

Proyecto Intermodular

2ºDAM

Aplicación TPV para la gestión de un restaurante



Participantes:

Alejandro Bados Palomino

Javier Jose Jarrín Escobar

Marcos Martín Vidal

Asier Berenguer Saiz

Índice:

1. Base de Datos

1.1 Requisitos Iniciales

1.2 Planificación

2. API

2.1 Requisitos iniciales

2.2 Estructura

2.3 Archivos y explicación

3. Android

3.1 Requisitos Iniciales

3.2 Planificación

3.3 Archivos y explicación

3.3.1 ApiClient

3.3.2 Login

3.3.3 Menú Principal

3.3.4 Listado de mesas

3.3.5 Menú de la Orden

3.3.5.1 Añadir Producto

3.3.5.2 Guardar Orden

3.3.5.3 Finalizar Orden

3.3.5.4 Cancelar Orden

3.3.6 Listado de productos

3.3.7 Detalle del producto

4. Windows Presentation Foundation (WPF)

4.1 Requisitos iniciales

4.2 Planificación

4.3 Archivos y explicación

4.3.1 Login

4.3.2 Orders - Client

4.3.3 Tables - Client

4.3.4 Products - Client

4.3.5 Menú - Admin

4.3.6 Products - Admin

4.3.7 Tables - Admin

4.3.8 Users - Admin

4.3.9 Orders - Admin

5. Configuración para usar el proyecto

5.1 Crear proyecto FireBase

5.2 Descargar key de administrador

5.2 Instalar dependencias e iniciar servidor

5.2 Configurar Android para conexión

Bibliografía:

1. Base de Datos

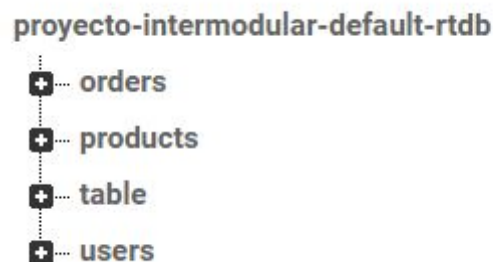
1.1 Requisitos Iniciales

La plataforma que utilizamos para nuestro proyecto es **Firebase**, plataforma desarrollada por Google.

Haremos uso en este proyecto de **Realtime Database**, que es donde tendremos nuestra base de datos guardada y actualizada en tiempo real y **Storage**, que es donde guardaremos todas las imágenes que utilizaremos en nuestro proyecto.

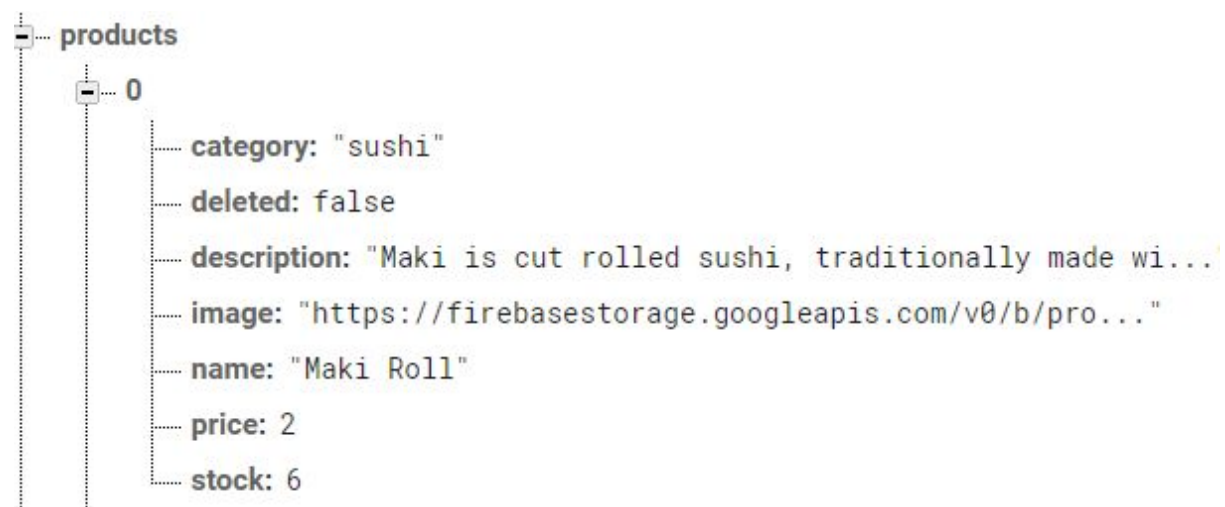
1.2 Planificación

Lo primero que hicimos fue crear un esquema con las tablas que íbamos a utilizar, que son las siguientes:



A partir de aquí ideamos la estructura que tenía que tener cada una de estas tablas para que tuviera sentido

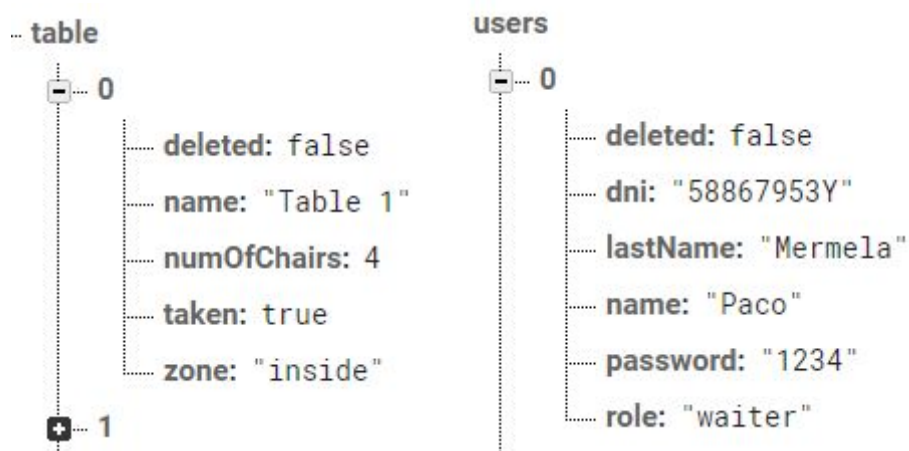
Para empezar creamos los **productos**. Para productos decidimos poner como campos los siguientes:



Los campos más especiales en este caso serían:

- **image:** Este campo contiene una cadena de texto con la dirección de imagen del producto en el storage de firebase, donde las tenemos todas subidas.
- **deleted:** Con este campo en el momento que desde el programa borremos un producto o lo que queramos borrar se mantendrá en la base de datos pero inhabilitado y marcado como borrado.

Las clases **table** y **users** constan de los siguientes campos:

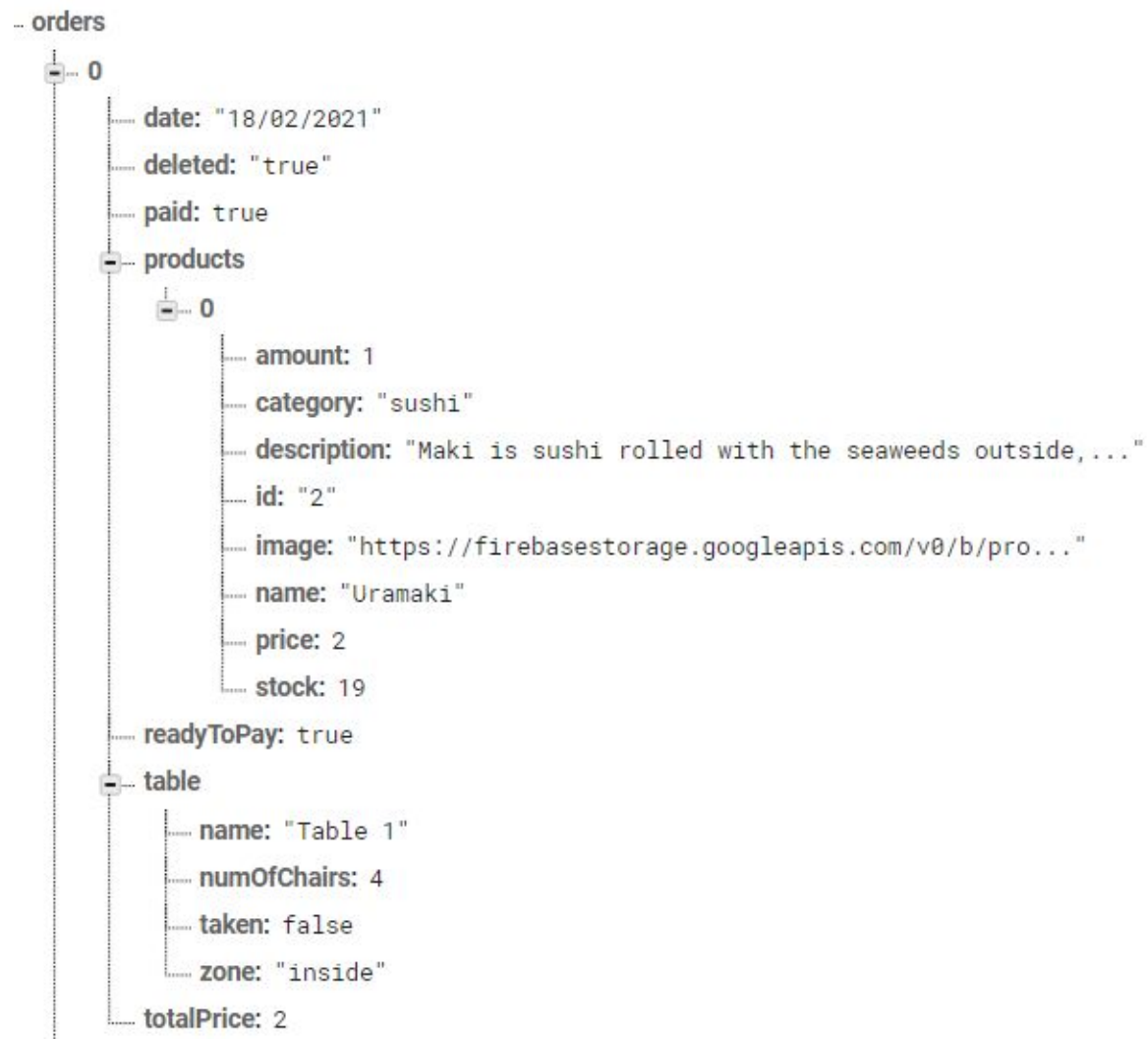


Los campos más especiales en este caso serían:

- **token:** Este campo nos muestra si la mesa está actualmente ocupada o no.
- **name** y **password:** Estos campos serán el nombre de usuario y la contraseña para entrar en la aplicación.

Proyecto Intermodular 2º DAM

La clase **orders** es de lejos la más grande y compleja de todas y es como se muestra a continuación:



Datos interesantes de la tabla:

- **paid**: En este campo se establece si el pedido ha sido pagado y ya se ha finalizado o si sigue abierto.
- **products**: Este campo es un array de todos los productos que se añaden a la orden, en cada uno de los productos se añaden los campos que ya tiene la clase productos más **amount** (cantidad de ese producto en la orden) e **id** (el identificador de producto en su clase).
- **table**: En este campo se le pasa la mesa entera, con todos los datos excepto el delete.
- **totalPrice**: Como el nombre indica en este campo se hace el cálculo para tener el precio total

2. API

2.1 Requisitos iniciales

Lo primero que necesitamos es express porque vamos a controlar todas las peticiones con `express.Router()` y vamos a utilizar el `app.listen` con el puerto 5000 para escuchar todas las peticiones que le vayan llegando a la API.

```
const express = require('express');
```

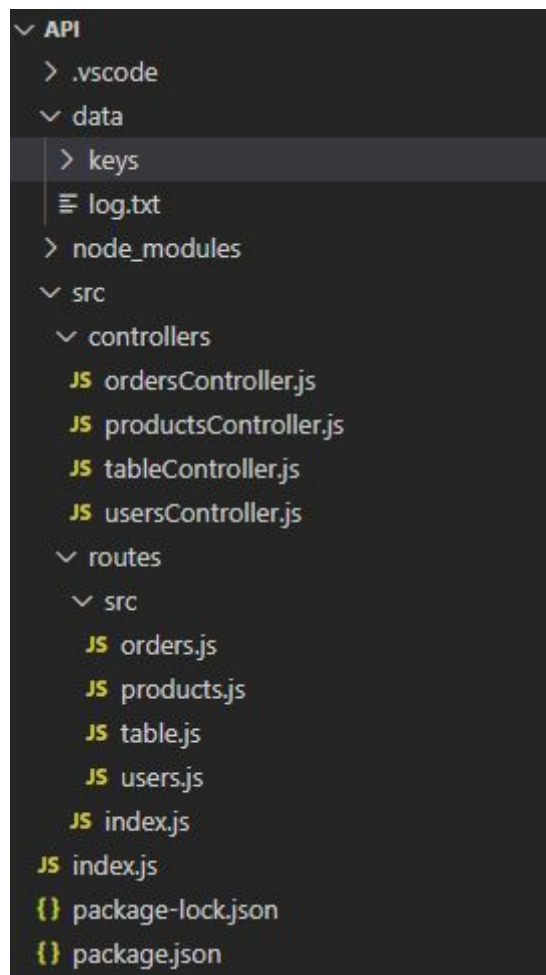
```
const router = express.Router();
```

También vamos a necesitar importar el `firebase-admin` y necesitaremos descargar la key de administrador e iniciar sesión:

```
var admin = require("firebase-admin");
var serviceAccount = require("../data/keys/key-intermodular.json");

admin.initializeApp({
  credential: admin.credential.cert(serviceAccount),
  databaseURL: "https://proyecto-intermodular-default-rtdb.europe-west1.firebaseio.com/"
});
```

2.2 Estructura



Como podemos ver en la foto tenemos el index de la aplicación con sus respectivos package, node_modules y .vscode al programar en visual studio code, por otra parte tenemos las carpetas del proyecto con el siguiente formato:

1. DATA
 - 1.1. KEY
 - 1.1.1. Key del administrador de firebase
2. SRC
 - 2.1. CONTROLLERS
 - 2.1.1. Con los controladores de cada EndPoint.
 - 2.2. ROUTES
 - 2.2.1. Tiene un index.js que será donde exportemos todos los EndPoint
 - 2.2.2. SRC
 - 2.2.2.1. Los archivos donde se relaciona cada ruta con su respectiva función dentro de su controlador.

2.3 Archivos y explicación

Index principal

```
const express = require('express');
const app = express();
const port = 5000;
var admin = require("firebase-admin");
var serviceAccount = require("../data/keys/key-intermodular.json");

admin.initializeApp({
  credential: admin.credential.cert(serviceAccount),
  databaseURL: "https://proyecto-intermodular-default-rtdb.europe-west1.firebaseio.com/"
});

//Todo nuestra API en un principio manejará json's
app.use(express.json());

require('./src/routes/index')(app);

app.listen(port, () => {
  console.log(`Servidor iniciado en el puerto ${port}`);
})
```

Aquí es donde empieza todo, requerimos las librerías de express, hacemos el login con administrador en firebase, parseamos todos los archivos que va a recibir y devolver la api a JSON. Incluimos el index de las rutas y activamos el listen de la API en el puerto 5000 mandando un mensaje de que se ha iniciado correctamente la escucha.

Index de rutas

```
function router(app) {
  app.use('/products', require('./src/products'));
  app.use('/table', require('./src/table'));
  app.use('/users', require('./src/users'));
  app.use('/orders', require('./src/orders'));
}
module.exports = router;
```

Le decimos a la app que deberá usar los endpoints correspondientes y que para cada uno necesitará usar su archivo de rutas adecuado.

Ejemplo de un archivo de rutas

```
const express = require('express');
const controller = require('../controllers/ordersController');

const router = express.Router();
router.get('/', controller.all);
router.get('/:id', controller.one);
router.post('/:id', controller.save);
router.patch('/:id', controller.update);
router.delete('/:id', controller.destroy);

module.exports = router;
```

En este ejemplo hacemos lo mismo que en todos los otros EndPoints, ya que todos van a tener 1 Get de todos los objetos, 1 Get de un objeto en concreto, 1 Post para subir archivos nuevos, 1 Patch para actualizar el objeto y 1 Delete para poder cambiar el deleted a "true" de un objeto en concreto.

Cada uno de estos tipos de peticiones lleva a una función dentro de su respectivo controlador.

Los GETS

```
const admin = require('firebase-admin');
let db = admin.database();
let type = 'products';

async function all(request, response) {
  let ref = db.ref(type);
  get(ref, response);
}

async function one(request, response) {
  let ref = db.ref(type + "/" + request.params.id);
  get(ref, response);
}

async function get(ref, response) {
  ref.once("value", function (snapshot) {
    response.writeHead(200, { 'Content-Type': 'application/json' });
    response.end(JSON.stringify(snapshot.val()));
  }, function (error) {
    console.log("Error get in " + type);
    response.writeHead(404);
    response.end();
  });
}
```

Puesto que los dos son gets y siguen el mismo formato se ha creado una función One para cuando es uno y una que es All para cuando son todos, estas dos funciones le pasan la referencia y la respuesta a la función get para que está termine de manejar y enviar los datos.

Post

```
async function save(request, response) {
  const category = request.body.category;
  const description = request.body.description;
  const image = request.body.image;
  const name = request.body.name;
  const price = request.body.price;
  const stock = request.body.stock;
  const id = request.params.id;
  let newProduct = db.ref(type).child(id);
  newProduct.update({
    category: category,
    description: description,
    image: image,
    name: name,
    price: price,
    stock: stock,
    deleted: false,
  }).then(function () {
    response.writeHead(200, { "Content-Type": "application/json" });
    response.end(JSON.stringify({ msg: "Success" }));
  }).catch(function (error) {
    console.log("Error save in " + type);
    response.writeHead(404, { "Content-Type": "application/json" });
    response.end(JSON.stringify({ msg: "Fail" }));
  });
}
```

Esta función crea un nuevo hijo dentro del tipo de objeto de la base de datos y le pasa todas sus propiedades.

Update

```
async function update(request, response) {
  let ref = db.ref(type + "/" + request.params.id);
  const category = request.body.category;
  const description = request.body.description;
  const image = request.body.image;
  const name = request.body.name;
  const price = request.body.price;
  const stock = request.body.stock;
  ref.update({
    category: category,
    description: description,
    image: image,
    name: name,
    price: price,
    stock: stock,
    deleted: false,
  }).then(function () {
    response.writeHead(200, { "Content-Type": "application/json" });
    response.end(JSON.stringify({ msg: "Success" }));
  }).catch(function (error) {
    console.log("Error update in " + type);
    response.writeHead(404, { "Content-Type": "application/json" });
    response.end(JSON.stringify({ msg: "Fail" }));
  });
}
```

En esta función directamente apunta al objeto con el id que se pasó por parámetros y le actualiza los datos

Delete

```
async function destroy(request, response) {  
  const id = request.params.id;  
  let newProduct = db.ref(type).child(id);  
  newProduct.update({  
    deleted: true  
  }).then(function () {  
    response.writeHead(200, { "Content-Type": "application/json" });  
    response.end(JSON.stringify({ msg: "Success" }));  
  }).catch(function (error) {  
    console.log("Error destroy in " + type);  
    response.writeHead(404, { "Content-Type": "application/json" });  
    response.end(JSON.stringify({ msg: "Fail" }));  
  });  
}
```

En este caso lo que va a hacer es un update únicamente cambiando el deleted a true, aunque desde el cliente la sensación que da es que ha borrado el producto.

3. Android

3.1 Requisitos Iniciales

El proyecto ha sido realizado para dispositivos con sistema operativo Android haciendo uso del lenguaje de programación de Java en el IDE de Android Studio anunciado por Google en 2013 como sustituto de Eclipse.

La aplicación está soportada por el SO de Android a partir de su versión 6.0 Marshmallow, con las siguientes dependencias:

```
implementation 'androidx.appcompat:appcompat:1.2.0'
implementation 'com.google.android.material:material:1.3.0'
implementation 'androidx.constraintlayout:constraintlayout:2.0.4'
testImplementation 'junit:junit:4.13.2'
androidTestImplementation 'androidx.test.ext:junit:1.1.2'
androidTestImplementation 'androidx.test.espresso:espresso-core:3.3.0'
```

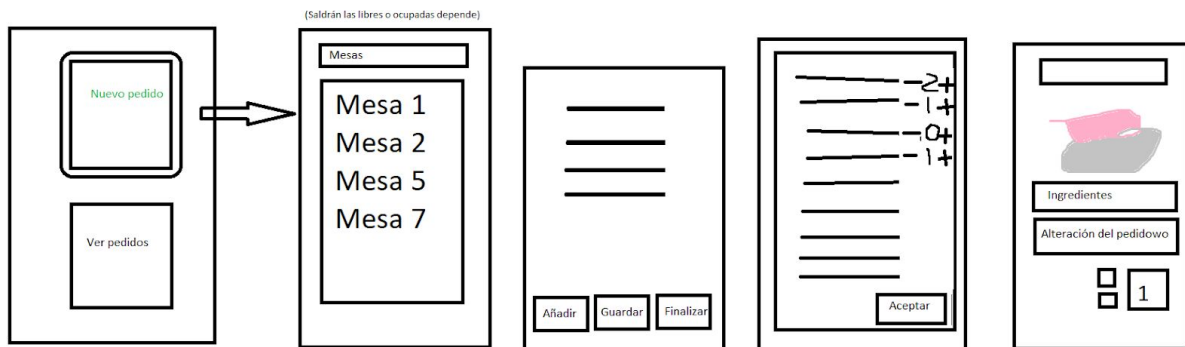
Y con las siguientes librerías externas:

```
implementation 'com.squareup.picasso:picasso:2.71828'
implementation 'com.google.code.gson:gson:2.8.6'
implementation 'com.loopj.android:android-async-http:1.4.9'
```

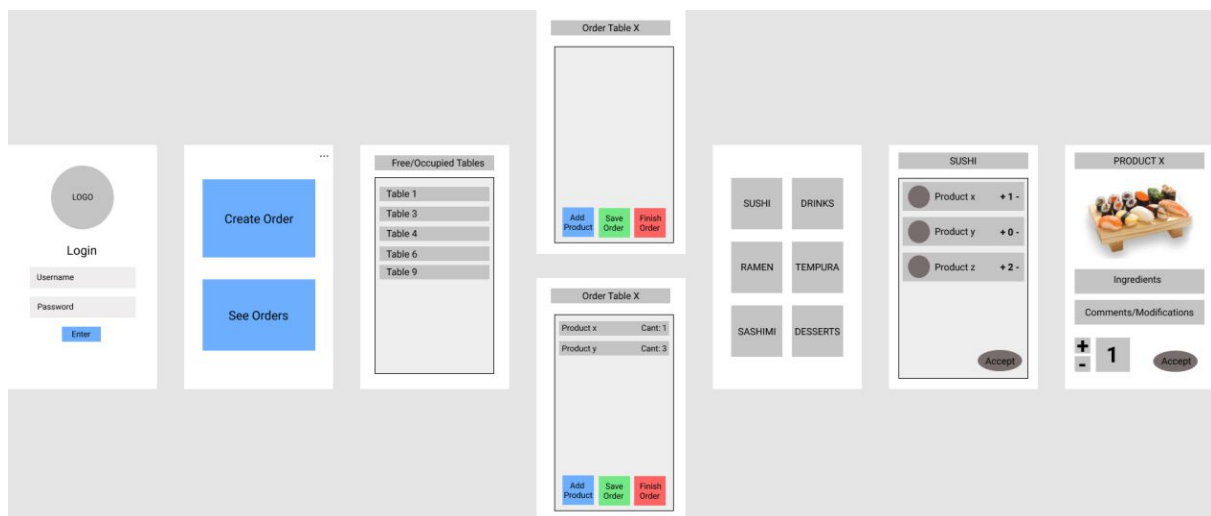
- **Picasso:** librería para Android Studio para la carga y descarga de imágenes, mediante la url de esta, usando el mínimo de memoria.
- **GSON:** librería desarrollada por Google para serializar y deserializar JSON en Java. También se puede usar para convertir un string de JSON a su equivalente clase en Java.
- **Loopj:** librería para realizar llamadas asíncronas en Android, en base a las librerías de Apache Http Client. Permite realizar diferentes peticiones POST, GET, PATCH o DELETE a una url, en nuestro caso a la API Rest.

3.2 Planificación

La planificación de las vistas fue desarrollada al principio usando Paint, y teniendo en mente las funcionalidades que le queríamos aplicar a la aplicación, como las diferentes listas con filtros:



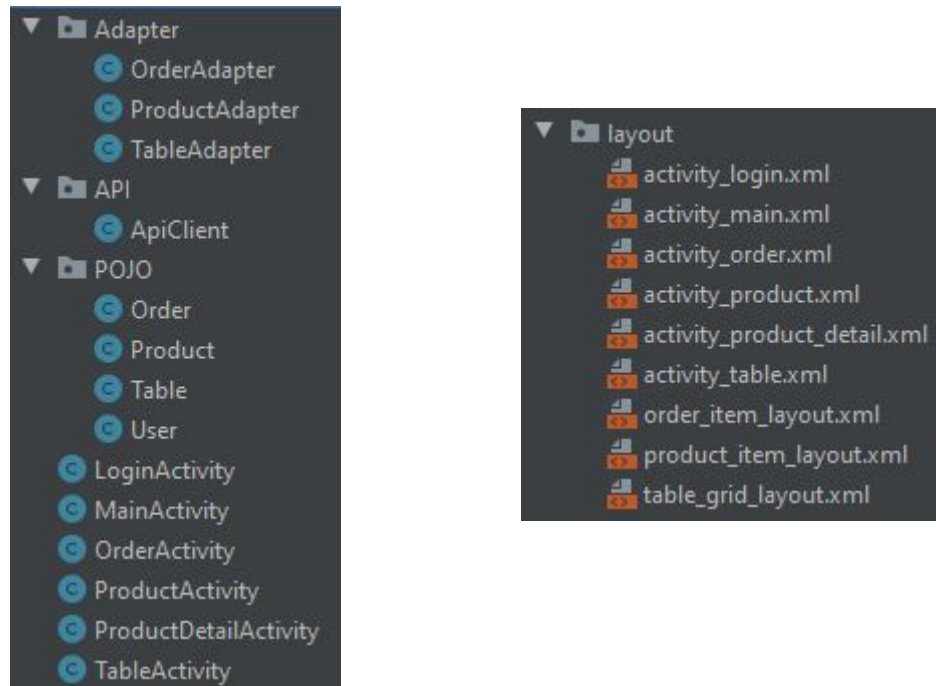
Una vez plasmada la idea procedimos a crear las vistas usando una página especializada para la creación de mockups llamada **Figma**, para que tuviera un aspecto más profesional y detallado.



Con el mockup desarrollado y una estructura clara para seguir, comenzó el desarrollo de la aplicación en Android Studio haciendo uso de las diferentes layouts de ejemplo y ayuda del IDE. Además de los diferentes recursos como iconos y documentación en Android Developers

3.3 Archivos y explicación

La versión final del proyecto de Android presenta la siguiente estructura de archivos:



- **Adapter:** contiene los diferentes adaptadores asociados a una RecyclerView concreta. Una RecyclerView es un tipo de lista para cargar grandes colecciones o conjuntos de datos con la particularidad de que va reciclando las vistas a medida que se va desplazando por ella.
- **API:** este paquete contiene el archivo ApiClient en el cual se usa la librería para realizar llamadas http a la API
- **POJO:** Plain Old Java Objects, las clases de las diferentes tablas de la base de datos de Firebase, con sus diferentes variables y constructores.
- **Activities:** las diferentes actividades, cada una relacionada a una vista específica
- **Layout:** cada vista tiene asociada una actividad la cual carga el layout correspondiente

3.3.1 ApiClient

Es el archivo clave para el correcto funcionamiento del proyecto, comunicándose con la API y realizando peticiones para traer y enviar json desde y hacia la base de datos.

Para ello se le debe indicar la ip del equipo en el que se esté ejecutando la API:

```
private static final String API_BASE_URL = "http://192.168.1.104:5000";
```

Y siguiendo la documentación de la librería de llamadas asíncronas utilizamos la siguiente función para añadir los endpoints deseados a la hora de realizar peticiones:

```
public String getApiUrl(String relativeUrl) { return API_BASE_URL + relativeUrl; }
```

El archivo tiene 4 funciones clave, las cuales realizan las peticiones principales a la API:

```
public void get(String url, AsyncHttpResponseHandler responseHandler) {  
    client.get(getApiUrl(url), responseHandler);  
}  
  
public void post(Context ctx, String url, StringEntity entity, String contentType,  
    AsyncHttpResponseHandler responseHandler){  
    client.post(ctx, getApiUrl(url), entity, contentType, responseHandler);  
}  
  
public void patch(Context ctx, String url, StringEntity entity, String contentType,  
    AsyncHttpResponseHandler responseHandler){  
    client.patch(ctx, getApiUrl(url), entity, contentType, responseHandler);  
}  
  
public void delete(String url, AsyncHttpResponseHandler responseHandler){  
    client.delete(getApiUrl(url), responseHandler);  
}
```

La función get y delete son las más simples, solamente se le indica el endpoint al que se quiere atacar y devuelve un json, el get recibe un array de objetos y el delete un mensaje si se ha realizado correctamente la petición.

Las funciones post y patch necesitan el contexto de la actividad en la que se encuentran, el endpoint al que atacar, el JSON que se quiere enviar en el body de la petición metiendolo dentro de una variable StringEntity, indicarle el tipo de contenido(en nuestro caso application/json) y por último el handler de la respuesta.

Proyecto Intermodular 2º DAM

Para poder realizar las peticiones post y patch de las órdenes, mesas o productos se han añadido también varios constructores que al pasarles por parámetro un objeto lo parsean a un objeto JSON con la misma estructura que el de la base de datos. Por ejemplo la función de un orden:

```
public JSONObject jsonOrder(Order order, Boolean ready) throws JSONException {
    JSONObject jsonTable = new JSONObject();
    jsonTable.put( name: "name", order.getTable().getName());
    jsonTable.put( name: "taken", order.getTable().getTaken());
    jsonTable.put( name: "numOfChairs", order.getTable().getNumOfChairs());
    jsonTable.put( name: "zone", order.getTable().getLocation());

    JSONArray jsonProducts = new JSONArray();
    if(order.getProducts() != null) {
        for (Product p : order.getProducts()) {
            JSONObject product = new JSONObject();
            product.put( name: "id", p.getId());
            product.put( name: "name", p.getName());
            product.put( name: "amount", p.getAmount());
            product.put( name: "category", p.getCategory());
            product.put( name: "image", p.getImage());
            product.put( name: "description", p.getDescription());
            product.put( name: "note", p.getNote());
            product.put( name: "price", p.getPrice());
            product.put( name: "stock", p.getStock());
            jsonProducts.put(product);
        }
    }

    JSONObject jsonOrder = new JSONObject();
    jsonOrder.put( name: "paid", order.getPaid());
    jsonOrder.put( name: "date", order.getDate());
    jsonOrder.put( name: "totalPrice", order.getTotalPrice());
    jsonOrder.put( name: "table", jsonTable);
    jsonOrder.put( name: "readyToPay", ready);
    jsonOrder.put( name: "deleted", value: false);
    jsonOrder.put( name: "products", jsonProducts);
    return jsonOrder;
}
```

Te pasa la mesa la mesa y los productos de la lista de la orden a objetos que posteriormente se le añaden a los campos del objeto jsonOrder

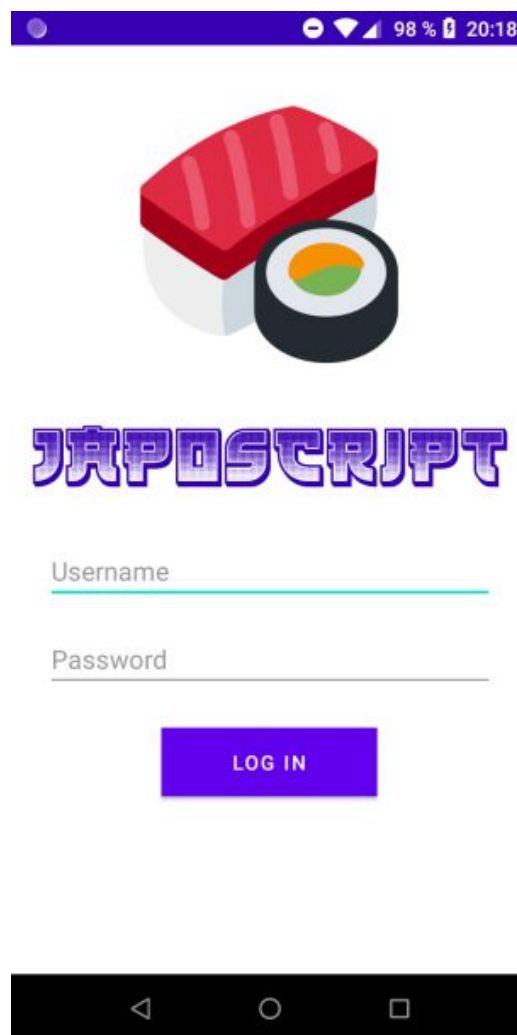
3.3.2 Login

Actividad inicial de la aplicación que aparece al iniciarse. Para ello hay que indicárselo al manifest:

```
<activity
    android:name=".LoginActivity"
    android:screenOrientation="portrait">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

D

Dicha actividad parte de un Constraint Layout, el cual consta de una imagen que contiene el icono de la aplicación, un título con el nombre del restaurante, 2 campos para rellenar, uno para el nombre de usuario y otro para la contraseña, y un botón para hacer el login si los campos son correctos.



Proyecto Intermodular 2º DAM

A la hora de hacer login, el empleado, el cual es el único que puede acceder a la aplicación de móvil, tiene tres intentos para iniciar sesión. Si se fallan esos tres intentos aparecerá una popup indicando que se han acabado los intentos y que se cerrará la aplicación.

Para validar con la base de datos se realiza una petición get al endpoint /users, el cual devuelve un array con todos los objetos usuarios:

```
private void fetchUsers(){
    userList = new ArrayList<User>();
    ApiClient client = new ApiClient();
    client.get( url: "/users", new JsonHttpResponseHandler(){
        @Override
        public void onSuccess(int statusCode, Header[] headers, JSONArray response) {
            Log.i( tag: "users", msg: "onSuccess: "+response.toString());
            try{
                for (int i = 0; i < response.length(); i++){
                    User u = gson.fromJson(response.get(i).toString(), User.class);
                    userList.add(u);
                }
                Log.i( tag: "user_list", msg: "onSuccess: "+response.toString());
            }catch (JSONException e){
                e.printStackTrace();
            }
        }

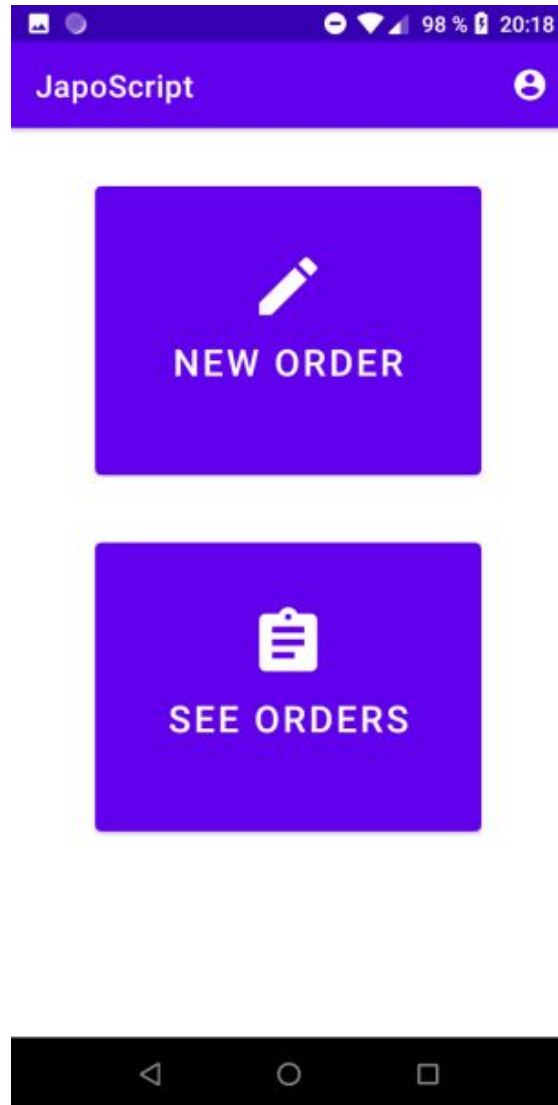
        @Override
        public void onFailure(int statusCode, Header[] headers, Throwable throwable, JSONObject errorResponse) {
            super.onFailure(statusCode, headers, throwable, errorResponse);
        }
    });
}
```

Si la función se ejecuta correctamente la función onSuccess se ejecuta y serializa todos los objetos de la respuesta JSON en objetos User que se añaden a un arraylist de usuarios. Si ha ocurrido algún error la función onFailure.

Entonces al validar que si el username y la contraseña son correctos se ejecutará el startActivity de un nuevo Intent cargará la siguiente actividad y pasará como un extra el objeto usuario del empleado que acaba de hacer login.

3.3.3 Menú Principal

El menú principal es la actividad donde el empleado podrá acceder a las dos acciones que se le permiten: crear órdenes y ver ordenes.



La actividad consta solamente de dos botones y un icono en la ActionBar la cual indica el nombre de usuario y un botón log out por si se quiere cerrar la sesión.

El botón NEW ORDER cargará una nueva actividad para poder crear nuevas órdenes, y el botón SEE ORDERS cargará la actividad que le permitirá ver, editar y borrar las órdenes ya creadas

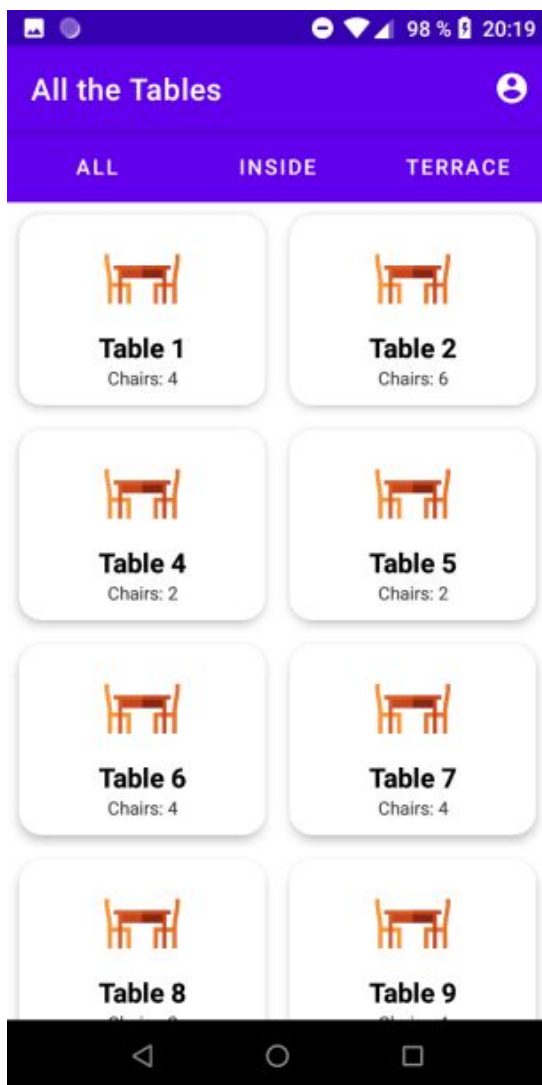
Si el usuario, en el caso de que le dé al botón de atrás, quiere volver a la anterior actividad, saltará un popup preguntando si quiere hacer logout, si le da a que si esta actividad finalizará y se volverá al login. Lo mismo pasa si se pulsa el botón de log out en la ActionBar.

3.3.4 Listado de mesas

Dependiendo de lo que elija el usuario en la actividad anterior, se cargará la actividad TableActivity la cual mediante un filtro mostrará:

- Pulsado NEW ORDER, las mesas que se encuentran libres
- Pulsado SEE ORDERS, las mesas ocupadas que tengan una orden asociada.

En cuanto al estilo, la actividad consta de tres botones en la parte superior indicando los filtros, por los cuales se puede filtrar la lista de abajo.



La lista se puede filtrar por las mesas que se encuentran dentro del restaurante, las mesas que se encuentran en la terraza o que te muestre todas las mesas. La primera opción es la que viene por defecto.

Además cada vez que pulsas sobre una de los filtros el título de la actividad cambiará.

Por último destacar el botón de usuario en el ActionBar de la actividad, el cual como en la actividad anterior muestra el usuario con el que se ha hecho login y un botón para hacer logout.

En este caso el botón de logout no solamente termina la actividad y vuelve al login, si no que elimina la pila de actividades que se ha ido cargando hasta ahora, evitando así que si por ejemplo si el usuario le da al botón de atrás no vuelva a una actividad anterior que siga cargada en la pila de actividades.

En cuanto a la lista de las mesas, esta se trata de un RecyclerView:

```
<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/tablelist"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

Una vez traídas todas las mesas mediante un get al endpoint de /table, en formato JSON y habiéndolas parseado a un objeto Table metiéndolas dentro de un arrayList de tipo Table, se filtran para así solo poder visualizar las mesas que estén libres si hemos elegido crear una orden o las mesas ocupadas en el caso contrario.

También se controla que las mesas no estén eliminadas, ya que cuando se le hace un delete a la api indicando el id del objeto, no se elimina si no que se le cambia el campo booleano llamado deleted pasando a true.

Dicho RecyclerView tiene asociado un adaptador personalizado que va cargando un layout llamado table_grid_layout.xml el cual contiene un CardView en el que se cargará la información de la mesa:



El adapter lo que hace es coger el nombre de la mesa y el número de sillas y las carga dentro de los dos TextView que hay dentro del CardView.

Desde la actividad se declara el tipo el tipo de manejador que el recyclerview usará, el cual en el caso de esta actividad será un GridLayoutManayer para mostrar las mesas en 2 columnas distintas, en vez de mostrarlo como si fuera una lista:

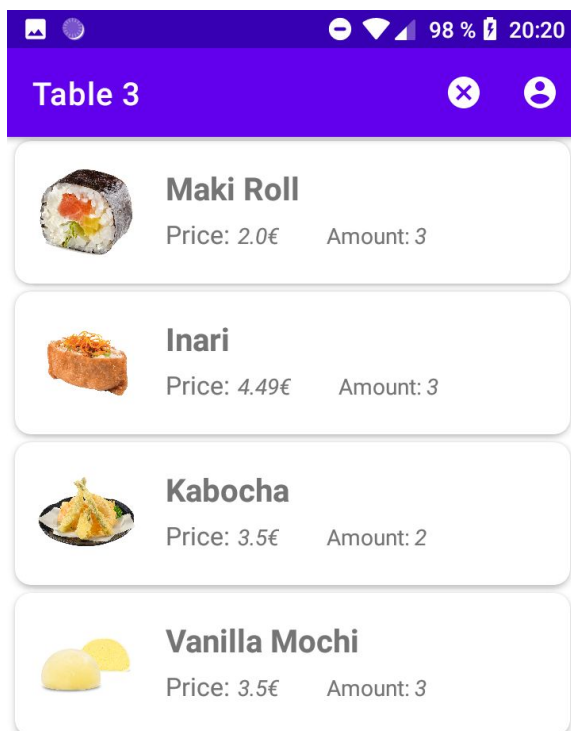
```
public void initRecyclerView(List<Table> tables){
    adapter = new TableAdapter(getApplicationContext(), tables, u);
    GridLayoutManager gridLayoutManager = new GridLayoutManager(
        getApplicationContext(),
        spanCount: 2,
        GridLayoutManager.VERTICAL,
        reverseLayout: false);
    tableList.setLayoutManager(gridLayoutManager);
    tableList.setAdapter(adapter);
}
```

Función para iniciar el RecyclerView de mesas.

3.3.5 Menú de la Orden

Cuando pulsas sobre una mesa en la actividad anterior se abrirá esta nueva actividad sobre la orden asociada a esa mesa. Si la mesa está libre se abrirá una orden vacía, si no, se cargará la orden asociada a esa mesa si se cumplen las siguientes condiciones:

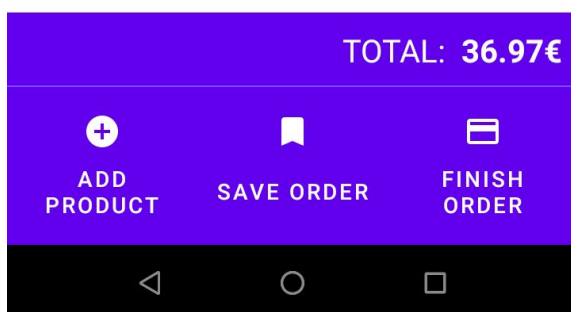
- La mesa coincide con la mesa de la orden
- La orden no está pagada
- La orden no está lista para pagar
- La orden no está eliminada



El estilo de la actividad presenta un RecyclerView que carga los productos de la orden, con la imagen del producto, el nombre, el precio y la cantidad seleccionada.

Justo debajo del RecyclerView tenemos un LinearLayout con orientación horizontal en el que se muestra el precio total de la orden.

Los tres botones de la parte inferior de la actividad están dentro de un LinearLayout con orientación horizontal y que tiene funcionalidades distintas



En el ActionBar, en la parte superior hay dos iconos, uno para cancelar la orden y otro que despliega el nombre del usuario y el botón del logout con la misma finalidad que el de la actividad de la mesas.

3.3.5.1 Añadir Producto

La funcionalidad de este botón es abrir una nueva actividad para poder seleccionar los productos para la orden, pasándole como extra un ArrayList de productos vacío.

3.3.5.2 Guardar Orden

La funcionalidad de este botón es la de guardar la orden cuando se le añaden productos a la lista de la orden. Si la lista de productos está vacía cuando se pulse el botón aparecerá un mensaje abajo indicando que no se puede guardar una orden si está vacía. En el caso de que la lista tenga productos se ejecutará la función createOrder:

```
public void createOrder() throws JSONException, UnsupportedEncodingException {
    order = new Order();
    order.setDate();
    order.setPaid(false);
    order.setProducts(productList);
    order.setTable(selectedTable);
    order.setTotalPrice(getTotalPrice());
    JSONObject json = client.jsonOrder(order, ready: false);
    StringEntity entity = new StringEntity(json.toString());
    String url = "/orders/"+ORDER_ID;
    client.post( context.this,url,entity, contentType: "application/json", new JsonHttpResponseHandler(){
        @Override
        public void onSuccess(int statusCode, Header[] headers, JSONObject response) {
            super.onSuccess(statusCode, headers, response);
            Toast.makeText( context OrderActivity.this, text "Order Created", Toast.LENGTH_SHORT).show();
            try{
                setTableTaken();
                updateStock();
            } catch (JSONException | UnsupportedEncodingException e){
                e.printStackTrace();
            }
        }

        @Override
        public void onFailure(int statusCode, Header[] headers, Throwable throwable, JSONObject errorResponse) {
            super.onFailure(statusCode, headers, throwable, errorResponse);
            Toast.makeText( context OrderActivity.this, text "Error: CreateOrder Failed", Toast.LENGTH_SHORT).show();
        }
    });
}
```

Se nos creará un nuevo objeto Orden al cual se le indicará la fecha actual, si está pagado por defecto a false, se le añadirá la lista de productos, el objeto mesa asociado a la orden y el precio total de la orden. Este objeto orden pasará a JSON usando las funciones de ApiClient para darle el formato adecuado y seguidamente pasará a un StringEntity para poder pasarlo a la petición post.

Si la petición se ha realizado correctamente se ejecutarán dos funciones más una para hacer un update a la mesa de la orden e indicarle que ha pasado de estar libre a esta ocupada, y otra para restarle el stock a los productos de la orden.

En el caso de que la orden ya exista y queramos cambiar cualquier producto de la lista, si pulsamos sobre guardar orden se ejecutará una función distinta, la función UpdateOrder:

```
public void updateOrder() throws JSONException, UnsupportedEncodingException {
    String url = "/orders/"+ORDER_ID;
    order.setProducts(productList);
    order.setTotalPrice(getTotalPrice());
    JSONObject obj = client.jsonOrder(order, ready: false);
    StringEntity entity = new StringEntity(obj.toString());
    client.patch(cbc this,url,entity, contentType: "application/json",new JsonHttpResponseHandler(){
        @Override
        public void onSuccess(int statusCode, Header[] headers, JSONObject response) {
            super.onSuccess(statusCode, headers, response);
            Toast.makeText( context: OrderActivity.this, text: "Order Updated", Toast.LENGTH_SHORT).show();
            try{
                updateStock();
            } catch (UnsupportedEncodingException | JSONException e) {
                e.printStackTrace();
            }
        }

        @Override
        public void onFailure(int statusCode, Header[] headers, Throwable throwable, JSONObject errorResponse) {
            super.onFailure(statusCode, headers, throwable, errorResponse);
            Toast.makeText( context: OrderActivity.this, text: "Error: Update Failed", Toast.LENGTH_SHORT).show();
        }
    });
}
```

Dicha función lo único que cambia es el array de productos y el precio total de la orden, pasando la orden otra vez a JSON para poder pasarlo a la petición patch y si se ejecuta sin problemas realizará otro update del stock.

Cabe destacar que si queremos eliminar un producto de la orden podemos hacer un swipe hacia la izquierda o hacia la derecha para quitar un ítem de la lista, y cuando se quita y se le da a guardar orden se realiza un update del stock restableciendo el stock del ítem que se acaba de borrar

3.3.5.3 Finalizar Orden

La funcionalidad de este último botón es el de ejecutar la función payOrder:

```
public void payOrder() throws JSONException, UnsupportedEncodingException {
    String url = "/orders/"+ORDER_ID;
    JSONObject obj = client.jsonOrder(order, ready: true);
    StringEntity entity = new StringEntity(obj.toString());
    client.patch( cbc this,url,entity, contentType: "application/json",new JsonHttpHandler(){
        @Override
        public void onSuccess(int statusCode, Header[] headers, JSONObject response) {
            Toast.makeText( context OrderActivity.this, text "Order ready to pay", Toast.LENGTH_SHORT).show();
        }

        @Override
        public void onFailure(int statusCode, Header[] headers, Throwable throwable, JSONObject errorResponse) {
            Toast.makeText( context OrderActivity.this, text "Error: not ready to pay", Toast.LENGTH_SHORT).show();
        }
    });
}
```

Dicha función lo que hace a realizar un patch con el id de la orden para cambiar el campo de readyToPay de la orden pasandola a true, ejecutando posteriormente la función clearTable que hace que la mesa pase de estar ocupada a estar libre.

En el caso de que se pulsará este botón y no hubiera ninguna orden cargada , ni asociada a la mesa , aparecerá un mensaje en la parte inferior de la actividad que dice que la orden que se quiere pagar no existe.

3.3.5.4 Cancelar Orden

En la parte superior de la actividad en el ActionBar hay un icono de una x, ese es el botón de cancelar orden el cual si al interactuar con el aun no se ha creado la orden aparecerá un mensaje diciendo que la orden todavía no existe.

Si pulsamos sobre el botón y hay una orden cargada aparecerá una ventana popup que nos preguntará si estamos seguros de que queremos cancelar la orden. Si le damos a no se cerrará la ventana y si le damos a si ejecutará la función CancelOrder, la cual mediante un loop pasa por todos los productos de la lista de la orden y ejecuta un restoreStock:

```
public void restoreStock(Product p) throws JSONException, UnsupportedEncodingException {
    client.get( url: "/products/"+p.getId(), (JsonHttpErrorHandler) onSuccess(statusCode, headers, response) + {
        Log.i( tag: "producto", msg: "onSuccess: "+response.toString());
        try {
            p.setStock(response.getInt( name: "stock"));
        } catch (JSONException e) {
            e.printStackTrace();
        }
    });
    int totalStock = p.getStock()+p.getAmount();
    JSONObject obj = client.jsonProductRestoreStock(p,totalStock);
    StringEntity entity = new StringEntity(obj.toString());
    client.patch(getApplicationContext(), url: "/products/"+p.getId(),entity, contentType: "application/json",new JsonHttpErrorHandler());
}
```

Restablece el stock de todos los productos sumándole la cantidad a el stock,
Función cancelOrder:

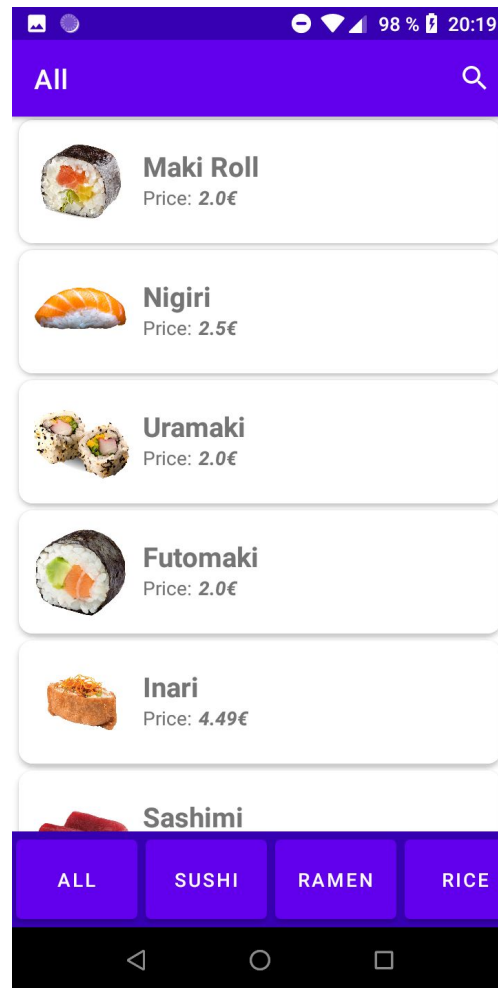
```
public void cancelOrder(MenuItem item){
    try {
        Apiclient client = new Apiclient();
        for (Product p : order.getProducts()){
            restoreStock(p);
        }
        client.delete( url: "/orders/"+ORDER_ID,new JsonHttpErrorHandler(){
            @Override
            public void onSuccess(int statusCode, Header[] headers, JSONObject response) {
                try {
                    clearTable();
                } catch (JSONException | UnsupportedEncodingException e) {
                    e.printStackTrace();
                }
                Toast.makeText( context: OrderActivity.this, text: "Order Canceled", Toast.LENGTH_SHORT).show();
                finish();
            }
        });

        @Override
        public void onFailure(int statusCode, Header[] headers, Throwable throwable, JSONObject errorResponse) {
            Toast.makeText( context: OrderActivity.this, text: "Error: cannot cancel order", Toast.LENGTH_SHORT).show();
        }
    });
} catch (JSONException | UnsupportedEncodingException e) {
    e.printStackTrace();
}
}
```

Una vez realizado el restablecimiento del stock se realiza un /delete al endpoint /orders/:id con el que la api realizará los cambios a la orden con el id que le hayamos indicado para que quede como deleted igual a true.

3.3.6 Listado de productos

Cuando pulsamos sobre añadir producto se nos abre la actividad que contiene la lista con todos los productos:



Dicha lista es un RecyclerView como todas las listas de la aplicación el cual tiene un adaptador personalizado que carga los datos necesarios de cada producto.

El adaptador carga la imagen mediante la librería de Picasso:

```
Picasso.get().load(Uri.parse(productList.get(position).getImage()))
    .error(R.drawable.ic_img_not_found_24)
    .into(holder.ivImage);
```

Se carga la imagen asociada al producto y si ocurre algún error con la url de la imagen se carga una imagen placeholder.

También se controla desde el adaptador la cantidad de stock que tiene cada producto, y si un producto tiene el stock igual a 0 cambiara el color de fondo y no se podrá interactuar con el:

```
if(productList.get(position).getStock() == 0){
    holder.tvName.setText(productList.get(position).getName());
    holder.card.setBackgroundColor(Color.parseColor( colorString: "#ff8080"));
} else {
    holder.card.setBackgroundColor(Color.parseColor( colorString: "#ffffff"));
    holder.tvName.setText(productList.get(position).getName());
}
```



En la parte inferior de la actividad se encuentra un ScrollView con orientación horizontal en el que hay varios botones con las diferentes categorías de productos:



Al pulsar sobre uno la lista de los productos se filtra con los productos de la categoría seleccionada, mediante la función de FilterList:

```
private void filterProductList(String status){
    selectedFilter = status;
    List<Product> filteredProductList = new ArrayList<>();

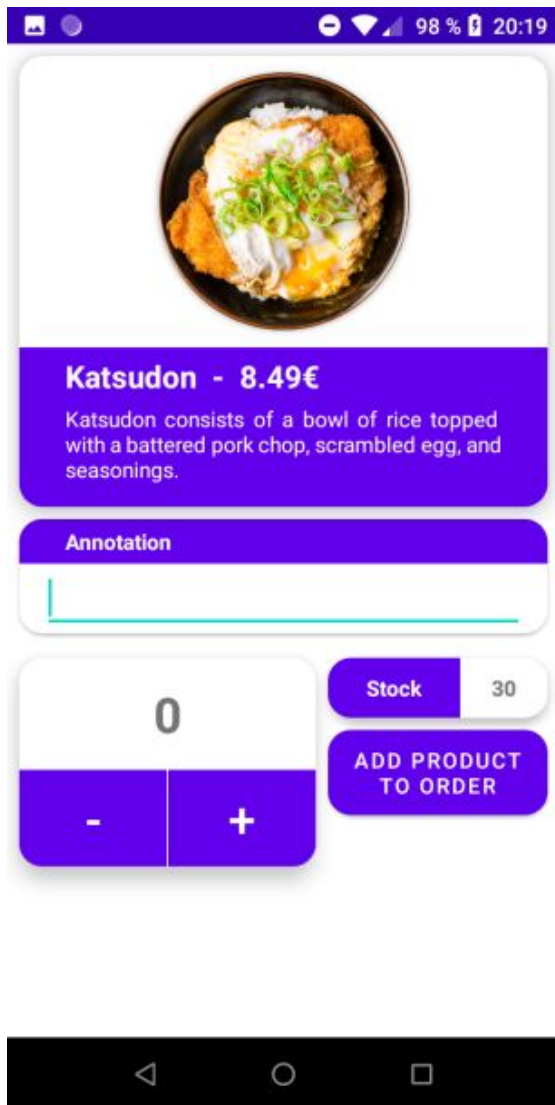
    for (Product p : productList){
        if(p.getCategory().toLowerCase().equals(status)){
            filteredProductList.add(p);
        }
    }

    initRecyclerView(filteredProductList);
}
```

Por último en la parte superior de la actividad hay una barra de búsqueda la cual se actualiza al cambiar el texto que se indique.

3.3.7 Detalle del producto

Dentro del listado de productos si pulsamos sobre cualquier producto que no esté bloqueado porque no tenga stock, se nos abrirá el detalla de ese producto:



El estilo de dicho detalle está formado por un `ConstraintLayout` compuesto de `CardsView`.

Todos los card contienen información del producto, como la imagen, el nombre, el precio, la descripción y el stock.

Además de un apartado para escribir una anotación, si no se escribe nada el campo quedará vacío.

La cantidad se puede sumar y restar, no superando el stock permitido. Y si le damos al botón de añadir a la orden y la cantidad es 0 aparecerá un mensaje diciendo que la cantidad no puede ser 0.

Si le das a atrás sin haberle dado a añadir el producto no se añadirá a la lista de la orden. Y si elegimos una cantidad y le damos a añadir se enviará ese producto a la actividad anterior del listado de productos añadiendo a la lista de la orden que se le pasó desde el menú de la orden.

Esta actividad también se puede abrir desde la actividad del menú de la orden, si pulsamos sobre cualquier producto que esté en la lista de la orden se nos abrirá su detalle con la cantidad que hemos seleccionado anteriormente.

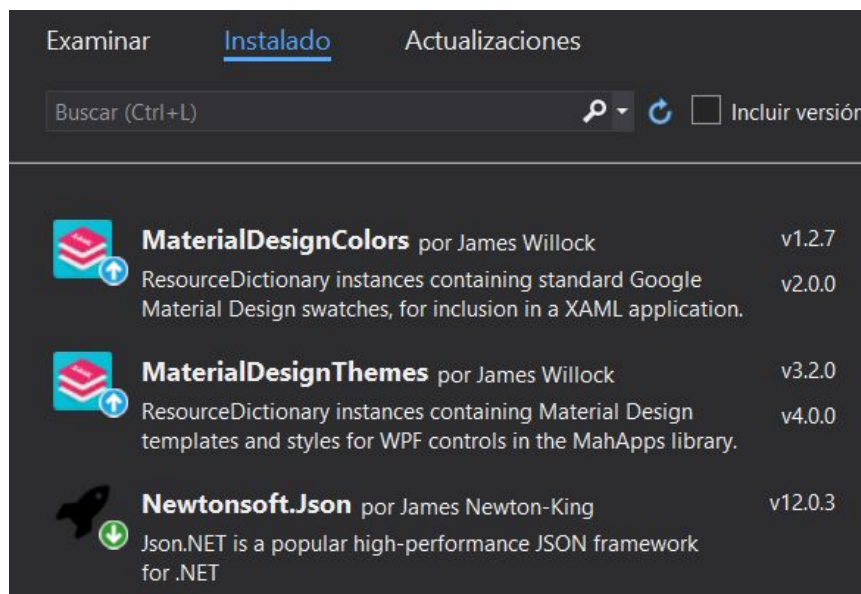
4. Windows Presentation Foundation (WPF)

4.1 Requisitos iniciales

La versión de escritorio fue desarrollada en el IDE Visual Studio haciendo uso de su lenguaje de programación C# y XAML que es el lenguaje de formato para la interfaz de usuario para la base de presentación.

En la aplicación disponemos de los siguientes paquetes que nos ayudaron en el desarrollo del proyecto:

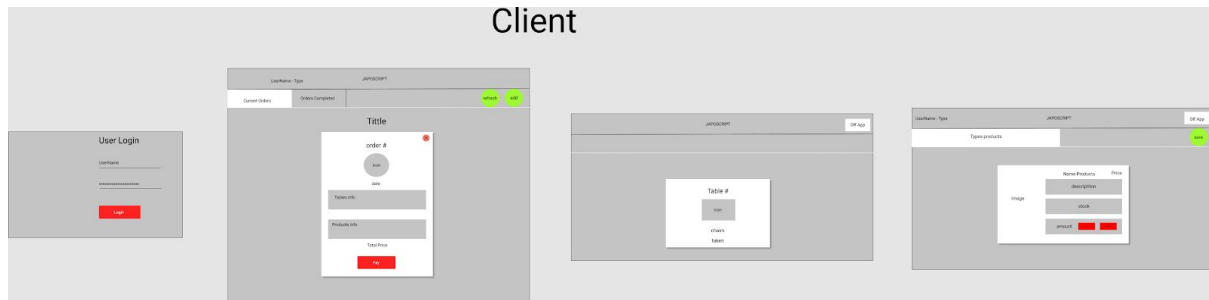
- **MaterialDesingColors And MaterialDesingThemes:** Paquetes de diseño enfocado a la visualización del sistema. Fue desarrollado por Google y anunciado en la conferencia Google I/O celebrada el 25 de junio de 2014.
- **Newtonsoft.Json:** Librería para el uso de archivos JSON.



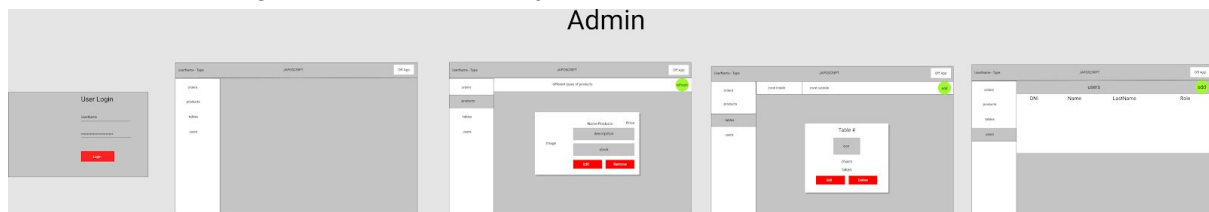
4.2 Planificación

La planificación de las diferentes vistas fueron desarrolladas en **Figma**, una aplicación para diseñar interfaces que se ejecuta en el navegador, y que como herramienta nos sirve para llevar a cabo un diseño mucho más óptimo.

En la versión cliente se desarrollaron cuatro vistas más el login.

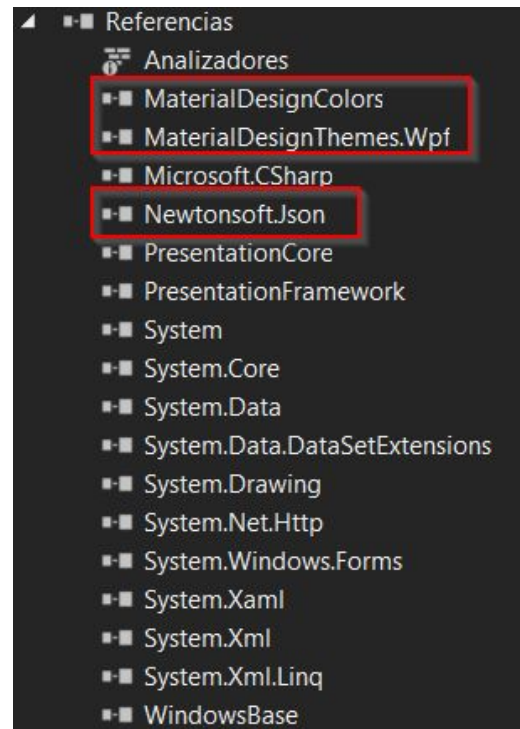
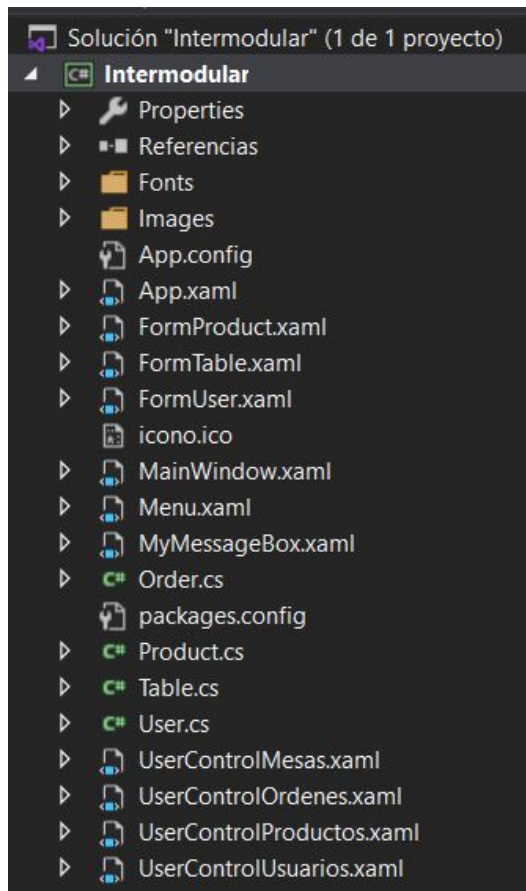


En la versión del administrador se finalizó con cuatro vistas, donde desde el menú se hace el llamado de las siguientes para una mejor optimización.



4.3 Archivos y explicación

Estructura final de archivos.

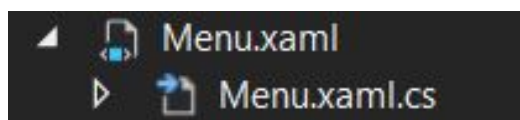


Referencias: Las referencias son básicamente una entrada de un archivo de proyecto que contiene la información que Visual Studio necesita para localizar el componente.

Archivos .xaml: Los archivos de extensión **.xaml** son nuestras vistas donde integramos todos los componentes de visualización, los cuales están asociadas con otro archivo **.xaml.cs**.

Archivos .xaml.cs: En estos archivos es donde manejamos la lógica e interacción con nuestras vistas.

Ejemplo del archivo **Menu.xaml**.



Archivos .cs: Estos archivos son nuestro modelo para la creación de objetos, se han definido cuatro en total que son **Order.cs**, **Product.cs**, **Table.cs**, **User.cs**.

Implementación de MaterialDesing: En los recursos a nivel de aplicación, hacemos una llamada a los paquetes de **MaterialDesing**.

```
<Application x:Class="Intermodular.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:Intermodular"
    StartupUri="MainWindow.xaml">
    <Application.Resources>
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary Source="pack://application:,,,/MaterialDesignThemes.Wpf;component/Themes/MaterialDesignTheme.Light.xaml" />
                <ResourceDictionary Source="pack://application:,,,/MaterialDesignThemes.Wpf;component/Themes/MaterialDesignTheme.Defaults.xaml" />
                <ResourceDictionary Source="pack://application:,,,/MaterialDesignColors;component/Themes/Recommended/Primary/MaterialDesignColor.Red.xaml" />
                <ResourceDictionary Source="pack://application:,,,/MaterialDesignColors;component/Themes/Recommended/Accent/MaterialDesignColor.Lime.xaml" />
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </Application.Resources>
</Application>
```

En cada vista implementaremos un referencia a **materialdesign** para su hacer uso de sus componentes a nivel del diseñador:

```
<UserControl x:Class="Intermodular.UserControlMesas"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:materialDesign="http://materialdesigninxaml.net/winfx/xaml/themes"
    xmlns:local="clr-namespace:Intermodular"
    mc:Ignorable="d"
    d:DesignHeight="650" d:DesignWidth="960">
```

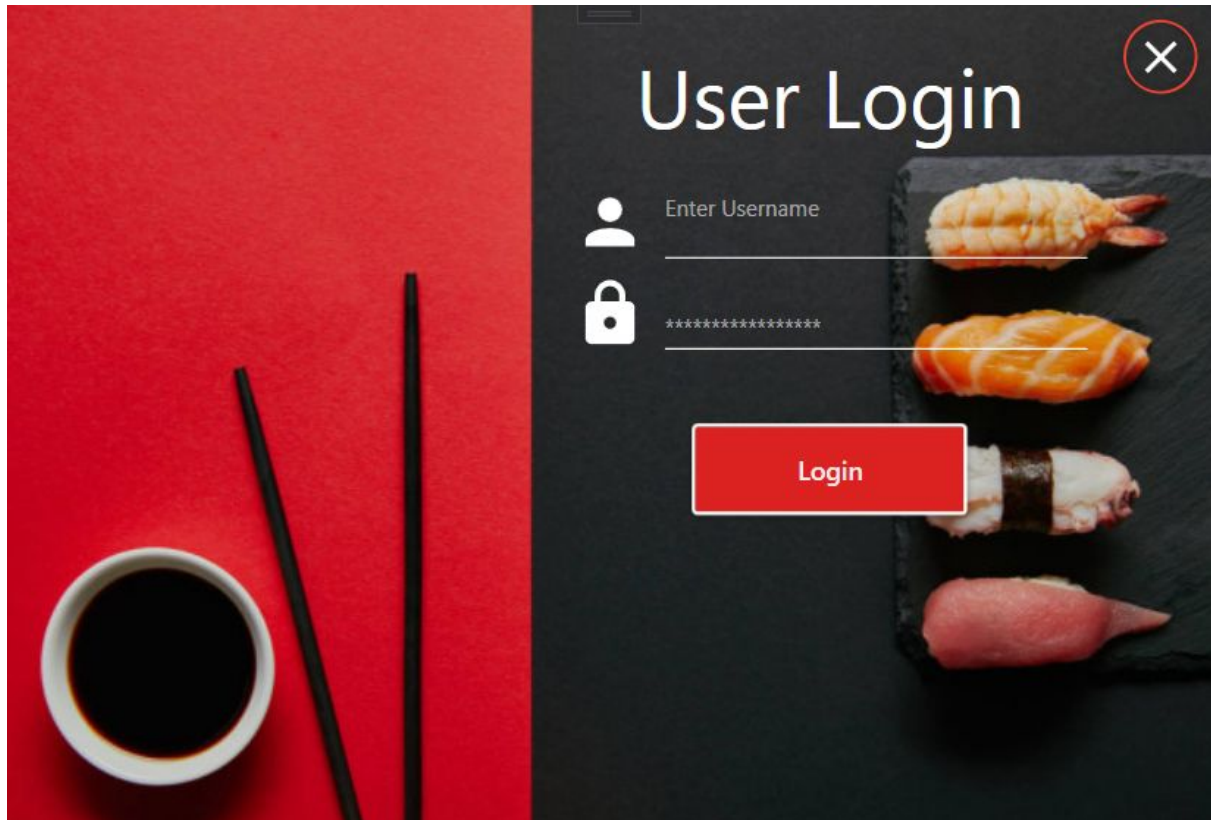
Para usar los componentes de **materialdesign** a nivel de clase lo importamos de la siguiente manera:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Controls.Primitives;
using System.Windows.Input;
using System.Windows.Media;
using Newtonsoft.Json;
using System.Net;
using System.Net.Http;
using System.Threading;
using MaterialDesignThemes.Wpf; //MD

namespace Intermodular
{
    /// <summary>
    /// Lógica de interacción para UserControlMesas.xaml
    /// </summary>
    4 referencias
    public partial class UserControlMesas : UserControl
    {
```

4.3.1 Login

El login es la primera ventana que se carga al ejecutarse la aplicación, consta de 2 campos de texto y un botón para validar que los usuarios existan en nuestra base de datos, y dependiendo que tipo de usuario sean, te llevará a una u otra ventana.



Para validar los usuarios, los obtenemos haciendo una petición tipo get a nuestra API que se encuentra en **localhost:5000/users** que nos devolverá un archivo JSON con los usuarios de la base de datos. Después vamos a deserializar el archivo para convertirlo en un objeto tipo List<User> y así poder hacer uso del mismo.

```
public static List<User> ListUsers = new List<User>();
int quantity = 0;
2 referencias
public UserControlUsuarios()
{
    InitializeComponent();
    RefreshDataGrid();
}

2 referencias
private void getUsers()
{
    ListUsers.Clear();
    quantity = 0;

    string url = "http://localhost:5000/users";
    WebClient wc = new WebClient();
    var datos = wc.DownloadString(url);
    List<User> pros = JsonConvert.DeserializeObject<List<User>>(datos);
    for (int i = 0; i < pros.Count; i++)
    {
        if (!pros[i].Deleted)
        {
            InsertUser(pros[i].Deleted,
                pros[i].Dni,
                pros[i].LastName,
                pros[i].Name,
                pros[i].Password,
                pros[i].Role,
                i
            );
        }
        quantity++;
    }
}
```

4.3.2 Orders - Client

Los usuarios de tipo rol “waiter” son los encargados de acceder a este apartado, el cual tiene dos ítems y dos botones, uno añadir y otro refrescar. El botón de añadir nos permitirá crear una nueva orden y el botón de refrescar se utilizará para estar al tanto de las órdenes que estén listas para pagar, ya que nos daremos cuenta al cambiarse el color de la **card** a verde.

En las **card** podremos observar información relevante de la orden como el número, la fecha, el número de mesa y número de sillas, y los productos elegidos con su cantidad y total a pagar.

En los ítems se visualizarán los órdenes actuales y finalizadas.

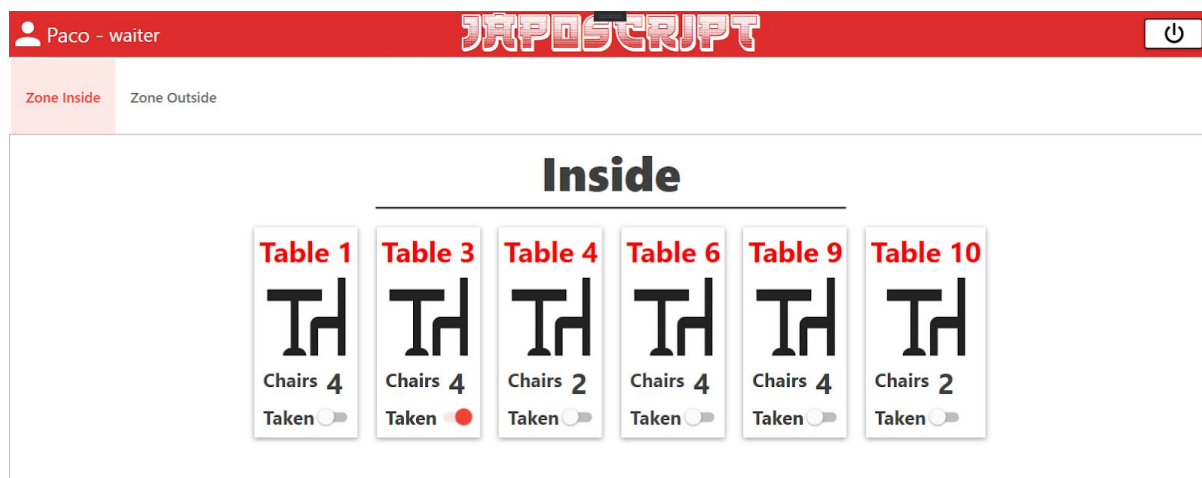
The screenshot shows the JapoScript waiter interface. At the top, a red header bar contains the user name 'Paco - waiter', the JapoScript logo, and a power button. Below the header, there are two tabs: 'Current Orders' (active) and 'Orders Completed'. To the right of the tabs are two green circular buttons with a refresh icon and a plus icon. The main area is titled 'Current Orders' and displays three order cards. Each card has a title (Order 26, Order 28, Order 29), a date (25/02/2021), a table number, number of chairs, and a list of products with their quantities and prices. The total price and a 'Pay' button are at the bottom of each card. Order 26 is white, Order 28 is white, and Order 29 is green.

Order	Date	Table	Num Of Chairs	Products	Total
Order 26	25/02/2021	Table 4	2	Maki Roll 2€ x 1 = 2 € Nigiri 2.5€ x 2 = 5 €	7 €
Order 28	26/02/2021	Table 6	4	Maki Roll 3€ x 2 = 6 € Nigiri 2.5€ x 2 = 5 €	11 €
Order 29	25/02/2021	Table 1	5	Ebi 1.5€ x 9 = 13,5 €	13,5 €

Los pasos para la creación del pedido van en orden, pasando por esta ventana, después seguida de **mesas** para elegir una, y por últimos pasando por los **productos** a elegir.

4.3.3 Tables - Client

En esta ventana se visualizan las mesas a elegir, seguido de dos ítems que hacen un pequeño filtro para ver las mesas en dos zonas, las cuales son **zone Inside** y **zone Outside**.



Para elegir la mesa solo basta con clicar en el card de la mesa donde no esté tomada (taken) ya que de lo contrario no dejaría, este método comprueba que se pueda elegir una mesa libre.

```

+ referencia
private void Card_MouseDown(object sender, MouseButtonEventArgs e)
{
    String date = DateTime.Now.ToString("dd/MM/yyyy");
    Card card = (Card)sender;
    Table table = new Table();

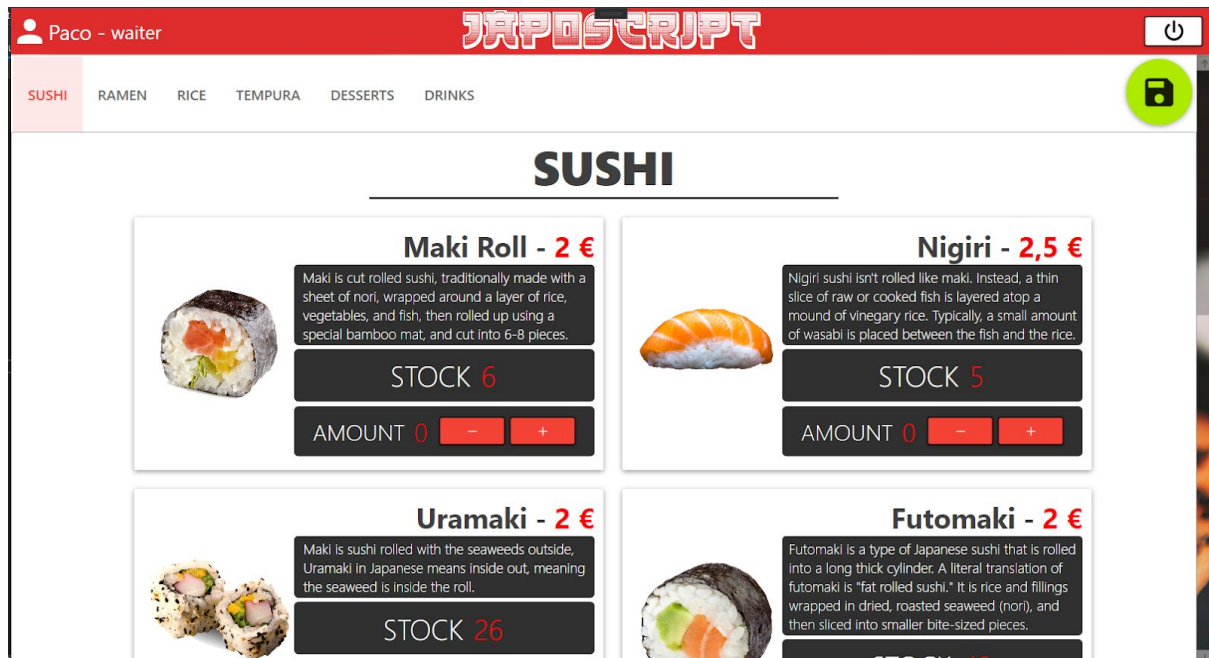
    for (int i = 0; i < ListTables.Count; i++)
    {
        if (ListTables[i].Name.Substring(ListTables[i].Name.Length - (card.Name.Length - 1), card.Name.Length - 1) == card.Name.Substring(1, card.Name.Length - 1))
        {
            table = ListTables[i];
        }
    }
    if (!table.Taken)
    {
        if (table.Name != "")
        {
            Order x = new Order(date, false, false, new List<Product>(), table, 0, false);
            Grid grid = (Grid)this.Parent;
            grid.Children.Add(new UserControlProductos(admin, x, numTable));
        }
        else
        {
            MyMessageBox.ShowBox("Error selecting the table", "Error", "Ok");
        }
    }
    else
    {
        MyMessageBox.ShowBox("Table Taken", "Warning", "Ok");
    }
}

```

4.3.4 Products - Client

En el apartado de productos se caracteriza por dividir los productos en categorías, cada producto se encuentra en un card con su nombre, precio, descripción, cantidad de productos que se encuentran actualmente, y un contador para elegir la cantidad.

Si el stock se encuentra en cero, la card cambia de color a amarillo para hacer referencia al stock.



Estos métodos los cuales están en la clase **Product** nos permiten añadir la cantidad de productos, si el stock está vacío no añadirá ninguno, y la cantidad tiene que ser mayor a cero para poder elegirlo.

```
1 referencia
public void addAmount()
{
    if (Stock > 0)
    {
        Amount++;
        Stock--;
    }
}

1 referencia
public void restAmount()
{
    if (Amount > 0)
    {
        Stock++;
        Amount--;
    }
}
```


4.3.5 Menú - Admin

La sección de administrador cuenta con cuatro opciones para cargar las vistas que son Orders, Products, Table y Users.

Cada vista se carga dentro del menú lo que hace que esta sección sea mucho más óptima.



Método que usa el menú para desplazarse por las diferentes vistas.

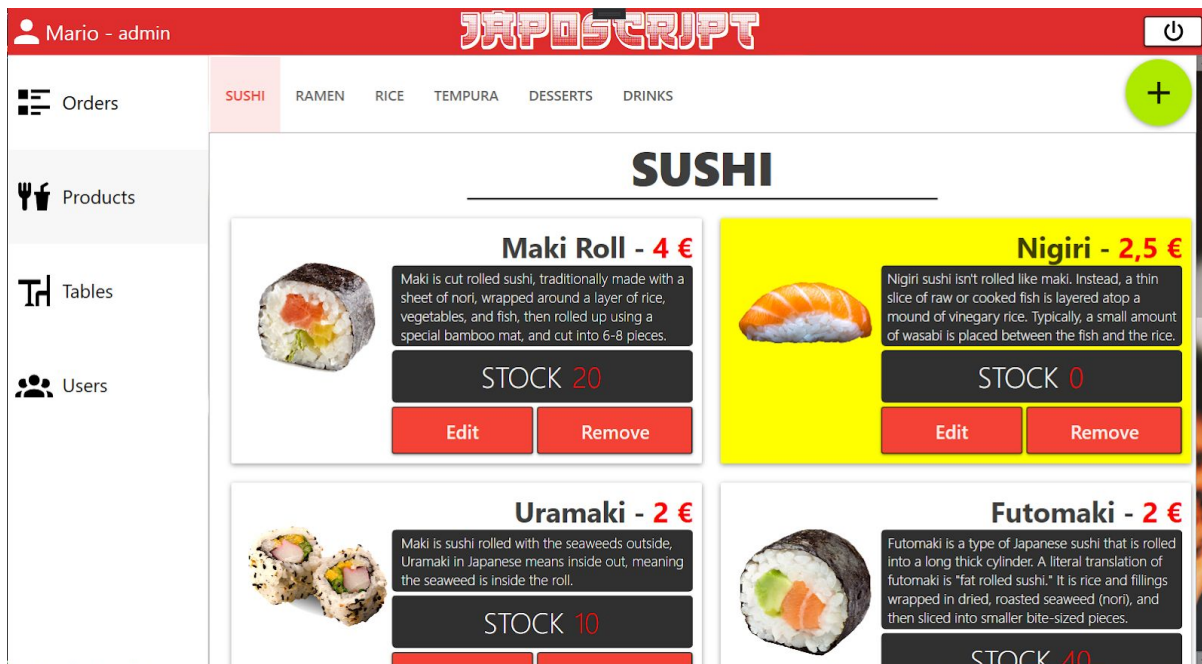
```
private void ListViewMenu_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    int index = ListViewMenu.SelectedIndex;

    switch (index)
    {
        case 0:
            GridPrincipal.Children.Clear();
            GridPrincipal.Children.Add(new UserControlOrdenes(admin));
            break;
        case 1:
            GridPrincipal.Children.Clear();
            GridPrincipal.Children.Add(new UserControlProductos(admin));
            break;
        case 2:
            GridPrincipal.Children.Clear();
            GridPrincipal.Children.Add(new UserControlMesas(admin, -1));
            break;
        case 3:
            GridPrincipal.Children.Clear();
            GridPrincipal.Children.Add(new UserControlUsuarios());
            break;
        default:
            break;
    }
}
```

4.3.6 Products - Admin

En el apartado de productos el administrador cuenta con las siguientes funciones:

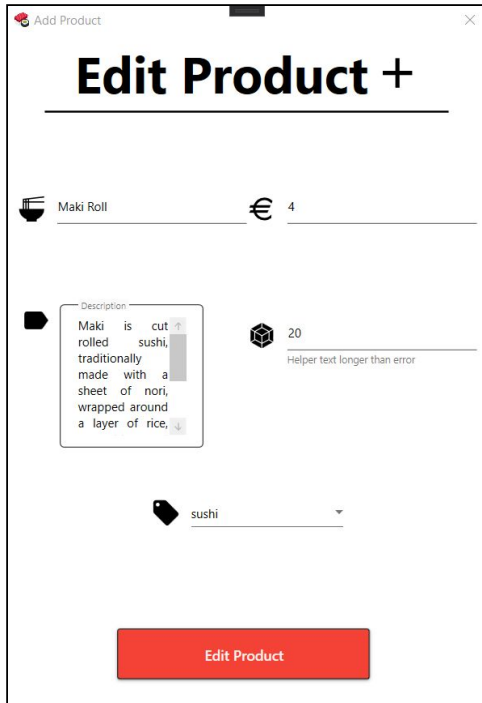
- Añadir un producto
- Editar un producto
- Eliminar un producto



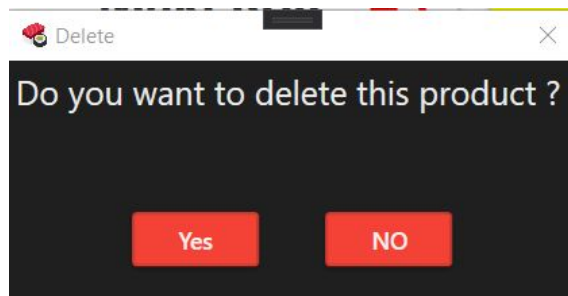
Añadir producto: Al pulsar el botón de añadir, nos saldrá un formulario que debemos completar con los siguientes campos de texto:

- nombre
- precio
- descripción
- stock
- categoría

Editar Producto: En editar producto hacemos uso del mismo formulario para evitar la creación de más vistas y reutilizar.

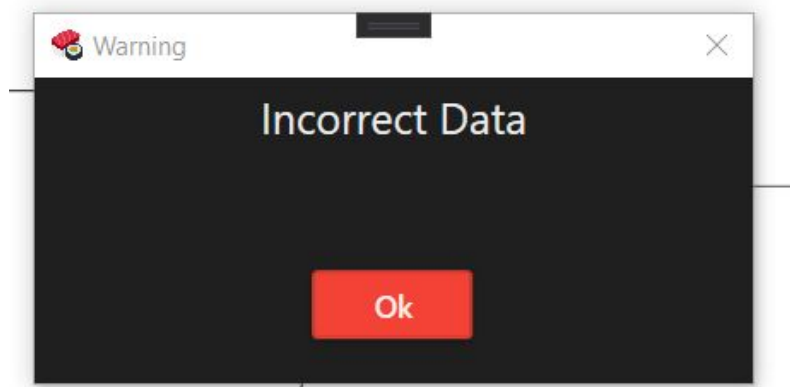


Eliminar producto: Antes de eliminar un producto nos saldrá un mensaje de confirmación para que el usuario esté seguro y no elimine ningún producto por error.



Todos los formularios cuentan con validación en sus campos, para así evitar cualquier error en la aplicación.

Si dejamos cualquier campo vacío, o introducimos un dato mal, nos saldrá este mensaje haciendo advertencia de que algo falló.



Proyecto Intermodular 2º DAM

En el siguiente método **AddproductUserControl()** recibe como parámetro las propiedades para la creación del producto, el cual pasamos al constructor de la clase.

Para enviar una respuesta a la API tipo post debemos serializar nuestro objeto en un JSON lo cual hacemos **con JsonConvert** que son devuelve el objeto en tipo json para después hacer el post.

```
1 referencia
public void AddProductUserControl(string category, string description, string name, double price, int stock, string image, string id)
{
    HttpClient wc = new HttpClient();
    Product product = new Product( category, description, name, price, stock, image, id);
    product.Image = "https://firebasestorage.googleapis.com/v0/b/proyecto-intermodular.appspot.com/o/icono.png?alt=media&token=0d765241-6284-4583-bd55-ea563b2b3588";
    string json = JsonConvert.SerializeObject(product);
    string url = "http://localhost:5000/products/" + quantity;
    var content = new StringContent(json, Encoding.UTF8, "application/json");
    wc.PostAsync(url, content);

    getProducts();
}

1 referencia
public void EditProductUserControl(string category, string description, string name, double price, int stock, string image, string id)
{
    HttpClient wc = new HttpClient();
    Product product = new Product(category, description, name, price, stock, image, id);
    string json = JsonConvert.SerializeObject(product);
    string url = "http://localhost:5000/products/" + id;
    var content = new StringContent(json, Encoding.UTF8, "application/json");
    _ = PatchAsync.patchAsync(wc, url, content, new CancellationToken());

    getProducts();
}
```

Clase **Product** que usamos como modelo para la creación del producto.

```
3 referencias
public Product(string category, string description, string name, double price, int stock, string image, string id)
{
    this.Category = category;
    this.Description = description;
    this.Name = name;
    this.Price = price;
    this.Image = image;
    this.Stock = stock;
    this.ID = id;
}
```

```
[JsonProperty("id")]
5 referencias
public string ID { get; set; }
[JsonProperty("category")]
6 referencias
public string Category { get; set; }
[JsonProperty("description")]
6 referencias
public string Description { get; set; }
[JsonProperty("name")]
9 referencias
public string Name { get; set; }
[JsonProperty("price")]
9 referencias
public double Price { get; set; }
[JsonProperty("image")]
9 referencias
public string Image { get; set; }
[JsonProperty("stock")]
12 referencias
public int Stock { get; set; }
[JsonProperty("deleted")]
1 referencia
public bool Deleted { get; set; }
[JsonProperty("note")]
1 referencia
public string note { get; set; }
[JsonProperty("amount")]
12 referencias
public int Amount { get; set; }
```

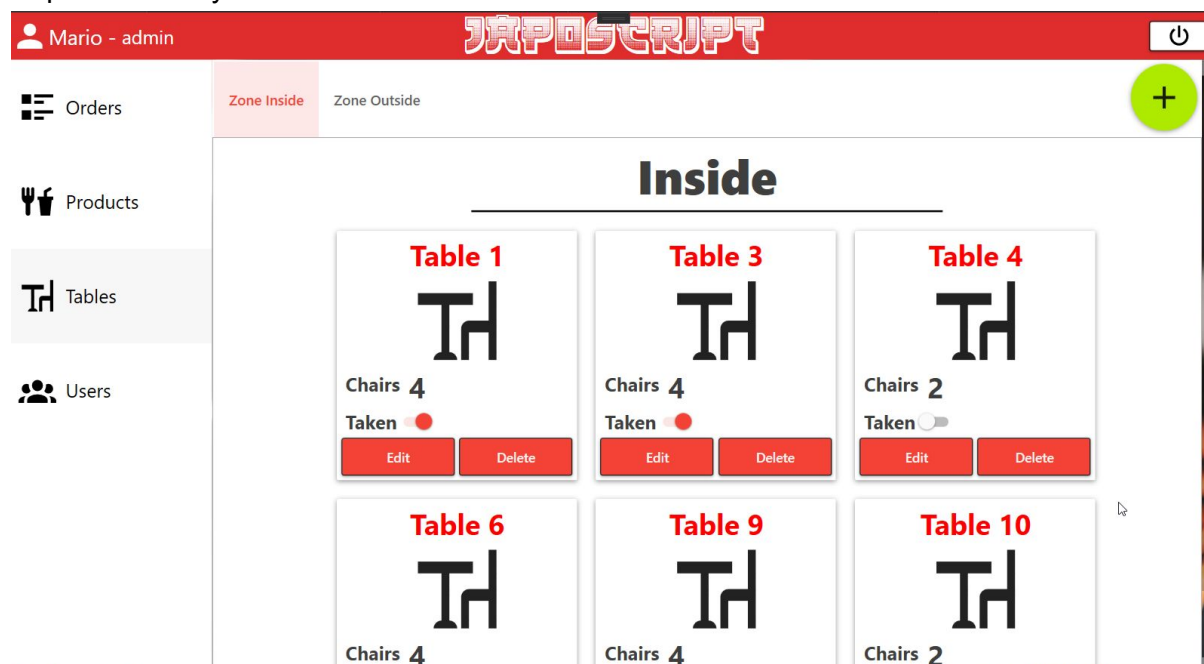
Importar para el Uso de **JsonProperty**

```
using Newtonsoft.Json;
```

Para la que la Serialización y Deserialización de los diferentes objetos funciones, las **JsonProperty** nos ayuda a modificar en nombre de propiedad.

4.3.7 Tables - Admin

En la sección de las mesas se encuentra un botón para la creación de las mesas, también se podrá editar y eliminar.



Hacemos uso de los diferentes formularios para editar y añadir una mesa

Eliminar una tabla:

```

1 referencia
private void DeleteTable(object sender, RoutedEventArgs e)
{
    string btnClicked = MyMessageBox.ShowBox("Do you want to delete this table ? ", "Delete", "Yes", "NO");
    if (btnClicked == "1")
    {
        Button button = ((Button)sender as Button);
        string btn = button.Name;
        string[] position = btn.Split('_');
        string url = "http://localhost:5000/table/"+position[1];

        HttpClient wc = new HttpClient();
        wc.DeleteAsync(url);
        getTables();
    }
}
    
```

4.3.8 Users - Admin

En el apartado de usuarios podemos observar el botón de añadir que abre un formulario en el cual debemos rellenar todos los campos para insertar un nuevo usuario.

DNI	Name	LastName	ROLE
58867953Y	Paco	Mermela	waiter
58667953Y	Mario	Sanchezos	admin
12345698B	Jose Luis	Garcioss	waiter
78954585Q	Agentes	Smithes	admin
78451236A	Elver	Gomez Torva	waiter
00000000A	Agente	007	admin

Add User

Name _____

LastName _____

DNI _____

Password _____ Role _____

Add User

Edit User

Agente _____

007 _____

00000000A _____

..... admin _____

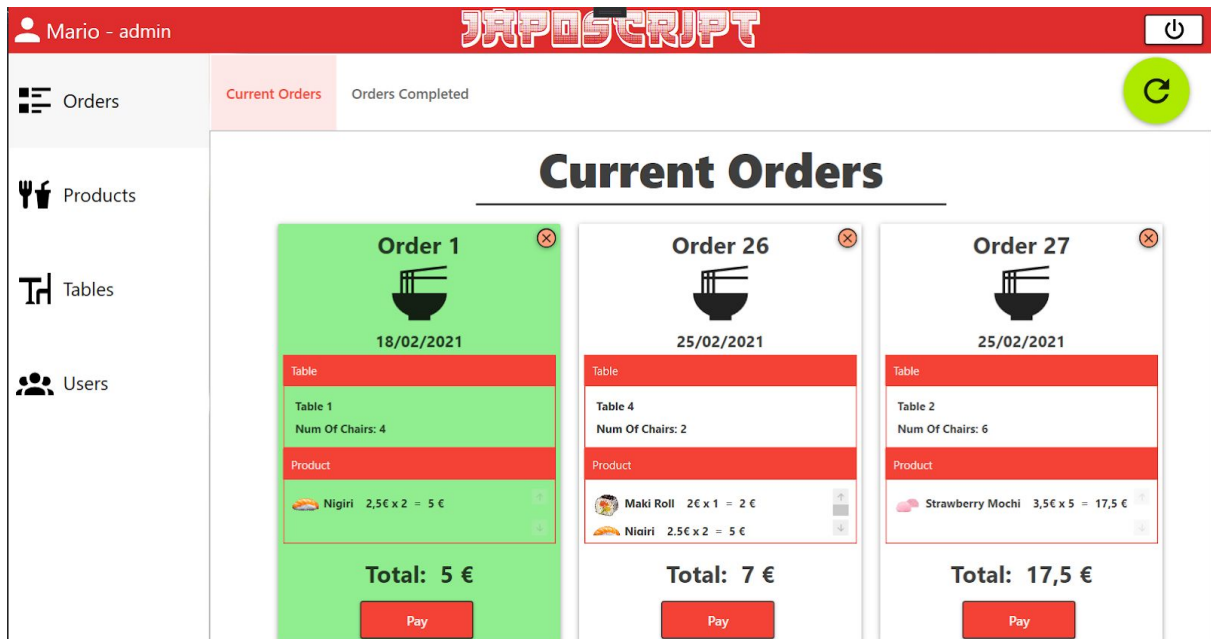
Edit User **Delete User**

Hacemos uso del mismo formulario para no hacer uso innecesario de vistas.

4.3.9 Orders - Admin

Como administrador también tenemos la opción de entrar en la sección de las órdenes para evitar cualquier percance, el administrador puede pagar si así la desea cualquier orden siempre y cuando esté lista para pagar.

Las **card** tiene una (x) para cancelar la orden en caso de cualquier inconveniente.



Método para pagar la orden, simplemente obtenemos la posición de esa mesa, que es un objeto tipo `table`, que después convertimos a json para la ejecución de un path hacia la API que determinara por medio de la variable `pay`, que la orden ha sido paga,

```
private void ButtonPay_Click(object sender, RoutedEventArgs e)
{
    string btnClicked = MyMessageBox.ShowBox("Do you want to Pay?", "Pay", "Yes", "NO");
    if (btnClicked == "1")
    {
        Button button = ((Button)sender as Button);
        string btn = button.Name;
        string[] position = btn.Split('_');
        //UPDATE TABLE Taken
        Table table = ListOrders[Convert.ToInt32(position[1])].Table;
        table.Taken = false;
        HttpClient wc = new HttpClient();
        string json = JsonConvert.SerializeObject(table);
        int numTable = Convert.ToInt32(table.Name.Split(' ')[1]) - 1;
        string url = "http://localhost:5000/table/" + numTable;
        var content3 = new StringContent(json, Encoding.UTF8, "application/json");
        _ = PatchAsync.patchAsync(wc, url, content3, new CancellationToken());

        HttpClient wc2 = new HttpClient();

        Order order = ListOrders[Convert.ToInt32(position[1])];
        order.Paid = true;

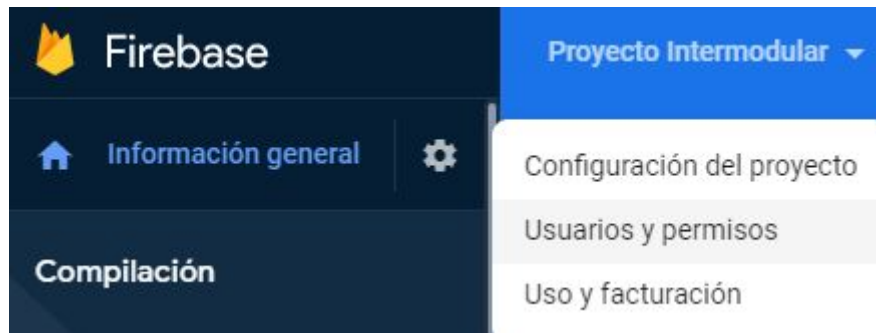
        string url2 = "http://localhost:5000/orders/" + position[1];
        string json2 = JsonConvert.SerializeObject(order);

        var content = new StringContent(json2, Encoding.UTF8, "application/json");
        _ = PatchAsync.patchAsync(wc2, url2, content, new CancellationToken());
        GetOrders();
    }
}
```

5. Configuración para usar el proyecto

5.1 Crear proyecto FireBase

Lo primero que necesitaremos será una cuenta en firebase y darle a crear nuevo proyecto, cuando nos vaya preguntando lo rellenaremos con nuestros datos y cuando nos pregunte si queremos activar el modo de prueba le daremos a que no, en cuanto a las opciones de google Analytics y etc, es a vuestra elección.



5.2 Descargar key de administrador

Nos dirigiremos a la configuración y seleccionaremos Usuarios y permisos

En el apartado de Cuentas de servicio tendremos lo siguiente:

Fragmento de configuración del SDK de administración

☒ Node.js ☐ Java ☐ Python ☐ Go

```
var admin = require("firebase-admin");

var serviceAccount = require("path/to/serviceAccountKey.json");

admin.initializeApp({
  credential: admin.credential.cert(serviceAccount),
  databaseURL: "https://proyecto-intermodular-default-rtdb.europe-we
});
```

Generar nueva clave privada

De aquí nos interesa copiar el `admin.initializeApp` y sustituirlo por el que hay en el `index` principal y por otra parte clicar en “Generar nueva clave privada” y guardar ese archivo en la carpeta `/data/key`

5.2 Instalar dependencias e iniciar servidor

Tenemos que tener Node.js instalado en el equipo

Tras sustituir la `key` por la de vuestro proyecto y la ruta del `initializeApp` por la vuestra ya hemos finalizado con Firebase y ahora nos toca instalar las dependencias e iniciar Node.js

Abriremos un terminal en la ruta de nuestro proyecto de Node.js y escribiremos “`npm install`” tras la descarga de las dependencias nos quedará escribir “`npm start`” para iniciar el servidor.

5.2 Configurar Android para conexión

Lo último que nos falta es hacer que android apunte a nuestra dirección IP, para ello vamos a abrir el proyecto Android y vamos a abrir la clase `ApiClient` donde tendremos que poner nuestra IP (Si no sabéis vuestra IP podéis ver la siguiente página:

<https://computerhoy.com/paso-a-paso/internet/como-saber-cual-es-direccion-ip-mi-or-denador-24347>)

Bibliografía:

Dependencias Android:

- <https://square.github.io/picasso/>
- <https://loopj.com/android-async-http/>
- <https://github.com/google/gson>

Mockups Android:

- https://www.figma.com/file/UdCTHvYHxEH0aVgR1TFBhq/Android_Mockup

Dependencias WPF:

- <http://materialdesigninxaml.net/>

Mockups WPF:

cliente

- <https://www.figma.com/file/O7E2ku4gpV8ih3eL7hOWHM/?node-id=109%3A1>

Administrador

- <https://www.figma.com/file/O7E2ku4gpV8ih3eL7hOWHM/?node-id=110%3A28>