

Modelo de Vistas "4 + 1"

Vista de Escenarios

Actores: 

1. Usuario final:

- Se registra y autentica.
- Realiza pedidos de productos y consulta sus pedidos.
- Cancela pedidos si cumple las condiciones.

2. Administrador:

- Crea y actualiza productos (atributos dinámicos, stock).
- Gestiona el estado de los pedidos.

3. Base de Datos:

- Almacena información de usuarios, productos y pedidos.

Principales Casos de Uso: 

1. **Registrar Usuario:** El usuario crea una cuenta y valida su correo.
2. **Login Usuario:** El usuario inicia sesión en su cuenta.
3. **Hacer Pedido:** Selecciona productos, verifica restricciones, confirma el pedido y decrece el stock.
4. **Cancelar Pedido:** Revierte el stock si se cumplen las condiciones.
5. **Gestión de Pedidos:** El administrador cambia estados de pedidos.
6. **Crear Producto:** El administrador agrega productos con atributos configurables.
7. **Recuperar Contraseña Usuario:** Si el usuario olvida su contraseña, la recupera mediante a 3 preguntas de seguridad.

Vista Lógica

Capas: 

1. Capa de Frontend

• Responsabilidades:

- Interactúa directamente con los usuarios finales (tanto usuarios como administradores).
- Realiza solicitudes a los endpoints del back-end utilizando JSON y POSTMAN.
- Presenta la información de forma interactiva y amigable, incluyendo formularios para el registro, autenticación, creación de pedidos, y visualización de pedidos confirmados.
- Maneja errores devueltos por el back-end, como reglas no cumplidas o falta de stock.

• Principales Componentes:

a. Componentes de Interfaz de Usuario:

- **Pantallas del Usuario:** Registro, inicio de sesión, creación de pedidos, lista de pedidos, cancelación de pedidos.
- **Pantallas del Administrador:** Gestión de productos (creación, edición, stock) y cambio de estados de pedidos.

b. Servicios HTTP:

- Consume endpoints expuestos por los controladores del back-end.

c. Manejo de Estados:

- Control del flujo de datos entre las solicitudes y la interfaz.
- Actualización dinámica de la interfaz en función de las respuestas del back-end (por ejemplo, mostrar un error si no se cumplen las reglas del pedido).

- **Interacciones con el Back-end:**

- El front-end se comunica con las siguientes capas del back-end:
 - **Capa de Controladores:** Envía solicitudes a los controladores (por ejemplo, `OrderController` para crear un pedido).
 - **Capa de Servicios:** Recibe respuestas procesadas desde los servicios del back-end.
- Los datos se envían y reciben en formato JSON para facilitar la interoperabilidad.

- **Tecnologías Utilizadas:**

- **JavaScript:** Para lógica del cliente y manipulación del DOM.
- **HTML/CSS:** Para la estructura y diseño visual.

2. Capa de Controladores:

- Maneja las solicitudes HTTP.
- Controladores como `OrderController`, `ProductController`, y `UserController` gestionan entradas y envían respuestas.
- Usa DTOs (`OrderDTO`, `UserDTO`) para la comunicación con el front-end.

3. Capa de Servicios:

- Contiene la lógica principal del negocio.
- Servicios como `OrderServices`, `ProductServices` y `UserServices` gestionan reglas dinámicas, stock, usuarios, y autenticación.
- Ejemplo: `OrderServices` valida pedidos contra las reglas dinámicas implementadas (Las cuales son objetos que extienden la clase abstracta `BaseRule`, por ejemplo `NoCompatibilityTypesRule`).

4. Capa de Dominio:

- Representa los objetos centrales: `User`, `Product`, `Order`, y sus estados (`PedidoState`, `CanceladoState`, etc.).
- `User`, `Product`, `Order` representan sus contrapartes en la base de datos.
- Aplica patrones como `State` para gestionar el flujo de estados de los pedidos.

5. Capa de Persistencia:

- Gestiona la interacción con la base de datos.
- Repositorios como `OrderRepository` y `ProductRepository` encapsulan las operaciones de almacenamiento y recuperación.

Algunas Interacciones:

- **Registrar Usuario:**

- `UserController` recibe datos y llama a `UserServices` para validación y persistencia.
- `UserServices` usa `UserRepository` para guardar la información en la base de datos.

- **Realizar Pedido:**

- `OrderController` valida la solicitud.
- `OrderServices` aplica reglas (`MaxWeightRule`, etc.), verifica stock (`ProductServices`) y guarda el pedido (`OrderRepository`).
- Envía confirmación al usuario.

- **Crear Producto:**

- `ProductController` valida la solicitud.
- `ProductServices` crea el nuevo producto con los atributos recibidos.
- Envía la confirmación al administrador.

Vista Desarrollo

Módulos Principales:

1. Controladores:

- `OrderController.java`, `ProductController.java`, `UserController.java`: Gestionan solicitudes HTTP y coordinan con servicios.

2. Servicios:

- `OrderServices.java` : Gestión de pedidos y reglas.
- `ProductServices.java` : Gestión de stock y atributos dinámicos.
- `JwtServices.java` : Manejo de autenticación.
- `UserServices.java` : Gestión de usuarios y datos personales.

3. Dominio:

- Clases como `Order` , `Product` , `User` encapsulan la lógica de negocio.
- Submódulos:
 - **Estados de Pedido:** Implementa *State Pattern* (`PedidoState` , `CanceladoState`).
 - **Reglas Dinámicas:** Clases como `MaxWeightRule` implementan validaciones extensibles.

4. Persistencia:

- Repositorios: `OrderRepository` , `ProductRepository` , `UserRepository` encapsulan la interacción con la base de datos.

Vista de Procesos

Flujo: Crear Pedido

1. **Solicitud HTTP:** El front-end envía un JSON con el pedido al endpoint del back-end.
2. **Controlador:** `OrderController` valida la solicitud y la envía a `OrderServices` .
3. **Lógica de Negocio:**
 - `OrderServices` verifica restricciones dinámicas (`Rule`).
 - `ProductServices` verifica stock.
4. **Persistencia:**
 - Si todo es válido, `OrderRepository` guarda el pedido y `ProductRepository` actualiza el stock.
5. **Notificación:** Se envía un email con la confirmación.

Flujo: Cancelar Pedido

1. El usuario envía una solicitud al back-end.
2. `OrderServices` verifica condiciones y llama a `ProductServices` para restituir el stock.
3. Actualiza el estado del pedido a "Cancelado".

Flujo: Registrar Usuario

1. **Solicitud HTTP:**
 - El front-end envía un JSON con los datos del usuario (nombre, apellido, email, contraseña, etc.) al endpoint correspondiente del back-end.
 - Ejemplo: `POST /api/users/register` .
2. **Controlador:**
 - `UserController` recibe la solicitud, valida los datos iniciales.
 - Llama a `UserServices` para procesar la lógica de negocio.
3. **Lógica de Negocio:**
 - `UserServices` :
 - Verifica que el email no esté registrado.
 - Hashea la contraseña para almacenamiento seguro.
 - Construye el usuario utilizando el patrón **Builder** para mantener consistencia.
4. **Persistencia:**
 - `UserRepository` guarda los datos del nuevo usuario en la base de datos.
5. **Notificación:**
 - Se envía un email de verificación para validar la dirección de correo electrónico.

6. Respuesta:

- El back-end responde con un estado de éxito.

Flujo: Loguear Usuario

1. Solicitud HTTP:

- El front-end envía un JSON con las credenciales del usuario (email y contraseña) al endpoint de autenticación.
- Ejemplo: `POST /api/users/login`.

2. Controlador:

- `UserController` recibe la solicitud y llama a `UserServices` para procesar las credenciales.

3. Lógica de Negocio:

- `UserServices`:
 - Busca al usuario en la base de datos utilizando el email.
 - Compara la contraseña ingresada con la almacenada.
 - Si las credenciales son válidas, genera un **JWT** (token de acceso) para la sesión.

4. Persistencia:

- `UserRepository` consulta la base de datos para verificar la existencia del usuario y obtener su información.

5. Respuesta:

- Devuelve un token JWT al front-end para autenticar futuras solicitudes o un mensaje de error si las credenciales son inválidas.

Flujo: Crear Producto

1. Solicitud HTTP:

- El administrador envía un JSON con los datos del producto (nombre, tipo, peso, atributos dinámicos, etc.) al endpoint correspondiente del backend.

2. Controlador:

- `ProductController` recibe la solicitud, valida los datos del producto.
- Llama a `ProductServices` para manejar la lógica de creación.

3. Lógica de Negocio:

- `ProductServices`:
 - Verifica que el producto no exista ya en la base de datos.
 - Crea una instancia de `Product` utilizando los datos recibidos y asigna atributos dinámicos si corresponde.

4. Persistencia:

- `ProductRepository` guarda el nuevo producto en la base de datos.

5. Respuesta:

- El back-end devuelve un estado de éxito con los detalles del producto creado o un mensaje de error si algo falla.

Vista Física [↗](#)

Despliegue: [↗](#)

1. Front-end:

- Implementado en **JavaScript**, interactúa con el back-end a través de endpoints REST.
- Pruebas realizadas con Postman.

2. Back-end:

- Contenedor de Docker ejecutando el servicio Java.
- Servicio expuesto mediante endpoints REST.

3. Base de Datos:

- **MySQL/MariaDB** desplegado como contenedor Docker.

- Tres bases de datos separadas para usuarios, productos y pedidos.


4. Correo Electrónico:

- Servicio externo configurado para enviar notificaciones.

Esquema de Despliegue:

[Frontend] --> [Backend (Docker)] --> [Database (MariaDB)]

Atributos de Calidad

En el desarrollo de nuestro proyecto, buscamos potenciar los atributos de calidad que consideramos esenciales para cumplir con los requisitos de la consigna. Nos centramos en cinco atributos clave que consideramos fundamentales para garantizar un sistema robusto, escalable y fácil de mantener, con un enfoque en la experiencia del usuario y la seguridad de los datos. 

Escalabilidad

Uno de los objetivos principales fue crear un **Producto Mínimo Viable (MVP)** que estuviera diseñado desde sus bases para soportar una futura escalabilidad. El sistema implementado tiene la capacidad de crecer tanto en términos de usuarios como en funcionalidades, siguiendo las directrices que hemos establecido en el *Informe Final*. Al centrarnos en un diseño modular y bien estructurado, aseguramos que nuestro MVP pueda ser la base de una versión más compleja y robusta. Esto se logró adoptando tecnologías y prácticas que permiten un crecimiento tanto vertical (mejoras en la infraestructura) como horizontal (adición de nuevas instancias de componentes o servicios).

Confiabilidad

La confiabilidad del sistema fue otro de los pilares fundamentales en el diseño y desarrollo del proyecto. Implementamos un enfoque riguroso en las **pruebas unitarias**, buscando alcanzar una cobertura amplia del código. Estas pruebas no solo abarcan los escenarios ideales, sino también aquellos que representan casos borde o conflictos.

El objetivo principal de estas pruebas fue garantizar que cada componente del sistema funcione correctamente en todas las situaciones posibles. Esto nos permitió identificar y resolver errores de manera temprana, asegurando así que el MVP cumpliera con los comportamientos esperados y ofreciera una experiencia consistente y confiable para los usuarios.

Seguridad

La seguridad de los datos y la protección del sistema frente a accesos no autorizados fueron prioridades indiscutibles en nuestro proyecto. Implementamos un modelo robusto de **autenticación y autorización** basado en **OAuth** y **JSON Web Tokens (JWT)**.

Este enfoque permitió:

1. Garantizar que solo los usuarios autenticados y autorizados pudieran acceder a las funcionalidades correspondientes a su rol.
2. Proteger tanto la información sensible de los usuarios como los datos del sistema contra vulneraciones externas.
3. Asegurar la integridad y confidencialidad de las transacciones realizadas dentro de la plataforma.

Usabilidad

La experiencia del usuario fue un factor clave en nuestro diseño. En el desarrollo de la interfaz, tanto para los usuarios finales como para el **Backoffice**, buscamos una **usabilidad óptima**, priorizando la claridad, simplicidad y comodidad en cada interacción.

Para lograr esto, diseñamos una interfaz gráfica intuitiva y visualmente agradable que:

1. Facilite la navegación y el acceso a las funcionalidades del sistema.
2. Reduzca la curva de aprendizaje para nuevos usuarios.
3. Garantice una experiencia positiva tanto para usuarios generales como para administradores.

Mantenibilidad

Un diseño que facilite el mantenimiento y evolución del sistema fue otro de los atributos clave que trabajamos.

Para ello, nos enfocamos en:

- **Distribuir responsabilidades de manera adecuada** entre las clases y los componentes.
- Seguir principios de diseño como los **criterios SOLID**.
- Adoptar una arquitectura en capas (**Layers**) y patrones de diseño como **State**, que ayudan a gestionar de manera efectiva la lógica de negocio y el flujo del sistema.