UADE - BDD2

Trabajo práctico obligatorio - Grupo 1

INTEGRANTES:

- Thiago Klees
- Valentín Julian Campestri
- Mariano Alvarez

PROFESOR:

- Joaquín Salas

UADE

MODELO BASE DE DATOS - Ecommerce

MONGO DB:
Utilizaremos una base de datos documental para almacenar los datos de **Usuarios, Pedidos, Facturas y Productos**, detallando la estructura propuesta para cada entidad y la optimización de las consultas.

MongoDB es una base de datos **documental NoSQL**, lo que la hace ideal para sistemas de comercio electrónico debido a las siguientes razones:

Flexibilidad del esquema: Los datos de usuarios, pedidos, facturas y productos pueden variar según el tipo de cliente o las necesidades del negocio, sin necesidad de migraciones complejas.
Escalabilidad horizontal: MongoDB permite **sharding**, lo que facilita el manejo de grandes volúmenes de pedidos y productos sin afectar el rendimiento.
Optimización para lecturas y escrituras rápidas: En un sistema de comercio electrónico, las consultas deben ser eficientes para brindar una experiencia fluida a los usuarios.
Soporte para datos anidados: Se pueden almacenar detalles de pedidos y facturas dentro de un solo documento, lo que reduce la cantidad de consultas necesarias.
Alta disponibilidad: La replicación automática de MongoDB garantiza que los datos estén disponibles incluso en caso de fallos del sistema.

Usuarios

```
{
  "_id": "645f2a9b7d3e4a0015c2e8a1",
  "nombre": "Carlos Pérez",
  "email": "carlos.perez@email.com",
  "telefono": "+54 11 1234 5678",
  "direccion": {
    "calle": "Av. Libertador 1234",
    "ciudad": "Buenos Aires",
    "pais": "Argentina"
  },
  "fecha_registro": "2024-02-09T12:00:00Z",
“es_admin”: true
  "categoria": "TOP",
  "historial_pedidos": ["pedidoID1", "pedidoID2"],
}
```

Dirección embebida, evitando la necesidad de JOINs.
Historial de compras embebido, permitiendo acceder rápidamente a los pedidos recientes.
Un ejemplo de consulta puede ser: db.usuarios.findOne({ "email": "carlos.perez@email.com" })

Con este podríamos filtrar rapidamente según el mail del usuario
Permite agregar nuevos atributos sin alterar el esquema general.

Se pueden almacenar los pedidos recientes dentro del usuario para optimizar consultas.

Pedidos
Cada pedido está compuesto por múltiples productos y requiere almacenar detalles como el estado, los descuentos aplicados y la dirección de envío.


```
{
  "_id": "12345",
  "usuario_id": "645f2a9b7d3e4a0015c2e8a1",
  "fecha": "2024-01-15T10:30:00Z",
  "estado": "Entregado",
  "productos": [
    {
      "producto_id": "001",
      "nombre": "Notebook Lenovo",
      "cantidad": 1,
      "precio_unitario": 1200.00
    },
    {
      "producto_id": "002",
      "nombre": "Mouse Logitech",
      "cantidad": 2,
      "precio_unitario": 50.00
    }
  ],
  "total": 1300.00,
  "direccion_envio": {
    "calle": "Av. Libertador 1234",
    "ciudad": "Buenos Aires",
    "pais": "Argentina"
  }
}
```


Se almacena el usuario que realizó el pedido.
Lista de productos embebidos para evitar consultas adicionales.
Dirección de envío almacenada en el pedido, ya que puede diferir de la dirección principal del usuario.

Ejemplo de consulta: Obtener los pedidos de un usuario en los últimos 30 días:
db.pedidos.find({ "usuario_id": "645f2a9b7d3e4a0015c2e8a1", "fecha": { $gte: ISODate("2024-01-01T00:00:00Z") } })

- Permite acceder a los pedidos rápidamente sin necesidad de hacer múltiples JOINs.

- Se puede escalar horizontalmente mediante **sharding** en la clave `usuario_id`.

FACTURAS
Cada factura debe estar vinculada a un pedido y registrar la forma de pago.

Estructura propuesta

```
{
  "_id": "F12345",
  "pedido_id": "12345",
  "usuario_id": "645f2a9b7d3e4a0015c2e8a1",
  "fecha": "2024-01-15T11:00:00Z",
  "total": 1300.00,
  "metodo_pago": "Tarjeta de Crédito",
  "estado": "Pagado"
}
```

- Almacena facturas como documentos individuales sin necesidad de normalización excesiva.
- Se pueden optimizar consultas creando un **índice en `usuario_id` y `pedido_id`**.

Producto

Los productos tienen múltiples variantes y precios que pueden cambiar con el tiempo.

Estructura

```
{
  "_id": "001",
  "nombre": "Notebook Lenovo",
  "descripcion": "Laptop con procesador Intel i7, 16GB RAM, 512GB SSD",
  "categorias": ["Tecnología", "Computadoras"],
  "precio_actual": 1200.00,
  "historial_precios": [
    { "fecha": "2023-12-01", "precio": 1250.00 },
    { "fecha": "2023-11-01", "precio": 1300.00 }
```

],

"stock": 50,

“imagen”: IMG

}

Historial de precios embebido para consultas rápidas.

Uso de categorías como array para mejorar la indexación en búsquedas.

Ventajas de MongoDB para Productos:

- Permite manejar productos con múltiples precios sin depender de relaciones complejas.
- Se pueden optimizar búsquedas creando un **índice de texto en `nombre` y `categorias`**.

El uso de **MongoDB** en esta plataforma de comercio electrónico se justifica por su flexibilidad, escalabilidad y rendimiento en consultas. Se recomienda implementar **sharding** en las colecciones de **pedidos y facturas** para manejar grandes volúmenes de datos.

REDIS (Base de datos clave - valor):

En una plataforma de comercio electrónico, la gestión de sesiones y carritos de compras requiere almacenamiento rápido, confiable y escalable. Para estas funcionalidades, **Redis** es una opción ideal debido a su **arquitectura en memoria y su modelo clave-valor**, que permite acceso a datos en **milisegundos**.

Alta velocidad: Redis almacena los datos en memoria, lo que permite tiempos de respuesta extremadamente rápidos.

Persistencia opcional: Aunque Redis es en memoria, permite configuraciones para persistir datos (AOF y RDB).

Soporte para TTL (Time-To-Live): Permite manejar sesiones con expiración automática.

Estructuras de datos avanzadas: Soporta listas, hashes y conjuntos, optimizando el manejo de carritos de compras.

Escalabilidad: Se puede usar Redis Cluster para distribuir la carga en múltiples nodos.

Uso de Redis para Autenticación y Sesión de Usuario

Justificación del Uso de Redis

La autenticación y la gestión de sesiones requieren una base de datos que:

1. **Permita accesos rápidos**, ya que la autenticación ocurre en cada solicitud.
2. **Almacene información temporal** y la elimine automáticamente después de cierto tiempo (TTL).
3. **Evite sobrecargar la base de datos principal (MongoDB)** con información que solo se necesita mientras el usuario está activo.

Modelo de almacenamiento en Redis

Redis almacena sesiones usando una **clave-valor**, donde la clave es el **ID de sesión** y el valor es un JSON con la información de la sesión del usuario.

Ejemplo de almacenamiento de sesión en Redis:

SETEX session:123456789 3600 '{"usuario_id": "C001", "nombre": "Carlos Pérez", "email": "carlos.perez@email.com", "rol": "cliente"}'

`SETEX`: Crea la clave con una expiración automática.

`"session:123456789"`: Identificador único de la sesión.

`3600`: Expira en 1 hora.

El valor es un **JSON con los datos de sesión** del usuario.

Recuperar una sesión activa

Cuando el usuario interactúa con la plataforma, su sesión se consulta en Redis: GET session:123456789

Donde esperaríamos el siguiente return: {"usuario_id": "C001", "nombre": "Carlos Pérez", "email": "carlos.perez@email.com", "rol": "cliente"}

Ventajas de Redis para sesiones:

- **Autenticación ultra rápida** (en memoria).
- **Expiración automática** para manejar sesiones de usuario.
- **Menor carga en MongoDB** (MongoDB solo almacena información persistente del usuario).

Uso de Redis para el Carrito de Compras

El carrito de compras debe permitir:

1. **Acceso rápido a los productos agregados.**
2. **Persistencia temporal** (eliminarse después de cierto tiempo si el usuario no finaliza la compra).
3. **Recuperar versiones anteriores del carrito**, en caso de que el usuario quiera volver a un estado anterior.

Almacenamiento en memoria con TTL: Permite borrar carritos inactivos automáticamente.

Estructuras de datos eficientes: Redis permite almacenar el carrito como un **hash o una lista ordenada**.

Soporte para historial de cambios: Redis ofrece **listas (List) o snapshots con copias previas**.

Modelo de Carrito de Compras en Redis

Almacenamiento del carrito en un `HASH`

Un carrito se almacena en Redis con el **ID del usuario como clave** y una estructura de hash con los productos y cantidades.

Un ejemplo sería: HSET carrito:C001 "producto:001" 2 "producto:002" 1

`"carrito:C001"` → Clave única del carrito del usuario `"C001"`.

`"producto:001" 2` → El usuario agrega **2 unidades** del producto `"001"`.

`"producto:002" 1` → El usuario agrega **1 unidad** del producto `"002"`.

Para obtener los productos del carrito: HGETALL carrito:C001

Donde debería retornar: {"producto:001": "2", "producto:002": "1"}

Ventaja: Recuperación instantánea del carrito sin necesidad de consultar una base de datos tradicional.

Guardar versiones anteriores del carrito

Si un usuario quiere **volver a un estado anterior del carrito**, podemos almacenar copias previas con listas (`LIST`). Ejemplo: Guardar un snapshot del carrito antes de modificarlo:

LPUSH historial:carrito:C001 '{"producto:001": 2, "producto:002": 1}'

Explicación:

- `LPUSH`: Agrega una nueva versión del carrito al inicio de la lista.

- `"historial:carrito:C001"`: Identificador del historial del carrito del usuario.

Recuperar el estado anterior del carrito

Si el usuario quiere volver a la versión anterior: LINDEX historial:carrito:C001 0

{"producto:001": 2, "producto:002": 1}

Ventajas:

- Permite **deshacer cambios en el carrito**.
- Se pueden almacenar múltiples versiones sin impactar el rendimiento.

El uso de **Redis como base de datos clave-valor** es ideal para manejar:

- **Autenticación y sesiones de usuario:** Permite almacenar sesiones de forma rápida y eficiente con **TTL**.
- **Carritos de compras:** Ofrece acceso rápido, persistencia temporal y un mecanismo para restaurar versiones anteriores.

Integración con MongoDB:

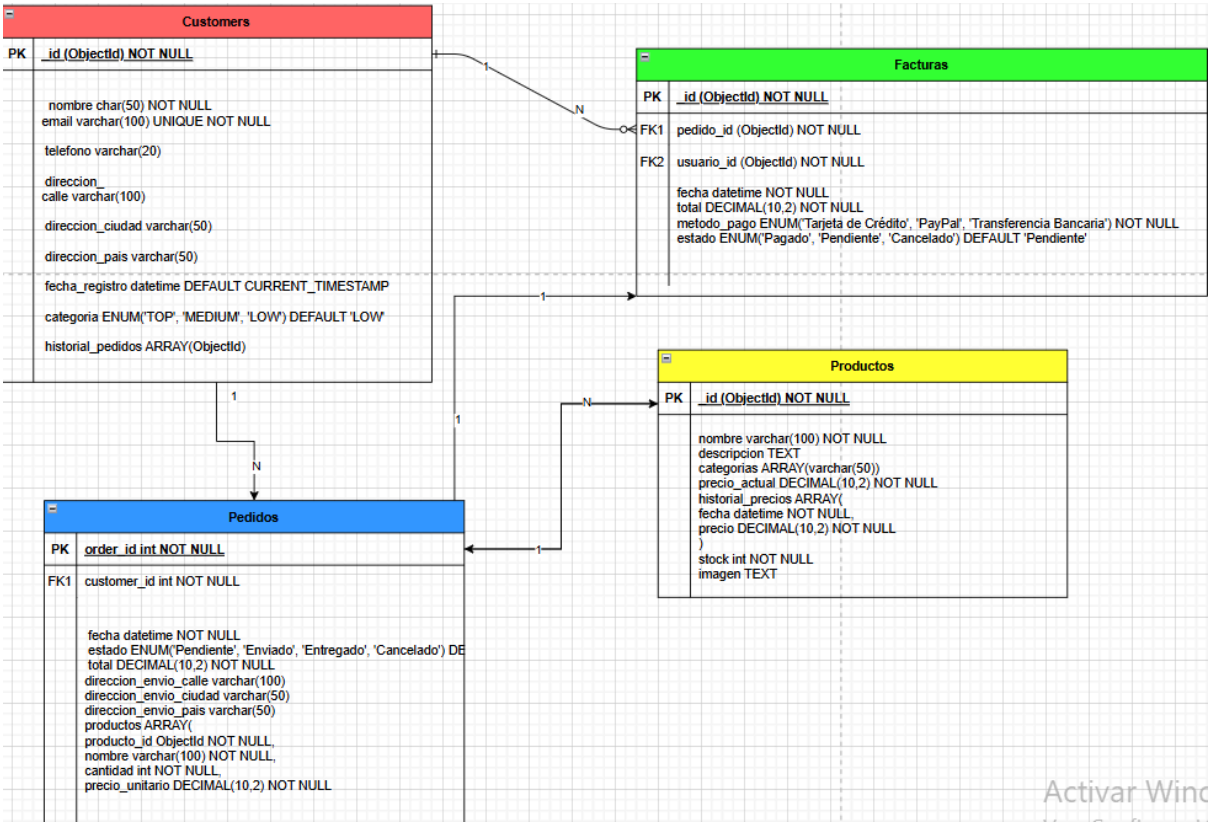
- Redis almacena **sesiones y carritos** de forma temporal.
- MongoDB almacena **datos persistentes** de usuarios y pedidos finalizados.

La combinación de MongoDB y Redis en esta arquitectura permite un equilibrio entre persistencia de datos y rendimiento optimizado.

- **MongoDB** se encarga del almacenamiento de datos estructurados de usuarios, pedidos, facturas y productos, asegurando escalabilidad mediante **sharding** y optimización con índices.
- **Redis** almacena sesiones y carritos de compras, garantizando respuestas en milisegundos sin sobrecargar MongoDB.
- **El backend gestiona la sincronización de datos entre ambas bases de datos**, permitiendo una integración eficiente y escalable.

Esta arquitectura permite un **alto rendimiento y escalabilidad**, asegurando que la plataforma pueda manejar grandes volúmenes de transacciones sin afectar la experiencia del usuario.

DER



ESTRUCTURA API

1. Introducción

Este documento presenta la estructura y funcionalidad de la API desarrollada para un sistema de comercio electrónico. La API gestiona usuarios, pedidos, facturas y productos mediante **Node.js, Express y MongoDB**.

A lo largo del informe se describirán los componentes principales del sistema, detallando su estructura, funcionalidad y la relación entre ellos.

---

2. Arquitectura General

El sistema sigue una arquitectura basada en **MVC (Modelo-Vista-Controlador)**, donde:

- **Modelos:** Definen la estructura de los datos almacenados en MongoDB.
- **Controladores:** Gestionan la lógica de negocio y las operaciones sobre los modelos.
- **Rutas:** Definen los puntos de acceso a la API y conectan con los controladores.

/ecommerce-api

│── models/ # Contiene los modelos de datos (MongoDB)

│── controllers/ # Contiene la lógica de negocio de la API

│── routes/ # Define los endpoints y su conexión con los controladores

│── app.js # Configuración principal del servidor

│── package.json # Dependencias del proyecto

/ecommerce-api

models/

── User.js # Modelo de usuario en Mongoose

── Order.js # Modelo de pedidos

── Product.js # Modelo de productos

── Invoice.js # Modelo de facturas

controllers/

── user.controller.js # Lógica de usuarios

── order.controller.js # Lógica de pedidos

── product.controller.js # Lógica de productos

── invoice.controller.js # Lógica de facturación

routes/

── user.routes.js # Rutas de usuario

── order.routes.js # Rutas de pedidos

── product.routes.js # Rutas de productos

── invoice.routes.js # Rutas de facturación

└── app.js # Configuración del servidor

3. Modelos de Datos

Los modelos de datos definen la estructura de la información almacenada en la base de datos. Se utilizan esquemas de **Mongoose** para establecer las relaciones entre los distintos elementos del sistema.

3.1. Modelo de Usuario (`User.js`)

Este modelo representa a los clientes de la plataforma e incluye información clave como su nombre, correo electrónico, dirección y pedidos realizados.

Atributos Principales

- **Información personal:** Nombre, correo electrónico, teléfono.
- **Seguridad:** Contraseña cifrada y estado de verificación del correo.
- **Categoría de usuario:** Clasificación en función de su historial de compras.
- **Historial de pedidos:** Referencia a los pedidos asociados al usuario.
- **Fecha de registro:** Para identificar cuándo se unió el usuario a la plataforma.

3.2. Modelo de Pedido (`Order.js`)

Este modelo gestiona los pedidos realizados por los usuarios, incluyendo información sobre los productos adquiridos, el estado del pedido y la dirección de envío.

Atributos Principales

- **Usuario:** Referencia al cliente que realizó el pedido.
- **Productos:** Lista de productos comprados con su cantidad y precio unitario.
- **Estado del pedido:** Puede ser "Pendiente", "Enviado", "Entregado" o "Cancelado".
- **Dirección de envío:** Ubicación donde se debe entregar el pedido.
- **Fecha y total:** Registro de la fecha de compra y el costo total del pedido.

3.3. Modelo de Factura (`Invoice.js`)

Este modelo almacena la información de las facturas generadas a partir de los pedidos.

Atributos Principales

- **Pedido asociado:** Referencia al pedido que originó la factura.
- **Usuario:** Identificación del cliente que realizó el pago.
- **Método de pago:** Puede ser "Tarjeta de Crédito", "PayPal" o "Transferencia Bancaria".
- **Estado del pago:** Puede ser "Pagado", "Pendiente" o "Cancelado".
- **Total y fecha:** Monto facturado y fecha de emisión de la factura.

3.4. Modelo de Producto (`Product.js`)

Este modelo representa los productos disponibles en la tienda.

Atributos Principales

- **Información básica:** Nombre, descripción y categoría del producto.
- **Precio actual:** Costo vigente del producto en la tienda.
- **Historial de precios:** Registros de cambios en el precio a lo largo del tiempo.
- **Stock disponible:** Cantidad de unidades en inventario.
- **Imagen del producto:** Enlace a la imagen representativa del producto.

4. Controladores

Los controladores gestionan la lógica de negocio y la interacción entre los modelos y las rutas. Son responsables de procesar las solicitudes de los clientes y devolver respuestas adecuadas.

4.1. Controlador de Usuarios (`user.controller.js`)

Gestiona todas las operaciones relacionadas con los usuarios, incluyendo:

- **Obtener usuarios:** Retorna una lista de todos los usuarios registrados.
- **Obtener usuario por ID:** Recupera la información de un usuario en específico.
- **Crear usuario:** Registra un nuevo usuario en la plataforma.
- **Actualizar usuario:** Permite modificar la información de un usuario existente.
- **Eliminar usuario:** Borra un usuario del sistema de forma permanente.

4.2. Controlador de Pedidos (`order.controller.js`)

Maneja las operaciones relacionadas con los pedidos.

- **Obtener pedidos:** Lista todos los pedidos registrados en la base de datos.
- **Obtener pedido por ID:** Permite consultar un pedido en particular.
- **Crear pedido:** Registra un nuevo pedido y calcula su total.
- **Actualizar pedido:** Modifica el estado o los productos dentro de un pedido.
- **Eliminar pedido:** Cancela y elimina un pedido si es necesario.

4.3. Controlador de Facturas (`invoice.controller.js`)

Gestiona la facturación de los pedidos realizados.

- **Obtener facturas:** Devuelve todas las facturas generadas en la plataforma.

- **Obtener factura por ID:** Busca una factura específica según su identificador.
- **Generar factura:** Crea una nueva factura basada en un pedido existente.
- **Actualizar estado de pago:** Modifica el estado de una factura (Pagado/Pendiente).
- **Eliminar factura:** Borra una factura si el pedido es cancelado.

4.4. Controlador de Productos (`product.controller.js`)

Se encarga de la administración de los productos disponibles en la tienda.

- **Obtener productos:** Devuelve el catálogo de productos disponibles.
- **Obtener producto por ID:** Consulta la información de un producto en particular.
- **Agregar nuevo producto:** Permite registrar un nuevo producto en el sistema.
- **Actualizar producto:** Modifica los datos de un producto existente, como precio o stock.
- **Eliminar producto:** Borra un producto si ya no está disponible en la tienda.

5. Rutas de la API

Las rutas definen los puntos de acceso a la API y conectan los controladores con el cliente.

5.1. Rutas de Usuarios (`user.routes.js`)

Establece las rutas para gestionar los usuarios, permitiendo obtener, crear, actualizar y eliminar registros de usuarios.

Endpoints:

```
router.post("/register", usersController.register);
```

```
router.post("/login", usersController.login);

router.post("/usuarios", usersController.createUser);

router.get("/:id", usersController.getUserById);

router.get("/", usersController.getAllUsers);

router.put("/:id", usersController.updateUser);

router.delete("/:id", usersController.deleteUser);
```

5.2. Rutas de Pedidos (`order.routes.js`)

Define los endpoints para gestionar pedidos, facilitando la consulta, creación, actualización y eliminación de pedidos.

Endpoints:

```
router.post("/pedidos", orderController.createOrder);

router.get("/:usuario_id", orderController.getOrdersByUser);

router.get("/", orderController.getAllOrders);

router.get("/:id", orderController.getOrderById);

router.put("/:id", orderController.updateOrder);

router.delete("/:id", orderController.deleteOrder);
```

5.3. Rutas de Facturas (`invoice.routes.js`)

Permite la consulta, generación y modificación de facturas a partir de pedidos realizados.

Endpoints:

```
router.post("/facturas", invoiceController.createInvoice);

router.get("/:usuario_id", invoiceController.getInvoicesByUser);

router.get("/", invoiceController.getAllInvoices);

router.get("/:id", invoiceController.getInvoiceById);

router.put("/:id", invoiceController.updateInvoice);

router.delete("/:id", invoiceController.deleteInvoice);
```

5.4. Rutas de Productos (`product.routes.js`)

Define los endpoints para administrar productos, incluyendo la gestión del catálogo y la actualización de precios.

Endpoints:

```
router.post("/", productController.createProduct);
```

```
router.get("/", productController.getAllProducts);;

router.get("/:id", productController.getProductById);

router.put("/:id", productController.updateProduct);

router.delete("/:id", productController.deleteProduct);
```

5.5. Rutas de Carrito(`cart.routes.js`)

Permitirá realizar el CRUD del carrito de compras

Endpoints

```
router.post("/", productController.createProduct);
router.get("/", productController.getAllProducts);;
router.get("/:id", productController.getProductById);
router.put("/:id", productController.updateProduct);
router.delete("/:id", productController.deleteProduct);
```

6. Configuración del Servidor (`app.js`)

El archivo **`app.js`** configura la API y establece las rutas principales. Se encarga de inicializar el servidor Express y conectar con la base de datos MongoDB.

Tareas principales:

- Cargar los módulos necesarios (Express, Mongoose, etc.).
- Conectar con la base de datos.
- Definir las rutas de la API.
- Iniciar el servidor en el puerto correspondiente.