

**Índice de Contenidos**

**Información del proyecto**

Datos

|  | Empresa / Organización | Perfulandia |
| --- | --- | --- |
| Fecha de inicio/fin | 16/03/2025 - 04/07/2025 |
| 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 |

**1.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. |
| --- |

**2. Justificación del proyecto – Contexto**

| En el contexto de la Experiencia 3, nos enfocamos no solo en consolidar esta arquitectura, sino también en asegurar su calidad y estabilidad mediante la implementación de controladores REST, el desarrollo de pruebas unitarias e integración utilizando herramientas como JUnit y Spring Boot Test, y la documentación estandarizada con OpenAPI Specification (OAS). Además, aplicamos buenas prácticas de desarrollo, control de versiones con Git y GitHub, y una estructuración adecuada del proyecto.  Este trabajo busca entregar una solución robusta, confiable y bien documentada, que responda a los desafíos tecnológicos de Perfulandia SPA, asegurando continuidad operativa, escalabilidad futura y una mejor experiencia para sus usuarios. |
| --- |

**3. Objetivos Específicos del Proyecto**

| El presente proyecto tiene como propósito lograr una transformación tecnológica efectiva mediante la implementación de microservicios. Para ello, se definieron los siguientes objetivos específicos:   1. Diseñar una arquitectura de microservicios modular que permita mejorar la escalabilidad del sistema. 2. Implementar servicios RESTful utilizando Spring Boot, asegurando el cumplimiento de buenas prácticas de desarrollo. 3. Aplicar pruebas unitarias e integración con JUnit 5, Mockito y MockMvc para validar el correcto funcionamiento del sistema. 4. Documentar los endpoints de la API utilizando la especificación OpenAPI (Swagger UI) para facilitar su comprensión y uso. 5. Implementar control de versiones utilizando Git y GitHub para una colaboración eficiente en equipo. |
| --- |

**4. Arquitectura de Microservicios Proyecto Perfulandia**

| El diagrama presentado describe la arquitectura moderna adoptada por el proyecto **Perfulandia**, basada en microservicios, herramientas cloud y prácticas DevOps. Esta estructura fue diseñada para garantizar escalabilidad, seguridad, modularidad y eficiencia operativa.  **3.1 diagrama de microservicios**    **4.2 Autenticación y Entrada**  **Cliente Web (Navegador o App):** Punto de acceso inicial del usuario.  **Amazon Cognito:** Servicio de autenticación gestionado por AWS, que se encarga de validar y gestionar identidades de usuario antes de acceder a los servicios internos.  **4.3 Backend y Microservicios**  **Spring Boot:** Framework principal para la implementación de la lógica de negocio. Desde aquí se orquesta el enrutamiento hacia los microservicios especializados.  **4.4 Los microservicios están divididos según responsabilidad funcional:**   * Producto Microservicio * Cliente Microservicio * Venta Microservicio * Sucursal Microservicio * Usuarios/Roles Microservicio   Cada microservicio gestiona su propia lógica y tiene acceso independiente a su base de datos, lo cual sigue el principio de database per service.  **4.5 Persistencia**  Cada microservicio se conecta a su propia base de datos MySQL, desplegada en la nube mediante Amazon RDS, asegurando almacenamiento confiable y escalable.  **4.6 DevOps y Observabilidad**  **CI/CD Pipeline (Jenkins / GitHub Actions):** Automatiza la integración y el despliegue continuo de los microservicios.  **Kubernetes (EKS):** Plataforma de orquestación utilizada para gestionar el despliegue, escalado y disponibilidad de los microservicios.  **Monitoreo & Logs (Grafana + CloudWatch):** Herramientas implementadas para monitorear métricas. visualizar logs y detectar errores o caídas de servicio en tiempo real. |
| --- |

**5. Plan de pruebas:**

| Para garantizar la calidad del software desarrollado en el proyecto **Perfulandia**, se utilizaron diversas herramientas y frameworks especializados en la creación y ejecución de pruebas. Estas herramientas nos permitieron implementar pruebas unitarias, pruebas de integración y documentación automatizada de los servicios REST.    Además de seleccionar cuidadosamente las herramientas, se implementaron pruebas unitarias en los servicios clave del proyecto mediante el uso de @Mock y @InjectMocks de Mockito, lo que permitió aislar la lógica de negocio de las dependencias externas. Estas pruebas se ejecutaron utilizando el comando mvn test, asegurando así la correcta ejecución de cada caso definido.  **5.1 Evidencia de Éxito en la Ejecución de Pruebas del Proyecto**  aquí podemos ver que todas las pruebas de proyecto funcionan correctamente, sin errores lo que indican lo distintos testeos de los microservicios del software backend, en este caso los rest |
| --- |

**6. Pruebas Unitarias.**

| Las pruebas unitarias permiten verificar el correcto funcionamiento de los componentes individuales del sistema, como los controladores REST. En este proyecto, se implementaron pruebas para validar operaciones CRUD sobre entidades clave: Usuario, Producto, Pedido, Sucursal y Gerente. Se utilizaron frameworks como JUnit 5, Mockito y MockMvc integrados en Spring Boot.  **6.1 tabla resumen de las pruebas**    Las pruebas unitarias implementadas cubren las funcionalidades principales del sistema, asegurando que los controladores REST manejen correctamente las operaciones más críticas. Esto contribuye directamente a la confiabilidad del sistema y facilita el mantenimiento futuro del código.  **6.2 ejemplos de prueba de unitaria**  **Prueba:** findAllTest del Servicio de Usuario  Esta prueba unitaria valida el correcto funcionamiento del método findAll() del servicio UsuarioServiceImpl, que es responsable de retornar todos los usuarios registrados en la base de datos (simulada).    **Prueba:** productoNoExisteTest  Esta prueba verifica que el servicio responda con un HTTP 404 (Not Found) cuando se intenta buscar un producto por ID y este no existe.  Se usa Mockito para simular que productoService.findById(10L) retorna Optional.empty() (es decir, no se encontró el producto). Luego se realiza una llamada simulada con MockMvc al endpoint /api/productos/10, y se valida que la respuesta tenga el estado 404.    **Prueba:** gerenteNoExisteTest  Esta prueba verifica el comportamiento del sistema cuando se hace una solicitud GET a un gerente con un ID que no está registrado. Se espera que el sistema devuelva un código HTTP 404 (Not Found).    **Prueba:** crearPedidoTest  Esta prueba unitaria valida que el sistema es capaz de crear un nuevo pedido correctamente al recibir una solicitud HTTP POST con los datos correspondientes. |
| --- |

**7. Pruebas de integración**

| Las pruebas de integración en este proyecto tienen como objetivo validar que los distintos componentes de la aplicación (controladores, servicios y repositorios simulados) trabajen de forma conjunta y correcta a través los endpoints REST.  Estas pruebas se implementaron utilizando Spring Boot Test junto con MockMvc, lo que permite simular peticiones HTTP reales sin necesidad de levantar un servidor completo. También se utilizó Mockito para simular respuestas de los servicios, permitiendo aislar las dependencias externas como la base de datos.  **7.1 ejemplos de prueba de integración**  **Prueba:** integración Listar gerentes  Esta prueba simula una solicitud GET al endpoint /api/gerentes y verifica que la respuesta sea exitosa (HTTP 200 OK).  Utilizando MockMvc, se valida que el controlador responda correctamente al recibir una lista simulada de gerentes desde el servicio.    **Prueba:** integración Crear usuario  Esta prueba valida que el endpoint /api/usuarios permita la creación correcta de un nuevo usuario. Utiliza MockMvc para enviar un objeto Usuario como JSON en una solicitud POST.  El servicio simulado (usuarioService) responde con un objeto Usuario ya guardado, y se espera que el controlador devuelva un estado HTTP 201 (Created) como confirmación.    **Prueba:** integración Ver una sucursal  Esta prueba de integración simula una solicitud HTTP GET al endpoint /api/sucursales/1 para consultar una sucursal por su ID. Se espera una respuesta exitosa (HTTP 200 OK) cuando el servicio retorna un objeto válido.  La prueba se apoya en MockMvc para generar la solicitud y en Mockito para simular el comportamiento del servicio sucursalService. El test valida la correcta integración entre el controlador y la lógica de negocio. |
| --- |

**8. Ejecución de Pruebas con Apoyo de Documentación OpenAPI (OAS):**

| En esta ocasión, la ejecución de pruebas se complementará con el uso de OpenAPI (OAS), lo cual permite documentar y validar visualmente los endpoints desarrollados. Esto proporciona una vista clara y estandarizada de la API, facilitando tanto el entendimiento del sistema como la verificación de su correcto comportamiento.  Las pruebas serán ejecutadas desde el entorno de desarrollo (IDE), utilizando el motor JUnit 5 junto con Mockito y MockMvc, para validar tanto pruebas unitarias como de integración. Además de los resultados mostrados en el entorno de pruebas, se utilizará SpringDoc OpenAPI para explorar en tiempo real los endpoints mediante la interfaz Swagger. |
| --- |

**9. Documentación de las pruebas ejecutadas ( imágenes ).**

| **9.1 Ejecución de Pruebas con Apoyo de Documentación OpenAPI (OAS)**  La imagen evidencia el uso de Swagger UI, una herramienta basada en OpenAPI Specification (OAS), para documentar, visualizar y probar de forma interactiva los endpoints de una API REST. En particular, se prueba el endpoint:    Esto demuestra que:  Se utiliza la documentación generada automáticamente por OpenAPI para entender y ejecutar el servicio.  Se prueba de forma directa el funcionamiento del endpoint, validando que responda con un código HTTP 200 y una lista de usuarios en formato application/json.  En el desarrollo del sistema, se utilizó la herramienta Swagger UI, respaldada por la especificación OpenAPI (OAS), para ejecutar y validar diversos endpoints de la API de manera interactiva.    Esto demuestra que:  Se implementó y documentó correctamente la API usando la especificación OpenAPI (OAS), permitiendo generar una interfaz interactiva (Swagger UI) para su exploración.  En el desarrollo del sistema, se utilizó la herramienta Swagger UI, respaldada por la especificación OpenAPI (OAS), para ejecutar y validar diversos endpoints de la API de manera interactiva.  d  Esto demuestra que:  La API está correctamente documentada usando la especificación OpenAPI (OAS), permitiendo la generación automática de una interfaz interactiva con Swagger UI.  Se realizaron pruebas efectivas y exitosas de distintos endpoints (/api/usuarios, /api/gerentes, /api/productos), obteniendo respuestas estructuradas en formato JSON y código de estado 200 OK.  **9.2 Manejo de Errores en la API REST (DELETE)**  Se realizó una prueba del endpoint:  DELETE /api/productos/1 — Elimina un producto por ID.  Al intentar eliminar un producto inexistente (ID 1), la API respondió correctamente con un código de error HTTP 404 (Not Found).    Esto demuestra que:  La API maneja correctamente los errores comunes, como intentos de eliminar un recurso inexistente, respondiendo con el código HTTP adecuado (404 Not Found).  Se siguen las convenciones de buenas prácticas en REST, ya que se informa claramente al cliente cuando una operación no puede completarse por la ausencia del recurso solicitado.  **9.3 Evidencia de Consulta Específica con OpenAPI (OAS)**  Se utilizó Swagger UI para ejecutar el endpoint GET /api/productos/{id} con el parámetro id = 2, obteniendo los detalles de un producto específico.    Esto demuestra que:  El endpoint para consulta por ID funciona correctamente, respondiendo con el código adecuado y la estructura de datos esperada.  La API permite obtener recursos específicos de manera controlada, lo que es esencial para operaciones de detalle, edición o verificación.  **9.4 Evidencia de Creación de Recursos con OpenAPI (OAS)**  Se utilizó Swagger UI para ejecutar el endpoint POST /api/productos, enviando un cuerpo en formato JSON con los datos de un nuevo producto.    Esto demuestra que:  La API permite la creación dinámica de recursos (productos) mediante el método POST, aceptando datos en formato application/json.  El contrato definido en OpenAPI se respeta correctamente, ya que se validan tanto la entrada como la salida del recurso. |
| --- |

**10. Consideraciones Éticas**

| Durante el desarrollo del proyecto se tomaron en cuenta varios enfoques éticos relevantes para el diseño de software:   1. **Privacidad de los datos:** Se consideró la necesidad de proteger los datos personales del usuario mediante mecanismos de autenticación y autorización robustos (como Amazon Cognito). 2. **Transparencia:** La documentación mediante OpenAPI proporciona una interfaz clara y accesible que permite a los usuarios comprender cómo se utilizan sus datos. 3. **Responsabilidad:** Se implementaron pruebas para garantizar el correcto comportamiento del sistema, minimizando errores que puedan afectar negativamente a los usuarios. 4. **Seguridad:** El diseño considera buenas prácticas de seguridad en la gestión de identidades y control de acceso a los servicios.   10.1 Propuestas de Mejora Ética   1. Implementar el uso de tokens con expiración y renovación automática para mayor seguridad en sesiones activas. 2. Añadir capas de encriptación de datos sensibles en tránsito y en reposo. 3. Incluir un mecanismo de trazabilidad que registre el acceso y uso de datos personales para reforzar la transparencia. 4. Ampliar la documentación del sistema incluyendo políticas de uso y tratamiento de datos desde la perspectiva del usuario final. |
| --- |

**11. Uso de Git y GitHub**

| bueno en el control de versiones se hizo un trabajo colaborativo, lo que permitió gestionar el historial de cambios, y a si poder integrar nuevas funcionalidades, y mantener una copia segura del proyecto. Se creó un repositorio en Github, donde se almaceno el código fuente completo, incluyendo los microservicios y los scripts de la base de datos ya creada.  Comandos que se utilizaron  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  Proyecto subido con código fuente y base de datos.  README.md incluye rutas:  /api/usuarios  /api/productos  /api/ventas.  Gracias a esto, fue posible trabajar de forma organizada, documentar adecuadamente cada cambio y facilitar la colaboración entre los integrantes del equipo |
| --- |

**12. Recomendaciones para Futuros Proyectos**

| Esto es lo que pensamos en agregar o tomar más en cuenta para futuros proyectos ya que en este igual tuvimos algunos errores al inicio pero luego con el tiempo pudimos arreglar de una o otra forma. lo bueno que todos aprendimos y estamos enfocados en realizar este proyecto esto nos ayudó bastante a poder encontrar la solución a los problemas que teníamos sobre todo al inicio. este listado es más que nada para poder ver qué podemos mejorar para futuros proyectos .   1. **Planificar con mayor detalle desde el inicio**, incluyendo tiempos, responsables y objetivos claros para cada etapa. 2. **Establecer roles definidos desde el comienzo** (Scrum Master, DevOps, Backend, etc.) para evitar confusión en las tareas. 3. **Dedicar más tiempo a la fase de diseño de arquitectura**, validando que todas las dependencias y servicios estén bien conectados antes de implementar. 4. **Realizar reuniones de seguimiento más frecuentes** para detectar errores tempranos y corregirlos a tiempo. |
| --- |

**conclusión**

| El desarrollo del proyecto *Perfulandia* representó un proceso de transformación tecnológica significativo, en el cual se migró desde una arquitectura monolítica hacia un enfoque basado en microservicios. Esta transición permitió mejorar la escalabilidad, flexibilidad y mantenibilidad del sistema, alineándose con las exigencias actuales de desarrollo moderno.  A lo largo del proyecto, se aplicaron buenas prácticas en programación, pruebas y documentación. El uso de herramientas como Spring Boot, JUnit 5, Mockito, GitHub y Swagger permitió garantizar la calidad técnica y funcional del sistema. Además, se validó la correcta integración entre componentes mediante pruebas unitarias y de integración, asegurando así la confiabilidad del software.  El trabajo colaborativo del equipo fue clave para superar los desafíos iniciales. A través de una constante revisión, aprendizaje y mejora continua, se logró construir una solución robusta y bien documentada, que cumple con los objetivos planteados. Este proyecto no solo fortaleció las competencias técnicas del equipo, sino que también potenció habilidades de trabajo en equipo, resolución de problemas y toma de decisiones éticas. |
| --- |