

**Índice de Contenidos**

Introducción

Justificación del Proyecto – Contexto

Alcance de esta Etapa

Diagrama de Arquitectura de Microservicios

Estructura del Proyecto

6.1 Entity

6.2 Repository

6.3 Service

6.4 Controller

Implementación de los Servicios REST

7.1 Endpoints CRUD

7.2 Inyección de Dependencias

Implementación de Vistas (GET)

Base de Datos

9.1 Estructura y Comandos SQL

9.2 Integración con Spring Boot

9.3 Ventajas

Archivo pom.xml y Dependencias

10.1 Dependencias Usadas

10.2 Estructura del Archivo

Uso de Git y GitHub

11.1 Comandos Básicos

11.2 Repositorio y Documentación

Conclusión

**Información del proyecto**

Datos

|  | Empresa / Organización | Perfulandia |
| --- | --- | --- |
| Fecha de inicio/fin | 16/03/2025 - 27/05/2025 |
| Cliente | Empresa de Perfumes |
| Jefe de Proyecto | Bastian |

Gerente de Proyecto

| **Nombre** | **Cargo** | **Departamento / División** |
| --- | --- | --- |
| Bastian Reyes | Scrum Master | Informática |

Lista de Interesados (stakeholders)

| **Nombre** | **Tipo** | **Cargo** | **Departamento / División** |
| --- | --- | --- | --- |
| Cristian Lizama | Cliente | Full Stack Dev | Informática |
| Joshua Mardones | Cliente | DevOps | Informática |

**Introducción**

| perfulandia SPA ha estado usando un sistema monolítico, pero . Con el paso del tiempo, este tipo de sistema se ha vuelto complejo de mantener, escalar y modificar, generando problemas en la disponibilidad del servicio, en la gestión interna y en la experiencia del cliente. Estas limitaciones han evidenciado la necesidad de adoptar una solución más flexible y robusta. Por eso, se está pensando en pasarse a una arquitectura de microservicios, que básicamente permite dividir el sistema en partes más pequeñas que se pueden manejar por separado.  Frente a este escenario que enfrenta perfulandia spa , nuestro equipo de desarrollo ha iniciado un proceso de transformación digital enfocado en migrar a una arquitectura basada en microservicios, de este modo dejaremos atrás la arquitectura monolítica. la nueva arquitectura de microservicios nos permite dividir el sistema en componentes independientes, autónomos y especializados. |
| --- |

Justificación del proyecto – Contexto

| SPA Perfulandia está en medio de un proceso de transformación digital que involucra migrar desde una arquitectura monolítica a una basada en microservicios para mejorar la escalabilidad, el rendimiento y la mantenibilidad de su sistema de información. Para lograrlo, se ha implementado un proyecto académico de varias etapas, esta es la segunda. Este documento es un informe de la implementación de tres microservicios funcionales: Usuarios, Productos y Ventas. Todos se crearon siguiendo buenas prácticas de diseño y usando tecnologías modernas del desarrollo backend, como Spring Boot, JPA y Maven. Luego, se plantearon las capacidades CRUD, asegurando la interacción efectiva con MySQL como base de datos y la validación a través de Postman. En esta fase, la comunicación entre servicios, la modularidad del código y la configuración propiamente del ciclo de vida del proyecto a través de Git y GitHub lo hicieron en plataformas adecuadas y de código abierto. Todo esto se alinea con la visión de mantener la arquitectura flexible y mantenible, lista para infraestructura en la nube con AWS y otros. El microservicio de productos implementado aporta un valor esencial a la administración del inventario y el flujo de ventas y es un paso crucial en la migración a un sistema distribuido robusto. Por lo tanto, cree que esta etapa es un hito importante y necesario en la modernización del sistema de información de SPA Perfulandia. |
| --- |

**Alcance de esta etapa**

| En resumen, en esta fase del proyecto se crearon e implementaron tres microservicios que funcionan: usuarios, productos y ventas. Los tres, de hecho, lo fueron para garantizar operaciones tipo CRUD completas, es decir, crear, leer, actualizar y eliminar. Estos utilizan tecnologías de hoy, incluidos Spring Boot, Spring Data JPA y MySQL, y, en particular, el servicio del microservicio titulado productos es un ejemplo de un servidor con endpoints RESTful nuevamente, GET, POST, PUT y DELETE de solicitud. Al recordar que fueron probados con Postman, son completamente funcionales, integrados con la base de datos, y todos respetan las buenas prácticas de arquitectura y desarrollo. Además, se crearon vistas básicas para permitir ver datos promovidos de cada servicio, y se documentó el ciclo de vida del software relatado completo: desarrollo, implementación y control de versiones a través de Git y Github. Nuestro trabajo sigue siendo trazable y colaborativo. |
| --- |

| **Diagrama de arquitectura de microservicios**  La arquitectura del microservicio de productos sigue el patrón clásico de capas en Spring Boot. A continuación, se describe cada uno de los componentes involucrados:  **Servicio:** Productos  Este es el módulo principal del microservicio, encargado de gestionar todas las operaciones relacionadas con productos como creación, actualización, eliminación y consulta.  **Base de Datos:** db\_crud (MySQL)  Es la base de datos que almacena los registros de productos. Contiene una tabla llamada producto con columnas para id, nombre, descripcion y precio.    **Cliente:** Postman  Se utilizó Postman como herramienta cliente para probar los endpoints REST mediante peticiones HTTP. Las pruebas incluyen operaciones GET, POST, PUT y DELETE sobre la entidad Producto.  **Estructura (flujo de responsabilidad):**  **Controller:** Es la capa que recibe las solicitudes HTTP del cliente. En este caso, la clase ProductoRestController define rutas como /api/productos.    **Service:** Contiene la lógica de negocio. La interfaz ProductoService y su implementación ProductoServiceImpl determinan cómo interactuar con los datos.    **Repository:** Accede directamente a la base de datos. ProductoRepository extiende CrudRepository, lo que permite usar métodos como findAll(), save(), deleteById() sin escribir consultas SQL.    **Entity:** Es la representación del objeto Producto como una clase Java mapeada a la tabla en la base de datos mediante anotaciones JPA como @Entity y @Id. |
| --- |

**Estructura del proyecto**

| El proyecto fue construido siguiendo una arquitectura en capas, común en aplicaciones Spring Boot. A continuación se detalla el rol de cada componente en el microservicio de productos:  **Entity:** Producto.java  Es la clase que representa la tabla producto de la base de datos. Está anotada con @Entity y contiene los atributos id, nombre, descripcion y precio. Cada atributo se corresponde con una columna en la base de datos y se gestiona mediante JPA.  **Repository:** ProductoRepository  Es una interfaz que extiende CrudRepository<Producto, Long>. Gracias a esto, se obtienen por defecto métodos para guardar, buscar, eliminar y listar productos sin necesidad de escribir SQL manual.  **Service:** ProductoService, ProductoServiceImpl  ProductoService es una interfaz que define los métodos que manejarán la lógica del negocio, como findAll, findById, save y deleteById.  ProductoServiceImpl es la clase que implementa esa interfaz. En ella se inyecta el repositorio y se define cómo se aplican las operaciones CRUD. Esta capa permite separar la lógica de negocio del acceso a datos.  **Controller:** ProductoRestController  Es la clase que expone la API REST. Utiliza anotaciones como @RestController y @RequestMapping para definir las rutas accesibles desde el navegador o herramientas como Postman. Incluye los métodos GET, POST, PUT y DELETE, los cuales interactúan con el servicio para manipular productos. |
| --- |

**Implementación de los servicios REST**

| CImplementación de los servicios REST  En este proyecto se implementó una API REST para gestionar productos, siguiendo el enfoque de controladores RESTful proporcionado por Spring Boot. A continuación, se detallan los métodos disponibles y su comportamiento:  GET /api/productos  Recupera todos los productos almacenados en la base de datos. Este método se implementa utilizando @GetMapping y llama al servicio para obtener una lista de objetos Producto.  GET /api/productos/{id}  Recupera un producto específico según su identificador. Si el producto no existe, el servicio puede retornar un error o un objeto vacío, dependiendo de la implementación.  POST /api/productos  Crea un nuevo producto. Se utiliza @PostMapping y se recibe un objeto Producto en formato JSON en el cuerpo de la solicitud. Este objeto es procesado y guardado en la base de datos a través del servicio.  PUT /api/productos/{id}  Actualiza un producto existente. El método @PutMapping recibe tanto el ID del producto como el nuevo contenido JSON, y actualiza los campos correspondientes. El producto modificado se guarda nuevamente en la base de datos.  DELETE /api/productos/{id}  Elimina un producto según su identificador. Se utiliza @DeleteMapping y, tras confirmar que el producto existe, se procede a su eliminación.  Todos estos métodos se integran mediante el controlador ProductoRestController, el cual utiliza anotaciones de Spring como @Autowired para inyectar dependencias, y delega la lógica de negocio al servicio ProductoServiceImpl. Esta división favorece la reutilización del código y la claridad en la estructura del sistema. |
| --- |

| **Base de Datos**  Para esta etapa del proyecto se utilizó el sistema de gestión de bases de datos relacional MySQL, seleccionado por su robustez, compatibilidad con Spring Boot y facilidad de uso. La base de datos se denomina db\_crud y fue configurada para trabajar junto al microservicio de productos.  La estructura principal en esta base de datos es la tabla producto, la cual se define con el siguiente esquema:  CREATE TABLE producto (  id INT AUTO\_INCREMENT PRIMARY KEY,  nombre VARCHAR(255),  descripcion VARCHAR(255),  precio INT  );  **Descripción de campos:**  **id:** clave primaria de tipo entero, autoincrementable. Identifica de forma única cada producto.  **nombre:** campo de texto para almacenar el nombre del producto.  **descripcion:** campo de texto para describir brevemente el producto.  **precio:** valor numérico que representa el precio del producto.  **Datos de ejemplo insertados durante pruebas:**    **Integración con el proyecto:**  En el archivo application.properties, se configuró la conexión con la base de datos, incluyendo la URL del host, nombre de la base (db\_crud), y las credenciales de acceso.  JPA y Hibernate se encargan de mapear automáticamente la clase Producto.java a la tabla producto, utilizando anotaciones como @Entity, @Id, @GeneratedValue.  La propiedad spring.jpa.hibernate.ddl-auto=update garantiza que, si la tabla no existe, será creada automáticamente al ejecutar la aplicación.  Ventajas de la implementación actual:  Permite realizar operaciones CRUD completas directamente sobre la base de datos desde el backend.  La configuración permite una futura migración a motores más complejos (como PostgreSQL o Amazon RDS) sin modificar la lógica de negocio. |
| --- |

| **Archivo pom.xml y dependencias**  El archivo pom.xml es el corazón de cualquier proyecto basado en Maven, ya que define su configuración y las dependencias necesarias para que el proyecto funcione correctamente. En este proyecto, se incluyeron las siguientes dependencias clave:    Explicación de las dependencias:  spring-boot-starter-data-jpa: Habilita el uso de Spring Data JPA, lo que permite interactuar con la base de datos a través de clases Java y repositorios, eliminando la necesidad de escribir SQL manual.  mysql-connector-java: Es el driver JDBC que permite la conexión entre la aplicación y una base de datos MySQL.  spring-boot-starter-web: Habilita la creación de controladores REST, configuración del servidor embebido (Tomcat), manejo de peticiones HTTP, entre otros.  spring-boot-devtools: Herramienta de desarrollo que permite la recarga automática del proyecto cuando se detectan cambios, agilizando el proceso de desarrollo.  Estructura adicional del pom.xml:  El archivo pom.xml también contiene otras secciones importantes:  <groupId>, <artifactId> y <version>: Identifican el proyecto dentro del repositorio Maven.  <properties>: Especifica la versión de Java a utilizar, generalmente 17 o superior.  <build>: Configura el empaquetado del proyecto, el plugin de Spring Boot, y el objetivo de construcción (formato JAR). |
| --- |

| **Uso de Git y GitHub**  Para el control de versiones y trabajo colaborativo se utilizó **Git**, permitiendo gestionar el historial de cambios, integrar nuevas funcionalidades y mantener una copia segura del proyecto. Se creó un repositorio en **GitHub**, donde se almacenó el código fuente completo, incluyendo los microservicios y los scripts de base de datos.  (FOTO )  Comandos utilizados:  git init  git add .  git commit -m "Implementación de servicios REST"  git push origin main  El archivo readme.md del repositorio incluye la descripción del proyecto y las rutas de acceso a los servicios REST implementados:  (FOTO )  Repositorio GitHub:  - Proyecto subido con código fuente y base de datos.  - README.md incluye rutas:  - /api/usuarios  - /api/productos  - /api/ventas  Gracias al uso de Git y GitHub, fue posible trabajar de forma organizada, documentar adecuadamente cada cambio y facilitar la colaboración entre los integrantes del equipo. Además, se garantiza la trazabilidad y disponibilidad del proyecto desde cualquier ubicación. |
| --- |

**conclusión**

| Implementar microservicios en el proyecto Perfulandia SPA ha sido un paso importante para modernizar su sistema. Esto significa que han pasado de un diseño monolítico a una estructura más flexible, escalable y fácil de mantener. En esta segunda etapa, lograron desarrollar con éxito tres microservicios funcionales: usuarios, productos y ventas. Cada uno cuenta con operaciones completas de creación, lectura, actualización y eliminación (CRUD), y están conectados a una base de datos MySQL. Además, probaron todo con herramientas como Postman para asegurarse de que funcionan correctamente. En particular, el microservicio de 'producto' se construyó usando tecnologías como Spring Boot, Spring Data JPA y Maven. Se siguieron buenas prácticas de desarrollo en backend para crear un diseño claro, reutilizable y preparado para futuras mejoras. Gracias a pruebas exhaustivas en Postman y una configuración sólida de la base de datos, podemos confiar en que funcionará bien en distintos entornos. Durante este proceso, también se establecieron prácticas importantes como el control de versiones con Git, el trabajo en equipo usando GitHub, y la documentación del proceso. Además, se crearon vistas básicas para facilitar la interacción con los servicios, lo que permite una forma sencilla de visualizar los datos que ofrecen las APIs. |
| --- |