Guía explicativa de la creación de la Aplicación ‘MidletMensajero’

1. Estructura General de la Aplicación

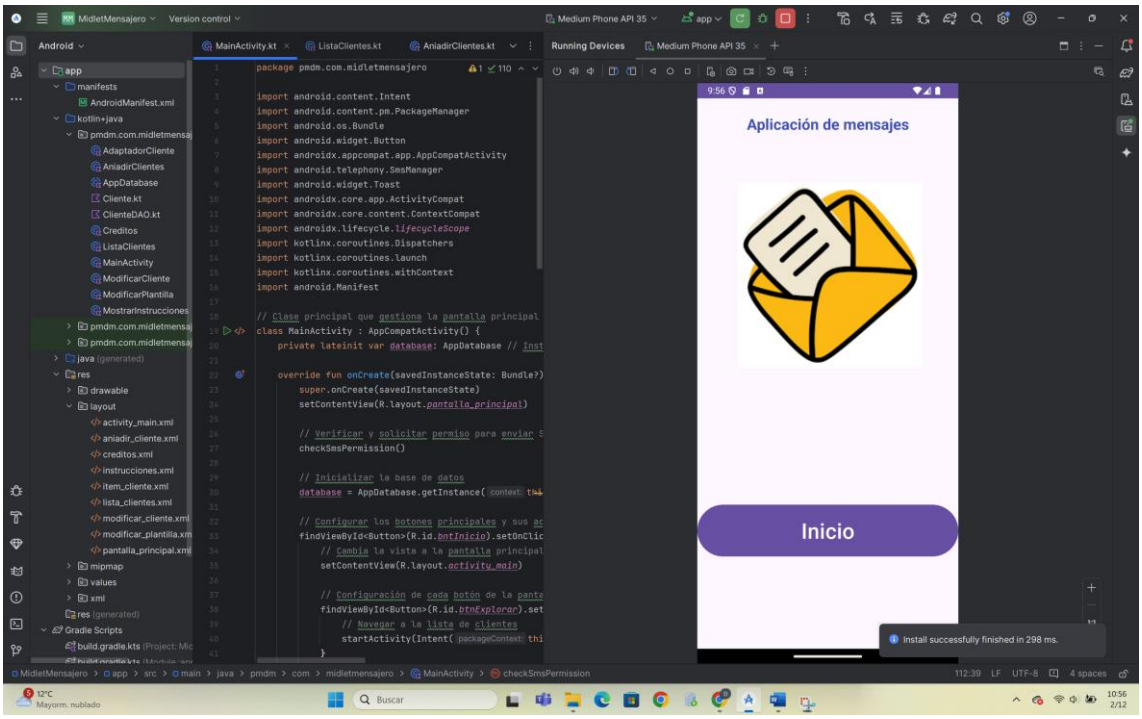
La aplicación está organizada para gestionar una pequeña base de datos de contactos de clientes para poder enviar mensajes publicitarios, permitiendo al usuario seleccionar crear contactos, modificarlo y eliminarlos, así como modificar la plantilla del mensaje y enviar mensajes a todos estos. Se han utilizado varias actividades (Activities) y archivos XML para la interfaz de usuario.

2. Diseño de la Interfaz (Archivos XML)

Los siguientes archivos XML definen la interfaz de usuario para las pantallas principales de la aplicación:

i. Pantalla Inicio(pantalla_principal.xml):

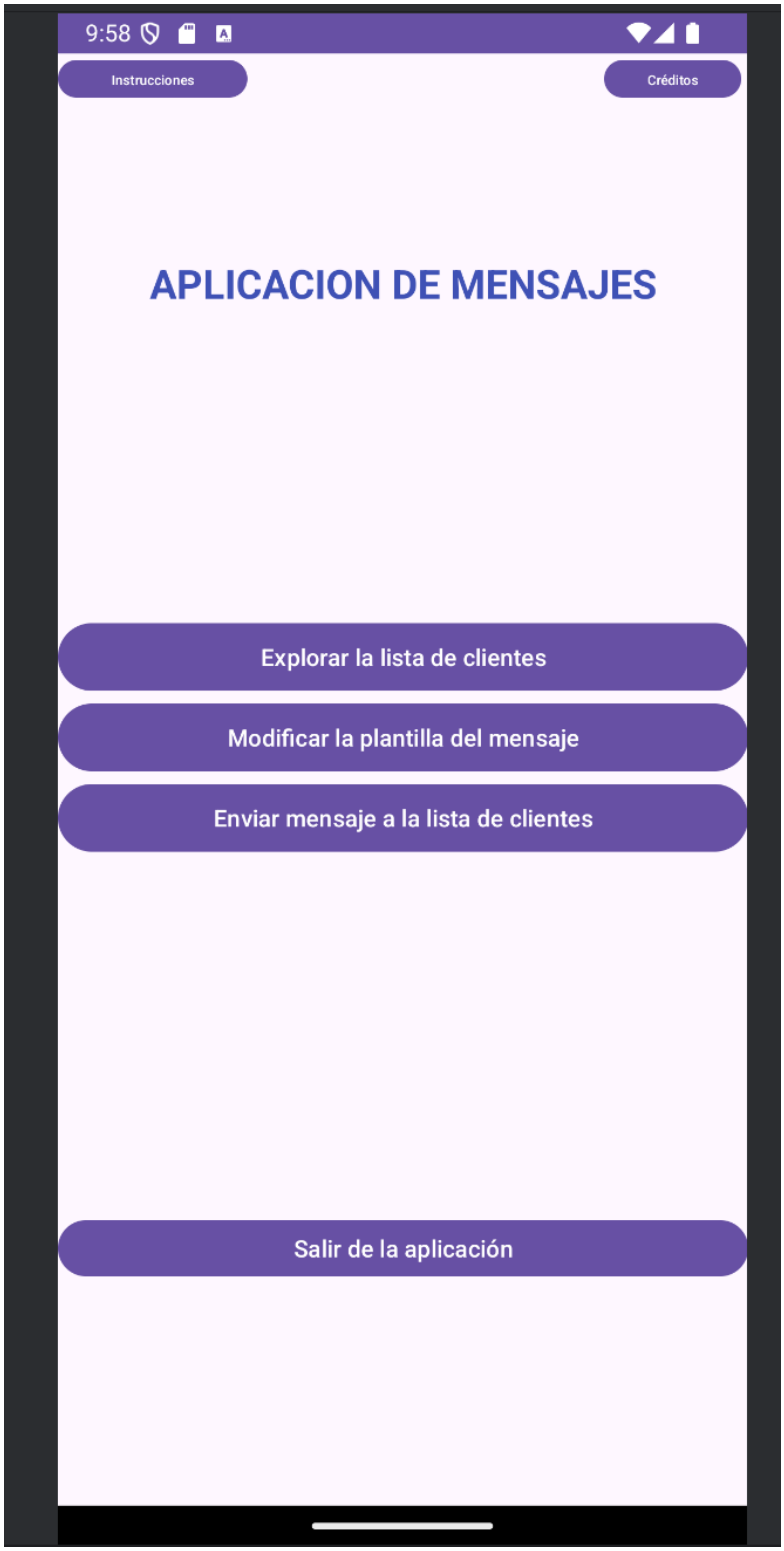
Se muestra al iniciar la aplicación, contiene el título de la aplicación, una imagen y el botón “Iniciar”.



ii. Pantalla Principal(activity_main.xml):

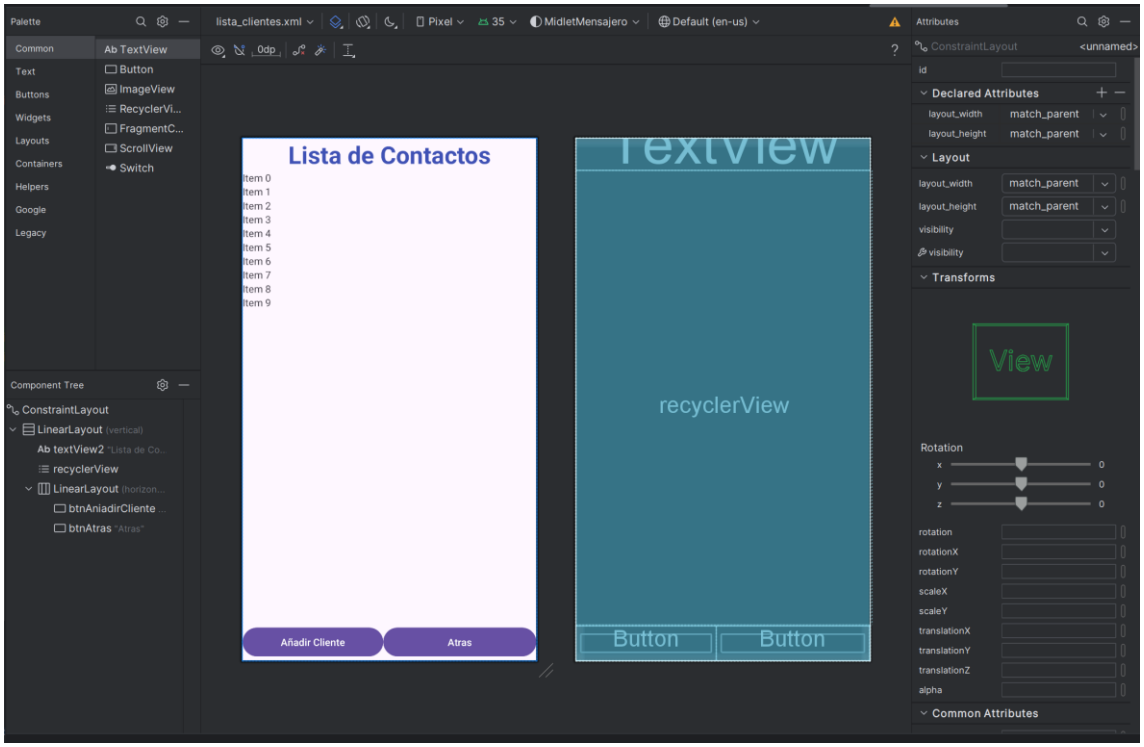
- Esta pantalla muestra el menú principal de la aplicación. Contiene los siguientes botones.
  - **btnInstrucciones:** Dirige a la pantalla instrucciones, en la cual se muestra una guía de uso de la aplicación.
  - **btnCréditos:** Dirige a la pantalla de créditos.
  - **btnExplorar:** Dirige a la página de lista_clientes donde se muestra la lista de contactos.

- **btnPlantilla:** Dirige a la página modificar_plantilla.xml donde se permite editar la plantilla del mensaje de texto.
- **btnEnviarMensaje:** Permite envisar el mensaje de la plantilla a todos los contactos.
- **btnSalir:** permite salir de la aplicación.



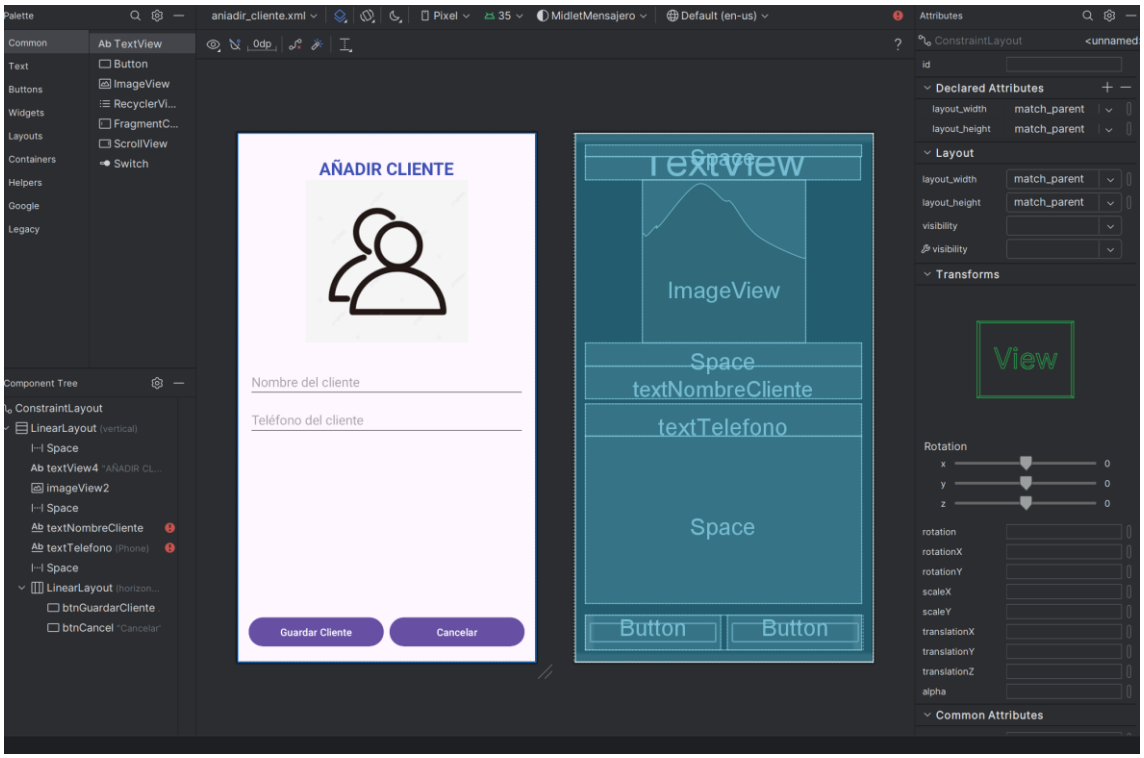
iii. **Pantalla de lista de contactos (lista_clientes.xml):**

- Contiene un título(viewText) en la parte superior con el nombre de la sección**.**
- Para mostrar la lista de contactos se ha hecho uso de un recyclerView.
- En la parte inferior se sitúan dos botones:
  - **btnAniadirCliente:** Rederidige a la página aniadir_cliente.xml para crear un nuevo contacto.
  - **btnAtras:** Cierra esta página y vuelve al menú principal.



iv. Pantalla para añadir clientes (aniadir_clientes.xml)

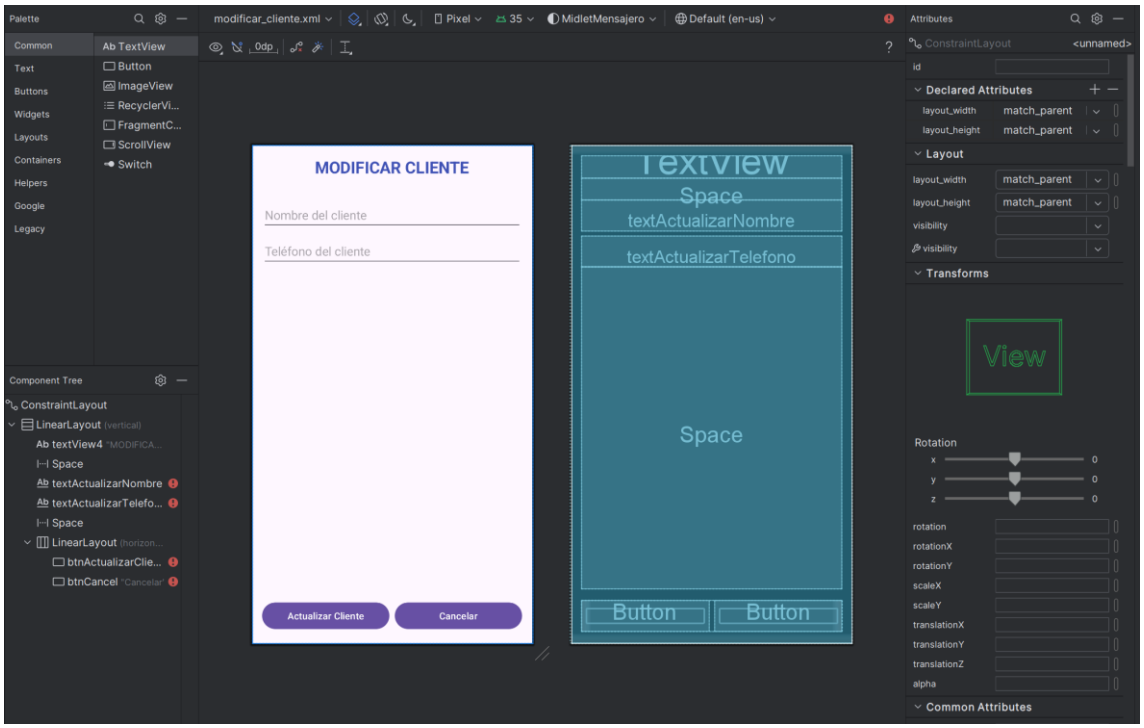
- Contiene un título(viewText) en la parte superior con el nombre de la sección**.**
- Debajo del título se ha insertado una imagen.
- Seguidamente se encuentran dos TextInput que permiten insertar el nombre y teléfono del contacto:
  - **textNombreCliente**
  - **textTeléfono**
- En la parte inferior se sitúan dos botones:
  - **btnguardarCliente:** guarda el contacto en la base de datos.
  - **btnCancelar:** vuelve a la pantalla anterior sin guardar el contacto.



v. Pantalla para modificar el cliente (modificar_cliente.xml)

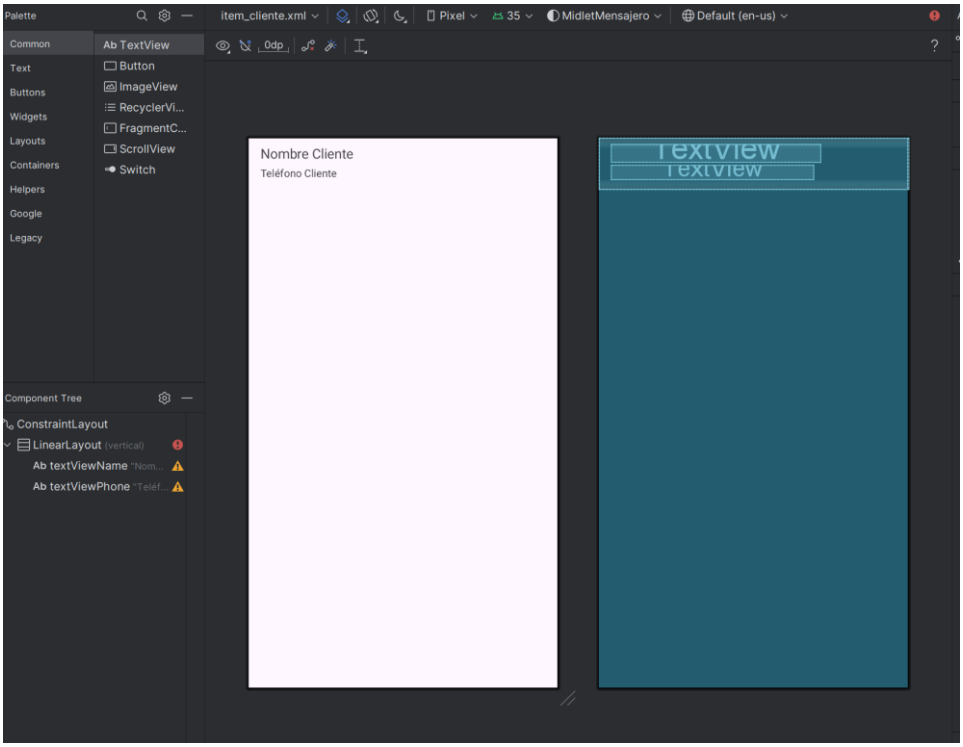
Esta pantalla permite modificar la información de un contacto previamente guardado. Contiene:

- Contiene un título(viewText) en la parte superior con el nombre de la sección**.**
- Seguidamente se encuentran dos TextInput que permiten insertar el nombre y teléfono del contacto:
  - **textActualizarNombre**
  - **textActualizarTeléfono**
- En la parte inferior se encuentran dos botones:
  - **btnActualizarCliente:** Guarda la nueva información del cliente.
  - **btnCancel:** Cancela la modificación del cliente, cerrando la ventana y volviendo a la anterior.



vi. Pantalla item cliente (item_cliente.xml)

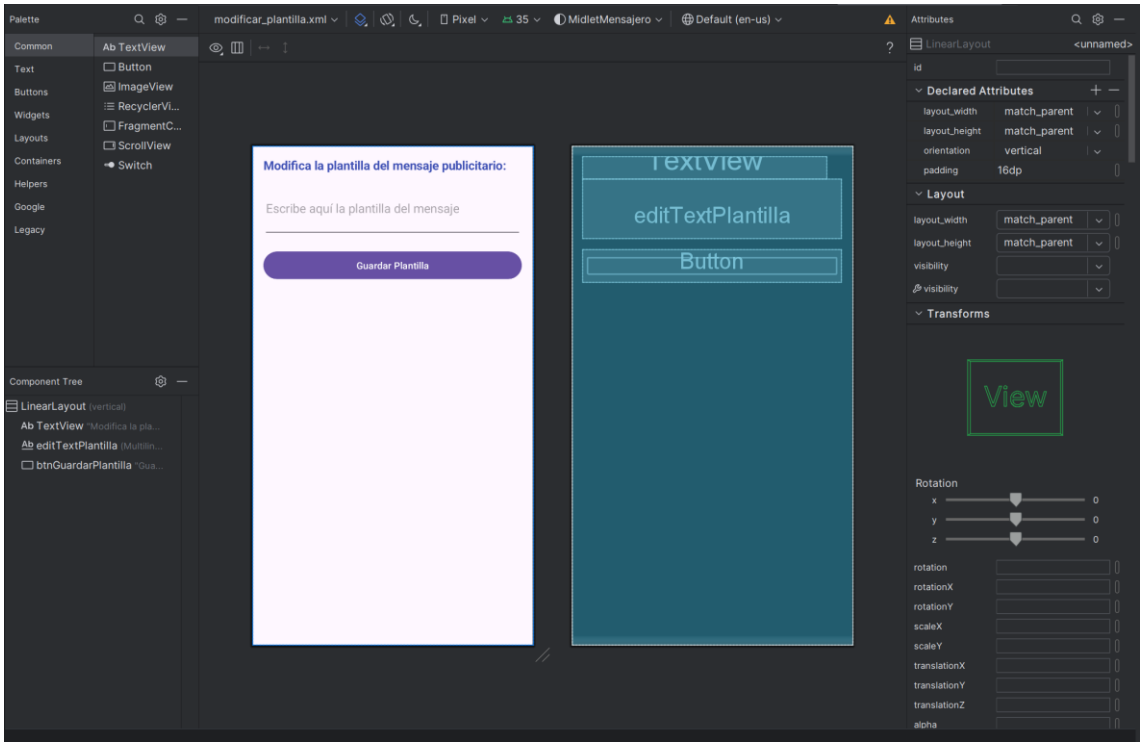
Esta pantalla es utilizada para insertar el item en el recyclerView de la listaClientes. Contiene dos textView que contendrán el nombre y el teléfono del contacto.



vii. Pantalla Modificar Plantilla (modificar_plantilla.xml)

Esta pantalla incluye la plantilla editable para poder establecer el mensaje personalizado que posteriormente se enviará a los clientes por sms. Esta pantalla contiene:

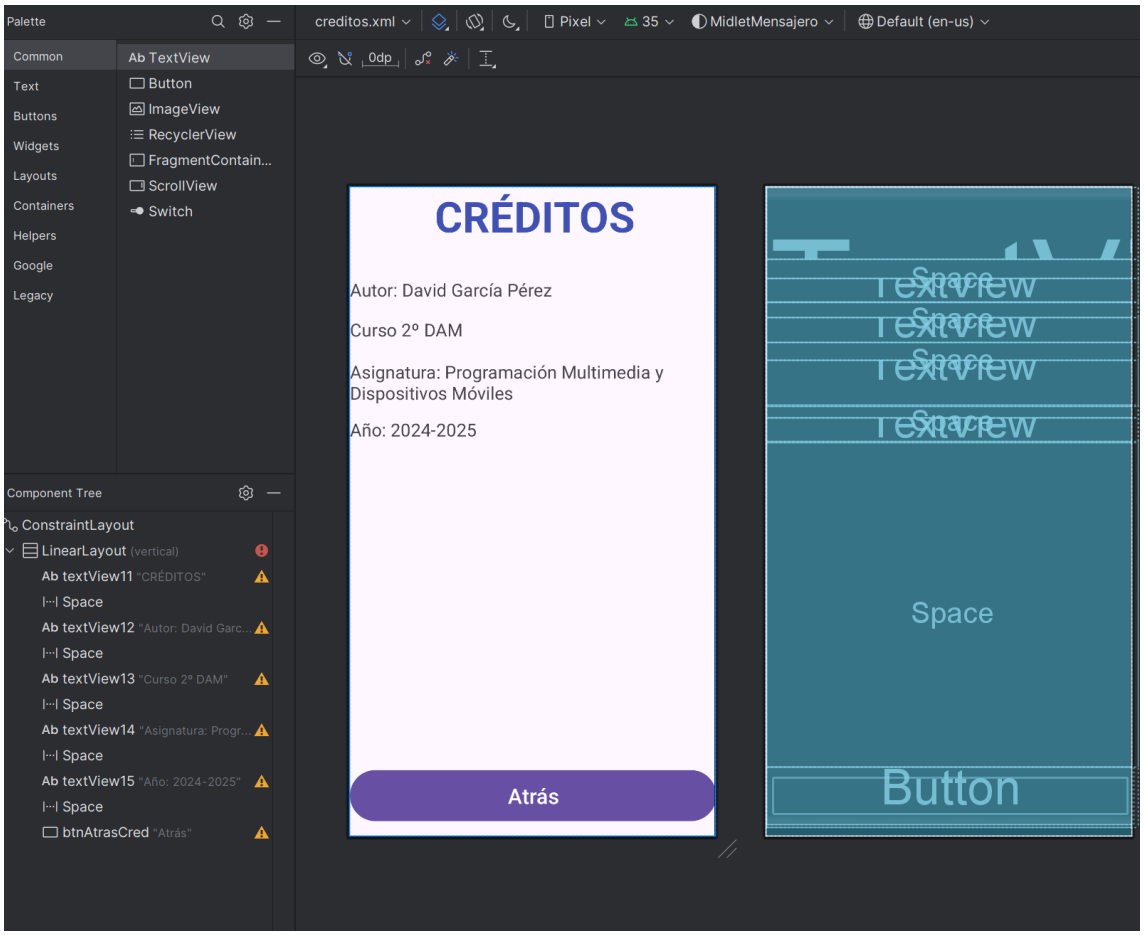
- Contiene un título(viewText) en la parte superior con la instrucción de esta sección**.**
- Seguidamente inclute un editText (**editTextPlantilla**) que permite introducir el mensaje que posteriormente se enviará.
- Finalmente incluye el botón **btnGuardarPlantilla.**

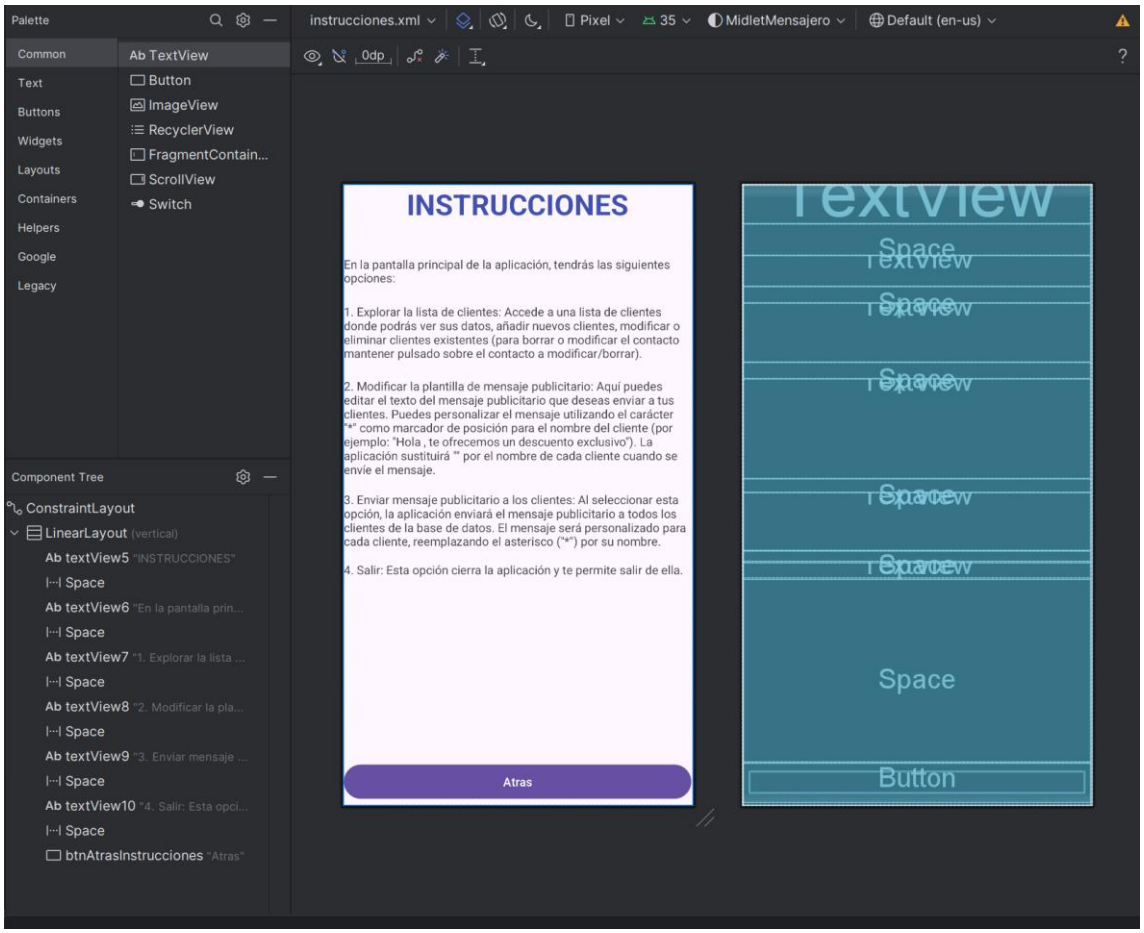


viii. Pantallas créditos e instrucciones

Por último, se incluyen las pantallas de instrucciones y créditos que contienen la información respectiva.

Programación Multimedia y Dispositivos Móviles





3. Clases en Kotlin

Las clases en Kotlin gestionan la lógica de la aplicación y la interacción con el usuario. A continuación, se detalla cada clase y su funcionalidad:

i. Clase MainClass:

Función: La clase MainActivity actúa como la actividad principal de la aplicación y sirve como punto de entrada para los usuarios. Proporciona opciones para realizar las siguientes acciones:

- Navegar por las funcionalidades principales de la aplicación, como gestionar clientes, modificar la plantilla de mensajes, y enviar mensajes personalizados.
- Solicitar permisos necesarios (como el permiso para enviar SMS).
- Gestionar la base de datos local mediante Room para acceder a los datos de clientes.

Análisis de Componentes:

Atributos

- **database: AppDatabase**
  Representa la base de datos local de la aplicación. Se inicializa en el método onCreate usando el patrón singleton proporcionado por AppDatabase.

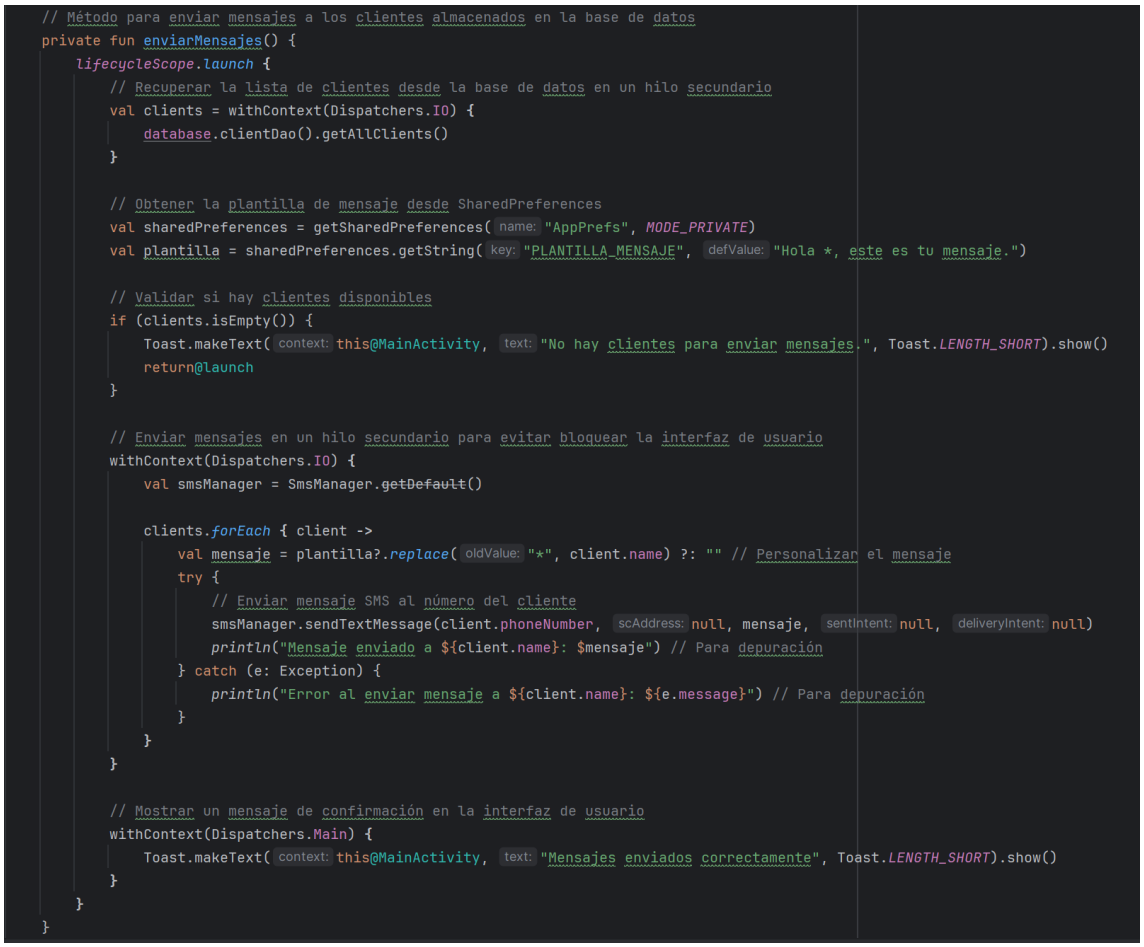
Métodos

1. **onCreate(savedInstanceState: Bundle?)**
    - **Responsabilidad**: Configura la interfaz de usuario inicial y los eventos asociados a los botones principales.
    - **Puntos Clave**:
        - Inicializa la base de datos usando AppDatabase.getInstance().
        - Verifica y solicita el permiso necesario para enviar mensajes SMS (checkSmsPermission).
        - Configura listeners para cada botón de la interfaz principal (btnExplorar, btnPlantilla, btnEnviarMensaje, etc.).
2. **enviarMensajes()**
    - **Responsabilidad**: Envía mensajes SMS personalizados a los clientes almacenados en la base de datos.
    - **Puntos Clave**:
        - Recupera la lista de clientes desde la base de datos en un hilo secundario utilizando lifecycleScope y Dispatchers.IO.
        - Obtiene la plantilla de mensaje almacenada en SharedPreferences.
        - Recorre la lista de clientes y personaliza el mensaje para cada uno.
        - Usa SmsManager para enviar mensajes SMS a los números de teléfono de los clientes.
        - Maneja errores usando un bloque try-catch.
3. **checkSmsPermission()**
    - **Responsabilidad**: Verifica si el permiso para enviar SMS ha sido otorgado y, si no, lo solicita.
    - **Puntos Clave**:
        - Usa ContextCompat.checkSelfPermission para verificar el estado del permiso.
        - Solicita el permiso con ActivityCompat.requestPermissions si es necesario.
4. **onRequestPermissionsResult()**
    - **Responsabilidad**: Maneja la respuesta del usuario al diálogo de solicitud de permisos.
    - **Puntos Clave**:

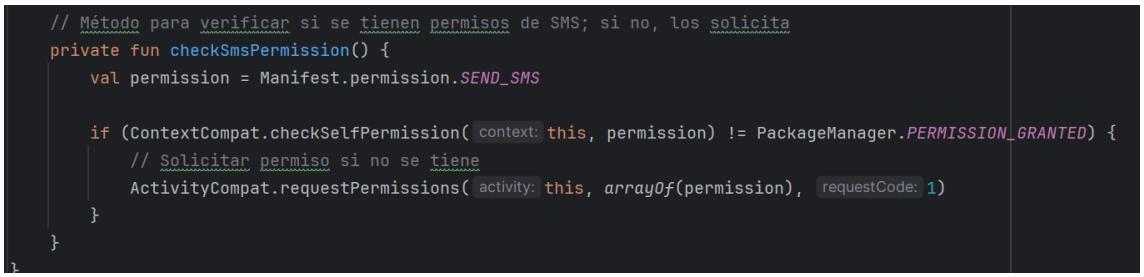
- Verifica si el permiso de envío de SMS fue otorgado y notifica al usuario en consecuencia.

```
// Clase principal que gestiona la pantalla principal y las interacciones iniciales de la aplicación
class MainActivity : AppCompatActivity() {
    private lateinit var database: AppDatabase // Instancia de la base de datos local

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.pantalla_principal)

        // Verificar y solicitar permiso para enviar SMS
        checkSmsPermission()

        // Inicializar la base de datos
        database = AppDatabase.getInstance( context: this)

        // Configurar los botones principales y sus acciones
        findViewById<Button>(R.id.bntInicio).setOnClickListener {
            // Cambia la vista a la pantalla principal
            setContentView(R.layout.activity_main)

            // Configuración de cada botón de la pantalla principal
            findViewById<Button>(R.id.btnExplorar).setOnClickListener {
                // Navegar a la lista de clientes
                startActivity(Intent( packageContext: this, ListaClientes::class.java))
            }

            findViewById<Button>(R.id.btnPlantilla).setOnClickListener {
                // Navegar a la pantalla para modificar la plantilla de mensaje
                startActivity(Intent( packageContext: this, ModificarPlantilla::class.java))
            }

            findViewById<Button>(R.id.btnEnviarMensaje).setOnClickListener {
                // Enviar mensajes a los clientes
                enviarMensajes()
            }
```

```
            findViewById<Button>(R.id.btnSalir).setOnClickListener {
                // Finalizar la actividad actual y salir de la aplicación
                finish()
            }

            findViewById<Button>(R.id.btnInstrucciones).setOnClickListener {
                // Navegar a la pantalla de instrucciones
                startActivity(Intent( packageContext: this, MostrarInstrucciones::class.java))
            }

            findViewById<Button>(R.id.btnCreditos).setOnClickListener {
                // Navegar a la pantalla de créditos
                startActivity(Intent( packageContext: this, Creditos::class.java))
            }
        }
    }
```



```
// Método para enviar mensajes a los clientes almacenados en la base de datos
private fun enviarMensajes() {
    lifecycleScope.launch {
        // Recuperar la lista de clientes desde la base de datos en un hilo secundario
        val clients = withContext(Dispatchers.IO) {
            database.clientDao().getAllClients()
        }

        // Obtener la plantilla de mensaje desde SharedPreferences
        val sharedPreferences = getSharedPreferences( name: "AppPrefs", MODE_PRIVATE)
        val plantilla = sharedPreferences.getString( key: "PLANTILLA_MENSAJE",  defValue: "Hola *, este es tu mensaje.")

        // Validar si hay clientes disponibles
        if (clients.isEmpty()) {
            Toast.makeText( context: this@MainActivity,  text: "No hay clientes para enviar mensajes.", Toast.LENGTH_SHORT).show()
            return@launch
        }

        // Enviar mensajes en un hilo secundario para evitar bloquear la interfaz de usuario
        withContext(Dispatchers.IO) {
            val smsManager = SmsManager.getDefault()

            clients.forEach { client ->
                val mensaje = plantilla?.replace( oldValue: "*", client.name) ?: "" // Personalizar el mensaje
                try {
                    // Enviar mensaje SMS al número del cliente
                    smsManager.sendTextMessage(client.phoneNumber,  scAddress: null, mensaje,  sentIntent: null,  deliveryIntent: null)
                    println("Mensaje enviado a ${client.name}: $mensaje") // Para depuración
                } catch (e: Exception) {
                    println("Error al enviar mensaje a ${client.name}: ${e.message}") // Para depuración
                }
            }
        }

        // Mostrar un mensaje de confirmación en la interfaz de usuario
        withContext(Dispatchers.Main) {
            Toast.makeText( context: this@MainActivity,  text: "Mensajes enviados correctamente", Toast.LENGTH_SHORT).show()
        }
    }
}
```



```
// Método para verificar si se tienen permisos de SMS; si no, los solicita
private fun checkSmsPermission() {
    val permission = Manifest.permission.SEND_SMS

    if (ContextCompat.checkSelfPermission( context: this, permission) != PackageManager.PERMISSION_GRANTED) {
        // Solicitar permiso si no se tiene
        ActivityCompat.requestPermissions( activity: this, arrayOf(permission),  requestCode: 1)
    }
}
```

ii. Clase AppDatabase

- Clase principal de la base de datos que define las entidades y los DAOs asociados.
- Anotada con @Database, especifica:
    - Las entidades que maneja (en este caso, Client).
    - La versión de la base de datos (version = 1).
    - Si debe exportarse el esquema de la base de datos (exportSchema = false).

1. **Método clientDao**:

- Devuelve una instancia del DAO (ClientDao) para realizar operaciones CRUD en la tabla Client.

2. **Patrón Singleton**:

- Se utiliza para garantizar que solo exista una instancia de la base de datos en toda la aplicación.
- La variable INSTANCE es @Volatile, lo que asegura la visibilidad de los cambios realizados por un hilo a otros hilos.

3. **Creación de la base de datos**:

- Se utiliza Room.databaseBuilder para construir la base de datos:
    - context.applicationContext: Asegura que el contexto no esté ligado a una actividad específica.
    - AppDatabase::class.java: Especifica la clase que define la base de datos.
    - "clientes.db": Nombre del archivo donde se almacenan los datos.

4. **Sincronización**:

- synchronized(this): Asegura que solo un hilo pueda ejecutar el bloque de código a la vez, previniendo la creación de múltiples instancias.

```
package pmdm.com.midletmensajero

import android.content.Context
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase

// Declaración de la base de datos utilizando Room
@Database(entities = [Client::class], version = 1, exportSchema = false)
abstract class AppDatabase : RoomDatabase() {
    // Método abstracto para obtener el DAO de clientes
    abstract fun clientDao(): ClientDao

    companion object {
        @Volatile
        private var INSTANCE: AppDatabase? = null // Instancia única de la base de datos

        // Método para obtener la instancia única de la base de datos
        fun getInstance(context: Context): AppDatabase {
            return INSTANCE ?: synchronized( lock: this) { // Bloque sincronizado para garantizar que solo una instancia sea creada
                val instance = Room.databaseBuilder(
                    context.applicationContext,
                    AppDatabase::class.java, // Clase de la base de datos
                     name: "clientes.db" // Nombre del archivo de la base de datos en el almacenamiento
                ).build()
                INSTANCE = instance
                instance
            }
        }
    }
}
```

iii. Clase Cliente

La entidad Client es la base para almacenar y recuperar la información de los clientes desde la base de datos. Los datos almacenados aquí serán utilizados por otras clases para operaciones como enviar SMS o mostrar listas de clientes.

1. **Anotación @Entity:**

Marca la clase como una entidad de Room, lo que significa que representará una tabla en la base de datos.

Por defecto, el nombre de la tabla será el mismo que el nombre de la clase (Client).

2. **Propiedad id:**

Es la clave primaria de la tabla, definida con la anotación @PrimaryKey.

autoGenerate = true: Permite que Room genere automáticamente valores únicos para esta columna.

3. **Propiedades name y phoneNumber:**

Representan las columnas adicionales de la tabla.

name: Almacena el nombre del cliente.

phoneNumber: Almacena el número de teléfono del cliente.

4. **Constructor con valores predeterminados:**

id tiene un valor predeterminado de 0, lo que permite su generación automática.

Esto facilita la creación de objetos Client sin necesidad de especificar manualmente un ID.

```kotlin
package pmdm.com.midletmensajero

import androidx.room.Entity
import androidx.room.PrimaryKey

// Define una entidad de Room que representa la tabla "Client" en la base de datos
@Entity
data class Client(
    @PrimaryKey(autoGenerate = true) val id: Int = 0, // ID único autogenerado para cada cliente
    val name: String, // Nombre del cliente
    val phoneNumber: String // Número de teléfono del cliente
)
```

iv. Clase ClienteDAO

El DAO ClientDao se usa en la clase AppDatabase para realizar operaciones sobre los datos almacenados en la base de datos. Implementa funcionalidades como:

- Añadir nuevos clientes.
- Consultar la lista completa de clientes.
- Eliminar o actualizar información de clientes

1. **Anotación @Dao**:

- Marca la interfaz como un DAO (Data Access Object).
- Room utiliza esta interfaz para generar el código necesario para interactuar con la base de datos.

2. **Métodos CRUD (Create, Read, Update, Delete)**:

- **insert(client: Client)**:
    - Inserta un nuevo cliente en la tabla Client.
    - Usa la anotación @Insert para que Room genere la lógica de inserción.
- **getAllClients()**:
    - Recupera todos los registros de la tabla Client.
    - La consulta SQL se define con la anotación @Query.
- **getClientById(id: Int)**:
    - Recupera un cliente específico según su ID.
    - Devuelve un objeto Client o null si no se encuentra el cliente.
- **delete(client: Client)**:
    - Elimina un cliente de la tabla.
    - La anotación @Delete genera la lógica de eliminación.
- **update(client: Client)**:
    - Actualiza los datos de un cliente existente.
    - La anotación @Update maneja la lógica para modificar los registros existentes.

```kotlin
package pmdm.com.midletmensajero

import androidx.room.*

// Define el DAO (Data Access Object) para gestionar operaciones en la tabla "Client"
@Dao
interface ClientDao {

    // Inserta un cliente en la base de datos
    @Insert
    fun insert(client: Client)

    // Recupera todos los clientes de la tabla
    @Query("SELECT * FROM Client")
    fun getAllClients(): List<Client>

    // Recupera un cliente específico basado en su ID
    @Query("SELECT * FROM Client WHERE id = :id")
    fun getClientById(id: Int): Client?

    // Elimina un cliente de la tabla
    @Delete
    fun delete(client: Client)

    // Actualiza los datos de un cliente existente
    @Update
    fun update(client: Client)
}
```

v. Clase ListaClientes

Gestiona la lista de clientes y permite al usuario realizar acciones clave como visualizar, modificar o eliminar clientes. Es una parte central de la funcionalidad de la app.

1. **Base de datos**:

- Se utiliza AppDatabase para obtener acceso a la tabla Client.
- Se consulta la base de datos en segundo plano usando coroutines (Dispatchers.IO).

2. **RecyclerView**:

- Muestra la lista de clientes utilizando un adaptador (AdaptadorCliente).
- El diseño del RecyclerView se organiza verticalmente con LinearLayoutManager.

3. **Botones de acción**:

- btnAniadirCliente: Navega a la actividad AniadirClientes para agregar nuevos clientes.
- btnAtras: Finaliza la actividad actual.

4. **Gestión de clientes**:

- **loadClients()**:
    - Obtiene la lista de clientes de la base de datos y la pasa al adaptador.
- **showOptionsDialog(client)**:
    - Muestra un cuadro de diálogo con opciones para modificar o borrar un cliente.
- **deleteClient(client)**:
    - Elimina un cliente de la base de datos.
    - Actualiza la lista para reflejar los cambios.
- **navigateToModifyClient(client)**:
    - Inicia la actividad ModificarCliente pasando el ID del cliente seleccionado.

```
class ListaClientes : AppCompatActivity() {

    // Declaración de variables para gestionar la interfaz y la base de datos
    private lateinit var recyclerView: RecyclerView
    private lateinit var clientAdapter: AdaptadorCliente
    private lateinit var database: AppDatabase

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.lista_clientes)

        // Inicializar la base de datos Room
        database = AppDatabase.getInstance( context: this)

        // Inicializar el RecyclerView
        recyclerView = findViewById(R.id.recyclerView)
        recyclerView.layoutManager = LinearLayoutManager( context: this)

        // Configurar el botón para añadir un nuevo cliente
        findViewById<Button>(R.id.btnAniadirCliente).setOnClickListener {
            // Navegar a la pantalla de añadir cliente
            startActivity(Intent( packageContext: this, AniadirClientes::class.java))
        }

        // Configurar el botón para volver atrás
        findViewById<Button>(R.id.btnAtras).setOnClickListener {
            finish() // Finaliza la actividad actual y vuelve a la anterior
        }

        // Cargar la lista de clientes desde la base de datos
        loadClients()
    }
```

```kotlin
// Este método se ejecuta cada vez que la actividad se reanuda
override fun onResume() {
    super.onResume()
    loadClients() // Recargar la lista de clientes
}

// Método para cargar la lista de clientes desde la base de datos
private fun loadClients() {
    lifecycleScope.launch {
        val clients = withContext(Dispatchers.IO) {
            database.clientDao().getAllClients() // Obtiene todos los clientes
        }

        // Configurar el adaptador con los clientes obtenidos
        clientAdapter = AdaptadorCliente(clients) { client ->
            showOptionsDialog(client) // Mostrar diálogo de opciones al hacer clic en un cliente
        }
        recyclerView.adapter = clientAdapter // Asignar el adaptador al RecyclerView
    }
}

// Mostrar un diálogo con opciones (Modificar o Borrar) para un cliente específico
private fun showOptionsDialog(client: Client) {
    val options = arrayOf("Modificar", "Borrar")
    AlertDialog.Builder( context: this)
        .setTitle("Seleccione una opción")
        .setItems(options) { _, which ->
            when (which) {
                0 -> navigateToModifyClient(client) // Navegar a la pantalla de modificar cliente
                1 -> deleteClient(client) // Eliminar el cliente
            }
        }
        .show()
}
```

```kotlin
// Método para eliminar un cliente de la base de datos
private fun deleteClient(client: Client) {
    lifecycleScope.launch {
        withContext(Dispatchers.IO) {
            database.clientDao().delete(client) // Elimina el cliente de la base de datos
        }
        Toast.makeText( context: this@ListaClientes, text: "Cliente eliminado", Toast.LENGTH_SHORT).show()
        loadClients() // Recargar la lista de clientes después de la eliminación
    }
}

// Navegar a la pantalla de modificar cliente
private fun navigateToModifyClient(client: Client) {
    val intent = Intent( packageContext: this, ModificarCliente::class.java)
    intent.putExtra( name: "CLIENT_ID", client.id) // Pasa el ID del cliente a la siguiente actividad
    startActivity(intent)
}
```

vi. Clase AniadirCliente

Esta clase permite a los usuarios agregar nuevos clientes a la base de datos. Es una funcionalidad clave para actualizar los datos de los clientes, ya que las otras pantallas dependen de estos datos.

1. **Base de datos**:

- AppDatabase.getInstance(this) inicializa la base de datos Room para permitir operaciones CRUD (Crear, Leer, Actualizar, Eliminar) sobre la tabla de clientes.

2. **Validación**:

- Antes de guardar el cliente, el método valida que ambos campos (nombre y teléfono) no estén vacíos. Si lo están, muestra un mensaje al usuario.

3. **Interacción con la base de datos**:

- Usa una coroutine con Dispatchers.IO para realizar la operación de guardar en segundo plano. Esto asegura que no se bloquee el hilo principal (UI) mientras se inserta el cliente en la base de datos.

4. **Guardar clientes**:

- Después de guardar, muestra un mensaje al usuario indicando que el cliente fue guardado correctamente.
- Finaliza la actividad con finish() para regresar automáticamente a la pantalla anterior.

```
class AniadirClientes : AppCompatActivity() {
    private lateinit var baseDatos: AppDatabase

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.aniadir_cliente)

        // Inicializar la base de datos
        baseDatos = AppDatabase.getInstance( context: this)

        // Referencias a los elementos de la interfaz
        val nombreIntroducido = findViewById<EditText>(R.id.textNombreCliente)
        val telefonoIntroducido = findViewById<EditText>(R.id.textTelefono)

        // Configurar el botón para guardar el cliente
        findViewById<Button>(R.id.btnGuardarCliente).setOnClickListener {
            // Obtener el texto introducido en los campos
            val nombre = nombreIntroducido.text.toString().trim()
            val telefono = telefonoIntroducido.text.toString().trim()

            // Validar que los campos no estén vacíos
            if (nombre.isEmpty() || telefono.isEmpty()) {
                Toast.makeText( context: this, text: "Introduce un nombre y un teléfono", Toast.LENGTH_SHORT).show()
            } else {
                guardarCliente(nombre, telefono) // Guardar cliente en la base de datos
            }
        }
    }

    // Método para guardar un cliente en la base de datos
    private fun guardarCliente(nombre: String, telefono: String) {
        lifecycleScope.launch {
            // Insertar cliente en un hilo secundario
            withContext(Dispatchers.IO) {
                val nuevoCliente = Client(name = nombre, phoneNumber = telefono) // Crear objeto Client
                baseDatos.clientDao().insert(nuevoCliente) // Insertar cliente en la base de datos
            }
            // Mostrar confirmación al usuario en el hilo principal
            Toast.makeText( context: this@AniadirClientes, text: "Cliente guardado correctamente", Toast.LENGTH_SHORT).show()
            finish() // Cierra la actividad y regresa a la anterior
        }
```

vii. Clase ModificarCliente

Esta clase permite al usuario actualizar los datos de un cliente existente en la base de datos. Se integra con otras clases como ListaClientes y ClientDao para lograr una interacción fluida entre la interfaz y la base de datos.

1. **Cargar datos iniciales**:

- Utiliza el ID del cliente, recibido a través del Intent, para buscar sus datos actuales en la base de datos (getClientById).
- Los datos se muestran en los campos de texto (EditText) para que el usuario los modifique.

2. **Actualizar cliente**:

- Al hacer clic en el botón "Actualizar Cliente", los valores de los campos se obtienen y se usan para crear un nuevo objeto Client.
- Este objeto actualizado se envía al método update del DAO para actualizar los datos en la base de datos.

3. **Validación mínima**:

- Se asume que los valores ingresados son válidos. En una implementación real, sería ideal agregar validaciones (por ejemplo, que los campos no estén vacíos o que el teléfono tenga un formato correcto).

4. **Notificación al usuario**:

- Después de actualizar el cliente, se muestra un mensaje de confirmación (Toast) y se cierra la actividad.

5. **Uso de corrutinas**:

- Las operaciones que interactúan con la base de datos se ejecutan en un hilo secundario (Dispatchers.IO) para no bloquear la interfaz de usuario.

6. **Layout modificar_cliente**:

- Este layout contiene:
    - Dos EditText para el nombre (textActualizarNombre) y teléfono (textActualizarTelefono).
    - Un Button para guardar los cambios (btnActualizarCliente).

```
class ModificarCliente : AppCompatActivity() {
    private lateinit var database: AppDatabase // Instancia de la base de datos
    private var clientId: Int = 0 // ID del cliente a modificar

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.modificar_cliente) // Enlaza el layout que permite modificar los datos del cliente

        database = AppDatabase.getInstance( context: this) // Inicializa la base de datos
        clientId = intent.getIntExtra( name: "CLIENT_ID",  defaultValue: 0) // Obtiene el ID del cliente desde el intent

        // Cargar los datos actuales del cliente para mostrarlos en los campos de texto
        lifecycleScope.launch {
            val client = withContext(Dispatchers.IO) {
                database.clientDao().getClientById(clientId)
            }
            if (client != null) {
                // Rellena los campos de texto con el nombre y teléfono del cliente
                findViewById<EditText>(R.id.textActualizarNombre).setText(client.name)
                findViewById<EditText>(R.id.textActualizarTelefono).setText(client.phoneNumber)
            }
        }

        // Configuración del botón "Actualizar Cliente"
        findViewById<Button>(R.id.btnActualizarCliente).setOnClickListener {
            // Obtiene los valores actualizados desde los campos de texto
            val newName = findViewById<EditText>(R.id.textActualizarNombre).text.toString()
            val newPhone = findViewById<EditText>(R.id.textActualizarTelefono).text.toString()

            lifecycleScope.launch {
                withContext(Dispatchers.IO) {
                    // Crea un nuevo objeto cliente con los datos actualizados
                    val updatedClient = Client(clientId, newName, newPhone)
                    // Actualiza el cliente en la base de datos
                    database.clientDao().update(updatedClient)
                }
                // Muestra una confirmación al usuario
                Toast.makeText( context: this@ModificarCliente,  text: "Cliente actualizado", Toast.LENGTH_SHORT).show()
                finish() // Cierra la actividad y regresa a la anterior
            }
        }
```

viii. Clase ModificarPlantilla

Esta clase permite al usuario modificar una plantilla de mensaje guardada previamente. Esta plantilla se guarda utilizando SharedPreferences. Al usar SharedPreferences, los cambios en la plantilla se mantienen entre sesiones de la aplicación, lo que proporciona persistencia de datos.

1. **SharedPreferences**:

- SharedPreferences se utiliza para almacenar y recuperar datos pequeños y simples (como configuraciones o preferencias del usuario).
- En este caso, se guarda la plantilla del mensaje bajo la clave "PLANTILLA_MENSAJE". Si no hay una plantilla guardada, se muestra una plantilla predeterminada: "Enhorabuena *, has sido seleccionado para...".

2. **Carga de la plantilla**:

- Al iniciar la actividad, la plantilla actual (si existe) se carga en el EditText para que el usuario pueda verla y modificarla.
- Si no se ha guardado previamente una plantilla, se carga una plantilla predeterminada.

3. **Guardar la plantilla**:

- Al pulsar el botón "Guardar Plantilla", el texto del EditText se obtiene y se guarda en SharedPreferences bajo la clave "PLANTILLA_MENSAJE".
- Si el campo está vacío, muestra un mensaje de error (Toast).
- Si la plantilla se guarda correctamente, muestra un mensaje de éxito y cierra la actividad.

4. **Notificación al usuario**:

- Utiliza un Toast para mostrar mensajes cortos de confirmación o error, lo cual proporciona retroalimentación al usuario.

5. **Layout modificar_plantilla**:

- Este layout contiene:
    - Un EditText para que el usuario ingrese o edite la plantilla (editTextPlantilla).
    - Un Button para guardar la plantilla (btnGuardarPlantilla).

ix. Clase AdaptadorCliente

La clase AdaptadorCliente es un adaptador personalizado para un RecyclerView que gestiona una lista de objetos Client. Esta clase se encarga de inflar las vistas y asignarles los datos correspondientes.

1. **Parámetros**:

- **clients**: Una lista de objetos Client que contiene los datos que se mostrarán en cada ítem de la lista.
- **onLongClick**: Una función que se ejecutará cuando el usuario haga un click largo en un ítem. Se pasa como parámetro una función de alto orden (callback) que recibe un objeto Client.

2. **Métodos clave**:

- **onCreateViewHolder**:
    - Infla el layout item_cliente para crear un nuevo ítem de la lista.
    - Retorna un ViewHolder que se utiliza para representar visualmente el ítem dentro del RecyclerView.
- **onBindViewHolder**:
    - Este método se llama para cada ítem en el RecyclerView. En él, se enlazan los datos del cliente con las vistas correspondientes (el nombre y el número de teléfono en los TextView).
    - También se configura un OnLongClickListener en cada ítem, lo que permite ejecutar la acción definida por el parámetro onLongClick cuando el usuario hace un click largo.

- **getItemCount**:
    - Devuelve el número de elementos en la lista de clientes, lo que permite al RecyclerView saber cuántos ítems tiene que mostrar.

3. **ViewHolder (ClientViewHolder)**:

- Esta clase interna extiende de RecyclerView.ViewHolder y sirve para representar los elementos visualmente. Contiene referencias a los TextView para el nombre y número de teléfono del cliente.

4. **Interacción con el RecyclerView**:

- Cada vez que el RecyclerView necesita mostrar un nuevo ítem, onCreateViewHolder infla la vista correspondiente, y onBindViewHolder asigna los datos de cada cliente a esa vista.
- Al hacer un click largo en cualquier ítem, se ejecuta la acción proporcionada en el parámetro onLongClick.

```
package pmdm.com.midletmensajero

import ...

// Adaptador para manejar la lista de clientes en el RecyclerView
class AdaptadorCliente(
    private val clients: List<Client>,  // Lista de clientes a mostrar
    private val onLongClick: (Client) -> Unit  // Acción a ejecutar cuando se realiza un click largo en un ítem
) : RecyclerView.Adapter<AdaptadorCliente.ClientViewHolder>() {

    // Crear la vista para cada ítem en el RecyclerView
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ClientViewHolder {
        // Infla el layout item_cliente para cada elemento de la lista
        val view = LayoutInflater.from(parent.context).inflate(R.layout.item_cliente, parent,  attachToRoot: false)
        return ClientViewHolder(view)  // Retorna un ViewHolder con la vista inflada
    }

    // Llenar la vista con los datos de un cliente
    override fun onBindViewHolder(holder: ClientViewHolder, position: Int) {
        val client = clients[position]  // Obtiene el cliente correspondiente a la posición
        holder.nameTextView.text = client.name  // Asigna el nombre del cliente al TextView
        holder.phoneTextView.text = client.phoneNumber  // Asigna el número de teléfono al TextView

        // Configura el evento de click largo
        holder.itemView.setOnLongClickListener {
            onLongClick(client)  // Ejecuta la acción definida en onLongClick
            true  // Indica que el evento fue manejado
        }
    }

    // Número de elementos que tiene el RecyclerView
    override fun getItemCount(): Int = clients.size  // Retorna el tamaño de la lista de clientes

    // ViewHolder para manejar la vista de cada ítem
    inner class ClientViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
        val nameTextView: TextView = itemView.findViewById(R.id.textViewName)  // TextView para el nombre del cliente
        val phoneTextView: TextView = itemView.findViewById(R.id.textViewPhone)  // TextView para el número de teléfono
    }
}
```