ANDROID

DEL DISEÑO DE LA ARQUITECTURA AL DESPLIEGUE PROFESIONAL

ÁNGEL VÁZQUEZ VÁZQUEZ
JUAN ANTONIO GÓMEZ GUTIÉRREZ
RAMÓN SERRANO VALERO

Contenidos Interactivos Web

Alfaomega

Marcombo

ANDROID: DEL DISEÑO DE LA ARQUITECTURA AL DESPLIEGUE PROFESIONAL

ANDROID: DEL DISEÑO DE LA ARQUITECTURA AL DESPLIEGUE PROFESIONAL

Ángel Vázquez Vázquez - Juan Antonio Gómez Gutiérrez - Ramón Serrano Valero

Alfaomega

Marcombo

Datos catalográficos
Vázquez, Ángel; Gómez, Juan; Serrano, Ramón
Android: del diseño de la arquitectura al despliegue profesional
Primera Edición

Alfaomega Grupo Editor, S.A. de C.V., México
ISBN: 978-607-538-391-0
Formato: 17 x 23 cm Páginas: 304

Diseño de la cubierta y maquetación: ArteMio
Correctora: Ester Valbona
Revisor técnico: Pablo Martínez Izurzu
Directora de producción: María Rosa Castillo

Android: del diseño de la arquitectura al despliegue profesional
Ángel Vázquez Vázquez; Juan Antonio Gutiérrez y Ramón Serrano Valero
ISBN: 978-84-267-2649-0, de la edición publicada por MARCOMBO, S.A., Barcelona, España
Derechos reservados © 2018 MARCOMBO, S.A.

Primera edición: Alfaomega Grupo Editor, México, enero 2019

© 2019 Alfaomega Grupo Editor, S.A. de C.V.
Pitágoras 1139, Col. Del Valle, 03100, México D.F.

Miembro de la Cámara Nacional de la Industria Editorial Mexicana
Registro No. 2317

Pág. Web: **http://www.alfaomega.com.mx**
E-mail: **atencionalcliente@alfaomega.com.mx**

ISBN: 978-607-538-391-0

Derechos reservados:
Esta obra es propiedad intelectual de su autor y los derechos de publicación en lengua española han sido legalmente transferidos al editor. Prohibida su reproducción parcial o total por cualquier medio sin permiso por escrito del propietario de los derechos del copyright.

Nota importante:
La información contenida en esta obra tiene un fin exclusivamente didáctico y, por lo tanto, no está previsto su aprovechamiento a nivel profesional o industrial. Las indicaciones técnicas y programas incluidos, han sido elaborados con gran cuidado por el autor y reproducidos bajo estrictas normas de control. ALFAOMEGA GRUPO EDITOR, S.A. de C.V. no será jurídicamente responsable por: errores u omisiones; daños y perjuicios que se pudieran atribuir al uso de la información comprendida en este libro y en el material de apoyo en la web, ni por la utilización indebida que pudiera dársele. d e s c a r g a do e n: e y b o o k s. c o m

Edición autorizada para venta en México y todo el continente americano.

Impreso en México. Printed in Mexico.

Empresas del grupo:

México: Alfaomega Grupo Editor, S.A. de C.V. – Dr. Isidoro Olvera (Eje 2 sur) No. 74, Col. Doctores, Ciudad de México – C.P. 06720. Tel.: (52-55) 5575-5022 – Fax: (52-55) 5575-2420 / 2490. Sin costo: 01-800-020-4396 – E-mail: atencionalcliente@alfaomega.com.mx

Colombia: Alfaomega Colombiana S.A. – Calle 62 No. 20-46, Barrio San Luis, Bogotá, Colombia, Tels.: (57-1) 746 0102 / 210 0415 – E-mail: cliente@alfaomega.com.co

Chile: Alfaomega Grupo Editor, S.A. – Av. Providencia 1443. Oficina 24, Santiago, Chile
Tel.: (56-2) 2235-4248 – Fax: (56-2) 2235-5786 – E-mail: agechile@alfaomega.cl

Argentina: Alfaomega Grupo Editor Argentino, S.A. – Av. Córdoba 1215, piso 10, CP: 1055, Buenos Aires, Argentina, – Tel./Fax: (54-11) 4811-0887 y 4811 7183 – E-mail: ventas@alfaomegaeditor.com.ar

Juan: Dedico mi libro a mi mujer y a mi hijo a los que quiero con todo mi corazón. A mis padres y hermana que tanto apoyo y cariño me han dado a lo largo de la vida. También dedico este libro a mis amigos Víctor G. y Jordi P. y a mis compañeros Jordi B. y Xavi T. con los que he vivido tantas cosas buenas y a los que tanto recuerdo. Y, como no, agradezco a mis amigos Ramón y Ángel las muchas clases de todo y los ánimos que me han dado desde el primer momento en que los conocí.

Ángel: Dedico este libro a mis padres que me han apoyado en todo lo que he llevado a cabo hasta el día de hoy. A mis compañeros y ya amigos Ramón y Juan, por todas esas conversaciones y trabajos realizados hasta la fecha; esta ya es nuestra segunda obra y estoy seguro que habrá más porque es un placer trabajar con vosotros.

Me gustaría compartir este libro con la que me acompaña en todas mis aventuras y hace que mi día a día sea genial. No será una obra como la de Serrat pero, ¡este libro es para ti Lucía!

Ramón: Llevaba tiempo queriendo escribir sobre arquitectura Android, de modo que cuando surgió la oportunidad de trabajar con mis mejores amigos en este mundo Android, no lo dudé. Dedico este libro a mi mujer Sara V., gracias por creer siempre en mí y apoyarme en que haga lo que me gusta, si no existieras tendría que inventarte. A mis amigos Ángel y Juan, ¡lo hemos conseguido! Gracias por esas reuniones de domingo para organizarnos y compartir conocimientos, finalmente pusimos un trocito de cada uno de nosotros en este libro. Hoy somos una mejor versión de nosotros mismos.

Quisiéramos dedicar este libro a nuestro gran compañero Pablo Fernández, no le fue posible estar en este proyecto. Quizá en la próxima aventura podamos recuperarlo, ya que es la cuarta pata de este gran equipo.

PLATAFORMA DE CONTENIDOS INTERACTIVOS

Para tener acceso al material de la plataforma de contenidos interactivos del libro, siga los siguientes pasos:

1. Ir a la página: **http://libroweb.alfaomega.com.mx**
2. Ir a la sección *Catálogo* y seleccionar la imagen de la portada del libro, al dar doble clic sobre ella, tendrá acceso al material descargable, el cual consiste en **el código fuente de los ejemplos**, complemento imprescindible de este libro, el cual podrá descomprimir con la clave ANDROID7

CONTENIDO

PARTE 1: BLOQUE TEÓRICO

CAPÍTULO 1: INTRODUCCIÓN 11

¿A quién va dirigido? 12
¿Cuál es el propósito? 12
Darás respuestas a 13
Estructura del libro 13
En primer lugar, algunos conceptos teóricos 13
Comienza la práctica 15
Ahora practicaremos más en serio 15
Kotlin para rematar 16

CAPÍTULO 2: ARQUITECTURA CLEAN 17

CAPÍTULO 3: PRINCIPIOS SOLID 21

CAPÍTULO 4: PATRONES DE DISEÑO 27

MVC (Modelo Vista Controlador) 28
MVP (Modelo Vista Presentador) 29
Comparando MVC y MVP 31
Patrón Observer 31

CAPÍTULO 5: INYECCIÓN DE DEPENDENCIAS 35

¿Qué es la inversión de dependencias? 36
¿En qué nos ayuda la inyección de dependencias? 36
¿En qué consiste? 36

CAPÍTULO 6: DAGGER 2 37

CAPÍTULO 7: ARQUITECTURA DEL PROYECTO ANDROID 41

CAPÍTULO 8: TEST UNITARIOS 43

JUnit 44
Aplicación de ejemplo 46
Test con JUnit 50
Test con Mockito 55

CAPÍTULO 9: BITBUCKET & JENKINS 67

Jenkins 74
Instalación 75

CAPÍTULO 10: TRELLO 93

CAPÍTULO 11: SLACK **97**
Creación de un grupo en Slack 100

CAPÍTULO 12: BITBUCKET **107**
Inicializar Git desde un directorio existente 109
Clonar repositorio en un directorio 111
Integrar Bitbucket con Slack 112

PARTE 2: PROYECTOS DE PRUEBA

CAPÍTULO 1: PROYECTO BASE DAGGER 2 **117**

CAPÍTULO 2: PROYECTO BASE DAGGER V.2.11 **123**

CAPÍTULO 3: PROYECTO CLEAN MVP CON DAGGER 2.11 Y RXJAVA **131**

PARTE 3: DESARROLLO DE UNA APP PROFESIONAL

CAPÍTULO 1: DESARROLLO APP PROFESIONAL **167**
Trello 168
Git 169
Firebase 172
GSON 179
Retrofit 2 180
Butterknife 182
Picasso 183
Realm Database 183
Desarrollo de la aplicación 186

PARTE 4: KOTLIN

CAPÍTULO 1: KOTLIN **265**
Hola Mundo (IntelliJ IDEA) 267
Variables y tipos 269
Arrays 270
Listas 272
Funciones 274
Colecciones y funciones 275
If-else-when 277
Hola Mundo en Kotlin 279

CAPÍTULO 2: KOTLIN MVP **285**
Módulo del dominio 286
Módulo de datos 290
Módulo de presentación 293

REFLEXIONES FINALES

PARTE 1:
BLOQUE TEÓRICO

CAPÍTULO 1:
Introducción

La evolución tecnológica nos obliga a ser cada día más competitivos y al mismo tiempo flexibles para poder adaptarnos rápidamente no tan solo a las necesidades de nuestros clientes sino, a la constante aparición de productos que pueden convertir en obsoletas nuestras flamantes aplicaciones. Si a esto le sumamos el problema de ser eficientes en nuestros desarrollos y de utilizar estándares y metodologías de trabajo para que cualquier miembro de un equipo pueda integrarse fácilmente a un proyecto, llegaremos a la conclusión de que no basta con conocer un lenguaje o tecnología, sino que, hemos de establecer una serie de normas, criterios y buenas prácticas basadas en la experiencia de la comunidad y nuestra propia.

Con el tiempo, todo cambia y como desarrolladores de software que somos, debemos realizar un aprendizaje continuo. La elección de una determinada tecnología o patrón de diseño, para resolver un problema software concreto hoy en día, mañana puede que exista una alternativa que resuelva mejor el problema, de forma más eficiente y limpia.

¿A quién va dirigido?

Este libro va dirigido a intrépidos desarrolladores de software, que persiguen, no solo crear aplicaciones utilizando un determinado lenguaje, sino además disfrutar del camino que los lleve a su objetivo, ya que es así como se consiguen los mejores resultados.

Va dirigido a ti, lector, que ya conoces Java y ves posibilidades en realizar una pequeña aplicación Android si no lo has hecho ya. En cualquier caso, como puedes imaginar o habrás podido comprobar, aunque se comparta lenguaje de programación, las cosas son algo diferentes cuando programas para dispositivos móviles.

Llevas tiempo programando. Sabes que tus aplicaciones funcionan e incluso, hay usuarios finales que lo están utilizando. Es posible que además hubieras subido tu app móvil al Play Store de Google, porque también dominas la firma de la app.

Seguramente cuando desarrollaste, lo hiciste de forma individual o en equipo y quieres mejorar tus habilidades y ser capaz de definir una arquitectura limpia reutilizable por todo el equipo de desarrollo.

¿Cuál es el propósito?

Este libro te aportará una visión del desarrollo Android diferente y una pequeña guía o una serie de recomendaciones que ofrecemos los autores para ayudarte a crear o mantener tu aplicación con menos esfuerzo y más control sobre la misma solo o en compañía de otros desarrolladores.

Sabes programar, pero la cuestión es, si lo estás haciendo de una forma organizada, utilizando un patrón de diseño determinado y de forma que tu desarrollo pueda ser utilizado por varios miembros de un equipo, mediante las herramientas que te proponemos en el libro.

Este libro te ayudará a crecer como arquitecto software, en el ámbito de la empresa, trabajando con equipos pequeños.

Algunos de los propósitos del libro son:

1. Aprenderás los principios SOLID y sabrás aplicarlos.
2. Entenderás lo que es una arquitectura limpia.
3. Conocerás distintos patrones de diseño.
4. Serás capaz de estructurar tu proyecto en n capas.
5. Dominarás la inyección de dependencias y Dagger 2.
6. Utilizarás control de versiones de código fuente.
7. Plantearás el desarrollo ágil, organizando las tareas.
8. Definirás casos de uso, orientando el desarrollo a implementarlos.
9. Ejecutarás pruebas unitarias, para validar la funcionalidad de tu código.
10. Aprenderás que es integración continua y utilizarás Jenkins.
11. Conocerás Kotlin, el nuevo lenguaje de desarrollo para Android e impulsado por Google.
12. Te convertirás en un desarrollador profesional Android, apto para el desarrollo en la empresa, en equipos de desarrollo.

Darás respuestas a...

Estás pensando en crear tu propia aplicación o ya la tienes desarrollada y colgada en la Play Store, ¿estás preparado para crecer a medida que lo demandan tus usuarios? Cuando subes una nueva versión ¿estás totalmente tranquilo de que lo que has tocado no genera errores en esa u otra funcionalidad? ¿Cada cuánto generas nueva versión? ¿Tienes ese proceso automatizado? ¿Conoces nuevos lenguajes como Kotlin que es recomendado y soportado oficialmente por Google para desarrollo Android?

Estructura del libro

El libro consta principalmente de cuatro bloques. Uno teórico, uno práctico, aplicando la teoría aprendida en el bloque teórico, otro práctico desarrollando una app móvil profesional y un último bloque teórico y práctico sobre Kotlin, la gran novedad en Android.

En primer lugar, algunos conceptos teóricos

El primer bloque del libro, contiene los aspectos teóricos y transversales al desarrollo software, en la que presentamos los distintos conceptos y herramientas que recomendamos tener en cuenta y utilizar a la hora de abordar un proyecto que cumpla con los niveles de calidad y eficiencia que consideramos necesarios e imprescindibles.

En primer lugar, proponemos el uso de la arquitectura CLEAN basada en los cinco principios básicos SOLID, que a modo de resumen indicamos a continuación y que presentaremos más detalladamente en capítulos posteriores:

- **Single Responsibility**: Cada unidad de código debe tener una única responsabilidad.
- **Open/Closed**: Podemos extender las funcionalidades de un objeto, no modificarlas.

- **Liskov**: Un objeto puede ser siempre sustituido por otro objeto subtipo.
- **Interface segregation**: Es preferible usar varios interfaces específicos que no uno general.
- **Dependency inversion**: Un objeto debe depender de abstracciones.

La definición de arquitectura limpia CLEAN, permite la separación entre las diversas capas de la aplicación y aplica SOLID.

Seguidamente se presentarán los patrones de diseño, destacando la importancia de utilizar patrones en general ya que nos permiten asegurar la validez del código y establecer un lenguaje común dentro del equipo y de la propia comunidad en general.

Existe una gran cantidad de patrones que nos permiten resolver eficazmente planteamientos probados por muchos usuarios anteriormente y que aportan una gran fiabilidad a la resolución de problemas.

Posteriormente presentaremos algunos ejemplos de patrones de diseño clásicos como son MVC (Model View Controller), MVP (Model View Presenter) o el patrón Observador.

Con estos primeros conceptos preparamos el terreno para los siguientes temas de este bloque teórico como son la Inyección de Dependencias, mediante la cual, podremos suministrar objetos a una clase sin obligar a que la propia clase quien los cree. Se trata de un patrón muy utilizado en Java, C# y otros lenguajes basados en orientación a objetos.

Hablaremos de Dagger 2 como framework que nos ayudará a trabajar con Inyección de dependencias.

Insistiremos sobre como plantear la arquitectura limpia de un proyecto que nos permita separar el modelo o núcleo de la lógica de tu aplicación de todo el código relacionado con el framework Android.

Separando la arquitectura en las capas Data (acceso a datos), Domain (modelos de negocio, casos de uso, parsers entre capa data y presentation y viceversa) y Presentation (Interfaz de usuario, vistas, presentadores, etc.) nos ayudará en la consecución de un desarrollo claro y eficiente.

De esta forma facilitamos en gran medida la evolución de la propia aplicación mediante el desarrollo de nuevas funcionalidades y las pruebas. Sin olvidarnos de introducir, un framework de testing como JUnit o Mockito, ayudándonos a automatizar las pruebas, simulando la existencia de recursos difíciles de conseguir o preparar.

Por este motivo, realizaremos una presentación de la librería Mockito la cual nos ayudará mucho en estas tareas.

Por otra parte, es indispensable disponer de un control de versiones (tanto si trabajamos solos como en compañía) que nos permita compartir nuestros fuentes con un equipo de trabajo y también, asegurar la progresión de un desarrollo de forma que, en caso de problemas o necesidad de ramificar un desarrollo, podamos mantener el estado de un proyecto en diversos momentos del tiempo. Así, si interesa recuperar la situación de una determinada versión o fecha, podemos fácilmente localizarla y utilizarla.

Por esta razón, presentaremos BitBucket como sistema de alojamiento remoto, basado en Web similar a GitHub que nos permite gestionar repositorios de controles de versiones como Git o Mercurial.

Definiremos, además, un flujo de integración continua para nuestra app mediante Jenkins.

La integración continua nos ha de permitir compilar y ejecutar pruebas de nuestro proyecto de forma automatizada y frecuente con el objetivo de ir añadiendo valor constantemente al mismo manteniendo el nivel de calidad que hayamos determinado. En este sentido, Jenkins es una herramienta que nos ayudará a automatizar la fabricación de 'builds' o versiones de nuestra aplicación obteniendo el código fuente del repositorio de control de versiones, recompilándolo, testeándolo y desplegándolo en uno o varios entornos según nos pueda interesar.

Para finalizar este primer bloque, durante el desarrollo de un proyecto es imprescindible llevar a cabo una gestión del mismo para poder tutelar su avance y consecución del objetivo pretendido, así como coordinar los esfuerzos de los participantes del mismo. Por ello, presentaremos Trello como herramienta de gestión de proyectos, la cual facilita el trabajo en equipo de forma instantánea. Utiliza tableros y tarjetas para estructurar las tareas y su utilización resulta bastante sencilla e intuitiva acercándonos mínimamente al tablero Kanban, usado en desarrollo ágil de SCRUM.

Como complemento a Trello, os introduciremos en Slack como aplicación de mensajería que permite integrar sistemas externos tales como Twitter, Dropbox, Google Docs, etc) y que serán utilizados para la comunicación entre los integrantes del equipo de desarrollo, así como un sistema donde organizar documentos asociados al proyecto. Podremos combinar el sistema Slack con la integración continua de Jenkins, de tal forma que cuando un miembro del equipo, realice un Push o suba código al repositorio BitBucket, se lance un evento capturado por Slack para notificar al grupo de que se ha realizado una subida.

Comienza la práctica

En el siguiente bloque se comienzan a poner en práctica los conceptos de inyección de dependencias y Dagger 2. Crearemos un pequeño proyecto para aplicar los conceptos teóricos aprendidos en el primer bloque: elección de arquitectura, patrón MVP, inyección de dependencias, patrón observador.

Si bien podremos observar, se compone de tres pequeños proyectos:

- El primero de ellos una aplicación para aplicar arquitectura CLEAN y Dagger 2.
- El segundo proyecto, será idéntico al anterior, con la diferencia de que aplicaremos Dagger 2 versión 2.11 o superior, ya que la forma de estructurar el proyecto nos parece más adecuada y limpia que la anterior.
- El tercero consiste en una aplicación sencilla, un diccionario de termino y traducción, realizado con todo lo explicado a en el bloque teórico y utilizando Dagger 2.11 o superior.

Ahora practicaremos más en serio

En este bloque plantearemos un posible proyecto real sobre hoteles que crearemos junto con otros desarrolladores, diseñadores, scrum managers o clientes. Este equipo de personas tiene como objetivo crear la mejor aplicación de hoteles posible.

Para ello tendrán que tomar las decisiones de negocio y desarrollo oportunas que podrán dejar plasmadas en una herramienta de comunicación como Slack.

Todo este trabajo generado se descompondrá en tareas que gestionaremos con Trello creando un tablón para que podamos comunicar el estado del proyecto y gestionar nuestros casos de uso internamente.

Para crear nuestro MVP de opiniones sobre hoteles nos integraremos con diferentes proveedores de autenticación como Google, Facebook, así como el clásico Login mediante usuario y contraseña. Todo ello mediante Firebase Authentication. Conoceremos también, el potencial de FirebaseUser y de todas las posibilidades que nos permite, a la hora de agilizar el desarrollo.

Para la persistencia remota de los datos, emplearemos Firebase Realtime Database, una base de datos en tiempo real proporcionada por Firebase y aprenderemos a configurar reglas de acceso a los datos, así como, consumir los datos mediante Api Rest de Firebase.

Obtendremos la imagen del perfil del usuario y además la subiremos a Firebase, haciendo uso del sistema de almacenamiento en la nube de Firebase Cloud Store.

Explicaremos la librería de GSON, para convertir los datos JSON a clases de nuestra aplicación.

Para la comunicación remota entre app móvil y web api de Firebase, emplearemos Retrofit 2, practicando todas las acciones GET, POST, PUT, DELETE, que tendremos la oportunidad de utilizar en la app, así como el tratamiento de las respuestas obtenidas por estas peticiones, interpretando las respuestas HTTP.

Además, utilizaremos Butterknife para que el desarrollo sea más limpio y en menos líneas, aprovechando el potencial de esta librería de Jake Warton.

El almacenamiento local probaremos la base de datos Realm, muy popular en la actualidad, con gran potencial y muy rápido, permitiendo trabajar con transacciones.

Transversalmente, aplicaremos los conceptos teóricos anteriormente vistos para estructurar nuestras aplicaciones, y utilizando las técnicas aprendidas con anterioridad.

Kotlin para rematar

Finalmente, os trasladaremos una de las novedades más excitantes del momento, la consolidación de Kotlin como lenguaje de programación de aplicaciones Android.

Kotlin es un lenguaje oficialmente soportado por Google y es sencillo de aprender para programadores Java. No obstante, en este capítulo verás aspectos que te enamorarán desde el primer momento.

Este bloque lo hemos separado en una parte más teórica introduciendo características propias del lenguaje para que te empieces a familiarizar y otra parte muy práctica en la que podrás jugar con este lenguaje y compararlo con Java.

Hemos elegido y reescrito el proyecto del bloque 2 en Kotlin con el fin didáctico de que puedas comparar ambos proyectos y no distraerte con nuevas funcionalidades.

Es posible que no quieras volver a programar en Java, ¡al menos para Android!

Esperamos que disfrutes este libro tanto como nosotros hemos disfrutado confeccionándolo.

¡A divertirse!

CAPÍTULO 2:
Arquitectura CLEAN

La denominación "arquitectura limpia" tiene su origen en un artículo escrito por Robert C. Martin, también conocido como Uncle Bob, titulado "The Clean Architecture". Este ingeniero de software estadounidense es autor de otro libro que lleva por título *Clean Code* en el que reflexiona acerca de las buenas prácticas y el estudio de patrones a la hora de escribir software.

Una arquitectura limpia es aquella que pretende conseguir unas estructuras modulares bien separadas, de fácil lectura, limpieza del código y testabilidad.

Basándonos en el artículo de Uncle Bob, los sistemas construidos con una arquitectura limpia han de ser:

- **Independientes del framework utilizado.** Las librerías empleadas en la construcción del sistema no deben condicionarnos, han de ser una herramienta más por utilizar. En nuestro caso no querremos que el framework de Android condicione nuestra forma de desarrollar, deberá ser una herramienta más.
- **Testeables.** La lógica de negocio de nuestra aplicación ha de ser testeable indistintamente de la interfaz gráfica, modelo, base de datos o peticiones a una API empleadas.
- **Independientes de la interfaz gráfica.** Debemos usar patrones que nos permitan cambiar fácilmente la interfaz gráfica. En otras palabras, tenemos que evitar acoplar el funcionamiento de la vista con el modelo. Para ello veremos patrones como MVP, MVVM o el clásico MVC.
- **Independientes de los orígenes de datos.** Podremos sustituir nuestro origen de datos fácilmente y sin importarnos si este está disponible en una base de datos local, ficheros, una base de datos relacional o no relacional o a través de peticiones a una API. Para ello haremos uso de patrones de diseño como el patrón Repositorio.
- **Independientes de factores externos.** Debemos aislar nuestras reglas de negocio en su mundo, de tal forma que no conozcan nada ajeno a ellas.

La otra parte importante del artículo escrito por Uncle Bob corresponde a un diagrama como el de la imagen FIG. 1.1., en el que se expone la idea de la separación de responsabilidades y las diferentes capas de nuestros sistemas construidos con una arquitectura limpia.

Nuestro modelo o lógica de negocio está en el interior, en el corazón del diagrama; hacia el exterior tenemos la capa de control, transporte o comunicación entre capas y finalmente la capa correspondiente a la vista, base de datos, interfaces externas, red, etc.

Además de las diferentes capas, la regla de dependencia nos indica que existe un sentido para atravesarlas y es que las dependencias de-

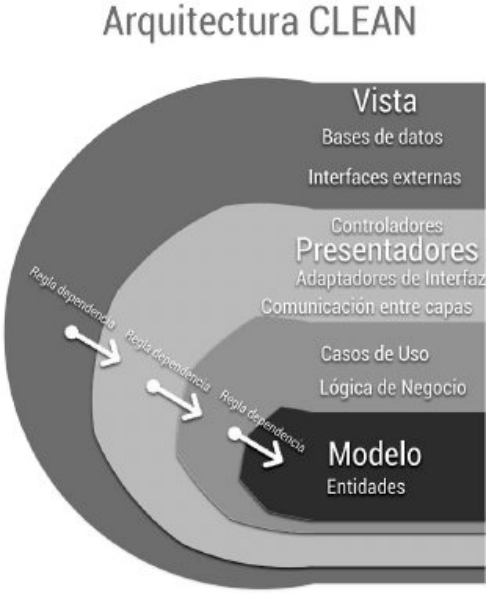


FIG. 1.1. Arquitectura CLEAN
Regla de dependencia.

ben apuntar desde el extremo de la figura hacia su corazón. Con esta separación, si comparamos dos capas, la capa interior no debe conocer nada de las capas exteriores a ella.

Nuestra lógica de negocio o casos de uso tienen que ser independientes, no deben utilizar ningún elemento software de cualquier otra capa. Veremos en siguientes capítulos cómo articulamos este punto en un pequeño proyecto Android de ejemplo, pero ya podemos adelantar que nuestro modelo debería ser solamente código Java y, por tanto, independiente de cualquier otra capa como los datos o el framework.

Los diferentes elementos que vemos en la figura son:

- **Entidades.** Son objetos de negocio de nuestra aplicación que poseen propiedades y métodos; deben ser reutilizables y poseen una estructura que varía con poca frecuencia.
- **Casos de uso.** Conocidos como "interactores", se encargan de implementar las reglas de negocio de la aplicación, orquestando el flujo de datos desde y hacia las entidades.
- **Adaptadores de interfaz.** Son adaptadores encargados de transformar los datos desde el formato más conveniente para los Casos de uso y Entidades al formato que mejor convenga a Base de datos o Interfaz de usuario. Los objetos transferidos son objetos DTO (Data Transfer Objects), son objetos simples, sin lógica de negocio, de tal forma que su formato pueda ser transformado por los adaptadores de interfaz al formato que más convenga al resto de las capas.
- **Frameworks y Drivers.** En esta capa se encuentran plataformas externas, Interfaz de usuario y Base de datos. Es la capa más externa, que debe comunicarse hacia las capas interiores.

CAPÍTULO 3:
Principios SOLID

SOLID es uno de los acrónimos más famosos en el mundo de la programación. Introducido por Robert C. Martin a principios del 2000, se compone de 5 principios de la programación orientada a objetos:

- **Principio de responsabilidad única (S):**

  Una clase debe encapsular una única funcionalidad; en caso de encapsular más de una funcionalidad, sería necesario separar la clase en múltiples clases.

  Es un principio fundamental basado en que una clase ha de hacer aquello que debe hacer y nada más. De esta forma conseguimos que la clase sea más entendible y fácil de mantener.

  En nuestro caso, al emplear casos de uso, estamos creando clases que encapsulan la lógica de negocio para una tarea concreta y nada más. Si estas clases encapsulasen más de una funcionalidad, deberíamos separar el caso en tantos casos de uso como funcionalidades diferentes debamos implementar.

- **Principio de ser abierto y cerrado (O):**

  Debemos preparar nuestro código para que esté abierto a extensiones y cerrado a modificaciones. La idea es que el código ya escrito, y que ha pasado las pruebas unitarias, esté cerrado a modificaciones, es decir, que aquello que funcione no se toque.

  Por otra parte, dado que nuestro sistema evolucionará con el tiempo, debe estar abierto a cambios, es decir, que podamos extender estas clases mediante el uso de clases abstractas, que nos permitan crear nuevos objetos que hereden de estas clases, sin alterar su funcionamiento.

  A primera vista suele ser complicado detectar qué partes del código fuente son más propensas a sufrir modificaciones. Sin embargo, las detectaremos con la práctica o cuando realmente surja la necesidad, y prepararemos esas partes para que cumplan este principio en la medida de lo posible.

  Por ejemplo, si tenemos una clase Figura, cuando estemos implementando el método areaFigura(), en lugar de utilizar un condicional para detectar el tipo de figura cuya área se calculará y seleccionar su método de cálculo, como por ejemplo, areaCuadrado(Figura) o areaCirculo(Figura), en caso de querer añadir un nuevo tipo de figura triángulo, deberíamos añadir en el condicional un nuevo tipo de figura y su respectivo método de cálculo: areaTriangulo().

```
public enum FiguraTipo{ CUADRADO,CIRCULO }

public class Figura {
    private FiguraTipo Tipo;
    public void setFiguraTipo(FiguraTipo tipo) { Tipo=tipo; }
    public FiguraTipo getFiguraTipo(){ return Tipo; }

    public double areaFigura(Figura figura)
    {
        if(getFiguraTipo().equals(FiguraTipo.CUADRADO))
         return areaCuadrado(figura);
        else if(getFiguraTipo().equals(FiguraTipo.CIRCULO))
return areaCirculo(figura);
…
   }
}
```

Para aplicar este principio deberíamos declarar una clase abstracta Figura y que todas las otras figuras que vayamos añadiendo extiendan de esta clase abstracta. De esta forma conseguimos que cada clase se encargue de calcular su propia área. Ahora simplemente invocando figura.areaFigura(), calculará su propia área, de acuerdo con su tipo.

```
public abstract class Figura
{   abstract double areaFigura();   }
public class Triangulo extends Figura
{   @Override public double areaFigura(){…}   }
public class Cuadrado extends Figura
{   @Override public double areaFigura(){…}   }
public class Circulo extends Figura
{   @Override public double areaFigura(){…}   }
```

Cabe destacar que también podríamos haber utilizado Interfaces en lugar de clases abstractas para resolverlo.

- **Principio de sustitución de Liskov (L):**

  Toda clase que extienda la funcionalidad de una clase padre base debe implementar la funcionalidad de la clase base sin alterar su comportamiento, de forma que cualquier subclase que herede de la clase padre podrá ser sustituida por otra subclase, sin afectar el comportamiento de la clase padre.

  En el caso de que alguna de las subclases hijas de una clase base no implemente una propiedad o método de la clase base, la clase hija no debería ser una subclase de la clase base, puesto que estaría violando el principio de sustitución de Liskov.

  Para aplicar correctamente este principio, definimos una clase abstracta OperacionMatematica, de tal forma que dos clases hijas extienden la funcionalidad de la clase base mediante “extend”. Como vemos ambas implementan la funcionalidad de la clase base, siendo calcular el método por sobrescribir. Por tanto, estas subclases pueden sustituirse entre sí, sin que afecte al comportamiento de la clase padre.

```
public abstract class OperacionMatematica {
    protected double numero;
    public OperacionMatematica(double numero)
    {  this.numero=numero;  }
    abstract double calcular();
}
public class OperacionMatematicaSuma extends OperacionMatematica {
    private double numeroB;
    public OperacionMatematicaSuma(double numeroA, double numeroB) {
        super(numeroA);
        this.numeroB=numeroB;
    }
    @Override double calcular() { return numero + numeroB; }
}

public class OperacionMatematicaRaizCu extends OperacionMatematica {
    public OperacionMatematicaRaizCu(double numero)

   { super(numero); }
    @Override double calcular() {  return Math.sqrt(numero);  }
}
```

Habríamos violado el principio si no hubiésemos implementado algún método o propiedad de la clase base de la cual extienden las clases hijas, sin ser estas necesarias para el correcto funcionamiento de la clase hija, simplemente dejándolas vacías o retornando un error de no implementación.

- **Principio de segregación de interfaz (I):**

  A medida que vamos construyendo nuestro sistema, nos encontramos con un problema frecuente en nuestras clases, sobre todo en las extendidas, ya que pueden contener métodos de interfaces, que no implementamos al no ser necesaria su implementación para la clase en cuestión.

  Debemos evitar, en la medida de lo posible, que ocurra esto. Es decir, tener métodos no implementados de interfaces que implementamos. Llegados a este punto, dividir la interfaz en interfaces más pequeñas es una buena práctica; de esta forma, todos los métodos que implemente una interfaz tendrán su propósito en la clase que la implemente.

- **Principio de inversión de dependencia (D):**

  Las dependencias entre clases son un problema cuando se introducen cambios en el sistema; instanciar una clase dentro de otra implica tener que conocer las clases que se instancian y de qué manera realizan su función. Como ocurre en el siguiente ejemplo, la entidad Post se corresponde con el post de un blog web con varias propiedades.

```
public class Post
{
    protected String Titulo;
    protected String Mensaje;
    protected Date Fecha;
}
```

El Blog se compone de diversos métodos, uno de los cuales es crearPost. Lo que queremos es que, cuando se escriba un post, se almacene en la base de datos local y, posteriormente, en caso de haberse almacenado con éxito, envíe una notificación por SMS a los subscriptores asociados al blog.

```
public class Blog {
    public void crearPost(Post post)
    {
        if(new AlmacenamientoLocal().guardarLocalmente(post))
         new NotificarSubscriptores().NotificarMail(post);
    }
}
public class AlmacenamientoLocal {
    public boolean guardarLocalmente(Post post) {
        //Guardar en base de datos Local el post
        return true; }
}
public class NotificarSubscriptores {
    public void NotificarMail(Post post){ //Enviar post por Email }
}
```

En este caso observamos que, en el método crearPost de la clase Blog, estamos creando una dependencia con la clase AlmacenamientoLocal y la clase Noti-

ficarSubscriptores. Además, la clase Blog conoce de qué forma se va a almacenar el post y de qué forma se va a notificar. Esto no debería ser así, ya que, en el caso de querer almacenar remotamente el post o incluso notificar mediante un email, tendríamos que modificar esta parte del código y todas donde se instancie.

Una vez que hemos visto la mala práctica de introducir dependencias, vamos a analizar lo que establece el principio de inversión de dependencias. Este principio establece que, en el dominio de nuestro sistema, debemos utilizar interfaces o abstracciones, elementos que cambian con poca frecuencia, de tal forma que sean las concreciones de menor nivel las que dependan de abstracciones o interfaces y no a la inversa.

Para cumplir este principio de inversión de dependencias, debemos modificar el ejemplo anterior, creando una interfaz IRepositorio para el almacenamiento e INotificador para el envío de notificaciones.

```
interface INotificador { void enviarNotificacion(Pos post); }
interface IRepositorio { boolean guardar(Post post); }
```

Como la forma de almacenar el post puede variar con el tiempo, lo mejor es crear clases que implementen la interfaz IRepositorio y, de esta forma, crearemos una clase para Almacenamiento Local y otra para Almacenamiento Remoto.

```
public class AlmacenamientoLocal implements IRepositorio {
  @Override public boolean guardar(Post post) { //Guardar localmente }
}
public class AlmacenamientoRemoto implements IRepositorio {
  @Override public boolean guardar(Post post) { //Guardar remotamente }
}
```

Lo mismo ocurre con la forma de notificar el post, podemos hacerlo mediante email o SMS. Debemos crear dos clases que implementen la interfaz INotificador, de tal forma que podamos añadir tantos métodos de notificación como queramos a lo largo del tiempo.

```
public class NotificarSubscriptoresSMS implements INotificador {
  @Override public void enviarNotificacion(Post post) { //Enviar SMS }
}
public class NotificarSubscriptoresMail implements INotificador {
  @Override public void enviarNotificacion(Post post) { //Enviar email }
}
```

De esta forma bastará con modificar la clase Blog, pasando por referencia los dos objetos que implementan la interfaz de IRepositorio e INotificador. La clase Blog no necesita conocer de qué forma se almacenarán los datos o de qué forma se enviará la notificación, ya que se encargarán de implementarlo los objetos pasados por referencia.

```
public class Blog {
    private IRepositorio repositorio;
    private INotificador notificador;
    public Blog(IRepositorio repositorio, INotificador notificador) {
        this.repositorio=repositorio;
        this.notificador=notificador;
    }
    public void crearPost(Post post) {
        if(repositorio.guardar(post))
            notificador.enviarNotificacion(post);
    }
}
```

Esta forma de pasar los objetos como parámetros del constructor se conoce como inyección de dependencias. Solucionan el problema de instanciar clases dentro de otras y se eliminan dependencias.

Llegados a este punto, es posible que nos preguntemos quién se encarga entonces de crear las instancias para pasarlas a estos constructores, y la respuesta es que se instancian una única vez en un lugar y la gestión de estas dependencias la realiza un framework, que, en nuestro caso de Android, será Dagger 2, aunque profundizaremos sobre este tema más adelante.

El uso de todos estos principios permite al desarrollador crear software de calidad, fácilmente escalable, entendible por todos, tolerante a test unitarios y fácil de mantener debido a la facilidad de sustitución de cada uno de los elementos que componen el sistema.

CAPÍTULO 4:
Patrones de diseño

El uso de patrones facilita la solución de problemas comunes existentes en el desarrollo de software. Los patrones de diseño tratan de resolver los problemas relacionados con la interacción entre interfaz de usuario, lógica de negocio y los datos.

Dos de los patrones más utilizados para la resolución de este tipo de problemas son los patrones MVC (Modelo Vista Controlador) y MVP (Modelo Vista Presentador).

Ambos tratan de separar la presentación de la lógica de negocio y de los datos. Dicha separación, además de facilitar el desarrollo de una aplicación, favorece su mantenimiento, ya que utiliza el principio de separación de conceptos, lo cual hace que el código sea más reutilizable. La idea principal es que cada una de las capas tenga su propia responsabilidad.

El uso de patrones permite a los desarrolladores no tener que reinventar la rueda, es decir, si ya existe una solución para un problema de diseño, utilizarla. Cuantos más patrones conozcamos, mejor sabremos en qué momento aplicarlos dependiendo del problema de diseño software con el que nos encontremos.

MVC (Modelo Vista Controlador)

El patrón de diseño MVC es uno de los más conocidos por la comunidad de desarrolladores de software. Plantea el uso de 3 capas para separar la interfaz de usuario de los datos y la lógica de negocio. Estas capas son:

- **Modelo.** Esta capa contiene el conjunto de clases que definen la estructura de datos con los que vamos a trabajar en el sistema. Además, su principal responsabilidad es el almacenamiento y persistencia de los datos de nuestra aplicación. Cabe destacar que el modelo es independiente de la representación de los datos en la vista.
- **Vista.** Esta capa contiene la interfaz de usuario de nuestra aplicación. Maneja la interacción del usuario con la interfaz de usuario para enviar peticiones al controlador. Podemos tener múltiples vistas para representar un mismo modelo de datos.
- **Controlador.** Esta capa es la intermediaria entre la Vista y el Modelo. Es capaz de responder a eventos, capturándolos por la interacción de usuarios en la interfaz, para posteriormente procesar la petición y solicitar datos o modificarlos en el modelo, retornando a la vista el modelo para representarlo en la interfaz.

Como observamos en la imagen 1.4.1.1, la secuencia de interacción entre componentes es la siguiente:

- El usuario interactúa con la vista realizando una acción, por ejemplo, pulsa un botón para indicar que quiere realizar una consulta. De este modo, la vista ejecuta una acción.
- Esta acción es atendida por el controlador, más concretamente por uno de sus métodos de acción o rutas de acceso, que procesa la petición.
- Seguidamente, el controlador solicita al modelo obtener o modificar los datos solicitados, en el modelo.
- Estos datos son atendidos por el controlador, que los devuelve a la vista seleccionada para presentarlos por pantalla.

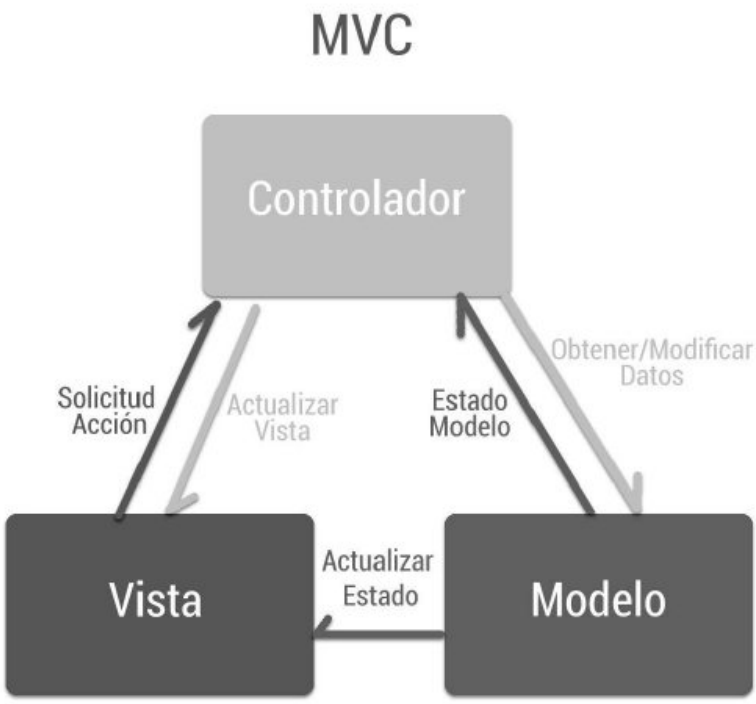


FIG. 1.4.1.1

MVP (Modelo Vista Presentador)

El patrón MVP deriva del MVC y nos permite separar aún más la vista de la lógica de negocio y de los datos. En este patrón, toda la lógica de la presentación de la interfaz reside en el Presentador, de forma que este da el formato necesario a los datos y los entrega a la vista para que esta simplemente pueda mostrarlos sin realizar ninguna lógica adicional.

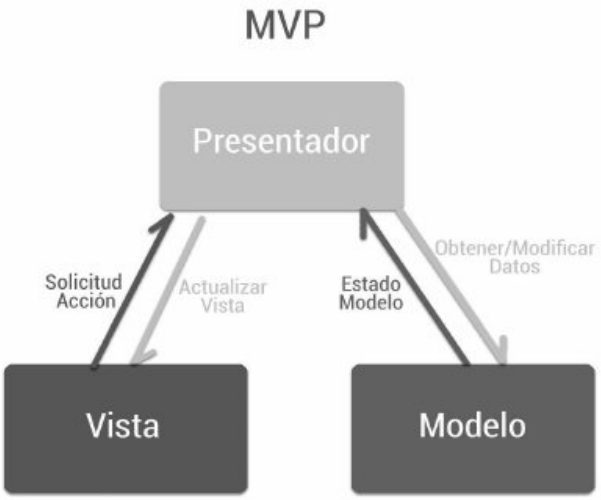


FIG. 1.4.2.1

Como observamos en la imagen 1.4.2.1, las capas que componen este patrón son:

- **Modelo.** Es la capa encargada de gestionar los datos; su principal responsabilidad es la persistencia y almacenamiento de datos. En esta capa encontramos la lógica de negocio de nuestra aplicación, utilizando los interactores para realizar peticiones al servidor con el fin de obtener o actualizar los datos y devolvérselos al presentador.
- **Vista.** La vista no es una Activity o un Fragment, simplemente es una interfaz de comportamiento de lo que podemos realizar con la vista. Sin embargo, son las Activity o Fragments los encargados de atender a la interacción del usuario por pantalla para comunicarse con el presentador. Únicamente deben implementar la interfaz de la vista, que servirá de puente de comunicación entre el presentador y las Activity o Fragments, de forma que a través de los métodos implementados representen por pantalla los datos.
- **Presentador.** Es la capa que actúa como intermediaria entre el modelo y la vista. Se encarga de enviar las acciones de la vista hacia el modelo de tal forma que, cuando el modelo procese la petición y devuelva los datos, el presentador los devolverá asimismo a la vista. El presentador no necesita conocer la vista, ya que se comunica con ella a través de una interfaz.

La forma de comunicación entre capas es posible gracias al uso de interfaces; ya que, por ejemplo, la vista conoce cuál es su presentador y puede invocar sus métodos directamente. El presentador puede comunicarse con la vista gracias a una interfaz de la vista, que se le pasa al presentador y que la vista implementa. Por otro lado, el presentador se comunica con el modelo a través del patrón Observer.

La secuencia de interacción entre componentes es la siguiente:

- El usuario interactúa con la interfaz de usuario realizando una acción, por ejemplo, pulsa un botón para indicar que quiere realizar una consulta, lo cual es capturado por la Activity o Fragment.
- La Activity recibe el evento clic y como la vista conoce cuál es su presentador, invoca al método del presentador, encargado de gestionar dicha acción.
- El presentador recibe la tarea que debe realizar por el método invocado por la vista.
- El presentador llama a un interactor del modelo para que realice la tarea, bien sea de acceso a datos remotos o locales.
- Cuando el interactor finaliza la petición de datos locales o remotos, envía los resultados al presentador, generalmente por el uso del patrón Observer.
- Tras recibir los resultados de la petición el presentador, y conociendo la interfaz de la vista, invocamos uno de los métodos de esta, responsable de la representación de los datos por pantalla, a través de uno de los métodos que implementa de la interfaz de la vista.

  Gracias a este patrón MVP, estamos separando del presentador los componentes de la interfaz Android, así como al modelo, de conocer cómo se representan los datos en la vista.

  Esta separación nos permite abstraernos del framework de Android y sus componentes, lo que nos facilita la tarea de testear cada uno de sus métodos, así como separar cada capa limpiamente.

Hasta ahora, el modelo MVP que hemos presentado delegaba la lógica de la vista en el presentador. A esto lo llamamos Vista Pasiva, ya que la vista únicamente se encarga de escuchar eventos de interacción del usuario por pantalla y de mostrar la información por pantalla.

Sin embargo, cabe mencionar que existe otro tipo de control en la lógica de la interfaz, conocido como Controlador Supervisado. Este delega en el presentador el control del evento detectado por la vista, para invocar posteriormente al modelo y que este actualice el estado, informando a la vista de este cambio del estado para que lo represente por pantalla. Este caso se parece bastante al patrón MVC.

Comparando MVC y MVP

En el patrón MVC, el controlador puede tener múltiples vistas en las que representar los datos de forma distinta, por lo que es el controlador el que decide qué vista utilizar. Por el contrario, en el MVP, cada vista tiene asignada su presentador, responsable de gestionar su lógica de representación.

Por otro lado, el modelo de MVC es un modelo sin apenas lógica, representa el estado del dominio; mientras que el modelo en MVP contiene lógica de negocio del sistema.

Otra diferencia es que el controlador de MVC contiene la lógica de negocio del sistema y sirve de intermediario entre el modelo y la vista, mientras que, por el contrario, el presentador de MVP únicamente se encarga de la lógica representación de la vista, así como de hacer de intermediario entre el modelo y la vista.

Por tanto, podríamos decir que el controlador y modelo de MVC sería el modelo de MVP, mientras que la vista de MVC englobaría el presentador y la vista de MVP. Es decir, que MVP estaría más enfocado a la vista, al dedicar dos capas a esta y su representación.

Debido a que las Activity o Fragment representan una vista, nos parece apropiado que exista un presentador responsable de la lógica de representación de la vista para separar las responsabilidades.

De no separar las responsabilidades, podríamos eliminar el presentador y dejar que la Activity o Fragment controlasen la lógica de negocio, a modo de controlador, así como de representación de la vista, todo en una misma clase. Esto conllevaría que, en cada cambio en la representación de los datos, tuviéramos que modificar el controlador y crear un modelo acoplado.

Por este motivo, ya que las Activity capturan los eventos de la interfaz de usuario, para separar las responsabilidades y abstraernos del framework de Android, nos parece interesante que la lógica de representación de la vista sea gestionada por un presentador. De modo que, si un día cambia la forma de representar los datos o de almacenarlos, simplemente modificaremos la capa responsable y nada más.

Patrón Observer

El patrón Observer se basa en dos objetos con una responsabilidad bien definida; por una parte, los objetos observables, y por otra, los observadores.

- **Observables.** Son objetos con un estado concreto, capaces de informar a los subscriptores subscritos al observable y que desean ser notificados sobre cambios de estado de estos objetos.
- **Observadores.** Son objetos que se subscriben a los objetos observables y que solicitan ser notificados cuando el estado de estos observables cambie.

  Para poder simular su funcionamiento, debemos hacer uso de interfaces. Para ello hemos construido un ejemplo sencillo de este patrón mediante un objeto observable "Animal". Queremos que dos observadores estén atentos a cambios de estado de peso y edad para notificar cuándo su edad ha alcanzado su esperanza de vida o el peso supera la media de peso para la especie, e informar sobre obesidad.

  Creamos una interfaz IObservable, que contendrá un método para subscribir los distintos observadores que deseen recibir notificaciones de cambios de estado, otro para cancelar la subscripción y dejar de recibir dichas notificaciones y, por último, un método para notificar el cambio de estado a aquellos observadores que se encuentren subscritos al objeto.

```
public interface IObservable {
 void subscribir(IObservador observador);
 void cancelarSubscripcion(IObservador observador);
 void notificarCambiosEstadoAObservadores();
}
```

Por otro lado, creamos la interfaz IObservador, que contendrá un método por el cual se notificará al observador el cambio de estado.

```
public interface IObservador {
 void actualizar(AnimalObservable animalObservable);
}
```

Una vez creadas las interfaces, crearemos un objeto observable. Queremos ser notificados ante cualquier cambio de estado en este, por ejemplo del peso y de la edad. Este objeto es "Animal" e implementa la interfaz de IObservable.

```
public class AnimalObservable implements IObservable {
 private int MAX_EDAD;
 private int MAX_PESO;
 private int edad=0;
 private int peso=0;
 ArrayList<IObservador> observadores=new ArrayList<>();
 public AnimalObservable(int edad,int peso,int MAX_EDAD,int MAX_PESO)
 {...}
 public void setEdad(int edad) {
  if(this.edad!=edad) {
  this.edad = edad;
   notificarCambiosEstadoAObservadores();
   }
 }
 public void setPeso(int peso) {
  if(this.peso!=peso) {
   this.peso = peso;
   notificarCambiosEstadoAObservadores();
  }
 }
```

Descargado en: eybooks.com

```
 @Override
 public void subscribir(IObservador observador) {
  if(!observadores.contains(observador)) observadores.add(observador);
 }
 @Override
 public void cancelarSubscripcion(IObservador observador) {
  observadores.remove(observador);
 }
 @Override
 public void notificarCambiosEstadoAObservadores() {
  for(IObservador observador : observadores)
  observador.actualizar(this);
 }
...
}
```

Como podemos observar, en el constructor, inicializamos el estado del objeto Animal. Este objeto tiene una lista de observadores, subscritos al objeto observable, que mantiene mediante los métodos subscribir() y cancelarSubscripcion(), para notificarlos posteriormente a través del método notificarCambiosEstadoAObservadores().

Cuando se añada peso o edad, en el caso de que el peso haya variado respecto a su estado inicial, se notifica a los observadores subscritos. Lo mismo ocurre cuando varíe la edad. En la notificación se recorren los observadores subscritos y se invoca a su método actualizar(), enviando el estado del objeto.

También hemos creado un observador que esté pendiente de cambios en el peso ObservadorSalud y otro que esté pendiente de cambios en la edad ObservadorVida, que implementan la interfaz de IObservador.

```
public class ObservadorVida implements IObservador {
 private IObservable observable;
 public ObservadorVida(IObservable observable)
 {
  this.observable=observable;
  observable.subscribir(this);
 }
 @Override
 public void actualizar(AnimalObservable animalObservable) {
  if(animalObservable.getEdad()>animalObservable.getMAX_EDAD())
  //El animal ha superado su esperanza de vida.
 }
}
```

En el constructor, cuando creamos el observador, le pasamos el objeto que tiene que observar, de tal forma que nos subscribimos a dicho objeto mediante el método subscribir. En el momento en que el observable cambia su estado, este invoca el método actualizar() del observador, donde se comprueba la edad del animal y su esperanza de vida, de tal forma que en caso de que su edad supere la esperanza de vida, se notifique al usuario.

Lo mismo ocurriría con un observador de peso llamado ObservadorSalud, que actuaría de igual forma que el anterior observador, pero comparando el peso ideal con el actual.

```
public class ObservadorSalud implements IObservador {
 private IObservable observable;
 public ObservadorSalud(IObservable observable)
 {
  this.observable=observable;
  observable.subscribir(this);
 }
 @Override
 public void actualizar(AnimalObservable animalObservable) {
  if(animalObservable.getPeso()>animalObservable.getMAX_PESO())
  //El animal ha superado su peso ideal.
 }
}
```

Por último quedaría su uso en la aplicación, se crearía un objeto observable de tipo Animal, en este caso un elefante, con una edad de 0, peso de 100 kg, una esperanza de vida de 60 años y 6 toneladas de peso ideal máximo. Y añadimos los dos observadores pasándole por referencia el observable, de tal forma que cuando vayamos alterando el estado del objeto, los observadores reciban estos cambios de estado para actuar en consecuencia.

```
AnimalObservable elefante =new AnimalObservable(0,100,60,6000);
IObservador observadorVida=new ObservadorVida(elefante);
IObservador observadorSalud=new ObservadorSalud(elefante);
elefante.setEdad(elefante.getEdad()+70);
elefante.setPeso(elefante.getPeso()+6500);
```

Este patrón será muy útil para utilizarlo en nuestras aplicaciones Android, ya que por ejemplo, en el caso de realizar una llamada http, podemos subscribirnos a un objeto que realice la petición, continuar operando en la aplicación y, cuando recibamos la respuesta o cambio de estado, recibir dicha información de los objetos observadores y llevar a cabo alguna acción en la interfaz de usuario.

CAPÍTULO 5:
Inyección de dependencias

Existe un problema en la programación tradicional a la que hemos estado acostumbrados. Se llama **dependencia entre objetos del dominio del sistema** y ocurre cuando una clase de alto nivel depende de otra y crea instancias a estas clases de las que depende.

Para resolver el problema de la dependencia entre objetos o acoplamiento, podemos utilizar la inversión de dependencias.

La inyección de dependencias es un patrón de diseño utilizado dentro del principio de inversión de dependencias.

¿Qué es la inversión de dependencias?

Es uno de los principios SOLID (como ya vimos en el capítulo dedicado a estos) y nos dice que nuestro código del modelo no debe depender de los detalles de implementación del framework, de cómo nos conectamos a la base de datos o a la red. Debemos apoyarnos en las abstracciones que hacemos de los objetos con los que interactuamos de capas exteriores.

¿En qué nos ayuda la inyección de dependencias?

Tal y como comentamos deberemos apoyarnos en las abstracciones del origen de datos o de las peticiones a una API a través de la red, de forma que, llegado el momento, podamos tener varias implementaciones que podamos intercambiar en el caso de que nos interese o sea estrictamente necesario por el cambio de un requisito.

¿En qué consiste?

Consiste en pasar o inyectar por referencia, al constructor de una clase, el objeto del que anteriormente dependía, en lugar de crear la instancia del objeto en el constructor de la clase. Gracias a esta inyección del objeto, en lugar de su instanciación se facilita el desacoplamiento de nuestro sistema y las tareas en la fase de pruebas, con lo que se evita una posible instanciación de un objeto, para ser construido; por ejemplo, que para ser creado requiriera acceso a la red, lo que ralentizaría todo.

De esta forma, para evitar la dependencia de una implementación de una clase, en la inyección de dependencias trabajamos con interfaces. Esto agiliza el desarrollo, sin escribir código espagueti, y permite que el código sea mantenible y reutilizable.

Algunos frameworks como Spring utilizan intensamente la inyección de dependencias para separar las abstracciones de las clases que implementan una funcionalidad. En Android tenemos diferentes alternativas, como Dagger, Guice o Dagger 2, que veremos en el siguiente apartado.

CAPÍTULO 6:
Dagger 2

Uno de los principales problemas en el desarrollo de software es la dependencia entre objetos. Esto ocurre en el momento en que una clase utiliza o depende de otro objeto para poder construirse correctamente. Para evitar esto, debemos utilizar el principio de inyección de dependencias, que consiste en la inyección del objeto del que depende, a través del constructor de la clase que necesita el objeto.

Este proceso se realiza mediante el uso de interfaces, de modo que los constructores de estas clases, a los cuales inyectamos las dependencias, incluirán un parámetro de tipo abstracto, de tal forma que implementen la misma interfaz distintos objetos dependientes.

Para entenderlo mejor pongamos, por ejemplo, el caso de que tenemos una clase "Almacenamiento" que requiere utilizar dos orígenes de datos distintos para almacenar un dato. Debería crear una interfaz abstracta, por ejemplo, "IDataBase", así como dos clases que hereden sus propiedades, por ejemplo "LocalDB" y "RemoteDB". De esta forma, en el constructor de la clase, únicamente tendríamos, por parámetro, un objeto de la interfaz "IDataBase", de modo que en el momento que quisiéramos almacenar un dato localmente, le inyectaríamos a la clase "Almacenamiento" la dependencia del objeto "LocalDB" o "RemoteDB" en caso de almacenamiento remoto.

```
public interface IDataBase { void guardar(Object objeto); }

public class LocalDB implements IDataBase {
    @Override public void guardar(Object objeto) {
        //Guardar remotamente
    }
}
public class RemoteDB implements IDataBase {
    @Override public void guardar(Object objeto) {
        //Guardar localmente
    }
}
public class Almacenamiento {
IDataBase dataBase;
    public Almacenamiento(IDataBase dataBase)
    { this.dataBase=dataBase; }

    public void almacenarInformacion(Object objeto)
    { dataBase.guardar(objeto); }
}
```

Gracias a la inyección de dependencias, no necesitamos crear nuevas instancias de objetos dentro de estas clases; de esta forma no creamos dependencias y el código resulta más limpio y organizado.

Para facilitar la tarea de la gestión de dependencias, surge Dagger 2, actualmente el framework más popular en la gestión de inyección de dependencias en Android. Su primera versión fue a cargo de Square y actualmente la mantiene Google.

Este framework se encarga de gestionar las dependencias de los objetos instancia; para ello crea un grafo de dependencias en tiempo de compilación y analiza una serie de anotaciones Java, localizadas en las cabeceras de clases o métodos. Además, Dagger 2 se compone principalmente de dos elementos: los módulos y los componentes.

Los **módulos** identifican las clases que forman parte del grafo de dependencias e indican a Dagger que dicha clase proveerá de dependencias en la aplicación que

deberán ser gestionadas por dicho framework. Dagger puede identificar que una clase es un módulo, especificándolo en una etiqueta o anotación **@Module**, ubicada sobre la definición de la clase.

Toda variable o método que provea dependencias o forme parte de la cadena de dependencia deberá definirse en los módulos. Además, todo método proveedor de dependencias que retorne un valor se encontrará definido en los módulos, que deberán incluir la anotación **@Provides** encima de la definición del método. Se recomienda, por convención, que estos métodos empiecen por la palabra *provide*, seguida del nombre del tipo de objeto retornado, para indicar que proveen de un objeto usado en la inyección de dependencias.

Por otra parte, indicamos que la instancia de un objeto obtenido en la inyección de dependencia se cree una única vez, y sea compartida por toda la aplicación, mediante la anotación **@Singleton**, ubicada en la definición del objeto.

Los **componentes**, otro elemento principal empleado por Dagger, son unas interfaces encargadas de inyectar las dependencias, referenciando los objetos Singleton a las Activity y otros elementos del ciclo de vida de la aplicación. Estos componentes utilizan la anotación **@Component**. Esta etiqueta incluye un parámetro, llamado *modules*, al que le asignamos los módulos que le corresponden al componente.

Toda Activity que utilice el componente deberá añadirse en esta interfaz componente con métodos **inject**. Estos componentes actúan como un puente entre módulos e inyección de dependencias.

A modo de ayuda hemos creado una imagen (IMG 1.6), que nos permitirá recordar rápidamente los pasos que hay que realizar en la configuración de una aplicación que utilice Dagger.

FIG. 1.6. Configuración rápida de Dagger 2.

Como resumen de las anotaciones de Dagger:

- **@Module**

Clase encargada de la gestión de las dependencias que incluye métodos que proveerán de dichas dependencias.

- **@Component**

Interfaz encargada de referenciar los objetos @Singleton a la Activity. Las Activity que utilicen el módulo deberán inyectarse en esta interfaz.

- **@Provide**

Se encarga de proveer dependencias en los módulos.

- **@Inject**

Mediante esta anotación obtenemos la dependencia.

- **@Scope**

Es el alcance que tienen nuestras dependencias, si son a nivel de la aplicación, como si fueran Singleton, o si únicamente tienen las dependencias un tiempo de vida por Activity, es decir, @Singleton a nivel local o del ciclo de vida de la Activity.

- **@Singleton**

La instancia de un objeto será creada únicamente la primera vez en la aplicación, después se reutilizará.

CAPÍTULO 7:
Arquitectura del proyecto Android

En el desarrollo de un proyecto, elegimos una arquitectura basada en CLEAN y dividimos el proyecto en tres capas bien diferenciadas, cada una de las cuales se corresponde con un módulo de un proyecto Android. Estas capas son:

- **Módulo Data**

  Capa en la que definimos el acceso a datos, bien sean de una API REST remota con la cual nos comuniquemos, la base de datos local o simplemente las preferencias de usuario del sistema, con el objetivo de obtener o almacenar información.

- **Módulo Domain**

  Es la capa de dominio de nuestro proyecto, en la que definimos nuestros modelos de negocio, así como los casos de uso de los que se compone nuestro proyecto. Definiremos una serie de conversores o *parsers*, capaces de convertir los datos al formato que más convenga entre la capa Data y la capa Presentación o viceversa.

  Esta conversión es necesaria, debido a que en la capa Presentación solo interesan datos en un formato que facilite su representación en la vista, mientras que esta misma representación no será la misma que la que utiliza la capa Data para enviar datos u obtenerlos local o remotamente.

  Hemos incluido el modelo del patrón MVP en esta capa, la de dominio, para separar la responsabilidad de cada una de las capas adecuadamente.

- **Módulo Presentation**

  Capa de presentación en la que se encuentran la interfaz de usuario, las vistas y los presentadores. En el caso de Dagger, debido al tratamiento de inyección de dependencias, dispondremos de una serie de módulos y componentes definidos por nosotros.

CAPÍTULO 8:
Test unitarios

Una de las tendencias de la programación es usar **TDD** (**Test Driven Development**) y diseñar los test que han de superarse antes de pasar al desarrollo de las clases que implementarán la aplicación.

Disponer de un buen conjunto de test no solamente nos ayudará a verificar que no dejamos nada al azar y que todo está controlado y supervisado, sino que también nos permitirá comprobar en el futuro que, cada vez que hagamos un cambio o añadamos una nueva funcionalidad, el resto de las funcionalidades se siguen cubriendo gracias a la posibilidad de ejecutar nuestros test tantas veces como sea necesario.

En Android, disponemos básicamente de 2 tipos de pruebas:

Pruebas de unidad local	Estas pruebas se ejecutan en la propia máquina virtual de nuestra máquina local.
Pruebas instrumentadas	Pruebas que se ejecutan en dispositivo o emulador de hardware. Dichas pruebas se compilan en un archivo *.apk* independiente.

A continuación, introduciremos algunos conceptos sobre **JUnit** y **Mockito**, ya que son unas de las herramientas más utilizadas en el apartado de los test unitarios.

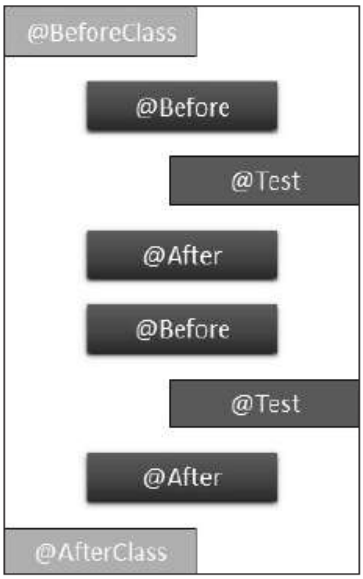
JUnit

JUnit utiliza anotaciones y métodos **Assert** en sus clases de test para configurar las pruebas. Las anotaciones más importantes son las siguientes:

Anotación	Comentario
@BeforeClass	Se ejecuta antes de que todos los métodos test de todas las clases puedan ser ejecutados. Usado para inicializaciones comunes a todos los test como, por ejemplo, establecer conexiones con bases de datos, inicializar clases, etc.
@AfterClass	Se ejecuta después de que todos los métodos test de todas las clases puedan ser ejecutados. Usado para acciones comunes a todos los test, como cerrar la conexión a base de datos.
@Before	Se ejecuta antes de cada método de test y sirve para preparar el entorno.
@After	Se ejecuta después de cada método de test y se usa para limpiar el entorno (datos temporales, estructuras de memoria, etc.).

@Test	Identifica que el método es un test.
@Test(expected=Exception.class)	Espera que el método lance la excepción indicada. Si no se lanza la excepción, el método falla.
@Test(timeout=<tiempo>)	Falla si el método tarda más del tiempo indicado en milisegundos.
@Ignore	Permite ignorar el test que viene a continuación. Es útil cuando el test todavía no está a punto o ha quedado obsoleto fruto de alguna modificación reciente.

El orden de ejecución es el siguiente:



Los métodos de **Assert** permiten confirmar si dos valores son iguales o no. En general compara el resultado del método con un valor previamente esperado.

Los métodos **Assert** son los siguientes:

Método	**Comentario**
assertEquals	Afirma que dos valores son iguales.
assertFalse	Afirma que una condición es falsa.
assertNotNull	Afirma que un objeto no es nulo.
assertNotSame	Afirma que dos objetos no se refieren al mismo objeto.

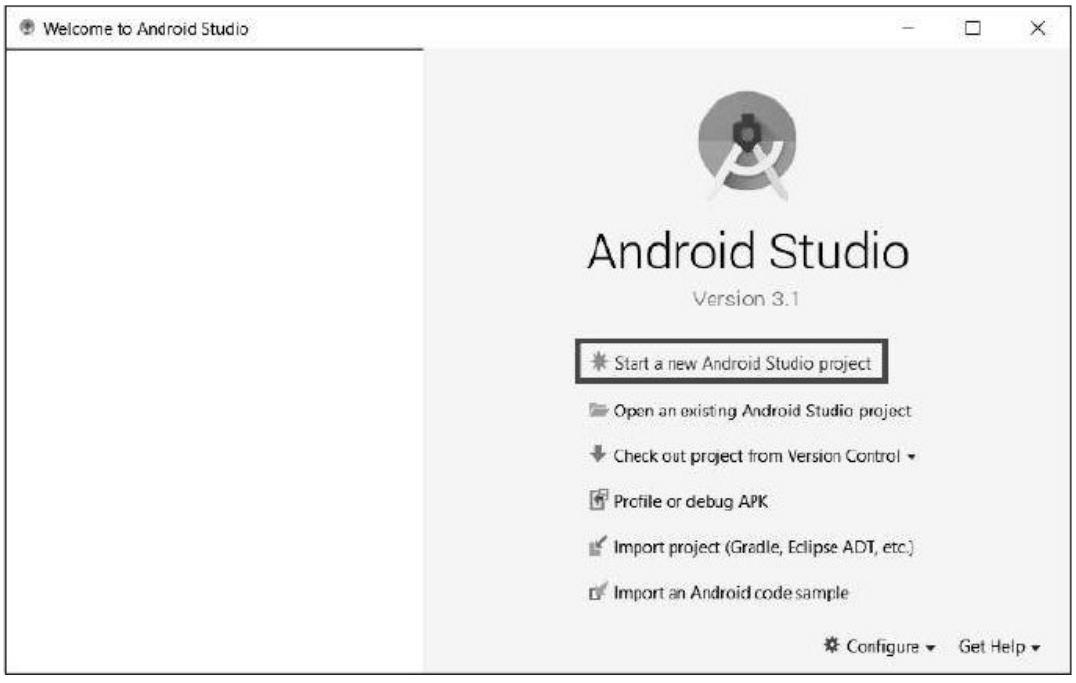
assertNull	Afirma que un objeto es nulo.
assertSame	Afirma que dos objetos se refieren al mismo objeto.
assertTrue	Afirma que una condición es verdadera.
fail()	Falla sin mensaje.
fail(mensaje)	Falla con el mensaje indicado. Por ejemplo, es útil para indicar que el método está pendiente de implementar.

A continuación, veremos un ejemplo de cómo implementar algunos test en una aplicación Android.

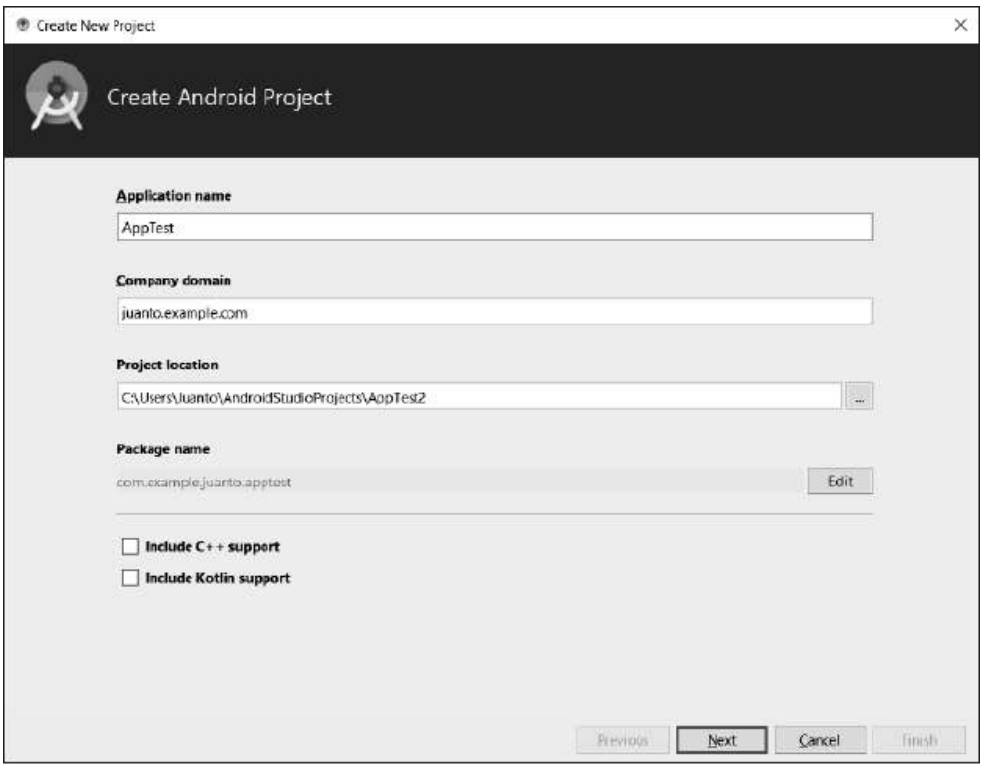
Aplicación de ejemplo

Vamos a generar la aplicación **AppTest** para crear una clase llamada **Calculadora** con unos cuantos métodos correspondientes a las operaciones básicas de **suma**, **resta**, **multiplicación** y **división**.

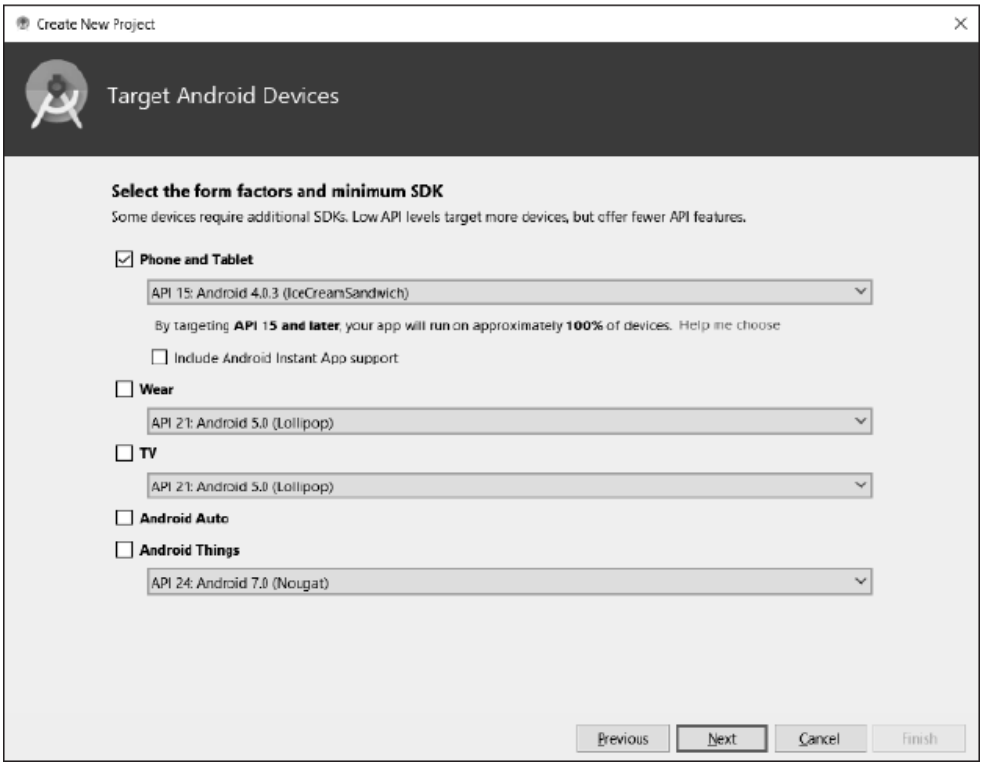
Para ello, arrancaremos Android Studio y seleccionaremos la opción **Start a new Android Studio project**:



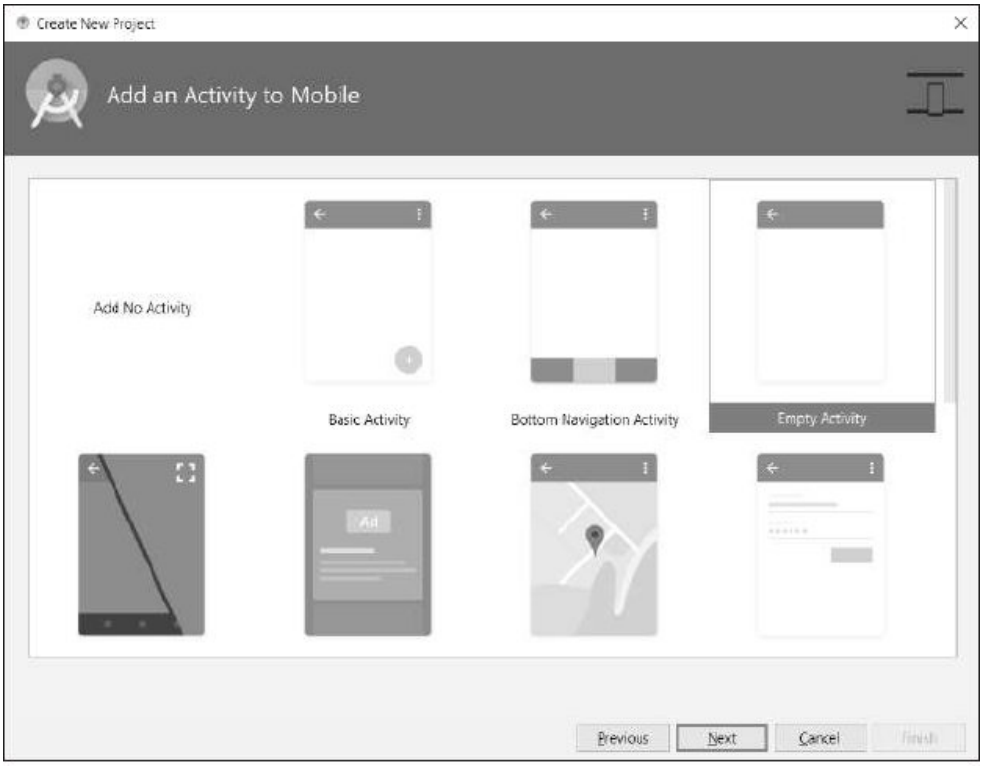
En el siguiente cuadro de diálogo, indicaremos **AppTest** en el campo **Application name**:



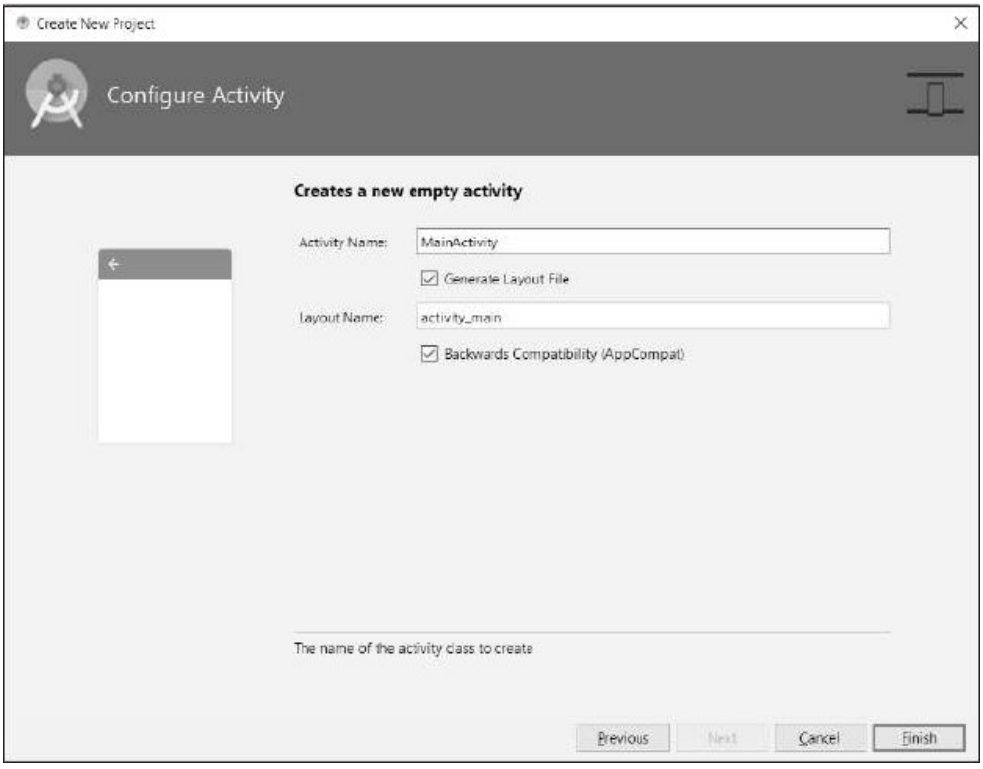
Pulsaremos sobre **Next** para pasar a la siguiente pantalla:



Podemos aceptar los valores propuestos y pulsar sobre **Next**.

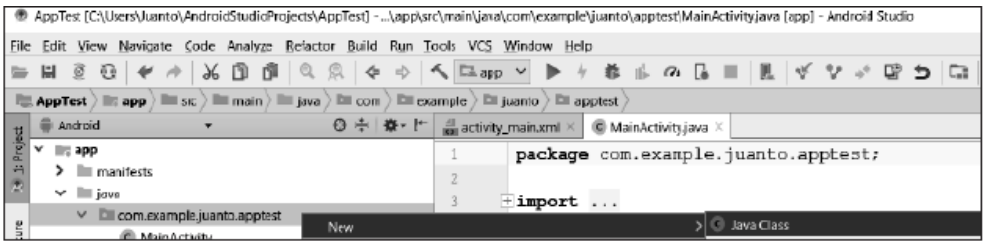


En esta pantalla seleccionaremos **Empty Activity** y pulsaremos **Next**.

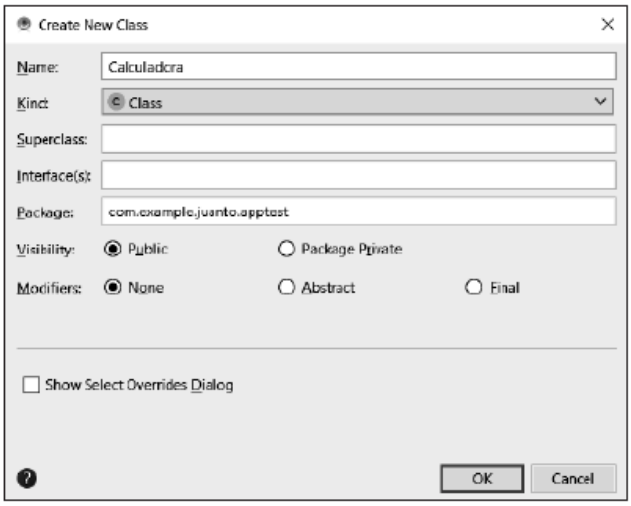


Por último, aceptaremos lo valores propuestos y pulsaremos sobre **Finish**.

Una vez que **Android Studio** nos muestre la aplicación, crearemos la clase **Calculadora** haciendo clic con el botón derecho del ratón sobre el nodo **app/com.example.juanto.apptest** y seleccionando **New/Java Class**:



En el siguiente cuadro de diálogo, escribiremos **Calculadora** en el campo **Name** y pulsaremos sobre **OK**:



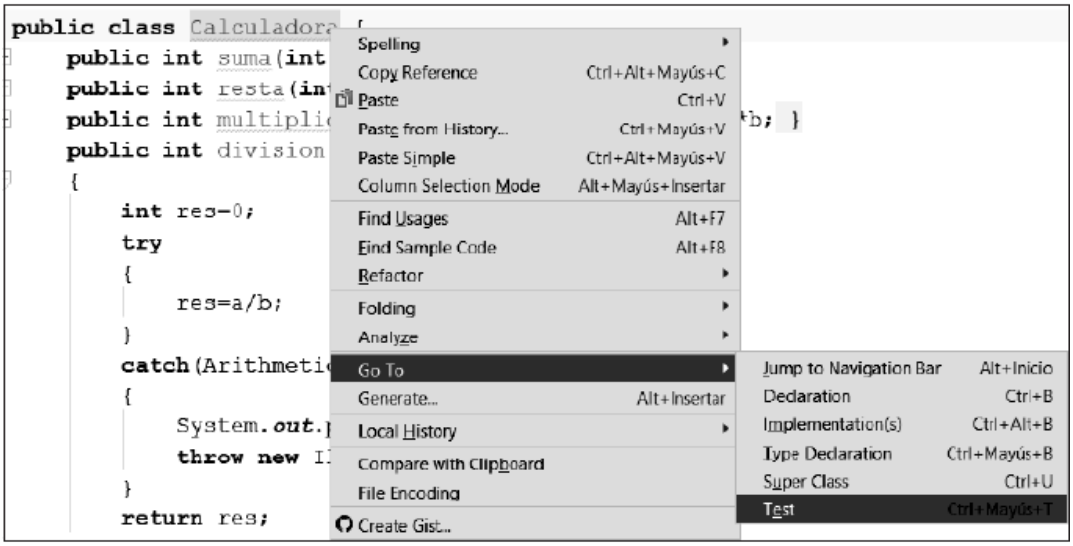
Observaremos que se abre el fichero en el editor y nos muestra el código para que empecemos a editarlo.En la definición de la clase introduciremos lo siguiente:

```
public class Calculadora {
    public int suma(int a,int b)
    {
        return a+b;
    }
    public int resta(int a,int b)
    {
        return a-b;
    }
    public int multiplicacion(int a,int b)
    {
        return a*b;
    }
    public int division(int a,int b)
    {
        int res=0;
        try
        {
            res=a/b;
        }
```

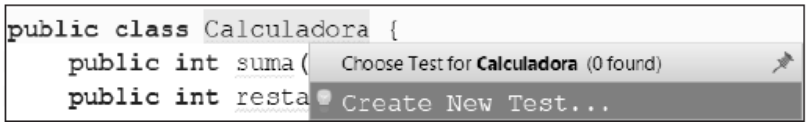
```
            catch(ArithmeticException ex)
            {
                System.out.println(" Error " + ex.getMessage());
                throw new IllegalArgumentException("Argumento 'divisor
                ' es 0");
            }
            return res;
        }
    }
```

Test con JUnit

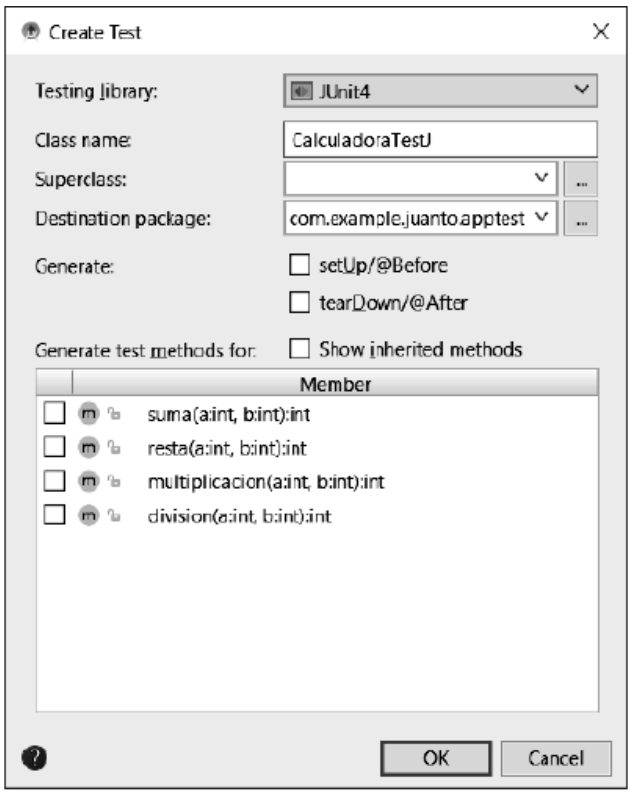
A continuación, crearemos un test para esta clase. Existen diferentes formas de crear dicho test pero, en este caso, dentro del fichero **Calculadora.java**, simplemente haremos clic con el botón derecho sobre el nombre de la clase **Calculadora** y seleccionaremos **Go To/Test**:



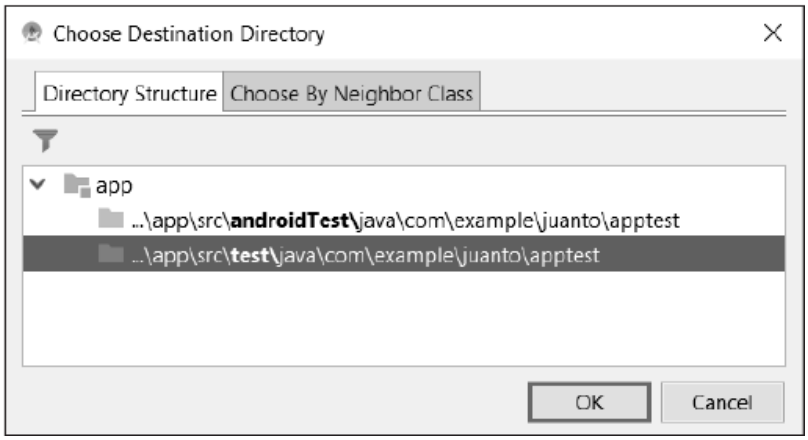
Aparecerá un menú flotante en el que seleccionaremos la opción **Create New Test...**



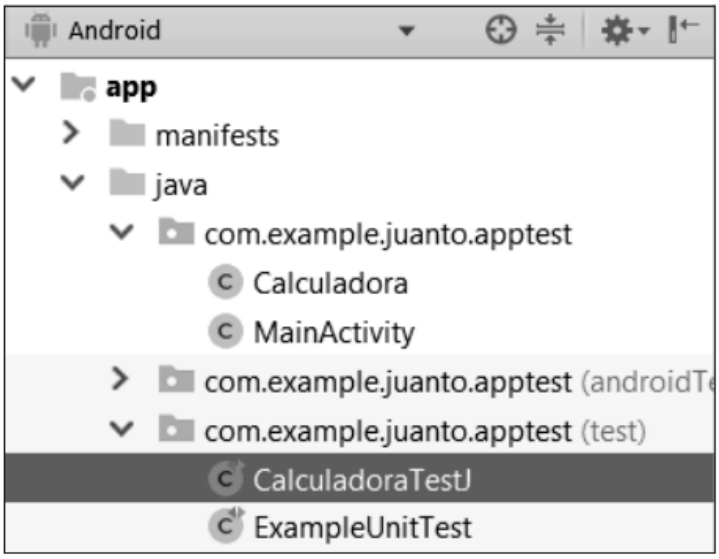
A continuación, aparecerá el siguiente cuadro de diálogo, en el que solo introduciremos el nombre **CalculadoraTestJ** en el campo **Class name**:



Pulsamos sobre **OK** y, seguidamente, en el siguiente cuadro de diálogo, elegiremos la opción **...\app\src\test\java\com\example\juanto\apptest** y pulsaremos sobre **OK**:



Observaremos que se ha creado una nueva clase dentro de la carpeta **com.example.juanto.apptest**:



En el editor, observaremos que también se muestra el código de dicha clase, y modificaremos su definición con el fin de incluir diversos test para probar los métodos incluidos en la clase **Calculadora** de la siguiente manera:

```
import org.junit.Assert;
import org.junit.BeforeClass;
import org.junit.Test;

import static junit.framework.Assert.fail;
import static org.junit.Assert.*;

/**
 * Created by Juanto on 18/03/2018.
 */
public class CalculadoraTestJ {
    private static Calculadora calc;

    @BeforeClass
    public static void setUp() throws Exception {
        calc = new Calculadora();
    }

    @Test
    public void sumaJ() throws Exception {
        int res = calc.suma(2, 2);
        int resEsperado = 4;
        Assert.assertEquals(resEsperado, res);
    }

    @Test
    public void restaJ() throws Exception {
        int res = calc.resta(2, 4);
        int resEsperado = -2;
        Assert.assertEquals(resEsperado, res);
    }

    @Test
    public void multiplicacionJ() throws Exception {
        int res = calc.multiplicacion(3, 5);
        int resEsperado = 15;
        Assert.assertEquals(resEsperado, res);
```

```
    }

    @Test
    public void divisionJ() throws Exception {
        int res = calc.division(10, 2);
        int resEsperado = 5;
        Assert.assertEquals(resEsperado, res);
    }

    @Test
    public void sumaJNotEqual() throws Exception {
        int res = calc.suma(2, 2);
        int resEsperado = 5;
        Assert.assertNotEquals(resEsperado, res);
    }

    @Test
    public void sumaJTrue() throws Exception {
        int res = calc.suma(2, 2);
        int resEsperado = 4;
        Assert.assertTrue(resEsperado == res);
    }

    @Test
    public void testQueFallara() throws Exception {
        fail("Falla porque est  pendiente de implementar");
    }
}
```

Podemos observar que, en la anotación **@BeforeClass**, el método **setUp()** nos permite inicializar un objeto de tipo **Calculadora**.

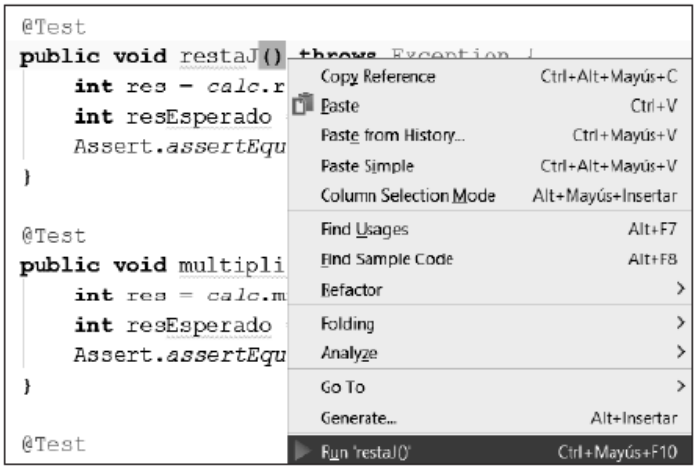
Seguidamente, añadimos un bloque de tipo **@Test** por cada una de las operaciones que deseamos probar, que en definitiva son las existentes en la clase **Calculadora** (**suma**, **resta**, **multiplicación** y **división**).

Cada bloque de cada operación básica contiene un **Assert** (**assertEquals**) que permite comparar el resultado de realizar una operación con la calculadora con un resultado esperado que definimos de antemano. Si ambos resultados coinciden, **Assert** devuelve *true* y el test se muestra como superado.

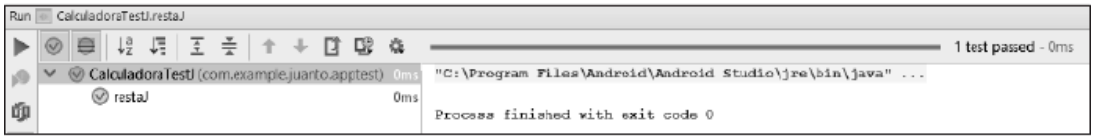
Hemos querido incluir también un ejemplo de **Assert** de tipo **assertNotEquals** y **assertTrue** para ver su funcionamiento. Por último, hemos incluido un ejemplo de **fail()** que simplemente nos permite lanzar una excepción y que puede servirnos para cuando tenemos algún test pendiente de implementar y queremos hacerlo notar durante su ejecución. Si quisiéramos evitar su ejecución, bastaría con añadir la anotación **@Ignore** justo antes de **@Test** para que no nos diga que el test falla:

```
@Ignore
@Test
public void testQueFallara() throws Exception {
    fail("Falla porque est  pendiente de implementar");
}
```

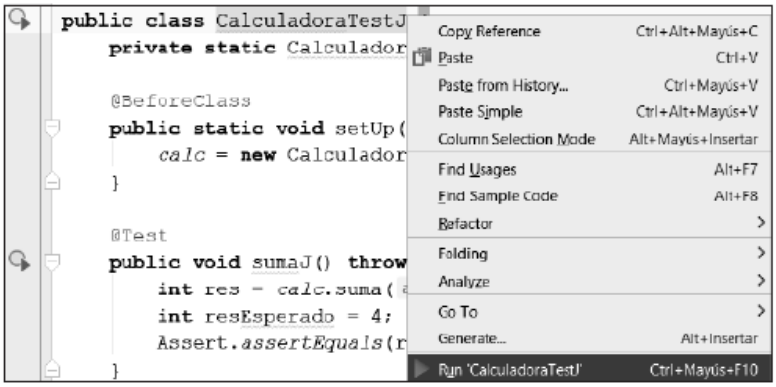
En este punto, podemos ejecutar los test por separado simplemente haciendo clic con el botón derecho del ratón sobre cualquier parte del bloque correspondiente al test en cuestión y seleccionando del menú flotante la opción **Run 'nombreTest'**. Por ejemplo, para probar el test **RestaJ()** veremos lo siguiente:



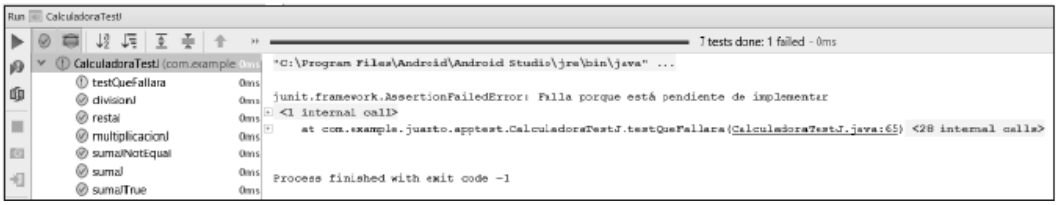
Al ejecutar este test en solitario, si todo va bien, veremos el siguiente resultado:



De forma similar, para ejecutar todos los test de la clase, podemos hacer clic sobre el nombre de la clase y, con el botón derecho del ratón, seleccionar **Run 'CalculadoraTestJ'**:

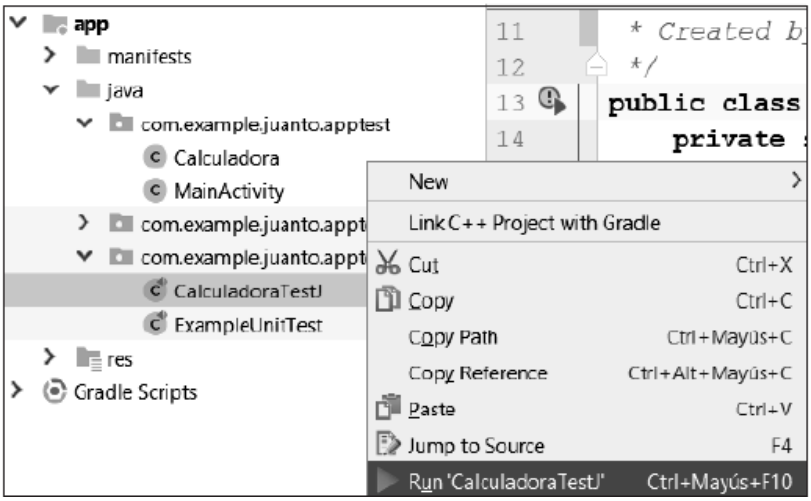


El resultado de los test se mostrará de la siguiente manera:



Vemos que todos los test se han realizado satisfactoriamente y que uno de ellos ha sido ignorado como esperábamos:

Otra posible forma de ejecutar todos los test sería localizar la clase **CalculadoraTestJ** dentro del nodo **com.example.juanto.apptest** y, con el botón derecho del ratón, localizar la opción **Run 'CalculadoraTestJ'**:



Test con Mockito

Para introducir algunos conceptos relacionados con **Mockito**, diremos que este es un framework de **Mocks** open source que permite crear objetos **mocks** y espías de forma sencilla y potente. Es decir, permite crear y configurar objetos simulados y probar la funcionalidad de una clase de forma aislada.

Empezaremos comentando que los **mock objects** son objetos que simulan parte del comportamiento de una clase y que contienen los métodos del objeto original pero que, en realidad, son falsos; al no contener ninguna implementación es imprescindible definir el comportamiento de los métodos que referenciemos.

Mockito permite generar **mock objects dinámicos** y está basado en **EasyMock**, lo que facilita la creación de mocks de clases concretas (y no solo de interfaces).

La idea de las pruebas, más que analizar los resultados obtenidos por los métodos por probar, es ejecutar y verificar las llamadas e interacciones de las clases por probar y las clases de apoyo.

Con **Mock**, disponemos también de diferentes anotaciones como son:

Anotación	Comentario
@Mock	Crea un mock de un determinado objeto sin necesidad de llamar a **Mockito.mock** manualmente.
@Spy	Permite crear un **wrapper** alrededor de una instancia real (no de un objeto simulado) contra la que podrá interactuar. Puede funcionar como un objeto real, pero también podemos simular el comportamiento de cualquier método.
@Captor	Se usa para capturar valores de argumento para otros assert.
@InjectMocks	Permite indicar a **Mockito** qué clase se desea inyectar.

Respecto a los métodos, tenemos:

Método	Comentario
when	Permite decidir cómo debe comportarse un mock y qué resultado ha de devolver dicho mock para un determinado método.
doAnswer	Cuando queremos que el método haga algo.
doThrow	Cuando se desea lanzar una excepción al llamar a un método de objeto.
verify	Permite verificar que se han llamado a ciertos métodos durante la ejecución con algunas condiciones determinadas.
ArgumentCaptor	Permite capturar argumentos pasados a los métodos.
doReturn	Cuando queremos devolver un valor específico al llamar a un método de objeto.
doCallRealMethod	Cuando queremos que realmente se ejecute el método (y cada una de sus líneas).
doNothing	Cuando queremos omitir la ejecución de un método y, por tanto, de su posible complejidad, que no afecta a la ejecución de la prueba.

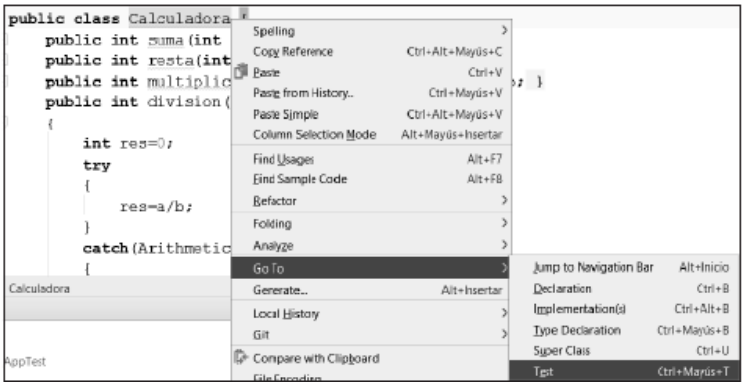
Con **when**, podemos usar algunos de los siguientes métodos:

Método	Comentario
thenReturn(returnValue)	Permite especificar el valor por devolver cuando se llame al método asociado.
thenThrow(exception)	Permite especificar la excepción por lanzar cuando es invocado un método.
thenCallRealMethod()	Permite llamar al método real.
thenAnswer()	Permite simular el comportamiento de métodos nulos.

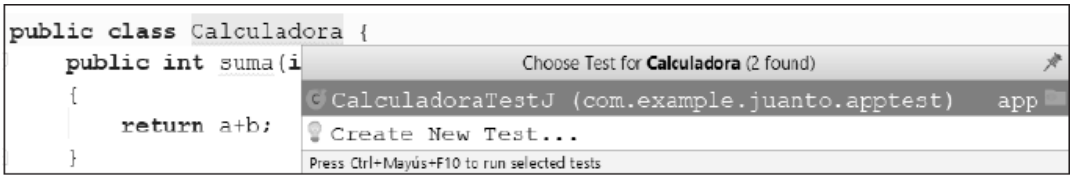
Con **verify**, podemos determinar si un método ha sido llamado alguna vez, un número de veces o ninguna:

Sintaxis	Comentario
times(int wantedNumberOfInvocations)	Número de llamadas realizadas.
atLeast(int wantedNumberOfInvocations)	Número de llamadas realizadas como mínimo.
atMost(int wantedNumberOfInvocations)	Número de llamadas realizadas como máximo.
only(int wantedNumberOfInvocations)	Solo se ha llamado a ese método en este mock.
atLeastOnce()	Ha sido llamado al menos una vez.
never()	Nunca ha sido llamado.

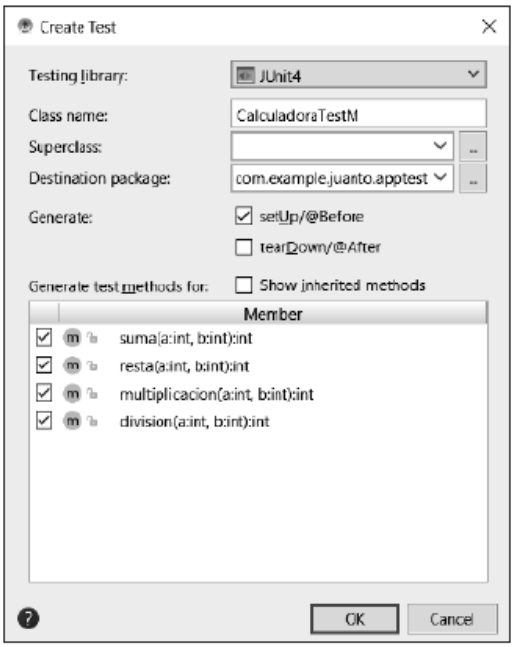
A continuación, crearemos un nuevo test para utilizar algunos de los métodos de **Mockito**. Para ello, de nuevo nos ubicaremos en la clase **Calculadora.java** y, tal y como explicamos para el ejemplo de **JUnit**, simplemente haremos clic con el botón derecho sobre el nombre de la clase **Calculadora** y seleccionaremos **Go To Test**:



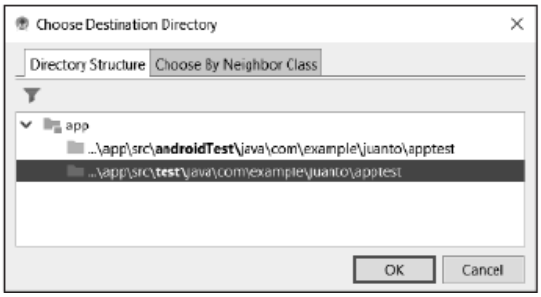
En esta ocasión, nos mostrará el test que hemos creado en el ejemplo anterior, y de nuevo seleccionaremos **Create New Test...**



En esta ocasión, pondremos **CalculadoraTestM** como nombre de **Class name** y marcaremos los casilleros **setUp/@Before** y todos los casilleros de los métodos **suma**, **resta**, **multiplicación** y **división**:



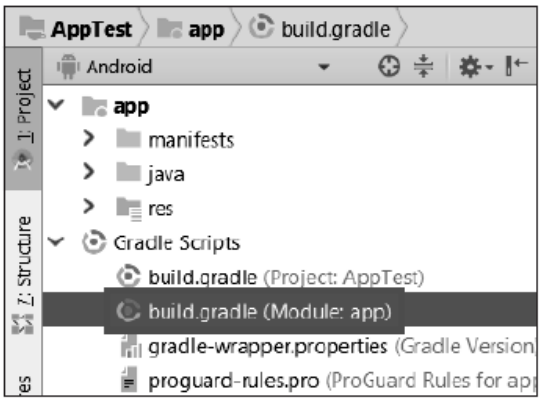
Pulsaremos sobre **OK** y, a continuación, seleccionaremos **...\app\src\test\...**:



Veremos que nos ha creado una nueva clase con el método **setUp()** y un método vacío por cada operación por probar:

```
public class CalculadoraTestM {
    @Before
    public void setUp() throws Exception {
    }
    @Test
    public void suma() throws Exception {
    }
    @Test
    public void resta() throws Exception {
    }
    @Test
    public void multiplicacion() throws Exception {
    }
    @Test
    public void division() throws Exception {
    }
}
```

Antes de continuar definiendo los métodos, modificaremos nuestro archivo **app** para añadir una nueva dependencia que nos permita trabajar con **Mockito**. Para ello, desplegaremos el nodo **Gradle Scripts** y localizaremos el archivo **build.gradle (Module:app)**:



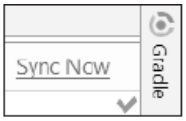
Una vez abierto en el editor, añadiremos en el bloque **dependencias** la última línea correspondiente a **Mockito** de la siguiente manera:

```
testImplementation 'org.mockito:mockito-all:1.10.19'
```

Quedará así:

```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation 'com.android.support:appcompat-v7:26.1.0'
    implementation 'com.android.support.constraint:constraint-layout:1.0.2'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'com.android.support.test:runner:1.0.1'
    androidTestImplementation 'com.android.support.test.espresso:espresso-core:3.0.1'
    testImplementation 'org.mockito:mockito-all:1.10.19'
}
```

A continuación, pulsaremos sobre **Sync Now** para sincronizar los archivos del proyecto:



Seguidamente, modificaremos el contenido de la clase para rellenar los métodos y cambiar los nombres de estos de la siguiente manera:

```
@RunWith(MockitoJUnitRunner.class)
public class CalculadoraTestM {
    private Calculadora calc;

    @Before
    public void setUp() throws Exception {
        calc = mock(Calculadora.class);
        when(calc.suma(2, 2)).thenReturn(4);
        when(calc.resta(2, 4)).thenReturn(-2);
        when(calc.multiplicacion(3, 5)).thenReturn(15);
        when(calc.division(10, 2)).thenReturn(5);
        when(calc.division(10, 0)).thenThrow(ArithmeticException.class);
    }

    @Ignore
    @Test(expected = ArithmeticException.class)
    public void divisionMEx2() {
        int res = calc.division(10, 0);
    }

    @Test
    public void sumaM() throws Exception {
        calc.suma(2, 2);
        verify(calc).suma(2, 2);
    }

    @Test
    public void restaM() throws Exception {
        calc.resta(2, 4);
        verify(calc).resta(2, 4);
    }

    @Test
    public void multiplicacionM() throws Exception {
        calc.multiplicacion(3, 5);
        verify(calc).multiplicacion(3, 5);
    }

    @Test
    public void divisionM() throws Exception {
        calc.division(10, 2);
        verify(calc).division(10, 2);
    }

    @Test(expected = ArithmeticException.class)
    public void divisionMEx() {
        int res = calc.division(10, 0);
    }

    @Test
    public void sumaMatLeast() throws Exception {
        calc.suma(2, 2);
        calc.suma(2, 4);
```

Descargado en: eybooks.com

```
        verify(calc, atLeast(2)).suma(anyInt(), anyInt());
    }

    @Test
    public void sumaMnever() throws Exception {
        verify(calc, never()).suma(2, 6);
    }
}
```

Observa que, en la línea anterior a la definición de la clase, hemos añadido la siguiente instrucción para indicar que el lanzador que queremos usar es **MockitoJUnitRunner** en lugar del lanzador por defecto **BlockJUnit4ClassRunner**:

```
@RunWith(MockitoJUnitRunner.class)
```

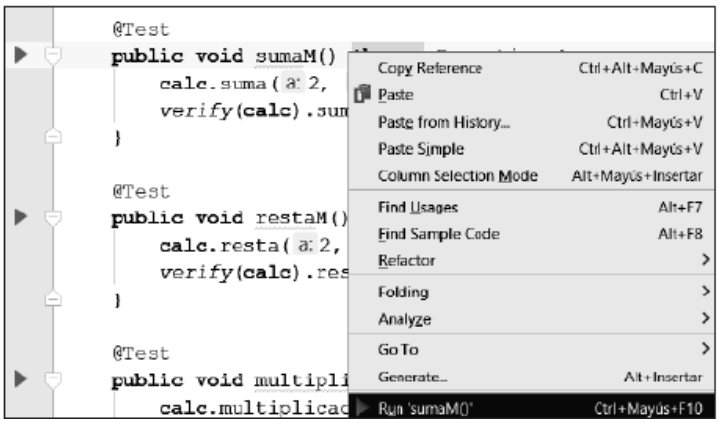
Debemos tener en cuenta que hemos tenido que incluir los siguientes **imports**:

```
import org.junit.Before;
import org.junit.Ignore;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.runners.MockitoJUnitRunner;
import static org.mockito.Matchers.anyInt;
import static org.mockito.Mockito.atLeast;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.never;
import static org.mockito.Mockito.verify;
import static org.mockito.Mockito.when;
```

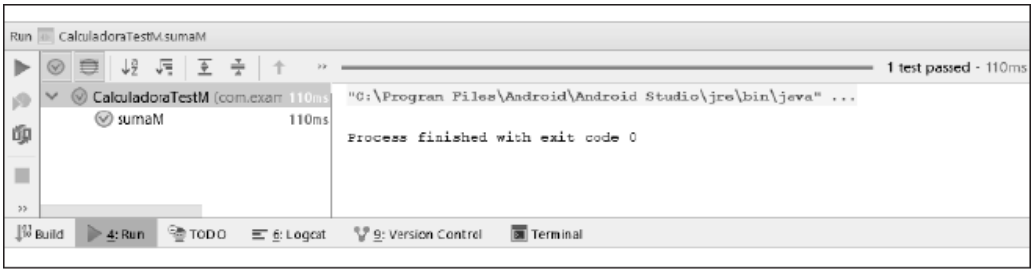
Observa también que, en el método **setUp()**, hemos inicializado el **mock** para el objeto **calc** y definido también los valores por devolver (mediante **when** y **thenReturn**) cuando se invoquen a los métodos de nuestra clase **Calculadora**.

Al igual que vimos en el ejemplo anterior, podemos lanzar los test de forma individual o bien todos juntos. Recuerda que con la anotación **@Ignored** podemos obviar la ejecución del test que viene a continuación.

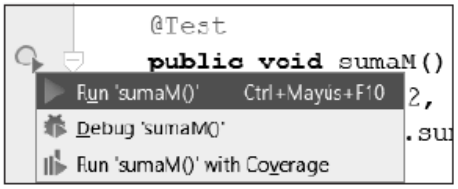
Para lanzar un test de forma individual, podemos hacer clic dentro del bloque que contiene el método y pulsar la combinación de teclas **Ctrl+Mayús+F10** o bien, con el botón derecho del ratón, seleccionar del menú flotante la opción **Run 'nombre del método'**. Por ejemplo, para ejecutar el test **sumaM**(), haremos lo siguiente:



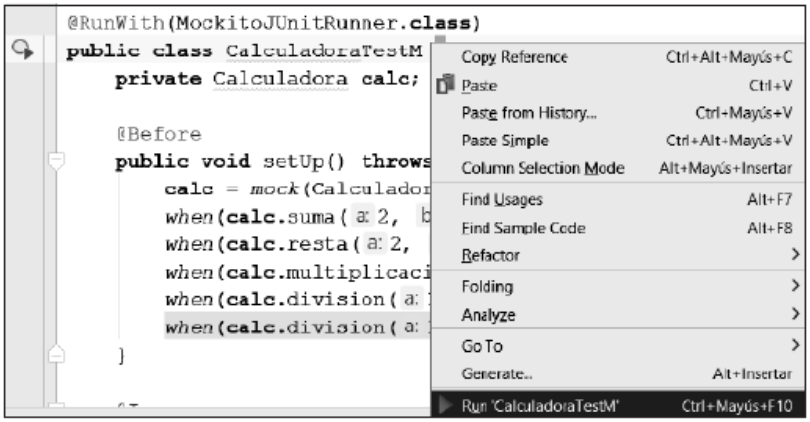
El resultado de este lo veremos en la vista **Run** tal y como mostramos a continuación:



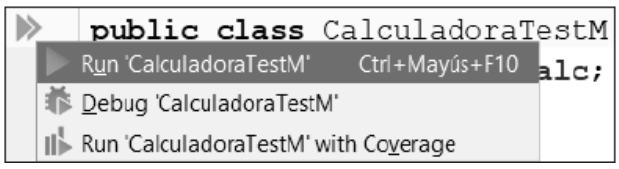
También podríamos lanzarlo pulsando sobre el circulito que se muestra en la parte izquierda de la declaración del método y seleccionando la opción **Run 'sumaM()'**:



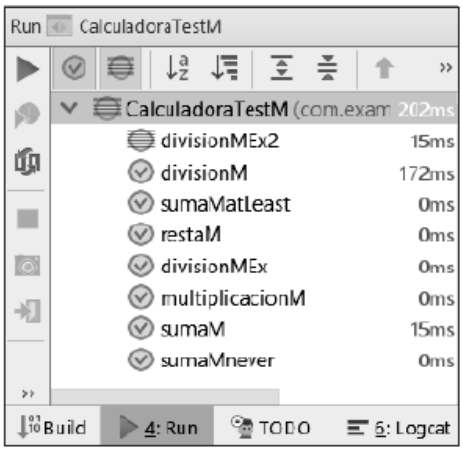
Para ejecutar todos los test de la clase **CalculadoraTestM**, podemos hacer clic con el botón derecho del ratón sobre la línea en la que se halla la sentencia **public class CalculadoraTestM** y, a continuación, ejecutar **Run 'CalculadoraTestM'**:



También podemos ejecutar todos los test pulsando sobre la doble flecha ubicada en la parte izquierda del código y seleccionando la opción **Run 'CalculadoraTestM'**:



El resultado será el siguiente:

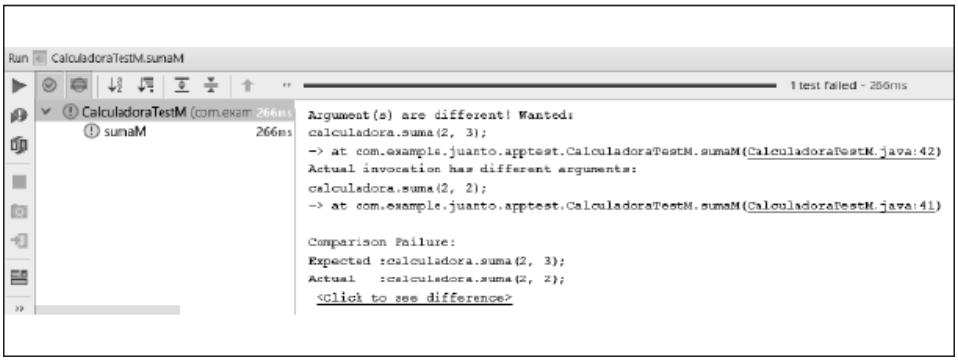


Podemos provocar un error para que uno de los test falle y así poder comprobar la salida prevista.

Por ejemplo, vamos a modificar el método **sumaM()** para que verifique la suma de los números **2** y **3**. En el editor veremos lo siguiente:

```
@Test
public void sumaM() throws Exception {
    calc.suma( a: 2,  b: 2);
    verify(calc).suma( a: 2,  b: 3);
}
```

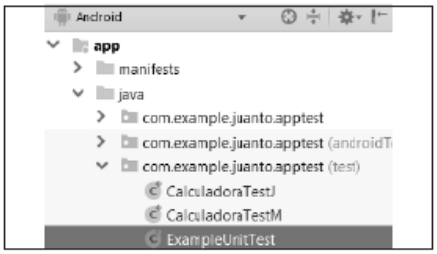
Si ejecutamos de nuevo el test **sumaM()**, veremos el siguiente resultado:



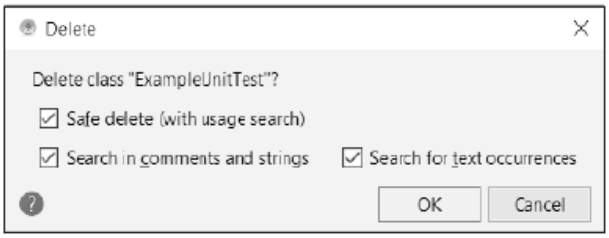
Vemos que el test ha fallado porque esperábamos (**Expected**) la ejecución del método **suma** con (2, 3) y realmente (**Actual**) se ha ejecutado la llamada con (2, 2). Observa que el icono asociado al método cambia y muestra que el resultado de la última ejecución falló:

Restablecemos el valor original para que el test no falle y pasamos a ver cómo podemos ejecutar todos los test previstos en la app.

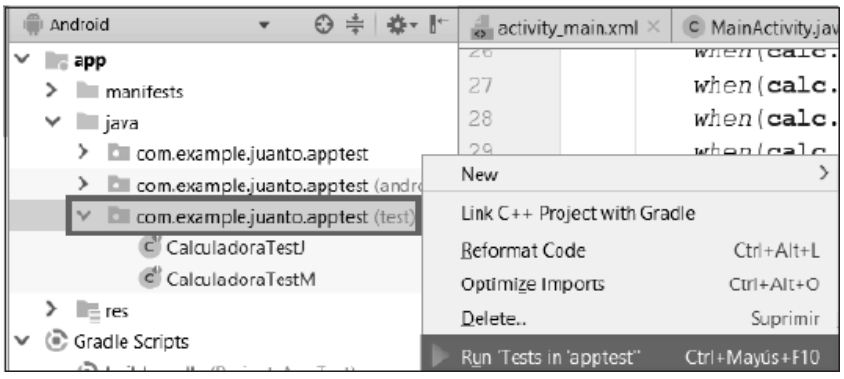
Podemos eliminar el test llamado **ExampleUnitTest** que se creó por defecto al crear la app y que veremos ubicado en el nodo **com.example.juanto.apptest**:



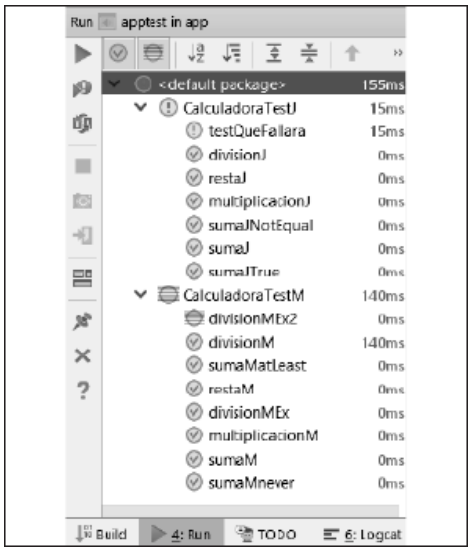
Para ello, simplemente seleccionaremos el test y pulsaremos sobre la tecla **Supr** para que aparezca el siguiente cuadro de diálogo, donde simplemente pulsaremos sobre **OK**:



En este punto, observaremos que el test de ejemplo ya ha desaparecido. Para ejecutar todos los test, simplemente seleccionaremos el nodo **com.example.juanto.apptest** y, con el botón derecho del ratón, seleccionaremos **Run 'Test in 'apptest''**:



El resultado será el siguiente:



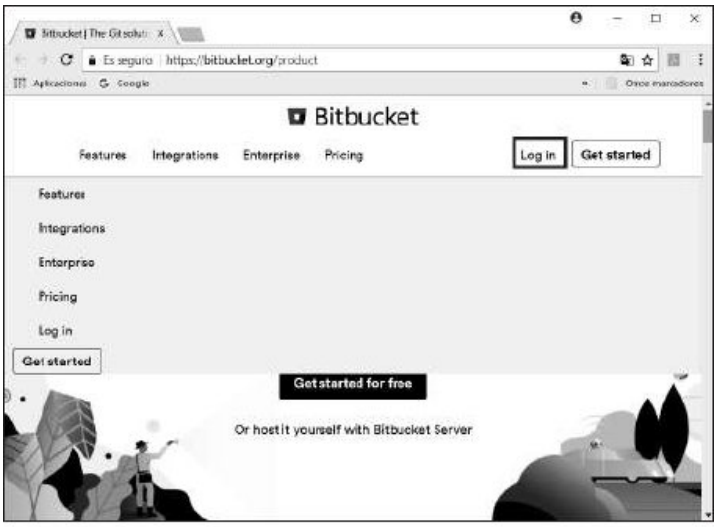
En definitiva, tendremos el conjunto de resultados que hemos obtenido anteriormente de forma separada.

CAPÍTULO 9:
Bitbucket & Jenkins

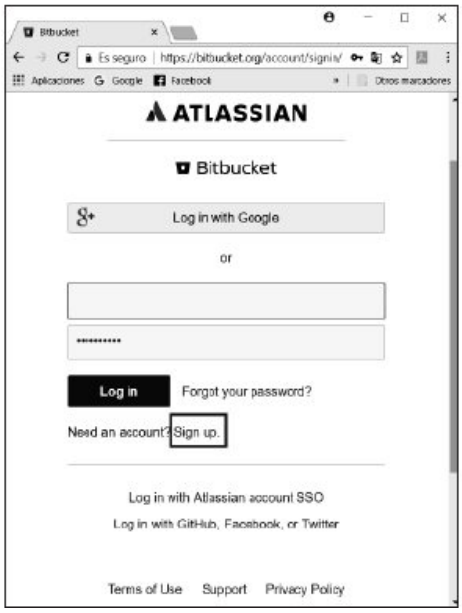
El uso de un control de versiones mediante el que podamos gestionar y compartir nuestras aplicaciones es una práctica muy extendida que nadie se cuestiona hoy en día. Existe un buen número de productos que podemos usar para realizar esta tarea, de entre los que destacamos **CVS**, **Subversion**, **Team Foundation Server**, **Git**, etc. Para nuestra explicación de Jenkins nos basaremos en **Git** y en un servicio de alojamiento denominado **Bitbucket**, del que aprovecharemos uno de sus planes gratuitos y sobre el que depositaremos nuestros proyectos de pruebas.

Así que lo primero que haremos será crear una cuenta en **Bitbucket**; para ello accederemos a la siguiente **URL** y nos registraremos en ella:

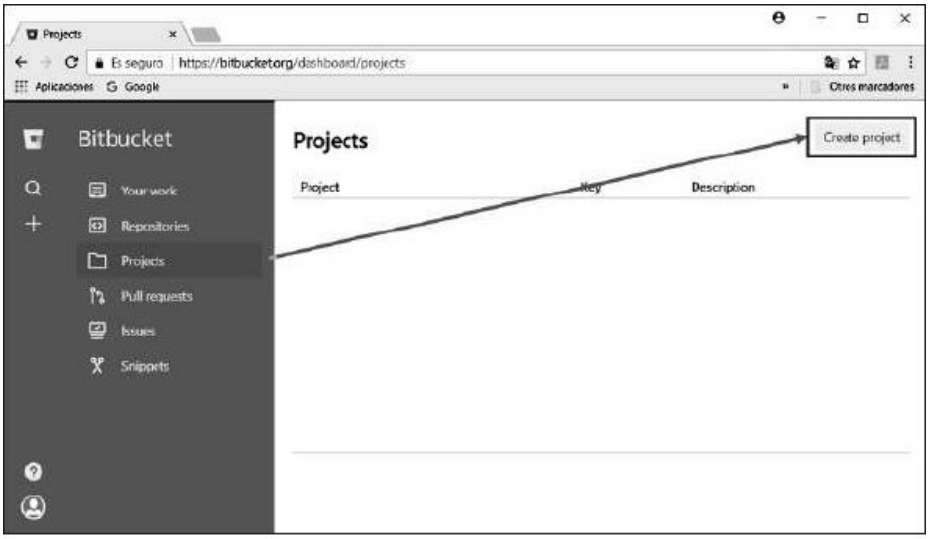
https://bitbucket.org



Si pulsamos sobre **Log in**, accederemos a la pantalla de acceso a **Bitbucket**, desde donde podemos registrarnos, si deseamos una cuenta nueva, pulsando a pie de pantalla sobre la opción **Sign up**:

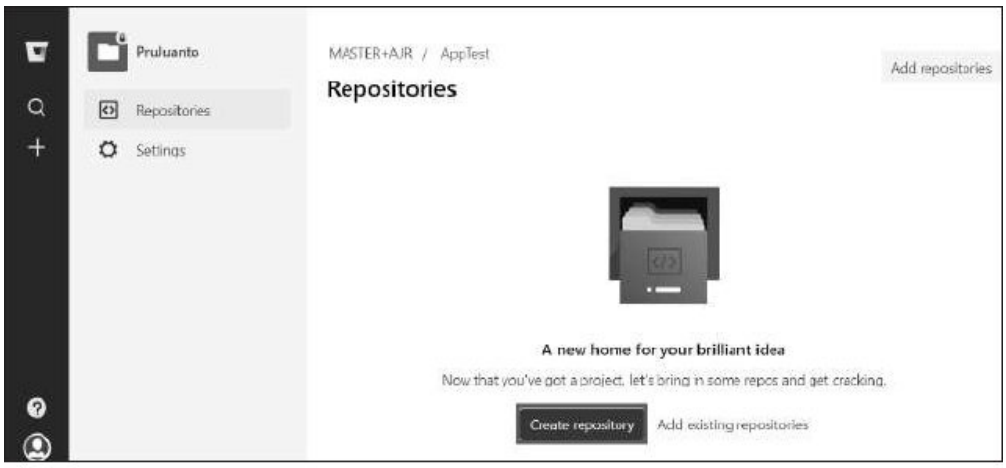


Una vez creada la cuenta, podemos acceder a **Bitbucket** y crear nuestro primer proyecto. Para ello, pulsaremos sobre la opción **Projects**, que se halla situada en la lista de opciones de la parte izquierda de la pantalla, y seguidamente sobre la opción **Create project**, situada en la parte superior derecha de la ventana:

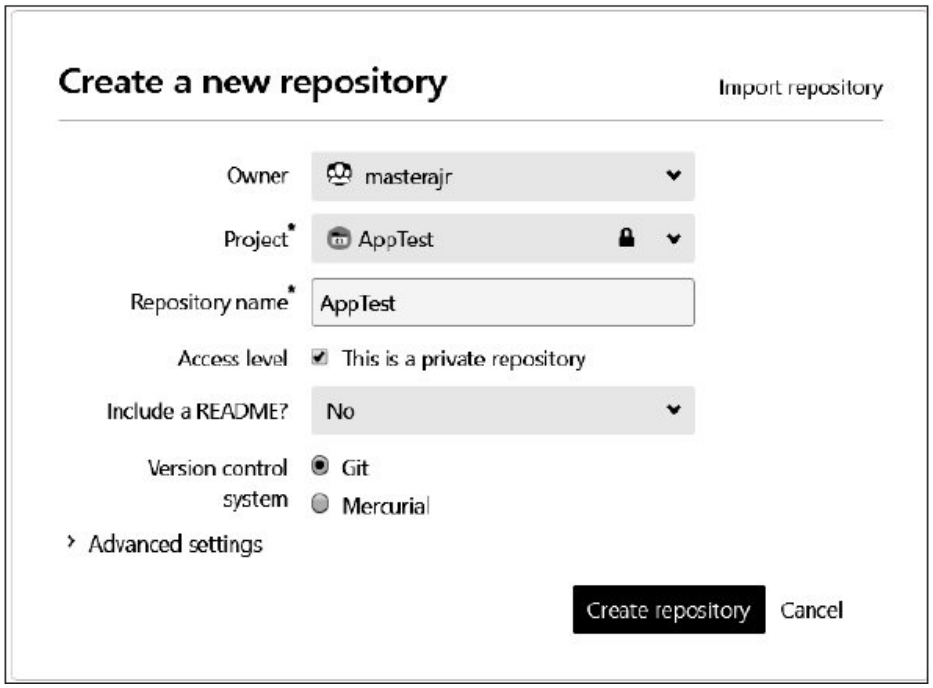


Al pulsar sobre esta opción, aparece la siguiente ventana, que rellenaremos de la siguiente forma:

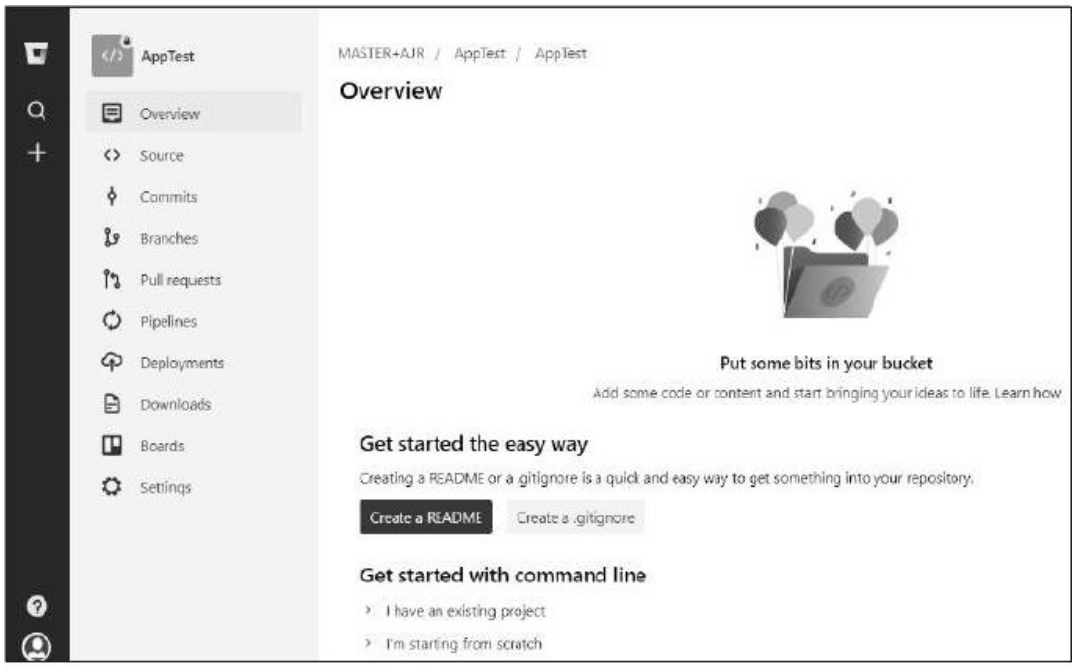
Una vez creado el proyecto, veremos la siguiente pantalla, a partir de la cual crearemos un repositorio nuevo pulsando sobre **Create repository**:



Tras pulsar esta opción pasaremos a la siguiente pantalla, que rellenaremos con el nombre de nuestro repositorio, y aceptaremos los valores propuestos por defecto:



Por último, al pulsar sobre **Create repository**, se crea el repositorio y nos muestra la siguiente pantalla a la espera de que realicemos más acciones sobre este:



En este punto, podemos recuperar nuestra aplicación **AppTest** que creamos para poder explicar algunos conceptos sobre **JUnit** y **Mockito**, y la conectaremos con nuestro proyecto recién creado.

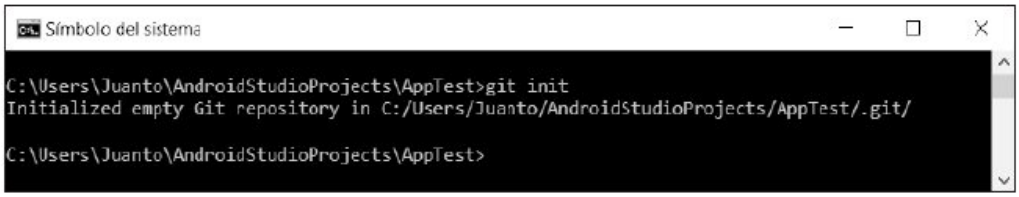
Para ello, podemos abrir una sesión de **CMD** en **Windows** y colocarnos en el directorio que contiene nuestro proyecto, que en mi ejemplo es:

```
C:\Users\Juanto\AndroidStudioProjects\AppTest
```

Una vez ubicados en dicho directorio, teclearemos:

```
git init
```

Esta acción crea un subdirectorio llamado **.git** con todo lo necesario para conectar el proyecto con **Git**.



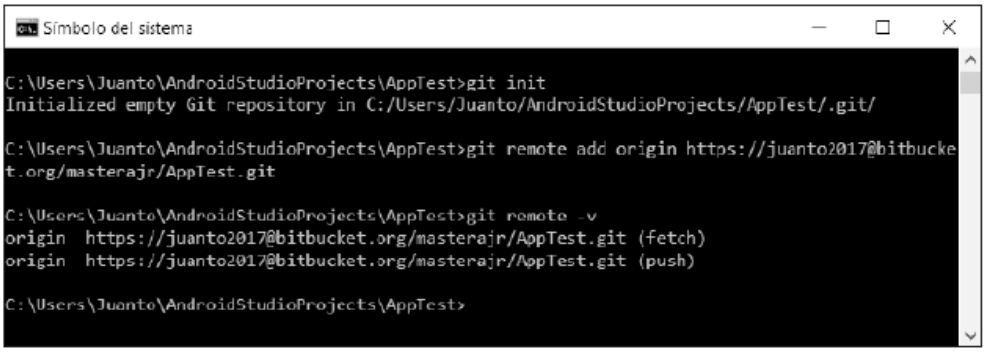
A continuación, añadiremos un nuevo repositorio **Git** remoto asignando un nombre. Para ello teclearemos lo siguiente:

```
git remote add origin https://juanto2017@bitbucket.org/masterajr/
AppTest.git
```

En el ejemplo, **juanto2017** se corresponde con el usuario que hemos usado en **Bitbucket**.

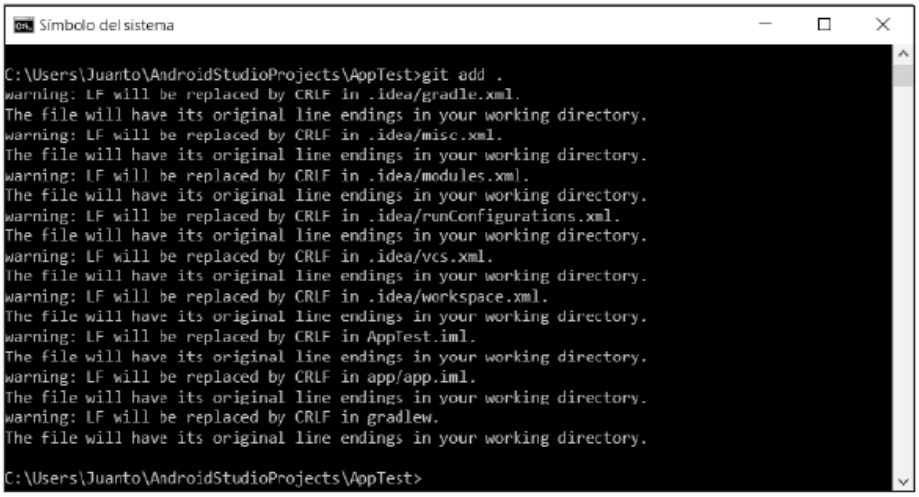
A continuación, listaremos las **URL** asociadas al repositorio tecleando:

```
git remote -v
```



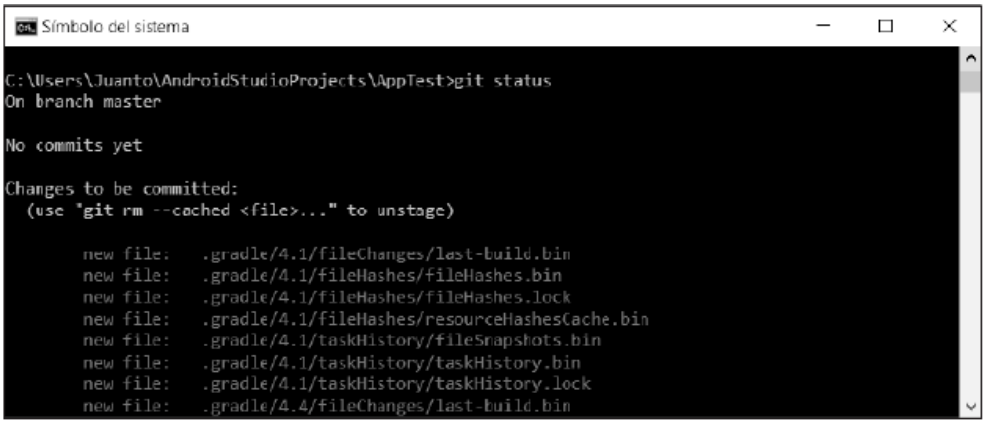
Seguidamente indicaremos qué archivos deseamos controlar usando el comando **add**:

```
git add .
```



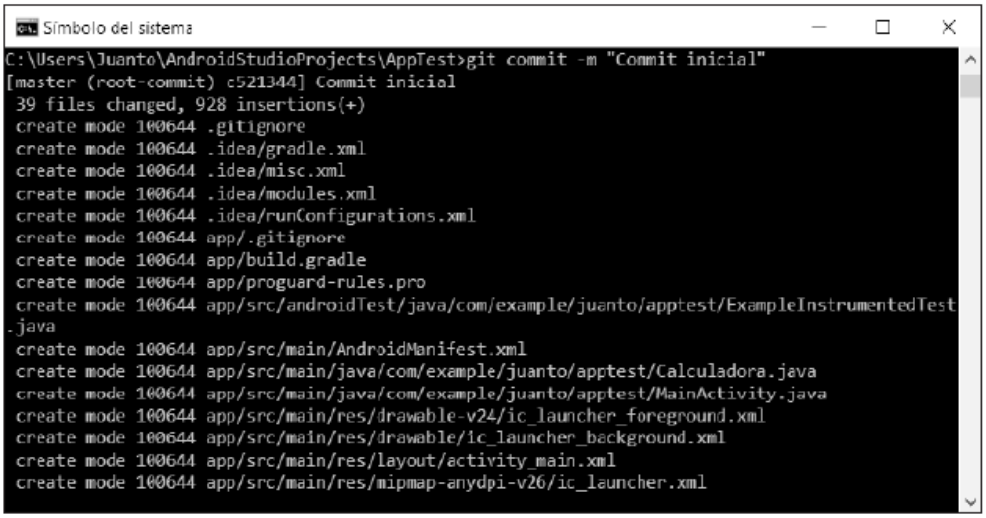
Podemos revisar las acciones pendientes mediante el comando **status**:

```
git status
```



En este punto, ya estamos en situación de realizar el **commit** mediante el siguiente comando, que va acompañado de un texto explicativo de la acción:

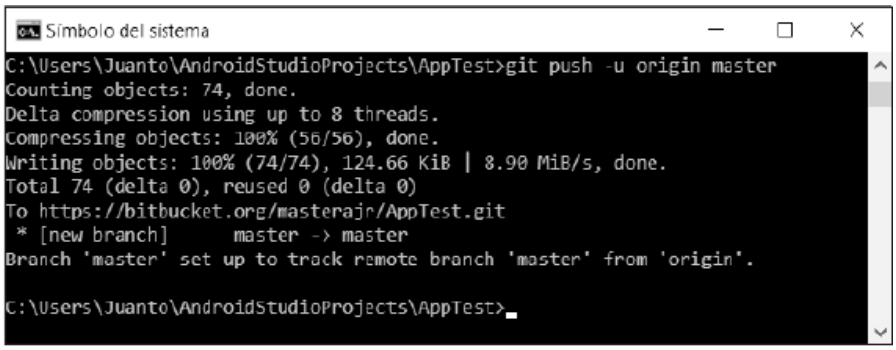
```
git commit -m "Commit inicial"
```



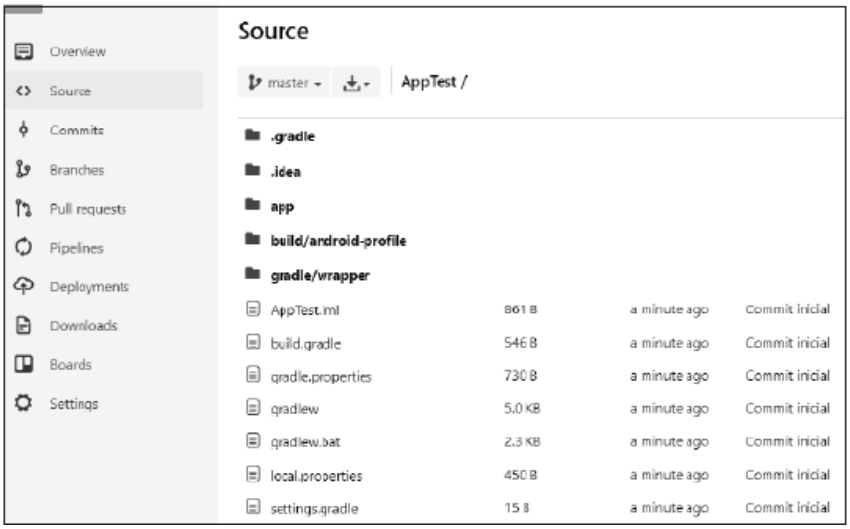
Por último, ejecutaremos el comando **push** para enviar nuestros cambios al repositorio de la siguiente manera:

```
git push -u origin master
```

El resultado será el siguiente:



Ahora, si acudimos a **Bitbucket**, comprobaremos que nuestro repositorio ya posee el contenido de nuestra app. Por ejemplo, dentro de **Bitbucket**, podemos acudir al apartado **Source** para comprobarlo:



A continuación, introduciremos algunos conceptos relacionados con **Jenkins** y sobre cómo podemos aprovecharnos de este para automatizar ciertos procesos, entre ellos la construcción de nuevas **builds**, en caso de que las pruebas realizadas fuesen satisfactorias.

Jenkins

Jenkins es un software de integración continua. Está escrito en **Java** y es **open source**.

La primera versión de Jenkins data de febrero de 2011, pero su desarrollo comenzó como una parte del proyecto Hudson, en 2004, por parte de **Sun Microsystems**.

Jenkins se lanza en un servidor que funciona como contenedor de servlets de forma similar a como lo hace **Apache Tomcat**.

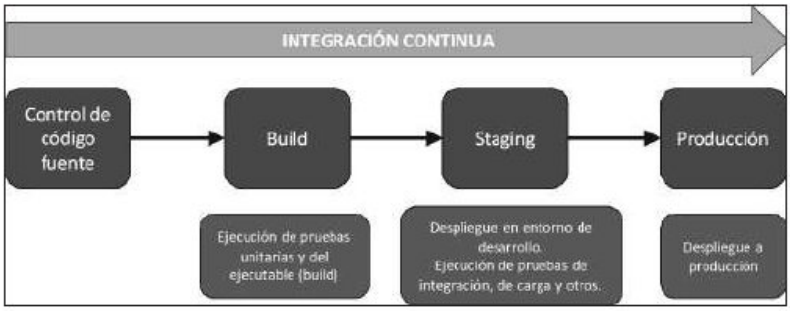
La integración continua propone que los desarrolladores vayan realizando despliegues de software, que pueden ser pequeños pero, sobre todo, frecuentes,

sobre un repositorio de forma que este pueda someterse a una serie de pruebas automáticas que den lugar a un producto cada vez más acabado y mejorado.

La integración continua pretende:

- **Hallar y arreglar errores rápidamente.**
- **Aumentar la calidad del software.**
- **Validar en menos tiempo.**
- **Publicar actualizaciones de software.**

A continuación, mostramos un gráfico con la secuencia clásica de acciones realizadas por Jenkins:



Cada vez que se genera y envía una revisión al repositorio para su publicación, se genera la creación de una **release** y se realizan sus correspondientes pruebas antes de su despliegue en producción definitivo. La idea es generar un ejecutable (**build**) en cada integración y que este se pueda crear de forma automatizada junto con sus pruebas y métricas de calidad adecuadas.

Jenkins se apoya en tareas que permiten indicar qué hacer con un **build**. Por ejemplo, podríamos hacer que cuando un desarrollador suba su código al control de versiones, automáticamente se realice una compilación y una ejecución de pruebas, de forma que, si algo falla, se envíe un aviso a QA para que solucione el problema. Por el contrario, si las pruebas van bien, podemos indicar a Jenkins que realice la integración.

Desde Jenkins podemos también lanzar métricas de calidad y ver los resultados en la propia herramienta. También podemos ejecutar **Unit Test** creados para cada aplicación y test automáticos.

Instalación

Para la instalación y posterior configuración de Jenkins y, teniendo en cuenta el ejercicio que realizaremos próximamente, necesitamos descargar el siguiente software (o verificar la existencia de este en nuestro equipo):

Software	Ubicaciones de descarga/comentarios
Jenkins	En esta explicación, usaremos Jenkins 2.110. https://jenkins.io/download/

Gradle	En esta explicación, usaremos Gradle 4.6. https://gradle.org/releases/
Java	En esta explicación, usaremos jdk1.8.0_162. http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

La instalación de **Gradle** y Java es bastante intuitiva y prácticamente solo hay que aceptar las opciones que se proponen por defecto.

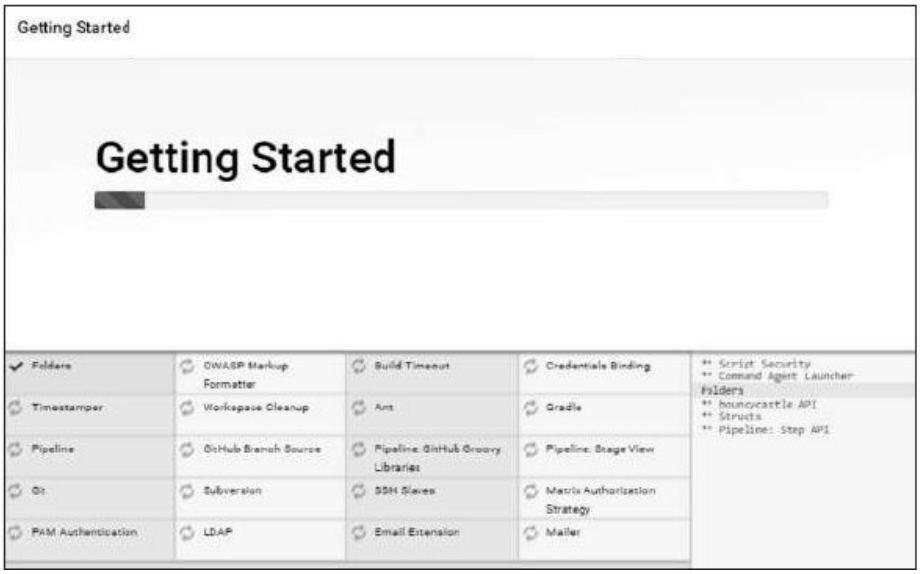
En el caso de Jenkins, también resulta bastante sencillo, pero lo explicaremos con algo más de detalle.

En primer lugar, descargaremos Jenkins desde la página oficial:
https://jenkins.io/download/

En esta página se puede descargar el instalador que corresponda a nuestro sistema operativo:

Al seleccionar un sistema operativo, se iniciará la descarga y, una vez finalizada, procederemos a su instalación descomprimiendo el archivo y ejecutando el instalador. Durante la instalación recibiremos diferentes propuestas que aceptaremos por defecto. Al arrancar la instalación veremos la siguiente pantalla:

Pulsaremos sobre **Install suggested plugins** y se iniciará la instalación:



Una vez que se haya realizado la primera parte de la instalación, se invitará a crear un usuario administrador:

Una vez cumplimentados los datos, podremos empezar a usar Jenkins:

Al pulsar sobre **Start using Jenkins**, nos encontraremos la primera pantalla del producto:

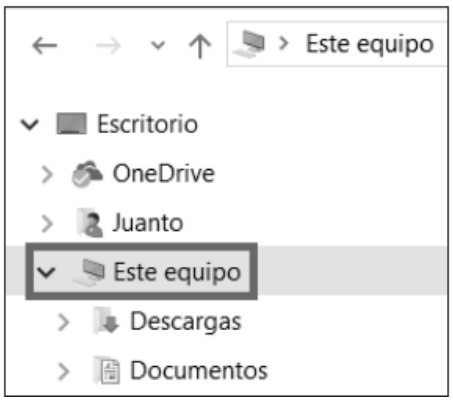
Para desconectarnos de Jenkins, basta con pulsar sobre la opción **Desconectar**:

Antes de empezar a configurar Jenkins, vamos a crear una serie de variables de entorno para indicar al sistema donde tenemos nuestros entornos de **Gradle**, **Android** y **Java**.

En mi caso, tengo ubicados dichos entornos en los directorios que indico a continuación y, por tanto, las variables de entorno tendrán los siguientes valores:

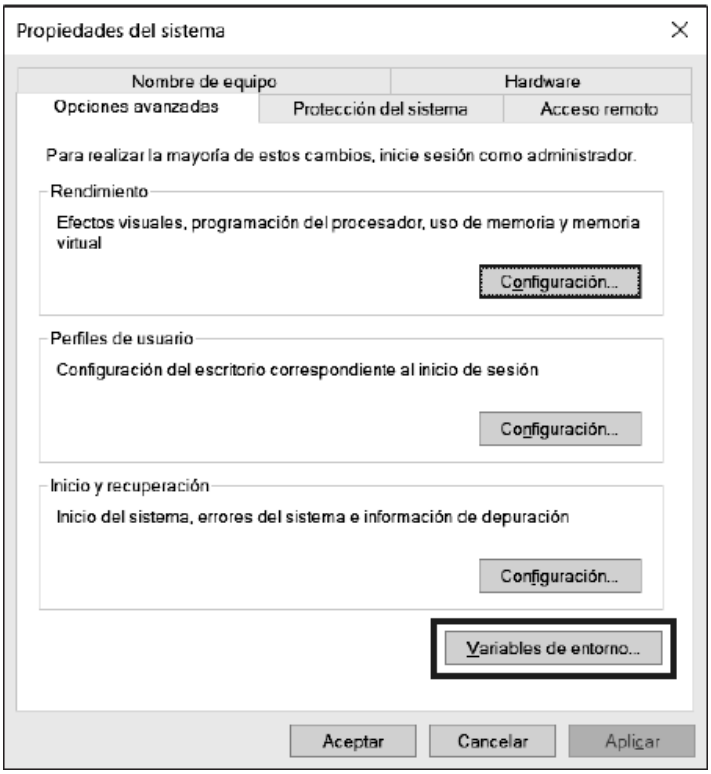
Variable de entorno	Valor
GRADLE_HOME	C:\Gradle\gradle-4.6
ANDROID_HOME	C:\Users\Juan\AppData\Local\Android\Sdk
JAVA_HOME	C:\Program Files\Java\jdk1.8.0_45

En **Windows**, para definir una variable de entorno, podemos abrir una sesión de Explorador de Windows y localizar el icono **Este equipo**:

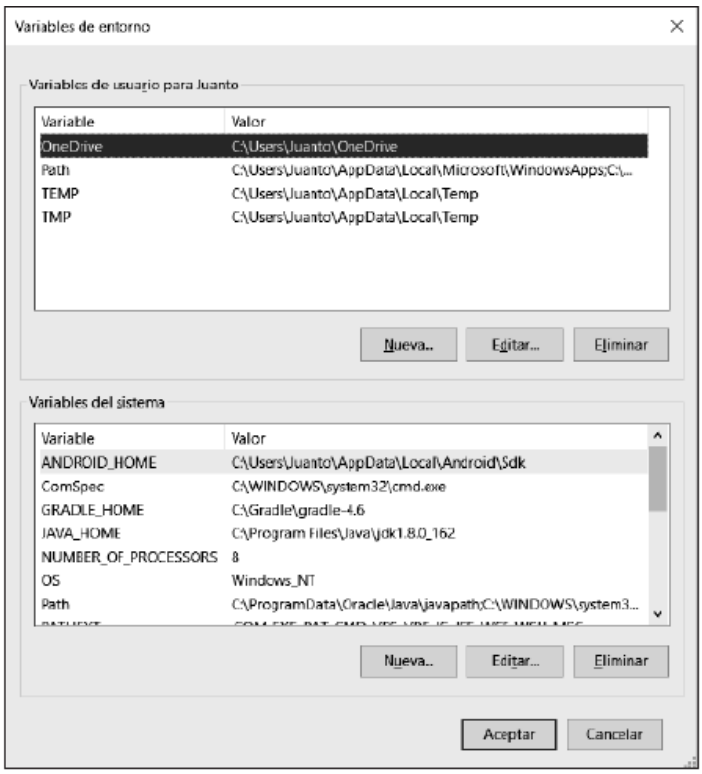


Una vez localizado este icono, haremos clic con el botón derecho del ratón sobre él y seleccionaremos la opción **Propiedades** para que aparezca la ventana principal del **Panel de control**:

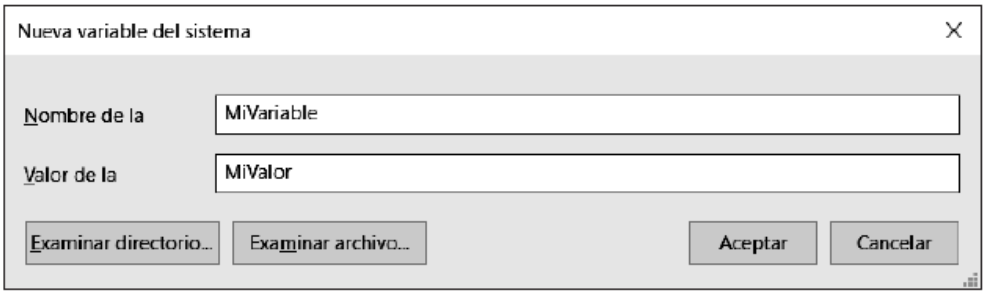
Al pulsar sobre **Configuración avanzada del sistema**, aparece la siguiente ventana, desde la que podremos acceder a gestionar las variables de entorno:



Si pulsamos sobre **Variables de entorno**, accedemos a la siguiente pantalla, desde la que podemos gestionar las variables de entorno que necesitamos:



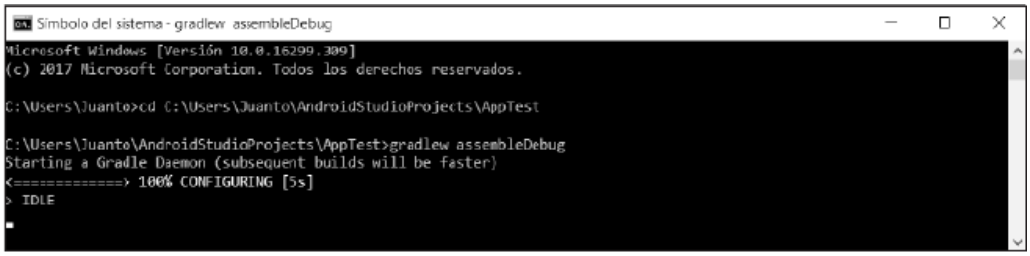
Al pulsar sobre **Nueva...**, aparece un cuadro de diálogo que solicita el nombre y el valor de la variable que deseamos crear:



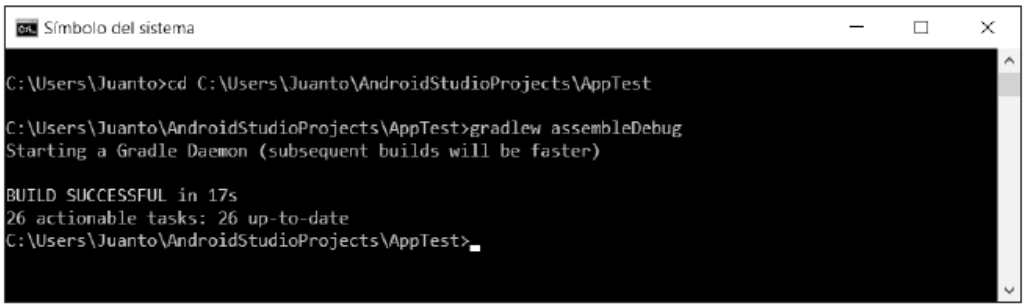
Una vez definidas las variables de entorno, vamos a realizar una pequeña prueba de compilación en nuestro equipo. Para ello, nos ubicaremos en el directorio donde tenemos la aplicación y ejecutaremos el comando **gradlew** de la siguiente manera:

```
gradlew assembleDebug
```

Observaremos cómo empieza el proceso:



Al final de este veremos el siguiente resultado:



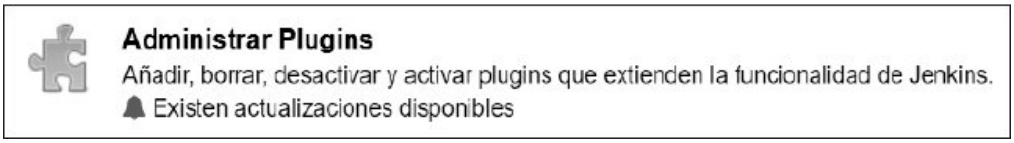
A continuación, vamos a configurar Jenkins para crear nuestra primera tarea, que consistirá en descargar la app que creamos en el ejercicio anterior, llamada **AppTest**, y compilarla para generar de pasos los test que creamos cuando explicamos algunos ejemplos de **JUnit** y **Mockito**. Para ello, acudiremos a la URL http://localhost:8080/ y accederemos a la pantalla de inicio de sesión:

En ella usaremos un usuario administrador para conectarnos y, una vez introducido el **Usuario** y la **Contraseña**, pulsaremos sobre **Entrar** para acceder al panel de control de Jenkins:

A continuación, necesitaremos instalar los siguientes plugins:

- **Bitbucket Plugin**
- **Gradle Plugin**
- **JUnit Plugin**
- **JUnit Realtime Test Reporter Plugin**
- **Workspace Cleanup Plugin**
- **Android Emulator Plugin**
- **Build Timeout**

Para ello, seleccionaremos **Administrar Jenkins** y seguidamente **Administrar Plugins**:



Una vez que accedamos a la pantalla del **Gestor de plugins**, observaremos diversas pestañas desde las que podremos verificar si los plugins que necesitamos ya están instalados o, en caso contrario, desde las que podremos instalarlos. Para comprobar si un plugin está instalado, accederemos a la pestaña de **Plugins instalados** y veremos si dicho plugin aparece marcado o no:

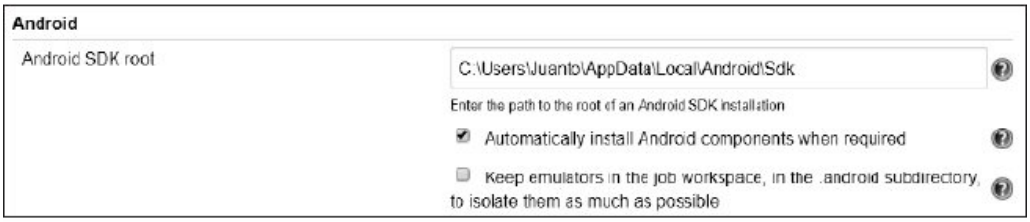
Observa que podemos usar el campo **Filter** para buscar el plugin que necesitamos tratar.

Si un plugin no está instalado, pulsaremos sobre la pestaña **Todos los plugins**, lo localizaremos y lo marcaremos:

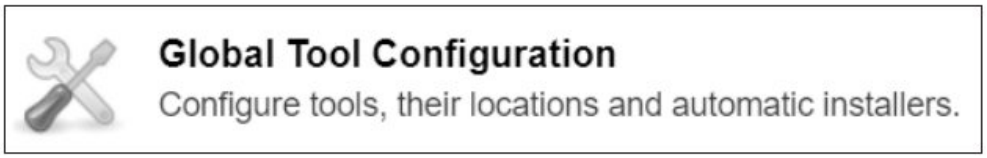
Luego, podemos decidir si lo instalamos sin reiniciar o después de reiniciar. En nuestro caso, por ejemplo, elegiremos **Instalar sin reiniciar**.

Una vez instalados los plugins continuaremos con la configuración de Jenkins accediendo a **Administrar Jenkins - Configurar el Sistema**:

En este apartado, definiremos el **path** donde tenemos ubicada nuestra instalación del **SDK** de **Android**. Para ello, localizaremos el apartado **Android**:



Seguidamente, configuraremos otros parámetros accediendo a **Global Tool Configuration**:

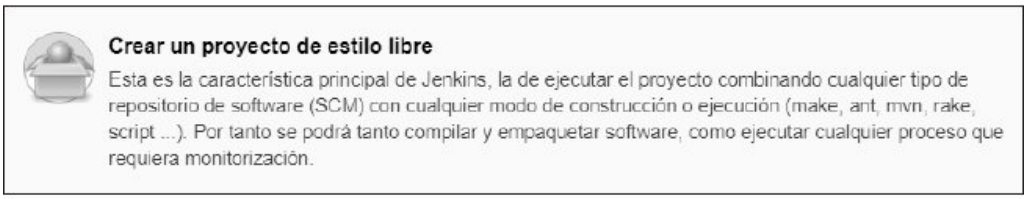


Al pulsar en esta pantalla, localizaremos el apartado **JDK**, donde indicaremos la instalación que vamos a utilizar, que en nuestro caso será la **jdk1.8.0_162**. Si no disponemos de ninguna, pulsaremos sobre el botón **Añadir JDK** e introduciremos el nombre y la ubicación en nuestro equipo donde se halla:

Ahora ya estamos en situación de crear un nuevo proyecto con el que realizaremos nuestro ejemplo. Para ello, nos ubicaremos en la pantalla principal de Jenkins y pulsaremos sobre **Nueva Tarea**:

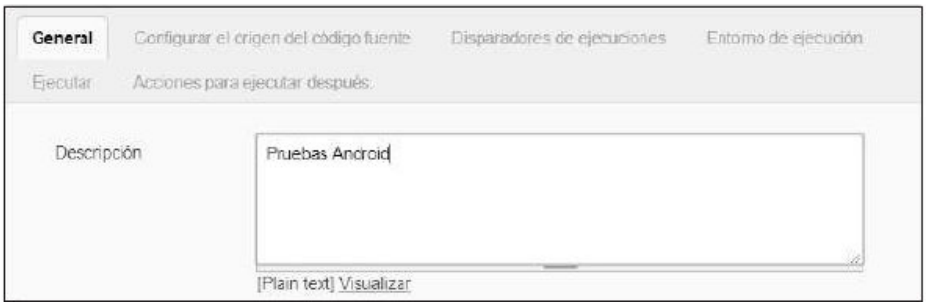
Una vez pulsada esta opción, aparece la siguiente pantalla, en la que nos solicita el nombre del proyecto y el tipo de este. Introduciremos el nombre **PruAppTest** y seleccionaremos el tipo **Crear un proyecto de estilo libre**:

Ampliando la imagen podemos ver algunos detalles más sobre los proyectos de estilo libre:



Una vez introducido el nombre y seleccionado el tipo pulsaremos sobre **OK** y pasaremos a definir las características de nuestro proyecto.

En la pantalla de nuestro proyecto, introduciremos una breve descripción de este:



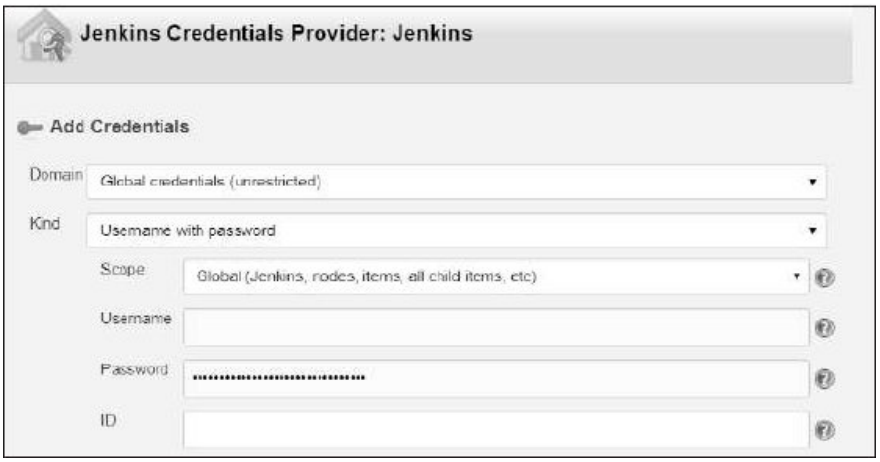
Seguidamente, en el apartado **Configurar el origen del código fuente**, seleccionaremos la opción **Git** para que nos muestre los campos que necesita para poderse

conectar a nuestro repositorio del proyecto en **Git**. Los campos necesarios son básicamente la **URL del repositorio** y las **credenciales** con las que accederemos.

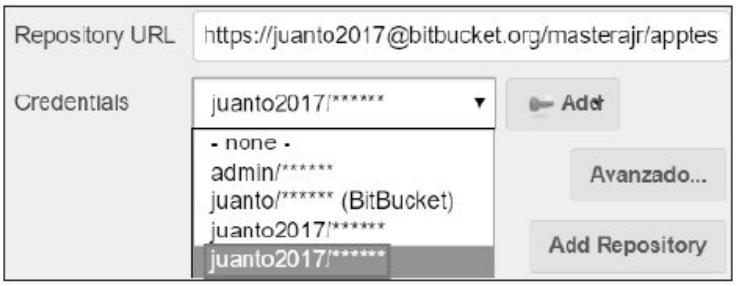
Podemos obtener fácilmente la URL del repositorio accediendo a Bitbucket y seleccionando la vista **Overview** de nuestra aplicación (AppTest). Allí, bastará con copiar la URL que aparece justo a la derecha del desplegable que muestra HTTPS:

En mi caso, es la siguiente:
https://juanto2017@bitbucket.org/masterajr/apptest.git
Esta es la **URL** que introduciremos en Jenkins:

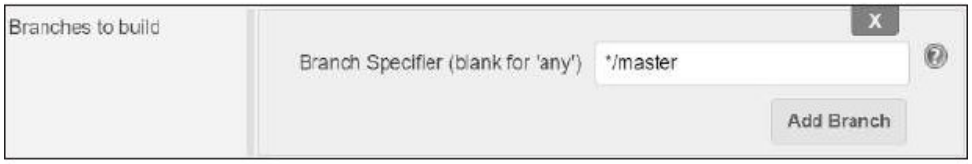
Para indicar las credenciales, primero deberemos crearlas pulsando sobre el botón **Add** y seleccionando **Jenkins**. En la pantalla que aparece, introduciremos nuestro usuario y contraseña, tal y como se indica a continuación:



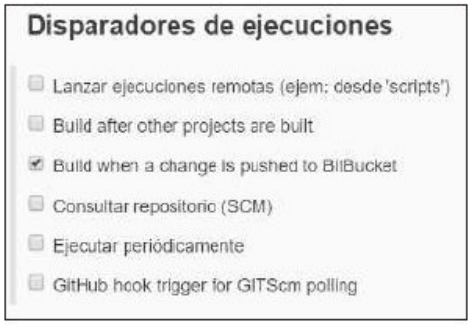
Una vez que pulsemos sobre **Add**, ya podremos desplegar la lista de valores **Credentials** y seleccionar nuestro usuario:



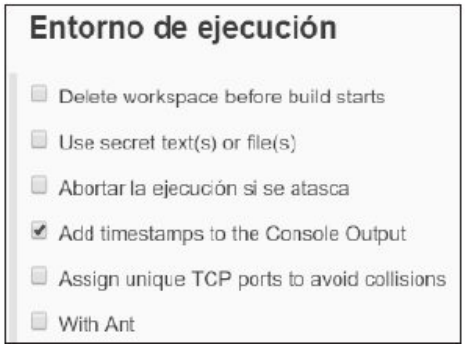
Dejaremos ***/master** en el apartado **Branches to build**:



Seguidamente, en el apartado **Disparadores de ejecuciones**, marcaremos **Build when a change is pushed to Bitbucket** para que la construcción se dispare cuando se realice un cambio en el repositorio:



También, en el **Entorno de ejecución**, marcaremos la opción **Add timestamps to the Console Output** para que, cuando analicemos la salida de la construcción, tengamos más datos sobre cuándo se realizó cada acción:

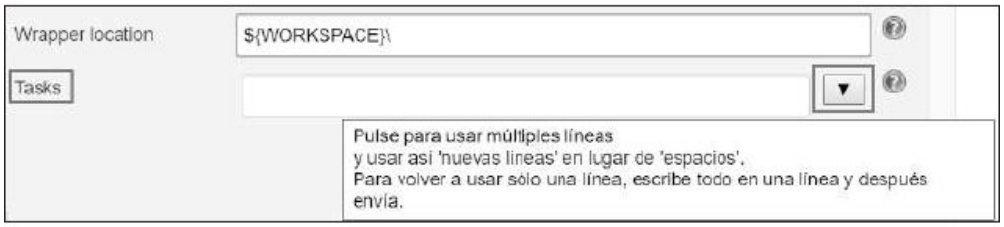


En el apartado **Ejecutar**, seleccionaremos la opción **Invoke Gradle script**:

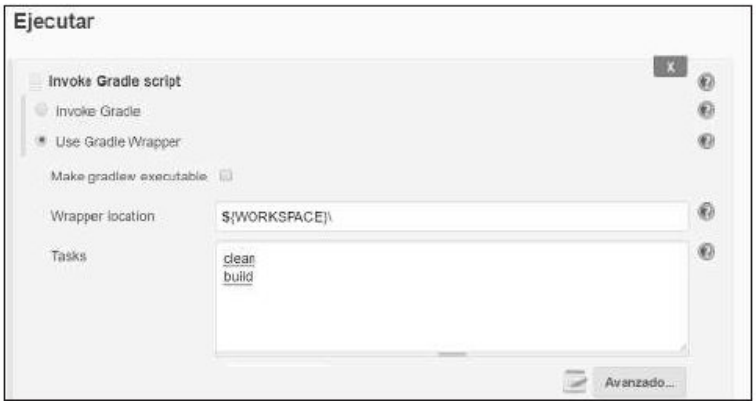
Seleccionaremos **Use Gradle Wrapper** e introduciremos los siguientes campos:

Wrapper location	**${WORKSPACE}**
Task	clean build

En el campo **Tasks** podemos pulsar sobre la flechita que se halla a su derecha para convertir el campo en multilínea:



Quedará de la siguiente manera:



A continuación, pulsaremos sobre el botón **Avanzado...** y marcaremos la opción **Force GRADLE_USER_HOME to use workspace**:

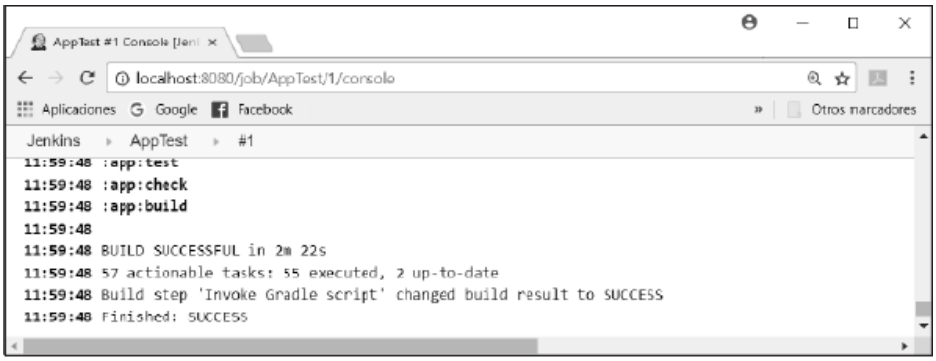
Una vez definidos estos apartados, ya podemos pulsar sobre **Guardar** para pasar a construir la **build**.

Desde la pantalla del proyecto, pulsaremos sobre **Construir ahora** para iniciar la construcción.

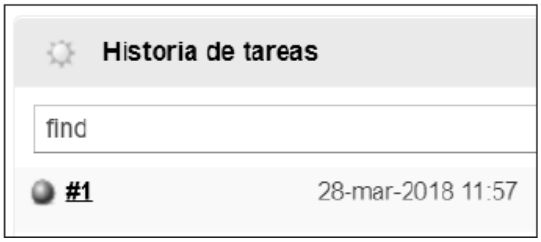
Al pulsar sobre **Construir ahora**, observaremos cómo en la lista de **Historia de tareas** aparece una nueva entrada que indica que hay un proceso en construcción:

Si pulsamos sobre dicha entrada, aparecerá una pantalla con diversas opciones relacionadas con ella. Pulsaremos sobre **Console Output** para ver lo que sucede durante la construcción:

Al final del proceso, deberíamos obtener un resultado como el siguiente:



Al final de esta, podremos revisar la lista de **Historia de tareas** y sus resultados:



CAPÍTULO 10:
Trello

Trello es una herramienta que, gracias a su flexibilidad y simplicidad, puede utilizarse en la organización de casi cualquier proyecto del tipo que sea, tanto individual como colaborativo.

Se basa en la visualización de un panel dividido en listas o columnas, a las que se puede dar nombre, y dentro de las cuales se pueden agregar tarjetas con el significado que se quiera.

Con esta definición tan simple y abierta podremos organizar cualquier tipo de proyecto de una forma muy visual, lo que nos permitirá observar en cada momento, de una forma fácil, su estado.

Otra de las grandes ventajas es la gestión del trabajo colaborativo, y es que cuando trabajamos dentro de un equipo una de las claves es la coordinación entre sus miembros. Trello sincroniza en todo momento los cambios que hagamos en los tableros de un proyecto, de modo que sean inmediatos para todos los miembros.

Para ilustrar los conceptos mencionados anteriormente podríamos crear un tablero de Trello en el que poder gestionar la aplicación que queremos construir. Desde un punto de vista de gestión de un proyecto software, como la creación de una aplicación móvil, querremos representar en nuestro tablero principalmente:

- ¿Qué tareas tenemos que llevar a cabo?
- ¿Quién del equipo se encarga de cada tarea?
- ¿Qué plazos tenemos para cada una?

Otra decisión que hay que tomar es la metodología de desarrollo software que se llevará a cabo. En la actualidad, los proyectos software suelen abordarse bajo metodologías ágiles, ya que nuestro dominio del problema o modelo de nuestro minimundo puede cambiar rápidamente y debemos poder adaptarnos sin demora.

Las metodologías ágiles nos permiten liberar versiones de nuestro desarrollo con más frecuencia y anticiparnos a posibles errores y malentendidos con un cliente, así como lanzar funcionalidades nuevas como prueba de concepto para ver si funcionan o no.

Una de las metodologías ágiles que engloba lo que acabamos de mencionar es **Scrum**. La utilizaremos junto con **Kanban**, que es un sistema que nos ayuda a saber qué vamos a producir en cada momento de nuestro software en el caso que nos ocupa.

Para ello haremos uso de nuestro tablero Trello y crearemos diferentes columnas, que podríamos denominar:

- Backlog
- Listo para desarrollo
- En proceso
- En pruebas
- Realizado

El siguiente paso sería crear las tareas que se llevarán a cabo y colocarlas en el **Backlog**, ya que es todo lo que tenemos pendiente por hacer.

Para nuestro tablero necesitamos saber los actores, quiénes son los miembros del equipo que intervendrán en las tareas.

Por último, para cada **sprint** colocaremos en **Listo para desarrollo** aquellas tareas que queramos planificar en el periodo de tiempo delimitado y que cada miembro del grupo, como responsable de cada tarea, tendrá que mover entre las diferentes columnas hasta que la funcionalidad se haya finalizado.

CAPÍTULO 11:
Slack

Es una utilidad que permite gestionar la mensajería de forma eficiente y favorecer la comunicación en tiempo real e integrada en un mismo sitio, todos los medios de comunicación que habitualmente usamos en nuestro día a día, tanto en la empresa como a nivel personal.

A su vez, integra gran cantidad de herramientas y servicios, como **Dropbox**, **Twitter**, **Skype**, **IFTTT**, **Bitbucket**, **Subversion**, **GitHub**, etc., lo que nos permite disponer de nuestros recursos habituales desde un único punto.

En la actualidad, más de 60.000 equipos y 3 millones de usuarios utilizan Slack. Dispone de apps nativas en **iOS**, **Android** y en **Windows Phone**, además de aplicaciones de escritorio para **Mac**, **Windows** y **Linux**.

Ofrece diferentes planes de contratación, pero para pequeñas empresas o para uso personal podemos utilizar gratuitamente los siguientes servicios:

- Buscar y navegar por los últimos 10.000 mensajes.
- Almacenar hasta 5 GB de datos.
- Integrar hasta 10 servicios.
- Número de miembros ilimitado.

A continuación, enumeramos algunos de los conceptos básicos de Slack:

- **Canales.** Espacios abiertos de comunicación creados para compartir información con los diferentes usuarios que acceden a ellos. Los usuarios se identifican mediante el signo # (hashtag). Existen canales públicos y canales privados.
- **Grupos.** Espacios privados donde solo usuarios invitados previamente pueden acceder. Se identifican mediante un candado (🔒).
- **Mensajes directos.** Conversaciones entre usuarios uno a uno.

Respecto a las menciones que podemos hacer en nuestras conversaciones enumeramos las siguientes:

- **Mencionar un usuario.** Al escribir **@nombre_usuario** en un canal o grupo, dirigimos un mensaje al usuario indicado.
- **Menciones especiales.** Podemos utilizar las siguientes menciones al utilizar un grupo o canal:
  - **@channel:** envía mensaje a todos los miembros del canal.
  - **@everyone:** envía mensaje a todos los miembros del equipo.
  - **@here:** envía mensaje a todos los miembros del canal que estén en línea.

Respecto a los mensajes, disponemos de:

- **Mensajes libres.** Mensajes que puedes escribir en un grupo o canal para que lo lean todos los usuarios registrados en él.
- **Mensajes directos.** Mensajes enviados directamente a alguno de los usuarios que aparezcan en la barra lateral o que busquemos mediante la opción **+ Invite People**.

Una de las posibilidades que tenemos en Slack es compartir diversos tipos de información de diferentes maneras. A continuación, relacionamos algunas de ellas:

- **Compartir archivos mediante drag and drop.** Podemos arrastrar y soltar archivos para compartirlos. Se abre una ventana para cumplimentar la acción (añadir un título, un comentario, etc.). Se puede realizar la misma acción pulsando

la tecla **Shift** para evitar el cuadro de diálogo. Podemos usar el signo + ubicado a la izquierda del cuadro de mensaje para acceder al archivo que queremos compartir.
- **Compartir enlaces.** Basta simplemente con copiarlos y pegarlos sobre el mensaje.
- **Compartir cualquier objeto del portapapeles.** Simplemente hacemos copiar y pegar.

Podemos dar un sencillo formato a nuestros mensajes mediante las siguientes convenciones:
- *Negrita*
- _Cursiva_
- ~Tachado~
- > Blockquote en una línea
- >>> Blockquote en varias líneas

En el cuadro de mensaje, disponemos de una serie de comandos que nos permiten realizar acciones como por ejemplo, añadir recordatorios para que Slack avise a un usuario. La siguiente tabla muestra los comandos que podemos realizar en dicho cuadro:

Comando	**Comentario**
/away	Alterna status 'dentro'/'fuera'.
/call	Inicia una llamada.
/collapse	Colapsa todos los ficheros del canal.
/dnd	Inicia o finaliza un mensaje de 'No molestar'.
/expand	Expande todos los ficheros del canal.
/feed	Gestiona suscripciones RSS.
/feedback	Envía un feedback a Slack.
/invite	Invita a un usuario al canal.
/invite_people	Invita a una persona a formar parte del equipo.
/leave	Sale del canal.
/me	Muestra el mensaje indicado.
/msg	Envía mensaje a otro usuario.

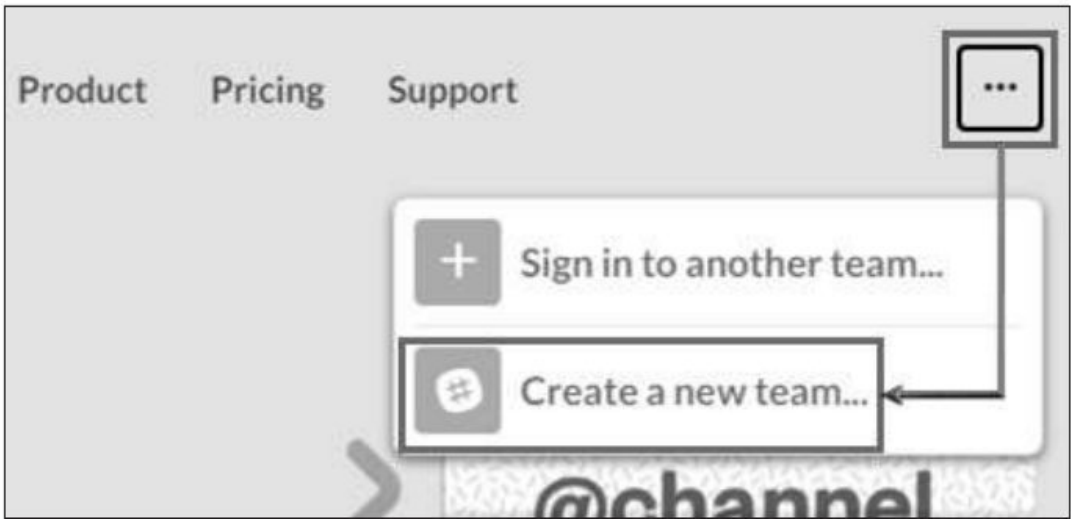
/mute	Silencia el canal en curso.
/open	Abre un canal.
/prefs	Muestra el cuadro de diálogo de preferencias.
/remind	Configura un recordatorio.
/rename	Renombra un canal.
/shorcuts	Abre el diálogo de 'atajos'.
/shrug	Añade ¯_( )_/¯ al mensaje.
/star	Destaca canal actual o conversación.
/status	Inicializa el status personalizado.
/who	Lista los usuarios del canal o grupo actual.

Creación de un grupo en Slack

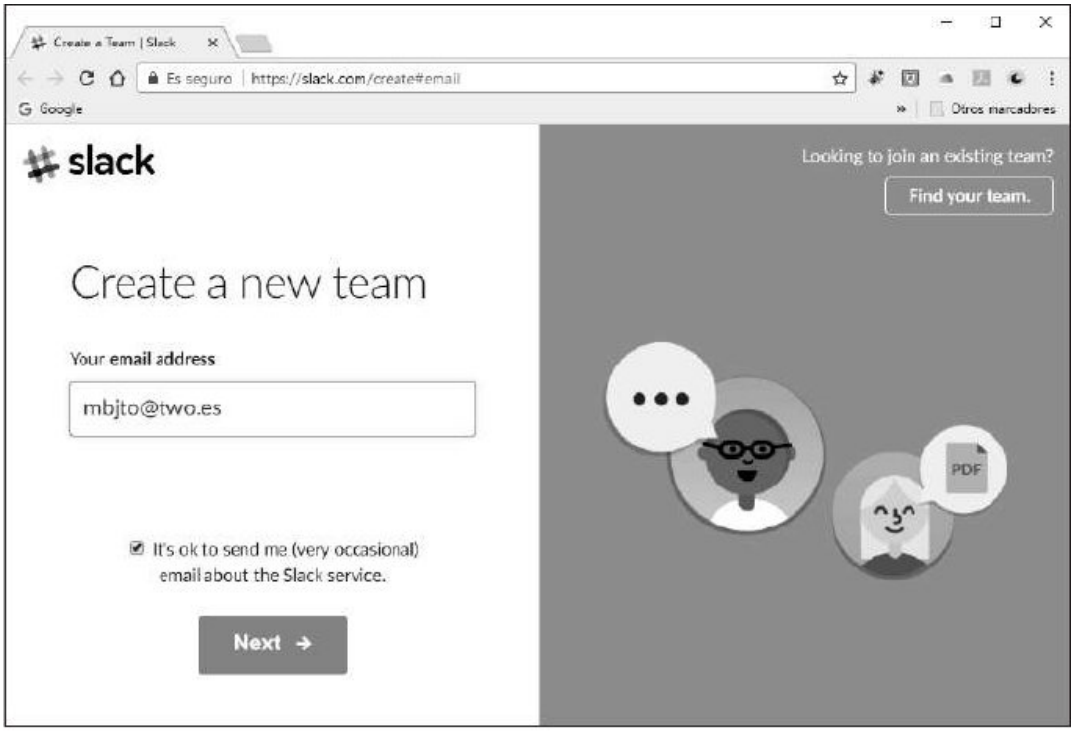
Para crear un grupo en Slack, en primer lugar, accederemos a:

https://slack.com/

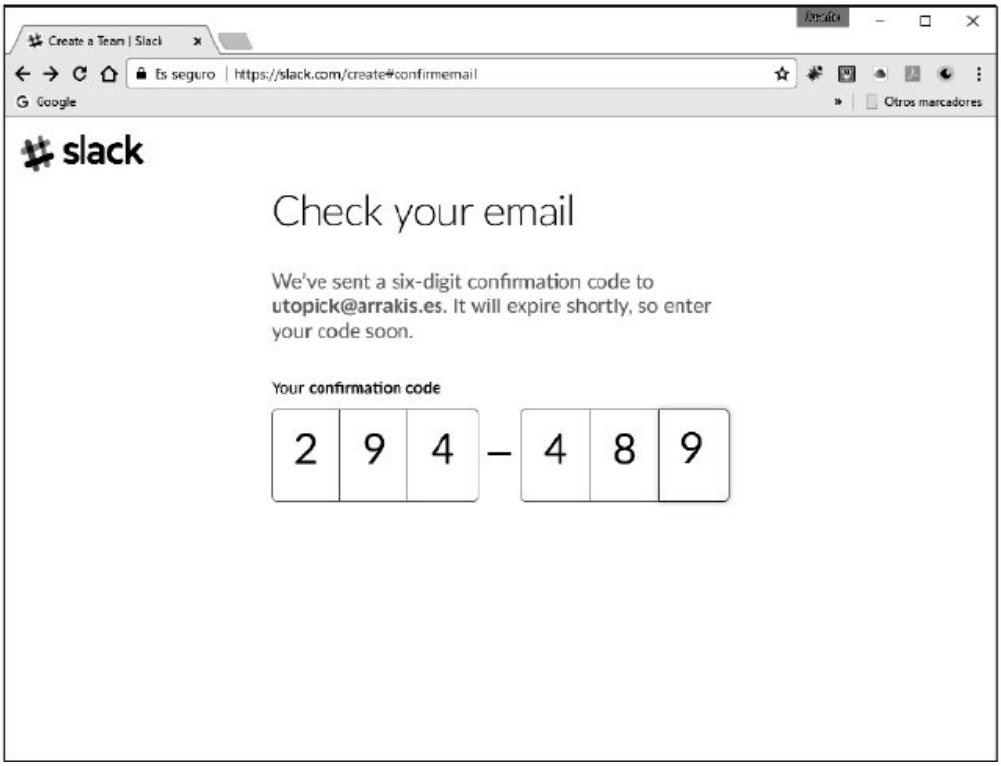
En la esquina superior derecha de la página veremos un botón que, si lo desplegamos, nos permitirá acceder a alguno de los grupos a los que ya pertenezcamos o bien crear uno nuevo. En nuestro caso, seleccionaremos la opción **Create a new team...**



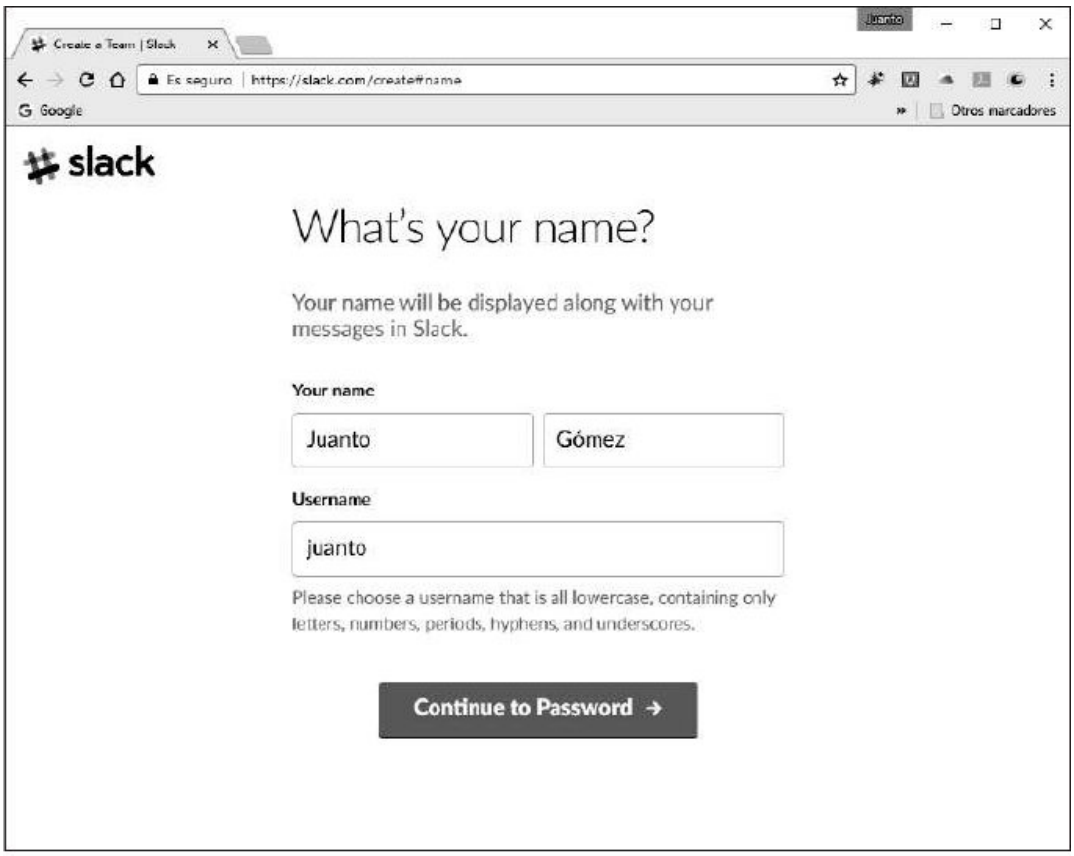
Una vez seleccionada esta opción, nos solicitará una dirección de correo:



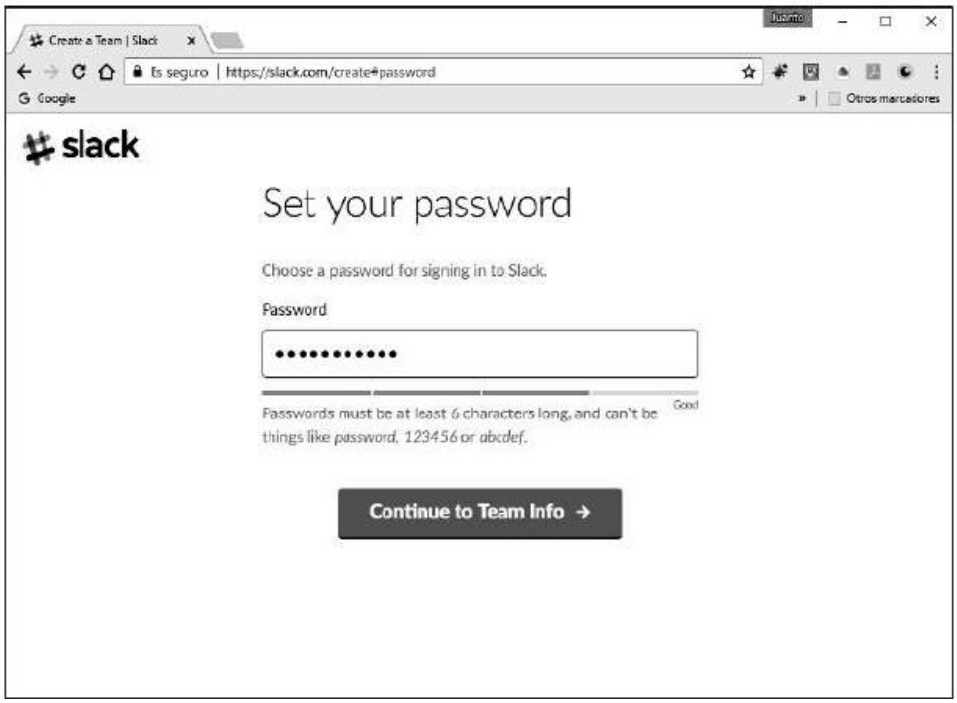
Al pulsar sobre **Next** recibiremos un código, por correo electrónico, que deberemos introducir en la siguiente pantalla (el siguiente código es un ejemplo):



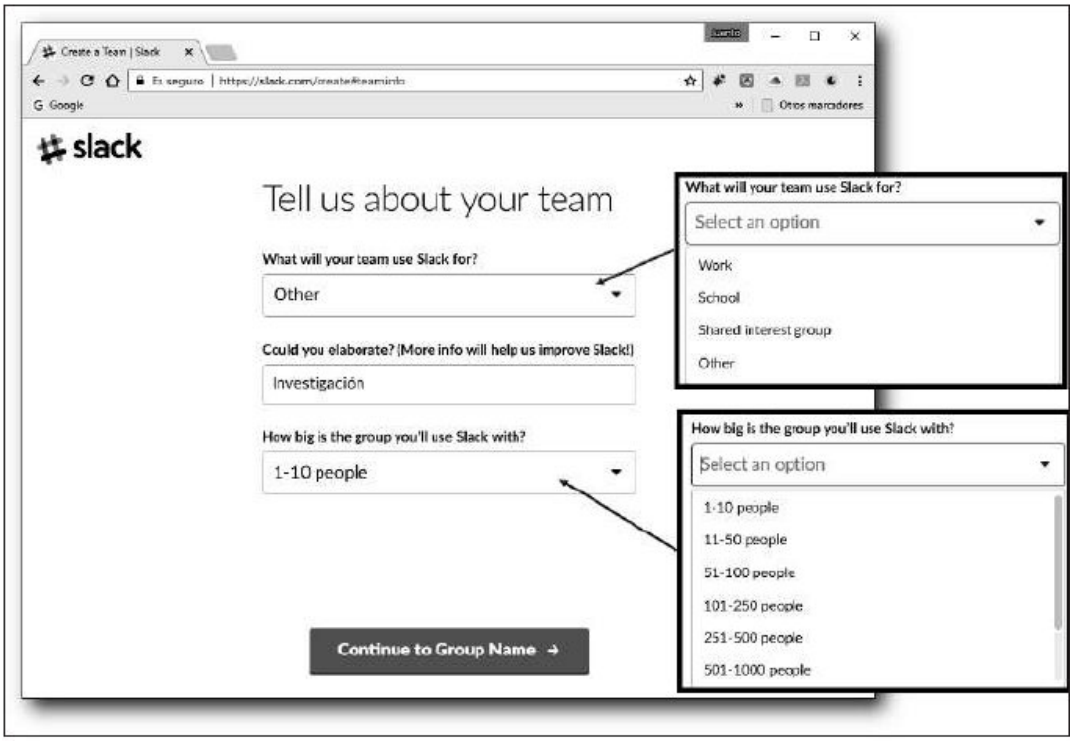
A continuación, introduciremos los siguientes datos (**nombre** y **usuario**):



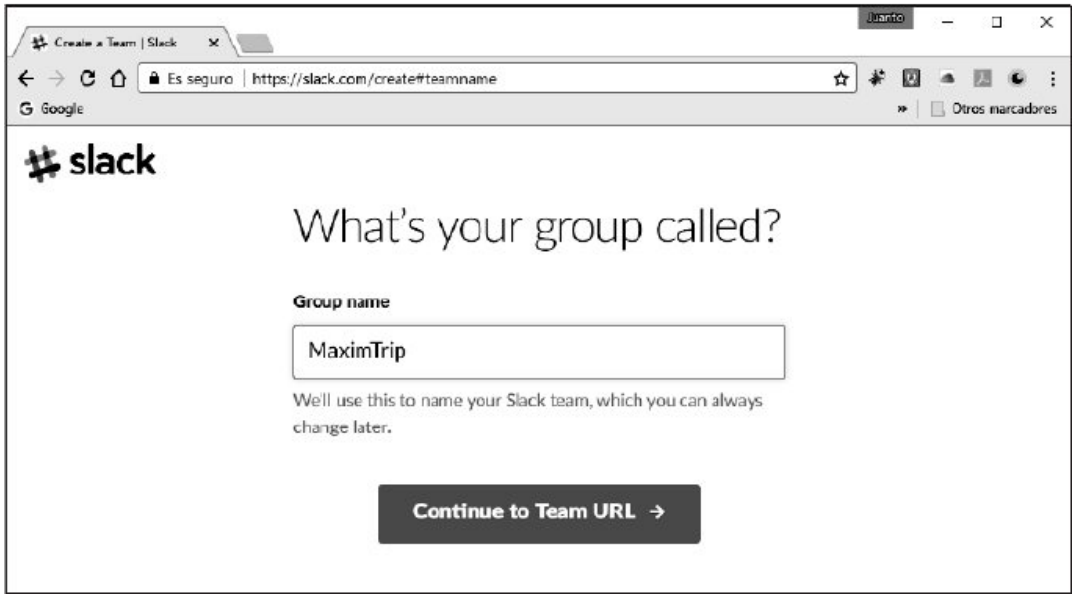
Al pulsar sobre **Continue to Password** se nos solicita el password que deseamos utilizar para acceder a Slack. Este debe contener al menos 6 caracteres:



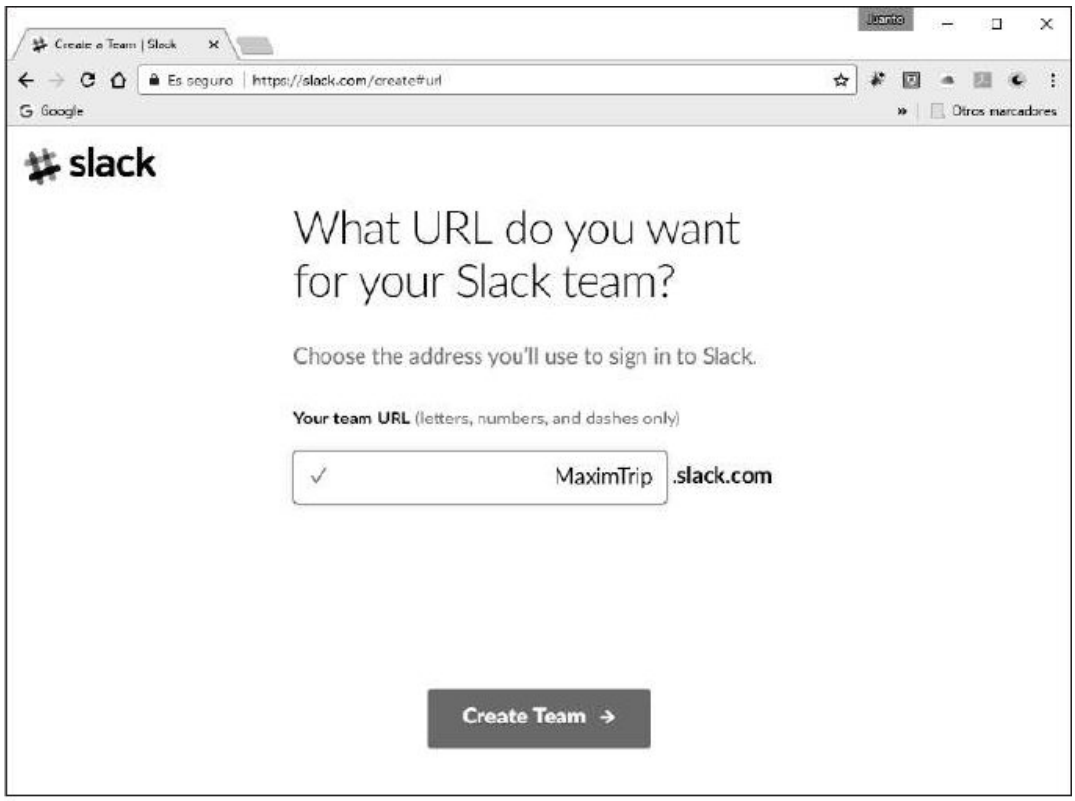
Pulsamos sobre **Continue to Team Info** e indicamos cuál es el objetivo de utilizar un grupo en Slack y el número de personas aproximado que lo compondrá:



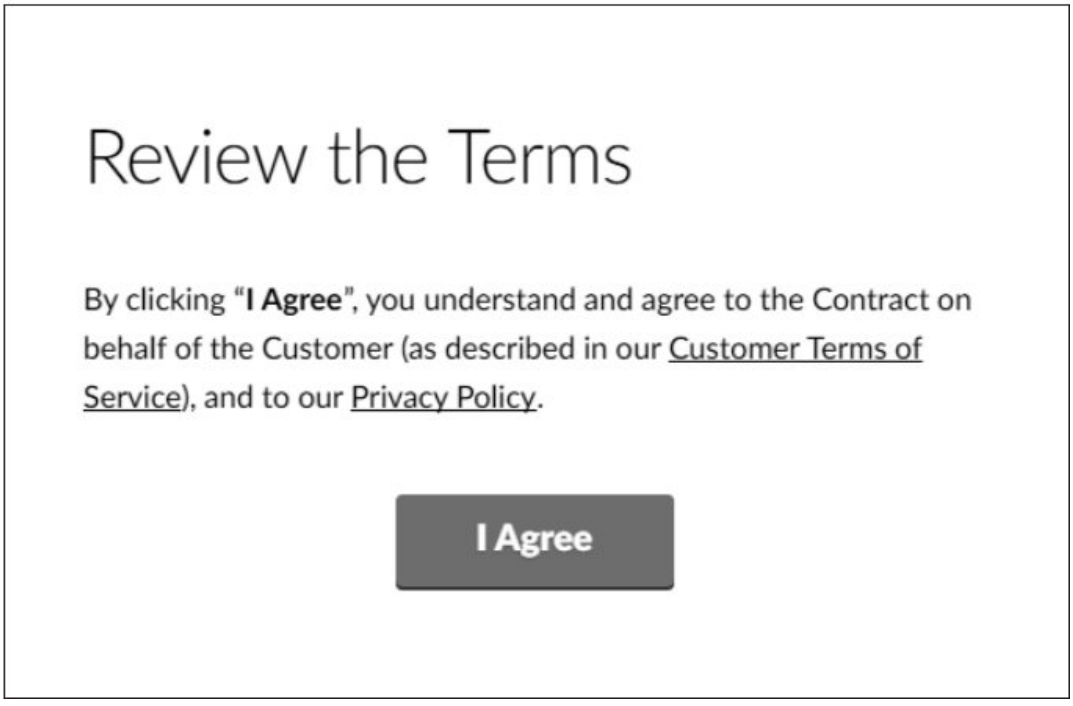
Pulsamos sobre **Continue to Group Name** y elegimos el nombre de nuestro grupo.



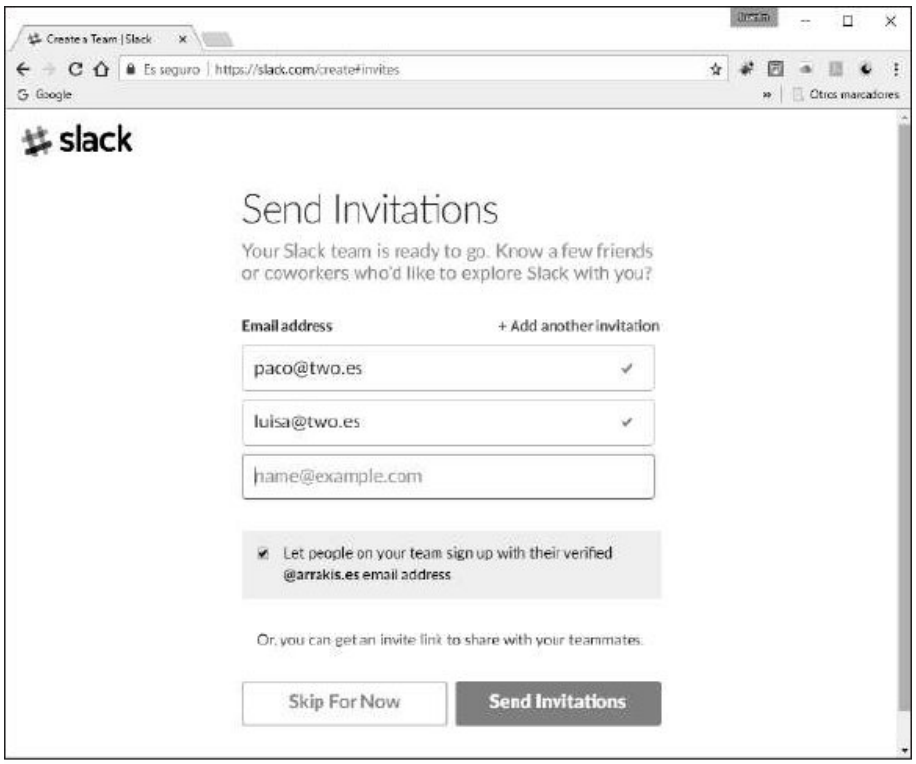
A continuación, se nos solicita una URL para el grupo y se verifica que el nombre de grupo esté disponible:



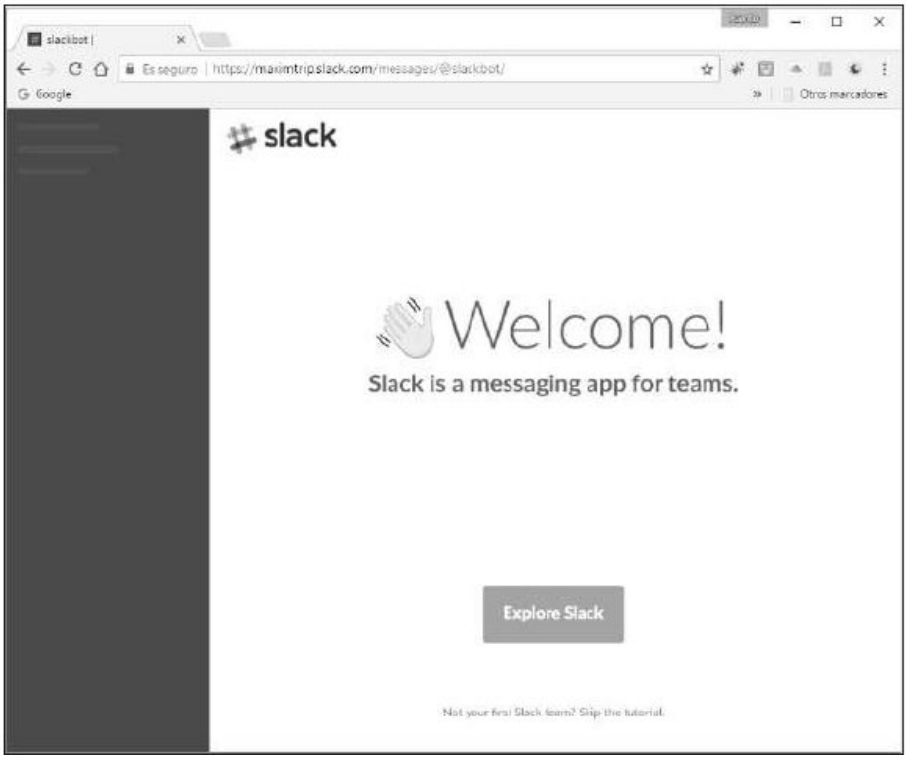
A continuación, debemos revisar los términos del servicio:



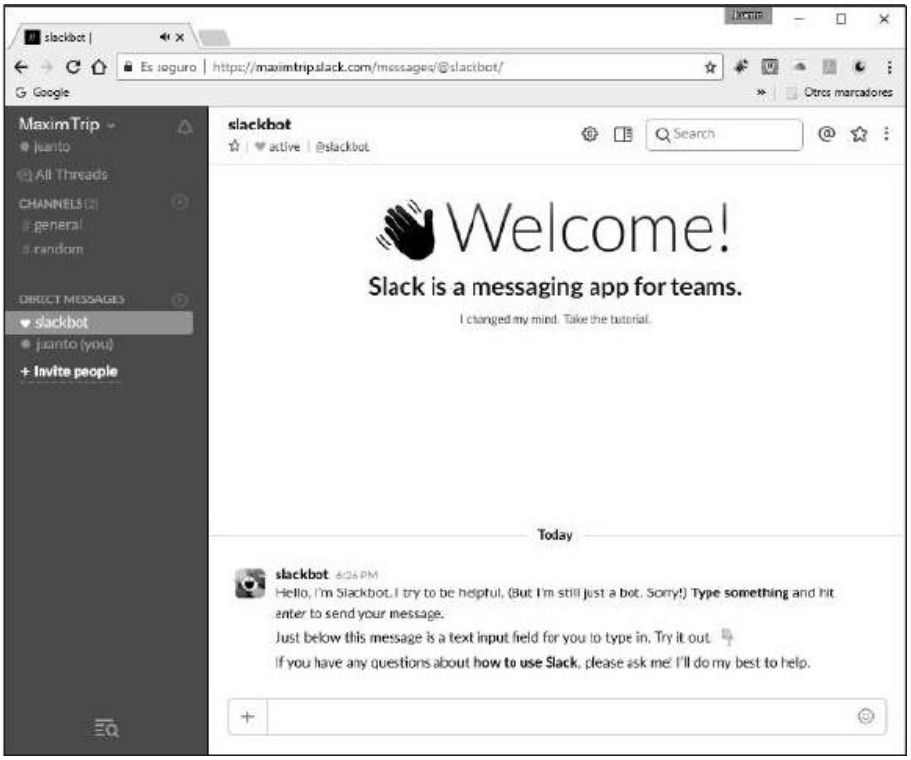
Pulsamos sobre **I Agree** y pasamos a invitar a los componentes del grupo:



Una vez que hayamos enviado las invitaciones (**Send Invitations**) o no (**Skip For Now**), ya tendremos el grupo creado:



A continuación, podemos dedicar unos minutos a seguir el tutorial o bien pulsar sobre **Skip the tutorial** para acceder directamente a nuestra pantalla de trabajo:


slackbot |
Es seguro | https://maximtrip.slack.com/messages/@slackbot/
Google
Otros marcadores
MaximTrip
juanto
All Threads
CHANNELS
general
random
DIRECT MESSAGES
slackbot
juanto (you)
+ Invite people
slackbot
active | @slackbot
Search
Welcome!
Slack is a messaging app for teams.
I changed my mind. Take the tutorial.
Today
slackbot 6:26 PM
Hello, I'm Slackbot. I try to be helpful. (But I'm still just a bot. Sorry!) Type something and hit enter to send your message.
Just below this message is a text input field for you to type in. Try it out.
If you have any questions about how to use Slack, please ask me! I'll do my best to help.

CAPÍTULO 12:
Bitbucket

Bitbucket es una herramienta proporcionada por Atlassian, una empresa australiana que crea software enfocado a desarrolladores y empresas. En concreto, Bitbucket es una herramienta creada para el control de versiones de uno o varios proyectos software, basada en el uso de **Git** de forma colaborativa entre los miembros de un equipo.

Git proporciona una serie de herramientas que permiten a un equipo de desarrolladores trabajar sobre un mismo proyecto software (en nuestro caso, un proyecto Android) de modo que, de forma colaborativa, todos los miembros del equipo puedan trabajar sobre el mismo código fuente, modificando y añadiendo aquellos ficheros que lo requieran, de tal forma que el código fuente se actualiza diariamente.

Esta herramienta permite trabajar, de forma gratuita, con equipos de hasta cinco personas. Para nuestro grupo de trabajo y el de muchas organizaciones, un equipo de cinco personas es muy común en el ámbito del desarrollo de proyectos móviles y, por tanto, nos parece interesante utilizarlo para nuestro propósito. Cabe mencionar que, si nuestra organización o equipo se amplía a más de cinco miembros, deberemos realizar el pago por uso de esta herramienta, aunque esta nos parece tan buena que no debería ser un problema pagar por una herramienta que facilita tanto la vida a los desarrolladores.

Existen una serie de conceptos que debemos comprender a la hora de utilizar una herramienta de control de versiones de código fuente:

- **Espacio de trabajo o workspace.** Espacio en el cual realizamos el desarrollo del código fuente. En este espacio se encuentran los ficheros del proyecto, así como un directorio, llamado **.git**, que contiene la estructura de archivos del proyecto.
- **Índice.** Espacio utilizado para organizar los ficheros, pertenecientes al proyecto, sobre los que queramos mantener un control de versiones. Para ello preparamos el fichero para ser incluido en el siguiente commit, añadiendo cambios al repositorio local.
- **Repositorio local.** Espacio local de nuestro equipo, donde se encuentran todos los ficheros que pasan del estado del "Index" al estado "Head", tras realizar un commit. Todos estos cambios se aplicarán en el repositorio remoto, mediante un push.
- **Repositorio remoto.** Espacio remoto donde se encuentra nuestro código fuente y el de todos los integrantes del equipo. En nuestro caso será Bitbucket, sin embargo, puede ser GitLab o cualquier otro repositorio.

Para crear una cuenta Bitbucket, debemos ir a la dirección web: https://bitbucket.org/account/signup.

Una vez creada la cuenta con un correo electrónico y contraseña válidos y verificado la cuenta en el correo recibido, nos pedirá que introduzcamos un nombre de usuario para la cuenta Bitbucket.

¡Perfecto! Ya estamos listos para crear equipos de trabajo, repositorios Git o diferentes proyectos.

Para crear equipos de trabajo, simplemente necesitaremos que el administrador invite a otros usuarios, ya registrados en Bitbucket o que sean invitados a través de un correo electrónico. En nuestro caso, nuestro grupo se compone de tres integrantes, de modo que los añadiremos al equipo.

Para crear un proyecto en Bitbucket, debemos utilizar la consola de Git. Para ello debemos instalar Git en nuestro equipo. En caso de no disponer de él en nuestro equipo, visitaremos la página https://git-scm.com/download, seleccionare-

mos el sistema operativo donde instalaremos Git y, una vez descargado, iniciaremos el instalador, todo con la configuración por defecto, pulsando sobre Siguiente hasta finalizar la instalación.

Una vez que tenemos Git instalado, podemos partir de un directorio de nuestro equipo para inicializar un repositorio o clonar desde un repositorio remoto de Bitbucket, por ejemplo.

Inicializar Git desde un directorio existente

Si hemos creado nuestro proyecto desde Android Studio, bastaría con situarnos en el directorio del proyecto *C:\ProyectosAndroid\HotelsRepo* que hemos creado mediante la consola de Windows.

Una vez que estamos en la ruta donde se encuentra nuestro proyecto, debemos crear la estructura del repositorio Git, que se almacenará en un directorio llamado **.git**. Para ello debemos ejecutar el comando **git init** para inicializar el repositorio, tal como se muestra en la imagen.

Fig. 1.12.1.1 Git init.

Una vez que tenemos inicializado el repositorio, se creará la carpeta **.git** con la estructura del repositorio.

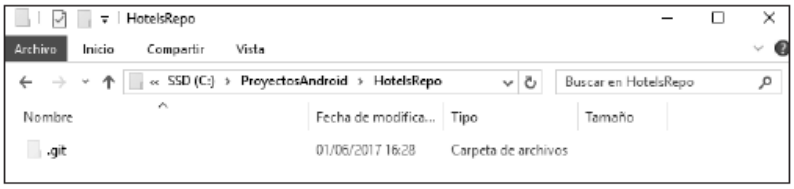


Fig. 1.12.1.2 Carpeta .git.

Debemos asociar el directorio local con nuestro repositorio remoto existente en Bitbucket. Este nombre lo encontramos en la pantalla resumen del repositorio.

Copiamos el nombre https://eclipselunar@bitbucket.org/masterajr/hotelsrepo.git.

Fig. 1.12.1.3 Git init.

Una vez que tenemos el nombre del repositorio remoto, conectamos nuestro directorio local con el repositorio remoto, mediante el comando **git remote add [alias] [URL]**, siendo el alias un nombre clave del repositorio, y URL, la dirección del repositorio remoto.

Fig. 1.12.1.4 Conectar con repo.

Como tenemos el directorio vacío, debemos crear algún fichero, por ejemplo un **Nuevo documento de texto.txt**. En el caso de tener los archivos del proyecto, no sería necesario.

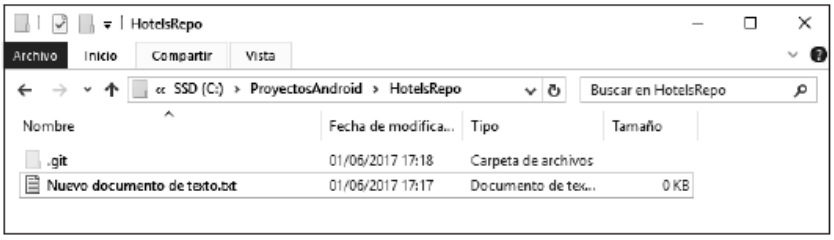


Fig. 1.12.1.5 Nuevo fichero en carpeta.

Ahora que ya disponemos de algún fichero nuevo para subir, que únicamente existe localmente, es el momento de añadir aquellos ficheros que queramos subir al repositorio. Para ello utilizaremos el comando **git add .**, el cual añadirá aquellos ficheros en estado creado o modificado.

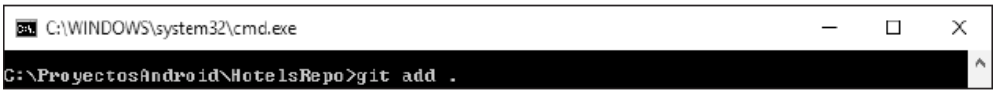


Fig. 1.12.1.6 Git add.

Una vez añadidos todos los ficheros con estado creado o modificado, estamos listos para añadir los cambios al repositorio local. Para ello utilizaremos el comando **git commit –m #DESCRIPCIÓN#**.

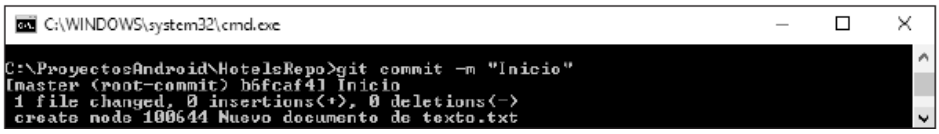


Fig. 1.12.1.7 Git commit.

Descargado en: eybooks.com

Finalmente subimos al repositorio remoto los archivos añadidos en el último commit realizado, mediante el comando **git push –u origin master**.

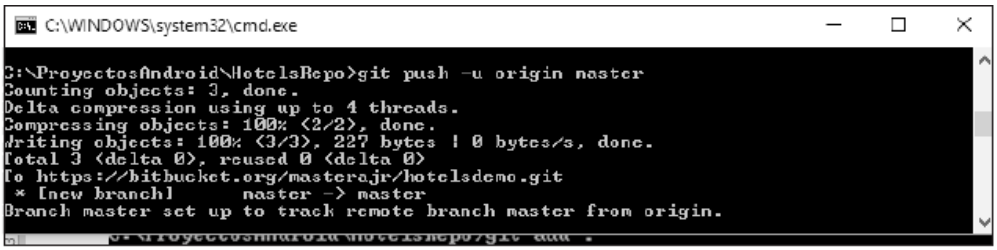


Fig. 1.12.1.8 Git push.

Como podemos observar, tenemos el fichero subido en el repositorio remoto de Bitbucket.

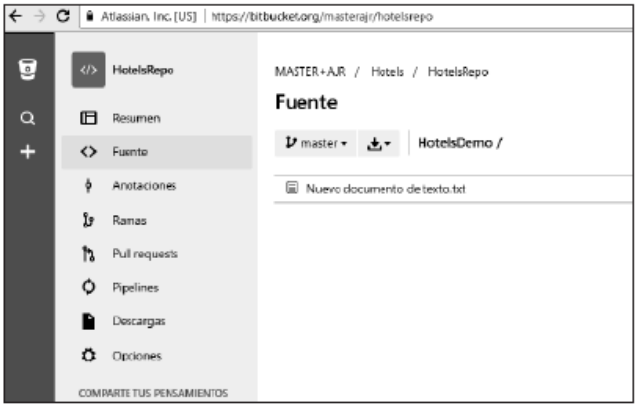


Fig. 1.12.1.9 Bitbucket.

Clonar repositorio en un directorio

Para clonar un repositorio remoto ya existente, debemos copiar la dirección de dicho repositorio, por ejemplo, nuestro repositorio de prueba de Bitbucket:

https://eclipselunar@bitbucket.org/masterajr/hotelsrepo.git.

Mediante la consola de comandos de nuestro equipo nos dirigimos al directorio donde queramos bajarnos este repositorio, y ejecutamos el siguiente comando:

```
git clone https://eclipselunar@bitbucket.org/masterajr/hotelsrepo.git
```

Y con esto ya tendríamos el repositorio clonado en nuestro equipo y listo para abrirlo, usarlo y modificarlo.

Integrar Bitbucket con Slack

Tal y como vimos en el capítulo anterior, Slack es una herramienta muy potente que permite la comunicación entre los integrantes del equipo, la subida de documentos importantes, reuniones y videoconferencias.

Pero eso no es todo, podemos hacer que cada vez que subamos código al repositorio remoto, es decir, cada vez que hagamos un push, se les notifique a los miembros del equipo a través de Slack, por medio de una notificación de un hook lanzado por Bitbucket.

Para crear esta integración, iniciamos sesión en Slack y creamos un equipo o elegimos uno ya existente.

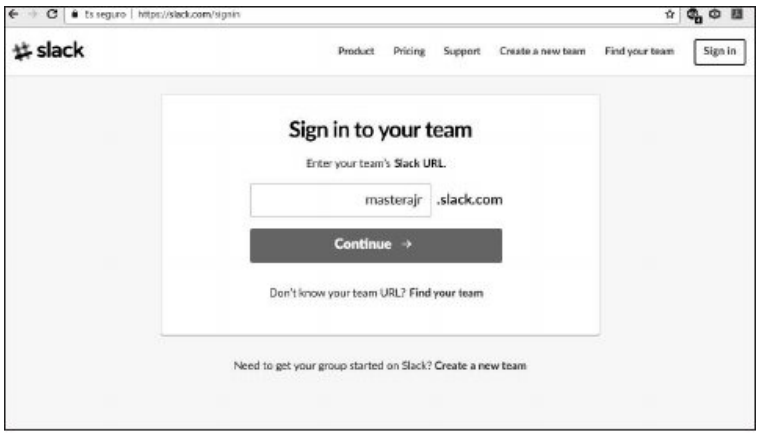


Fig. 1.12.3.1 Login Slack.

Una vez en Slack, creamos un canal de comunicación, le asignamos el nombre que queramos (en nuestro caso, "hotels") e indicamos los integrantes del canal. Tras esto, se creará el canal y veremos el chat.

En el icono de configuración del canal, se desplegará un menú, donde seleccionaremos **Add an app or integration**.

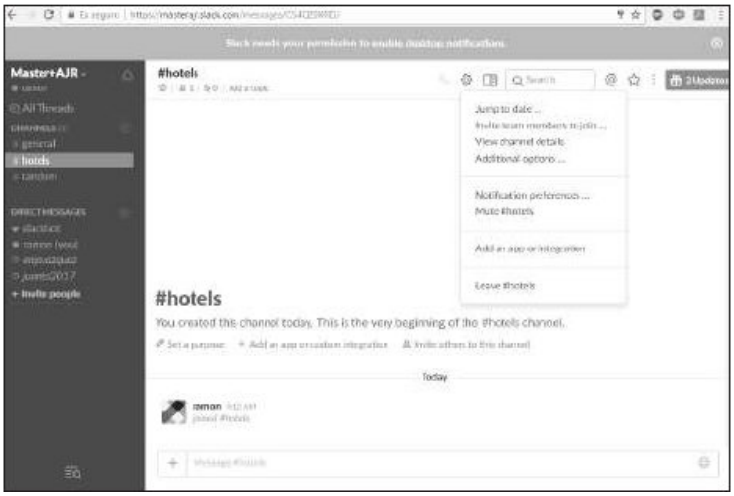


Fig. 1.12.3.2 Canal creado.

Tras esto, deberemos seleccionar la aplicación con la cual queremos integrar Slack, en este caso será con Bitbucket. La elegimos y pulsamos sobre el botón **Install**. Esto nos permitirá realizar una nueva configuración, donde seleccionaremos el canal por el cual queremos que Bitbucket nos notifique y pulsamos sobre **Add Bitbucket Integration**. Esto nos proporcionará una URL, que debemos copiar.

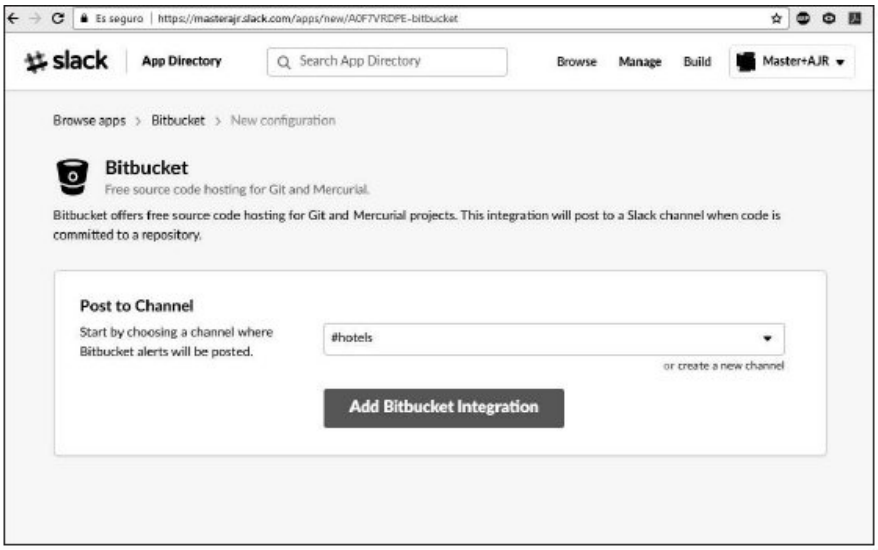


Fig. 1.12.3.3 Obtener URL de Slack.

Una vez copiada la URL proporcionada por Slack, abrimos Bitbucket. Allí, abrimos el repositorio sobre el cual queremos recibir notificaciones de subidas de código y abrimos las opciones del repositorio.

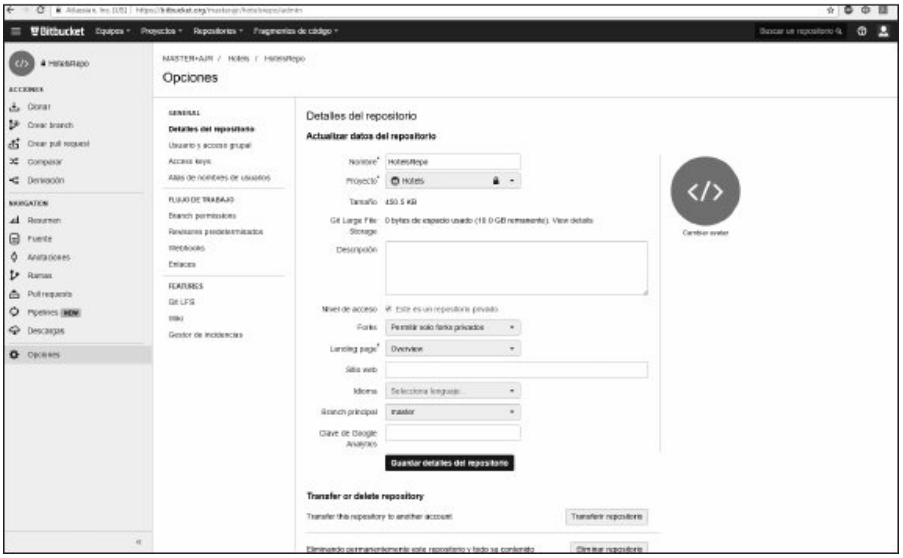


Fig. 1.12.3.4 Opciones de Bitbucket.

Una vez en esta pantalla de configuración, pulsamos sobre la opción **Webhooks** y crearemos un Webhook pulsando sobre **Add new webhook**. Un webhook realiza

una petición http a la URL que le indiquemos, de forma que las respuestas de esta petición serán tratadas asíncronamente.

Será aquí donde incluiremos la URL que copiamos de Slack anteriormente y, tras guardar esta configuración, el Webhook estará activado.

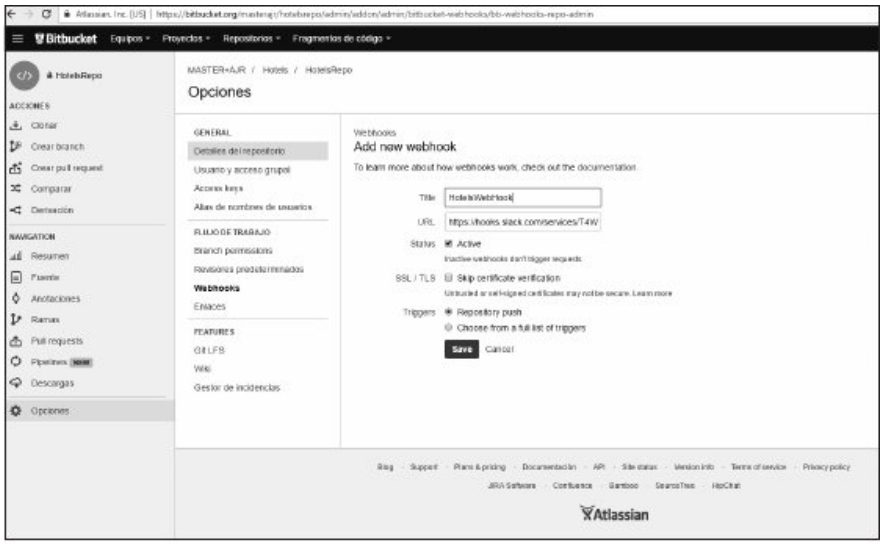


Fig. 1.12.3.5 Bitbucket Webhook.

Volvemos a Slack y veremos un mensaje que nos informa de que nos hemos integrado correctamente.

Fig. 1.12.3.6 Bitbucket Webhook.

Ahora, cada vez que subamos código al repositorio, se nos notificará a través del chat del canal que hemos sincronizado con Bitbucket. Esto es muy interesante, ya que, tras recibir una notificación en el chat, podremos comentarlo rápidamente con los miembros del equipo, así como estar al tanto de cada subida que se realice.

PARTE 2:
PROYECTOS DE PRUEBA

CAPÍTULO 1:
Proyecto base Dagger 2

Para entender bien Dagger, no hay mejor forma que trabajar con un proyecto básico de ejemplo cuya única función sea, por ahora, la de arrancar la aplicación configurada con Dagger sin errores. Este proyecto de ejemplo, llamado **Dagger2**, podemos encontrarlo en el código proporcionado por la editorial.

Lo primero es crear nuestra aplicación base con el nombre que queramos, en nuestro caso, Dagger2 (Figura 2.1.1). Opcionalmente podemos crearle una Activity principal vacía (Figura 2.1.2), ya que el objetivo no es trabajar en la parte gráfica de la interfaz, como hemos comentado, sino que nuestra aplicación arranque a nivel funcional.

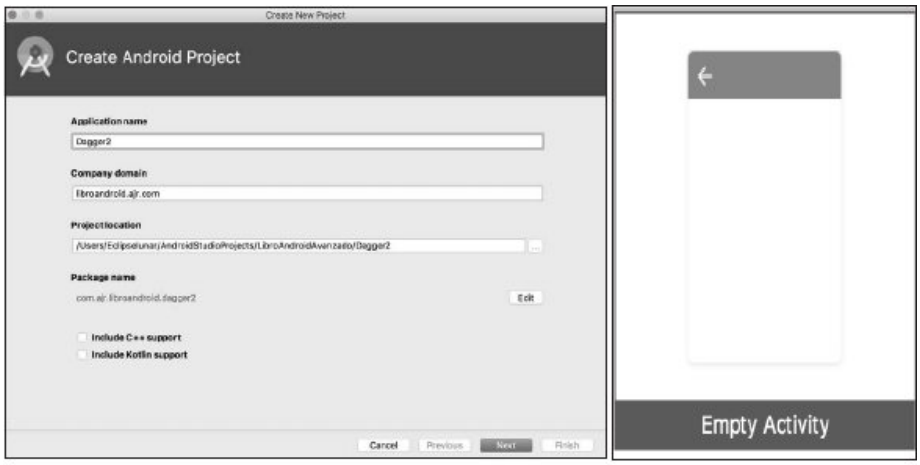


FIG. 2.1.1. Creación de nueva aplicación. FIG. 2.1.2. Creación de Activity vacía.

Asignaremos **MainActivity** como nombre de la actividad, es decir, el nombre que viene por defecto en la creación de nueva actividad (Figura 2.1.3).

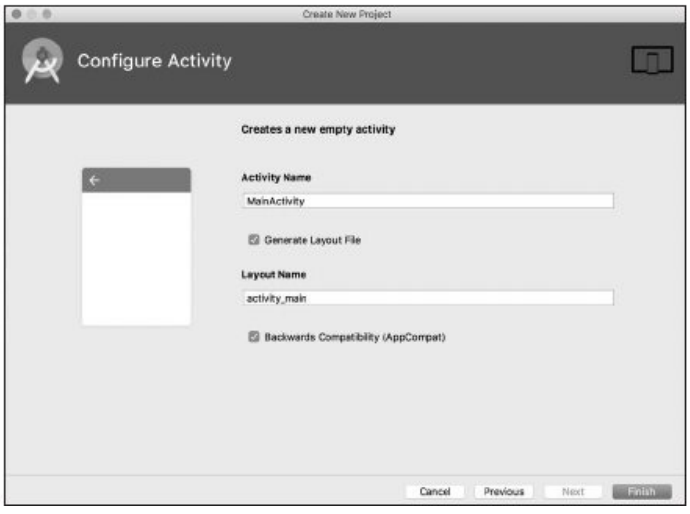


Fig. 2.1.3. Nombre de Activity

Ya tenemos creado el proyecto. Configurar correctamente Dagger 2, en nuestro proyecto Android, es muy sencillo, basta con seguir estos pasos:

1. En el fichero **build.gradle** del proyecto Android añadimos la dependencia del plugin **android-apt**, para habilitar el uso de Dagger 2.

```
dependencies {
…
classpath 'com.neenbedankt.gradle.plugins:android-apt:1.8'
}
```

Sincronizamos Gradle para aplicarlo, mediante **Sync now**.

2. Además debemos aplicar el plugin en el fichero **build.gradle** del módulo; para ello, añadiremos:

```
apply plugin: 'com.neenbedankt.android-apt'
```

3. Sin embargo, toda la configuración anterior únicamente es necesaria para versiones de Gradle inferiores a 2.2. A partir de la versión de Gradle 2.2, el plugin **adt** ya no es necesario; para usar Dagger bastará con añadir, en las dependencias del fichero **build.gradle** del módulo, la API pública de Dagger 2:

```
dependencies {
…
compile 'com.google.dagger:dagger:2.10'
…
}
```

4. Añadimos, en las dependencias, el procesado de anotaciones realizado anteriormente por el plugin **android-apt** y que ahora lo realiza este plugin:

```
dependencies {
…
annotationProcessor 'com.google.dagger:dagger-compiler:2.10'
…
}
```

Sincronizamos Gradle para aplicarlo, mediante **Sync now**.

En estos momentos ya tendríamos configurada la parte de paquetes de Dagger; ahora hay que crear la estructura base sobre la que construir nuestro proyecto.

Para agrupar todo lo relativo a la inyección de dependencias, creamos un paquete en el proyecto con el nombre **di**. Será aquí donde creemos los módulos y componentes de la aplicación. Y por ende, crearemos adicionalmente dos paquetes, nombrados **module** y **component**, ubicados dentro del paquete **di**.

Ya estamos listos para crear nuestro primer Módulo. Recordemos que será en los módulos donde Dagger gestionará las dependencias. Para ello creamos una clase con el nombre que queramos, por ejemplo **AppModule**, y le asignamos la anotación **@Module** encima de la definición de la clase:

```
@Module
public class AppModule { … }
```

En los módulos debemos definir todas las variables u objetos, así como métodos que provean dependencias y que formen parte de la cadena de dependencia.

Dentro de los módulos, los métodos que retornen un valor deberán incluir la anotación **@Provides**. Para facilitar la comprensión, es recomendable que estos métodos empiecen por la palabra *provide*, para indicar que nos proveen de un objeto usado en la inyección de dependencias.

La anotación **@Singleton** establece que una instancia del objeto obtenido para la inyección de dependencias deberá ser creada y compartida únicamente una vez en la aplicación.

Todo esto podemos verlo mejor en el siguiente ejemplo: en la clase **AppModule** creada anteriormente, definimos los métodos que proporcionan dependencias; en nuestro caso, proveemos de la dependencia de **Application**. Para ello creamos el método **provideApplication()**. Como observamos, incluye el prefijo "provide" + "la dependencia que se provee", en este caso, el contexto de la aplicación.

Además hemos utilizado la anotación **@Provides** para indicar que es un método proveedor de dependencias; así como la anotación **@Singleton**, que indica el tiempo de vida de la dependencia. En este caso, su alcance o scope se aplicará al tiempo de vida de la aplicación.

```
@Module
public class AppModule {

    private Application application;

    public AppModule(Application application)
    { this.application = application; }

    @Provides
    @Singleton
    public Application provideApplication(){ return this.application; }
}
```

El siguiente paso consiste en indicar dónde queremos inyectar la dependencia. Este paso lo realizamos en una interfaz inyectora, llamada **componente**, encargada de referenciar los objetos Singleton a las Activity.

Estos componentes deben incluir la anotación **@Component**, así como asignar un parámetro a la anotación, denominado como **modules**, en el que le asignamos el o los módulos que le pertenecen al componente, en nuestro caso, el módulo **AppModule**.

Las Activity que utilicen el componente deberán ser añadidas en esta interfaz, con métodos **inject**. Es decir, los componentes actúan de puente entre el módulo e inyección de dependencia.

```
@Singleton
@Component(modules = AppModule.class)
public interface AppComponent { void inject (MainActivity mainActivity); }
```

Una vez que tenemos el módulo y componente definidos, podemos crear una clase que herede de **Application** para el uso de Dagger.

Para ello creamos, en la raíz del proyecto, una clase llamada **App** que herede de **Application**, con un método que retorne el componente de la aplicación, **AppComponent**, llamado **getAppComponent()**.

La estructura del proyecto debería parecerse a esta imagen:

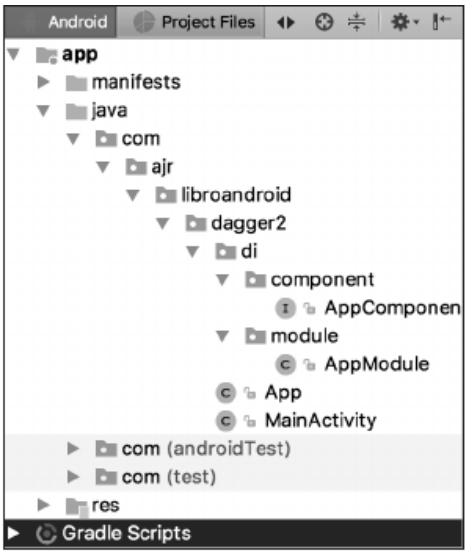


Fig. 2.1.4. Estructura proyecto Android Dagger 2.

Por otro lado sobrescribiremos el método **onCreate()** y obtendremos la instancia del componente. Esta instancia del componente puede requerir un módulo; en nuestro caso, será el módulo de la App **appModule**, cuyo módulo constructor requiere como parámetro el contexto actual; en este caso, nuestra Application **App**, accesible mediante **this**. Una vez instanciado el módulo, mediante el método **build()** construimos el componente, que posteriormente se asignará a la variable local **AppComponent**.

```
public class App extends Application {
    private AppComponent appComponent;
    @Override
    public void onCreate()
    {
        super.onCreate();
        appComponent= DaggerAppComponent
                      .builder()
                      .appModule(new AppModule(this))
                      .build();
    }
    public AppComponent getAppComponent(){ return this.appComponent; }
}
```

Véase que en ningún momento hemos definido **DaggerAppComponent**, sino que esto es generado automáticamente por Dagger, al detectar que **AppComponent** es un componente creado por nosotros que utiliza Dagger. Para que Dagger genere dicho fichero, debemos compilar la aplicación. En ese momento, si todo va bien, nos indicará un mensaje de error de que "DaggerAppComponent" no está

definido. Para solucionarlo, resolvemos con la importación de dicha clase y ya podríamos usarla.

Posteriormente, para que nuestra aplicación **App** sea la primera en ser inicializada, debemos indicarlo en el **AndroidManifest.xml**, dándole el nombre de la aplicación inicial, **android:name=".App"**.

Finalmente, para inyectar nuestra activity al componente, en el método **onCreate()** de nuestra **MainActivity**, obtenemos el contexto de la aplicación mediante un casting, indicando que el contexto es de tipo App, ya que nuestra clase hereda de Application. De esta forma, podemos obtener el componente asociado mediante **getAppComponent()** y le inyectamos el contexto de la MainActivity mediante el método **inject** definido en el componente.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    ((App)getApplication()).getAppComponent().inject(this);
}
```

Pues bien, ya tenemos configurada nuestra primera aplicación que usa Dagger. Ahora ya solo queda profundizar un poco más en el tema Dagger, así como seguir estos pasos cada vez que creemos nuevos módulos o componentes, obteniéndolos en la Activity que corresponda.

Fig. 2.1.5. Resultado de ejecución de la aplicación.

Al ejecutar la aplicación, debería abrirse y mostrarse como en esta imagen, es decir, simplemente debe arrancar sin que se detenga la aplicación.

CAPÍTULO 2:
Proyecto base Dagger v.2.11

Probablemente, a estas alturas, ya te habrás dado cuenta de que todo ha cambiado, y así es, Dagger ha evolucionado respecto a su versión 2.10.

A partir de la versión 2.11 de Dagger, la inyección de dependencias se hace de una manera más limpia y organizada y, a pesar de que mantiene la compatibilidad con la versión anterior, hemos considerado que debíamos realizar un proyecto ejemplo; por tanto, desde este momento, en nuestros próximos proyectos utilizaremos esta versión más reciente.

Crearemos, por tanto, un proyecto básico de ejemplo cuya única función sea arrancar la aplicación configurada con Dagger v.2.13 sin errores. Este proyecto de ejemplo, llamado **Dagger211**, podemos encontrarlo en el código proporcionado por la editorial.

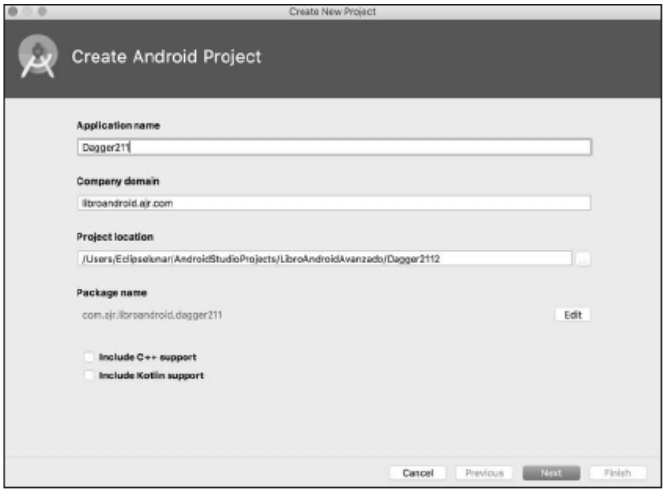


Fig. 2.2.1. Creación de nueva aplicación.

Lo primero es crear nuestra aplicación base con el nombre que queramos, en nuestro caso, **Dagger211** (Figura 2.2.1). Opcionalmente podemos crearle una actividad principal vacía, ya que el objetivo no es trabajar en la parte gráfica de la interfaz, como hemos comentado, sino que arranque nuestra aplicación a nivel funcional.

El nombre de la actividad será **MainActivity**, es decir, el nombre que viene por defecto en la creación de nueva actividad.

En estos momentos, tenemos creado el proyecto. Configurar correctamente Dagger con la versión 2.13, en nuestro proyecto Android, es muy sencillo, basta con seguir estos pasos:

1. Desde la versión de Gradle 2.2, únicamente es necesario añadir en las dependencias del fichero **build.gradle** del módulo, la API pública de Dagger 2, para poder hacer uso de Dagger:

```
dependencies { compile 'com.google.dagger:dagger:2.13' }
```

2. En caso de utilizar las librerías de soporte de Dagger, debemos añadir dicha dependencia a este mismo fichero **build.gradle**:

```
dependencies { compile 'com.google.dagger:dagger-android-support:2.13' }
```

3. El procesado de las anotaciones, que usa Dagger, lo realiza este plugin, que también debemos añadir al **build.gradle**:

```
dependencies {annotationProcessor 'com.google.dagger:
dagger-compiler:2.13' }
```

4. Además añadiremos, en las dependencias, el procesado de anotaciones Android, como por ejemplo, la anotación **@ContributesAndroidInjector**, que explicaremos más adelante:

```
dependencies {
annotationProcessor 'com.google.dagger:dagger-android-processor:2.13'
}
```

Sincronizamos Gradle para aplicarlo, mediante **Sync now**.

En estos momentos ya tendríamos configurada la parte de paquetes de Dagger. Ahora hay que crear la estructura base sobre la que construir nuestro proyecto.

Al igual que en el proyecto anterior, agruparemos todo lo relativo a la inyección de dependencias, creando un paquete en el proyecto con el nombre **di**, que significa "Dependency Injection". Será aquí donde creemos los módulos, componentes, builders o constructores y alcances o scopes de la aplicación. Adicionalmente crearemos los paquetes **module**, **component**, **builder** y **scope**, dentro del paquete **di**.

Empecemos pues con la creación del módulo de la aplicación con el nombre **AppModule**, donde Dagger gestionará las dependencias. Esta clase contendrá la anotación **@Module** encima de la definición de la clase para indicar a Dagger que esta clase es un módulo y que debe tenerla en cuenta a la hora de elaborar el grafo de dependencias. Es en los módulos donde definimos todas las variables u objetos, así como los métodos que provean dependencias y que formen parte de la cadena de dependencia; proveeremos del contexto de la aplicación.

A diferencia de la versión anterior, desde la versión 2.11, los módulos pueden proveer dependencias utilizando la anotación **@Binds** para métodos abstractos, es decir, sin incluir implementación, o bien utilizar **@Provides** como ya vimos, con la peculiaridad de que, si declaramos el módulo como clase abstracta, este método deberá ser estático, es decir **static**, e incluir implementación.

Sin embargo, en módulos no abstractos, no podremos utilizar **@Binds**; continuará siendo válido el uso de **@Provides** sin la necesidad de que sean métodos estáticos. Para el ejemplo, crearemos el módulo de la aplicación de tipo abstracto.

```
@Module
public abstract class AppModule {
    @Provides
    @Singleton
    static Context provideContext(App application){
        return application.getApplicationContext();
    }
}
```

Como podemos observar, proveemos del contexto de la aplicación en un método estático cuyo alcance es **@Singleton**, es decir, a nivel de aplicación.

Como hemos visto en anteriores capítulos, existen diferentes tipos de alcances o scopes (seguro que te suena el **scope @Singleton**). Pues bien, un **@Singleton** se inicializa una única vez a lo largo del ciclo de vida de la aplicación.

Sin embargo, en determinados escenarios, puede que queramos que una dependencia dure el ciclo de vida de una Activity. Para ello, crearemos una interfaz en el paquete **scope**, con el nombre **PerActivity**, reteniéndola en tiempo de ejecución. El alcance se declara mediante la anotación **@Scope**; la dependencia en tiempo de ejecución es retenida por la máquina virtual mediante la anotación **@Retention**.

```
@Scope
@Retention(RUNTIME)
public @interface PerActivity {}
```

Cada una de las Activity o fragments que tengamos en nuestra aplicación deberán definir su propio módulo. En nuestro caso, únicamente tenemos la actividad principal, por tanto, crearemos un módulo dentro del paquete **di/module** del proyecto con el nombre **MainActivityModule**.

Al igual que todo módulo en Dagger, este contiene las dependencias provistas por la actividad. Actualmente no proveeremos de ninguna dependencia; sin embargo, en futuros proyectos, será aquí donde declararemos las dependencias de la vista y del presentador utilizadas por la Activity al emplear el patrón MVP. Para que el ejemplo resulte más claro lo dejaremos vacío, dado que no vamos a crear vistas ni presentadores en este momento.

```
@Module
public abstract class MainActivityModule{...}
```

Una de las novedades introducidas por la nueva versión de Dagger es la anotación **@ContributesAndroidInjector**, que genera por nosotros gran cantidad de código y nos libera de crear componentes con la anotación **@SubComponent**, por cada uno de los componentes que tengamos de cada actividad.

Proveeremos en una nueva clase abstracta cada una de las Activity o fragments que tengamos en nuestra aplicación, dentro del paquete **di/builder**, que llamaremos **BuildersModule**, aunque también podríamos denominarla **ActivitiesBuilderModule**, donde se encontrarían nuestros subcomponentes para que Dagger pudiera inyectarlos.

Únicamente deberemos indicarle a la anotación los módulos que queremos instalar en el subcomponente en cuestión; en nuestro caso, será el módulo de la actividad principal **MainActivityModule**, lo que nos proporcionará el subcomponente con las dependencias de la activity, como por ejemplo, la vista y presentador utilizado.

En caso de tener más activities o fragments en la aplicación, serían declaradas en esta misma clase **BuildersModule**. Además asignaremos un alcance o tiempo de vida de la dependencia, a nivel del ciclo de vida de la activity, mediante el scope **@PerActivity**, definido por nosotros anteriormente.

```
@Module
public abstract class BuildersModule {
    @PerActivity
    @ContributesAndroidInjector(modules = MainActivityModule.class)
    abstract MainActivity contributeMainActivity();
}
```

Nuestra aplicación debe tener su propio componente, que será utilizado por Dagger en la tarea de elaborar el grafo de inyección de dependencias. Por tanto, debemos crear una interfaz, con el nombre **AppComponent**, que incluirá la anotación **@Component** como en la versión anterior.

No obstante, a la hora de incluir los módulos de los que depende el componente, inicialmente debemos instalar el **AndroidSupportInjectionModule**. Con este módulo nos aseguramos de que todo aquello que enlacemos a nivel de dependencias se encuentre disponible.

Seguiremos incluyendo el módulo de la aplicación **AppModule** y, como novedad, incluiremos la clase creada anteriormente, **BuildersModule**, donde definíamos la inyección de los subcomponentes de las Activity.

```
@Singleton
@Component(modules = { AndroidSupportInjectionModule.class,
            AppModule.class, BuildersModule.class })
public interface AppComponent {

    @Component.Builder
    interface Builder {

        @BindsInstance
        Builder application(App application);

        AppComponent build();
    }

    void inject(App app);
    Context context();
}
```

Si observas el código anterior, verás que hemos incluido una anotación un tanto extraña **@Component.Builder**; pues bien, esto no es otra cosa que un constructor que nos permite definir cómo enlazamos el componente de la aplicación **AppComponent** con nuestra aplicación.

Mediante la anotación **@BindsInstance**, cargamos una instancia de la aplicación en el componente; será aquí donde tendremos cargada dicha instancia para ser posteriormente inyectada.

Definimos, además, el método constructor del componente **build()**, mediante el cual el componente de la aplicación será retornado ya construido.

Por último definimos el método **inject(App)** en el componente, para inyectar nuestra aplicación. Y por tanto nuestro componente ya estará definido y podrá ser utilizado por Dagger.

Una vez que tenemos el módulo y componente de la aplicación definidos, podemos crear una clase de nombre **App** en la raíz del proyecto, que herede de **Application**.

Como diferencia respecto de la versión anterior, para poder inyectar las Activity, debemos implementar en esta clase, a interfaz **HasActivityInjector**, la cual proporciona un **DispatchingAndroidInjector<Activity>**, que indica que las Activi-

ty participan en la inyección de dependencias de Dagger en la aplicación, lo cual nos interesa en Android, ya que trabajamos con activities.

Así pues, al implementar la interfaz **HastActivityInjector**, estamos obligados a sobrescribir el método **activityInjector()** y retornar un objeto de tipo **DispatchingAndroidInjector<Activity>** inyectado previamente en la aplicación mediante **@Inject**.

```
public class App extends Application implements HasActivityInjector {
    @Inject
    DispatchingAndroidInjector<Activity> dispatchingAndroidInjector;

    @Override
    public void onCreate()
    {
        super.onCreate();

        DaggerAppComponent
                .builder()
                .application(this)
                .build()
                .inject(this);
    }

    @Override
    public AndroidInjector<Activity> activityInjector() {
        return dispatchingAndroidInjector;
    }
}
```

Si observamos el código anterior, en el método **onCreate()** construimos el componente de la aplicación mediante el constructor **@Component.Builder** que habíamos definido en el componente de la aplicación.

Para ello escribimos el nombre "Dagger" + "Componente de la aplicación", es decir, **DaggerAppComponent**. Fíjate en que, en ningún momento, hemos definido la clase **DaggerAppComponent**, sino que es generada automáticamente por Dagger si todas las dependencias están correctamente definidas, en tiempo de compilación, al detectar que **AppComponent** es un componente creado por nosotros que utiliza Dagger.

Para que Dagger genere dicho fichero, debemos compilar la aplicación. En ese momento, si todo va bien, nos indicará un mensaje de error de que "DaggerAppComponent" no está definido. Para solucionarlo, resolvemos con la importación de dicha clase y ya podríamos usarla.

A partir de ahora, podemos utilizar el builder creado en el AppComponent para construir el componente, llamando al método **application(this)**, e inyectándole la instancia de la aplicación a dicho componente. Posteriormente utilizaremos el método **build()** para obtener el componente de la aplicación correctamente construido.

Para finalizar, inyectamos nuestra aplicación en el método **inject()** del componente de la aplicación. Y con esto ya tendríamos construida la aplicación Dagger.

Si hemos seguido los pasos anteriores, la estructura del proyecto debería asemejarse a la siguiente imagen

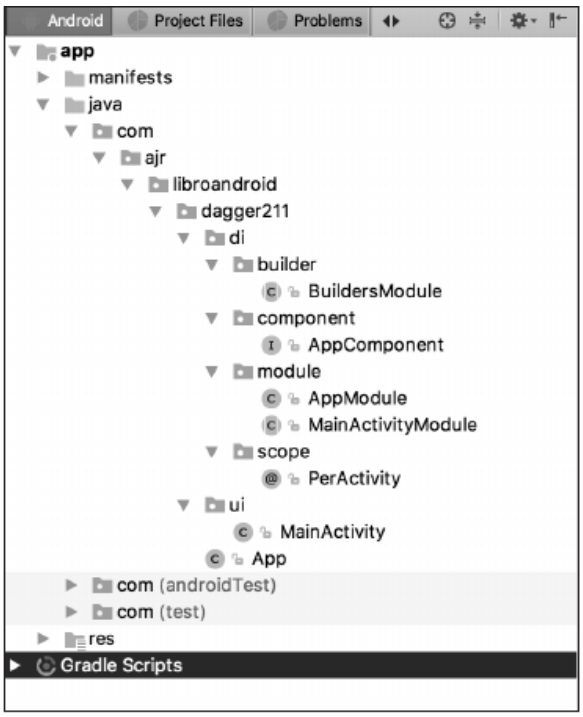


Fig. 2.2.2. Estructura proyecto Android Dagger 2.11.

Uno de los últimos pasos que nos quedan por realizar es establecer, en el AndroidManifest.xml, el nombre de la aplicación inicial; en nuestro caso, **App**, en el tag de <application></application>, con el label **android:name=".App"**, para que nuestra aplicación **App** sea la primera en ser inicializada por la aplicación y por Dagger.

Finalmente, inyectaremos nuestra actividad principal a la aplicación mediante el método **inject** de la clase inyectora **AndroidInjection**. Esto será definido en el método **onCreate()** de nuestra **MainActivity**, antes de llamar al **super()**, es, decir antes de que ocurra la inyección.

Esta inyección se producirá en aquella clase que implemente el **AndroidInjection**, es decir, la interfaz **HasActivityInjector** implementada en nuestra clase **App**, a través del método sobrescrito **AndroidInjector<Activity> activityInjector**.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
        AndroidInjection.inject(this);

        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

Pues bien, ya tenemos configurada nuestra primera aplicación que usa Dagger versión 2.11 o superior. Como puedes apreciar, su presentación es mucho más organizada y limpia que en versiones anteriores. Ahora ya solo queda profundizar un poco más en el tema Dagger, así como seguir estos pasos cada vez que creemos nuevos módulos o componentes.

Si ejecutamos la aplicación, debería abrirse y mostrarse como en la imagen; es decir, simplemente que arranque sin que se detenga la aplicación.

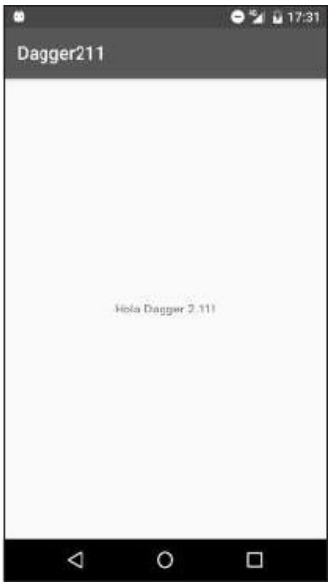


Fig. 2.2.3. Resultado de ejecución de la aplicación.

CAPÍTULO 3:
Proyecto Clean MVP con Dagger 2.11 y RxJava

El propósito de este capítulo es construir un proyecto base, utilizando una arquitectura limpia que utilice Dagger 2 para gestionar las dependencias. Construiremos la arquitectura basándonos en el patrón MVP (Modelo, Vista, Presentador), mediante el cual crearemos cada uno de los elementos que intervienen en el sistema. Adicionalmente, creemos interesante añadir una API que facilite el flujo de datos, como RxJava (Reactive Java), empleado en este caso para la comunicación entre distintas capas del proyecto.

El propósito funcional de este proyecto es construir un diccionario traductor de palabras; en nuestro caso, será un traductor de inglés a español, aunque puede ser del idioma que queramos.

La arquitectura del sistema estaría reflejada en el siguiente diagrama:

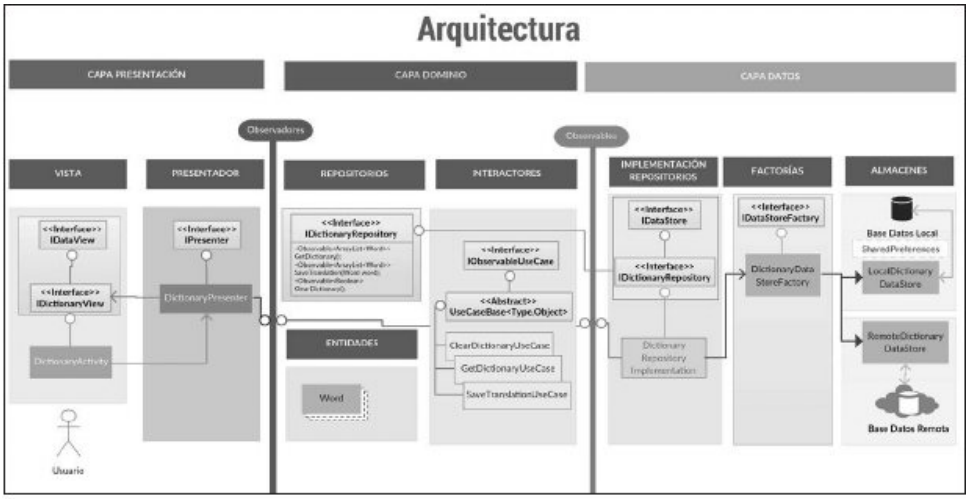


Fig. 2.3.1. Arquitectura.

Empecemos con la creación de un nuevo proyecto limpio, al que daremos el nombre de **CleanMVPDagger211**. Crearemos además una Activity principal vacía de nombre **DictionaryActivity**. Por simplificar, esta será la única Activity que tendremos en nuestra aplicación. Este proyecto de ejemplo, llamado CleanMVPDagger211, podemos encontrarlo en el código proporcionado por la editorial.

Hasta ahora no hemos creado mucho; lo primero que haremos será dividir el proyecto en tres módulos. El primero de ellos, creado por defecto con el nombre **app**, lo renombraremos con el nombre **presentation**, indicando que el módulo contendrá todo lo relativo a la capa de presentación. Para ello realizamos un **Refactor** sobre el nombre del módulo y cambiamos "app" por "presentation", como observamos en las imágenes (Figura 2.3.2) y (Figura 2.3.3).

Fig. 2.3.2. Cambiar el nombre del módulo.

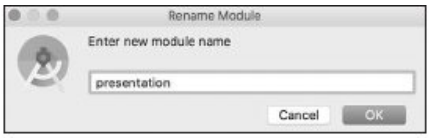


Fig. 2.3.3. Asignar el nombre.

Completaremos la estructura de la arquitectura del sistema con la creación de los dos módulos restantes. Para el bloque del dominio, añadiremos un nuevo módulo al proyecto (Figura 2.3.4), de tipo **Librería Java** (Figura 2.3.5), y le asignaremos el nombre **domain**, ya que se trata del módulo de la capa del dominio del sistema, independiente del framework de Android.

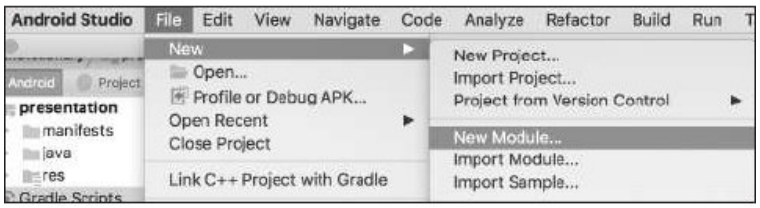


Fig. 2.3.4. Agregar nuevo módulo.

Fig. 2.3.5. Librería Java.

Finalizamos la división de bloques creando la capa de acceso a datos. Creamos un nuevo módulo llamado **data**, de tipo **Librería Android** (Figura 2.3.6), ya que necesitaremos tener acceso a red y al contexto de la aplicación.

Fig. 2.3.6. Librería Android.

Ahora debemos crear las dependencias que existen entre los módulos acabados de crear. Empecemos por el módulo responsable del acceso a datos, es decir, el módulo **data**.

En el fichero **build.gradle** de este módulo, añadiremos la dependencia con el módulo **domain**, debido a la necesidad de convertir los objetos recibidos al objeto del dominio y viceversa.

```
dependencies { compile project(':domain') }
```

De igual manera ocurre con el módulo **presentation**, aunque este contiene una dependencia con el módulo **data** para el acceso a la capa de datos, así como una dependencia con el módulo **domain** al representar los objetos del dominio.

Por tanto debemos añadir, en el fichero **build.gradle** del módulo **presentation**, la dependencia a ambos módulos.

```
dependencies {
compile project(':domain')
compile project(':data')
}
```

Finalmente sincronizamos el proyecto, que no debería darnos ningún error de configuración.

Acabamos de construir la base estructural del proyecto y ya estamos listos para seguir ampliando horizontes.

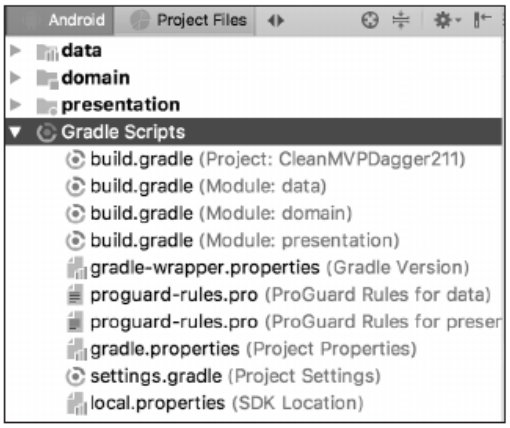


Fig. 2.3.7. Estructura del proyecto.

En el fichero build.gradle del módulo **presentation**, configuraremos todo lo relativo al framework de Dagger versión 2.13. Para ello añadiremos las dependencias de la API pública de Dagger 2, así como las anotaciones que permitan a Dagger interpretar nuestras clases como clases que intervienen o no en el grafo de dependencias de Dagger. Estas librerías nos permiten usar Dagger:

```
dependencies {
        compile 'com.google.dagger:dagger:2.13'
        annotationProcessor 'com.google.dagger:dagger-compiler:2.13'
        annotationProcessor 'com.google.dagger:dagger-
```

```
        android-processor:2.13'
        compile 'com.google.dagger:dagger-android-support:2.13'
}
```

Dado que ya hemos visto estas librerías en el capítulo anterior, no entraremos en detalle sobre su funcionalidad.

Además de añadir estas dependencias, añadiremos la dependencia de la API de Dagger en el módulo **domain**, ya que vamos a realizar inyecciones en el dominio mediante la anotación **@Inject**, que veremos más adelante.

```
dependencies {  compile 'com.google.dagger:dagger:2.13' }
```

Tras sincronizar el proyecto, deberían instalarse sin problemas las librerías que acabamos de añadir.

Procedamos a configurar Dagger 2. Para ello realizaremos la misma configuración que en el capítulo anterior sobre Dagger 2 versión 11. Para evitar ser repetitivos, únicamente detallaremos aquello que sea relevante.

Nuestro proyecto únicamente contendrá una actividad, llamada **DictionaryActivity**, creada al inicio del proyecto. Esta debe tener su propio módulo proveedor de dependencias. En el capítulo anterior, este proveedor lo nombramos **MainActivityModule**. Siguiendo con la nomenclatura, crearemos un módulo proveedor de dependencias, para esta Activity, con el nombre **DictionaryModule**, alojado en el paquete **di/module**, donde se encuentran los módulos.

```
@Module
public abstract class DictionaryModule { }
```

Por ahora dejaremos esta clase vacía. Aquí será donde proveamos del presentador y de la vista, que utiliza dicha Activity.

En el paquete **di/builder** deberíamos tener una clase llamada **BuildersModule**, donde definiremos cada una de las inyecciones de nuestras Activity o Fragments. En nuestro caso, dado que tenemos una Activity, declararemos su inyección y el módulo utilizado, en nuestro caso, **DictionaryModule**.

```
@Module
public abstract class BuildersModule {
@PerActivity
    @ContributesAndroidInjector(modules = DictionaryModule.class)
    abstract DictionaryActivity contributeDictionaryActivity();
}
```

En la raíz del módulo **presentation**, creamos la clase **App**, que hereda de **Application**. Esta aplicación se iniciará al arrancar la app y Dagger la utilizará para inicializar el contexto de la aplicación.

Indicamos que las Activity participan en la inyección de dependencias de Dagger mediante el **ActivityInjector**. Más adelante, en **onCreate()** de esta clase, instanciaremos el componente de la aplicación; de momento, lo dejamos vacío.

```
public class App extends Application implements HasActivityInjector {
    @Inject DispatchingAndroidInjector<Activity>
  dispatchingAndroidInjector;
    @Override
    public void onCreate(){ super.onCreate(); }
    @Override
    public AndroidInjector<Activity> activityInjector() {
        return dispatchingAndroidInjector;
    }
}
```

Creamos el módulo de la aplicación **AppModule** en el paquete **di/module**; aunque, por ahora, únicamente proveeremos del contexto de la aplicación.

```
@Module
public class AppModule {
    @Provides @Singleton
    static Context provideContext(App application){
        return application.getApplicationContext();
    }
}
```

Una vez creado este módulo, construiremos su componente; para ello, creamos, en el paquete **di/component**, una clase llamada **AppComponent**, que usará los módulos **AndroidSupportInjectionModule**, **AppModule** y **BuildersModule**.

Creamos un constructor para enlazar el componente de nuestra aplicación con la aplicación. Este constructor lo tenemos declarado con la anotación **@Component.Builder**.

También creamos un método de inyección de la aplicación, **inject()**.

```
@Singleton
@Component( modules = { AndroidSupportInjectionModule.class,
                  AppModule.class, BuildersModule.class } )
public interface AppComponent {
    @Component.Builder
    interface Builder {
        @BindsInstance
        Builder application(App application);
        AppComponent build();
    }
    void inject(App app);
}
```

Si regresamos a la clase **App**, en su método **onCreate()**, construimos el componente de la aplicación inyectándole la aplicación mediante el constructor creado anteriormente en el **AppModule**.

```
@Override
public void onCreate()
{
    super.onCreate();
    DaggerAppComponent.builder()
            .application(this).build()
            .inject(this);
}
```

Si te fijas en lo anterior, verás que hemos usado una clase inexistente en nuestra aplicación. Es una clase creada por Dagger, en tiempo de compilación, compuesta por "Dagger" + "Nombre del componente de la aplicación". La generamos compilando la solución y agregando la referencia a la clase.

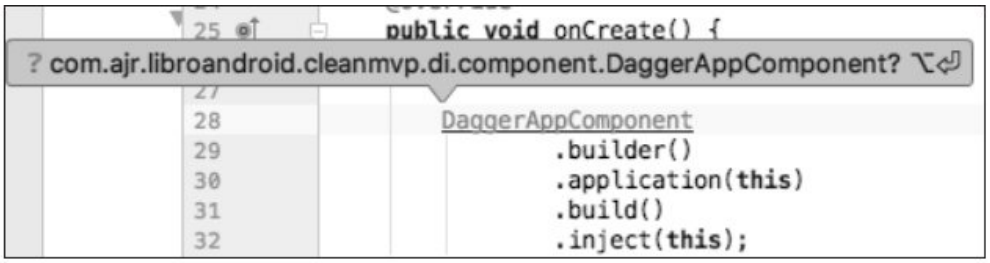


Fig. 2.3.8. Resolver DaggerAppComponent.

Como último paso de configuración, declaramos en el fichero **AndroidManifest.xml**, del módulo **presentation**, que nuestra clase **App** sea la primera aplicación en arrancar al ejecutarse el proyecto.

```
<application android:name=".App" ...
```

El siguiente paso será definir las entidades del dominio de la aplicación. Recordemos que debemos definir el modelo de la aplicación en el módulo **domain**; por tanto, crearemos un paquete con el nombre **model**, donde alojaremos las entidades del dominio de la aplicación.

Únicamente necesitamos la clase **Word** para representar la palabra de un diccionario. Esta contiene dos propiedades de tipo **string**: una para el término llamada **Term** y otra para la traducción, que llamaremos **Translation**.

```
public class Word {
    private String Term;
    private String Translation;
}
```

En la comunicación entre las diferentes capas de la arquitectura del sistema, necesitamos declarar, en las dependencias del módulo del dominio, la API Reactive Java o **RxJava**, lo que nos permite utilizar el patrón Observer.

```
dependencies { … compile 'io.reactivex.rxjava2:rxjava:2.0.2' …}
```

Recordemos que el patrón Observer se basa en el uso de dos objetos con una responsabilidad bien definida. Por una parte, los objetos observables tienen un estado concreto y son capaces de informar a sus suscriptores sobre cambios en él.

Por otro lado, los observadores son objetos que se subscriben a los observables, y solicitan ser notificados ante cambios de estado en estos.

Otros elementos del dominio son los **repositorios**, interfaces con unos métodos invocados por nuestros casos de uso e implementados en la capa de datos. Estos posibilitan la comunicación entre la capa del dominio y la de datos.

Declaremos entonces la interfaz repositorio **IDictionaryRepository**, con los tres métodos que necesita nuestro diccionario:

- **GetDictionary().** retorna el listado de palabras almacenadas.
- **SaveTranslation().** almacenaremos una palabra en el diccionario.
- **ClearDictionary().** nos permitirá vaciar el diccionario.

```
public interface IDictionaryRepository {
    Observable<ArrayList<Word>> GetDictionary();
    Observable<ArrayList<Word>> SaveTranslation(Word word);
    Observable<Boolean> ClearDictionary();
}
```

Todos retornan un observable, al cual nos subscribimos mediante observers, que definiremos más adelante en el módulo de presentación. Por ahora, únicamente es necesario saber que la finalidad de los repositorios es declarar los métodos que usan nuestros casos de uso.

Una vez definida la interfaz del repositorio, debemos crear la clase que implementa la interfaz del diccionario. Esta implementación se realizará en el módulo **data**, aunque antes necesitamos crear una serie de clases e interfaces.

Dado que el módulo **data** es el responsable del acceso a datos remotos o locales, primero definiremos un paquete en dicho módulo llamado **datastore**. Será aquí donde encontremos toda clase responsable de acceder remota o localmente a sistemas de almacenamiento.

Para que resulte sencillo, iremos paso a paso en este módulo. Comenzaremos por la definición de una interfaz vacía llamada **IDataStore**, que representa la interfaz que toda clase almacén de datos debe cumplir.

```
public interface IDataStore { }
```

El hecho de que esta interfaz esté vacía se debe a que, en el caso de tener otros almacenes de datos, todos heredarían de la misma interfaz, por lo que dependeríamos de abstracciones en lugar de concreciones. Realmente, no sería necesario definir esta interfaz si se posee un único almacén de datos, es decir, el del diccionario. No obstante, si tuviéramos otros almacenes de datos, como por ejemplo, uno de usuarios, este heredaría también de esta interfaz.

Ahora creemos la interfaz del almacén de datos del repositorio del diccionario, en el paquete **datastore** del módulo **data**, con el nombre **IDictionaryDataStore**, que hereda del almacén de datos **IDataStore**, así como de la interfaz del repositorio del dominio **IDictionaryRepository**.

```
public interface IDictionaryDataStore extends IDictionaryRepository,
IDataStore { }
```

Como queremos que los datos se puedan almacenar o consultar tanto remota como localmente, debemos crear un **factory** o factoría que nos permita seleccio-

Descargado en: eybooks.com

nar el tipo de almacenamiento. Por tanto crearemos el paquete **factory** en el módulo **data**, así como una interfaz con el nombre **IDataStoreFactory**, que representa a los métodos utilizados por las factorías.

Estos métodos son dos y ambos retornan un objeto **IDataStore**. Aquí es donde entra en juego depender de abstracciones en lugar de concreciones, al retornar una clase de acceso a datos que herede de **IDataStore** o de **IDictionaryDataStore**. Esto lo veremos más adelante al definir dichas clases.

- **Remote.** retorna la instancia de la clase que accede remotamente a los datos.
- **Local.** retorna la instancia de la clase que accede localmente a los datos.

```
public interface IDataStoreFactory {
    IDataStore Remote();
    IDataStore Local();
}
```

Tras definir la interfaz de la factoría, que toda factoría debe cumplir, definiremos la factoría de acceso a datos del diccionario. Para ello crearemos, dentro del paquete **factory** del módulo **data**, la factoría del diccionario con el nombre **DictionaryDataStoreFactory**, que implementa la interfaz de factorías. Por tanto, debemos sobrescribir los métodos que retornan las clases de almacenamiento remoto y local.

Por claridad, dejemos estos dos métodos sin implementar, ya volveremos a ellos más tarde. Simplemente destacamos que, en el constructor de la clase, inyectamos el contexto de la aplicación mediante la anotación **@Inject**. Al mantener Dagger dicha dependencia, puede reutilizarla.

Queremos, además, que esta clase persista durante el ciclo de vida de la aplicación, instanciándose una única vez. Se lo indicaremos a Dagger mediante la anotación **@Singleton** definida a dicha clase.

```
@Singleton
public class DictionaryDataStoreFactory implements IDataStoreFactory{
    Context context;
    @Inject
    public DictionaryDataStoreFactory(Context context)
    { this.context=context; }
    @Override public IDictionaryDataStore Remote()
{throw new UnsupportedOperationException("not_implemened_exception");}
    @Override
    public IDictionaryDataStore Local()
   {throw new UnsupportedOperationException
   ("not_implemened_exception"); }
}
```

Por fin podemos crear la clase que implementa la interfaz del repositorio del diccionario IDictionaryRepository. Creemos, por tanto, en el módulo **data**, una clase llamada **DictionaryRepositoryImplementation**.

Inyectamos a su constructor la factoría del diccionario definido anteriormente, **DictionaryDataStoreFactory**, necesario para seleccionar entre almacenamiento local y remoto, a través de los métodos implementados en esta clase.

```
@Singleton public class DictionaryRepositoryImplementation
         implements IDictionaryRepository
{
    DictionaryDataStoreFactory dictionaryDataStoreFactory;
@Inject public DictionaryRepositoryImplementation
                          ( DictionaryDataStoreFactory
dictionaryDataStoreFactory )
{ this.dictionaryDataStoreFactory=dictionaryDataStoreFactory; }
    @Override
    public Observable<ArrayList<Word>> GetDictionary()
{ throw new UnsupportedOperationException("not_implemented_exception"); }

    @Override
    public Observable<ArrayList<Word>> SaveTranslation(Word word)
{ throw new UnsupportedOperationException("not_implemented_exception"); }
    @Override
    public Observable<Boolean> ClearDictionary()
{ throw new UnsupportedOperationException("not_implemented_exception"); }
}
```

Observa que los métodos que implementamos son los definidos en la interfaz del repositorio del diccionario. Por ahora los dejaremos sin implementar.

Volvamos ahora al módulo del dominio. Allí tenemos que definir los casos de uso, también conocidos como **interactores**. Estos implementan las reglas de negocio de la aplicación, orquestando la comunicación desde y hacia las entidades. Por tanto, definamos en el dominio los interactores que utilizaremos.

Lo primero que haremos será crear, en el dominio, el paquete **interactor**. Será aquí donde creemos la interfaz **IObservableUseCase**, que todo caso de uso, al utilizar observables, debe implementar. Contendrá dos acciones:

- **subscribe().** para subscribir un observador al objeto **Observable**.
- **cancelSubscription().** para cancelar dicha subscripción.

```
public interface IObservableUseCase {
    void subscribe(DisposableObserver observer);
    void cancelSubscription();
}
```

Todo caso de uso deberá heredar de la clase base **UseCaseBase**, que crearemos en el paquete **interactor**. Es de tipo abstracto e implementará la interfaz **IObservableUseCase** para que subscriba y cancele la subscripción de los observadores en los observables.

Esta clase tendrá un parámetro de tipo **Type**, utilizado para indicar el tipo de datos que retorna un observable, así como otro parámetro **Object**, empleado como objeto que requiere el observable para ser construido.

Tenemos además una lista de observadores **observers** de todo observador que se subscriba al observable. Esta subscripción se realiza a través del método **subscribe()**, al que se le pasa el observador que quiere ser notificado de cambios de estado del observador y que, por tanto, añadiremos al listado de subscriptores en caso de que no se haya subscrito ya una vez.

Por otra parte, cuando no deseemos seguir escuchando cambios de estado del observable, como en el cierre de la aplicación, utilizaremos el método **cancel-**

Subscription(), el cual recorrerá cada uno de los subscriptores subscritos en el observable y los eliminará.

Definiremos un método abstracto **implementUseCase()**, que todo caso de uso que herede de esta clase base implementará. Dado que nuestros casos de uso retornarán observables con un tipo, será este el tipo retornado por dicho método. Además, como parámetros de entrada, necesitará un objeto observador **observer** y otro **object** para construir el observable, como veremos posteriormente.

Por último crearemos el método **createUseCase()**, donde ocurre toda la magia. Le pasamos a este método un observable, de modo que subscribimos este observable a un hilo diferente del principal, capaz de lanzar tareas asíncronas, del pool de hilos existente, mediante **.subscribeOn(Schedulers.io())**, de tal forma que el código del observable se ejecute en un hilo diferente.

Estos hilos, creados a la espera de ejecutar tareas, en lugar de destruirlos y crear nuevos hilos, se reutilizan para ejecutar tareas según se vayan completando.

Sin embargo, dado que estamos indicando que el observable se ejecutará en un hilo diferente, nuestro observador también lo hará por el mismo hilo y esto es un problema, si tenemos llamadas en la vista que únicamente se pueden gestionar desde el hilo principal. Como el método **onNext()** que veremos más adelante.

Para solucionarlo, indicaremos que el observador observe en el hilo principal mediante **.observeOn()**, siendo **schedulerThread** un parámetro que representa al hilo principal **AndroidSchedulers.mainThread()**, que más adelante definiremos.

Por último indicamos que el observador **observer** se subscriba al observable mediante **.subscribeWith()** para recibir notificaciones de dicho objeto observable. Y añadimos el observador a la lista de observadores para cancelar su subscripción en cualquier momento.

```
public abstract class UseCaseBase<Type,Object> implements
IObservableUseCase
{
    ArrayList<DisposableObserver<Type>> observers=new ArrayList<>();

    public void createUseCase(Observable observable,DisposableObserver<Type>
observer,Scheduler schedulerThread)
    {
        observable.subscribeOn(Schedulers.io())
                .observeOn(schedulerThread)
                .subscribeWith(observer);
        subscribe(observer);
    }
    public abstract Observable<Type> implementUseCase
(DisposableObserver observer, Object object);

    @Override public void subscribe(DisposableObserver observer) {
        if(!observers.contains(observer)) observers.add(observer);
    }
    @Override public void cancelSubscription() {
         ArrayList<DisposableObserver<Type>> observersAux =
(ArrayList<DisposableObserver<Type>>) observers.clone();
        for(DisposableObserver observer : observersAux)
            if(observer!=null && observers.contains(observer)) {
                observer.dispose();
                observers.remove(observer);
            }
    }
}
```

Perfecto, en estos momentos ya tendríamos implementada la parte de subscripción de observadores en objetos observables.

Es el momento de crear los casos de uso empleados en el proyecto; sin embargo, para construirlos, necesitaremos inyectar la clase del repositorio **IDictionaryRepository**, así como el hilo principal **Scheduler**, permitiéndole a la vista gestionar a través del método **onNext()** del observador, al indicarle previamente que este observe en el hilo principal.

Dado que vamos a necesitar acceder al planificador de tareas de Android o **AndroidSchedulers**, para seleccionar hilos de ejecución, necesitamos importar la dependencia al fichero **build.gradle** del módulo **presentation**, la API de **RxAndroid**, para hacer uso de los planificadores de tareas.

```
dependencies { … compile 'io.reactivex.rxjava2:rxandroid:2.0.1' }
```

Necesitamos ir al proveedor de dependencias de la aplicación AppModule, en el módulo **presentation**, donde proveeremos de dos nuevas dependencias.

Una retornará un Scheduler, que representará al hilo principal de la aplicación Android. Este retornará el hilo principal, del listado de schedulers de Android, a través del método **mainThread()** o hilo principal.

También proveeremos de la clase **DictionaryRepositoryImplementation**, que implementa el repositorio, para poder utilizarla e invocar sus métodos.

```
@Provides
@Singleton
static Scheduler provideScheduler(){return
AndroidSchedulers.mainThread();}

@Provides
@Singleton
static IDictionaryRepository provideDictionaryRepository(
DictionaryRepositoryImplementation dictionaryRepositoryImplementation)
{       return dictionaryRepositoryImplementation;     }
```

Recordemos que, al usar las anotaciones **@Provides** y **@Singleton**, indicamos que proveemos de una dependencia, que se instanciará una sola vez y tendrá un tiempo de vida a nivel de aplicación.

Ahora ya podemos inyectar estas dependencias en cualquier parte de la aplicación mediante la anotación **@Inject**.

Una vez creadas todas las dependencias, crearemos cada uno de los casos de uso. Serán tres, uno por cada método definido en la interfaz del repositorio **IDictionaryRepository**.

Empecemos pues por el primero de ellos; representa la obtención de todas las palabras que hemos almacenado en el diccionario. Este caso de uso será un **interactor**, perteneciente al dominio del sistema.

Creemos entonces una clase, en el paquete **interactor**, con el nombre **GetDictionaryUseCase**, extendiendo de la clase base **UseCaseBase** definida anteriormente. Además, dado que este caso de uso debe retornar un observable con un listado de palabras, lo indicamos en su herencia *UseCaseBase< **ArrayList<Word>**,-*

Void>, usando **Void** para indicar que no requerimos parámetros para obtener la información.

Definimos la anotación **@Inject**, sobre el constructor de la clase, recibiendo la interfaz del repositorio del diccionario y el planificador de tareas Scheduler como parámetros. Esta anotación permite inyectar la instancia de ambas clases, provistas por el proveedor de dependencias de la aplicación AppModule. Con ello evitamos tener que instanciar dichas clases y emplearlas ya inicializadas.

En el constructor recibimos estos dos parámetros inicializados, que podemos utilizar en esta clase. Crearemos, por tanto, dos variables privadas accesibles únicamente por la propia clase, a las que asignaremos estas inyecciones.

Por otra parte, al heredar de la clase base de los casos de uso, debemos implementar su método abstracto **implementUseCase()** y retornar un objeto **Observable** con una lista de palabras.

Este método representa la implementación del caso de uso, es decir, tenemos que implementarlo. Aquí entra en juego el repositorio y el Scheduler.

Para empezar invocaremos el método del repositorio **GetDictionary()**, proveedor de información. Este retornará un observable de lista de palabras.

Una vez obtenido el observable **dictionaryObservable**, tras invocar al método del repositorio, llamamos al método **CreateUseCase()**, de la clase **UseCaseBase**, para crear dicho caso de uso. Este requiere, como parámetros, el observable del diccionario, el observador para subscribirlo al observable, así como el planificador de tareas del hilo principal s**chedulerThread**, que representa al hilo principal.

```
public class GetDictionaryUseCase extends UseCaseBase<ArrayList<Word>,
Void> {
    private IDictionaryRepository iDictionaryRepository;
    private Scheduler schedulerThread;

  @Inject public GetDictionaryUseCase(
 IDictionaryRepository iDictionaryRepository, Scheduler schedulerThread)
  {
        this.iDictionaryRepository = iDictionaryRepository;
                this.schedulerThread=schedulerThread;
  }
  @Override public Observable<ArrayList<Word>> implementUseCase(

DisposableObserver observer, Void object)
  { Observable<ArrayList<Word>> dictionaryObservable

= iDictionaryRepository.GetDictionary();
this.createUseCase(dictionaryObservable,observer,schedulerThread);
return dictionaryObservable;
  }
}
```

Una vez creado el caso de uso, retornaremos el observable, que más adelante comentaremos en la capa de presentación. La razón de que se reciban estos dos parámetros del método **implementUseCase** es que son proporcionados por el presentador en la capa de presentación, de modo que allí es donde los instanciamos. Aunque esto lo veremos más adelante.

Definiremos los otros dos casos de uso de la misma forma que el anterior; sigamos por el caso de uso de almacenamiento de la traducción de una palabra.

Para ello creamos la clase **SaveTranslationUseCase** en el paquete de los interactores, es decir, paquete **interactor**, del módulo del dominio del sistema.

Al igual que en el caso de uso anterior, este hereda también de **UseCaseBase** y retorna un listado de palabras. No obstante, a diferencia del anterior, este requiere un parámetro palabra o **Word** para realizar la petición de datos. Representa a una palabra que almacenaremos en el almacén de datos.

Además debemos implementar el caso de uso; aunque en este caso, para obtener el observable con el listado de palabras **dictionaryObservable**, necesitamos invocar el método del repositorio del diccionario **SaveTranslation()**, el cual recibe la palabra. Por todo lo demás, es idéntico al caso de uso anterior.

```
public class SaveTranslationUseCase extends UseCaseBase<ArrayList
<Word>, Word> {
    private IDictionaryRepository iDictionaryRepository;
    private Scheduler schedulerThread;

@Inject public SaveTranslationUseCase( IDictionaryRepository
 iDictionaryRepository,
     Scheduler schedulerThread){
        this.iDictionaryRepository = iDictionaryRepository;
        this.schedulerThread=schedulerThread;
}
@Override public Observable<ArrayList<Word>> implementUseCase
( DisposableObserver observer, Word word) {
Observable<ArrayList<Word>> dictionaryObservable =
         iDictionaryRepository.SaveTranslation(word);
this.createUseCase(dictionaryObservable,observer,schedulerThread);
         return dictionaryObservable;
    }
}
```

Por último quedaría el caso de uso responsable de vaciar de memoria las palabras almacenadas. Para ello, debemos crear, en el mismo paquete **interactor**, la clase **ClearDictionaryUseCase**, heredando del caso de uso base. Sin embargo, retornará un valor booleano e indicará si se han vaciado satisfactoriamente de memoria, las palabras. Cabe destacar que no requiere ningún parámetro para eliminar el listado de palabras; por tanto, su parámetro será un **Void**.

Implementaremos el caso de uso; pero, en este caso, para obtener el objeto observable con un valor booleano **dictionaryObservable**, necesitamos invocar el método del repositorio del diccionario **ClearDictionary()** sin requerir parámetros. Por todo lo demás, es idéntico al caso de uso anterior.

```
public class ClearDictionaryUseCase extends UseCaseBase<Boolean,Void> {
    private IDictionaryRepository iDictionaryRepository;
    private Scheduler schedulerThread;
@Inject public ClearDictionaryUseCase (IDictionaryRepository
         iDictionaryRepository, Scheduler schedulerThread) {
        this.iDictionaryRepository = iDictionaryRepository;
        this.schedulerThread=schedulerThread;
    }
    @Override public Observable<Boolean> implementUseCase(
         DisposableObserver observer, Void object) {
         Observable<Boolean> dictionaryObservable=
         iDictionaryRepository.ClearDictionary();
        this.createUseCase(dictionaryObservable,observer,schedulerThread);
        return dictionaryObservable;
    }
}
```

Todos los casos de uso necesarios para el funcionamiento de nuestra aplicación ya han sido creados. Ya que estamos en el módulo de dominio, y únicamente queda una clase por declarar en este módulo, procederemos a crearla.

Se trata de una clase que nos facilitará la vida a la hora de crear nuevas instancias de palabras. Creemos un paquete con el nombre **factories**, en dicho módulo, además de una clase **Factory** que contiene una clase estática llamada **WordFactory**, que implementa un método **Create()**. Este método recibe como parámetros dos campos de tipo **String**, uno para el término y otro para la traducción, retornando una nueva instancia de palabra o **Word**.

```
public class Factory {
public static final class WordFactory {
        public static Word Create(String term, String translation)
        { return new Word(term,translation); }
}
}
```

Y, con esto, hemos terminado el bloque del módulo dominio. Pasemos, pues, al módulo de datos, donde tenemos todavía cosas pendientes de explicar.

Por lo general, a los datos se debe poder acceder, remota o localmente, mediante almacenes remotos o locales, respectivamente. Sin embargo, por claridad, hemos creído que no debíamos implementar en este capítulo el acceso a datos remotos, ya que el uso de **retrofit** para acceder a datos remotos lo veremos con detenimiento en el próximo capítulo. No obstante, declararemos la clase de almacén de datos remoto sin implementación.

Empecemos por la creación del paquete **remote**, incluido dentro del paquete **datastore** del módulo de datos.

Una vez creado el paquete, crearemos la clase **RemoteDictionaryDataStore**, que implementa la interfaz del almacén de datos del repositorio del diccionario y, por tanto, sus tres métodos, que dejaremos sin implementar.

Estos métodos son:

- **GetDictionary.** obtención de lista de palabras.
- **SaveTranslation.** almacenar nuevas palabras.
- **ClearDictionary.** vacía el listado de palabras.

Opcionalmente, en el constructor de la clase, establecemos que requiere el contexto de la aplicación como parámetro.

```
public class RemoteDictionaryDataStore implements IDictionaryDataStore
{         private Context context;
 public RemoteDictionaryDataStore(Context context)
{ this.context=context; }
 @Override
 public Observable<ArrayList<Word>> GetDictionary()
 { throw new UnsupportedOperationException("not_implemented_exception"); }
 @Override
 public Observable<ArrayList<Word>> SaveTranslation(Word word)
 { throw new UnsupportedOperationException("not_implemented_exception"); }
 @Override
 public Observable<Boolean> ClearDictionary()
 { throw new UnsupportedOperationException("not_implemented_exception"); }
}
```

Antes de crear el almacén de datos local, dado que vamos a necesitar utilizar expresiones lambda, incluiremos en el fichero **build.gradle**, del módulo **data**, la compatibilidad de Java versión 1.8, de tal forma que soporte expresiones lambda. Son de gran utilidad a la hora de desarrollar código ágilmente.

Haremos lo mismo en el fichero **build.grandle** del módulo **presentation**, para mantener la compatibilidad de Java 1.8 en el proyecto.

```
android {        …
compileOptions {
    targetCompatibility JavaVersion.VERSION_1_8
    sourceCompatibility JavaVersion.VERSION_1_8
}
}
```

Una expresión lambda se compone de uno o varios parámetros que representan a una instancia, seguidos de un "**->**" y una expresión o bloque, encerrado por llaves de apertura "**{**" y cierre "**}**".

Además, para almacenar localmente los objetos **Word** o palabra, necesitaremos serializarlos en formato JSON. Para poder usar JSON, incluiremos la dependencia de la librería **gson** en el fichero **build.gradle** del módulo de datos.

```
dependencies { … compile 'com.google.code.gson:gson:2.8.2' }
```

Sincronizando el Gradle ya tendríamos configurado el módulo de datos.

Siguiendo los pasos que hicimos en el almacén de datos remoto, es hora de crear un paquete llamado **local**, dentro del paquete **datastore** del módulo de datos. Una vez creado el paquete, crearemos la clase **LocalDictionaryDataStore**, que heredará de la interfaz del almacén de datos del repositorio del diccionario y, por tanto, tendremos que implementar sus tres métodos.

Para almacenar localmente las palabras de nuestro diccionario, podemos utilizar la API **SharedPreferences** de Android. Nos viene perfecto, ya que permite almacenar una colección de registros clave-valor en memoria localmente, y podemos definir el nivel de acceso del fichero de registros en modo privado, es decir, únicamente invocado por la aplicación que lo crea, como es nuestro caso, o en modo compartido, es decir, por otras aplicaciones.

Podemos tener acceso a este archivo de preferencias compartidas o SharedPreferences, instanciándola mediante el uso del contexto de la aplicación y el método **getSharedPreferences()**.

Una vez conocido el modo de almacenar los datos, analicemos la clase **LocalDictionaryDataStore**.

Definimos una serie de variables globales en la clase, el contexto de la aplicación, la instancia del fichero de preferencias, así como dos constantes de tipo **String**, que representan la carencia de valor **NOVALUE**, así como el nombre del fichero de preferencias **DICTIONARYRECORDS**.

Su constructor recibe, como parámetro, el contexto, necesario para poder acceder al fichero de preferencias. Es aquí donde instanciamos el fichero de prefe-

rencias mediante el nombre del fichero **DictionaryRecords**, cuyo acceso será en modo privado, accesible únicamente a través de nuestra aplicación.

```
public class LocalDictionaryDataStore implements IDictionaryDataStore {
Context context;
    private final SharedPreferences sharedPreferences;
private static final String NOVALUE = null;
private static final String DICTIONARYRECORDS = "DictionaryRecords";
public LocalDictionaryDataStore(Context context) {
        this.context = context;
        this.sharedPreferences =
context.getSharedPreferences(DICTIONARYRECORDS,Context.MODE_PRIVATE);
} …
```

Definimos un método privado **getRecords()**, reutilizable por los métodos de esta clase, el cual obtendrá una cadena en formato JSON que contiene cada una de las palabras que hemos serializado en el fichero. En nuestro caso hemos almacenado el listado de palabras, serializado en JSON, convertido a tipo **String** y, por tanto, almacenamos una cadena como valor del fichero de preferencias.

Para obtener la cadena de tipo **String**, que contiene serializada en JSON la lista de palabras, utilizamos el método **getString()** del objeto **sharedPreferences**, representando la instancia del fichero de preferencias. A este le pasamos por parámetro la clave que buscar, **DICTIONARYRECORDS**, y el valor por defecto que retornar, **NOVALUE**, en caso de no encontrar una clave con dicho nombre.

Si tenemos algún valor, debemos serializarlo al tipo deseado. En nuestro caso, queremos serializar el String a un listado de palabras, es decir, a un **ArrayList<Word>**. Para ello obtenemos el tipo buscado mediante **TypeToken**, especificando el tipo, y posteriormente obtenemos el tipo mediante **getType()**.

Una vez que tenemos el tipo que queremos utilizar para serializar, instanciamos la API Gson y deserializamos la cadena de texto **dictionaryRecords** mediante el método **fromJson** de la API de Gson, al cual le pasamos el tipo al que queremos que lo deserialice, es decir, un listado de palabras.

```
private ArrayList<Word> getRecords() {
String dictionaryRecords =
                  sharedPreferences.getString(DICTIONARYRECORDS, NOVALUE);
if (dictionaryRecords != null) {
       Type listOfWordsType = new TypeToken<ArrayList<Word>>() {}.getType();
       Gson gson = new Gson();
       return gson.fromJson(dictionaryRecords, listOfWordsType);
    }
    return new ArrayList<Word>();
}
…
```

Es interesante tener un método **existsWord()** que nos indique si una palabra existe en el listado de palabras y, en caso afirmativo, retorne la posición que ocupa en la lista.

```
…
private int existsWord(ArrayList<Word> records, Word wordToFind) {
    int index = -1;
    for (Word word : records) { index++;
        if (wordToFind.getTerm().compareToIgnoreCase(word.getTerm()) == 0)
                return index;
    }
    return -1;
}
…
```

Otro de los métodos que hemos querido incorporar es **SortRecords()**, el cual nos permite retornar la lista de palabras ordenadas alfabéticamente.

```
…
private ArrayList<Word> SortRecords(ArrayList<Word> records) {
    Collections.sort(records, (Word word1, Word word2) -> {
        return word1.getTerm().compareToIgnoreCase(word2.getTerm()); });
return records;
}
…
```

Por fin llegamos a los métodos que implementa la interfaz del repositorio del diccionario. Empecemos por la obtención de la lista de palabras **GetDictionary()**.

Este método debe retornar un objeto **Observable**, que retornará una lista de palabras. Para crear un observable debemos utilizar su método **create()**. Cabe destacar que tenemos la opción de construirlo mediante la interfaz que contiene un método de subscripción o emplear expresiones lambda. Mostraremos ambos casos para ver la diferencia. Sin expresiones lambda, el código sería el siguiente:

```
@Override public Observable<ArrayList<Word>> GetDictionary() {

return Observable.create( new ObservableOnSubscribe<ArrayList<Word>>() {
         @Override public void subscribe(
         ObservableEmitter<ArrayList<Word>> emitter) throws Exception {
                try {
                    ArrayList<Word> records = getRecords();
                    records = SortRecords(records);
                    emitter.onNext(records);
                    emitter.onComplete();
                }
                  catch (Exception e) { emitter.onError(e); }
            }
        });
}
```

Empleando expresiones lambda, el mismo código anterior pasaría a ser de la siguiente forma:

```
…
@Override public Observable<ArrayList<Word>> GetDictionary() {
return Observable.create(emitter -> {
         try {
                  ArrayList<Word> records = getRecords();
                  records = SortRecords(records);
                  emitter.onNext(records);
                  emitter.onComplete();
```

```
                }
            catch (Exception e) { emitter.onError(e);}
            });
    }
```

En nuestro caso elegiremos las expresiones lambda por simplificar el código. Lo que interesa conocer en este momento es que **emitter** es una instancia de un **ObservableEmitter**, capaz de lanzar una serie de eventos. Estos eventos son:

- **onNext.** lanzado en caso de que la petición de datos haya sido satisfactoria y sin errores, retorna los datos solicitados por parte del observable, en nuestro caso, un **ArrayList<Word>**.
- **onComplete.** lanzado cuando hemos completado la acción, por si queremos capturar el evento y realizar alguna acción.
- **onError.** lanzado en caso de error, retorna la excepción capturada.

Estos eventos serán capturados por parte de observadores (Observers), que definiremos en la capa de presentación más adelante.

Siguiendo con la explicación del método, obtenemos a través de **getRecords()** la lista de palabras almacenadas localmente en el fichero de preferencias. Una vez obtenido el listado, lo ordenaremos alfabéticamente mediante el método **SortRecords()**, definido anteriormente por nosotros.

Una vez ordenados los registros, en caso de no producirse ningún error, los retornaremos a través del método **onNext()**. Finalmente, si hemos terminado la acción correctamente, lanzamos el método **onComplete()**. Sin embargo, en caso de error, lanzaremos el método **onError()**, que se encuentra en el **try catch**.

En la implementación del siguiente método de la interfaz del repositorio, trataremos de almacenar localmente una nueva palabra. Para ello, lo primero es buscar el tipo **ArrayList<Word>**, por el que vamos a serializar.

Para no duplicar palabras en el listado de memoria, debemos obtener el diccionario, que se encuentra almacenado localmente, mediante nuestro método **getRecords()**. En caso de que la clave de la palabra, es decir, el término, exista, entonces actualizaremos su valor en el listado de palabras; en caso contrario, la añadiremos como una nueva.

Acto seguido, ordenaremos alfabéticamente las palabras mediante nuestro método **SortRecords()**, para retornarlas en la interfaz de usuario sin tener que repetir la misma tarea en la interfaz.

Por último accedemos a la instancia del fichero de preferencias e indicamos, mediante el método **edit()**, que queremos abrir el fichero en modo edición. Una vez que nos encontremos en modo edición, ya podremos definir el tipo de datos que vamos a almacenar: un número, texto o booleano.

Dado que queremos almacenar un listado de palabras, lo mejor es serializar el objeto y convertirlo a tipo texto; por este motivo utilizamos **putString()**, al que indicamos la clave del fichero, así como el objeto serializado mediante **gson.toJson()**. Este último método nos lo proporciona la API Gson para la serialización de datos en formato Json. Por último, aplicamos los cambios en el fichero **sharedPreferences** mediante **apply()** y ya está almacenado en la memoria local.

Pues bien, ya que hemos obtenido los datos, los retornaremos por el método **onNext()** e invocaremos **onComplete()** para indicar el fin de la acción. En caso de que se haya producido algún error, lanzaremos el método **onError()** del emisor.

```
@Override public Observable<ArrayList<Word>> SaveTranslation(Word word) {
      return Observable.create(emitter ->
      {
              try {
    Type listOfWordsType = new TypeToken<ArrayList<Word>>() {}.getType();
    ArrayList<Word> records = getRecords();
    word.setTerm(word.getTerm().toUpperCase());
    int indexWordInRecords = existsWord(records, word);
    if (indexWordInRecords > -1)records.set(indexWordInRecords, word);
    else if (!word.getTerm().isEmpty())

    records.add(word);
    records = SortRecords(records);

    Gson gson = new Gson();
    sharedPreferences.edit().putString(DICTIONARYRECORDS,
    gson.toJson(records, listOfWordsType)).apply();
    emitter.onNext(records);
    emitter.onComplete();
             }
    catch (Exception e) { emitter.onError(e); }
    });
    }
```

Por último, debemos implementar el vaciado del diccionario de la memoria local a través del método **ClearDictionary()**. Únicamente será necesario abrir el fichero de preferencias en modo edición, como hicimos anteriormente, e invocar al método **remove()**, indicando la clave del registro por eliminar hasta finalmente aplicar los cambios. Como observamos, retornamos un valor booleano de éxito en caso de que todo haya ido correcto por el método **onNext()**. Esto ya nos resulta familiar; todos los observables invocan los mismos tres métodos de éxito, completado y error.

```
@Override public Observable<Boolean> ClearDictionary() {
return Observable.create(emitter -> {
  try {
    sharedPreferences.edit().remove(DICTIONARYRECORDS).apply();
    emitter.onNext(true);
    emitter.onComplete();
    }
    catch (Exception e) { emitter.onError(e); }
        });
    }
… }
```

Con la implementación de estos métodos, hemos conseguido llegar al núcleo del acceso a datos locales. Si bien es cierto que hemos elegido almacenamiento de datos en ficheros de preferencias, no es el único método, ni siquiera el mejor; sin embargo, para el ejemplo rápido de una pequeña cantidad de datos, puede servir. Incluso es muy adecuado usar este tipo de recursos para almacenar sesiones de usuario y tokens de autorización.

Otra posibilidad habría sido utilizar **SQLite** o incluso **Realm**, esta última muy interesante en la actualidad por su velocidad.

Terminemos de definir la factoría de acceso a datos del diccionario **DictionaryDataStoreFactory**, que anteriormente creamos en el módulo de datos, ya que implementa la interfaz de factorías. Nos habíamos dejado sin implementar los métodos que retornan la clase de almacenamiento remoto y la clase de almacenamiento local; no obstante, tras acabarlas de crear, podemos añadirlas.

```
@Singleton
public class DictionaryDataStoreFactory implements IDataStoreFactory
{ …
@Override public IDictionaryDataStore Remote()
        { return new RemoteDictionaryDataStore(context); }
@Override public IDictionaryDataStore Local()
        { return new LocalDictionaryDataStore(context); }
}
```

Si recordamos lo que dijimos con anterioridad, los datos pueden almacenarse local o remotamente; es en esta clase donde instanciamos la clase **RemoteDictionaryDataStore**, encargada del almacenamiento remoto. Además instanciamos la clase **LocalDictionaryDataStore**, que, como ya hemos mencionado, implementa el almacenamiento local.

Largo es el camino recorrido en esta capa de datos; únicamente nos quedaría implementar la clase **DictionaryRepositoryImplementation**, que implementa la interfaz del repositorio **IDictionaryRepository**, definida en la capa de dominio.

Recordemos que tenemos tres métodos por implementar: obtención del diccionario de palabras, almacenamiento de palabras y limpieza del diccionario. De modo que, utilizando la instancia de la factoría del diccionario, accedemos a la instancia local mediante el método **Local()**. Es en este punto donde elegiríamos el tipo de almacenamiento por realizar. En caso de querer realizar un acceso a datos remoto, bastaría con invocar al método **Remote()**, de tal forma que obtuviéramos la instancia de la clase de almacenamiento remoto.

Por este motivo, dado que únicamente tenemos implementada la capa de acceso a datos locales, utilizaremos el método **Local()**. Una vez obtenida la instancia del almacén local, invocaremos a sus respectivos métodos, que nos retornarán un observable del tipo de datos requerido.

```
@Singleton
public class DictionaryRepositoryImplementation
                        implements IDictionaryRepository
{ …
@Override public Observable<ArrayList<Word>> GetDictionary()
{ return dictionaryDataStoreFactory.Local().GetDictionary(); }

   @Override public Observable<ArrayList<Word>> SaveTranslation(Word word)
{ return dictionaryDataStoreFactory.Local().SaveTranslation(word); }

@Override public Observable<Boolean> ClearDictionary()
{ return dictionaryDataStoreFactory.Local().ClearDictionary(); }
}
```

Con esta última codificación hemos terminado con la capa de acceso a datos, de tal forma que, al invocar a estos métodos del repositorio, se retornan objetos de tipo **Observable** que serán recopilados por los casos de uso de la capa de dominio y finalmente comunicados a la capa de presentación, como veremos seguidamente.

La capa de presentación tiene la responsabilidad de recibir la interacción del usuario por pantalla y comunicarse con la capa de dominio, además de representar los datos por la interfaz de usuario.

El proceso de comunicación es el siguiente:

1. Inicialmente, el usuario realiza una acción por pantalla.
2. La Activity capturará la acción, por ejemplo, la pulsación de un botón.
3. Una vez que este evento es capturado por la actividad, esta la redirige al método del presentador asociado a la Activity, capaz de resolver el evento.
4. El método del presentador que recibe dicho evento creará un objeto de tipo **Observer**, es decir, un observador, y se lo pasará a un caso de uso de la capa de dominio.
5. Dicho caso de uso construirá un objeto **Observable**, al cual subscribirá dicho observador.
6. El caso de uso invocará al método de la interfaz del repositorio, el cual estará implementado en la capa de datos.
7. La capa de datos ejecutará el método invocado, para retornar la solicitud de datos a través de un observable.
8. Este objeto es retornado al caso de uso, que a su vez lo retorna al presentador. No obstante, lo que importa es que la comunicación queda delegada al objeto **Observer** creado por el presentador; de forma que quedará a la espera de recibir información por uno de sus eventos.
9. Cuando la capa de datos resuelva con éxito la tarea de acceso a datos, invocará el método **onNext()**, el cual será recibido por el observer de la capa de presentación en la implementación de dicho método. Lo mismo ocurre en caso de evento completado o de error, por sus métodos correspondientes.
10. Una vez recibida la respuesta, el observador se lo comunicará a la vista, invocando un método de la vista, cuya implementación se encuentra en la Activity; por tanto, solo necesitaremos representar por pantalla los datos obtenidos.

Una vez conocido el proceso de comunicación entre capas, explicaremos los actores del patrón MVP que intervienen en la capa de datos. Dado que el modelo es implementado, el acceso en la capa de datos y la definición de entidades en el dominio del sistema, quedaría ubicar la vista y el presentador.

Una Activity tiene asociados un presentador o presenter, así como una vista. La vista es una interfaz que define una serie de métodos que habilitan la comunicación entre la Activity y el presentador. Será la Activity la encargada de implementar la vista.

A su vez, el presentador es el responsable de la comunicación con la capa de dominio para realizar peticiones, así como con la Activity. Esto es posible gracias a inyectar, en el presentador, la interfaz de la vista de la Activity, invocando sus métodos, implementados en la Activity. De esta forma, el presentador no depende del framework de Android.

Una vez conocida la teoría, es el momento de pasar a la práctica. Trabajaremos de aquí en adelante en la capa de presentación, por lo que todo lo que indiquemos será creado en esta capa.

Empecemos por crear el paquete **mvp**, que incluirá a su vez dos paquetes adicionales: uno para las vistas, con el nombre **mvp/view**, y otro, para los presentadores, con el nombre **mvp/presenter**.

Una vez creados los paquetes, antes de definir la vista, definamos la interfaz **IDataView**, en el paquete **view**, con los métodos que toda vista implementará:

- **showLoading().** mostrará el icono barra de progreso o ProgressBar.
- **hideLoading().** ocultará la barra de progreso.
- **showMessage().** mostrará un mensaje por pantalla.

```
interface IDataView {
void showLoading();
void hideLoading();
void showMessage(String message);
}
```

Una vez definida esta interfaz genérica de las vistas, crearemos una nueva interfaz **IDictionaryView**, que herede de la interfaz genérica de vistas **IDataView**.

- **renderData().** mostrará, por pantalla, el listado de palabras.
- **dictionaryCleared().** vaciará el listado mostrado por pantalla.

```
public interface IDictionaryView extends IDataView {
 void renderData(ArrayList<Word> dictionary);
 void dictionaryCleared(Boolean value);
}
```

Con esto último, la vista estaría definida. Antes de hacer lo mismo con el presentador, definamos los objetos **Observer** que instanciará el presentador. Empecemos por crear el paquete **observer** en la raíz de paquetes de esta capa de presentación y posteriormente la clase **DictionaryObserver**, que representa un objeto observador u **Observer**, que espera un listado de palabras.

En el constructor de la clase, le pasaremos la instancia de la vista, así como el contexto de la aplicación. Por otro lado, recordemos que, al heredar de **DisposableObserver**, estamos convirtiendo nuestra clase en un observador y, por tanto, tenemos que implementar los tres métodos del observador. Estos son los métodos que se invocan desde la capa de datos y son capturados aquí.

Por un lado, en caso de éxito de la solicitud, nos retornará el observable, a través del método **onNext()**, una lista de palabras. Pues bien, enviaremos esta lista a la vista para que la muestre por pantalla a través del método **renderData()**.

Por otro lado, en caso de error, entrará por el método **onError()**, donde informaremos a la vista que oculte la barra de progreso y muestre un mensaje, por pantalla, con el texto del error de que no hay registros.

Por último, en caso de tarea completada, el evento será capturado por el método **onComplete()**, que informará a la vista para que oculte la barra de progreso, dado que la acción ha finalizado.

```
public class DictionaryObserver extends DisposableObserver
<ArrayList<Word>>{
 private IDictionaryView iDictionaryView;
 private Context context;
 public DictionaryObserver( IDictionaryView iDictionaryView,
     Context context) {
     this.iDictionaryView=iDictionaryView;
     this.context=context;
 }
@Override public void onNext(ArrayList<Word> value)
        { iDictionaryView.renderData(value); }

@Override public void onError(Throwable e) {
          iDictionaryView.hideLoading();
```

```
        iDictionaryView.showMessage(
                        context.getString(R.string.error_no_
records));
    }

@Override public void onComplete() { iDictionaryView.hideLoading(); }
}
```

De igual manera que antes, definimos otro observador para el almacenamiento de nuevas palabras en el diccionario, con el nombre **DictionaryTranslatorObserver**, el cual retornará un listado de palabras.

```
public class DictionaryTranslatorObserver extends
DisposableObserver<ArrayList<Word>> {
    private IDictionaryView iDictionaryView;
    private Context context;
public DictionaryTranslatorObserver(IDictionaryView iDictionaryView,
Context context) {
        this.iDictionaryView=iDictionaryView;
        this.context=context;
    }
@Override public void onNext(ArrayList<Word> value)
         { iDictionaryView.renderData(value); }
    @Override public void onError(Throwable e)
         { iDictionaryView.hideLoading();
         iDictionaryView.showMessage(
         context.getString(R.string.error_on_saving_translation));
         }
    @Override public void onComplete() { iDictionaryView.hideLoading(); }
}
```

Por último, del mismo modo, crearemos el observador responsable de capturar eventos del vaciado del diccionario **DictionaryClearObserver**. Retornamos un valor booleano que indica el éxito del vaciado del diccionario. En caso de obtener un valor por el método **onNext()**, invocaremos al método de la vista, **dictionaryCleared()**, encargado de vaciar la lista de palabras en pantalla.

```
public class DictionaryClearObserver extends DisposableObserver
<Boolean> {
    private IDictionaryView iDictionaryView;
    private Context context;
    public DictionaryClearObserver(IDictionaryView iDictionaryView,
         Context context) {
        this.iDictionaryView=iDictionaryView;
        this.context=context;
    }
    @Override public void onNext(Boolean value)
                { iDictionaryView.dictionaryCleared(value); }
    @Override public void onError(Throwable e)
                { iDictionaryView.hideLoading();
                  iDictionaryView.showMessage(
                context.getString(R.string.error_not_cleared));
            }
    @Override public void onComplete() { iDictionaryView.hideLoading(); }
}
```

Perfecto, ya tenemos todo lo necesario para crear el presentador. Creemos, en el paquete **presenter**, una interfaz **IPresenter** que todo presentador cumplirá, definiendo un método **destroy()**, para cancelar las subscripciones de los observadores en los observables de los casos de uso, cuando la aplicación pase a segundo plano. En tal caso no querremos recibir actualizaciones de estado de los observables.

```
public interface Presenter { void destroy(); }
```

Una vez creada dicha interfaz, crearemos una clase con el nombre **DictionaryPresenter** que herede de la interfaz anterior del presentador.

Inicialmente inyectaremos, en el constructor de la clase del presentador, el contexto de la aplicación, la interfaz de la vista y cada uno de los casos de uso que queramos ejecutar.

```
public class DictionaryPresenter implements IPresenter {
private IDictionaryView iDictionaryView;
private GetDictionaryUseCase getDictionaryUseCase;
private SaveTranslationUseCase saveTranslationUseCase;
private ClearDictionaryUseCase clearDictionaryUseCase;
private Context context;
@Inject public DictionaryPresenter(Context context
                  , IDictionaryView iDictionaryView
                          , GetDictionaryUseCase getDictionaryUseCase
                  , SaveTranslationUseCase saveTranslationUseCase
                  , ClearDictionaryUseCase clearDictionaryUseCase    ) {
                         this.iDictionaryView = iDictionaryView;
        this.context = context;
        this.getDictionaryUseCase = getDictionaryUseCase;
        this.saveTranslationUseCase = saveTranslationUseCase;
        this.clearDictionaryUseCase = clearDictionaryUseCase;
} …
```

Posteriormente crearemos el método **initialize()**, el cual invocará al método **getDictionary()**, para obtener el listado de palabras del diccionario. Antes de invocar al caso de uso, informaremos a la vista para que muestre un icono de barra de progreso.

Posteriormente, invocaremos al caso de uso, responsable de la tarea de obtener el diccionario de palabras, siendo este la instancia del caso de uso **getDictionaryUseCase**, al cual recordemos que para invocarlo debemos pasarle un objeto observador y un parámetro en caso de que lo necesite. Por esta razón, instanciaremos un nuevo objeto observador **DictionaryObserver**, el cual requiere la instancia de la vista y el contexto para instanciarse. Por último, dado que este caso de uso no requiere ningún parámetro, enviaremos un valor nulo.

```
…
    public void initialize() { getDictionary(); }
    public void getDictionary() {
       iDictionaryView.showLoading();
         getDictionaryUseCase.implementUseCase(
                   new DictionaryObserver(iDictionaryView,context), null);
    }
…
```

Para el almacenamiento de nuevas palabras, creamos un método llamado **saveTranslation()**, el cual implementaremos de igual forma que el anterior, con la salvedad de que, en este caso, utilizaremos el caso de uso **saveTranslationUseCase**, responsable de almacenar nuevas palabras. Además crearemos una instancia del observador **DictionaryTranslatorObserver**, responsable de observar eventos del observable de este caso de uso. Como parámetro, a diferencia del anterior, enviaremos una palabra, utilizando la factoría de construcción de palabras, para crear una nueva instancia de palabra.

```
…
    public void saveTranslation(String term, String translation) {
       iDictionaryView.showLoading();
         saveTranslationUseCase.implementUseCase(
                  new DictionaryTranslatorObserver(iDictionaryView,
context),
                  Factory.WordFactory.Create(term, translation));
    }
…
```

Otro de los métodos creados es **clearDictionary()**, responsable de ejecutar el vaciado del diccionario. Se construye igual que los dos métodos anteriores, salvo que, en este caso, invocamos al caso de uso **clearDictionaryUseCase** con el observador **DictionaryClearObserver**, destinado a dicha tarea. Dado que no requiere parámetros, enviamos un valor nulo.

```
…
   public void clearDictionary() {
    iDictionaryView.showLoading();
    clearDictionaryUseCase.implementUseCase(
    new DictionaryClearObserver(iDictionaryView,context), null);
}
…
```

Por último, implementamos el método **destroy()**, definido por la interfaz **IPresenter**, y que cancelará la subscripción de los observadores a los observables, dado que este método será invocado por la interfaz cuando la aplicación pase a segundo plano.

```
…
    @Override
    public void destroy() {
        this.iDictionaryView = null;
         if (getDictionaryUseCase != null)
                  getDictionaryUseCase.cancelSubscription();
         if (saveTranslationUseCase != null)
                  saveTranslationUseCase.cancelSubscription();
         if (clearDictionaryUseCase != null)
                           clearDictionaryUseCase.cancelSubscription();
    }
}
```

Una vez definido el presentador y la vista de la Activity, debemos proveerlos en el módulo de la Activity, es decir, en **DictionaryModule**. Recuerda que, al inicio de

este capítulo, mientras lo configurábamos, lo dejamos vacío. Ahora ya podemos proveer del presentador **DictionaryPresenter** y de la vista **IDictionaryView**.

```
@Module
public abstract class DictionaryModule {
    @Provides
static DictionaryPresenter provideDictionaryPresenter(Context context
            ,IDictionaryView iDictionaryView
            ,GetDictionaryUseCase getDictionaryUseCase
            ,SaveTranslationUseCase saveTranslationUseCase
            ,ClearDictionaryUseCase clearDictionaryUseCase
    )
{ return new DictionaryPresenter( context, iDictionaryView,
getDictionaryUseCase,
saveTranslationUseCase, clearDictionaryUseCase);
}
    @Binds abstract IDictionaryView provideDictionaryView
(DictionaryActivity dictionaryActivity);
}
```

Gracias a este módulo, la Activity podrá hacer uso de su presentador y su vista asociada.

En la representación de los datos, nos serviremos de un adaptador sencillo, compuesto por un listado de elementos con dos líneas, un título y un subtítulo. Para ello, vamos a utilizar un ArrayAdapter de palabras, que crearemos en el paquete **di/adapter**, con el nombre **DictionaryAdapter**.

Al constructor de esta clase adaptador le pasaremos el contexto y el listado de palabras, mediante la plantilla **layout simple_list_item_2** de Android, compuesto por dos vistas de texto, para representar cada elemento del listado. Uno de los TextView **text1** servirá para representar el término, mientras que el otro TextView **text2** se utilizará para representar la traducción.

```
public class DictionaryAdapter extends ArrayAdapter<Word> {
    ArrayList<Word> words;
public DictionaryAdapter(@NonNull Context context,
                                                        @NonNull
 ArrayList<Word> words) {
         super(context, android.R.layout.simple_list_item_2,

         android.R.id.text1, words);
         this.words=words;
    }
    @Override
    public View getView(int position, View convertView,
    ViewGroup parent) {
        View view = super.getView(position, convertView, parent);

        TextView text1 = view.findViewById(android.R.id.text1);
        TextView text2 = view.findViewById(android.R.id.text2);

        text1.setText(words.get(position).getTerm());
        text2.setText(words.get(position).getTranslation());
        return view;
    }
}
```

Una vez que tenemos el adaptador construido, podemos centrarnos en la interfaz de usuario (Figura 2.3.9), compuesta por dos campos de texto para intro-

ducir términos y traducciones, un botón para guardar la palabra, una lista para representar las palabras, un botón flotante para crear nuevas palabras y dos elementos más que no vemos en la vista previa: el menú con un icono para vaciar el listado y un icono de barra de progreso.

Fig. 2.3.9. Vista previa de la interfaz de usuario.

Hemos creado los estilos para las vistas de la interfaz de usuario en el directorio de recursos de estilos **styles.xml** del módulo de presentación. No van más allá de configuración de anchura, altura, el centrado de la vista y color de texto. Por tanto, se han creado por comodidad a la hora de definir las vistas de la interfaz para hacerlas más sencillas de leer.

Hemos considerado útil que nuestra Activity tuviera un icono como elemento de menú en la barra de herramientas o Toolbar. Por ello hemos creado un nuevo menú (Figura 2.3.10), llamado **menu_dictionary.xml**, en el directorio de recursos del módulo de presentación, con un icono de papelera de reciclaje, mostrado siempre en la ToolBar de la Activity.

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">
    <item android:id="@+id/clear_dictionary"
        android:icon="@drawable/ic_delete_white_24dp"
        android:title="@string/clear"
        app:showAsAction="always">
    </item>
</menu>
```

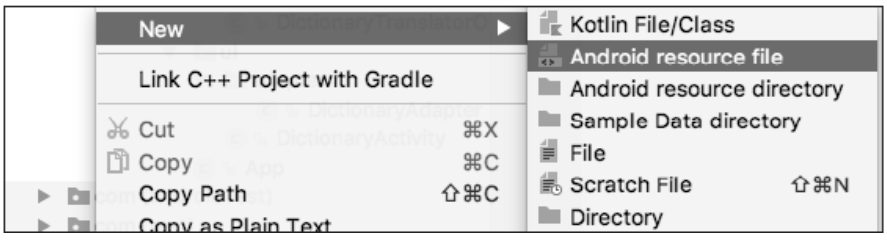


Fig. 2.3.10. Crear Recurso Menú.

Ahora definiremos la plantilla empleada por la actividad principal, es decir, la interfaz de usuario **activity_dictionary.xml**. Definimos un **RelativeLayout**, compuesto por un **LinearLayout** que contiene dos campos de texto y un botón. Por otra parte, definimos un **SwipeRefreshLayout**, que permitirá actualizar el listado **ListView** que contiene cada vez que hagamos un gesto con nuestro dedo sobre la lista, hacia abajo. Por último, definimos un botón flotante y una barra de progreso o ProgressBar.

Volviendo a la codificación, únicamente nos queda codificar la actividad principal **DictionaryActivity**, la cual implementa la interfaz de la vista **IDataView**, así como el listener de refrescado del **SwipeRefreshLayout**, que nos permitirá capturar evento de refresco.

Inicialmente inyectamos el presentador **DictionaryPresenter**, dado que lo hemos provisto en el módulo de la Activity y, por tanto, podemos usarlo.

```
public class DictionaryActivity extends AppCompatActivity implements
 IDictionaryView, SwipeRefreshLayout.OnRefreshListener {
    Button btn_save_word;
    FloatingActionButton btn_add;
    EditText term_edit_text;
    EditText translation_edit_text;
    ListView list_view_dictionary;
    LinearLayout linear_layout_word_edition;
    ProgressBar loading_progress;
    SwipeRefreshLayout swipe_layout;
@Inject DictionaryPresenter dictionaryPresenter;
…
```

Por otra parte, en el método **onCreate** de la Activity, antes de que ocurra la inyección de dependencias, debemos inyectar nuestra Activity a la aplicación a través del método **inject()** de la clase inyectora **AndroidInjection**.

En **onCreate**, además, buscaremos cada una de las vistas de la interfaz y las definiremos globalmente. En posteriores capítulos veremos una forma más limpia de realizar todo esto mediante **ButterKnife**.

```
…
@Override protected void onCreate(Bundle savedInstanceState) {

        AndroidInjection.inject(this);

        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_dictionary);

        loading_progress = findViewById(R.id.loading_progress);
        swipe_layout = findViewById(R.id.swipe_layout);
        swipe_layout.setOnRefreshListener(this);
        dictionaryPresenter.initialize();
        term_edit_text = findViewById(R.id.term_edit_text);
        translation_edit_text = findViewById(R.id.translation_edit_text);
        btn_save_word = findViewById(R.id.btn_save_word);
        list_view_dictionary = findViewById(R.id.list_view_dictionary);

          linear_layout_word_edition =
          findViewById(R.id.linear_layout_word_edition);
       btn_add = findViewById(R.id.btn_add);
         btn_save_word.setOnClickListener(saveTranslationClick);
         btn_add.setOnClickListener(buttonAddClick);
}
…
```

Cuando el usuario pulse el botón de guardado, la Activity capturará el evento clic de este botón mediante el listener **saveTranslationClick**. Tras capturar dicho evento, invocaremos al método **saveTranslation()** del presentador, responsable de guardar la traducción, y le pasaremos como parámetro los valores escritos por pantalla en los campos de texto.

```
…
View.OnClickListener saveTranslationClick= new View.OnClickListener() {
    @Override public void onClick(View v) {
        dictionaryPresenter.saveTranslation(
                  term_edit_text.getText().toString(),
                  translation_edit_text.getText().toString());
    }};
…
```

Cuando el usuario pulse el botón flotante de añadir nuevas palabras, la plantilla con los dos campos de texto, este se mostrará en caso de que no fuera visible o se ocultará en caso contrario.

```
…
View.OnClickListener buttonAddClick=new View.OnClickListener() {
    @Override public void onClick(View v) {
        if (linear_layout_word_edition.getVisibility() == View.VISIBLE)
            linear_layout_word_edition.setVisibility(View.GONE);
        else
            linear_layout_word_edition.setVisibility(View.VISIBLE);
    }};
…
```

Implementamos el método de la vista **showLoading()**, responsable de mostrar el icono de barra de progreso. Este método mostrará el icono de carga, cada petición de datos nueva.

```
…
@Override
public void showLoading() { loading_progress.setVisibility(View.
VISIBLE); }
…
```

Implementamos el método de la vista **hideLoading()**, responsable de ocultar el icono de barra de progreso. Este método ocultará el icono de carga tras finalizar una petición de datos. Además, dado que en la interfaz utilizamos también el **SwipeRefreshLayout**, en caso de que hubiera mostrado este elemento un icono de refresco, lo ocultaremos mediante **setRefreshing(false)**.

```
…
@Override public void hideLoading() {
loading_progress.setVisibility(View.GONE);
swipe_layout.setRefreshing(false);
}
…
```

De igual forma haremos con el método sobrescrito de la vista **showMessage()**, el cual mostrará un mensaje por pantalla, empleando el clásico pero útil **Toast**.

```
…
@Override public void showMessage(String message)    { Toast.
makeText(this,message,Toast.LENGTH_SHORT).show(); }
…
```

Creamos un método con el nombre **refreshAdapter()**, responsable de crear una instancia del adaptador de palabras **DictionaryAdapter**, al cual pasamos como parámetro el listado de palabras que ha de representar la plantilla del adaptador. Este adaptador se lo asignaremos al ListView **list_view_dictionary**, y se representará por pantalla.

```
…
private void refreshAdapter(ArrayList<Word> dictionary) {
 ArrayAdapter<Word> adapter = new DictionaryAdapter(this, dictionary);
 list_view_dictionary.setAdapter(adapter);
}
…
```

Sobrescribimos el método de la vista **renderData()**, responsable de recibir el listado de palabras por parte del presentador y refrescar la vista con este listado. Además ocultaremos los campos de introducción de palabras y vaciaremos sus valores.

```
…
@Override public void renderData(ArrayList<Word> dictionary) {
        refreshAdapter(dictionary);
          linear_layout_word_edition.setVisibility(View.GONE);
        term_edit_text.setText(“”);
        translation_edit_text.setText(“”);
}
…
```

Sobrescribimos el método de la vista **dictionaryCleared()**, responsable de recibir el valor de éxito o fracaso por parte del presentador tras borrar el listado de palabras. En caso de éxito refrescaremos el adaptador con un listado de palabras vacío.

```
…
@Override public void dictionaryCleared(Boolean value) {
         if (value) {
                  ArrayAdapter<Word> adapter =
                                    new DictionaryAdapter(this, new
ArrayList<Word>());
             list_view_dictionary.setAdapter(adapter);
        }
    }
…
```

Sobrescribimos el método **onRefresh()**, responsable de capturar el gesto de refresco de datos del listado, al implementar el listener del **SwipeRefreshLayout.OnRefreshListener**. Invocaremos desde aquí una nueva petición al presentador, para que obtenga el listado de palabras actualizado.

```
…
@Override public void onRefresh() { dictionaryPresenter.getDictionary(); }
…
```

Construimos el menú de la actividad mediante la plantilla definida por el fichero **menu_dictionary.xml**, que definimos anteriormente y que únicamente muestra un icono en la Toolbar, sobrescribiendo el método **onCreateOptionsMenu**.

```
…
@Override public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.menu_dictionary, menu);
        return true;
}
…
```

Para la detección del pulsado sobre el icono del menú, que realiza el borrado del listado de palabras, requerimos sobrescribir el método **onOptionsItemSelected**. Tras detectar que hemos pulsado el icono de borrado, invocaremos el método del presentador, responsable de vaciar la lista.

```
…
@Override public boolean onOptionsItemSelected(MenuItem item) {
        int id = item.getItemId();
        if (id == R.id.clear_dictionary) {
            dictionaryPresenter.clearDictionary();
            return true;
        }
        return super.onOptionsItemSelected(item);
}
…
```

Finalmente, antes de que la actividad se destruya, invocamos el método del presentador, responsable de cancelar las subscripciones de los observadores en los observables, para dejar de recibir actualizaciones de estado.

```
…
@Override protected void onDestroy() { super.onDestroy();
        dictionaryPresenter.destroy();
}
}
```

Con esto tendríamos implementado nuestro proyecto sencillo de un diccionario de palabras, que emplea almacenamiento local, Dagger 2, RxJava, patrón MVP y arquitectura Clean. Si hemos seguido todos los pasos, la estructura final del proyecto debería ser similar a esta:

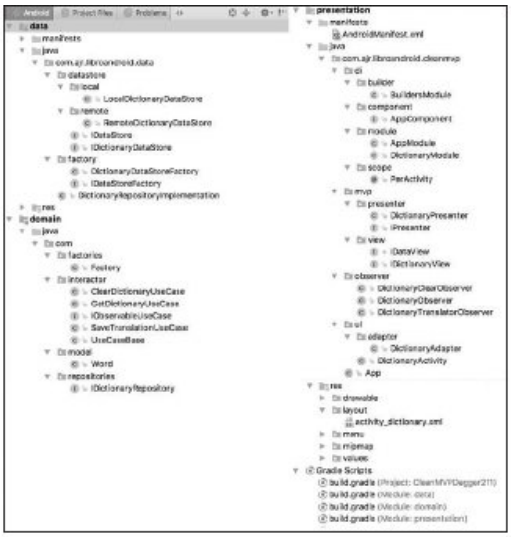


Fig. 2.3.11. Estructura final.

Al ejecutar la aplicación, deberíamos ver algo parecido a la primera imagen (Figura 2.3.12), una lista vacía con un botón flotante y un icono en el menú.

Si pulsamos el botón flotante, insertamos palabras como mostramos en la segunda imagen (Figura 2.3.13), donde tras pulsar el botón de guardado, se ocultarían los campos de texto y se mostraría la lista con las palabras del diccionario (Figura 2.3.14). En caso de pulsar el icono papelera del menú, se vaciará la lista y se vería como la primera imagen (Figura 2.3.12).

Finalmente, en caso de querer actualizar el listado con un gesto, veríamos el icono de refresco que mencionamos, como muestra la última imagen (Figura 2.3.15).

Por el momento, esto sería todo; en próximos capítulos veremos un ejemplo más avanzado, que incluirá el acceso a datos remotos.

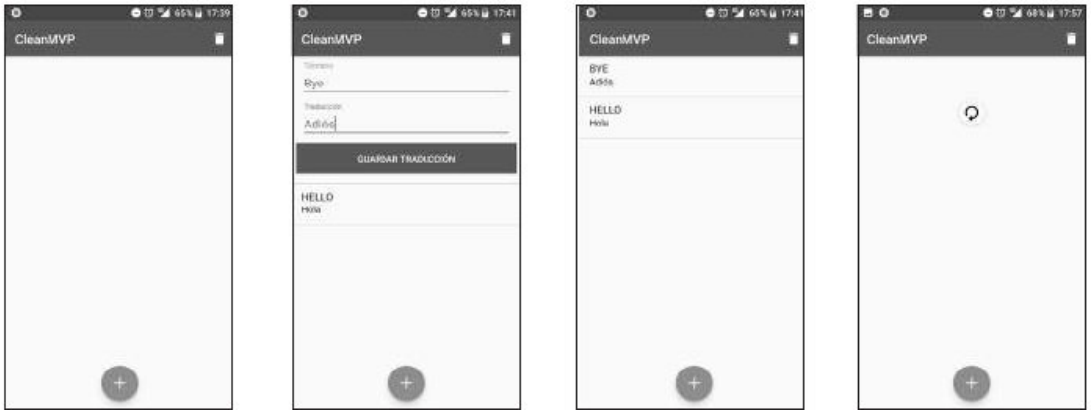


Fig. 2.3.12. Pantalla inicial. Fig. 2.3.13. Pantalla Crear palabra.
Fig. 2.3.14. Pantalla listado. Fig. 2.3.15. Pantalla Refrescar.

PARTE 3:
DESARROLLO DE UNA APP PROFESIONAL

CAPÍTULO 1:
Desarrollo app profesional

Una vez que hemos aprendido toda la base para crear aplicaciones ágiles, usando una arquitectura limpia y de forma estructurada, llega el momento de incrementar la dificultad y crear una aplicación que perfectamente podría encargarnos una empresa.

Toda aplicación debe presentar al usuario información útil, de forma clara, y poner a su disposición operaciones sobre los datos. En función de cómo se almacenen estos datos, se emplearán orígenes diversos, ya sea en memoria, en disco, en base de datos relacional, no relacionales NoSQL o mediante peticiones a través de Internet.

El propósito de este capítulo es crear una aplicación de hoteles completamente profesional, poniendo a su disposición datos a través de Internet. Para ello, dado que el propósito de este libro no es crear un backend, utilizaremos Firebase. Este nos proporcionará todo lo que necesitamos para simular un backend, proporcionándonos almacenamiento en la nube de nuestros datos de hoteles, gestión de usuarios y sesiones activas, así como el almacenamiento de ficheros. Este proyecto de ejemplo, llamado **Hotels**, podemos encontrarlo en el código proporcionado por la editorial.

Por ello recomendamos que, una vez descargado el código fuente de este proyecto, lo utilices como referencia, dado que en este capítulo explicaremos cada caso de uso, las partes más importantes. Aunque puede verse en profundidad todo el código en el proyecto descargado desde la editorial.

Trello

En la construcción de este proyecto vamos a utilizar Trello y definiremos cada uno de los casos de uso que vamos a realizar. Disponemos de tres columnas: pendiente, en proceso y finalizado.

Por defecto, todas las tareas las tenemos en estado pendiente, sin asignar a ningún miembro del equipo; de tal forma que, cada vez que un miembro del equipo realice una tarea, deberá mover la tarea de la columna pendiente al estado en proceso. De esta forma indicamos que estamos realizando el caso de uso.

Una vez que finalicemos la tarea, la moveremos a su estado correspondiente, es decir, finalizado. Así, tras haber verificado su correcto funcionamiento, indicamos que dicha tarea está finalizada y podemos pasar a otra de las columnas de tareas pendientes de realizar.

Podemos observar el caso inicial en la siguiente imagen. De esta forma, conforme vayamos realizando las tareas, iremos moviéndolas a su estado correspondiente y así informaremos a los miembros del equipo de que uno de ellos está trabajando en una tarea y que ellos pueden realizar otra, como puede verse en la Figura 3.2.

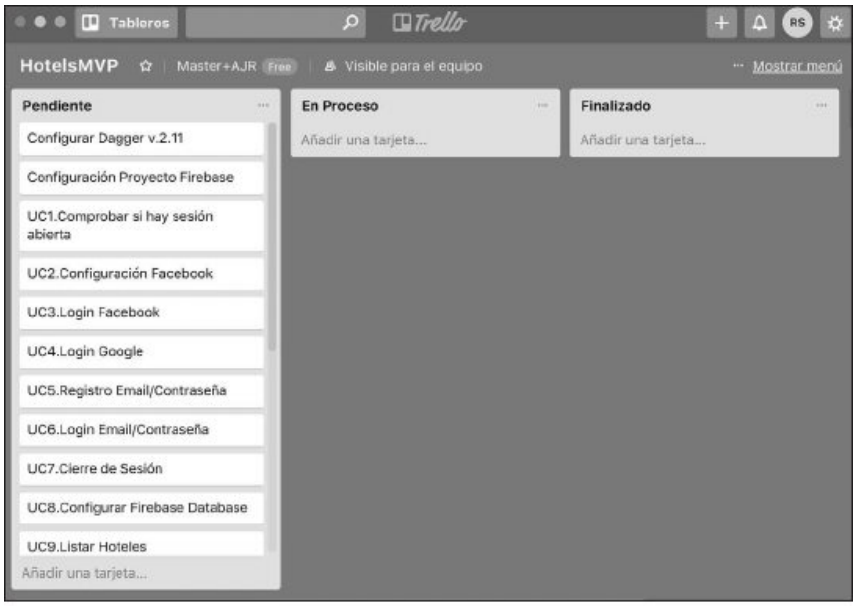


Fig. 3.1 Trello. Definición de casos de uso.

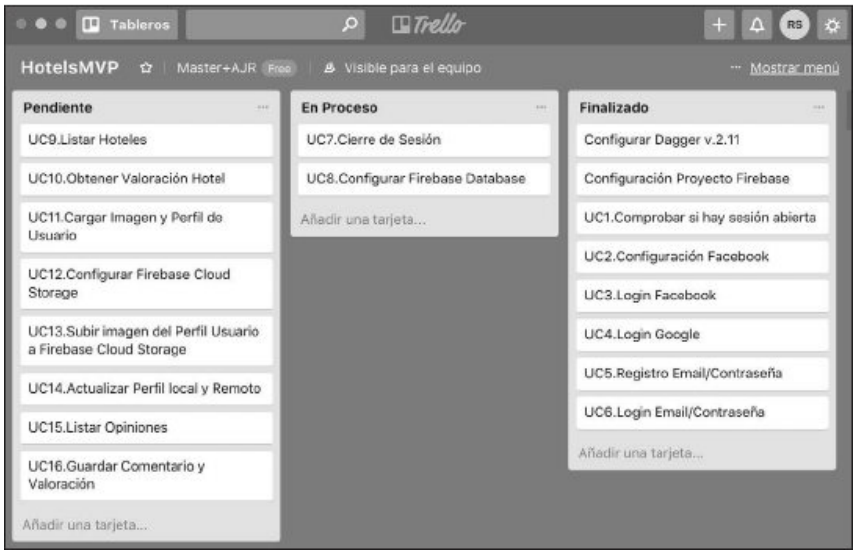


Fig. 3.2 Trello. Estado intermedio.

Git

Hemos subido nuestro proyecto a Bitbucket, a un repositorio privado; no obstante, el proyecto completo de este capítulo se proporcionará como material adicional, descargable a través de la editorial Marcombo.

La idea es crear un proyecto desde cero; de modo que, tras crear una cuenta en Bitbucket o GitLab, elige el que más te guste, crearemos un repositorio en dicho sistema.

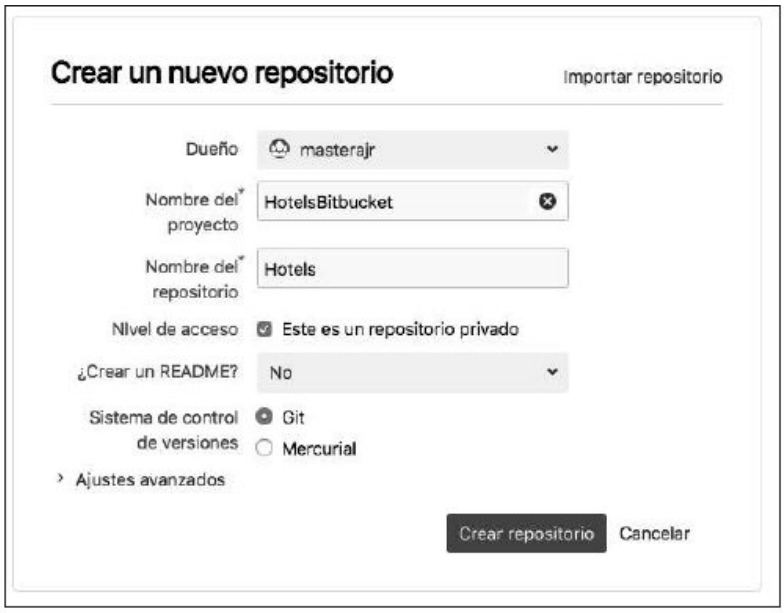


Fig. 3.3 Crear repositorio en Bitbucket.

Tras crear el repositorio en Bitbucket, tendremos una ruta similar a esta:
https://miusuariogit@bitbucket.org/masterajr/hotels.git

Tenemos creado nuestro proyecto en un directorio de nuestro ordenador Mac o Windows. De modo que abrimos una ventana de terminal o consola y nos posicionamos por comandos en el directorio del proyecto.

Tras posicionarnos en el directorio de nuestro proyecto Android, a través de la consola de comandos, inicializamos Git mediante el comando:

```
git init
```

Tras inicializar Git, ejecutamos el siguiente comando para asociar nuestro directorio local, con el repositorio remoto de Bitbucket, adquirido anteriormente.

```
git remote add origin https://miusuariogit@bitbucket.org/masterajr
/hotels.git
```

Indicamos que queremos añadir todos los ficheros que Android Studio haya generado al estado "Index", que es un estado previo al "commit". Es decir, un estado entre nuestro directorio local y los cambios que queremos aplicar en el repositorio remoto. Para ello, ejecutamos el comando:

```
git add .
```

Tras ejecutar este comando, los ficheros se ponen en un estado previo al "commit". El punto añadido al final del comando anterior indica que queremos añadir todos los ficheros que no estuvieran ya añadidos.

Una vez que los ficheros que inicialmente deberían estar en un estado "new file", ya que es la primera vez que los añadimos al estado "Index", debemos pa-

sarlos al estado “commit”, un estado de presubida al repositorio remoto. Para ello ejecutaremos el siguiente comando:

```
git commit -m ‘Comentario Inicial, Primer Commit’
```

Por último, una vez que tenemos los ficheros en el estado del “Head”, es hora de subirlos al repositorio de Bitbucket. Únicamente se subirán aquellos cambios que se encuentren en el “Head”. Para ello ejecutamos el comando:

```
git push –u origin master
```

De este modo, indicamos que queremos subir los ficheros del estado “Head” de nuestro equipo, es decir, del origen de nuestro repositorio local, a la rama principal del repositorio remoto, que por defecto se llama **master**. Y con esto, ya tendríamos subidos los ficheros en el repositorio, como vemos en la imagen.

Fig. 3.4. Push Bitbucket.

Por último, cada vez que finalicemos un caso de uso, ejecutaremos los siguientes comandos en este orden:

1. Para añadir los ficheros nuevos que hayamos creado en el caso de uso:

```
git add .
```

2. Para pasar del estado “Index” al estado “Head”:

```
git commit -m ‘Caso de uso X, Finalizado’
```

3. Para pasar del estado “Head” de nuestro equipo al repositorio:

```
git push -u origin master
```

Realizaremos estos pasos con cada caso de uso, de tal forma que todos los miembros del equipo podamos subir al repositorio el trabajo que vayamos realizando.

Por otra parte, si trabajamos en grupo, antes de realizar un push de los cambios, debemos actualizarnos con lo que los compañeros hayan realizado mediante el comando:

```
git pull
```

Posteriormente, el pull nos bajará los cambios realizados por los compañeros y nosotros, con un push, subiremos lo que tengamos en el estado del "Head".

Firebase

Firebase es una plataforma de apoyo al desarrollo móvil y web que pone a nuestra disposición una gran variedad de funcionalidades en la nube. Clasificado como BaaS (Backend as a Service), nos permitirá desarrollar nuestra aplicación más rápidamente, nos ahorrará el tiempo y dinero que invertiríamos en implementar esos servicios por nuestra cuenta.

De entre sus funcionalidades podemos destacar: la gestión de la autenticación, notificaciones, almacenamiento de archivos, hosting, gestión de informes de error de tu aplicación, analíticas, base de datos en tiempo real o una plataforma para testear tus aplicaciones en una gran variedad de dispositivos.

Este servicio ofrecido por Firebase nos descarga de mucho trabajo que tendríamos que hacer en la parte de backend y nos permite centrarnos en nuestras aplicaciones.

Para autenticarnos en la plataforma, únicamente necesitaremos tener una cuenta de Google. A partir de ese momento podemos acceder a la consola de desarrollo Firebase y comenzar a crear nuestros propios proyectos.

Firebase Authentication

En algún momento del ciclo de vida de nuestra app, es posible que necesitemos gestionar el acceso de usuarios a partes privadas de nuestra aplicación. Lo normal sería implementar nosotros mismos la capa de gestión de usuarios en nuestro backend; sin embargo, gracias a Firebase, podemos ahorrarnos semanas de desarrollo de esta gestión de usuarios.

Firebase nos proporciona una serie de métodos de inicio de sesión. Desde el más tradicional, como viene a ser el correo electrónico y contraseña, hasta multitud de proveedores muy populares, como por ejemplo Facebook, Google o Twitter.

Muchos de nosotros ya poseemos algún tipo de cuenta con estos proveedores, lo que nos permitirá no tener que recordar las credenciales para acceder a la app. Bastará con hacer clic sobre el login de cualquiera de estos proveedores para obtener las credenciales, que serán enviadas al SDK de autenticación de Firebase.

Una vez registrados en Firebase, podremos acceder al perfil del usuario, así como obtener el token de acceso, que tiene una caducidad de 1 hora.

Este token de acceso será requerido en caso de necesitar invocar a la API REST de Firebase para obtener y actualizar información, siempre y cuando los datos sean privados, y así lo hayamos establecido mediante reglas de acceso a datos.

Sesión activa de usuario

Tras haber configurado nuestro proyecto para utilizar Firebase, habremos obtenido un fichero de configuración JSON, que utilizaremos en nuestro proyecto.

Para saber si tenemos alguna sesión activa en Firebase, debemos obtener una instancia de **FirebaseUser**, no nula, invocando:

```
FirebaseAuth.getInstance().getCurrentUser();
```

Una vez que tengamos la instancia del usuario de Firebase, podremos acceder a sus propiedades, tales como nombre, email, foto e identificador de usuario. Podremos acceder a estas mediante getters de la instancia del objeto **FirebaseUser**.

Debemos saber que un identificador de usuario o UID es único e identifica a un usuario en el sistema, sin caducidad, a no ser que manualmente, en la consola de desarrolladores de Firebase, revoquemos dicho identificador.

No debemos confundir el UID del usuario con el TokenID, ya que el token identifica la sesión abierta de un usuario y tiene una hora de caducidad.

Actualizar perfil del usuario

En la fase de registro de un usuario, según el proveedor utilizado, se proporcionan unas propiedades que son almacenadas en Firebase y que podemos modificar en cualquier momento. Para ello necesitamos tener una sesión de usuario abierta en Firebase e invocar al método **updateProfile**, al cual le proporcionaremos una instancia del objeto **UserProfileChangeRequest** para cambiar el nombre y la imagen.

```
UserProfileChangeRequest profileUpdates = new UserProfileChangeRequest
          .Builder().setDisplayName(“#NOMBRE#”).setPhotoUri(“#URI#”).
          build();
firebaseUser.updateProfile(profileUpdates)…
```

Autenticación por email y contraseña

El método clásico de autenticación por email y contraseña es posible en Firebase. Nos permite tener un listado de usuarios, cuyo email debe ser único, así como una serie de funcionalidades ya implementadas, como la recuperación de credenciales. En nuestro caso realizaremos el registro de una nueva cuenta, así como el acceso a través de una cuenta ya existente.

Crear una nueva cuenta es tan sencillo como acceder a la instancia de **FirebaseAuth** e invocar al método **createUserWithEmailAndPassword**, pasándole como parámetros el email y la contraseña.

```
firebaseAuth.createUserWithEmailAndPassword(email, password)
```

El login de un usuario ya existente se realiza de una forma similar, pero en este caso invocaremos al método **signInWithEmailAndPassword**, proporcionándole los mismos parámetros que en el caso anterior.

```
firebaseAuth.signInWithEmailAndPassword(email, password)
```

Por último, cuando queramos cerrar la sesión con la que hemos accedido, bastará con realizar un **signOut**, sobre la instancia de **FirebaseAuth**.

```
firebaseAuth.signOut();
```

Autenticación con Facebook

Facebook es quizás uno de los proveedores más utilizados por la comunidad de usuarios y es que la gran mayoría de nosotros tenemos una cuenta con ellos. Por esta razón hemos querido incorporar este proveedor al proyecto.

Para empezar, Facebook no nos lo pone tan fácil, debido a que hay una serie de pasos previos que debemos configurar fuera de Firebase, en el sitio web de Facebook Developers, al que deberemos entrar con una cuenta de Facebook para crear una aplicación Facebook.

Tras crearla y configurarla apropiadamente, como veremos más adelante, se nos proporcionará un identificador y un secreto de la aplicación. Pues bien, estos dos datos son los que debemos proporcionar a Firebase al habilitar este método de autenticación.

Una vez que lo hemos configurado todo, tendremos dos opciones para realizar el registro y login de usuario. Podemos utilizar un botón proporcionado por el SDK de Facebook, llamado **LoginButton**, que nos proporciona un botón en la interfaz con la apariencia habitual de Facebook, o bien crear nuestro propio botón en la interfaz y utilizar el **LoginManager**, responsable de controlar eventos de registro y login, el cual solicitará los permisos de acceso al email y al perfil público.

Inicialmente debemos tener una instancia del **CallbackManager** de Facebook, responsable de controlar las llamadas entre el SDK de Facebook y la Activity.

```
CallbackManager.Factory.create();
```

Tras obtener una instancia de esta clase, para utilizar nuestro propio botón con la apariencia deseada, necesitamos invocar al **LoginManager**, manejando los permisos de lectura de email y del perfil público de Facebook.

```
LoginManager.getInstance()
.logInWithReadPermissions(this, Arrays.asList("email", "public_profile"));
```

Manejando además la respuesta obtenida de Facebook, en caso de éxito, e invocando al método **registerCallback**.

```
LoginManager.getInstance().registerCallback(callbackManager,
new FacebookCallback<LoginResult>() {
    @Override public void onSuccess(LoginResult loginResult) {…
```

El **CallbackManager** capturará la respuesta de la validación de Facebook en el **onActivityResult** de la Activity, que en caso de éxito será finalmente capturada por el método sobrescrito anterior **onSuccess**, donde tendremos que gestionar el resultado obtenido **LoginResult**.

LoginResult nos proveerá del token de acceso mediante:

```
loginResult.getAccessToken().getToken();
```

Finalmente, debemos obtener la credencial de Facebook, **AuthCredential**, mediante el **FacebookAuthProvider** y el **AccessToken** obtenido anteriormente.

```
FacebookAuthProvider.getCredential(accountIdToken);
```

Para realizar el **SignIn** con las credenciales de Facebook en Firebase, crearemos una nueva cuenta en Firebase en caso de que no exista o simplemente accederemos empleando el usuario que ya existe.

```
firebaseAuth.signInWithCredential(credential) …
```

Por último, en caso de querer cerrar sesión en Facebook, debemos realizar:

```
LoginManager.getInstance().logOut();
```

Autenticación con Google

Google es otro de los proveedores más utilizados; la mayoría de nosotros poseemos una cuenta Google.

Una vez habilitado este proveedor en Firebase, configuramos el acceso de nuestra app con Google. Configuremos, pues, las **GoogleSignInOptions**, solicitando el identificador de usuario, email y el perfil del usuario.

```
GoogleSignInOptions googleSignInOptions = new GoogleSignInOptions
    .Builder(GoogleSignInOptions.DEFAULT_SIGN_IN)
    .requestIdToken(application.getString(R.string.default_web_client_id))
    .requestEmail().build();
```

Tras esto, añadimos la configuración al cliente de la API de Google, **GoogleApiClient**.

```
GoogleApiClient googleApiClient = new GoogleApiClient.Builder(this)
      .addApi(Auth.GOOGLE_SIGN_IN_API, googleSignInOptions).build();
```

Ya tendríamos la API de Google configurada y obtenida su instancia. Al igual que en Facebook, Google nos permite utilizar su botón de login **SignInButton**, o bien realizarlo manualmente con un botón personalizado.

Para efectuar el registro o login en Google, debemos obtener el intent de acceso, pasándole como parámetro la instancia de la API de Google:

```
Auth.GoogleSignInApi.getSignInIntent(googleApiClient);
```

Esto nos proporcionará un intent que deberemos lanzar. El resultado lo capturaremos en el **onActivityResult** de la Activity que lo lanzó. Este resultado obtenido es un **GoogleSignInResult**, a través del cual, en caso de éxito, podremos obtener información sobre nuestra cuenta de acceso a Google, es decir, la **GoogleSignInAccount**. Esta cuenta nos proporcionará el token de acceso que necesitamos para obtener las credenciales.

```
googleSignInResult = Auth.GoogleSignInApi.getSignInResultFromIntent(data);
googleSignInAccount = googleSignInResult.getSignInAccount();
String accountIdToken = googleSignInAccount.getIdToken();
```

Finalmente, obtenemos la credencial de Google, **AuthCredential**, mediante el **GoogleAuthProvider** y el **AccessToken** obtenido anteriormente.

```
firebaseAuth.signInWithCredential(credential) …
```

Por último, para cerrar sesión en Google, cerramos la conexión del cliente de la API y desconectamos la recepción de respuestas de conexión de este proveedor.

```
googleApiClient.registerConnectionCallbacks(
new GoogleApiClient.ConnectionCallbacks() {
    @Override public void onConnected(@Nullable Bundle bundle) {
         Auth.GoogleSignInApi.revokeAccess(googleApiClient)
         .setResultCallback( status -> {
             googleApiClient.disconnect();
             googleApiClient.unregisterConnectionCallbacks(this);  });…
```

Firebase Realtime Database

Firebase pone a nuestra disposición una base de datos, con persistencia en la nube, en la que los datos son almacenados en formato JSON. Como peculiaridad, cabe destacar que hay dos tipos de acceso a estas bases de datos. Por una parte, podemos utilizar una instancia de la base de datos en nuestra aplicación y, a partir de ahí, invocar sus métodos implementados por Firebase, o simplemente consumir una API REST. Esta última es la que más nos interesa para evitar toda dependencia posible.

Si nos decantáramos por emplear la instancia de base de datos Firebase en nuestra aplicación, tendríamos a nuestra disposición los datos sincronizados entre nuestra app y la base de datos remota, en caso de que algún dato cambie.

Inicialmente, tras crear un proyecto en Firebase que utilice Realtime Database, la plataforma nos proporciona la URL donde nuestra base de datos se encuentra alojada y, de momento, sin datos.

Esta base de datos no contiene tablas ni registros del tipo fila/columna, sino datos en formato JSON. Todo nodo JSON tiene que poseer una clave, de modo que

manualmente podemos ir creando nodos clave/valor en la consola de Firebase o simplemente importar un fichero con nuestros datos en formato JSON. Esta última es la manera más sencilla y limpia de hacerlo.

Es importante estructurar bien nuestro JSON para no necesitar bajarnos todos los datos en una única llamada, sino solo aquella información que necesitemos en cada momento. Además, el máximo número de niveles de cada nodo JSON es de 32, más que suficiente; sobre todo porque, al bajarnos un nodo de un determinado nivel, nos bajamos también sus niveles inferiores.

Dado que en nuestro caso no vamos a utilizar la instancia de la base de datos Firebase, es decir, el **DatabaseReference**, no entraremos en detalle en este punto, ya que vemos más interesante utilizar nuestras propias llamadas a la API REST para acceder a los datos. No obstante, quien quiera depender de la API de bases de datos Firebase puede hacerlo, la verdad es que es sencillo de usar y muy útil, sobre todo en cuanto a la sincronización de los datos en tiempo real, así como trabajar sin conexión con los datos.

Reglas de acceso

Llegados a este punto, es posible que nos preguntemos por la privacidad de los datos y quién puede acceder a estos. Pues bien, toda la configuración de seguridad de acceso a los datos se realiza en la consola de Firebase, en el apartado **Reglas** de la sección de Realtime Database.

Las reglas son sencillas de configurar, incluso Firebase proporciona un simulador de reglas para verificar si alguien, autorizado o no, tiene acceso a un cierto nodo.

Las reglas se definen en un JSON cuyo nodo raíz tiene una clave llamada **rules**. Dentro de las reglas podemos establecer el nivel de protección o acceso que queramos a cada nodo, asignándole permisos de lectura o escritura. Los métodos de acceso disponibles son:

- **Usuarios autenticados.** En este caso, lectura o escritura, o ambas, requieren que la conexión con la base de datos la realice un usuario autenticado. Esto se define mediante **"auth != null"**. Un ejemplo de lectura y escritura solo permitida para usuarios autenticados sería:

```
{ "rules": { ".read": "auth != null", ".write": "auth != null" } }
```

- **Acceso público.** En este caso, lectura o escritura, o ambas, son accesibles de forma pública, es decir, exponemos nuestros datos a que cualquiera pueda leer o escribir. Un ejemplo que permite públicamente la lectura y escritura sería:

```
{ "rules": { ".read": true, ".write": true } }
```

- **Acceso privado.** Este caso sería el opuesto al anterior, es decir, no se permite lectura, escritura o ninguna. Un ejemplo de no permitir el acceso de lectura y escritura sería:

```
{ "rules": { ".read": false, ".write": false } }
```

- **Acceso usuario.** Este caso permite que un usuario acceda, por ejemplo, a unos datos privados que únicamente deban ser accesibles para el usuario propietario de los datos. Un ejemplo de esto sería definir un nodo, que hemos llamado **zonaPrivada**, donde cada usuario almacenará información privada. Cada subnodo de la zona privada contiene un nodo de clave, el identificador de usuario (UID), y como valor otro nodo JSON con múltiples propiedades.

  Cuando el usuario accede a la base de datos, identificamos quién es mediante la instrucción **auth.uid**, que retorna el **UID** del usuario validado. Entonces definimos un nombre de variable, por ejemplo **$uid**, que representa la clave UID del usuario, cuya información cuelga del nodo **zonaPrivada**, y dentro de esa variable definimos las reglas de lectura y escritura sobre dicho nodo. De esta forma, únicamente si coincide la clave del nodo usuario **$uid**, con el UID del usuario autenticado, tendrá acceso de lectura y/o escritura:

```
{"rules": {
    "zonaPrivada": {
         "$uid": {
                          ".read": "$uid === auth.uid",
                  ".write": "$uid === auth.uid"
                          }
} } }
```

Con saber esto sobre las reglas de acceso en Firebase nos basta para proteger nuestros datos de la forma deseada.

Firebase Cloud Storage

Firebase ha pensado una vez más en facilitarnos las cosas y ha creado por nosotros un sistema de almacenamiento remoto de ficheros. En caso de querer privatizar el acceso a los ficheros que hemos subido, podemos crear reglas específicas de acceso.

Este sistema, llamado **Cloud Storage**, nos permite subir o descargar ficheros persistentes remotamente en él. Tras crear el proyecto en Firebase, nos proporciona una URL donde almacenaremos todos los ficheros que necesitemos, de igual forma que ocurría con la base de datos en tiempo real.

Lo primero que necesitamos es obtener una instancia del **FirebaseStorage** para poder acceder al almacenamiento remoto de ficheros:

```
FirebaseStorage.getInstance();
```

Una vez obtenida la instancia del repositorio de almacenamiento, referenciamos al directorio de almacenamiento, empleado a la hora de subir o bajar ficheros. Esta referencia es un **StorageReference**. A modo de ejemplo, esta sería una referencia para obtener o almacenar la imagen del usuario logueado:

```
String path = "directorio/"+firebaseAuth.getCurrentUser().getUid()+".png";
storage.getReference(path);
```

Existen varios modos de subir un fichero, aunque el más sencillo consiste en pasarle un array de bytes, que hemos llamado **data**, con los datos del fichero. Para ello, creamos una tarea de subida de ficheros, llamada **UploadTask**, que requiere que pasemos, a la referencia de almacenamiento, los bytes por subir.

Finalmente, le asignamos un listener para capturar el evento de subida satisfactoria, el cual nos retornará un objeto **TaskSnapshot**, que contiene propiedades como la URL, los bytes transmitidos, los metadatos enviados al servidor, así como la referencia de almacenamiento **StorageReference** que se utilizó en la subida de ficheros.

A nosotros nos interesa obtener la URI con la ruta del fichero que acabamos de subir a Firebase; por tanto, obtendremos dicha propiedad invocando al método **getDownloadUrl**() de la **TaskSnapshot**.

```
UploadTask uploadTask = storageRef.putBytes(data);
uploadTask.addOnSuccessListener(new OnSuccessListener
<UploadTask.TaskSnapshot>() {
   @Override public void onSuccess(UploadTask.TaskSnapshot taskSnapshot) {
      Uri downloadUrl = taskSnapshot.getDownloadUrl();
   }});
```

Para descargar el fichero, el proceso será similar; aunque, en lugar de subir bytes con **putBytes**, debemos obtener dicho array de bytes mediante **getBytes**, al que indicaremos la cantidad mínima de bytes por descargar. No obstante, nosotros realizaremos este proceso de otra forma, que veremos más adelante.

GSON

GSON es una librería creada por Google para convertir objetos JSON a objetos Java y viceversa. A la hora de analizar resulta más ágil el formato JSON que el formato XML, lo que reduce el coste de procesamiento.

Para utilizar GSON en nuestro proyecto debemos añadir, al gradle, la librería:

```
compile 'com.google.code.gson:gson:2.8.2'
```

En caso de que sea un tipo complejo, debemos obtener su tipo de la siguiente forma, por ejemplo, un listado de un objeto:

```
Type listType = new TypeToken<ArrayList<Object>>() { }.getType();
```

Una vez que sabemos el tipo del objeto que queremos serializar a JSON, lo que hacemos es obtener una instancia de GSON e invocar su método **toJson**. Este método retornará una cadena de texto con el objeto serializado en formato JSON.

```
Gson gson = new Gson();
String jsonObject = gson.toJson(records, listType);
```

La operación inversa es convertir una cadena en formato JSON a un objeto de Java. Para ello, con la misma instancia de GSON, invocamos al método **fromJson**, el cual nos proporcionará un objeto Java deserializado.

```
ArrayList<Object> lObjects = gson.fromJson(jsonObject, listType);
```

Hemos visto que no es nada complicado realizar una serialización y deserialización de datos, gracias a la librería de código abierto de Google GSON.

Retrofit 2

Retrofit es una librería creada por **Square**, y que podemos encontrar en GitHub, que nos permite consumir una Web API REST desde nuestra aplicación Android. Enviaremos peticiones como GET, POST, PUT, DELETE, con la posibilidad además de enviar en estas peticiones parámetros de tipo **@Body**, **@Path**, **@Header** y **@Query**.

Inicialmente definimos una interfaz, que represente al servicio con las llamadas que necesite para acceso a datos. Esta interfaz la llamamos **IService**.

Partiendo de que toda petición debe apuntar a la misma base URL donde se encuentra nuestra API, dentro de los TAG de tipo de petición debemos incluir el resto de la llamada:

- **GET, Para retornar todos los objetos.** Dentro del GET incluimos la llamada al controlador objetos, responsable de esta tarea.

```
@GET("objetos")
Call<List<Object>> getAllObjects();
```

- **GET requiere un parámetro.** Debemos definir este parámetro en la ruta, entre llaves **{ }**, así como definir el tag **@Path** para asociar el parámetro de la ruta con el valor del parámetro del método.

```
@GET("objetos/{id}")
Call<Object> getObject(@Path(value = "id", encoded = true) int ObjectId);
```

- **GET requiere una request Header.** Deberemos incluir el tag **@Header**, lo que permitirá enviarle un token de acceso en la cabecera de la petición.

```
@GET("objetos")
Call<List<Object>> getAll(@Header("Authorization") String authorization);
```

- **POST permite almacenar un objeto remotamente.** Debemos emplear el tag **@Body**, que es el objeto por enviar en el cuerpo de la petición.

```
@POST("objetos")
Call<Object> createObject(@Body Object objeto);
```

Descargado en: eybooks.com

- **PUT permite actualizar un objeto remotamente.** Le pasamos, en el body, el objeto por actualizar. En el path, como vimos anteriormente, indicamos el identificador del objeto por actualizar y finalmente, en caso de ser necesario, podemos utilizar el tag **@Query**, que nos permitirá añadir parámetros adicionales en la petición del tipo "&parametro=MiValor".

```
@PUT("objetos/{id}")
Call<Object> updateObject(@Path(value = "id", encoded = true) int id ,
          @Query("extra") String extra ,
          @Body Object objeto);
```

- **DELETE permite eliminar un objeto.** Le pasamos, en el path, el identificador del objeto por borrar y lo borrará si la API así lo interpreta.

```
@DELETE("objetos/{id}")
Call<Object> deleteObject(@Path(value = "id", encoded = true) int id );
```

Las llamadas anteriores seguirían las reglas de Api RestFul, donde cada tipo de petición está definido en un controlador de la Web API del backend y realiza lo que hemos indicado anteriormente.

Volviendo a Retrofit, para configurarlo en nuestro proyecto, inicialmente debemos importar la librería de Retrofit, además de configurar, mediante un constructor de este, la URL base donde se encuentra nuestra API remota.

```
retrofit = new Retrofit.Builder()
       .baseUrl(context.getResources().getString(R.string.apiBaseURL))
         .addConverterFactory(GsonConverterFactory.create()).build();
```

Si observamos el código anterior, establecemos mediante **baseUrl** la dirección donde se encuentra nuestra Web API. Por otra parte, dado que los datos enviados y recibidos estarán en formato JSON, es posible que necesitemos algún conversor que permita serializar los datos obtenidos a su mejor representación. Por esta razón añadimos un conversor. Por defecto, nosotros utilizaremos el **GsonConverterFactory**; sin embargo, en ocasiones, necesitaremos crear nuestro propio conversor, dependiendo de cómo queramos transformar los datos.

Por último, Retrofit creará la implementación de la interfaz del servicio que hemos definido **IService**, para que podamos invocar las llamadas del servicio.

```
IService iService = retrofit.create(IService.class);
```

Cuando tengamos la implementación de los métodos de la interfaz del servicio por parte de Retrofit, podremos invocarlos. Para ello bastará con utilizar la instancia del servicio e invocar a uno de sus métodos, por ejemplo:

```
Call<Object> call = iService.getOpinion(id);
```

Una vez creada la llamada síncrona o asíncrona, mediante **Call**, que recibe un objeto **Object**, para ejecutarla y recibir una respuesta http, invocamos el método

execute() de la llamada que acabamos de instanciar. Esto nos devuelve una respuesta de tipo Response, que contendrá un **código** http de la petición, y un **body** con el valor retornado de la petición, en este caso, el objeto en formato Json.

```
Response<Object> response = call.execute();
```

Los códigos de respuesta http más comunes son:

- **200.** La solicitud se ha realizado correctamente.
- **401.** No está autorizado a realizar la petición.
- **400.** La solicitud produjo un error.
- **403.** Acceso prohibido, no puede acceder al recurso.
- **404.** Recurso no encontrado.

Butterknife

Butterknife es una librería muy útil, creada por Jake Wharton, que facilita la vida al desarrollador, ya que nos permite desarrollar código más limpio e inyectar las vistas de manera organizada.

Recordemos que, sin Butterknife, el método tradicional de cargar una vista que se encuentra en el layout de la Activity es mediante **findViewById**, pero esto hace que nuestro código se "ensucie" y dificulte la lectura.

Inicialmente importamos la librería de Butterknife al gradle, así como las anotaciones, para poder usar los tags que veremos a continuación.

```
compile 'com.jakewharton:butterknife:8.6.0'
annotationProcessor 'com.jakewharton:butterknife-compiler:8.6.0'
```

Tras sincronizar el proyecto, debemos inyectar las vistas del layout, en cada Activity donde queramos hacer uso de Butterknife, en el método **onCreate**.

```
ButterKnife.bind(this);
```

Bien, ya tenemos preparada la Activity para utilizar Butterknife y sus anotaciones. Una de las ventajas de usar esta librería es que la inyección de vistas se realiza simplemente mediante la anotación **@BindView**, indicándole el identificador del recurso por inyectar.

```
@BindView(R.id.myView)
TextView myView;
```

Una vez declarado lo anterior, ya tendríamos la vista cargada y lista para ser utilizada. Otro de los recursos que nos proporciona esta librería es declarar el evento clic de un botón, con su método, empleando la anotación **@OnClick**.

```
@OnClick(R.id.myButton)
public void doMyButtonClick() { … }
```

En el código anterior, habríamos asociado a la vista botón el evento clic, de modo que, cuando pulsáramos el botón, se ejecutaría ese método definido.

Además de todo lo anterior, podríamos profundizar más y cargar recursos como textos, colores o imágenes, etc.

```
@BindString(R.string.myText)
String myText;
@BindColor(R.color.myColor)
int myColor;
```

Picasso

Otra de las librerías creadas por Square es Picasso, increíblemente útil para cargar imágenes remotamente desde una URL.

Utilizar esta librería es tan sencillo como añadirla al gradle:

```
compile 'com.squareup.picasso:picasso:2.5.2'
```

Una vez que la tenemos incorporada a nuestro proyecto, ya podemos utilizarla. Es muy útil para cargar imágenes asíncronamente en los adaptadores del RecyclerView, dado que cargamos los elementos conforme los vamos necesitando y liberamos los recursos cuando estos no aparecen en la vista del Recycler. Por tanto, esta librería nos proporciona una ayuda en esta tarea.

Para cargar una imagen partiendo de una URL, para asignársela a un recurso de tipo ImageView, bastará con indicar a Picasso el contexto actual, indicado en **with**, para cargar la URL donde se encuentra la imagen, indicado en **load**, y una vez que lo resuelva, lo cargue en un recurso de tipo imagen, indicándolo en **into**.

```
Picasso.with(context).load("URL").into(R.id.image);
```

Con esta instrucción, ya tendríamos cargada la imagen desde URL. Además, para optimizar el consumo de memoria al cargar las imágenes, podemos reducir su tamaño mediante **.resize(50,50)**, lo cual reduciría ancho y alto a la mitad de sus dimensiones.

Realm Database

Realm es una base de datos utilizada en multitud de plataformas, como Java, Swift o .Net entre otras. Es increíblemente rápido a la hora de realizar lecturas y escrituras, ya que efectúa el mapeado de objetos instantáneamente.

Hemos utilizado Realm, en la capa de datos, para la persistencia de datos locales, y trabajado con transacciones para que, en caso de que algo falle, no se almacene en la base de datos y poder volver al estado anterior del error.

Importamos el plugin de Realm al gradle de nuestro proyecto Android:

```
classpath "io.realm:realm-gradle-plugin:4.2.0"
```

Y aplicamos el plugin que acabamos de importar en el gradle del módulo que utilice Realm.

```
apply plugin: 'realm-android'
```

Todo se organiza en contenedores de objetos, de tal forma que, para acceder a estos, tenemos que instanciar un objeto **Realm**. Si queremos empezar a trabajar con esta base de datos, primero debemos inicializar Realm con el método **init**, pasándole el contexto de la aplicación.

```
Realm.init(this);
```

Posteriormente, debemos crear una configuración **RealmConfiguration** por defecto y asignársela a Realm.

```
RealmConfiguration realmConfiguration = new RealmConfiguration.Builder().
build();
Realm.setDefaultConfiguration(realmConfiguration);
```

Como podemos observar, la configuración anterior está vacía, pero a modo de configuración por defecto, nos sirve para inicializar Realm.

Nuestros objetos del dominio de la aplicación pueden ser objetos **Realm**, simplemente extendiendo la definición de la clase a **RealmObject**. Tras esta acción, nuestra clase se convierte en un objeto de tipo **Realm**.

```
public class MiObjeto extends RealmObject …
```

En cuanto a los tipos de datos soportados por Realm, encontramos String, int, float, double, boolean, byte, long, short, Date y byte[].

Por otra parte, podemos utilizar anotaciones encima de la definición de una propiedad de la clase, como **@Required**, para indicar que dicha propiedad será requerida a la hora de guardar valores, o la notación de clave primaria **@PrimaryKey**, que podrá ser un **int** o un **String**, indexando automáticamente dicho valor en la base de datos. O simplemente podemos ignorar el guardado en base de datos, de una propiedad de la clase, mediante la anotación **@Ignore**.

Estos tipos de base de datos son contenedores registrados con un nombre **name** y que acaban en **.realm**. Cuando alguna propiedad del objeto **Realm** almacenado varíe, incrementaremos el número de versión del esquema.

Si no queremos perder los datos almacenados en el contenedor, tras modificar el esquema, deberemos aplicar una migración de los datos que permita pasar de versiones anteriores a más recientes mediante **addMigration**. No obstante, en las fases iniciales de la construcción de la base de datos, dado que no suelen haber apenas datos, nos puede interesar utilizar el **deleteRealmIfMigrationNeeded**, lo que nos permite recrear el esquema del objeto **Realm**.

```
RealmConfiguration config = new RealmConfiguration.Builder()
                .name("MiObjeto.realm").schemaVersion(1)
        .deleteRealmIfMigrationNeeded().build();
```

La configuración anterior cargaría la versión 1 del esquema del contenedor "MiObjeto.realm", de modo que en caso de ser necesaria una migración, se recrearía el esquema. Una vez definida la configuración, para poder trabajar con el objeto **Realm** que hemos indicado, tenemos que obtener su instancia:

```
Realm objectRealm = Realm.getInstance(config);
```

Estamos preparados para trabajar con la instancia de nuestro contenedor. En cada acción que realicemos sobre la base de datos, bien sea lectura o escritura, vamos a utilizar transacciones. Recordemos que una transacción es una acción realizada sobre la base de datos, en la que en caso de que realicemos una modificación de datos y tenga éxito, se confirmará la persistencia de estos en la base de datos; sin embargo, en caso de que suceda algún error, se volverá al estado anterior a la modificación de estos.

Tenemos dos opciones por cada acción que realicemos con la base de datos: por un lado, establecer manualmente cuando se inicia **beginTransaction** y finaliza una transacción aplicando los cambios en la base de datos, **commitTransaction.**

```
objectRealm.beginTransaction();
...//Todo lo que tengamos que modificar en la base de datos viene aqu .
objectRealm.commitTransaction();
```

Por otro lado, tenemos la posibilidad de utilizar el método **executeTransaction**, el cual encapsula toda esta funcionalidad anterior.

```
objectRealm.executeTransaction(new Realm.Transaction() {
    @Override public void execute(Realm realm) {
 …//Todo lo que tengamos que modificar en la base de datos viene aqu .
 } });
```

Finalmente, debemos cerrar toda instancia que abramos de cada contenedor Realm. Esto suele indicarse en los **finally** tras un **try/catch**, de modo que pase lo que pase con la transacción, aunque dé error, nos aseguramos de cerrar la instancia mediante **close**.

```
objectRealm.close();
```

Para trabajar con un modelo Realm, lo primero que debemos hacer es utilizar el método **where** de una instancia abierta Realm, de tal forma que inicie un **RealmQuery**, que nos permitirá realizar consultas sobre un contenedor. Los métodos que ejecuta RealmQuery son:

- **findAll.** Obtiene todos los objetos que cumplen una condición y retorna una colección de resultados **RealmResults**, que contienen un listado de instancias del contenedor. De esta forma, para obtener todas las instancias mapeadas a la clase del contenedor utilizado:

```
RealmResults<Object> results = objectRealm.where(Object.class).findAll();
```

- **findFirst.** Obtiene el primer elemento que cumple una condición.

```
Object object =objectRealm.where(Object.class).findFirst();
```

- **findAllSorted.** Obtiene los elementos ordenados por un campo que especifiquemos, así como un tipo de ordenamiento.

```
.findAllSorted("Field", Sort.DESCENDING);
```

Tenemos dos opciones para el almacenamiento de instancias en el contenedor:

- **copyToRealm.** Método que una instancia abierta de un contenedor Realm puede ejecutar para almacenar un objeto soportado por el contenedor, siempre y cuando no exista el objeto en la BD con la misma clave privada.
- **copyToRealmOrUpdate.** Método que, a diferencia del anterior, en caso de que intentemos almacenar un objeto ya existente, lo sobrescribirá.

Finalmente, el borrado de instancias del contenedor se realizará mediante:

- **deleteFirstFromRealm.** Elimina la primera instancia de una colección.
- **deleteLastFromRealm.** Eelimina la última instancia de una colección.
- **deleteFromRealm.** Elimina una instancia específica.
- **deleteAllFromRealm.** Elimina todas las instancias de una colección.

Desarrollo de la aplicación

En este capítulo, acabamos de describir cada uno de los elementos que utilizaremos en el desarrollo de la aplicación. Se trata de una app de hoteles, con sección pública y privada, cuyos requisitos son:

Sección privada:

- Mostrar perfil del usuario.
- Modificar nombre del usuario.
- Modificar imagen del usuario.
- Insertar nuevas opiniones.
- Insertar valoración de un hotel.

Sección pública:

- Listar hoteles.
- Listar opiniones de un hotel.
- Realizar login.
- Registrar usuarios.

Para explicar el desarrollo de esta aplicación, lo haremos por casos de uso. Pese a que ciertos casos de uso podrán ser reutilizados en varias partes de la app, creemos que es la mejor forma de cubrir cada una de las capas de la arquitectura.

Siguiendo con el planteamiento de capítulos anteriores, seguimos apostando por la arquitectura limpia, con tres capas: **data** para el acceso a datos, **domain** para los casos de uso y definición de las entidades del dominio y, por último, la capa **presentation**, para todo lo relativo a la interfaz, vista y presentadores.

En la comunicación entre capas seguimos empleando el patrón **MVP**, Modelo Vista Presentador, en combinación con el patrón observador de **RxJava**. Utilizaremos Dagger v.2.11 en la inyección de dependencias como habíamos visto hasta ahora.

Para no ser repetitivos, vamos a obviar la configuración de Dagger en el proyecto; por tanto, partiremos de que hemos realizado dicha configuración en el nuevo proyecto, así como la división de las tres capas de la arquitectura.

Entidades del dominio

Las entidades del dominio las definimos en el paquete **domain/model**, en la capa de dominio. Dado que vamos a gestionar sesiones de usuarios, necesitaremos una clase que represente la información que almacenaremos sobre la sesión del usuario. Por claridad, omitimos los getters y setters.

```
public class User {
    private String UserId;
    private String Name;
    private String Email;
    private String PhotoUrl;
    private String Provider;
    private String AuthToken;
}
```

Representamos la información de los hoteles mediante la clase **Hotel**.

```
public class Hotel {
    private int id;
    private String country;
    private Location location;
    private String name;
    private Double rating;
    private String town;
    private String web;
    private String photoUrl;
}
public class Location {
         private double latitude;
         private double longitude;
}
```

Para las opiniones de usuarios de un hotel, utilizamos la clase **Opinion**.

```
public class Opinion {
         private int rating;
         private String message;
         private Date creationDate;
         private String name;
}
```

La valoración de hotel de cada usuario viene definida por la clase **Rating**.

```
public class Rating {
        private String userId;
        private int rate;
}
```

Por último, tenemos una serie de constantes de posibles valores que utilizará el usuario para identificar el tipo de proveedor de la sesión.

```
public class SessionProvider {
        public final static String GOOGLE = "google.com";
        public final static String FACEBOOK = "facebook.com";
        public final static String EMAIL = "password";
        public final static String NONE = "none";
}
```

Crear proyecto en Firebase

Entramos en la consola de desarrolladores Firebase https://console.firebase.google.com, con nuestra cuenta Google, creamos un nuevo proyecto, **HotelsMVP**, y pulsamos sobre el botón **Crear proyecto**.

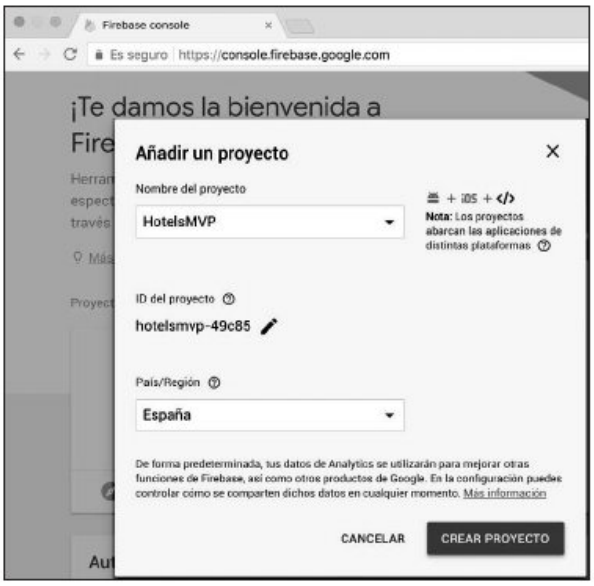


Fig. 3.7.2.1. Nuevo proyecto Firebase.

Para continuar con la configuración del proyecto, pulsamos sobre el icono de configuración que se encuentra en **Project Overview**, para añadir Firebase a nuestra aplicación Android, generando el fichero de configuración de los servicios de Google.

Algunos servicios de Google requieren el certificado SHA-1, que podemos obtener ejecutando el comando siguiente en la consola terminal de Mac:

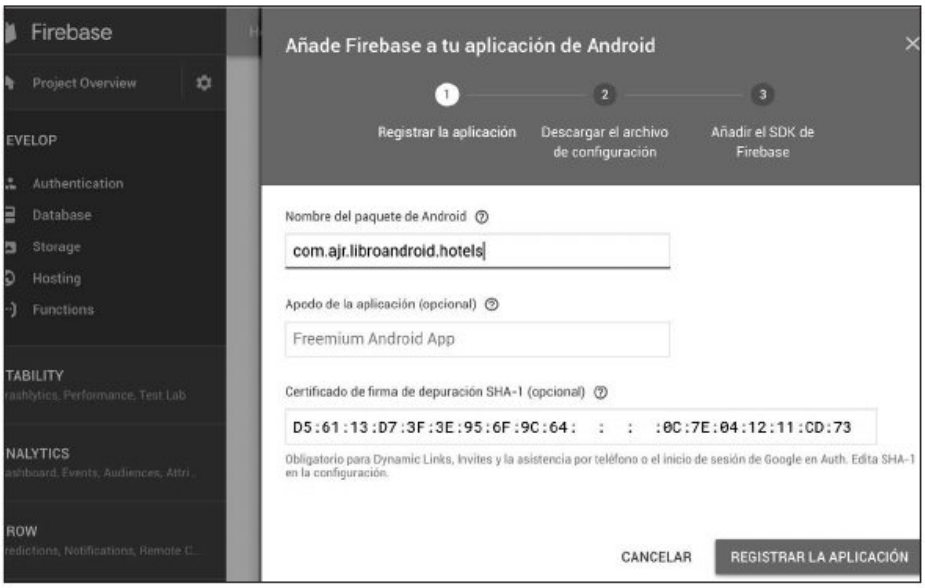
```
keytool -exportcert -list -v -alias androiddebugkey -keystore ~/
.android/debug.keystore
```

O desde la consola de Windows, cambiando la última parte del comando anterior por:

```
%USERPROFILE%\.android\debug.keystore
```

Aquí es donde se encuentra la clave de almacenamiento por defecto de Android, son los **keystore**. Cuando nos solicite contraseña, no indicaremos ninguna y, tras pulsar Intro, nos mostrará el certificado de firma **SHA-1**.

Pegaremos este certificado SHA-1 en la pantalla de configuración que acabamos de abrir en Firebase y lo utilizaremos para obtener el fichero de configuración de servicios de Google, **google-services.json**, que pegaremos en la raíz del módulo de presentación de nuestro proyecto.



Tras añadir este fichero que acabamos de generar, incluiremos los servicios de Google en las dependencias del proyecto del archivo gradle:

```
classpath 'com.google.gms:google-services:3.1.0'
```

Aplicamos el plugin de los servicios de Google, indicándolo al final del fichero gradle del módulo presentación del proyecto:

```
apply plugin: 'com.google.gms.google-services'
```

Por último añadimos, en el gradle del módulo presentación, así como también en el gradle del módulo de datos, las dependencias de las librerías necesarias de Firebase, es decir, la de autenticación:

```
compile 'com.google.firebase:firebase-auth:11.8.0'
```

Una vez que sincronicemos el proyecto, ya lo tendremos asociado con Firebase.

Caso de uso - Comprobar sesión de usuario activa

Como hemos visto en capítulos anteriores, hemos definido los casos de uso en el dominio de la aplicación, en el paquete **interactor**. Hemos considerado empezar la aplicación realizando la comprobación de si hay alguna sesión de usuario abierta o no. Para ello, describiremos qué debemos efectuar en cada capa para completar este caso de uso. Observemos el siguiente diagrama para tener una visión global de cada área afectada.

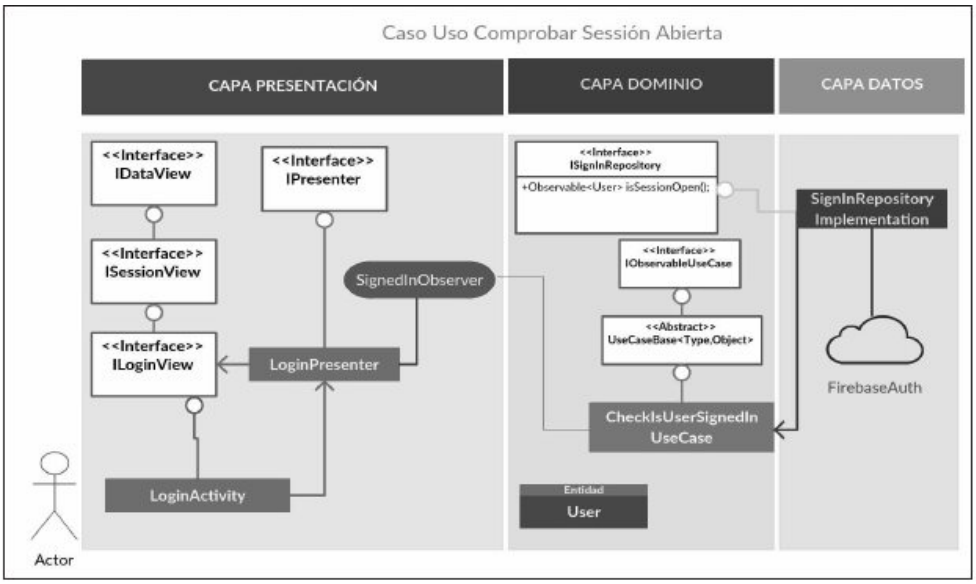


Fig. UC1D. Diagrama caso de uso.

Comenzamos por definir la actividad **LoginActivity**, encargada de gestionar login, registro y cierre de sesión. Esta actividad estará compuesta por una serie de botones: un botón para iniciar sesión con Facebook, otro para Google, otro para iniciar sesión con email y contraseña, otro para crear una nueva cuenta con email y contraseña y, por último, un botón de cierre de sesión.

Pues bien, la idea es que, cuando tengamos una sesión abierta, únicamente mostremos el botón de cerrar la sesión abierta; en caso contrario, mostraremos el resto de los botones y ocultaremos el de cierre de sesión.

En la capa de dominio, debemos crear el paquete **repositories** y una interfaz **ISignInRepository** que defina los métodos del repositorio; por ahora solo crearemos el método que necesitamos.

```
public interface ISignInRepository { Observable<User> isSessionOpen(); }
```

En esta misma capa, pero en el paquete **interactor**, definimos el caso de uso **CheckIsUserSignedInUseCase**, que implementaremos para saber si hay una sesión activa o no.

```
public class CheckIsUserSignedInUseCase extends UseCaseBase<User, Void> {…
 @Inject public CheckIsUserSignedInUseCase(
         ISignInRepository iSignInRepository, Scheduler schedulerThread) {
```

```
        this.iSignInRepository = iSignInRepository;
        this.schedulerThread = schedulerThread;
 }
 @Override public Observable<User> implementUseCase(
        DisposableObserver observer, Void parameters) {
                Observable<User> observable = iSignInRepository.
isSessionOpen();
                this.createUseCase(observable, observer, schedulerThread);
                return observable;
 }
}
```

Recordemos que ya hemos visto **UseCaseBase** en otros capítulos; así que, por simplificar, no lo explicaremos de nuevo, ya que se trata de la misma clase.

En la capa de datos, dado que vamos a trabajar con el objeto de usuario que nos retorna Firebase, **FirebaseUser**, debemos mapear el usuario de Firebase a uno entendible por nuestro dominio, es decir, a nuestra clase **User**. Para ello creamos la clase **FirebaseUserToUser** en el paquete **mapper**, de modo que pasándole el usuario de Firebase lo convirtamos a nuestra clase **User**.

Identificaremos, a través de la instancia de usuario Firebase, el tipo de proveedor con el que se ha abierto la sesión, mediante **getProviders**, así como el identificador único del usuario, mediante **getUid**, y el nombre **getDisplayName** o la imagen del perfil **getPhotoUrl**.

Por claridad, no mostramos el código completo, pero sí las llamadas que nos proporcionan la información deseada para mapear el objeto **FirebaseUser**.

```
String provider = firebaseUser.getProviders().get(0);
String uid = firebaseUser.getUid();
String email = firebaseUser.getEmail();
String name = firebaseUser.getDisplayName();
String photoUrl = firebaseUser.getPhotoUrl().toString();
```

Implementamos la interfaz **ISignInRepository** mediante la clase **SignInRepositoryImplementation**, a la que le pasamos por ahora el contexto y la instancia de **FirebaseAuth** que está inyectada en la capa de presentación.

```
@Singleton
public class SignInRepositoryImplementation implements
ISignInRepository {
    private FirebaseAuth firebaseAuth;
    private Context context;
    @Inject
    public SignInRepositoryImplementation(Context context,FirebaseAuth
firebaseAuth) {
        this.context = context;
        this.firebaseAuth = firebaseAuth;
    }
…
```

Ahora implementamos el método **isSessionOpen**, que comprobará si tenemos usuario. Al utilizar la instancia de FirebaseAuth e invocando su método **getCurrentUser**, sabremos que tenemos un usuario con sesión abierta, en caso de que retorne un FirebaseUser no nulo. Sin embargo, en caso de retornar un usuario válido, llamaremos al método **getTokenOnSuccessFulSignIn** y buscaremos su AccessToken para asignárselo al objeto **User** que retornaremos.

```
…
    @Override
    public Observable<User> isSessionOpen() {
        return Observable
                .create(emitter -> {
                    try {
                        if (firebaseAuth.getCurrentUser() == null)
                 emitter.onError(new Exception(context.getString
(R.string.there_is_not_active_user)));
                        else
                            getTokenOnSuccessFulSignIn(emitter);
                    } catch (Exception e) {
                        emitter.onError(e);
                    }
                });
    }
…
```

Al obtener la sesión de usuario que hay abierta, lo convertimos a un objeto **User** reconocible por nosotros mediante el mapper que creamos anteriormente.

```
…
private void getTokenOnSuccessFulSignIn(ObservableEmitter<User> emitter) {
User user = FirebaseUserToUser.Create(firebaseAuth.getCurrentUser());
…
```

Posteriormente obtenemos el token del usuario, sin forzar el refresco, mediante el método **getIdToken**, retornando un resultado con el token activo del usuario. Nosotros lo asignaremos a nuestro objeto **User** y lo retornaremos hacia capas inferiores.

```
…
        firebaseAuth.getCurrentUser().getIdToken(false)
                .addOnCompleteListener(result -> {
                    user.setAuthToken(result.getResult().getToken());
                        emitter.onNext(user);
                        emitter.onComplete();
                });
    }
}
```

Más adelante veremos por qué no hemos guardado el objeto usuario, añadiendo una línea al método descrito anteriormente.

Volviendo a la capa de presentación, definimos la vista **ILoginView** de nuestra LoginActivity, que a su vez extiende de otra interfaz reutilizable, llamada **ISessionView**, de modo que definiremos ambas.

```
public interface ILoginView extends ISessionView { }
public interface ISessionView extends IDataView {
         void updateUI(User user);
void onSignedIn();
         void onSignedOut();
}
```

La interfaz **ISessionView** será utilizada por toda Activity que requiera implementar métodos relacionados con la sesión de usuario:

- **updateUI.** Muestra información del usuario conectado.
- **onSignedIn.** Indica que hay sesión abierta.
- **onSignedOut.** Indica que no hay sesión abierta.

Para observar los cambios de estado del caso de uso de comprobar si existe una sesión abierta, definimos en el paquete **Observer** de esta capa un observador **SignedInObserver**, que en caso de éxito recibirá un usuario por el método **onNext**, e invocaremos al método **updateUI** de la interfaz **ISessionView**, así como al método **onSignedIn**, para mostrar el botón de cierre de sesión. En caso de error, invocamos al método **onSignedOut** para mostrar los botones de login.

```
public class SignedInObserver extends DisposableObserver<User> {…
    public SignedInObserver(ISessionView iSessionView)
{ this.iSessionView = iSessionView; }
    @Override
public void onNext(User value) {
        iSessionView.updateUI(value);
        iSessionView.onSignedIn();
    }
    @Override
    public void onError(Throwable e) { iSessionView.hideLoading();
        iSessionView.onSignedOut(); }
    @Override
    public void onComplete() { iSessionView.hideLoading(); }
}
```

En principio, el resto de los observadores que crearemos más adelante serán muy similares a este; así que, por claridad, cuando describamos otros observadores, tomaremos como plantilla este observador e indicaremos qué métodos invoca en caso de éxito o error.

En el paquete de **mvp/presenter**, debemos crear el presentador **LoginPresenter** para la LoginActivity. Inicialmente, su constructor tendrá un caso de uso, el de este capítulo; sin embargo, lo iremos ampliando a medida que expliquemos otros casos de uso.

```
public class LoginPresenter implements IPresenter {…
    @Inject public LoginPresenter(ILoginView iLoginView,
                                  CheckIsUserSignedInUseCase
checkIsUserSignedInUseCase) {
        this.iLoginView = iLoginView;
        this.checkIsUserSignedInUseCase = checkIsUserSignedInUseCase;
    }
…
```

Inicializamos el presentador, pasándole el contexto de la Activity, e invocamos al método **isSignedIn** para que compruebe mediante el caso de uso **checkIsUserSignedInUseCase** si hay alguna sesión activa. Requiere un observador **SignedInObserver** que hemos creado anteriormente.

```
…
    public void initialize(Context context) {
        this.context = context;
        isSignedIn();
    }
    public void isSignedIn() {
         checkIsUserSignedInUseCase.implementUseCase(
         new SignedInObserver(iLoginView),null);
}
…
```

En el paquete **module**, creamos el módulo **LoginModule** para la Activity, en el que proveemos la vista **ILoginView** y el presentador **LoginPresenter**. De momento, en el constructor de este presentador incluimos el caso de uso que estamos explicando; sin embargo, más adelante, tendremos que ampliar este constructor para añadir un par de casos de uso adicionales.

```
@Module
public abstract class LoginModule {
@Provides static LoginPresenter provideLoginPresenter(
         ILoginView iLoginView,
         CheckIsUserSignedInUseCase checkIsUserSignedInUseCase) {
         return new LoginPresenter(iLoginView,
         checkIsUserSignedInUseCase); }
         @Binds abstract ILoginView provideLoginView(LoginActivity
loginActivity);
}
```

En la clase **BuildersModule**, recordemos que tenemos que especificar cada una de las Activity que vayamos creando, con su módulo, es decir, para el caso del login incluiremos estas líneas, aunque irá creciendo conforme vayamos incorporando nuevas Activity.

```
@Module public abstract class BuildersModule {
@PerActivity
@ContributesAndroidInjector(modules = LoginModule.class)
abstract LoginActivity contributeLoginActivity();
}
```

Por último tendríamos que proveer de las dependencias que hemos necesitado en las capas superiores, en el módulo **AppModule**, como la instancia de autenticación de Firebase, **FirebaseAuth**, así como la instancia de la clase que implementa el repositorio **ISignInRepository**.

```
@Provides @Singleton static ISignInRepository providesSignInRepository(
SignInRepositoryImplementation signInRepositoryImplementation)
                 { return signInRepositoryImplementation; }
@Provides @Singleton static FirebaseAuth provideFirebaseAuth()
                 { return FirebaseAuth.getInstance(); }
```

Una vez configurado lo anterior, podemos centrarnos ya en la parte Activity. Lo primero que haremos será importar la librería Butterknife a las dependencias del fichero gradle del módulo de presentación.

```
compile 'com.jakewharton:butterknife:8.6.0'
annotationProcessor 'com.jakewharton:butterknife-compiler:8.6.0'
```

Esto nos permitirá inyectar las vistas en la Activity y utilizarlas siempre que inicialicemos esta librería en el método **onCreate** de **LoginActivity**.

```
ButterKnife.bind(this);
```

Definimos con Butterknife cada uno de estos botones, incluido un **progress-bar**, a través de **@BindView**.

```
@BindView(R.id.login_progress)                          ProgressBar login_
progress;
@BindView(R.id.login_signInFacebook_btn) Button login_signInFacebook_btn;
@BindView(R.id.login_signInGoogle_btn)   Button login_signInGoogle_btn;
@BindView(R.id.login_email_btn)                Button login_email_btn;
@BindView(R.id.login_register_btn)       Button login_register_btn;
@BindView(R.id.login_signOut_btn)              Button login_signOut_btn;
```

Inyectamos la Activity para que Dagger la tenga en consideración.

```
AndroidInjection.inject(this);
```

La Activity debe implementar la interfaz de la vista **ILoginView**; además debemos inyectar en la Activity, mediante **@Inject**, el presentador **LoginPresenter**, así como inicializar el presentador con el contexto, para que compruebe si tenemos o no sesión activa.

```
public class LoginActivity extends AppCompatActivity implements
ILoginView {…
@Inject LoginPresenter loginPresenter;
@Override protected void onCreate(Bundle savedInstanceState)
{ … loginPresenter.initialize(this); … }
```

Tras comprobar si tenemos sesión activa, en caso de tenerla, en el método sobrescrito de la interfaz **updateUI**, recibiremos el usuario y mostraremos en un Toast su nombre; además, en el método **onSignedIn**, mostraremos el botón de cierre de sesión y ocultaremos el resto de los botones.

Sin embargo, en caso contrario, al no tener sesión activa, ocultaremos el botón de cierre de sesión y mostraremos los restantes botones. Al ejecutar la app, como vemos en la siguiente imagen, se mostrarán inicialmente todos los botones.

Fig. UC1. Comprobar sesión activa.

Caso de uso - Configurar Facebook

Queremos incorporar a nuestra aplicación el login de usuario a través de una cuenta Facebook, para que un usuario pueda acceder a la parte privada de la aplicación sin la necesidad de introducir nombre de usuario o contraseña para acceder.

Firebase nos proporciona este sistema de autenticación, de modo que debemos habilitar en la consola de Firebase, en la sección de **Authentication**, el método de inicio de sesión de Facebook. Sin embargo, necesitamos introducir un identificador de aplicación, así como el secreto de la aplicación para habilitarlo. Para obtener esta información, debemos ir a la consola de desarrolladores de Facebook, https://developers.facebook.com, con nuestra cuenta Facebook y darnos de alta como desarrolladores de aplicaciones para Facebook (Figura UC2-1), en caso de que no lo hayamos realizado con anterioridad. Posteriormente verificaremos nuestra cuenta, tras recibir un SMS con un código que debemos introducir (Figura UC2-2).

Fig. UC2-1. Registro desarrollador Facebook.

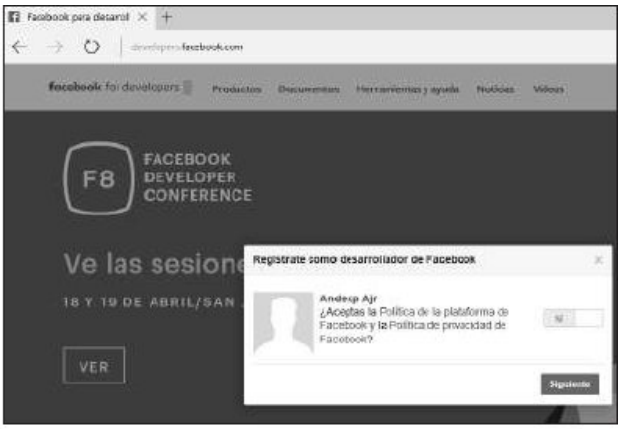


Fig. UC2-2. Verificar cuenta.

Una vez que nos hemos dado de alta como desarrolladores de Facebook, ya podemos crear una aplicación. Para ello, introducimos un nombre de aplicación y nuestro email, como vemos en la imagen.

Fig. UC2-3.
Crear aplicación Facebook.

Una vez creada, se mostrará el panel de control de la aplicación; en él debemos añadir el producto de inicio de sesión con Facebook (Figura UC2-4), pulsando sobre su botón **Configurar**, y posteriormente seleccionaremos el inicio rápido para la plataforma **Android**.

Fig. UC2-4. Configurar inicio de sesión Facebook.

Al configurar el inicio de sesión con Facebook desde Android, debemos seguir una serie de pasos:

1. Importar el SDK de Facebook a las dependencias del fichero gradle de nuestro módulo presentación y sincronizar el proyecto.

```
implementation 'com.facebook.android:facebook-login:[4,5)'
```

2. En el wizard de Facebook, indicamos el nombre del paquete de nuestro proyecto, es decir, **com.ajr.libroandroid.hotels**, y el nombre de la Activity que gestione enlaces profundos, en nuestro caso, **com.ajr.libroandroid.hotels.ui.LoginActivity**, y guardamos dicha configuración para pasar al siguiente punto.
3. Necesitamos añadir los hashes clave de desarrollo; para ello debemos tener instalado el entorno de desarrollo Java, Java Development Kit o JDK, que nos permita utilizar la herramienta **keytool** de generación de claves. Probablemente ya lo tengamos instalado en nuestro ordenador tras configurar el entorno de desarrollo; en caso contrario, deberemos instalarlo.

Mediante la ejecución del siguiente comando en la consola **Terminal**, obtenemos una clave hash que contiene el carácter "=" al final de la cadena:

Desde Mac:

```
keytool -exportcert -alias androiddebugkey -keystore
~/.android/debug.keystore | openssl sha1 -binary | openssl base64
```

Desde Windows:

```
keytool -exportcert -alias androiddebugkey -keystore "C:\Users\USERNAME\.
android\debug.keystore" | "PATH_TO_OPENSSL_LIBRARY\bin\openssl" sha1
-binary |          "PATH_TO_OPENSSL_LIBRARY\bin\openssl" base64
```

El comando ejecutado es **keytool -exportcert**, al que le indicamos el alias y almacén de claves generado previamente. Tras su ejecución, nos solicitará que introduzcamos la contraseña de acceso al almacén de claves, con lo que obtendremos la clave de desarrollo **1WET1z8+lW+cZElfvgx+BBIRzXM=**, que deberemos copiar y pegar en la sección **Añadir hashes de clave de desarrollo y activación** del wizard de desarrolladores de Facebook. A continuación pulsaremos sobre **Guardar**.

4. Vamos al proyecto Android, abrimos el recurso **strings.xml** de la capa de presentación y añadimos dos claves:

```
<string name="facebook_app_id">180194992773675</string>
<string name="fb_login_protocol_scheme">fb180194992773675</string>
```

5. En el proyecto Android, abrimos el **AndroidManifest.xml** y pegamos el siguiente elemento meta-data dentro del elemento **application**.

```
<meta-data android:name="com.facebook.sdk.ApplicationId"
    android:value="@string/facebook_app_id"/>
<activity android:name="com.facebook.FacebookActivity"
    android:configChanges="keyboard|keyboardHidden|screenLayout
|screenSize|orientation"
    android:label="@string/app_name" />
<activity android:name="com.facebook.CustomTabActivity"
    android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.BROWSABLE" />
        <data android:scheme="@string/fb_login_protocol_scheme" />
    </intent-filter>
</activity>
```

Volvemos a la consola de desarrolladores de Facebook y buscamos la sección **Configuración > Información básica**, donde se encuentra información sobre nuestra app Android, por ahora inactiva.

Para activarla, necesitamos indicar una URL de política de privacidad. Si no tenemos una página de política de privacidad, nos crearemos una, por ejemplo, en https://es.wix.com, donde podemos elegir una plantilla y crear una página web gratuita con dicha información. Existen webs que generan plantillas de privacidad. Bastará con poner el nombre de vuestra compañía y se generará la política de privacidad.

Pegamos nuestra URL donde hemos subido nuestra política de privacidad, en el apartado correspondiente de la información básica, indicamos la categoría de "ocio" y, finalmente, activamos el modo de desarrollo a público mediante el botón que encontraremos en la zona superior derecha de la interfaz, que por defecto está desactivado. Podemos ver cómo hemos configurado esta pantalla en la Figura UC2-5.

Es en esta pantalla donde tenemos información sobre la Id de la aplicación que requiere Firebase y la clave secreta de la aplicación, de modo que los copiamos para pegarlos en la consola Firebase, en la sección de **Authentication**, (Figu-

ra UC2-6). Además copiaremos la URI de redirección de OAuth, que aparece más abajo, para permitir esta URI en Facebook.

Esta URI de redirección copiada de Firebase la pegamos en Facebook, en la sección **Inicio de sesión con Facebook > Configuración > Configuración del cliente de OAuth > URI de redireccionamiento de OAuth válidos** y guardamos.

https://hotelsmvp-49c85.firebaseapp.com/___/auth/handler

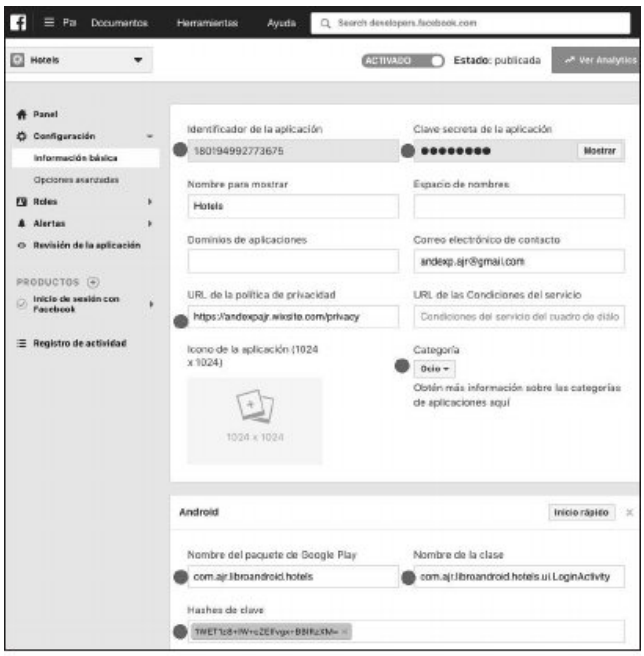


Fig. UC2-5. Configuración básica Facebook.

Con esto ya tendríamos habilitado Facebook en nuestra aplicación Android, así como en Firebase, de modo que este estaría autorizado para gestionar cuentas de Facebook.

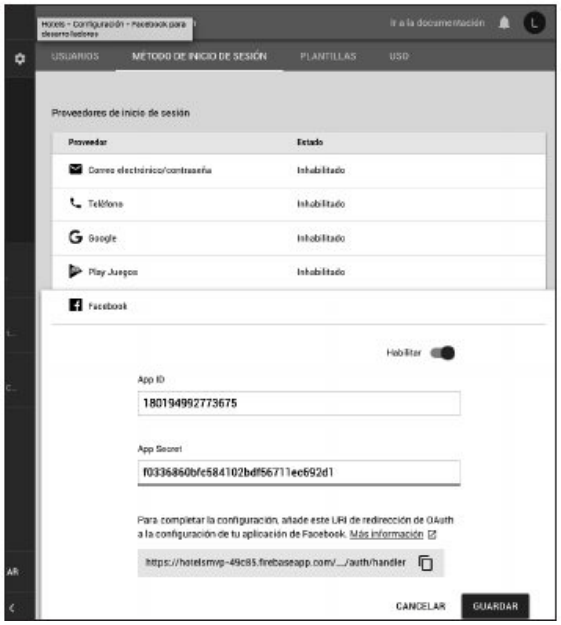


Fig. UC2-6. Habilitar Facebook en Firebase.

Caso de uso - Login Facebook

Para continuar con la funcionalidad, vamos a preparar la aplicación para soportar login de Facebook. Describiremos qué debemos realizar en cada capa para completar este caso de uso. Observemos el diagrama de la imagen para tener una visión global de cada área afectada.

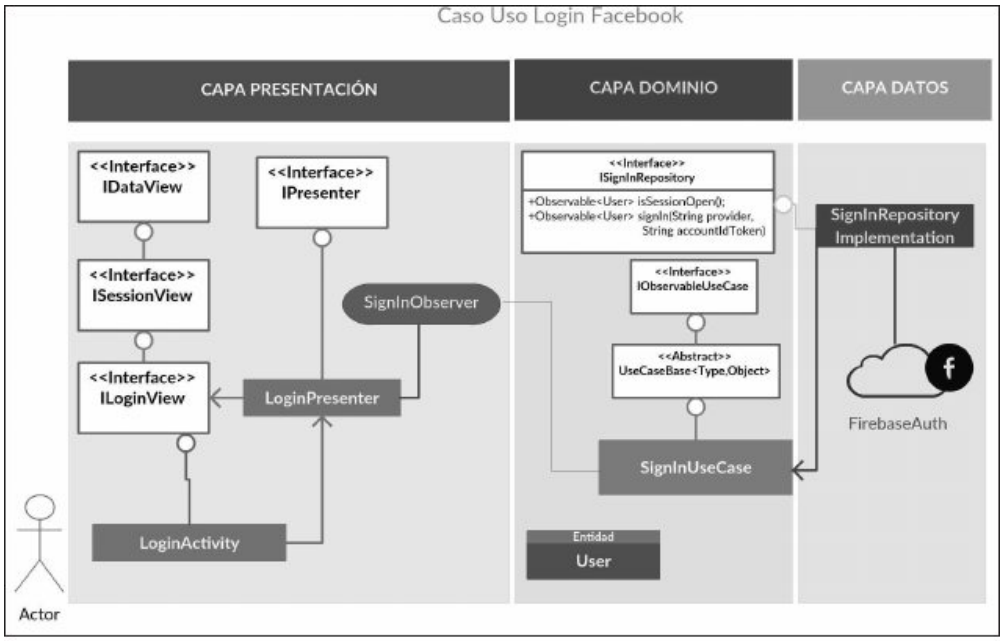


Fig. UC3D. Diagrama del login.

Empecemos por la capa del dominio; en el paquete de repositorios, añadimos a la interfaz del repositorio **ISignInRepository** un nuevo método responsable de retornar un observable con la sesión del usuario conectado, tras enviarle el TokenID y el proveedor elegido para obtener las credenciales.

```
Observable<User> signIn(String provider, String accountIdToken)
```

Para el caso de uso que vamos a crear, dado que requiere una serie de parámetros como el tipo de proveedor y el TokenID, debemos crear una clase llamada **SignInParameters**, que contendrá esta información, así como un método **Create** para instanciar esta clase con los valores rellenos. Colocaremos esta clase dentro del paquete **domain/interactor/parameters**.

```
public class SignInParameters {
    public static final class Parameters {
         private String sessionProvider;
         private String accountIdToken;
…
         public static Parameters Create(
                 String sessionProvider, String accountIdToken) {
```

Descargado en: eybooks.com

```
                return new Parameters(sessionProvider, accountIdToken);
        }
}
}
```

Para terminar con la capa de dominio, definimos el caso de uso responsable de realizar el login de Facebook, en el paquete interactor. Este caso de uso retornará un Observable de tipo **User**, y se recibirán como parámetros el proveedor seleccionado para el login y el TokenID necesario para la obtención de las credenciales.

```
public class SignInUseCase extends UseCaseBase<User,SignInParameters.
Parameters> {
private ISignInRepository iSignInRepository;
private Scheduler schedulerThread;
@Inject
public SignInUseCase(ISignInRepository iSignInRepository,
                        Scheduler schedulerThread) {
        this.iSignInRepository = iSignInRepository;
        this.schedulerThread = schedulerThread;
}
@Override public Observable<User> implementUseCase(DisposableObserver
observer,
SignInParameters.Parameters parameters) {
Observable<User> observable = iSignInRepository
.signIn(parameters.getSessionProvider(),parameters.getAccountIdToken());
this.createUseCase(observable, observer, schedulerThread);
return observable;
}
}
```

En la capa de datos, en la clase que implementa el repositorio **ISignInRepository**, es decir, **SignInRepositoryImplementation**, implementamos el método **signIn** que definimos anteriormente.

En caso de que el proveedor sea Facebook, utilizaremos el proveedor de Facebook, **FacebookAuthProvider**, para obtener las credenciales **AuthCredential**, al que le pasamos el TokenID de acceso y nos proporcionará las credenciales necesarias para hacer login en Firebase.

Mediante la instancia de autenticación de Firebase, **firebaseAuth**, invocamos su método de inicio de sesión en Facebook, **signInWithCredential**, empleando las credenciales que acabamos de obtener de Facebook. Finalmente le asignamos un listener para que nos avise cuando haya terminado.

A través de las credenciales en Facebook, obtenemos mediante una **task** la respuesta de inicio de sesión por parte de Firebase. El caso de éxito lo establece su método **isSuccessful**, de modo que obtendríamos el usuario que acaba de abrir sesión y llamaríamos a **getTokenOnSuccessFulSignIn**, que ya vimos con anterioridad, responsable de retornar el usuario a las capas inferiores.

Cabe destacar que el método de **signInWithCredential**, en caso de que no exista el usuario de Facebook en Firebase, lo creará y, en caso contrario, es decir, que sí exista en Firebase, realizará el login directamente.

```
@Override
public Observable<User> signIn(String provider, String accountIdToken) {
    return Observable.create(emitter -> {
try {
        AuthCredential credential = null;
```

```
        if (provider.equals(SessionProvider.FACEBOOK))
                credential = FacebookAuthProvider.
getCredential(accountIdToken);             firebaseAuth.
signInWithCredential(credential).addOnCompleteListener(
        task -> {
                        if (task.isSuccessful())
getTokenOnSuccessFulSignIn(emitter);
                        else emitter.onError(task.getException());
                });
} catch (Exception e) { emitter.onError(e); } });
}
```

Pasamos a la capa de presentación y creamos un observador que esté a la escucha de la respuesta del login. En caso de éxito mostraremos, por la interfaz de usuario, mediante **updateUI**, información del usuario conectado, así como el botón de cierre de sesión.

```
public class SignInObserver extends DisposableObserver<User> {
…
@Override public void onNext(User value) {
        iSessionView.updateUI(value);
        iSessionView.onSignedIn();
}
…
}
```

Añadimos al módulo **LoginModule** el nuevo caso de uso que acabamos de añadir, dado que el constructor del presentador **LoginPresenter** incorporará el nuevo caso de uso creado.

```
@Provides static LoginPresenter provideLoginPresenter(
…, SignInUseCase signInUseCase, …)
{ return new LoginPresenter(…,signInUseCase,…);}
```

En el presentador **LoginPresenter**, que se encuentra en el paquete **presenter**, le añadimos este caso de uso en el constructor.

```
@Inject public LoginPresenter(ILoginView iLoginView, SignInUseCase
signInUseCase, CheckIsUserSignedInUseCase checkIsUserSignedInUseCase) {
```

Debido a que tenemos que controlar el resultado del **callbackManager** de Facebook, definimos el método **handleFacebookResult** en el **LoginPresenter**, el cual recibe por parámetro un **LoginResult** que contiene información de acceso del usuario a Facebook.

Es aquí donde obtenemos el token de acceso, necesario para obtener las credenciales en la capa de datos. Lo obtenemos invocando al método **getAccessToken()**.**getToken()**, del **LoginResult**, para retornar el token en una cadena, utilizado por el caso de uso por ejecutar, **signInUseCase**.

```
public void handleFacebookResult(LoginResult loginResult) {
   iLoginView.showLoading();
   String accessToken = loginResult.getAccessToken().getToken();
   signInUseCase.implementUseCase( new SignInObserver(iLoginView, context),
 SignInParameters.Parameters.Create(SessionProvider.FACEBOOK,
```

```
accessToken));
}
```

En **LoginActivity**, inyectamos el CallbackManager de Facebook.

```
@Inject CallbackManager callbackManager;
```

Definimos el evento clic del botón de login de Facebook con Butterknife. Al no utilizar el botón nativo de login de Facebook, necesitamos el LoginManager.

Obtenemos la instancia del LoginManager de Facebook e indicamos que queremos recibir información sobre email y perfil público, para que nos concedan acceso de lectura y escritura.

```
@OnClick(R.id.login_signInFacebook_btn)
protected void doFacebookSignIn() {
 LoginManager.getInstance().logInWithReadPermissions(this, Arrays.
asList("email", "public_profile"));
…
```

Asociamos un **callbackManager** a la instancia del LoginManager de Facebook. En caso de éxito **onSuccess**, el presentador maneja la respuesta de Facebook.

```
…
    LoginManager.getInstance().registerCallback(callbackManager,
            new FacebookCallback<LoginResult>() {
                @Override public void onSuccess(LoginResult loginResult)
                            { loginPresenter.
handleFacebookResult(loginResult); }
          @Override public void onCancel() { }
          @Override public void onError(FacebookException error) { }});}
```

Cuando recibimos respuesta del intent lanzado por Facebook en el inicio de sesión, si el código de respuesta es el de login, invocamos al **callbackManager** de Facebook, indicando que hemos recibido una respuesta; el FacebookCallback se encargará entonces de analizar si hubo éxito o no en el login hacia Facebook.

```
@Override
public void onActivityResult(int requestCode, int resultCode,
Intent data) {    super.onActivityResult(requestCode, resultCode
, data);
if (requestCode ==CallbackManagerImpl.RequestCodeOffset.Login.
toRequestCode())
         callbackManager.onActivityResult(requestCode, resultCode, data);
}
```

Por último proveemos de la dependencia del **CallbackManager** de Facebook para proveer su instancia en la clase **AppModule**.

```
@Provides @Singleton static CallbackManager provideCallbackManager()
                    { return CallbackManager.Factory.create(); }
```

Ejecutamos la app y pulsamos el botón de login de Facebook. Aparecerá un popup donde aceptaremos los permisos para obtener información del email y

perfil de la cuenta Facebook con la que nos conectamos, como vemos en la Figura UC3-1.

Tras continuar como usuario de Facebook, se recibiría el activityResult y el callbackManager capturaría el LoginResult, invocando al presentador para que realice el login en Firebase. Tras la respuesta de éxito de login satisfactorio en Firebase, mostraríamos el botón de cierre de sesión e información del usuario conectado mediante un Toast, como observamos en la Figura UC3-2.

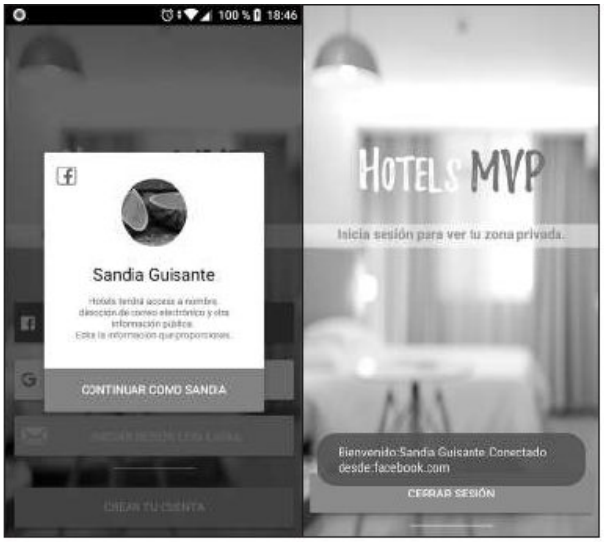


Fig. UC3-1. Popup Facebook.
Fig. UC3-2. Logged Facebook.

Caso de uso - Login Google

Google, uno de los proveedores que vamos a utilizar en nuestro proyecto, permite realizar login a través de una cuenta de Google propia. Muchos usuarios tienen una cuenta Google y más teniendo en cuenta que se trata de una app para Android, donde vinculamos nuestra cuenta Google con el dispositivo.

Para empezar, habilitamos en Firebase la autenticación mediante proveedor Google, en la misma pantalla de Firebase donde efectuamos el caso de Facebook. Sección **Authentication > Método de Inicio de sesión > Google > Habilitar**.

En las capas de dominio y datos, vamos a reutilizar el caso de uso de Facebook, añadiendo en la capa de datos, en el método **signIn**, de la clase **SignInRepositoryImplementation**, la siguiente comprobación.

Identificamos si el tipo de proveedor que le indicamos por parámetro es Google, en cuyo caso utilizamos el proveedor de Google **GoogleAuthProvider** para adquirir las credenciales. Tras obtenerlas, reutilizaríamos la llamada de Firebase **signInWithCredential** pasándole estas credenciales.

```
if (provider.equals(SessionProvider.GOOGLE))
 credential = GoogleAuthProvider.getCredential(accountIdToken,null);
else …
```

En la capa de presentación, importamos la librería de autenticación de Google al fichero gradle del módulo, para poder utilizar el login con Google.

```
compile 'com.google.android.gms:play-services-auth:11.8.0'
```

Por otra parte, proveemos la dependencia del cliente de la API de Google, **GoogleApiClient**, en el módulo de la aplicación AppModule. Para ello, antes indicamos la configuración de realizar login con la API de Google, definiendo en las **GoogleSignInOptions** que necesitamos solicitar el identificador del token de acceso mediante **requestIdToken**, así como el email a través de **requestEmail**.

Una vez realizada la configuración, indicamos al cliente de la API de Google que queremos utilizar la API de autenticación de Google, es decir, la **GOOGLE_SIGN_IN_API**, y le pasamos la configuración que acabamos de realizar necesaria para esta API. Obtendremos, de esta forma, la instancia de la API de autenticación de Google, que será lo que proveamos.

```
@Provides @Singleton
static GoogleApiClient provideGoogleApiClient(App application) {
 GoogleSignInOptions googleSignInOptions = new GoogleSignInOptions
.Builder(GoogleSignInOptions.DEFAULT_SIGN_IN)
.requestIdToken(application.getString(R.string.default_web_client_id))
.requestEmail()
.build();
 GoogleApiClient googleApiClient = new GoogleApiClient.Builder(application)
.addApi(Auth.GOOGLE_SIGN_IN_API, googleSignInOptions)
.build();
 return googleApiClient;
}
```

Ampliamos los parámetros del constructor del presentador, añadiendo la dependencia de GoogleApiClient, primero en el **LoginModule**.

```
@Provides static LoginPresenter provideLoginPresenter(…,
GoogleApiClient googleApiClient)
{ return new LoginPresenter(…, googleApiClient); }
```

Así como también en el constructor del **LoginPresenter**.

```
private GoogleApiClient googleApiClient;
@Inject public LoginPresenter(… ,GoogleApiClient googleApiClient)
{        … this.googleApiClient = googleApiClient; }
```

Definimos un método llamado **googleSignIn**, en el LoginPresenter, para obtener un intent, a través de la API del cliente de Google, que retornará a la Activity, cuando lo llame, el intent para lanzarlo.

```
public Intent googleSignIn() {
    iLoginView.showLoading();
    return Auth.GoogleSignInApi.getSignInIntent(googleApiClient);
}
```

En este mismo presentador, definimos un método para manejar el resultado obtenido tras invocar el intent de inicio de sesión con Google, y obtenemos el resultado de autenticación invocando al método **getSignInResultFromIntent**, de la API de Google.

En caso de éxito, con este resultado **GoogleSignInResult**, obtenemos la cuenta de Google validada **GoogleSignInAccount**, para finalmente obtener el identificador del token, a través de esta cuenta, mediante **getIdToken**. Este token es enviado al caso de uso **signInUseCase** para, mediante el proveedor de Google y el token, obtener las credenciales en la capa de datos y que, de esta forma, Firebase realice el login o registro de dicho usuario.

```
public void handleGoogleResult(Intent data) {
 GoogleSignInResult googleSignInResult =
          Auth.GoogleSignInApi.getSignInResultFromIntent(data);
if (googleSignInResult.isSuccess()) {
          GoogleSignInAccount googleSignInAccount =
                          googleSignInResult.getSignInAccount();
 signInUseCase.implementUseCase( new SignInObserver(iLoginView, context),
 SignInParameters.Parameters.
Create( SessionProvider.GOOGLE, googleSignInAccount.getIdToken()));
}
}
```

Definimos el evento clic del botón del login de Google, utilizando Butterknife, en la **LoginActivity**. Tras pulsarlo, invocamos al presentador para obtener el intent de Google, para lanzarlo y obtener un **activityResult** por parte de Google.

```
@OnClick(R.id.login_signInGoogle_btn) protected void doGoogleSignIn() {
Intent intentGoogle = loginPresenter.googleSignIn();
startActivityForResult(intentGoogle, Flags.GOOGLE_SIGN_IN);
}
```

Finalmente, en el método sobrescrito de la actividad **onActivityResult**, comprobamos si el intent de respuesta es de Google; en caso afirmativo, manejamos dicho resultado en el presentador, como ya indicamos anteriormente.

```
if (requestCode == Flags.GOOGLE_SIGN_IN)
         loginPresenter.handleGoogleResult(data);
```

Al ejecutar la aplicación, pulsamos el botón de login con Google. Aparecerá una ventana para seleccionar una cuenta Google (Figura UC4-1). Seleccionamos una y, tras recibir la respuesta de Google con la cuenta y, por tanto, su credencial, Firebase registrará el usuario en el sistema, mostrará información del usuario conectado y el botón de cierre de sesión (Figura UC4-2).

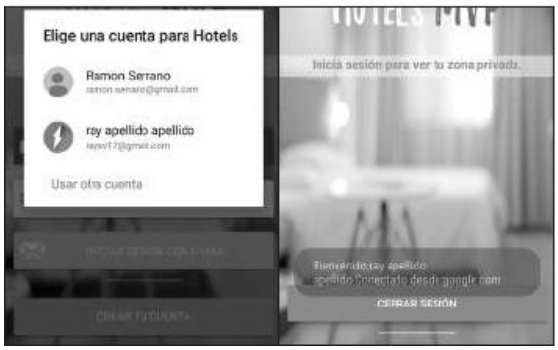


Fig. UC4-1. Selección de cuenta Google.
Fig. UC4-2. Logueado con cuenta Google.

Caso de uso - Registro email con contraseña

Queremos dotar a nuestra app del registro de usuarios, de la forma tradicional, sin proveedores, es decir, mediante un email y una contraseña. Firebase nos facilita la vida, dado que soporta este tipo de validación de usuarios. En el siguiente diagrama podremos hacernos una idea global de este caso de uso.

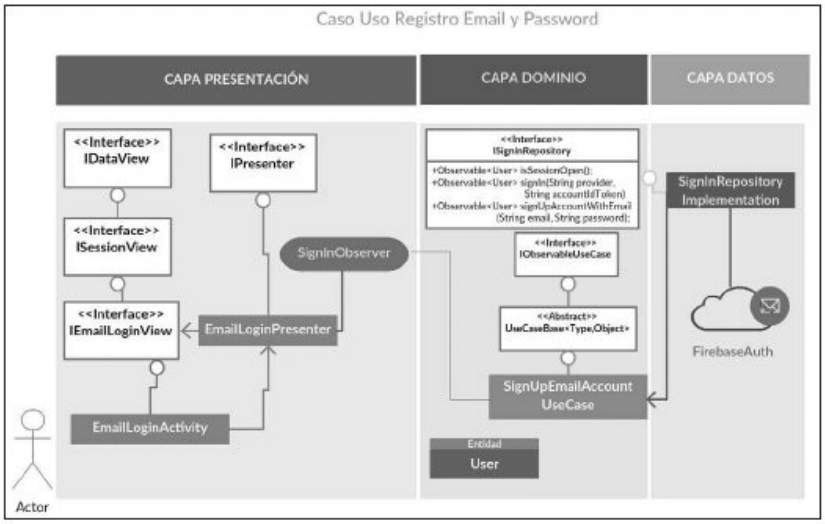


Fig. UC5D. Diagrama Caso de uso registro email.

Para habilitarlo en Firebase, debemos ir a la sección **Authentication > Método de Inicio de sesión > Correo electrónico/contraseña > Habilitar**.

En la capa de dominio, en la interfaz del repositorio **ISignInRepository**, creamos un método para el registro por email.

```
Observable<User> signUpAccountWithEmail(String email, String password);
```

Dado que nuestro caso de uso necesita dos parámetros, uno para el email y otro para la contraseña, creamos una clase que soporte estos dos, en el paquete de parámetros de los interactores del dominio, con el nombre **EmailParameters**.

```
public class EmailParameters {
public static final class Parameters {
private String email, password;…
        public static Parameters Create(String email, String password)
        { return new Parameters(email, password); }
}
}
```

Creamos nuestro caso de uso **SignUpAccountUseCase** en los interactores del dominio. Retornando un observable de tipo usuario y recibiendo como parámetros

el email y la contraseña, necesarios para invocar al método **signUpAccountWithEmail** del repositorio.

```
public class SignUpEmailAccountUseCase
                  extends UseCaseBase<User, EmailParameters.Parameters> {…
  @Inject public SignUpEmailAccountUseCase(
         ISignInRepository iSignInRepository, Scheduler schedulerThread) {
         this.iSignInRepository = iSignInRepository;
         this.schedulerThread = schedulerThread;
  }
  @Override public Observable<User> implementUseCase(
DisposableObserver observer, EmailParameters.Parameters parameters) {
Observable<User> observable =
iSignInRepository.signUpAccountWithEmail(
                  parameters.getEmail(),parameters.getPassword());
         this.createUseCase(observable, observer, schedulerThread);
         return observable;
}
}
```

En la capa de datos, implementamos el método sign**UpAccountWithEmail**, en la clase que implementa la interfaz del repositorio **SignInRepositoryImplementation**. Utilizamos la instancia de autenticación de Firebase **firebaseAuth** e invocamos su método **createUserWithEmailAndPassword**, al que le pasamos un email y una contraseña para registrar el usuario en Firebase.

```
@Override public Observable<User> signUpAccountWithEmail(
                                    String email, String password) {
    return Observable.create(emitter -> {
                  try {
         firebaseAuth.createUserWithEmailAndPassword(email, password)
                             .addOnCompleteListener(task -> {
                                    if(task.isSuccessful())
getTokenOnSuccessFulSignIn(emitter);
                                                 else emitter.onError(task.
getException());
                          });
            } catch (Exception e) { emitter.onError(e); }
});
}
```

Hasta ahora habíamos utilizado solamente la Activity del login; sin embargo, cuando pulsemos sobre el botón **Crear tu cuenta** de la LoginActivity, queremos que se abra una nueva Activity, responsable del login y registro por email. Para ello creamos una actividad vacía **EmailLoginActivity**, inyectándola en el método **onCreate** para Dagger.

```
AndroidInjection.inject(this);
```

Esta Activity tiene su propia vista y presentador, de modo que crearemos primero la interfaz de la vista que implementará, con el nombre **IEmailLoginView**, en el paquete **mvp/view** de la capa de presentación. Definimos una serie de métodos que muestran mensajes de error en caso de email o contraseña incorrecta en la interfaz o los borran al subsanarse el error.

```
public interface IEmailLoginView extends ISessionView {
    void showEmailMessageError(String s);
void showEmailNoErrors();
void showPasswordMessageError(String s);
void showPasswordNoErrors();
}
```

EmailLoginPresenter será el presentador de esta Activity, que por ahora ejecutará el caso de uso que acabamos de definir: **SignUpEmailAccountUseCase**. Su constructor está compuesto por la vista y el caso de uso de registro por email. Añadiremos un método de inicialización, que recibirá el contexto de la Activity.

```
public class EmailLoginPresenter implements IPresenter {
@Inject public EmailLoginPresenter(IEmailLoginView iEmailLoginView,
         SignUpEmailAccountUseCase signUpEmailAccountUseCase) {
         this.iEmailLoginView = iEmailLoginView;
         this.signUpEmailAccountUseCase = signUpEmailAccountUseCase;
}
public void initialize(Context context) { this.context = context; }
…
```

Añadiremos también el método responsable de construir el caso de uso, para registrar al usuario mediante un email y contraseña. En caso de que el email y contraseña sean válidos, ejecutaremos el caso de uso **signUpEmailAccountUseCase** y reutilizaremos el observable **SignInObserver**.

```
…
 public void registerWithEmailAndPassword(String email, String password) {
        if (isValidEmail(email) && isValidPassword(password)) {
            signUpEmailAccountUseCase.implementUseCase(
                    new SignInObserver(iEmailLoginView, context),
                    EmailParameters.Parameters.Create(email, password));
        }
    }
…
```

Como ya tenemos la vista y el presentador definidos, podemos definir el módulo de esta Activity, dentro del paquete **di/module**, de la capa de presentación, con el nombre **EmailLoginModule**.

```
@Module public abstract class EmailLoginModule {
@Provides static EmailLoginPresenter       provideEmailLoginPresenter(
                          IEmailLoginView iLoginView,
                                  SignUpEmailAccountUseCase
signUpEmailAccountUseCase)
  { return new EmailLoginPresenter(iLoginView, signUpEmailAccountUseCase); }

@Binds abstract IEmailLoginView provideEmailLoginView(
                          EmailLoginActivity emilLoginActivity);
}
```

Por último añadimos esta nueva Activity a la clase **BuildersModule**, del paquete **di/builder** de la capa de presentación.

```
@PerActivity @ContributesAndroidInjector(modules = EmailLoginModule.class)
abstract EmailLoginActivity contributeEmailLoginActivity();
```

Inyectamos el presentador en la activity **EmailLoginActivity** y hacemos que la Activity implemente la vista **IEmailLoginView**.

```
@Inject EmailLoginPresenter emailLoginPresenter;
public class EmailLoginActivity extends AppCompatActivity implements
IEmailLoginView { … }
```

Inyectamos las vistas de la interfaz de la Activity mediante Butterknife y el evento clic del botón de registro de nueva cuenta, en el método **doRegisterClick**, el cual invocará al método del presentador responsable de esta tarea de registro.

```
@OnClick(R.id.emailLogin_register_btn)
public void doRegisterClick(View view) {…
emailLoginPresenter.registerWithEmailAndPassword(
         emailLogin_email.getText().toString(),
         emailLogin_password.getText().toString());
}
```

Inicializamos Butterknife y el presentador, en el método **onCreate** de la actividad.

```
lLoginPresenter.initialize(this);
```

Por último, en el AndroidManifest.xml de esta capa, indicamos que esta Activity tiene una Activity padre, que es **LoginActivity**. De esta forma, cuando pulsemos atrás, volverá a la Activity que la llamó, es decir, la del login.

```
<activity android:name=".ui.EmailLoginActivity"  android
:parentActivityName=".ui.LoginActivity" />
```

Y como la actividad padre es LoginActivity, esta será responsable de abrir la actividad **EmailLoginActivity**, con un parámetro que indica el modo de apertura: en modo login o en modo registro. De esta forma, capturamos el clic sobre el botón **Crear cuenta nueva**, en LoginActivity, para abrir la actividad de registro.

```
@OnClick(R.id.login_register_btn) protected void doEmailRegister
(View view) {
    Intent intent = new Intent(this, EmailLoginActivity.class);
 intent.putExtra(Constants.IntentEmailLoginMode,  Constants.
EmailLoginMode.Register);
    startActivityForResult(intent, Flags.EMAIL_SIGN_IN);
}
```

Si el registro se ha completado con éxito, cuando regresamos a LoginActivity, capturamos el resultado del intent en el método **onActivityResult**; en caso de que fuera invocada desde la Activity de registro, refrescaríamos la interfaz del login para comprobar si hay una sesión activa.

```
…if (requestCode == Flags.EMAIL_LOGIN || requestCode == Flags.
EMAIL_SIGN_IN)            loginPresenter.isSignedIn();
```

Al ejecutar la aplicación y pulsar el botón **Crear cuenta nueva**, debería abrirse la Activity de registro de cuenta nueva (Figura UC5-1), de tal forma que, tras introducir los datos de email y contraseña y pulsar el botón **Crear tu cuenta**, nos registraría en Firebase, si todo ha ido bien, cerraría la Activity de registro y mostraría la Activity del login con el botón de cierre de sesión en caso de detectar una sesión abierta (Figura UC5-2).

Fig. UC5-1. Crear cuenta con email. Fig. UC5-2. Logueado con email.

Caso de uso - Login email con contraseña

Anteriormente realizamos la fase de registro de una nueva cuenta mediante email. Nos faltaba el caso de uso del login de usuario ya existente en el sistema, mediante email y contraseña. Se puede ver en el siguiente diagrama:

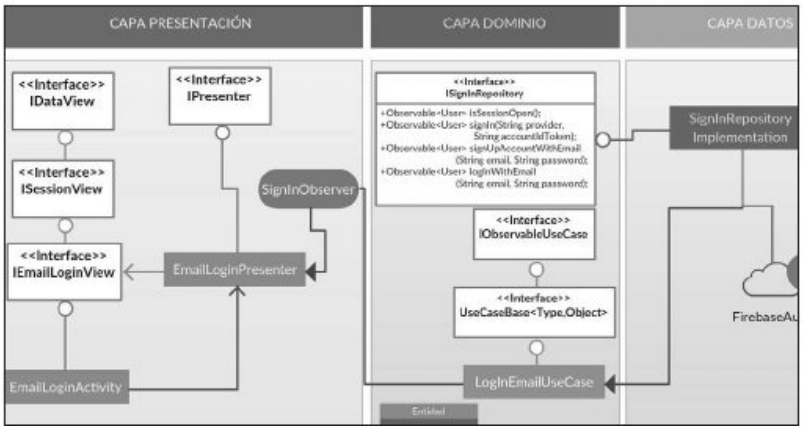


Fig. UC6D. Diagrama caso de uso login email.

En la capa de dominio, en el repositorio **ISignInRepository**, creamos un nuevo método para realizar el login con email.

```
Observable<User> logInWithEmail(String email, String password);
```

Para construir el caso de uso responsable de la acción del login mediante email, creamos una clase **LogInEmailUseCase**, que retornará el usuario conectado, en caso de éxito, y requiere dos parámetros: email y contraseña.

```
public class LogInEmailUseCase  extends UseCaseBase<User,
EmailParameters.Parameters> {…
    @Inject public LogInEmailUseCase(ISignInRepository
iSignInRepository,         Scheduler schedulerThread) {
         this.iSignInRepository = iSignInRepository;
         this.schedulerThread = schedulerThread;
}
@Override public Observable<User> implementUseCase( DisposableObserver
 observer,

EmailParameters.Parameters parameters) {
         Observable<User> observable = iSignInRepository.logInWithEmail(
                  parameters.getEmail(), parameters.getPassword());

            this.createUseCase(observable, observer, schedulerThread);
         return observable;
}
}
```

En la capa de datos, implementamos el método que acabamos de crear en el repositorio, en la clase que lo implementa, **SignInRepositoryImplementation**. En este caso, utilizaremos la instancia de autenticación de Firebase para invocar su método de login **signInWithEmailAndPassword**, pasándole el email y contraseña como parámetros. Esto retornará el usuario conectado, en caso de éxito.

```
@Override public Observable<User> logInWithEmail(String email, String
password) {
    return Observable.create(emitter -> {
         try { firebaseAuth.signInWithEmailAndPassword(email, password)
                                  .addOnCompleteListener(task -> {
                  if (task.isSuccessful())
getTokenOnSuccessFulSignIn(emitter);
                  else emitter.onError(task.getException());});
                  } catch (Exception e) { emitter.onError(e); }
});
}
```

En la capa de presentación, abrimos el **EmailLoginPresenter** y añadimos al constructor, como nuevo parámetro, el caso de uso que acabamos de crear **LogInEmailUseCase**; además de un método para realizar el login, que ejecutará el caso de uso en el que estamos trabajando, pasándole el email y la contraseña.

```
public void loginWithEmailAndPassword(String email, String password) {
    if (isValidEmail(email) && isValidPassword(password)) {
        logInEmailUseCase.implementUseCase(
                new SignInObserver(iEmailLoginView, context),
                EmailParameters.Parameters.Create(email, password));
    }
}
```

Descargado en: eybooks.com

De la misma forma, dado que el presentador ha aumentado en un parámetro el constructor, modificamos dicho constructor en el módulo **EmailLoginModule**.

```
@Provides static EmailLoginPresenter provideEmailLoginPresenter(
…
,LogInEmailUseCase logInEmailUseCase, … )
          { return new EmailLoginPresenter(…, logInEmailUseCase, …); }
```

Por último, en esta capa de presentación, abrimos la LoginActivity y capturamos el evento del botón **Iniciar sesión con su E-mail**, para abrir la actividad EmailLoginActivity, pasándole el modo login en los extras del intent.

Además, en la actividad EmailLoginActivity, tras introducir el email y contraseña válidos, capturaremos el botón **Iniciar sesión con su E-mail**, para llamar al método responsable del login por email del presentador.

```
@OnClick(R.id.emailLogin_login_btn) public void doLoginClick(View view) {
 emailLoginPresenter.loginWithEmailAndPassword(emailLogin_email.getText().
toString(),
  emailLogin_password.getText().toString());
}
```

Si ejecutamos la app, tras pulsar sobre el botón **Iniciar sesión con su E-mail**, se nos abriría la pantalla de login por email (Figura UC6-1). Tras anotar email y contraseña, pulsamos sobre **Iniciar sesión**. Si todo va bien, se validará en Firebase y obtendremos al usuario conectado (Figura UC6-2).

Fig. UC6-1. Login con email. Fig. UC6-2. Logueado con email.

Por último verificamos que los usuarios que hemos ido registrando en Firebase están registrados. Para ello vamos a la consola de desarrolladores de Firebase, a la sección **Usuarios**, y allí encontraremos nuestros usuarios con UID y cuentas de Facebook, Google e email registrados, como podemos ver en la siguiente imagen.

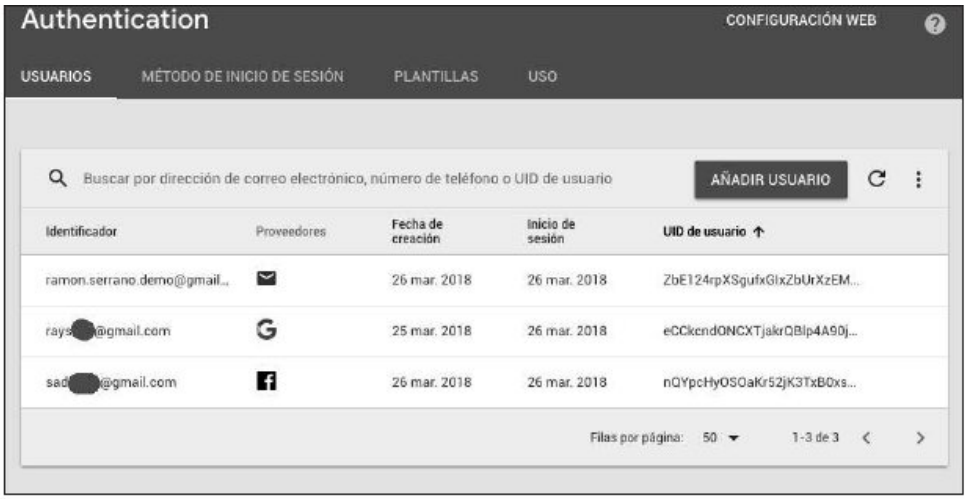


Fig. UC6-3. Login con email.

Caso de uso - Cierre de sesión

Llegados a este punto, necesitamos cerrar la sesión del usuario. Para ello elaboramos un caso de uso que cierre toda sesión abierta, independientemente del proveedor utilizado. Podemos verlo en el siguiente diagrama:

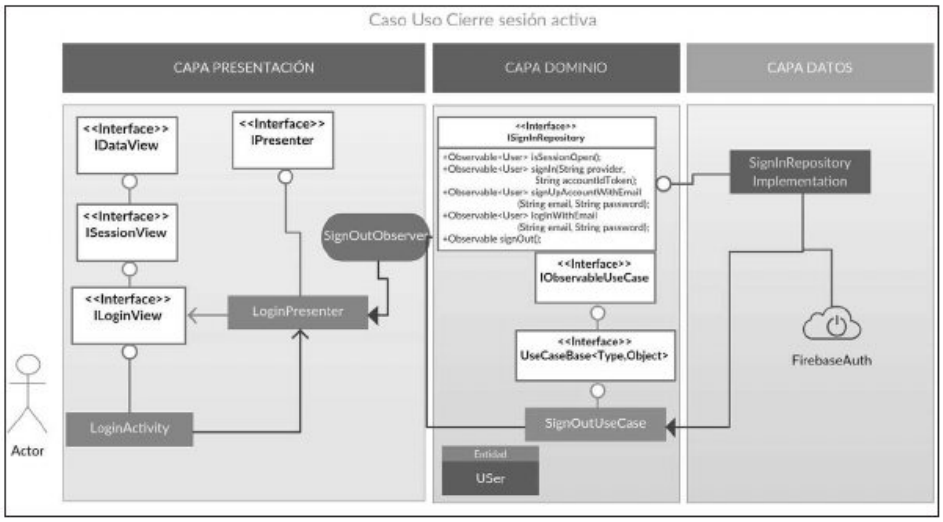


Fig. UC7D. Diagrama caso de uso cierre de sesión.

Seguimos con el proceso realizado hasta ahora; vamos a la capa del dominio y, en el repositorio **ISignInRepository**, creamos un método para el cierre de sesión.

```
Observable signOut();
```

Continuamos en la misma capa y creamos ahora el caso de uso del cierre de sesión, sin retornar ningún valor ni requerir parámetro alguno.

```
public class SignOutUseCase extends UseCaseBase<Void, Void> {
@Inject public SignOutUseCase(ISignInRepository iSignInRepository,
Scheduler schedulerThread) {
                this.iSignInRepository = iSignInRepository;
                this.schedulerThread = schedulerThread;
}
@Override public Observable implementUseCase(DisposableObserver
observer, Void parameters) {
        Observable observable = iSignInRepository.signOut();
        this.createUseCase(observable, observer, schedulerThread);
        return observable;
}
}
```

Agregamos al gradle del módulo **data**, la dependencia de Facebook para poder utilizar el LoginManager en el cierre de sesión; lo mismo ocurrirá con la librería de Google para el mismo propósito.

```
implementation 'com.facebook.android:facebook-login:[4,5)'
compile 'com.google.android.gms:play-services-auth:11.8.0'
```

En la capa de datos, en la clase que implementa el repositorio, **SignInRepositoryImplementation**, inyectamos al constructor el cliente de la API de Google, **GoogleApiClient**, para poder realizar el cierre de sesión en Google.

Además sobrescribimos el método **signOut** del repositorio, responsable del cierre de sesión. Lo hemos explicado en los comentarios, para que quede más claro el código. El más especial es el de Google, donde tenemos que desconectarnos del cliente de la API de Google.

```
@Override public Observable signOut() {
return Observable.create(emitter -> {      try {
        //Hacemos Logout de Firebase.
        firebaseAuth.signOut();
        //Hacemos Logout de Faceboook.
        LoginManager.getInstance().logOut();

        //Hacemos Logout de Google.
        googleApiClient.registerConnectionCallbacks(new GoogleApiClient
        .ConnectionCallbacks() {
        @Override public void onConnected(@Nullable Bundle bundle) {
                //Revocamos el acceso al cliente de la API de Google.
                  Auth.GoogleSignInApi.revokeAccess(googleApiClient)
                                                  .setResultCallback(
                          status -> {
                          //Desconectamos el cliente de la API.
                                    googleApiClient.disconnect();
                          //Dejamos de escuchar eventos de conexi n.
                  googleApiClient.unregisterConnectionCallbacks(this);
                                       emitter.onComplete();
                                          });
                }
          @Override public void onConnectionSuspended(int i) { }
        });
        //Reconectamos el cliente de la API de Google.
        googleApiClient.connect();
} catch (Exception e) { emitter.onError(e); }
  });
}
```

Nos vamos a la capa de presentación, donde definimos los observadores y creamos uno llamado **SignOutObserver**, aunque como retorna un Void, únicamente queremos capturar el evento de completado o de error, invocando al método **onSignedOut** en caso de que el cierre se haya realizado con éxito.

```
public class SignOutObserver extends DisposableObserver<Void> { …
@Override public void onComplete()
{       iSessionView.hideLoading();
        iSessionView.onSignedOut(); }
}
```

Dado que hemos creado un nuevo caso de uso, debemos incorporarlo en el constructor del LoginPresenter, así como en el constructor que instancia al LoginPresenter en el módulo **LoginModule**.

```
@Provides static LoginPresenter provideLoginPresenter(… ,SignOutUseCase
 signOutUseCase,…)
               { return new LoginPresenter(…, signOutUseCase,… ); }
```

Una vez definido en el constructor del LoginPresenter el **signOutUseCase**, ya podemos ejecutarlo. De modo que creamos un método en este presentador, llamado **signOut**, que ejecutará dicho caso de uso sin proporcionarle parámetros.

```
public void signOut(){ signOutUseCase.implementUseCase(new
SignOutObserver(iLoginView), null); }
```

Solo queda capturar, en la LoginActivity, el clic sobre el botón de cierre de sesión para llamar al método del presentador, responsable del cierre de sesión.

```
@OnClick(R.id.login_signOut_btn) protected void doSignOut() {
loginPresenter.signOut(); }
```

Si ejecutamos la app, en caso de tener sesión activa, como ocurría en la Figura UC6-2, el botón de cierre de sesión es visible. De modo que, al pulsar el botón, cerraremos cualquier sesión que tengamos activa, independientemente del proveedor que proporcionó la sesión, como observamos en la siguiente imagen:

Fig. UC7-1. Cierre de sesión.

Caso de uso - Configurar Firebase Realtime Database

Necesitamos una base de datos para la app. Firebase nos la proporciona mediante **Realtime Database**; lo único que necesitamos es subir nuestros datos en formato JSON.

Lo primero que tenemos que hacer es abrir la consola de Firebase en la sección **Database** del proyecto. Nos preguntará si queremos empezar con una base de datos en tiempo real (Figura UC8-1).

Fig. UC8-1. Crear Realtime Database.

Al pulsar sobre **Empezar** se generará una dirección URL donde se almacenará nuestra base de datos. Esto es lo que, en próximos capítulos, utilizaremos como BaseURL. Pues bien, la nuestra se encuentra en: https://hotelsmvp.firebaseio.com.

Nuestra base de datos necesitará los siguientes nodos:

- **hotels.** Contendrá información pública de los hoteles que hemos registrado, la clave será el identificador numérico del hotel y el valor, sus propiedades.

```
"hotels": {
"0": {
    "id":0,
    "country": "Espa a",
    "location": {"latitude": "41.234555", "longitude": "1.806757"},
    "name": "Hotel Don Juan Platja",
    "town": "Sitges",
    "web": "www.hoteldonjuanplatja.com",
    "photoUrl":"/o/hotels%2Fhotel1Mini.jpg?alt=media&token=
d930a03f-32a2-46d3-a678-b8482e41943d"
  },…
}
```

- **opinions.** Contendrá información pública de los comentarios que los usuarios han hecho. Su clave es un identificador numérico que se corresponde con el identificador del hotel, dado que pueden existir múltiples opiniones del mismo hotel. El valor será un listado de nodos cuya clave será el UID del usuario que ha realizado el comentario y su valor serán las propiedades del comentario que ha dejado, así como la valoración otorgada. Solo un usuario puede escribir en su comentario.

```
"opinions": {
"0": {
        "ymeq73ZEWAeyQbJHmn3nIqTK0kh2":{
                "rating":4,
                "message":"Ha sido increible la estancia en
este hotel...",
                "creationDate":"2018-03-12T21:30:00"
        },
        "pSB4wKvDmoVZIHNhVvZolY6stEC2":{
                "rating":3,
                "message":"La habitaci n muy c moda...",
                "creationDate":"2018-03-12T21:40:00"
        }…
},…
}
```

- **public_profile.** Contiene información pública de los nombres públicos de los usuarios. Su clave es el UID de un usuario y su valor, el nombre del usuario.

```
"public_profile":{
"ymeq73ZEWAeyQbJHmn3nIqTK0kh2":"Ram n",
"pSB4wKvDmoVZIHNhVvZolY6stEC2":"Ximo"
}
```

- **ratings.** Contiene información pública de las valoraciones de cada usuario acerca de un hotel determinado. Su clave es el identificador numérico de hotel y su valor es un listado de nodos, donde la clave es el UID del usuario que ha introducido la valoración y el valor es la puntuación de la valoración.

```
"ratings":{
"0": {
        "ymeq73ZEWAeyQbJHmn3nIqTK0kh2":4,
        "pSB4wKvDmoVZIHNhVvZolY6stEC2":3
        },…
}
```

Almacenamos toda esta información en un JSON y, de esta forma, ya tendríamos preparado el fichero para subirlo a Firebase. Para ello lo importamos desde la consola de Firebase, en la sección **Datos**, pulsando el icono de tres puntos, como observamos en la siguiente imagen:

Fig. UC8-2. Importar JSON en Firebase.

Tras importar los datos, estos se harán visibles, como se aprecia en la imagen:

Fig. UC8-3. Base de datos importada.

Para crear las reglas de acceso para esta base de datos, las definimos en el apartado **Reglas** de la sección de Realtime Database y las configuramos de esta forma:

- **public_profile.** Cualquiera puede leer; sin embargo, solo un usuario podrá editar su nombre público, ya que el UID clave del public_profile debe coincidir con el UID del usuario que accede a la información.
- **hotels.** Cualquiera puede leer, pero nadie puede escribir.
- **ratings.** Cualquiera puede leer; sin embargo, dentro de un hotel, **$id**, solo podrá escribir el usuario cuyo UID coincida con el de la clave UID del comentario.
- **opinions.** Idéntico al de ratings.

```
{“rules”:
  {
    “public_profile”:{“.read”:true,”$uid”:{“.write”: “auth.uid=== $uid”} },
    “hotels”:{“.read”:true,”.write”: “false” },
    “ratings”:{
         “.read”:true,
         “$id”:{“.read”:true,”$uid”: {“.write”: “auth.uid === $uid”}}
},
“opinions”:{
    “.read”:true,
    “$id”:{“.read”:true,”$uid”: {“.write”: “auth.uid === $uid” }
    }
   }
  }
}
```

Subimos estas reglas a Firebase y pulsamos sobre el botón **Publicar**. Si todo ha ido bien, no aparecerá ningún error y ya estaremos listos para consumir la base de datos desde la app de hoteles de Android.

Caso de uso - Listar hoteles

Una vez que tenemos definida en Firebase la base de datos con la que vamos a trabajar, es el momento de obtener un listado de hoteles de esta base de datos mediante llamadas a la API de Firebase; para ello utilizaremos **Retrofit 2**, tal como se puede observar en el siguiente diagrama:

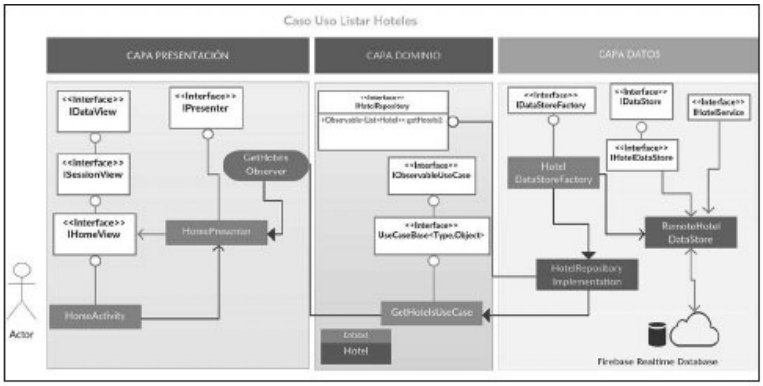


Fig. UCD9. Diagrama caso de uso listar hoteles.

Definimos un nuevo repositorio en la capa de dominio **IHotelRepository**, con un método **getHotels**, que retornará un observable de listado de hoteles.

```
public interface IHotelRepository { Observable<List<Hotel>> getHotels(); }
```

En los interactores del dominio, creamos un nuevo caso de uso para listar los hoteles **GetHotelsUseCase**. Retornará un listado de hoteles y no requerirá ningún parámetro para invocar al método del repositorio, responsable de buscar datos.

```
public class GetHotelsUseCase extends UseCaseBase<List<Hotel>, Void> {
@Inject public GetHotelsUseCase(IHotelRepository iHotelRepository,
                Scheduler schedulerThread)
{this.iHotelRepository = iHotelRepository;
this.schedulerThread = schedulerThread; }
@Override public Observable<List<Hotel>> implementUseCase
                (DisposableObserver observer, Void parameters) {
Observable<List<Hotel>> observable = iHotelRepository.getHotels();
this.createUseCase(observable, observer, schedulerThread);
return observable;
  }
}
```

En la capa de datos, definimos la interfaz **IHotelDataStore**, que debe implementar cualquier acceso a datos remota o local de los datos de los hoteles, con el método del caso de uso **getHotels**. Esto nos permitirá que, en la factoría de almacenes de datos, el acceso remoto o local sea del mismo tipo.

```
public interface IHotelDataStore extends IDataStore { Observable<List
<Hotel>> getHotels(); }
```

Para el acceso remoto a la base de datos de Firebase será necesario el uso de Retrofit 2; por ello, en el fichero gradle de la capa de datos, responsable de la comunicación, debemos importar las siguientes librerías:

- **okhttp.** Necesaria para lanzar peticiones http y recibir sus respuestas.

```
compile 'com.squareup.okhttp3:okhttp:3.7.0'
```

- **retrofit.** Necesaria para invocar las llamadas http y usar anotaciones como @ GET, @POST, @ PUT, @DELETE, entre otras.

```
compile 'com.squareup.retrofit2:retrofit:2.2.0'
```

- **converter-gson: necesaria para el mapeo entre JSON y objetos Java.**

```
compile 'com.squareup.retrofit2:converter-gson:2.2.0'
```

- **logging-interceptor.** Necesaria para mostrar una traza de cada petición.

```
compile 'com.squareup.okhttp3:logging-interceptor:3.7.0'
```

Una vez que tenemos las librerías importadas al proyecto, ya podemos hacer uso de las anotaciones de Retrofit, necesarias en la definición de los servicios. Creamos un nuevo servicio en la capa de datos, en el paquete **service**, llamado **IHotelService**. Aquí definiremos cada una de las acciones a base de datos remotas; en nuestro caso, la petición GET, para obtener todos los hoteles de la base de datos. Dentro de **@GET** incluimos la ruta que debemos invocar para obtener los datos. Esta ruta se compone de una URL base, que es la URL donde se encuentra nuestra base de datos o API de Firebase, es decir, https://hotels-mvp.firebaseio.com, seguido del controlador o nodo que queremos obtener, en este caso, "hotels", y como queremos en formato JSON la respuesta, le añadimos ".json".

Normalmente, las peticiones a web API no suelen incluir ".json"; no obstante, en Firebase, las peticiones lo requieren.

```
public interface IHotelService    { @GET("hotels.json") Call<List
<Hotel>> getHotels(); }
```

Debido a que cada vez que vayamos a utilizar retrofit tendremos que inicializar el cliente http, hemos decidido crear una clase de utilidades en la capa de datos, llamada **Utilities**, con un método que retorna inicializado el cliente http.

En él, incluimos un interceptor de trazas de cada petición http que realicemos, llamado **HttpLoggingInterceptor**, e indicamos mediante **Level.BODY** que queremos capturar trazas tanto de la petición como de la respuesta. Además asignamos un tiempo de espera aceptable, antes de provocar un timeout, de 30 segundos.

```
public static OkHttpClient.Builder initializeHttpClient(Context
context) {
HttpLoggingInterceptor logging = new HttpLoggingInterceptor();
logging.setLevel(HttpLoggingInterceptor.Level.BODY);
```

```
        OkHttpClient.Builder httpClient = new OkHttpClient.Builder();
         httpClient.connectTimeout(30, TimeUnit.SECONDS);
         httpClient.readTimeout(30, TimeUnit.SECONDS);
         httpClient.addInterceptor(logging);
         return httpClient;
}
```

La clase responsable de acceso a datos remotos es **RemoteHotelDataStore**, que definimos en el paquete **data/remote**. Esta implementa la interfaz de almacenamiento de datos de hoteles IHotelDataStore.

En el constructor de la clase, preparamos Retrofit indicándole la URL base donde se encuentra la API de Firebase mediante **baseUrl**, además de añadirle una factoría de conversión de datos JSON. Seguidamente, instanciamos el cliente http de las peticiones mediante **client** y la utilidad que anteriormente definimos.

Por último hacemos que Retrofit implemente las llamadas definidas por el servicio IHotelService, para poder ser invocados.

```
public class RemoteHotelDataStore implements IHotelDataStore {…
 public RemoteHotelDataStore(final Context context) {
         this.context = context;
         retrofit = new Retrofit.Builder()
                 .baseUrl(context.getResources().getString(R.string.
apiBaseURL))
          .addConverterFactory(GsonConverterFactory.create())
          .client(Utilities.initializeHttpClient(context).build()).
build();
        iHotelService = retrofit.create(IHotelService.class);
    }
…
```

Implementamos el método del almacén de datos, es decir, getHotels. Será aquí donde invoquemos la llamada al servicio responsable de obtener los hoteles y obtener una llamada **Call**. Necesitamos ejecutar esta llamada call mediante el método **execute**, el cual nos retornará una respuesta con el tipo de datos esperado, es decir, una **Response<List<Hotel>>**.

De esta respuesta podemos obtener el código de respuesta http. Si ha ido todo bien, será http 200 y, por tanto, en el **body** de esta respuesta encontraremos el listado de hoteles.

Por cada hotel obtenido tenemos una propiedad que es la ruta donde se encuentra la imagen del hotel. En nuestro caso las tenemos almacenadas en el Storage de ficheros de Firebase, por lo que al ser su URL base conocida, hemos pensado añadir a cada photoUrl del hotel la URL base donde se encuentra el repositorio de ficheros de Firebase; de este modo, ahorramos datos móviles.

```
…
    @Override public Observable<List<Hotel>> getHotels() {
        return Observable.create(emitter -> {
            try {
                Call<List<Hotel>> call = iHotelService.getHotels();
                Response<List<Hotel>> response = call.execute();
                if (response.code() == 200) {
                    List<Hotel> hotels = response.body();
                    List<Hotel> hotelsResponse = new ArrayList<>();
                    for (Hotel hotel : hotels) {
                        if (hotel != null) {
```

```
                            Hotel hotelRes = hotel;
                            hotelRes.setPhotoUrl(context.getString(R.
string.dataStorageBaseURL) + hotel.getPhotoUrl());
                             hotelsResponse.add(hotelRes);
                        }
                    }
                    emitter.onNext(hotelsResponse);
                    emitter.onComplete();
                          }else emitter.onError(
                                  new Exception( response.errorBody().
toString()));
            } catch (Exception e) { emitter.onError(e); }
        });
    }
}
```

Siguiendo con la capa de datos, definimos la factoría de almacenes de datos de los hoteles, en el paquete **factory**, con el nombre **HotelDataStoreFactory**, donde, dependiendo del tipo de acceso requerido, retornaremos acceso remoto o local. Para los hoteles será remoto por el momento.

```
@Singleton public class HotelDataStoreFactory implements IData
StoreFactory {
        @Inject public HotelDataStoreFactory(Context context)
                { this.context = context; }
@Override public IHotelDataStore Remote()
       { return new RemoteHotelDataStore(context); }
@Override public IHotelDataStore Local() { return null; }
}
```

Terminamos en esta capa implementando la clase que implementa al repositorio de hoteles, **HotelRepositoryImplementation**. Este utilizará la factoría de almacenes de datos de hoteles para invocar al método remoto de obtener hoteles.

```
@Singleton public class HotelRepositoryImplementation implements
IHotelRepository {
                @Inject public HotelRepositoryImplementation(
         Context context, HotelDataStoreFactory hotelDataStoreFactory)
         {        this.context = context;
                  this.hotelDataStoreFactory = hotelDataStoreFactory;
         }
@Override public Observable<List<Hotel>> getHotels()
         { return hotelDataStoreFactory.Remote().getHotels(); }
}
```

Pasamos a la capa de presentación y proveemos la dependencia de la clase que implementa al repositorio de hoteles, en el módulo **AppModule**.

```
@Provides @Singleton static IhotelRepository providesHotelRepository(
HotelRepositoryImplementation hotelRepositoryImplementation)
                    { return hotelRepositoryImplementation; }
```

Creamos la actividad vacía **HomeActivity** en el paquete **ui**, marcándola como actividad principal en el manifest. Su layout se compone de un **CoordinatorLayout**, que encapsula una **Toolbar** con el logo de la app y el icono del **DrawerLayout**.

Además, disponemos de un DrawerLayout que contiene un **SwipeRefreshLayout**, capaz de refrescar su **RecyclerView** hijo, responsable de la representación de los datos de los hoteles.

Por último, nuestro DrawerLayout contiene un **NavigationView**, representado por un menú de tres opciones: "Iniciar sesión", "Ir al perfil" y "Cerrar sesión".

Creamos la vista **IHomeView** de esta actividad con un método que retorna el listado de hoteles.

```
public interface IHomeView extends ISessionView { void renderData
(List<Hotel> value); }
```

Creamos un observador **GetHotelsObserver** que, en caso de éxito de la petición, retornará el listado de hoteles por el método **renderData** de la interfaz de la vista **IHomeView**.

Una vez definida la vista, le llega el turno al presentador **HomePresenter** de la actividad, el cual encapsulará en su constructor, la vista, y el caso de uso **GetHotelsUseCase**. Su método **initialize** comprueba primero si existe alguna sesión activa y posteriormente busca los hoteles, ejecutando el caso de uso responsable de esta tarea sin pasarle ningún parámetro.

```
public class HomePresenter implements IPresenter {
@Inject public HomePresenter(IHomeView iHomeView, GetHotelsUseCase
getHotelsUseCase
                , CheckIsUserSignedInUseCase checkIsUserSignedInUseCase) {
        this.iHomeView = iHomeView;
        this.getHotelsUseCase = getHotelsUseCase;
        this.checkIsUserSignedInUseCase = checkIsUserSignedInUseCase;
}
public void initialize() {        isSignedIn();    getHotels(); }
public void isSignedIn() { checkIsUserSignedInUseCase.implementUseCase(
                                                    new
SignedInObserver(iHomeView), null); }
public void getHotels() { getHotelsUseCase.implementUseCase(
                                                          new
GetHotelsObserver(iHomeView), null);}
}
```

Como siempre, definimos un módulo **HomeModule** proveyendo de la vista y del presentador.

```
@Module public abstract class HomeModule {
@Provides static HomePresenter provideHomePresenter(IHomeView iHomeView,
             GetHotelsUseCase getHotelsUseCase ,
CheckIsUserSignedInUseCase checkIsUserSignedInUseCase)
        {return new HomePresenter(iHomeView, getHotelsUseCase,
checkIsUserSignedInUseCase); }
@Binds abstract IHomeView provideHomeView(HomeActivity homeActivity);
}
```

Volviendo a la Activity, hacemos que implemente la vista **IHomeView**, inyectamos el presentador, instanciamos Butterknife e inicializamos el presentador para que vaya a buscar los hoteles.

Una vez inicializado el presentador, lo primero que hará este será ir a buscar los hoteles. Cuando obtengamos una respuesta con los hoteles listados, a través

del método sobrescrito en la HomeActivity, **renderData**, debemos configurar un adaptador para representar los datos en el RecyclerView.

Para representar cada hotel necesitaremos diseñar un layout que contenga un CardView con la imagen del hotel, un título, un subtítulo con la población del hotel y, por último, una barra de valoraciones en forma de estrellas.

Tras crear este layout, pasaremos a crear una clase que represente las vistas de este layout, para el adaptador del RecyclerView, y crearemos en el paquete **adapter** la clase **HotelsViewHolder**, que inyectará las vistas en el RecyclerView.

```
public class HotelsViewHolder extends RecyclerView.ViewHolder {
@BindView(R.id.card_view)                 CardView card_view;
@BindView(R.id.card_image)                ImageView card_image;
@BindView(R.id.card_title)                TextView card_title;
@BindView(R.id.card_location)     TextView card_location;
@BindView(R.id.card_rating)               RatingBar card_rating;
public HotelsViewHolder(View itemView)
{        super(itemView);
         ButterKnife.bind(this, itemView); }
}
```

Nuestra vista va a necesitar cargar imágenes desde una URL; por esta razón, vamos a importar la librería Picasso, desarrollada por el equipo Square, en el gradle del módulo de presentación, ideal para cargar imágenes desde URL, cacheándolas y liberándolas de memoria cuando no las representa.

```
compile 'com.squareup.picasso:picasso:2.5.2'
```

Una vez que tenemos inyector de vistas del recycler, creamos el adaptador, que representará cada hotel mediante este manejador de vistas para cada elemento de una colección de hoteles. Creamos este adaptador en el mismo paquete **ui/adapter** con el nombre **HotelsListAdapter**. Como podemos observar en el código, utilizamos el layout **card_view_hotel** para representar cada hotel, de tal forma que, cuando representemos cada elemento mediante **onBindViewHolder**, cargaremos en las respectivas vistas la información requerida: nombre de hotel, nombre de población y valoración.

Además emplearemos la librería de Picasso para cargar la vista de la imagen del hotel, sabiendo su URL, de tal forma que será Picasso el que cargue esta información en la vista cuando la tenga lista.

```
public class HotelsListAdapter extends RecyclerView.
Adapter<HotelsViewHolder>{…
public HotelsListAdapter(List<Hotel> hotelList, Context context)
{ this.items = hotelList;          this.mContext=context; }

@Override public HotelsViewHolder onCreateViewHolder(ViewGroup parent,
int viewType)
{        View v = LayoutInflater.from(parent.getContext()).inflate
(R.layout.card_view_hotel, parent, false);
         return new HotelsViewHolder(v);
}
@Override public void onBindViewHolder(HotelsViewHolder holder,
int position) {
         final Hotel hotel = this.items.get(position);
                  holder.card_title.setText(hotel.getName());
```

```
                holder.card_rating.setRating(hotel.getRating().
floatValue());
                holder.card_location.setText(new StringBuilder()
                                .append(hotel.getTown()).append(" (")
                                .append(hotel.getCountry()).append
(")"));
                Picasso.with(mContext).load(hotel.getPhotoUrl()).
into(holder.card_image);
}
public Hotel getItemAtPosition(int position) { return items.get
(position); }
@Override public int getItemCount() { return items.size(); }
}
```

Una vez que hemos creado el adaptador, debemos alimentarlo con los datos obtenidos por el presentador en la búsqueda de hoteles. Instanciamos el adaptador con los datos obtenidos, establecemos que el RecyclerView se representará verticalmente mediante un **LinearLayoutManager** y, por último, asociamos el adaptador al RecyclerView.

```
@Override public void renderData(List<Hotel> data) {
      hotelsListAdapter = new HotelsListAdapter(data, this);
  LinearLayoutManager linearLayoutManager = new LinearLayoutManager(this,

LinearLayoutManager.VERTICAL, false);
      home_hotels_recycler.setLayoutManager(linearLayoutManager);
  home_hotels_recycler.setAdapter(hotelsListAdapter);
…
```

Si necesitamos refrescar la información, utilizaremos el SwipeRefreshLayout, donde invocaremos al método de refresco del presentador **refreshItems**, que a su vez realizará la misma acción que la del método **getHotels** del presentador.

```
…

    home_swipeRefreshLayout.setOnRefreshListener(()-> homePresenter.
refreshItems());
}
```

Si ejecutamos la app, debería cargarse el RecyclerView con el listado de hoteles obtenido desde nuestra base de datos de Firebase, como se aprecia en la imagen:

Fig. UC9-1. Cargar hoteles en RecyclerView.

Caso de uso - Obtener valoración de hoteles

En la base de datos creamos un nodo llamado **ratings** que contenía la puntuación que un listado de usuarios habían asignado a un hotel. Este es el nodo que queremos consumir para representar la valoración media obtenida por los usuarios en cada hotel del RecyclerView. Observemos su diagrama en la siguiente imagen:

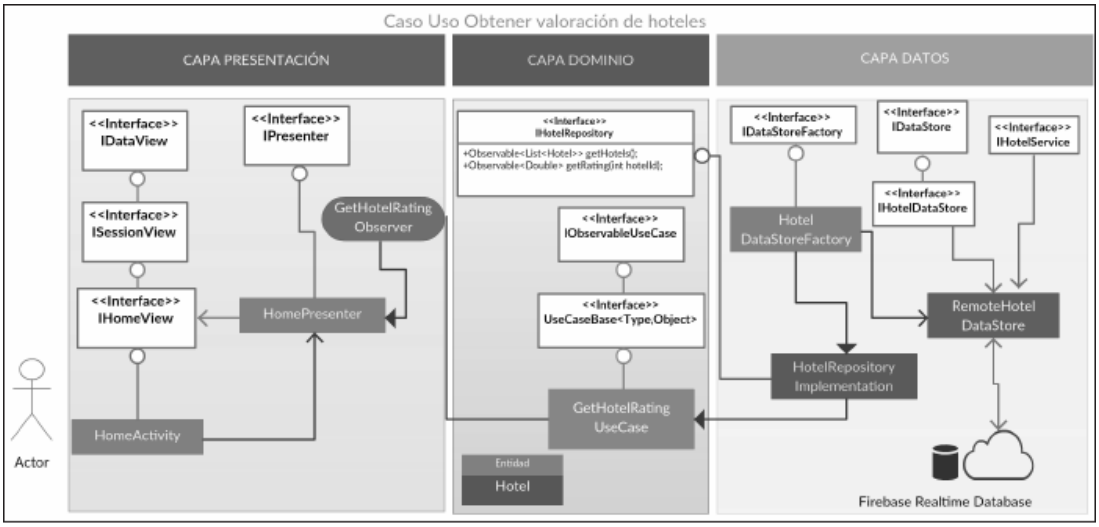


Fig. UCD10. Diagrama del caso de uso.

En el repositorio del hotel, **IHotelRepository**, definimos un nuevo método para obtener valoraciones en el dominio. Retornará un observable con una puntuación media para un hotel determinado.

```
Observable<Double> getRating(int hotelId);
```

Nuestro caso de uso requiere que le pasemos un parámetro que contenga un identificador de hotel, por ello creamos una clase, en el paquete **interactor/parameters**, para gestionar este tipo de parámetros.

```
public class HotelParameters {
public static final class Parameters {
        private int hotelId;
        public static HotelParameters.Parameters Create(int hotelId) {
                return new HotelParameters.Parameters(hotelId);
        }
}
}
```

Con estos parámetros ya podemos crear el caso de uso GetHotelRatingUseCase, que retornará un valor decimal y requiere, como parámetro, el identificador de hotel para filtrar por hotel y obtener su puntuación media.

```
public class GetHotelRatingUseCase extends UseCaseBase<Double,
HotelParameters.Parameters> {
 @Inject public GetHotelRatingUseCase(IHotelRepository iHotelRepository,
Scheduler schedulerThread)
```

```
                {       this.iHotelRepository = iHotelRepository;
                        this.schedulerThread = schedulerThread;    }
@Override public Observable<Double> implementUseCase( DisposableObserver
 observer,
HotelParameters.Parameters parameters) {
                Observable<Double> observable = iHotelRepository.
getRating(parameters.getHotelId());
                this.createUseCase(observable, observer, schedulerThread);
                return observable;        }
}
```

En la interfaz IHotelDataStore de la capa de datos, añadimos el método que se va a implementar por el acceso remoto a datos del hotel, para obtener valoraciones.

```
Observable<Double> getRating(int hotelId);
```

En la clase **HotelRepositoryImplementation**, sobrescribimos el método del repositorio para la obtención remota de valoraciones por hotel.

```
@Override public Observable<Double> getRating(int hotelId)
         { return hotelDataStoreFactory.Remote().getRating(hotelId); }
```

Definimos la llamada de obtener las valoraciones, basándonos en un identificador de hotel, en el servicio IHotelService. Como observamos, invocamos al nodo ratings/"identificador de hotel". Esto nos retornará un JSON con las puntuaciones que cada usuario haya realizado sobre el hotel en cuestión.

```
@GET("ratings/{id}.json")
Call<Map<String, Integer>> getRating(@Path(value = "id",
encoded = true) int hotelId);
```

En la clase **RemoteHotelDataStore**, implementamos el método responsable de obtener remotamente las valoraciones de Firebase. Para ello, creamos la llamada Call del servicio y lo ejecutamos con execute; de tal forma que, en caso de recibir respuesta http 200, calcularemos la media de valoraciones obtenidas y retornaremos un valor numérico como valor promedio de la puntuación del hotel.

```
@Override public Observable<Double> getRating(int hotelId) {
return Observable.create(emitter -> {
try {    Call call = iHotelService.getRating(hotelId);
                Response<Map<String, Integer>> response =
call.execute();
       double ratingValue = 0;
                if (response.code() == 200) {
                Map<String, Integer> ratings = response.body();
                        if (ratings != null) {
                                for (String key : ratings.keySet())
                                        ratingValue += ratings.
get(key).intValue();
                                ratingValue = ratingValue / ratings.
size();
                }
                        emitter.onNext(ratingValue);
                        emitter.onComplete();
```

```
                } else emitter.onError(
                                new Exception(response.errorBody().
toString()));
        } catch (Exception e) { emitter.onError(e); }
});
}
```

En la capa de presentación, en la vista **IHomeView**, añadimos el método **renderRating** para mostrar la puntuación de un hotel por la interfaz de usuario.

```
void renderRating(Double value, int hotelId);
```

Creamos un observador llamado **GetHotelRatingObserver**, el cual recibirá en caso de éxito, un valor numérico de tipo Double, con la puntuación media obtenida de un hotel. En este caso invocamos al método renderRating de la interfaz de la vista IHomeView, al que le pasamos la puntuación y el identificador de hotel.

Inyectamos el caso de uso **GetHotelRatingUseCase** al constructor del presentador **HomePresenter** y definimos el método responsable de obtener las puntuaciones de un hotel, que ejecutará el caso de uso en cuestión y le proporcionará el identificador de hotel por el que filtrar.

```
public void getRating(int hotelId) {
getHotelRatingUseCase.implementUseCase(new GetHotelRatingObserver(
                iHomeView, hotelId), HotelParameters.Parameters.
Create(hotelId));
}
```

Tras añadir este caso de uso al presentador, lo añadiremos también al módulo **HomeModule** para construir bien la instancia del presentador.

```
@Provides static HomePresenter provideHomePresenter(…,GetHotel
RatingUseCase getHotelRatingUseCase, …)              { return new
HomePresenter(… , getHotelRatingUseCase,… ); }
```

Finalmente, en la actividad HomeActivity, en el método de obtención de los hoteles **renderData**, recorremos cada uno de los hoteles recibidos, de tal forma que, por cada hotel, lanzamos una llamada para obtener sus puntuaciones.

```
@Override public void renderData(List<Hotel> data) {
for (Hotel hotel : data) homePresenter.getRating(hotel.getId());
…
```

Por fin, recibimos las respuestas de las valoraciones a través del método **renderRating**, definido en la interfaz de la vista. Tras recibir la valoración, refrescaremos el adaptador del RecyclerView para actualizar la información.

```
@Override public void renderRating(Double value, int hotelId) {
    hotelsListAdapter.getItemAtPosition(hotelId).setRating(value);
    hotelsListAdapter.notifyDataSetChanged();
}
```

Si ejecutamos la app, se cargarán los hoteles (Figura UC10-1) y posteriormente, por cada hotel, se irán actualizando las valoraciones medias de cada hotel (Figura UC10-2).

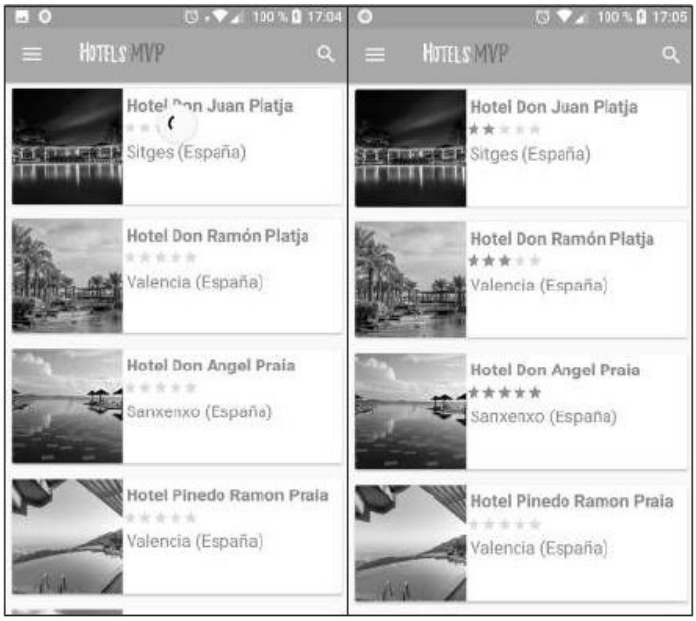


Fig. UC10-1. Cargar valoraciones. Fig. UC10-2. Valoraciones cargadas.

Caso de uso - Cargar imagen y perfil de usuario

Hemos querido dotar a la app de una zona privada, únicamente accesible para el usuario que se encuentre con una sesión activa. En esta sección, el usuario podrá visualizar información relativa a su perfil de usuario, como nombre, email y proveedor con el que se ha conectado, así como la imagen de su perfil cargada. Podemos ver el diagrama del caso de uso en la siguiente imagen:

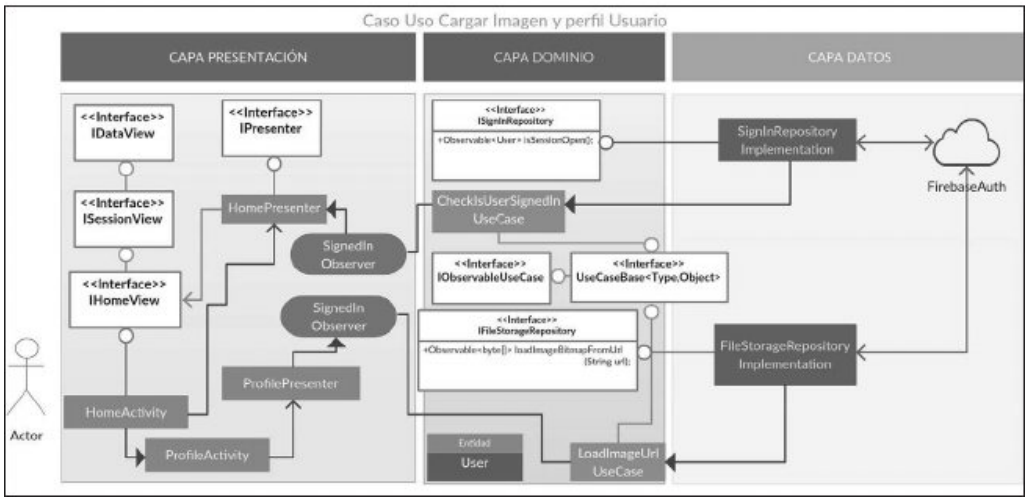


Fig. UCD11. Diagrama caso de uso carga de imagen del perfil.

En el dominio del proyecto, definimos el repositorio **IFileStorageRepository**, que debe implementar la capa de datos. En él, declaramos un método de tipo observable, que retorna un array de bytes con el contenido de la imagen que hemos ido a buscar, de acuerdo con el parámetro URL proporcionado.

```
public interface IFileStorageRepository { Observable<byte[]>
loadImageBitmapFromUrl(String url); }
```

Para la carga de la imagen del perfil de usuario, hemos definido un caso de uso, como siempre en los interactores, con el nombre **LoadImageUrlUseCase**, el cual invocará al método que hemos definido anteriormente en el repositorio. Este caso de uso retornará un array de bytes y, como parámetro, un String con la dirección donde se encuentra la imagen buscada.

```
public class LoadImageUrlUseCase extends UseCaseBase<byte[], String> {
@Inject public LoadImageUrlUseCase(IFileStorageRepository
iFileStorageRepository,

Scheduler schedulerThread)
{        this.iFileStorageRepository = iFileStorageRepository;
                                    this.schedulerThread =
schedulerThread;
                           }
@Override public Observable<byte[]> implementUseCase(DisposableObserver
observer, String url)
{           Observable<byte[]> observable = iFileStorageRepository.
loadImageBitmapFromUrl(url);
        this.createUseCase(observable, observer, schedulerThread);
        return observable;
    }
}
```

Pasamos a la capa de datos, donde definimos la clase **FileStorageRepositoryImplementation**, que implementa la interfaz del repositorio que acabamos de definir.

Sobrescribimos el método **loadImageBitmapFromUrl**, que retornará los datos de la imagen buscada en un array de bytes. Para ello abrimos una conexión con la URL conocida mediante un InputStream y openStream.

ByteArrayOutputStream nos permitirá almacenar tantos bytes como vayamos leyendo por cada iteración, en bloques de 1.024 bytes, retornando el array de bytes de la imagen obtenida por URL, que hemos reunido en el byteBuffer, tras convertirlo a array mediante toByteArray.

```
@Singleton public class FileStorageRepositoryImplementation implements
IFileStorageRepository {
@Inject public FileStorageRepositoryImplementation() { }

@Override public Observable<byte[]> loadImageBitmapFromUrl(String url)
{        return Observable.create(emitter -> {
                InputStream inputStream = new java.net.URL(url).
openStream();
                ByteArrayOutputStream byteBuffer = new
ByteArrayOutputStream();
                int bufferSize = 1024;
                byte[] buffer = new byte[bufferSize];
                int len = 0;
```

```
                while ((len = inputStream.read(buffer)) != -1)
                        byteBuffer.write(buffer, 0, len);
                emitter.onNext(byteBuffer.toByteArray());
                emitter.onComplete();
        });
    }
}
```

Debemos proveer de la dependencia la clase que implementa el repositorio que hemos definido. Esto lo realizaremos como hasta ahora, declarándolo en el módulo **AppModule**.

```
@Provides @Singleton static IFileStorageRepository
providesFileStorageRepository(
 FileStorageRepositoryImplementation fileStorageRepositoryImplementation)
                                        { return
fileStorageRepositoryImplementation; }
```

Ya estamos listos para centrarnos en todo lo relacionado con la actividad que crearemos, pero antes definamos la vista **IProfileView** con un método preparado para retornar el array de bytes de la imagen obtenida, así como el recurso ImageView, sobre el que cargaremos los datos de la imagen.

```
public interface IProfileView extends ISessionView { void photo
Loaded(byte[] value, int resourceId); }
```

Todo caso de uso tiene su observador, de modo que definiremos el observador **LoadImageObserver**, que recibirá el array de bytes en caso de éxito, así como el recurso donde debe representarlos. Se pasará esta información a la vista mediante el método photoLoaded.

```
public class LoadImageObserver extends DisposableObserver<byte[]> {
 public LoadImageObserver(IProfileView iProfileView, int resourceId)
{       this.iProfileView = iProfileView;
        this.resourceId = resourceId; }
@Override public void onNext(byte[] data) {
        iProfileView.photoLoaded(data, resourceId);
}…
}
```

El perfil requerirá una serie de casos de usos que ya hemos visto, como son CheckIsUserSignedInUseCase, para obtener el usuario conectado y por tanto sus datos, una vez que sepamos la URL de la imagen del usuario, la busquemos con el caso de uso LoadImageUrlUseCase y, por último, el caso de uso de cierre de sesión. Todos estos serán declarados en el constructor del presentador **ProfilePresenter**.

```
public class ProfilePresenter implements IPresenter {…
@Inject public ProfilePresenter(IProfileView iProfileView
                        , CheckIsUserSignedInUseCase
checkIsUserSignedInUseCase
                , LoadImageUrlUseCase loadImageUrlUseCase
                , SignOutUseCase signOutUseCase)
{       this.iProfileView = iProfileView;
        this.checkIsUserSignedInUseCase = checkIsUserSignedInUseCase;
```

Descargado en: eybooks.com

```
        this.loadImageUrlUseCase = loadImageUrlUseCase;
        this.signOutUseCase = signOutUseCase;
}
…
Comprobamos si hay una sesi n activa mediante isSignedIn.
…
 public void isSignedIn(){ checkIsUserSignedInUseCase.
implementUseCase(
                                    new SignedInObserver(iProfileView),
null); }
…
```

Buscamos los bytes de la URL de la imagen a través del método **loadImageUrl**, donde ejecutaremos el caso de uso que acabamos de crear, proporcionándole la URL de la imagen.

```
…
public void loadImageUrl(String url, int resourceId)
{       loadImageUrlUseCase.implementUseCase(new
LoadImageObserver(iProfileView, resourceId), url); }
…
}
```

Creamos la actividad **ProfileActivity** con un layout compuesto de varios CardView, uno para la imagen del perfil del usuario en un ImageView, otros tres para el tipo de conexión (Facebook, Google o email) y, por último, dos TextView, uno para el nombre y otro para el email. Además constará de un botón para cerrar la sesión activa.

Tras tener la actividad definida, la vista y el presentador, llega el turno de definir el módulo del perfil **ProfileModule**, donde proveeremos del presentador y la vista.

```
@Module public abstract class ProfileModule {
@Provides static ProfilePresenter provideProfilePresenter(IProfileView
iProfileView
        , CheckIsUserSignedInUseCase       checkIsUserSignedInUseCase
        , LoadImageUrlUseCase loadImageUrlUseCase
        , SignOutUseCase signOutUseCase)
{ return new ProfilePresenter(iProfileView, checkIsUserSignedInUseCase,
 loadImageUrlUseCase,
                        signOutUseCase); }
@Binds abstract IProfileView provideProfileView(     ProfileActivity
profileActivity);
}
```

Añadimos, a la clase **BuildersModule**, la actividad que acabamos de crear **ProfileActivity**.

```
@PerActivity @ContributesAndroidInjector(modules = ProfileModule.class)
abstract ProfileActivity contributeProfileActivity();
```

Posteriormente, abrimos la actividad ProfileActivity, hacemos que implemente la interfaz de la vista **IProfileView** y, por tanto, sobrescribimos sus métodos, y creamos una variable global **currentUser** para representar al usuario actual conectado.

Finalmente inyectamos el presentador **ProfilePresenter** e invocamos, en el **onCreate**, al método **isSignedIn** del presentador para obtener información del usuario conectado, que reuniremos en el método **updateUI**.

Tras recibir la información del usuario conectado, lo asignamos a la variable global y rellenamos las vistas del layout de la actividad. Además, conocida la URL de la imagen del usuario, invocamos al presentador para que obtenga los bytes de la imagen.

```
@Override public void updateUI(User user)  {        currentUser = user;
        profile_email.setText(user.getEmail());
        profilePresenter.loadImageUrl(user.getPhotoUrl(), profile_image.
getId());
…
}
```

Una vez que hemos recibido los bytes de la imagen, utilizamos la factoría BitmapFactory para convertir los bytes a Bitmap. Con este bitmap ya podemos representar la imagen en el ImageView del recurso solicitado.

```
@Override public void photoLoaded(byte[] data, int resourceId) {
        Bitmap bitmap = BitmapFactory.decodeByteArray(data, 0,
data.length);
((ImageView) findViewById(resourceId)).setImageBitmap(bitmap);
}
```

Para acceder a esta área privada, lo haremos a través de la HomeActivity. Recordemos que lo primero que hacemos en la HomeActivity del listado de hoteles, aparte de obtener estos, es comprobar si algún usuario tenía sesión activa. Pues bien, tras recibir la información de sesión, por **onSignedIn** de la actividad HomeActivity, mostramos el ítem de **Bienvenido** y el de **Cierre de sesión** del menú del NavigationView.

```
@Override public void onSignedIn() {
    Menu menu = home_navigation_view.getMenu();…
    MenuItem myActionMenuItemProfile = menu.findItem(R.id.nav_profile);
    myActionMenuItemProfile.setVisible(true);
}
```

De modo que, una vez pulsada la opción del menú, en caso de no tener sesión, solo podremos ir al login. En caso de tener sesión y pulsar sobre **Bienvenido** o **Cerrar sesión**, abrirá la actividad del perfil de usuario o zona privada.

```
@Override public boolean onNavigationItemSelected(@NonNull MenuItem
item) {
  switch (item.getItemId()) {
case R.id.nav_login:
                Intent intentLogin = new Intent(this, LoginActivity.
class);
        startActivityForResult(intentLogin, Flags.RequestLogin);
break;
case R.id.nav_profile:
case R.id.nav_session:
                Intent intentProfile = new Intent(this, ProfileActivity.
class);
        startActivityForResult(intentProfile, Flags.RequestProfile)
```

```
;break;
 }…
}
```

Si ejecutamos la app, se abrirá el listado de hoteles y se cargarán las opciones del NavigationView (Figura UC11-1). Al pulsar sobre la opción **Bienvenido**, abriremos la actividad del perfil del usuario y se cargarán la imagen del usuario y el tipo de proveedor utilizado (Figura UC11-2).

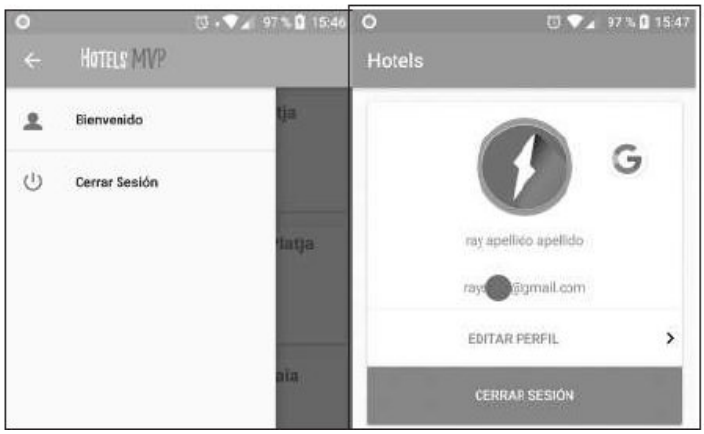


Fig. UC11-1. Abrir perfil en zona privada. **Fig. UC11-2.** Cargar imagen del perfil.

Caso de uso - Configurar Firebase Cloud Storage

Una vez más, Firebase nos proporciona un sistema de almacenamiento remoto en la nube, que posibilita la subida y descarga de archivos en el servidor. Este servicio se llama **Firebase Cloud Storage**. Nosotros lo emplearemos principalmente para almacenar las imágenes de cada uno de los hoteles, así como en la subida de imágenes del perfil del usuario.

La subida de imágenes se puede realizar manualmente a través de la consola de desarrolladores de Firebase o mediante los servicios de Firebase en nuestra aplicación Android. Pues bien, cada vez que realicemos una subida al repositorio que Firebase ha generado tras crear nuestro proyecto, obtendremos una URL donde estos recursos serán accesibles.

Podemos proteger estos ficheros mediante reglas, de igual forma que ocurría en el caso de los datos de los nodos de la base de datos.

De este modo, tras conocer la dirección base del repositorio de los ficheros **gs://hotelsmvp.appspot.com/**, creamos dos directorios que descienden del raíz, uno llamado **hotels**, donde almacenamos las imágenes de cada hotel, y otro llamado **profiles**, donde almacenamos las imágenes que subamos del perfil de cada usuario, como puede verse en la siguiente imagen:

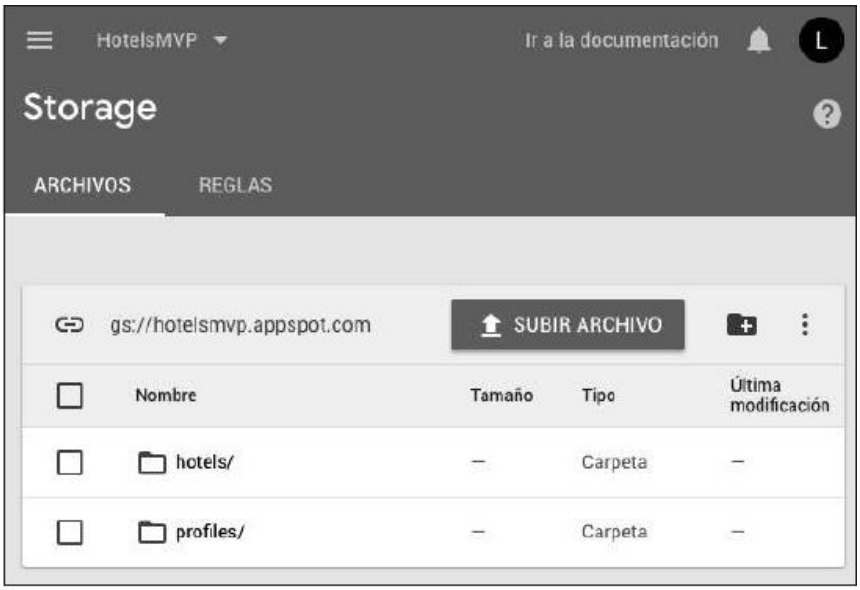


Fig. UC12-1. Firebase Cloud Storage.

Tras subir un fichero, podemos ver propiedades como nombre, tamaño, tipo, fecha de subida, así como la ubicación donde se encuentra y la URL de descarga de la imagen, que puede revocarse manualmente. Podemos observarlo en la siguiente imagen:

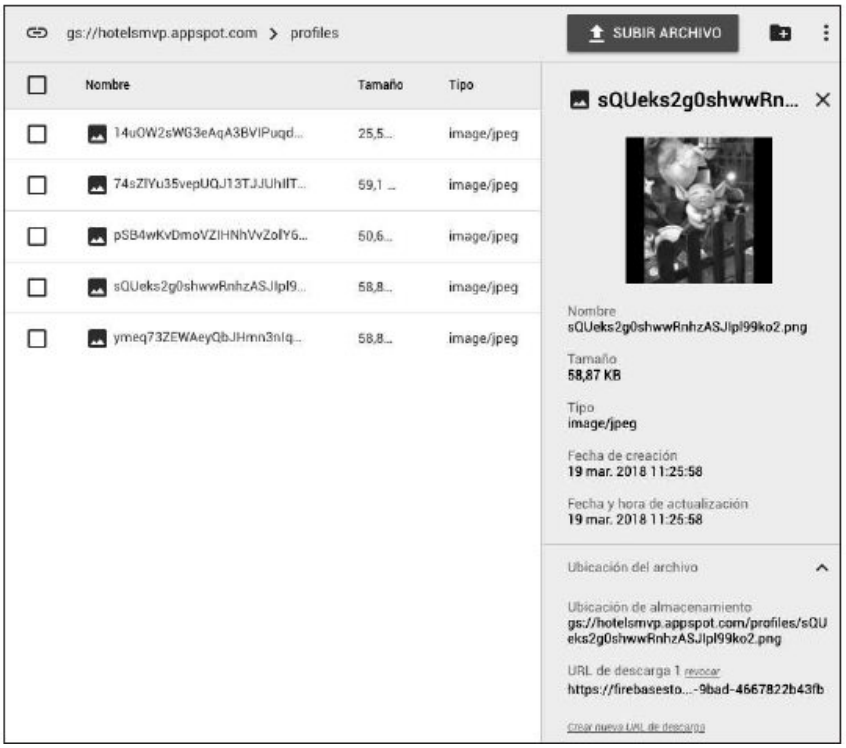


Fig. UC12-2. Firebase Cloud Storage Detalle.

Caso de uso - Subir imagen del perfil a Firebase Cloud Storage

Para la subida de imágenes emplearemos Firebase Cloud Storage. En el siguiente diagrama tenemos representado este caso de uso.

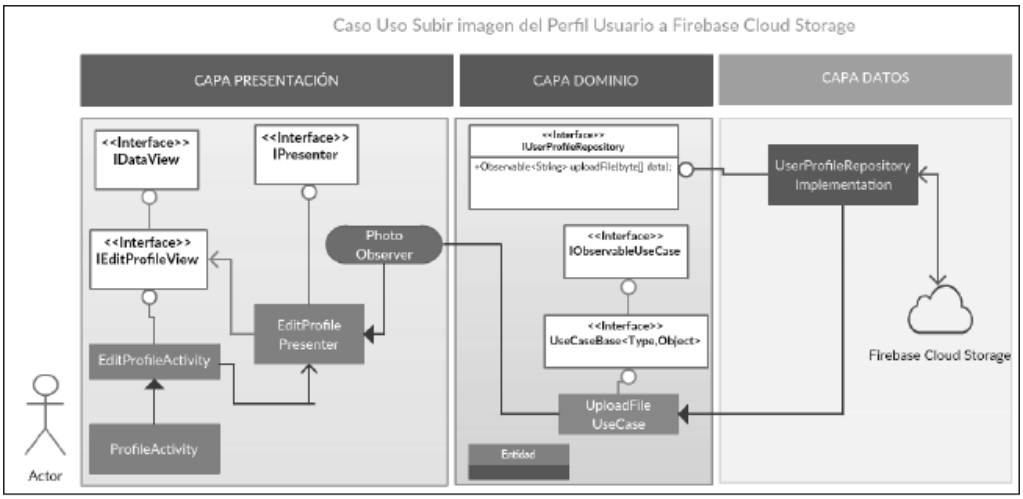


Fig. UC13D. Diagrama caso de uso subida de imagen.

Como siempre, empezamos por definir, en la capa del dominio, el repositorio **IUserProfileRepository**, con un método que recibirá un array de bytes de la imagen que queremos subir a la nube. Con ello se nos devolverá un enlace de descarga de la misma imagen.

```
public interface IUserProfileRepository     {       Observable<String>
uploadFile(byte[] data); }
```

Por tanto, una vez definido este repositorio, nos queda una tarea por realizar en el dominio: la definición del caso de uso **UploadFileUseCase** en los interactores. Su implementación recibe como parámetro los bytes de la imagen y retorna un observable con la URL obtenida tras almacenar estos bytes en Firebase.

```
public class UploadFileUseCase extends UseCaseBase<String, byte[]> {
@Inject public UploadFileUseCase( IUserProfileRepository
iUserProfileRepository
, Scheduler schedulerThread)
{       this.iUserProfileRepository = iUserProfileRepository;
                this.schedulerThread = schedulerThread; }

@Override public Observable<String> implementUseCase(DisposableObserver
observer, byte[] data) {
                Observable<String> observable = iUserProfileRepository.
uploadFile(data);
                        this.createUseCase(observable, observer,
schedulerThread);
                        return observable;
        }
}
```

Para poder utilizar este servicio de almacenamiento, debemos importar la librería de Firebase Cloud Storage en el fichero gradle de la capa de datos, que es desde donde se accede a este servicio.

```
compile 'com.google.firebase:firebase-storage:11.8.0'
```

Implementamos el repositorio del perfil del usuario en la capa de datos, en la clase **UserProfileRepositoryImplementation**, creando la instancia del almacén de Firebase Storage en el constructor de esta clase.

```
@Singleton public class UserProfileRepositoryImplementation
implements IUserProfileRepository {…                @Inject public
UserProfileRepositoryImplementation(Context context,
                        FirebaseAuth firebaseAuth, Scheduler
schedulerThread) {
        this.context = context;
        this.sessionDataStoreFactory = sessionDataStoreFactory;
        this.firebaseAuth = firebaseAuth;
        storage = FirebaseStorage.getInstance();
        this.schedulerThread = schedulerThread;
}
…
```

Sobrescribimos el método de subida de la imagen **uploadFile**, que recibe como parámetro los bytes de la imagen. Después, definimos una variable **path** con la ruta donde guardaremos la imagen que estamos subiendo, es decir, en el directorio **profiles** del repositorio base de nuestro almacén de ficheros.

Además, para que un usuario tenga una única imagen de perfil, cada vez que un usuario sube una imagen a este repositorio, esta se sobrescribe; por ello utilizamos el identificador único de usuario como nombre de imagen.

Una vez construido este path, mediante la instancia de Firebase Storage y el path que acabamos de construir, generamos una referencia **StorageReference**, que se corresponde con el sistema de almacenamiento Firebase.

```
…
@Override public Observable<String> uploadFile(byte[] data) {
        return Observable.create(emitter -> {
                String path = "profiles/" + firebaseAuth.getCurrentUser().
getUid() + ".png";
                storageRef = storage.getReference(path);
…
```

Empleando esta referencia que hemos obtenido, invocamos al método **putBytes**, al que le pasaremos los bytes de la imagen que debe subir a Firebase, con lo que obtenemos una tarea **UploadTask**.

Por último, para saber si se ha producido un error, añadimos un listener a la tarea y capturamos dicho evento por onFailure; si, por el contrario, la imagen se ha subido satisfactoriamente, recibiremos por el método onSuccess la URL de descarga a través del método **getDownloadUrl**, que es lo que queremos retornar a las capas inferiores.

```
…
         UploadTask uploadTask = storageRef.putBytes(data);
         uploadTask.addOnFailureListener(new OnFailureListener() {
@Override public void onFailure(@NonNull Exception exception)
                          { emitter.onError(exception); } } )
                          .addOnSuccessListener(new
OnSuccessListener<UploadTask.TaskSnapshot>()
                                { @Override public void
onSuccess(UploadTask.TaskSnapshot taskSnapshot)
                             {     Uri downloadUrl = taskSnapshot.
getDownloadUrl();
                                emitter.onNext(downloadUrl.toString());
                                emitter.onComplete(); }
                             });
                                });
          }
}
```

Tras definir la clase que implementa al repositorio del perfil del usuario, en la capa de presentación, proveemos de dicha dependencia en el módulo **AppModule**.

```
@Provides @Singleton static IUserProfileRepository
providesUserProfileRepository(
                                UserProfileRepositoryImplementation
userProfileRepositoryImplementation)
                                    { return
userProfileRepositoryImplementation; }
```

Creamos la vista **IEditProfileView** con un método que retorna la URL de la imagen que acabamos de subir.

```
public interface IEditProfileView extends IDataView {        void
photoUploaded(String value);       }
```

En esta misma capa, crearemos el observador **PhotoObserver**, el cual retornará, en caso de éxito, la ruta de la imagen que acabamos de subir.

```
public class PhotoObserver extends DisposableObserver<String> {
…
@Override public void onNext(String value) { iEditProfileView.
photoUploaded(value); }
…
}
```

Una vez que tenemos la vista y el observador, es el momento ideal para crear el presentador, que en este caso es **EditProfilePresenter**, y definir en su constructor la vista y el caso de uso de subida de la imagen.

Además implementamos el método **uploadProfilePhoto**, responsable de invocar al caso de uso de subida de imágenes, pasándole el observador como parámetro y el array de bytes de la imagen.

```
public class EditProfilePresenter implements IPresenter {
    @Inject public EditProfilePresenter(IEditProfileView iEditProfileView,
                                    UploadFileUseCase uploadFileUseCase)
{       this.iEditProfileView = iEditProfileView;
```

```
        this.uploadFileUseCase = uploadFileUseCase; }
public void uploadProfilePhoto(byte[] data)
{       uploadFileUseCase.implementUseCase(new
PhotoObserver(iEditProfileView), data); }
}
```

Creamos el módulo **EditProfileModule** para proveer del presentador y la vista de la activity EditProfileActivity que crearemos.

```
@Module public abstract class EditProfileModule
{
@Provides static EditProfilePresenter provideEditProfilePresenter(
IEditProfileView iEditProfileView,                    UploadFileUseCase
uploadFileUseCase) {
                        return new EditProfilePresenter(iEditProfile
View, uploadFileUseCase);
                      }
       @Binds abstract IeditProfileView provideEditProfileView(
EditProfileActivity editProfileActivity);
}
```

Añadimos a la clase **BuildersModule** la actividad que acabamos de crear **EditProfileActivity**.

```
@PerActivity @ContributesAndroidInjector(modules = EditProfileModule.class)
        abstract EditProfileActivity contributeEditProfileActivity();
```

Creamos una nueva actividad **EditProfileActivity** para editar el perfil del usuario. Su layout se compone de un ImageView para editar la imagen del usuario y un EditText para editar el nombre.

Esta actividad implementa la vista **IEditProfileView**, por lo que sobrescribiremos sus métodos. En concreto nos interesa el método **photoUploaded**, que es por donde recibiremos la URL del recurso que acabamos de subir a Firebase.

Inicialmente inyectamos, en la actividad, el presentador **EditProfilePresenter**; posteriormente, en el método **onCreate**, inyectamos mediante el AndroidInjection la actividad para Dagger, como habíamos hecho hasta ahora, y por último inicializamos Butterknife y llamamos al método **renderUser**, responsable de obtener desde el intent y representar la información en la interfaz.

Como observamos, obtenemos el intent que la actividad del perfil nos ha enviado y capturamos en sus extras el objeto usuario que hemos serializado, así como la imagen del perfil también serializada.

Deserializados los datos, los asignamos a una variable global currentUser y, por último, representamos estos datos en la interfaz, tanto el nombre como el email. Finalmente convertimos los datos de array de bytes a Bitmap y se lo asignamos al ImageView.

```
private void renderUser() {    Intent intent = getIntent();
Gson gson = new Gson();
String currentUserJson = intent.getExtras().getString(Constants.
ExtraCurrentUser);
byte[] data =     intent.getExtras().getByteArray(Constants.
ExtraCurrentProfileImage);
currentUser = gson.fromJson(currentUserJson, User.class);
editprofile_email.setText(currentUser.getEmail());
```

```
editprofile_name.setText(currentUser.getName());
    Bitmap bitmap = BitmapFactory.decodeByteArray(data, 0, data.length);
    if (bitmap != null) editprofile_image.setImageBitmap(bitmap);
}
```

Capturamos el evento clic sobre la imagen a través del método **doUploadProfileImage**, lanzando un intent a la galería de imágenes de Android, para seleccionar una y asignarla como imagen.

```
@OnClick(R.id.editprofile_image_button) public void doUploadProfile
Image() {
        Intent intent = new Intent(Intent.ACTION_GET_CONTENT);
intent.setType("image/*");
intent.addCategory(Intent.CATEGORY_OPENABLE);
startActivityForResult(intent, Flags.FILE_SELECTOR);
}
```

Una vez elegida la imagen de la galería recibimos, en el onActivityResult, los datos de la imagen seleccionada en una URI mediante el método **getData**. Esta URI se utilizará para obtener un bitmap que se asociará al ImageView.

```
@Override public void onActivityResult(int requestCode, int resultCode,
 Intent data) {           super.onActivityResult(requestCode, resultCode,
 data);
    if (requestCode == Flags.FILE_SELECTOR && resultCode == RESULT_OK) {
        Uri selectedimage = data.getData();
        try {
editprofile_image.setImageBitmap(MediaStore.Images.Media
 .getBitmap(this.getContentResolver(),selectedimage));
…
```

Posteriormente, obtenemos el array de bytes de la imagen, para proceder a subirla a Firebase Cloud Storage. Podemos comprimirla y perder un poco de calidad, aunque nosotros la dejaremos al 100 % de calidad.

Por último, subiremos estos bytes a Firebase a través del método **uploadProfilePhoto** del presentador.

```
…

            editprofile_image.setDrawingCacheEnabled(true);
            editprofile_image.buildDrawingCache();
            Bitmap bitmap = editprofile_image.getDrawingCache();
            ByteArrayOutputStream baos = new ByteArrayOutputStream();
            bitmap.compress(Bitmap.CompressFormat.JPEG, 100, baos);

            byte[] data2 = baos.toByteArray();
            editProfilePresenter.uploadProfilePhoto(data2);
        } catch (IOException e) { e.printStackTrace(); }
    }
}
```

Recibimos de Firebase la URL de la imagen que acabamos de almacenar y asignamos esta URL a la variable global del usuario para guardarla si queremos.

```
@Override public void photoUploaded(String value) { currentUser.
setPhotoUrl(value); }
```

Podemos acceder a la actividad de edición del perfil mediante el botón **Editar perfil** de la actividad **ProfileActivity**. Será responsable de serializar el usuario y la imagen, información que pasará como extras a través del intent.

```
@OnClick(R.id.profile_edit) public void doEdit() {
    Intent intent = new Intent(this, EditProfileActivity.class);

    Gson gson = new Gson();
    String currentUserJson = gson.toJson(currentUser, User.class);

    intent.putExtra(Constants.ExtraCurrentUser, currentUserJson);
 Bitmap bitmap =  ((BitmapDrawable) profile_image.getDrawable()).
getBitmap();
    ByteArrayOutputStream stream = new ByteArrayOutputStream();
    bitmap.compress(Bitmap.CompressFormat.PNG, 100, stream);
    byte[] byteArray = stream.toByteArray();

    intent.putExtra(Constants.ExtraCurrentProfileImage, byteArray);
    startActivityForResult(intent, Flags.REQUEST_EDIT_PROFILE);
}
```

Al ejecutar la app y entrar en la sección privada del perfil, pulsamos sobre el botón **Editar perfil** y abrimos la actividad de edición del perfil. Será aquí donde, tras hacer clic en la imagen, seleccionaremos la imagen que queremos como imagen del perfil (Figura UC13-1), que subiremos a Firebase tras seleccionarla (Figura UC13-2).

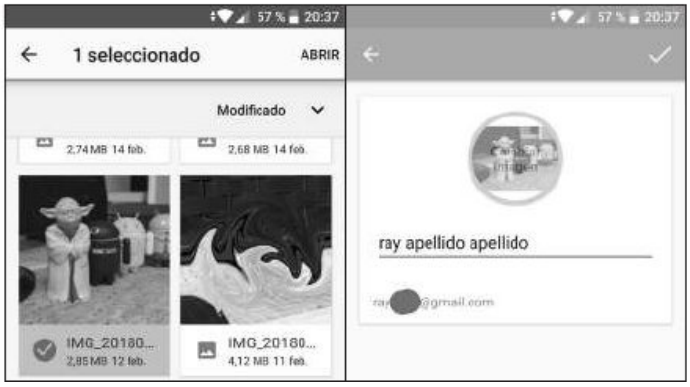


Fig. UC13-1. Seleccionar imagen. Fig. UC13-2. Subir imagen.

Caso de uso - Guardar perfil local y remotamente

Como hemos visto hasta ahora, tras autenticarnos en Firebase, obtenemos un usuario con una información. Esta información, tras registrar nuestro usuario en Firebase, se almacena en el sistema, aunque podemos modificarla desde nuestra app. Para ello hemos hecho que este caso de uso actualice remotamente el nombre del perfil del usuario, así como la URL de la imagen del usuario. Además, a través de Realm, podremos almacenar localmente información de la sesión del usuario. Podemos observarlo en el diagrama siguiente:

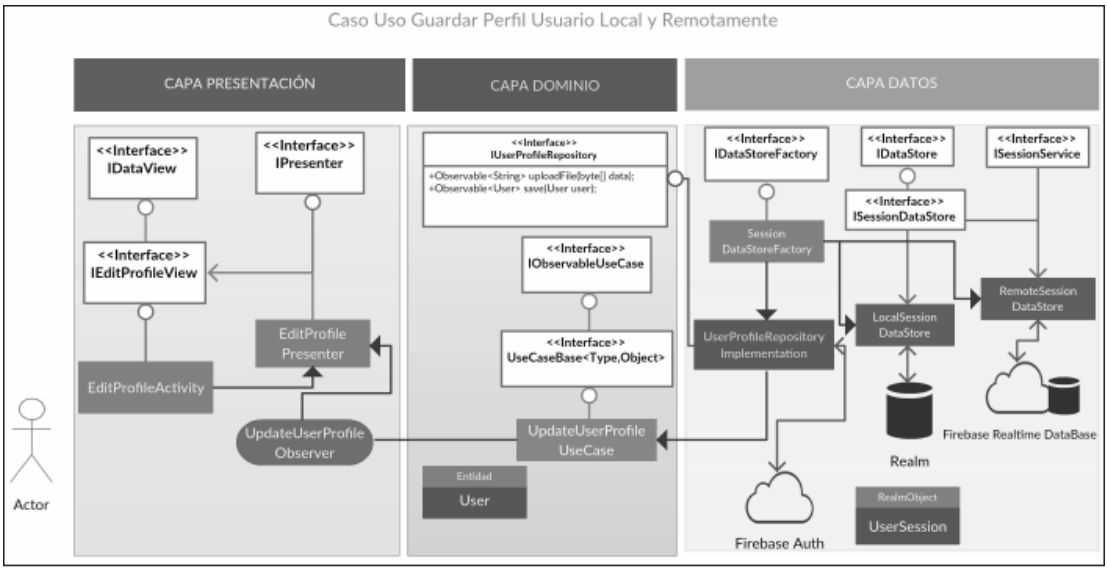


Fig. UC14D. Diagrama caso de uso guardar perfil.

Empecemos por crear un método nuevo en la interfaz del repositorio del perfil del usuario **IUserProfileRepository** para almacenar el usuario.

```
Observable<User> save(User user);
```

El caso de uso responsable de esta tarea será **UpdateUserProfileUseCase**, el cual retornará un observable con el usuario que acaba de almacenar y requerirá un parámetro del mismo tipo para referirse al usuario.

```
public class UpdateUserProfileUseCase extends UseCaseBase<User, User> {
@Inject public UpdateUserProfileUseCase(IUserProfileRepository
iUserProfileRepository
, Scheduler schedulerThread)
{       this.iUserProfileRepository = iUserProfileRepository;
        this.schedulerThread = schedulerThread;
}
@Override public Observable<User> implementUseCase(DisposableObserver
observer, User user) {
        Observable<User> observable = iUserProfileRepository.save(user);
        this.createUseCase(observable, observer, schedulerThread);
        return observable;
}
}
```

Hemos comentado que vamos a necesitar la base de datos Realm para el almacenamiento local; para ello, debemos añadir la dependencia de Realm a nivel global de la aplicación, declarándolo en el build.gradle del proyecto.

```
classpath "io.realm:realm-gradle-plugin:4.2.0"
```

Además, al principio del fichero del build.gradle del módulo **data**, que es donde vamos a utilizar Realm, aplicaremos el plugin de esta librería en Android.

```
apply plugin: 'realm-android'
```

Conectaremos remotamente con Firebase para almacenar la información y aprovecharemos para guardar, como perfil público, el nombre que guardemos al editar el perfil de usuario.

Para realizar esta última acción, debemos definir la interfaz del servicio **ISessionService**, en la capa de datos, para crear un nodo en la base de datos de Firebase del tipo **public_profile/ #UID# /#NOMBRE#**.

Esto nos resultará útil para asociar identificadores de usuario con un nombre público en los comentarios que hagan los usuarios sobre los hoteles.

```
public interface ISessionService {
@PUT("public_profile/{uid}.json") Call<String> createPublicProfile(
                  @Path(value = "uid", encoded = true) String uid, @
Query("auth") String authorization
                  , @Body String name);
}
```

Los métodos que implementarán los almacenes de datos de la sesión de usuario, remoto y local, los definimos en la interfaz **ISessionDataStore**, que se encuentra en la capa de datos, paquete **datastore**. Uno de estos métodos será para almacenamiento local, llamado **saveSession**, y el otro para el almacenamiento remoto, llamado **savePublicProfile**.

```
public interface ISessionDataStore extends IDataStore {
boolean saveSession(User user);
Observable<User> savePublicProfile(User user);
}
```

El almacenamiento remoto de la sesión se implementará en la clase **RemoteSessionDataStore**, ubicada en el paquete **datastore/remote**. En el constructor instanciamos el servicio de sesión para su invocación por parte de Retrofit.

```
public class RemoteSessionDataStore implements ISessionDataStore {
 public RemoteSessionDataStore(final Context context)
 {… iSessionService = retrofit.create(ISessionService.class); }
…
```

Para almacenar el nombre del perfil del usuario como perfil público de la base de datos de Firebase, invocamos el método **createPublicProfile** y le pasamos el UID del usuario, utilizado para asociar un identificador de usuario con un nombre. También enviaremos el token de autenticación, dado que únicamente podrá editar su registro el usuario cuya autenticación coincida con el UID que pretende modificar. Y por último enviamos el nombre.

```
…
 @Override public Observable<User> savePublicProfile(User user) {
 return Observable.create(emitter -> {
 try {   Call<String> call =      iSessionService.createPublicProfile
(user.getUserId(),
```

```
                        user.getAuthToken(),user.getName());
                        Response<String> response = call.execute();
                        if (response.code() == 200) {
                                emitter.onNext(user);
                                emitter.onComplete();
                        } else   emitter.onError(new Exception
(response.errorBody().toString()));
                } catch (Exception e) { emitter.onError(e); }
        });
}
}
```

Para almacenar localmente la sesión del usuario, mediante Realm, y no crear una dependencia de Realm en el dominio, hemos preferido replicar la información de la clase User del dominio en un objeto llamado **UserSession**, que hemos creado en la capa de datos en el paquete **realm/DTO**. Recordemos que, para que Realm reconozca nuestra clase como un objeto de tipo Realm, debemos extenderlo a **RealmObject**.

```
public class UserSession extends RealmObject {
@PrimaryKey private String UserId;
private String Name;
private String Email;
private String PhotoUrl;
private String ProfilePhoto;
private String Provider;
private String AuthToken;
…
}
```

Llega el turno de implementar el almacenamiento local mediante la clase **LocalSessionDataStore**, en el paquete **datastore/local**. En su constructor, creamos la configuración RealmConfiguration, donde establecemos el nombre del contenedor Realm, la versión del esquema y, en caso de que hayamos modificado la estructura del objeto Realm que tenemos almacenado, recrearemos el esquema del objeto en lugar de realizar una migración, dado que estamos comenzando y no hay datos útiles.

```
public class LocalSessionDataStore implements ISessionDataStore {
        public LocalSessionDataStore(Context context) {
                RealmConfiguration myConfig = new RealmConfiguration.
Builder()
                        .name("UserSession.realm").schemaVersion(3)
                        .deleteRealmIfMigrationNeeded().build();
                Realm.setDefaultConfiguration(myConfig);
                myUserRealm = Realm.getInstance(myConfig);
 }
…
```

Mediante el método **saveSession**, le pasamos el objeto de usuario del dominio, relleno, de tal forma que lo mapeamos al tipo de objeto **UserSession**.

Posteriormente, utilizamos la instancia del contenedor Realm para ejecutar una transacción, donde asignaremos a la propiedad setLastAccess de la instancia de la sesión de usuario, la fecha actual como fecha del último acceso. Finalmente,

mediante copyToRealmOrUpdate, almacenamos este objeto Realm, de tal forma que, en caso de existir, lo sobrescribiremos y no dará error de solapamiento.

```
…
@Override public boolean saveSession(User user) {
        UserSession userSession = UserToUserSession.Create(user);
        try {
                myUserRealm.executeTransaction(new Realm.Transaction() {
                        @Override public void execute(Realm realm) {
                                userSession.setLastAccess(new Date());
                                realm.copyToRealmOrUpdate(userSession);
                        }
                });
              }catch (Exception error) { return false; }
        finally { myUserRealm.close(); }
              return true;
    }
}
```

Siguiendo con la capa de datos, definimos la factoría de almacenes de datos de la sesión del usuario **SessionDataStoreFactory**, en el paquete **factory**, donde de acuerdo con el tipo de acceso requerido retornaremos acceso remoto o local.

```
@Singleton public class SessionDataStoreFactory implements IDataStore
Factory {
@Inject public SessionDataStoreFactory(Context context) { this.context =
 context;}
@Override public ISessionDataStore Remote()
{       return new RemoteSessionDataStore(context);         }
@Override public ISessionDataStore Local()
{       return new LocalSessionDataStore(context);          }
}
```

En la implementación de la interfaz del repositorio del perfil del usuario UserProfileRepositoryImplementation, añadimos el nuevo método **save**. Será aquí donde, tras recibir por parámetro el usuario que se va a almacenar, actualizaremos información del nombre y URL de la imagen del perfil, en el sistema de usuarios de Firebase creando una petición **UserProfileChangeRequest**, que utilizaremos invocando el método **updateProfile** del usuario de Firebase conectado.

```
@Override public Observable<User> save(User user) {

        UserProfileChangeRequest profileUpdates =    new
UserProfileChangeRequest.Builder()
                                    .setDisplayName(user.getName())
                                            .setPhotoUri(Uri.parse(user.
getPhotoUrl())).build();

        firebaseAuth.getCurrentUser().updateProfile(profileUpdates)
                                    .addOnCompleteListener(new
OnCompleteListener<Void>() {
                                    @Override public void onComplete
(@NonNull Task<Void> task) {
…
```

Una vez actualizada la información en Firebase, en caso de éxito, almacenamos localmente el usuario a través de Realm.

```
…
                    if(task.isSuccessful()) { sessionDataStoreFactory.
Local().saveSession(user); }
            }
});
 …
```

Por último almacenamos el perfil público, remotamente, en la BD de Firebase.

```
…
return sessionDataStoreFactory.Remote().savePublicProfile(user);
}
```

Aprovechando el almacenamiento local, vamos a ampliar la funcionalidad para almacenar localmente el usuario cada vez que obtengamos un token de autenticación en la capa de datos, clase **SignInRepositoryImplementation**, método **getTokenOnSuccessFulSignIn**.

```
…        if (sessionDataStoreFactory.Local().saveSession(user))      {
emitter.onNext(user); emitter.onComplete();}
…
```

Inicializamos la base de datos Realm con la configuración básica, en la capa de presentación, en el método **onCreate** de la clase de la aplicación **App**.

```
Realm.init(this);
RealmConfiguration realmConfiguration = new RealmConfiguration.Builder().
build();
Realm.setDefaultConfiguration(realmConfiguration);
```

Creamos un nuevo método **updatedUser**, en la vista **IEditProfileView**, a través del cual se recibirán las respuestas del observador, al recibir cambios de estado tras almacenar con éxito el perfil del usuario.

```
void updatedUser(User value);
```

Creamos el observador UpdateUserProfileObserver, responsable de controlar los cambios de estado de la actualización del perfil del usuario. En caso de éxito, recibimos el usuario que acabamos de almacenar y, de esta forma, respondemos a la vista que se ha actualizado el usuario.

```
public class UpdateUserProfileObserver extends DisposableObserver<User>
{        @Override public void onNext(User value) { iEditProfileView.
updatedUser(value); }     }
```

Al constructor del presentador **EditProfilePresenter** le añadimos el caso de uso UpdateUserProfileUseCase, así como también al constructor del presentador, que proveemos en el **EditProfileModule**. Creamos además el método **saveProfile**, res-

ponsable de ejecutar el caso de uso, pasándole el observador y, como parámetro, el usuario por almacenar.

```
@Inject public EditProfilePresenter(… , UpdateUserProfileUseCase
 updateUserProfileUseCase,…)
{…
           public void saveProfile(User user) {
                       updateUserProfileUseCase.implementUseCase(new
UpdateUserProfileObserver(iEditProfileView), user);
}
```

Por último, en la actividad EditProfileActivity, capturamos el evento clic sobre el icono **Guardar**, del menú de la actividad, para lanzar el método de almacenar la información del presentador.

```
@Override public boolean onOptionsItemSelected(MenuItem item) {
    int id = item.getItemId();
    if (id == R.id.menu_save) {
        currentUser.setName(editprofile_name.getText().toString());
        currentUser.setEmail(editprofile_email.getText().toString());
        editProfilePresenter.saveProfile(currentUser);
        return true;
    }
    return super.onOptionsItemSelected(item);
}
```

Tras capturar el éxito de almacenamiento, sobrescribimos el método **updatedUser** y cerramos la actividad.

```
@Override public void updatedUser(User value) { finish(); }
```

Tras ejecutar la aplicación y entrar en la pantalla de edición del perfil, si modificamos el nombre y pulsamos sobre el icono del menú **Guardar** (Figura UC14-1), guardamos esta información local y remotamente. Tras almacenarse correctamente, se cierra la edición del perfil y se regresa a la actividad de visualización del perfil (Figura UC14-2).

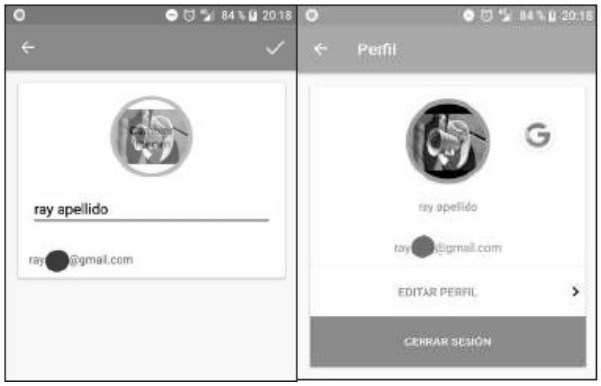


Fig. UC14-1. Actualizar perfil. Fig. UC14-2. Perfil actualizado.

Caso de uso - Listar opiniones

Los hoteles pueden contener comentarios realizados por usuarios del sistema; por esta razón, hemos habilitado esta funcionalidad. Todo hotel tendrá una lista de opiniones, así como una valoración media basada en las opiniones. Podemos verlo en el siguiente diagrama:

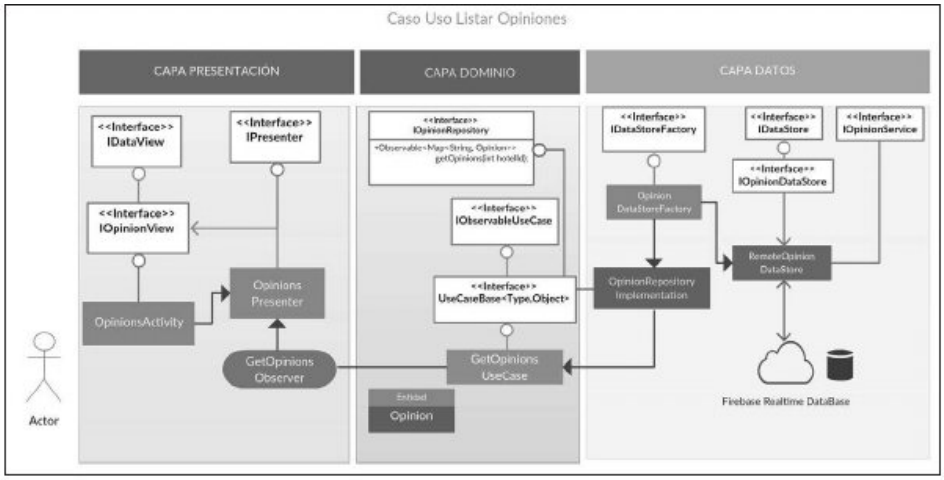


Fig. UC15D. Diagrama listado de opiniones.

Creamos, en la capa de dominio, la interfaz del repositorio **IOpinionRepository**, con el método de obtención de opiniones de acuerdo con un identificador de hotel numérico.

```
public interface IOpinionRepository { Observable<Map<String, Opinion>>
getOpinions(int hotelId); }
```

En esta misma capa, generamos el caso de uso GetOpinionsUseCase y lo almacenamos en los interactores. Este recibirá un valor entero con el número de hotel buscado y retornará una colección de opiniones de usuarios para dicho hotel.

```
public class GetOpinionsUseCase extends UseCaseBase<Map<String,
Opinion>, Integer> {
@Inject public GetOpinionsUseCase(IOpinionRepository iOpinionRepository,
                        Scheduler schedulerThread) {
        this.iOpinionRepository = iOpinionRepository;
        this.schedulerThread = schedulerThread;
}
@Override public Observable<Map<String,Opinion>>   implementUseCase(
 DisposableObserver observer,
Integer hotelId)
{       Observable<Map<String,Opinion>> observable =
iOpinionRepository.getOpinions(hotelId);
                this.createUseCase(observable, observer,
schedulerThread);
                return observable;
}
}
```

Nos posicionamos en la capa de datos, donde crearemos la interfaz IOpinionService, en el paquete de servicios que invocaremos. En él, definimos un método de tipo GET para obtener el listado de opiniones de un hotel.

```
public interface IOpinionService {
@GET("opinions/{id}.json") Call<Map<String, Opinion>> getOpinions(
        @Path(value = "id", encoded = true) int hotelId);
}
```

En esta misma capa, en el paquete de almacenes de datos, creamos una interfaz IOpinionDataStore, con un método que implementaremos en el acceso remoto a estos datos.

```
public interface IOpinionDataStore extends IDataStore
{       Observable<Map<String, Opinion>> getOpinions(int hotelId); }
```

La clase responsable de implementar el acceso a estos datos remotamente es **RemoteOpinionDataStore**, ubicada en el paquete **remote** de los almacenes de datos de la capa de datos. Como novedad, añadimos al constructor de Retrofit la factoría de conversión de datos, de tipo JSON, indicamos el formato de fecha que vamos a convertir y, mediante serializeNulls, indicamos también que serialice aquellos datos del JSON que se encuentren con valor nulo.

```
public class RemoteOpinionDataStore implements IOpinionDataStore {
public RemoteOpinionDataStore(final Context context) {
        Gson gson = new GsonBuilder()
                                .setDateFormat("yyyy-MM-dd'T'HH:mm:ss")
                                .serializeNulls()
                                .create();
       retrofit = new Retrofit.Builder()
                .baseUrl(context.getResources().getString(R.string.
apiBaseURL))
                .addConverterFactory(GsonConverterFactory.create(gson))
                .client(Utilities.initializeHttpClient(context).build()).
build();
       iOpinionService = retrofit.create(IOpinionService.class);
    }
…
```

De esta forma, invocamos el método del servicio para obtener las opiniones y retornarlas al observador de la capa de presentación en caso de éxito.

```
…
@Override public Observable<Map<String, Opinion>> getOpinions
(int hotelId)
{ return Observable.create(emitter -> {
                    try { Call<Map<String, Opinion>> call = iOpinion
Service.getOpinions(hotelId);
               Response<Map<String, Opinion>> response = call.execute();
               if (response.code() == 200) {
                        Map<String, Opinion> opinions = response.body();
                        emitter.onNext(opinions);
                        emitter.onComplete();
        } else emitter.onError(new Exception(response.errorBody().
toString()));
```

```
            } catch (Exception e) { emitter.onError(e); }
        });
    }
}
```

Para elegir entre almacenamiento local o remoto, necesitamos la factoría de almacenes de datos **OpinionDataStoreFactory**, de la cual únicamente aprovecharemos la opción de almacenamiento remoto, dado que no almacenaremos nada localmente por lo que respecta a opiniones.

```
@Singleton public class OpinionDataStoreFactory implements
IDataStoreFactory
{
@Override public IOpinionDataStore Remote() {      return new
RemoteOpinionDataStore(context); }
}
```

En la capa de datos implementamos la interfaz del repositorio, la clase **OpinionRepositoryImplementation**. En ella utilizamos el método del repositorio **getOpinions** para invocar al método del acceso remoto a datos de almacenes de las opiniones.

```
@Singleton public class OpinionRepositoryImplementation implements
IOpinionRepository {
@Inject public OpinionRepositoryImplementation(OpinionDataStoreFactory
opinionDataStoreFactory)
{       this.opinionDataStoreFactory = opinionDataStoreFactory;     }
@Override public Observable<Map<String, Opinion>> getOpinions(int hotelId)
{       return opinionDataStoreFactory.Remote().getOpinions(hotelId);
}
}
```

Pasamos a la capa de presentación, donde definiremos inicialmente la interfaz de la vista **IOpinionsView** con un método para mostrar por pantalla las opiniones recibidas de un hotel.

```
public interface IOpinionsView extends ISessionView
{       void renderData(Map<String, Opinion> value);       }
```

Creamos el observador **GetOpinionsObserver**, que, en caso de éxito de la petición, retorne la colección de opiniones de los usuarios en un hotel determinado.

```
public class GetOpinionsObserver   extends
DisposableObserver<Map<String,Opinion>>
{ @Override public void onNext(Map<String,Opinion> value)    {
iOpinionsView.renderData(value);   } }
```

Una vez que tenemos la vista definida, el caso de uso que se va a invocar y el observador que se utilizará, estamos listos para declarar el presentador de las opiniones: **OpinionsPresenter**.

```
public class OpinionsPresenter implements IPresenter {
    @Inject public OpinionsPresenter(IOpinionsView iOpinionsView
                                                        ,
CheckIsUserSignedInUseCase checkIsUserSignedInUseCase
                                                    , GetOpinionsUseCase
getOpinionsUseCase) {
                this.iOpinionsView = iOpinionsView;
                this.getOpinionsUseCase = getOpinionsUseCase;
                this.checkIsUserSignedInUseCase =
checkIsUserSignedInUseCase;
}
…
```

Inicializamos el presentador y buscamos primero si tenemos una sesión abierta para mostrar u ocultar el botón de registro de nuevas opiniones. Paralelamente obtenemos las opiniones del hotel que indicamos por parámetro, ejecutando para ello el caso de uso responsable de realizar esta tarea.

```
…
        public void initialize(int hotelId) { isSignedIn();
getOpinions(hotelId); }

public void getOpinions(int hotelId)
{       getOpinionsUseCase.implementUseCase(new
GetOpinionsObserver(iOpinionsView), hotelId);       }
}
```

Creamos, en la capa de presentación, un adaptador llamado **OpinionsMapAdapter** para listar las opiniones. Es prácticamente igual al que realizamos para los hoteles, con la salvedad de que en lugar de ser un listado de hoteles, es decir List<Hotel>, los elementos del adaptador, los ítems o elementos son de tipo Map<String,Opinion>, de modo que la clave de cada registro de esta colección es un String que identifica al identificador de usuario UID que registró la opinión. Además, el valor se representa por un objeto **Opinion**, compuesto por la fecha de registro del comentario, la puntuación otorgada por el usuario y el comentario.

Para la inyección de vistas del layout card_view_opinion, que representa la vista de cada elemento de la lista, hemos creado un ViewHolder llamado **OpinionsViewHolder**.

Por tanto, la representación de cada opinión o elemento será implementada en el adaptador a través de su método **onBindViewHolder**.

```
@Override public void onBindViewHolder(OpinionsViewHolder holder, int
position)
{       final Opinion opinion = (new ArrayList<>(this.items.values()).
get(position));
holder.card_message.setText(opinion.getMessage());
SimpleDateFormat dateFormat = new SimpleDateFormat(mContext.getString
(R.string.date_pattern),
Locale.getDefault());
holder.card_creationdate.setText(dateFormat.format(opinion.getCreation
Date().getTime()));
holder.card_rating.setRating(opinion.getRating());
holder.card_name.setText(opinion.getName() != null ?  opinion.getName()
                                                                    :
 mContext.getString(R.string.anonymous));
}
```

Creamos la actividad **OpinionsActivity**, que implementará la interfaz de su vista **IOpinionsView**, como hemos hecho hasta ahora e inyectamos en la actividad su presentador **OpinionsPresenter**.

Será en su método **onCreate** donde inyectemos la actividad a Dagger, inicialicemos Butterknife y capturemos en los extras del intent, lanzado por la actividad del listado de hoteles HomeActivity, el hotel seleccionado desde dicha actividad. De esta forma, podremos reunir información relativa al hotel y centrarnos únicamente en obtener el listado de opiniones asociadas al hotel.

Para obtener las opiniones, bastará con invocar al método de inicialización del presentador, al cual le pasaremos el identificador de hotel por el cual se filtrarán las opiniones.

```
opinionsPresenter.initialize(currentHotel.getId());
```

Una vez obtenida la colección de opiniones de los distintos usuarios que han comentado el hotel, a través del método sobrescrito de la interfaz de la vista **renderData**, asignaremos esta información al adaptador OpinionsMapAdapter, asignando este adaptador al RecyclerView de la actividad.

```
@Override public void renderData(Map<String, Opinion> data)
{        opinionsMapAdapter = new OpinionsMapAdapter(data, this);
LinearLayoutManager linearLayoutManager =
                                                    new
LinearLayoutManager(this, LinearLayoutManager.VERTICAL, false);
opinions_recycler.setLayoutManager(linearLayoutManager);
opinions_recycler.setAdapter(opinionsMapAdapter);
}
```

Finalmente nos falta crear el módulo **OpinionsModule**, para Dagger, proveyendo del presentador y la vista de la actividad de las opiniones.

```
@Module public abstract class OpinionsModule {
@Provides static OpinionsPresenter provideOpinionsPresenter
( IOpinionsView iOpinionsView
        , CheckIsUserSignedInUseCase checkIsUserSignedInUseCase
        , GetOpinionsUseCase getOpinionsUseCase)
{ return new OpinionsPresenter(iOpinionsView, checkIsUserSigned
InUseCase, getOpinionsUseCase); }
@Binds abstract IOpinionsView provideOpinionsView(  OpinionsActivity
opinionsActivity);
}
```

Agregamos a la clase **BuildersModule** la nueva actividad con su módulo.

```
@PerActivity @ContributesAndroidInjector(modules = OpinionsModule.class)
abstract OpinionsActivity contributeOpinionsActivity();
```

Por último proveemos del repositorio IOpinionRepository, en el módulo de la aplicación, para poder ser inyectado en capas superiores.

```
@Provides @Singleton static IOpinionRepository providesOpinionRepository
                                    ( OpinionRepositoryImplementation
opinionRepositoryImplementation)
```

```
                                        {     return
opinionRepositoryImplementation;   }
```

Llegados a este punto, nos preguntaremos cómo llegamos a la pantalla de opiniones de un hotel. La respuesta es haciendo clic sobre un hotel del listado de hoteles de la HomeActivity. Dado que usamos un RecyclerView para listar los hoteles, debemos implementar el evento clic sobre el hotel y que nos abra esta actividad de opiniones.

Por ello inicialmente creamos una clase **RecyclerViewClickListener** en el paquete de adaptadores de la capa de presentación, de modo que capture el evento clic sobre elementos de un RecyclerView mediante GestureDetector, el evento de un clic simple, que será interceptado.

```
public class RecyclerViewClickListener implements RecyclerView.
OnItemTouchListener {
public interface OnItemClickListener {     void onItemClick(View view,
int position); }

public RecyclerViewClickListener(Context context, OnItemClickListener
listener) {…
                mGestureDetector = new GestureDetector(context,
                        new GestureDetector.SimpleOnGestureListener() {
                    @Override public boolean onSingleTapUp(MotionEvent
 e) { return true; }
                });
}
@Override public boolean onInterceptTouchEvent(     RecyclerView rv,
MotionEvent e) {
                                    View childView = rv.findChildViewUnder
(e.getX(), e.getY());
        if (childView != null && mListener != null  && mGestureDetector.
onTouchEvent(e))
                mListener.onItemClick(childView,
rv.getChildAdapterPosition(childView));     return false;
        }
}
```

Definimos esta captura de evento clic en la actividad HomeActivity, en el método **renderData**, donde capturamos la posición del elemento de la lista que ha sido pulsada y, a partir de esta, obtenemos la instancia del hotel que hemos seleccionado. Con esta información, abrimos la actividad OpinionsActivity.

```
home_hotels_recycler.addOnItemTouchListener(new RecyclerViewClick
Listener(         this,(view, position) -> {
        Hotel hotel = ((HotelsListAdapter) home_hotels_recycler.
getAdapter()).getItemAtPosition(position);
        Intent intent = new Intent(       getApplicationContext(),
OpinionsActivity.class);
        Gson gson = new Gson();
        intent.putExtra(Constants.ExtraCurrentHotel, gson.toJson(hotel));
        startActivity(intent);
}));
```

Se ejecutamos la app, tras seleccionar un hotel de la actividad principal, se abrirá la actividad de las opiniones del hotel seleccionado, como observamos en

la imagen. En caso de tener una sesión abierta, se habilitará el botón flotante de crear nuevos comentarios.

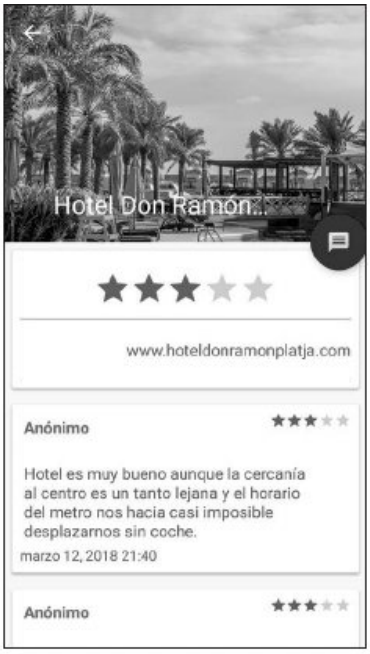


Fig. UC15-1. Listado de opiniones.

Caso de uso - Guardar comentario

Hemos creado este caso de uso para almacenar una nueva opinión de un usuario sobre un hotel. Almacenamos la valoración y opinión de un cliente en la base de datos remota de Firebase, como observamos en el siguiente diagrama:

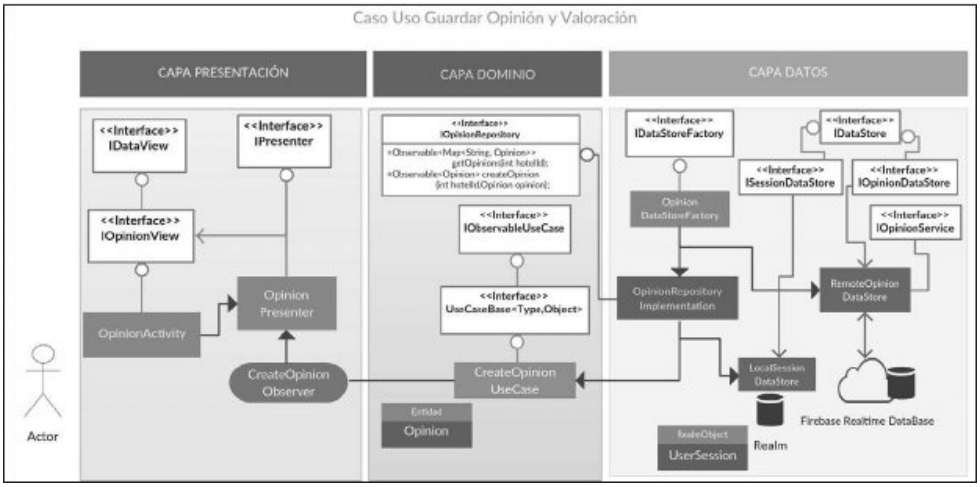


Fig. UC16D. Diagrama caso de uso guardar opinión y valoración.

Como habitualmente hacemos, empezamos por crear un método nuevo en la interfaz del repositorio de opiniones **IOpinionRepository** del dominio.

```
Observable<Opinion> createOpinion(int hotelId,Opinion opinion);
```

En los interactores de esta misma capa, creamos el caso de uso **CreateOpinionUseCase**, el cual devolverá un observable con el comentario que hemos almacenado y que requiere, por parámetro, el identificador del hotel donde se almacenará el comentario, así como el propio comentario y valoración del usuario.

```
public class CreateOpinionUseCase  extends UseCaseBase<Opinion,
OpinionParameters.Parameters> {
     @Inject public CreateOpinionUseCase(IOpinionRepository
iOpinionRepository, Scheduler schedulerThread) {
         this.iOpinionRepository = iOpinionRepository;
         this.schedulerThread = schedulerThread;
}
@Override public Observable<Opinion> implementUseCase(
DisposableObserver observer

         , OpinionParameters.Parameters parameters)
{        Observable<Opinion> observable =  iOpinionRepository.
createOpinion(
                                                    parameters.
getHotelId(),parameters.getOpinion());
                        this.createUseCase(observable, observer,
schedulerThread);
                 return observable;
}
}
```

Para el almacenamiento de este comentario, necesitamos añadir un método en la interfaz del servicio de opiniones **IOpinionService**, de la capa de datos. Este método es **createOpinion**, y necesita el identificador de usuario para saber, dentro de los comentarios de un hotel, qué usuario los ha escrito. También necesitamos el parámetro auth, que incluiremos en la URL de la llamada y que contiene el token de autorización del usuario que quiere insertar un registro. Y, por último, necesitaremos un parámetro con la opinión que se va a insertar.

```
@PUT("opinions/{id}/{uid}.json")
Call<Opinion> createOpinion(@Path(value = "id", encoded = true)
int hotelId
, @Path(value = "uid", encoded = true) String uid, @Query("auth")
String authorization
, @Body Opinion opinion);
```

Como necesitamos saber el token de autorización del usuario que se ha validado en el sistema, leemos esta información del último usuario logueado en nuestra app, almacenado localmente en Realm. Para ello declaramos un método **getSession** en la interfaz ISessionDataStore del paquete **datastore**.

```
User getSession();
```

Dado que debemos pasarle por parámetro el token de autenticación del usuario, para comprobar si este tiene permiso o no de escritura sobre el registro que quiere escribir, obtenemos la sesión del último usuario validado en el sistema en nuestra aplicación, que hemos almacenado localmente con Realm en capítulos anteriores.

Para ello, en el almacén local de la sesión, sobrescribimos el método **getSession** de la clase **LocalSessionDataStore** del paquete **datastore/local** de la capa de datos. Para almacenar la sesión de usuario utilizamos la instancia del contenedor de Realm, donde buscaremos todos los registros de los usuarios validados en la app, ordenados por el campo fecha de último acceso, como podemos ver en el método **findAllSorted**.

```
@Override public User getSession() { UserSession userSession = null;
    try {
                userSession = myUserRealm.where(UserSession.class)
                        .findAllSorted("LastAccess", Sort.DESCENDING)
                        .first(null);
                return UserToUserSession.Create(userSession);
} catch (Exception error) { return null; }
finally { myUserRealm.close(); }
}
```

Definimos el método de creación de opiniones en la interfaz **IOpinionDataStore**, que deben sobrescribir los almacenes de datos remoto y local de las opiniones. Nosotros, sin embargo, únicamente implementaremos el remoto.

```
Observable<Opinion> createOpinion(User user, int hotelId, Opinion opinion);
```

Implementemos entonces el método que acabamos de definir, sobrescribiéndolo en el acceso a datos remoto de las opiniones **RemoteOpinionDataStore** del paquete **datastore/remote** de la capa de datos.

Invocaremos el método del servicio responsable de realizar una actualización de registro remoto, es decir, una petición de tipo PUT a través de Retrofit. En caso satisfactorio, obtendremos la opinión o comentario que acabamos de almacenar en Firebase.

```
@Override public Observable<Opinion> createOpinion(User user, int
hotelId  , Opinion opinion) {
return Observable.create(emitter -> {
        try { Call<Opinion> call = iOpinionService.createOpinion
(hotelId, user.getUserId()
                , user.getAuthToken(), opinion);
                    Response<Opinion> response = call.execute();
                if (response.code() == 200) {
                        Opinion opinions = response.body();
                        emitter.onNext(opinions);
                        emitter.onComplete();
                } else emitter.onError(new Exception(response.
errorBody().toString()));
        } catch (Exception e) {   emitter.onError(e);       }
});
}
```

Sobrescribimos, por tanto, el método **createOpinion** en la clase **OpinionRepositoryImplementation**, responsable de implementar el repositorio de opiniones. Será aquí donde inicialmente obtengamos de forma local el último usuario validado que tenemos almacenado, y utilizaremos su información con el fin de obtener el UID y el token para poder crear la opinion remotamente en Firebase.

```
@Override public Observable<Opinion> createOpinion(int hotelId
, Opinion opinion) {
        User user = sessionDataStoreFactory.Local().getSession();
        return opinionDataStoreFactory.Remote().createOpinion(user,
hotelId, opinion);
}
```

Ya en la capa de presentación, creamos la vista **IOpinionView** con el método responsable de informar a la interfaz de usuario de que la opinión se ha almacenado satisfactoriamente.

```
public interface IOpinionView extends IDataView    {       void
opinionCreated(Opinion opinion);   }
```

Definimos el observador **CreateOpinionObserver**, responsable de escuchar cambios de estado en el almacenamiento de opiniones. En caso de éxito, informaremos a la vista mediante el método **opinionCreated** para que cierre la actividad.

```
public class CreateOpinionObserver extends DisposableObserver<Opinion> {
@Override public void onNext(Opinion value) {      iOpinionView.
opinionCreated(value);    }
}
```

Una vez creada la vista, el observador y el caso de uso, ya podemos definir el presentador **OpinionPresenter**, que usaremos en este caso para ejecutar el caso de uso de almacenar opiniones, pasándole el identificador de hotel y la opinión.

```
public class OpinionPresenter implements IPresenter {
@Inject public OpinionPresenter(IOpinionView iOpinionView
                                , CreateOpinionUseCase
createOpinionUseCase)
                {       this.iOpinionView = iOpinionView;
                        this.createOpinionUseCase = createOpinion
UseCase;
                }
public void createOpinion(int hotelId, Opinion opinion)
{
        createOpinionUseCase.implementUseCase(
                        new CreateOpinionObserver(iOpinionView),
                        OpinionParameters.Parameters.Create(hotelId,
opinion));
}
}
```

Creamos la actividad **OpinionActivity**, que implementará la interfaz de la vista **IOpinionView**. Su layout se compone simplemente de un EditText para introducir el comentario, un RatingBar para asignar la valoración y, por último, un icono de guardado en el menú de la actividad. En esta actividad, inyectaremos como siempre su presentador **OpinionPresenter** y, en su método **onCreate**, inyectaremos la actividad a Dagger e inicializaremos Butterknife.

Introducimos nuestro comentario en la interfaz de usuario que queremos almacenar, así como la puntuación. Tras pulsar sobre el botón **Guardar** del menú,

invocaremos al método del presentador, responsable de guardar este comentario y valoración en Firebase.

```
@Override public boolean onOptionsItemSelected(MenuItem item) {
int id = item.getItemId();
if (id == R.id.menu_save) {
        Opinion opinion = Factory.OpinionFactory.Create(Math.
round(opinion_rating.getRating())
                                          , opinion_message.getText().
toString(), new Date() );
        opinionPresenter.createOpinion(hotelId, opinion);
        return true;
}
return super.onOptionsItemSelected(item);
}
```

Tras almacenar con éxito el comentario del usuario recibiremos, por el método sobrescrito de la vista **opinionCreated**, la opinión que acabamos de guardar, de tal forma que cerraremos esta actividad y volveremos a la del listado de opiniones.

```
@Override public void opinionCreated(Opinion opinion) {onBackPressed();
}
```

Creamos el módulo **OpinionModule** para proporcionar a Dagger el presentador y la vista.

```
@Module public abstract class OpinionModule {
@Provides static OpinionPresenter provideOpinionPresenter(
        IOpinionView iOpinionView, CreateOpinionUseCase
createOpinionUseCase)     {        return new OpinionPresenter(iOpinion
View, createOpinionUseCase);       }
@Binds abstract IOpinionView provideOpinionView(OpinionActivity
opinionActivity);
}
```

Por último proporcionamos, en **BuildersModule**, la actividad que acabamos de crear y su módulo.

```
@PerActivity @ContributesAndroidInjector(modules = OpinionModule.class)
abstract OpinionActivity contributeOpinionActivity();
```

Al ejecutar la app con una sesión abierta y seleccionar un hotel, veremos el listado de opiniones que han dejado los usuarios sobre este. De modo que, si pulsamos el botón de añadir nuevo comentario, se abrirá la actividad de introducir nuevo comentario y se podrá dejar una opinión (Figura UC16-1). Tras guardarla, volveremos a la pantalla anterior (Figura UC16-2).

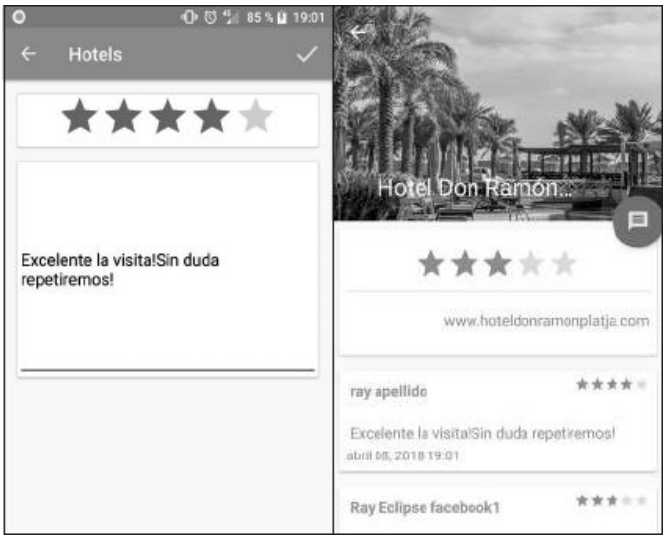


Fig. UC16-1. Guardar opinión. Fig. UC16-2. Opinión guardada.

Aplicación completada - Conclusiones

Llegados a este punto, podemos observar cómo, a la hora de implementar cada caso de uso, hemos seguido un patrón de comportamiento. Hemos empezado por la capa de dominio, la capa más interna del sistema y, por tanto, de la que dependerán las capas más externas. Hemos seguido por la capa de acceso a datos locales o remotos y, por último, por la capa de presentación.

Hacerlo de esta forma nos permite cambiar la forma de acceso remoto, mediante Retrofit, a cualquier otro que aparezca en el mercado o, en el caso de almacenamiento local, quitar nuestra dependencia con Realm y utilizar SQLite o cualquier otro que elijamos. Nosotros hemos seleccionado estos por su rápida ayuda a la implementación, ya que nos proporcionan unas herramientas muy potentes.

Como buenas prácticas, inicialmente deberíamos definir en Trello, o cualquier otro sistema de gestión de tareas, aquellos casos de uso que queremos que nuestra aplicación implemente. De esta forma, tendremos en todo momento el control de lo que queremos hacer.

Una vez que hemos definido las tareas, podemos hacer dos cosas: asignarlas a miembros del equipo de desarrollo, según sus fortalezas, o bien no asignarlas a nadie en el caso de que todos los miembros del equipo tengn similares fortalezas y puedan realizar cualquiera de las tareas. En este último caso, cada miembro deberá elegir una tarea, asignarla a su nombre y colocar este en la columna que indica que lo está realizando. Posteriormente, cuando finalice la tarea, deberá moverla a la columna de tareas terminadas para su revisión.

Como norma general, cada vez que finalicemos un caso de uso, deberíamos subir el código al repositorio para no perder el trabajo realizado. Pueden realizarse subidas intermedias, si así lo requerimos, aunque siempre al finalizar una funcionalidad.

La aplicación Android que acabamos de empezar contiene todos los elementos que podemos encontrarnos a la hora de abordar un desarrollo Android en el ámbito profesional.

En esta aplicación no hemos abordado el desarrollo propio de una API, ni de la base de datos, ni del sistema del almacenamiento de ficheros, ni hemos desarrollado un sistema de autenticación de usuarios porque ya nos lo ha proporcionado Firebase. Todos estos desarrollos adicionales suele realizarlos la empresa; nuestra app únicamente tiene que consumirlos. No obstante, en caso de que nuestra app requiera alguno de estos sistemas y no dispongamos de él, deberemos implementarlos, a no ser que nos baste con el backend proporcionado por Firebase.

Si miramos hacia atrás, nuestra app tiene una sección pública y otra privada. Cualquier usuario que abra la aplicación podrá ver el listado de hoteles y los comentarios de cada hotel realizados por los usuarios. También poseemos una sección privada donde, únicamente tras registrarnos en el sistema con alguno de los proveedores proporcionados, podremos crear nuevos comentarios de un hotel, así como cambiar la información de nuestro perfil de usuario.

El patrón utilizado no es el único que puede aplicarse; de hecho, podríamos quitar la programación reactiva de observadores e implementar otro sistema. También podríamos quitar el patrón modelo vista presentador y cambiarlo por cualquier otro. Aunque nuestra recomendación es utilizar el patrón MVP, visto en el libro en combinación con programación reactiva y observadores, ya que nos permite separar las responsabilidades de cada capa y sustituir elementos del sistema fácilmente.

Para finalizar, destacaremos la necesidad de aprender Kotlin, un lenguaje por el que se está apostando en el desarrollo en Android. En capítulos posteriores lo veremos con algún ejemplo teórico y práctico.

PARTE 4: KOTLIN

CAPÍTULO 1:
Kotlin

Kotlin es un lenguaje de programación que surge como alternativa a Java para corregir algunos de los problemas detectados en este último, como pueden ser los **NullPointerException** o la falta de extensibilidad.

Se ejecuta sobre la **Máquina Virtual de Java** y puede compilarse a código fuente de **JavaScript**. Puede interoperar con código Java y sus librerías; por ello, cualquier código escrito en Java puede usarse desde Kotlin.

Su nombre procede de la isla de Kotlin, ubicada cerca de San Petersburgo, que es donde fue desarrollado en 2010 por desarrolladores de **JetBrains** (responsables de **IntelliJ**).

Kotlin se ha convertido en un lenguaje de **Android** gracias a que **Google** y **JetBrains** han reforzado su colaboración y **Google** se ha posicionado favorablemente.

Debido al alto nivel de interoperabilidad entre Kotlin y Java, es posible usar la mayoría de las librerías y frameworks de Java en proyectos Kotlin.

La curva de aprendizaje es sencilla y además es posible convertir código Java a Kotlin, con lo que se facilita poder establecer la correspondencia entre sentencias de ambos lenguajes.

Algunas de las características destacables de su sintaxis son las siguientes:

- La declaración de variables y listas de parámetros en Kotlin describen el tipo de dato después del identificador seguido de dos puntos. Por ejemplo:

```
var miVariable: Int = 5
```

- Los puntos y comas son opcionales como final de sentencia, ya que el compilador deduce que un salto de línea implica el fin de una sentencia en muchos casos.
- Posee los métodos y clases clásicas usadas en la programación orientada a objetos.
- Soporta programación por procedimientos y el uso de funciones.
- Se utiliza la función **main()** como punto de entrada a un programa Kotlin, a la que se le pasan los argumentos de entrada desde línea de comandos mediante un array.

```
fun main(args : Array<String>) {
…
}
```

- Distingue entre tipos **nullables** y **no-nullables** (el valor puede ser nulo o no), que se declaran con un '**?**' después del tipo:

```
fun miFuncion(miVariable : String?) {
…
}
```

Android Studio soporta Kotlin a partir de la versión **3.0.** Para usar Kotlin en un nuevo proyecto, basta con marcar el check que incluye soporte a Kotlin durante su creación.

Si deseas realizar algunos ejemplos, puedes descargar el IDE **IntelliJ IDEA** de la siguiente URL:

https://www.jetbrains.com/idea/

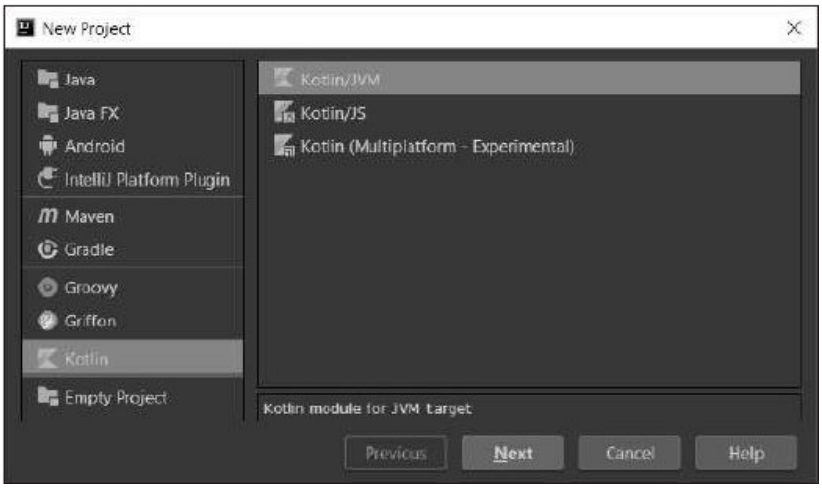
La instalación es muy sencilla e intuitiva; prácticamente solo tienes que aceptar los valores propuestos por defecto.

Veamos a continuación algunos temas relacionados con el lenguaje.

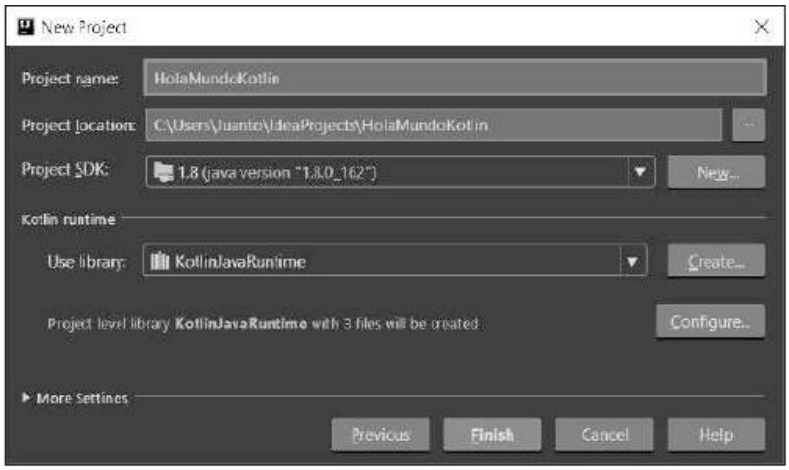
Hola Mundo (IntelliJ IDEA)

Si has descargado el IDE **IntelliJ IDEA**, empezaremos por arrancar el IDE y crear un proyecto seleccionando la opción **Create New Project**:

En la siguiente pantalla, seleccionaremos **Kotlin** como tipo de proyecto y **Kotlin/JVM** en la parte de la derecha:

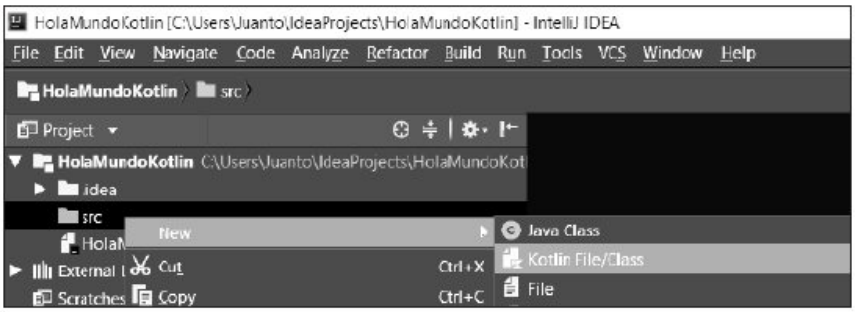


Pulsaremos sobre **Next** e introduciremos "HolaMundoKotlin" en el campo **Project name**:

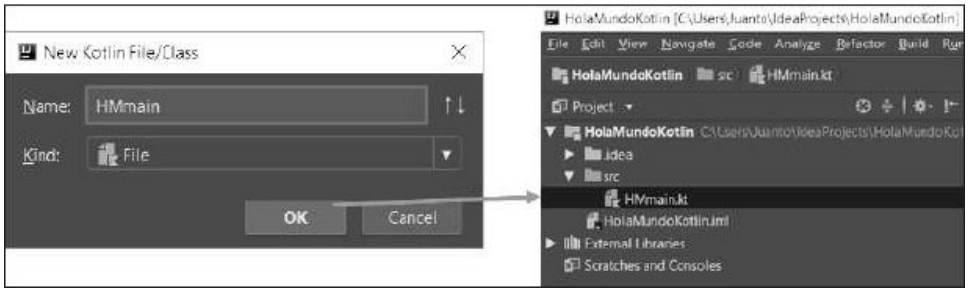


Pulsaremos sobre **Finish** y observaremos cómo se abre el proyecto en nuestro **IDE**.

Seguidamente, crearemos un fichero seleccionando la carpeta **src** de nuestro proyecto y, con el botón derecho del ratón, seleccionaremos **New - Kotlin File/ Class**:



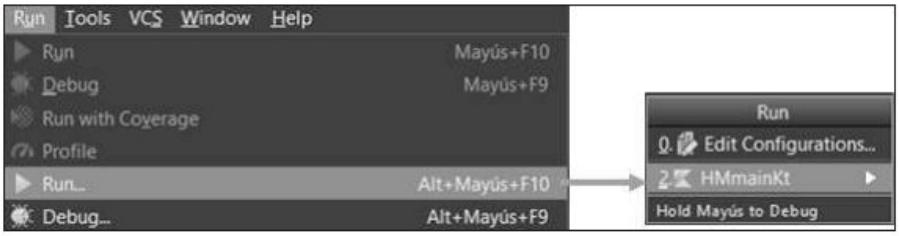
En la ventana que aparece introduciremos "HMmain" como **Name** y pulsaremos sobre **OK**:



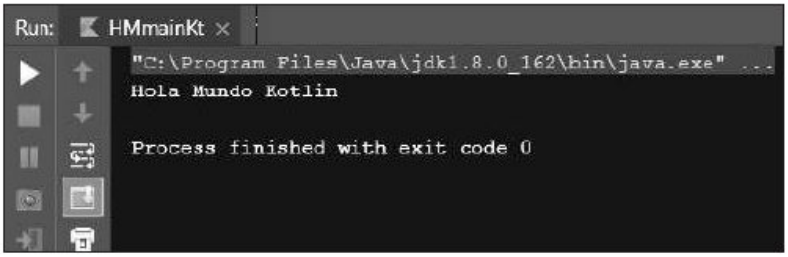
En este punto ya disponemos de nuestro fichero **HMmain.kt**, sobre el que definiremos nuestro punto de entrada a la aplicación gracias a la función **main()**, y simplemente imprimiremos el string "Hola Mundo Kotlin" tal y como se muestra a continuación:

```
fun main(args: Array<String>) {
    println("Hola Mundo Kotlin")
}
```

Ahora solo nos queda ejecutar la aplicación; para ello seleccionamos la opción **Run** del menú, de nuevo **Run...** y **HMmainKt**:



En la ventana de resultados veremos lo siguiente:



Variables y tipos

Las variables se declaran usando la palabra clave **var** seguida del nombre de la variable, dos puntos (:) y su **tipo**. Opcionalmente podemos inicializarla con un contenido. Por ejemplo:

```
var miVariable: Int = 5
```

También hay que indicar que las variables pueden ser **mutables** e **inmutables** dependiendo de si se pueden modificar o no. Es equivalente a la marca de '**final**' que usamos en Java.

Para indicar que una variable es inmutable, en lugar de **var** usaremos **val**. Siguiendo el ejemplo anterior:

```
val miVariable: Int = 5
```

El tipo de datos puede inferirse dependiendo del valor que asignamos a la variable, con lo que no sería estrictamente necesario especificar el tipo de datos. No obstante, es una buena práctica especificarlo de forma explícita para dar claridad al código.

La siguiente tabla resume los tipos de variables que podemos utilizar:

	Tipo	Observaciones
Numéricas	**Int**	Números naturales entre -2.147.483.647 y 2.147.483.647. **Ejemplo:** var miInt: Int = -123
	Long	Igual que Int pero entre -9.223.372.036.854.775.807 y 9.223.372.036.854.775.807. **Ejemplo:** var miLong: Long = 1234567890
	Float	Números reales que soportan hasta 6 decimales. Se añade una 'f' al final del valor (ej. 1.23f). **Ejemplo:** var miFloat: Float = 5.678f
	Double	Similar a Float y soporta hasta 14 decimales. **Ejemplo:** var miDouble: Double= 1.234567890
Alfanuméricas	**char**	Almacena un carácter. **Ejemplo:** var miChar: Char = 'a'
	String	Almacena una cadena de caracteres y se definen entre comillas dobles. **Ejemplo:** var miString: String = "- Prueba alfanuméricos -"
Booleanas	**Boolean**	Almacena valores true o false. **Ejemplo:** var bFalso: Boolean = false var bVerdadero: Boolean = true

Arrays

Los arrays se definen y utilizan como en otros muchos lenguajes. En definitiva, almacenan un conjunto de datos del mismo tipo que comparten un mismo nombre y a los que se accede a través de un índice o posición que identifica el lugar que ocupan dentro de dicho conjunto. El primer valor se halla en la posición 0; el segundo, en la 1 y así sucesivamente.

Un ejemplo de definición de array sería el siguiente:

```
val estaciones = arrayOf(“Primavera”, “Verano”, “Oto o”, “Invierno”)
```

También podemos definir arrays de la siguiente manera:

```
val amigos = kotlin.arrayOfNulls<String>(3)
amigos[0] = “Juan”
amigos[1] = “Paco”
amigos[2] = “Toni”
println(Arrays.toString(amigos)) // Devuelve [Juan, Paco, Toni]
```

Para el ejemplo anterior, deberemos importar al inicio de nuestro fichero la librería:

```
import java.util.*

fun main(args: Array<String>) {
…
```

Para acceder a una posición de un array, usaremos la función **get()** de la siguiente forma:

```
println(estaciones.get(0))
```

Para obtener la dimensión de un array, disponemos de la propiedad **size**:

```
println(estaciones.size)
```

Uno de los problemas de los arrays es que, una vez definidos, mantienen su tamaño y no hay posibilidad de alterarlo. Sin embargo, sí podemos alterar su contenido mediante la función **set()**:

```
estaciones.set(2, “Autumn”)
```

Para recorrer los arrays, podemos usar diferentes mecanismos, ya que de antemano conocemos su dimensión. No obstante, aunque no nos basemos en su tamaño para realizar un **for**, podemos usar la siguiente expresión para recorrerlos, como se muestra a continuación:

```
for (estacion in estaciones) {
    println(estacion)
}
```

También podemos usar la siguiente expresión para mostrar la posición y el valor:

```
for ((idx, dato) in estaciones.withIndex()) {
    println(“Posici n $idx - dato $dato”)
}
```

Listas

Las listas, al igual que los arrays, son elementos que nos permiten almacenar colecciones de objetos, pero poseen diferencias notables. Mientras que los arrays manejan un número fijo de elementos, las listas mutables pueden ir añadiendo elementos sobre la marcha. Por el contrario, el manejo de la memoria es menos eficiente.

A continuación, mostramos un ejemplo similar al descrito para los arrays, pero adaptado en primer lugar a una lista inmutable:

```
val lestaciones: List<String> = listOf("Primavera", "Verano", "Oto o",
"Invierno")
```

Efectivamente, si imprimimos **lestaciones** tenemos:

```
println(lestaciones) // Imprime: [Primavera, Verano, Oto o, Invierno]
```

Para acceder a una posición concreta de la lista podemos usar la función **get()**, teniendo en cuenta que el índice empieza en el **0**. Por ejemplo:

```
println(lestaciones.get(2)) // Imprime: Oto o
```

También podemos acceder al primer y último elemento usando las funciones **first()** y **last()**:

```
println(lestaciones.first()) // Imprime: Primavera
println(lestaciones.last()) // Imprime: Invierno
```

Si accedemos a la lista mediante un **iterator**, podemos usar un filtro de la siguiente manera:

```
val miColeccion: List<Int> = listOf(1, 3, 6, 9, 12)
val res = miColeccion.filter { it >= 6 }
println("resultado $res )")// Devuelve resultado [6, 9, 12])
```

Las listas mutables nos dan la posibilidad de añadir y eliminar elementos dinámicamente. Un ejemplo de lista mutable podría ser el siguiente:

```
var amigos: MutableList<String> = mutableListOf("Paco", "Miguel", "Toni")
println("Tengo ${amigos.size} amigos: $amigos") // Imprime: Tengo 3
amigos: [Paco, Miguel, Toni]
```

Para añadir un elemento, basta con utilizar la función **add** de la siguiente manera:

```
amigos.add("Tintu")
println("Tengo ${amigos.size} amigos: $amigos") // Imprime: Tengo 4
amigos: [Paco, Miguel, Toni, Tintu]
```

Para eliminar un elemento de una determinada posición usaremos la función **removeAt()**. Por ejemplo:

```
amigos.removeAt(0)
println("Tengo ${amigos.size} amigos: $amigos") // Imprime: Vacia? False
```

Para añadir un elemento en una posición, hay que indicar la posición donde se añadirá. Por ejemplo, para incluir un amigo al inicio de la colección lo haremos de la siguiente manera:

```
amigos.add(0,"Jordi")
println("Tengo ${amigos.size} amigos: $amigos") // Imprime: Tengo
4 amigos: [Jordi, Miguel, Toni, Tintu]
```

Para cambiar el contenido de un elemento usaremos la función **set()**:

```
amigos.set(1,"Sonia")
println("Tengo " + amigos.size+ " amigos: $amigos") // Imprime: Tengo
4 amigos: [Jordi, Sonia, Toni, Tintu]
```

Además de usar la propiedad **size** para saber cuántos elementos tiene una lista, podemos averiguar si esta está vacía o no usando la función **none()**:

```
println("Vacia? ${amigos.none()}")
```

También es posible indicar que nos devuelva un **null**, si el contenido del elemento está vacío, usando la función **elementAtOrNull()** y, en caso de necesitar el primer o último elemento, usaremos **firstOrNull()** y **lastOrNull()** respectivamente. Veamos los siguientes ejemplos:

```
if (amigos.elementAtOrNull(2).isNullOrEmpty()){
    println("Es nulo") // Imprime: Es nulo
}

println("elementAtOrNull (${amigos.elementAtOrNull(2)})") // Imprime:
 elementAtOrNull ()
println("firstOrNull (${amigos.firstOrNull()})") // Imprime: firstOrNull
(Jordi)
println("lastOrNull (${amigos.lastOrNull()})") // Imprime: lastOrNull
(Tintu)
```

Al igual que vimos en los arrays, podemos recorrer las listas de la siguiente manera:

for (amigo **in** amigos) {
  println(amigo)
}
	// Salida
	// Jordi
	// Sonia
	//
	// Tintu

for ((idx, amigo) **in** amigos.*withIndex*()) {
  println("**Posición $**idx **- amigo $**amigo")
}
	// Salida
	// Posición 0 - amigo Jordi
	// Posición 1 - amigo Sonia
	// Posición 2 - amigo
	// Posición 3 - amigo Tintu

Por último, también podemos apoyarnos en el **iterator** para recorrer la lista con **foreach** de la siguiente manera:

```
amigos.forEach {
    println("Amigo ${it}")
}
// Salida
//    Amigo Jordi
//    Amigo Sonia
//    Amigo
//    Amigo Tintu
```

Funciones

Las funciones son bloques de código que pueden ser llamadas tantas veces como sea necesario y, por tanto, permiten reutilizar código, que es uno de los aspectos que siempre perseguimos en la programación.

Se declaran mediante la palabra clave **fun** seguida del nombre de la función, la lista de argumentos de entrada entre paréntesis y, si tiene parámetro de salida, dos puntos (:) y dicho parámetro. El cuerpo de la función se indica entre 2 llaves (**{}**). La sintaxis general es:

```
fun nombreFuncion(nomParam1: tipoParam1, nomParam2: tipoParam2) :
tipoParamSalida{
    ...
    return paramSalida
}
```

Supongamos que definimos una función donde solo saludamos. Por ejemplo:

```
fun saluda(){
    println("Hola Mundo Kotlin")
}
```

Podemos invocarla, por ejemplo, desde nuestra función **main()** de la siguiente manera:

```
fun main(args : Array<String>) {
    saluda() // Imprime: Hola Mundo Kotlin
}
```

Al igual que ocurre en otros lenguajes, podemos "sobrepasar" la función, simplemente definiéndola con argumentos de entrada diferentes. De esta forma, al variar la signatura, no hay ningún tipo de conflicto en cuanto a duplicidad de nombres de función. Por ejemplo, podemos pasar el nombre como argumento en la llamada, tal y como se muestra a continuación:

```
fun main(args : Array<String>) {
    saluda("Juan") // Imprime: Hola Mundo Juan
}
fun saluda(nombre: String){
    println("Hola Mundo $nombre")
}
```

Por último, fabricaremos una función que nos devuelva el texto correspondiente al saludo que queremos realizar. Por ejemplo:

```
fun main(args : Array<String>) {
    var res = saludaV("Paco")
    println(res) // Imprime: Hola Mundo Paco
}
fun saludaV(nombre: String): String{
    return "Hola Mundo $nombre"
}
```

Efectivamente, vemos que la función **saludaV** devuelve un **String** que se asigna a una variable para su posterior impresión.

Si unificamos los tres ejemplos, tendremos que, en un único fichero, se hallará el siguiente código:

```
fun main(args : Array<String>) {
    saluda() // Imprime: Hola Mundo Kotlin
    saluda("Juan") // Imprime: Hola Mundo Juan
    var res = saludaV("Paco")
    println(res) // Imprime: Hola Mundo Paco
}
fun saluda(){
    println("Hola Mundo Kotlin")
}
fun saluda(nombre: String){
    println("Hola Mundo $nombre")
}
fun saludaV(nombre: String): String{
    return "Hola Mundo $nombre"
}
```

Colecciones y funciones

Las operaciones sobre colecciones de datos son de los elementos más potentes del lenguaje y que lo hacen verdaderamente interesante.

Podremos probar estos ejemplos desde Android Studio mediante la entrada de menú **Tools > Kotlin > Kotlin REPL.** Esta línea de comandos interactiva nos permite probar las características del lenguaje.

Podemos definir una lista inmutable de enteros de la siguiente manera:

```
val myList = listOf(2,3,4,5,6,7,8)
```

Sobre esta veremos algunos de los operadores que podemos aplicarle.

El operador **any** comprueba elemento a elemento si alguno cumple el predicado; si lo cumple, devuelve true. En caso contrario, false:

```
myList.any { it % 2 == 0} devuelve true
```

El operador **all** comprueba elemento a elemento si todos cumplen el predicado; si lo cumplen, devuelve true; de lo contrario, false:

```
myList.all { it % 2 == 0} devuelve false
```

El operador **none** comprueba elemento a elemento si ninguno cumple el predicado; si no lo cumplen, devuelve true. En caso contrario, false:

```
myList.none { it % 2 == 0} devuelve false
```

El operador **count** comprueba elemento a elemento si cumplen el predicado y devuelve el total de elementos que lo cumplen:

```
myList.count { it % 2 == 0} devuelve 4
```

El operador **min** devuelve el mínimo valor de la lista:

```
myList.min() devuelve 2
```

El operador **max** devuelve el máximo valor de la lista:

```
myList.max() devuelve 8
```

El operador **drop** devuelve una lista sin los *n* primeros elementos:

```
myList.drop(2) devuelve {4,5,6,7,8}
```

El operador **filter** devuelve una lista de los elementos de la lista original que cumplen el predicado:

```
myList.filter{ it % 2 == 0 } devuelve {2,4,6,8}
```

El operador **filterNot** devuelve una lista de los elementos de la lista original que no cumplen el predicado:

```
myList.filterNot{ it % 2 == 0 } devuelve {3,5,7}
```

El operador **slice** obtiene de la lista los elementos en los índices que se le pasan y crea otra lista con los resultados:

```
myList.slice(listOf(2,4)) devuelve {4,6}
```

El operador **take** devuelve una lista con los *n* primeros elementos:

```
myList.take(3) devuelve {2,3,4}
```

El operador **take** devuelve una lista con los *n* últimos elementos:

```
myList.takeLast(2) devuelve {7,8}
```

El operador **map** devuelve una lista transformada por la función que se le pasa como predicado:

```
myList.map{ it + 1 } devuelve {3,4,5,6,7,8,9}
```

If-else-when

Las instrucciones condicionales son un clásico en la inmensa mayoría de lenguajes y, en el caso de Kotlin, su utilización es idéntica.

El caso más simple es el uso de **if**. Lo que hacemos con **if** es verificar si una condición se cumple. En caso de que se cumpla, se ejecutarán las instrucciones asociadas al bloque **if**. Por ejemplo:

```
fun main(args : Array<String>) {
    var cierto = true
    if (cierto)
        println(“Es cierto”) // Imprime: Es cierto
}
```

Si el bloque **if** contiene varias sentencias, podemos definir dicho bloque entre llaves (**{}**) de la siguiente manera:

```
if (cierto) {
    println(“Es cierto”) // Imprime: Es cierto
}
```

La condición total por evaluar puede estar formada a su vez por diferentes condiciones encadenadas por operadores **AND** (**&&**) y/o operadores **OR** (**||**) que cumplen la siguiente sintaxis:

```
if (condicion1 && condicion2 || condicion3)
```

Por ejemplo, si quisiéramos permitir el acceso a personas con más de 16 años que tengan entrada o simplemente a las que tengan invitación independientemente de la edad, podríamos indicar algo parecido a lo siguiente:

```
var edad: Int = 20
var tieneEntrada = true
var tieneInvitacion = true
if (edad >16 && tieneEntrada || tieneInvitacion)
    println("Acceso permitido") // Imprime: Acceso permitido
```

Para negar una condición, basta con anteponer el símbolo de admiración al inicio de la condición:

```
cierto=false
if (!cierto)
    println("Es falso") // Imprime: Es falso
```

La condición **else** se cumple cuando no se cumple la condición del **if**. En el siguiente ejemplo, se indica que "si no es cierto", es que "es falso":

```
cierto=false
if (cierto) {
    println("Es cierto") // Imprime: Es cierto
}else {
    println("Es falso") // Imprime: Es falso
}
```

Podemos anidar los **if-else** tantas veces como necesitemos. Veamos el siguiente ejemplo:

```
var tipo: Int = 3
if (tipo==1) {
    println("Es tipo 1") // Imprime: Es tipo 1
}else if (tipo==2) {
    println("Es tipo 2") // Imprime: Es tipo 2
}else if (tipo==3) {
    println("Es tipo 3") // Imprime: Es tipo 3
}else{
    println("Tipo desconocido") // Imprime: Tipo desconocido
}
```

Si hacemos que el valor de tipo sea **5**, el resultado que obtendremos será "Tipo desconocido".

Una forma de hacer lo mismo que en el caso anterior, pero tal vez más elegante, sería utilizar la sentencia **when** de la siguiente manera:

```
var tipo: Int = 3
when (tipo) {
    1 -> print("Es tipo 1")
    2 -> print("Es tipo 2")
    3 -> print("Es tipo 3")
    else -> {
        print("Tipo desconocido")
    }}
```

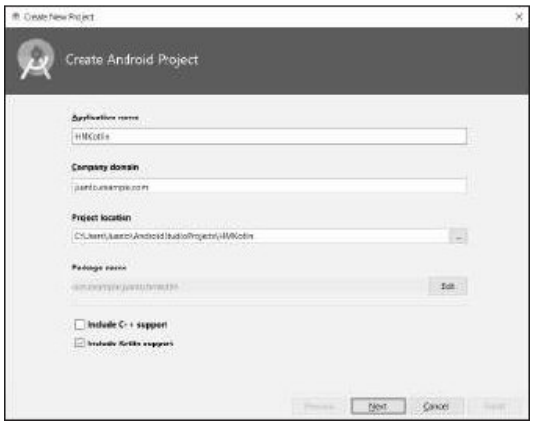
En este ejemplo, el resultado será: “Es tipo 3”.

Hola Mundo en Kotlin

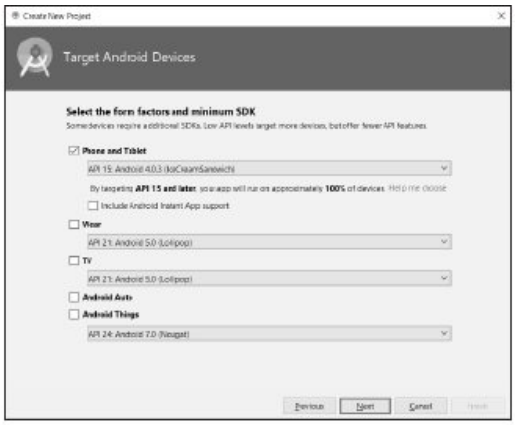
A continuación, vamos a realizar una sencilla aplicación en Android, utilizando Kotlin, que nos permita mostrar el texto “Hola Mundo Kotlin”.

Para ello, ejecutaremos **Android Studio** (recuerda utilizar una versión igual o superior a la 3.0) y seleccionaremos **Start a new Android Studio project**:

En el siguiente cuadro de diálogo, indicaremos el nombre de la app (por ejemplo, **HMKotlin**) y dejaremos marcado el check asociado a **Include Kotlin suppor**t:

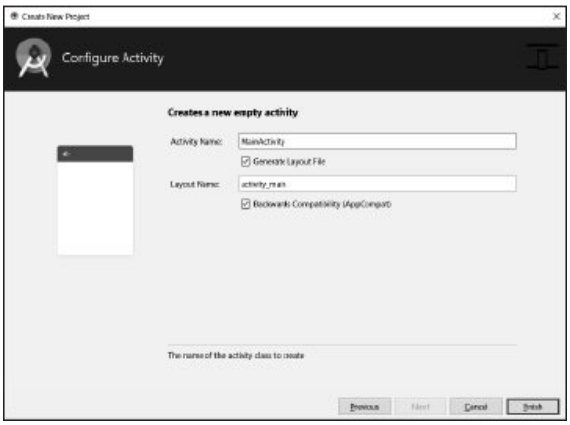


Al pulsar sobre **Next** aparecerá el siguiente cuadro de diálogo:



Aceptamos los valores propuestos por defecto y de nuevo pulsamos sobre **Next**:

En este punto, seleccionamos una actividad vacía (**Empty Activity**) y pulsamos sobre **Next**:



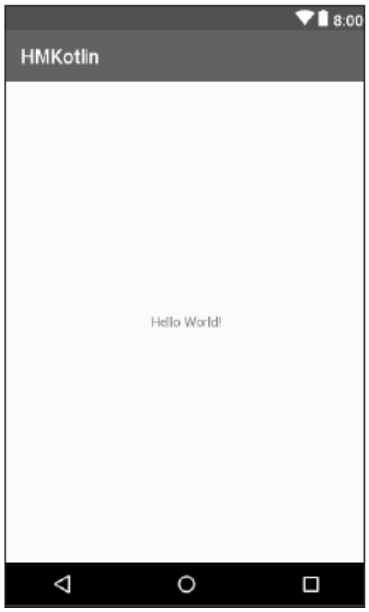
En este cuadro, simplemente pulsaremos sobre **Finish** y observaremos cómo en breve aparece el IDE con nuestra nueva aplicación. Observaremos también cómo se muestra nuestra Activity principal contenida en el fichero **MainActivity.kt**:

```
package com.example.juan.hmkotlin

import android.support.v7.app.AppCompatActivity
import android.os.Bundle

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}
```

Según el **company domain** que hayamos definido durante la creación del proyecto, el nombre del **package** variará; por eso, en mi caso, se llama **com.example.juan.hmkotlin**. En tu caso, esto variará según el valor que hayas dejado en ese punto.

Si observamos el fichero **activity_main.xml**, veremos que, por defecto, ya incluye un **TextView** con el texto "Hello World!":

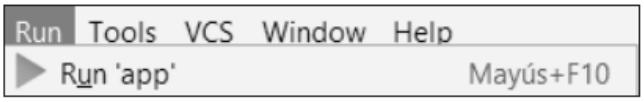
```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout xmlns:android=
"http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="com.example.juan.hmkotlin.MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</android.support.constraint.ConstraintLayout>
```



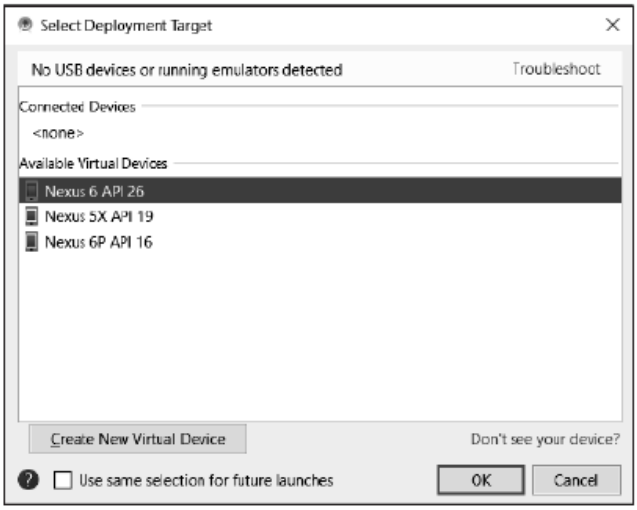
Podemos cambiar su contenido a:

```
android:text="Hola Mundo Kotlin"
```

A partir de este punto, ya podríamos ejecutar la aplicación pulsando la opción **Run** del menú **Run**:



Nos invitará a seleccionar el dispositivo sobre el cual queremos ejecutar la **app**; si aún no hemos definido ninguno, será un buen momento para hacerlo. En mi caso, voy a utilizar el **Nexus 6 API 26**:



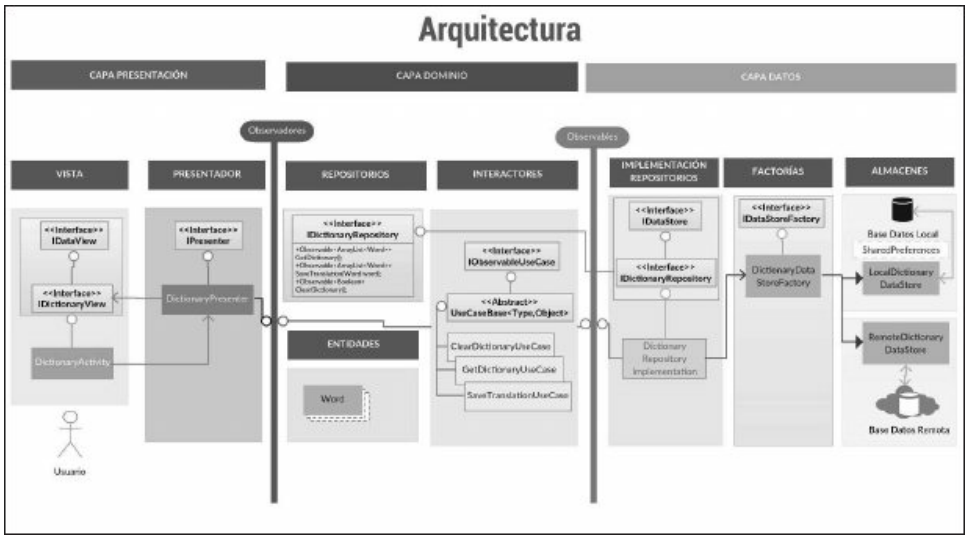
Al pulsar sobre **OK**, se inicia el proceso de despliegue de la **app** sobre el dispositivo y, seguidamente, se ejecutará sobre él y dará el siguiente resultado:

Evidentemente, a partir de aquí podríamos continuar complicando el ejemplo al añadir más elementos a la interfaz y dando más funcionalidad a la app, pero el objetivo de este capítulo era simplemente dar una pincelada sobre Kotlin y su posible utilización para programar en Android. Por tanto, te invitamos a que continúes profundizando sobre el uso de Kotlin en Android Studio. Es posible convertir ficheros Java a ficheros Kotlin usando la opción del menú principal: **Code - Convert Java File to Kotlin File**.

CAPÍTULO 2:
Kotlin MVP

Hasta ahora hemos dado los primeros pasos en este nuevo lenguaje y hemos construido nuestra primera aplicación: Hola Mundo. Llegados a este punto daremos un paso más y crearemos una aplicación completa basándonos en el proyecto Clean MVP de capítulos anteriores.

Vamos a utilizar este proyecto como referencia de lo que queremos construir: utilizaremos la misma estructura de capas (presentación, modelo, datos), los mismos patrones (MVP, Repository, observadores), las mismas librerías (RxJava, Dagger 2), con la diferencia de que lo codificaremos todo en Kotlin en lugar de Java.



Recuperamos esta imagen, que nos servirá como recordatorio de la estructura del proyecto que vamos a crear:

Seguimos los pasos que ya hemos hecho en anteriores capítulos para crear un nuevo proyecto que llamaremos **cleanmvpdagger211kotlin** y también la separación en los tres módulos de la aplicación (**app**, **model** y **data**) como hicimos con la aplicación de ejemplo **CleanMVPDagger211.**

A partir de aquí podríamos empezar creando nuestra estructura de módulos y componentes para dar soporte a la inyección de dependencias, nuestra actividad principal (**DictionaryActivity.kt**), las clases **DictionaryPresenter.kt** y **IDictionaryView.kt** para la implementación del patrón MVP, las clases del dominio en la capa modelo, los casos de uso, los repositorios y las diferentes implementaciones, etc.Sin embargo, nuestro objetivo no es repetir en este capítulo lo que ya hemos visto en anteriores, sino conocer Kotlin y ver las diferencias de codificación con Java. Por esta razón nos pareció buena idea replicar el proyecto que ya conocemos, volver a codificarlo en Kotlin y ponerlo a la disposición del lector en los ejemplos proporcionados por la editorial para tenerlo como referencia y comentar en este capítulo aquellas partes más interesantes.

Módulo del dominio

Empezaremos por la parte core de nuestro dominio, la capa modelo que hemos definido dentro del módulo **model**.

El clásico objeto **POJO** de Java es sustituido por lo que en Kotlin se denomina **data class**; es una clase que solo mantiene un estado y sin operaciones más allá de las que modifiquen el propio estado. Para definir esta data class basta con que definas su nombre y el de sus atributos; el resto de código nos lo ahorramos, ya que nos lo proporciona Kotlin.

Así, nuestra clase Word anteriormente sería algo así:

```
public class Word {
    private String Term;
    private String Translation;

    public Word(String term, String translation) {
        setTerm(term);
        setTranslation(translation);
    }

    public String getTerm() {
        return Term;
    }

    public void setTerm(String term) {
        Term = term;
    }

    public String getTranslation() {
        return Translation;
    }

    public void setTranslation(String translation) {
        Translation = translation;
    }

    @Override
    public String toString() {
        return "Word{" +
                "Term='" + Term + '\'' +
                ", Translation='" + Translation + '\'' +
                '}';
    }
}
```

Y pasaría a convertirse en algo como esto:

```
data class Word(val term:String, val translation:String)
```

Bastante más resumido y con la misma información, sabemos que tenemos un objeto Word que tiene como atributos término y su traducción.

Además de esta clase hemos creado otra, a diferencia del proyecto del que partimos, que se llama **Dictionay.kt**. Esta es otra clase del dominio que representa el diccionario y almacena un listado de palabras.

```
data class Dictionary (val wordList:List<Word>)
```

Siguiendo en el módulo **model** tendríamos la traducción simple a Kotlin de nuestro caso de uso base, que pasaría de algo como esto:

```
public abstract class UseCaseBase<Type,Object> implements
IObservableUseCase
{
    ArrayList<DisposableObserver<Type>> observers=new ArrayList<>();


    public void createUseCase(Observable observable,
DisposableObserver<Type> observer,Scheduler schedulerThread)
    {
        observable.subscribeOn(Schedulers.io())
                .observeOn(schedulerThread)
                .subscribeWith(observer);
        subscribe(observer);
    }
    public abstract Observable<Type> implementUseCase
(DisposableObserver observer, Object object);

    @Override
    public void subscribe(DisposableObserver observer) {
        if(!observers.contains(observer)) observers.add(observer);
    }

    @Override
    public void cancelSubscription() {
        for(DisposableObserver observer : observers)
        {
            if(observer!=null && observers.contains(observer)) {
                observer.dispose();
                observers.remove(observer);
            }
        }
    }
}
```

a algo como esto:

```
abstract class UseCaseBase<Type, Params>:IObservableUseCase<Type> {
    private var observers:MutableList<DisposableObserver<Type>> =
mutableListOf()

    fun createUseCase(observable: Observable<Type>,
observer:DisposableObserver<Type>, schedulerThread:Scheduler){
        observable.subscribeOn(Schedulers.io())
                .observeOn(schedulerThread)
                .subscribeWith(observer)
        subscribe(observer)
    }

    override fun subscribe(observer:DisposableObserver<Type>) {
        if(!observers.contains(observer)) observers.add(observer)
    }

    override fun cancelSubscription() {
        observers.map {
            it.dispose()
            observers.remove(it)
        }
    }

    abstract fun implementUseCase(observer:DisposableObserver<Type>,
params:Params):Observable<Type>
}
```

Hay que destacar en esta clase dos detalles:

El primero ver cómo se expresa la herencia en Java, en la que usamos la palabra **implements**

```
public abstract class UseCaseBase<Type,Object> implements
IObservableUseCase
```

a diferencia de Kotlin, en la que solamente lo indicamos como un tipo, que como en toda variable se indica después de los dos puntos:**abstract class** UseCase-Base<Type, Params>:IObservableUseCase<Type>

El segundo es la colección de observadores que mantenemos en Java:

```
ArrayList<DisposableObserver<Type>> observers=new ArrayList<>();
```

Pasa a ser una colección **mutable**, algo que tenemos que indicar específicamente, ya que en Kotlin las colecciones por defecto son **inmutables**. Esto lo podemos ver en la forma en la que definimos con la palabra reservada **var** y la inicialización de la lista:

```
private var observers:MutableList<DisposableObserver<Type>>
= mutableListOf()
```

Si vamos a la definición de un caso de uso concreto, podremos ver:

```
class GetDictionaryUseCase @Inject constructor(private val
iDictionaryRepository:IDictionaryRepository, private val
schedulerThread:Scheduler) : UseCaseBase<Dictionary, Void?>() {

    override fun implementUseCase(observer: DisposableObserver
<Dictionary>, params: Void?): Observable<Dictionary> {
        val dictionaryObs = iDictionaryRepository.getDictionary()
        this.createUseCase(dictionaryObs, observer, schedulerThread)
        return dictionaryObs
    }
}
```

En este caso tenemos otra característica del lenguaje con respecto a los constructores, se indica en la definición de la clase el único constructor:

```
constructor(private val iDictionaryRepository:IDictionaryRepository,
private val schedulerThread:Scheduler)
```

Si tuviéramos más de uno, podríamos hacerlo dentro del cuerpo de la clase:

```
class GetDictionaryUseCase : UseCaseBase<Dictionary, Void?>() {

    @Inject
    constructor(iDictionaryRepository:IDictionaryRepository,
schedulerThread:Scheduler){
        val name = "GetDictionaryUseCase";
    }
    constructor(iDictionaryRepository:IDictionaryRepository){
        val name = "GetDictionaryUseCase";
    }
}
```

Módulo de datos

Pasamos ahora al módulo **data**, en el que se implementa el patrón repositorio y se obtienen los datos que vamos a manejar en la aplicación.

Repository

Atendiendo a la implementación del repositorio en el proyecto que hemos visto tendríamos:

```
public class DictionaryRepositoryImplementation implements
IDictionaryRepository {

    DictionaryDataStoreFactory dictionaryDataStoreFactory;

    @Inject
    public DictionaryRepositoryImplementation(DictionaryDataStore
Factory dictionaryDataStoreFactory)
    {
        this.dictionaryDataStoreFactory=dictionaryDataStoreFactory;
    }

    @Override
    public Observable<ArrayList<Word>> GetDictionary() {
        return dictionaryDataStoreFactory.Local().GetDictionary();
    }

    @Override
    public Observable<ArrayList<Word>> SaveTranslation(Word word) {
        return dictionaryDataStoreFactory.Local().SaveTranslation(word);
    }

    @Override
    public Observable<Boolean> ClearDictionary() {
        return dictionaryDataStoreFactory.Local().ClearDictionary();
    }
}
```

Su traducción a Kotlin sería:

```
class DictionaryRepositoryImplementation(private val
dictionaryDataStoreFactory:DictionaryDataStoreFactory):IDictionary
Repository {

    override fun getDictionary(): Observable<Dictionary> {
        return dictionaryDataStoreFactory.local().getDictionary()
    }

    override fun saveTranslation(word: Word): Observable<Dictionary> {
        return dictionaryDataStoreFactory.local().saveTranslation(word)
    }

    override fun clearDictionary(): Observable<Boolean> {
        return dictionaryDataStoreFactory.local().clearDictionary()
    }
}
```

Con la diferencia de que en este caso, como hemos mencionado anteriormente, sustituiremos el Observable<List<Word>> por el que envuelve a nuestra clase diccionario Observable<Dictionary>.

Guardando en local

Una vez vista nuestra definición del repositorio centrémonos en la implementación del repositorio local.

En este caso almacenamos las palabras de nuestro diccionario en memoria y las persistimos utilizando el mecanismo de preferencias que nos proporciona Android.

Centrándonos en el caso de uso de guardar una nueva palabra con su término y traducción teníamos esto en nuestro proyecto original:

```
@Override
public Observable<ArrayList<Word>> SaveTranslation(Word word) {
    return Observable
            .create(emitter ->
            {
                try {
                    Type listOfWordsType = new TypeToken<Array
List<Word>>() { }.getType();

                    ArrayList<Word> records = getRecords();

                    word.setTerm(word.getTerm().toUpperCase());

                    int indexWordInRecords = existsWord(records, word);
                    if (indexWordInRecords > -1) {
                        records.set(indexWordInRecords, word);
                    } else if (!word.getTerm().isEmpty()) {

                        records.add(word);
                    }
                    records = SortRecords(records);

                    Gson gson = new Gson();
                    sharedPreferences.edit().putString
(DICTIONARYRECORDS, gson.toJson(records, listOfWordsType)).apply();

                    emitter.onNext(records);
                    emitter.onComplete();

                } catch (Exception e) {
                    emitter.onError(e);
                }
            });
}
```

Y tenemos esto en nuestro proyecto con Kotlin:

```
override fun saveTranslation(word: Word): Observable<Dictionary> {
 return Observable.create({
    val dictionary = getDictionaryFromMemory()
    if(dictionary.wordList.any {it.term.equals(word.term, true)}) {
      val modifyDictionaryTerm: (Word, String) -> Word = {
         w, term
          -> if (term.equals(w.term, true))
```

```
            w.copy(translation = word.translation)
        else w
        }

        val dictCopy = dictionary.copy(wordList = dictionary.wordList.map
    { modifyDictionaryTerm(it, word.term) })
                    prefHelper.customPrefs(DICTIONARYRECORDS).edit().
putString(DICTIONARYRECORDS, Gson().toJson(dictCopy)).apply()
                    it.onNext(dictionary.copy(wordList = dictionary.
wordList.map { modifyDictionaryTerm(it, word.term) }))
                    it.onComplete()
                }
                else {
                    val dictCopy = dictionary.copy(wordList = dictionary.
wordList.plus(word))
                    prefHelper.customPrefs(DICTIONARYRECORDS).edit().
putString(DICTIONARYRECORDS, Gson().toJson(dictCopy)).apply()
                    it.onNext(dictCopy)
                    it.onComplete()
                }
        })
    }
```

¿Lo ves claro? Seguramente no, así que lo desgranaremos un poco e iremos por partes. Solamente adelantarte que, aunque esta no sea la mejor implementación, sí que descubre elementos muy potentes del lenguaje.

En común con nuestro ejemplo origen tenemos que la función **saveTranslation** tiene como argumento una palabra nueva (Word) y nos devuelve el diccionario bien sea como clase diccionario (Dictionary) o simplemente como lista de palabras (List<Word>).

Igualmente recuperamos un objeto que contiene las palabras de nuestro diccionario de las preferencias.

```
val dictionary = getDictionaryFromMemory()
```

A partir de aquí ya empezamos a ver los cambios, pues teniendo el diccionario podemos comprobar si nuestra palabra existe con una simple función sobre la lista de palabras.

La función **any** nos devolverá una variable boolean dependiendo de si en la colección de palabras existe alguna con el término que queremos guardar.

```
if(dictionary.wordList.any {it.term.equals(word.term, true)}){}
```

En el caso de que exista, definiremos una función que, para cada palabra (Word) y término (String), si existe ese término, nos devuelva una copia de la palabra con la traducción modificada o la propia palabra en otro caso:

```
val modifyDictionaryTerm: (Word, String) -> Word = {
    w, term -> if (term.equals(w.term, true)) w.copy(translation =
word.translation) else w }
```

Hacemos una copia del objeto para respetar la inmutabilidad de nuestras variables.

Pues bien, ya tenemos nuestra propia función, pero ¿qué nos faltaría? Aplicarla a nuestra lista de palabras del diccionario:

```
val dictCopy = dictionary.copy(wordList = dictionary.wordList.map {
modifyDictionaryTerm(it, word.term) })
```

Para ello usamos la función **map** que, a partir de una colección, nos devuelve otra (recuerda la inmutabilidad) a la que se le ha aplicado una función para cada elemento de esta.

Así, para cada palabra del diccionario, se aplicará la función que hemos definido y nos devolverá otra lista con la nueva traducción del término buscado.

Finalmente haremos una copia de nuestro diccionario para almacenarlo en preferencias y emitirlo a nuestros observadores.

En el caso de que no exista el término en nuestro diccionario, lo que haremos será añadirlo. Para ello, crearemos una copia de nuestro diccionario y guardaremos el contenido en preferencias:

```
val dictCopy = dictionary.copy(wordList = dictionary.wordList.plus(word))
```

Módulo de presentación

Hemos visto las capas de **model** y **data**, centrémonos ahora en el módulo de presentación (**app**).

Observers

Si comparamos los observadores que hemos definido, veremos que el código queda más limpio y resumido.

Frente a lo que teníamos en Java:

```
public class DictionaryObserver extends DisposableObserver<Array
List<Word>> {

    private IDictionaryView iDictionaryView;

    private Context context;

    public DictionaryObserver(IDictionaryView iDictionaryView, Context
 context)
    {
        this.iDictionaryView=iDictionaryView;
        this.context=context;
    }

    @Override
    public void onNext(ArrayList<Word> value) {
        iDictionaryView.renderData(value);
    }

    @Override
    public void onError(Throwable e) {
        iDictionaryView.hideLoading();
        iDictionaryView.showMessage(context.getString(R.string.
```

```
error_no_records));
    }

    @Override
    public void onComplete() {
        iDictionaryView.hideLoading();
    }
}
```

Tenemos en Kotlin:

```
class DictionaryObserver(private val iDictionaryView:
IDictionaryView):DisposableObserver<Dictionary>() {

    override fun onComplete()= iDictionaryView.hideLoading()

    override fun onNext(value: Dictionary)= iDictionaryView.
renderData(value)

    override fun onError(e: Throwable?) {
        iDictionaryView.hideLoading()
        iDictionaryView.showMessage("No se ha podido obtener registros")
  }
}
```

La característica fundamental para clarificar y resumir el código es la expresividad del lenguaje. Mostramos, como ejemplo, esta función usando una forma contraída:

```
override fun onComplete()= iDictionaryView.hideLoading()
Frente a la otra posibilidad, también correcta:
override fun onComplete() {
     iDictionaryView.hideLoading()
}
```

Presenter

Si nos fijamos ahora en la clase del presentador, también apreciaremos una menor claridad en:

```
public class DictionaryPresenter implements IPresenter {

    private IDictionaryView iDictionaryView;

    private GetDictionaryUseCase getDictionaryUseCase;
    private SaveTranslationUseCase saveTranslationUseCase;
    private ClearDictionaryUseCase clearDictionaryUseCase;
    private Context context;

    @Inject
    public DictionaryPresenter(Context context
                               ,IDictionaryView iDictionaryView
                               ,GetDictionaryUseCase getDictionary
UseCase
                               ,SaveTranslationUseCase save
TranslationUseCase
                               ,ClearDictionaryUseCase clear
DictionaryUseCase
    ) {
        this.iDictionaryView = iDictionaryView;
```

```
        this.context=context;
        this.getDictionaryUseCase = getDictionaryUseCase;
        this.saveTranslationUseCase = saveTranslationUseCase;
        this.clearDictionaryUseCase=clearDictionaryUseCase;

    }

    public void initialize() {
        getDictionary();
    }

    public void getDictionary() {

        iDictionaryView.showLoading();

        getDictionaryUseCase.implementUseCase( new
DictionaryObserver(iDictionaryView,context), null);
    }

    public void saveTranslation(String term, String translation) {
        iDictionaryView.showLoading();

        saveTranslationUseCase.implementUseCase( new
DictionaryTranslatorObserver(iDictionaryView,context),
Factory.WordFactory.Create(term, translation));
    }

    public void clearDictionary()
    {
        iDictionaryView.showLoading();

        clearDictionaryUseCase.implementUseCase( new
DictionaryClearObserver(iDictionaryView,context), null);
    }

    @Override
    public void destroy() {
        this.iDictionaryView = null;
        if (getDictionaryUseCase != null) getDictionaryUseCase.
cancelSubscription();
        if (saveTranslationUseCase != null) saveTranslationUseCase.
cancelSubscription();
        if (clearDictionaryUseCase != null) clearDictionaryUseCase.
cancelSubscription();
    }
}
```

Comparado con:

```
class DictionaryPresenter @Inject constructor(private val
iDictionaryView:IDictionaryView
                              ,private val
getDictionaryUseCase:GetDictionaryUseCase
                              ,private val
saveTranslationUseCase:SaveTranslationUseCase
                              ,private val
clearDictionaryUseCase:ClearDictionaryUseCase) {

    fun initialize() = getDictionary()

    fun getDictionary() {
        iDictionaryView.showLoading()
        getDictionaryUseCase.
implementUseCase(DictionaryObserver(iDictionaryView), null)
    }
```

```
    fun saveTranslation(term: String, translation: String) {
        iDictionaryView.showLoading()
        saveTranslationUseCase.
implementUseCase(DictionaryTranslatorObserver(iDictionaryView),
Word(term,translation))
    }
    fun clearDictionary() {
        iDictionaryView.showLoading()
        clearDictionaryUseCase.
implementUseCase(DictionaryClearObserver(iDictionaryView), null)
    }

    fun destroy(){
        getDictionaryUseCase?.cancelSubscription()
        saveTranslationUseCase?.cancelSubscription()
        clearDictionaryUseCase?.cancelSubscription()
    }

}
```

Activity

Finalmente, si comparamos las actividades codificadas en los diferentes lenguajes, veremos un par de aspectos muy interesantes de Kotlin.

```
public class DictionaryActivity extends AppCompatActivity implements
IDictionaryView, SwipeRefreshLayout.OnRefreshListener {

    Button btn_save_word;
    FloatingActionButton btn_add;
    EditText term_edit_text;
    EditText translation_edit_text;
    ListView list_view_dictionary;
    LinearLayout linear_layout_word_edition;
    ProgressBar loading_progress;
    SwipeRefreshLayout swipe_layout;

    @Inject
    DictionaryPresenter dictionaryPresenter;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        AndroidInjection.inject(this);

        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_dictionary);

        loading_progress = findViewById(R.id.loading_progress);
        swipe_layout = findViewById(R.id.swipe_layout);
        swipe_layout.setOnRefreshListener(this);

        dictionaryPresenter.initialize();

        term_edit_text = findViewById(R.id.term_edit_text);
        translation_edit_text = findViewById(R.id.translation_edit_text);
        btn_save_word = findViewById(R.id.btn_save_word);
        list_view_dictionary = findViewById(R.id.list_view_dictionary);
        linear_layout_word_edition = findViewById(R.id.linear_
layout_word_edition);
        btn_add = findViewById(R.id.btn_add);

        btn_save_word.setOnClickListener(saveTranslationClick);
        btn_add.setOnClickListener(buttonAddClick);

    }
```

```
    View.OnClickListener saveTranslationClick = new View.
OnClickListener() {
        @Override
        public void onClick(View v) {
            dictionaryPresenter.saveTranslation(term_edit_text.
getText().toString(), translation_edit_text.getText().toString());
        }
    };

    View.OnClickListener buttonAddClick = new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            if (linear_layout_word_edition.getVisibility() == View.
VISIBLE)
                linear_layout_word_edition.setVisibility(View.GONE);
            else
                linear_layout_word_edition.setVisibility(View.VISIBLE);
        }
    };

    @Override
    public void showLoading() {
        loading_progress.setVisibility(View.VISIBLE);
    }

    @Override
    public void hideLoading() {
        loading_progress.setVisibility(View.GONE);
        swipe_layout.setRefreshing(false);
    }

    @Override
    public void showMessage(String message) {
        Toast.makeText(this, message, Toast.LENGTH_SHORT).show();
    }

    @Override
    public void renderData(ArrayList<Word> dictionary) {
        refreshAdapter(dictionary);
        linear_layout_word_edition.setVisibility(View.GONE);
        term_edit_text.setText("");
        translation_edit_text.setText("");
    }
    @Override
    public void dictionaryCleared(Boolean value) {
        if (value) {
            ArrayAdapter<Word> adapter = new DictionaryAdapter(this,
new ArrayList<Word>());
            list_view_dictionary.setAdapter(adapter);
        }
    }
    @Override
    public void onRefresh() {
        dictionaryPresenter.getDictionary();
    }

    private void refreshAdapter(ArrayList<Word> dictionary) {

        ArrayAdapter<Word> adapter = new DictionaryAdapter(this,
dictionary);
        list_view_dictionary.setAdapter(adapter);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.menu_dictionary, menu);
        return true;
```

```
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        int id = item.getItemId();
        if (id == R.id.clear_dictionary) {
            dictionaryPresenter.clearDictionary();
            return true;
        }
        return super.onOptionsItemSelected(item);
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        dictionaryPresenter.destroy();
    }
}
```

Obtener referencias de las vistas

En la definición de la actividad codificada en Java vemos partes en las que podríamos ganar muchísimo en legibilidad, tales como la declaración de variables:

```
Button btn_save_word;
FloatingActionButton btn_add;
EditText term_edit_text;
EditText translation_edit_text;
ListView list_view_dictionary;
LinearLayout linear_layout_word_edition;
ProgressBar loading_progress;
SwipeRefreshLayout swipe_layout;
```

Y su posterior inicialización dentro del **onCreate** del ciclo de vida de la actividad:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    AndroidInjection.inject(this);

    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_dictionary);

    loading_progress = findViewById(R.id.loading_progress);
    swipe_layout = findViewById(R.id.swipe_layout);
    swipe_layout.setOnRefreshListener(this);

    term_edit_text = findViewById(R.id.term_edit_text);
    translation_edit_text = findViewById(R.id.translation_edit_text);
    btn_save_word = findViewById(R.id.btn_save_word);
    list_view_dictionary = findViewById(R.id.list_view_dictionary);
    linear_layout_word_edition = findViewById(R.id.linear_layout_word_
edition);
    btn_add = findViewById(R.id.btn_add);

    btn_save_word.setOnClickListener(saveTranslationClick);
    btn_add.setOnClickListener(buttonAddClick);

}
```

Frente a esto, en Kotlin tendríamos algo como:

```
override fun onCreate(savedInstanceState: Bundle?) {
    AndroidInjection.inject(this)
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_dictionary)

    swipe_layout.setOnRefreshListener { dictionaryPresenter.
getDictionary() }
    dictionaryPresenter.initialize()

    list_view_dictionary.layoutManager = LinearLayoutManager(this)
    list_view_dictionary.adapter = DictionaryAdapter(emptyList())

    btn_add.setOnClickListener(addWordListener)
    btn_save_word.setOnClickListener(saveWordListener)
}
```

Y es que no necesitamos hacer el **findViewById** (en Java podíamos evitarlo mediante el uso de librerías de terceros como Butterknife) sin hacer nada más que importar en nuestro proyecto el plugin de extensiones del lenguaje.

```
apply plugin: 'kotlin-android-extensions'
```

De esta forma, tan solo con asignarle un ID a cada elemento de nuestro layout tendremos a nuestra disposición la referencia en nuestra Activity.

```
list_view_dictionary.layoutManager = LinearLayoutManager(this)
list_view_dictionary.adapter = DictionaryAdapter(emptyList())
```

Otra de las cosas interesantes que tenemos en Kotlin es la definición de escuchadores como funciones:

```
private val addWordListener = View.OnClickListener{
    if(linear_layout_word_edition.visibility == View.VISIBLE){
        linear_layout_word_edition.visibility = View.GONE
    }
    else{
        linear_layout_word_edition.visibility = View.VISIBLE
    }
}
```

Frente al código más engorroso que teníamos en Java:

```
View.OnClickListener buttonAddClick = new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        if (linear_layout_word_edition.getVisibility() == View.VISIBLE)
            linear_layout_word_edition.setVisibility(View.GONE);
        else
            linear_layout_word_edition.setVisibility(View.VISIBLE);
    }
};
```

Además podemos acceder a las propiedades de las clases directamente sin necesidad de explicitar **get** o **set**.

Extensiones de clases

Otra de las características que hemos visto en el resumen del punto anterior es el de poder extender clases que no sean hechas por nosotros sin necesidad de modificarlas. Definimos extensiones de las clases para añadir características nuevas.

De esta forma podríamos definir la extensión de la clase **Context** para añadir una operación que muestre un mensaje al usuario de la siguiente forma:

```
package com.ajr.libroandroid.presentation.extensions

import android.content.Context
import android.widget.Toast

fun Context.toast(message: CharSequence) = Toast.makeText(this,
message, Toast.LENGTH_SHORT).show()
```

Simplemente con esta definición podremos utilizar esta función en cualquier clase del tipo **Context**.

Así, en Activity (que hereda de Context), podremos utilizarla de esta manera:

```
override fun showMessage(message: String) {
    toast(message)
}
```

Esto ejecutará la función y mostrará un mensaje al usuario.

REFLEXIONES FINALES

Lejos de pretender ser una enciclopedia, el objetivo de este libro es proporcionarte herramientas sencillas pero extensas. Confiamos en haber sido capaces de presentarte las herramientas necesarias e introducirte lo suficiente como para animarte a lanzarte a su utilización y a su descubrimiento por la vía de la propia experiencia.

Este libro es el resultado de nuestras lecturas, trabajo y experiencia. No hemos inventado nada nuevo. Simplemente hemos ordenado y puesto en común las técnicas y herramientas esenciales para ponerlas a tu disposición y explicar nuestra forma de entender el desarrollo profesional.

La tecnología avanza muy rápidamente, pero, con las bases expuestas en esta obra, estamos convencidos de que serás capaz de entender y adaptarte a la evolución de cualquiera de los temas tratados. Esperamos que hayas sido capaz de abrir la mente hacia la posibilidad de crear software de mayor calidad, con una mejor organización y estructura, definiendo la arquitectura en capas, bien diferenciadas del proyecto.

Asimismo, hemos pensado que debías tener una visión más reciente del estado del desarrollo Android. Por este motivo, hemos considerado presentar Kotlin, un elemento novedoso que dará que hablar durante al menos los próximos años.

Queremos que te quedes con la idea de que una aplicación Android se puede realizar rápidamente, pero que te cuestiones si estarías dispuesto a sacrificar algo más de tu tiempo para diseñar una arquitectura que puedas reaprovechar en otros proyectos. Sigue realizando buenas prácticas de desarrollo, divide el problema en capas, haz que una clase haga lo que tenga que hacer, haz tu código reutilizable y, por supuesto, nunca te repitas.

Creemos que la teoría basada en ejemplos facilita el aprendizaje a través de la propia experiencia, poniendo en práctica los ejemplos que se consideren oportunos y contrastando el resultado con el código fuente aportado y asociado a este libro.

Para finalizar, deseamos de todo corazón que hayas disfrutado de la lectura, tanto como nosotros lo hemos hecho escribiendo este libro con tanta ilusión. Estamos encantados de compartir este mundo Androide contigo.

Descargado en: eybooks.com