ESCUELA POLITÉCNICA NACIONAL

FACULTAD DE INGENIERÍA ELÉCTRICA Y ELECTRÓNICA

DESARROLLO DE UN PROTOTIPO DE PUNTO DE VENTA (POS) MÓVIL PARA PYMES QUE COMERCIALIZAN ROPA Y CALZADO

TRABAJO DE TITULACIÓN PREVIO A LA OBTENCIÓN DEL TÍTULO DE INGENIERO EN ELETRÓNICA Y REDES DE INFORMACIÓN

KEVIN CRISTHOPER CALVACHE QUIMBIULCO

kevin.calvachi@epn.edu.ec

DIRECTOR: M.Sc. RAÚL DAVID MEJÍA NAVARRETE

david.mejia@epn.edu.ec

Quito, febrero 2019

AVAL

Certifico que el presente trabajo fue desarrollado por Kevin Cristhoper Calvache Quimbiulco, bajo mi supervisión.

ING. RAÚL DAVID MEJÍA NAVARRETE, MSc.
DIRECTOR DEL TRABAJO DE TITULACIÓN

DECLARACIÓN DE AUTORÍA

Yo, Kevin Cristhoper Calvache Quimbiulco, declaro bajo juramento que el trabajo aquí descrito es de mi autoría; que no ha sido previamente presentado para ningún grado o calificación profesional; y, que he consultado las referencias bibliográficas que se incluyen en este documento.

A través de la presente declaración dejo constancia de que la Escuela Politécnica Nacional podrá hacer uso del presente trabajo según los términos estipulados en la Ley, Reglamentos y Normas vigentes.

KEVIN CRISTHOPER CALVACHE QUIMBIULCO

ÍNDICE DE CONTENIDO

RESUMEN

En el presente Proyecto Técnico se realizará el desarrollo de un prototipo de punto de venta (POS) móvil para pequeñas y medianas empresa empresas (PyMES) que comercializan ropa y calzado, con lo cual se busca agilizar el proceso de venta y mantener un control automatizado de los productos. El prototipo está compuesto de una base de datos, una web API (*Application Programming Interface*) y un cliente Android.

Este documento presenta una visión general del trabajo realizado y está conformado por 4 capítulos:

El primer capítulo contiene la información teórica necesaria para este Proyecto Técnico, se explican los fundamentos de bases de datos relacionales, *framework* web API, arquitectura de servicios REST (*Representational State Transfer*), desarrollo de aplicaciones móviles y la metodología ágil Scrum.

En el segundo capítulo y con base en la metodología propuesta, se hace un análisis de las aplicaciones existentes que junto con entrevistas permiten definir los requerimientos de usuario, se planifican las tareas a realizar por cada *sprint* y finalmente se escribe el código de cada tarea.

En el tercer capítulo se presentan los resultados obtenidos, así como las propuestas realizadas a raíz de una encuesta de satisfacción.

El cuarto capítulo contiene las conclusiones y recomendaciones recogidas a lo largo del desarrollo de este Proyecto Técnico.

En los anexos se incluyen la entrevista para el levantamiento de información, el *script* para la creación de la base de datos, la encuesta de validación, el código de la web API, código del cliente Android, el manual de usuario del prototipo y el manual de instalación del mismo.

PALABRAS CLAVE: sistema POS, web API, REST, JSON, cliente Android.

ABSTRACT

The current Technical Project consists of the development of a mobile prototype point of sale (POS) for small and medium enterprises (SMEs) that market clothing and footwear, which seeks to streamline the sales process and maintain an automated control of the products. The prototype is composed of a database, a web API (Application Programming Interface) and an Android client.

This document presents an overview of the work carried out and consists of 4 chapters:

The first chapter contains the theoretical information necessary for this Technical Project, basics of relational databases, web API framework, REST services architecture (Representational State Transfer), mobile application development and the Scrum agile methodology are explained.

In the second chapter, an analysis is made of the existing applications that together with surveys allow to define the user requirements, the tasks to be performed are planned for each sprint and finally the code of each task is written.

In the third chapter the results obtained are presented, as well as the proposed changes as a result of a satisfaction survey.

The fourth chapter contains the conclusions and recommendations collected throughout the development of this Technical Project.

In the appendix, the interview for the gathering of information, the script for the creation of the database, the UML (Unified Modeling Language) class diagram for server side, the validation survey, the prototype code, the prototype user manual and the prototype installation manual are presented.

KEYWORDS: POS system, web API, REST, JSON, Android client

1. INTRODUCCIÓN

Los puntos de venta (POS) basan su funcionamiento en el apoyo a procesos de venta, facturación y control de inventario en locales comerciales. A pesar de la amplia oferta de sistemas móviles orientados a este fin, estas aplicaciones poseen tres características que las vuelven poco accesibles a emprendimientos, así como pequeñas y medianas empresas (PyMES).

Por un lado, estas soluciones intentan satisfacer los requerimientos generales de la mayor parte de comercios dejando de lado requisitos específicos que son de gran peso en el giro de negocio del usuario final. Luego y como factor común las funciones de facturación están disponibles previo pago de membresías relativamente costosas. Por último, estas aplicaciones hacen uso de componentes de hardware extra como impresoras y lectores de código de barras. Todos estos factores mencionados hacen que por un lado las aplicaciones no se puedan adaptar por completo a una actividad comercial en específico y por otro lado que sean poco accesibles a las PyMES por las altas inversiones que el despliegue de estos sistemas supone [1].

El objetivo de este Proyecto Técnico es desarrollar un prototipo POS enfocado en negocios de ropa y calzado con el fin de atender las necesidades de este segmento de mercado. Con la finalidad de reducir costos de inversión, por un lado, este prototipo hace uso de la cámara del dispositivo móvil en reemplazo de lectores de código de barra y, por otro lado, integra una impresora térmica genérica de bajo costo para generar *tickets* de ventas al finalizar una venta.

El prototipo de punto de venta ha sido desarrollado bajo un ambiente distribuido con el uso de servicios web lo que permite una fácil implementación de nuevas funcionalidades y la flexibilidad de trabajar entre varios empleados de un mismo negocio simultáneamente. Finalmente se destaca el hecho de que el prototipo está destinado para dispositivos móviles con sistema operativo Android pues es el sistema operativo con mayor aceptación en el país según datos del INEC [2].

1.1 OBJETIVOS

El objetivo general de este Proyecto Técnico es: desarrollar un prototipo de punto de venta distribuido para dispositivos móviles Android enfocado en PyMES que comercializan ropa y calzado.

Mientras que los objetivos específicos son:

1. Analizar las herramientas tecnológicas necesarias para el desarrollo del prototipo de punto de venta.
2. Diseñar los componentes de *backend* y *frontend* necesarios para el desarrollo del prototipo: base de datos, web API[1] con arquitectura REST[2] y aplicación Android.
3. Implementar los componentes del prototipo de punto de venta con base en el diseño realizado.
4. Analizar los resultados obtenidos de las pruebas ejecutadas en el prototipo de punto de venta.

1.2 ALCANCE

El prototipo de punto de venta funcionará en un ambiente distribuido como se muestra en la Figura 1.1, lo que dará la flexibilidad de manipular la información de varios negocios, así como trabajar entre varios empleados en un mismo negocio simultáneamente. Esto supone el desarrollo de un *backend* y un *frontend* para la aplicación. El *backend* será desarrollado con la ayuda del *framework* web API y bases de datos relacionales SQL, además se usará como objeto de asociación la tecnología Entity Framework. El *backend* será responsable de procesar los datos recogidos del lado del usuario y almacenarlos en la base de datos. Por otro lado, el *frontend* se implementará con el uso de tecnologías nativas de Android.

El prototipo contará con dos perfiles de usuario, uno para el administrador y otro de vendedor, el perfil de administrador permitirá gestionar el inventario de la tienda comercial, así como realizar la venta de productos, el perfil de vendedor solo permitirá la realización de ventas. Con el uso de la cámara, el prototipo será capaz de leer el código de barras de

---

[1] API (*Application Programming Interface*): es un conjunto de reglas que las aplicaciones siguen para comunicarse entre ellas.

[2] REST (*Representational State Transfer*): es un estilo arquitectónico para proveer reglas de comunicación entre sistemas basados en la web, se caracteriza por trabajar en un ambiente cliente-servidor en donde el servidor no mantiene el estado de las peticiones.

los productos que el administrador registre y de manera opcional para la venta de los mismos por parte del administrador y vendedores. Al finalizar una venta se emitirá un *ticket* el cual podrá ser impreso en una impresora térmica con conexión bluetooth.

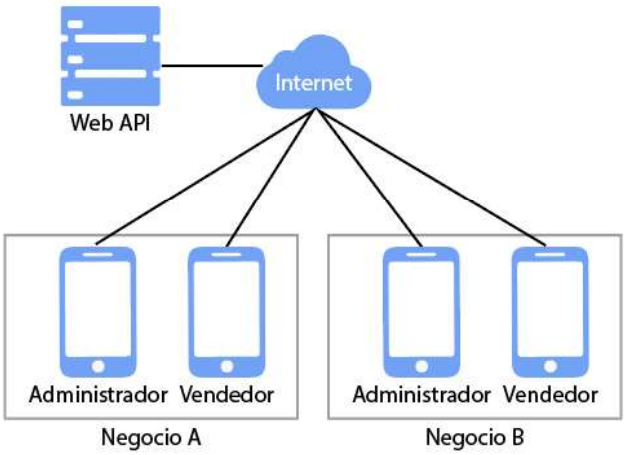


Figura 1.1 Elementos del prototipo de punto de venta

1.3 MARCO TEÓRICO

1.3.1 SERVICIOS WEB

Son un modelo de desarrollo de aplicaciones distribuidas a los cuales se puede acceder de forma remota a través de la red con el uso de identificadores estándar conocidos como URI[3], estos servicios en comparación con otros modelos de computación distribuida ofrecen como ventaja primordial la interoperabilidad, es decir que sin importar la plataforma de desarrollo del servicio estos estarán siempre disponible para su consumo a través de peticiones HTTP[4], también destaca la usabilidad, reusabilidad y una implementación eficiente y segura sobre tecnologías de Internet [3].

Los servicios web son una de las implementaciones más conocidas de la arquitectura SOA[5] (*Service Oriented Architecture*) y extienden sus características enfocándose en ser sistemas completamente interoperables [4] . Al heredar los principios de esta arquitectura,

---

[3] URI (*Uniform Resource Identifier*): es un conjunto de caracteres presentado en un formato definido para identificar recursos en internet.

[4] HTTP (*HyperText Transfer Protocol*): es un protocolo de capa de aplicación para la transferencia de recursos a través de internet.

[5] SOA (*Service-Oriented Architecture*): es un estilo arquitectónico para el desarrollo de aplicaciones con base en servicios disponibles.

los servicios web hacen uso de protocolos como TCP[6], IP[7] y HTTP para el transporte de la información a través de Internet. Y en función de los estándares que se usen para el formato de intercambio de mensajes, así como el acceso a recursos de la aplicación, los servicios web pueden ser de tipo SOAP[8] o REST.

1.3.2 ARQUITECTURA REST

Es un conjunto de lineamientos que permiten desarrollar una interfaz entre sistemas que usan HTTP para la transmisión y procesamiento de información. REST se presentó en el año 2000 como una alternativa a SOAP y su nombre es la contracción de *REpresentation State Transfer*.

La arquitectura REST se fundamenta en 6 principios [5].

- Arquitectura cliente-servidor: como base de las aplicaciones en donde se haga uso de REST para separar funciones entre los dos actores.
- *Stateless*: REST es una arquitectura sin estado, es decir que para cada petición se debe incluir un identificador de quien realiza la solicitud.
- Interfaz uniforme: Todos los recursos son accesibles a través de una URI la cual siempre hace referencia a un recurso no a una acción como en SOAP. Las acciones son definidas a través del uso de verbos HTTP.
- Navegación hipermedia: es el nivel más alto de desarrollo al que se alcanza con REST, dado que el servidor no almacena estados, estos se manejan a través de información incluida en las respuestas del servidor para la navegabilidad de la aplicación.
- Manejo de cache: siempre en el cliente, con la finalidad de reducir la carga del servidor y mejorar el tiempo de respuesta en el.
- Aplicación multicapa: en términos generales, el servidor debe estar compuesto por capas de presentación y datos e información. Cada una independiente de la otra y con una interacción adyacente únicamente.

---

[6] TCP (*Transport Control Protocol*): es un protocolo para el establecimiento de un canal de comunicación el cual garantiza la entrega de datos sin errores y en el mismo orden de envío.

[7] IP (*Internet Protocol*): es un protocolo que se encarga del envío y recepción de mensajes en la red.

[8] SOAP (*Simple Object Access Protocol*): es un conjunto de reglas para la comunicación de aplicaciones intercambiando información en formato XML.

Uno de las principales diferencias entre SOAP y REST es que este último está basado en recursos, a diferencia de SOAP que está basado en acciones. Para definir la acción a realizar REST hace uso de los verbos del protocolo HTTP (ver Tabla 1.1), también llamados métodos, y los agrega al encabezado de una petición.

Tabla 1.1 Principales verbos HTTP

Método	Descripción
`GET`	Se emplea para solicitar recursos.
`POST`	Se utiliza para enviar información, en el destino generalmente provoca cambio de estados o crecimiento de los registros almacenados, el método POST se usa también para solicitar recursos con base en varios parámetros de entrada.
`PUT`	Reemplaza una o varias entidades de un recurso con la información incluida en la petición.
`DELETE`	Usado para eliminar recursos.

Uno de los pilares de los servicios web basados en la arquitectura REST es el uso de JSON como formato de intercambio de datos. Esta convención está basada en el estándar ECMA-262 y se caracteriza porque su formato de texto es independiente a todo lenguaje de programación, es más ligero que XML[9] y es de fácil compresión y análisis para computadores y personas [6].

JavaScript Object Notation (JSON) está constituido en su forma más simple por una colección de pares clave-valor rodeados por llaves y separados por el carácter ‘,’ como se muestra en la Figura 1.2 donde el campo clave siempre es un identificador de tipo `string` [6].

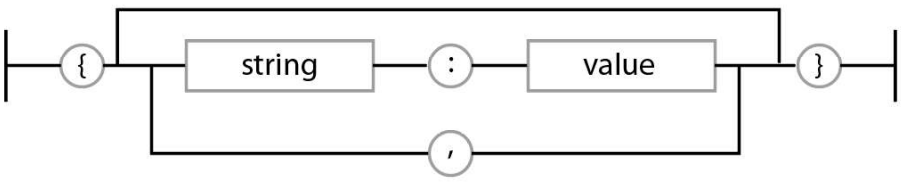


Figura 1.2 Estructura básica de un objeto JSON

[9] XML (*Extensible Markup Language*): es un lenguaje de marcado de propósito general para enviar datos a través de internet, la definición propósito general hace referencia a que XML permite definir etiquetas propias.

En el campo valor se pueden alojar objetos de tipos simples, la representación JSON de un objeto se ejemplifica en el Código 1.1 donde los valores de las claves `id` y `edad` son de tipo `integer`, los valores para las claves `nombre` y `apellido` son de tipo `string`, para la clave `altura` el valor es de tipo `decimal` y un objeto de tipo `bool` en la clave `estado`.

```json
{
  "id": 1,
  "nombre": "Alexandra",
  "apellido": "Noriega",
  "edad": 21,
  "altura": 1.65,
  "estado": true
}
```

Código 1.1 Ejemplo de un objeto JSON

En el campo valor se puede representar un objeto complejo, como se muestra en el Código 1.2 donde se tiene el objeto principal explicado previamente al cual se agrega un atributo (clave) `acceso`. Para representar esto, JSON define que el objeto anidado en esta última clave debe empezar y finalizar con los caracteres llaves y dentro de ellas se representa las claves del objeto siguiendo estructura básica de representación JSON.

```json
{
  "id": 1,
  "nombre": "Alexandra",
  "apellido": "Noriega",
  "edad": 21,
  "altura": 1.65,
  "estado": true,
  "acceso":{
    "correo": "alexandra_noriga@gmail.com",
    "clave": "nbcsjkdbvjerbgvklsdnclnsdvjkebrdfkjvncsldk..."
  }
}
```

Código 1.2 Ejemplo de objetos anidados en JSON

Finalmente, dentro de un objeto JSON se puede integrar una lista de objetos complementarios. La representación de listas de objetos supone que dichos objetos deben estar rodeados por corchetes como se ejemplifica en el Código 1.3 donde el objeto principal contiene una lista de objetos representadas en la clave `empresas` y como se mencionó previamente inicia y finaliza con corchetes. Al igual que en el caso anterior cada objeto de la lista se encuentra bordeado por llaves dentro de los cuales se detallan las claves y sus respectivos valores.

```json
{
  "id": 1,
  "nombre": "Alexandra",
  "apellido": "Noriega",
  "edad": 21,
  "altura": 1.65,
  "estado": true,
  "acceso":{
    "correo": "alexandra_noriga@gmail.com",
    "clave": "nbcsjkdbvjerbgvklsdnclnsdvjkebrdfkjvncsldk..."
  },
  "empresas":[
    {
      "id": 51,
      "nombre": "Distribuidora Jossbel"
    },
    {
      "id": 66,
      "nombre": "Calzado Elizabeth"
    }
  ]
}
```

Código 1.3 Ejemplo de lista de objetos en JSON

Como se mencionó previamente SOAP y REST son dos opciones para implementar servicios web. REST es la opción más usada para conectar aplicaciones como redes sociales, mensajería en tiempo real y servicios de alta disponibilidad pues se adaptan naturalmente a los estándares de Internet, al ser estructuralmente simples soportan grandes demandas de peticiones en el servidor y debido al uso de JSON, permite un intercambio de datos más ligero y por ende una comunicación más rápida, sin embargo, uno de las principales desventajas es la seguridad debido a que esta arquitectura no está fuertemente tipada.

Para los casos en donde se requiera seguridad en el intercambio de datos, se prefiere SOAP por lo que esta opción es altamente valorada en aplicaciones orientadas a la banca, telecomunicaciones y pasarelas de pago [7, 8].

1.3.3 FRAMEWORK WEB API

Web API es un *framework* de desarrollo propiedad de Microsoft que se integra al ya existente ASP.NET en sus versiones CORE y MVC, y que permite la comunicación y procesamiento de la información entre sistemas cliente-servidor a través de servicios REST.

Este *framework* fue desarrollado como una alternativa ligera a WCF[10] y está pensada para exponer servicios a una mayor gama de clientes como navegadores, teléfonos inteligentes y tabletas de una manera simple y general a través del protocolo HTTP. A la vez que es más ligero también permite un uso más sencillo pues está basado en verbos HTTP e integra a JSON como formato de intercambio de información por defecto [9].

El *framework* web API está basado en el modelo de desarrollo MVC por lo cual la implementación de la lógica de negocio se alojará en clases llamadas controladores [10], este *framework* está alojado sobre el espacio de nombres `System.Web.Http`, y la principal directiva de desarrollo es heredar de la clase `ApiController` como se muestra en el Código 1.4. Debido a la relación que el *framework* guarda con el modelo MVC y la convención de nombres que este mantiene, el nombre de una clase siempre terminará con la palabra *Controller* y este nombre a su vez será el recurso de la URI para acceder a sus métodos.

```
namespace webApiTesis.Controllers.api
{
    //METODO:  api/Cliente
    public class ClienteController : ApiController
    {
        //Implemetacion de la logica del negocio
    }
}
```

Código 1.4 Estructura básica de un controlador web API

Una clase que hereda de `ApiController` implementa de manera predefinida los métodos `Get`, `Post`, `Put` y `Delete` para controlar las acciones definidas en el protocolo HTTP cuando estas sean invocadas en una petición. Como se muestra en el Código 1.5 los métodos `GET` y `DELETE` reciben un objeto de tipo `integer` como parámetro de entrada el cual lo toma desde la URL, para el método `POST` se agrega el atributo `FromBody` a su firma, el cual permite extraer la información del cuerpo de la petición como un objeto JSON. Por su parte, el método `Put` recibe tanto un objeto `integer` desde la URL como información en formato JSON dentro del cuerpo de la petición.

[10] WCF (*Windows Communication Foundation*): es un *framework* para el desarrollo de aplicaciones orientadas a servicios, permite enviar mensaje en arquitecturas cliente-servidor, el servidor define una interfaz donde se definen los métodos a los que el cliente puede acceder a través de peticiones SOAP.

```
//GET: api/Cliente
public void Get()
{}

//GET: api/Cliente/5
public void Get(int id)
{ }

//POST: api/Cliente
public void Post([FromBody]JObject data)
{ }

//PUT: api/Cliente/5
public void Put(int id, [FromBody]JObject data)
{ }

//DELETE: api/Cliente/5
public void Delete(int id)
{ }
```

Código 1.5 Métodos principales de un controlador web API

Aunque el *framework* web API implementa de manera predefinida los métodos antes descritos, también brinda la flexibilidad de definir métodos personalizados a través del uso de atributos de acción y ruta como se muestra en el Código 1.6.

```
[HttpPost]
[Route("api/Cliente/filtro")]
// POST: Cliente/filtro
public void FiltroClientes([FromBody]JObject data)
{
    //Implementacion de logica de negocio
}
```

Código 1.6 Métodos personalizados HTTP con web API

1.3.4 BASES DE DATOS

Las bases de datos son almacenes de información, es decir, una colección de datos relacionados y organizados que aseguran la integridad y seguridad de dicha información. Para almacenar esta información las bases de datos cuentan con componentes jerárquicos como campos, registros y tablas [11] como se ejemplifica en la Figura 1.3.

El campo es un espacio de almacenamiento para un dato de un único tipo, un registro es un conjunto de campos relacionados que proporcionan información, aquí cada campo puede contener un tipo de dato diferentes, y finalmente la tabla, que es un conjunto de registros.

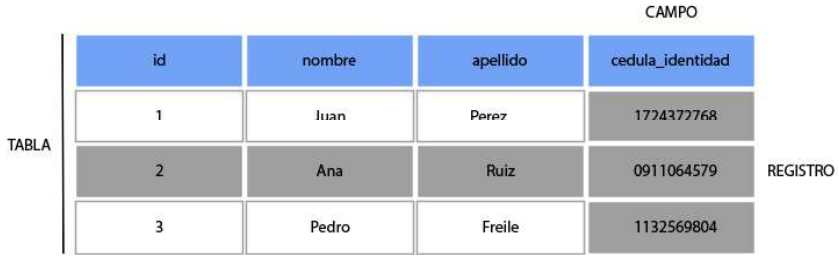


Figura 1.3 Elemento de una base de datos

Adicionalmente, las bases de datos cuentan con herramientas de *software* que sirve como interfaz entre la base de datos y el usuario, la función principal de estas herramientas es la administración y consulta de la información, pero también controlan los tipos de datos, las restricciones sobre la información y el acceso de usuarios a la misma, estas herramientas son conocidas como sistemas gestores de base de datos y entre las más conocidas se puede mencionar a MySQL, Microsoft SQL Server y PostgreSQL.

1.3.5 MODELO DE DATOS

Los modelos de bases de datos buscan representar de una manera generalmente gráfica la estructura lógica de una base de datos [12]. Desde el año 1963 se han conceptualizado varios modelos para describir estructuras complejas de datos que se presentan en la vida real, las relaciones existentes entre los objetos, la semántica y las restricciones de consistencia [13]. De todos modelos de datos destaca el modelo relacional de base de datos.

El modelo relacional de base de datos fue desarrollado por IBM en los años 70, es el modelo de base de datos más popular en la actualidad. Este modelo define los siguientes parámetros.

- Entidades: es una persona u objeto que se puede gestionar en la base de datos
- Atributos: son las propiedades o características de una entidad.
- Relaciones: representan los vínculos entre entidades.

Las relaciones hacen uso de identificadores en cada entidad llamados llaves primarias y foráneas, las llaves primarias permiten identificar de manera única a una entidad mientras que las llaves foráneas sirven para referenciar una entidad en otra. Cada relación tiene asociada una representación numérica llamada cardinalidad, esto permite conocer el contexto numérico de la relación.

La representación gráfica del modelo relacional de base de datos se ejemplifica en la Figura 1.4 donde se muestran 3 entidades `clientes`, `clientes_x_empresa` y `empresas` cada una con sus respectivos atributos y relaciones entre las mismas.

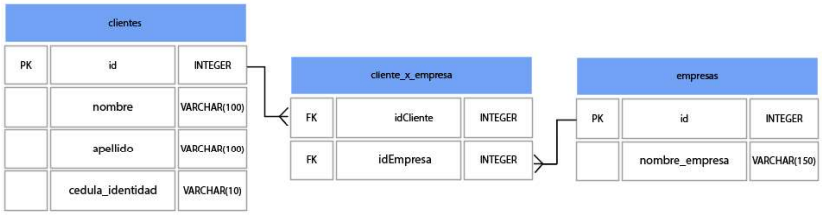


Figura 1.4 Modelo relacional de bases de datos

En el proceso de diseño de base de datos es común hacer uso de un esquema para la representación de una base de datos llamado diagrama entidad-relación o diagrama E-R para representar de una forma gráfica las entidades, atributos y relaciones definidas en el modelo relación (ver Figura 1.5).

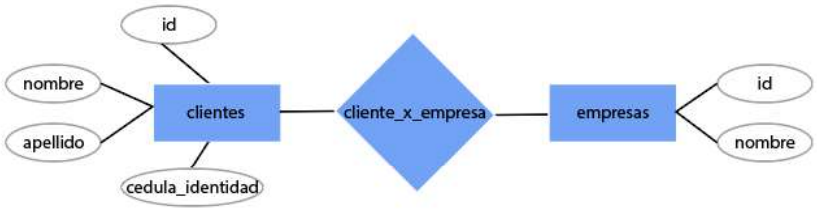


Figura 1.5 Ejemplificación de diagrama entidad-relación

En los diagramas E-R se hacen uso de los componentes descritos en la Tabla 1.2.

Tabla 1.2 Componentes de los diagramas entidad-relación

Componente	Descripción
Rectángulo	Representa conjuntos de campos o entidades
Elipse	Representa un atributo o campo
Rombos	Representa relaciones entre conjuntos de entidades
Líneas	Unen los atributos de cada entidad y las relaciones entre ellas

1.3.6 ENTITY FRAMEWORK

Entity Framework es un conjunto de librerías que permiten implementar la técnica de asociación objeto-relacional con la cual es posible convertir los tipos de datos de un lenguaje de programación orientado a objetos en tipos de datos con los que trabaja un sistema de base de datos. Los *frameworks* que implementan esta técnica son conocidos

como ORM[11] y Entity Framework no es el único ejemplo, también se puede mencionar a Hybernate en entornos de desarrollo con Java o Doctrine para aquellos que trabajan con PHP[12]. Los ORM pretenden agilizar el acceso a la capa de datos de una aplicación pues se encargan de traducir las tablas, relaciones, procesos almacenados entre otros a objetos utilizables en lenguaje de programación de manera automática [14] para así agilizar el proceso de desarrollo y evitar potenciales errores en el acceso a la capa datos. Estos objetos reposan sobre clases llamadas POCO (*Plain Old* CLR[13] *Objects*).

Entity Framework está basado en la tecnología *ActiveX Data Objects* (ADO) y su componente principal es el `EntityDataModel` en el cual se define los conjuntos de entidades, atributos y relaciones. A su vez, este se compone de tres elementos (ver Figura 1.6) que permiten una descripción completa de la base de datos.

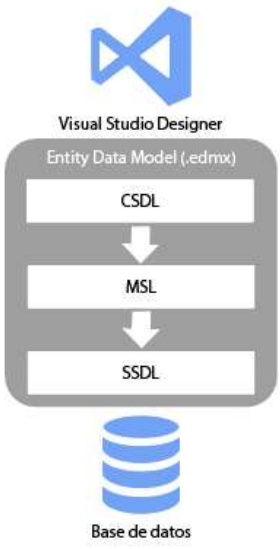


Figura 1.6 Estrutura de `EntityDataModel`

A continuación, se define cada componente del `EntityDataModel` [15].

---

[11] ORM (Object-Relational Mapping): es una técnica para convertir entidades de una base de datos a objetos utilizables en lenguaje de programación orientada a objetos.

[12] PHP (*Hypertext Preprocessor*): es un lenguaje de programación de código del lado del servidor, fue originalmente creado para el desarrollo web de contenido dinámico, aunque en la actualidad se han desarrollado framework que permiten su uso en el desarrollo de cualquier sistema.

[13] CLR (*Common Language Runtime*): es un entorno de ejecución de código para la plataforma .NET [37]

- *Storage Squema Definition Language* (SSDL): describe la estructura de una base de datos, es decir, tablas, relaciones, vistas y procesos almacenados.
- *Conceptual Squema Definition Language* (CSDL): describe las entidades del modelo conceptual y permite manejar las relaciones como propiedades.
- *Mapping Specification Language* (MSL): es el encargado de mantener una relación entre el SSDLy el CSDL.

Entity Framework cuenta con tres enfoques de desarrollo llamados *Code First, DataBase First* y *Model First*.

- *Code First* es un enfoque destinado a crear una nueva base de datos a partir de clases definas previamente, es el enfoque más popular debido a que se tiene el control total de la base de datos a través de las clases y las configuraciones que se describan en el código.
- *DataBase First* es un enfoque alternativo, es utilizado cuando ya se cuenta con una base de datos definida y al contrario de *Code First*, la asociación para este esquema consistirá en crear las clases a partir de la base de datos ya existente. Se dice que es un enfoque alternativo ya que para su administración o modificación es necesario un entorno de trabajo extra el cual puede ser Microsoft SQL Server Manager.
- *Model First* es un enfoque de diseño gráfico, en el cual se hace uso de la herramienta Visual Studio Designer por medio del cual se crean tablas, columnas y relaciones sin necesidad de código, Entity Framework se encarga de la traducción de las tablas gráficas a entidades POCO y con ello la creación de una base de datos.

La asociación de una base de datos con el ORM Entity Framework bajo el esquema *DataBase First* crea una clase de contexto en el archivo denominado `DatabaseModel.Context.cs`. Esta clase es una de las más importantes pues es la puerta de acceso a toda la capa de datos, esta clase hereda de la clase `DBContext` la cual a su vez se encuentra en el espacio de nombres `System.Data.Entity`, y contiene la colección de tablas asociadas a las cuales se puede tener acceso.

El contenido de esta clase de contexto se muestra en el Código 1.7. El constructor de la clase permite especificar configuraciones del comportamiento de Entity Framework, las líneas de código definidas en este método permite desactivar la carga automática de objetos relacionadas con el fin de mejorar el tiempo de respuesta de las búsquedas en la

base de datos. La clase de contexto contiene también una lista de propiedades de tipo `DbSet` las cuales son las tablas asociadas a objetos a los cuales se puede tener acceso.

```
public partial class tesisPoliEntities : DbContext
{
    public tesisPoliEntities(): base("name=tesisPoliEntities")
    {
        this.Configuration.LazyLoadingEnabled = true;
        this.Configuration.ProxyCreationEnabled = false;
    }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        throw new UnintentionalCodeFirstException();
    }

    public virtual DbSet<cargos> cargos { get; set; }
    public virtual DbSet<categorias> categorias { get; set; }
    public virtual DbSet<clientes> clientes { get; set; }
    public virtual DbSet<clientes_x_negocios> clientes_x_negocios { get; set; }
    public virtual DbSet<impuestos> impuestos { get; set; }
    public virtual DbSet<negocios> negocios { get; set; }
    public virtual DbSet<productos> productos { get; set; }
    public virtual DbSet<productos_x_tallas> productos_x_tallas { get; set; }
    public virtual DbSet<productos_x_ventas> productos_x_ventas { get; set; }
    public virtual DbSet<tallas> tallas { get; set; }
    public virtual DbSet<tipos_tallas> tipos_tallas { get; set; }
    public virtual DbSet<usuarios> usuarios { get; set; }
    public virtual DbSet<usuarios_x_cargos_x_negocios> usuarios_x_cargos_x_negocios { get; set; }
    public virtual DbSet<ventas> ventas { get; set; }
    public virtual DbSet<formas_pago> formas_pago { get; set; }
    public virtual DbSet<pagos_ventas> pagos_ventas { get; set; }
    public virtual DbSet<empleados> empleados { get; set; }
    public virtual DbSet<sociedades_fiscales> sociedades_fiscales { get; set; }
}
```

Código 1.7 Archivo de contexto en Entity Framework

1.3.7 LINQ

Lenguage Integrate Query es un *framework* de .NET que permite la consulta de múltiples fuentes de datos con una sola sintaxis unificada, para ello el *framework* hace uso de expresiones de consulta del lado del cliente y las transforma al lenguaje nativo en el lado de la fuente de datos. Como se menciona una fuente de datos puede ser una lista simple de objetos o una base de datos compleja desarrollada en MySQL, SQL Server u otros gestores.

LINQ es una alternativa cada vez más usada en reemplazo a DAO[14] pues reduce significativamente la cantidad de código, evita a los programadores aprender un lenguaje específico para cada fuente de datos usada y ayuda a un entendimiento y mantenimiento más fácil y rápido del código. Las sintaxis de consulta están basadas en lenguaje SQL y pueden hacer uso de consultas textuales o métodos [16].

[14] DAO (*Data Access Object*): es un patrón de diseño para crear la capa de datos de una aplicación [37].

1.3.7.1 Sintaxis de consulta LINQ

Es la forma más básica de consulta, utiliza las palabras reservadas `from` para especificar una fuente de datos, `select` para especificar el resultado esperado y opcionalmente palabras `where` como condicional de búsqueda, y, `join` para relazar búsquedas complejas trabajando sobre múltiples fuentes de datos. Además, hace uso de un alias como iterador de objetos sobre los cuales se realiza la consulta, este alias permite acceder a las propiedades de cada objeto (ver Figura 1.7).

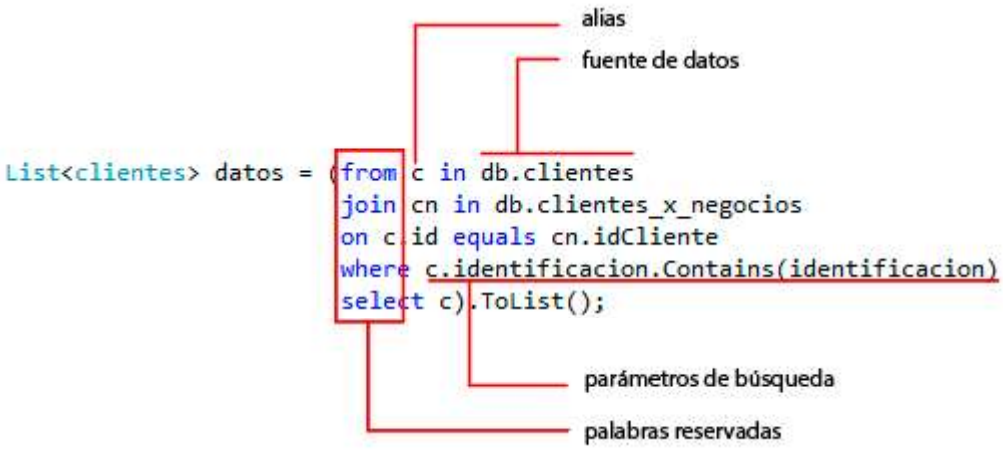


Figura 1.7 Estructura de sintaxis de consulta LINQ

1.3.7.2 Sintaxis de métodos LINQ

Hace uso de métodos definidos en las clases `Enumerable` y `Queryable` del *framework* .NET, así también hace uso de parámetros de búsqueda conocidos como expresiones lamba para filtrar el resultado de la búsqueda. Al igual que en la sintaxis de consulta se cuenta con un alias como iterador de objetos y este permite tener acceso a las propiedades de cada objeto (ver Figura 1.8).

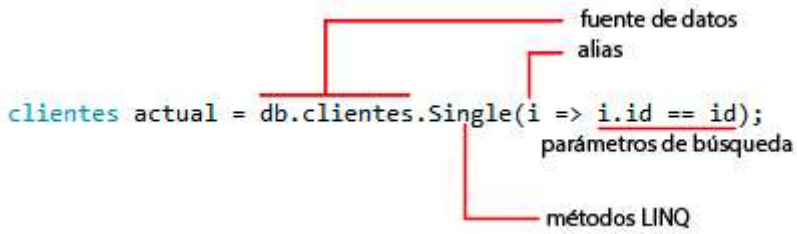


Figura 1.8 Estructura de sintaxis de métodos LINQ

Los métodos más usados bajo esta sintaxis de consulta se detallan en la Tabla 1.3:

Tabla 1.3 Principales métodos de LINQ

Método	Descripción
`Single`	Permite obtener un solo registro de una fuente de datos en relación a los parámetros de búsqueda definidos como argumento de entrada, si no existe coincidencias, este método provocará una excepción.
`Find`	Genera el mismo resultado que la función `Single`, este método difiere en que si no existen coincidencias se genera un valor nulo pero no una excepción.
`Where`	Permite obtener una colección de registros de una fuente de datos que coincidan con las condiciones de búsqueda definidos como argumento de entrada. La salida de este método es un objeto `IQueryable`. Si no existen coincidencias, este método provocará una excepción.
`First`	Permite obtener el primer registro de una colección, si la colección no contiene elementos este método provocará una excepción.
`FirstOrDefault`	Genera el mismo resultado de que la función `First`, la diferencia es que si la colección no contiene elementos el resultado será un valor nulo, pero no una excepción.
`Last`	Permite obtener el último registro de una colección, si la colección no contiene elementos este método provocará una excepción.
`LastOrDefault`	Genera el mismo resultado de que la función `Last`, la diferencia es que si la colección no contiene elementos el resultado será un valor nulo, pero no una excepción.
`ToList`	Permite transformar un objeto de la clase `IQueryable` en un objeto de la clase `List`.

1.3.8 ANDROID

Android es un sistema operativo de código abierto para dispositivos móviles impulsado por Google y muchas otras compañías dedicadas a las telecomunicaciones, *software* y desarrollo electrónico, su núcleo está basado en Linux. Desde 2017 Kotlin es su lenguaje oficial de programación, aunque mantiene vigente a Java como una alternativa, el sistema operativo Android se caracteriza principalmente por su sencillez y total interoperabilidad con Java. Al ser un sistema operativo de código abierto cuenta con una gran comunidad de desarrolladores como respaldo, así también un código abierto brinda mayor transparencia y con esto seguridad y mantenimiento ágil y rápido. Para el desarrollo de aplicaciones móviles en Android se requiere de un conjunto de librerías para el control del *hardware* del dispositivo, integración de servicios y un emulador para probar la aplicación

desarrollada, este conjunto de librerías es conocido como Android SDK. Las librerías son generadas por la comunidad de desarrolladores, pero es posible agregar librerías externas a los proyectos de trabajo. Por otro lado, es necesario un IDE[15] como Android Studio que permite una interacción de forma gráfica con las librerías y herramientas antes mencionadas [17].

Android Studio está basado en IntelliJ IDEA el cual es la pieza clave del desarrollo unificado de aplicaciones para todo tipo de dispositivos, un compilador conocido como Gradle y una herramienta llamada Instant Run para aplicar cambios en vivo durante las pruebas. En referencia a la estructura de un proyecto, se destacan tres módulos de desarrollo (ver Figura 1.9).

- Módulos de aplicación: el cual abarca a todos los archivos que permiten la funcionalidad de la aplicación.
- Bibliotecas: donde se alojan los recursos externos para la aplicación.
- Módulos de configuración los cuales contienen los archivos de configuración y permisos del dispositivo y de la aplicación, de los cuales destaca el archivo de configuración del compilador `build.gradle`.

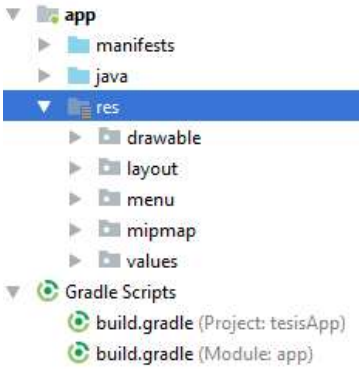


Figura 1.9 Estructura del proyecto Android

El módulo de aplicación está compuesto por tres directorios:

- `manifest`: el cual contiene al archivo `AndroidManifest.xml`
- `java`: el cual contiene los archivos fuentes para la funcionalidad de la aplicación.

[15] IDE (*Integrated Development Environment*): es una aplicación que brinda las herramientas básicas para el desarrollo y prueba de *software*.

- `Res`: el cual contiene todos los diseños gráficos en formato XML, imágenes, estilos, biblioteca de colores, entre otros.

1.3.8.1 Archivo AndroidManifest.xml

Este archivo brinda información esencial sobre la aplicación al sistema operativo, el archivo `AndroidManifest.xml` se encarga de nombrar el paquete de Java para la aplicación a través del atributo `package`, como se muestra en el Código 1.8, este nombre es único y sirve como identificador de la aplicación una vez que esta se publica en la tienda de Google Play.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.kevin.calvache.tesisapp">
```

Código 1.8 Atributo `package` en el archivo `AndroidManifest.xml`

Otro de los objetivos de este archivo es describir los parámetros básicos de la aplicación como el icono mediante los atributos `android:icon` y `android:roundIcon`, el nombre de la aplicación que se muestra en la pantalla de los dispositivos móviles mediante el atributo `android:label`, el resguardo de datos en la cuenta de Google Drive a través de los atributos `android:allowBackup` y `android:fullBackupOnly`, como se puede ver en el Código 1.9.

```
<application
    android:name=".Model.Mensaje.App"
    android:allowBackup="false"
    android:fullBackupOnly="false"
    android:icon="@mipmap/ic_launcher_foreground"
    android:label="Tesis App"
    android:roundIcon="@mipmap/ic_launcher_foreground"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">
    <service...>
    <activity...>
    <activity...>
</application>
```

Código 1.9 Parámetros básicos de una app en archivo `AndroidManifest.xml`

Adicionalmente en el archivo `AndroidManifest.xml` se detallan las actividades contenidas en la aplicación, las cuales se definen con la etiqueta `<activity>`, los

servicios y los receptores de mensajes a través de la etiqueta `<service>`. Estas definiciones notifican al sistema Android los componentes y las condiciones necesarias para el lanzamiento de una aplicación [18].

El último objetivo de este archivo es especificar al sistema operativo los permisos de acceso hacia los componentes de hardware integrados en el dispositivo móvil que se debe otorgar para el correcto funcionamiento de la app, esta definición se hace dentro de la etiqueta `<uses-permission>` y `<uses-feature>` como se muestra en el Código 1.10 donde se definen los permisos para el uso de la conexión a Internet, el acceso a la cámara haciendo uso de la característica de enfoque automático (*autofocus*) y la conexión Bluetooth.

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.CAMERA" />
<uses-feature android:name="android.hardware.camera" />
<uses-feature android:name="android.hardware.camera.autofocus" />
<uses-permission android:name="android.permission.BLUETOOTH" />
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
```

Código 1.10 Definición de permisos en el `AndroidManifest.xml`

1.3.8.2 Archivos build.gradle

Son archivos de configuración del compilador. Dado que un proyecto de Android Studio puede contener varios módulos de desarrollo, lo cual es similar al manejo de soluciones y proyectos en el entorno de desarrollo Visual Studio, existe un archivo con la configuración global del proyecto llamado `build.gradle(Project)` y tantos archivos de configuración especifica como módulos existan en el proyecto, estos son denominados `build.gradle(Module)` [19].

Los archivos `gradle` tiene a cargo dos funciones, por una parte, y dentro del apartado `android` que se muestra en el Código 1.11 se encargan de especificar en el campo `minSDKVersion` la versión mínima del sistema operativo sobre el cual puede ejecutarse una aplicación, identifican a la aplicación a través de un nombre único del paquete a través del campo `applicationId` y llevan un control de la versión de desarrollo con el uso de los campos `versionCode` y `versionName`. El campo `targetSdkVersion` define la versión de sistema operativo sobre la cual la aplicación brindará las mayores prestaciones.

```
android {
    compileSdkVersion 27
    defaultConfig {
        applicationId "com.kevin.calvache.tesisapp"
        minSdkVersion 18
        targetSdkVersion 26
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }
```

Código 1.11 Apartado `android` del archivo `build.gradle`

Y por otro lado dentro del apartado `dependencies`, se especifica las librerías incluidas o excluidas para el desarrollo de la aplicación. En el Código 1.12 se muestra un ejemplo con las librerías o paquetes por defecto que Android Studio incluye en un proyecto, la inclusión de una librería se realiza a través de la palabra reservada `implementation` seguida del nombre del paquete.

```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation 'com.android.support:appcompat-v7:27.0.0'
    implementation 'com.android.support.constraint:constraint-layout:1.1.2'
    implementation 'com.android.support:design:27.1.1'
    implementation 'com.android.support:support-v4:27.1.1'
    implementation 'com.android.support:recyclerview-v7:27.1.1'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'com.android.support.test:runner:1.0.2'
    androidTestImplementation 'com.android.support.test.espresso:espresso-core:3.0.2'
}
```

Código 1.12 Apartado `dependencies` en archivo `build.gradle`

Para fines prácticos y en relación al alcance de este Proyecto Técnico de aquí en adelante cuando se haga referencia a los archivos `gradle` se entenderá que se hace referencia específica al archivo `biuld.gradle(Module)`.

1.3.9 LIBRERÍAS EXTERNAS DE ANDROID

Como se mencionó en el apartado previo, el desarrollo de aplicaciones móviles Android permite la inclusión de librerías desarrollados por terceros. Algunos ejemplos de dichas librerías son los siguientes:

1.3.9.1 Zxing

Es una librería de código abierto para el procesamiento de imágenes de código de barras en una y dos dimensiones, la librería Zxing está escrita en Java y soporta múltiples formatos de códigos de barras como se muestra en la Tabla 1.4. Para la inclusión de la librería en el proyecto es necesario agregar el nombre del paquete `com.google.zxing:core:3.2.1`

al aparatado `dependencies` del archivo `gradle` y agregar las líneas que se muestran en el Código 1.13 en el archivo `AndroidManifest.xml` para especificar al sistema operativo que la aplicación necesita acceso a la cámara del dispositivo y a la característica de auto enfoque de existir.

Tabla 1.4 Estándares aceptados para lectura de código de barras [20]

Productos 1D	Industrial 1D	2D
UPC-A	Code 39	QR Code
UPC-E	Code 93	Data Matrix
UEAN-8	Code 128	MaxiCode
EAN-13	Codabar	RSS-14
	ITF	RSS-Expanded

```
<uses-permission android:name="android.permission.CAMERA" />
<uses-feature android:name="android.hardware.camera" />
<uses-feature android:name="android.hardware.camera.autofocus" />
```

Código 1.13 Permisos de acceso a cámara declarados en `AndroidManifest.xml`

1.3.9.2 Bluetooth Thermal SDK

Es una librería propietaria de Xiamen Electronic Technology Co. Permite la impresión de texto, imágenes, y códigos de barras en una y dos dimensiones. Está escrita en C++ y está diseñada para trabajar con una amplia gama de impresoras. Para la inclusión de esta librería es necesario solicitar el código fuente empaquetado en un archivo `.jar` a la compañía vía correo electrónico o contacto directo disponible en su página web y agregarlo a la carpeta `/app/lib` del proyecto. La librería cuenta con la clase `BluetoothDevice` para representar un dispositivo y concentra la lógica de impresión en la clase `BluetoothService` dentro de la cual se encuentra el método `sendMessage`. Dado que esta librería hace uso de la conexión *bluetooth* del dispositivo es necesario definir los permisos para el manejo de la conexión *bluetooth* en el dispositivo móvil en el archivo `AndroidManifest.xml` con las líneas que se muestran en el Código 1.14.

```
<uses-permission android:name="android.permission.BLUETOOTH" />
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
```

Código 1.14 Permisos de acceso a *bluetooth* declarados en `AndroidManifest.xml`

1.3.9.3 MPAndroid Chart

Es una librería de código abierto que permite crear gráficos estadísticos como líneas temporales, gráficos de barras simples y múltiples, pasteles entre muchos otros. Para la inclusión de esta librería es necesario agregar el nombre del paquete `com.github.PhilJay:MPAndroidChart:v3.1.0-alpha` dentro del apartado `dependencies` del archivo `gradle`. Cuenta con una amplia documentación y varios ejemplos de desarrollo en su página oficial de GitHub [21].

1.3.9.4 Firebase Cloud Messaging

Firebase es un sistema distribuido en la nube que Google pone a disposición como herramienta para el desarrollo de aplicaciones móviles y web. Firebase Cloud Messaging (FCM) es uno de los servicios ofertados que permite incluir la función de mensajería remota en tiempo real entre un servidor y un cliente. FCM establece un servicio confiable de mensajería a través de su infraestructura como intermediario (ver Figura 1.10) y permite enviar notificaciones, así como datos estructurados para la presentación de los mismos en una aplicación [22].

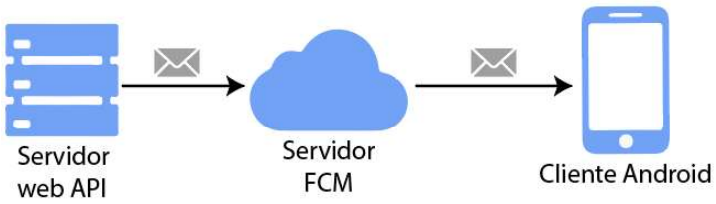


Figura 1.10 Arquitectura FCM

Además, del lado del cliente, FCM genera un *token* que identifica de manera única a un dispositivo móvil, este identificador se debe registrar en la base de datos con el objetivo de enviar mensajes o notificaciones de manera *unicast* o *multicast* a los usuarios. Para el uso de este servicio en el lado del cliente se debe incluir los nombres de los paquetes necesarios en el archivo `gradle` como se muestra en el Código 1.15.

```
implementation 'com.google.firebase:firebase-core:16.0.4'
implementation 'com.google.firebase:firebase-messaging:17.3.3'
```

Código 1.15 Inclusión de FCM en cliente Android

Firebase Cloud Messaging genera un hilo de escucha permanente para el arribo de mensajes en el cliente. Para capturar los datos es necesario definir una clase y heredarla

de `FirebaseMessagingService [23]`, esto supone la implementación del método `onNewToken` en donde se obtienen el *token* de identificación antes mencionado como se muestra en el Código 1.16.

```
@Override
public void onNewToken(String token) {

}
```

Código 1.16 Método `onNewToken` de FCM en cliente Android

Y el método `onMessageReceived` el cual contiene los datos que arriban desde el servidor FCM para su procesamiento ya sea a partir de una notificación o de un mensaje de texto en cuyo caso la información arribará en formato JSON (ver Código 1.17).

```
@Override
public void onMessageReceived(RemoteMessage remoteMessage) {
    super.onMessageReceived(remoteMessage);

    if(remoteMessage.getNotification() != null) { //arribo de notificación
        String datosNotificaion = remoteMessage.getNotification().getBody();
    }
    else{//arribo de mensaje de datos
         Map<String, String> datosMensaje = remoteMessage.getData();
    }
}
```

Código 1.17 Captura de datos en el método `OnMessageReceive`

Para el envío de información desde el servidor web API hacia los clientes móviles se debe generar una solicitud de tipo REST hacia el servidor FCM (ver Figura 1.10) dado que este actúa como intermediario, con la información deseada para su reenvío (ver Código 1.18), esta información debe ser estructurada en formato JSON de acuerdo a lo definido en la documentación oficial. La estructura de la información tanto para el envío de notificaciones y mensajes de datos, así como ejemplos de desarrollo se encuentra en la página oficial de Firebase Cloud Messaging.

```
public static void sendNotification(FirebaseNotif data)
{
    var client = new RestClient("https://fcm.googleapis.com/fcm/send");
    var request = new RestRequest(Method.POST);
    request.AddHeader("Authorization", "key=...");
    request.AddParameter("application/json", new JavaScriptSerializer().Serialize(data),
                         ParameterType.RequestBody);
}
```

Código 1.18 Petición a servidor FCM

1.3.10 METODOLOGÍA DE DESARROLLO SCRUM

Scrum es un marco de trabajo diseñado por Ken Schwaber y Jeff Sutherland para el desarrollo ágil de proyectos. Esta metodología es una de las más usadas actualmente pues brinda transparencia en los procesos, incrementa la flexibilidad del trabajo de los colaboradores y reduce los riesgos técnicos y financieros del negocio [24]. A pesar de que Scrum tradicionalmente ha sido utilizado en el desarrollo de productos y proyectos de software, es de gran valor en la administración de proyectos comerciales, administrativos e incluso para la planificación de actividades cotidianas.

Como visión general en Scrum se especifican dos etapas. En la primera etapa conocida como etapa de diseño, el dueño del producto llamado *product owner* en conjunto con el jefe del equipo de desarrollo llamado *scrum master* definen las características y funcionalidades del producto software a desarrollar, dichas funcionalidades se enlistan, organizan, priorizan y se estima su tiempo de desarrollo obteniendo un primer documento llamado *product backlog*, las tareas de esta lista se dividen en relación a su complejidad y tiempo de desarrollo para formar grupos de tareas que no tomen más allá de 4 semanas en su culminación, este periodo de tiempo es llamado *sprint*. Estos grupos de tareas formados se registran en documentos conocidos como *sprint backlogs*.

En la segunda etapa conocida como implementación, se tiene 4 sub etapas: planificación del *sprint* donde en una reunión de hasta 2 horas, los miembros del equipo conocidos como *scrum team* junto con el *scrum master* definen objetivos, revisan la documentación y proponen cambios de ser necesario, luego, la etapa de desarrollo donde se realiza la codificación de las tareas y se obtiene un producto potencial para el *product owner*. En esta etapa se hace uso de una herramienta conocida como tablero Scrum la cual tiene como objetivo el seguimiento de las actividades planificadas de manera constante. Las etapas de revisión del *sprint* y retroalimentación buscan la aceptación del producto potencial generado al final del *sprint* por parte del *product owner* y se realizan cambios de ser necesario (ver Figura 1.11).

En Scrum los integrantes del grupo de desarrollo forman equipos multidisciplinarios y auto dirigidos a quienes no se les prescribe métodos de trabajo en particular sino la tarea a ejecutar. Finalmente, y como diferencia a los métodos tradicionales de desarrollo en donde se genera exhaustiva documentación Scrum define una serie de reuniones cortas a lo largo de todo el desarrollo con el objetivo de tener una mayor cercanía a los problemas que se presentación y al avance del proyecto [25].

Figura 1.11 Proceso Scrum **[25]**

A continuación, se describe más a detalle cada uno de los actores, herramientas y condiciones definidas en la metodología Scrum.

1.3.10.1 Roles

En Scrum se definen tres roles: *product owner*, *scrum master* y *scrum team*.

- *Product owner*: es el encargado de definir las características y funcionalidades deseadas en el producto, en función de quien propone el producto, este rol puede ser interpretado por un cliente en caso de aplicaciones comerciales o un representante interno al grupo de trabajo en caso de aplicaciones de uso local dentro de una compañía o grupo de trabajo. Sin importar quien interprete este rol, el *product owner* es el encargado de interactuar con el equipo de trabajo para definir las actividades a desarrollar, la prioridad de las mismas y la verificación de su cumplimiento. Scrum dicta que el *product owner* es un rol unipersonal, es decir, que si quienes piden un producto es un grupo de personas, ellos deben designar a un representante que esté a cargo de todas las actividades mencionadas.
- *Scrum team*: son los responsables de la ejecución del proyecto, el *scrum team* está constituido por desarrolladores, analistas, diseñadores etc. en grupos de entre 5 y 9 personas dentro de los cuales no hay ningún nivel de jerarquía en relación a sus funciones o experiencia, todos son miembros de un mismo equipo y tiene la misma importancia. Estos grupos tienen como precepto la auto organización en función a las necesidades que una tarea demande.

- *Scrum master:* es un representante del Scrum team, es el responsable de generar la documentación necesaria para el desarrollo de tareas, liderar y guiar al equipo en el desarrollo de las tareas, interactuar con el *product owner* sobre posibles diferencias o cambios en las actividades planificadas, hacer cumplir y sobre todo formar al cliente y equipo en los procesos definidos en esta metodología.

1.3.10.2 Artefactos

Los artefactos son documentos que rigen el proceso de desarrollo del producto *software*, se los define en la etapa de diseño. Scrum cuenta con tres artefactos:

- *Product backlog*: es una lista priorizada de las funcionalidades que el *product owner* define para su producto como se muestra en la Figura 1.12, el *backlog* es la hoja de ruta que seguirá el proyecto razón por la cual tiene que ser tan detallado como sea posible y cada actividad listada debe definir una estimación de tiempo en horas o días. El *product backlog* no contiene historias de usuario, contiene elementos, son los elementos, quienes se pueden representar a través de historias de usuario o casos de uso [24].

Prioridad	Elemento
1	Como administrador, quiero añadir un empleado, modificar sus datos y eliminarlos de ser necesario
2	Como empleado, quiero poder generar una venta y asociarlo a un cliente o como consumidor final
3	Como empleado, quiero registrar clientes y poder ver las ventas realizadas

Figura 1.12 *Product backlog*

- *Sprint backlog*: es una lista de las actividades que el *scrum team* tiene que realizar en un periodo tiempo (ver Figura 1.13), también llamado *sprint*. Las tareas del *sprint backlog* no son priorizadas ni son asignadas a un miembro del equipo en particular, pues es el *scrum team* en conjunto quien determina que tarea cumplir, quien va a realizarla y cuando hacerla [26].

Elemento del *backlog*	Tarea del *sprint*	Responsable
Como administrador, quiero añadir un empleado, modificar sus datos y eliminarlos de ser necesario	Codificar proceso de registro	
	Codificar vista(UI)	
	Levantar tabla en BDD	
	.....	

Figura 1.13 *Sprint backlog*

1.3.10.3 Reuniones

Las reuniones son la segunda pieza más importante de la metodología pues en estas se analizan potenciales problemas, se verifican resultados y de ser necesario se hacen correctivos a equipos y procesos para mantener el correcto ritmo de trabajo.

- Planificación del *sprint*: se lleva a cabo entre el equipo de desarrollo y el *scrum master*, a partir del *product backlog* se diseña el *sprint backlog*, se lleva a cabo previo al inicio de cada *sprint*. El *product owner* puede participar en caso de que se necesite replantear o priorizar requisitos.
- Scrum diario: es una reunión diaria que tiene una duración aproximada de 15 minutos, en ella participan los miembros del equipo, el *scrum master* y el *product owner* aunque su presencia no es siempre requerida. La finalidad de esta reunión es conocer el avance del proyecto. Todos los miembros tienen que responder a 3 preguntas esenciales: ¿qué se hizo el día de ayer?, ¿qué se va a hacer hoy? y ¿qué obstáculos se presentaron?, si esta última pregunta es afirmativa se planifica una reunión por separado entre las partes necesarias para resolver el conflicto. En el desarrollo diario y como herramienta de transparencia en los procesos se hace uso de una tabla llamada “Tablero Scrum” donde se plasma el avance real de cada una de las tareas del *sprint*. Contiene 3 columnas: *ToDo* en donde se alojarán las tareas pendientes que no se han iniciado, *Doing* en donde se ubicaran las tareas que actualmente se están realizando y finalmente la columna *Done* para las tareas finalizadas (ver Figura 1.15), la actualización del estado de una tarea está a cargo de miembro del equipo responsable de ejecutar dicha tarea.

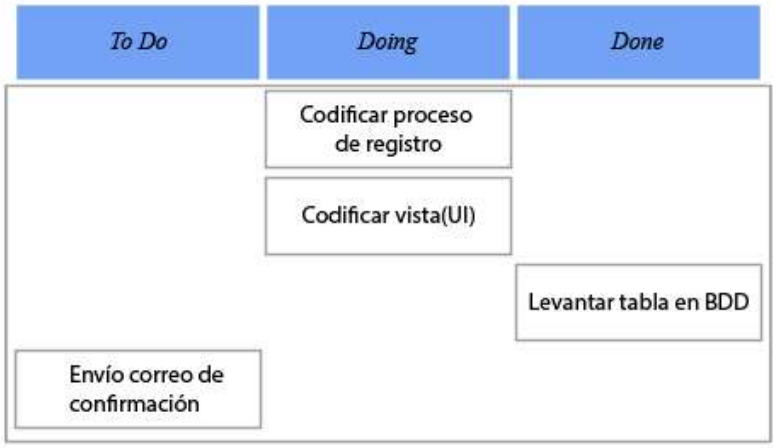


Figura 1.14 Tablero Scrum

- Revisión del *sprint*: al finalizar cada *sprint* se convoca a una reunión entre todos los miembros involucrados incluido de manera obligatoria el *product owner,* tiene una duración promedio de 2 horas y su finalidad es de presentar las nuevas funcionalidades o características implementadas.
- Retrospectiva del *sprint*: se realiza al finalizar cada sprint, es una reunión corta de máximo 30 minutos en donde todos los miembros del equipo buscan posibles errores en el producto entregado. Los errores encontrados se corregirán en paralelo al desarrollo del nuevo *sprint*.

2. METODOLOGÍA

Este Proyecto Técnico se desarrollará a través de una investigación aplicada en el cual se pondrán en práctica los fundamentos teóricos y prácticos relacionados a bases de datos, programación orientada a objetos, desarrollo de aplicaciones móviles y aplicaciones distribuidas, con el fin de generar un prototipo de punto de venta que permita el control del inventario en locales de venta de ropa y calzado.

Para el desarrollo de este prototipo se adoptó la metodología de desarrollo de software ágil Scrum, la cual plantea dos fases.

En la fase inicial conocida como fase de diseño se planteó la visión general del proyecto, se estudió las funcionalidades, los procesos de venta y control de inventario establecidos en aplicaciones como Square [27] y Loyverse [28], complementariamente se realizaron entrevistas con las cuales se levantaron los requerimientos funcionales desde la perspectiva del usuario y no funcionales desde la perspectiva del sistema.

Los requerimientos del prototipo, desde la perspectiva de usuario, fueron plasmados en historias de usuario. Se estableció la arquitectura del prototipo y con los requerimientos definidos se generó el diagrama entidad-relación, el diagrama relacional de base de datos y el diagrama de clases de la web API, también se generó el *product backlog*, se planificaron las tareas por cada *sprint* y dichas tareas fueron agrupadas formando los *sprint backlogs*. Adicionalmente se generaron los *sketches* y *wireframes* como guía de desarrollo para las interfaces de usuario.

En la segunda fase, conocida como etapa de implementación se codificaron las tareas planificadas por *sprint*. Cada tarea comprendió la codificación de la lógica en el *backend* con el uso del *framework* web API y el lenguaje de programación C# como la implementación de interfaces gráficas y sus respectivas clases de control en el *frontend* con el uso de tecnologías nativas de Android.

La metodología propuesta estipula como etapas complementarias al desarrollo de cada *sprint*, las etapas de revisión y retrospectiva, donde el producto obtenido debe ser revisado con el fin de verificar el correcto funcionamiento del mismo y de ser necesario realizar correcciones.

2.1 LEVANTAMIENTO DE INFORMACIÓN

Para el desarrollo del prototipo de punto de venta se analizaron las funcionalidades de las aplicaciones Loyverse [28] y Square [27].

Loyverse es una aplicación móvil desarrollada para manejar los procesos de venta, controlar el inventario de productos y generar reportes de venta en pequeños comercios. [29]. La aplicación permite al usuario registrar categorías y asociar productos a dichas categorías. En la vista principal se muestra el catálogo de productos registrados, del cual el usuario puede seleccionar un producto y la cantidad deseada para armar una venta. La venta puede o no asociarse a un cliente y al finalizar la misma la aplicación permite imprimir un *ticket* de venta a través de impresoras conectadas vía Bluetooth o por conexión de red.

Square es otra aplicación móvil POS que a diferencia de Loyverse presenta como vista inicial los productos seleccionados para la venta y un botón auxiliar para agregar más productos. Square además integra funciones para trabajar de forma simultánea entre varios empleados en un local comercial, y llevar un control administrativo de los mismos. Una de las ventajas competitivas de este sistema es que cuenta con varios *gadgets*[16] propietarios que se acoplan al dispositivo móvil para cobrar el valor de una venta a través de tarjetas de crédito y débito, así como también para permitir la lectura de códigos de barra o la impresión de *tickets* de venta [29].

A partir de las aplicaciones antes descritas, se tomaron las ideas centrales para el prototipo de punto de venta:

- Permitir la conexión con impresoras Bluetooth para emitir *tickets* de venta.
- Integrar la cámara del dispositivo móvil como lector de código de barras.
- Permitir la interacción entre varios empleados de un mismo negocio.
- Permitir la administración de clientes.

De forma complementaria se llevaron a cabo entrevistas a administradores y encargados de 15 PyMES de ropa y calzado, el listado de negocios entrevistado se encuentra en el Anexo H. El formato de la entrevista realizada se encuentra en el Anexo A y los resultados de la misma se presentan tabulados a continuación.

[16] *Gadget*: es un dispositivo electrónico destinado a una función específica.

Tabla 2.1 Tabulación de entrevistas (Parte I de II)

N.	Pregunta	Respuesta(%)
1	**¿Considera útil que usted pueda conocer la información de su negocio en cualquier momento y sin necesidad de estar presente en el?**	
	Si	100.00
	No	0.00
2	**Aplicaciones móviles como Facebook, hacen uso de un correo y una clave personal para el ingreso a la aplicación. De las siguientes combinaciones. ¿Cuál preferiría como acceso a la aplicación?**	
	Correo electrónico y clave personal	40.00
	Número de teléfono celular y clave personal	20.00
	Nombre de usuario y clave personal	20.00
	N. de identificación (Cédula/Pasaporte) y clave personal	20.00
3	**¿Le gustaría que la aplicación recuerde los datos con los que ingresa a la misma?**	
	Si	80.00
	No	20.00
4	**¿Qué funcionalidades desearía que esta aplicación tuviera?**	
	Historial de ventas por empleado	66.67
	Historial de ventas por fechas	73.33
	Registro de clientes	80.00
	Impresión de comprobantes de ventas	80.00
	Categorización de productos	26.67
5	**¿Qué datos sería de utilidad conocer de sus empleados?**	
	Nombres	100.00
	Cédula/Pasaporte	100.00
	País de residencia	20.00
	Ciudad de residencia	13.33
	Dirección domiciliaria	60.00
	Número de teléfono local/celular	73.33
	Correo electrónico	33.33
	Cargo	0.00
6	**¿Qué datos sería de utilidad conocer de sus clientes?**	
	Nombres	100.00
	Cédula/Pasaporte/RUC	66.67
	País de residencia	6.67
	Ciudad de residencia	13.33
	Dirección domiciliaria	66.67
	Número de teléfono local/celular	93.33
	Correo electrónico	46.67

Tabla 2.1 Tabulación de entrevistas (Parte II de II)

N.	Pregunta	Respuesta(%)
7	**De los siguientes parámetros, ¿cuáles consideraría necesario para registrar su producto?**	
	Tallas	93.33
	Imagen	86.67
	Cantidad	86.67
	Categoría	33.33
	Precio de compra	20.00
	Precio de venta	100.00
8	**Si la aplicación generará comprobantes de venta, ¿qué datos le gustaría que se muestre?**	
	Nombre del local	100.00
	RUC del local	93.33
	Dirección del local	93.33
	Número de teléfono del local	93.33
	Nombre del cliente	73.33
	N. de identificación del cliente	66.67
	Dirección del cliente	93.33
	Número de teléfono el cliente	93.33
	Nombre del producto	100.00
	Detalles monetarios (valor con IVA, valor sin IVA) de cada producto	80.00
	Detalles monetarios (valor con IVA, valor sin IVA) del total de la venta	86.67
	Otros	13.33
9	**¿Qué parámetros emplearía para la búsqueda de una venta?**	
	Fecha de realización de venta	73.33
	Número de comprobante de venta	73.33
	Nombre del cliente	80.00
	Otros	0.00
10	**¿Qué parámetros emplearía para la búsqueda de un producto?**	
	Nombre del producto	73.33
	Categoría	46.66
	Otros	0.00

La pregunta 1 busca conocer la percepción de utilidad de este prototipo de punto de venta por parte de los usuarios, para con ello validar el desarrollo de este Proyecto Técnico. Con base en las respuestas obtenidas se contabilizó un porcentaje de aprobación del 100 por ciento y con ello el desarrollo de ese Proyecto Técnico queda justificado.

La pregunta 2 buscó definir los datos necesarios para el acceso al prototipo ante lo cual la opción más aceptada fue la de correo y contraseña. Complementariamente la pregunta 3 buscó validar el almacenamiento de los datos de acceso en el prototipo ante lo cual se tiene un 100 por ciento de aceptación.

La pregunta 4 fue planteada con el objetivo de validar las ideas centrales referentes a la administración de clientes y uso de impresoras para la emisión de *tickets*. Las dos opciones previamente expuestas fueron aceptadas y además se puso en evidencia la necesidad que este prototipo cuente con funciones secundarias como reportes de ventas y categorización de productos.

Las preguntas 5, 6 y 7 tuvieron el objetivo de permitir modelar las entidades de empleado, cliente y productos respectivamente. Como parámetro general de aprobación se consideraron aquellas opciones con más del 30 por ciento. Con base en las respuestas, tanto la entidad `cliente` como la de `empleado` contarán con los siguientes atributos: `nombres`, `cedula/pasaporte`, `telefono/celular`, `direccion` y `correo`; por su parte la entidad `producto` contará con los siguientes atributos: `cantidad`, `categoria` y `precio de venta`, para las tallas se creará una entidad de `tallas_x_producto`. Los atributos antes descritos son la base de cada entidad, pero estos pueden incrementar durante el desarrollo del prototipo, los atributos definitivos de cada entidad se presentarán en el diagrama entidad relación.

La pregunta 8 tuvo el objetivo de estructurar el *ticket* que se va a emitir al finalizar una venta, se mantuvo el criterio de aceptación de las preguntas anteriores, con lo que todas las opciones planteadas fueron aceptadas.

Las preguntas 9 y 10 tuvieron relación a los parámetros de búsqueda para los registros de ventas y productos respectivamente. Para el listado de ventas se implementarán filtros por fecha de realización de venta, número de comprobante y nombre del cliente. Para el listado de productos se implementarán filtros por categoría y nombre del producto.

2.1.1 HISTORIAS DE USUARIO

Los resultados de las entrevistas realizadas junto con el análisis previo de aplicaciones permitieron identificar los requerimientos desde la perspectiva de usuario y con ellos se realizaron las historias de usuario presentadas en la Tabla 2.2.

Tabla 2.2 Historias de usuario

ID	Título	Descripción
HU01	Administración de clientes	Como usuario se necesita administrar los datos de clientes para con ellos asociar una venta.
HU02	Categorización de productos	Como usuario es necesario categorizar los productos para llevar un control más sencillo del inventario y buscar más fácilmente dentro del mismo.
HU03	Generación de reportes de venta	Como administrador es necesario conocer el monto total de ventas diarias y mensuales, así como el volumen de ventas por empleado en los mismo periodos.
HU04	Impresión de *tickets* de venta	Como usuario es necesario emitir un *ticket* de venta como constancia de compra y entregarlo al comprador.
HU05	Lectura de código de barras de productos	Como usuario es necesario contar con un lector de código de barras para determinar el producto que se va a vender.
HU06	Sincronización de inventario	Como usuario es necesario sincronizar la cantidad de productos en inventario tras cada venta para evitar error en la disponibilidad de productos en ventas futuras.

2.1.2 REQUERIMIENTOS

Las historias de usuario mostradas en el apartado previo permitieron definir los requerimientos del prototipo que se alinean con el alcance de este Proyecto Técnico.

1. Permitir la administración de datos de negocio.
2. Permitir la administración de datos de usuario (administrador, vendedor).
3. Iniciar sesión.
4. Cerrar sesión.
5. Mostrar un menú con funciones específicas en referencia al perfil de usuario (administrador, vendedor).
6. Permitir la administración de datos de categorías.
7. Permitir la administración de datos de productos.
8. Permitir la administración de tallas por productos.
9. Permitir la búsqueda de productos por categoría y nombre del producto.
10. Permitir la administración de datos de clientes.
11. Permitir la administración de ventas.
12. Permitir la administración de pagos de una venta.
13. Permitir la búsqueda de productos por fecha de realización, nombre del cliente y número de *ticket*.

14. Emitir *tickets* de venta con la ayuda de impresoras térmicas Bluetooth.
15. Hacer uso de la cámara del dispositivo móvil como lector de código de barras.
16. Generar reportes de ventas totales y de ventas por usuarios en un periodo de tiempo deseado.

2.1.3 DEFINICIÓN DE TAREAS

Cada uno de los requerimientos listados previamente fueron desglosados en tareas específicas y simples lo que se muestra en la Tabla 2.3.

Tabla 2.3 Tareas por requerimientos (Parte I de II)

	Requerimiento	Tarea	
1	Permitir la administración de datos de negocio	1	Registrar datos de negocio
		2	Mostrar los datos de negocio
		3	Editar los datos del negocio, solo administrador
		4	Eliminar datos de un negocio, solo administrador
2	Permitir la administración de datos de usuario (administrador, vendedor)	5	Registrar datos de usuario
		6	Mostrar los datos del usuario
		7	Editar datos de usuario
		8	Eliminar datos de usuario
3	Iniciar sesión	9	Autenticar mediante correo y contraseña
		10	Guardar datos de inicio de sesión en el lado del cliente
		11	Obtener token de registro del servicio FCM (Firebase Cloud Messaging)
		12	Registrar token en la base de datos
4	Cerrar sesión	13	Eliminar datos de inicio de sesión del lado del cliente
		14	Borrar token FCM registrado
5	Mostrar un menú	15	Mostrar opciones de menú en función del perfil de usuario
6	Permitir la administración de datos de categorías	16	Mostrar categorías
		17	Registrar una categoría
		18	Editar datos de categoría
		19	Eliminar datos de categoría
7	Permitir la administración de datos de productos	20	Mostrar productos
		21	Registrar un producto
		22	Editar datos de producto
		23	Eliminar datos de producto
		24	Sincronización de inventario

Tabla 2.3 Tareas por requerimientos (Parte II de II)

	Requerimiento	Tarea	
8	Permitir la administración de tallas por productos	25	Mostrar tallas de un producto
		26	Registrar una talla
		27	Editar datos de una talla
		28	Eliminar datos de una talla
9	Permitir la búsqueda de productos por categoría y nombre del producto	29	Buscar productos por categoría asociada
		30	Buscar productos por su nombre
10	Permitir la administración de datos de clientes	35	Mostrar listado de clientes
		36	Registrar un cliente
		37	Editar datos de un cliente
		38	Eliminar datos de un cliente
11	Permitir la administración de ventas	39	Mostrar listado de productos
		40	Seleccionar un producto, la talla del mismo y la cantidad deseada para agregarlos al carrito de compras
		41	Seleccionar cliente
		42	Seleccionar método de pago de una venta
		43	Registrar venta
		44	Mostrar listado de ventas
12	Permitir la administración de pagos de una venta	45	Mostrar listado de pagos
		46	Registrar un pago
		47	Modificar pago
13	Permitir la búsqueda de productos por fecha de realización, nombre del cliente y número de *ticket*	48	Permitir la búsqueda de productos por fecha de realización, nombre del cliente y número de *ticket*
14	Emitir *tickets* de venta con la ayuda de impresoras térmicas Bluetooth.	49	Conectar vía bluetooth una impresora térmica
		50	Imprimir *ticket* al finalizar una venta
15	Hacer uso de la cámara del dispositivo móvil como lector de código de barras	51	Configurar la cámara como lector de códigos de barra
		52	Filtrar productos por código de barras
16	Generar reportes de ventas totales y por usuarios en un periodo de tiempo	53	Generar reportes de venta totales
		54	Generar reportes de venta por usuarios

2.1.4 DEFINICIÓN DE SPRINTS

Una vez analizado cada requerimiento, se dividieron las tareas conforme se estipula en la metodología Scrum, con esto se generaron los tableros de cada *sprint* los cuales se muestra en la Tabla 2.4.

Tabla 2.4 Requerimientos por *sprint*

Sprint	Nombre	Requerimiento	
***Sprint* 1**	Registro e inicio de sesión	1	Permitir la administración de datos de negocio
		2	Permitir la administración de datos de usuario (administrador, vendedor)
		4	Cerrar sesión
		5	Mostrar un menú con funciones en referencia al perfil de usuario (administrador, vendedor)
***Sprint* 2**	Manejo de productos	6	Permitir la administración de datos de categorías
		7	Permitir la administración de datos de productos
		8	Permitir la administración de tallas por productos
		9	Permitir la búsqueda de productos por categoría y nombre del producto
***Sprint* 3**	Ventas I	10	Permitir la administración de datos de clientes
		11	Permitir el registro de una venta
***Sprint* 4**	Ventas II	12	Permitir la administración de pagos de una venta
		13	Permitir la administración de una venta
		14	Permitir la búsqueda de ventas por fecha de realización, nombre de cliente y número de *ticket*
***Sprint* 5**	Conectividad externa	15	Permitir la conexión de impresoras térmicas para la emisión de *tickets* de venta
		16	Hacer uso de la cámara del dispositivo móvil como lector de código de barras
***Sprint* 6**	Reportes	17	Generar reportes de ventas generales en un periodo de tiempo
		18	Generar reportes de venta por usuarios en un periodo de tiempo

En relación a la metodología propuesta, se usó la herramienta en línea iceScrum [30] sobre la cual se creó el tablero Scrum con las columnas `ToDo`, `Doing` y `Done`, a manera de ejemplo las tareas correspondientes al *sprint* 1 se muestran en la Figura 2.1. La herramienta iceScrum permite de manera muy sencilla cambiar el estado de las tareas

conforme al avance del proyecto asi como añadir notas referentes a problemas de desarrollo o datos de relevancia para tareas futuras.

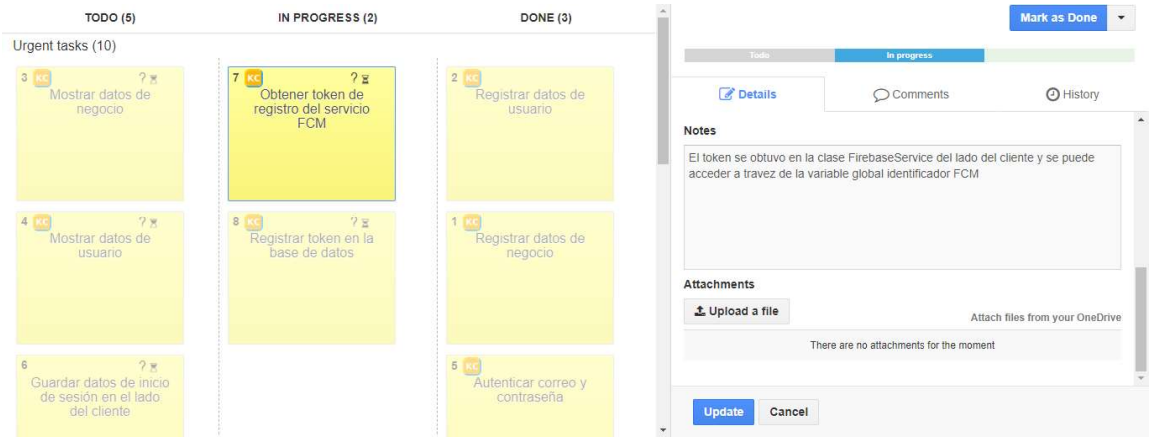


Figura 2.1 Tablero Scrum en iceScrum

2.1.5 ARQUITECTURA DEL PROTOTIPO

El prototipo fue desarrollado con una arquitectura de tipo cliente-servidor, como se muestra en la Figura 2.2. El prototipo cuenta con tres componentes:

- Base de datos: donde se almacena la información.
- Cliente Android: donde se genera la información.
- Web API: donde se procesa la información proveniente del cliente Android y permite que sea almacenada en la base de datos.

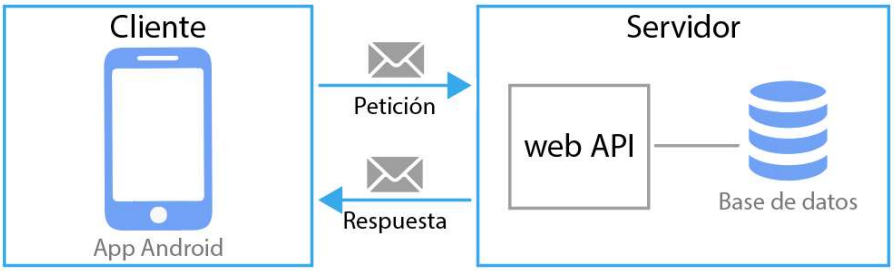


Figura 2.2 Arquitectura del prototipo

2.2 DISEÑO CONCEPTUAL

Con las historias de usuario detalladas, así como las tareas embebidas en cada una de ellas se inició el diseño de la base de datos necesaria para atender los requerimientos. Para esto se hizo uso de un esquema conceptual y de un esquema lógico.

El esquema conceptual se lo plasmo a través del diagrama E-R representado en la Figura 2.3, de este diagrama se destacan las relaciones `negocio - usuario` y `usuario - cargo` con cardinalidad `N:N` y `1:N` respectivamente, pues el prototipo contempla soporte para los escenarios donde un usuario puede trabajar en más de un negocio y en cada uno de ellos ostente un cargo diferente, la relación `negocio - cliente` fue planteada con la misma idea de la relación anterior dado que un cliente puede frecuentar varias tiendas y debería estar registrado en cada una de ellas, la relación `producto - talla` se planteó con cardinalidad `N:N` para soportar el registro de una lista de tallas por cada producto y debido al hecho de que estas tallas se reutilizarán en nuevos productos para evitar registros de talla duplicados.

Dado que una base de datos no puede contener relaciones muchos a muchos (`N:N`), las relaciones previamente descritas serán posteriormente reemplazadas por relaciones 1 a muchos (`1:N`) con el uso de tablas intermedias.

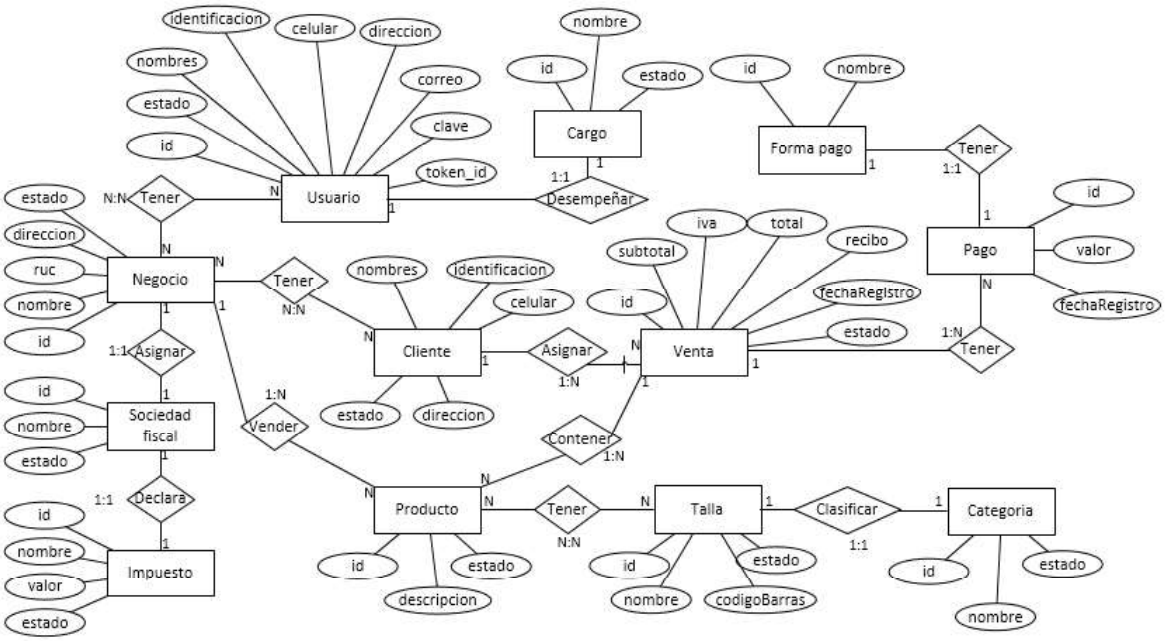


Figura 2.3 Diagrama E-R

Para el esquema lógico se generó el diagrama relacional de base de datos que se presenta en la Figura 2.4. Donde se plasma las entidades y atributos definitivos que contendrá la base de datos, el *script* para generar la base de datos se encuentra en el Anexo B y a continuación, se listan las entidades creadas y una breve descripción de su funcionalidad.

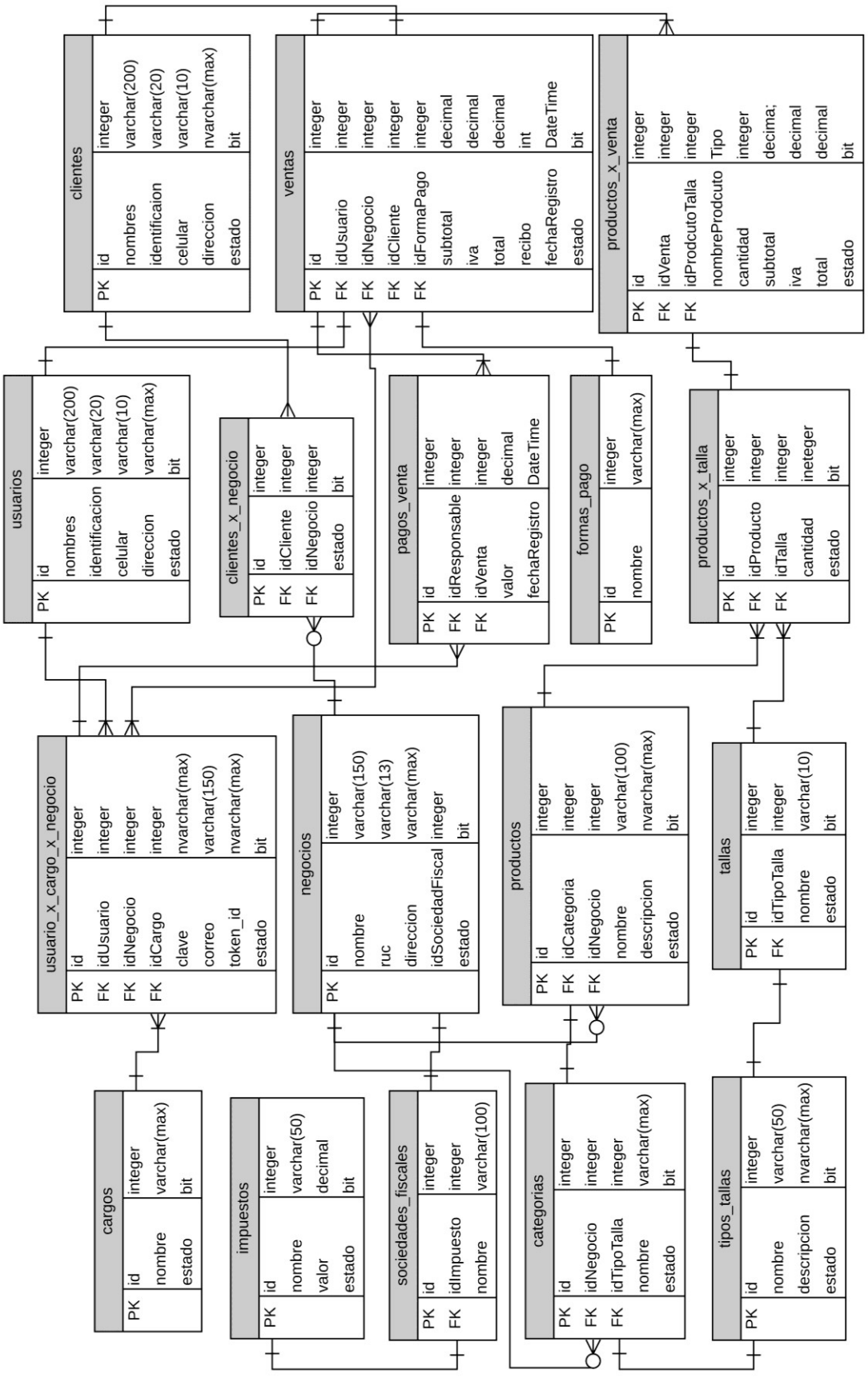


Figura 2.4 Diagrama relacional de base de datos

- `impuestos`: esta entidad permite almacenar los valores de los impuestos (0%,12%) que el administrador de un local comercial está obligado a declarar sobre sus ventas, los registros aquí alojados son considerados datos maestros, es decir que solo está permitida la lectura desde el cliente Android y su administración está reservada para el encargado del mantenimiento del prototipo.
- `sociedades_fiscales`: al igual que la entidad previa, almacena registros maestros de las representaciones fiscales sobre las que un negocio puede identificarse. El prototipo contará con 2 sociedades fiscales: "Persona natural" el cual está asociado con un valor de impuesto del 12% y "Artesano" asociado con un impuesto del 0%
- `cargos`: permite manejar los cargos con los que el prototipo trabajará. Así, se tiene 2 cargos: Administrador y Vendedor.
- `usuarios`: sobre esta entidad se almacenarán los datos de un usuario (Administrador, Vendedor)
- `negocios`: sobre esta entidad se almacenarán los datos de un negocio.
- `clientes`: sobre esta entidad se almacenarán los datos de un cliente.
- `usuarios_x_cargos_x_negocios`: es una entidad auxiliar y se creó para eliminar la relación `N:N` que existe entre las entidades `usuarios` y `negocios`.
- `clientes_x_negocio`: es una entidad auxiliar y se creó para eliminar la relación `N:N` que existe entre las entidades `clientes` y `negocios`.
- `productos`: sobre esta entidad se almacenarán los datos básicos de un producto como su nombre y descripción.
- `tallas`: es una entidad que aloja datos maestros, sobre esta se representarán los datos de cada talla que un producto pueda tener. El prototipo contará con registros para tallas numéricas de la 0 a la 46 y para tallas alfabéticas XS, S, M, L, XL y XXL.
- `tipos_tallas`: es una entidad creada con el fin de brindar cierto nivel de organización en el registro de productos, un registro de esta entidad representa a una lista de tallas similares. El prototipo contará con los registros; "Numérico bebe" que agrupará a las tallas de la 0 a la 18. "Numérico niño" que agrupara a las tallas de la 20 a la 34, "Numérico adulto" para agrupar las tallas de las 36 en adelante y "Alfabético" para agrupar las tallas XS, S, M, L, XL y XXL.
- `categorías`: sobre esta entidad se almacenarán los datos de una categoría, esta entidad a su vez está asociada a una entidad `tipos_tallas`.

- productos_x_talla: es una entidad auxiliar y se creó para eliminar la relación N:N que existe entre las entidades productos y tallas.
- ventas: sobre esta entidad se almacenarán los datos básicos de una venta como el valor total de la venta, el valor del impuesto generado y el número de recibo asignado, esta entidad se relaciona con las entidades usuarios_x_cargos_x_negocios que permite conocer quien realizó la venta, clientes_x_negocio para conocer los datos del cliente y productos_x_ventas para conocer los productos vendidos.
- productos_x_ventas: sobre esta entidad se almacenará la lista de productos entregados en cada venta.
- formas_pago: almacena registros maestros de las formas de pago que el prototipo soporta. Contará con 2 registros: "Contado" y "Crédito".
- pagos_venta: sobre esta entidad se almacenarán los detalles de los pagos realizados de una venta.

Por otro lado, para describir la estructura de la web API así como del cliente Android, las clases que cada uno contiene, los métodos y relaciones entre las mismas se realizaron 3 diagramas de clases.

El primer diagrama contiene las clases asociadas a las tablas de la base de datos que fueron generadas por Entity Framework en la web API y de forma manual en el cliente Android y que son comunes para ambos componentes.

Así, por ejemplo, si en la base de datos se tiene una tabla categorías, en la web API se tendrá un archivo que contiene la clase, llamado categorias.cs, mientras que en el lado del cliente Android el archivo se llamará categorias.java y para ambas clases se tendrán las propiedades id, idNegocio y idTipoTalla de tipo integer, además contarán con la propiedad estado cuyo tipo de dato se representa como un objeto bool (bit en la base de datos) y nombre de tipo string (varchar en base de datos) como se muestra en la Figura 2.5. En la web API al asociar la base de datos, se generó, además, un archivo que contiene la clase de contexto, llamado tesisPoliEntities.cs que permite establecer una sesión con la base de datos y manipular la información de cualquiera de las tablas existentes.

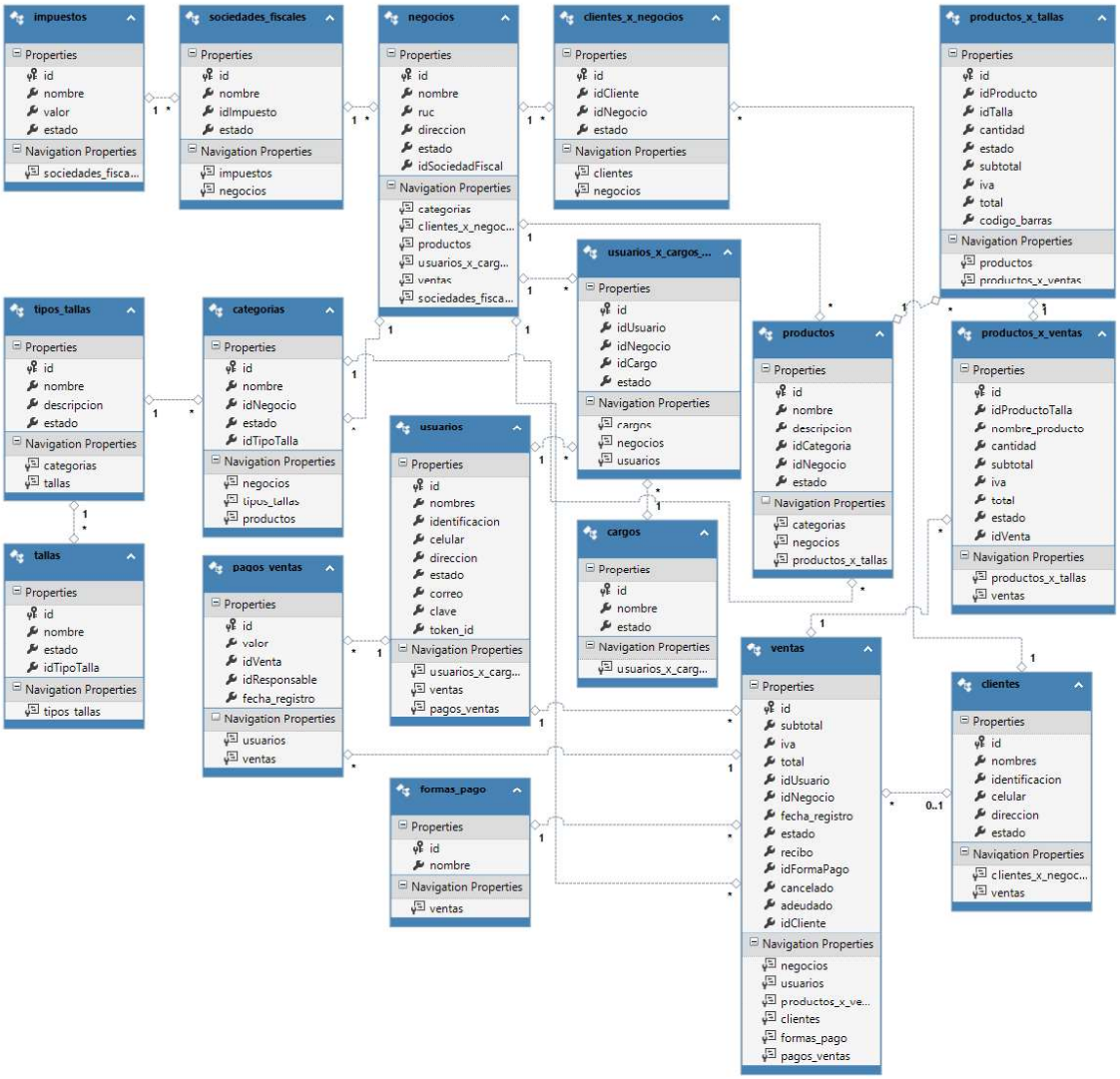


Figura 2.5 Diagrama de clases del modelo Entity Framework

El segundo diagrama se presenta en la Figura 2.6 y contiene las clases controladoras de la web API que permiten atender las solicitudes realizadas del lado del cliente Android. El objetivo de las clases se detalla a continuación:

- `CargoController`: Contiene los métodos de acción para leer los registros de cargos para los usuarios del prototipo.
- `CatalogoController`: Contiene un solo método para obtener los productos que se muestran en la pantalla principal del cliente Android.
- `CategoriaController`: Contiene los métodos para leer y manipular la información de la tabla `categorías`. Los métodos incluyen la creación (`POST`), edición (`PUT`) y eliminación (`DELETE`) de registros.

- `ClienteController`: Contiene los métodos para leer y manipular la información de la tabla `clientes`. Los métodos incluyen la creación (`POST`), edición (`PUT`) y eliminación (`DELETE`) de registros.
- `DetalleVentaController`: Contiene un solo método para obtener el listado de productos entregados en una venta.
- `FormaPagoController`: Contiene los métodos de acción para leer los registros de formas de pago establecidas en el alcance de este Proyecto Técnico.
- `NegocioController`: Contiene los métodos para leer y manipular la información de la tabla `negocios`. Los métodos incluyen la creación (`POST`), edición (`PUT`) y eliminación (`DELETE`) de registros.
- `PagoController`: Contiene los métodos para leer y manipular la información de la tabla `pagos_venta`. Los métodos incluyen la creación (`POST`), edición (`PUT`) y eliminación (`DELETE`) de registros.
- `ProductoController`: Contiene los métodos para leer y manipular la información de la tabla `productos`. Los métodos incluyen la creación (`POST`), edición (`PUT`) y eliminación (`DELETE`) de registros.
- `ProductoTallaController`: Contiene los métodos para leer y manipular la información de la tabla `productos_x_tallas`. Los métodos incluyen la creación (`POST`), edición (`PUT`) y eliminación (`DELETE`) de registros.
- `ReporteController`: Contiene un solo método para obtener los datos de reportes de tipo general y por usuarios.
- `SociedadController`: Contiene los métodos de acción para leer los registros de sociedades fiscales del prototipo.
- `TallaController`: Contiene los métodos de acción para leer los registros de tallas y tallas por categorías.
- `TipoTallaController`: Contiene los métodos de acción para leer los registros de tipos de tallas.
- `UsuarioController`: Contiene los métodos para leer y manipular la información de la tabla `usuarios` y `usuarios_x_cargos_x_negocios`. Los métodos incluyen la creación (`POST`), edición (`PUT`) y eliminación (`DELETE`) de registros.
- `UsuarioDetalleController`: Contiene métodos complementarios para cambiar la clave de usuarios, generación de una nueva vía correo electrónico entre otros.

- `VentaController`: Contiene los métodos para registrar y eliminar información de la tabla `ventas`.

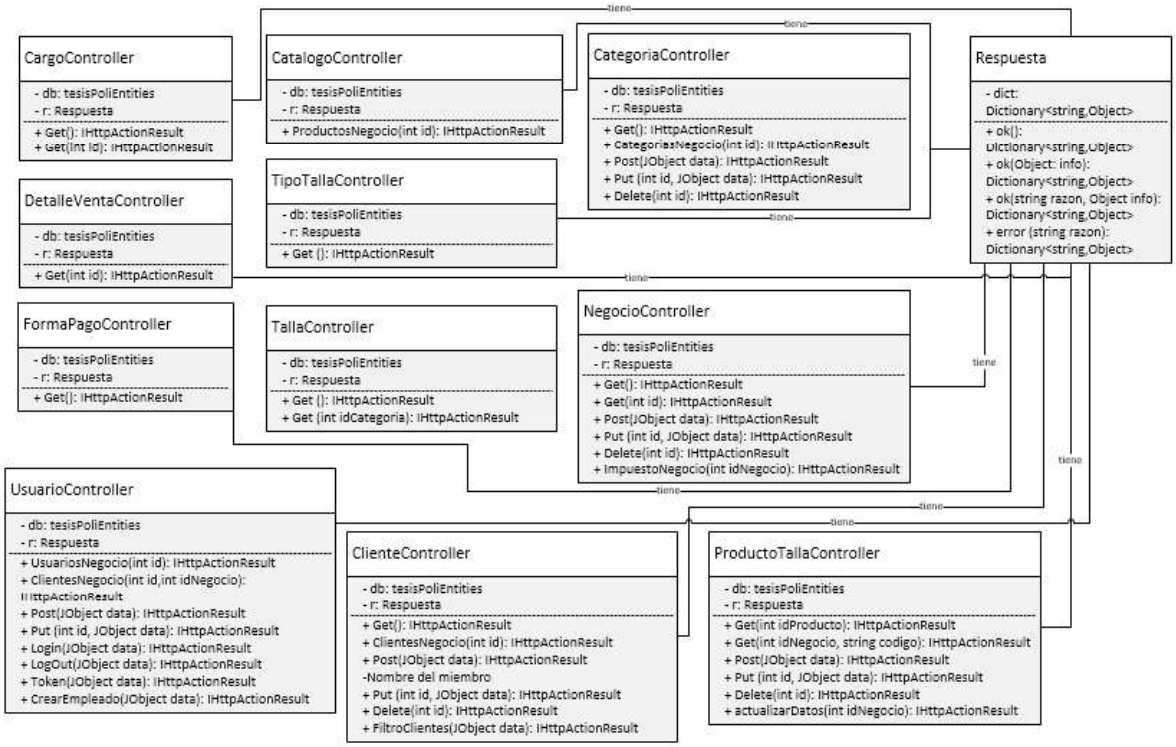


Figura 2.6 Diagrama de clases web API (Parte I de II)

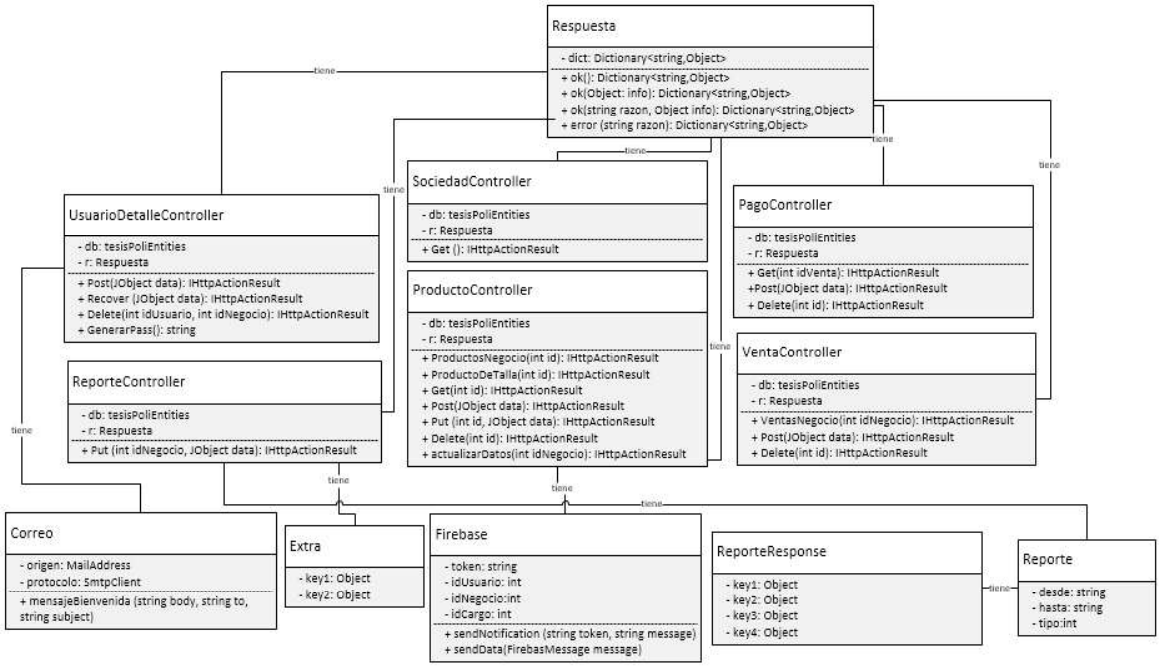


Figura 2.6 Diagrama de clases web API (Parte II de II)

En el diagrama también se incluyen clases adicionales:

- `Respuesta`: Ayuda a establecer un formato común de intercambio de información entre la web API (lado del servidor) y cliente Android.
- `Firebase`: Sus métodos permiten enviar información hacia el servicio FCM para sincronizar el inventario entre varios dispositivos móviles.
- `Correo`: Permite la conexión con un servidor de correo, así como el envió de mensajes electrónicos.
- `Extra`: Clase destinada a extender información que se envía hacia el cliente Android.
- `Reporte`: Clase para definir parámetros (rango de fechas y tipo de reporte) para la generación de reportes.
- `ReporteResponse`: Clase para representar un reporte.

El código completo de la web API se encuentra en el Anexo C.

El tercer diagrama se presenta en la Figura 2.7 y contiene las clases creadas en el cliente Android para controlar la presentación de información en las diferentes vistas, el control de las interacciones del usuario en el cliente Android y el intercambio de información con el lado del servidor. De forma general, el diagrama presenta clases como `TallaFragment`, `ProductoFragment`, `VentaFragment` y `ClienteFragment` las cuales representan a las vistas con las listas de tallas, productos registrados, venta realizadas y clientes respectivamente, los métodos presentes en estas clases permiten solicitar información al lado del servidor, manipular dicha información, presentarla gráficamente en componentes visuales de tipo `RecyclerView` y eliminar un registro deseado.

También se encuentran clases como `TallaDetalleFragment`, `ProductoDetalleFragment`, `VentaDetalleFragment` y `ClienteDetalleFragment` las cuales controlan las acciones sobre las vistas para el registro y modificación de datos de tallas, productos y la presentación del detalle de clientes y productos vendidos respectivamente. Estas vistas están compuestas por cuadros de texto (`EditText`), etiquetas (`TextView`), listas de selección (`Spinner`), botones (`Button`) entre otros.

El código completo del cliente Android se encuentra en el Anexo D

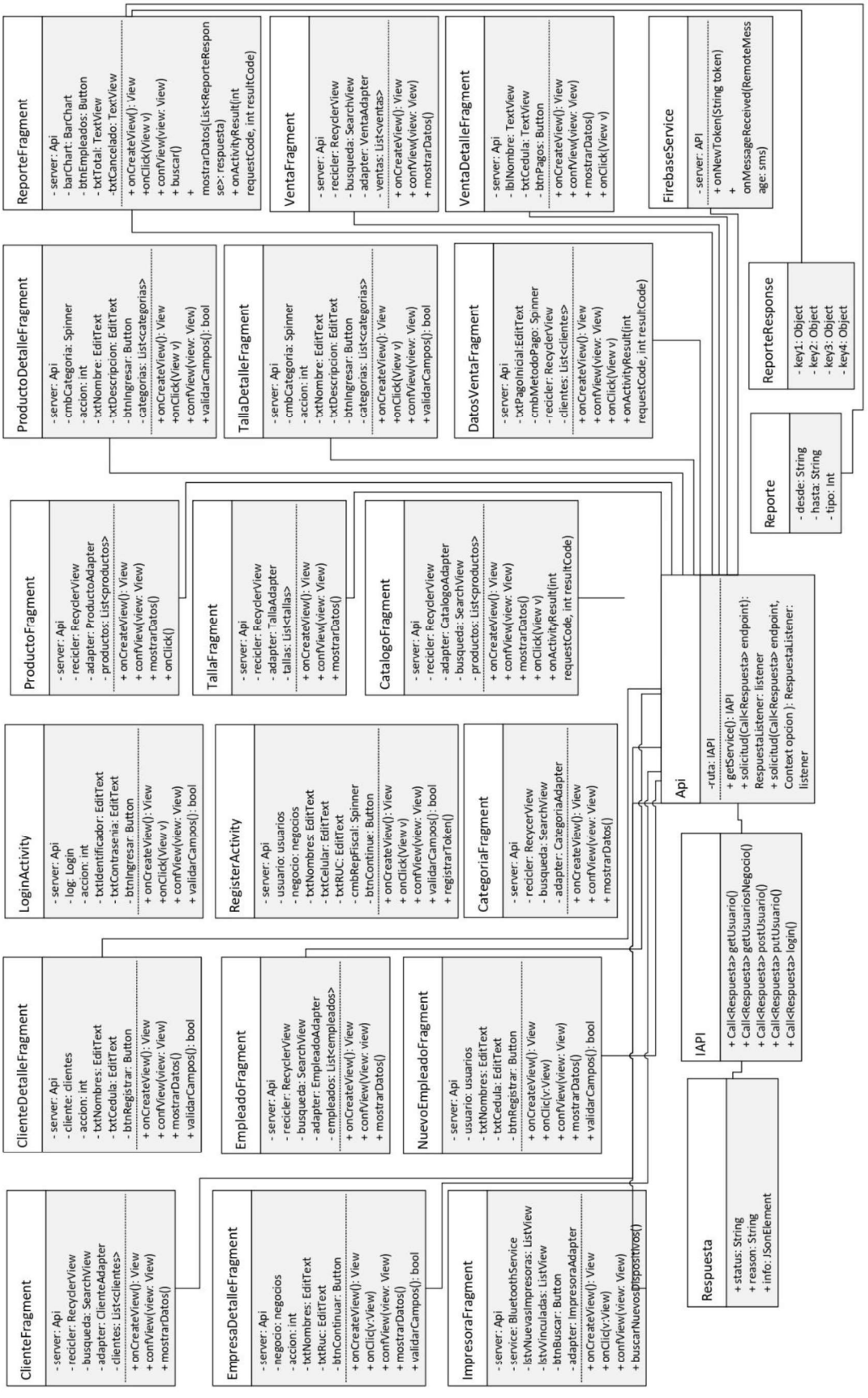


Figura 2.7 Diagrama de clases cliente Android

Al igual que en el lado del servidor, se cuentan con clases adicionales que complementan las funciones del lado del cliente como se detallan a continuación:

- `Respuesta`: Ayuda a establecer un formato común de intercambio de información entre la web API (lado del servidor) y cliente Android.
- `FirebaseService`: Sus métodos permiten receptar mensajes desde el servicio FCM para sincronizar el inventario en un dispositivo móvil.
- `Api`: Permite la conexión e intercambio de información con el lado del servidor.
- `IAPI`: Permite definir *endpoints* de conexión con el servidor.
- `Reporte`: Clase para definir parámetros (rango de fechas y tipo de reporte) para la generación de reportes.
- `ReporteResponse`: Clase para representar un reporte.

2.3 HERRAMIENTAS DE DESARROLLO

A continuación, se detallan las características de las herramientas utilizadas para el desarrollo de este Proyecto Técnico.

Servidor web: Para levantar el servidor web se hizo uso de la herramienta en línea Somee [30]. Para el uso de esta herramienta solo es necesario crear una cuenta con una cuenta de correo válida. En este servidor se alojará tanto la base de datos como la web API.

Una vez dentro del sistema, la creación de la base de datos se realiza seleccionando la opción *MS SQL> Databases* del panel izquierdo de la pantalla y a continuación se selecciona la primera opción (*Free hosting package*). Acto seguido, se presenta la pantalla de configuración, la cual se muestra en la Figura 2.8, donde se especifica el nombre de la base de datos, las credenciales de acceso a la misma las cuales por defecto son las mismas que para el acceso a la plataforma y la versión del servidor deseado.

Figura 2.8 Creación de base de datos en Somee

Así mismo, para la creación del sitio web sobre el cual se alojará el código de la web API se selecciona la opción *websites* del panel izquierdo de la pantalla principal. En el panel derecho se mostrará una lista de opciones para crear el sitio de la cual se selecciona la primera opción (*Free hosting package*).

A continuación, se presenta la pantalla de configuración, la cual se muestra en la Figura 2.9, donde se define el subdominio del sitio, la versión del servidor IIS[17] entre otros parámetros básicos.

Figura 2.9 Creación del sitio web en Somee

De la creación del sitio web se destaca el subdominio seleccionado con el cual se forma la URL `http://titulacionpoli.somee.com/`, la cual, a su vez, será la dirección de referencia de los *endpoints*[18] que se usarán en el lado del cliente.

Web API: El desarrollo de este componente fue realizado sobre el IDE Microsoft Visual Studio Community en su versión 15.9.3. El *framework* .NET utilizado para este prototipo fue el 4.7 mientras que el *framework* web API utilizado fue la versión 5.2.3. Para la creación de un proyecto con el *framework* web API dentro del IDE Visual Studio se selecciona la pestaña *File* luego la opción *New* y posteriormente la opción *Project.* En la pantalla que aparece para la selección del tipo de proyecto se selecciona la opción *Web* del panel izquierdo y finalmente el tipo ASP.NET *Web Application*. En la configuración del proyecto se selecciona la opción Web API. Por defecto se marcan las casillas MVC y web API como

[17] IIS (*Internet Information Service*): es un conjunto de servicios para plataformas Windows que permiten transformar a un PC en un servidor web ya sea en internet o una intranet.

[18] *Endpoint*: es una URL de una API que responde a una petición generada.

referencias del proyecto, esto permite desarrollar un servicio REST y una página web en el mismo proyecto.

Para la publicación de la web API en el servidor web Somee, se emplea el subdominio definido en la creación de dicho servidor, las credenciales de usuario y FTP como método de transferencia de datos, como se muestra en la Figura 2.10.

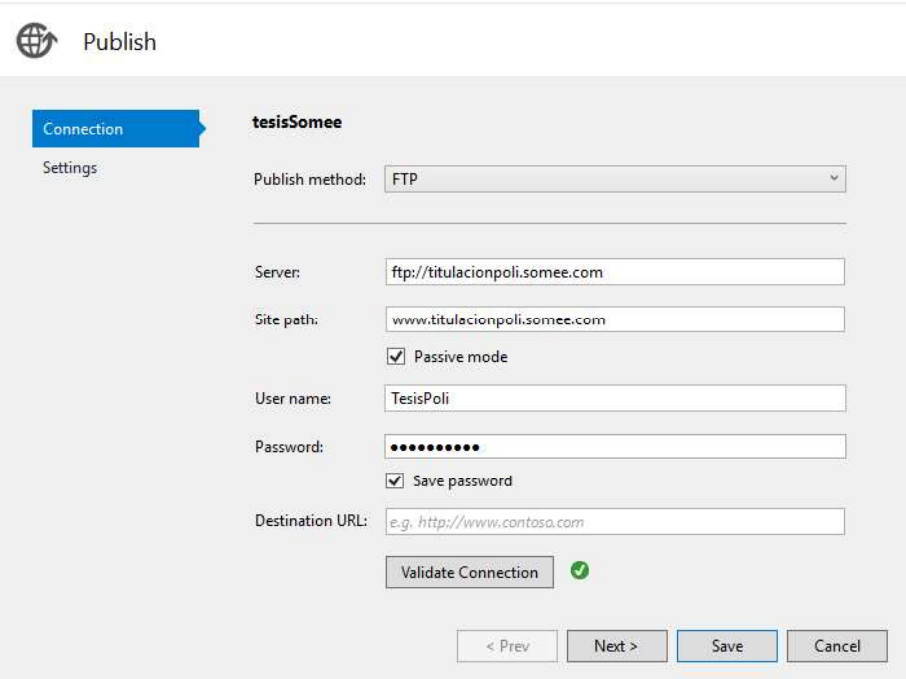


Figura 2.10 Parámetros de publicación de web API

Cliente Android: Para el desarrollo del cliente Android del prototipo de punto de venta se usó el IDE Android Studio en su versión 3.0.1.

Para la creación de un proyecto en Android Studio es necesario seleccionar la opción *Start a new Android Studio Project* en la pantalla inicial, esto lanza una secuencia de pantallas para la configuración inicial del proyecto.

En la primera pantalla se da un nombre al proyecto y al paquete que identificara a la aplicación, en la segunda pantalla se define el *target* de la aplicación, por defecto se tienen los dispositivos móviles con sistema operativo Android 4.1 (*Jelly Bean*) o superiores.

En la tercera pantalla se eligió la plantilla *Empty Activity* para evitar código autogenerado que pueda provocar conflictos en el desarrollo del prototipo. La ultima pantalla permite

nombrar la actividad raíz de la aplicación, por convención esta lleva el nombre de `MainActivity`.

Todos estos parámetros definidos durante la configuración inicial pueden ser modificados en los archivos `AndroidManifest.xml` y `buid.gradle` según corresponda.

2.4 SPRINT 1: REGISTRO E INICIO DE SESIÓN

El objetivo de este *sprint* es levantar las interfaces de navegación iniciales del prototipo de punto de venta. Al finalizar las tareas, un usuario administrador podrá registrar sus datos personales, así como los datos de su negocio para con ellos ingresar al prototipo. También podrá actualizar datos personales, visualizar las opciones disponibles en el prototipo, y crear cuentas para sus vendedores.

2.4.1 ESQUEMA GRÁFICO

En este *sprint* se han considerado los *sketckes* de la Figura 2.11, de izquierda a derecha se presenta el *sketch* para el inicio de la aplicación desde la cual un usuario puede dirigirse al registro (*sketch* 2) o al iniciar sesión a través de su correo y contraseña como se muestra en el *sketch* 3. También se hará uso de una actividad como menú principal (*sketch* 4) la cual permitirá al usuario navegar entre las diferentes opciones del prototipo. Las opciones mostradas corresponden al perfil de un usuario administrador mientras que para el perfil de un vendedor se limita a las opciones “Vender”, “Clientes”, “Historial de ventas”. “Impresoras” y “Cerrar sesión”.

Figura 2.11 *Sketches* de registro e inicio de sesión

Si los procesos de registro o de inicio de sesión son exitosos se inicia la actividad con el menú como se muestra en la Figura 2.12 mientras que si existen inconsistencias entre los datos ingresados y la base de datos se desplegará un mensaje de error como se ejemplifica en la Figura 2.13.

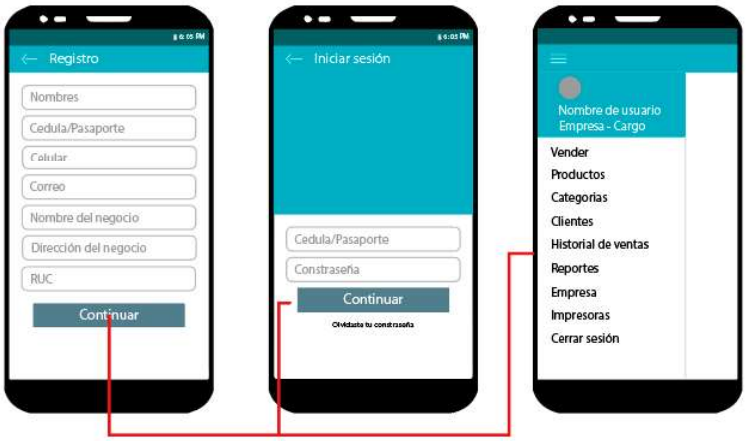


Figura 2.12 *Wireframe* de registro e inicio de sesión correcto

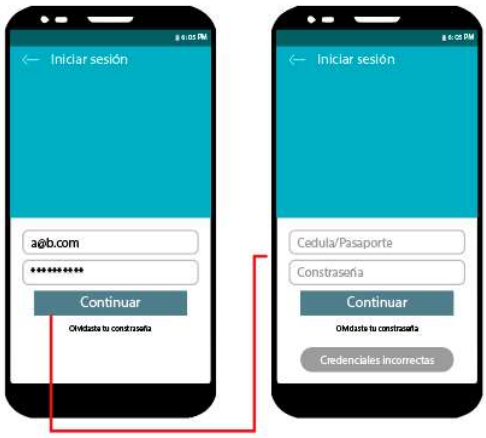


Figura 2.13 *Wireframe* de inicio de sesión erróneo

Este proceso de notificación de errores se mantendrá como esquema general en todas las vistas del cliente Android. Finalmente, en este *sprint* se implementaron las vistas de edición de datos de usuario y de negocio con el mismo estilo gráfico que el formulario de registro (*sketch* 1, Figura 2.12).

2.4.2 ESQUEMA CONCEPTUAL

Para la administración de datos de usuario y negocio se definieron las entidades `impuestos`, `sociedades_fiscales`, `cargos`, `usuarios`, `negocios` y `usuarios_x_cargos_x_negocios` en la base de datos.

En el lado del servidor se crearon las clases referente a los controladores (ver Figura 2.6) y clases auxiliares como `Correo`, `Login` y `Respuesta` las cuales se muestra en la Figura 2.14.

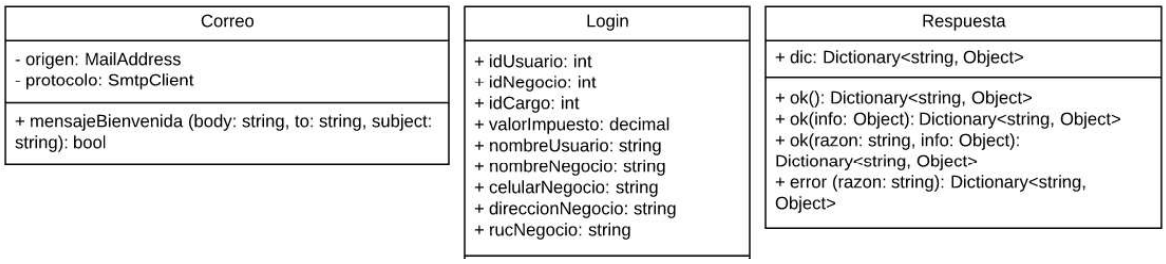


Figura 2.14 Diagrama de clases auxiliares UML *sprint* 1

El método `mensajeBienvenida` de la clase `Correo` permite enviar un correo de confirmación con la clave de acceso al finalizar el registro de un vendedor. Los métodos definidos en la clase `Respuesta` están destinados a enviar información de control y respuesta ante una petición del usuario, mientras que la clase `Login` fue destinada específicamente para disponer de los datos de inicio de sesión, los cuales serán enviados al cliente.

De la misma manera en el lado del cliente se crearon las clases auxiliares: `Login` para formar un objeto con las credenciales de acceso al prototipo, la clase `Api` y la interfaz `IApi` para manejar el envío de peticiones hacia el servidor. Además, la clase `Respuesta` y la interfaz `RespuestaListener` permiten administrar la información de control y respuesta que arriba desde el servidor ante una petición realizada. La transferencia de información entre los actores se hace en formato JSON, como se muestra en la Figura 2.15, donde se ejemplifica el envío de información para el registro de negocio.

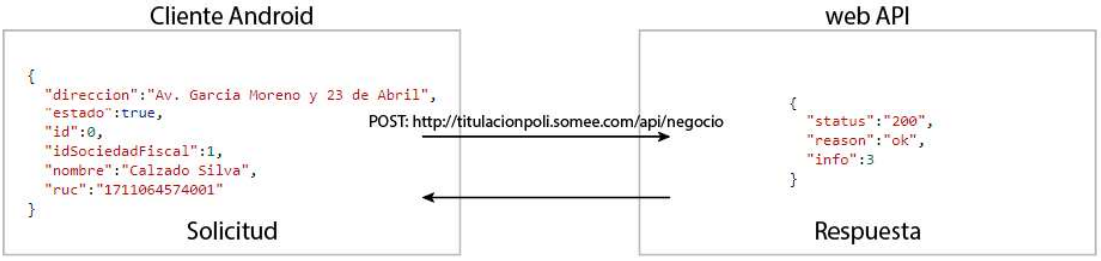


Figura 2.15 Transferencia de información para registro de negocio

La respuesta de la web API corresponde a un objeto del tipo `Respuesta`, que contiene un diccionario con las claves: `status`, `reason` e `info`. La clave `status` puede contener los

valores 200 o 500 para referirse a un proceso exitoso o fallido respectivamente, la clave `reason` contiene una cadena de texto con la explicación del error producido durante el proceso, mientras que la clave `info` alberga de manera opcional un objeto JSON que se desea enviar como respuesta, el cliente Android será el encargado de transformar esta información en el objeto de interés.

2.4.3 DESARROLLO

El *sprint* comenzó con la codificación en el lado del servidor de los métodos que permiten atender las solicitudes de tipo `POST` de los controladores `UsuarioController` y `NegocioController`. Ambos métodos mantienen una misma estructura dividida en tres secciones: obtención de datos, validación de datos y registro de información en la base de datos, pero difieren en las entidades con las que trabajan, la fuente de datos sobre la cual se guarda la información y la información que se retorna al lado del cliente.

En el Código 2.1 se muestra el método `Post` del controlador `NegocioController`, el cual acepta como entrada un objeto JSON (línea 27). Este objeto de entrada se transforma en un objeto `negocios` (línea 32), cuya información se contrasta con la información presente en la base de datos para evitar posibles registros duplicados (línea 34). Si no existen datos duplicados el negocio se activa (línea 40) y almacena en la base de datos con ayuda de sentencias LINQ (líneas 41 y 42) y se retorna al cliente el identificador asignado a dicho negocio (línea 43). Si por el contrario existen datos duplicados se envía un mensaje de error al cliente con la razón de dicho error (línea 45).

```
27      public IHttpActionResult Post([FromBody]JObject data)
28      {
29          try
30          {
31              //obtencion de datos
32              negocios empresa = data.ToObject<negocios>();
33              //validacion
34              negocios controlEmpresa = db.negocios.Where(i => i.estado == true
35                                                          && i.ruc == empresa.ruc)
36                                                          .FirstOrDefault();
37              if (controlEmpresa == null)
38              {
39                  //registro
40                  empresa.estado = true;
41                  db.negocios.Add(empresa);
42                  db.SaveChanges();
43                  return Ok(resp.ok(empresa.id));
44              }
45              else
46              {
47                  return Ok(resp.error("Ya existe una empresa con este RUC"));
48              }
49          }
50          catch (Exception ex)
51          {
52              return Ok(resp.error(ex.Message));
53          }
54      }
```

Código 2.1 Registro de negocio en el servidor

Luego, se codificó el método `Login` (Ver Código 2.2) al cual se tiene acceso mediante la petición `POST: api/Usuario/Login` definida mediante las etiquetas de acción y método de las líneas 104 y 105. Este método toma como parámetro de entrada un objeto JSON del cual se extrae el correo y la contraseña del usuario, y con ellos se busca el registro de usuario coincidente en la base de datos (línea 111). En caso de existir el usuario, se busca información acerca de los negocios y cargos a los cuales está registrado (línea 113) y con ello se arma una lista de objeto `Login` que son devueltos al lado del cliente (línea 137). Caso contrario se retorna un mensaje de error (línea 139).

```
104     [HttpPost]
105     [Route("api/Usuario/Login")]
        0 references
106     public IHttpActionResult Login([FromBody]JObject data)
107     {
108         string cor = (string)data["correo"];
109         string pass = (string)data["clave"];
110
111         usuarios usuario = db.usuarios.Where(i => i.correo == cor && i.clave == pass).FirstOrDefault();
112         if (usuario != null){
113             List<usuarios_x_cargos_x_negocios> listAux = db.usuarios_x_cargos_x_negocios.Include("negocios")
114                                                          .Where(i => i.idUsuario == usuario.id && i.estado == true
115                                                          && i.negocios.estado == true).ToList();
116             List<Login> listaCredenciales = new List<Login>();
117             foreach (usuarios_x_cargos_x_negocios aux in listAux)...
137             return Ok(resp.ok(listaCredenciales));
138         }else{
139             return Ok(resp.error("Credenciales no encontradas"));
140         }
141     }
```

Código 2.2 Inicio de sesión en el lado del servidor

A continuación, se codificaron los métodos que permiten atender las peticiones de tipo `GET`, en ambos controladores, para la lectura de datos de usuario y negocio. En el Código 2.3 se ejemplifica la búsqueda de datos de un usuario (líneas 23-25) a partir de su identificador y el identificador de negocio como parámetros de entrada para enviarlos hacia el cliente (línea 26).

```
20      public IHttpActionResult Get(int id, int idNegocio)
21      {
22          usuarios usuario = db.usuarios.Single(i => i.id == id);
23          usuario.usuarios_x_cargos_x_negocios = db.usuarios_x_cargos_x_negocios
24                                                 .Where(i => i.idNegocio == idNegocio && i.idUsuario == id)
25                                                 .ToList();
26          return Ok(resp.ok(usuario));
27      }
```

Código 2.3 Lectura de datos de usuario desde el servidor

Luego, se codificaron los métodos referentes a las acciones `PUT` en los controladores `NegocioController` y `UsuarioController` para controlar la edición de los registros correspondientes. El código de estos métodos guarda la misma estructura que la de los

métodos `POST` previamente explicados. En el lado del cliente se inició codificando las interfaces gráficas descritas en el apartado Esquema visual con la ayuda del lenguaje de marcado XML. Los nombres de las actividades (o fragmentos) y su función se describen en la Tabla 2.5.

Tabla 2.5 *Layouts* para *sprint* 1

Layout	**Descripción**
`activity_home`	Es la actividad inicial, cuenta con botones para direccionar al inicio de sesión o al registro de usuario.
`activity_login`	Actividad de inicio de sesión, cuenta con campos de texto para capturar las credenciales de usuario y un botón para realizar la autenticación.
`activity_register`	Actividad para el registro de usuario y negocio, cuenta con varios componentes de texto para capturar datos y un botón para guardar la información.
`activity_order`	Es la actividad principal de la aplicación, cuenta con un componente `NavigationView` que representa el menú.
`fragment_empresa_detalle`	Fragmento para mostrar datos de negocio o editarlos, cuenta con campos de texto y un botón para modificar la información.
`fragment_perfil`	Fragmento para mostrar y editar datos del usuario registrado, cuenta con varios campos de texto y un botón para modificar la información.
`fragment_new_employee`	Fragmento para registrar un vendedor, cuenta con campos de texto para capturar la información del mismo y un botón para guardar la información.

Con las interfaces gráficas levantadas se procedió a codificar la interfaz `IApi` para consumir los *endpoints* del servidor. En el Código 2.4 se muestra la implementación de esta interfaz y ejemplos de algunos *endpoints* para la administración de datos de negocio.

```
26  public interface ApiService {
27      //*********NEGOCIO******
28      @GET("negocio")
29      Call<Respuesta> getNegocio(@Query("id") int id);
30      @GET("negocio/Impuesto")
31      Call<Respuesta> getNegocioImpuesto(@Query("idNegocio") int idNegocio);
32      @POST("negocio")
33      Call<Respuesta> postNegocio(@Body Negocio negocio);
34      @PUT("negocio")
35      Call<Respuesta> putNegocio(@Query("id") int id, @Body Negocio negocio);
36  }
```

Código 2.4 Interface `IApi`

En la clase `Api` que implementa la interfaz se codificó el método `getService`, el cual permite ensamblar los paquetes HTTP con ayuda de la librería Retrofit y el método `solicitud`.

En el código 2.5 se muestra la implementación de este método, el cual recibe una URL como parámetro de entrada, genera el paquete HTTP con ayuda del método `enqueue` y la clase `Callback` (línea 47) la cual provoca que el hilo de ejecución se detenga hasta que la repuesta del servidor arribe. Si la respuesta del servidor es exitosa, se ejecuta el método `OnResponse` (línea 49), en el cual se extrae el cuerpo de la respuesta que corresponde a un objeto de tipo `Respuesta` en formato JSON (línea 52) y se valida que las claves de control `status` y `reason` no adviertan un error (línea 53), para finalmente devolver la información contenida en la clave `info` (línea 54). Si las claves antes analizadas advierten de un error, la razón del mismo se muestra en pantalla (línea 57).

```
45      public void solicitud(Call<Respuesta> endpoint, final RespuestaListener listener){
46          Call<Respuesta> call = endpoint;
47          call.enqueue(new Callback<Respuesta>() {
48              @Override
49              public void onResponse(Call<Respuesta> call, Response<Respuesta> response) {
50                  if (response.isSuccessful())
51                  {
52                      Respuesta r = response.body();
53                      if (r.getStatus().equalsIgnoreCase( anotherString: "200") && r.getReason().equalsIgnoreCase( anotherString: "ok")){
54                          listener.onData(r.getInfo());
55                      }
56                      else{
57                          Mensaje.simple(r.getReason(), duracion: 1);
58                      }
59                  }
60                  else
61                  {
62                      Mensaje.simple(response.message(), duracion: 1);
63                  }
64              }
65              @Override
66              public void onFailure(Call<Respuesta> call, Throwable t) {
67                  Mensaje.simple(t.getLocalizedMessage(), duracion: 1);
68              }
69          });
70      }
```

Código 2.5 Método `solicitud` de la clase `Api`

Acto seguido se procedieron a codificar los métodos que controlan los eventos `onClick` de los botones en cada actividad. Al igual que en el lado del servidor, estos métodos tienen una estructura similar basada en cuatro secciones: validación de datos, ejecución de petición, lectura de datos de respuesta y código complementario. Adicionalmente en cada actividad o fragmento se codificó el método `validarCampos` el cual contiene una serie de lazos `if-else` que permiten validar la información ingresada por el usuario en cada campo de texto de la vista, a la vez que forma el objeto necesario en dicha actividad o fragmento y retorna un objeto de tipo `bool` como respuesta a la validación.

El primer método desarrollado fue en la actividad `activity_register`. Su implementación se muestra en el Código 2.6, se puede ver que una vez validada la información ingresada por el usuario (línea 57) se realiza una petición de tipo `POST` al servidor para registrar el negocio (línea 60), esta llamada retorna los datos de respuesta en formato JSON, la cual se transforma en un entero (línea 65) y corresponde al identificador del negocio registrado en la base de datos, acto seguido se registra al administrador del negocio (línea 67), se transforma el dato de retorno (línea 71) que corresponde al identificador del usuario en la base de datos. Con todos los datos necesarios se crea un objeto de tipo `LoginResponse` (línea 72) y se guarda dicho objeto de manera local en el dispositivo móvil (línea 75 a 77). Finalmente, se inicia la actividad principal `MainActivity` (líneas 78 y 79).

```
56    public void onClick(View v) {
57        if (validarCampos())//validacion de datos
58        {
59            //ejecuacion de peticion
60            new Api().solicitud(Api.getService().postNegocio(negocio), (data) -> {
63                    //lectura de datos de respuesta
64                    int idNegocio = new Gson().fromJson(data, int.class);
65                    //codigo complementario
66                    usuario.setIdNegocio(idNegocio);
67                    new Api().solicitud(Api.getService().postUsuario(usuario), (data) -> {
70                        int idUsuario = new Gson().fromJson(data, int.class);
71                        new LoginResponse().setIndexPosition(0);
72                        LoginResponse log = new LoginResponse(idUsuario,txtNombres.getText().toString(),
73                                usuario.getIdNegocio(),txtNombreNegocio.getText().toString(),impuesto, idCargo: 1,
74                                usuario.getClave());
75                        List<LoginResponse> listaLogin = new ArrayList<>();
76                        listaLogin.add(log);
77                        new LoginResponse().guardarDatosIniciales(listaLogin);
78                        Intent actOrder = new Intent(App.getAppContext(), MainActivity.class);
79                        startActivity(actOrder);
80                    });
82            });
84        }
85    }
```

Código 2.6 Método `onClick` registro de usuario y negocio

Una vez registrados los datos iniciales se procedió a codificar el inicio de sesión en la actividad `activity_login` con una estructura similar a la detallada en el Código 2.6, en la cual se validan los datos, se realiza la petición al servidor, se forma un objeto `LoginResponse` para guardar los datos de inicio de sesión en el dispositivo y se inicia la actividad principal. Una vez codificados los métodos de acceso a la actividad principal, se escribió el código de control para el menú. Dado que este componente estará presente en todas las interfaces del prototipo, el código se definió en la clase `MainActivity`, donde se implementaron los métodos `configMenu` y `onNavigationItemSelected`. El primer método permite mostrar o esconder las opciones del menú en función del cargo que el usuario tiene, esto se logró haciendo uso de los métodos `getItem` y `setVisible` de la clase `Menu` que representa al componente gráfico.

El segundo método permite navegar entre vistas con base en la selección de un *ítem* del menú. En el Código 2.7 se presenta un fragmento de este método, donde se define la vista (`Fragment`) a la que se desea dirigir (líneas 162 a 165) en función al ítem del menú seleccionado y se inicia la misma (línea 168). Como se puede notar el inicio de una vista se realiza con la ayuda del método `getSupportFragmentManager`, esto se debe a que aquí se trabajaba con fragmentos a diferencia de los métodos para registro e inicio de sesión donde se trabaja con actividades (Ver Código 2.6 líneas 76 y 77).

```
159             Fragment fragment = null;
160             switch(item.getItemId()){
161                 case R.id.nav_catalogo:
162                     fragment = new CatalogoFragment();
163                     break;
164                 case R.id.nav_categoria:
165                     fragment = new CategoriaFragment();
166                     break;
167             }
168             getSupportFragmentManager().beginTransaction().replace(R.id.content_main,fragment)
169                                                           .addToBackStack(null).commit();
```

Código 2.7 Menú principal del prototipo

Los fragmentos son interfaces gráficas que pueden mostrarse como parte de otra para así mantener elementos comunes entre ellos y optimizar la reutilización de código, de manera lógica. El desarrollo con fragmentos permite controlar componentes e información de manera global en la clase principal (`MainActivity`) así como componentes e información independiente entre cada interfaz gráfica secundaria (`Fragment`) [31].

Acto seguido se codificaron los métodos necesarios para mostrar y editar los datos de usuario y negocio. Un ejemplo de la lógica necesaria para mostrar datos de negocio se muestra en el Código 2.8, donde a raíz del identificador de negocio almacenado de manera local se realiza la petición de datos al servidor (línea 77), la información recibida se transforma en un objeto de tipo `Negocio` (línea 80) y con esto se imprime la información en los campos de texto (línea 81 a 83) del fragmento. Este método se llama al iniciar la presentación del fragmento dentro del método `onCreateView`.

```
75      public void mostrarDatos(){
76          int idNegoio = new LoginResponse().leerDatos().getIdNegocio();
77          new Api().solicitud(Api.getService().getNegocio(idNegoio), (data) → {
80                  negocio = new Gson().fromJson(data, Negocio.class);
81                  txtNombre.setText(negocio.getNombre());
82                  txtDireccion.setText(negocio.getDireccion());
83                  txtRuc.setText(negocio.getRuc());
84          });
86      }
```

Código 2.8 Presentación de datos de negocio

Finalmente, se presenta un ejemplo para la modificación de datos de negocio en el Código 2.9, donde se validan los datos ingresados por el usuario en la vista (línea 55), se realiza la petición de tipo `PUT` al servidor con los nuevos datos (línea 56), se guarda de manera local los datos modificados (líneas 59 y 60) y se cierra dicha vista (línea 61).

```
53    public void onClick(View v) {
54        final LoginResponse log = new LoginResponse().leerDatos();
55        if (validarDatos()){
56            new Api().solicitud(Api.getService().putNegocio(log.getIdNegocio(), negocio), (data) → {
59                    log.setNombreNegocio(txtNombre.getText().toString());
60                    log.guardarDatos();
61                    getActivity().getSupportFragmentManager().popBackStack();
62            });
64        }
65    }
```

Código 2.9 Edición de datos de negocio

2.4.4 ENTREGABLE

En este *sprint* se implementó la funcionalidad que permite a los administradores de un negocio registrar los datos del mismo, así como sus datos personales en el sistema. Una vez registrado, el administrador accede a la actividad principal del prototipo y al menú principal de navegación desde donde puede navegar hacia las vistas respectivas para modificar los datos previamente registrados y registrar a vendedores. Tanto los administradores como los vendedores pueden acceder al prototipo haciendo uso de sus credenciales (correo y contraseña) desde la vista del inicio de sesión, el menú de navegación presenta opciones limitadas para los vendedores en comparación con un administrador.

2.5 SPRINT 2: MANEJO DE PRODUCTOS

El objetivo de este *sprint* es levantar las interfaces de navegación que permitan la administración de categorías y productos en el prototipo de punto de venta. Al finalizar las tareas, un administrador podrá registrar sus categorías, productos y tallas asociadas, editar los datos respectivos, eliminar registros de categoría o productos y filtrar los productos en función de la categoría asociada o de su nombre.

2.5.1 ESQUEMA GRÁFICO

En este *sprint* se han considerado los *sketckes* de la Figura 2.16, de izquierda a derecha se presenta una lista de las categorías registradas (*sketch* 1), esta vista contiene también un cuadro de texto que permite la búsqueda de datos, el *sketch* 2 contiene un cuadro de

texto y un componente `Spinner` donde se mostrarán los detalles de una categoría seleccionada, esta vista también permitirá la edición y registro de una nueva categoría.
De manera similar el *sketch* 3 contiene una lista con los productos ya creados y un campo de texto para filtrar los registros existentes mientras que el *sketch* 4 está compuesto por varias cajas de texto y listas de selección donde se mostrarán los detalles de un producto. Este *sketch* permite además modificar los datos de un producto, así como la creación de uno nuevo.

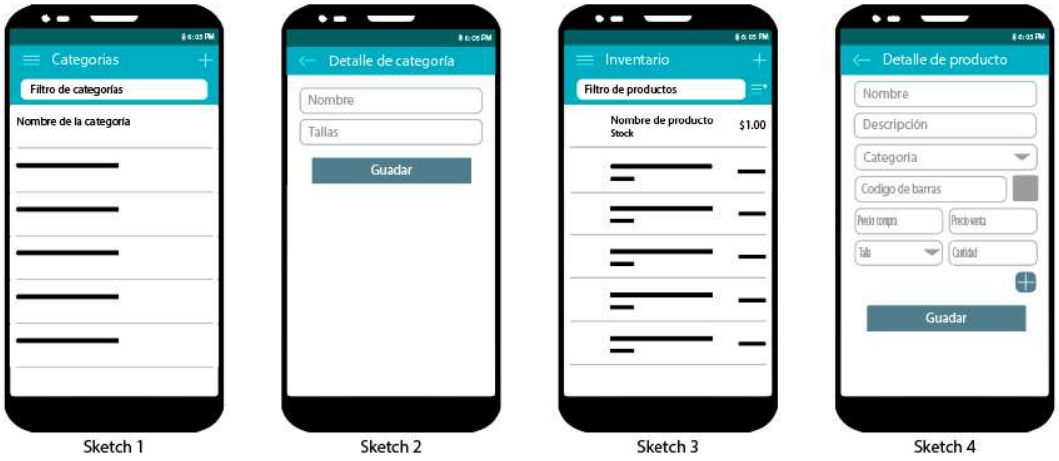


Figura 2.16 *Sketches* para la administración de categorías y productos

Cada *sketch* que contiene una lista tiene un botón en la barra superior que permitirá abrir una vista para registrar una nueva entidad como se ejemplifica en la Figura 2.17 con la interacción de los *sketches* 1 y 2. Además, se implementará la interacción `swipeDelete` en la lista para eliminar un registro como se muestra en la Figura 2.18.

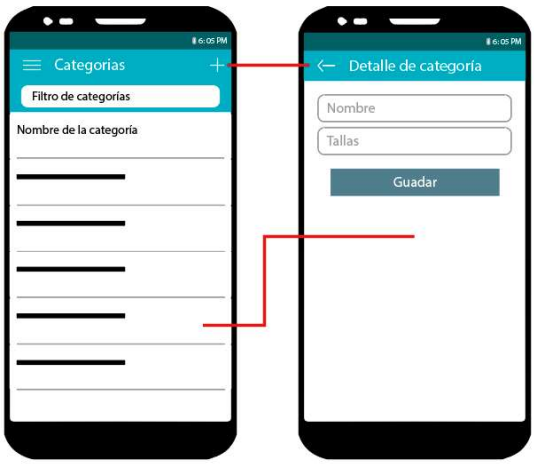


Figura 2.17 Creación de una nueva categoría

Figura 2.18 Eliminar registros con `swipeDelete`

2.5.2 ESQUEMA CONCEPTUAL

Para la administración de datos de categorías y productos se definieron las entidades `tipos_tallas`, `tallas`, `categorías`, `productos` y `productos_x_talla` en la base de datos cuyos objetivos se detallaron previamente en la sección 2.1.6. A más de las clases que representan a las entidades mencionadas y los controladores, en la web API se crearon las clases auxiliares que se muestran en la Figura 2.20.

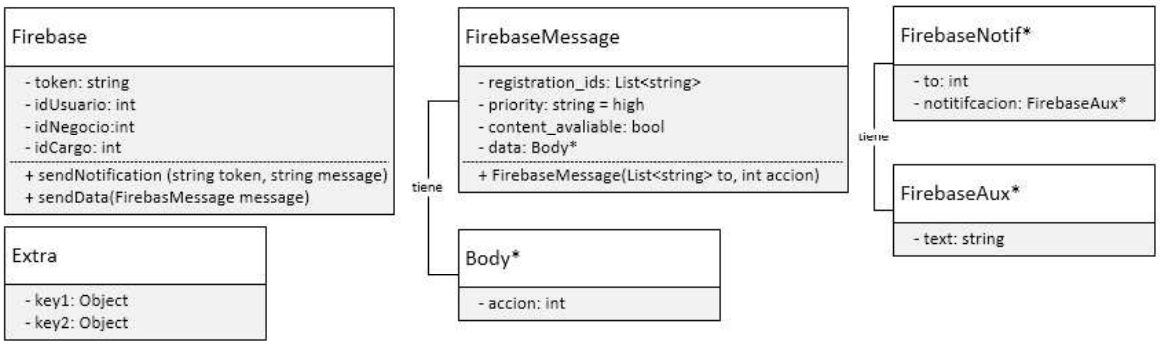


Figura 2.19 Diagrama UML de clases auxiliares web API - *sprint* 2

La clase `Extra` permite extender la información de un modelo que se envía hacia el cliente. La clase `Firebase` cuenta con el método `sendData` para enviar información hacia los clientes y con ella mantener la sincronización entre dispositivos y el método `sendNotification` para enviar alertas o textos generales hacia los clientes. La clase `FirebaseMessage` cuenta entre sus parámetros con el campo `registration_ids`, el cual contiene una lista de *tokens* de los dispositivos móviles hacia donde se enviarán los mensajes o notificaciones. Para que un cliente pueda interpretar la acción a realizar en función de los datos que arriba desde el servidor FCM, en la estructura `Body` la cual

representa al cuerpo del mensaje se ha definido una propiedad de tipo `integer` y su significado se presenta en la Tabla 2.6.

Tabla 2.6 Códigos de acción para sincronización de datos

Código	Descripción
1	Se genera ante el registro de una venta, en el cliente se tendrá que actualizar la lista de productos.
2	Se genera ante la modificación de un producto, en el cliente se tendrá que actualizar el registro correspondiente.
3	Se genera ante el registro de la talla de un producto, en el cliente se tendrá que actualizar el registro correspondiente.
4	Se genera ante el cambio en el valor de un impuesto, en el cliente se tendrá que actualizar dicho registro.

De la misma manera, en el lado del cliente se crearon las clases que representan a las entidades de la base de datos (ver Figura 2.7) involucradas en este *sprint* y las clases auxiliares que se presentan en la Figura 2.20. La clase `Extra` permite manejar la información adicional enviada desde el servidor, `FirebaseService` para manejar el arribo de datos desde el servidor FCM y `FirebaseToken` para enviar el identificador asignado al dispositivo hacia el servidor y registrarlo en la base de datos.

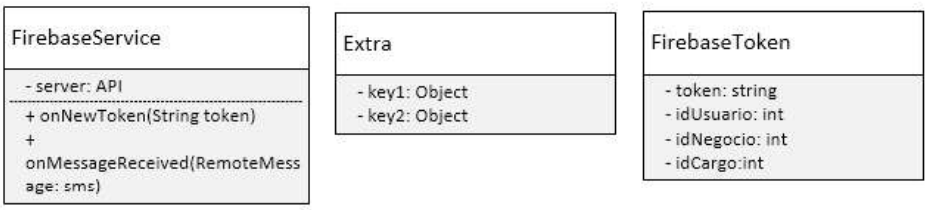


Figura 2.20 Diagrama UML de clases auxiliares en el cliente- *sprint* 2

2.5.3 DESARROLLO

Este *sprint* comenzó con la codificación en el lado del servidor de los métodos que permiten poblar las tablas de categorías y productos respectivamente desde el lado del cliente.

El método referente al verbo `GET` del controlador `ProductoController` se muestra en el Código 2.10 donde se pone en evidencia el uso de las etiquetas de acción (línea 17) y ruta (línea 18) para definir el acceso al mismo. Este método define como parámetro de entrada el identificador del negocio con el cual se busca en la base de datos los productos

registrados en dicho negocio con ayuda de sentencias LINQ, en esta búsqueda también se hizo uso del método `OrderBy` (línea 21) para organizar alfabéticamente los resultados en función de su nombre. Con la ayuda de la clase `Extra` se agregó a cada producto el *stock* y el precio más bajo (líneas 27-39). La lista de productos generada se envía al cliente (línea 33).

```
17      [HttpGet]
18      [Route("api/Producto/negocio")]
19      public IHttpActionResult ProductosNegocio(int id)
20      {
21          List<productos> productos = db.productos.Where(i => i.estado == true && i.idNegocio == id)
22                                                  .OrderBy(i => i.nombre).ToList();
23          foreach (var item in productos)
24          {
25              List<productos_x_tallas> tallas = db.productos_x_tallas.Where(i => i.idProducto == item.id
26                                                                          && i.estado == true).ToList();
27              item.extra = new Extra()
28              {
29                  key1 = tallas.OrderByDescending(i => i.total).FirstOrDefault().total,
30                  key2 = tallas.Sum(i => i.cantidad),
31              };
32          }
33          return Ok(r.ok(productos));
34      }
```

Código 2.10 Búsqueda de productos por negocio

Luego, se codificaron los métodos para controlar las acciones `POST` y `PUT` de los controladores en cuestión, para manejar el registro y modificación de información en la base de datos. Estos métodos mantienen una lógica de codificación similar a los métodos previamente implementados en los controladores `UsuarioController` y `NegocioController` del *sprint* 1 (ver Código 2.1).

A continuación, fueron implementados los métodos para dar de baja un registro de categoría y producto. El método para dar de baja un producto se presenta en el Código 2.11 donde a partir del identificador de producto, se busca el registro correspondiente en la base de datos (línea 119) con la ayuda de sentencias LINQ, este registro se da de baja cambiando el campo `estado` a `false` (línea 120), se actualiza la base de datos (línea 121) y se retorna un mensaje de confirmación (línea 122).

```
119     public IHttpActionResult Delete (int id)
120     {
121         productos item = db.productos.Single(i => i.id == id);
122         item.estado = false;
123         db.SaveChanges();
124         return Ok(r.ok());
125     }
```

Código 2.11 Lógica para eliminar un producto

En referencia a la sincronización de datos entre dispositivos clientes, se implementó el método `sendData` en la clase `Firebase` en el lado del servidor como muestra el Código

2.12, en el cual se define la URL hacia donde se va dirigir la petición (línea 32), su método (línea 33) y el tipo de contenido del cuerpo (línea 35). El mensaje a enviar que se tiene como parámetro de entrada se serializa en formato JSON (línea 36) y se añade a la petición (línea 37). Para terminar la petición se usa el método `Execute` (línea 38). Este método se llama al finalizar los procesos de registro y modificación de datos de productos.

```
30      public void sendData(FirebaseMessage message)
31      {
32          var client = new RestClient("https://fcm.googleapis.com/fcm/send");
33          var request = new RestRequest(Method.POST);
34          request.AddHeader("Authorization", "key=.......");
35          request.AddHeader("Content-Type", "application/json");
36          string body = new JavaScriptSerializer().Serialize(message);
37          request.AddParameter("application/json", body, ParameterType.RequestBody);
38          IRestResponse response = client.Execute(request);
39      }
```

Código 2.12 Método `sendData`

En el lado del cliente, se integró el servicio FCM, para esto se agregaron las rutas definidas en el archivo `gradle` y se estableció la ruta a la clase de control de este servicio en el archivo `AndroidManifest.xml` haciendo uso de la etiqueta `service` como se muestra en el Código 2.13.

```
22  <service
23      android:name=".Model.Conexion.FirebaseServices"
24      android:stopWithTask="false">
25      <intent-filter>
26          <action android:name="com.google.firebase.MESSAGING_EVENT" />
27      </intent-filter>
28  </service>
```

Código 2.13 Conexión con Firebase Cloud Messaging

La clase `FirebaseService`, como se explicó en el Capítulo 1, requiere la implementación de los métodos `onNewToken` y `onMessageReceived`. En el primer método se agregó la lógica necesaria para almacenar en el dispositivo móvil el *token* asignado por el servicio FCM.

Para registrar el identificador asignado a un dispositivo móvil en la base de datos se llama a un método en la clase `UsarioController`, mediante una petición `POST: api/usuario/token` al finalizar los procesos de registro e inicio de sesión, que fueron implementados en el *sprint* 1 (ver Código 2.6). Para estas peticiones fue necesario formar un objeto de tipo `FirebaseToken` y agregarlo al cuerpo de la petición. A continuación, se levantaron las interfaces gráficas correspondientes a los fragmentos o actividades, de acuerdo a lo descrito en la Tabla 2.7.

Tabla 2.7 *Layouts* para *sprint* 2

Layout	Descripción
`fragment_categoria`	Lista de categorías, se compone de un `RecyclerView` para la lista y un `SearchView` para filtrar datos.
`subtitle_item`	Representa a una fila de la lista de categorías. Cuenta con un componente `SwipeLayout` como elemento principal.
`fragment_categoria_detalle`	Fragmento para el registro y edición de una categoría.
`fragment_producto`	Lista de productos, cuenta con el componente `RecyclerView` como elemento principal y un componente `SearchView` para filtrar datos.
`producto_item`	Representa a una fila de la tabla productos. Cuenta con un componente `SwipeLayout` como elemento principal
`fragment_producto_detalle`	Vista para el registro y edición de un producto.
`swipe_delete`	Contiene la vista que se mostrara al usuario al momento de generar el evento *swipeDelete*.

Una vez levantadas las interfaces gráficas se escribió el código para poblar la lista de categorías y eliminar registros mediante la interacción `swipeDelete`. Para estos se usaron las clases: `CategoriaFragment` que representa a la vista con la lista de categorías, y un adaptador llamado `CategoriaAdapter`. En Android, un adaptador es una clase que sirve de puente entre los datos y un componente visual de tipo `ListView`. `GridView`, `Spinner` o `RecyclerView` [32].

La lógica implementa para poblar la lista de categorías se muestra en el Código 2.14, donde se realiza una petición al servidor para obtener los registros de categoría, en dicha solicitud se envía también el identificador del negocio (línea 67), los datos que arriban del servidor en formato JSON se transforman a una lista de objetos de tipo `Categoria` (línea 71) los cuales a su vez son enviados al adaptador (línea 72) y finalmente se asocia el adaptador con el componente `ReciclerView` (línea 73).

```
66      public void  mostrarDatos(){
67          new Api().solicitud(Api.getService().getCategorias(new LoginResponse().leerDatos().getIdNegocio()),
68                  (data) → {
71                  List<Categoria> categorias = Arrays.asList(new Gson().fromJson(data, Categoria[].class));
72                  adapter = new CategoriaAdapter(categorias, getFragmentManager());
73                  recicler.setAdapter(adapter);
74          });
76      }
```

Código 2.14 Adaptador de lista

En el adaptador `CategoriaAdapter` se implementó el método `onBindViewHolder`, el cual permite imprimir los datos de cada objeto recibido, como se muestra en el Código 2.15

mediante el método `setText` de cada componente `TextView`.

```
62      public void onBindViewHolder(final SimpleViewHolder viewHolder, final int position) {
63          final Categoria categoria = categorias.get(position);
64          viewHolder.lblTitle.setText(categoria.getNombre());
65          viewHolder.lblSubtitle.setText("");
66      }
```

Código 2.15 Método `onBindViewHolder`

Dentro del método `onBindViewHolder` se implementó también la respuesta al evento `onClick` de cada *item* de la lista mediante el método `setOnClickListener`, cuyo objetivo es presentar la información de una categoría en la vista de edición de datos. La lógica implementada se presenta en el Código 2.16, donde con la ayuda la clase `Bundle` se define la información que se pasa a la vista de edición de datos `CategoriaDetalleFragment` (líneas 70 a 72). El número 1 representa edición, mientras que el número 0 indica que la vista debe borrar toda la información presente para permitir crear un nuevo registro. La línea 75 inicia la vista de edición mencionada para presentar los datos de la categoría seleccionada.

```
67  viewHolder.swipeLayout.getSurfaceView().setOnClickListener((v) → {
70          Bundle arg = new Bundle();
71          arg.putSerializable("categoriaSel",categoria);
72          arg.putInt("accion",1);
73          Fragment detalles = new CategoriaDetalleFragment();
74          detalles.setArguments(arg);
75          fragment.beginTransaction().replace(R.id.content_main,detalles)
76                                     .addToBackStack(null).commit();
77  });
```

Código 2.16 Implementación de evento `onClick` en listas

De la misma manera, se codificó la respuesta a la interacción `swipeDelete` para eliminar un registro. Esto se muestra en el Código 2.17, donde se activa dicha interacción sobre cada *item* de la lista (línea 80). La línea 83 envía una petición de tipo `DELETE` hacia el servidor para eliminar el registro, si la petición se procesa correctamente, se elimina el objeto (línea 86) en el cliente Android y se actualizan los datos en la lista gráfica (líneas 87 y 88).

```
80  viewHolder.swipeLayout.setShowMode(SwipeLayout.ShowMode.PullOut);
81  viewHolder.swipeLayout.addDrag(SwipeLayout.DragEdge.Right,viewHolder.swipeLayout.findViewById(R.id.bottom_wraper));
82  viewHolder.delete.setOnClickListener((v) → {
85          new Api().solicitud(Api.getService().deleteCategoria(categoria.getId()), (data) → {
88                  categorias.remove(position);
89                  notifyItemRemoved(position);
90                  notifyItemRangeChanged(position,categorias.size());
91          });
93  });
```

Código 2.17 Implementación de evento `swipeDelete` en listas

A continuación, se implementó la clase `CategoriaDetalleFragment` la cual representa a la vista para el registro y edición de una categoría, al iniciar dicha vista se llama al método `confView` que se muestra en el Código 2.18, el cual permite con base en la variable `accion` recibida de la clase `CategoriaAdapter` (líneas 62 y 63) presentar los datos de un objeto `Categoria` (líneas 65 a 67) para la edición del mismo o limpiar los campos de texto para el ingreso de información (línea 69).

```
61      private void confView(View v) {
62          final Bundle arg = getArguments();
63          accion = arg.getInt( key: "accion");
64          if (accion == 1) {
65              categoria = (Categoria) arg.getSerializable( key: "categoriaSel");
66              btnRegistar.setText("Modificar");
67              mostrarDatos();
68          }else{
69              btnRegistar.setText("Registrar");
70          }
71      }
```

Código 2.18 Configuración de vista de registro y edición de categorías

También se implementó el método `onClick` como se muestra en el Código 2.19, donde previo una validación de la información ingresada por el usuario (línea 114) y con base en la variable `acción` se ejecuta una petición de tipo `POST` (línea 116) para registrar una categoría en la base de datos, o una petición de tipo PUT (línea 124) para modificar un registro de categoría. Las líneas 119 y 127 notifican al usuario que las solicitudes enviadas al servidor fueron realizadas satisfactoriamente mientras que las líneas 120 y 128 permiten iniciar la vista con la lista de categorías.

```
113     public void onClick(View v) {
114         if (validarCampos()) {
115         if (accion == 0) {//registro de datos
116             new Api().solicitud(Api.getService().postCategoria(categoria), (data) → {
119                     Mensaje.simple( mensaje: "Categoría registrada correctamente", duracion: 1);
120                     getActivity().getSupportFragmentManager().popBackStack();
121             });
123         } else {//edicion de datos
124             new Api().solicitud(Api.getService().putCategoria(categoria.getId(), categoria), (data) → {
127                     Mensaje.simple( mensaje: "Categoría modificada correctamente", duracion: 1);
128                     getActivity().getSupportFragmentManager().popBackStack();
129             });
131         }
132         }
133     }
```

Código 2.19 Método `onClick` para registro y edición de categorías

Continuando con el desarrollo de este *sprint* y siguiendo los ejemplos de los Códigos 2.14, Código 2.15, Código 2.16, Código 2.17, Código 2.18 y Código 2.19 para la administración de categorías, se implementó todo el control para la presentación de productos,

eliminación, edición y creación de los mismos en las clases `ProductoFragment`, `ProductoAdapter` y `ProductoDetalleFragment`.

Para la sincronización de datos se implementó el método `onMessageReceived` en la clase `FirebaseService` como se muestra en el Código 2.20, donde a partir de un dato de tipo `integer` que arriba desde el servidor (línea 41) se crea un anuncio con la ayuda de la clase `LocalBroadcastManager` (línea 43), esta clase permite enviar información a través de un mensaje hacia todos los fragmentos y actividades de la aplicación [33]. Este mismo método atendió la posible actualización del valor de un impuesto, cuando del servidor llega el valor 4. Para esto se obtiene el identificador del negocio guardado localmente (línea 49) y se consulta al servidor por el nuevo valor del impuesto (línea 50) finalmente este valor es actualizado localmente (líneas 54 y 55).

```
41  int accion = Integer.parseInt(remoteMessage.getData().get("accion"));
42  if (accion >=1 && accion <=3){
43      LocalBroadcastManager broadcaster = LocalBroadcastManager.getInstance(App.getAppContext());
44      Intent intent = new Intent( action: "firebase_update");
45      intent.putExtra( name: "accion",accion);
46      broadcaster.sendBroadcast(intent);
47  }
48  else if (accion == 4){
49      final LoginResponse log = new LoginResponse().leerDatos();
50      new Api().solicitud(Api.getService().getNegocioImpuesto(log.getIdNegocio()), (data) → {
53              float impuesto = new Gson().fromJson(data, float.class);
54              log.setValorImpuesto(impuesto);
55              log.guardarDatos();
56              Log.e( tag: "Mensaje_FIREBASE",  msg: "Impuesto modificado");
57      });
59  }
```

Código 2.20 Interpretación de códigos de acción en el cliente

2.5.4 ENTREGABLE

Al finalizar este *sprint*, el prototipo permite a un administrador registrar categorías, datos de sus productos y tallas de los mismos, estos registros pueden ser asociados para mantener una mejor organización del inventario.

2.6 SPRINT 3: VENTAS I

El objetivo de este *sprint* es levantar las interfaces para la administración de clientes, así como para el registro de una venta. Al finalizar las tareas de este *sprint*, un usuario (administrador o vendedor) podrá registrar clientes y registrar una venta seleccionando los productos deseados y la cantidad de cada uno de ellos.

2.6.1 ESQUEMA GRÁFICO

En este *sprint* se han considerado los *sketckes* de la Figura 2.21 para la administración de clientes, el *sketch* 1 contiene una lista con los clientes registrados; mientras que el *sketch* 2 presenta un formulario con varios campos de texto para registrar un cliente o modificar los datos de uno ya existente. La interacción entre estas es similar a la definida para la administración de categorías en la Sección 2.4.1.

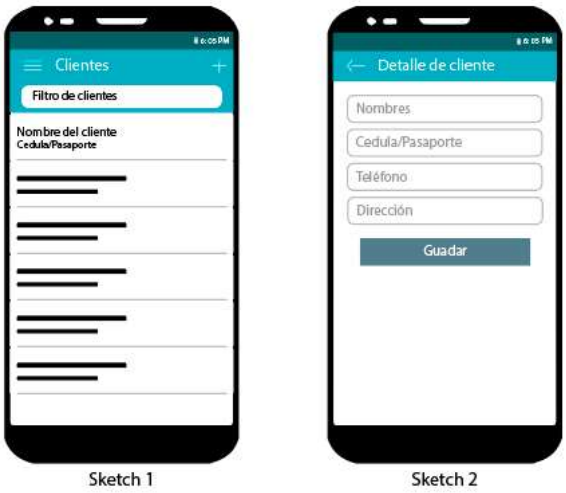


Figura 2.21 *Sketches* para administración de clientes

Para la generación de una venta se consideraron los *sketches* de la Figura 2.22. En donde se tiene un fragmento con el catálogo de productos (*sketch* 3), una lista que muestra los productos seleccionados (*sketch* 4) y un fragmento para seleccionar el cliente y el método de pago (*sketch* 5). Las interacciones entre estas últimas 3 vistas de muestran en la Figura 2.23.

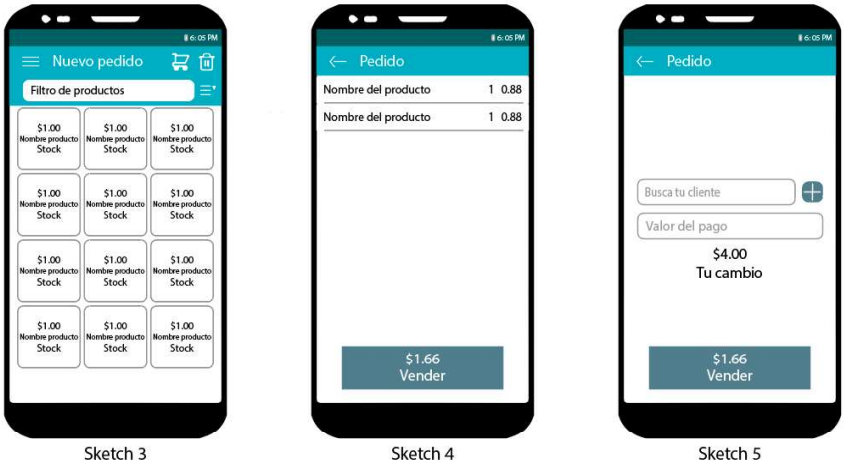


Figura 2.22 *Sketches* para generación de ventas

En el catálogo de productos (*sketch* 3) se alojan 2 botones en la barra superior, uno de ellos permite eliminar la lista con los productos seleccionados hasta ese momento mientras que el otro (carrito) permite acceder a una vista con la lista de dichos productos (*sketch* 4). En el *sketch* 5 se incluye un botón (+) el cual despliega una actividad para registrar un nuevo cliente y un botón `Vender` para finalizar la venta.

Figura 2.23 *Wireframes* para el registro de venta

2.6.2 ESQUEMA CONCEPTUAL

Para las tareas propuestas en este *sprint* se definieron las entidades `clientes`, `clientes_x_negocio`, `ventas`, `productos_x_venta` y `formas_pago` en la base de datos, cuya finalidad se detalló en la Sección 2.1.6.

A más de las clases que representan a las entidades mencionadas y los controladores correspondientes, en el lado del servidor se creó el controlador `CatalogoController` el cual no representa a ninguna entidad, sino que fue creado para tener una mejor organización del código fuente, este controlador cuenta con un único método que se llama al recibir peticiones de tipo `GET` y que permite obtener los registros que se mostrarán en la lista de productos (*sketch* 3).

En este *sprint* también se hace uso de la clase `Extra` tanto en el lado del servidor como en el lado del cliente para incluir información necesaria para construir la lista de productos que se mostrará en el catálogo.

2.6.3 DESARROLLO

Este *sprint* comenzó con la codificación en el lado del servidor de los métodos que se invocan al recibir una petición de tipo `GET`, `POST`, `PUT` o `DELETE` del controlador `ClienteController`. La implementación de cada uno de los métodos guarda la misma

estructura y lógica de codificación que sus similares presentados en el Código 2.1, Código 2.3 y Código 2.11.

También se implementó el método `ProductosNegocio` en el controlador `CatalogoController` el cual guarda cierta relación con el método del mismo nombre, pero del controlador `ProductoController` mostrado en el Código 2.10. La diferencia radica en que primer método a más de incorporar el *stock* y el precio mínimo en cada uno de los ítems de la lista de productos incorpora también una lista de tallas disponibles.

A continuación, se describe el método `Post` del controlador `VentaController` para registrar una venta en la base de datos. En el Código 2.21 las sentencias escritas entre las líneas 34 y 43 permiten transformar la información que arriba desde el lado del cliente en formato JSON a un objeto de tipo `ventas`, asignar un número de recibo y registrar la venta, los productos por venta y un registro de pago en la base de datos con la ayuda de sentencias LINQ. La iteración definida entre las líneas 45 y 50 reduce el *stock* de cada uno de los productos vendidos. Para sincronizar el nuevo stock en los dispositivos móviles, en la línea 51 se obtienen los *tokens* de identificación FCM de los usuarios del negocio donde se generó la venta y con ellos se realiza la petición al servidor FCM (líneas 54 y 55) para la redirección de la notificación de actualización a los clientes. Finalmente se retorna al cliente el número de recibo asignado por la base de datos a la venta registrada, este dato se usará en el cliente posteriormente para armar un comprobante de venta e imprimirlos a través de impresoras Bluetooth.

```
30  public IHttpActionResult Post([FromBody]JObject data)
31  {
32      try
33      {
34          ventas nuevo = data.ToObject<ventas>();
35          ventas ultima = db.ventas.Where(i => i.idNegocio == nuevo.idNegocio).ToList().LastOrDefault();
36          nuevo.recibo = (ultima == null) ? 1 : ultima.recibo + 1;
37          nuevo.idCliente = (nuevo.idCliente == 0) ? null : nuevo.idCliente;
38          nuevo.fecha_registro = DateTime.UtcNow;
39          foreach (pagos_ventas item in nuevo.pagos_ventas)...
42          db.ventas.Add(nuevo);
43          db.SaveChanges();
44
45          foreach (productos_x_ventas item in nuevo.productos_x_ventas)
46          {
47              productos_x_tallas prodTalla = db.productos_x_tallas.Single(i => i.id == item.idProductoTalla);
48              prodTalla.cantidad -= item.cantidad;
49              db.SaveChanges();
50          }
51          List<String> to = db.usuarios_x_cargos_x_negocios.Include("usuarios").Where(i => i.idNegocio == nuevo.idNegocio
52                                                                  && i.estado == true && i.usuarios.token_id != null)
53                                                                  .Select(i => i.usuarios.token_id).ToList();
54          FirebaseMessage mensaje = new FirebaseMessage(to, 1);
55          new Firebase().sendData(mensaje);
56          return Ok(r.ok(nuevo.recibo));
57      }
58      catch (Exception ex)
59      {
60          return Ok(r.error(ex.Message));
61      }
62  }
```

Código 2.21 Registro de una venta

En el lado del cliente se inició codificando las interfaces gráficas detalladas en el apartado Esquema visual. Los nombres de los recursos, así como una breve descripción de su estructura se describen en la Tabla 2.8.

Tabla 2.8 *Layouts* para *sprint* 3

Layout	**Descripción**
`fragment_cliente`	Es la lista de clientes, cuenta con un `RecyclerView` y un `SearchView` para filtrar datos.
`fragment_cliente_detalle`	Fragmento para el registro y edición de una categoría.
`fragment_catalogo`	Es la lista de productos, cuenta con un `RecyclerView` y un componente `SearchView` para filtrar datos.
`catalogo_item`	Representa a cada ítem de la lista de productos.
`fragment_car`	Contiene la lista de productos seleccionados para vender, cuenta con el componente `RecyclerView`.
`producto_detalle_item`	Representa a cada ítem de la lista de productos en el carrito.
`fragment_datos_venta`	Fragmento para seleccionar el cliente y finalizar la venta.
`activity_pop_talla`	Actividad que permite seleccionar la talla y cantidad de un producto para vender.
`activity_pop_cliente`	Actividad para registrar un nuevo cliente.

Una vez implementadas las interfaces gráficas se procedió a escribir el código fuente para controlar las vistas de administración de clientes sobre las clases `ClienteFragment`, `ClienteDetalleFragment` y `ClienteAdapter`. El código de cada clase guarda una relación directa con sus similares implementados en el *sprint* 2 (ver Sección 2.4.3). Así también y con base en el detalle del trabajo previamente realizado se implementó el código de control para la lista principal de productos a través de la clase `CatalogoFragment` y del adaptador `CatalogoAdapter`.

Para la selección de la talla y cantidad de un producto para la venta se hizo uso de una actividad `PopTallaActivity` la cual inicia al seleccionar un *item* del catálogo como se muestra en el Código 2.22 a través del método `onClick`. En la línea 180 se obtiene el producto seleccionado mientras que las líneas 181 a 184 permiten definir a dicho producto como parámetro de entrada de la actividad `PopTallaActivity`, el método `startActivityForResult` en la línea 184 permite iniciar la misma en un segundo hilo

de ejecución mientras que el hilo principal se detiene en espera de que el hilo secundario finalice y devuelva un dato u objeto resultante.

```
179        public void onClick(View v) {
180            Producto producto = productos.get(Global.catalogoIndex);
181            Intent i = new Intent(getActivity().getBaseContext(), PopTallaActivity.class);
182            i.putExtra( name: "productoSel",producto);
183            i.putExtra( name: "accion", value: 1);
184            startActivityForResult(i,  requestCode: 1);
185        }
186    }
```

Código 2.22 Inicio de vista para selección de tallas

Por otro lado, el método `onClick` en la clase `PopTallaActivity` (ver Código 2.23), valida que la cantidad ingresada por el usuario sea menor al *stock* existente (línea 88) y agrega el producto seleccionado al carrito de compras (línea 89). Las líneas 90 a 92 permiten definir la respuesta (`Activity.RESULT_OK`) que se envía al hilo de ejecución principal y finalizar el hilo secundario.

```
86  public void onClick(View v) {
87      if (v.getId() == R.id.btnAceptarTalla){
88          if (validateFields()){
89              Global.agregarAcarrito(itemVenta);
90              Intent returnIntent = getIntent();
91              setResult(Activity.RESULT_OK, returnIntent);
92              finish();
```

Código 2.23 Cierre de vista para selección de tallas

El retorno al hilo principal, así como la captura de la respuesta mencionada se captura en el método `onActivityResult` (ver Código 2.24) de la clase `CatalogoFragment`, en este método se presentan o se esconden los botones de la barra superior (línea 193) para que con ello un usuario pueda acceder a la vista con la lista de productos seleccionados.

```
189     public void onActivityResult(int requestCode, int resultCode, Intent data) {
190         super.onActivityResult(requestCode, resultCode, data);
191         if(requestCode == 1 || requestCode == 3){
192             if (Global.productosAventa.size() != 0) {
193                 getActivity().invalidateOptionsMenu();
194             }
195         }
```

Código 2.24 Obtención de datos resultantes

En la implementación lógica de la lista de productos seleccionados (carrito de compras) se usó las clases `CarFragment` y `DetalleVentaAdapter`. En esta vista también se hizo de la clase `PopTallaActivity` para presentar una actividad al seleccionar un ítem de

la lista y permitir la modificación de datos si es necesario. El control de esta actividad siguió la misma estructura lógica detallada en los Código 2.22 a 2.24.

En la vista para la selección de clientes y métodos de pago representada con la clase `DatosVentaFragment` se implementó la búsqueda de clientes, como se muestra en el Código 2.25, donde se definen los parámetros de búsqueda de un cliente en un objeto `Extra` (líneas 126 a 128) y se realiza la petición hacia el servidor (línea 130), si existen coincidencias se transforma la información de arribo a una lista de objetos `Cliente` (línea 134) y dicha lista de objetos se presenta gráficamente (líneas 135 a 138). Si por el contario, el servidor no devuelve coincidencias se esconde la lista visual (línea 141) y se asigna la propiedad `idCliente` el valor de 0 (línea 142) que representa a consumidor final.

```
126     Extra filtro = new Extra();
127     filtro.setKey1(new LoginResponse().leerDatos().getIdNegocio());
128     filtro.setKey2(txtFiltro.getText().toString());
129
130     new Api().solicitud(Api.getService().buscarCliente(filtro), (data) → {
133             if (data != null && !data.isJsonNull()){
134                 clientes = new LinkedList<>(Arrays.asList(new Gson().fromJson(data,Cliente[].class)));
135                 recicler.setVisibility(View.VISIBLE);
136                 adapter = new ClienteAdapter2(clientes, getFragmentManager());
137                 recicler.setAdapter(adapter);
138                 adapter.setOnClickListener(new ClienteSelected());
139                 btnCancelar.setVisibility(View.VISIBLE);
140             }else {
141                 recicler.setVisibility(View.GONE);
142                 venta.setIdCliente(0);
143             }
```

Código 2.25 Búsqueda de clientes

Finalmente, en la clase `CatalogoFragment` se implementó un receptor de anuncios con la ayuda de la clase `BroadcastReceiver`, la cual permite manejar el arribo de un anuncio para la sincronización de datos, como se muestra en el Código 2.26.

```
232     private BroadcastReceiver receiver = (context, intent) → {
235             if (intent != null) {
236                 int accion = intent.getIntExtra( name: "accion", defaultValue: 0);
237                 Log.e( tag: "FIREBASE-UPDATE", msg: "llego a orden");
238                 mostrarDatos();
239             }
240     };
```

Código 2.26 Actualización de catálogo con FCM

2.6.4 ENTREGABLE

Al terminar este *sprint,* el prototipo permite a los usuarios (administradores o vendedores) registrar y administrar datos de sus clientes, así como armar pedidos de ventas seleccionando los productos deseados del inventario previamente registrado en el *sprint* 2,

su talla y número de ítems de venta. Una vez seleccionados todos los productos el usuario puede asociar la venta a un cliente o marcarla como venta para consumidor final. El prototipo también brinda la flexibilidad de elegir entre contado o crédito como método de pago.

2.7 SPRINT 4: VENTAS II

El objetivo de este *sprint* es complementar las acciones para el manejo de ventas, así como implementar la administración de pagos en cada una de ellas. Al finalizar las tareas, un usuario (administrador o vendedor) podrá revisar su historial de ventas y el listado de productos en cada una, valor total de venta y de ser el caso el valor adeudado. Finalmente, un usuario podrá registrar y modificar pagos realizados.

2.7.1 ESQUEMA GRÁFICO

En este *sprint* se han considerado los *sketckes* de la Figura 2.24, de izquierda a derecha se presenta los *sketches* con una lista de ventas generadas y el detalle de productos entregados en cada venta en el *sketch* 2. Para la administración de pagos se realizaron los *sketches* 3 y 4 los cuales muestran la lista de pagos realizados y el formulario para registrar un nuevo pago o modificarlo. Como ayuda visual los *sketches* 2, 3 y 4 cuentan con una barra al pie de la vista con el valor total de ventas, así como el monto adeudado de la misma, estos *sketches* también cuentan con una barra superior donde se alojan botones de acción que permiten la navegabilidad entre *sketches*.

Figura 2.24 *Sketches sprint* 4

En la Figura 2.25 se muestran las interacciones entre *sketches* para presentar el detalle de venta y para buscar ventas por fecha o rango de fechas con la ayuda de una vista que contiene un calendario.

Figura 2.25 *Wireframe* para administración de ventas

2.7.2 ESQUEMA CONCEPTUAL

Para las tareas propuestas en este *sprint* se hizo uso de las entidades `clientes`, `clientes_x_negocio`, `ventas`, `productos_x_venta`, `formas_pago` y `pagos_venta` de la base de datos.

A más de las clases que representan a las entidades mencionadas y los controladores correspondientes, se creó la clase `Reporte` (ver Figura 2.21) que contiene las propiedades `desde` y `hasta` para definir los rangos de fecha sobre los cuales se buscarán los registros de venta en el lado del servidor. El campo `tipo` se define siempre en 0, puesto que el mismo no tendrá utilidad en esta sección, pero ganará funcionalidad en la Sección 2.1.7.

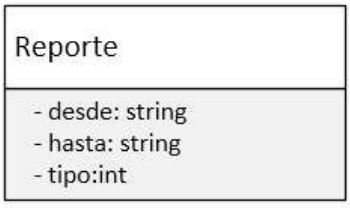


Figura 2.26 Clase `Reporte`

En el lado del cliente, también se replicó la clase `Reporte` definida en el lado del servidor, además se crearon las clases necesarias para controlar las interfaces gráficas y se creó una actividad para controlar la presentación de una vista para la selección de una fecha o rango de fechas con la ayuda de la librería `SlyCalendarView`.

2.7.3 DESARROLLO

El *sprint* comenzó con la codificación en el lado del servidor de los métodos que permiten controlar las peticiones `GET` de los controladores `VentaController` y `DetalleVentaController`, estos métodos permitirán poblar la lista de ventas y el detalle de productos por venta. El primer método mencionado se muestra en el Código 2.27, donde a partir del identificador de negocio enviado en la petición desde el cliente se busca en la base de datos registros coincidentes con ayuda de sentencias LINQ (línea 22), además con la ayuda del método `OrderByDescending` los registros obtenidos son ordenados de manera que el primer resultado en la lista sea la venta realizada más recientemente. Dado que al registrar la venta esta se guarda con una hora en formato UTC[19] fue necesario recorrer la hora registrada a la hora local de Ecuador (UTC - 5) como se muestra en la línea 25.

```
18      [HttpGet]
19      [Route("api/Venta/negocio")]
20      public IHttpActionResult VentasNegocio(int idNegocio)
21      {
22          List<ventas> ventas = db.ventas.Include("clientes").Where(i => i.idNegocio == idNegocio && i.estado == true)
23                                                              .OrderByDescending(i => i.fecha_registro).ToList();
24          foreach (ventas item in ventas) {
25              item.fecha_registro = item.fecha_registro.AddHours(-5);
26          }
27          return Ok(r.ok(ventas));
28      }
```

Código 2.27 Obtención del listado de ventas

En el controlador `VentaController` se implementó el método para responder a una petición `DELETE`, donde se elimina el registro respectivo, una venta no es borrada de la base de datos, sino que se modifica el campo `estado` de la misma. Dado que una venta no se puede modificar de manera directa no se implementó el método para controlar la acción `PUT` en este controlador.

Por otra parte, se creó el controlador `PagoController` sobre el cual se implementaron los métodos para controlar las acciones `GET`, `POST`, `PUT` y `DELETE` para permitir la administración de pagos. A la lógica principal de los métodos referentes a las acciones `POST`, `PUT` y `DELELE` se agregaron las líneas que permiten modificar el valor de los atributos `cancelado` y `adeudado` de la entidad `ventas` correspondiente.

[19] UTC (*Universal Time Coordinated*): es el principal estándar de tiempo para regular la fecha y hora mundial

En el lado del cliente se codificó las interfaces gráficas descritas en el apartado Esquema Visual, los nombres de las actividades (o fragmentos) y su función se describen en la Tabla 2.9.

Tabla 2.9 *Layouts* para *sprint* 4

Layout	**Descripción**
`fragment_venta`	Lista de ventas, cuanta con un `RecyclerView` y un componente `SearchView` para filtrar datos.
`fragment_venta_detalle`	Fragmento para el detalle de productos por venta, cuenta con un `RecyclerView` como elemento principal.
`fragment_pago`	Lista de pagos, cuenta con un `RecyclerView`.
`fragment_pago_detalle`	Fragmento para el registro o modificación de un pago, cuenta con un campo de texto para detallar el valor de pago y un componente `CalendarView` para seleccionar la fecha de registro del pago.

Una vez implementadas las interfaces gráficas, se implementó la lógica de control para la vista con la lista de ventas en las clases `VentaFragment` para obtener los registros de venta del servidor y `VentaAdapter` para controlar las acciones `onClick` y `swipeDetele` sobre cada *item* de la lista, los métodos implementados guardan similitud con el Código 2.14, Código 2.15, Código 2.16 y Código 2.17 del *sprint* 2.

De la misma manera, se implementaron las vistas para la administración de pagos con ayuda de las clases `PagoFragment`, `PagoDetalleFragment` y `PagoAdapter`, estos métodos permitieron presentar los registros de pagos de una venta, así como agregar un nuevo registro de pago, modificarlo o eliminarlo de ser necesario.

Finalmente, en este *sprint* se implementó la búsqueda por fechas en la vista con la lista de ventas para lo cual se usó la librería externa `SlyCalendarView` [34], la librería externa se usó en reemplazo de componentes nativos como el `CalendarView` y su uso se justifica debido a que el componente antes mencionado solo permite la selección de una fecha, en contraposición, la librería usada brinda la flexibilidad para la selección de un rango de fechas (ver Figura 2.27).

Figura 2.27 Implementación gráfica de `SlyCalendarView`

La creación y presentación del componente `SlyCalendarView` se muestran en el Código 2.28 donde destacan el método `setSingle` y `setCallback`. El primer método define si el calendario permite la selección de una sola fecha o de un rango de fechas, el segundo método permite implementar un hilo de escucha (`listener`) para obtener las fechas seleccionados una vez se presione el botón `Save` y se cierre el componente.

```
182             new SlyCalendarDialog().setSingle(false).setCallback(listener)
183                     .setHeaderColor(R.color.colorPrimary)
184                     .setSelectedColor(R.color.colorPrimary)
185                     .show(getActivity().getSupportFragmentManager(),  tag: "TAG_SLYCALENDAR");
```

Código 2.28 Inicialización del componente `SlyCalendarView`

La implementación del hilo de escucha mencionado se muestra en el Código 2.29, donde se tienen los métodos `onCancelled` y `onDataSelected` que se ejecutan al presionar los botones `Cancel` y `Save` respectivamente (ver Figura 2.27). Dentro del método `onDataSelected` se obtienen las fechas seleccionadas y con ellas se arma un objeto de tipo `Reporte` (líneas 173 a 175), para finalmente invocar al método `buscarPorFecha` (línea 202) el cual realiza una petición al servidor y presenta los registros retornados en la lista gráfica.

```
165     SlyCalendarDialog.Callback listener = new SlyCalendarDialog.Callback() {
166         @Override
167         public void onCancelled() {...}
170         @Override
171         public void onDataSelected(Calendar firstDate, Calendar secondDate, int hours, int minutes) {
172             if (firstDate != null) {
173                 desde = format.format(firstDate.getTime());
174             }
175             hasta = (secondDate == null) ? format.format(firstDate.getTime()): format.format(secondDate.getTime());
176             buscarPorFecha();
177         }
178     };
```

Código 2.29 Obtención de datos de `SlyCalendarView`

2.7.4 ENTREGABLE

Al finalizar este *sprint* el prototipo permite a los administradores o vendedores armar pedidos de venta, registrarlos en la base de dato, listar las ventas realizadas, buscar una venta por nombre del cliente asociado, número de *ticket* asignado o fecha de realización, además de presentar los detalles de la misma. El prototipo permite administrar los pagos realizados de una venta. Cada vez que se realiza o se elimina una venta, el prototipo automáticamente actualiza el inventario y sincroniza el mismo entre los diferentes dispositivos de los usuarios registrados en una empresa.

2.8 SPRINT 5: CONECTIVIDAD EXTERNA

Al finalizar este *sprint* el prototipo será capaz de emitir un comprobante al concluir la venta y hacer uso de la cámara del dispositivo móvil para la lectura de códigos de barra tanto para el registro de un producto como para la búsqueda de los mismos.

2.8.1 ESQUEMA GRÁFICO

Para este *sprint* se generaron los *sketches* presentados en la Figura 2.28. El primer *sketch* muestra tanto las impresoras vinculadas como las recientemente encontradas. El *sketch* 2 representa a la cámara trabajando como lector de códigos de barra, mientras que el *sketch* 3 es el fragmento con el catálogo (implementada en el *sprint* 3) al cual se agregó un botón flotante para iniciar el lector de código de barras.

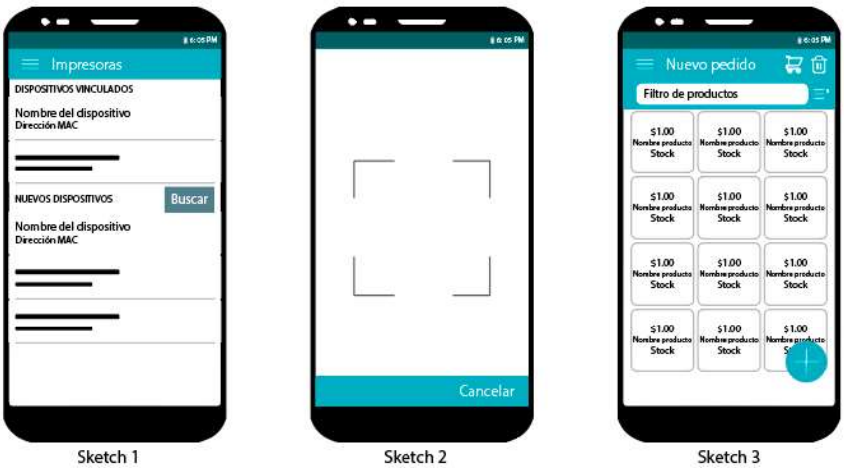


Figura 2.28 *Sketches* para el *sprint* 5

En la Figura 2.29 se muestra también la interacción para leer el código de barra y plasmarlo en la vista de registro o edición de productos previamente implementada en el *sprint* 2. Al

igual que en la vista con el catálogo, en esa vista se cuenta con un botón para iniciar el lector de códigos de barras.

Figura 2.29 *Wireframes* para lectura de códigos de barra

2.8.2 ESQUEMA CONCEPTUAL

La conexión con impresoras térmicas se realizó con la ayuda de la *librería Bluetooth Thermal Printer SDK*, de la cual, se hizo uso de la clase `BluetoothAdapter` para controlar el adaptador Bluetooth incorporado en el dispositivo móvil y administrar el uso de esta conexión desde el prototipo, la clase `BluetoothService` para manejar una conexión activa y cuya instancia representa a una impresora conectada la cual puede invocar el método `sendMessage` para imprimir texto y la clase `BluetoothDevice` que permite representar a dispositivos Bluetooth tanto encontrados a raíz de una nueva búsqueda como los previamente vinculados en el dispositivo móvil.

Para la lectura de códigos de barra se hizo uso de la librería ZXING de la cual su usaron las clases `IntentIntegrator` para inicializar el lector de códigos de barra e `IntentResult` para obtener el resultado de la lectura y pasarlo a la vista donde este dato será usado, es decir, a la vista con el catálogo para buscar el producto y añadirlo a la lista de productos seleccionados o a la vista de registro y edición de producto para registrar dicho código de barras en la base de datos.

2.8.3 DESARROLLO

En el lado del servidor se implementó el método para controlar las peticiones de tipo `GET` en el controlador `ProductoTallaController`, el cual permite buscar productos usando su código de barras. Este método se muestra en el Código 2.29 donde a partir del

identificador de negocio y el código de barras se filtran los productos coincidentes de la base de datos (línea 31), con estos datos se arma un objeto de tipo `productos_x_venta` (línea 36) y se retorna esta información al cliente (línea 44).

```
27      [HttpGet]
28      [Route("api/ProductoTalla/codigo")]
29      public IHttpActionResult Get(int idNegocio, string codigo)
30      {
31          productos_x_tallas itemTalla = db.productos_x_tallas.Include("productos").Single(i => i.codigo_barras == codigo
32                                                                  && i.estado == true && i.productos.idNegocio == idNegocio);
33          if (itemTalla != null)
34          {
35              itemTalla.talla = db.tallas.Single(i => i.id == itemTalla.idTalla);
36              productos_x_ventas itemVenta = new productos_x_ventas()...;
44              return Ok(r.ok(itemVenta));
45          }
46          else {
47              return Ok(r.ok());
48          }
49      }
```

Código 2.30 Filtro productos por código de barras

En el lado del cliente se codificaron las interfaces gráficas descritas en el apartado Esquema Visual. Los nombres de los fragmentos, actividades y su función se describen en la Tabla 2.10.

Tabla 2.10*Layouts* para *sprint* 5

Layout	**Descripción**
`activity_camara`	Permite abrir la cámara e iniciar el lector de código de barras.
`Catalogo_fragment`	Contiene un componente `FloatingButton` para iniciar la cámara y buscar productos por código de barras.
`fragment_impresora`	Fragmento con las listas de impresoras registradas y nuevas.

En el lado del cliente se inició con la activación de la conexión Bluetooth desde el dispositivo móvil en la clase `MainActivity`, como se muestra en el Código 2.31 donde en las líneas 95 y 96 se obtiene una instancia del adaptador Bluetooth, en la línea 97 se instancia el servicio Bluetooth que representa a una conexión activa a la cual se le asignó un hilo de escucha (`BluetoothHandler`) para manejar los diferentes eventos de una conexión, las líneas 98 y 99 inician una vista con un mensaje de confirmación para iniciar la conexión (ver Figura 2.30).

```
95      final BluetoothManager bluetoothManager = (BluetoothManager) getSystemService(Context.BLUETOOTH_SERVICE);
96      mBluetoothAdapter = bluetoothManager.getAdapter();
97      blueService= new BluetoothService( context: this,BluetoothHandler);
98      Intent enableIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
99      startActivityForResult(enableIntent, MainActivity.REQUEST_ENABLE_BT);
```

Código 2.31 Activación de conexión Bluetooth

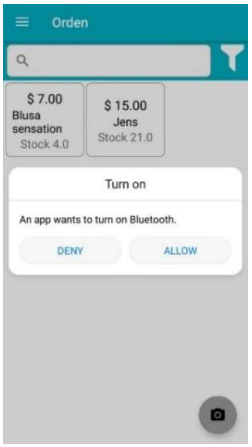


Figura 2.30 Confirmación de activación Bluetooth

La vista de la Figura 2.30 cuenta con dos botones `Deny` y `Allow` los cuales se controlan en el método `onActivityResult` de la clase `MainActivity` (ver Código 2.32). Si el botón presionado es el `Deny`, se ejecuta la sentencia `finish` de la línea 243 con la cual la aplicación se cierra mientras que si el botón presionado fue el `Allow` se ejecuta el método `contectarImpresora` (línea 241) con el cual la aplicación busca una impresora guardada localmente e intenta conectarse a la misma.

```
236    public void onActivityResult(int requestCode, int resultCode, Intent data) {
237        super.onActivityResult(requestCode, resultCode, data);
238        switch (requestCode) {
239            case REQUEST_ENABLE_BT:
240                if (resultCode == Activity.RESULT_OK) {
241                    conectarImpresora();
242                } else {
243                    finish();
244                }
245                break;
246        }
```

Código 2.32 Confirmación de conexión Bluetooth

Al iniciar la conexión *bluetooth* se inicia también un hilo de escucha para administrar los eventos de la conexión, como se muestra en el Código 2.33, donde se ejemplifican los eventos referentes a una conexión exitosa (línea 212), perdida de conexión (línea 218) y dispositivo fuera de rango (línea 221). Cada estado de la conexión genera una alerta al usuario. Además, como medida de comprobación al conectar correctamente una impresora imprime un texto con ayuda del método `sendMessage` (línea 213). En este método, a más de especificar el texto a imprimir, toma como parámetro de entrada el conjunto de caracteres aceptados, el estándar seleccionada fue el ISO-8859-1 pues este representa al alfabeto latino y permite el uso de caracteres propio del idioma como la ñ y vocales con tilde.

```
205    private final Handler BluetoothHandler = handleMessage(msg) → {
208            Intent intent;
209            switch (msg.what) {
210                case BluetoothService.MESSAGE_STATE_CHANGE:
211                    switch (msg.arg1) {
212                        case BluetoothService.STATE_CONNECTED:
213                            blueService.sendMessage( message: "\n\n", charset: "iso-8859-1");
214                            Toast.makeText(getApplicationContext(),  text: "Conección exitosa", Toast.LENGTH_SHORT).show();
215                            break;
216                    }
217                    break;
218                case BluetoothService.MESSAGE_CONNECTION_LOST:
219                    Toast.makeText(getApplicationContext(),  text: "Conección perdida", Toast.LENGTH_SHORT).show();
220                    break;
221                case BluetoothService.MESSAGE_UNABLE_CONNECT:
222                    Toast.makeText(getApplicationContext(),  text: "Imposible conectar", Toast.LENGTH_SHORT).show();
223                    break;
224            }
```

Código 2.33 Manejo de estados de conexión *bluetooth*

A continuación, se implementó la búsqueda de nuevos dispositivos en el fragmento `fragment_impresora` haciendo uso del método `startDiscovery` del servicio `blueService` establecido en la clase `MainActivity`. Para procesar la información de los nuevos dispositivos encontrados se emplea la clase (`BroadcastReceiver`) como se muestra en el Código 2.34, donde se obtiene el nuevo dispositivo (línea 100), este se añade a la lista de dispositivos y se lo presenta en la lista gráfica (línea 102 y 103).

```
95     private final BroadcastReceiver mReceiver = (context, intent) → {
98             String action = intent.getAction();
99             if (BluetoothDevice.ACTION_FOUND.equals(action)) {
100                BluetoothDevice device = intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);
101                if (device.getBondState() != BluetoothDevice.BOND_BONDED) {
102                    newDevices.add(device);
103                    newAdapter.notifyDataSetChanged();
104                }
105            }
106    };
```

Código 2.34 Captura de nuevos dispositivos

Para evitar que el usuario tenga que conectar una impresa al prototipo cada vez que ingresa al mismo se implementó la lógica mostrada en el Código 2.35, donde al seleccionar una impresora de la lista antes generada (Código 2.34) se guarda en el dispositivo móvil la información de dicha impresora (línea 45) para que en el próximo ingreso el prototipo se conecte a la impresora de manera automática (Código 2.32), las líneas 46 y 47 establecen la conexión con la impresora (líneas 46 y 47).

```
41  row.setOnClickListener((v) → {
44          MainActivity.macBluetooth = item.getAddress();
45          new LoginResponse().setMacAddress(item.getAddress());
46          MainActivity.blueDevice = MainActivity.blueService.getDevByMac(MainActivity.macBluetooth);
47          MainActivity.blueService.connect(MainActivity.blueDevice);
48  });
```

Código 2.35 Registro y conexión de dispositivo *bluetooth*

A continuación, se codificó la lógica para generar un comprobante de venta como se muestra en el Código 2.36, donde una vez registrada la venta en la base de datos (línea 187) se obtiene el número del recibo asignado (línea 190) y a partir de objetos de tipos `Venta`, `Cliente` y una colección de objetos de tipo `DetalleVenta` se arma el texto del comprobante con ayuda del método `modelado` de la clase `Ticket` (línea 192), la cadena de texto generada se imprime con la ayuda del método `sendData` (línea 195).

```
187  new Api().solicitud(Api.getService().postVenta(venta), (data) → {
190      venta.setRecibo(new Gson().fromJson(data, Integer.class));
191      venta.setFecha_registro(Conversion.currentStringDate());
192      String ticket = new Ticket().modelado((venta.getIdCliente() == 0)? null: cliente,
193                                            venta,Global.productosAventa);
194      try {
195          MainActivity.blueService.sendMessage(ticket, charset: "iso-8859-1");
196      }
197      catch (Exception ex){Mensaje.simple( mensaje: "No hay impresora", duracion: 1);}
198      Global.productosAventa = new ArrayList<>();
199      Fragment fragment = new CatalogoFragment();
200      getActivity().getSupportFragmentManager().beginTransaction().replace(R.id.content_main, fragment)
201                                                                  .addToBackStack(null).commit();
```

Código 2.36 Impresión de *ticket* de venta

Finalmente, para implementar la lectura de código de barras con el uso de la cámara del dispositivo móvil se agregó las líneas que se muestran en el Código 2.37, en el método `onCreate` de la clase `FullscreenActivity`. La clase `IntentIntegrator` es propia de la librería ZXING y el método `initiateScan` de la línea 76 inicia la lectura de código de barras.

```
74          final Activity activity = this;
75          IntentIntegrator integrator = new IntentIntegrator(activity);
76          integrator.initiateScan();
```

Código 2.37 Inicialización del lector de código de barras

Para obtener el resultado de la lectura se empleó la clase `IntentResult`, como se muestra en el Código 2.38, en las líneas 97 y 103 con el método `Result.getContents`, dado que este código se usa tanto para obtener un código de barras para el registro de un producto (`accion` =1) así como para buscar un producto del catálogo (`acción` = 2) se tiene el lazo `if-else` de las líneas 96 a 101. En el primer caso (bloque `if`) el código leído se devuelve a la vista para el registro de productos donde se procesa y se genera la solicitud respectiva al servidor, en el segundo caso (bloque `else`), se realiza una solicitud al servidor para buscar el producto con el código de barras obtenido (línea 103) y si existe un producto coincidente este se agrega al carrito de compras (líneas 107 a 109).

```
95          IntentResult Result = IntentIntegrator.parseActivityResult(requestCode , resultCode ,data);
96          if (accion == 1) {
97              Global.codigoBarras = Result.getContents();
98              Intent returnIntent = new Intent();
99              setResult(RESULT_OK, returnIntent);
100             finish();
101         }else{
102             LoginResponse log = new LoginResponse().leerDatos();
103             new Api().solicitud(Api.getService().getTallaProductoByCodigo(log.getIdNegocio(), Result.getContents()), (data) → {
106                     if (data != null){
107                         DetalleVenta aux = new Gson().fromJson(data, DetalleVenta.class);
108                         DetalleVenta itemVenta = new DetalleVenta(aux.getIdProductoTalla(),aux.getNombre_producto(),aux.getCantidad()
109                         Global.agregarAcarrito(itemVenta);
110                         Intent returnIntent = new Intent();
111                         setResult(RESULT_OK, returnIntent);
112                         finish();
113                     }else{
114                         Intent returnIntent = new Intent();
115                         setResult(RESULT_OK, returnIntent);
116                         finish();
```

Código 2.38 Resultado de la lectura de código de barras

2.8.4 ENTREGABLE

Al finalizar este *sprint* el prototipo permite buscar y conectar una impresora térmica Bluetooth, además de recordarla y buscarla cada vez que la aplicación inicia con el objeto de simplificar el proceso de conexión. Una vez conectada una impresora al prototipo, este permite imprimir comprobantes de venta con el detalle de la misma, datos del cliente, la empresa, entre otros.

2.9 SPRINT 6: REPORTES

Al finalizar este *sprint* un administrador podrá generar dos tipos de reportes de venta. Un reporte de venta general con el detalle del total vendido, el valor recibido y los valores adeudados; y un reporte por empleados (administrador y vendedores), que solo entrega el total vendido por cada empleado mostrando como primer resultado aquel que haya logrado el mayor volumen de ventas.

2.9.1 ESQUEMA GRÁFICO

En este *sprint* se han considerado los *sketckes* de la Figura 2.31, el *sketch* 1 está destinado para mostrar los resultados de un reporte general, cuenta con un componente `BarChart` para presentar de manera gráfica un historial del volumen de ventas mientras que el *sketch* 2 contiene una lista donde se presenta la lista de empleados y el correspondiente volumen de ventas.

En el *sketch* 1 se puede apreciar un botón `Reportes por empleados` el cual al presionar inicia una vista basada en el *sketch* 2, también cuenta con un botón ubicado en la barra superior derecha para abrir una actividad que permite seleccionar un rango de fechas para generar un reporte.

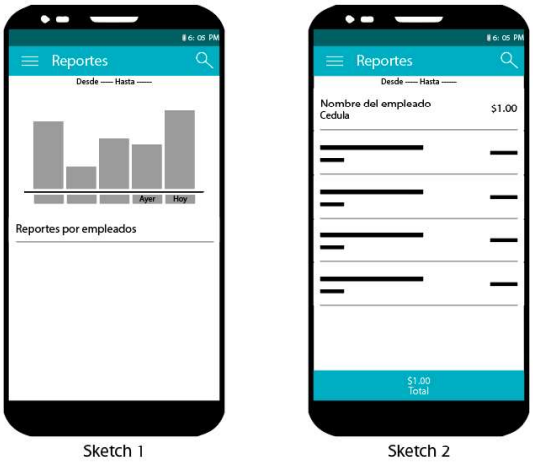


Figura 2.31 *Sketches* para reportes

2.9.2 ESQUEMA CONCEPTUAL

Para la generación de reportes se hizo uso de las entidades `ventas` y `clientes` de la base de datos. A diferencia de los pasados *sprints* en este solo se buscó información y no se implementaron métodos para generar ni modificar registros.

Tanto en el lado del servidor como del cliente se reutilizó la clase `Reporte` a la vez que se creó la clase `ReporteResponse` (ver Figura 2.32). La primera clase se usa para enviar desde el cliente los rangos de fecha seleccionados, así como el tipo de reporte deseado en la propiedad `tipo`. Siendo el 1 para solicitar reportes de venta generales y 2 para solicitar un reporte de venta por empleados. La clase `ReporteResponse` fue usada para enviar los resultados obtenidos desde el servidor e interpretarlos en el cliente, las propiedades de este objeto son de tipo `Object` con la finalidad de que en esta misma clase se pueda representar los dos tipos de reportes mencionados.

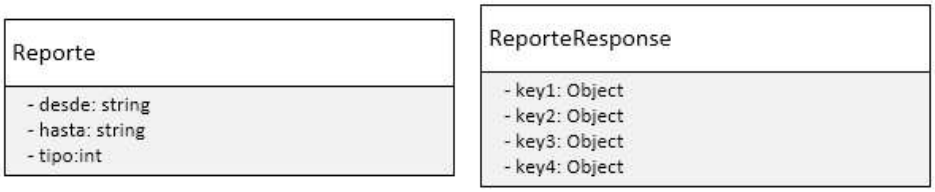


Figura 2.32 Clases para generación de reportes

En el lado del cliente, la implementación del diagrama de barras para el reporte general de ventas se lo hizo con ayuda de la librería MPAndoridChart dentro de la cual se usó el componente gráfico `BarChart` y las clases `BarChart, BarEntry, LegendEntry` y `BarDataSet`. La función de cada clase será detallada en la Sección 2.8.3.

2.9.3 DESARROLLO

En este *sprint* se implementó en lado del servidor el método `GenerarReporte` en el controlador `ReporteController`, al cual se llama a través de una petición de tipo `POST` y permite generar los dos tipos de reportes. La lógica para generar el reporte de ventas generales se muestra en el Código 2.39. Con el objetivo de presentar información más clara en el lado del cliente se incorporó un lazo `if` para obtener el listado de ventas del día actual (línea 34) esto genera una sola lista resultante, un lazo `if` para obtener el listado de ventas con un periodo de hasta una semana (línea 39) en donde se obtiene una colección de listas de ventas por día, y un último lazo `if` para periodos con más de una semana (línea 49) en donde también se obtendrá una sola lista de ventas como resultado. La línea 35 permite obtener el listado de ventas diarias en el periodo correspondiente donde se hizo uso de las funciones `TruncateTime` y `AddHours` para desplazar la hora originalmente en formato UTC a la hora local de Ecuador, en cuanto que la línea 38 identifican la fecha de cada listado de ventas obtenido. El lazo `foreach` de la línea 57 itera la lista de ventas obtenida para generar una lista de objetos de tipo `ReporteResponse` a la vez que da valor a cada uno de sus propiedades (línea 59). Esta lista de objetos `ReporteResponse` se envía al cliente (línea 69).

```
32      if (parametros.tipo == 1)
33      {//general vendido, adeudado y pagado
34          if (numeroDias == 0){
35              ventasDiarias.Add(db.ventas.Where(i => i.idNegocio == idNegocio &&
36                      DbFunctions.TruncateTime(DbFunctions.AddHours(i.fecha_registro,-5)) == DbFunctions.TruncateTime(desde))
37                      .ToList());
38              fechas.Add(desde.ToString("dd MMM"));
39          }
40          else if (numeroDias > 0 && numeroDias <= 7)...
49          else...
56          int c = 0;
57          foreach (var item in ventasDiarias)
58          {
59              ReporteResponse itemReporte = new ReporteResponse()...;
65              reportes.Add(itemReporte);
66              desde.AddDays(1);
67              c++;
68          }
69          return Ok(r.ok(reportes));
```

Código 2.39 Reporte general de ventas

Por otro lado, la lógica para obtener el reporte de ventas por empleado se presenta en el Código 2.40, donde se busca un listado de ventas en un rango de fechas (líneas 74 a 83). Con este listado resultante se agrupan las ventas por usuario y se crea un objeto de tipo `ReporteResponse` por cada grupo formado, es decir por cada empleado (línea 84). Esta lista de objetos al igual que en el primer tipo de reporte se envía al cliente (línea 94).

En el lado del cliente se implementó las interfaces gráficas descritas en el apartado Esquema visual según el detalle de la Tabla 2.11.

```
72  {//ventas por empleados
73
74      if (desde == hasta){
75          aux2 = db.ventas.Where(i => i.idNegocio == idNegocio &&
76                  DbFunctions.TruncateTime(DbFunctions.AddHours(i.fecha_registro, -5)) == DbFunctions.TruncateTime(desde))
77                  .ToList();
78      }else{
79          aux2 = db.ventas.Where(i => i.idNegocio == idNegocio &&
80                  DbFunctions.TruncateTime(DbFunctions.AddHours(i.fecha_registro, -5)) >= DbFunctions.TruncateTime(desde)
81                  && DbFunctions.TruncateTime(DbFunctions.AddHours(i.fecha_registro, -5)) <= DbFunctions.TruncateTime(hasta))
82                  .ToList();
83      }
84      List<ReporteResponse> listaDatos = aux2.GroupBy(i => i.idUsuario).Select(g=> new ReporteResponse()
85                                          { key1 = g.Key,
86                                            key2 = db.usuarios.Single(i =>i.id == g.Key).nombres,
87                                            key3 =new ReporteResponse()...
93                                           }).ToList();
94      return Ok(r.ok(listaDatos));
95  }
```

Código 2.40 Reportes de venta por empleado

Tabla 2.11 *Layouts* para *sprint* 6

Layout	**Descripción**
`fragment_empleado_reporte`	Vista para el reporte de ventas por empleado.
`fragment_reporte`	Fragmento de reportes generales de venta, cuanta con un componte `BarChart` para presentar los datos en un diagrama de barras.
`SlyCalendarView`	Actividad con calendario para la generación de reportes en una fecha o rango de fechas.

Una vez implementadas las interfaces gráficas se implementó la lógica necesaria para poblar el componente `BarChart` con los datos obtenidos como respuesta del servidor. Como se muestra en el Código 2.41 se representó los datos obtenidos desde el servidor en objetos de tipo `BarEntry` (línea 118) en el cual se fijó el valor de cada barra del gráfico, la clase `LegendEntry` (líneas 120 a 123) por otra parte se usó para definir la leyenda (fecha) de cada barra en el gráfico. Complementariamente se calcula el total de los valores monetarios cancelados y adeudados que se presentaran en la vista como datos extras al reporte (líneas 125 a 127).

Con la lista de objetos `BarEntry` previamente generada se configuró el componente `BarChar` y se asignaron los datos como se muestra en el Código 2.42. La línea 130 permite representar un área de trabajo con el conjunto de barras previamente generadas, la línea 131 genera un color diferente para cada barra como ayuda visual. Las líneas 133 a 135 añaden la lista de leyendas al componente gráfico mientras que en las líneas 136 y 138 se asigna el área de trabajo al componente y se imprimen todos los datos generados gráficamente.

```
109         List<String> fechas = new ArrayList<>();
110         List<BarEntry> entradas = new ArrayList<>();
111         List<LegendEntry> leyendas = new ArrayList<>();
112         int[] colors = ColorTemplate.JOYFUL_COLORS;
113         int i =0;
114         float total = 0F;
115         float adeudado = 0F;
116         float cancelado = 0F;
117         for (ReporteResponse item: respuesta) {
118             entradas.add(new BarEntry(i,Float.valueOf(item.getKey1().toString())));
119             fechas.add(item.getKey4().toString());
120             LegendEntry leg = new LegendEntry();
121             leg.formColor = colors[i];
122             leg.label = item.getKey4().toString();
123             leyendas.add(leg);
124             i++;
125             total += Float.valueOf(item.getKey1().toString());
126             cancelado += Float.valueOf(item.getKey2().toString());
127             adeudado += Float.valueOf(item.getKey3().toString());
128         }
```

Código 2.41 Preparación de datos para reporte

```
130         BarDataSet sets= new BarDataSet(entradas, label: "");
131         sets.setColors(ColorTemplate.JOYFUL_COLORS);
132         BarData data = new BarData(sets);
133         Legend l = barChart.getLegend();
134         l.setEnabled(true);
135         l.setCustom(leyendas);
136         barChart.setData(data);
137         barChart.getDescription().setText("");
138         barChart.invalidate(); // refresh
139
140         txtTotal.setText("$"+ total);
141         txtCancelado.setText("$"+ cancelado);
142         txtAdeudado.setText("$"+ adeudado);
```

Código 2.42 Asignación de datos a `BarChar`

2.9.4 ENTREGABLE

Al finalizar este *sprint*, se obtuvo el prototipo final, donde el administrador de un negocio puede registrar los datos de los mismo y sus datos personales para registrarlo en el prototipo. Una vez iniciada la sesión, el prototipo dispone de un menú de navegación lateral desde el cual se accede a las interfaces para la administración de categorías, productos, clientes, datos de empresa y datos personales. Con la ayuda de una impresora térmica, se pueden emitir comprobantes de venta al finalizar la misma. Además, el prototipo permite obtener reportes de venta con los registros almacenados en la base de datos.

3. RESULTADOS Y DISCUSIÓN

En este capítulo se presentan los resultados de las pruebas de integración y aceptación realizadas. Las pruebas de integración se llevaron a cabo con el fin de verificar el correcto funcionamiento de los componentes de *backend* y *frontend* desarrollados, así como el intercambio de información entre los mismos. Las pruebas de aceptación, por otro lado, tuvieron el objetivo de corroborar el funcionamiento del prototipo desde la perspectiva del usuario, para estas pruebas se eligieron a 10 administradores de locales comerciales dedicados a la venta de ropa y calzado, quienes usaron el prototipo y sus experiencias fueron recolectadas a través de una entrevista.

Además, al final de este capítulo se presentan los errores encontrados en el proceso de desarrollo y en las pruebas, así como sus respectivas soluciones. También se detallan las modificaciones realizadas en el prototipo.

3.1 PRUEBAS DE INTEGRACIÓN

Las pruebas de integración se realizaron con base en la lista de requisitos planteada en la Sección 2.1.2. Para algunas de las pruebas se hizo uso de la librería Logging Interceptor del lado del cliente Android la cual funciona como un *sniffer* [20] para capturar el tráfico entrante y saliente en el dispositivo móvil, el resultado de la captura se presenta en la pestaña `RUN` de Android Studio, con esto se pudo evidenciar las peticiones realizadas así como la información embebida en el cuerpo de las peticiones de tipo `POST` y `PUT` como se ejemplifica en la Figura 3.1, con una petición para modificar datos de usuario.

```
D/OkHttp: --> PUT http://titulacionpoli.somee.com/api/usuario?id=4 http/1.1
    Content-Type: application/json; charset=UTF-8
    Content-Length: 318
D/OkHttp: {"celular":"0984743739","estado":true,"id":4,"idCargo":0,"idNegocio":0,"identificacion":"1725450108"
    --> END PUT (318-byte body)
```

Figura 3.1 Captura de tráfico con Logging Interceptor

En el lado del servidor, por su parte, se hizo uso de la extensión Restlet Client de Google Chrome para corroborar la correcta ejecución de los métodos expuestos en el *backend*, y de la aplicación Microsoft SQL Manager Studio para verificar la persistencia de información en la base de datos.

[20] *Sniffer*: Es un producto que permite capturar tramas que circulan en una red informática.

3.1.1 REGISTRO E INICIO DE SESIÓN

Las pruebas realizadas sobre la vista para el registro de datos de usuario y de negocio fueron dos, por un lado, se comprobó que en dicha vista no permita campos vacíos ni datos erróneos, como se ejemplifica en la Figura 3.2. Para esta prueba se ingresó como número de RUC una secuencia de 10 dígitos que no finalizaba con los caracteres 001, ante esto se presentó una advertencia con el mensaje RUC inválido.

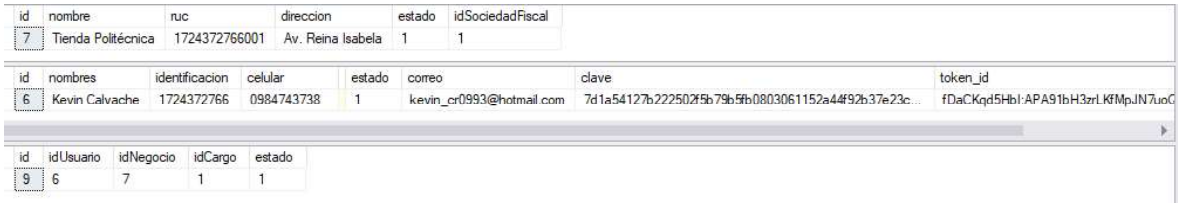
Figura 3.2 Registro de usuario y negocio

Por otro lado, se verificó el procesamiento de datos entre el cliente Android y el servidor. Al ingresar datos en la vista se envían paquetes de tipo `POST` con los datos recolectados en la misma. En la Figura 3.3 se muestra la captura del paquete para el registro de datos de negocio.

```
D/OkHttp: --> POST http://titulacionpoli.somee.com/api/negocio http/1.1
D/OkHttp: Content-Type: application/json; charset=UTF-8
D/OkHttp: Content-Length: 128
D/OkHttp: {"direccion":"Av. Reina Isabel","estado":false,"id":0,"idSociedadFiscal":1,"nombre":"Tienda Politécnica","ruc":"1724372766001"}
D/OkHttp: --> END POST (128-byte body)
```

Figura 3.3 Envío de paquete para registro de negocio

El servidor procesó la información recibida y la guardó en tres tablas de la base de datos. En la Figura 3.4 se muestran los registros almacenados en las tablas (de arriba hacia abajo) `negocios`, `usuarios` donde destacan los atributos `clave` cuyo valor representa una cadena de texto cifrada con el algoritmo SHA256 y el atributo `token_id` donde se almacena el identificador del dispositivo móvil para controlar la sincronización de datos entre dispositivos con la ayuda del servicio Firebase Cloud Messaging (FCM). Finalmente, el último registro se encuentra en la tabla `usuario_x_cargo_x_negocio` y permite la relación entre un negocio y un usuario (administrador o vendedor).



id	nombre	ruc	direccion	estado	idSociedadFiscal
7	Tienda Politécnica	1724372766001	Av. Reina Isabela	1	1

id	nombres	identificacion	celular	estado	correo	clave	token_id
6	Kevin Calvache	1724372766	0984743738	1	kevin_cr0993@hotmail.com	7d1a54127b222502f5b79b5fb0803061152a44f92b37e23c...	fDaCKqd5Hbl:APA91bH3zrLKfMpJN7uoC

id	idUsuario	idNegocio	idCargo	estado
9	6	7	1	1

Figura 3.4 Registro de usuario y negocio en base de datos

Para el inicio de sesión la aplicación generó una petición `POST: api/usuario/login` al servidor con el correo y contraseña ingresados por el usuario, si los datos son correctos el servidor arma una lista con información de los negocios y cargos del usuario (ver Figura 3.5) y se la envía de retorno al cliente Android, donde se almacenan con la ayuda del proveedor de contenido `SharedPreferences`. De esta información almacenada se destaca el identificador del usuario (`idUsuario`), identificador del negocio (`idNegocio`) e identificador del cargo (`idCargo`) que ostenta en dicho negocio.


```
info : [
    {
        idUsuario : 6,
        idNegocio : 7,
        idCargo : 1,
        valorImpuesto : 0.12,
        nombresUsuario : "Kevin Calvache",
        nombreNegocio : "Tienda Politécnica",
        celularNegocio : "0984743738",
        direccionNegocio : "Av. Reina Isabela",
        rucNegocio : "1724372766001"
    }
]
```


Figura 3.5 Datos de inicio de sesión

3.1.2 ADMINISTRACIÓN DE DATOS DE NEGOCIO

Las pruebas de integración que se realizó en esta sección fueron para comprobar la modificación de datos de negocio y dar de baja el mismo. Para la primera prueba se comprobó que la aplicación genera una llamada al método `PUT: api/negocio?id=7` en el cual se encuentra embebida la nueva información del negocio en el cuerpo del mismo al presionar el botón `Modificar` de la vista que presenta el detalle de datos de empresa (ver Figura 3.6).

Para esta prueba, en específico, se cambió el nombre del negocio de “Tienda Politécnica” a “Almacén Politécnico”.

Figura 3.6 Presentación de datos de negocio

La Figura 3.7 presenta una comparativa del registro de negocio correspondiente en la base de datos antes (arriba) y después (abajo) de llamar al método en cuestión, donde se evidencia el cambio de información en el campo `nombre`.

id	nombre	ruc	direccion	estado	idSociedadFiscal
7	Tienda Politécnica	1724372766001	Av. Reina Isabela	1	1

id	nombre	ruc	direccion	estado	idSociedadFiscal
7	Almacén Politécnico	1724372766001	Av. Reina Isabela	1	1

Figura 3.7 Modificación de datos de negocio

Para validar la interacción entre el cliente Android y el servidor ante una solicitud para eliminar un negocio, se comprobó que la aplicación genere una petición HTTP de tipo `DELETE` como se muestra en la Figura 3.8.

```
D/OkHttp: --> DELETE http://titulacionpoli.somee.com/api/negocio?id=7 http/1.1
D/OkHttp: --> END DELETE
W/libEGL: EGLNativeWindowType 0x7f55b17010 disconnect failed
V/InputMethodManager: Reporting focus gain, without startInput
D/OkHttp: <-- 200 OK http://titulacionpoli.somee.com/api/negocio?id=7 (688ms)
```

Figura 3.8 Solicitud para eliminar negocio

En el servidor los datos fueron procesados y el campo `estado` del registro correspondiente (ver Figura 3.7) se cambió a `false` para impedir nuevos inicios de sesión a usuarios de ese negocio.

3.1.3 ADMINISTRACIÓN DE DATOS DE USUARIO

En esta sección se presentan los resultados de la edición de datos de usuario. Para esto, en el fragmento donde se presenta el detalle de datos de usuario (ver Figura 3.9) al

presionar el botón `Modificar` se comprobó que la aplicación genere una solicitud `PUT:api/usuario?id=6` (ver Figura 3.10) al servidor y que este, genere su respectiva respuesta.

Figura 3.9 Presentación de datos de usuario

```
D/OkHttp: --> PUT http://titulacionpoli.somee.com/api/usuario?id=6 http/1.1
D/OkHttp: Content-Type: application/json; charset=UTF-8
D/OkHttp: Content-Length: 481
D/OkHttp: {"celular":"0984743738","clave":"7d1a54127b222502f5b79b5fb0803061152a44f92b37e23c6527baf665d4da9a","correo":"kevin.calvachi@epn.edu.ec",
D/OkHttp: --> END PUT (481-byte body)
V/FA: Inactivity, disconnecting from the service
D/OkHttp: <-- 200 OK http://titulacionpoli.somee.com/api/usuario?id=6 (462ms)
```

Figura 3.10 Solicitud para modificación de datos de usuario

En el servidor la solicitud fue procesada y la nueva información fue almacenada en la base de datos. Para esta prueba en particular, se modificó el correo registrado de "kevin_cr0993@hotmail.com" a "kevin.calvachi@epn.edu.ec". Una comparativa del registro antes (arriba) y después (abajo) de realizar el llamado a la acción `PUT` se presenta en la Figura 3.10

id	nombres	identificacion	celular	direccion	estado	correo	clave	token_id
6	Kevin Calvache	1724372766	0984743738	NULL	1	kevin_cr0993@hotmail.com	7d1a54127b222502f...	fDaCKqd5HbI:APA91bH3...

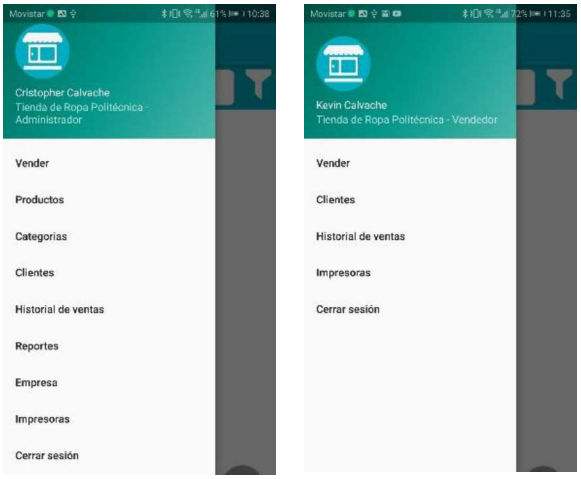
id	nombres	identificacion	celular	direccion	estado	correo	clave	token_id
6	Kevin Calvache	1724372766	0984743738	NULL	1	kevin.calvachi@epn.edu.ec	7d1a54127b222502f5...	fDaCKqd5HbI:APA91bH...

Figura 3.11 Modificación de datos de usuario

3.1.4 DESPLIEGUE DE MENÚ PRINCIPAL

Con base en la variable `idCargo` que llegó al cliente Android en el inicio de sesión (ver Figura 3.5), el menú principal presenta las opciones mostradas en la parte a de la Figura 3.12 si dicha variable representa a un usuario administrador, o las opciones presentadas en la parte b de la misma si la variable `idCargo` representa a un usuario vendedor.

Además, en la cabecera de los mismo se encuentra el nombre del negocio, nombre del usuario y cargo.



(a) (b)

Figura 3.12 Menú principal

3.1.5 CERRAR SESIÓN

El cierre de sesión es la última opción en el menú principal. Al seleccionar dicha opción se verificó que la aplicación envíe una petición `POST: api/usuario/logout` con los datos del usuario. El servidor, por su parte, elimina el identificador del servicio FCM registrado en la base de datos en el campo `token_id` de la tabla `usuario`, como se muestra en la Figura 3.13.

id	nombres	identificacion	celular	direccion	estado	correo	clave	token_id
6	Kevin Calvache	1724372766	0984743738	NULL	1	kevin.calvachi@epn.edu.ec	7d1a54127b222502f5b79b5fb0803061152a44f92b37e23c...	NULL

Figura 3.13 Cierre de sesión en base de datos

3.1.6 ADMINISTRACIÓN DE CATEGORÍAS

En la Figura 3.14 se presentan los datos ingresados por el usuario en la vista para el registro de categorías.

Figura 3.14 Información para registro de categorías

Al presionar el botón `Registrar`, el cliente Android genera una petición de tipo `POST` en tanto que el servidor procesa la información recibida y la guarda en la base de datos. La Figura 3.15 presenta el registro almacenado en la tabla `categorias` como resultado de la petición realizada.

id	nombre	idNegocio	estado	idTipoTalla
15	Blusas y tops	7	1	4

Figura 3.15 Registro de categoría generado

Dichos registros fueron recuperados al iniciar la vista con la lista de categorías mediante un solicitud `GET: api/categoria/negocio?id=7` al servidor y presentados gráficamente, como se muestra en la Figura 3.16.

Figura 3.16 Recuperación de registros de categorías

3.1.7 ADMINISTRACIÓN DE PRODUCTOS

En esta sección se presentan los resultados de la prueba realizadas en las vistas de la Figura 3.17. En la parte a de la figura se presenta la vista para agregar datos generales de un producto, como su nombre y descripción; mientras que la vista de la parte b permite ingresar los datos del precio, talla y cantidad de ítems disponibles.

(a) (b)

Figura 3.17 Vistas para registro de productos y tallas

Al presionar el botón `Registrar` (Figura 3.17 parte b) el cliente Android genera dos peticiones `POST`, para el registro del producto y talla respectivamente. La Figura 3.18 presenta los registros almacenados en la base de datos, el registro de la parte superior se encuentra en la tabla `productos` y contiene la información ingresada en la parte a de la Figura 3.17, el registro de la parte inferior por otro lado se encuentra en la tabla `productos_x_tallas`, la cual permite relacionar una talla y un producto.

id	nombre	descripcion	idCategoria	idNegocio	estado
20	Blusa casual	Ropa de la linea formal y semi formal	15	7	1

id	idProducto	idTalla	cantidad	estado	subtotal	iva	total	codigo_barras
31	20	31	10	1	13.39	0.12	15.00	

Figura 3.18 Registros de productos y talla

Para la recuperación de productos, el cliente Android genera una petición `GET: api/producto/negocio?id=7` y el servidor retorna como respuesta el listado de productos en formato JSON (ver Figura 3.19), dentro de la cada ítem de la lista se encuentra un objeto `extra` el cual a su vez tiene las claves `key1` y `key2` que albergan el menor precio del producto y el número de prendas disponibles.


```
info : [
    {
        id : 20,
        nombre : "Blusa casual ",
        descripcion : "Ropa de la linea formal y semi formal ",
        idCategoria : 15,
        idNegocio : 7,
        estado : true,
        extra : {
            key1 : 15,
            key2 : 18
        },
        categorias : null,
        negocios : null,
        productos_x_tallas : [
            {id : 31, idProducto : 20, idTalla : 31, cantidad : 10, estado : true,…},
            {id : 32, idProducto : 20, idTalla : 32, cantidad : 8, estado : true,…}
        ]
    }
]
```


Figura 3.19 Listado de productos en formato JSON

La información que llega del servidor en formato JSON es procesada y transformada en objetos, para finalmente presentarlos de forma gráfica en una lista, como se muestra en la Figura 3.20. De esta imagen se destaca el uso de la información contenida en la clave `key1` (ver Figura 3.19) como precio referencial en cada fila de la lista.

Figura 3.20 Recuperación de registros de productos

3.1.8 BÚSQUEDA DE PRODUCTOS

En esta sección se presentan los resultados de las pruebas realizadas en la vista de la Figura 3.21 donde se usó el componente `SearchView` en la parte superior de la misma para la búsqueda de productos por nombre y el botón contiguo para la búsqueda por categorías.

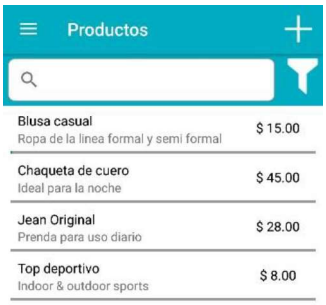


Figura 3.21 Búsqueda de producto por nombre

Para la búsqueda por nombre se escribe un fragmento del nombre en el componente `SearchView`, los resultados se presentan en la Figura 3.22.

Figura 3.22 Búsqueda de productos por nombre

Por otro lado, para la búsqueda por categorías se selecciona la opción deseada en la vista con el listado de categorías disponibles como se muestra en la Figura 3.23.

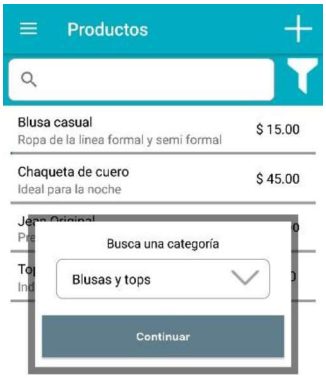


Figura 3.23 Filtro de productos por categoría

Al presionar el botón `Continuar` la lista de productos se actualiza y presenta los productos pertenecientes a la categoría seleccionada (ver Figura 3.24).

Figura 3.24 Resultado de la búsqueda por categorías

3.1.9 ADMINISTRACIÓN DE CLIENTES

La primera prueba realizada en esta sección fue para comprobar el registro de un cliente en la base de datos, para esto en la vista de la Figura 3.25 al presionar el botón `Registrar` se realiza una petición de tipo `POST: api/cliente` con los datos ingresados como se presentan en la Figura 3.26.

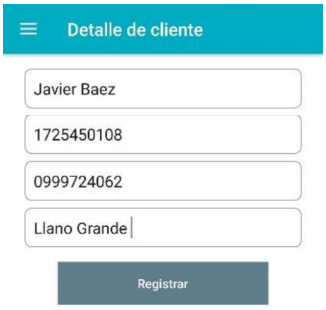


Figura 3.25 Vista para registro de clientes

```
D/OkHttp: --> POST http://titulacionpoli.somee.com/api/cliente http/1.1
    Content-Type: application/json; charset=UTF-8
    Content-Length: 141
D/OkHttp: {"celular":"0999724062","direccion":"Llano Grande","estado":false,"id":0,"idNegocio":7,"identificacion":"1725450108","nombres":"Javier Baez"}
D/OkHttp: --> END POST (141-byte body)
```

Figura 3.26 Petición para registro de clientes

El servidor, procesa la información y genera dos registros en la base de datos, el primero en la tabla `clientes` (parte superior de la Figura 3.27) y el segundo en la tabla `clientes_x_negocios` (parte inferior de la Figura 3.27) para relacionar el cliente registrado con un negocio.

id	nombres	identificacion	celular	direccion	estado
6	Javier Baez	1725450108	0999724062	Llano Grande	1

id	idCliente	idNegocio	estado
9	6	7	1

Figura 3.27 Registro de cliente en base de datos

Una vez comprobado el registro de cliente se procede a realizar las pruebas para recuperar dichos registros, al iniciar la vista con la lista de clientes, la aplicación envía al servidor una petición `GET: api/cliente/negocio?id=7`, los datos que se envian en formato JSON son transformados en objetos y presentados en la lista gráfica, como se muestra en la Figura 3.28

Figura 3.28 Recuperación de registros de clientes

En cada fila de la lista presentada en la Figura 3.28 se tiene la acción desplazar para borrar (`swipeToDelete`), al ejecutar esta acción el cliente Android genera una petición `DELETE: api/cliente/negocio?idCliente=6&idNegocio=7`, la cual provoca que en el registro correspondiente en la tabla `clientes_x_negocios` de la base de datos cambie el campo `estado` a `false (0)` como se muestra en la Figura 3.29.

id	idCliente	idNegocio	estado
9	6	7	0

Figura 3.29 Eliminar registro de cliente

3.1.10 ADMINISTRACIÓN DE VENTAS

En esta sección se corroboró la funcionalidad para armar una venta, registrarla en la base de datos, recuperar los registros y eliminar los mismos, de forma simultánea también se

comprobó el funcionamiento del servicio Firebase Cloud Messaging (FCM) verificando la reducción del *stock* de un producto al finalizar una venta.

Para armar una venta se selecciona un producto y se define la talla, la cantidad deseada y el precio final de venta de dicho artículo en la vista que se presenta en la Figura 3.30. Nótese que el producto seleccionado fue una “Blusa casual” en talla “M” con un *stock* inicial de 10 prendas y se escogen 2 unidades.

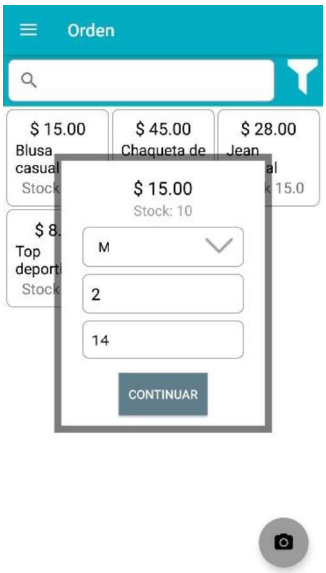


Figura 3.30 Selección de un producto para venta

Una vez armado el pedido y seleccionado el cliente a quien se realiza la venta, la aplicación genera una petición `POST: api/venta` con los productos seleccionados embebidos en el cuerpo de dicha petición como se muestra en la Figura 3.31, para guardar la venta en la base de datos.

```
D/OkHttp: --> POST http://titulacionpoli.somee.com/api/venta http/1.1
    Content-Type: application/json; charset=UTF-8
    Content-Length: 526
D/OkHttp: {"adeudado":0.0,"cancelado":27.999999999999996,"estado":true,"id":0,"idCliente":6,"idFormaPago":1,"idNegocio":7,"idUsuario":6,"iva":3.3599999999999994,
 "pagos_ventas":[{"estado":true,"id":0,"idResponsable":6,"idVenta":0,"valor":27.999999999999996}],"productos_x_ventas":[{"cantidad":2,"estado":true,"id":0,
 "idProductoTalla":31,"idVenta":0,"iva":2.999999999999996,"nombre_producto":"M-Blusa casual ","subtotal":24.999999999999996,"total":27.999999999999996}],"recibo":0,
 "subtotal":24.999999999999993,"total":27.999999999999996}
    --> END POST (526-byte body)
```

Figura 3.31 Petición para registro de venta

Al procesar la petición antes mencionada se generan 3 registros en la base de datos como se muestra en la Figura 3.32, de arriba hacia abajo se presenta el registro almacenado en la tabla `ventas` donde se encuentra la información general de una venta, en este registro destaca el campo `recibo` que es el número de *ticket* que la base de datos asigna a la

misma, el segundo registro corresponde a la tabla `productos_x_ventas` donde se encuentra el detalle de cada producto vendido mientras que el tercer registro corresponde a la tabla `pagos_ventas`.

id	subtotal	iva	total	idUsuario	idNegocio	fecha_registro	estado	recibo	idFormaPago	cancelado	adeudado	idCliente
75	25.00	3.36	28.00	6	7	2019-09-05 14:38:46.027	1	2	1	28.00	0.00	6

id	idProductoTalla	nombre_producto	cantidad	subtotal	iva	total	estado	idVenta
93	31	M-Blusa casual	2	25.00	3.00	28.00	1	75

id	valor	idVenta	idResponsable	fecha_registro
79	28.00	75	6	2019-09-05 14:38:46.027

Figura 3.32 Registro de venta en base de datos

Una vez registrada la venta en el base de datos y con el objetivo de verificar la integración del servicio FCM en la vista con el catálogo de productos, se selecciona nuevamente el *item* “Blusa casual” y se comprobó que el *stock* se redujo en 2 unidades como se muestra en la Figura 3.32

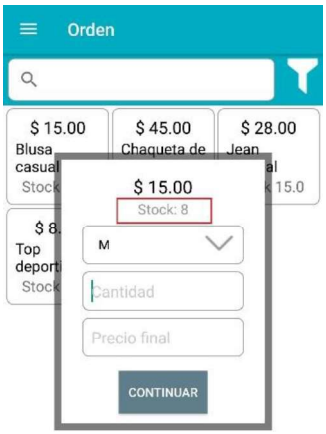


Figura 3.33 Reducción de stock

Para recuperar los registros de venta y detalle de cada una la aplicación genera las solicitudes `GET: api/venta/negocio?idNegocio=7` y `GET: api/detalleVenta?id=75` respectivamente, la información que retorna del servidor en cada solicitud se presenta gráficamente en las vistas de las Figuras 3.34 y 3.35.

Figura 3.34 Presentación de registros de venta

Figura 3.35 Presentación de detalle de venta

Finalmente, en cada fila de la lista presentada en la Figura 3.34 se tiene la acción desplazar para borrar (`swipeToDelete`), al ejecutar esta acción el cliente Android genera una petición `DELETE: api/venta?id=75,` la cual provoca que el registro correspondiente en la tabla `ventas` de la base de datos cambie el campo `estado` a `false` a la vez que el *stock* de cada producto vendido incrementa como se muestra en la Figura 3.29 donde después de eliminar la venta generada (ver Código 3.31) el número de prendas en *stock* del producto “Blusa casual” vuelve a ser 10.

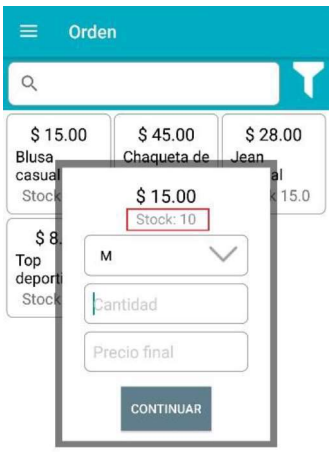


Figura 3.36 Reposición de stock al eliminar venta

3.1.11 ADMINISTRACIÓN DE PAGOS

La primera prueba realizada en esta sección fue verificar la recuperación y presentación de registros en el lado del cliente.

Para esto, el cliente al iniciar la vista con la lista de registros genera una petición `GET: api/pago?idVenta=74`, la información que el servidor retorna se procesa y presenta en la vista mencionada, un ejemplo del registro de pagos se muestra en la Figura 3.37.

Figura 3.37 Presentación de registros de pagos

Previo al registro de un nuevo pago, se verificó que los algoritmos de control eviten el ingreso de fechas y valor de pago incorrecto. En la Figura 3.38 se ejemplifica el primer algoritmo de control, donde en el calendario se seleccionó una fecha posterior a la fecha actual y al presionar el botón `Registrar` se presentó un mensaje de error con el texto "Fecha inválida".

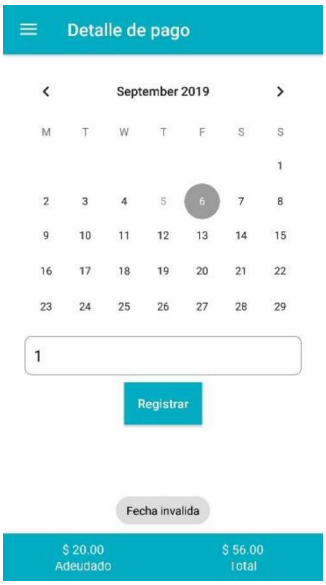


Figura 3.38 Control de fecha para registro de pago

Por otro lado, el control para el valor de pago incorrecto se presenta en la Figura 3.39, donde al pie de la misma se aprecia que el valor adeudado es de 20 dólares, mientras que el valor ingresado fue de 25 dólares y al presionar el botón `Registrar` se presentó un mensaje de error con el texto “Valor de pago incorrecto”.

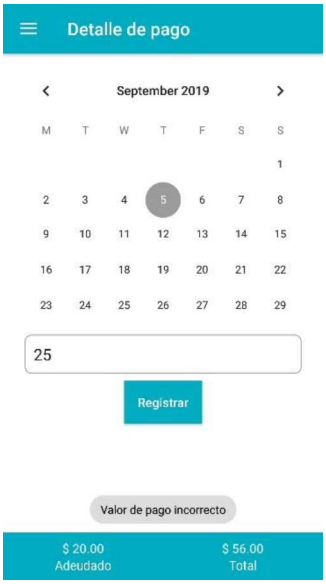


Figura 3.39 Control para valor de pago

Al seleccionar el botón Registrar el cliente Android ejecuta la petición para el registro de pago, el servidor al recibir dicha petición genera un registro en la tabla `pagos_ventas` de la base de datos como se muestra en la Figura 3.40

id	valor	idVenta	idResponsable	fecha_registro
90	20.00	76	6	2019-10-16 14:17:04.623

Figura 3.40 Registro de pago en base de datos

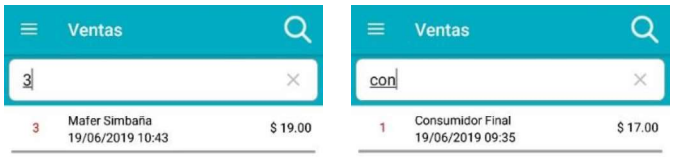
A la vez que se comprobó el registro de pagos, se corroboró la actualización del campo `adeudado` en el registro correspondiente en la tabla `ventas`. El registro de venta se muestra en la Figura 3.41 donde se aprecia que en el campo `adeudado` contiene un nuevo valor correspondiente a la resta del valor adeudado inicial (20 dólares) menos el valor del nuevo pago (20 dólares).

id	subtotal	iva	total	idUsuario	idNegocio	fecha_registro	estado	recibo	idFormaPago	cancelado	adeudado	idCliente
76	50.00	6.72	56.00	6	7	2019-09-05 21:26:52.247	1	3	2	56.00	0.00	6

Figura 3.41 Actualización de valor adeudado de venta

3.1.12 BÚSQUEDA DE VENTAS

Las opciones de búsqueda se implementaron con la ayuda del componente `SearchView`. En la Figura 3.42 se presentan los resultados de la búsqueda de ventas por número de *ticket* (parte a) y por nombre de cliente (parte b) donde se puede notar que solo es necesario un fragmento del número de *ticket* o del nombre para realizar la búsqueda.



(a) (b)

Figura 3.42 Búsqueda de ventas

3.1.13 EMISIÓN DE COMPROBANTES DE VENTA

El *ticket* emitido al finalizar una venta se muestra en la Figura 3.43, el mismo contiene los datos tanto del negocio, así como del cliente, vendedor y detalle de productos.

Figura 3.43 *Ticket* impreso con impresora *bluetooth*

3.1.14 LECTURA DE CÓDIGOS DE BARRA

La vista implementada para la lectura de un código de barras se muestra en la Figura 3.44 donde destaca una línea referencial de color rojo, donde se ubicó el código de barra deseado para la lectura.

Figura 3.44 Cámara para la lectura de códigos de barra

Una vez reconocido el código de barras, la información obtenida pasa a la vista desde la cual se inició el lector. En la Figura 3.45 se muestra la vista para el registro de la talla de un producto, donde se puede ver el código de barras obtenido desde el lector.

Figura 3.45 Resultado de lectura de código de barras

3.1.15 REPORTES DE VENTA

En esta sección se presentan los resultados obtenidos al generar un reporte general de ventas, así como un reporte de ventas por usuario. Al iniciar la vista con el reporte de ventas generales el cliente Android genera una petición de tipo `PUT` para obtener dicho reporte, la cual se muestra en la Figura 3.46, en la misma se aprecia que en el cuerpo de la petición se definen los rangos de fechas y en la clave `tipo` se encuentra el valor 1, que se refiere a este tipo de reportes.

```
D/OkHttp: --> PUT http://titulacionpoli.somee.com/api/reporte?idNegocio=7 http/1.1
D/OkHttp: Content-Type: application/json; charset=UTF-8
    Content-Length: 70
    {"desde":"2019-09-02 12:00:00","hasta":"2019-09-06 12:00:00","tipo":1}
D/OkHttp: --> END PUT (70-byte body)
```

Figura 3.46 Petición de reporte general de ventas

El servidor procesa la información de las ventas realizadas en el periodo seleccionado y retorna una lista de objetos JSON como se presenta en la Figura 3.47, cada objeto de la lista representa a un objeto de tipo `Extra`.

```
{
    status :  "200",
    reason :  "ok",
    info : [
        { key1 :  0,  key2 :  0,  key3 :  0,  key4 :  "02 Sep"},
        { key1 :  0,  key2 :  0,  key3 :  0,  key4 :  "03 Sep"},
        { key1 :  0,  key2 :  0,  key3 :  0,  key4 :  "04 Sep"},
        { key1 :  112,  key2 :  93,  key3 :  19,  key4 :  "05 Sep"},
        { key1 :  28,  key2 :  28,  key3 :  0,  key4 :  "06 Sep"}
    ]
}
```


Figura 3.47 Resultado de reporte general de ventas

En el cliente Android la información en formato JSON es transformada en objetos, los cuales son presentados como se muestra en la Figura 3.48, donde cada objeto de la lista de la Figura 3.47 representa una barra en el gráfico.

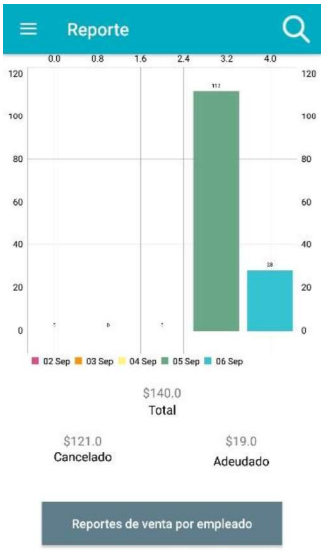


Figura 3.48 Reporte por rango de fecha

Por otro lado, para obtener un reporte de ventas por usuario, la aplicación genera una petición como en la Figura 3.49, donde el valor en la clave `tipo` cambia a 2.

```
D/OkHttp: --> PUT http://titulacionpoli.somee.com/api/reporte?idNegocio=7 http/1.1
    Content-Type: application/json; charset=UTF-8
    Content-Length: 70
D/OkHttp: {"desde":"2019-09-02 12:00:00","hasta":"2019-09-06 12:00:00","tipo":2}
    --> END PUT (70-byte body)
```

Figura 3.49 Petición para reporte de ventas por usuario

Al igual que en el reporte general de ventas, el servidor responde con una lista de objetos de tipo `Extra` representados en formato JSON mientras que en el cliente Android la información es presentada en una vista como la de la Figura 3.50.

Figura 3.50 Reporte de ventas por usuario

3.2 PRUEBAS DE ACEPTACIÓN

Para las pruebas de aceptación se entregó el prototipo a 10 administradores o vendedores de negocios dedicados a la comercialización de ropa y calzado, después de un periodo de pruebas de 3 días se realizó una entrevista que permitió validar el funcionamiento del prototipo desde la perspectiva del usuario. La entrevista se presenta en el Anexo E.

Los resultados obtenidos se presentan a continuación en la Tabla 3.1

Tabla 3.1 Resultados entrevista de aceptación

N.	Pregunta	Respuesta(%)	
		SI	NO
1	¿Pudo realizar una venta?	100.00	0.00
2	¿Pudo usar la opción de crédito como método de pago?	100.00	0.00
3	¿Pudo imprimir un *ticket* de venta?	80.00	20.00
4	¿Pudo reimprimir un *ticket* de venta?	80.00	20.00
5	¿Pudo registrar un producto y asociarlo a una categoría existente?	90.00	10.00
6	¿Pudo registrar el código de barras de un producto?	100.00	0.00
7	¿Pudo buscar un producto del catálogo a través de su código de barras?	100.00	0.00
8	¿Pudo registrar nuevos pagos?	90.00	10.00
9	¿Pudo generar reportes generales de venta?	100.00	0.00
10	¿Pudo generar reportes de venta por empleados?	100.00	0.00
11	¿Pudo registrar los datos de sus clientes?	100.00	0.00
12	Durante el uso de la aplicación. ¿Existió algún error?	20.00	80.00

13	Si la respuesta previa fue afirmativa, Describa de forma breve el error	Respuesta (%)
	Error al imprimir ticket de venta	20.00
	No se puede registrar nuevos pagos	10.00
	Asociación de tallas incorrectas	10.00

La primera pregunta hace referencia a la creación de una venta, esta pregunta permitió validar la selección de productos existentes del catálogo y el registro en la base de datos. El nivel de aceptación para este apartado fue total y por lo tanto la creación de una venta fue aceptada.

La segunda pregunta tiene el objetivo de validar una opción de pago implementada durante el desarrollo del prototipo la cual no estuvo considerada en el diseño inicial del mismo. La característica implementada permite al vendedor registrar una venta a crédito y tuvo un 100% de aprobación.

La tercera pregunta hace referencia a la impresión de un *ticket* de venta y fue planeada con dos objetivos, por un lado, corroborar la conexión de una impresora a través del prototipo y por otro, la impresión de información a través de dicho dispositivo. El nivel de aceptación de esta funcionalidad fue del 80%, el restante 20% tuvo problemas para generar un *ticket* de venta, el error encontrado se detalla en la pregunta 13.

Al igual que la pregunta 2, la pregunta 4 permitió validad una característica agregada al prototipo durante el desarrollo. Esta pregunta hace referencia a la reimpresión de un *ticket* desde la vista del detalle de venta. Con base en los resultados tabulados, el 80% de los encuestados aceptó esta característica mientras que el 20% restante tuvo inconvenientes para usarla.

La pregunta 5, tuvo el objetivo de validar el registro de productos y la categorización de los mismos. El 90% validó estas funciones mientras que el 10% restante tuvo problemas, el error encontrado se detalla en la pregunta 13.

La pregunta 6 tuvo el objetivo de validar la lectura de códigos de barras a través de la cámara del dispositivo Android para el registro del mismo en la base de datos. Esta función contó con una aprobación del 100%.

La pregunta 7 es un complemento a la pregunta 6. En esta se buscó validar la búsqueda de productos para la generación de una venta, el nivel de aceptación fue del 100%.

La pregunta 8 tuvo el objetivo de validar la administración de pagos realizados en una venta. De los encuestados, el 90% aprobó esta funcionalidad mientras que el 10% restante tuvo problema, el error encontrado se detalla en la pregunta 13.

Las preguntas 9 y 10 tuvieron el objetivo de aprobar la generación de reportes. La pregunta 9 se centra en la generación de reportes de venta generales mientras que la pregunta 10 se orienta a la generación de reportes por empleados. En ambas preguntas se obtuvo el 100% de aprobación y por lo tanto se validó la implementación de estas funcionalidades.

La pregunta 11 buscó validar el registro de clientes y con la misma la presentación y eliminación de registros de clientes previamente creados. Esta opción contó con un 100% de aceptación.
Las preguntas 12 y13 fueron planteadas con el objetivo de reconocer posibles errores no controlados durante el proceso de desarrollo. El 20% de los encuestados encontró errores en la ejecución del prototipo. Los errores encontrados fueron:

- De las preguntas 3 y 4 se encontró un error en el proceso de control de la conexión de una impresora, pues cuando la impresora perdió conexión con el cliente Android no se presentó ningún mensaje del evento, esto provocó que los usuarios no pudieran generar un *ticket* al finalizar la venta ni al reimprimirlo.
- Respecto a la pregunta 5 el usuario detalló que una vez registrado un producto, se procedió a registrar un segundo producto, pero las tallas de este segundo producto se asociaban al primero.
- De la pregunta 8 se detalló que, al iniciar la vista con el historial de pagos, se mostró un valor adeudado diferente de 0 pero no se mostraba el botón de acceso a la vista para el registro de un nuevo pago, lo cual impidió a los usuarios probar esta funcionalidad, cabe mencionar que el error se producía con valores adeudados menores a 1 dólar debido a una conversión a decimal del mismo.

A más de los errores listados, durante la realización de cada encuesta se recabaron varias opiniones para mejorar la experiencia de usuario. Estas ideas y las correcciones necesarias para solventar los errores listados serán presentadas en la siguiente sección.

3.3 DISCUSIONES

Las correcciones y modificaciones que se presentan en esta sección nacieron en primer lugar de las reuniones para las presentaciones de entregables realizadas al director de este Proyecto Técnico y luego a comentarios y sugerencias realizadas en la entrevista de aceptación del mismo.

De las reuniones realizadas con el director de este Proyecto Técnico nacieron las siguientes modificaciones:

- En las llamadas al servidor desde el cliente Android no existía ningún indicativo de espera al usuario lo cual generaba errores y peticiones duplicadas, antes esto, se modificó el método `solicitud` de la clase `Api` en el lado del cliente, para mostrar un componente `ProgressDialog` con el objetivo de advertir a los usuarios de tiempos de espera necesarios en los procesos de petición y envió de datos hacia el servidor.
- Debido a la nula utilidad del campo `precioCompra` en la ejecución del prototipo se optó por eliminarlo de la tabla `productos_x_tallas`, el cual alojaba el precio al que un administrador de una tienda adquiría sus productos.
- Existió errores en la generación de reportes de venta y búsqueda de ventas por fechas. Los errores se debieron que las fechas con las que se registra una venta en la base de datos se encuentra en formato UTC y las fechas que se envían desde el lado del cliente para los procesos mencionados se encontraban en la zona horaria local (UTC-5) haciendo que la selección de registros no sea correcta. Para esto, se desplazaron las fechas registradas en la base de datos a UTC-5 en los métodos correspondientes.
- En la vista de detalle de ventas se cambió el icono del botón para acceder al historial de pagos inicialmente ubicado en la barra superior (ver Figura 3.51 parte a) por un botón al pie de la misma (ver Figura 3.51 parte b). El cambio se justifica debido a que el icono inicialmente implementado no fue intuitivo.

(a) (b)

Figura 3.51 Modificaciones acceso al historial de pagos

Por otra parte, las soluciones a los errores mencionados durante la realización de las pruebas de aceptación fueron:

- Para solventar los errores en la impresión y reimpresión del *ticket* se agregó un mensaje “Pérdida de conexión con impresora” cuando se genere este evento y, por

otra parte, previo a la impresión de un *ticket* si no se tiene una impresora conectada, el cliente Android intentará conectarse con la impresora registrada.

- En referencia al error en el registro de productos consecutivos, el error se debió a una variable mal controlada, esta variable almacena el identificador del producto a lo largo del proceso de registro y modificación de una talla y tuvo que ser definida como `null` tras el correcto registro de cada una, con este cambio el error fue solventado satisfactoriamente.
- En referencia al error de registro de nuevos pagos, se modificó el método de transformación de `string` a `decimal` para con esto tener un mejor control de cuando mostrar el boton de acceso a la vista para el registro de pagos.

Finalmente, se presentan algunas modificaciones realizadas en base a comentarios adicionales realizados por los usuarios durante la entrevista de aceptación.

- Los entrevistados mencionaron que al finalizar un proceso de registro se presenta el listado con el nuevo registro, pero no se muestra ningún mensaje de confirmación, ante esto, se agregaron mensajes de confirmación al finalizar los procesos de registro y modificación de datos con el objetivo de implementar la retroalimentación deseada.
- En las listas de selección se agregó una etiqueta con texto que indique el parámetro a seleccionar (ver Figura 3.52), al igual que en el punto anterior los usuarios mencionaron que a diferencia de los campos de texto donde se usó `placeholders` para indicar el contenido del mismo, las listas de selección no contaban con textos de ayuda.

Figura 3.52 Etiquetas en `Combobox`

4. CONCLUSIONES Y RECOMENDACIONES

A continuación, se presentan las conclusiones y recomendaciones recogidas a lo largo del desarrollo de este Proyecto Técnico, así como posibles trabajos futuros relacionados.

4.1 CONCLUSIONES

- El prototipo resultante de este Proyecto Técnico cuenta con tres componentes: base de datos para almacenar la información relacionada a usuarios, productos y ventas entre otros, una aplicación Android desde la cual los usuarios (administradores, vendedores) gestionen sus datos y una web API como puente entre los dos componentes antes mencionados que recepta y procesa la información de cada solicitud.
- Con la ayuda de este prototipo de punto de venta los administradores y vendedores de ropa y calzado pueden mantener un control de sus ventas y sus clientes de una manera centralizada sin riesgo de perder la información como pasa en negocios donde hacen uso de métodos de control manuales.
- Este prototipo también permite imprimir un comprobante al finalizar una venta en una impresora térmica con conexión Bluetooth, así como hacer uso de la cámara del dispositivo para registrar y leer códigos de barras relacionados a un producto.
- Durante el desarrollo del prototipo se incluyó la opción para realizar ventas a crédito lo cual resulto ser una opción con gran aceptación y de gran utilidad para las PyMES que comercializan ropa y calzado.
- El análisis de aplicaciones previamente existentes junto con las entrevistas realizadas a los administradores de PyMES que comercializan ropa y calzado permitieron definir requerimientos funcionales y no funcionales necesarios para este prototipo, así como organizar las principales estructuras gráficas presentes en el.
- El uso de la clase `Respuesta` para enviar información desde la web API hacia el cliente Android permitió simplificar el control y presentación de errores, así como el reducir el código escrito para manejar la respuesta de las peticiones realizadas.
- Las peticiones que se generaron en el cliente Android hacia el servidor se realizaron con la ayuda de la librería Retrofit debido a su fácil implementación y compatibilidad con la librería Gson la cual se usó para el manejo de información en formato JSON, así como transformar información en dicho formato a objetos que se pudieran manipular con lenguaje Java de Android.

- En el desarrollo del prototipo de punto de venta se optó por el desarrollo de la base de datos como punto de inicio pues del modelado del diagrama E-R resultaron muchas entidades y relaciones, las cuales fueron más sencillas plasmar a través de código SQL.
- En el diseño de la base de datos se agregó el atributo `estado` a varias entidades para desactivar los registros e impedir que sean utilizados sin que sean borrados de la base de datos, ya que eliminar un registro puede producir errores relaciones con la integridad referencial de la base de datos debido al gran número de relaciones entre entidades.
- El uso de la tecnología Entity Framework con el paradigma *DataBase First* en la web API facilito la asociación y acceso a la capa de bases de datos, el uso de esta tecnología eliminó el potencial riesgo de errores al crear clases y asociaciones de forma manual.
- LINQ constituye un pilar en el desarrollo pues se usó para guardar, modificar, y eliminar información en la base de datos, así como para buscar registros que fueron presentados en el cliente Android.
- El uso de las recomendaciones establecidas en la metodología ágil Scrum permitió definir tareas que se agruparon en *sprints* y con ello lograr un desarrollo incremental del prototipo, generando entregables con pocas funciones que juntos formaron el producto final demostrable.
- Para el desarrollo del cliente Android se hizo uso de fragmentos lo cual permitió reusar componentes gráficos comunes en toda la aplicación como menús y barras de búsqueda.
- El uso de peticiones asíncronas en el lado del cliente Android permitió crear un nuevo hilo de ejecución para peticiones con gran carga de datos y con ello se dio la libertad de que el usuario pueda seguir trabajando en la aplicación mientras dichas peticiones se ejecutaban de manera simultánea.
- El almacenamiento de datos de inicio de sesión con el uso del proveedor de contenidos `SharedPreferences` eliminó la necesidad de que un usuario tenga que ingresar sus credenciales cada vez que inicia el cliente Android.
- El uso del servicio Firebase Cloud Messaging (FCM) en el cliente Android y en la web API permitió a los usuarios que trabajan de manera simultánea mantener actualizado el inventario de productos en tiempo real.

- Para estandarizar los estilos de las vistas en el lado del cliente se crearon plantillas XML que después se adjuntaron a los componentes `EditText`, `Spinner` y `Button` a través de la propiedad `android:background`.

4.2 RECOMENDACIONES

- En el lado del servidor se cuenta con bloques de código que pueden ser reemplazados por procedimientos almacenados en la base de datos, por ejemplo, la reducción del *stock* de un producto al realizar una venta, este cambio podría reducir el tiempo de respuesta ante estas peticiones.
- En el lado del cliente se generaron varios adaptadores para controlar la información de cada ítem de una lista, muchos de ellos con el mismo esquema gráfico. Si bien esto ayudo a mantener cierto nivel de compresión e independencia durante el desarrollo esto supone un problema al momento de escalar la aplicación por lo que se recomienda concentrar todo el código escrito en un solo adaptador.
- Una recomendación realizada por los usuarios durante las encuestas de aceptación fue añadir el logo de su negocio en el *ticket* de venta, esto no fue incluido en el prototipo debido al alcance del mismo, pero se podría incluir en trabajos futuros para lo cual sería necesario un repositorio en el servidor donde se puedan almacenar las imágenes mientras que en el cliente Android se debería incluir un método para la impresión de dichas imágenes con la ayuda de la clase `PrinterCommands` de la librería Bluetooth Thermal SDK.
- Para la presentación de mensajes de error se puede cambiar el uso del componente `Toast` por el `AlertDialog`, esto daría mayor realce a estos mensajes y podría ser de utilidad para los administradores y vendedores durante el uso del cliente Android.
- El prototipo resultante de este Proyecto Técnico contempla el uso de una impresora térmica para papel de 58mm de ancho, se recomienda continuar el desarrollo del mismo para brindar soporte para impresoras que alojen papel de 80mm de ancho, pues este es otro formato de *ticket* de gran uso en los negocios.
- El prototipo resultante de este Proyecto Técnico contempla el uso de una impresora con conexión Bluetooth, se recomienda continuar el desarrollo del mismo para brindar soporte para impresoras con conexión WiFi, para esto se puede hacer uso de la misma librería Bluetooth Thermal SDK la cual ya incluye métodos para la conexión a través de esta tecnología de comunicación inalámbrica.

- Se recomienda continuar el desarrollo de este prototipo de punto de venta con el fin de incluir facturación electrónica ya que esta es una función que las aplicaciones (Loyverse y Square) analizadas no poseen y podría ser una ventaja competitiva.
- Se recomienda continuar el desarrollo de este prototipo de punto de venta con el objetivo de incluir una pasarela de pagos como PayPal o PayPhone y con esto los clientes puedan realizar pagos con tarjeta de crédito o débito.
- Con el objetivo de minimizar la transferencia de información entre el servidor y el cliente se puede extender el uso del proveedor de contenido `SharedPreferences` para almacenar también datos de productos, ventas y clientes. Esta extensión debería contemplar también un reajuste en el envió de datos a través del servicio Firebase Clod Messaging en el que se envíe solamente los datos pertinentes a la modificación o registro de la información realizada.
- Se recomienda el uso de *Data Annotations* en las clases de la webAPI como un segundo método de control para evitar errores e incoherencias en la información que proviene del cliente Android.
- Las *Data Annotations* mencionadas deben ser incluidas una vez se finalice el prototipo ya que durante las actualizaciones de las clases generadas por Entity Framework en respuesta a cambios en la estructura de bases de datos las *Data Annotations* se borran.
- El uso del servicio Firebase Cloud Messaging se justifica en este Proyecto Técnico debido al alcance del mismo ya que por su limitado flujo de intercambio de datos su uso no supone ninguna inversión económica, sin embargo, para la puesta en producción del prototipo se recomienda buscar alternativas que permitan mayor flujo de datos con un plan gratuito o en su defecto alguna alternativa cien por ciento gratuita.
- De la misma manera para la puesta en producción de este prototipo se recomienda migrar la web API y la base de datos a servidores con mayores niveles de seguridad y mayores prestaciones como Microsoft Azure o Amazon Web Services.
- De manera complementaria, para la puesta en producción de este prototipo se recomienda integrar un certificado SSL a la web API para así mantener una transferencia de información cifradas con el cliente Android.
- Con el objetivo de ampliar el alcance al cual se dirige este prototipo, como trabajo futuro se puede agregar un perfil de usuario para cajeros y contadores, así como el control de inventario por sucursales, estas adiciones permitirían que el prototipo se adapte a los procesos de ventas y control de inventario de grandes negocios.

5. REFERENCIAS BIBLIOGRÁFICAS

[1] Trend Micro, [En línea]. Available: https://www.trendmicro.com/es_mx/business.html. [Último acceso: 08 12 2018].

[2] INEC, «Ecuador en cifras,» [En línea]. Available: http://www.ecuadorencifras.gob.ec/documentos/web-inec/Estadisticas_Sociales/TIC/2017/Tics%202017_270718.pdf. [Último acceso: 12 04 2019].

[3] Universidad de Alicante, «Servicios web,» [En línea]. Available: https://rua.ua.es/dspace/bitstream/10045/16740/18/14-Web%20services.pdf. [Último acceso: 12 04 2019].

[4] A. Los Santos, «Revisión de los servicios web SOAP/REST: Caracteristicas y rendimiento,» [En línea]. Available: http://www.albertolsa.com/wp-content/uploads/2009/07/mdsw-revision-de-los-servicios-web-soap_rest-alberto-los-santos.pdf. [Último acceso: 12 04 2019].

[5] Universidad de Valencia, «Representational State Transfer: REST,» [En línea]. Available: http://bibing.us.es/proyectos/abreproy/11247/fichero/Memoria%252F8-Representational+State+Transfer+(REST).pdf. [Último acceso: 12 03 2019].

[6] D. Crockford, «Introducing JSON,» [En línea]. Available: https://www.json.org/. [Último acceso: 20 02 2019].

[7] Chakray, «¿Qué diferencias hay entre SOAP y REST?,» [En línea]. Available: https://www.chakray.com/que-diferencias-hay-entre-rest-y-soap/. [Último acceso: 15 03 2019].

[8] C. Wodehouse, «SOAP vs. REST: A Look at Two Different API Styles,» [En línea]. Available: https://www.upwork.com/hiring/development/soap-vs-rest-comparing-two-apis/. [Último acceso: 30 03 2019].

[9] Microsoft, «ASP.NET Web APIs,» [En línea]. Available: https://dotnet.microsoft.com/apps/aspnet/apis. [Último acceso: 30 03 2019].

[10] A. Mittal, «Web API 2 With Entity Framework 6 Code First Migrations,» [En línea]. Available: https://www.c-sharpcorner.com/article/learning-web-api-2-with-entity-framework-6-code-first-migrations/. [Último acceso: 15 03 2019].

[11] M. E. Millán, Fundamente de bases de datos, Cali: Universidad del Valle, 2017.

[12] Lucidchart, «Qué es un modelo de base de datos,» [En línea]. Available: https://www.lucidchart.com/pages/es/que-es-un-modelo-de-base-de-datos. [Último acceso: 02 04 2019].

[13] E. Gómez Ballester, P. Martínez Barco, P. Moreda Pozo, A. Suárez Cueto, A. Montoyo Guijarro y E. Saquete Boro, «Apuntes de bases de datos 1,» pp. 15-23.

[14] Entity Framework Tutorial, «What is Entity Framework?,» [En línea]. Available: https://www.entityframeworktutorial.net/what-is-entityframework.aspx. [Último acceso: 17 03 2019].

[15] Microsoft, «Entity Framework Overview,» [En línea]. Available: https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ef/overview. [Último acceso: 18 03 2019].

[16] TutorialsTeacher, «Why LINQ?,» [En línea]. Available: https://www.tutorialsteacher.com/linq/why-linq. [Último acceso: 20 03 2019].

[17] J. Luján, «Android Studio. Aprende a desarrollar aplicaciones,» Mexico D.F., 2019, pp. 118-146.

[18] Android Developers, «Descripción general del manifiesto de una app,» [En línea]. Available: https://developer.android.com/guide/topics/manifest/manifest-intro?hl=es-419. [Último acceso: 02 08 2019].

[19] Android Developers, «Configura tu compilación,» [En línea]. Available: https://developer.android.com/studio/build. [Último acceso: 20 03 2019].

[20] S. Owen, «ZXing Documentation,» [En línea]. Available: https://github.com/zxing/zxing. [Último acceso: 01 04 2019].

[21] Weeklycoding, «MPAndroidChart Documentation,» [En línea]. Available: https://weeklycoding.com/mpandroidchart-documentation/getting-started/. [Último acceso: 01 04 2019].

[22] Firebase, «Firebase Cloud Messaging,» [En línea]. Available: https://firebase.google.com/docs/cloud-messaging/. [Último acceso: 02 04 2019].

[23] Firebase, «Configura una app cliente de Firebase Cloud Messaging en Android,» [En línea]. Available: https://firebase.google.com/docs/cloud-messaging/android/client. [Último acceso: 02 04 2019].

[24] P. Deemer, G. Benefield, C. Larman y B. Vodde, «Introducción básica a la teoría y práctica de Scrum,» 17 02 2019. [En línea]. Available: http://scrumprimer.org/primers/es_scrumprimer20.pdf.

[25] M. Itzcoalt Alvarez, «Desarrollo ágil con Scrum,» [En línea]. Available: http://cic.puj.edu.co/wiki/lib/exe/fetch.php?media=materias:sg07.p02.scrum.pdf. [Último acceso: 02 08 2019].

[26] A. Peralta, «–Metodología SCRUM–,» [En línea]. Available: https://fi.ort.edu.uy/innovaportal/file/2021/1/scrum.pdf. [Último acceso: 17 02 2019].

[27] Square Inc., «Grow your business your way with Square tools,» [En línea]. Available: https://squareup.com/us/en. [Último acceso: 04 04 2019].

[28] Loyverse, «Software TPV Gratis y Control de Inventario,» [En línea]. Available: https://loyverse.com/es. [Último acceso: 04 04 2019].

[29] Finances Online, «Loyverse POS Review,» [En línea]. Available: https://reviews.financesonline.com/p/loyverse-pos/. [Último acceso: 12 04 2019].

[30] Somee, «Somee,» [En línea]. Available: https://somee.com/default.aspx. [Último acceso: 21 04 2019].

[31] Android Developers, «Fragmentos,» [En línea]. Available: https://developer.android.com/guide/components/fragments?hl=es-419. [Último acceso: 29 04 2019].

[32] Android Developers, «Adapters,» [En línea]. Available: https://developer.android.com/reference/android/widget/Adapter. [Último acceso: 30 04 2019].

[33] A. Hathibelagal, «Entendiendo la Transmisión de Android,» [En línea]. Available: https://code.tutsplus.com/es/tutorials/android-from-scratch-understanding-android-broadcasts--cms-27026. [Último acceso: 30 04 2019].

[34] GitHub, «Sly Calendar View,» [En línea]. Available: https://github.com/psinetron/slycalendarview. [Último acceso: 04 05 2019].

[35] Revolvy, «Common Language Runtime,» [En línea]. Available: https://www.revolvy.com/page/Common-Language-Runtime. [Último acceso: 30 03 2019].

[36] M. Berger, «Data Access Object Pattern,» [En línea]. Available: https://pdfs.semanticscholar.org/1b11/5f5253567964283e16081488daf6af4f5351.pdf. [Último acceso: 30 03 2019].

[37] Kagilum SAS, «Herramienta de gestión deproyectos realmente ágil,» [En línea]. Available: https://www.icescrum.com/es/. [Último acceso: 22 04 2019].

ANEXOS

Anexo A Entrevista para el levantamiento de información

Anexo B Script para creación de la base de datos

Anexo C Código fuente de la web API

Anexo D Código fuente del cliente Android

Anexo E Encuesta de validación

Anexo F Manual de instalación del cliente Android

Anexo G Manual de usuario del cliente Android

Anexo H Listado de PYMES entrevistas

Los anexos se han incluido en un CD adjunto a este documento