Taller Estructuración Proyecto Front-End y Back-End

Diego Fernando Rodríguez Arauz

Emerson Andrés Vargas Torres

Danny Livzka Vargas Rico

Kevin Camilo Arias Lizarazo

Yohan alexander Rangel Méndez

1. Investiga como se estructura un proyecto escalable

Organización del Código y Estructura de Carpetas

La organización del código es esencial para la mantenibilidad. En el front-end, se recomienda dividir el proyecto en carpetas como components, services, assets, utils y views. En el back-end, se deben estructurar las carpetas en controllers, services, models, repositories y middlewares. Además, es importante mantener una separación clara entre la lógica de negocio y la lógica de presentación. El uso de convenciones de nomenclatura y documentación adecuada facilita la colaboración entre desarrolladores.

Cada módulo debe contar con su propia documentación y pruebas unitarias. Se recomienda utilizar JSDoc o Swagger para documentar las APIs y facilitar la integración con otros sistemas. La estructura de carpetas debe permitir la escalabilidad, evitando dependencias innecesarias entre módulos. Además, el uso de monorepos con herramientas como Nx puede ser beneficioso para proyectos grandes, permitiendo una mejor gestión del código y facilitando la colaboración entre equipos.

Gestión de Dependencias y Versionado

El uso de herramientas como npm, yarn o pip para gestionar dependencias es crucial. Se recomienda mantener un archivo package.json o requirements.txt bien documentado y actualizado. Además, el versionado del código con Git y la implementación de estrategias como Git Flow permiten un desarrollo ordenado y colaborativo. La integración de CI/CD (Integración y Despliegue Continuo) ayuda a automatizar pruebas y despliegues, reduciendo errores en producción.

Es recomendable utilizar Semantic Versioning (SemVer) para gestionar versiones de software, asegurando compatibilidad entre distintas versiones. La automatización de pruebas con herramientas como Jenkins, GitHub Actions o GitLab CI/CD mejora la calidad del código y reduce el riesgo de errores en producción. Además, la implementación de feature flags permite realizar despliegues progresivos y minimizar el impacto de cambios en el sistema.

Base de Datos y Gestión de Datos

La elección de la base de datos depende de los requisitos del proyecto. Para proyectos escalables, se recomienda el uso de bases de datos relacionales como PostgreSQL o NoSQL como MongoDB. La estructura de la base de datos debe seguir principios de normalización para evitar redundancias y mejorar la eficiencia. Además, el uso de ORMs (Object-Relational Mapping) como Sequelize o TypeORM facilita la gestión de datos y mejora la seguridad.

Es importante definir una estrategia de sharding y replicación para mejorar la escalabilidad y disponibilidad de los datos. La implementación de indexación eficiente y

optimización de consultas reduce los tiempos de respuesta y mejora el rendimiento del sistema. Además, el uso de ETL (Extract, Transform, Load) permite gestionar grandes volúmenes de datos de manera eficiente, facilitando la integración con sistemas de análisis y reportes.

Seguridad y Autenticación

La seguridad es un aspecto crítico en proyectos escalables. Se recomienda implementar JWT (JSON Web Tokens) para autenticación, junto con OAuth 2.0 para autorización. Además, el uso de middleware de seguridad en el back-end, como Helmet.js y CORS, protege contra ataques comunes. En el front-end, se deben evitar vulnerabilidades como XSS (Cross-Site Scripting) y CSRF (Cross-Site Request Forgery) mediante buenas prácticas de desarrollo.

Es fundamental realizar auditorías de seguridad periódicas y aplicar pruebas de penetración para identificar vulnerabilidades. La implementación de Zero Trust Architecture mejora la seguridad del sistema, asegurando que cada solicitud sea verificada antes de acceder a los recursos. Además, el uso de cifrado de datos con AES-256 y TLS protege la información sensible contra accesos no autorizados.

Escalabilidad y Despliegue

Para garantizar la escalabilidad, se recomienda el uso de contenedores con Docker y orquestación con Kubernetes. La implementación de balanceadores de carga y caching con Redis mejora el rendimiento del sistema. Además, el despliegue en servicios en la nube como AWS, Azure o Google Cloud permite escalar la infraestructura según la demanda.

La adopción de estrategias como blue-green deployment y canary deployment minimiza riesgos en actualizaciones.

El uso de serverless computing con plataformas como AWS Lambda o Google Cloud Functions permite reducir costos y mejorar la escalabilidad. La implementación de auto-scaling en la infraestructura garantiza que los recursos se ajusten dinámicamente a la demanda. Además, el monitoreo con herramientas como Datadog y New Relic permite detectar problemas en tiempo real y optimizar el rendimiento del sistema.

Pruebas y Mantenimiento

Las pruebas son esenciales para garantizar la calidad del software. Se recomienda implementar pruebas unitarias con Jest o Mocha, pruebas de integración y pruebas de carga para evaluar el rendimiento. Además, el monitoreo con herramientas como Prometheus y Grafana permite detectar problemas en tiempo real. La documentación del código y la capacitación del equipo aseguran la continuidad del proyecto.

Es recomendable utilizar pruebas automatizadas con herramientas como Selenium y Cypress para validar la funcionalidad del sistema. La implementación de observabilidad con OpenTelemetry permite analizar el comportamiento del sistema y detectar cuellos de botella. Además, la adopción de DevOps mejora la colaboración entre equipos de desarrollo y operaciones, asegurando una entrega continua y eficiente.

Optimización y Mejora Continua

Un proyecto escalable debe estar en constante mejora. La optimización del código mediante lazy loading, minificación y tree shaking en el front-end mejora la velocidad de

carga. En el back-end, la optimización de consultas y el uso de indexación en bases de datos reduce tiempos de respuesta. La recopilación de métricas y el análisis de rendimiento permiten tomar decisiones informadas para mejorar la eficiencia del sistema.

La implementación de A/B testing permite evaluar distintas versiones del sistema y optimizar la experiencia del usuario. El uso de machine learning para predecir patrones de uso y mejorar la eficiencia del sistema es una estrategia avanzada para proyectos escalables. Además, la adopción de arquitecturas event-driven mejora la capacidad de respuesta y escalabilidad del sistema.

2. Estructuración Proyecto Gestión de Inventarios Sector Automotriz

Descripción General del Proyecto

El proyecto de software denominado “Sistema de Gestión de Inventario para La Casa del Embrague RJC” tiene como finalidad automatizar y optimizar los procesos operativos asociados al manejo de productos, pedidos, ventas, usuarios y reportes. Este sistema está orientado a cubrir las necesidades específicas del sector automotriz, enfocándose en la comercialización y trazabilidad de autopartes. El desarrollo se basa en una arquitectura de microservicios, que permite distribuir las funcionalidades en servicios independientes, mejorando la escalabilidad y el mantenimiento. Desde su concepción, el proyecto ha sido diseñado con un enfoque moderno y profesional, utilizando tecnologías líderes en la industria del software.

El sistema proporciona funcionalidades como registro de productos, gestión de inventario, cálculo automático de precios mayoristas y minoristas, integración con facturación electrónica, generación de reportes dinámicos, control de usuarios con roles

diferenciados, y más. Su diseño modular permite que cada componente pueda evolucionar independientemente, facilitando futuras integraciones con otros sistemas (por ejemplo, ERP o CRM). El sistema es accesible desde múltiples dispositivos y proporciona una experiencia de usuario intuitiva y fluida, incluso para personal sin formación técnica.

Tecnologías Utilizadas

Tecnologías Utilizadas en el Proyecto

El sistema está diseñado con un conjunto de tecnologías modernas y ampliamente adoptadas en la industria, garantizando rendimiento, estabilidad, escalabilidad y portabilidad. La selección de herramientas se basa en su compatibilidad con arquitecturas de microservicios, su madurez en el mercado, el respaldo de una comunidad activa y su facilidad de integración con otros sistemas. Cada tecnología cumple un rol fundamental en la construcción de una plataforma robusta y eficiente, asegurando una experiencia óptima para los usuarios y una administración eficaz de los recursos.

Front-End: React.js

El front-end del sistema está desarrollado con React.js, una biblioteca de JavaScript ampliamente utilizada para la construcción de interfaces de usuario interactivas y reactivas. React permite la creación de componentes reutilizables, lo que facilita el mantenimiento y la escalabilidad del código. Su enfoque basado en Virtual DOM optimiza la actualización de la interfaz, mejorando el rendimiento de la aplicación. Además, se integra con herramientas como Redux para la gestión del estado global y Styled Components para la estilización dinámica. La modularidad de React permite una separación clara de responsabilidades, asegurando una experiencia fluida y eficiente para los usuarios.

Back-End: Java con Spring Boot

El back-end está construido con Java y Spring Boot, una combinación poderosa para el desarrollo de microservicios REST robustos, seguros y escalables. Spring Boot simplifica la configuración y el despliegue de aplicaciones, permitiendo una integración eficiente con bases de datos y servicios externos. Su arquitectura modular facilita la separación de lógica de negocio en capas bien definidas, mejorando la mantenibilidad del código. Además, incorpora mecanismos avanzados de seguridad, como autenticación con OAuth 2.0 y gestión de sesiones con JWT, garantizando la protección de los datos. La compatibilidad con Spring Cloud permite la orquestación de microservicios y la implementación de patrones como circuit breakers para mejorar la resiliencia del sistema.

Bases de Datos: MariaDB, PostgreSQL y MongoDB

El sistema emplea una combinación de bases de datos relacionales y NoSQL, optimizando el almacenamiento y la gestión de datos según las necesidades específicas de cada módulo. MariaDB se utiliza para operaciones transaccionales estructuradas, asegurando integridad y consistencia en los datos. PostgreSQL proporciona capacidades avanzadas de consultas y escalabilidad, ideal para manejar grandes volúmenes de información. MongoDB, por su parte, permite el almacenamiento flexible de datos no estructurados, facilitando la gestión de documentos JSON y la integración con sistemas dinámicos. La combinación de estas tecnologías garantiza un acceso rápido y eficiente a la información, optimizando el rendimiento del sistema.

Api Rest: Comunicación entre Componentes

La comunicación entre los distintos módulos del sistema se realiza mediante API REST, asegurando un intercambio de datos estándar y desacoplado. Este enfoque permite que los microservicios operen de manera independiente, facilitando la escalabilidad y el mantenimiento del sistema. La API está documentada con Swagger, lo que mejora la comprensión y el uso de los endpoints por parte de los desarrolladores. Además, se implementan estrategias de caching y rate limiting para optimizar el rendimiento y evitar sobrecargas en el servidor. La seguridad se refuerza con autenticación basada en tokens, garantizando que solo usuarios autorizados puedan acceder a los recursos.

Orquestación: Docker y Docker Compose

Para garantizar la portabilidad y escalabilidad, el sistema utiliza Docker para la contenerización de servicios, permitiendo que cada microservicio se ejecute de manera aislada y optimizada. Docker Compose facilita la gestión de múltiples servicios, asegurando una configuración eficiente y reproducible en distintos entornos. La integración con Kubernetes permite la orquestación avanzada de contenedores, asegurando un despliegue automatizado y una administración eficiente de los recursos. Este enfoque minimiza conflictos entre dependencias y mejora la estabilidad del sistema, permitiendo una rápida adaptación a cambios y actualizaciones.

Control de Versiones: Git y GitHub

El sistema emplea Git como VCS (Version Control System), asegurando un seguimiento preciso de los cambios en el código y facilitando la colaboración entre desarrolladores. GitHub se utiliza como repositorio remoto, permitiendo la gestión

eficiente de ramas y la implementación de estrategias como Git Flow para un desarrollo estructurado. La integración con CI/CD (Continuous Integration/Continuous Deployment) mediante GitHub Actions automatiza pruebas y despliegues, reduciendo errores en producción y mejorando la estabilidad del sistema. Este enfoque garantiza una entrega continua y eficiente, optimizando el ciclo de desarrollo.

Entorno de Desarrollo: Herramientas Clave

El desarrollo del sistema se lleva a cabo en un entorno optimizado con herramientas especializadas. Visual Studio Code se utiliza como editor principal, proporcionando una interfaz intuitiva y extensible con plugins para mejorar la productividad. Git CLI permite la gestión eficiente de versiones desde la terminal, facilitando la integración con repositorios remotos. Docker CLI se emplea para la administración de contenedores, asegurando un despliegue controlado y reproducible. Postman se utiliza para pruebas de endpoints, permitiendo la validación de la API REST y la detección temprana de errores. Este conjunto de herramientas garantiza un flujo de trabajo ágil y estructurado, optimizando el desarrollo y la implementación del sistema.

Estructura General del Proyecto

El backend del sistema está organizado en capas bajo el patrón MVC (Modelo-Vista-Controlador), separando claramente responsabilidades. A nivel de carpetas, se definen paquetes para controller, service, repository y model. Cada microservicio cuenta con su propia estructura y se gestiona de forma independiente. Además, existen archivos de configuración global como `application.properties`, `Dockerfile`, `pom.xml` y `docker-compose.yml`. El frontend, por su parte, sigue la estructura de componentes típica de

React, con organización por vistas, componentes reutilizables, servicios y hooks.

La base del proyecto backend está bajo el directorio `src/main/java`, donde se aloja toda la lógica del negocio. En `resources` se encuentran los archivos de configuración y plantillas. Las pruebas están en `src/test/java`. Cada microservicio puede ser desplegado y probado por separado, y se integran de manera coherente en un entorno común a través de Docker. Esta estructura facilita el mantenimiento, la trazabilidad del código y el trabajo en equipo.

Estructura del Proyecto con Microservicios

developmen\api

├── .mvn\wrapper
│ └── maven-wrapper.properties
├── src
│ ├── main
│ │ ├── java\dev\gosh\api
│ │ │ ├── controller
│ │ │ │ └── UserController.java
│ │ │ ├── model
│ │ │ │ └── User.java
│ │ │ ├── repository
│ │ │ │ └── UserRepository.java
│ │ │ ├── service
│ │ │ │ └── UserService.java
│ │ │ └── ApiClutchHouseApplication.java

│ ├── resources

│ │ └── application.properties

│ └── test\java\dev\gosh\api

│ └── ApiClutchHouseApplicationTests.java

├── target

│ ├── classes

│ │ └── application.properties

│ ├── generated-sources

│ ├── maven-status

│ └── test-classes

├── .gitignore

├── docker-compose.yml

├── Dockerfile

├── mvnw / mvnw.cmd

├── pom.xml

└── Snake

Principales Características de la Arquitectura

La arquitectura del sistema está basada en microservicios, lo que permite distribuir las responsabilidades en servicios independientes, desacoplados y autónomos. Esto mejora la escalabilidad, facilita la implementación continua y permite asignar recursos por servicio. Cada microservicio expone su propia API REST y se comunica con otros a través de HTTP. Se utiliza contenedores Docker para asegurar la portabilidad y consistencia del entorno de ejecución.

Los servicios están diseñados para ser tolerantes a fallos, permitiendo la actualización o reemplazo de componentes sin detener todo el sistema. También se considera la futura incorporación de un orquestador como Kubernetes. El sistema maneja distintos tipos de base de datos según el caso: relacionales para operaciones estructuradas y MongoDB para datos más dinámicos o flexibles, como registros de auditoría y logs.

Buenas Prácticas de Desarrollo Aplicadas

Durante el desarrollo del proyecto se aplican buenas prácticas recomendadas por la industria, tales como: uso de control de versiones con ramas separadas (feature, develop, main), documentación de código con Javadoc, pruebas automatizadas con JUnit y Postman, así como revisión de código entre pares. También se emplea integración continua a través de workflows en GitHub Actions para compilar y probar automáticamente cada push.

Además, se documentan todos los endpoints de las APIs REST usando Swagger/OpenAPI, se validan los datos de entrada a través de anotaciones de Spring, y se usan DTOs (Data Transfer Objects) para evitar exponer directamente las entidades de base de datos. Las configuraciones sensibles se gestionan mediante variables de entorno, promoviendo la seguridad y facilitando el despliegue multiambiente (dev, test, prod).

Escalabilidad y Mantenimiento.

La elección de una arquitectura de microservicios permite escalar horizontalmente los componentes del sistema según la demanda. Por ejemplo, si el módulo de búsqueda de productos es el más utilizado, puede desplegarse múltiples veces en distintos contenedores sin afectar al resto. Esta flexibilidad es clave para sistemas que deben operar 24/7 o soportar

múltiples transacciones concurrentes. Docker y su integración con orquestadores futuros permitirán un escalado automático más eficaz.

En cuanto al mantenimiento, el sistema sigue una filosofía de separación de preocupaciones, lo que permite identificar, aislar y corregir errores rápidamente. La estructura modular y el uso de patrones como MVC, DTO y repositorios promueven la reutilización y evitan duplicación de código. Asimismo, el monitoreo de logs, la trazabilidad con herramientas como Git y la documentación técnica aseguran que el sistema pueda mantenerse de forma eficiente a largo plazo.