActivity().getSharedPreferences(PREFS_NAME, 0).getLong("userID", 4L);

8. list_favorites(lon, new Callback<ArrayList<Recipe>>();

9. Api api = Api.getInstance();

10. api.service().recipe()list_favorites(lon, new Callback<CookingstardustMessagesUserRecipeListResponse>();

11. Cookingstardust.Recipe.ListFavorites call = request.listFavorites(lon);

12. new ApiCallAsyncTask<CookingstardustMessagesUserRecipeListResponse>(call, callback).execute();

13. call.execute();

14. User.get_user_favorites(request)

15. Recipe.to_message()

16. Recipe.parse(item)

17. callback.onCompleted(null, response);

18. Si el usuario tiene recetas favoritas:

    18a. createListView(result);

19. Si no:

    19a. toast("No tienes recetas favoritas!");

29. Ver Recomendaciones

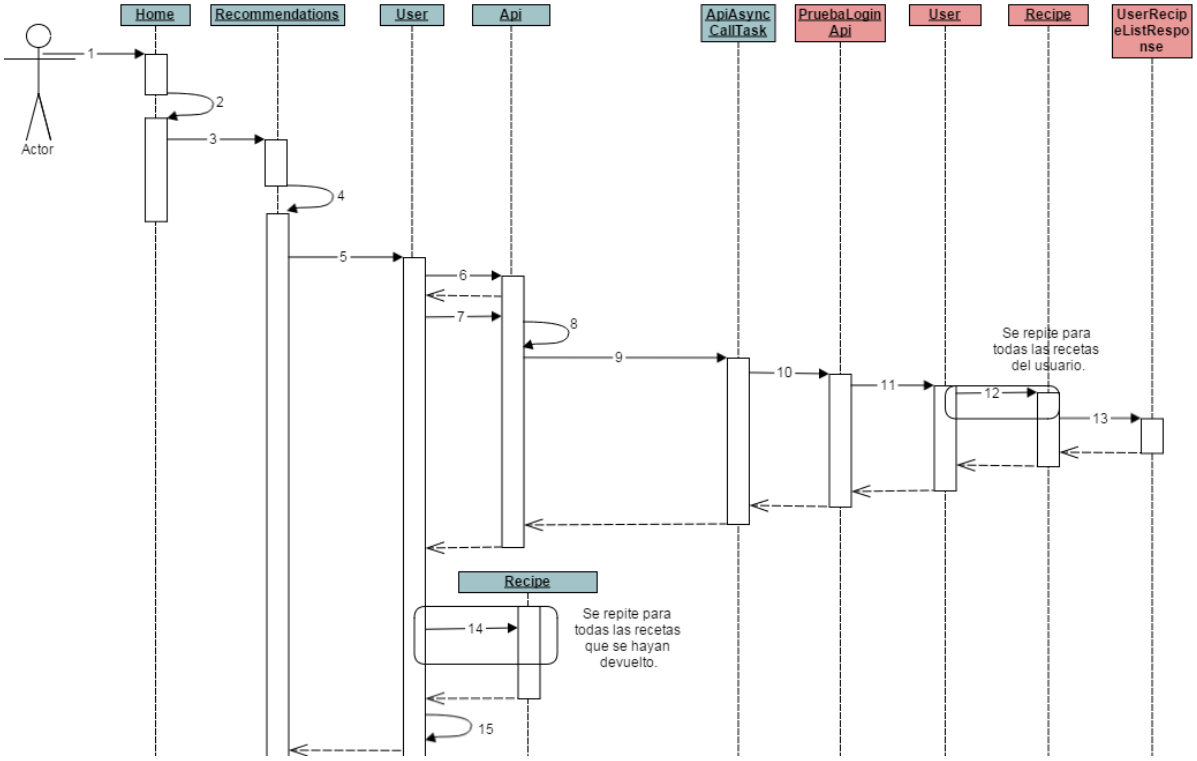


Ilustración 162: Diag. Sec. Ver Recomendaciones 1

1. El usuario accede a la pantalla Home.

2. onCreate(): void

3. El usuario pincha en "Recommendations" en el menú deslizable de la izquierda. -
startActivity(this, Recommendations.class);

4. onCreate(): void

5. User.list_favorites(user, new Callback<ArrayList<Recipe>>());

6. Api api = Api.getInstance();

7. api.service().recipe().list_favorites(lon, new
Callback<CookingstardustMessagesUserRecipeListResponse>();

8. Cookingstardust.Recipe.ListFavorites call = request.listFavorites(lon);

9. new ApiCallAsyncTask<CookingstardustMessagesUserRecipeListResponse>(call,
callback).execute();

10. call.execute();

11. User.get_user_favorites(request)

12. Recipe.to_message()

13. UserRecipeListResponse(recipes=finalrecipes)

14. Recipe.parse(item)

15. callback.onCompleted(null, response);

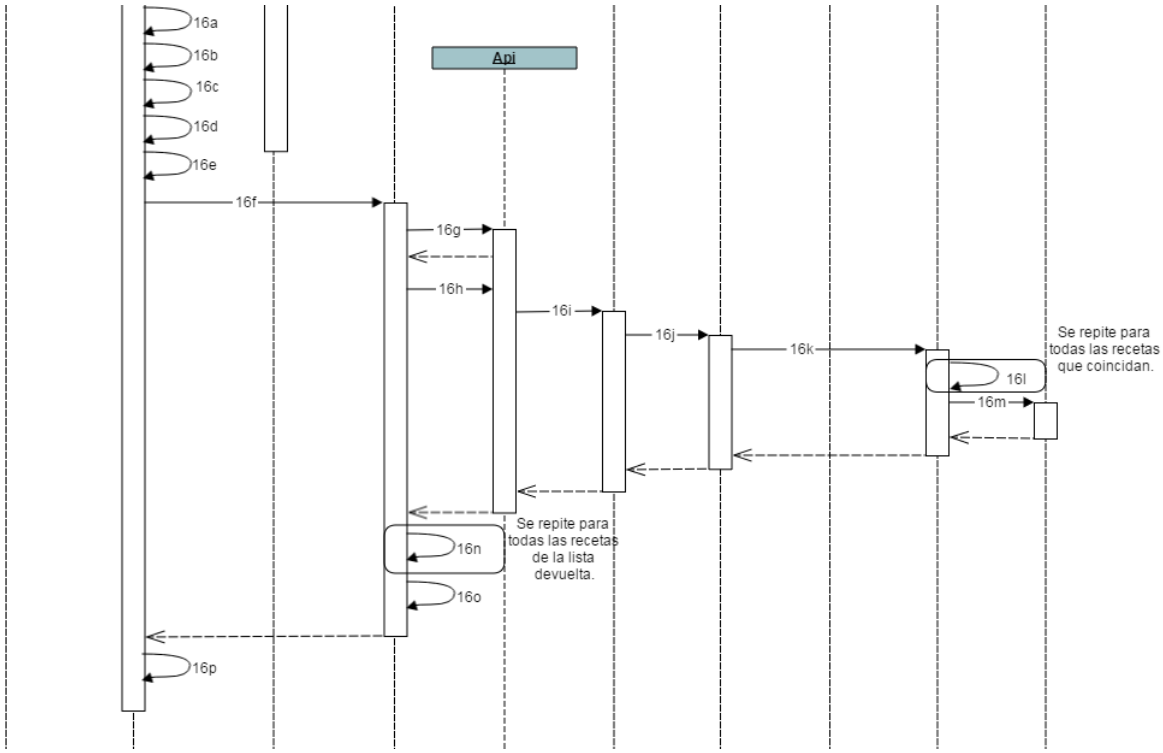


Ilustración 163: Diag. Sec. Ver Recomendaciones 2

16. Si el listado de recetas favoritas no es vacío:

    16a. calculateRecommendations(result);

16b. checkDieta(rec.getDiet());

16c. checkCocina(rec.getCuisine());

16d. checkPlato(rec.getRecipePlato());

16e. ordenar();

16f. Recipe.getRecommendations(query, user, new Callback<ArrayList<Recipe>>());

16g. Api api = Api.getInstance();

16h. api.service().recipe().recommendations(query, user, new Callback<CookingstardustMessagesUserRecipeListResponse>());

16i. new ApiCallAsyncTask<CookingstardustMessagesUserRecipeListResponse>(call, callback).execute();

16j. call.execute();

16k. Recipe.recomendaciones(request)

16l. re.to_message()

16m. UserRecipeListResponse(recipes=finalrecipes)

16n. Recipe.parse(item)

16o. callback.onCompleted(null, response);

16p. createListView(result);

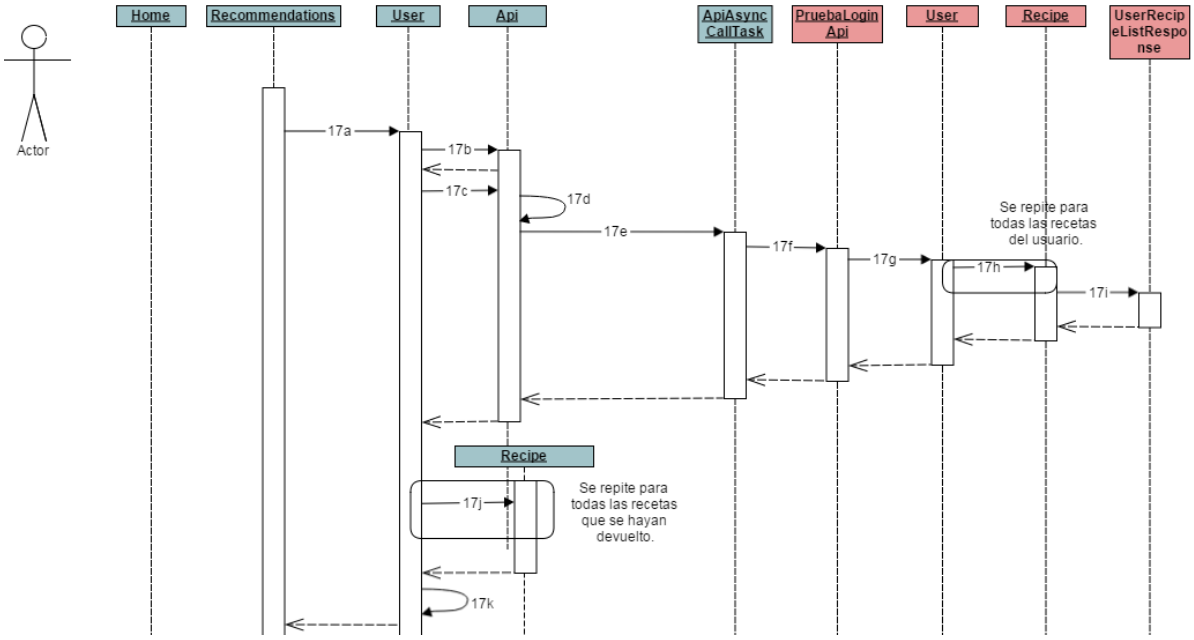


Ilustración 164: Diag. Sec. Ver Recomendaciones 3

17. Si el listado de recetas favoritas es vacío:

17a. User.get_best_recipes(user, new Callback<ArrayList<Recipe>>());

17b. Api api = Api.getInstance();

17c. api.service().recipe().get_best_recipes(lon, new Callback<CookingstardustMessagesUserRecipeListResponse>();

17d. Cookingstardust.Recipe.Bestrecipesuser call = request.bestrecipesuser(lon);

17e. new ApiCallAsyncTask<CookingstardustMessagesUserRecipeListResponse>(call, callback).execute();

17f. call.execute();

17g. User.best_recipes_user(request)

17h. Recipe.to_message()

17i. UserRecipeListResponse(recipes=finalrecipes)

17j. Recipe.parse(item)

17k. callback.onCompleted(null, response);

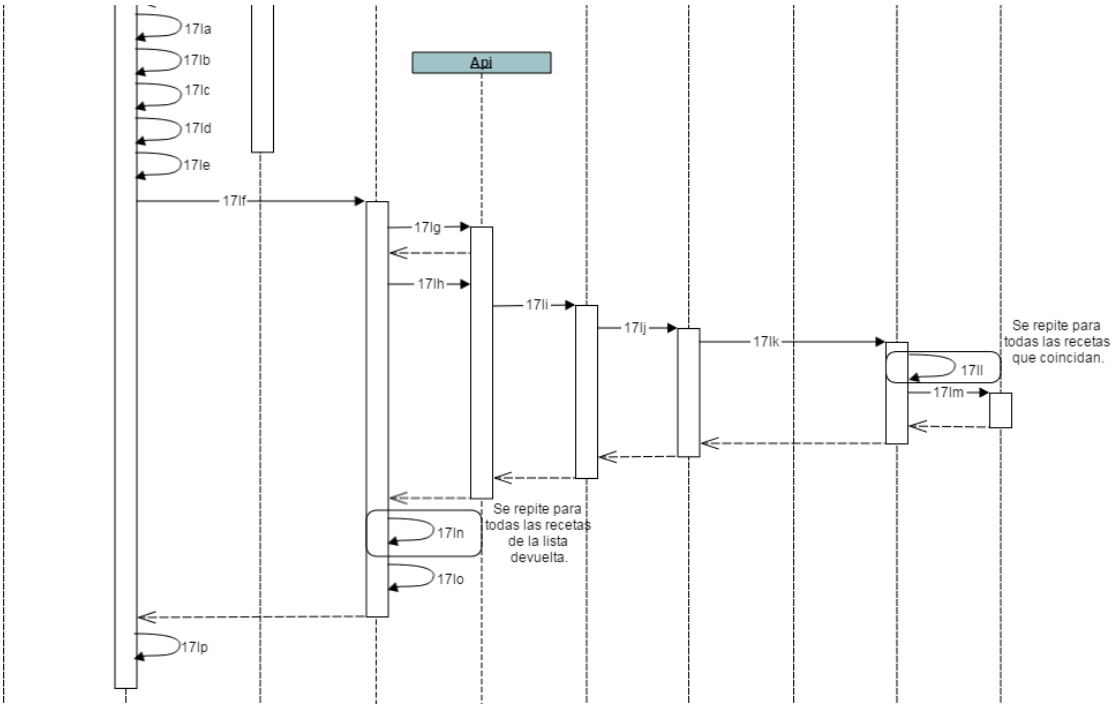


Ilustración 165: Diag. Sec. Ver Recomendaciones 4

17l. Si el listado de las recetas más puntuadas del usuario no es vacío:

17la. calculateRecommendations(result);

17lb. checkDieta(rec.getDiet());

17lc. checkCocina(rec.getCuisine());

17ld. checkPlato(rec.getRecipePlato());

17le. ordenar();

17lf. Recipe.getRecommendations(query, user, new Callback<ArrayList<Recipe>>());

17lg. Api api = Api.getInstance();

17lh. api.service().recipe().recommendations(query, user, new Callback<CookingstardustMessagesUserRecipeListResponse>());

17li. new ApiCallAsyncTask<CookingstardustMessagesUserRecipeListResponse>(call, callback).execute();

17lj. call.execute();

17lk. Recipe.recomendaciones(request)

17ll. re.to_message()

17lm. UserRecipeListResponse(recipes=finalrecipes)

17ln. Recipe.parse(item)

17lo. callback.onCompleted(null, response);

17lp. createListView(result);

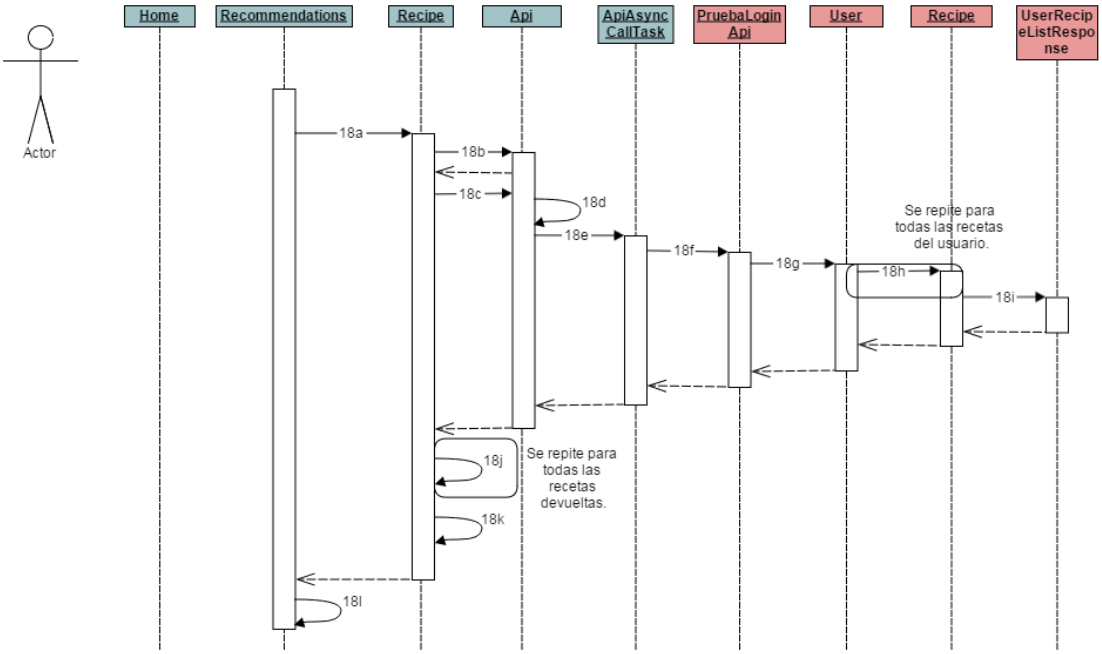


Ilustración 166: Diag. Sec. Ver Recomendaciones 5

18. Si es vacío:

18a. Recipe.getBestRecipes("nohacefalta", new Callback<ArrayList<Recipe>>());

18b. Api api = Api.getInstance();

18c. api.service().recipe().best_recipes(lon, new Callback<CookingstardustMessagesUserRecipeListResponse>();

18d. Cookingstardust.Recipe.BestRecipes call = request.bestrecipes(lon);

18e. new ApiCallAsyncTask<CookingstardustMessagesUserRecipeListResponse>(call, callback).execute();

18f. call.execute();

18g. User.best_recipes(request)

18h. Recipe.to_message()

18i. UserRecipeListResponse(recipes=finalrecipes)

18j. Recipe.parse(item)

18k. callback.onCompleted(null, response);

18l. createListView(result);

30. Ver perfil de un amigo

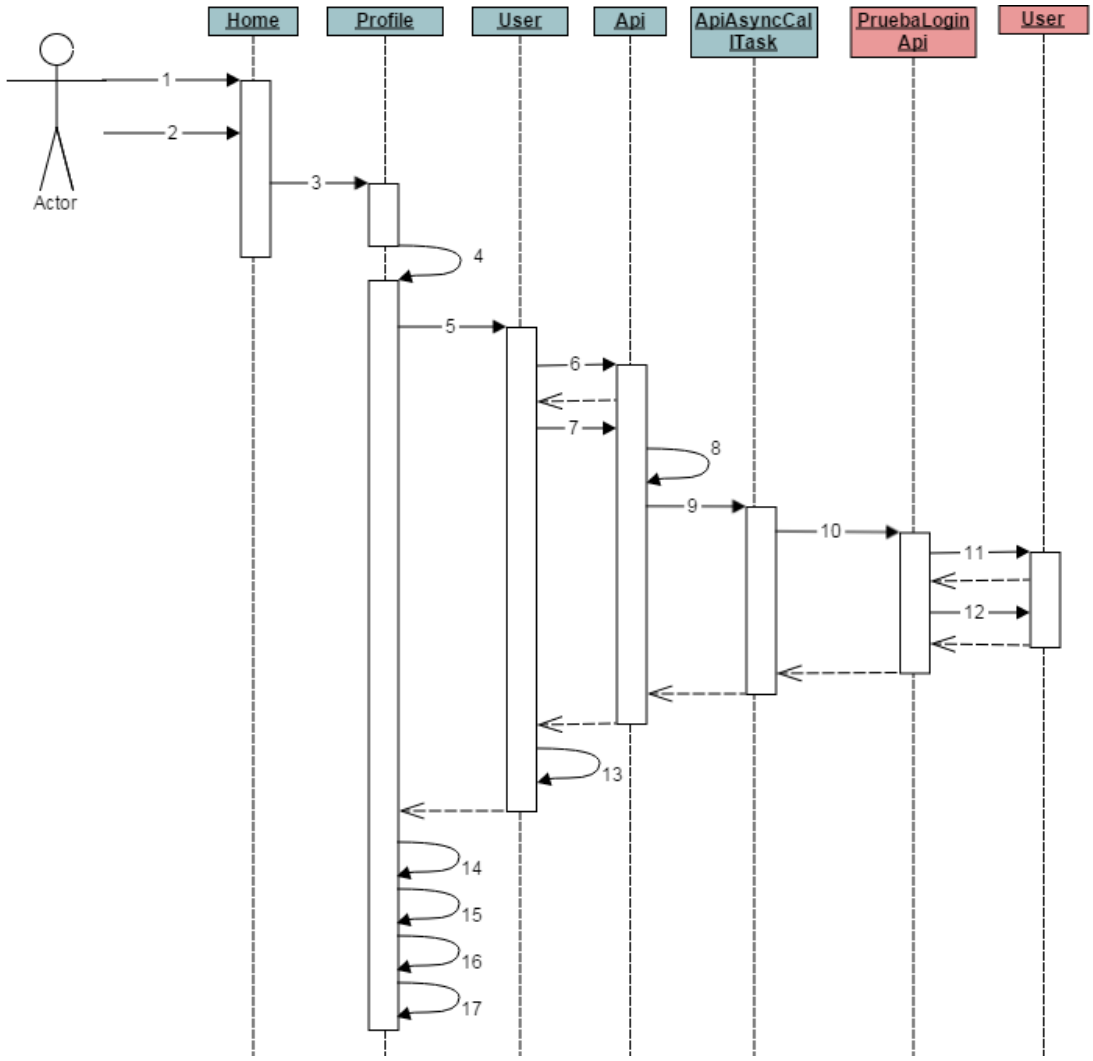


Ilustración 167: Diag. Sec. Ver Perfil Amigo

1. El usuario accede a su listado de usuarios que son sus amigos.
2. El usuario pincha sobre el usuario del que quiera consultar el perfil.
3. startActivity(new Intent(this, Profile.class));
4. onCreate(): void
5. getUserInfo(lon, new Callback<User>());
6. Api api = Api.getInstance();
7. api.service().user().get_info(lon, new Callback<CookingstardustMessagesUserAllResponseMessage>());
8. Cookingstardust.User.GetUser call = request.getUser(lon);
9. new ApiCallAsyncTask<CookingstardustMessagesUserAllResponseMessage>(call, callback).execute();
10. call.execute();
11. User.get_user_info(request)

12. user.to_message()
13. callback.onCompleted(null, User.parse(result));
14. comprobarFollowersFollowing(result);
15. rl.setVisibility(View.VISIBLE);
16. followButton.setText("Siguiendo");
17. followButton.setVisibility(View.VISIBLE);

31. Ver perfil de un un usuario no amigo

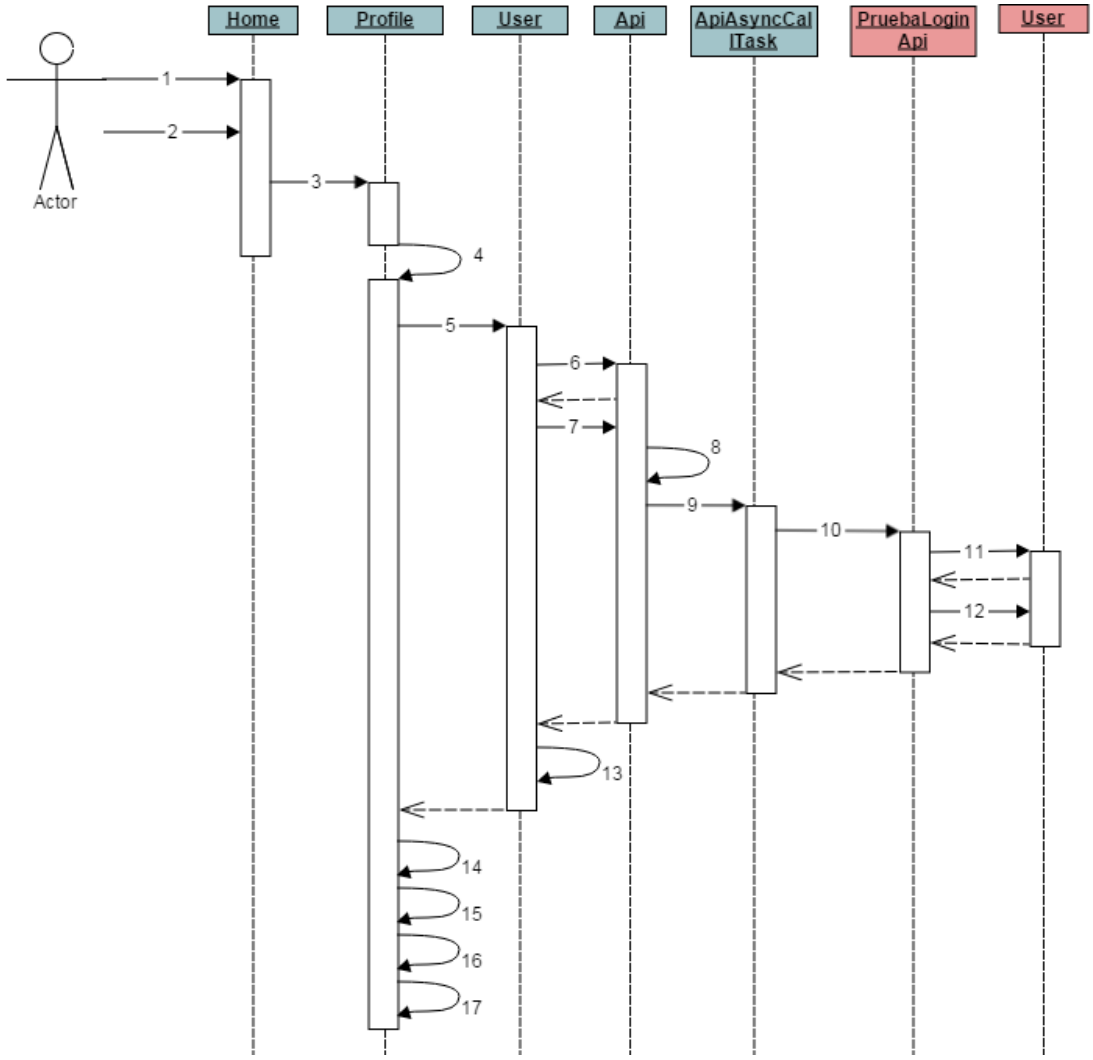


Ilustración 168: Diag. Sec. Ver Perfil No Amigo

1. El usuario accede a su listado de usuarios que no son sus amigos, ya sea mediante búsqueda o gente que le sigue pero que él no ha agreagado a su lista de amigos.
2. El usuario pincha sobre el usuario del que quiera consultar el perfil.
3. startActivity(new Intent(this, Profile.class));
4. onCreate(): void
5. getUserInfo(lon, new Callback<User>());

6. Api api = Api.getInstance();
7. api.service().user().get_info(lon, new
Callback<CookingstardustMessagesUserAllResponseMessage>());
8. Cookingstardust.User.GetUser call = request.getUser(lon);
9. new ApiCallAsyncTask<CookingstardustMessagesUserAllResponseMessage>(call,
callback).execute();
10. call.execute();
11. User.get_user_info(request)
12. user.to_message()
13. callback.onCompleted(null, User.parse(result));
14. comprobarFollowersFollowing(result);
15. rl.setVisibility(View.VISIBLE);
16. followButton.setText("Dejar de seguir");
17. followButton.setVisibility(View.VISIBLE);

32.Modificar Receta

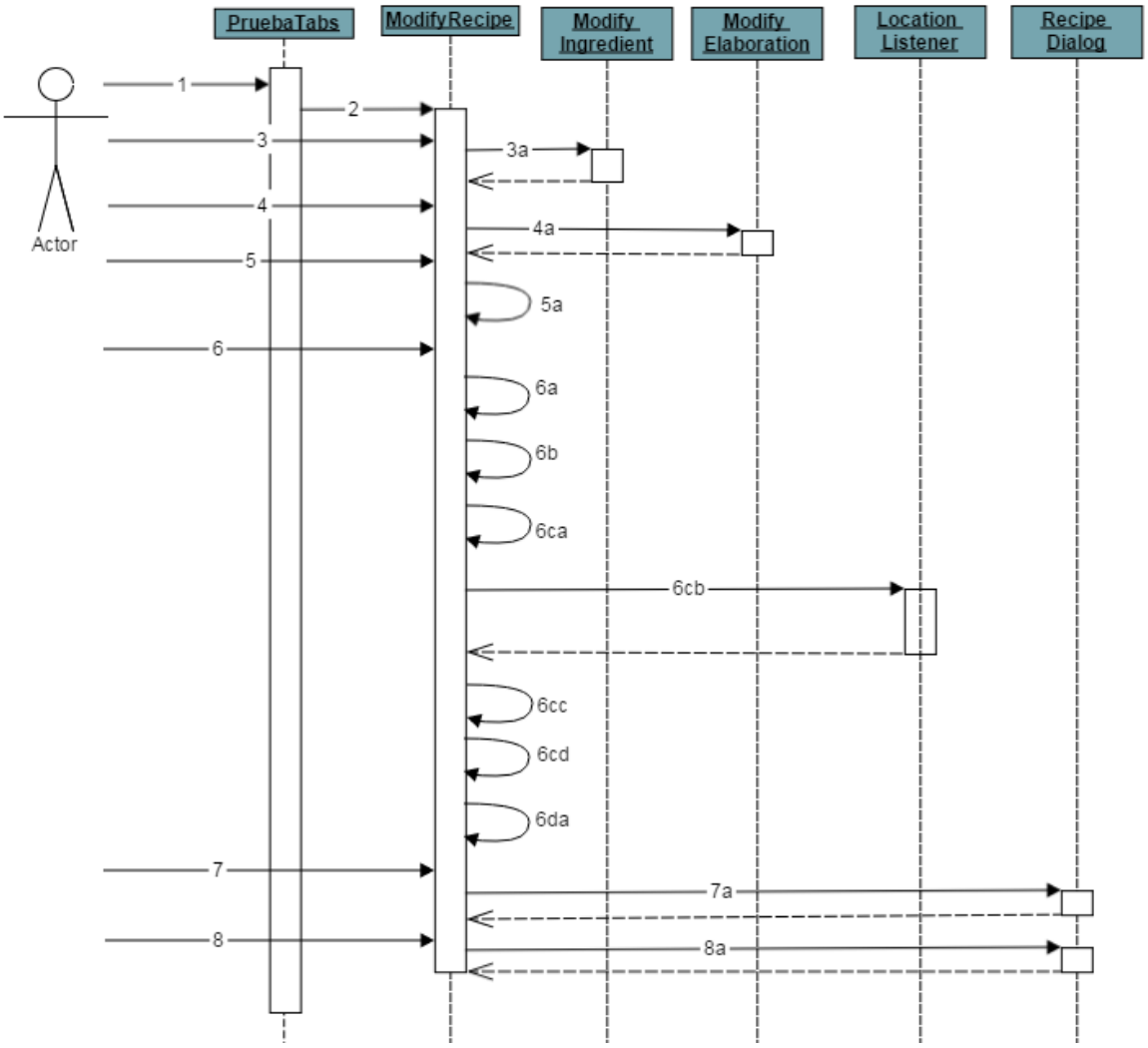


Ilustración 169: Diag. Sec. Modificar Receta 1

1. El usuario accede a cualquier receta suya y pincha "Modificar" en las opciones.
2. startActivity(this, ModifyRecipe.class)
3. El usuario pulsa el botón de "Modificar Ingrediente":
    3a. EXTEND CU "Modificar Ingredientes"
4. El usuario pulsa el botón de "Modificar Pasos":
    4a. EXTEND CU "Modificar Pasos".
5. El usuario pulsa el botón de "Cancelar":
    5a. this.finish();
6. El usuario pulsa el botón de "Localización":
    6a. SharedPreferences location = getSharedPreferences(PREFS_NAME, 0);
    6b. location.getBoolean("location", false);
    6c. Si la localización está activada:
        6ca. getSystemService(Context.LOCATION_SERVICE);
        6cb. LocationListener locationListener = new LocationListener();

6cc. latitude = Double.toString(location.getLatitude());

6cd. longitude = Double.toString(location.getLongitude());

6d. Si no lo está:

6da. toast("Por favor, activa la localizacion en la pantalla de Configuracion.");

7. El usuario pulsa el botón de "Modificar Foto":

7a. EXTEND CU "Añadir Foto".

8. El usuario pulsa el botón de "Modificar Foto por Paso":

8a. EXTEND CU "Añadir Foto por Paso".

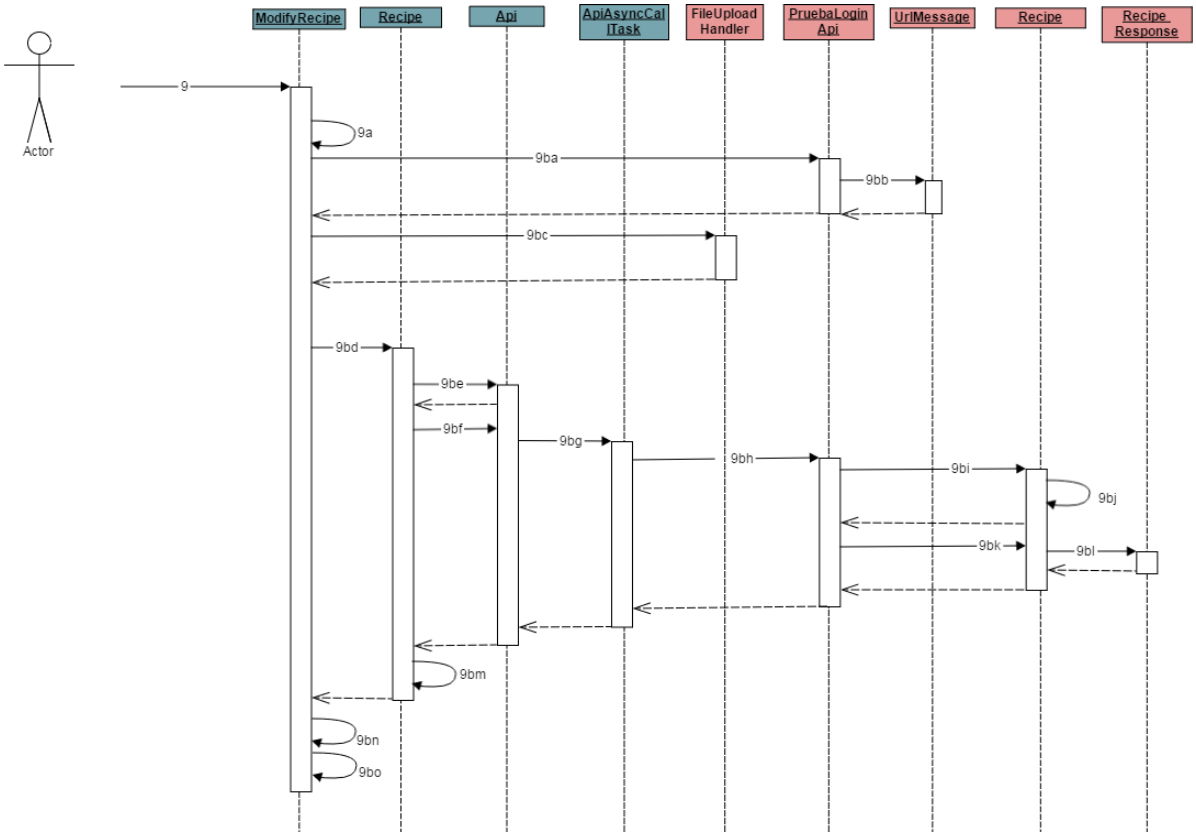


Ilustración 170: Diag. Sec. Modificar Receta 2

9. El usuario pulsa el botón "Modificar":

Si el nombre, los ingredientes, los pasos, o la dificultad están vacíos:

9a. toast("El nombre, ingredientes, elaboración, dificultad o tiempo no pueden estar vacíos.");

Sino:

9b. Si el usuario ha escogido una foto:

9ba. Ion.with(this).load(URL).asJsonObject().setCallback(new FutureCallback<JsonObject>(); //nos devuelve la url donde se guardará la foto

9bb. UrlMessage(url=url)

9bc.
Ion.with(getApplicationContext()).load(finalUrl).setMultipartParameter("file", "image/jpeg").setMultipartFile("file", new File(picturePath)).asJsonObject().setCallback(new FutureCallback<JsonObject>();
9bd. Recipe.modifyRecipe(titleView, ingredients, elaboracion, dietView, cuisineView, difficultyView, time_final, people_final, platoView, lon, urlKey, location, latitude, longitude, new Callback<Recipe>();
9be. Api api = Api.getInstance();
9bf. api.service().recipe().modifyRecipe(recipe, new Callback<CookingstardustMessagesRecipeResponse>();
9bg. new ApiCallAsyncTask<CookingstardustMessagesRecipeResponse>(call, callback).execute();
9bh. call.execute();
9bi. Recipe.modify_recipe(request)
9bj. recipe.put()
9bk. recipe.to_message()
9bl. return RecipeResponse()
9bm. callback.onCompleted(null, Recipe.parse(result));
9bn. toast("Receta " + result.getTitle() + "guardada exitosamente!");
9bo. this.finish();

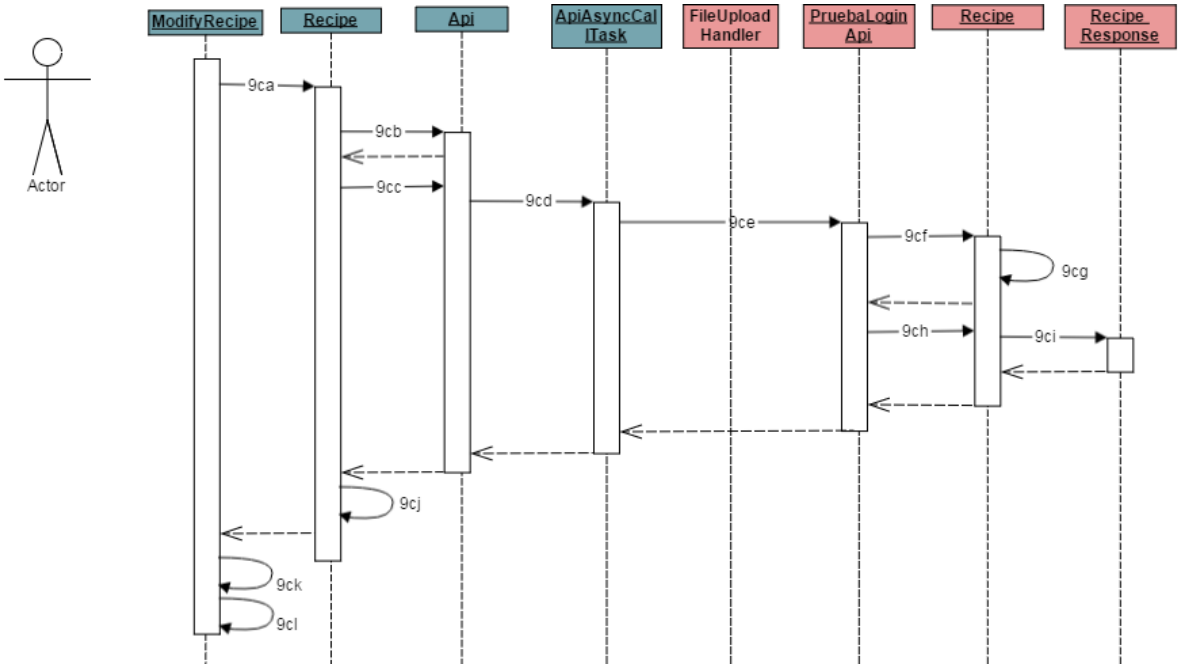


Ilustración 171: Diag. Sec. Modificar Receta 3

9c. Si el usuario no ha escogido ninguna foto:

9ca. Recipe.modifyRecipe(titleView, ingredients, elaboracion, dietView, cuisineView, difficultyView, time_final, people_final, platoView, lon, urlKey, location, latitude, longitude, new Callback<Recipe>();

9cb. Api api = Api.getInstance();

9cc. api.service().recipe().modifyRecipe(recipe, new Callback<CookingstardustMessagesRecipeResponse>();

9cd. new ApiCallAsyncTask<CookingstardustMessagesRecipeResponse>(call, callback).execute();

9ce. call.execute();

9cf. Recipe.put_from_message(request)

9cg. recipe.put()

9ch. recipe.to_message()

9ci. return RecipeResponse()

9cj. callback.onCompleted(null, Recipe.parse(result));

9ck. toast("Receta " + result.getTitle() + "guardada exitosamente!");

9cl. this.finish();

33.33Modificar Ingrediente

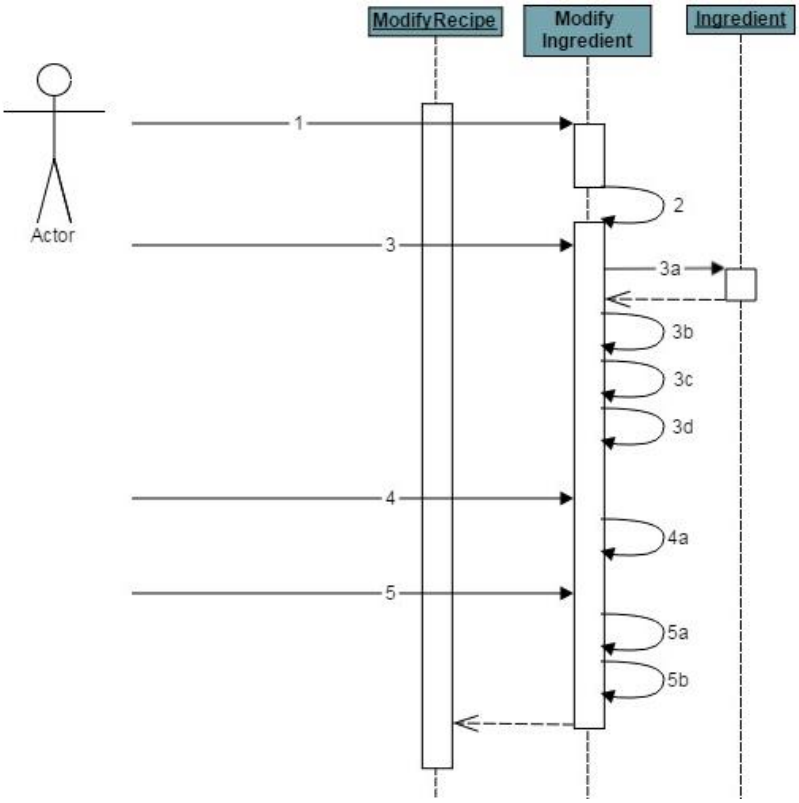


Ilustración 172: Diag. Sec. Modificar Ingrediente

1. startActivityForResult(new Intent(this, ModifyIngredients.class), REQUEST_CODE_INGS);
2. onCreate(): void
3. Si el usuario ha pinchado en añadir:
    3a. Ingredient ing = new Ingredient();
    3b. ing.setIngName(editIng.getText().toString());
    3c. ing.setQuantity(Long.decode(editCantidad.getText().toString()));
    3d. ingredients.add(ing);
4. Sino, si pulsa cancelar:
    4a. this.finish();
5. Si pulsa en Guardar:
    5a. Intent data = new Intent();
    5b. setResult(RESULT_OK, data);

34.Modificar Paso

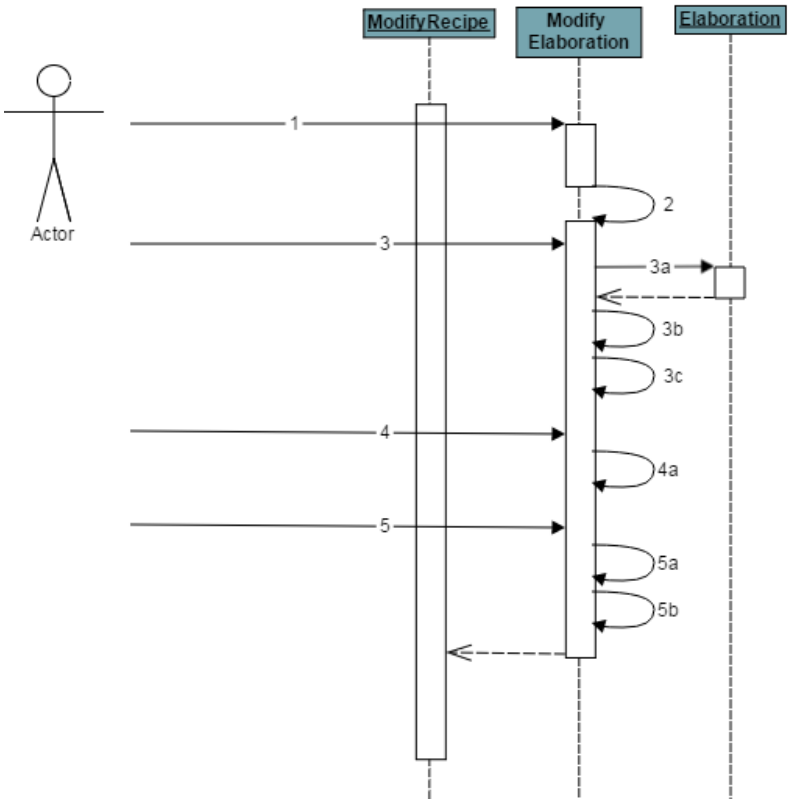


Ilustración 173: Modificar Paso

1. startActivityForResult(new Intent(this, ModifyElaboration.class), REQUEST_CODE_INGS);
2. onCreate(): void
3. Si el usuario ha pinchado en añadir:
    3a. Elaboration e = new Elaboration();
    3b. e.setStep(editStep.getText().toString());
    3c. elaboration.add(e);
4. Sino, si pulsa cancelar:
    4a. this.finish();
5. Si pulsa en Guardar:
    5a. Intent data = new Intent();
    5b. setResult(RESULT_OK, data);

35.Cocinar receta

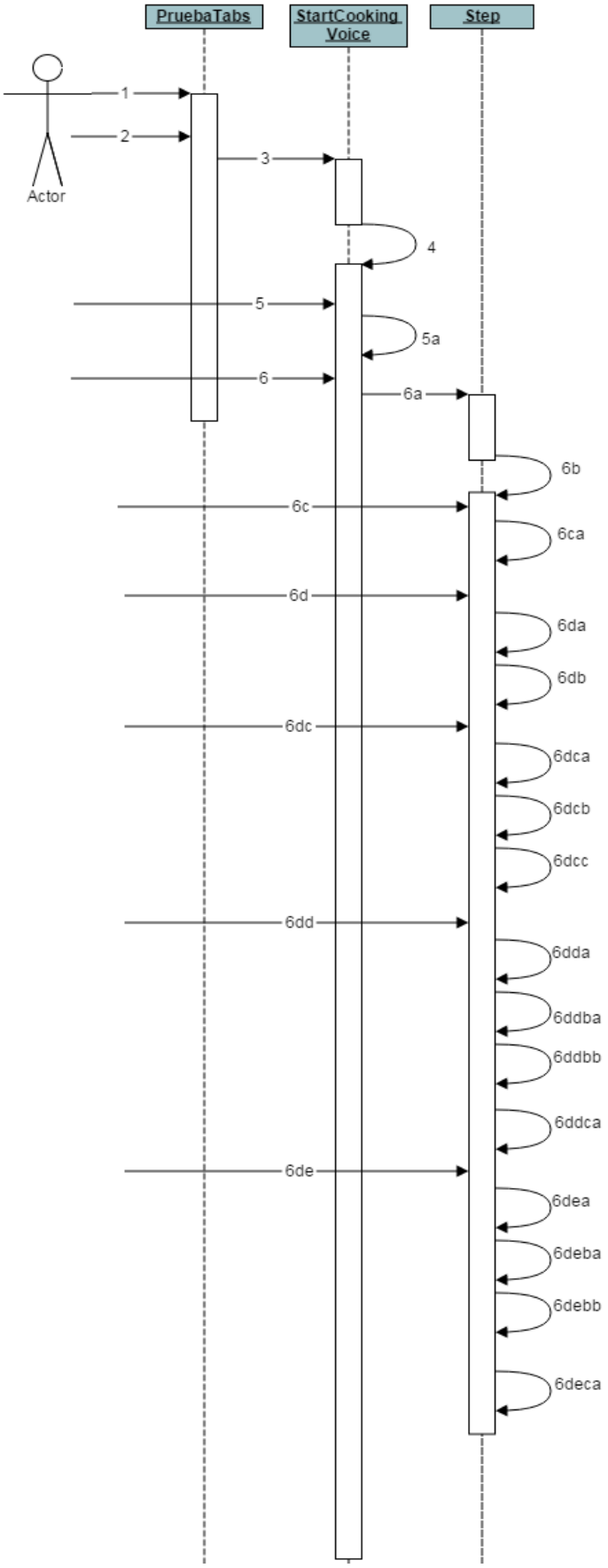


Ilustración 174: Cocinar Receta

1. El usuario accede a cualquier receta.
2. El usuario pulsa en el botón de "Cocinar".
3. startActivity(this, StartActivityCooking.class);
4. onCreate(): void
5. El usuario pincha en Cancelar:
    5a. this.finish();
6. El usuario pincha en Aceptar:
    6a. startActivity(this, Step.class);
    6b. onCreate(): void
    6c. El usuario pulsa en Ayuda:
        6ca. alertDialog.show();
    6d. El usuario pulsa en Activar voz.
        6da. launchRecognitionIntent();
        6db. sr.startListening(intent);
        6dc. El usuario dice "Finalizar":
            6dca. onResults(Bundle results): void
            6dcb. sr.stopListening();
            6dcc. finishActivity();
        6dd. El usuario dice "Siguiente":
            6dda. onResults(Bundle results): void
            6ddb. Si hay pasos para avanzar:
                6ddba. comprobarBotones();
                6ddbb. launchRecognitionIntent();
            6ddc: Si no los hay:
                6ddca: toast("No hay más pasos.");
        6de. El usuario dice "Siguiente":
            6dea. onResults(Bundle results): void
            6deb. Si hay pasos para avanzar:
                6deba. comprobarBotones();
                6debb. launchRecognitionIntent();
            6dec: Si no los hay:
                6deca: toast("No hay más pasos.");