Da Vinci
PRIMERA ESCUELA DE ARTE MULTIMEDIAL

Plataformas de desarrollo

Proyecto E-commerce Synchro

Alumno: Ignacio Joaquin Rolon Marecos
Comisión: ACN4BV
Turno: Noche
Cuatrimestre: 4º

Informe del Proyecto Synchro – E-Commerce de Relojes

1. Introducción

Synchro es una aplicación web de e-commerce orientada a la venta de relojes de distintos tipos: clásicos, deportivos e inteligentes.
Se compone de:

- Un FrontEnd desarrollado en React, que ofrece la interfaz gráfica para que el usuario final navegue, seleccione productos y realice compras.
- Un BackEnd implementado con Node.js y Express, que expone una API REST para gestionar productos y órdenes.
- Una base de datos en Firebase Firestore, utilizada para persistir productos y órdenes de compra.

El objetivo del sistema es simular un flujo real de compra online, integrando consulta de catálogo, carrito, confirmación de compra y almacenamiento de la orden.

---

2. Descripción general de la aplicación

La aplicación permite al usuario:

- Navegar por una **página de inicio** con productos destacados.
- Explorar el **catálogo por categorías** (clásicos, deportivos, inteligentes).
- Acceder al **detalle de cada producto**, donde puede ajustar la cantidad y agregarlo al carrito.
- Visualizar y modificar el **carrito de compras**, eliminando ítems o vaciándolo por completo.

- Completar un **formulario de compra** con datos personales (nombre, apellido, dirección, email y confirmación de email).

- Enviar la orden al servidor y recibir un **ID de orden** como comprobante.

La lógica de presentación y la experiencia de usuario se gestionan en el FrontEnd, mientras que la lógica de negocio y el acceso a datos se concentran en el BackEnd.

---

3. Tecnologías utilizadas

3.1 FrontEnd

- **React**

- **React Router** para el manejo de rutas internas.

- **Bootstrap 5** y **CSS personalizado** para la maquetación y estilos.

- **LocalStorage** del navegador para persistir el carrito de compras incluso si se recarga la página.

3.2 BackEnd

- **Node.js** como entorno de ejecución.

- **Express** como framework para construir la API REST.

- **CORS** para permitir la comunicación entre el front (puerto 5173) y el back (puerto 3000).

- **Firebase Admin SDK** para conectarse a **Firestore**.

- **Middlewares personalizados** para validación de datos y registro de peticiones.

---

4. Estructura de carpetas

4.1 BackEnd

```
BackEnd/
├── server.js
├── package.json
└── src/
    ├── App.js
    ├── config/
    │   └── firebase.js
    ├── controller/
    │   ├── producto_controller.js
    │   └── order_controller.js
    ├── middlewares/
    │   └── validarOrden.js
    ├── routes/
    │   ├── productos.js
    │   └── order.js
    └── services/
        ├── producto_service.js
        └── order_service.js
```

Descripción breve de archivos clave:

- **server.js**
  Carga las variables de entorno, importa `App.js` y levanta el servidor en el puerto definido (`process.env.PORT || 3000`).

- **src/App.js**
    - Importa `express`, `cors`, `path` y las rutas de productos y órdenes.
    - Define un middleware `requestLogger` que registra fecha, método HTTP y URL de cada petición.
    - Aplica los middlewares:
        - `cors()` → habilita CORS.
        - `express.json()` → parseo de JSON del body.
        - `requestLogger` → logging.
        - `express.static(ROOT_FOLDER)` → servir archivos estáticos si fuese necesario.

    - Monta las rutas:
        - `app.use(productosRoutes);`
        - `app.use(orderRoutes);`

- **src/config/firebase.js**
  Carga `firebase-admin`, inicializa la app con una **cuenta de servicio** (ruta obtenida de la variable `SERVICE_ACCOUNT_PATH`) y expone `db = admin.firestore()`.

- **src/services/producto_service.js**
  Define `getProductos()` que:
    - Accede a `db.collection('productos')`.
    - Obtiene el snapshot y arma un arreglo de productos con `id` y el resto de los campos (`doc.data()`).

- **src/services/order_service.js**
  Define `crearOrderFirebase(data)` que:
    - Construye un objeto `order` con `comprador`, `items`, `total` y `fecha`.
    - Lo guarda en `db.collection('orders')`.
    - Retorna el `id` del documento creado.

- **src/middlewares/validarorden.js**
  Extrae `{ comprador, items, total }` de `req.body` y valida:
    - Que los tres existan.
    - Que `items` sea un array y no esté vacío.
    - Que `total` sea numérico y mayor a cero.
      En caso de error devuelve `400` con un mensaje descriptivo.

- **src/routes/productos.js**
  Rutas:
    - `GET /api/productos` → obtiene todos los productos.
    - `GET /api/productos/:id` → producto individual según el ID.

- **src/routes/order.js**
  Ruta:
    - `POST /api/orders` → aplica el middleware de validación y luego llama al controlador para crear una orden.

- **src/controller/producto_controller.js**
  Implementa:
    - `getAllProductos` → usa `getProductos()` del servicio y responde con un array JSON.
    - `getProductById` → filtra el array por `id` (el que llega como parámetro de ruta) y responde con el producto o error 404.

- **src/controller/order_controller.js**
  Implementa:
    - `createOrder` → lee `req.body`, llama a `crearOrderFirebase`, y si todo sale bien responde con:
        - `status 201`
        - `{ success: true, orderId, message: 'Orden creada exitosamente' }`.

4.2 FrontEnd

```
FrontEnd/
├── public/
```

```
├── src/
│   ├── componentes/
│   ├── pages/
│   └── ...
├── index.html
├── package.json
└── vite.config.js
```

El FrontEnd se comunica con el BackEnd mediante llamadas `fetch` o `axios` a la API (`http://localhost:3000/api/...`), y maneja el estado del carrito

---

5. Flujo general de la aplicación

A continuación se describe el flujo de uso desde que el usuario ingresa a la aplicación hasta que completa una compra.

1. **Ingreso a la aplicación**
   El usuario accede a la URL del FrontEnd (por defecto `http://localhost:5173`). React renderiza la página de inicio, mostrando productos destacados y secciones por tipo de reloj.

2. **Navegación por el catálogo**

   - El usuario puede hacer clic en una categoría (clásicos, deportivos, inteligentes).

   - React navega a la ruta correspondiente (por ejemplo `/categoria/clasicos`) usando React Router.

   - En el montaje del componente, el FrontEnd realiza una petición `GET /api/productos` al BackEnd para obtener todos los productos y filtra por categoría en el cliente, o bien podría pedir solo los de esa categoría (según cómo se haya implementado).

3. **Detalle del producto**

   - Al seleccionar un producto, el usuario es dirigido a una vista de detalle.

- El FrontEnd puede solicitar datos específicos con `GET /api/productos/:id` o reutilizar la lista que ya tiene en memoria.

- En esta pantalla el usuario selecciona la cantidad y presiona **"Agregar al carrito"**.

4. **Gestión del carrito**

   - Los productos seleccionados se almacenan en el estado de React y en `LocalStorage`, para que el carrito no se pierda al recargar la página.

   - El usuario puede ver el carrito, modificar cantidades, eliminar productos o vaciarlo.

5. **Formulario de compra**

   - Cuando el usuario decide finalizar la compra, se lo redirige a una pantalla de checkout con un formulario que solicita: nombre, apellido, dirección, email y confirmación de email.

   - El FrontEnd realiza las validaciones básicas: campos obligatorios y coincidencia de los dos correos electrónicos.

6. **Envío de la orden al BackEnd**

   - Si las validaciones del formulario son correctas, el FrontEnd construye un objeto JSON con los datos del comprador, los items del carrito y el total, y lo envía mediante `POST /api/orders` al BackEnd.

7. **Validación y almacenamiento en el BackEnd**

   - Express recibe la petición en la ruta `/api/orders`.

   - Se ejecuta el middleware `validarorden.js` que valida estructura y tipos del JSON.

   - Si hay errores, responde con un `400 Bad Request` y un mensaje descriptivo.

   - Si la validación es correcta, el controlador `order_controller.js` llama al servicio `order_service.js`, que guarda la orden en Firestore y devuelve el `orderId`.

8. **Respuesta al FrontEnd y confirmación al usuario**

El BackEnd responde con un JSON del tipo:

```
{
  "success": true,
  "orderId": "ORD-8921",
  "message": "Orden creada exitosamente"
}
```

-
- El FrontEnd muestra una pantalla de **confirmación de compra** con el ID de la orden y un mensaje de agradecimiento, permitiendo volver al inicio o seguir navegando.

---

6. Ejemplos de intercambio de datos (JSON request/response)

En este apartado se muestran ejemplos concretos de las estructuras JSON que se envían y reciben entre el FrontEnd y el BackEnd.

6.1 Obtener todos los productos

Request

- Método: `GET`
- URL: `/api/productos`

No requiere body.

Response (200 OK)

```
[
  {
    "id": "9cJcMpI34jvoG6P75XtW",
    "nombre": "Marca C",
    "descripcion": "Reloj clasico moderno con correa de cuero...",
    "precio": 330,
    "categoria": "reloj clasico",
    "stock": 5,
    "imagen": "https://i.ibb.co/3yyWLngT/Img-reloj-3.jpg"
  },
```

```
  {
    "id": "GEn3fblvl8P371XuoHTd",
    "nombre": "Marca B",
    "descripcion": "Reloj clasico elegante con acabado dorado...",
    "precio": 200,
    "categoria": "reloj clasico",
    "stock": 5,
    "imagen": "https://i.ibb.co/1G7SQg2Y/Img-reloj-2.jpg"
  }
]
```

6.2 Obtener un producto por ID

Request

- Método: `GET`
- URL: `/api/productos/abc123`

Response (200 OK)


```
{
  "id": "JOfWHmRwy738vfh3FgGR",
  "nombre": "Marca H",
  "descripcion": "eloj inteligente con correa de silicona...",
  "precio": 210,
  "categoria": "Relojes Inteligentes",
  "stock": 6,
  "imagen": "https://i.ibb.co/JfGtK4g/Img-reloj-8.jpg"
}
```


Response de error (404 Not Found) – ejemplo posible:


```
{
  "error": "Producto no encontrado"
```

}

6.3 Crear una orden

Request

- Método: POST
- URL: `/api/orders`
- Cabeceras: `Content-Type: application/json`

Body

```
{
  "comprador": {
    "nombre": "Ignacio",
    "apellido": "Rolon ",
    "email": "ignacio@gmail.com",
    "direccion": "olleros 3000"
  },
  "items": [
    {
      "id": "JOfWHmRwy738vfh3FgGR",
      "nombre": "Marca C",
      "cantidad": 2,
      "precio": 330
    },
    {
      "id": "GEn3fblvl8P371XuoHTd",
      "nombre": "Marca B",
      "cantidad": 1,
      "precio": 200
    }
  ],
  "total": 860
}
```

Response (201 Created)

```
{
  "success": true,
  "orderId": "ORD-8921",
  "message": "Orden creada exitosamente"
}
```

Response de error de validación (400 Bad Request)

```
{
  "error": "Items debe ser un array no vacío"
}
```

---

7. Consideraciones de seguridad y buenas prácticas

- **Manejo de credenciales**
  La clave de servicio de Firebase no se incluye en el repositorio. La ruta a ese archivo se obtiene mediante la variable de entorno `SERVICE_ACCOUNT_PATH` desde el archivo `.env`.

- **Validación en servidor**
  Aunque el FrontEnd valida el formulario, la validación real se realiza en el BackEnd mediante el middleware `validarorden.js` para evitar que datos maliciosos lleguen a la base de datos.

- **CORS**
  El uso de `cors()` en Express permite el acceso controlado desde el origen del FrontEnd (localhost en desarrollo).

- **Logging**
  El middleware `requestLogger` registra cada petición con timestamp, método y URL, lo que facilita el debug y monitoreo básico.