Manual Técnico para Programadores - Proyecto GameStore

# 1. Introducción

Este manual proporciona una descripción detallada del funcionamiento interno del proyecto GameStore. Está diseñado para ayudar a programadores a comprender la estructura, lógica y conexiones entre los distintos componentes del sistema backend y frontend.

# 2. Estructura del Proyecto

La siguiente es la estructura general del proyecto GameStore extraída del paquete proporcionado:

* gamestore/docker-compose.yml
* gamestore/backend/.env
* gamestore/backend/Dockerfile
* gamestore/backend/package.json
* gamestore/backend/src/index.js
* gamestore/backend/src/models/Order.js
* gamestore/backend/src/models/Product.js
* gamestore/backend/src/routes/orders.js
* gamestore/backend/src/routes/products.js
* gamestore/backend/src/routes/upload.js
* gamestore/frontend/.flutter-plugins
* gamestore/frontend/.flutter-plugins-dependencies
* gamestore/frontend/.gitignore
* gamestore/frontend/.metadata
* gamestore/frontend/analysis\_options.yaml
* gamestore/frontend/Dockerfile
* gamestore/frontend/gamestore.iml
* gamestore/frontend/pubspec.lock
* gamestore/frontend/pubspec.yaml
* gamestore/frontend/.dart\_tool/package\_config.json
* gamestore/frontend/.dart\_tool/package\_config\_subset
* gamestore/frontend/.dart\_tool/version
* gamestore/frontend/.dart\_tool/dartpad/web\_plugin\_registrant.dart
* gamestore/frontend/.dart\_tool/extension\_discovery/vs\_code.json
* gamestore/frontend/.idea/modules.xml
* gamestore/frontend/.idea/workspace.xml
* gamestore/frontend/.idea/libraries/Dart\_SDK.xml
* gamestore/frontend/.idea/libraries/KotlinJavaRuntime.xml
* gamestore/frontend/.idea/runConfigurations/main\_dart.xml
* gamestore/frontend/assets/images/cover.png
* gamestore/frontend/lib/main.dart
* gamestore/frontend/lib/config/env.dart
* gamestore/frontend/lib/models/cart\_item.dart
* gamestore/frontend/lib/models/product.dart
* gamestore/frontend/lib/services/api\_service.dart
* gamestore/frontend/lib/services/auth\_service.dart
* gamestore/frontend/lib/services/cart\_service.dart
* gamestore/frontend/lib/services/order\_service.dart
* gamestore/frontend/lib/views/add\_product\_view.dart
* gamestore/frontend/lib/views/admin\_dashboard\_view.dart
* gamestore/frontend/lib/views/admin\_login\_view.dart
* gamestore/frontend/lib/views/admin\_orders\_view.dart
* gamestore/frontend/lib/views/cart\_view.dart
* gamestore/frontend/lib/views/checkout\_view.dart
* gamestore/frontend/lib/views/edit\_product\_view.dart
* gamestore/frontend/lib/views/home\_view.dart
* gamestore/frontend/lib/views/manage\_products\_view.dart
* gamestore/frontend/lib/views/order\_confirmation\_view.dart
* gamestore/frontend/lib/views/product\_detail\_view.dart
* gamestore/frontend/lib/views/product\_list\_view.dart
* gamestore/frontend/lib/widgets/custom\_app\_bar.dart
* gamestore/frontend/lib/widgets/payment\_method\_form.dart
* gamestore/frontend/lib/widgets/summary\_dialog.dart
* gamestore/frontend/linux/.gitignore
* gamestore/frontend/linux/CMakeLists.txt
* gamestore/frontend/linux/main.cc
* gamestore/frontend/linux/my\_application.cc
* gamestore/frontend/linux/my\_application.h
* gamestore/frontend/linux/flutter/CMakeLists.txt
* gamestore/frontend/linux/flutter/generated\_plugins.cmake
* gamestore/frontend/linux/flutter/generated\_plugin\_registrant.cc
* gamestore/frontend/linux/flutter/generated\_plugin\_registrant.h
* gamestore/frontend/linux/flutter/ephemeral/.plugin\_symlinks/file\_selector\_linux
* gamestore/frontend/linux/flutter/ephemeral/.plugin\_symlinks/image\_picker\_linux
* gamestore/frontend/linux/flutter/ephemeral/.plugin\_symlinks/path\_provider\_linux
* gamestore/frontend/macos/.gitignore
* gamestore/frontend/macos/Flutter/Flutter-Debug.xcconfig
* gamestore/frontend/macos/Flutter/Flutter-Release.xcconfig
* gamestore/frontend/macos/Flutter/GeneratedPluginRegistrant.swift
* gamestore/frontend/macos/Flutter/ephemeral/Flutter-Generated.xcconfig
* gamestore/frontend/macos/Flutter/ephemeral/flutter\_export\_environment.sh
* gamestore/frontend/macos/Runner/AppDelegate.swift
* gamestore/frontend/macos/Runner/DebugProfile.entitlements
* gamestore/frontend/macos/Runner/Info.plist
* gamestore/frontend/macos/Runner/MainFlutterWindow.swift
* gamestore/frontend/macos/Runner/Release.entitlements
* gamestore/frontend/macos/Runner/Assets.xcassets/AppIcon.appiconset/app\_icon\_1024.png
* gamestore/frontend/macos/Runner/Assets.xcassets/AppIcon.appiconset/app\_icon\_128.png
* gamestore/frontend/macos/Runner/Assets.xcassets/AppIcon.appiconset/app\_icon\_16.png
* gamestore/frontend/macos/Runner/Assets.xcassets/AppIcon.appiconset/app\_icon\_256.png
* gamestore/frontend/macos/Runner/Assets.xcassets/AppIcon.appiconset/app\_icon\_32.png
* gamestore/frontend/macos/Runner/Assets.xcassets/AppIcon.appiconset/app\_icon\_512.png
* gamestore/frontend/macos/Runner/Assets.xcassets/AppIcon.appiconset/app\_icon\_64.png
* gamestore/frontend/macos/Runner/Assets.xcassets/AppIcon.appiconset/Contents.json
* gamestore/frontend/macos/Runner/Base.lproj/MainMenu.xib
* gamestore/frontend/macos/Runner/Configs/AppInfo.xcconfig
* gamestore/frontend/macos/Runner/Configs/Debug.xcconfig
* gamestore/frontend/macos/Runner/Configs/Release.xcconfig
* gamestore/frontend/macos/Runner/Configs/Warnings.xcconfig
* gamestore/frontend/macos/Runner.xcodeproj/project.pbxproj
* gamestore/frontend/macos/Runner.xcodeproj/project.xcworkspace/xcshareddata/IDEWorkspaceChecks.plist
* gamestore/frontend/macos/Runner.xcodeproj/xcshareddata/xcschemes/Runner.xcscheme
* gamestore/frontend/macos/Runner.xcworkspace/contents.xcworkspacedata
* gamestore/frontend/macos/Runner.xcworkspace/xcshareddata/IDEWorkspaceChecks.plist
* gamestore/frontend/macos/RunnerTests/RunnerTests.swift
* gamestore/frontend/web/favicon.png
* gamestore/frontend/web/index.html
* gamestore/frontend/web/manifest.json
* gamestore/frontend/web/icons/Icon-192.png
* gamestore/frontend/web/icons/Icon-512.png
* gamestore/frontend/web/icons/Icon-maskable-192.png
* gamestore/frontend/web/icons/Icon-maskable-512.png
* gamestore/frontend/windows/.gitignore
* gamestore/frontend/windows/CMakeLists.txt
* gamestore/frontend/windows/flutter/CMakeLists.txt
* gamestore/frontend/windows/flutter/generated\_plugins.cmake
* gamestore/frontend/windows/flutter/generated\_plugin\_registrant.cc
* gamestore/frontend/windows/flutter/generated\_plugin\_registrant.h
* gamestore/frontend/windows/flutter/ephemeral/.plugin\_symlinks/file\_selector\_windows
* gamestore/frontend/windows/flutter/ephemeral/.plugin\_symlinks/image\_picker\_windows
* gamestore/frontend/windows/flutter/ephemeral/.plugin\_symlinks/path\_provider\_windows
* gamestore/frontend/windows/runner/CMakeLists.txt
* gamestore/frontend/windows/runner/flutter\_window.cpp
* gamestore/frontend/windows/runner/flutter\_window.h
* gamestore/frontend/windows/runner/main.cpp
* gamestore/frontend/windows/runner/resource.h
* gamestore/frontend/windows/runner/runner.exe.manifest
* gamestore/frontend/windows/runner/Runner.rc
* gamestore/frontend/windows/runner/utils.cpp
* gamestore/frontend/windows/runner/utils.h
* gamestore/frontend/windows/runner/win32\_window.cpp
* gamestore/frontend/windows/runner/win32\_window.h
* gamestore/frontend/windows/runner/resources/app\_icon.ico
* gamestore/uploads/1748308397672-1366\_2000.jpg
* gamestore/uploads/1748309194325-610.webp
* gamestore/uploads/1748312363338-1366\_2000.jpg
* gamestore/uploads/1748314526298-616klipzdtL.jpg
* gamestore/uploads/1748314711167-CONSOLA-SERIES-S-XBOX-RRS-00004-3002159\_b.webp
* gamestore/uploads/1748314835052-descargar.jfif
* gamestore/uploads/1748314978026-control-dualsense-inalambrico-ps5-negro-711719574309.webp
* gamestore/uploads/1748315079873-711719540892.webp
* gamestore/uploads/1748315241613-81oYUkXWukL.\_AC\_UF1000,1000\_QL80\_.jpg
* gamestore/uploads/1748377165736.webp
* gamestore/uploads/1748377242970.webp
* gamestore/uploads/1748377363305.jfif
* gamestore/uploads/1748389953620-PS5-Slim\_-DS5.webp

# 3. Backend - Descripción de Archivos y Funciones

## 3.1 index.js

Este archivo es el punto de entrada principal del backend. Utiliza el framework Express para crear un servidor HTTP y conecta la aplicación con MongoDB usando Mongoose.

* Componentes clave:
* - express: Framework web para crear el servidor.
* - mongoose: Cliente de MongoDB para Node.js.
* - cors: Middleware que permite peticiones desde otros orígenes (útil para comunicación con el frontend).
* - path: Módulo para manejar rutas de archivos.
* Flujo general del archivo:
* Se importan las rutas desde los módulos de productos, órdenes y subida de archivos.
* Se configuran middlewares globales (`cors` y `express.json()`).
* Se habilita el acceso estático a imágenes en la carpeta `uploads`.
* Se asocian rutas: `/api/products`, `/api/orders` y `/api/upload` a sus respectivos manejadores.
* Se conecta a MongoDB (contenedor 'mongo').
* Se inicia el servidor en el puerto 3000.
* Dependencias principales utilizadas:
* Express
* Mongoose
* CORS

## 3.2 Archivo: src/models/Product.js

Este archivo define el esquema del modelo `Product` para MongoDB utilizando Mongoose. Representa un producto en la tienda GameStore.

* Campos definidos en el esquema:
* `name` (String, requerido): Nombre del producto.
* `description` (String): Descripción del producto.
* `price` (Number, requerido): Precio del producto.
* `imageUrl` (String): URL de la imagen del producto.
* `quantity` (Number): Cantidad disponible en stock.

Este modelo se utiliza principalmente en las rutas de productos (`routes/products.js`) para registrar, actualizar, listar y eliminar productos en la base de datos.

## 3.3 Archivo: src/models/Order.js

Este archivo define el esquema del modelo `Order` para Mongoose. Representa una orden de compra realizada por un usuario.

* Campos definidos en el esquema:
* `items` (Array): Lista de productos comprados. Cada item incluye `product` (ID del producto) y `quantity`.
* `total` (Number): Valor total de la compra.
* `address` (String): Dirección de envío proporcionada por el usuario.
* `createdAt` (Date): Fecha de creación automática.

Este modelo es usado por la ruta de órdenes (`routes/orders.js`) para registrar nuevas compras y consultar el historial de pedidos.

## 3.4 Archivo: src/routes/products.js

Este archivo define las rutas del API relacionadas con los productos. Utiliza el modelo `Product` para interactuar con la base de datos.

* Rutas disponibles:
* `GET /api/products`: Lista todos los productos disponibles.
* `GET /api/products/:id`: Obtiene un producto específico por su ID.
* `POST /api/products`: Crea un nuevo producto (requiere datos en el cuerpo del request).
* `PUT /api/products/:id`: Actualiza los datos de un producto existente.
* `DELETE /api/products/:id`: Elimina un producto por su ID.

Cada una de estas rutas utiliza funciones de Mongoose como `find()`, `findById()`, `save()`, `findByIdAndUpdate()` y `findByIdAndDelete()` sobre el modelo `Product`.

## 3.5 Archivo: src/routes/orders.js

Este archivo define las rutas relacionadas con las órdenes de compra. Utiliza el modelo `Order`.

* Rutas disponibles:
* `GET /api/orders`: Lista todas las órdenes registradas.
* `POST /api/orders`: Crea una nueva orden con productos, total y dirección de envío.

Las operaciones utilizan `Order.find()` para listar y `new Order({...}).save()` para registrar nuevas órdenes.

## 3.6 Archivo: src/routes/upload.js

Este archivo gestiona la subida de archivos, como imágenes de productos, mediante `multer`, un middleware para manejo de `multipart/form-data`.

* Ruta disponible:
* `POST /api/upload`: Permite subir un archivo que se almacena en la carpeta `/uploads` del servidor.

El archivo se guarda en el servidor y se retorna su nombre para luego ser referenciado en productos.

## 3.7 Archivo: .env

Este archivo contiene variables de entorno usadas para configurar parámetros sensibles o cambiantes del sistema.

* Ejemplos típicos que podrían encontrarse aquí incluyen:
* URIs de conexión a base de datos
* Claves secretas de APIs

Este archivo debe incluirse en `.gitignore` para evitar exponer datos sensibles.

## 3.8 Archivo: Dockerfile

Define la imagen del contenedor del backend. Contiene instrucciones para construir el entorno Node.js dentro de Docker.

* Instrucciones destacadas:
* `FROM node:18`: Usa una imagen base oficial de Node.js versión 18.
* `WORKDIR /app`: Define el directorio de trabajo en el contenedor.
* `COPY . .`: Copia todos los archivos del backend al contenedor.
* `RUN npm install`: Instala las dependencias declaradas en `package.json`.
* `CMD ["node", "src/index.js"]`: Ejecuta el backend cuando el contenedor inicia.

## 3.9 Archivo: package.json

Este archivo define las dependencias del backend y scripts de ejecución.

* Dependencias clave:
* `express`: Framework web usado para construir la API.
* `mongoose`: Cliente para conectar y manipular bases de datos MongoDB.
* `cors`: Middleware que permite solicitudes entre dominios distintos.
* `multer`: Middleware para la gestión de formularios con archivos.

También incluye metadatos del proyecto como nombre, versión y autor, así como scripts como `start` para ejecutar la aplicación.

# 4. Análisis del Frontend (Flutter)

## 4.1 Archivo: lib/main.dart

Este archivo es el punto de entrada de la aplicación Flutter. Aquí se configura el entorno de ejecución, se inicializan servicios y se establece la estructura básica de navegación.

* Componentes principales identificados:
* `runApp(MyApp())`: Inicia la aplicación con el widget raíz `MyApp`.
* `MaterialApp`: Widget que configura rutas, tema global, y la pantalla inicial.
* `routes`: Define la navegación entre pantallas mediante rutas nombradas. Incluye vistas como `/`, `/cart`, `/checkout`, `/admin`.
* `home`: Pantalla inicial definida como `HomeView()` (vista del catálogo de productos).

El archivo establece la estructura general de navegación y enlaza todas las vistas de la app a través de rutas explícitas.

## 4.2 Widgets Reutilizables

### 4.2.1 custom\_app\_bar.dart

Este widget crea una barra de navegación personalizada utilizada en varias vistas.

* Características principales:
* Integra título centralizado y botones de acción.
* Permite personalizar el título de la vista mediante un parámetro.
* Facilita la reutilización del mismo diseño en distintas vistas (por ejemplo: `HomeView`, `CartView`).

### 4.2.2 summary\_dialog.dart

Este widget genera un diálogo emergente tipo resumen que se usa al confirmar órdenes.

* Características principales:
* Muestra un resumen de productos, dirección y método de pago.
* Se usa desde `checkout\_view.dart` antes de crear la orden.
* Reutilizable para futuras confirmaciones o revisiones de carrito.

### 4.2.3 payment\_method\_form.dart

Este widget muestra un formulario para capturar datos de pago.

* Características principales:
* Campos como número de tarjeta, fecha de expiración y CVV.
* Valida datos antes de proceder al resumen de la orden.
* Integrado en `checkout\_view.dart`.

## 4.3 Vistas de la Aplicación

## 4.3 Vistas de la Aplicación

### 4.3.1 AdminDashboardView 🔒 (Vista exclusiva del administrador)

Pantalla principal del administrador luego de iniciar sesión. Resume métricas generales como el número de productos, órdenes y permite acceso rápido a secciones como gestión de productos y revisión de órdenes. Integra navegación condicional según permisos. Utiliza servicios como orderService y apiService para cargar datos.

### 4.3.2 AdminLoginView 🔒

Vista de inicio de sesión exclusiva para el administrador. Muestra un formulario de autenticación simple (mock), validado localmente por authService. Controla acceso a vistas como AdminDashboardView, ManageProductsView y AdminOrdersView.

### 4.3.3 AdminOrdersView 🔒

Vista donde el administrador puede visualizar todas las órdenes realizadas por los usuarios. Muestra detalles como productos comprados, dirección de envío y total. Los datos son obtenidos desde el backend mediante orderService.fetchOrders(). Usa tablas o listas con scroll adaptativo.

### 4.3.4 AddProductView 🔒

Formulario exclusivo para crear nuevos productos. Permite subir una imagen (/api/upload), ingresar nombre, precio, descripción y cantidad. Al guardar, se invoca apiService.post() para registrar el producto en MongoDB. Incluye validaciones básicas y retroalimentación visual.

### 4.3.5 EditProductView 🔒

Pantalla similar a AddProductView, pero precargada con los datos de un producto existente. Permite modificar atributos, cambiar la imagen y actualizar el producto con apiService.put(). Se accede desde la lista de productos con un botón de edición contextual.

### 4.3.6 ManageProductsView 🔒

Panel administrativo donde se listan todos los productos de la tienda. Desde aquí se puede editar o eliminar productos usando botones individuales. Integra llamadas a apiService.get() y delete() para manipulación remota del inventario. Usa CustomAppBar y tablas dinámicas.

### 4.3.7 CartView

Vista accesible por cualquier usuario, muestra los productos añadidos al carrito. Permite modificar cantidades o eliminar ítems. Se actualiza dinámicamente con cartService y muestra el total calculado. Usa el modelo CartItem y está enlazada con la vista CheckoutView.

### 4.3.8 CheckoutView

Formulario para completar la compra. Solicita dirección de envío y método de pago (simulado). Incluye el widget PaymentMethodForm. Al finalizar, muestra el resumen (SummaryDialog) y luego envía la orden mediante orderService.createOrder() al backend.

### 4.3.9 HomeView

Vista principal del usuario. Muestra todos los productos en un grid, cargados desde el backend con apiService.get(). Cada producto tiene acceso al detalle individual y permite añadir al carrito. Utiliza widgets como CustomAppBar.

### 4.3.10 OrderConfirmationView

Se muestra tras una compra exitosa. Resume los datos de la orden: productos, dirección y total. Puede integrarse con confirmaciones por correo en futuras mejoras. Proporciona botón para volver a inicio o seguir comprando.

### 4.3.11 ProductDetailView

Vista individual de un producto. Muestra su imagen, descripción, precio y cantidad disponible. Desde aquí el usuario puede añadir el producto al carrito. Utiliza animaciones para confirmar la acción y conecta con cartService.

### 4.3.12 ProductListView

Vista alternativa que muestra todos los productos disponibles en formato lista en lugar de grid. Puede ser utilizada en dispositivos con pantallas pequeñas o para navegaciones más accesibles. También permite acceder al detalle de cada producto.

## 4.4 Servicios

### 4.4.1 api\_service.dart

Este servicio gestiona la comunicación general con el backend utilizando HTTP.

* Funciones clave:
* `get()`, `post()`, `put()`, `delete()`: Envían solicitudes HTTP genéricas usando `http` package.
* Utilizado internamente por otros servicios como `order\_service.dart` y `cart\_service.dart` para consumir la API REST del backend.

### 4.4.2 auth\_service.dart

Este archivo gestiona la autenticación simple para vistas del administrador.

* Características:
* Simula validación de usuario administrador con credenciales fijas o mock.
* Controla el acceso a vistas como `admin\_dashboard\_view.dart` y `admin\_orders\_view.dart`.

### 4.4.3 cart\_service.dart

Este servicio mantiene y modifica el estado del carrito de compras.

* Funciones clave:
* `addProduct()`, `removeProduct()`: Añade o elimina productos del carrito.
* `items`, `total`: Proveen acceso al estado actual del carrito.
* Notifica a los widgets para actualizar su estado usando `ChangeNotifier`.

### 4.4.4 order\_service.dart

Este servicio se encarga de registrar órdenes en el backend.

* Funciones clave:
* `createOrder()`: Envía los productos del carrito, total y dirección al servidor.
* `fetchOrders()`: Recupera las órdenes existentes para visualizarlas desde la vista `admin\_orders\_view.dart`.

## 4.5 Modelos

### 4.5.1 product.dart

Este modelo representa un producto en la tienda dentro del frontend Flutter.

* Atributos principales:
* `id` (String): Identificador único del producto.
* `name`, `description` (String): Nombre y descripción del producto.
* `price` (double): Precio del producto.
* `imageUrl` (String): Ruta de la imagen del producto.
* `quantity` (int): Cantidad disponible en inventario.

Incluye métodos de serialización como `fromJson()` y `toJson()` para intercambiar datos con el backend.

### 4.5.2 cart\_item.dart

Este modelo representa un ítem en el carrito de compras.

* Atributos:
* `product` (Product): Objeto producto asociado.
* `quantity` (int): Cantidad del producto seleccionado por el usuario.

Utilizado principalmente por `cart\_service.dart` y la vista `cart\_view.dart` para manejar y mostrar productos en el carrito.

# 5. Posibles Mejoras

A continuación se enumeran mejoras que podrían implementarse para escalar o robustecer el sistema:

* Agregar autenticación real con JWT o OAuth2 para separar roles de usuario y administrador.
* Integrar almacenamiento en la nube (por ejemplo, Cloudinary o AWS S3) para las imágenes en lugar del servidor local.
* Aplicar validaciones más estrictas en backend para los formularios y peticiones.
* Implementar paginación y filtrado en las listas de productos y órdenes.
* Agregar pruebas unitarias y de integración para asegurar la estabilidad del sistema.
* Optimizar la experiencia móvil en Flutter con diseños adaptativos y feedback visual adicional.
* Incluir historial de órdenes por usuario y posibilidad de reenviar confirmación de pedido.
* Conectar la app con una pasarela de pagos real (Stripe, PayU, etc.) en lugar de simular pagos.