



Universidad Central de Venezuela

Facultad de Ciencias
Escuela de Computación
Aplicaciones de Tecnología en Internet



Desarrollo de una Aplicación Móvil para la Recarga de Saldo Auto Gestionable con Consola de Administración Web

Trabajo Especial de Grado presentado ante la Ilustre
Universidad Central de Venezuela
por los Bachilleres:

Cesar J. Rodríguez D.
C.I.: 21.306.130
cesarjgrd.18@gmail.com

Daniel A. Romero A.
C.I.: 23.624.323
dara323@gmail.com

Para optar al título de Licenciado en Computación

Tutores:
Prof. Eric Gamess
Prof. Roger Bello

Caracas, Mayo 2017



Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación
Laboratorio de Comunicación y Redes



ACTA DE VEREDICTO

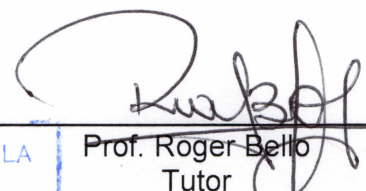
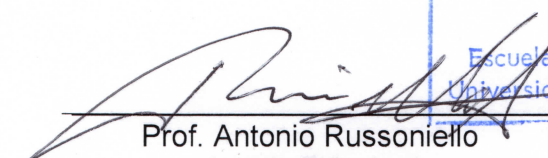
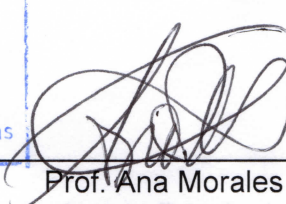
Quienes suscriben, miembros del jurado designado por el Consejo de Escuela de Computación, para examinar el Trabajo Especial de Grado presentado por los Bachilleres Cesar J. Rodríguez C.I. V-21.306.130 y Daniel A. Romero C.I. V-23.624.323, con el título "**Desarrollo de una Aplicación Móvil para la Recarga de Saldo Auto Gestionable con Consola de Administración Web**", a los fines de cumplir con el requisito legal para optar al título de Licenciado en Computación, dejan constancia de lo siguiente:

Leído el nombrado trabajo por cada uno de los miembros del jurado, éste fijó el día miércoles 17 de mayo de 2017, para que sus autores lo defendieran en forma pública, en la Sala I de la Escuela de Computación, mediante una exposición oral de su contenido, luego de la cual respondieron satisfactoriamente a las preguntas que les fueron formuladas por el jurado, todo ello conforme a lo dispuesto en la Ley de Universidades y demás normativas vigentes de la Universidad Central de Venezuela.

Finalizada la defensa pública del Trabajo Especial de Grado, el jurado decidió aprobarlo.

Es de aclarar que el Profesor Eric Gamess se encuentra de permiso fuera del país. Por esta razón, el Profesor Robinson Rivas, director de la Escuela de Computación, firma el presente documento en su lugar.

En fe de lo cual se levanta el presente acta en Caracas a los 17 días del mes de mayo del año 2017, dejando constancia que el Profesor Roger Bello actuó como coordinador del jurado.

 por el Prof. Eric Gamess Tutor	 Prof. Roger Bello Tutor
 Prof. Antonio Russoniello Jurado Principal	 Prof. Ana Morales Jurado Principal

REPUBLICA DE VENEZUELA
Facultad de Ciencias
Escuela de Computación
Universidad Central - Caracas

Dedicatoria

*En memoria de mi querida hermana, Marianella Gago.
Quien siempre supo cómo vencer las adversidades de la vida y luchar por sus metas.
Quien siempre fue mi apoyo y el de mi familia.
A quien recordaremos con cariño y amor.*

Daniel Romero.

“Quiero dedicarles este trabajo a mis padres, Cesar Enrique Rodríguez y Marieli Díaz, quienes son un increíble ejemplo de superación con valores y ética intachables, que me enseñaron lo importante que es estudiar para alcanzar las metas que nos trazamos en la vida. A mis hermanas Maurinelly, Mariana y Marielí quienes han estado a mi lado guiándome y apoyándome en cada paso que he dado. Por último, a mi segunda casa, la que vence las sombras, la Ilustre Universidad Central de Venezuela y a todas las personas que forman parte de ella que me han hecho crecer como persona, estudiante y profesional”

Cesar José Rodríguez

“Quiero dedicarle este trabajo a mis queridos padres, Carmen Araujo y Guillermo Romero, quienes son un ejemplo de esfuerzo y dedicación además de ser un gran apoyo para mí, que día tras día me siguen empujando para dar lo mejor de mí. A mi hermana Marianella Gago y a mis sobrinas Victoria Veroes y Paola Veroes que siempre estuvieron a mi lado, apoyándome, siendo motivos de inspiración. A mis padrinos, Eleonora Acosta y Manuel Padilla, que son figuras ejemplares para mí y siempre me alentaron a dar lo mejor. A todos mis profesores, que impartieron sus conocimientos sin compromiso alguno, para que todos seamos profesionales con valores y ética. A mi compañero de tesis, quien siempre apostó por mis conocimientos y fue de gran ayuda para mi crecimiento profesional. Y por último a la Universidad Central de Venezuela, que me abrió sus puertas y me permitió crecer como individuo”

Daniel A. Romero

“Especial agradecimiento para Jesús Yepez, quien nos brindó la oportunidad para emprender este proyecto. Confiando en nosotros para llevarlo a cabo brindándonos no solo la infraestructura necesaria, sino un mundo de conocimientos y experiencia para culminarlo satisfactoriamente. Otro agradecimiento a nuestros tutores, el Profesor Eric Gamess y el Profesor Roger Bello, quienes a lo largo de estos meses nos guiaron y encaminaron en la elaboración del presente trabajo”

Daniel y Cesar

Resumen

TÍTULO

Desarrollo de Aplicación Móvil para Recarga de Saldo Auto Gestionable con Consola de Administración Web.

AUTORES

Cesar J. Rodríguez D.
Daniel A. Romero A.

TUTORES

Prof. Eric Gamess y Prof. Roger Bello

Debido al creciente mercado latino en EEUU, la empresa TopUp City desarrolló un API para realizar recargas de saldo desde dicha nación hacia países latinoamericanos. Lo que generó la necesidad de un cliente móvil para facilitar las recargas a los usuarios y una consola web para administrar los productos ofrecidos.

En el presente Trabajo Especial de Grado, se estudiaron diversas tecnologías y enfoques para la creación de una aplicación móvil la cual pudiera consumir el API de TopUp City, realizar el cobro al cliente y además tener una interfaz de usuario usable; Así como las distintas opciones tecnológicas para una satisfactoria administración de los países, operadores telefónicos y productos ofrecidos en la aplicación vía la consola web.

En cuanto al cliente móvil, se implementó una aplicación para sistemas operativos Android usando el enfoque nativo, y PayPal como pasarela de pago móvil. En cuanto a la consola web, se implementó usando Spring Framework, en conjunto con Java Persisten API (JPA), un API de Hibernate, como ORM para poder realizar las operaciones elementales de una base de datos y PrimeUI como framework de interfaz gráfica.

Palabras Claves: Consola Web, Aplicación Móvil, Android, PayPal, API.

Tabla de Contenido

Índice de Figuras	15
Índice de Tablas.....	19
1. Introducción.....	21
2. El Problema.....	23
2.1 Planteamiento y Justificación del Problema.....	23
2.2 Objetivo General	23
2.3 Objetivos Específicos.....	24
2.4 Alcance.....	24
3. Aplicaciones Web.....	25
3.1 Paradigma Cliente-Servidor.....	25
3.1.1 Cliente	26
3.1.2 Servidor	26
3.1.3 Ventajas	26
3.1.4 Desventajas	27
3.1.5 Separación de Niveles Lógicos	27
3.2 Frontend	28
3.2.1 HTML5.....	28
3.2.2 CSS3.....	28
3.2.3 JavaScript.....	29
3.3 Backend	29
3.4 Sistema Manejador de Bases de Datos	30
3.5 Tipos de Aplicaciones Web.....	31
3.6 Frameworks y Lenguajes de Programación	32
3.7 Java.....	33
3.7.1 Java en Aplicaciones Web.....	33
3.7.2 Servlet	34
3.7.3 Enterprise JavaBeans (EJB).....	34
3.7.4 Servlet vs Enterprise JavaBean.....	35
4. Aplicaciones Móviles.....	37
4.1 Aplicaciones Nativas	38
4.2 Aplicaciones Móviles basadas en la Web	40
4.3 Aplicaciones Híbridas	41
5. Sistema Operativo Android	43
5.1 Arquitectura de Android	44
5.1.1 Máquina Virtual Dalvik	45

5.1.2	Conjunto de Instrucciones	46
5.1.3	Kernel y Proceso de Inicio	46
5.1.4	Librería Bionic	47
5.1.5	Almacenamiento de Datos y Sistema de Archivos	47
5.1.6	Administración de Energía.....	48
5.2	Aplicaciones de Android	48
5.2.1	Actividad.....	48
5.2.2	Servicio.....	50
5.2.3	Intento	51
5.2.4	Proveedores de Contenido	51
6.	API: Interfaz de Programación de Aplicaciones	53
6.1	API Privado	54
6.2	API Público.....	55
7.	Pasarela de Pago.....	57
7.1	Principales Pasarelas de Pago.....	58
8.	Trabajos Relacionados	61
8.1	Aplicación de Recarga de Saldo de Technology & Solutions LTDA.....	61
8.2	Aplicación de Recarga de Saldo Ezetop Ding.....	62
9.	Marco Metodológico	67
9.1	Método Mobile-D	67
9.2	Fase de Exploración	68
9.2.1	Arquitectura de la Solución	68
9.3	Fase de Inicialización.....	69
9.3.1	Tecnologías a Utilizar	70
9.3.2	Prototipo de Interfaz del Cliente Móvil.....	70
9.3.3	Prototipo de Interfaz de la Consola Web.....	74
9.4	Fases de Producción y Estabilización	78
9.5	Fase de Pruebas.....	78
10.	Marco Aplicativo.....	79
10.1	Análisis General de la Aplicación Móvil.....	79
10.1.1	Prueba de Concepto de Inicio de Sesión Social	79
10.1.2	Prueba de Concepto Integración con PayPal	81
10.1.3	Flujo Principal de la Aplicación	83
10.1.4	Integración con API Privado de TopUp City LLC	87
10.1.5	Módulo de Autenticación.....	89
10.1.6	Desarrollo del Historial de Transacciones.....	91
10.1.7	Mejora de las UI con un Diseño Adaptativo	92

10.1.8 Configuración del Proyecto.....	98
10.2 Análisis General de la Consola Administrativa.....	101
10.2.1 Configuración del Ambiente.....	101
10.2.2 Desarrollo de la Pantalla de Login.....	104
10.2.3 Desarrollo del Tablero o Pantalla Inicial.....	109
10.2.4 Desarrollo de la Administración de Usuarios	112
10.2.5 Desarrollo de la Administración de Cuentas	118
10.2.6 Desarrollo de la Administración de Países.....	120
10.2.7 Desarrollo de la Administración de Monedas.....	122
10.2.8 Desarrollo de la Administración de Telefonías.....	125
10.2.9 Desarrollo de la Administración de Prefijos.....	126
10.2.10 Desarrollo de la Administración de Productos	128
10.2.11 Desarrollo de la Administración de Transacciones	129
10.3 Resultado de Pruebas	131
11. Conclusiones	133
Referencias	135

Índice de Figuras

Figura 3.1: Modelo Cliente-Servidor	25
Figura 3.2: Separación de Niveles de Funciones.....	27
Figura 3.3: Servlet vs EJBs.....	35
Figura 4.1: Teléfonos Básicos Conocidos como Features Phones	37
Figura 4.2: Aplicación Nativa: Interacción con el Dispositivo Móvil	39
Figura 4.3: Aplicación Híbrida: Interacción con el Dispositivo Móvil	42
Figura 5.1: Número Acumulado de Aplicaciones Descargadas de Google Play	43
Figura 5.2: Número de Aplicaciones Disponibles en la Tienda de Google Play.....	44
Figura 5.3: Arquitectura de Android	45
Figura 5.4: Ciclo de Vida de una Actividad	50
Figura 6.1: Estructura e Interacción de un Web API	53
Figura 6.2: Estructura e Interacción de un Web API Privado	54
Figura 6.3: Estructura e Interacción de un Web API Público	55
Figura 7.1: Pasarelas de Pago por Minoristas en EEUU	58
Figura 8.1: Pantalla de Confirmación y Aprobación de la Recarga.....	61
Figura 8.2: Pantalla Inicial de Ezetop Ding	62
Figura 8.3: Pantalla Inicial para Recarga de Saldo	63
Figura 8.4: Teclado para Ingresar Números Telefónicos	63
Figura 8.5: Selección del País	64
Figura 8.6: Selección del Operador	64
Figura 8.7: Selección de Producto o Monto	65
Figura 8.8: Sección de Pago.....	65
Figura 9.1: Ciclo de Desarrollo de Mobile-D	68
Figura 9.2: Arquitectura de la Solución	69
Figura 9.3: Pantalla de Bienvenida	70
Figura 9.4: Menú Lateral para Invitados	71
Figura 9.5: Menú Lateral para Usuarios Registrados.....	71
Figura 9.6: “ <i>Fragment</i> ” con Formulario de Recarga.....	72
Figura 9.7; “ <i>Fragment</i> ” con Detalle del Producto	72
Figura 9.8: Prototipo de Interfaz de Inicio de Sesión y Registro	73
Figura 9.9: Prototipo de Interfaz para Listado de Recargas.....	73
Figura 9.10: Pantalla de Inicio de Sesión.....	74
Figura 9.11: Página de Inicio o Tablero de la Consola Administrativa	75
Figura 9.12: Página Principal de los Recursos Administrables	76
Figura 9.13: Ejemplo de Formularios para Agregar o Editar Registros	77
Figura 10.1: Diagrama de Flujo del Inicio de Sesión Social	80
Figura 10.2: “ <i>Activities</i> ” de la Prueba de Concepto de Inicio de Sesión.....	80
Figura 10.3: “ <i>Activity</i> ” con la Integración de PayPal.....	81
Figura 10.4: Importando SDK de PayPal	82
Figura 10.5: Configuración del SDK de PayPal	82
Figura 10.6: Llamada al SDK de PayPal para Realizar Pago	83
Figura 10.7: Flujo Principal de la Aplicación	83
Figura 10.8: Interfaz Gráfica del <i>LaunchScreenActivity</i>	84
Figura 10.9: Menú Lateral “Tipo Cajón”	85
Figura 10.10: Interfaz Gráfica del <i>SelectProductFragment</i>	86

Figura 10.11: Interfaz Gráfica del <i>DetailProductFragment</i>	86
Figura 10.12: “ <i>PaymentActivity</i> ” del SDK de PayPal	87
Figura 10.13: Java Interface de Retrofit	88
Figura 10.14: Ejemplo de Petición HTTP con Retrofit	89
Figura 10.15: Interfaz Gráfica de las Pestañas del “ <i>SignInActivity</i> ”	89
Figura 10.16: Flujo de Autenticación con Redes Sociales	90
Figura 10.17: Histórico de Recargas	91
Figura 10.18: Ejemplo con Vista Previa de un “ <i>Vector Drawable</i> ”	93
Figura 10.19: Directorio de Recursos del Proyecto de Android	93
Figura 10.20: Nuevo Flujo Principal de la Aplicación	94
Figura 10.21: Pantallas Promocionales del “ <i>IntroActivity</i> ”	95
Figura 10.22: Interfaz Gráfica del “ <i>HomeFragment</i> ”	95
Figura 10.23: Interfaz Gráfica del “ <i>NumpadFragment</i> ”	96
Figura 10.24: Nueva Interfaz Gráfica del “ <i>DetailFragment</i> ”	97
Figura 10.25: Nueva Interfaz Gráfica del “ <i>SelectProductFragment</i> ”	97
Figura 10.26: Contenido del Archivo “ <i>build.gradle</i> ” de Nivel Superior	98
Figura 10.27: Contenido del Archivo “ <i>settings.gradle</i> ”	99
Figura 10.28: Contenido del Archivo “ <i>gradle.properties</i> ”	99
Figura 10.29: Contenido del Archivo “ <i>build.gradle</i> ” de Nivel de Módulo	100
Figura 10.30: Archivo “ <i>pom.xml</i> ”	102
Figura 10.31: Dependencia de MySQL Requerida por Maven	102
Figura 10.32: Contenido del Archivo “ <i>application.properties</i> ”	103
Figura 10.33: Spring Beans para Definir la Capa de la Vista	103
Figura 10.34: Contenido del Archivo “ <i>tiles-definitions.xml</i> ”	104
Figura 10.35: Ejemplo del Archivo “ <i>message.properties</i> ”	104
Figura 10.36: Ejemplo de Inclusión de una Plantilla	105
Figura 10.37: Formulario para Iniciar Sesión	105
Figura 10.38: Controlador Utilizado para Acceder al Login	106
Figura 10.39: Clase LoginForm	106
Figura 10.40: Modelo de Usuarios	107
Figura 10.41: Interfaz DAO para los Usuarios	108
Figura 10.42: Diagrama de Flujo del Login	108
Figura 10.43: Menú Superior Utilizado en la Plantilla General	109
Figura 10.44: Menú Superior	109
Figura 10.45: Variables en Archivos JSP	110
Figura 10.46: Menú Lateral	110
Figura 10.47: Contenedor de Datos del Home	111
Figura 10.48: Controlador del Home	111
Figura 10.49: Resultado Final del Home	112
Figura 10.50: Estructura de la Tabla de los Recursos Administrables	112
Figura 10.51: Script para Crear las Tablas con <i>Prime UI</i>	113
Figura 10.52: Objeto <i>Paginator</i> de la Función <i>Puidatatable</i>	113
Figura 10.53: Controlador para Acceder a la Lista de Usuarios	113
Figura 10.54: Objeto <i>Columns</i> de la Función <i>Puidatatable</i>	114
Figura 10.55: Petición Ajax del Objeto <i>Datasource</i>	114
Figura 10.56: Controlador Utilizado para Pagar los Registros	115
Figura 10.57: Resultado Final de la Tabla de Usuarios	115
Figura 10.58: Formulario para Crear o Editar Usuarios	116
Figura 10.59: Diagrama de Flujo de Añadir Usuario	116

Figura 10.60: Controlador Utilizado para Llenar el Formulario que Permite Editar un Usuario	117
Figura 10.61: Modal para Eliminar Registro	118
Figura 10.62: Modelo de Cuentas	119
Figura 10.63: Modelo de Países	120
Figura 10.64: Formulario para Crear o Editar Países	121
Figura 10.65: Controlador Utilizado para Llenar el Formulario para Editar un País.	121
Figura 10.66: Función <i>AjaxMethodDelete</i> para Eliminar Registros.....	122
Figura 10.67: Modelo de Monedas	123
Figura 10.68: Formulario para Crear o Editar Monedas	123
Figura 10.69: Etiqueta <i>Select</i> de Spring para Seleccionar un País.....	124
Figura 10.70: Controlador para Acceder al Formulario de Añadir Moneda	124
Figura 10.71: Modelo de Telefonías	125
Figura 10.72: Formulario para Crear o Editar Telefonías.....	126
Figura 10.73: Modelo de Prefijos	127
Figura 10.74: Formulario para Crear o Editar Prefijos	127
Figura 10.75: Modelo de Productos	128
Figura 10.76: Formulario para Crear o Editar Productos	129
Figura 10.77: Modelo de Transacciones	130
Figura 10.78: Resultado Final de la Tabla de Transacciones	130

Índice de Tablas

Tabla 4.1: Leguajes, Herramientas y Canales de Distribución de Apple iOS y Android	38
Tabla 4.2: Comparación de los Enfoques de Aplicaciones basadas en Web	40

1. Introducción

Desde hace algunos años, gracias a la evolución tecnológica han surgido nuevas tecnologías lo que ha permitido el desarrollo de nuevos tipos de aplicaciones tanto web como móviles que han revolucionado la forma en que se ve y hasta se usa los dispositivos, y esto se refiere a que ha dejado de un lado los dispositivos más grandes y alámbricos para trasladarnos a dispositivos más pequeños e inalámbricos debido a las facilidades que nos provee el Internet conjunto a las aplicaciones que en este medio se desarrollan.

Hoy en día el uso de aplicaciones es prácticamente una necesidad humana ya que es la forma en la que nos comunicamos, lo que ha producido una alta competitividad a nivel empresarial de desarrollar constantemente aplicaciones para buscar innovar el mercado.

Las aplicaciones en la actualidad van desde lo más sencillo, como una página informativa, hasta incluso hacer uso de una pasarela de pagos para permitirle a los usuarios realizar compras desde un sitio web o incluso de una aplicación móvil; para realizar estas aplicaciones se deben hacer uso de distintas tecnologías, y ese es uno de los objetivos de esta investigación, contemplar, analizar y conceptualizar algunas de estas tecnologías para el desarrollo de una aplicación móvil que permita recargas de saldo y sea auto gestionable mediante una consola web.

Habiendo mencionado lo anterior, este documento se encuentra estructurado en ocho capítulos descritos a continuación:

Capítulo 2. El problema

De acuerdo a la base teórica desarrollada, se plantea el problema que será resuelto, los objetivos a alcanzar, la metodología y las pruebas.

Capítulo 3. Aplicaciones Web

En este capítulo se describen los conceptos usados en una aplicación web, arquitectura de aplicaciones web, tipos de aplicaciones, y algunos lenguajes utilizados para el desarrollo de aplicaciones web.

Capítulo 4. Aplicaciones Móviles

En el cuarto capítulo, se describen los distintos tipos de aplicaciones móviles y su funcionamiento.

Capítulo 5. Sistema Operativo Android

En el quinto capítulo, se describe la arquitectura utilizada en los dispositivos que poseen un sistema operativo Android, así como los elementos para utilizarlos dentro de las aplicaciones móviles provenientes de dicha arquitectura.

Capítulo 6. API: Interfaz de Programación de Aplicaciones

En el sexto capítulo, se describe el concepto de un API, así como los tipos y la interacción de estas con las aplicaciones web y móviles.

Capítulo 7. Pasarela de Pago

Es un servicio intermediario entre un cliente de comercio electrónico y un banco. En el séptimo capítulo se profundiza el concepto y se realiza un análisis de las pasarelas de pago más utilizadas.

Capítulo 8. Trabajos Relacionados

En este capítulo se realiza un análisis de dos aplicaciones móviles para determinar los aspectos más importantes en la creación de una aplicación móvil.

Capítulo 9. Marco Metodológico

En este capítulo se explica la metodología a utilizar y cómo influye en el desarrollo del proyecto, además se exponen los prototipos de las interfaces de ambas aplicaciones.

Capítulo 10. Marco Aplicativo

En el décimo capítulo se detallará el desarrollo de ambas aplicaciones.

2. El Problema

En este capítulo, se presenta y detalla el planteamiento del problema que consiste en mantener los celulares con saldo, tocando diversos temas que nos llevarán a realizar una aplicación móvil para recargas de saldo de operadoras telefónicas en Latinoamérica, así como su justificación, los objetivos a lograr y el alcance de este proyecto.

2.1 Planteamiento y Justificación del Problema

El mercado de teléfonos en Latinoamérica está en crecimiento, en especial el de teléfonos inteligentes, que para finales del año 2016 cerca del 60% de los 705 millones de suscriptores poseían un *Smartphone* [1]. Además de una mejor conectividad y una amplia variedad de aplicaciones, mejora el uso de los servicios en línea a lo largo del día. Para muchos consumidores, los teléfonos inteligentes constituyen la primera forma de experimentar los servicios de Internet, ya que muchos tienen acceso limitado a los computadores ya la banda ancha fija.

Las suscripciones a banda ancha móvil seguirán aumentando junto con un estilo de vida más conectado. La conectividad permanente y de alta calidad es esencial para apoyar este nuevo ecosistema en el que la gente, la sociedad y las industrias dependen de canales digitales.

Por otro lado, EEUU cuenta con una alta presencia de latinos, existe una población de aproximadamente 55 millones de hispanos [2], los cuales equivalen al 17% [3] de la población total en dicho país. Gracias a esta gran cantidad de presencia latina, se ha generado un nicho de mercado que ha cobrado gran importancia, ya que para el 2013 tenía un poder adquisitivo de un billón de dólares, contando con una expectativa de crecimiento de hasta el 50% para el año 2015, siendo superada esta meta de manera satisfactoria.

Muchos latinos no tienen el poder adquisitivo suficiente para mantener un plan de datos activos en sus dispositivos móviles, por lo que dependen de redes WiFi para conectarse a Internet y estar en contacto con sus familiares en EEUU. Es por esto que en conjunto con la infraestructura de TopUp City LLC se busca desarrollar un sistema que permita realizar recargas de saldo desde EEUU a Colombia y México haciendo uso de una aplicación móvil, llevando un control de transacciones realizadas, estadísticas y configuración del sistema a través de un panel de administración web.

2.2 Objetivo General

Desarrollar una solución que permita realizar recargas de saldos desde dispositivos móviles soportados para Sistemas Operativos Android.

2.3 Objetivos Específicos

Para satisfacer el objetivo general, se plantean los siguientes objetivos específicos:

- Desarrollar un módulo para la aplicación móvil que permita conectarse e interactuar con el API privado de TopUp City LLC.
- Implementar la aplicación móvil.
- Implementar un módulo autenticación social en la aplicación móvil.
- Implementar el módulo de cobro del servicio desde la aplicación móvil.
- Implementar una Consola Web que permita la gestión y configuración del aplicativo móvil.
- Desarrollar pruebas de integración de componentes y funcionales.

2.4 Alcance

El presente trabajo tiene como finalidad la creación de una aplicación móvil nativa para sistemas operativos Android, que deberá integrarse con el API privado de TopUp City LLC para realizar las recargas de saldo para operadoras telefónicas de México y Colombia; y para recuperar el historial de recargas de cada usuario. También, se implementará el cobro mediante el uso de PayPal como pasarela de pago móvil y la autenticación social mediante los SDKs de Facebook y Google. Además, deberá cumplir con las guías de estilo “*Material Design*” de Google para enriquecer las interfaces gráficas y la experiencia de usuario. El desarrollo de esta aplicación llegará a fase beta, la cual se llamará MVP por sus siglas en inglés “*Minimal Value Product*”, y cumplirá con los requerimientos funcionales y no funcionales.

A su vez la consola web deberá integrarse con la base de datos de TopUp City LLC para realizar todas las operaciones inherentes al modelo de datos para así permitir el manejo en tiempo real de usuarios, productos y configuración del sistema.

3. Aplicaciones Web

En el presente capítulo se desarrollará lo que es una aplicación web, arquitectura sobre la cual opera, cuáles son los tipos de aplicaciones web más frecuentes, así como frameworks y lenguajes más utilizados

Una aplicación web, es un tipo especial de aplicación basada en la arquitectura cliente-servidor, donde el cliente (Navegador o Explorador) y el servidor (Servidor Web) se comunican mediante el protocolo HTTP [4]. Las aplicaciones web se encuentran dentro del servidor web y se desarrollan en dos niveles diferentes, como son el frontend y el backend.

3.1 Paradigma Cliente-Servidor

El paradigma cliente-servidor, es una arquitectura de red en la cual las tareas se reparten entre los servidores para poder dar una respuesta efectiva a los clientes. Usualmente, los servidores son máquinas potentes para poder optimizar el tiempo de respuesta, mientras que los clientes son máquinas menos potentes que utilizan los recursos de los servidores [5].

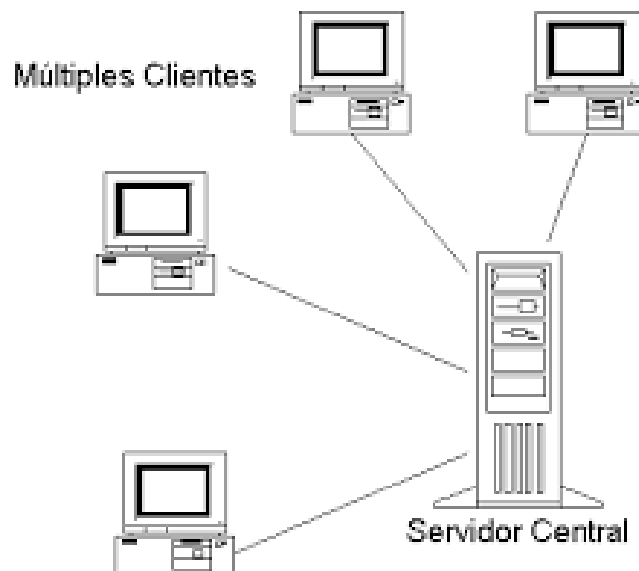


Figura 3.1: Modelo Cliente-Servidor

La Figura 3.1 presenta como múltiples clientes consumen los recursos de un servidor central. La idea es tratar a un computador como instrumento, que por sí sola pueda realizar muchas tareas, pero con la consideración de que realice aquellas que son más adecuadas a sus características. La forma más común de trabajar con esta arquitectura es haciendo uso de clientes o interfaces gráficas para el usuario y haciendo uso de servidores centrales de tipo mainframe, que se encarguen del manejo de los datos,

además de su seguridad e integridad. El objetivo principal es lograr fomentar el desarrollo modular de una aplicación, haciendo uso de esta arquitectura, para facilitar el desarrollo y mejorar su mantenimiento [6].

3.1.1 Cliente

El cliente, es el proceso que se encarga de captar los requerimientos de los usuarios mediante interfaces gráficas, usualmente haciendo uso de formularios, también es conocido como *frontend* y es el encargado mayormente de la captación y despliegue de datos, además de hacer uso de los servicios. Entre algunas funciones que cumplen los clientes, se encuentran las siguientes:

- Interactuar con el usuario.
- Realizar validaciones sencillas sobre los datos.
- Obtener las respuestas del servidor.
- Formatear los datos, para mostrárselos de forma sencilla al usuario.

3.1.2 Servidor

El servidor o también conocido como *backend*, es el proceso que atiende las peticiones de múltiples clientes que intentan acceder o hacer uso de un servicio que el administra. Es el encargado de manejar todas las funciones relacionadas con las reglas del negocio y el manejo de datos [6]. Entre algunas funciones que cumplen los servidores, se encuentran:

- Aceptar las peticiones que envían los clientes.
- Garantizar el acceso a los datos de forma segura.
- Aplicar la lógica del negocio.
- Realizar validaciones a nivel de bases de datos.

3.1.3 Ventajas

Como toda arquitectura de red posee ciertas ventajas, entre las cuales se pueden destacar [6]:

- Reduce los costos del proyecto, debido a que permite el uso de diferentes componentes, tanto de hardware como de software, lo que facilita el uso de máquinas más económicas provenientes de varios proveedores.
- Distribuye la carga de las tareas por ejecutar para así lograr concurrencia de procesos.
- Permite la escalabilidad horizontal, es decir, mejorar el rendimiento del sistema añadiendo nuevos nodos o servidores.
- Permite la escalabilidad vertical, agregando componentes de hardware a un nodo para así mejorar el rendimiento global.

- Facilita el intercambio de datos entre diferentes sistemas, los cuales no necesariamente deben tener la misma arquitectura o incluso el mismo lenguaje de programación.
- Utilizar herramientas o frameworks para agilizar el desarrollo de la aplicación, minimizando tiempo y costos.
- La estructura modular facilita la integración de nuevas tecnologías, favoreciendo así la escalabilidad del sistema.

3.1.4 Desventajas

En este orden de ideas, esta arquitectura posee ciertas limitaciones o desventajas, las cuales se desarrollan a continuación [6]:

- El mantenimiento del sistema podría provocar una serie de complicaciones, debido a la posible interacción de múltiples componentes de hardware y software provenientes de distintos proveedores.
- Las herramientas disponibles son escasas para la administración y ajuste del desempeño de los sistemas.
- Es importante tener en cuenta que todos los servidores deben trabajar con los mismos mecanismos de comunicación, ya que se encontrarán en un ambiente distribuido.
- Presencia de complicaciones en la persistencia y el acceso a los datos, es necesario contar con mecanismos de control de concurrencia para balancear las cargas y de esta manera mantener la consistencia de los datos.
- Realizar validaciones a nivel del cliente y del servidor para asegurar que el sistema sea seguro.
- Presencia de inconvenientes tales como: congestión en la red, cuellos de botellas, entre otros.

3.1.5 Separación de Niveles Lógicos

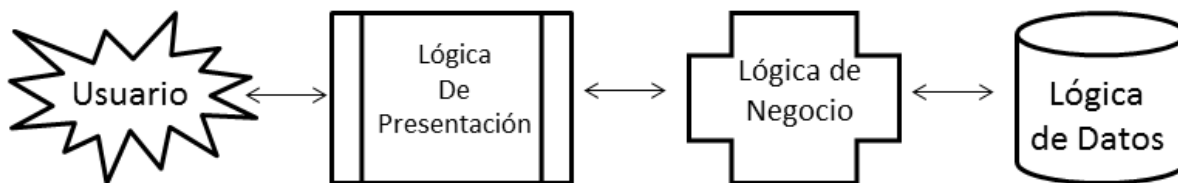


Figura 3.2: Separación de Niveles de Funciones

Como se mencionó con anterioridad entre las ventajas que posee, la arquitectura cliente-servidor permite tener diferentes niveles de abstracción, tal como lo muestra la Figura

3.2, lo que permite la separación de la lógica del sistema para así lograr realizar cambios en el sistema sin afectar el funcionamiento de la aplicación. Estos niveles son:

- Lógica de presentación: Es la capa donde se presentan los datos al usuario. La presentación de los datos es una función independiente del resto [6].
- Lógica de datos: Este nivel permite definir de qué forma se guardarán los datos y a su vez definir el acceso a ellos.
- Lógica de aplicación o negocio: Los flujos de la aplicación se pueden modificar en el tiempo, según los requerimientos del negocio. Este nivel permite realizar dichos cambios sin alterar los demás, siempre y cuando se trabaje con los mismos datos y se envíen con el mismo formato a la lógica de presentación.

3.2 Frontend

El frontend, es aquella parte visual de una aplicación web; visto en términos de la arquitectura Modelo-Vista-Controlador (MVC) es todo lo asociado a las vistas. Las tecnologías más utilizadas para ello son: HTML5, CSS3, junto a algunos frameworks tales como Bootstrap o Materializer y algunos lenguajes de scripting, como, por ejemplo: JavaScript, TypeScript, CoffeeScript, junto a algunos frameworks de JavaScript como JQuery. Cada una de estas tecnologías cumple una función específica, detalladas a continuación:

3.2.1 HTML5

HTML5 es el lenguaje de marcado predominante usado para describir contenido o data, incluye todos los elementos de HTML4 y XHTML 1.0, aunado a esto ha sido diseñado con algunos principios para asegurar su funcionamiento y lograr que sea compatible con plataformas más antiguas y navegadores.

HTML5 incluye cambios en algunos elementos de marcado, así como nuevas etiquetas para permitir a los diseñadores ser más expresivos al momento de describir el contenido de las páginas. Algunas de estas nuevas etiquetas se encuentran: article, section, header, footer, entre otras.

El término HTML5 ha sido utilizado para referirse a nuevas tecnologías y APIs, algunas de estas incluyen elementos como canvas, *offline storage*, elementos de audio y video, funcionalidad *drag-and-drop*, entre algunas otras [7].

3.2.2 CSS3

Las páginas de estilo en cascada o CSS, son un lenguaje de estilos que describe la apariencia de la página y la manera como es presentada a los usuarios. CSS3 es la última versión presentada de CSS.

Esta última versión contiene todo lo que anteriormente estaba incluido en CSS2, igualmente añade nuevas características que ayudan a los desarrolladores en la resolución de conflictos de estilos sin la necesidad de recurrir a scripts o plugins. Algunas de las características que incluyen esta nueva versión son: Soporte para selectores adicionales, efecto de sombras, bordes redondeados, animaciones, transparencias, entre otros [7].

3.2.3 JavaScript

JavaScript es un lenguaje de programación moderno que trabaja de la mano con HTML y CSS, permitiendo manejar el comportamiento de la página. Algunas de las actividades que se pueden llevar a cabo con JavaScript y otros lenguajes de scripting son:

- Atender eventos, como el click del mouse.
- Modificar dinámicamente el DOM.
- Añadir animaciones.
- Transferir data entre el servidor y el navegador mediante llamadas asíncronas.

3.3 Backend

El backend involucra todos los procesos que deben ocurrir del lado del servidor, que no son apreciables por el usuario final, proporcionando así una capa de abstracción para manejar los datos implementando toda la lógica de negocio.

Dentro del servidor se debe desarrollar un acceso a los datos que se encuentran almacenados en algún repositorio central de datos y programar los servicios o controladores para dar una respuesta a las peticiones generadas por la(s) página(s).

Hoy en día debido al auge de las aplicaciones móviles, se estila crear un API del lado del servidor para que tanto las aplicaciones web como las aplicaciones móviles consuman los mismos servicios, asegurando de esta manera que la data que se muestra a los usuarios finales sea la misma, sin importar la plataforma que se esté utilizando para acceder al sistema.

Es importante definir las medidas de seguridad que se deben tomar del lado del servidor para asegurar que los datos que entren y salgan de éste sean confiables. Dentro de las medidas de seguridad más empleadas hoy en día se encuentra la utilización de tokens al momento de realizar una petición al API, de forma que, al recibir este token, se descifra y se valida, en caso de que no sea válido o no se encuentre dentro de la petición se procede a rechazar el recurso solicitado. Estos tokens usualmente se generan a partir de una data del usuario para lograr validarlo. Otra medida de seguridad importante, es no almacenar datos cruciales de los usuarios en texto plano, aplicando un algoritmo hash sobre estos campos para asegurar que no se pueda obtener el valor inicial, ya que las funciones hash son unidireccionales.

El backend de acuerdo a la arquitectura MVC, abarca el Modelo sobre el cual se debe proveer un acceso seguro y estándar a los datos para mantener la modularidad del sistema y en caso de presentarse futuros cambios en el modelo, no se vean directamente afectados los controladores de forma drástica. Ahora bien, el backend también cumple una función y es la de proveer servicios para las Vistas de forma sencilla y esto se logra mediante los controladores. Esta capa provee un enlace entre el modelo y las vistas, se debe asegurar que, en caso de realizar cambios en la lógica de negocio no afecte las vistas.

Hoy en día se utilizan arquitecturas de software RESTful, esto asegura un conjunto de operaciones bien definidas que surgen de una correspondencia entre las operaciones básicas HTTP (POST, GET, UPDATE, DELETE) y el CRUD (Create, Read, Update, Delete) que se puede realizar sobre los registros de una base de datos. Es importante destacar, que se hace uso de los ORMs (Objeto Modelo-Relacional) dado que estos proveen una abstracción de la correspondencia entre las operaciones HTTP y las operaciones que se pueden realizar sobre los objetos de un lenguaje de programación orientado a objetos.

3.4 Sistema Manejador de Bases de Datos

Un sistema manejador de bases de datos consiste en una colección de datos interrelacionados y un conjunto de programas para acceder a dichos datos. El objetivo principal de un SMDDB es proporcionar un mecanismo para almacenar y recuperar los registros guardados en una base de datos de manera eficiente [8].

Antes de la llegada de los sistemas de gestión de bases de datos, las organizaciones almacenaban la información haciendo uso de sistemas de archivos, sin embargo, este último sistema mencionado posee ciertas desventajas o inconvenientes que los sistemas manejadores de bases datos procuran resolver. Estos inconvenientes son:

- Redundancia e inconsistencia de datos: esto se refiere a poseer el mismo dato repetido una o más veces en varios archivos o incluso en el mismo documento, esto conduce a un mayor coste de almacenamiento y con mucha certeza inconsistencia en los datos.
- Dificultad en el acceso a los datos: dado que, aunque los archivos puedan tener una estructura rígida la única forma para realizar búsquedas sobre archivos es recorriéndolos en su totalidad, lo que puede conducir a un alto coste de procesamiento y esto a su vez disminuir el rendimiento del sistema.
- Aislamiento de datos: debido a que los datos están dispersos en diferentes archivos y estos puedan estar en diferentes formatos, es difícil escribir aplicaciones que puedan recuperar los datos de manera apropiada.
- Problemas de integridad: es complicado restringir nuevos datos debido al alto costo de acceso a los datos.
- Problemas de atomicidad: todo sistema está sujeto a fallo, por lo que es crucial asegurar que una vez que un fallo ha ocurrido se puedan regresar los cambios

hasta llegar a un estado consistente, esto es difícil de lograr en un sistema de archivos convencional.

En la actualidad, el sistema gestor de bases de datos más utilizado es MySQL el cual es de código abierto, éste es lo suficientemente flexible para trabajar en cualquier ambiente incluyendo aplicaciones web. La arquitectura de éste sistema manejador, permite asegurar las propiedades ACID de toda base de datos relacional y a su vez provee ciertas funcionalidades las cuales se detallan a continuación:

- Manejo de conexiones: cuando un cliente se conecta al servidor de MySQL, éste le proporciona su propio hilo el cual reside en un núcleo del CPU.
- Seguridad: la autenticación en MySQL se basa en un usuario, un host y una contraseña, una vez el cliente se ha conectado, el servidor verifica si éste posee los privilegios necesarios para poder ejecutar determinadas consultas.
- Optimización y ejecución: cuando MySQL recibe una consulta, éste aplica cierta variedad de optimizaciones las cuales pueden incluir: reescribir el query, determinar el orden en que serán leídas las tablas, y seleccionar los índices a usar.
- Control de concurrencia: provee control de concurrencia lo que incluye bloqueos al momento de lectura y escritura de tablas.
- Transacciones: MySQL permite ejecutar un grupo de consultas y tratarlas de forma atómica, es decir, como una sola unidad de trabajo. Una transacción debe cumplir con las propiedades ACID de todo sistema manejador de bases de datos relacionales. Estas propiedades son:
 - Atomicidad: una transacción debe ser ejecutada como un solo bloque de instrucciones, es decir, se ejecuta todo o todos los cambios se revierten.
 - Consistencia: la base de datos debe moverse de un estado consistente a otro igual, es decir, se deben aplicar los cambios de exitosa y estos deben ser verificables.
 - Aislamiento: una transacción debe estar aislada de las demás, lo que quiere decir que los resultados intermedios y el resultado final de una transacción deben ser invisibles para cualquier otra transacción.
 - Durabilidad: una vez que se realicen los cambios, estos deben ser durables y permanentes.

3.5 Tipos de Aplicaciones Web

Con el constante desarrollo de las tecnologías es importante hacer uso de aplicaciones web para cubrir desde las necesidades más básicas de un cliente o empresa hasta las más complejas. Esto da lugar a distintos tipos de aplicaciones web, entre las cuales se pueden destacar:

- Aplicación Web Estática: Estas son aplicaciones web donde se muestra poca información y no suelen cambiar mucho. Desarrolladas en HTML5, CSS3 y JavaScript. Un ejemplo de este tipo de aplicación son las páginas informativas,

portafolios, curriculum digital, es común encontrar este tipo de aplicaciones web en una empresa que este iniciando y posea una aplicación estática para mostrar sus datos de contacto y lograr entrar en un mercado específico.

- **Aplicación Web Dinámica:** Son un poco más complejas, ya que requieren el uso de bases de datos para almacenar la información cambiante y actualizar los datos que surgen en el tiempo y que se muestran a los usuarios cuando acceden a las diferentes vistas de la aplicación. En la actualidad se utilizan Manejadores de Contenido, como Wordpress o Joomla o incluso se puede hacer uso de lenguajes de programación como PHP o Java, siendo estos los más comunes.
- **Tienda Virtual o Comercio Electrónico:** Este tipo de aplicaciones tienen un grado mayor de complejidad, ya que requieren la utilización de una pasarela de pago para permitir pagos con tarjetas de crédito o débito dentro de la aplicación para poder funcionar. Dentro de estas aplicaciones se pueden encontrar sitios de comercio electrónico como Amazon y Mercadolibre.
- **Portal o Foro:** Este tipo de aplicaciones web permite diferentes funcionalidades a nivel de interacción de usuarios, ya que permite establecer chats, crear fotos, noticias de innovación, buscadores, entre otros.
- **Consolas Administrativas:** Son aplicaciones web que se desarrollan con la finalidad de realizar operaciones CRUD (Create, Read, Update, Delete) sobre los registros almacenados en bases de datos, su utilidad va en miras de manejar configuraciones de otras aplicaciones tanto web como móviles en tiempo real, sin la necesidad de realizar cambios manualmente ni publicar una nueva versión, en caso de una aplicación móvil.

3.6 Frameworks y Lenguajes de Programación

Actualmente debido a la cantidad de aplicaciones web que se desarrollan de forma vertiginosa a nivel mundial, se ha procurado crear frameworks en distintos lenguajes de programación que van desde lenguajes de alto nivel como JavaScript hasta lenguajes como Python y C#. Estos frameworks, en su mayoría, ofrecen un desarrollo ágil y rápido gracias a la cantidad de facilidades que ofrecen, como la correspondencia de objeto-relacional (ORM por sus siglas en inglés), que permite realizar operaciones a nivel de base de datos como si fuesen objetos de un lenguaje orientado a objetos, sin la necesidad de realizar consultas manualmente. Sin embargo, estos también permiten la creación de consultas más específicas. Dentro de los lenguajes más utilizados a nivel empresarial, se encuentran PHP por popularidad y Java por su portabilidad y en ambos casos, por la cantidad de documentación y la capacidad de conexión con otros sistemas o incluso con distintos manejadores de base de datos. En éste caso de estudio se usará Java.

3.7 Java

Java es un lenguaje orientado a objetos de propósito general, el cual comenzó a ser conocido como un lenguaje de programación de applets que se ejecutan en el entorno de un navegador web, pudiendo utilizarse para construir cualquier tipo de proyecto.

La principal característica de Java es la de ser un lenguaje compilado e interpretado. Todo programa en Java ha de compilarse y el código que se genera, llamado bytecodes, es interpretado por una máquina virtual. De este modo se consigue la independencia de la máquina real, el código compilado se ejecuta en la máquina virtual la cual si depende de la plataforma.

En el diseño de Java se prestó especial atención a la seguridad. Existen varios niveles de seguridad en Java, desde el ámbito del programador, hasta el ámbito de la ejecución en la máquina virtual. Con respecto al programador, Java realiza comprobación estricta de tipos durante la compilación, evitando con ello problemas tales como el desbordamiento de la pila. Sin embargo, es durante la ejecución donde se encuentra el método adecuado según el tipo de la clase receptora del mensaje; siendo posible forzar un enlace estático declarando un método como *final*. Todas las instancias de una clase se crean con el operador *new()*, de manera que un recolector de basura se encarga de liberar la memoria ocupada por los objetos que ya no están referenciados.

La máquina virtual de Java gestiona la memoria dinámicamente. Igualmente, el programador siempre está obligado a tratar las posibles excepciones que se produzcan en tiempo de ejecución. Dentro de la amplia gama de características de Java se tiene que esta preparado para la programación concurrente sin necesidad de utilizar ningún tipo de biblioteca. Finalmente, Java posee un gestor de seguridad con el que se puede restringir el acceso a los recursos del sistema [9].

3.7.1 Java en Aplicaciones Web

La web dinámica se ha desarrollado desde un sistema de información distribuido hypermedia basado en red que ofrecía información estática hasta un conjunto de portales y aplicaciones en Internet que ofrecen un conjunto variado de servicios.

Las soluciones de primera generación incluyeron CGI, que es un mecanismo para ejecutar programas externos a un servidor web. En una segunda instancia vienen los servidores que proveían plugins o incluso APIs, pero estos eran muy específicos, posteriormente Microsoft proporcionó las páginas activas del servidor (ASP) las cuales facilitaron la creación de contenido dinámico, sin embargo, tenían una limitante ya que su solución sólo funcionaba con Microsoft. Otra tecnología de segunda generación son los Servlets, estos hacen que sea más sencillo escribir aplicaciones del lado del servidor usando la tecnología Java. El problema con los CGIs o los servlets, sin embargo, se debe seguir el ciclo de vida de escribir, compilar y desplegar.

Las páginas JSP son una solución de tercera generación que se pueden combinar fácilmente con algunas soluciones de la segunda generación, creando el contenido dinámico, y haciendo más expedito construir las aplicaciones basadas en web que trabajan con una variedad de otras tecnologías: servidores web, navegadores web, servidores de aplicación y otras herramientas de desarrollo. La tecnología JSP nos permite poner segmentos de código servlet directamente dentro de una página HTML estática. Cuando el navegador carga una página JSP, se ejecuta el código del servlet y el servidor de aplicaciones crea, compila, carga y ejecuta un servlet en segundo plano para ejecutar los segmentos de código servlet y devolver una página HTML o imprimir un informe XML.

Los programadores de Java mediante la especificación J2EE escriben componentes de software auto-contenidos y se ensamblan dentro de una aplicación J2EE. La especificación define los siguientes componentes:

- Componentes de aplicación de clientes: Páginas JSP.
- Componentes JavaBeans Enterprise: Clase de Negocios (EJBS).
- Componentes servlet.
- Applets: pequeñas aplicaciones del lado del cliente.

3.7.2 **Servlet**

Los servlets permiten seguir la trayectoria de un usuario, esto se puede lograr mediante el uso de Cookies, Session Tracking y Requests. Este es especialmente útil en comercios web o aplicaciones que requieran mantener un estado conversacional con el cliente.

Una de las tareas más frecuentemente realizadas por los servlets es la conexión a base de datos mediante JDBC. El JDBC es una parte del API de Java que proporciona clases para conectarse con bases de datos.

3.7.3 **Enterprise JavaBeans (EJB)**

Un Enterprise JavaBean (EJB), es un componente del lado del servidor que encapsula la lógica del negocio de una aplicación. En cualquier aplicación, los beans enterprise implementan los métodos de la lógica del negocio, que pueden ser invocados por clientes remotos para acceder a los servicios más importantes proporcionados por la aplicación. Los EJBs simplifican el desarrollo de grandes aplicaciones empresariales seguras y distribuidas por las siguientes razones:

- El contenedor de EJBs proporciona servicios a nivel del sistema como control de transacciones y autorizaciones de seguridad.
- Los desarrolladores no deben preocuparse por la conectividad con la base de datos, solo deben preocuparse en el desarrollo de la lógica del negocio.
- Los EJBs son portables, por lo tanto, los desarrolladores pueden crear aplicaciones de beans ya existentes.

3.7.4 Servlet vs Enterprise JavaBean

Hay una diferencia importante entre ambos componentes, y esta es que los EJBs no pueden aceptar peticiones HTTP, es decir, peticiones provenientes directamente de un cliente web mientras que los servlets si pueden [10]. Nótese esto en la Figura 3.3.

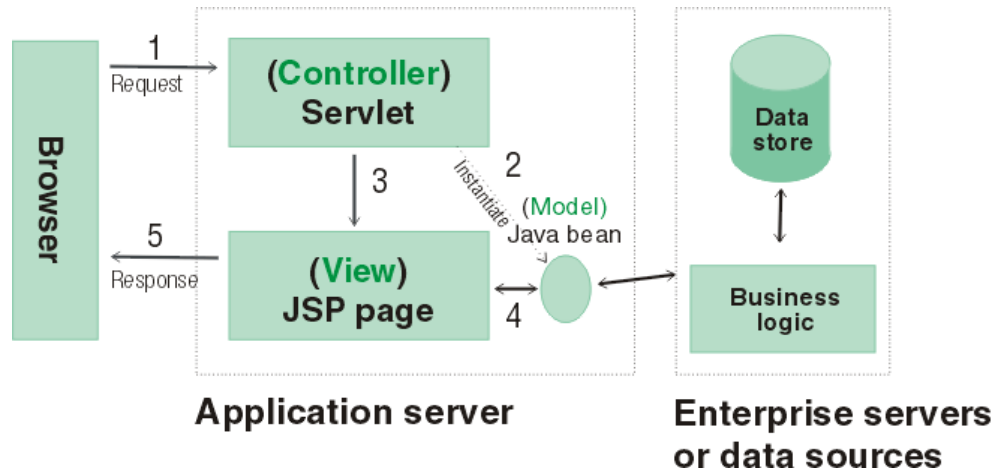


Figura 3.3: Servlet vs EJBs

4. Aplicaciones Móviles

Primero se debe definir lo que son teléfonos móviles o celulares para saber cómo se originaron las aplicaciones móviles, estos son dispositivos electrónicos de comunicación, normalmente de diseño reducido que usa ondas de radio para conectarse a la red telefónica, es decir, que la realización de llamadas no es dependiente de ningún terminal fijo y que no requiere de ningún tipo de cableado para llevar a cabo la conexión a la red telefónica [11]. Sin embargo, esta definición es un poco básica, más apegada a los que se conoce hoy en día como *feature phones* (Figura 4.1) que para su época ya contaban con aplicaciones o *apps* básicas enfocadas en mejorar la productividad personal como calculadoras, alarmas, cronómetros, calendarios y clientes de correo.



Figura 4.1: Teléfonos Básicos Conocidos como Features Phones

En la actualidad lo primero que nos viene a la mente al pensar en teléfono móvil es lo que se conoce como *smartphones* o teléfonos inteligentes. Estos dispositivos soportan más funciones que un teléfono común debido a que acceden continuamente a Internet [12]. Este cambio en la percepción de lo que es un teléfono se produjo gracias a la llegada del *iPhone* en el 2007, ya que con él se generaron nuevos modelos de negocio que hicieron las aplicaciones más rentables tanto para los desarrolladores como para los mercados de aplicaciones, como App Store, Google Play y Windows Store [13].

Ahora que se está en contexto se puede decir que una aplicación móvil no es más que un *software* diseñado para ser ejecutado en teléfonos inteligentes, tabletas y cada día en más dispositivos móviles como relojes y automóviles.

Al momento de plantearnos el desarrollo de una aplicación móvil se tiene que definir el enfoque de desarrollo, ya sea una nativa, Web o híbrida; dado que influenciará el resultado. Cada enfoque posee limitaciones y beneficios inherentes que se exploran a continuación.

4.1 Aplicaciones Nativas

Las aplicaciones nativas son desarrolladas específicamente para ser ejecutadas en el sistema operativo en el cual es instalada. Tras la instalación y ejecución, la aplicación nativa se conecta directamente al sistema operativo móvil, sin necesidad de intermediarios ni contenedores. La aplicación puede acceder sin restricción a todas las APIs provistas por el sistema operativo.

Para poder desarrollar aplicaciones nativas, cada sistema operativo ofrece un conjunto de herramientas y recursos llamados *SDK* o Kit de Desarrollo de Software, por sus siglas en inglés. Si bien el proceso de desarrollo suele ser similar para los diferentes sistemas operativos, el SDK es específico para cada plataforma. Las distintas herramientas, lenguajes de programación y canales de distribución para los dos sistemas operativos más usados son expuestas en la Tabla 4.1.

	Apple iOS	Android
Lenguajes	Objective-C, C y C++	Java, algunos en C y C++
Herramientas	Xcode	Android Studio
Tiendas	Apple App Store	Google Play

Tabla 4.1: Lenguajes, Herramientas y Canales de Distribución de Apple iOS y Android

Gracias a estas diferencias entre las distintas plataformas se genera la principal desventaja del enfoque de desarrollo nativo. El código escrito para una plataforma no puede ser reutilizado para otra, por lo que el desarrollo y mantenimiento de las aplicaciones nativas para múltiples sistemas operativos se torna en una ardua y costosa tarea [14]. A pesar de esta costosa desventaja muchas empresas eligen este enfoque de desarrollo, debido a las APIs y guías de estilo que ofrece cada sistema operativo, las cuales se exponen para entender mejor este enfoque.

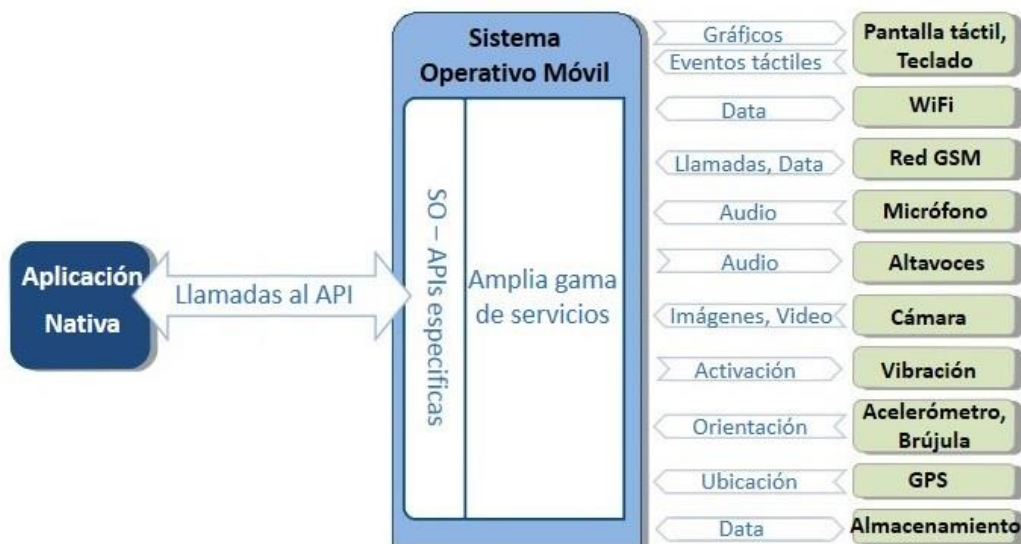


Figura 4.2: Aplicación Nativa: Interacción con el Dispositivo Móvil

Primero se tienen los *APIs de bajo nivel*, con las cuales se tiene acceso a todos los elementos de hardware disponibles. Se puede interactuar directamente con la pantalla táctil o el teclado, y así mostrar gráficos, conectarse a redes, procesar audio recibido por el micrófono, reproducir sonidos por el altavoz o auriculares, o recibir imágenes y videos de la cámara. También acceder al GPS, recibir información sobre orientación y, por supuesto, leer y escribir archivos en el disco como se puede observar en la Figura 4.2.

Además de proporcionar los servicios de bajo nivel para acceder al hardware, los sistemas operativos móviles ofrecen un conjunto de *APIs de alto nivel* accesible para aplicaciones nativas, lo que permite acceder a muchos servicios que son importantes para la experiencia móvil del usuario. Dichos servicios incluyen procesos tales como navegar por Internet, gestionar el calendario y los contactos, acceder álbumes de fotos, así como hacer llamadas telefónicas o enviar y recibir mensajes de texto. Otras APIs permiten que las aplicaciones accedan a servicios en la nube ofrecidos por el distribuidor del sistema operativo, como notificaciones push o compras en tiendas de aplicaciones.

Cada sistema operativo móvil viene con una gama única de componentes para la interfaz del usuario, como botones, campos de introducción de datos, menús, barras de pestañas, cuadros de diálogo, etc. Las aplicaciones que utilizan estos componentes heredan las características y funciones de ese sistema operativo móvil específico, lo que genera una mejor experiencia para el usuario.

Si bien las APIs son específicas del sistema operativo, agregan bastante complejidad y costos para el desarrollo de múltiples aplicaciones nativas, estos elementos son la única manera de crear aplicaciones móviles completas que utilicen todas las funcionalidades que los dispositivos móviles modernos tienen para ofrecer [14].

4.2 Aplicaciones Móviles basadas en la Web

En la actualidad los dispositivos móviles cuentan con poderosos navegadores que soportan las nuevas funcionalidades de HTML5, Cascading Style Sheets 3 (CSS3) y JavaScript. Con los últimos avances de HTML5, esta tecnología dejó de ser un “*Lenguaje de definición de páginas*” para convertirse en un estándar de desarrollo de aplicaciones basadas en navegador.

Dentro de las aplicaciones basadas en Web existen dos enfoques muy distintos que se pueden distinguir. Primero están los sitios Webs adaptables, mejor conocidos por su nombre en inglés “*Responsive Web*”, donde los sitios reconocen cuando se accede a ellos a través de un dispositivo móvil y preparan páginas HTML que han sido diseñadas para brindar una “experiencia táctil” cómoda en una pantalla pequeña. El otro enfoque va más allá y mejora la experiencia del usuario desarrollando un sitio Web que se asemeja a una aplicación nativa y se puede ejecutar con un acceso directo igual al que usa para lanzar las aplicaciones nativas. La Tabla 4.2 compara brevemente ambos enfoques.

Característica	Apl. Web solo móviles	Sitios Web solo móviles
Herramientas y conocimientos	Escritas totalmente en HTML, CCS y JavaScript	Escritas totalmente en HTML, CCS y JavaScript
Ejecución	Acceso directo “ <i>instalado</i> ”, lanzado mediante apl. nativa	Navegando mediante URL (Uniform Resource Locator)
Experiencia del usuario	Touch-friendly, interactive UI	UI mediante navegación entre páginas que muestran datos estáticos
Desempeño	UI reside localmente, aplicación con capacidad de respuesta y acceso offline	Todo el código se ejecuta desde un servidor, el rendimiento depende de la red

Tabla 4.2: Comparación de los Enfoques de Aplicaciones basadas en Web

Las aplicaciones móviles basadas en Web son muy prometedoras ya que, para ayudar a los desarrolladores a construir la interfaz de usuario, se han desarrollado muchas librerías JavaScript, tales como dojox.mobile, Sencha Touch y jQuery Mobile, las cuales generan interfaces de usuario con una apariencia similar a las aplicaciones nativas. Ambos enfoques se ejecutan completamente dentro del navegador del dispositivo móvil y utilizan las últimas características de JavaScript, CSS y HTML5 disponibles.

La principal ventaja de una aplicación Web es su soporte para múltiples plataformas y el bajo costo de desarrollo. La mayoría de los navegadores móviles utilizan WebKit como motor de navegación Web, que es un proyecto de código abierto conducido principalmente por Google y Apple, y ofrece la más completa implementación de HTML5 disponible en la actualidad.

A pesar del prometedor potencial de las tecnologías web en el espacio móvil, estas aún poseen limitaciones. A diferencia de las aplicaciones nativas que se ejecutan independientemente y se conectan directamente con el sistema operativo, las aplicaciones Web se ejecutan dentro del navegador.

El navegador es una aplicación nativa que tiene acceso directo a los APIs del sistema operativo, pero muy pocas de esos APIs pueden ser utilizadas por las aplicaciones Web que se ejecutan dentro del mismo. Mientras que las aplicaciones nativas tienen acceso completo al dispositivo, en las aplicaciones web muchas funcionalidades no están disponibles o sólo en forma parcial. Aunque gracias a los grandes avances en el desarrollo de HTML es posible que esto cambie en poco tiempo [14].

4.3 Aplicaciones Híbridas

En el enfoque híbrido, se mezcla el desarrollo nativo con la tecnología Web. Se desarrolla para múltiples plataformas haciendo uso de las tecnologías web, a la vez que se mantiene el acceso a las APIs nativas cuando es necesario. Esto es gracias a que la porción nativa de la aplicación funciona como puente entre el navegador y las APIs del dispositivo, permitiendo así que la aplicación híbrida aproveche todas las características que ofrecen los dispositivos [14].

Al momento de empezar un desarrollo bajo este enfoque se puede optar por codificar un puente propio o utilizar soluciones existentes como PhoneGap, que es un framework de código abierto que provee una interfaz JavaScript para el acceso a funcionalidades del dispositivo que son iguales en todos los sistemas operativos, para tener una mejor idea se puede observar la Figura 4.3.

La porción web de la aplicación puede ser un conjunto de recursos web alojados en un servidor o estos recursos pueden estar incorporados en el código de aplicación y/o almacenados localmente. Si los recursos Web están alojados en un servidor se pueden realizar pequeñas actualizaciones sin necesidad de pasar por las tiendas. Este enfoque elimina cualquier disponibilidad offline, ya que el contenido no es accesible cuando el dispositivo no está conectado a la red.

Por otro lado, almacenar el código web localmente puede mejorar el desempeño y la accesibilidad, pero no acepta actualizaciones remotas. Lo mejor de ambos mundos se puede lograr combinando los dos enfoques, alojando los recursos HTML en un servidor Web mejorando la flexibilidad, y al mismo tiempo tenerlos localmente en el dispositivo móvil para mejorar el desempeño.

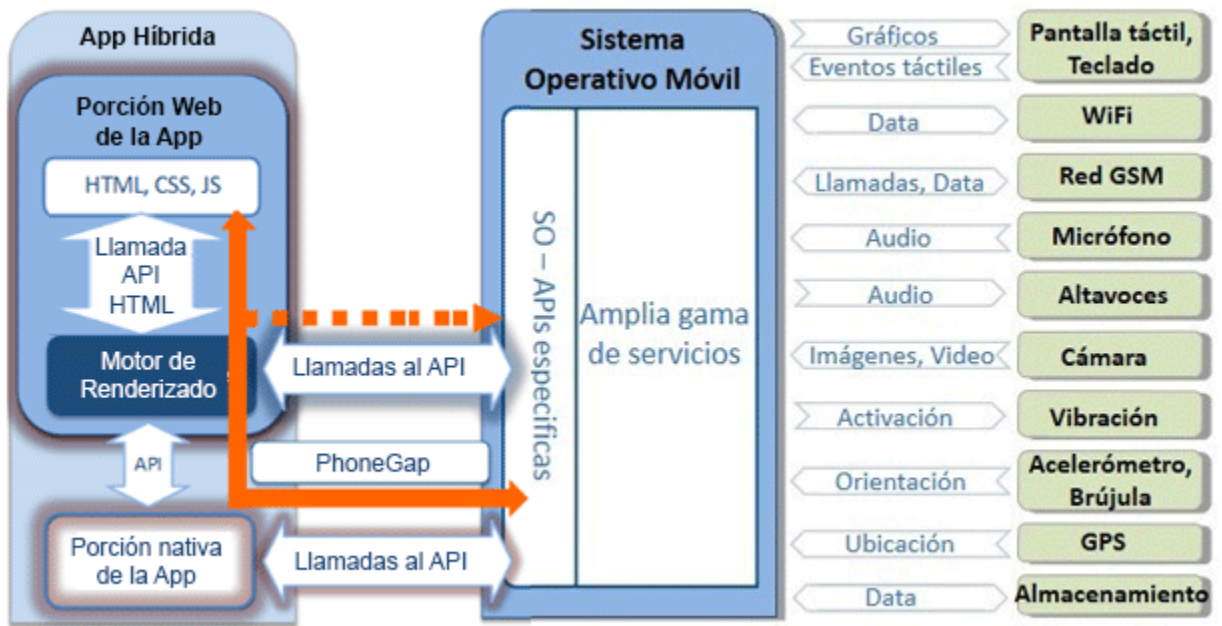


Figura 4.3: Aplicación Híbrida: Interacción con el Dispositivo Móvil

5. Sistema Operativo Android

El término Android se utiliza comúnmente para referirse al sistema operativo, eso está bien, pero incompleto. Android es una plataforma de software de código abierto que incluye el sistema operativo que está basado en Linux, sin embargo, está específicamente diseñado para ejecutarse en dispositivos. Aunado a esto, la plataforma Android incluye un completo framework que te permite crear y desplegar sus propias aplicaciones personalizadas [15]. Android fue desarrollado inicialmente por Android Inc, empresa que fue adquirida por Google en 2005 y fue presentado en 2007 en conjunto con la fundación Open Handset Alliance que es un consorcio de empresas de hardware, software y telecomunicaciones enfocados en construir estándares abiertos para dispositivos móviles. Actualmente, mantenimiento y desarrollo de Android es llevado a cabo por el Android Open Source Project (AOSP).

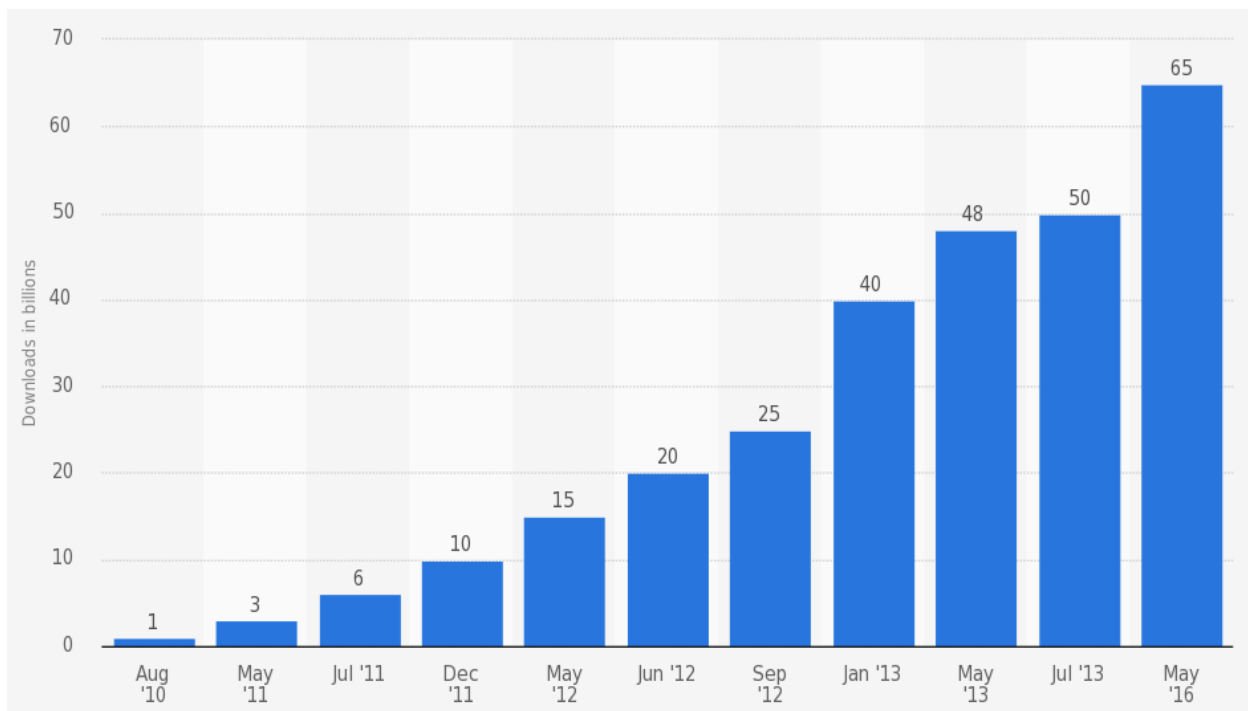


Figura 5.1: Número Acumulado de Aplicaciones Descargadas de Google Play

Android cuenta con una tienda llamada Google Play Store, el cual les permite a los desarrolladores tener el canal de distribución digital de aplicaciones más importante de la actualidad. En donde, como muestra la Figura 5.1, contó en los últimos tres años con 15 billones de descargas. Esta plataforma les permite a los usuarios navegar y descargar cualquier aplicación entre las más de 2 millones que hay disponibles para febrero de 2016 como se observa en la Figura 5.2.

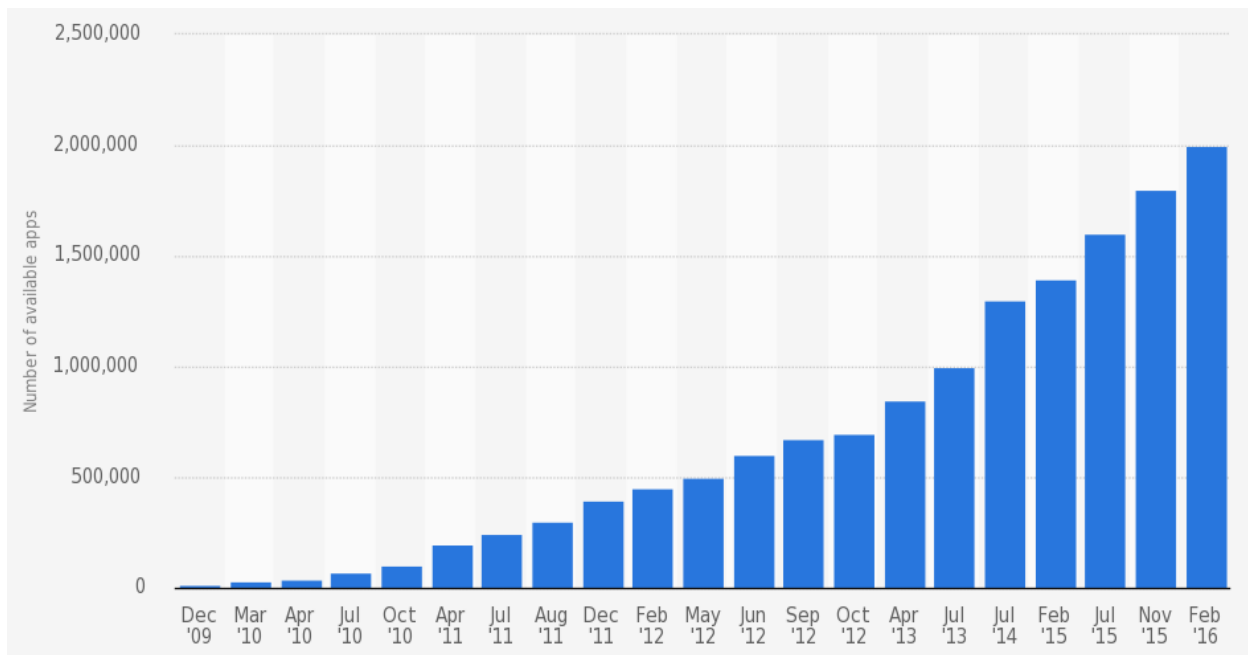


Figura 5.2: Número de Aplicaciones Disponibles en la Tienda de Google Play

El ambiente operativo de Android puede ser etiquetado como una plataforma abierta de desarrollo móvil, una referencia de diseño para hardware de dispositivos móviles, un sistema impulsado por el kernel modificado de Linux 2.6, un entorno en tiempo de ejecución y un framework de aplicaciones e interfaces de usuarios [16].

5.1 Arquitectura de Android

La Figura 5.3 engloba las capas que conforman la arquitectura de Android. El kernel modificado de Linux opera como una capa de abstracción del hardware, y provee controladores del dispositivo, manejo de memoria, manejo de procesos, así como funcionalidades de redes. La capa de librerías se interconecta a través de Java, es aquí en donde se encuentra Bionic, que es la librería estándar de C desarrollada por Google específicamente para Android. El administrador de superficies se encarga de manejar las ventanas de interfaz de usuario (UI). Android runtime es la capa que contiene la Máquina Virtual Dalvik (DVM) y las librerías del núcleo como Java. La mayoría de las funcionalidades disponibles en Android provienen de las librerías del núcleo.

El armazón de aplicaciones aloja las APIs. En esta capa, el administrador de actividades gobierna el ciclo de vida de las aplicaciones. El proveedor de contenidos le permite a las aplicaciones acceder a datos de otras aplicaciones o compartir sus propios datos. El administrador de recursos provee acceso a recursos sin código, como imágenes, mientras que el administrador de notificaciones le permite a las aplicaciones generar alertas personalizadas. En la parte superior del armazón se tienen las aplicaciones auto-contenidas y las de usuario, en donde cada una se ejecuta con su propio espacio de proceso dentro de su propia instancia de la Máquina Virtual Dalvik.

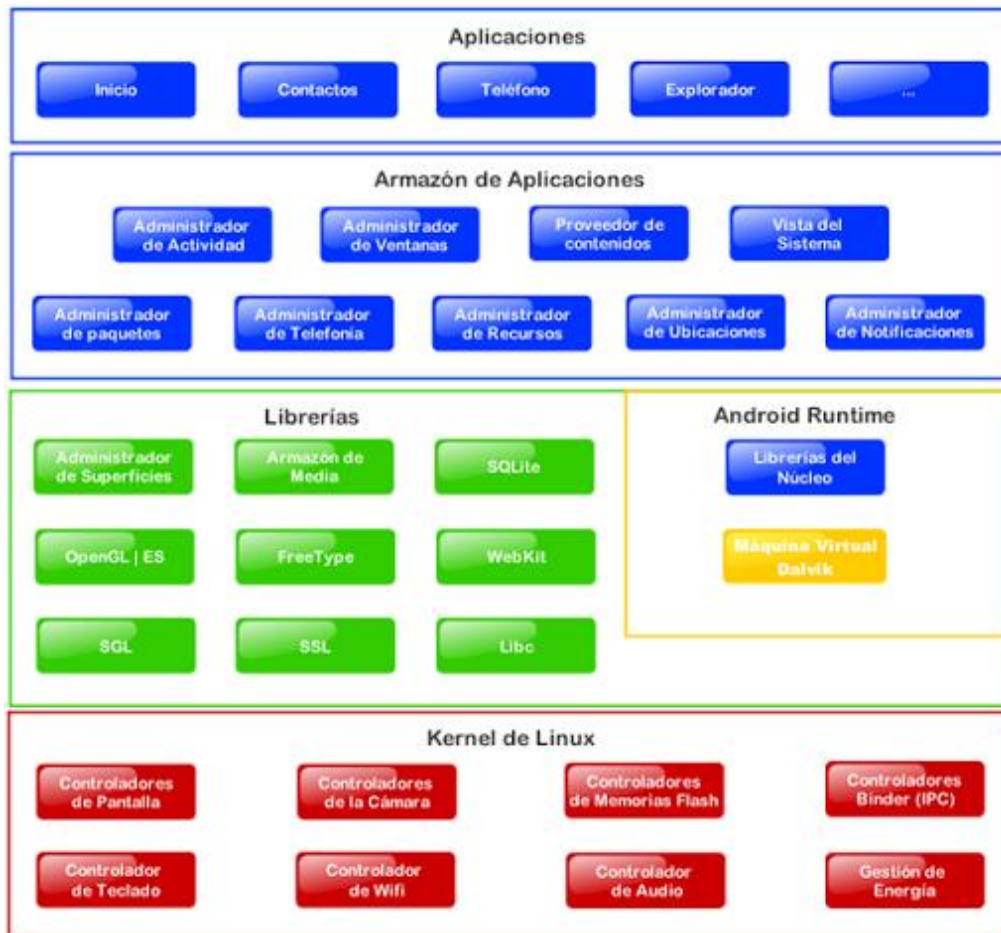


Figura 5.3: Arquitectura de Android

5.1.1 Máquina Virtual Dalvik

Los sistemas basados en Android utilizan su propia máquina virtual, también conocida como Máquina Virtual Dalvik o DVM por sus siglas en inglés. La DVM cuenta con un *byte-code* especial, por lo que el *byte-code* de Java no se puede ejecutar directamente. Android posee una herramienta que permite convertir las clases de Java en archivos Dalvik ejecutables o Dex. La implementación de la Máquina Virtual Dalvik está optimizada para tener el rendimiento más eficiente y efectivo que sea posible en dispositivos móviles normalmente equipados con procesadores modestos de dos o cuatro núcleos, recursos limitados de memoria, sin espacio de intercambio y batería limitada. La DVM se implementó de manera que permite a un dispositivo ejecutar múltiples máquinas virtuales de manera bastante eficiente. También debe señalar que la DVM se basa en el kernel de Linux modificado para potenciar el manejo de procesos y la gestión de memoria de bajo nivel.

Con la versión 2.2 de Android, se llevaron a cabo algunos cambios importantes en la infraestructura de la Máquina Virtual de Java. Hasta esta versión la JVM era un *intérprete*

real, muy parecido a la solución original de la JVM en la versión 1.0 de Java. Aunque las soluciones de Android siempre reflejaban ser un intérprete muy eficiente, seguía siendo un intérprete, por lo que no se generaba ningún código nativo. Con el lanzamiento de Android 2.2, se incorporó a la pila de soluciones un compilador llamado JIT (*just-in-time*), que traduce el byte-code de Dalvik en un código de máquina más eficiente parecido al código del compilador de C. A medida que pase el tiempo, otras funciones serán agregadas al JIT y al colector de basura que potenciarán el rendimiento del sistema [16].

5.1.2 Conjunto de Instrucciones

La mayoría de los dispositivos basados en Linux 2.6 utilizan una arquitectura basada en sistemas x86, mientras que la mayoría de los dispositivos móviles son basados en productos ARM. Mientras que ARM representa una arquitectura de 32-bits con un conjunto reducido de intrusiones (RISC), los sistemas x86 se basan principalmente en arquitecturas con un conjunto complejo de instrucciones. Se puede decir que los sistemas ARM (RISC) ejecutan instrucciones más simples, pero en mayor cantidad que los sistemas x86 (CISC). En comparación con los sistemas x86, los sistemas ARM tienen un mayor enfoque en el consumo bajo de energía lo cual es más apropiado para los dispositivos móviles.

Como se ha mencionado, la memoria es un recurso limitado en los dispositivos móviles. ARM se encarga de esto proporcionando un segundo conjunto de instrucciones de 16-bits que puede ser utilizado en conjunto con el conjunto de instrucciones regular de 32-bits. Este conjunto extra de instrucciones puede reducir el código hasta un tercio a costa de algunas limitaciones de rendimiento.

En la mayoría de los primeros dispositivos Android incorporaban el mismo procesador ARM Qualcomm, por lo tanto, su rendimiento era muy similar. Con la llegada del Motorola Droid se introdujo una nueva generación de chips con soporte para procesadores gráficos. Hoy en día, básicamente existen tres conjuntos de chips que están usando los dispositivos con Android. Para ilustrarlo, HTC está usando el Qualcomm Snapdragon, Motorola el Texas Instruments OMAP, mientras que Samsung ha diseñado su propio chipset llamado Hummingbird. Vale destacar que los tres procesadores están basados en la arquitectura ARM Cortex-v8 con ajustes específicos de cada proveedor que ofrece características únicas [16].

5.1.3 Kernel y Proceso de Inicio

A pesar que Android está basado en Linux 2.6, este no usa un kernel estándar de Linux. Algunas de las mejoras del kernel de Android son:

- Controlador de alarmas (provee un temporizador para despertar el dispositivo).
- Controlador de memoria compartida.
- Binder (para la comunicación entre procesos).

- Manejador de energía.
- Depurador de kernel

Durante el encendido de Android, el componente Linux de Android llama al proceso *init* (nada inusual comparado con kernel estándar de Linux). El proceso *init* accede a los archivos *init.rc* e *init.device.rc*, este último es específico para cada dispositivo, después es llamado el proceso etiquetado como *zygote*. El proceso *zygote* carga las clases núcleo de Java y ejecuta los primeros pasos de procesamiento. Estas clases pueden ser reutilizadas por las aplicaciones de Android, por lo tanto, este paso agiliza el proceso de inicio global. Después del proceso de carga inicial, *zygote* espera en un socket por futuras peticiones.

Cada aplicación de Android se ejecuta en su propio ambiente de proceso. Un controlador especial etiqueta al *binder* para permitir la comunicación entre procesos (de manera eficiente). Los objetos son almacenados en la memoria compartida, con lo que se optimiza la comunicación entre procesos, siendo menor la cantidad de data que debe ser transmitida. Comparado con la mayoría de los ambientes UNIX, Android no tiene ningún espacio de intercambio, por lo tanto, la memoria virtual es gobernada por la cantidad de memoria física disponible.

5.1.4 Librería Bionic

Android incorpora su propia librería C llamada *Bionic*, esta librería no es compatible con la librería *glibc* de Linux. En comparación con *glibc*, *Bionic* utiliza menos memoria. Para ilustrarnos, la librería *Bionic* incorpora un manejo especial de hilos que primero, mejora el consumo de memoria de cada hilo, y segundo, reduce el tiempo de creación de cada hilo. Android provee llamadas al kernel en tiempo de ejecución, por lo que el comportamiento de este puede ser alterado dinámicamente por los procesos/hilos que tengan los permisos necesarios [16]. Para generalizar, la librería C de Android fue construida para operar bajo unas condiciones limitadas de CPU y memoria comunes en la plataforma de Android. Además, se diseñaron e implementaron especiales consideraciones de seguridad para mantener la integridad del sistema.

5.1.5 Almacenamiento de Datos y Sistema de Archivos

Cuando se trata de armar y configurar un dispositivo móvil, los discos duros tradicionales son muy grandes, frágiles y consumen mucha energía para ser útiles. Por otro lado, las memorias flash normalmente ofrecen un tiempo de acceso a lectura relativamente rápido, así como mayor resistencia a golpes en comparación a los discos duros. Dos de las memorias flash más comunes son las NAND y NOR, por lo general, las basadas en NOR proveen baja densidad y se caracteriza por ser relativamente lenta para escribir y rápida para leer. Por otro lado, las basadas en NAND ofrecen bajo costo, alta densidad y suelen ser relativamente rápidas para escribir y lentas para leer. Algunos sistemas ofrecen

dispositivos flash basados en NAND para almacenar los datos y componentes basados en NOR para el código en el ambiente de ejecución.

Desde el punto de vista del sistema de archivos, en la versión 2.3 de Android, se comenzó a utilizar el bien conocido *ext4*. Antes de sistema de archivos *ext4*, normalmente Android empleaba *YAFFS* (llamado así por sus siglas en inglés, *yet another flas file system*), que es conocido como el primer sistema de archivos NAND optimizado en Linux.

5.1.6 Administración de Energía

Cuando se habla de dispositivos móviles, la administración de la energía es sumamente importante. Sin embargo, gracias a la creciente demanda de energía en los sistemas informáticos de hoy en día, el manejo de la energía en cualquier sistema de TI con cualquier sistema operativo necesita un buen manejo de energía. Por ejemplo, para reducir y gestionar el consumo de energía, los sistemas basados en Linux ofrecen característica de ahorro de energía como *clock gating*, escalado de voltaje, activando el modo dormir, o deshabilitando la memoria cache. Cada uno de estas técnicas reduce el consumo de energía, pero con un costo de latencia en el sistema. Los sistemas basados en Android poseen su propia infraestructura para la administración de la energía, llamada *Power Manager*, que fue diseñado para la creencia que el procesador no debe consumir energía mientras no hay procesos o servicios que requieran energía.

5.2 Aplicaciones de Android

Las aplicaciones de Android se empaquetan en un archivo *.apk* (Android Package) a través del *Android Asset Packaging Tool* (AAPT). Para coordinar el proceso de desarrollo, Google provee unas herramientas de desarrollo en Android o ADT por sus siglas en Ingles. El ADT agiliza la conversión de los archivos *class* a *dex*, y genera el archivo *.apk* durante el despliegue. De una manera muy sencilla, se puede decir que las aplicaciones de Android esta compuestas por:

- Actividades
- Intentos
- Servicios
- Proveedor de contenidos

5.2.1 Actividad

Una Actividad es un componente que provee una pantalla con la que el usuario puede interactuar para realizar una determinada acción, como marcar un número telefónico, tomar una foto, mandar un correo electrónico, o ver un mapa. A cada actividad se le asigna una ventana en la cual dibuja la interfaz de usuario. Normalmente la ventana cubre la pantalla, pero podría ser más pequeña y flotar sobre otra ventada.

Una aplicación por lo general se compone de varias actividades que están ligadas entre sí. Por lo general, en una aplicación se debe especificar una actividad como la actividad principal (*main activity* en inglés), la cual es presentada al usuario al iniciar la aplicación por primera vez. Cada actividad se puede iniciar otra actividad con el fin de realizar diferentes acciones. Cada vez que se inicia una nueva actividad, se detiene la actividad anterior, pero el sistema conserva la actividad en una pila (*back stack*). Cuando se inicia una nueva actividad, se inserta en la pila y se le presenta al usuario. La pila sigue el mecanismo básico de “*último en entrar, primero en salir*”, por tanto, cuando el usuario termina con la actividad actual y presiona el botón atrás, entonces se extrae la actividad de la pila (y se destruye) y reanuda la actividad anterior.

Cuando se detiene una actividad debido a una nueva actividad es iniciada, se notifica el cambio en el estado del ciclo de vida a través de métodos de respuesta (*callback*) de la actividad. Hay varios métodos de respuesta que podría recibir una actividad ocasionado por un cambio en su estado, si el sistema la crea, detiene, reanuda, o destruye; Cada uno de los métodos de respuesta ofrece la oportunidad de realizar un trabajo específico y apropiado para el cambio de estado. Por ejemplo, cuando se detuvo, su actividad debe liberar todos los objetos grandes, tales como bases de datos de red o conexiones. Cuando se reanuda la actividad, se puede volver a adquirir los recursos necesarios y reanudar las acciones que fueron interrumpidos. Estas transiciones de estado son parte del ciclo de vida de la actividad [17].

Durante la vida de una actividad, el sistema llama a un conjunto de métodos del ciclo de vida en una secuencia similar a una pirámide escalonada. Es decir, cada etapa del ciclo de vida de una actividad es un paso independiente en la pirámide. A medida que el sistema crea una nueva actividad, cada método de retorno mueve el estado de la actividad un paso hacia la parte superior. La parte superior de la pirámide es el punto en el que la actividad se ejecuta en el primer plano y el usuario puede interactuar con ella.

A medida que el usuario comienza a abandonar la actividad, el sistema llama a otros métodos que mueven el estado de la actividad hacia abajo la pirámide con el fin de dismantelar la actividad. En algunos casos, la actividad se moverá sólo parcialmente hacia abajo en la pirámide (por ejemplo, cuando el usuario cambia a otra aplicación) y espera, a partir de ese momento la actividad se puede mover de nuevo a la parte superior (si el usuario vuelve a la actividad) y se reanuda donde el usuario la dejó.

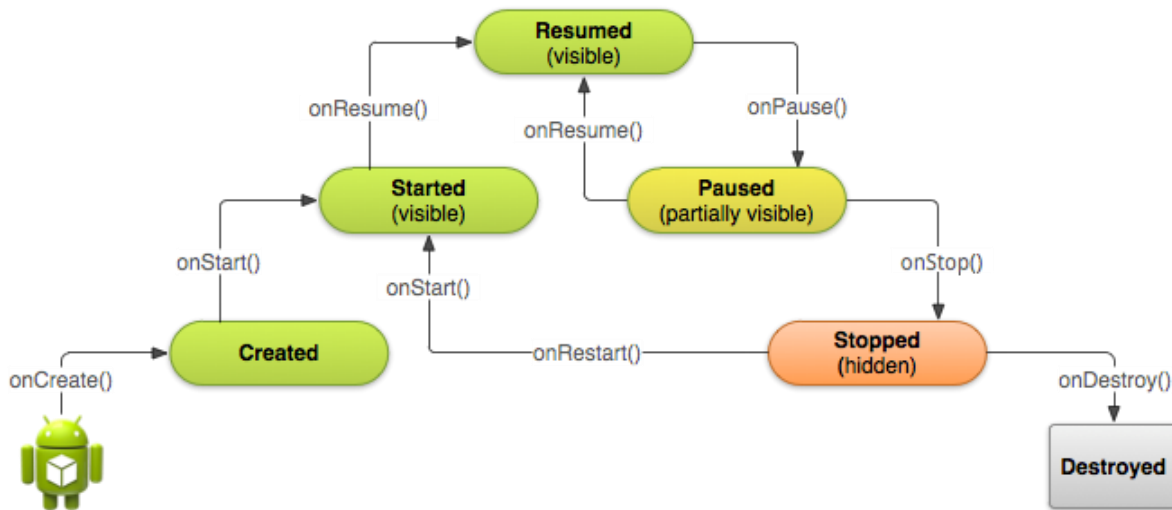


Figura 5.4: Ciclo de Vida de una Actividad

La Figura 5.4 es una ilustración simplificada del ciclo de vida de una actividad, expresado como una pirámide escalonada. Esto demuestra cómo, por cada método de retorno la actividad da un paso hacia el tope hasta llegar estado “*reanudado*” en la parte superior, también hay un método retorno que lleva a la actividad a dar un paso hacia abajo. La actividad también puede volver al estado reanudado desde el estado de pausa y se detuvo [18].

5.2.2 Servicio

Un servicio es un componente de aplicación que puede realizar operaciones de larga ejecución en segundo plano y no proporciona una interfaz de usuario. Otro componente de aplicación puede iniciar un servicio y seguirá funcionando en segundo plano, incluso si el usuario cambia a otra aplicación. Además, un componente puede vincularse a un servicio para interactuar con él e incluso realizar la comunicación entre procesos (IPC). Por ejemplo, un servicio puede manejar las transacciones de red, reproducir música, ejecutar el archivo de E / S, o interactuar con un proveedor de contenidos, todo ello desde el segundo plano [19]. Un servicio esencialmente puede tomar dos formas:

- **Iniciado (*started*):** Un servicio es “*iniciado*” cuando un componente de aplicación (tal como una actividad) lo inicia mediante una llamada al método `StartService()`. Una vez iniciado, un servicio puede ejecutarse en segundo plano de forma indefinida, incluso si el componente que se inició, se destruye. Por lo general, un servicio iniciado realiza una sola operación y no devuelve un resultado a la persona que llama. Por ejemplo, puede descargar o cargar un archivo a través de la red. Cuando se realiza la operación, el servicio debe detenerse.
- **Ligado (*bind*):** Un servicio está “*ligado*” cuando un componente de aplicación se une a ella llamando `bindService()`. Un servicio ligado ofrece una interfaz de cliente-

servidor que permite que los componentes interactuar con el servicio, enviar solicitudes, obtener resultados, e incluso lo hacen a través de procesos con la comunicación entre procesos (IPC). Un servicio ligado se ejecuta sólo el tiempo que otro componente de la aplicación está vinculado a la misma. Múltiples componentes pueden unirse al servicio de una sola vez, pero cuando todos ellos se desligan, el servicio es destruido.

5.2.3 Intento

Un intento es un objeto de mensajería se puede usar para solicitar una acción de otro componente de la aplicación [20]. Aunque los intentos facilitan la comunicación entre los componentes de varias maneras, hay tres casos de uso fundamentales:

- Para iniciar una actividad: Puede comenzar una nueva instancia de una actividad al pasar un intento al método `startActivity()`. El intento describe la actividad a iniciar y lleva todos los datos necesarios.
- Para iniciar un servicio: Puede iniciar un servicio para realizar una operación de una sola vez (por ejemplo, descarga un archivo) pasando un intento al método `StartService ()`. El intento describe el servicio a iniciar y lleva todos los datos necesarios.
- Para entregar una difusión: Una difusión es un mensaje que cualquier aplicación puede recibir. El sistema ofrece varias difusiones de los eventos del sistema, como por ejemplo cuando el sistema arranca o el dispositivo comienza a cargar. Se puede enviar una difusión a otras aplicaciones al pasar un intento al método `sendBroadcast ()`, `sendOrderedBroadcast ()`, o `sendStickyBroadcast ()`.

A su vez, hay dos tipos de intentos:

- Los *intentos explícitos* especifican el componente a iniciar por su nombre (el nombre de clase completo). Así, normalmente se utiliza un intento explícito para iniciar un componente de su propia aplicación, ya que se conoce el nombre de la clase de la actividad o servicio que desea iniciar.
- Los *intentos implícitos* no nombran a un componente específico, sino que declaran una acción general para llevar a cabo, lo que permite que un componente de otra aplicación poder manejarla.

5.2.4 Proveedores de Contenido

Los proveedores de contenidos gestionan el acceso a un conjunto estructurado de datos. Ellos encapsulan los datos, y proporcionan mecanismos para la definición de seguridad de los datos. Los proveedores de contenidos son la interfaz estándar que conecta los datos en un solo proceso con el código que se ejecuta en otro proceso. Android incluye

proveedores de contenido que manejan datos tales como audio, vídeo, imágenes, e información de contacto personal.

Cuando se desea acceder a los datos de un proveedor de contenido, se utiliza el objeto `ContentResolver` del contexto de la aplicación para comunicarse con el proveedor como cliente. El objeto `ContentResolver` se comunica con el objeto proveedor, una instancia de una clase que implementa `ContentProvider`. El objeto de proveedor recibe solicitudes de datos de clientes, realiza la acción solicitada, y devuelve los resultados.

6. API: Interfaz de Programación de Aplicaciones

La Interfaz de Programación de Aplicaciones o API (por sus siglas en inglés) es una capa de abstracción que nos provee una forma de conectarnos a componentes de software. En otras palabras, las APIs hacen posible que las organizaciones reúsen sus datos y funcionalidades para crear nuevos servicios. Esto es logrado facilitando la interacción entre los módulos de código, aplicaciones y los sistemas TI del backend. Las APIs especifican la forma en que estos distintos componentes de software pueden interactuar entre sí, y permite el paso de datos entre ellos.

Las APIs no son un concepto nuevo. Pero gracias a que los sistemas informáticos son cada vez más complejos, la necesidad de las APIs ha aumentado, como se puede ver con su prevalencia en los sistemas operativos, lenguajes de programación, redes y la Web. En los últimos años la Web se ha convertido en la red principal para la integración de sistemas y creando una necesidad de buscar formas de conectar los activos de tecnología de información a los portales de Internet y aplicaciones móviles, como resultado las APIs han crecido rápidamente.

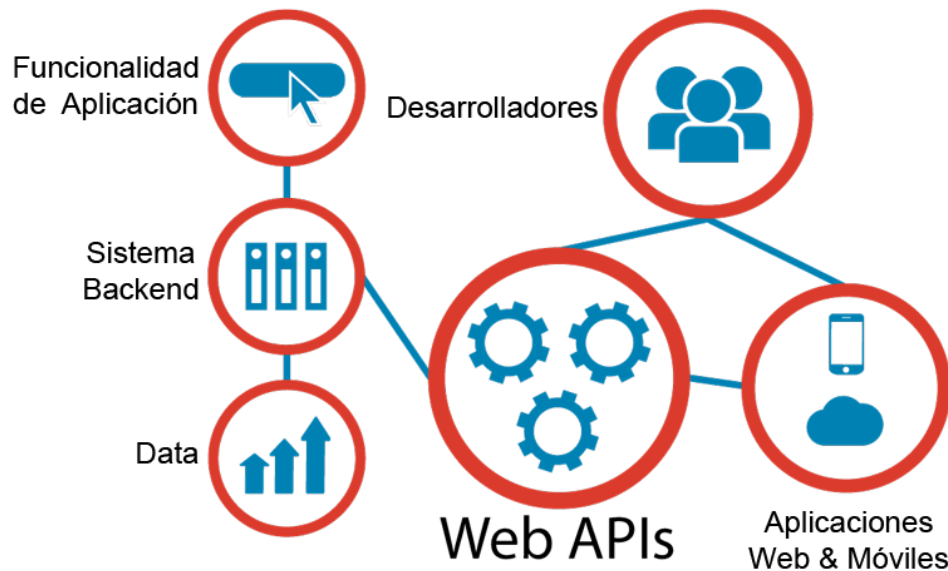


Figura 6.1: Estructura e Interacción de un Web API

Actualmente cuando la gente habla de API, por lo general se hace referencia a los “Web APIs”. Un API de web puede ser definido como cualquier interfaz de software expuesta a través del protocolo HTTP con el fin de facilitar el desarrollo de aplicaciones web, móviles y en la nube. Los Web APIs son particularmente importantes en las redes sociales, ya que, por ejemplo, a través de ellos se podría obtener acceso a grandes volúmenes de datos para analizar el comportamiento de un mercado en especial o permitir a los desarrolladores crear una aplicación cliente para publicar actualizaciones de estado de los dispositivos móviles. Sin embargo, los Web APIs son cada vez más

importante para las organizaciones de todos los sectores. En la Figura 6.1 se aprecia la estructura e interacción entre los niveles de abstracción generales de un web API.

6.1 API Privado

Un API privado es una interfaz que permite el acceso a los datos y habilita ciertas funcionalidades de una empresa para ser usada por los desarrolladores que trabajan dentro (o contratistas que trabajan para) de dicha organización. Las nuevas aplicaciones que estos desarrolladores crean pueden ser distribuidas al público, pero la propia interfaz no está disponible para cualquier persona que no trabaja directamente para la empresa dueña del API.

Los APIs privados pueden reducir significativamente el tiempo de desarrollo y los recursos necesarios para integrar los sistemas informáticos internos, la construcción de nuevos sistemas y crear aplicaciones de cara al cliente. En lugar de crear aplicaciones desde cero, los desarrolladores pueden sacar provecho de los activos de software internos.



Figura 6.2: Estructura e Interacción de un Web API Privado

Es fundamental mantener continuamente cualquier API, para garantizar que la seguridad y el rendimiento de los sistemas backend se mantengan a largo plazo. El mantenimiento de API privado puede parecer fácil ya que las interfaces sólo están expuestas a los desarrolladores internos, lo que reduce los riesgos de seguridad; Y los diseñadores del API tienen acceso directo a estos desarrolladores, por lo que es más fácil crear interfaces amigables. Sin embargo, es importante recordar que la exposición de la interfaz del software siempre crea una serie de problemas de seguridad y de mantenimiento. Por ejemplo, en muchos casos, las aplicaciones cliente se comunican con los APIs a través del Internet público o redes móviles. También existen desafíos asociados con la

integración de los sistemas que utilizan diferentes protocolos y estándares, en particular en los sistemas antiguos (legado) son a menudo inadecuados para su uso en dispositivos móviles o la Web. La Figura 6.2 ejemplifica como los distintos sistemas interactúan con el API privado.

6.2 API Público

Un API Público es una interfaz que ha sido diseñado para ser accesible a una población más amplia de desarrolladores web y móviles. Esto significa que un API Público puede ser utilizado tanto por los desarrolladores dentro de la empresa dueña del API o por cualquier desarrollador externo que desee registrarse para tener acceso a la interfaz.

Las empresas que desarrollan su API Público por lo general buscan aprovechar la creciente comunidad de desarrolladores de aplicaciones. Esto le permite a la organización estimular el desarrollo de aplicaciones innovadoras que aporten valor a su negocio principal, sin necesidad de invertir directamente en los esfuerzos de desarrollo, lo que al mismo tiempo aumenta la producción de nuevas ideas y disminuye los costos de desarrollo. Por lo tanto, para los gerentes de empresas y diseñadores de la interfaz, el objetivo fundamental debe ser aumentar tanto la cantidad como la calidad del uso del API. Esto significa llegar a un público específico de desarrolladores, entregar la interfaz junto a la documentación que permita satisfacer a la audiencia.

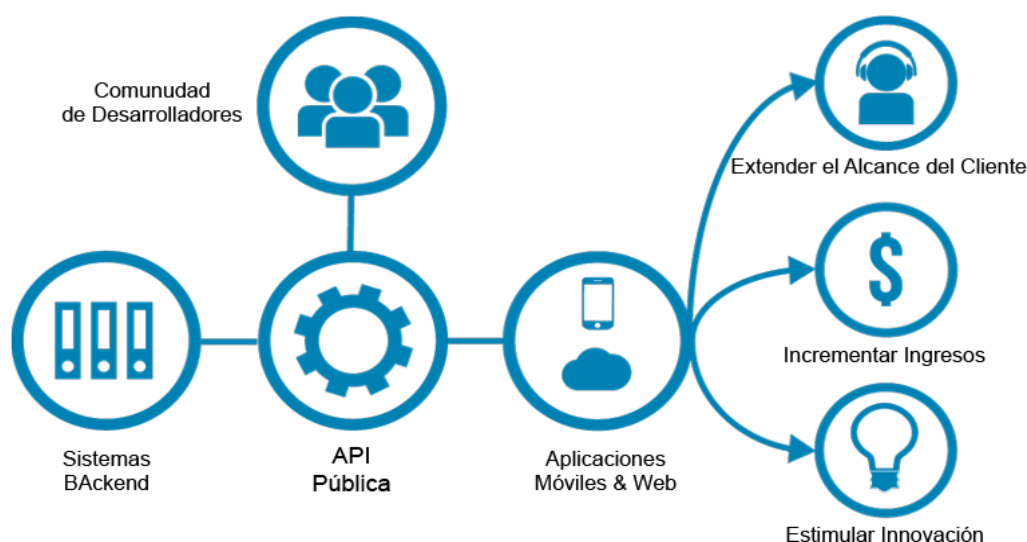


Figura 6.3: Estructura e Interacción de un Web API Público

Una interfaz pública para desarrolladores externos puede aumentar considerablemente los problemas de mantenimiento y de seguridad asociados al API. Por ejemplo, con muchas de las aplicaciones de terceros activas en el mercado, puede ser muy difícil garantizar que las actualizaciones de la interfaz no dañaran la funcionalidad de la aplicación. El aumento de los riesgos de seguridad representa otro reto importante

asociado con la exposición pública de las interfaces de software. No sólo la publicación de un API teóricamente significa que cualquier desarrollador puede acceder a los sistemas de backend a la vista, sino que también podría llamar la atención de los hackers. La interacción de los distintos actores con el API público puede ser apreciada en la Figura 6.3.

7. Pasarela de Pago

Una Pasarela de Pago es un servicio intermediario entre un cliente de comercio electrónico y un banco cuando se ejecutan transacciones bancarias online. Se integra directamente al cliente y almacena información del banco que maneja las cuentas de compradores y vendedores. Para el proceso de pago o transferencias bancarias es necesaria la utilización de tarjetas de crédito o débito como medios electrónicos de pago. La Pasarela de Pagos valida la veracidad de la tarjeta y organiza la transferencia del dinero de la cuenta del comprador a la cuenta del vendedor.

Las transacciones bancarias online deben realizarse bajo un estricto control de seguridad para proteger el sistema de ataques a la autenticidad, confidencialidad, integridad, disponibilidad y el no repudio de la información. La comunicación establecida entre los entes participantes debe utilizar una conexión a través de un servidor seguro haciendo uso de protocolos seguros permitiendo que la información viaje cifrada entre el cliente y el servidor. Esta estrategia evita que los datos sean interceptados por terceras personas, comprometiendo datos sensibles. Al usar protocolos como SSL para autenticar el servidor y HTTPS para utilizar certificados de clave pública para verificar la identidad de los participantes, se fomenta la seguridad entre las entidades participantes cuando se realizan transacciones bancarias online.

Son varios los procesos de pago utilizado por las Pasarelas de Pago, como lo son el pago usando tarjetas de crédito o débito y el pago mediante transferencia. En el primer caso, el cliente accede a un sitio de comercio electrónico y elige la lista de artículos a comprar. La aplicación calcula el importe a cobrar y cuando el cliente está listo para pagar, es dirigido a la pasarela, la cual le muestra el monto a pagar y los datos a introducir como el número de tarjeta. La pasarela se encarga de codificar la información la cual viaja de forma segura hacia el banco. En este se comprueba rápidamente que la tarjeta sea válida (que no sea robada o que esté vencida) y que el cliente tenga los fondos suficientes para comprar los artículos. De estar todo en orden se ingresa el dinero en la cuenta del vendedor, la misma debe pertenecer al banco en cuestión o este debe tener relación con el banco que posee la cuenta del vendedor. La pasarela le comunica al comercio y al cliente el resultado de la transacción (si el pago se efectuó o no).

El proceso de transferencia sería similar: el cliente se encargará de informar a que cuenta desea ingresar la transferencia, la pasarela manipulará la información para que viaje confidencialmente hacia la cuenta destino, si no se presentan inconvenientes o si se presentaran, el cliente siempre será notificado. Finalmente, todas las transacciones realizadas se actualizan en la base de datos de la pasarela.

7.1 Principales Pasarelas de Pago

Actualmente existe gran cantidad de Pasarelas de Pagos en todo el mundo. Cada una de ellas se diferencia entre sí por las restricciones que poseen, facilidades de desarrollo, el costo que cobran por transacciones, y los países con los cuales trabajan.

La consultora Boston Retail Partners realizó una encuesta [21] donde se puede observar (Figura 7.1) las pasarelas de pago más usadas por los minoristas en EEUU, así como los planes de implementarlas dentro de los próximos 5 años. Las pasarelas de pago con mejor proyección fueron PayPal, Apple Pay y Google Wallet.

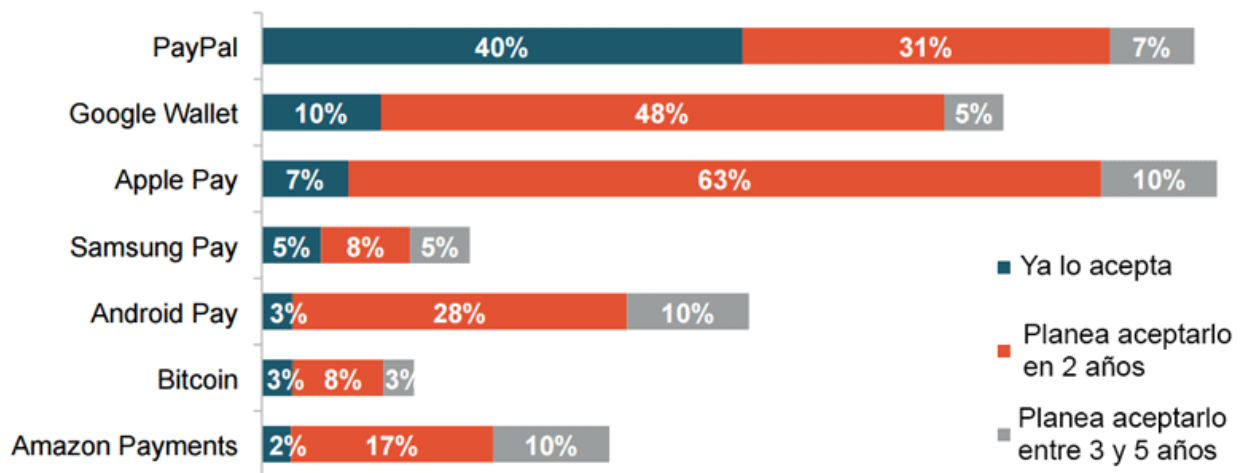


Figura 7.1: Pasarelas de Pago por Minoristas en EEUU

PayPal es una pasarela de pagos que permite realizar pagos y transferencias de dinero online de forma segura. Es la más conocida a nivel mundial pertenece a los EEUU y está extendida a más de 202 países, y 56 monedas. Su servicio permite la transferencia de dinero entre usuarios que posean cuenta con PayPal y para ellos solo es necesario que tengan correo electrónico. Con el uso de esta pasarela se pueden realizar peticiones de pago en comercio electrónico desde aplicaciones Web y móviles de terceros. Los métodos de transferencia y pago por medio del correo electrónico tienen asociado una tarjeta de crédito (tales como Visa, Mastercard, American Express y Diners de compradores de EEUU y Europa) [22].

Google Wallet y Android Pay son los sistemas de pago creados por Google, donde Google Wallet evolucionó en una aplicación que permite enviar y recibir dinero de manera rápida y gratuita con tu cuenta de Gmail. Mientras que Android Pay permite a sus usuarios almacenar tarjetas de débito, crédito, fidelidad y tarjetas regalo en su teléfono móvil Android. Este sistema utiliza Near Field Communication (NFC) para hacer pagos rápidos y seguros con un simple toque del teléfono en cualquier terminal habilitado al

momento de pagar y solo está disponible por ahora en EEUU, Reino Unido, Singapur y Australia.

Apple Pay es el servicio de pago móvil creado por Apple que fue lanzado en octubre de 2014. El servicio permite a los usuarios añadir tarjetas de débito y crédito al iPhone para pagar su compra con un solo toque usando tecnología Near Field Communication (NFC) en conjunto con Touch ID y Passbook. También puede ser usado para realizar compras dentro de aplicaciones y en sitios web. Está disponible solo en EEUU para los equipos de última generación.

8. Trabajos Relacionados

A continuación, se describirán un par de trabajos relacionados con nuestra investigación, ambos tienen un objetivo general idéntico al nuestro, sin embargo, el primero a mencionar fue realizado en el año 2009 y el segundo es una aplicación que se encuentra actualmente en el mercado, con esto se quiere hacer referencia a las guías de diseño que se están usando hoy en día para poder hacer uso de estas en nuestra aplicación y como han cambiado las aplicaciones móviles a lo largo de los años.

8.1 Aplicación de Recarga de Saldo de Technology & Solutions LTDA

El primero es una aplicación de recarga de saldo desarrollada para una empresa, la cual se nombrará más adelante, la cual sirvió como tesis para la Universidad Industrial de Santander bajo el título “*Desarrollo de la aplicación para la gestión de recargas en línea a teléfonos celulares para la empresa Technology & Solutions LTDA*”. Esta tenía como objetivo general realizar una implementación de una aplicación móvil que permitiera recargas en línea desde teléfonos celulares a distintos operadores, haciendo uso de J2ME (Java 2 Mobile Environment) [23]. Se busca con esta aplicación facilitar las recargas de saldo para los usuarios, sin importar el tipo de operador, dado que hasta en aquel entonces se realizaban recargas mediante el uso de tarjetas prepagadas. Se muestra en la Figura 8.1, un ejemplo de la pantalla de confirmación y la recarga aprobada de esta aplicación.

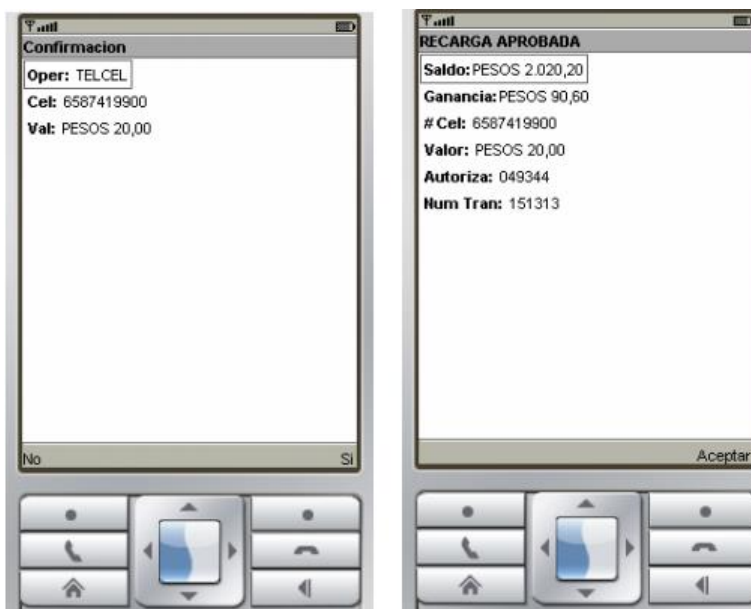


Figura 8.1: Pantalla de Confirmación y Aprobación de la Recarga

El estudio de este trabajo relacionado antes presentado, es de gran importancia para el desarrollo de este trabajo especial de grado. Se observa el avance de las tecnologías y la facilidad que provee el framework de android sobre J2ME para el desarrollo de aplicaciones móviles, así como las ventajas al momento del desarrollo de las interfaces de usuario.

8.2 Aplicación de Recarga de Saldo Ezetop Ding

La siguiente es una aplicación que se encuentra disponible en el Play Store, llamada Ezetop Ding, esta permite recarga de saldo a más de 100 países provee registros sociales con Facebook y Google, además permite la facilidad de hacer uso de una tarjeta de crédito o incluso una cuenta de PayPal. A continuación, se muestran las pantallas con una breve explicación.

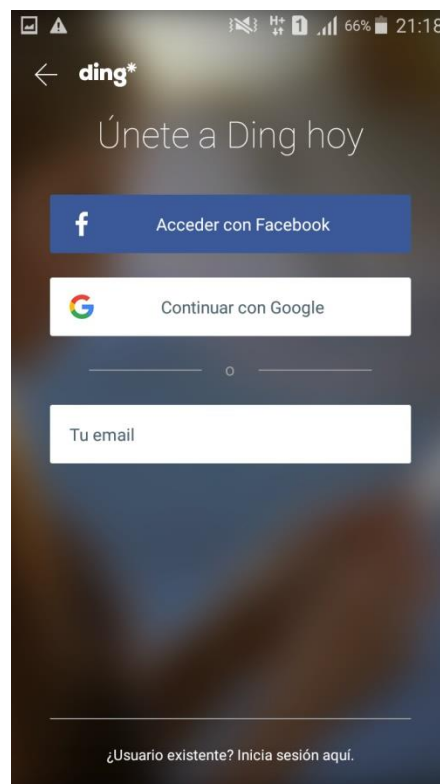


Figura 8.2: Pantalla Inicial de Ezetop Ding

En la Figura 8.2 se puede apreciar la pantalla de registro, donde se puede utilizar la cuenta de Facebook o Google para registrarnos o incluso el correo personal.

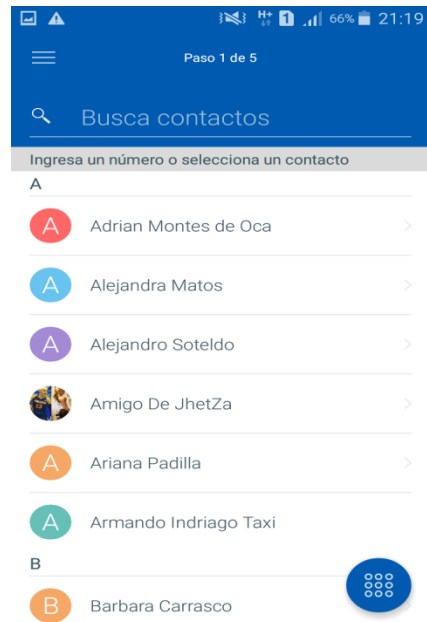


Figura 8.3: Pantalla Inicial para Recarga de Saldo

Una vez iniciada la sesión, la primera pantalla será la lista de los contactos que se encuentran almacenados en el teléfono como se muestra en la Figura 8.3, se pueden seleccionar para realizar recargas o incluso se puede escribir un número que no se encuentre registrado en el celular como se refleja en la Figura 8.4.

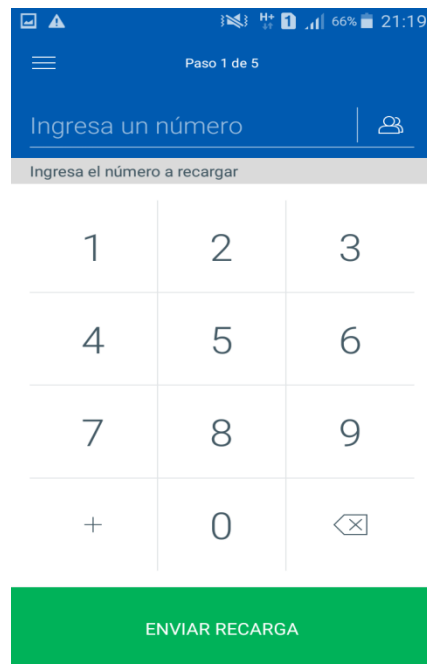


Figura 8.4: Teclado para Ingresar Números Telefónicos

Una vez seleccionado el contacto o ingresado el número telefónico, esta información se mostrará en la parte superior de la pantalla para poder observar los datos que se van suministrando. El siguiente paso es indicar el país al que pertenece el número telefónico como se muestra en la Figura 8.5, esto con la finalidad de poder obtener todos los operadores con los cuales se tiene acceso a sus servicios

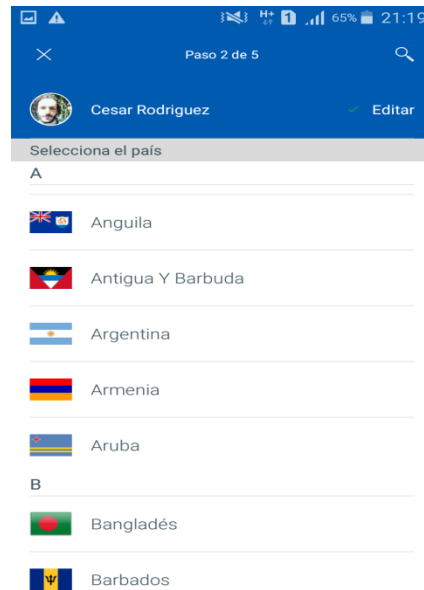


Figura 8.5: Selección del País

Como se mencionó anteriormente el siguiente paso es seleccionar el operador como se muestra en la Figura 8.6.

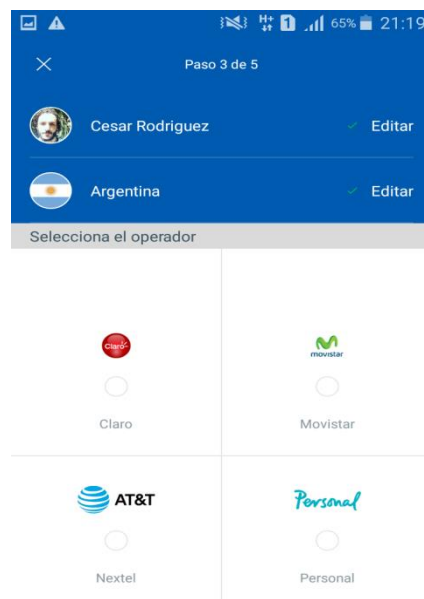


Figura 8.6: Selección del Operador

El último paso es la selección del monto a recargar como se refleja en la Figura 8.7, esto también depende del operador anteriormente seleccionado ya que no todos ofrecen los mismos productos.

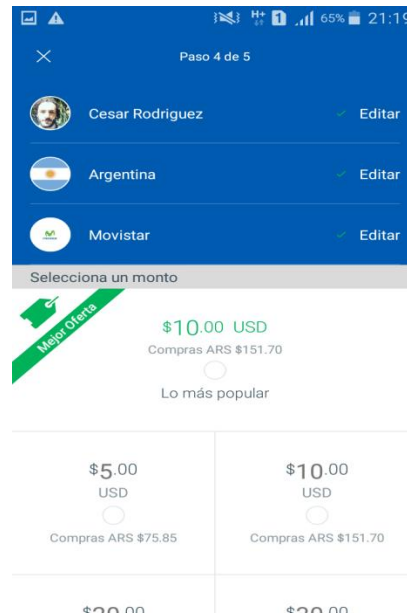


Figura 8.7: Selección de Producto o Monto

Luego de la selección de todos los datos necesarios, se habilitará un botón el cual nos permitirá ir a la sección de pago como se muestra en la Figura 8.8, donde se podrá elegir una tarjeta o una cuenta de PayPal y así finalizar el proceso de recarga de saldo.

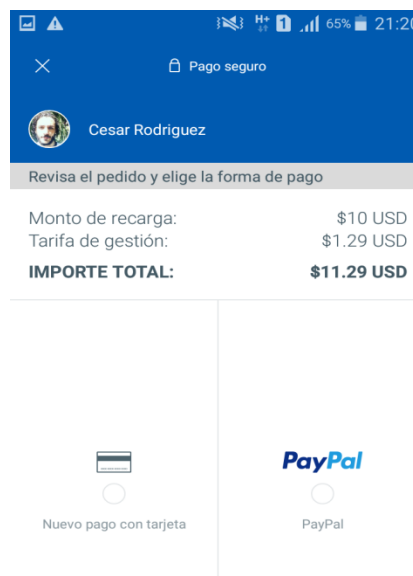


Figura 8.8: Sección de Pago

Al estudiar esta aplicación comercial se aprecian las guías de estilo, elementos visuales e incluso la distribución de las pantallas que utiliza una aplicación similar que se encuentra actualmente disponible en el Play Store de Google. También, es importante notar que esta y muchas otras aplicaciones, hacen uso de una autenticación social con Facebook y Google además de proveer la posibilidad de un pago móvil incorporando el SDK de PayPal.

9. Marco Metodológico

Para alcanzar los objetivos planteados en el Capítulo 2, es necesario definir una metodología de trabajo que permita el desarrollo estructurado y organizado de cada uno de los requerimientos de la aplicación. A continuación, se presenta la especificación de Mobile-D, la metodología utilizada, que está pensada para el desarrollo de aplicaciones móviles y puede ser empleada también en el desarrollo de la consola administrativa, además del atractivo que posee por el desarrollo rápido, enfocado en la implementación.

9.1 Método Mobile-D

Mobile-D es una metodología ágil diseñada principalmente para el desarrollo de aplicaciones móviles; busca conseguir ciclos de desarrollo muy rápidos basando sus prácticas en las pruebas, programación en pequeños equipos e integración continua [24].

Como se ilustra en la Figura 9.1, la metodología consta de cinco fases que son: exploración, iniciación, producción, estabilización y prueba del sistema. Cada una de ellas tiene un número de etapas, tareas y prácticas asociadas. A continuación, se describen las fases:

- **Exploración:** El equipo de desarrollo se centra en la planificación y conceptos básicos del proyecto. Este se realiza en tres etapas, que son el establecimiento de actores, como por ejemplo los clientes; definición del alcance y establecimiento de proyectos. Se diferencia del resto de fases ya que no posee un día de planificación y otro de entrega.
- **Inicialización:** Preparación e identificación de los recursos necesarios. Se preparan los planes para las siguientes fases y se establece el entorno técnico como los recursos físicos, tecnológicos, de comunicación y capacitación del equipo. Este se divide a su vez en cuatro etapas, la puesta en marcha del proyecto, la planificación inicial, el día de prueba y día de salida.
- **Producción:** Se repite iterativamente la programación de tres días, como son planificación, trabajo y liberación hasta implementar todas las funcionalidades. Esta iteración se planifica en términos de requisitos y tareas a realizar. Las pruebas de iteración son preparadas previamente, las tareas se llevan a cabo durante el día de trabajo y durante el último día se realiza la integración del sistema en caso de haber grupos trabajando de forma independiente, para finalmente realizar las pruebas de aceptación.
- **Estabilización:** Se llevan a cabo las acciones de integración para asegurar el correcto funcionamiento de todo el sistema. Los desarrolladores realizan tareas similares a las de producción, buscando la integración del sistema. En esta fase es importante la integración multi-equipo con diferentes subsistemas en

desarrollo. Adicionalmente, en esta fase se contempla la producción de la documentación.

- **Pruebas:** Se realizan pruebas y se corrigen errores sin desarrollar algo nuevo, hasta llegar a una versión estable según lo establecido en las primeras fases por el cliente.

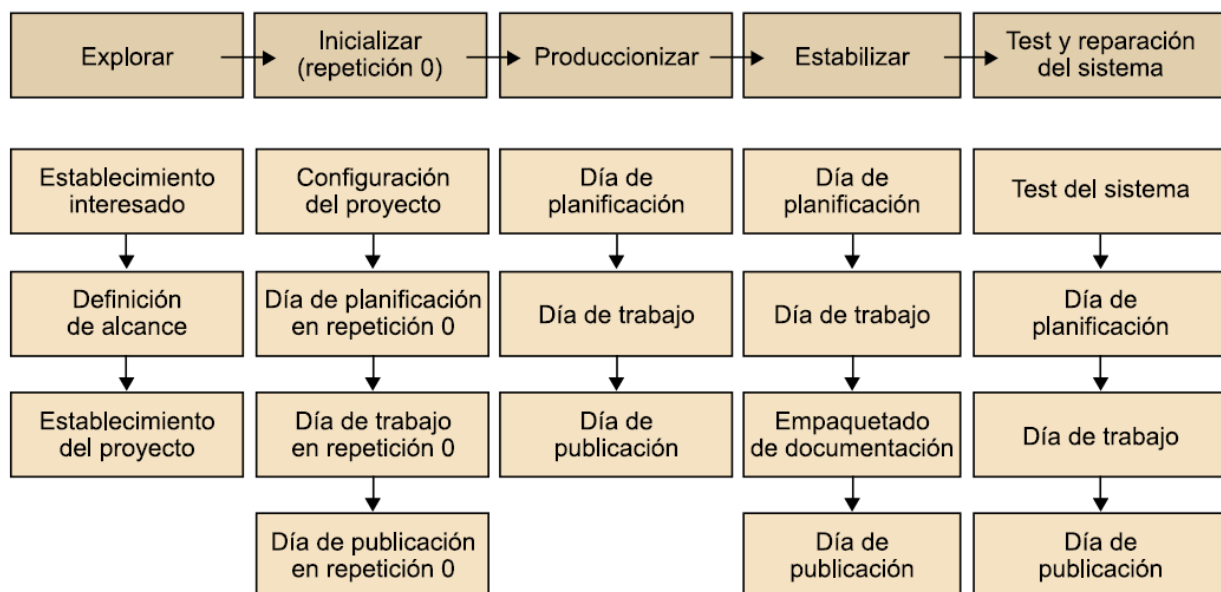


Figura 9.1: Ciclo de Desarrollo de Mobile-D

9.2 Fase de Exploración

Durante esta primera fase de la metodología Mobile-D se planificó realizar dos aplicaciones, un cliente móvil para sistemas operativos Android que se integre con el API privado de TopUp City LLC, para realizar las recargas de saldo. Y una aplicación web que deberá integrarse con la base de datos de TopUp City LLC para administrar los recursos del sistema. A demás se definió la arquitectura de la solución que se presenta a continuación.

9.2.1 Arquitectura de la Solución

La Figura 9.2 muestra un diagrama con la arquitectura del sistema, en donde la aplicación móvil (1) usará los servicios de Facebook (2) y Google (3) para realizar la autenticación social, también se integrará con la pasarela de pago de PayPal (4) para realizar el cobro de los productos. La aplicación deberá sincronizada en todo momento con el API privado de TopUp City LLC (5), que al momento de realizar una recarga se conecta con los proveedores del servicio de recarga como Emida. La consola de administración (6) se conectara a la base de datos de TopUp City para administrar sus recursos.

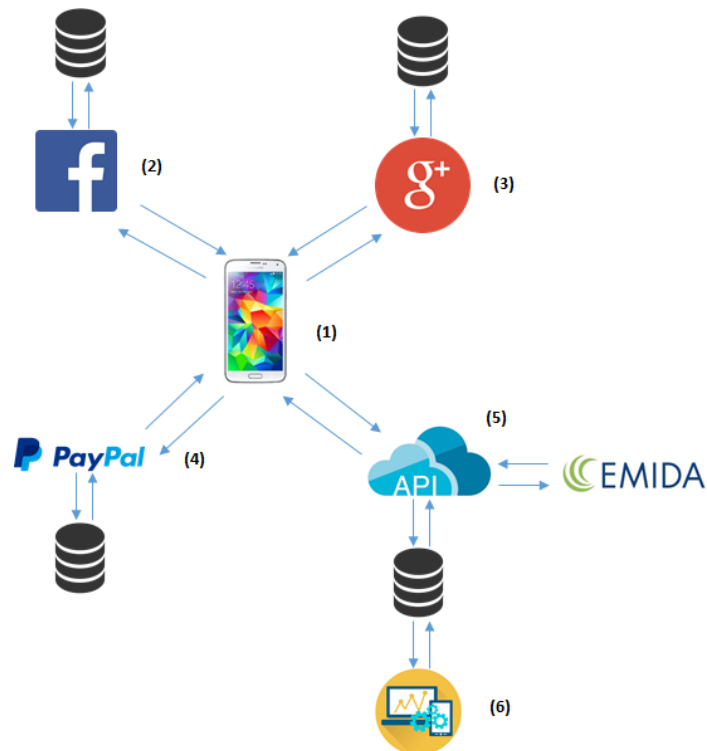


Figura 9.2: Arquitectura de la Solución

Para explicar mejor el flujo de la aplicación, hay que aclarar que para poder realizar una recarga de saldo el usuario deberá autenticarse en la aplicación y para esto se ofrece la posibilidad de iniciar sesión haciendo uso de la autenticación social de Facebook o Google además de un servicio propio de registro o inicio de sesión. En caso de que el usuario decida hacer uso de este, el cliente android hará uso del SDK apropiado para poder conectarse a los servicios de autenticación de la red social y estos accederán a sus modelos propios para poder devolver los datos requeridos, en este caso solo es necesario el nombre y el correo del usuario. Al obtener dichos datos, se enviarán al API para la creación de un nuevo registro.

Una vez autenticado el usuario, este podrá seguir el flujo de venta de un producto el cual termina haciendo uso del SDK de PayPal para permitir el cobro. Este proveerá la opción de hacer uso de una cuenta en PayPal o realizar la transacción con una tarjeta de crédito. Al obtener la respuesta y en caso de que esta sea exitosa se procede a conectarse con el API para realizar una nueva transacción y poder acceder a los servicios pre-pagados, nuevamente en caso de éxito se almacena un registro en el modelo de TopUp City LLC y se responde al usuario que se ha logrado realizar la transacción.

9.3 Fase de Inicialización

En la esta fase se definieron las tecnologías a utilizar y las características de las interfaces de usuario para ambas aplicaciones.

9.3.1 Tecnologías a Utilizar

En la presente sección, se listan y describen brevemente cada una de las tecnologías y herramientas a usar para implementar la solución:

- Java, que es un lenguaje de programación de propósito general, concurrente y orientado a objetos. Será utilizado tanto para el desarrollo de la aplicación Android nativa, como para la consola Web.
- El Kit de Desarrollo de Software (SDK) de Android, el cual posee el conjunto de herramientas necesarias para crear y desplegar una aplicación Android nativa.
- La Plataforma Integral del Desarrollo (IDE) seleccionada es Android Studio que en conjunto con el complemento ADT (Android Developer Tools plugin) es la única plataforma soportada oficialmente por el SDK de Android.
- Spring Boot, para crear la consola Web basada en una aplicación de Spring Framework.
- Para el manejo de la base de datos MySQL, se utilizará el framework ORM (Object-Relational Mapping) Hibernate, el cual a su vez posee su propia implementación del API de Persistencia de Java (JPA).
- Amazon Web Services (AWS), para el despliegue de la consola Web.
- Git, como manejador de versiones en ambas aplicaciones.
- La plataforma de Jira, para de seguimiento del proyecto y sus incidencias.

9.3.2 Prototipo de Interfaz del Cliente Móvil

Ésta aplicación debe usar los componentes de interfaz nativos ofrecidos por el framework de Android y seguir las guías de estilo “*Material Design*” para mantener un estándar con las demás aplicaciones de esta plataforma. También se debe hacer uso de las librerías “*Support*” para proveer compatibilidad hacia atrás a las versiones antiguas de Android.



Figura 9.3: Pantalla de Bienvenida

Primero se necesita una pantalla de bienvenida (Figura 9.3) la cual le mostrará al usuario el logo de TopUp City LLC mientras la aplicación se inicializa por completo.

El “*Main Activity*” de la aplicación será el encargado de llevar el proceso de recarga de saldo. En la Figura 9.4 se aprecia el menú lateral con que debe contar la aplicación para la navegación de los usuarios no registrados (invitados). Para este tipo de usuario, el menú le permitirá navegar por los países disponibles para hacer recargas y posteriormente verificar las operadoras y productos existentes para el país seleccionado; además de permitir al usuario ir al “*Activity*” correspondiente para realizar el inicio de sesión.

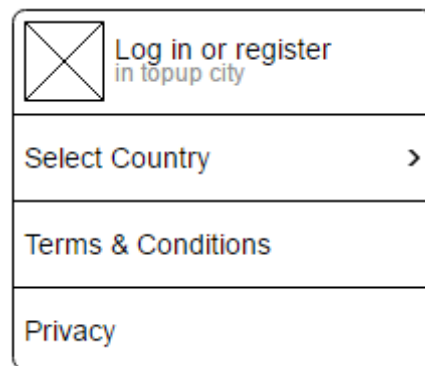


Figura 9.4: Menú Lateral para Invitados

Para los usuarios que se hayan registrado e iniciado sesión, el menú lateral será como el de la Figura 9.5, que además de permitir la selección de un país de los disponibles, dejará al usuario navegar hasta la lista de recargas realizadas en la sección “*My Purchases*” y también podrá cerrar la sesión en el dispositivo.

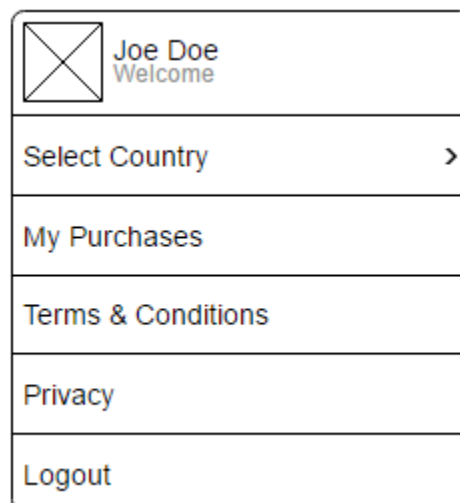


Figura 9.5: Menú Lateral para Usuarios Registrados

El proceso de recarga de saldo consta de varios “*fragments*”, el primero de ellos es el encargado de capturar el número a donde se desea realizar la recargar y darle a elegir al usuario el producto según el operador seleccionado, como se observa en la Figura 9.6.

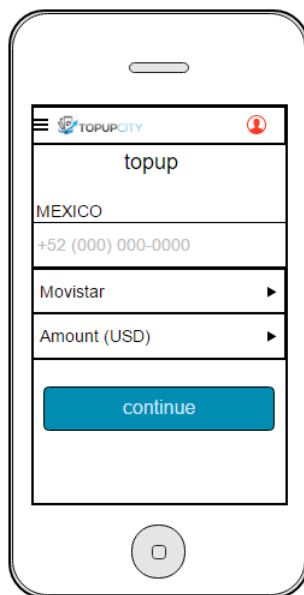
A mobile application interface for a topup service. At the top, there is a header bar with a menu icon, the 'TOPUPCITY' logo, and a red notification icon. Below the header, the word 'topup' is displayed. The form contains several input fields: 'MEXICO', a phone number '+52 (000) 000-0000', a dropdown menu for 'Movistar', and another dropdown menu for 'Amount (USD)'. At the bottom of the form is a blue button labeled 'continue'.

Figura 9.6: “*Fragment*” con Formulario de Recarga

Una vez validado el número, se muestra al usuario el detalle del producto que va a adquirir y se le muestran las opciones de pago disponibles como se aprecia en la Figura 9.7.

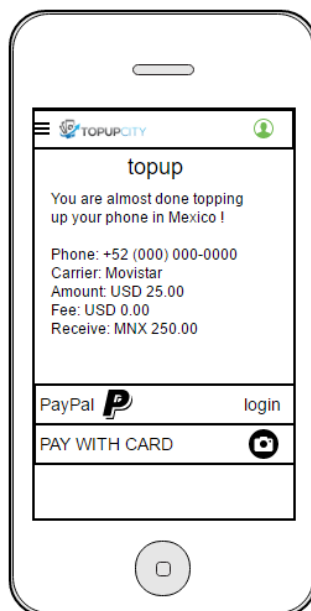
A mobile application interface showing the details of a topup transaction. The header bar is similar to the previous screen but includes a green checkmark icon. The word 'topup' is displayed, followed by a confirmation message: 'You are almost done topping up your phone in Mexico !'. Below this, the transaction details are listed: 'Phone: +52 (000) 000-0000', 'Carrier: Movistar', 'Amount: USD 25.00', 'Fee: USD 0.00', and 'Receive: MNX 250.00'. At the bottom, there are two payment options: 'PayPal' with a 'login' link, and 'PAY WITH CARD' with a card icon.

Figura 9.7; “*Fragment*” con Detalle del Producto

El “*activity*” encargado del registro y del inicio de sesión de los usuarios contará con una pestaña para acción, en cada una se encontrarán dos botones para iniciar sesión con Google o Facebook y un formulario para utilizar el usuario de TopUp City si así lo desea el usuario. En la Figura 9.8 se pueden observar con más claridad.

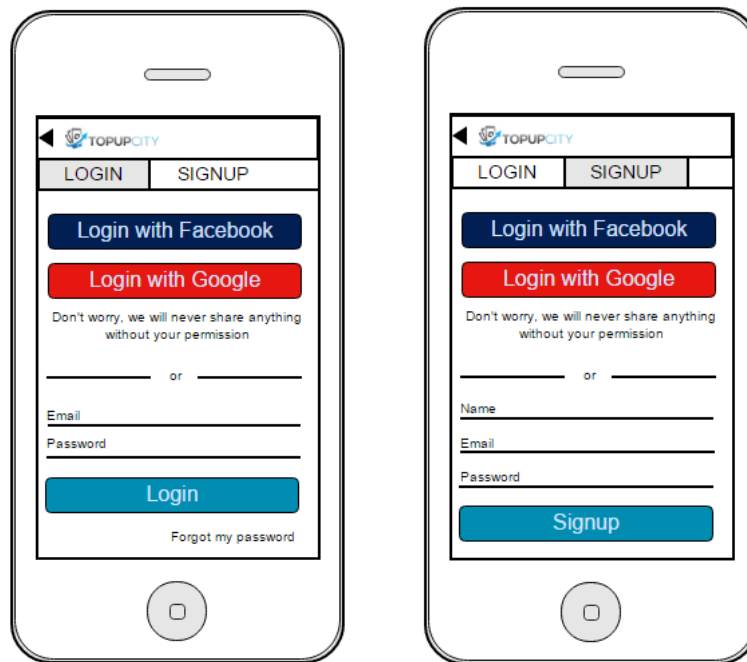


Figura 9.8: Prototipo de Interfaz de Inicio de Sesión y Registro

El último prototipo de la aplicación móvil es el del “*activity*” encargado de, como se aprecia en la Figura 9.9, listar todos los productos comprados a través de la aplicación.

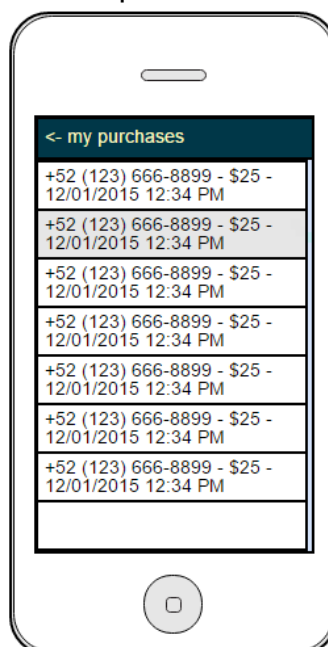


Figura 9.9: Prototipo de Interfaz para Listado de Recargas

9.3.3 Prototipo de Interfaz de la Consola Web

A continuación, se muestran los prototipos de interfaz de usuario para la Consola Administrativa. Todas las vistas utilizan una misma plantilla, la cual cuenta con un menú lateral izquierdo para administrar los recursos de la aplicación móvil y un botón en la esquina superior derecha para el control de usuarios de la consola administrativa. Estos y todos los demás componentes restantes como tablas, formularios y botones son “*responsive*” o adaptativos, lo que quiere decir que se ajustarán según la resolución de la pantalla en la que se encuentren.

En la Figura 9.10 se muestra la pantalla de inicio de sesión, la cual posee un botón y dos campos de texto para introducir el nombre de usuario y la contraseña. Esta pantalla será constante para cualquier resolución.

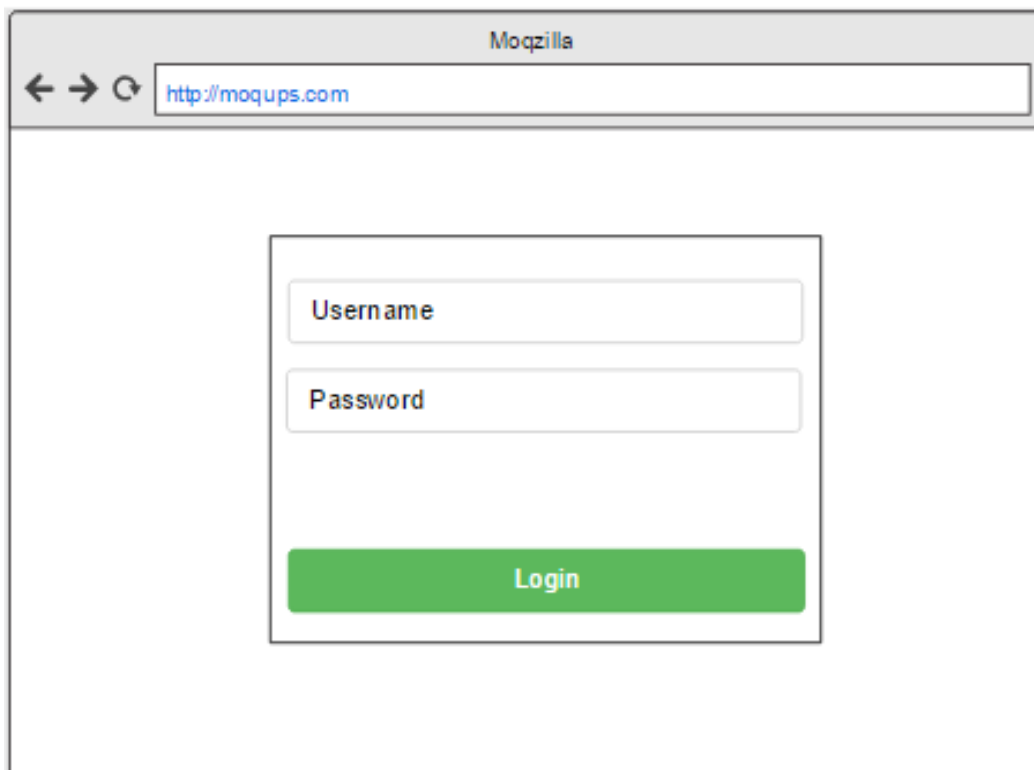
El prototipo muestra una ventana de navegador con el título "Mozilla". La barra de direcciones contiene "http://moqups.com". El contenido principal es un formulario centrado con dos campos de texto: "Username" y "Password", ambos con un ícono de ojo para alternar visibilidad. Debajo de estos campos hay un botón rectangular de color verde con el texto "Login" en blanco.

Figura 9.10: Pantalla de Inicio de Sesión

Una vez iniciada la sesión, se ingresará al tablero de la consola administrativa como se muestra en la Figura 9.11. La interfaz cuenta con unos indicadores que contienen los contadores de los elementos más importantes del sistema como cantidad de usuarios, países, monedas, productos, operadoras y transacciones registradas en el sistema.

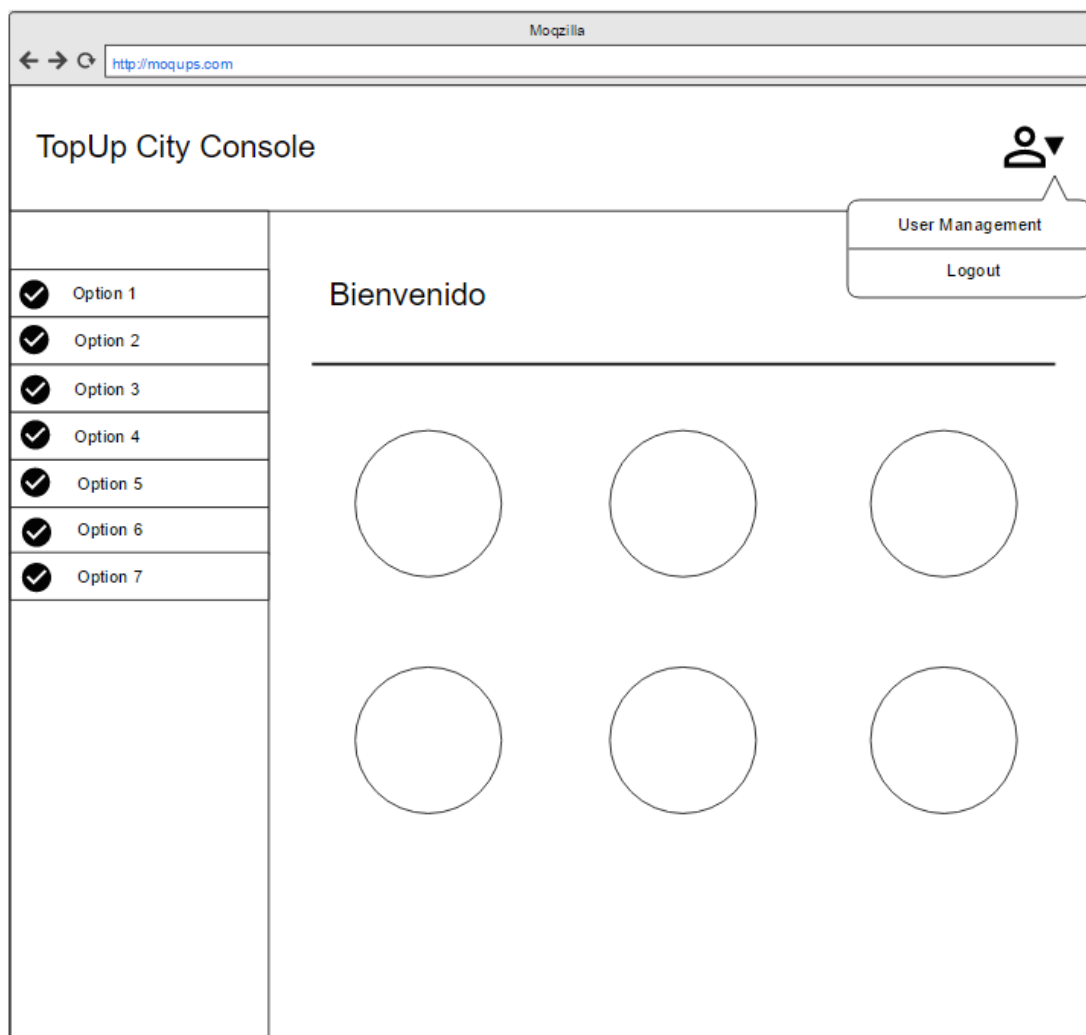


Figura 9.11: Página de Inicio o Tablero de la Consola Administrativa

Para poder apreciar los recursos administrables, estos serán dispuestos en tablas como lo muestra la Figura 9.12, donde las columnas serán los atributos de los recursos y las filas los registros almacenados en el sistema.

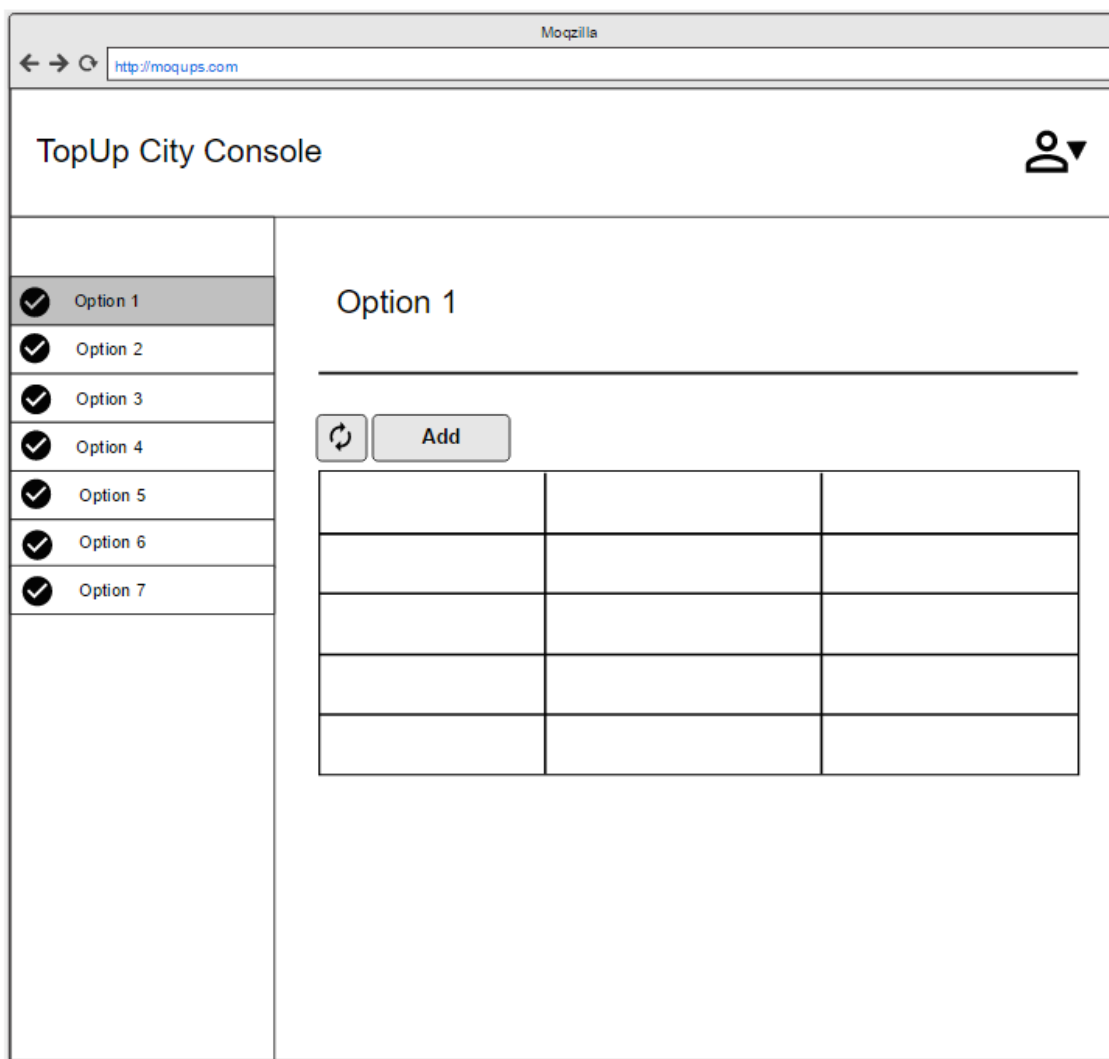


Figura 9.12: Página Principal de los Recursos Administrables

Así mismo, para poder agregar o eliminar registros es necesario hacer uso de formularios que contengan los campos necesarios para la inserción o edición de estos, un ejemplo de estos formularios se muestra en la Figura 9.13.

Todos los formularios contarán con un botón de Regresar para así facilitar la navegación en la aplicación web. Es importante notar que en la vista principal de los recursos administrables se encuentra un botón con la palabra “Add” o Añadir dependiendo del idioma del navegador, el cual direccionará al formulario para agregar un nuevo registro. El caso de eliminar y editar son opciones que se encontrarán en la última columna de las tablas con los iconos de una papelera de reciclaje y un lápiz respectivamente.

Moqzilla

← → ↻ <http://moqups.com>

TopUp City Console



✓ Option 1

✓ Option 2

✓ Option 3

✓ Option 4

✓ Option 5

✓ Option 6

✓ Option 7

Option 1

Username

Email

Password

Regresar

Action

Figura 9.13: Ejemplo de Formularios para Agregar o Editar Registros

9.4 Fases de Producción y Estabilización

Las fases de producción y estabilización se tomaron como una sola fase, ya que solo dos personas trabajarían en el proyecto y cada uno desarrollara una aplicación distinta. En esta fase se realizaron varias iteraciones para desarrollar tanto la aplicación móvil como la consola administrativa, dichas iteraciones son descritas en el Capítulo 10 del Marco Aplicativo.

9.5 Fase de Pruebas

La fase de prueba se llevó a cabo al finalizar cada una de las iteraciones de desarrollo descritas en el próximo capítulo, en ella se realizaron pruebas funcionales para verificar el correcto funcionamiento del módulo desarrollado según el alcance del mismo. Además se hicieron pruebas de integración del módulo desarrollado en cada iteración con los desarrollados en las iteraciones previas.

Al finalizar el desarrollo de la aplicación móvil y de la consola web se realizaron pruebas para verificar que los recursos se estuvieran administrando correctamente.

10. Marco Aplicativo

En este capítulo se explica el diseño de la solución a través de las iteraciones definidas en base a los lineamientos de la metodología Mobile-D, descrita en el Capítulo 9. Cada iteración se define en base al desarrollo de las fases de Diseño, Codificación y Pruebas.

10.1 Análisis General de la Aplicación Móvil

La aplicación Android nativa se desarrolló utilizando el lenguaje de programación Java en conjunto con XML (eXtensible Markup Language) para la implementación de los “*layouts*” o interfaces gráficas. De igual forma se utilizaron diversas librerías y SDKs para realizar determinadas funcionalidades, que serán mencionadas más adelante.

El Entorno de Desarrollo Integrado (IDE por sus siglas en inglés) usado para el desarrollo de la aplicación móvil fue *Android Studio*, basado en IntelliJ IDEA y en el *Android Developer Tool* (ADT) y posee una excelente integración con Java y su máquina virtual, el sistema de compilación automático “*Gradle*”, emuladores de dispositivos Android y demás herramientas que están a disposición del desarrollador.

Para ejecutar la aplicación se utilizó un dispositivo Motorola Moto G de primera generación, corriendo el sistema operativo Android en su versión 5.1.1, también conocida como “*Lollipop*”.

A continuación, se describirán cada una de las iteraciones del proceso de desarrollo de la aplicación.

10.1.1 Prueba de Concepto de Inicio de Sesión Social

Para la primera iteración era necesario desarrollar un módulo que ayudara al desarrollador a familiarizarse con el ambiente y el “*framework*” de Android, es por ello, que se tomó la decisión de empezar por realizar una prueba de concepto sobre el inicio de sesión, usando redes sociales como Facebook y Google, ya que ambas plataformas poseen APIs bien documentadas y además tienen una gran comunidad de desarrolladores dispuestos a ayudar.

En la Figura 10.1 se observa el diagrama de flujo básico de esta prueba de concepto, en donde, al correr la aplicación se debe verificar si el usuario ya ha iniciado sesión. En caso de ser positivo, se deben mostrar los datos previamente guardados y permitir el cierre de sesión, y en caso contrario, se debe permitir al usuario escoger entre Facebook y Google para iniciar sesión, y posteriormente recuperar, guardar y mostrar los datos básicos del usuario.

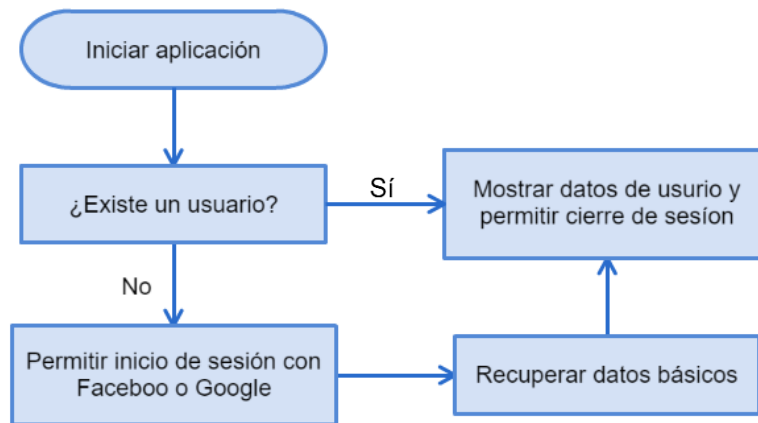


Figura 10.1: Diagrama de Flujo del Inicio de Sesión Social

Para lograr esta prueba de concepto se crearon dos “*Activities*” como se aprecia en la Figura 10.2, el primero es el encargado de verificar la existencia de un usuario y del inicio de sesión con las redes sociales, y el segundo se encarga de mostrar los datos del usuario y del cierre de sesión. Para la persistencia de los datos del usuario se usó el “*Share Preference*”, un mecanismo de persistencia local el cual utiliza pares clave-valor.

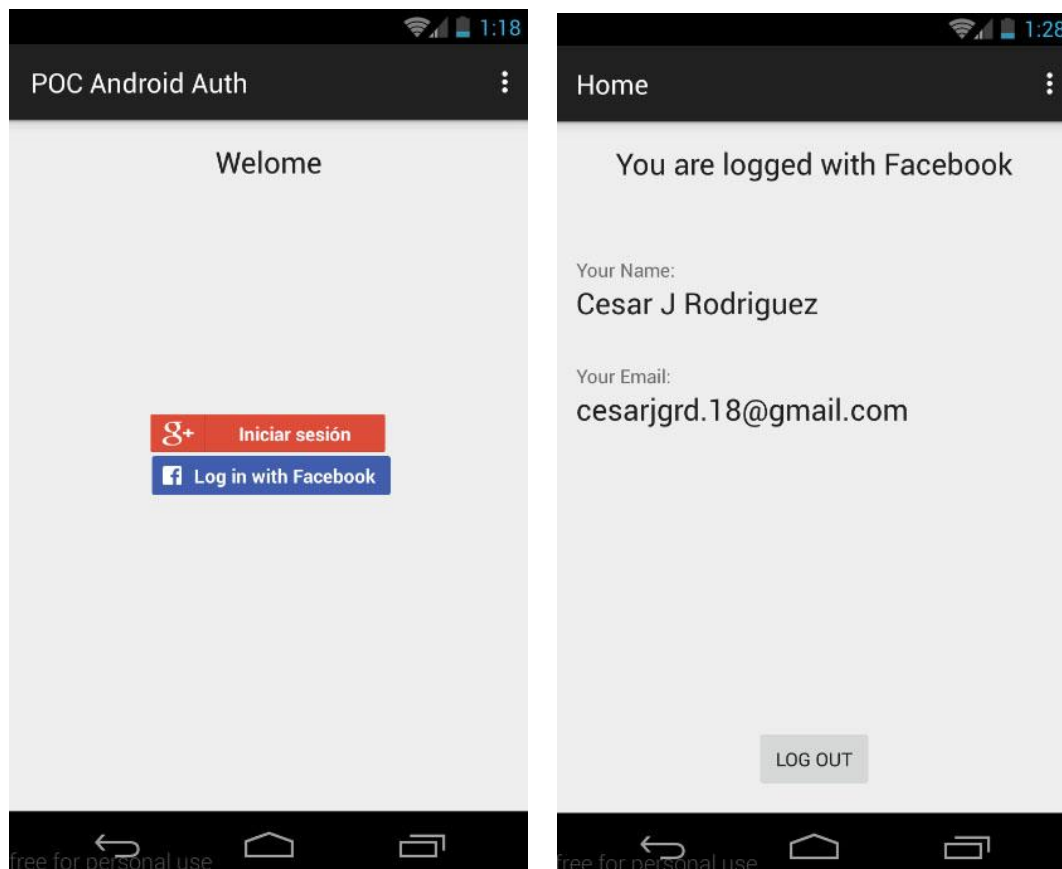


Figura 10.2: “*Activities*” de la Prueba de Concepto de Inicio de Sesión

En cuanto al inicio de sesión con las redes sociales, se consultaron tanto las guías de desarrollo de Facebook como las de Google. Ambas redes requieren un proceso de configuración e implementación que se puede resumir en los siguientes pasos:

- Creación de proyecto: dentro de cada una de las plataformas (Facebook y Google) se debe crear un proyecto, al cual se le debe especificar la plataforma desde donde se va a hacer uso del SDK de la red social, además de los permisos que se le van solicitar al usuario, ya sea para ver su perfil, correo y demás datos del mismo.
- Configuración de la aplicación: se deben añadir las dependencias de cada plataforma a la aplicación a través de “*Gradle.build*”, añadir los respectivos IDs que Facebook genera y el archivo de configuración que genera Google.
- Implementación: los SDKs de cada una de las plataformas, deben ser implementados y así hacer uso de las funciones necesarias que se necesiten.

10.1.2 Prueba de Concepto Integración con PayPal

La segunda iteración del proceso de desarrollo, tuvo como objetivo crear una prueba de concepto sobre el cobro, haciendo uso del SDK de PayPal, implementando la integración con dicha pasarela de pago. El alcance de esta prueba solo requería complementar el inicio de sesión social, con un campo de texto en donde se ingresara un monto, como se muestra en la Figura 10.3, para luego invocar el SDK de PayPal en modo “*SandBox*” (ambiente virtual de pruebas que replica el ambiente de producción) para realizar el cobro y capturar la respuesta del SDK.

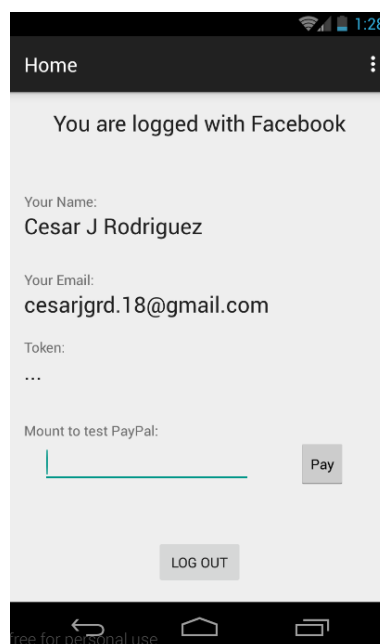


Figura 10.3: “*Activity*” con la Integración de PayPal

Entre las formas de pago que ofrece el SDK de PayPal, la prueba de concepto se enfocó en el “*single payment*”, en el que se ejecuta el pago una sola vez y de forma inmediata. Lo primero que se hizo fue añadir el SDK a la aplicación, como se muestra en la Figura 10.4.

```
27
28 dependencies {
29     compile fileTree(dir: 'libs', include: ['*.jar'])
30
31     //PAYPAL SDK
32     compile 'com.paypal.sdk:paypal-android-sdk:2.15.3'
33 }
```

Figura 10.4: Importando SDK de PayPal

El siguiente paso para integrar PayPal, fue crear el objeto “*PayPalConfiguration*”, encargado de toda la configuración del SDK, por ejemplo: el ambiente en donde se va a ejecutar, ya sea en modo “sin internet” en donde se simulan todos los datos, en modo de pruebas o “*sandbox*”, o en producción; y también se identifica a quien le pertenece la aplicación con el número de cliente otorgado por PayPal. En la Figura 10.5 se demuestra como se configura este SDK.

```
103
104 // Start with mock environment. When ready, switch to sandbox (ENVIRONMENT_SANDBOX)
105 // or live (ENVIRONMENT_PRODUCTION)
106 private static PayPalConfiguration config = new PayPalConfiguration()
107     .environment(PayPalConfiguration.ENVIRONMENT_NO_NETWORK)
108     .clientId(Constant.PAYPAL_CLIENT_ID);
109
```

Figura 10.5: Configuración del SDK de PayPal

Antes de invocar al SDK para realizar un cobro, se debe iniciar el servicio de PayPal al empezar el ciclo de vida del “*activity*”, y al destruirse este, se debe detener dicho servicio. Para realizar el cobro se debe crear un “*intent*” para la clase “*PaymentActivity*” perteneciente al SDK de PayPal y agregar toda la información del pago, posteriormente iniciar el “*activity*” para obtener un resultado como se observa en la Figura 10.6.

Por último, solo resta obtener el resultado de la operación, haciendo uso del método de retorno de Android para los “*activities*” iniciados para resultado llamado “*onActivityResult*”, en donde se recibe un parámetro de tipo “*intent*” que posea todos los datos referentes a la transacción, como errores en caso de ser cancelada por el usuario, falta de fondos, o algún otro error; y en caso de éxito, se obtienen algunos datos, entre los que se puede destacar el *timestamp* de la transacción, el monto, la moneda, la descripción y la comisión. Estos se utilizan para poder crear una transacción en el sistema haciendo uso del API privado de TopUp City.

```

761 public void onBuyPressed(View pressed) {
762
763     // PAYMENT_INTENT_SALE will cause the payment to complete immediately.
764     // Change PAYMENT_INTENT_SALE to
765     // - PAYMENT_INTENT_AUTHORIZE to only authorize payment and capture funds later.
766     // - PAYMENT_INTENT_ORDER to create a payment for authorization and capture
767     // later via calls from your server.
768
769     PayPalPayment payment = new PayPalPayment(new BigDecimal("1.75"), "USD", "sample item",
770         PayPalPayment.PAYMENT_INTENT_SALE);
771
772     Intent intent = new Intent(this, PaymentActivity.class);
773
774     // send the same configuration for restart resiliency
775     intent.putExtra(PayPalService.EXTRA_PAYPAL_CONFIGURATION, config);
776
777     intent.putExtra(PaymentActivity.EXTRA_PAYMENT, payment);
778
779     startActivityForResult(intent, 0);
780 }

```

Figura 10.6: Llamada al SDK de PayPal para Realizar Pago

10.1.3 Flujo Principal de la Aplicación

En la tercera iteración de desarrollo, se planificó hacer las interfaces del flujo principal de la aplicación mostrado en la Figura 10.7, además de la navegación entre los principales “activities” y uno de los componentes principales como lo es el menú lateral “tipo cajón”.

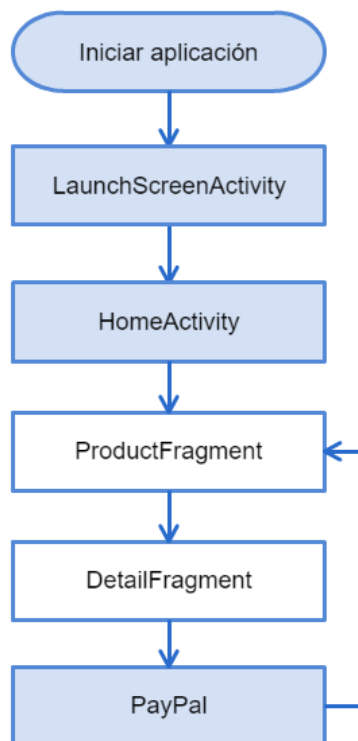


Figura 10.7: Flujo Principal de la Aplicación

Al iniciar la aplicación, el primer *activity* en ejecutarse es el *LaunchScreenActivity*, que muestra el logo de TopUp City LLC mientras se inicia, así como se puede ver en la Figura 10.8. Este *activity* será el encargado de hacerle la solicitud al API privado de un token de autenticación, el cual se almacena con el *SharePreferences* como se mencionó anteriormente, con la finalidad de poder interactuar con este. Al terminar la carga inicial de la aplicación, el *LaunchScreenActivity* inicia el *HomeActivity* y finaliza su ejecución haciendo uso del método *finish()* de la clase *Activity*.



Figura 10.8: Interfaz Gráfica del *LaunchScreenActivity*

Toda aplicación en Android tiene un *activity* principal, en esta aplicación se llamó *HomeActivity* y se encuentra contenido el flujo de pago, que está constituido por varios *fragments*, cada uno con una función en específico, además del menú lateral “tipo cajón” que le permitirá al usuario navegar entre las distintas secciones de la aplicación y cambiar el país a donde se realizará la recarga. El *HomeActivity* también es el encargado de realizar la conexión con PayPal a través de su SDK para realizar el pago de las recargas.

El menú lateral “tipo cajón”, tiene dos versiones que se pueden observar en la Figura 10.9, la primera cuando el usuario es un invitado, es decir, que aún no se ha registrado o no ha iniciado sesión en la aplicación y la segunda para los que sí se hayan autenticado. Ambas versiones del menú tienen elementos en común como “Seleccionar País” (para cambiar el país a cual se realizará la recarga), “Términos y Condiciones”, “Privacidad” y “Ayuda”; para los invitados el *header* del menú lo insta a iniciar sesión, mientras que para los usuarios que se hayan autenticado el *header* muestra sus datos y

para estos mismos el menú tendrá dos opciones extras que son la de “Mis TopUps” para ver el historial de compras y la opción de “Cerrar Sesión”.

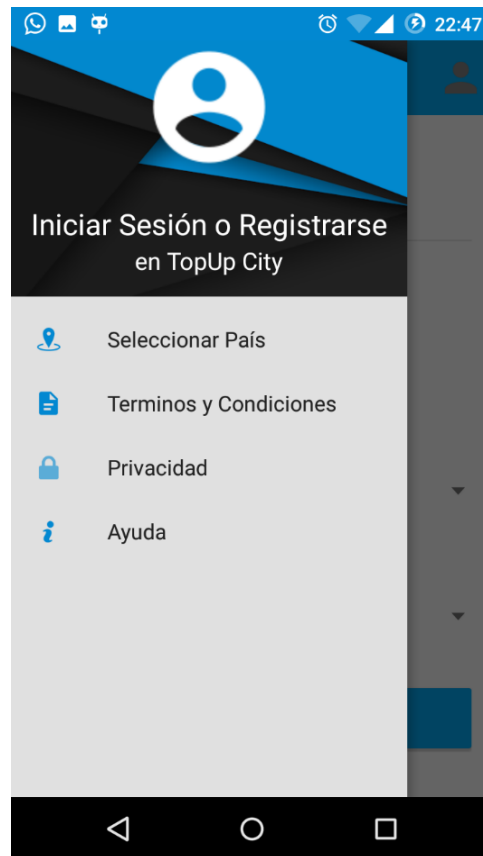


Figura 10.9: Menú Lateral “Tipo Cajón”

Por sí solo, el *HomeActivity* no cumple ninguna función, necesita de los *framents* que conducirán al usuario en el proceso de realizar la recarga. Al iniciar el ciclo de vida del *HomeActivity*, este busca en la información del usuario el país que tiene guardado y se inicia el *SelectProductFragment* en un contenedor dentro de su propio *layout*, mostrando las operadoras y productos de dicho país. En caso que el usuario no tenga un país guardado o sea un invitado, se toma el país por defecto.

En la Figura 10.10, se puede apreciar que además de poder seleccionar un operador y un producto en el *SelectProductFragment*, también se debe introducir el número que se desea recargar para continuar con el proceso. Este *fragment* verifica la validez del número antes de ejecutar el *DetailProductFragment*, que es el encargado del siguiente paso en el proceso de recarga.

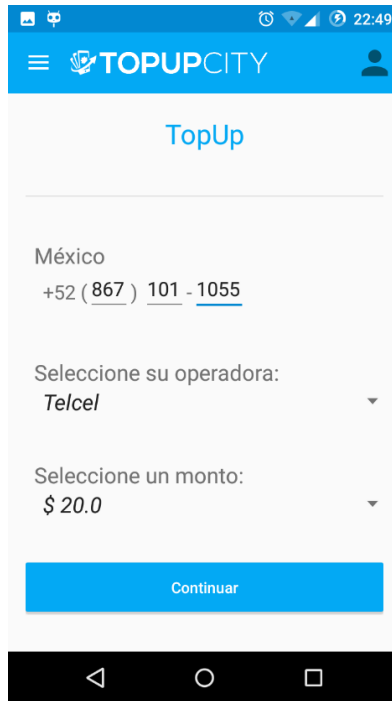


Figura 10.10: Interfaz Gráfica del *SelectProductFragment*

Cuando el *DetailProductFragment* es iniciado, este consultará el API para recuperar y mostrar toda la información referente al producto que se ha seleccionado, tal cual como se observa en la Figura 10.11, si el usuario está conforme, puede continuar para realizar el pago con PayPal.



Figura 10.11: Interfaz Gráfica del *DetailProductFragment*

Si el usuario decide continuar con la compra, se invoca el SDK de PayPal con los datos del producto, tal como se aprecia en la Figura 10.12. Finalizado el proceso de pago, se captura la respuesta de PayPal y de ser exitosa se realiza una llamada al API privado de TopUp City LLC con los datos del producto y del pago para ejecutar la recarga.

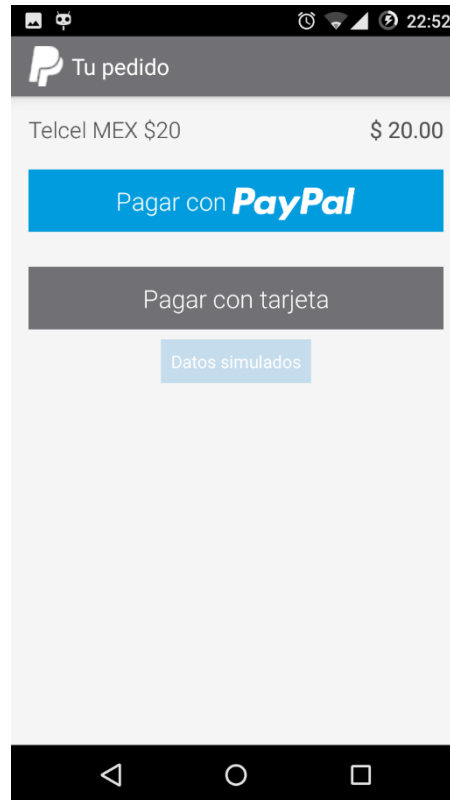


Figura 10.12: “PaymentActivity” del SDK de PayPal

10.1.4 Integración con API Privado de TopUp City LLC

Hasta el final de la tercera iteración la aplicación móvil solo contaba con la navegación básica del proceso de pago y datos por defecto, en esta cuarta iteración de desarrollo se planificó integrar la aplicación con el API privado de TopUp City LLC.

El REST API cuenta con trece servicios que van desde la creación de la sesión, usuarios y transacciones; hasta la autenticación. Cada servicio se mencionará brevemente a continuación:

- **POST auth/req_session:** permite que una aplicación cliente se autentique con el servidor y obtenga un ID de sesión (sid) que pueda utilizarse en llamadas consecutivas a la API REST.
- **POST auth/login:** servicio de autenticación estándar haciendo uso del correo y contraseña del usuario.
- **POST auth/logout:** invalida el ID de sesión (sid) actual en el sistema.

- **POST account/settings/create:** crea un usuario con la configuración de la cuenta.
- **POST account/settings/update:** actualiza la configuración de la cuenta del usuario.
- **GET account/settings:** devuelve todos los ajustes asociados a la cuenta de usuario.
- **GET countries:** devuelve todos los países de destino actualmente disponibles para recargar en el sistema.
- **GET countries/:id/carriers:** devuelve las operadoras actualmente disponibles para un país específico.
- **GET products/:id:** devuelve el detalle completo de un producto específico disponible en el sistema.
- **POST transaction:** envía una solicitud de transacción (recarga) al sistema, esta llamada se realiza una vez que el flujo de pago con PayPal se ha completado correctamente. El sistema intentará procesar la recarga y confirmar el pago con PayPal si la recarga también tiene éxito, de lo contrario se revertirá el pago y se enviará un mensaje de error a la aplicación cliente.
- **GET transactions:** devuelve las últimas X transacciones procesadas por un usuario basadas en una fecha y hora como punto de origen.

Para consumir estos servicios se utilizó la librería “*Retrofit 2*” que es un cliente HTTP para Java y Android desarrollado por “*Square, Inc*”. Retrofit hace que sea relativamente fácil el envío y recepción de JSON (u otro tipo de dato estructurado) a través de un servicio web basado en REST, como se observa en la Figura 10.13.

```

19 public interface TopUpApiServices {
20     @GET("account/settings")
21     Call<MyResponse> getAccountSettings(@Query("sid") String sid);
22
23     @GET("countries")
24     Call<MyResponse> getCountries(@Query("sid") String sid);
25
26     @GET("countries/{countryId}/carriers")
27     Call<MyResponse> getCountriesCarriers(@Path("countryId") int countryId,
28                                         @Query("sid") String sid);
29
30     @GET("products/{id}")
31     Call<MyResponse> getProducts(@Path("id") int productId, @Query("sid") String sid);
32

```

Figura 10.13: Java Interface de Retrofit

Retrofit implementa la interfaz y por cada “*Call*” creado se puede realizar una petición HTTP asíncrona al servidor web remoto. Para realizar las peticiones solo se debe encolar la llamada e implementar el “*callback*” o método de retorno, en donde se recibirá la respuesta como se aprecia en la Figura 10.14. Otra ventaja de Retrofit es que puede convertir el cuerpo de la respuesta de la petición en un objeto del tipo que se defina en la intefaz haciendo uso de la librería de serialización “*Gson*”.

```

42 private void requestSession() {
43     TopUpApiServices apiServices = TopUpApiClient.getService();
44     final Call<MyResponse> reqSession = apiServices
45         .postAuthReqSession(new Session(getApplicationContext()));
46     reqSession.enqueue(new Callback<MyResponse>() {
47         @Override
48         public void onResponse(Call<MyResponse> call, Response<MyResponse> response) {
49
50         }
51
52         @Override
53         public void onFailure(Call<MyResponse> call, Throwable t) {
54
55         }
56     });
57 }

```

Figura 10.14: Ejemplo de Petición HTTP con Retrofit

10.1.5 Módulo de Autenticación

El desarrollo de la quinta iteración se basó en un “*activity*” encargado del proceso de autenticación del usuario en la aplicación. Como se observa en la Figura 10.15 la interfaz cuenta con dos pestañas, la primera para iniciar sesión y la segunda para registro.

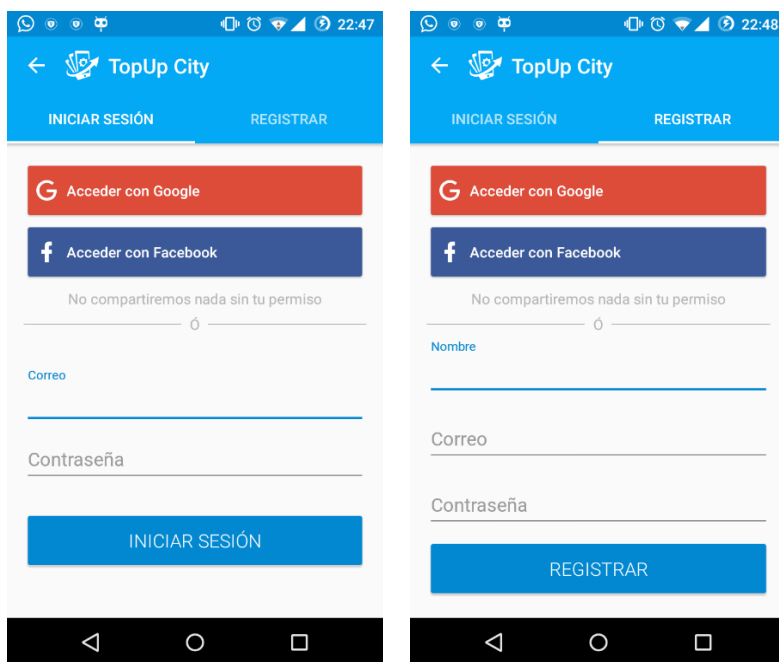


Figura 10.15: Interfaz Gráfica de las Pestañas del “*SignInActivity*”

El “*SignInActivity*” se encarga de manejar las pestañas, las llamadas al API REST de TopUp City LLC y de la implementación de los SDKs de Facebook y Google, reusando parte del código de la Prueba de Concepto de Inicio de Sesión Social (capítulo 10.1.1).

En la segunda pestaña, la de registro, el usuario puede llenar el formulario con su nombre, correo y contraseña para registrarse; la aplicación valida los datos y hace una llamada al servicio “*POST account/settings/create*” del API REST. Si la llamada retorna exitosamente, el usuario inicia sesión automáticamente y se retorna al “*HomeActivity*”. Para la primera pestaña el usuario puede llenar el formulario para iniciar sesión con su correo y contraseña previamente registrados en la aplicación. En este caso la llamada al API REST es “*POST auth/login*”, siguiendo el mismo comportamiento del registro en caso de éxito.

En cuanto al inicio de sesión social, ambas pestañas poseen los botones y ejecutan el mismo algoritmo (Figura 10.16) para autenticarse con Facebook y Google. Una vez autenticado con la red social de preferencia del usuario, se solicitan a esta los datos básicos (nombre y correo) del usuario. Con esta información se procede a realizar el registro a través de la llamada “*POST account/settings/create*” al API REST. Si la llamada devuelve un error que no sea de usuario ya existente se le notificará al usuario el error, en caso de que el servicio retorne exitosamente o con el error de que el usuario ya existe, se procede al inicio de sesión con la llamada “*POST auth/login*”, si la llamada es exitosa la aplicación retorna al “*HomeActivity*”, de lo contrario se le notifica el error al usuario.

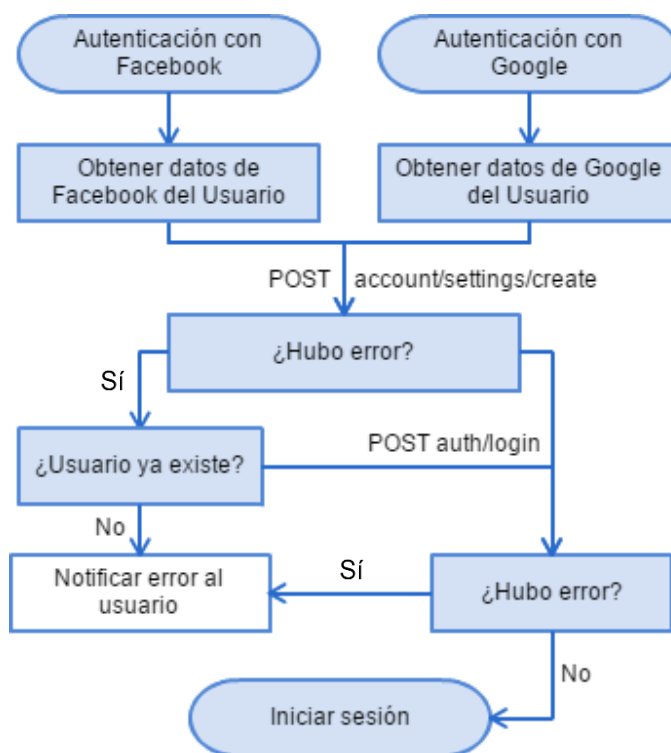


Figura 10.16: Flujo de Autenticación con Redes Sociales

10.1.6 Desarrollo del Historial de Transacciones

En la tercera iteración (Sección 10.1.3) se hizo mención del menú lateral de la aplicación, dicho menú cuenta con un elemento solo visible para usuarios autenticados llamado “Mis TopUps”, que inicia el componente “*MyTopUpsActivity*” en el que está basado la presente iteración.

Dicho “*activity*” es el encargado de listar todas las recargas realizadas por el usuario, haciendo uso del API REST con la llamada “*GET transactions*”. Para lograr el resultado que se observa en la Figura 10.17, se hizo uso de la librería de soporte para compatibilidad regresiva de Android conocida como “*Support v7*”, específicamente se usó un “*widget*” de interfaz de usuario llamada “*RecyclerView*”, que permite mostrar en pantalla grandes colecciones de datos de manera eficiente, ya que recicla las vistas de los elementos anteriores que ya no son visibles en la pantalla, mejorando así el tiempo de creación y destrucción de la vista de cada elemento; el segundo “*widget*” de interfaz de usuario que se utilizó fue el “*CardView*”, siendo este un contenedor para crear tarjetas con apariencia similar, soporta esquinas redondeadas y sombras de elevación de manera nativa.

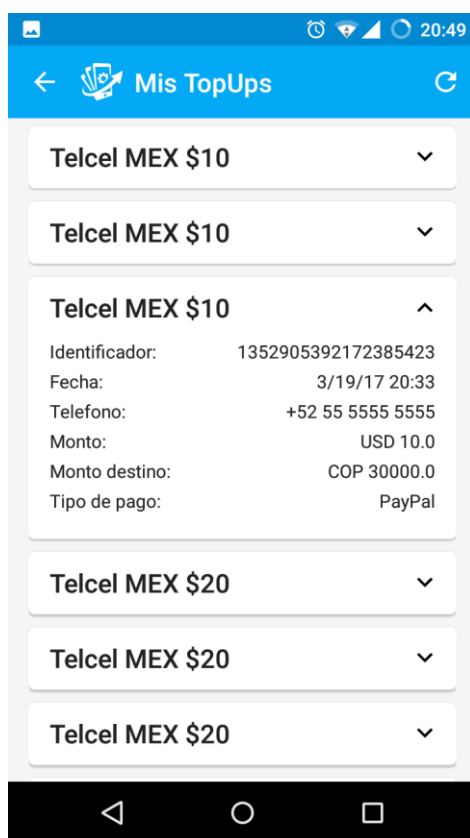


Figura 10.17: Histórico de Recargas

Cada recarga realizada se lista en una tarjeta que tiene como título el nombre de dicho producto, por defecto la tarjeta se encuentra contraída y con un botón de expansión. La información mostrada al expandir la tarjeta es inherente a la recarga, entre los datos de esta que se muestran, se tienen: el identificador de la transacción que es un entero que identifica inequívocamente a ésta, la fecha y hora en que se realizó la transacción, el número destino de la recarga, el monto cobrado por la recarga, el monto recibido en moneda del país destino de la recarga y el medio o forma de pago.

10.1.7 Mejora de las UI con un Diseño Adaptativo

Hasta la iteración anterior se contaba con el MVP totalmente funcional, pero la interfaz de usuario deja mucho que desear, es por ello que para esta séptima y última iteración de desarrollo, se planificó mejorar la interfaz de usuario del flujo de pago, así como el ajuste de todas las interfaces de la aplicación para adaptarse correctamente a diversos tamaños y densidades de pantalla.

Al momento de realizar un “*layout*”, nombre que recibe el recurso encargado de definir la estructura visual de una interfaz de usuario, se deben tomar en cuenta dos aspectos muy importantes de la pantalla del dispositivo donde se va a mostrar dicha interfaz. El primero es la densidad de pantalla que hace referencia a la cantidad de píxeles dentro de un área física de la pantalla, y el segundo es el tamaño de la pantalla.

Android utiliza los “píxeles independientes de densidad” (*dp*) para facilitar la construcción de los “*layouts*”. Un *dp* es una unidad de píxeles virtuales que se debe usar al definir el diseño de interfaz de usuario, para expresar las dimensiones o la posición del diseño con independencia de la densidad. El píxel independiente de la densidad es equivalente a un píxel físico en una pantalla de 160 dpi (puntos por pulgada), valor que representa la densidad de referencia que considera Android para una pantalla de densidad “media”. En tiempo de ejecución, el sistema maneja de forma transparente cualquier ajuste de las unidades *dp*, cuando resulta necesario, según la densidad actual de la pantalla en uso. La conversión de unidades *dp* a píxeles de pantalla es simple: $px = dp * (dpi / 160)$. Por ejemplo, en una pantalla de 240 dpi, 1 *dp* es igual a 1,5 píxeles físicos. Siempre se debe usar unidades *dp* cuando se define la interfaz de usuario de una aplicación, para asegurar que el diseño se muestre de manera apropiada en pantallas con diferentes densidades.

Considerando esto, en esta séptima iteración, se migraron todas las imágenes e iconos que estaban en formatos gráficos como PNG a “*vector drawable*”, que es un gráfico vectorial definido en un archivo XML como un conjunto de puntos, líneas y curvas junto con su información de color asociada como se observa en la Figura 10.18. La mayor ventaja de utilizar un “*vector drawable*” es la escalabilidad de la imagen. Se puede escalar sin pérdida de calidad de visualización, lo que significa que el mismo archivo se

redimensiona para diferentes densidades de pantalla sin pérdida de calidad de imagen. Esto resulta en archivos APK más pequeños y menos mantenimiento del desarrollador.

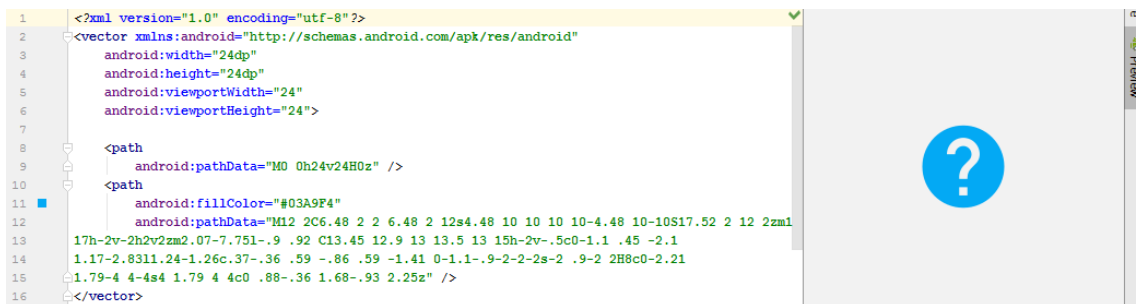


Figura 10.18: Ejemplo con Vista Previa de un “Vector Drawable”

Para lidiar con los diferentes tamaños de pantalla, se usaron los “calificadores de configuración”. Android admite varios que permiten controlar la forma en que el sistema selecciona los recursos alternativos según las características de la pantalla del dispositivo en uso. Un calificador de configuración, es una cadena de caracteres que se puede concatenar al directorio de recursos del proyecto de Android como se aprecia en la Figura 10.19 y especifica la configuración para la que se diseñan los recursos incluidos.

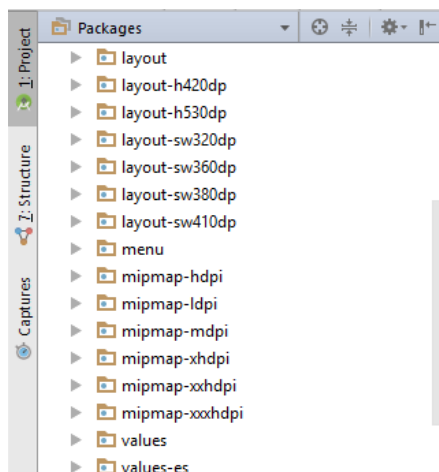


Figura 10.19: Directorio de Recursos del Proyecto de Android

Para los “layout” se utilizaron los nuevos calificadores de tamaños introducidos en la versión 3.2 de Android. Entre ellos se usó el calificador de “menor anchura” de la pantalla indicado por la dimensión más corta del área de la pantalla, es decir, es la parte más corta de la altura y el ancho disponibles, se usa al concatenar al directorio de recursos un guión seguido de las letras “sw” de “smallestWidth” con el tamaño en *dp*, como se observa en la Figura 10.19. El segundo calificador usado fue el de “altura” que especifica la altura de pantalla mínima, en unidades *dp*, a la que se deben usar los recursos contenidos en el directorio. Se debe destacar que el valor correspondiente del sistema para la altura, cambia cuando la orientación de la pantalla se alterna entre los modos horizontal y vertical; la cadena que debe ser concatenada al directorio de recursos para

ser usado este calificador, empieza con el guión seguido de la “h” de altura en inglés y el tamaño en *dp*, por ejemplo, *res/layout-h420dp*.

Como se observa en la Figura 10.19, los tamaños seleccionados fueron 320 *dp* para las pantallas más pequeñas, 360, 380 y hasta 410 *dp* para las más grandes. Por otra parte, los únicos “*layouts*” que necesitaron usar el calificador de “altura” fueron los relacionados a la autenticación debido a que, como ya se mostró anteriormente, estos poseen una serie de botones y formularios a lo largo de la pantalla, que para su correcto ajuste el calificador de altura se comporta perfectamente. Para este caso se seleccionaron los tamaños 420 y 530 *dp*.

Una vez adaptados todos los “*layouts*”, menos el de “*SelectProductFrament*” que se mejorará complemente más adelante, se empezó la mejora de la interfaz de usuario para el flujo principal de la aplicación a la que se le agregaron, como se observa en la Figura 10.20 , 3 pasos adicionales para mejorar la experiencia del usuario.

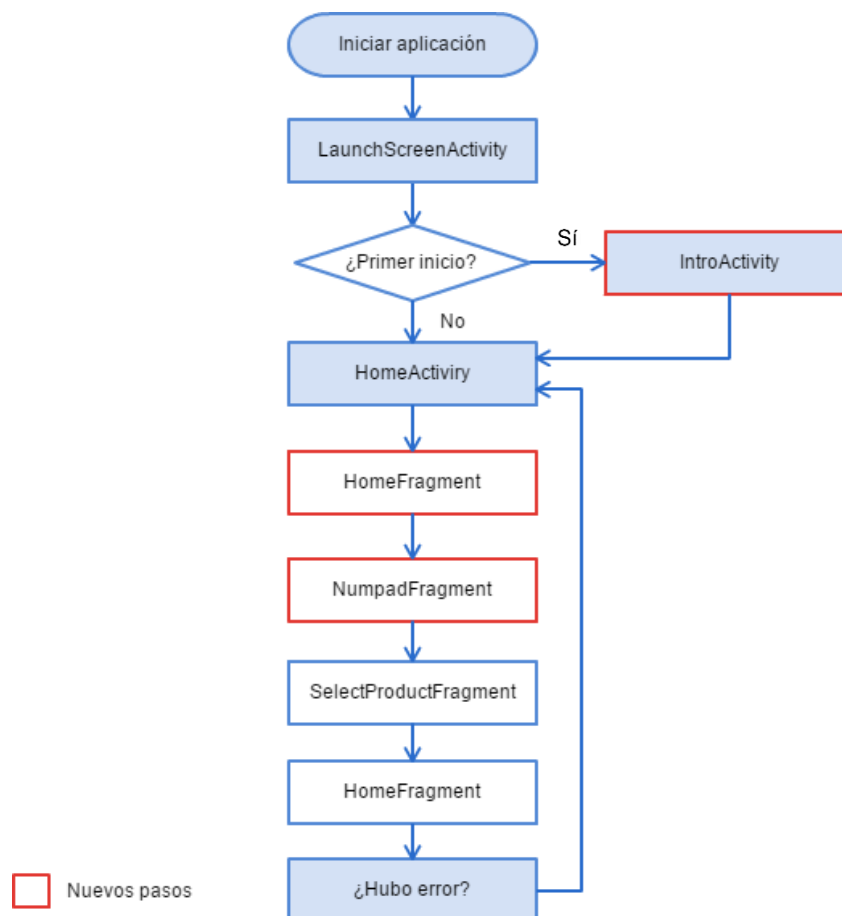


Figura 10.20: Nuevo Flujo Principal de la Aplicación

El primer paso agregado al flujo principal de la aplicación es la ejecución del “*IntroActivity*”, este muestra tres pantallas provicionales de la aplicación, como se ve en

la Figura 10.21, dicha ejecución debe suceder unicamente la primera vez que el usuario ejecuta la aplicación.

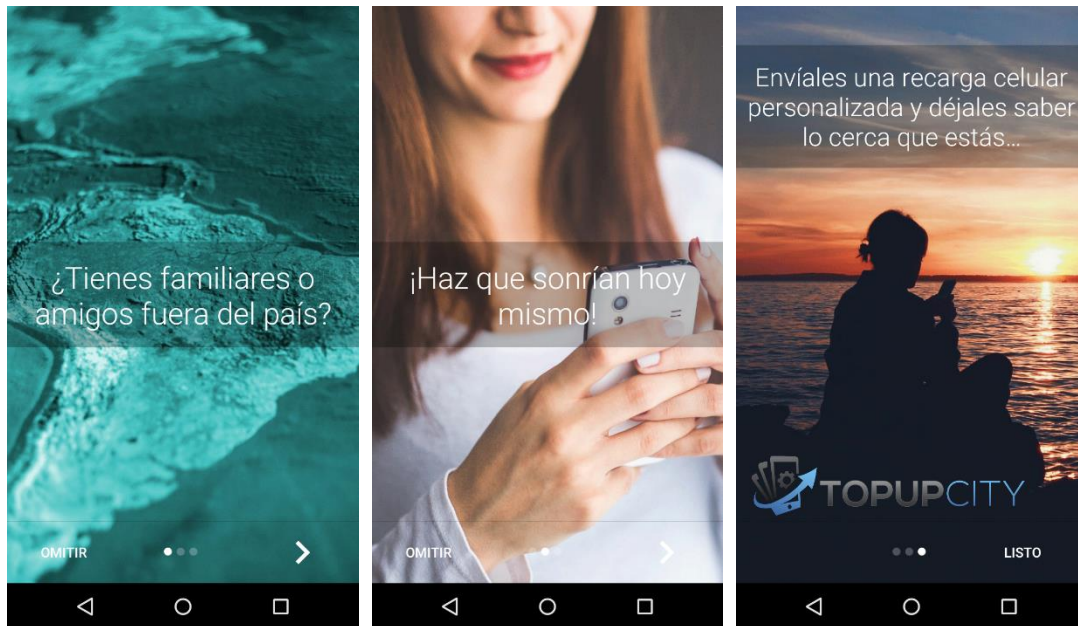


Figura 10.21: Pantallas Promocionales del “IntroActivity”

Una vez presionado el botón de “Listo” de la última pantalla promocional, se inicia el ya conocido “HomeActivity”, encargado de mostrar el menú lateral, manejar los “*fragments*” del proceso de pago y la integración con el SDK de PayPal. Anteriormente se iniciaba de primero el “SelectProductActivity”, en el que el usuario introducía el número destino y seleccionaba el producto, ahora este proceso se expandió a tres “*fragments*”, el primero de ellos es el “HomeFragment” observado en la Figura 10.22 encargado de recibir y dar bienvenida al usuario.

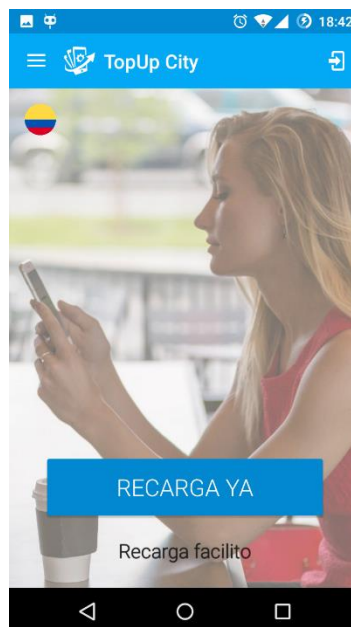


Figura 10.22: Interfaz Gráfica del “HomeFragment”

Además de mostrar el país que está seleccionado actualmente, el “*HomeFragment*”, también posee un botón que inicia el siguiente “*fragment*” que está encargado de capturar y validar el número destino de la recarga. Como se muestra en la Figura 10.23, el denominado “*NumpadFragment*” también muestra el país seleccionado actualmente junto con el código telefónico internacional y un campo de texto para introducir el número, este se puede ingresar haciendo uso del teclado numérico o activando el botón de contactos para buscar uno en la lista de contactos almacenados en el teléfono.

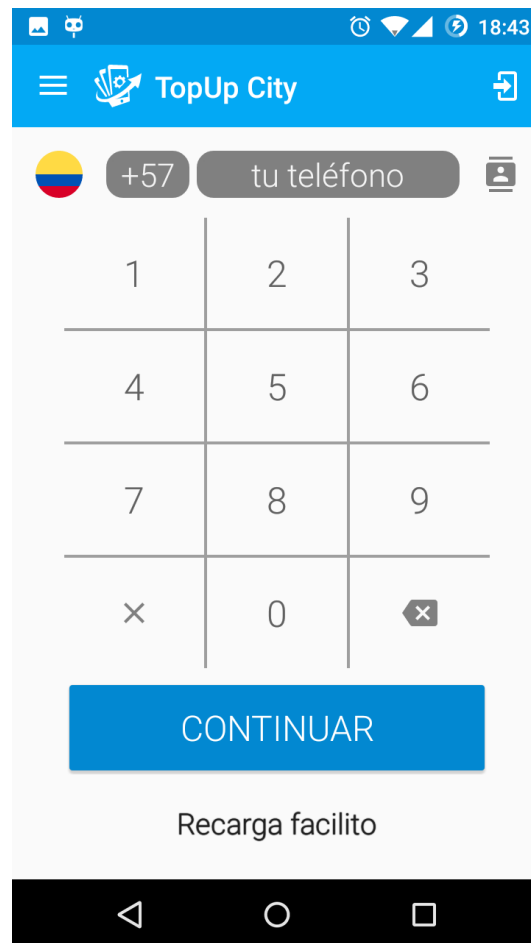


Figura 10.23: Interfaz Gráfica del “*NumpadFragment*”

Una vez introducido y validado el número destino de la recarga se inicia el “*SelectProductFragment*” con una interfaz gráfica mejorada como se aprecia en la Figura 10.25. Ahora las operadoras disponibles en el país seleccionado se muestran en las ya conocidas tarjetas o “*CardView*” con el nombre de la operadora como título de esta. Mientras que los productos se listan dentro la tarjeta a la cual pertenecen.

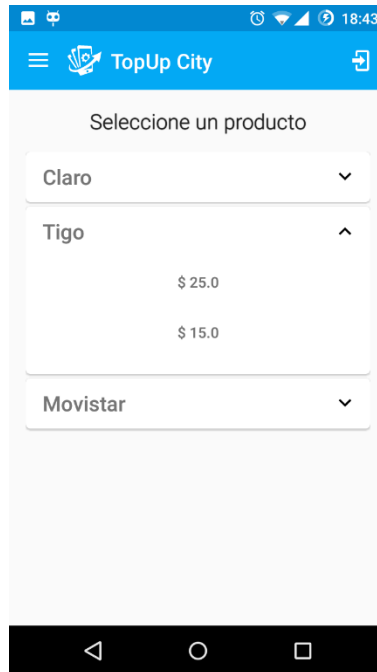


Figura 10.25: Nueva Interfaz Gráfica del “*SelectProductFragment*”

Por último, una vez seleccionado el producto, se inicia el ya conocido “*DetailFragment*”, con una interfaz gráfica mejorada, más limpia y agradable para el usuario. Entre los datos de la recarga que se muestran en este paso, se tiene el número y operadora destino, el monto del producto, comisión y total a pagar en dólares, y el monto a recargar en moneda del país destino como se observa en la Figura 10.24.

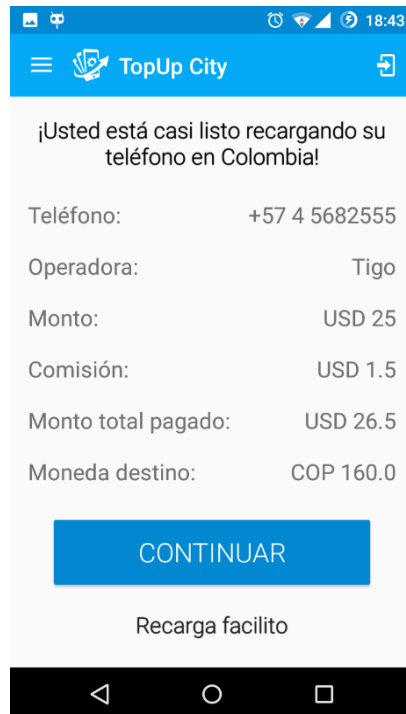


Figura 10.24: Nueva Interfaz Gráfica del “*DetailFragment*”

10.1.8 Configuración del Proyecto

A continuación, se explicará uno de los primeros pasos que se deben realizar al empezar el desarrollo de una aplicación para Android, como es la configuración del proyecto y de la aplicación. Se decidió explicar esto de último ya que a medida que se avanza el proceso de desarrollo, el contenido de los archivos y configuraciones que me mostrarán cambian continuamente, de esta forma se mostrará la configuración final de la aplicación.

Al crear un nuevo proyecto en Android Studio, este genera todo los archivos y directorios necesarios para empezar a desarrollar con configuraciones básicas por defecto. Cabe destacar, que un proyecto de Android puede contener varios módulos, es por ello que primero se hablará de la configuración del proyecto, esta es común y tendrá efecto para todos los módulos contenidos en el mismo.

El sistema de compilación de Android compila recursos y código fuente de la aplicación y los empaqueta en archivo APK que se pueden probar, implementar, firmar y distribuir. Android Studio usa “*Gradle*”, un paquete de herramientas de compilación avanzadas, para automatizar y administrar el proceso de compilación, y al mismo tiempo permite definir configuraciones de compilación personalizadas y flexibles.

El archivo “*build.gradle*” de nivel superior que se muestra en Figura 10.26, ubicado en el directorio raíz del proyecto, define configuraciones de compilación que se aplican a todos los módulos de un proyecto. De forma predeterminada, el archivo de nivel superior usa el bloque “*buildscript { }*” para definir los repositorios y las dependencias de “*Gradle*” comunes a todos los módulos del proyecto.

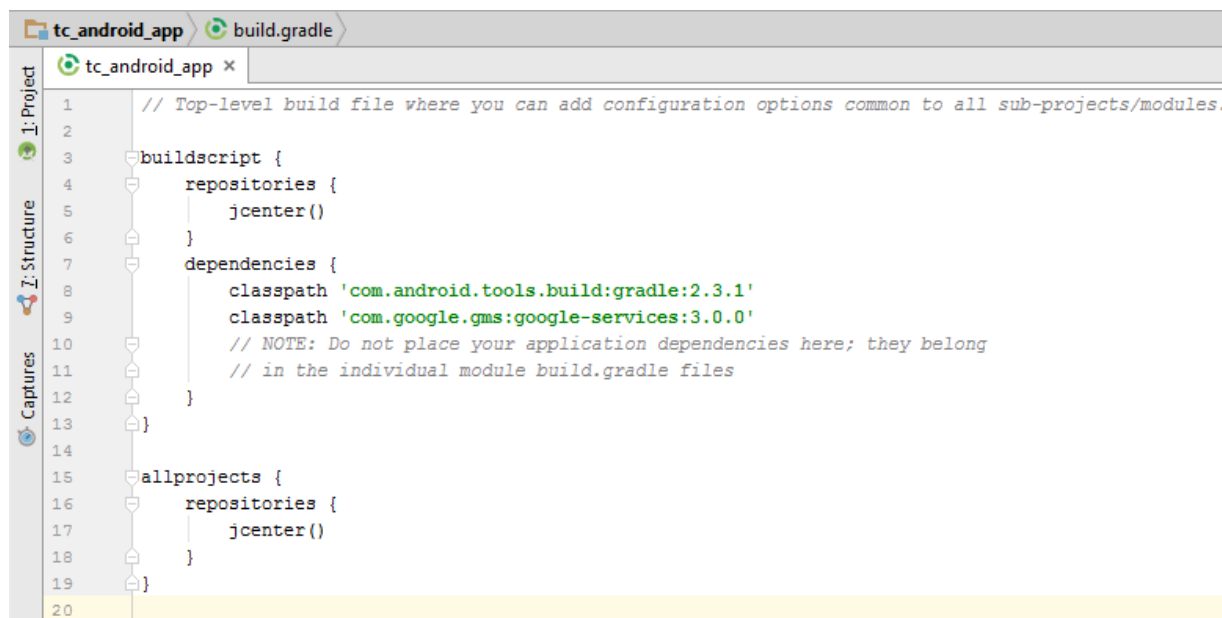


Figura 10.26: Contenido del Archivo “*build.gradle*” de Nivel Superior

El archivo “*settings.gradle*”, que también está ubicado en el directorio raíz del proyecto, indica a “*Gradle*” los módulos que debe incluir al compilar la aplicación. Para la mayoría de los proyectos, el archivo es sencillo como se observa en la Figura 10.27.

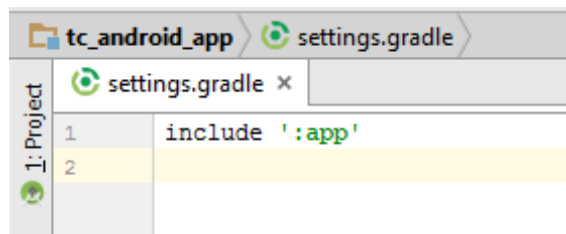


Figura 10.27: Contenido del Archivo “*settings.gradle*”

Gradle también incluye un archivo de propiedades, ubicado en el directorio raíz del proyecto, que se puede usar para especificar configuraciones del paquete de herramientas de compilación de Gradle. En el “*gradle.properties*” mostrado en la Figura 10.28, que se usa para almacenar las distintas constantes necesarias, utilizadas por los SDKs y el API REST.

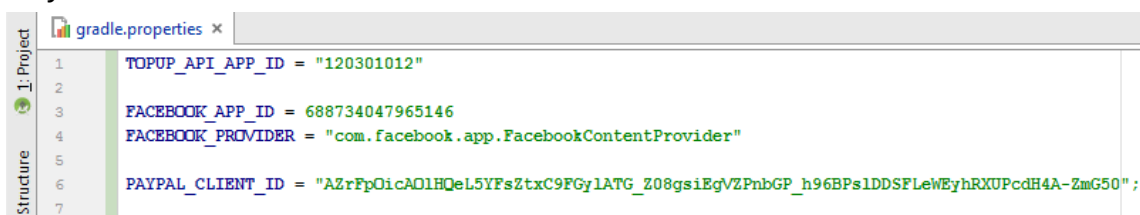


Figura 10.28: Contenido del Archivo “*gradle.properties*”

El siguiente archivo del cual se hablará es el archivo “*build.gradle*” de nivel de módulo, ubicado en cada directorio de “*tc_android_app/app/*”, que permite configurar ajustes de compilación para el módulo específico en el que se encuentra. La configuración de esos ajustes de compilación permite proporcionar opciones de empaquetado personalizadas, como tipos de compilación y tipos de productos adicionales, adicionalmente anular las configuraciones en el manifiesto de la aplicación o en el archivo “*build.gradle*” de nivel superior.

La primera línea del “*build.gradle*” aplica el “*plugin*” Android para que “*Gradle*” reconozca el bloque “*android {}*” está disponible para las opciones de compilación específicas para Android. Es en este bloque, en donde se especifica la versión del API de Android y la versión de las herramientas de compilación (*build tools*) del SDK de Android que debe usar “*Gradle*” para compilar la aplicación. Dentro de este, se encuentran los bloques “*defaultConfig {}*” encargados de identificar y versionar la aplicación; y el “*buildTypes {}*” donde se configuran los distintos tipos de compilación, por defecto Android Studio crea la configuración de la versión de “*publicación*”. En el proyecto se agregó también la versión de “*pruebas*” añadiendo las constantes creadas en el archivo “*build.properties*”.

En el bloque “*dependencies { }*” del archivo “*build.gradle*” de nivel de módulo, se especifican las dependencias necesarias para compilar dicho módulo, como lo son las librerías de compatibilidad, los SDKs de Facebook, Google y PayPal, las librerías HTTP como *Retrofit* y *Okhttp*, la librería de serialización *Gson* y por último una librería usada para la validación y formateo de números telefónicos llamada *libphonenumber*.

```

1  apply plugin: 'com.android.application'
2
3  android {
4      compileSdkVersion 25
5      buildToolsVersion "25.0.2"
6
7      defaultConfig {
8          applicationId "com.topupcity"
9          minSdkVersion 17
10         targetSdkVersion 25
11         versionCode 3
12         versionName "1.0.0"
13     }
14     buildTypes {
15         release {
16             minifyEnabled true
17             proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
18             buildConfigField "String", "TOPUP_API_APP_ID", project.property('TOPUP_API_APP_ID')
19             resValue "string", "FACEBOOK_APP_ID", project.property('FACEBOOK_APP_ID')
20         }
21         debug {
22             buildConfigField "String", "TOPUP_API_APP_ID", project.property('TOPUP_API_APP_ID')
23             resValue "string", "FACEBOOK_APP_ID", project.property('FACEBOOK_APP_ID')
24         }
25     }
26 }
27
28 dependencies {
29     compile fileTree(dir: 'libs', include: ['*.jar'])
30
31     //PAYPAL SDK
32     compile 'com.paypal.sdk:paypal-android-sdk:2.15.3'
33
34     //Soporte
35     compile 'com.android.support:appcompat-v7:25.3.1'
36     compile 'com.android.support:recyclerview-v7:25.3.1'
37     compile 'com.android.support:cardview-v7:25.3.1'
38     compile 'com.android.support:design:25.3.1'
39     compile 'com.android.support:support-v4:25.3.1'
40
41     //Redes Sociales
42     compile 'com.google.android.gms:play-services-plus:10.2.1'
43     compile 'com.google.android.gms:play-services-auth:10.2.1'
44     compile 'com.facebook.android:facebook-android-sdk:[4,5]'
45
46     //UI
47     compile 'com.mobsandgeeks:android-saripaar:2.0.3'
48     compile 'com.github.paolorotolo:appintro:4.1.0'
49
50     //Api connections and http
51     compile 'com.squareup.retrofit2:retrofit:2.1.0'
52     compile 'com.squareup.retrofit2:converter-gson:2.1.0'
53     compile 'com.squareup.okhttp3:okhttp:3.5.0'
54     compile 'com.squareup.okhttp3:logging-interceptor:3.5.0'
55
56     //Util
57     compile 'com.google.code.gson:gson:2.8.0'
58     compile 'com.googlecode.libphonenumber:libphonenumber:8.3.1'
59 }
60
61 apply plugin: 'com.google.gms.google-services'

```

Figura 10.29: Contenido del Archivo “*build.gradle*” de Nivel de Módulo

10.2 Análisis General de la Consola Administrativa

La consola administrativa es una aplicación web desarrollada con Spring Boot, un framework basado en Java para aplicaciones web, y dentro de sus funciones permite programar los controladores de la aplicación, cabe acotar que se empleó una base de datos MySQL del lado del modelo. Además, se utilizó “*PrimeUI*”, una librería basada en JavaScript para realizar la integración con los controladores y las vistas mediante llamadas *Ajax*.

El IDE usado para desarrollar la consola administrativa fue IntelliJ IDEA 2016.2, el cual provee una fácil integración con Spring y “*Maven*”, un sistema de compilación automático.

Para el desarrollo se utilizó Spring Boot 1.3.3 haciendo uso de JDK 1.8 y el conector de MySQL 5.1.38, para poder acceder a la base de datos. Por su parte, la aplicación fue desarrollada en un servidor Tomcat 8 en una MacBook Pro corriendo el sistema operativo OS X Yosemite (versión 10.10.5). Del lado del cliente se utilizó PrimeUI 4.1.0 y Google Chrome como navegador web.

A continuación, se describirán cada una de las iteraciones del proceso de desarrollo de la aplicación.

10.2.1 Configuración del Ambiente

Antes de comenzar el desarrollo de la aplicación, fue necesario configurar el ambiente, lo que implicó definir el archivo “*pom.xml*” de Maven para automatizar el proceso de compilación e inclusión de librerías, configurar el archivo “*application.properties*” que contiene los datos de acceso a la base de datos y otras variables de configuración. Adicionalmente, fue necesario definir la carpeta donde se encuentran las vistas y las plantillas utilizadas y para esto último se utilizó Apache Tiles, que requiere un archivo “*tiles-definitions.xml*” para definir los templates y por último se necesitó crear diferentes archivos “*message.properties*”, para la internacionalización de la aplicación web. Todo esto será explicado con detalle a continuación.

Como se mencionó con anterioridad, para empezar, fue necesario definir el Modelo Objeto de Proyecto por sus siglas en inglés “*Project Object Model*”, este es un archivo XML que contiene información acerca del proyecto y detalles de configuración usados por Maven para lograr construir el proyecto. En este caso se definió un “*pom.xml*” bastante sencillo, que contiene el nombre del proyecto y una descripción, a su vez posee la versión del proyecto, la forma en que va a ser empaquetado y en este caso será un Archivo de Aplicación Web o *WAR* por sus siglas en inglés “*Web Application Archive*” y el nombre del paquete o “*groupId*”, todo esto se aprecia en la Figura 10.30.

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
4    <modelVersion>4.0.0</modelVersion>
5
6    <groupId>net.topupcity</groupId>
7    <artifactId>admin</artifactId>
8    <version>0.0.1-SNAPSHOT</version>
9    <packaging>war</packaging>
10
11    <name>admin</name>
12    <description>Demo project for Spring Boot</description>
13
14    <parent...>
15
16    <properties>
17      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
18      <java.version>1.8</java.version>
19    </properties>
20
21    <dependencies...>
22
23    <build...>
24
25  </project>

```

Figura 10.30: Archivo “pom.xml”

Es necesario destacar que en el pom.xml, no sólo se definen variables de configuración, también es posible incluir dependencias al proyecto, lo que facilita la forma en la que se importan las librerías al proyecto, en la aplicación se añadieron algunas dependencias necesarias, como por ejemplo, el conector de MySQL, esto se logró al poner una etiqueta XML, que contiene el nombre del paquete, el ID de la dependencia y la versión deseada en el documento para que Maven pudiera determinar que dependencia incluir en el proyecto, como se muestra en la Figura 10.31.

```

72  <dependency>
73    <groupId>mysql</groupId>
74    <artifactId>mysql-connector-java</artifactId>
75    <version>5.1.38</version>
76  </dependency>

```

Figura 10.31: Dependencia de MySQL Requerida por Maven

Spring Boot permite externalizar la configuración del proyecto mediante un archivo llamado “application.properties”, lo que permite trabajar la misma aplicación en diferentes ambientes. En este caso, contiene los datos de acceso a la base de datos como nombre de usuario, contraseña y la URL de acceso, como se muestra en la Figura 10.32.

```

1  spring.datasource.url=URL
2  spring.datasource.username=USER
3  spring.datasource.password=PASSWORD
4  #spring.datasource.url=jdbc:mysql://localhost/topup_city_db?logger=com.mysql.jdbc.log.Slf4JLogger&profileSQL=true
5  #spring.datasource.username=root
6  #spring.datasource.password=
7  spring.datasource.driver-class-name=com.mysql.jdbc.Driver

```

Figura 10.32: Contenido del Archivo “*application.properties*”

Para poder configurar el directorio de las vistas y las plantillas, fue necesario crear “*Spring Beans*” como se muestra en la Figura 10.33, estos son objetos que contienen la metadata de la configuración. En este caso, se definieron dos Beans, el primero para definir la carpeta de las vistas y el sufijo de los archivos, que en este caso fueron de tipo JSP y el segundo para definir el archivo de configuración de Apache Tiles donde se define la estructura de las plantillas a utilizar.

```

@Bean
public ViewResolver getViewResolver(){
    InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();
    viewResolver.setPrefix("/WEB-INF/pages/");
    viewResolver.setSuffix(".jsp");
    return viewResolver;
}

@Bean
public TilesConfigurer tilesConfigurer(){
    TilesConfigurer tilesConfigurer = new TilesConfigurer();
    tilesConfigurer.setDefinitions("/WEB-INF/tiles/tiles-definitions.xml");
    return tilesConfigurer;
}

```

Figura 10.33: Spring Beans para Definir la Capa de la Vista

Apache Tiles permite definir fragmentos de página que pueden ser ensamblados para completar las vistas en tiempo de ejecución. En este caso, fue necesario definir dos plantillas, la primera es la pantalla de Login, la cual no contiene elementos reutilizables según se ve en los prototipos definidos en el capítulo anterior y la segunda plantilla contiene tanto la cabecera y el menú lateral que se aprecian en la Figura 9.11. La configuración de Apache Tiles se logra mediante un archivo XML donde se definen los nombres de las plantillas y el archivo JSP asociado, así como se aprecia en la Figura 10.34.

```

<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE tiles-definitions PUBLIC
    "-//Apache Software Foundation//DTD Tiles Configuration 3.0//EN"
    "http://tiles.apache.org/dtds/tiles-config_3_0.dtd">
<tiles-definitions>

    <definition name="defaultTemplate" template="/WEB-INF/templates/template.jsp">
        <put-attribute name="header" value="/WEB-INF/templates/header.jsp" />
    </definition>

    <definition name="loginTemplate" template="/WEB-INF/templates/loginTemplate.jsp">
    </definition>

</tiles-definitions>

```

Figura 10.34: Contenido del Archivo “tiles-definitions.xml”

Para finalizar con la configuración del ambiente se crearon dos archivos, “message_es.properties” y “message_en.properties”, que contienen todos los mensajes de la aplicación en español y en inglés respectivamente, de modo que al necesitar un mensaje se utiliza el nombre de la variable y se mostrará en el idioma con el cual se encuentra configurado el navegador. En caso de que el navegador se encuentre con un idioma distinto a estos, se utilizarán los mensajes definidos por defecto en el archivo “message.properties”. Un ejemplo de algunos de estos mensajes, se puede apreciar en la Figura 10.35.

<pre> #Header template messages lbl.header.admin = TopUp City Admin lbl.header.user.management = User Management lbl.header.logout = Logout lbl.header.option.dashboard = Dashboard lbl.header.option.accounts = Accounts lbl.header.option.countries = Countries lbl.header.option.currencies = Currencies lbl.header.option.carriers = Carriers lbl.header.option.prefixes = Prefixes lbl.header.option.carrier.sub = Carrier Products lbl.header.option.transaction = Transactions </pre>	<pre> #Header template messages lbl.header.admin = Consola de TopUp City lbl.header.profile = Perfil lbl.header.user.management = Administraci\u00f3n de usuarios lbl.header.logout = Cerrar sesi\u00f3n lbl.header.option.dashboard = Tablero lbl.header.option.accounts = Cuentas lbl.header.option.countries = Pa\u00edses lbl.header.option.currencies = Monedas lbl.header.option.carriers = Telefon\u00e1das lbl.header.option.prefixes = Prefijos lbl.header.option.carrier.sub = Productos de telefon\u00e1das lbl.header.option.transaction = Transacciones </pre>
--	---

Figura 10.35: Ejemplo del Archivo “message.properties”

10.2.2 Desarrollo de la Pantalla de Login

Para iniciar esta iteración, lo primero fue maquetar la vista siguiendo los prototipos presentados anteriormente. Esto se logró haciendo uso de la plantilla para la vista del *login* previamente definida la cual contenía enlaces a archivos CSS y JavaScript únicamente. Para poder hacer uso de esta, se requiere utilizar una etiqueta de Apache Tiles para poder incluirla como se puede ver en la Figura 10.36. La segunda etiqueta permite definir el contenido de la página y este es construido haciendo uso de etiquetas

HTML5 y clases de *bootstrap* para aplicar estilos a estos elementos obteniendo un diseño adaptivo.

```
<%@ taglib prefix="tiles" uri="http://tiles.apache.org/tags-tiles"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<%@ taglib prefix="spt" uri="http://www.springframework.org/tags" %>

<tiles:insertDefinition name="loginTemplate">
  <tiles:putAttribute name="content">

    <div class="row"...>

      <hr><br>

      <script type="text/javascript"...>

    </tiles:putAttribute>
  </tiles:insertDefinition>
```

Figura 10.36: Ejemplo de Inclusión de una Plantilla

El contenido de la vista del login se basó en un panel que contiene un título, y en este punto, se puede evidenciar el uso de los mensajes internacionalizados ya que, no se utiliza un texto plano, sino que se utiliza una referencia al mensaje almacenado en la variable “*lbl.login.signin*”, como se observa en la Figura 10.37. Además, posee tres campos tipo *input*, el primero para el nombre de usuario, el segundo para la contraseña y el último es un botón para poder confirmar que se desea enviar los datos del formulario. Cabe acotar que no se utilizaron simples etiquetas HTML, también se hizo uso de etiquetas de Spring a la par, nótese que al crear el formulario se definió un atributo denominado “*commandName*” y el valor de este determina el tipo de objeto que será recibido en los controladores, al comenzar el formulario se define una sección de errores en donde se mostrarán mensajes según sea el caso.

```
<div class="login-panel panel panel-default">
  <div class="panel-heading">
    <h3 class="panel-title"><spt:message code="lbl.login.signin"/></h3>
  </div>
  <div class="panel-body">
    <form:form commandName="loginForm" method="POST" role="form">
      <form:errors path="*" element="div" cssClass="spring-errors" />
      <fieldset style="margin-top: 15px;">
        <div class="form-group">
          <form:input path="login" cssClass="form-control" cssErrorClass="validation-error form-control" placeholder="Login" />
        </div>
        <div class="form-group">
          <form:password path="password" cssClass="form-control" cssErrorClass="validation-error form-control" placeholder="Password" />
        </div>
        <input type="submit" value="Login" class="btn btn-lg btn-success btn-block" />
      </fieldset>
    </form:form>
  </div>
</div>
```

Figura 10.37: Formulario para Iniciar Sesión

La vista del *login* tiene asociado dos controladores, el primero de tipo GET que se utiliza para poder acceder al *home* en caso que se encuentre una sesión activa, sino simplemente se accede a la vista del login, así como lo indica la Figura 10.38. El segundo controlador es de tipo POST y se utiliza para recibir los datos del formulario, validarlos y dar acceso al sistema o rechazar la petición y dar algún mensaje de error. Esta es la forma en la que se trabajó las vistas, en este caso, se explicarán ambos controladores, pero para evitar redundar en las siguientes iteraciones únicamente se dará detalle de los controladores que poseen la lógica del negocio.

```
@RequestMapping(value = Paths.LOGIN)
public String viewLogin(Map<String, Object> model, LoginForm loginForm, HttpSession httpSession){
    log.info("Entrando al login GET...");
    User user = (User) httpSession.getAttribute(TopupCityConstants.USER);
    if (user != null)
        return "home";
    model.put("serverTime", new Date());
    return "login";
}
```

Figura 10.38: Controlador Utilizado para Acceder al Login

Cuando se hace uso de Spring y se quiere definir un nuevo controlador, se debe hacer uso de la anotación *@RequestMapping* donde se puede definir la ruta del controlador, el tipo de petición HTTP y el tipo de dato que consume y produce. Ahora, si se quiere acceder a una vista, se tendría que retornar un *string* que contenga el nombre del archivo.

Como se puede ver en la Figura 10.38, se tiene una función llamada *viewLogin*, que retorna un *string*, esta recibe por parámetros una variable de tipo *Map<String, Object>* que se utiliza para enviar datos a las vistas y una sesión HTTP, la cual posee almacenado un usuario en caso de que ya se haya iniciado sesión anteriormente. Básicamente el controlador accede a la sesión en búsqueda de un usuario activo, en caso de que exista se podrá acceder al *home* y sino simplemente se accede a la vista del *login*.

```
public class LoginForm implements Serializable {

    private String login;
    private String password;

    @Size(min = 1, max = 20)
    public String getLogin() { return login; }

    public void setLogin(String login) { this.login = login; }

    @Size(min = 1, max = 100)
    public String getPassword() { return password; }

    public void setPassword(String password) { this.password = password; }
}
```

Figura 10.39: Clase LoginForm

El segundo controlador es el encargado de recibir los datos, validarlos y permitir o rechazar el acceso. El formulario que recibe este controlador es de tipo *LoginForm* y contiene los campos requeridos para iniciar sesión como se muestra en la Figura 10.39.

Estos datos son validados con el modelo de datos, y en el caso de Spring se establece una correspondencia con el modelo, haciendo uso de una clase que contenga dos anotaciones que indican que son una entidad y el nombre de la tabla con la que se quiere establecer la relación, así como se muestra en la Figura 10.40. Es necesario aclarar que esta clase debe contener los mismos atributos que la tabla en la base de datos con todos los métodos de acceso, por motivos prácticos se omite.

```
@Entity
@Table(name = "users")
public class User implements Serializable {

    private static final long serialVersionUID = 1L;
    private Integer id;
    private String login;
    private String password;
    private String email;
    private String firstName;
    private String lastName;
    private Date creationDate;

}
```

Figura 10.40: Modelo de Usuarios

Sin embargo, esto no es suficiente, para poder alterar o crear registros en la base de datos, se tiene que hacer uso del ORM de Spring o en este caso JPA. Y para esto se tuvo que definir una interfaz que tenía que extender de alguna otra interfaz definida por JPA, en este caso se utilizó *JpaRepository* que provee métodos para realizar operaciones CRUD, paginación y ordenamiento de registros. Es importante destacar que se debe especificar el modelo al cual está asociado y el tipo de dato de la clave primaria.

En caso de necesitar alguna búsqueda un poco más específica, basta con definir una firma de una función que empiece con alguno de estos prefijos: *findBy*, *readBy*, *queryBy*, *countBy*, *getBy*, seguidos por el o los campos que se quieren en la búsqueda, unidos por un *and* y usando el estilo de escritura *camelCase*, así como se muestra en la Figura 10.41.

```

public interface UserDao extends JpaRepository<User, Integer> {

    User findByLogin(String login);

    User findById(Integer id);

}

```

Figura 10.41: Interfaz DAO para los Usuarios

En la Figura 10.42 se observa el diagrama de flujo del segundo controlador, donde en una primera instancia se validan los datos recibidos, de forma que se pueda asegurar que sean el tipo de datos esperados y que cumpla con los requerimientos del negocio. Luego se utiliza la interfaz anteriormente definida para buscar al usuario y en caso de que exista, se verifica la contraseña que este posee almacenada con la que se está recibiendo, y sólo en caso de que estos tres escenarios sean exitosos, se podrá acceder al *home*, en caso contrario se regresa al *login*.

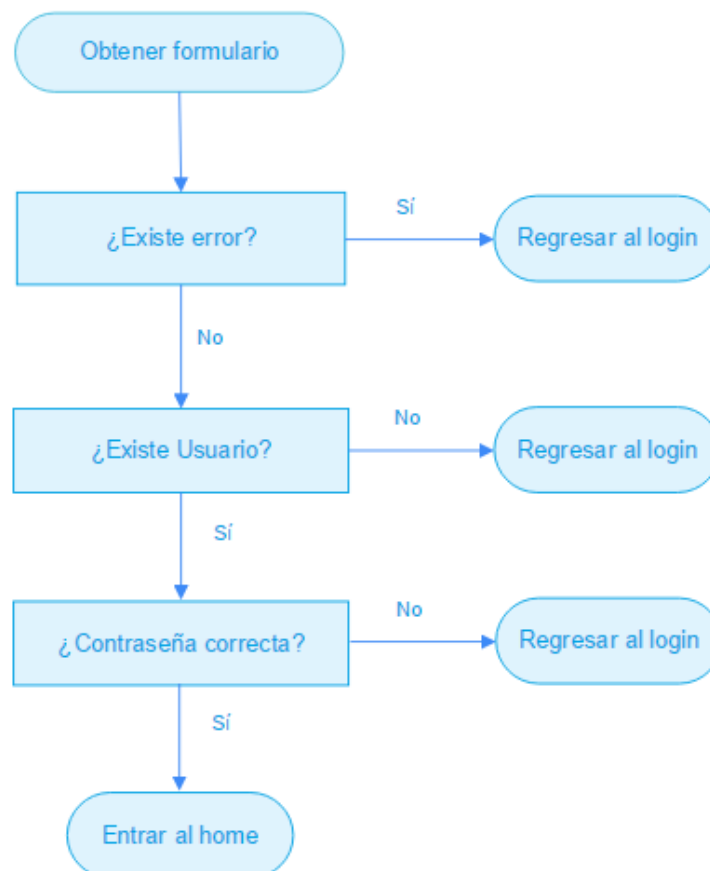


Figura 10.42: Diagrama de Flujo del Login

10.2.3 Desarrollo del Tablero o Pantalla Inicial

Para iniciar esta iteración, lo primero fue maquetar la plantilla general que iba a ser utilizada en toda la aplicación, que consta de un menú en la barra superior para el control de usuarios que acceden a la consola web y para cerrar sesión, además de un menú en la barra lateral izquierda para poder navegar entre los recursos administrables. Esto se logra haciendo uso de etiquetas HTML y clases de *Bootstrap* para facilitar y agilizar el trabajo. En la Figura 10.43, se pueden observar las estructuras utilizadas para obtener el menú que se encuentra en la barra superior.

```
<!-- BARRA MENU DERECHO -->
<ul class="nav navbar-top-links navbar-right navbar-collapse">
  <li class="dropdown pull-right" style="...">
    <a class="dropdown-toggle" data-toggle="dropdown" href="#">
      <i class="fa fa-user fa-fw"></i> <i class="fa fa-caret-down"></i>
    </a>
    <ul class="dropdown-menu dropdown-user">
      <li><a href="{userManagement}"><i class="fa fa-user-plus fa-fw"></i><spt:message code="lbl.header.user.management"/></a>
      </li>
      <li class="divider"></li>
      <li>
        <a href="{loginUrl}"><i class="fa fa-sign-out fa-fw"></i><spt:message code="lbl.header.logout"/></a>
      </li>
    </ul>
  </li>
</ul>
<!-- FIN BARRA MENU DERECHO -->
```

Figura 10.43: Menú Superior Utilizado en la Plantilla General

El resultado de utilizar la estructura HTML mencionada con estas clases Bootstrap en específico, permite obtener el menú que se aprecia en la Figura 10.44. Además, facilita el trabajo a la hora de obtener un diseño adaptativo.

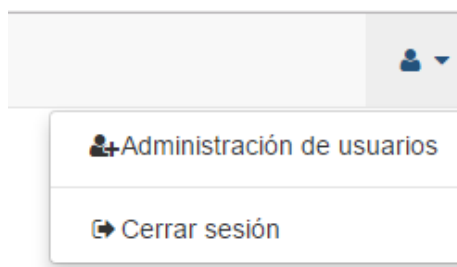


Figura 10.44: Menú Superior

Nótese que en la Figura 10.43, se puede apreciar que las referencias de hipertexto o atributos *href* no poseen texto plano, esto debido a que en los archivos JSP es posible incluir código Java, así como definir variables y en este caso para tratar en lo posible de reutilizar texto, se almacenaron las rutas en variables. Se puede ver en la Figura 10.45.

```

<spt:url value="/" var="homeUrl" htmlEscape="true"/>
<spt:url value="/accounts" var="accountsUrl" htmlEscape="true"/>
<spt:url value="/countries" var="countriesUrl" htmlEscape="true"/>
<spt:url value="/currencies" var="currenciesUrl" htmlEscape="true"/>
<spt:url value="/carriers" var="carriersUrl" htmlEscape="true"/>
<spt:url value="/logout" var="loginUrl" htmlEscape="true"/>
<spt:url value="/users" var="userManagement" htmlEscape="true"/>
<spt:url value="/prefixes" var="prefixesUrl" htmlEscape="true"/>
<spt:url value="/products" var="productsUrl" htmlEscape="true"/>
<spt:url value="/transactions" var="transactionsUrl" htmlEscape="true"/>

```

Figura 10.45: Variables en Archivos JSP

Para la realización del menú lateral izquierdo se creó una lista y cada elemento era un enlace a un recurso en específico, además se utilizaron iconos de la librería *Font Awesome* para aumentar la usabilidad de la aplicación. En la Figura 10.46, se puede apreciar el resultado final.



Figura 10.46: Menú Lateral

Al finalizar el desarrollo de la plantilla general, se procedió a trabajar en la vista del *home*, donde se pretende mostrar un conteo de los registros del sistema. Para lograr esto se comenzó maquetando el contenido de la vista, teniendo en cuenta los prototipos antes presentados, es decir, se crearon seis contenedores en forma de círculos para luego mostrar los contadores. Esto se logró utilizando una etiqueta *div* a la cual se aplicaron estilos de CSS3 para redondear los bordes y obtener un contenedor circular. En la Figura 10.47, se pueden observar las etiquetas utilizadas, así como la clase de CSS definida para obtener el resultado final.

```

<div class="col-lg-4 col-md-4 col-sm-4 circle">
  <div class="row circle-title">
    <p><spt:message code="lbl.header.option.countries"/></p>
  </div>
  <div class="circle-body">
    <p id="countCountries">${countCountries}</p>
  </div>
</div>

```

```

}.circle {
  width: 169px;
  height: 169px;
  -moz-border-radius: 50%;
  -webkit-border-radius: 50%;
  border-radius: 50%;
  background: #f8f8f8;
  margin-left: 10%;
  margin-top: 2%;
  border-color: #0e90d2;
  border-width: 2px;
  border-style: solid;
}

```

Figura 10.47: Contenedor de Datos del Home

Como se puede observar en la Figura 10.47, los contenedores poseen un título y un cuerpo. En este último es donde se ubicaron los contadores proporcionados por el controlador, que en este caso se encarga de contabilizar los registros. Esto se logró utilizando las interfaces de JPA definidas para cada recurso administrable del sistema, al obtener el conteo de los registros se almacenan en una variable tipo *Map*. Este funcionamiento se aprecia en la Figura 10.48.

```

@RequestMapping(value = Paths.INDEX)
public String home(Map<String, Object> model){
    Date date = new Date();
    SimpleDateFormat format = new SimpleDateFormat( pattern: "yyyy-MM-dd HH:mm:ss");

    log.info("Entrando al Home...");
    model.put("countUser", accountDao.count());
    model.put("countTrans", transactionDao.count());
    model.put("countCountries", countryDao.count());
    model.put("countCurrencies", currencyDao.count());
    model.put("countCarriers", carrierDao.count());
    model.put("countProd", carrierProductsDao.count());
    model.put("serverTime", format.format(date));

    return "home";
}

```

Figura 10.48: Controlador del Home

El resultado final de integrar la vista con el controlador que provee el conteo de los recursos, se puede apreciar en la Figura 10.49.



Figura 10.49: Resultado Final del Home

10.2.4 Desarrollo de la Administración de Usuarios

Una vez llegado a este punto se contaba ya con todas las plantillas y los recursos listos, por lo que de ahora en adelante lo único necesario era desarrollar el contenido de las vistas y los controladores asociados.

En esta iteración fue necesario desarrollar las cuatro operaciones básicas que son *Read*, *Create*, *Update* y *Delete*, será en este mismo orden que se explicarán en esta iteración y todas las restantes.

Para empezar a listar todos los usuarios administradores fue necesario definir la estructura a utilizar, por lo que se empezó a emplear la librería *Prime UI* que fue la encargada de facilitar la integración entre nuestros controladores y las tablas a utilizar en este y en los demás recursos administrables.

Para hacer uso del *Datatable* de la librería, lo primero que se tuvo que hacer fue definir la estructura HTML de la tabla, la misma se puede apreciar en la Figura 10.50.

```
<a href="{addUser}" class="btn btn-default"><i class="fa fa-plus fa-fw"></i><spt:message code="lbl.user.add" /></a>
<a href="{userManager}" class="btn btn-default"><i class="fa fa-history fa-fw"></i></a>
<div style="margin-top: 4px;"class="row">
  <div class="col-lg-12">
    <div id="tblUserList"></div>
  </div>
</div>
```

Figura 10.50: Estructura de la Tabla de los Recursos Administrables

Nótese que básicamente se tiene una etiqueta *div* con una clase *tblUserList*. Luego se crea la tabla haciendo uso de la librería, para esto bastó con hacer uso de un selector de

JQuery para obtener el elemento del DOM con la clase *tblUserList* y utilizar una función llamada *puidatatable* que es parte de la librería, como se observa en la Figura 10.51.

```
$('#tblUserList').puidatatable({
    lazy: true,
    caption: '<spt:message code="lbl.user.list" />',
    responsive: true,
    paginator: {...},
    columns: [...],
    datasource: function(callback, ui) {...}
});
```

Figura 10.51: Script para Crear las Tablas con *Prime UI*

La función *puidatatable* requiere de ciertos parámetros indispensables para poder funcionar correctamente, entre los que más interesan se encuentran los objetos *paginator*, *columns*, y *datasource*. El primero utilizado para paginar la tabla, este objeto posee dos variables de gran interés, la primera es usada para determinar cuántos registros se deseaban mostrar por página y la segunda para indicar cuantos registros se tenían almacenados en su totalidad, se aprecia mejor en la Figura 10.52.

```
paginator: {
    rows: 5,
    totalRecords: ${totalRecords}
},
```

Figura 10.52: Objeto *Paginator* de la Función *Puidatatable*

Se puede observar en la Figura 10.52 que para indicar la cantidad de registros que se poseen almacenados, se utiliza una variable *totalRecords*, este valor se obtiene de un controlador que poseen todas las vistas principales de los recursos administrables y solo se explicará en este primer caso para evitar redundar.

Los controladores se pueden separar en controladores *Ajax* y *NonAjax*, al acceder a cualquier recurso administrable del sistema, el primer controlador que se necesita es de tipo *NonAjax* y esto se debe a que se accede a él realizando una llamada síncrona, que se encarga de contabilizar los registros y almacenarlos en la variable *totalRecords*, se puede observar con detalle en la Figura 10.53.

```
@RequestMapping(value = Paths.USERS_LIST_PATH)
public String loadView(Map<String, Object> model){
    log.info("##### entrando a userList #####");
    long count = userDao.count();
    model.put("totalRecords", count);
    return "userList";
}
```

Figura 10.53: Controlador para Acceder a la Lista de Usuarios

El segundo objeto que recibe la función *puidatatable* es *columns*, es un arreglo de objetos JavaScript y cada uno de estos indica que datos de la respuesta serán mostrados, así como el nombre de la columna y si es posible ordenarlos o no. Nótese que en la Figura 10.54, la última columna no posee un campo de la respuesta, sino que posee una función JavaScript y esto se hizo con el propósito de colocar una columna de opciones desde donde es posible editar, eliminar o ver el detalle de un registro en específico. Esto se logra retornando una variable que contenga los botones necesarios.

```
columns: [
  {field: 'id', headerText: '<spt:message code="lbl.user.tbl.header.id" />', sortable:true},
  {field: 'login', headerText: '<spt:message code="lbl.user.tbl.header.login" />', sortable:true},
  {field: 'email', headerText: '<spt:message code="lbl.user.tbl.header.email" />', sortable:true},
  {field: 'firstName', headerText: '<spt:message code="lbl.user.tbl.header.fisrtName" />', sortable:true},
  {field: 'lastName', headerText: '<spt:message code="lbl.user.tbl.header.lastName" />', sortable:true},
  {headerText: '<spt:message code="lbl.user.tbl.header.options" />',
    content: function(row){...}
  },
],
```

Figura 10.54: Objeto *Columns* de la Función *Puidatatable*

El último objeto requerido para armar la tabla es el *datasource*, una función que ejecutará una llamada asíncrona al controlador para obtener los registros y así poder llenar cada una de las páginas de la tabla de forma dinámica. Esta función utiliza la llamada Ajax de JQuery, que recibe un conjunto de pares clave-valor especificando el tipo de petición HTTP, la data a enviar, la URL del controlador y el tipo de dato que será tratado. A su vez, posee múltiples *callbacks* para poder identificar el estado de la petición, sin embargo, se estudiarán los estados *success* y *error*. Con el primero se pudo identificar que la llamada se había ejecutado correctamente además de enlazar la tabla con la respuesta y con el segundo que había ocurrido un error en la petición. La estructura de esta función fue utilizada en las iteraciones restantes y se puede observar en la Figura 10.55.

```
$.ajax({
  type: "GET",
  data:data,
  url: uri,
  dataType: "json",
  context: this,
  success: function(response) {
    $('#spinner').removeClass("fa fa-refresh fa-spin");
    console.log("Exitoso...");
    callback.call(this, response);
  },
  error: function(err){
    $('#spinner').removeClass("fa fa-refresh fa-spin");
    console.log("Error...");
    console.log(err);
    loadErrorDlg(err);
  }
})
```

Figura 10.55: Petición Ajax del Objeto *Datasource*

Una vez teniendo listo todo del lado del cliente, se puede analizar cómo funciona el controlador que permite llenar la tabla. Como se mencionó anteriormente, al utilizar *JpaRepository*, se tiene acceso a funciones de paginación que resultan prácticas en este caso. Es necesario crear un objeto *PageRequest* con el que sea posible especificar el index para que inicie la paginación, el tamaño de la página, si se quiere ordenar de forma ascendente o descendente y el campo de ordenamiento. Al tener esto, fue necesario hacer uso de la interfaz DAO para los usuarios, por sus siglas en inglés Data Access Object u Objeto de Acceso a Datos, y hacer un *findAll* indicando nuestro objeto de paginación para luego obtener una lista de usuarios, finalmente retornar esta lista como un objeto JSON que puede ser especificado al momento de crear un controlador con la anotación *@RequestMapping*. El proceso recientemente descrito puede ser observado en la Figura 10.56.

```
@RequestMapping(value = Paths.USERS_GRID_PATH, produces = "application/json")
@ResponseBody
public List<User> listUser(@RequestParam Integer start, @RequestParam String sortField,
                          @RequestParam String sortOrder, @RequestParam Integer rows){

    log.info("##### pagination info start:{}, rows:{}, sortField:{}, sortOrder:{}, page :{}", start, rows, sortField, sortOrder, (start / rows));
    PageRequest pageRequest = new PageRequest((start/rows), rows, Sort.Direction.fromString(sortOrder), StringUtils.isEmpty(sortField) ? "id" : sortField);

    Page<User> userPage = userDao.findAll(pageRequest);
    log.info("Pagina cargada");
    List<User> userList = userPage.getContent();
    log.info("USER_LIST:{}", userList);

    return userList;
}
```

Figura 10.56: Controlador Utilizado para Pagar los Registros

Teniendo el cliente y el servidor listos, es preciso realizar una prueba para apreciar el resultado final, el cual se puede ver en la Figura 10.57.

+ Añadir usuario

Lista de Usuarios					
Id	Usuario	Correo	Nombre	Apellido	Opciones
3	admin	dara323@gmail.com	admin	admin	 
1	default	info@topupcity.net	Default	User	 

«

«

1

»

»

Figura 10.57: Resultado Final de la Tabla de Usuarios

Una vez finalizado el listado de usuarios administradores, se procedió a la creación de un usuario, para esto se tuvo que definir una clase *UserForm* que tuviera los campos requeridos para poder añadir o editar un usuario, esta clase se puede ver en la Figura 10.58.

```

public class UserForm implements Serializable {

    private static final long serialVersionUID = 1L;

    @NotNull
    @Size(min = 4, max = 20)
    private String login;

    @NotNull
    @Size(min = 4, max = 100)
    private String password;

    @NotNull
    @Email(regexp="[a-z0-9!#$%&'*/=?^_`{|}~]+(?:\\. [a-z0-9!#$%&'*/=?^_`{|}~]+)*@([a-z0-9]([a-z0-9-]*[a-z0-9])?\\.)+[a-z0-9]([a-z0-9-]*[a-z0-9])?")
    private String email;

    @NotNull
    @Size(min = 1, max = 45)
    private String firstName;

    @NotNull
    @Size(min = 1, max = 45)
    private String lastName;
}

```

Figura 10.58: Formulario para Crear o Editar Usuarios

Al tener esta clase fue necesario maquetar el contenido de la vista para poder crear un usuario, nuevamente se utilizó la etiqueta *form* de Spring para crear un formulario y se definen los *input* para crear los campos que permitan llenar el objeto de la clase previamente creada. Para añadir registros se cuenta con dos controladores, ambos *NonAjax*, el primero simplemente para visualizar el formulario y el segundo que permite insertar un nuevo registro en el modelo. El diagrama de flujo de este controlador se puede apreciar en la Figura 10.59.

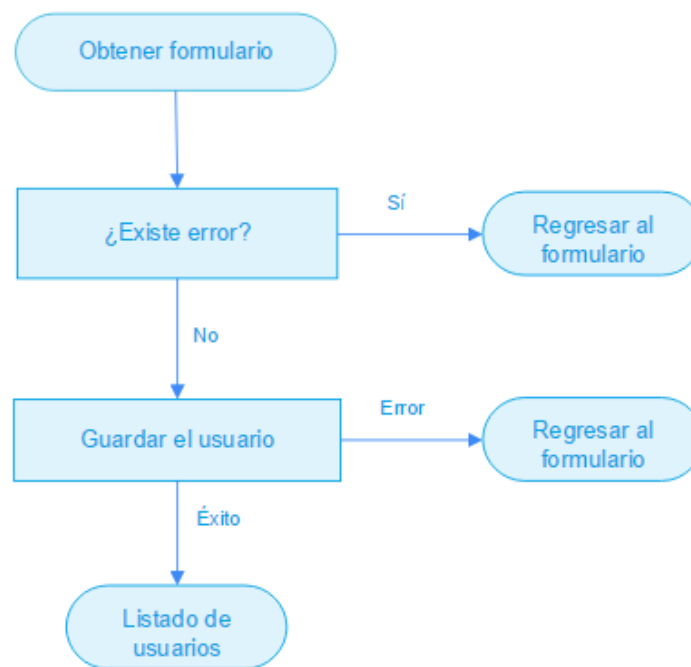


Figura 10.59: Diagrama de Flujo de Añadir Usuario

Teniendo listo el añadir usuario, se procedió a desarrollar la operación de edición y en este caso se definió una nueva vista con un nuevo formulario ya que hay campos que no pueden ser editables, sin embargo, se utilizó la misma clase *UserForm* para recibir los datos en el controlador. En este caso, no es posible editar campos sensibles como lo son la contraseña y el nombre de usuario, únicamente es posible editar el email, el nombre y el apellido, por lo que se tuvo que definir un *form* que contenga tres *input* para los campos antes mencionados. En el caso de la edición, también se cuenta con dos controladores *NonAjax*, el primero se encarga de buscar el usuario que se requiere editar y esto es posible ya que se envía el ID del usuario como un *PathVariable*, es decir se encuentra en la URL, al encontrar el usuario se llenan los campos del objeto *UserForm* que se encuentra enlazado al formulario, este controlador se puede apreciar en la Figura 10.60.

```
@RequestMapping(value = Paths.USER_EDIT)
public String editUser(Map<String, Object> model, UserForm userForm, @PathVariable String userID){
    User user = userDao.findById(Integer.parseInt(userID));
    if (user!=null){
        model.put("id", user.getId());
        userForm.setLogin(user.getLogin());
        userForm.setEmail(user.getEmail());
        userForm.setFirstName(user.getFirstName());
        userForm.setLastName(user.getLastName());
        return "userEdit";
    }else{
        long count = userDao.count();
        model.put("totalRecords", count);
        return "userList";
    }
}
```

Figura 10.60: Controlador Utilizado para Llenar el Formulario que Permite Editar un Usuario

El segundo controlador se encarga de recibir los datos modificados, validar los nuevos datos y en caso de que no exista ningún error, se guardan los cambios.

Para finalizar esta iteración, se explicará el desarrollo de la eliminación de un usuario. En este caso, para evitar eliminar algún registro por equivocación se implementó un modal para verificar que realmente se desea eliminar el recurso. Esto se realizó haciendo uso de una función llamada *puidialog* que pertenece a la librería *Prime UI*, esta función recibe un conjunto de pares clave-valor, sin embargo, únicamente es de interés el objeto llamado *buttons*, donde es posible definir los botones para aceptar o rechazar y las acciones que estos realizarán. La invocación a esta función se observa en la Figura 10.61.

```

$('#dlg').puidialog({
    showEffect: 'fade',
    hideEffect: 'fade',
    minimizable: true,
    maximizable: true,
    responsive: true,
    minWidth: 200,
    modal: true,
    buttons: [{
        text: '<spt:message code="lbl.user.del.yes" />',
        icon: 'fa-check',
        click: function() {
            AjaxMethodDelete(identifier);
            $('#dlg').puidialog('hide');
        }
    },
    {
        text: '<spt:message code="lbl.user.del.no" />',
        icon: 'fa-close',
        click: function() {
            $('#dlg').puidialog('hide');
        }
    }
    ]
});

```

Figura 10.61: Modal para Eliminar Registro

En caso de que la respuesta sea afirmativa, se invocará la función *AjaxMethodDelete* que realiza una petición asíncrona al controlador de eliminar un usuario y cualquiera que sea el resultado se mostrará un mensaje indicando la respuesta en la esquina superior derecha por un corto periodo de tiempo.

El controlador, para eliminar un usuario recibe un ID, busca el usuario y en caso de encontrarlo lo elimina y luego responde con un booleano indicando que se eliminó exitosamente, en caso de algún error se responde de forma negativa.

10.2.5 Desarrollo de la Administración de Cuentas

En esta iteración no fue necesario desarrollar las cuatro operaciones básicas, dado que solamente se contempló el listado de las cuentas registradas en la aplicación móvil. Se empezará con explicar el modelo asociado a las cuentas. Como se mencionó anteriormente, es posible establecer una correspondencia entre una tabla de la base de datos y una clase haciendo uso de las notaciones *@Entity* y *@Table* pero eso no es suficiente, se debe especificar que atributo de la clase es la clave primaria con la anotación *@Id* y además detallar con que campo de la tabla estará asociado cada atributo de la clase, esto se logra haciendo uso de la anotación *@Column*. Como se puede ver en la Figura 10.62, también es posible describir las relaciones de las tablas,

haciendo uso de la anotación `@OneToMany` en caso que exista una relación uno a muchos de la tabla actual con el atributo o también se podría usar `@ManyToOne` en caso de que exista una relación muchos a uno.

```

@Entity
@Table(name = "accounts")
public class Account implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @Column(name = "id")
    private Long id;
    @Column(name = "sourceidentity")
    private String sourceidentity;
    @Column(name = "fullname")
    private String fullname;
    @Column(name = "email")
    private String email;
    @Column(name = "cellphone")
    private String cellphone;
    @Column(name = "status")
    private String status;
    @Column(name = "password")
    private String password;
    @OneToMany(mappedBy = "accountsId")
    private Collection<DeviceSession> deviceSessionCollection;
    @OneToMany(cascade = CascadeType.ALL, mappedBy = "accountsId")
    private Collection<Transaction> transactionCollection;
    @OneToMany(cascade = CascadeType.ALL, mappedBy = "accountsId")
    private Collection<AccountLog> accountLogCollection;
    @JoinColumn(name = "default_countries_id", referencedColumnName = "id")
    @ManyToOne(optional = false)
    @JsonIgnore
    private Country defaultCountriesId;
}

```

Figura 10.62: Modelo de Cuentas

Es de aclarar que es posible definir la clave foránea con la que se establece la relación e incluso si la tabla posee una, para el primer caso se utilizó el atributo `mappedBy` al momento de establecer la relación, en caso contrario se utilizó la anotación `@JoinColumn`. En la Figura 10.62, se pueden observar los campos, así como las relaciones de la tabla, este modelo posee una colección de tres objetos donde el primero es el que se utilizó para almacenar el *sid* que es usado en las llamadas al API, el segundo es un objeto para ver los datos de una transacción y el último es un log de la cuenta.

Para continuar y tener el cliente listo bastó con utilizar la misma estructura HTML y el *script* antes creado para armar la tabla, del lado del servidor se requirió definir la interfaz

de *JpaRepository* y desarrollar nuevamente dos controladores al igual que el caso anterior, el primero de tipo *NonAjax* para contar la totalidad de los registros y acceder a la vista, mientras el segundo de tipo *Ajax* para poder paginar los registros de la tabla.

10.2.6 Desarrollo de la Administración de Países

Para esta quinta iteración se planificó el desarrollo de las cuatro operaciones básicas sobre los países, se explicarán en el orden antes mencionado, sin embargo, se tiene que explicar el modelo asociado a los países.

La tabla de los países consta de un ID como clave primaria, un nombre, sigue la especificación *ISO 3166-1* la cual define tres códigos para identificar un país, el primero es *ISO 3166-1 numeric* que consta de tres dígitos, el segundo es el *ISO 3166-1 alpha-2* contiene dos letras y el último, es el *ISO 3166-1 alpha-3* que posee tres letras para identificar al país, a su vez un país tiene asociado un conjunto de telefonías, prefijos y monedas, por lo que estas últimas son relaciones de uno a muchos, sin embargo, se omite en la Figura 10.63 por motivos prácticos.

```
@Entity
@Table(name = "countries")
public class Country implements Serializable {

    private static final long serialVersionUID = 1L;

    private Integer id;
    private String name;
    private String isonumcode;
    private String isoalpha2code;
    private String isoalpha3code;
    private Collection<Carrier> carrierCollection;
    private Collection<Prefix> prefixCollection;
    private Collection<Currency> currencyCollection;

}
```

Figura 10.63: Modelo de Países

Seguido a esto fue necesario definir la interfaz extendiendo de *JpaRepository* para poder acceder al modelo y poder trabajar en los controladores. Para comenzar el desarrollo del cliente y listar los países, se debe recordar que se utilizará la misma estructura HTML y el *script* desarrollado, teniendo en cuenta que se editaron las columnas a mostrar. Para finalizar esta primera operación básica, se cuenta con dos controladores, el primero para lograr acceder a la vista y el segundo que permite paginar el contenido de la tabla.

La siguiente operación a desarrollar fue *create* o añadir un país, para esto fue necesario definir una clase *CountryForm* con los campos requeridos para poder insertar un nuevo

registro como fueron el nombre y datos de la especificación *ISO 3166-1*. Esta clase se puede observar en la Figura 10.64.

```
public class CountryForm implements Serializable {

    private static final long serialVersionUID = 1L;

    private String name;
    private String isonumcode;
    private String isoalpha2code;
    private String isoalpha3code;

}
```

Figura 10.64: Formulario para Crear o Editar Países

Al tener esta clase se procedió a maquetar el contenido de la vista necesario para poder crear un país, nuevamente se utiliza la etiqueta *form* de Spring para crear un formulario y se definió los *input* de los cuatro campos antes especificados. Para añadir registros se tienen igualmente dos controladores, ambos *NonAjax*, el primero simplemente para visualizar el formulario y el segundo que permite insertar un nuevo registro en el modelo.

Teniendo listo el añadir país, se procedió a desarrollar la operación de edición utilizando la misma clase *CountryForm* para recibir los datos en el controlador. En el caso de la edición, también se tiene dos controladores *NonAjax*, el primero se encarga de buscar el país por el ID que se envía como un *PathVariable*, al encontrar el país se llenan los campos del objeto *CountryForm* que se encuentran enlazados al formulario, este controlador se puede apreciar en la Figura 10.65.

```
@RequestMapping(value = Paths.COUNTRY_EDIT)
public String editCountry(Map<String, Object> model, CountryForm countryForm, @PathVariable String countryID) {
    Country country = countrydao.findById(Integer.parseInt(countryID));
    if(country!=null){
        model.put("id", country.getId());
        countryForm.setName(country.getName());
        countryForm.setIsonumcode(country.getIsonumcode());
        countryForm.setIsoalpha2code(country.getIsoalpha2code());
        countryForm.setIsoalpha3code(country.getIsoalpha3code());

        return "countryEdit";
    }else{
        long count = countrydao.count();
        model.put("totalRecords", count);
        return "countriesList";
    }
}
```

Figura 10.65: Controlador Utilizado para Llenar el Formulario para Editar un País.

Para finalizar esta iteración, se explica el desarrollo de la eliminación de un país. Como en el caso de la administración de los usuarios y las siguientes iteraciones, para evitar eliminar algún registro por equivocación, se implementó un modal para verificar que realmente se desea eliminar el recurso, que contiene dos botones, uno para rechazar y otro para aceptar. En caso afirmativo se invocará una función llamada *AjaxMethodDelete*, la cual realiza una petición asíncrona al controlador y cualquiera que sea el resultado se mostrará un mensaje indicando la respuesta en la esquina superior derecha por un corto período de tiempo, esta función se aprecia en la Figura 10.66.

```
function AjaxMethodDelete(id) {
    var data = {"id": id};
    var uri = "/delete/country";
    $.ajax({
        type: "GET",
        url: uri,
        data: data,
        dataType: "json",
        success: function(response) {
            $('#dlg').puidialog('hide');
            if (response[0]){
                $('#messages').puigrowl('show', [{severity: 'info', summary: 'Country deleted. ID: ', detail: id}]);
            }else{
                $('#messages').puigrowl('show', [{severity: 'warn', summary: 'Country was not deleted. ID: ', detail: id}]);
            }
            window.location.replace("/countries");
        },
        error: function(err) {
            $('#dlg').puidialog('hide');
            $('#messages').puigrowl('show', [{severity: 'warn', summary: 'There was an error: ', detail: err}]);
            alert(JSON.stringify(err));
            window.location.replace("/countries");
        }
    });
}
```

Figura 10.66: Función *AjaxMethodDelete* para Eliminar Registros

El controlador para eliminar un país es de tipo *Ajax* y recibe un ID, busca el registro y en caso de encontrarlo lo elimina y luego responde con un booleano indicando que se eliminó exitosamente, en caso de algún error se responde de forma negativa.

10.2.7 Desarrollo de la Administración de Monedas

Para esta sexta iteración se planificó el desarrollo de las cuatro operaciones básicas sobre las monedas, se empezará por explicar el modelo asociado.

La tabla de las monedas consta de un ID como clave primaria, un nombre, sigue la especificación *ISO 4217* que define un código de tres letras para las divisas del mundo, a su vez una moneda tiene asociado un país y nótese que en la Figura 10.67, al momento de especificar el país con el cual está asociada la moneda, es necesario un objeto de tipo *Country* y no únicamente un ID.

```

@Entity
@Table(name = "currencies")
public class Currency implements Serializable {

    private static final long serialVersionUID = 1L;

    private Integer id;
    private String name;
    private String isocode;
    private Country countriesId;

}

```

Figura 10.67: Modelo de Monedas

Seguido a esto se definió la interfaz DAO extendiendo de *JpaRepository* para poder acceder al modelo y poder trabajar en los controladores. De igual forma se utilizó la misma estructura HTML y el *script* antes desarrollados, teniendo en cuenta que se cambiaron las columnas a mostrar. Para finalizar esta primera operación básica se tuvo que desarrollar dos controladores, el primero para lograr acceder a la vista y el segundo que permite paginar el contenido de la tabla.

La siguiente operación a desarrollar fue *create*, para esto se definió una clase *CurrencyForm* con los campos requeridos para poder insertar una nueva moneda, que fueron; el nombre, el código *ISO 4217* y el nombre del país. Esta clase se puede observar en la Figura 10.68.

```

public class CurrencyForm implements Serializable{

    private static final long serialVersionUID = 1L;

    private String name;
    private String isocode;
    private String country;

}

```

Figura 10.68: Formulario para Crear o Editar Monedas

Al tener esta clase se maquetó el contenido de la vista necesario para poder crear una moneda, nuevamente se utilizó la etiqueta *form* de Spring para crear un formulario y se definieron los *input* de los tres campos antes especificados, en este caso para poder elegir el país asociado se definió una etiqueta *select* para evitar errores humanos y asegurar que se indicara un país existente en el sistema. Para llenar la lista del select se utilizó una variable que se obtiene del primer controlador que se explicará a continuación, la estructura HTML utilizada para el *select* se observa en la Figura 10.69.

```

<form:select class="form-control" path="country">
  <form:option class="form-control" value="NONE" disabled="true"><spt:message code="lbl.currency.placeholder.country" /></form:option>
  <form:options class="form-control" items="{listOfCountries}"></form:options>
</form:select>

```

Figura 10.69: Etiqueta *Select* de Spring para Seleccionar un País

Para añadir monedas se desarrollaron igualmente dos controladores, ambos *NonAjax*, el primero se encarga de almacenar los nombres de los países en una lista para enviarla a la vista del formulario para crear un nuevo registro y el segundo que permite insertar un nuevo registro en el modelo. El primer controlador se puede observar en la Figura 10.70.

```

@RequestMapping(value = Paths.CURRENCY_ADD_FORM)
public String viewAddCurrency(Map<String, Object> model, CurrencyForm currencyForm) {

    log.info("Entrando al currency add form...");
    List<Country> countries = countrydao.findAll();
    List<String> listOfCountries = new ArrayList<>();
    for(int i=0; i<countries.size(); i++)
        listOfCountries.add(countries.get(i).getName());

    model.put("listOfCountries", listOfCountries);
    return "addCurrency";
}

```

Figura 10.70: Controlador para Acceder al Formulario de Añadir Moneda

Teniendo listo el añadir moneda, se procedió a la edición utilizando de la misma clase *CurrencyForm* para recibir los datos en el controlador. En el caso de la edición, también se contó con dos controladores *NonAjax*, el primero se encarga de buscar la moneda por el ID que se envía como un *PathVariable* y almacenar en una lista de los nombres de los países, al encontrar el registro se llenan los campos del objeto *CurrencyForm* que se encuentra enlazado al formulario.

El segundo controlador se encarga de recibir los datos modificados, validar los nuevos datos y en caso de que no exista ningún error, se guardan los cambios.

Para finalizar esta iteración, la eliminación de una moneda se implementó mediante un modal para verificar que realmente se desea eliminar el recurso, que consta con dos botones uno para rechazar y otro para aceptar, en caso afirmativo se invocará una función llamada *AjaxMethodDelete*, que realizará una petición asíncrona al controlador y cualquiera que sea el resultado se mostrará un mensaje indicando la respuesta.

El controlador para eliminar una moneda es de tipo *Ajax* y recibe un ID, busca el registro y en caso de encontrarlo lo elimina y luego responde con un booleano indicando que se eliminó exitosamente, en caso de algún error se responde de forma negativa.

10.2.8 Desarrollo de la Administración de Telefonías

Para esta séptima iteración se planificó el desarrollo de las cuatro operaciones básicas sobre las telefonías, se empezará por explicar el modelo asociado.

En la Figura 10.71, se aprecia la tabla de las telefonías que consta de un ID como clave primaria, un nombre, seguido de un código *E.212*, definido por la ITU por sus siglas en inglés “*International Telecommunication Union*”, que logra definir un código para el país denominado MCC por sus siglas en inglés “*Mobile Country Code*” de tres dígitos y un código para red denominado MNC “*Mobile Network Code*” que puede variar entre dos y tres dígitos, a su vez una telefonía tiene un estatus que indica si está habilitada o no, posee asociado un país y un conjunto de productos.

```
@Entity
@Table(name = "carriers")
public class Carrier implements Serializable {

    private static final long serialVersionUID = 1L;

    private Integer id;
    private String code;
    private String name;
    private String status;
    private Country countriesId;
    private Collection<CarrierProduct> carrierProductCollection;

}
```

Figura 10.71: Modelo de Telefonías

Seguido a esto fue necesario definir la interfaz DAO extendiendo de *JpaRepository* para poder acceder al modelo y poder trabajar en nuestros controladores. En esta primera operación básica se desarrollaron dos controladores, el primero para lograr acceder a la vista y el segundo que permite paginar el contenido de la tabla.

La siguiente operación desarrollada fue *create*, para esto fue necesario definir una clase *CarrierForm* con los campos requeridos para poder insertar una nueva telefonía: el nombre, código *E.212*, nombre del país y el estatus, esta clase se puede observar en la Figura 10.72.

Al tener esta clase fue necesario maquetar el contenido de la vista para poder crear una telefonía, nuevamente se creó un formulario y se definieron los *input* de los cuatro campos antes especificados, nuevamente para poder elegir el país asociado se definió una etiqueta *select*, al igual que para definir si la telefonía estará habilitada o no.

```

public class CarrierForm implements Serializable {

    private static final long serialVersionUID = 1L;

    private String code;
    private String name;
    private String status;
    private String country;

}

```

Figura 10.72: Formulario para Crear o Editar Telefonías

Para añadir telefonías se tienen igualmente dos controladores, ambos *NonAjax*, el primero se encarga de almacenar los nombres de los países en una lista para enviarla a la vista del formulario para crear un nuevo registro y el segundo que permite insertar un nuevo registro en el modelo.

Teniendo listo el añadir moneda, se procedió a la edición utilizando la misma clase *CarrierForm* para recibir los datos en el controlador. En el caso de la edición, se tienen dos controladores *NonAjax*, el primero se encarga de buscar la telefonía por el ID que se envía como un *PathVariable* y almacenar en una lista de los nombres de los países, al encontrar el registro se llenan los campos del objeto *UserForm* que se encuentra enlazado al formulario.

El segundo controlador se encarga de recibir los datos modificados, validar los nuevos datos y en caso de que no exista ningún error, se guardan los cambios.

Para finalizar esta iteración, la eliminación de una telefonía se implementó mediante un modal para verificar que realmente se desea eliminar el recurso, que consta de dos botones, uno para rechazar y otro para aceptar, en caso afirmativo se invocará una función llamada *AjaxMethodDelete* la cual realizará una petición asíncrona al controlador, este se encarga de busca el registro y en caso de encontrarlo lo elimina y luego responde con un booleano indicando que se eliminó exitosamente, en caso de algún error se responde de forma negativa.

10.2.9 Desarrollo de la Administración de Prefijos

Para esta octava iteración se planificó el desarrollo de las cuatro operaciones básicas sobre los prefijos, se iniciará por explicar el modelo asociado.

En la Figura 10.73, se puede observar la tabla de los prefijos que consiste de un ID como clave primaria, sigue el prefijo que es el código internacional para marcar a un país, a su vez un prefijo tiene asociado un país.

```

@Entity
@Table(name = "prefixes")
public class Prefix implements Serializable {

    private static final long serialVersionUID = 1L;

    private Integer id;
    private String prefix;
    private Country countriesId;

}

```

Figura 10.73: Modelo de Prefijos

Seguido a esto se definió la interfaz DAO para poder acceder al modelo y poder trabajar en los controladores. Para comenzar a listar los prefijos se utilizó la misma estructura HTML y el *script* antes desarrollado. Para finalizar esta primera operación básica se desarrollaron dos controladores, el primero para lograr acceder a la vista y el segundo que permite paginar el contenido de la tabla.

La siguiente operación a desarrollar fue *create*, para esto fue necesario definir una clase *PrefixForm* con los campos requeridos para poder insertar un nuevo prefijo, que eran el prefijo y el país, esta clase se puede observar en la Figura 10.74.

```

public class PrefixForm implements Serializable{

    private static final long serialVersionUID = 1L;

    private String prefix;
    private String country;

}

```

Figura 10.74: Formulario para Crear o Editar Prefijos

Al tener esta clase fue necesario maquetar el contenido de la vista para poder crear un prefijo, nuevamente se utilizó la etiqueta *form* de Spring para crear un formulario y se definieron los *input* de los dos campos antes especificados, nuevamente para poder elegir el país asociado se definió una etiqueta *select*.

Para añadir prefijos se cuenta con dos controladores, ambos *NonAjax*, el primero se encarga de almacenar los nombres de los países en una lista para enviarla a la vista del formulario para crear un nuevo registro y el segundo que permite insertar un nuevo registro en el modelo.

Teniendo listo el añadir prefijo, se procedió a la edición utilizando la misma clase *PrefixForm* para recibir los datos en el controlador. En el caso de la edición, también cuenta con dos controladores *NonAjax*, el primero se encarga de buscar el prefijo por el

ID que se envía como un *PathVariable* y almacenar en una lista de los nombres de los países, al encontrar el registro se llenan los campos del objeto *PrefixForm* que se encuentra enlazado al formulario.

El segundo controlador se encarga de recibir los datos modificados, validar los nuevos datos y en caso de que no exista ningún error, se guardan los cambios.

Para finalizar esta iteración, la eliminación de un prefijo se implementó mediante un modal para verificar que realmente se desea eliminar el recurso, así como en las iteraciones previas.

El controlador para eliminar un prefijo es de tipo *Ajax* y recibe un ID, busca el registro y en caso de encontrarlo lo elimina y luego responde con un booleano indicando que se eliminó exitosamente, en caso de algún error se responde de forma negativa.

10.2.10 Desarrollo de la Administración de Productos

Para esta novena iteración se planificó el desarrollo de las cuatro operaciones básicas sobre los productos, se dará inicio por explicar el modelo asociado.

La tabla de los productos consta de un ID como clave primaria, un código, un nombre, un monto, un impuesto y a su vez está asociado con una telefonía; estos mencionados son de especial importancia, sin embargo, en la Figura 10.75 se aprecia el modelo de los productos.

Seguido a esto se definió la interfaz DAO para poder acceder al modelo y poder trabajar en nuestros controladores. Para comenzar a listar los productos se hizo uso de la misma estructura HTML y el *script* antes desarrollado.

```
@Entity
@Table(name = "carrier_products")
public class CarrierProduct implements Serializable {

    private static final long serialVersionUID = 1L;

    private Integer id;
    private String code;
    private String name;
    private String shortname;
    private BigDecimal amount;
    private BigDecimal fee;
    private Carrier carriersId;
    private Collection<ProductMapping> productMappingCollection;

}
```

Figura 10.75: Modelo de Productos

En esta iteración se decidió que la operación de crear se iba a agregar a la tabla de telefonías para evitar confusiones, así como se también incluir un botón para poder listar los productos de una telefonía específica. Se dará inicio por explicar el listado de productos específicos, para esto se desarrolló un controlador que recibe un ID de alguna telefonía, luego busca todos los productos del sistema y evalúa si pertenece a la telefonía indicada, en tal caso, se almacena en una lista que finalmente es enviada a la vista. La operación de agregar se realizó haciendo uso de un formulario llamado *CarrierProductForm* que cuenta con los atributos necesarios para poder agregar un registro nuevo, se aprecia mejor en la Figura 10.76. Para poder terminar esta operación se crearon dos controladores, el primero para lograr acceder a la vista que contiene el formulario y el segundo que se encarga de recibir los datos del formulario y el ID de la telefonía, validar los datos y en caso de éxito añadir el producto en el modelo.

```
public class CarrierProductForm implements Serializable{
    private String code;
    private String name;
    private BigDecimal amount;
    private BigDecimal fee;
}
```

Figura 10.76: Formulario para Crear o Editar Productos

Teniendo listo el añadir producto, se procedió a realizar la edición utilizando la misma clase *CarrierProductForm* para recibir los datos en el controlador. En el caso de la edición, se cuenta con dos controladores *NonAjax*, el primero se encarga de buscar el producto por el ID que se envía como un *PathVariable* y almacenar el resultado en los campos del objeto *CarrierProductForm* que se encuentra enlazado al formulario.

El segundo controlador se encarga de recibir los datos modificados, validar los nuevos datos y en caso de que no exista ningún error, se guardan los cambios.

Para finalizar esta iteración, la eliminación de un producto se implementó mediante un modal para verificar que realmente se desea eliminar el recurso, que contiene dos botones uno para rechazar y otro para aceptar, en caso afirmativo se invocará una función llamada *AjaxMethodDelete*, en la que realizará una petición asíncrona al controlador y cualquiera que sea el resultado se mostrará un mensaje indicando la respuesta.

10.2.11 Desarrollo de la Administración de Transacciones

En esta iteración no fue necesario desarrollar las cuatro operaciones básicas, dado que solamente se contempló el listado de las transacciones. Se iniciará con explicar el modelo asociado estas. En la Figura 10.77, se pueden apreciar los campos que posee esta tabla, por motivos prácticos se removieron las anotaciones y los métodos de acceso. Una transacción almacenará datos sensibles e importantes como son el monto, la

moneda, la cuenta y el teléfono al cual se realizó la recarga, el impuesto, el proveedor, entre algunos otros.

```

@Entity
@Table(name = "transactions")
public class Transaction implements Serializable {

    private static final long serialVersionUID = 1L;
    private Long id;
    private BigDecimal amount;
    private String cellphone;
    private BigDecimal fee;
    private BigDecimal destinationamount;
    private String currency;
    private String destinationcurrency;
    private BigDecimal exchangerate;
    private Date datetime;
    private String paymentauthorization;
    private Date paymentdate;
    private BigDecimal paymentfee;
    private String paymentinfo;
    private Account accountsId;
    private CarrierProduct carrierProductsId;
    private Carrier carriersId;
    private Country countriesId;
    private ProviderProduct providerProductsId;
    private Provider providersId;
    private ResultCode resultcodesId;

}

```

Figura 10.77: Modelo de Transacciones

Para continuar y tener el cliente listo, bastó con utilizar la misma estructura HTML y el *script* antes creado para armar la tabla, sin embargo, este requería de un cambio en el código debido a que se poseen demasiados campos en la respuesta y para aumentar la legibilidad de la tabla se concatenaron los montos con sus respectivas monedas. Del lado del servidor, se requirió definir la interfaz de *JpaRepository* para acceder al modelo y desarrollar nuevamente dos controladores, el primero de tipo *NonAjax* para contar la cantidad de transacciones y lograr acceder a la vista, mientras el segundo de tipo *Ajax* para poder paginar los registros de la tabla. En la Figura 10.78, se muestra el resultado final.

Lista de Transacciones						
Id ↕	Fecha ↕	Emisor	Telefono ↕	Impuesto	Monto	Monto Destino
9091035135837625000	Wed Feb 12 2014 18:29:49 GMT-0400 (Hora estándar de Venezuela)	Cesar Rodriguez	571234666666	1.5	20 USD	160 MXN

Figura 10.78: Resultado Final de la Tabla de Transacciones

10.3 Resultado de Pruebas

La aplicación móvil fue probada como se mencionó anteriormente en un dispositivo Motorola Moto G 2013 corriendo la versión 5.1.1 de Android, además se utilizaron dispositivos virtuales emulados con el fin de probar la aplicación en pantallas de varias resoluciones.

El módulo de autenticación social de la aplicación móvil se probó con la aplicación de Facebook instalada y posteriormente sin dicha aplicación instalada, ambas pruebas fueron satisfactorias logrando autenticar al usuario. Por su parte, la autenticación con Google se probó satisfactoriamente en dispositivos con los “*Google Play Services*” instalados, en dispositivos sin estos servicios instalados no se pudo realizar la autenticación pero se le indica al usuario el error y se invita a instalarlo.

La integración con la pasarela de pago móvil de PayPal se probó haciendo uso del ambiente “sin internet” el cual genera datos simulados, porque siempre retorna a la aplicación con una transacción exitosa. Por esto se realizó una segunda prueba configurando el ambiente del SDK de PayPal como “sandbox” el cual usa cuentas y tarjetas de crédito de pruebas, con lo que se pudo probar satisfactoriamente todo los escenarios incluyendo los de error por falta de fondos o datos erróneos.

En principio, el módulo de conexión con el API privado de TopUp City se desarrolló usando la clase `HttpRequest` que proveía Android pero el código final tenía no era eficiente ni legible, por lo que se migró a la librería “*Retrofit 2*” descrita en el capítulo anterior que a su vez usa la librería `OkHttp` para realizar la petición. `OkHttp` se configura con el nivel de registro que se desea utilizar para imprimir por consola todas las peticiones realizadas y respuestas obtenidas por la aplicación, lo permitió probar satisfactoriamente la integración el API de TopUp City.

La primera prueba realizada en la consola web, fue probar el inicio de sesión con datos válidos y al asegurar su correcto funcionamiento se probaron datos erróneos para confirmar que los mensajes de error se muestran de manera adecuada.

Al finalizar el inicio de sesión, se procedió a confirmar que la segunda iteración funcionaba de manera correcta, para esto se eliminaron registros de forma manual en la base de datos y se confirmó que los indicadores de la pantalla inicial realmente reflejaban los cambios.

A partir de la tercera iteración hasta finalizar el desarrollo, se comprobaron las operaciones CRUD desarrolladas de la siguiente manera. Se procedió a crear un nuevo registro haciendo uso de los formularios desarrollados en la consola web, empezando por utilizar datos inválidos para asegurar que las validaciones funcionaran de manera correcta. Al agregar el registro exitosamente en el sistema, se verificaba su existencia en las tablas ubicadas en las páginas principales de cada recurso administrable. Luego de

verificar que la lectura del modelo funcionaba de manera adecuada, se procedió a verificar que la edición funcionaba de la forma esperada. Para esto, se procedió a editar el registro recientemente agregado y se verificó que los datos se cargaban de manera exitosa en la vista, para luego realizar una modificación y guardar los cambios. Nuevamente, se verificó la vista principal de los recursos para asegurar que se reflejaban los cambios recientes. Para finalizar, la última operación a probar fue la eliminación, para esto bastó con tratar de eliminar el registro y recargar la página principal que confirmar que no se mostraba en la tabla. Este proceso descrito se realizó en cada una de las iteraciones para asegurar que los cambios producidos al desarrollar un nuevo módulo no afectaron a los anteriormente desarrollados.

Las últimas pruebas realizadas fue verificar que los cambios realizados a nivel de la consola web se reflejaban en la aplicación móvil, así como asegurar que los registros producidos en el cliente android se mostraran de manera satisfactoria en la aplicación web. Para esto, se registró un usuario haciendo uso de los servicios provistos por el API de TopUp City LLC y se verificó su presencia en la tabla de usuarios, luego se realizó una transacción con este y se comprobó la existencia en la tabla de transacciones. En este punto surgió un problema, el listado estaba siendo ordenado por el id de las transacciones, por lo cual no se reflejó al inicio de tabla. Esto fue solucionado modificando el campo de ordenamiento, estableciendo el campo *timestamp* como atributo de ordenación y listando de mayor a menor.

Luego se procedió a modificar los productos que se ofrecen en la aplicación móvil, se editaron tanto el impuesto como el monto origen del producto. Se procedió a realizar una transacción con este producto recientemente editado y verificar que tanto el historial del cliente android como la tabla de transacciones de la consola web reflejaran en el detalle los cambios realizados. Esto funcionó de manera correcta.

11. Conclusiones

Una aplicación móvil es un software que es ejecutado desde un dispositivo móvil que hoy en día se han convertido en una necesidad, ya que poseen una gran cantidad de herramientas que simplifican las tareas cotidianas como, comunicación, productividad y entretenimiento.

Se han establecido estándares para el desarrollo de aplicaciones móviles que van desde guías de diseño, como debería ser el pase de mensajes entre el servidor y el cliente, e incluso hasta como debería ser la comunicación entre pares. Teniendo como norte aumentar la interoperabilidad y la interconexión de aplicaciones.

En el presente trabajo se desarrolló con éxito el *MVP* de una aplicación móvil nativa para sistemas operativos Android, con el fin de integrarse con el API privado de TopUp City LLC para realizar las recargas de saldo a teléfonos celulares de operadoras telefónicas de México y Colombia; contando con funcionalidades como recuperar el historial de recargas de cada usuario, cobro mediante el uso de PayPal como pasarela de pago móvil y la autenticación social mediante los SDKs de Facebook y Google. Además, cumple con las guías de estilo “*Material Design*” de Google enriqueciendo las interfaces gráficas y la experiencia de usuario.

Cualquier procesamiento de pago mediante tarjetas de crédito involucra un riesgo inherente a este tipo de transacción, ya que son muchos los fraudes que se cometen a diario. Por esto se tuvo que, deshabilitar el procesamiento directo de tarjetas de crédito a través del SDK de PayPal, dejando solo la posibilidad de realizar el pago haciendo uso de la cuenta de PayPal de usuario. Esto representa una clara limitación de la aplicación que podría ser solucionada usando *Braintree Payments*, un servicio de PayPal que tiene una herramienta de detección de fraudes, la cual no se pudo integrar con la aplicación ya que según las políticas de uso el modelo de negocio es de alto riesgo.

A su vez, se desarrolló exitosamente la Consola Web, encargada de administrar los recursos utilizados por la aplicación móvil, siguiendo la arquitectura MVC. Las vistas de la aplicación web fueron desarrolladas con ciertas herramientas para poder facilitar y agilizar el trabajo a realizar, entre los cuales se pueden destacar a Bootstrap que permitió obtener vistas completamente adaptativas, Apache Tiles que facilitó la creación de plantillas con el fin de modularizar el contenido de las vistas y, por último, Prime UI que proporcionó estructuras HTML y JavaScript para realizar la integración de las vistas con los controladores. Estos últimos se desarrollaron haciendo uso de Spring Boot y JPA para acceder al modelo. Este último de gran importancia, ya que facilitó realizar las consultas sin la necesidad de escribir código PL/SQL además de proveer un control de transacciones.

A pesar de los problemas que surgieron durante el desarrollo de ambas aplicaciones y teniendo en cuenta los resultados finales, se puede concluir que se lograron los objetivos específicos propuestos, obteniendo así una aplicación móvil que se conecta a un API privado para realizar recargas de saldo y una aplicación web que permite gestionar los recursos del sistema. Se deja abierta la posibilidad de desarrollar un módulo de seguridad basado en *Feedzai* para utilizar técnicas de Aprendizaje Automático o “Machine Learning” para prevenir fraudes en transacciones que involucren tarjetas de crédito.

Referencias

- [1] Ericsson, "Ericsson Mobility Report - Latin America and The Caribbean", Noviembre 2016.
- [2] Pew Research Center, *Statistical Portrait of Hispanics in the United States*. The Pew Charitable Trusts in: <http://www.pewhispanic.org/2016/04/19/statistical-portrait-of-hispanics-in-the-united-states>
- [3] F. Muciño, ¿*Por qué EE.UU. Debe "Seducir" a los Hispanos?*, Forbes México in: <http://www.forbes.com.mx/por-que-eu-debe-seducir-a-los-hispanos>
- [4] L.M. Sergio, "Qué es una Aplicación Web", in *Programación en Internet: Clientes Web*, Alicante, Editorial Club Universitario, 2001.
- [5] L.M. Sergio, "Arquitecturas Clientes/Servidor", in *Programación en Internet: clientes web*, Alicante: Editorial Club Universitario, 2001.
- [6] B.M. Márquez Avedaño, J.M. Zulaica Rugarcía, in *Implementación de un Reconocedor de Voz Gratuito a el Sistema de Ayuda a Invidentes Dos-Vox en Español*, Escuela de Ingeniería, Universidad de las Américas Puebla, Tesis Licenciatura, 2004.
- [7] G. Alexis, L. Louis, W. Estelle, "Introducing HTML5 and CSS3 for The Real Word", 2nd Edition, Editorial SitePoint, 2015.
- [8] S. Abraham, K. Henry, S. Surdashaan, "Capitulo 1. Introducción", in *Fundamentos de bases de datos, 4ta Edición*, Editorial Mc Graw Hill, 2002.
- [9] O. Belmonte, *I. Introducción al Lenguaje de Programación Java: Una Guía Básica*, Universitat Jaume I, Castelló, España. Revisión 2.0, 2005.
- [10] V. Daniel, *Introducción a las Aplicaciones Web con Java*. ESI-UCLM in: <http://www.inf-cr.uclm.es/www/mpolo/asig/0708/tutorJavaWeb.pdf>
- [11] D.L. La Red, "Dispositivos Móviles", Trabajo adscripción, Facultad de Ciencias Exactas, Naturales y Agrimensura, Universidad Nacional de Nordeste, Corrientes, Argentina, 2007.
- [12] J.D. Morrillo, "Introducción a los Dispositivos Móviles", Universidad Abierta de Cataluña, Cataluña, España, 2011.
- [13] J. Cuello, J. Vittone, "Capítulo 1. Las Aplicaciones", *Diseñando Apps para Móviles*, 2013.
- [14] IBM Corporation, *El Desarrollo de Aplicaciones Móviles Nativas, Web o Híbridas*, Thought Leadership White Paper, 2012.
- [15] J. Bernardo, P. Matos, T. Pimpao, *Brief Look to Android History*, http://eraizes.ipsantarem.pt/moodle/pluginfile.php/25180/mod_resource/content/7/Brief-look-to-Android-history.pdf
- [16] D. Herger, *Mobile Devices - An Introduction to the Android Operating Environment*, DHTechnologies in: <http://www.dhtusa.com/media/AndroidInternals.pdf>

- [17] Activities, Google in: <https://developer.android.com/guide/components/activities.html>
- [18] Starting an Activity, Google in: <https://developer.android.com/training/basics/activity-lifecycle/starting.html>
- [19] Services, Google in: <https://developer.android.com/guide/components/services.html>
- [20] Intents and Intent Filters, Google in: <https://developer.android.com/guide/components/intents-filters.html>
- [21] Boston Retail Partners “2015 E-Commerce Survey”, Septiembre 2015.
- [22] D. Fonseca, W. Pérez, M. Morilla, “Pasarela de Pagos para la Seguridad de Transacciones Bancarias en Línea”, *3C Empresa*, Nro. 15, pp. 62 – 86, Agosto 2013.
- [23] F. Jhonatan, in *Desarrollo de la aplicación para la gestión de recargas en línea a teléfonos celulares para la empresa Technology & Solutions LDTA*, Escuela de Ingeniería de Sistemas e Informática, Universidad Industrial de Santander, Tesis Licenciatura, 2009.
- [24] Y. Amaya, “Metodologías Ágiles en el Desarrollo de Aplicaciones para Dispositivos Móviles. Estado Actual”, *Revista de Tecnología*, Nro. 12, Vol. 2, pp. 118 – 126, 2013.