Universidad de la Sabana

Sistema Gestion Inmobiliaria

Proyecto final

Autor:
Harold Steven Vargas Henao
Diego Fernando Ramirez Tenjo
David Eduardo Lopez Jimenez

Profesor(a):
Cesar Augusto Vega Fernandez

Curso:
Ingeniería Informática

Fecha:
18 de Noviembre del 2024

Departamento de Ingeniería

1 Resumen Ejecutivo

1.1 Breve Descripción del Proyecto

El proyecto denominado *Sistema de Gestión Inmobiliaria* es una aplicación diseñada para gestionar propiedades inmobiliarias, pagos, contratos de arrendamiento y usuarios. Utiliza tecnologías como **Spring Boot** para el desarrollo de la aplicación backend, **Jenkins** para la integración continua (CI/CD), y **Docker** para la contenerización y despliegue. La aplicación incluye características como el manejo de pagos mediante diferentes métodos, la gestión de propiedades, y el seguimiento de contratos de arrendamiento.

1.2 Objetivos Alcanzados

Los principales objetivos alcanzados en el proyecto incluyen:

- Desarrollo y despliegue de un sistema completo de gestión inmobiliaria utilizando tecnologías modernas como **Spring Boot**, **Docker**, y **Jenkins**.
- Implementación de pruebas unitarias y de integración, lo que garantiza la calidad y el rendimiento del sistema.
- Implementación de un pipeline de integración continua (CI/CD) que automatiza el proceso de compilación, pruebas y despliegue.
- Generación de informes de calidad de código y pruebas a través de **SonarQube** y **Allure**.
- Despliegue del sistema en contenedores **Docker** para facilitar su escalabilidad y portabilidad.
- Realización de pruebas de carga utilizando **JMeter** para evaluar el rendimiento del sistema bajo condiciones de uso intenso.

1.3 Soluciones Implementadas

Durante el desarrollo del proyecto se implementaron varias soluciones para asegurar la eficiencia, escalabilidad y calidad del sistema:

- **Integración Continua y Entrega Continua (CI/CD)**: Se configuró un pipeline en Jenkins que incluye la ejecución de pruebas unitarias, análisis de calidad con SonarQube, y despliegue automático de la aplicación en Docker. El proceso fue optimizado para garantizar tiempos de ejecución rápidos, con un tiempo promedio de compilación de 18 minutos.
- **Contenerización Docker**: La aplicación fue contenerizada utilizando **Docker**, lo que permite que sea fácilmente desplegada en diferentes entornos sin importar las configuraciones del sistema operativo subyacente.
- **SonarQube y Allure para Análisis de Calidad y Reportes**: Se integró **SonarQube** en el pipeline para análisis de código estático y calidad, mientras que se utilizaron los informes generados por **Allure** para presentar los resultados de las pruebas de manera visual y detallada.
- **Pruebas Unitarias y Autónomas**: El proyecto incluye una suite de pruebas unitarias y autónomas para asegurar el correcto funcionamiento de los servicios del sistema. Estas pruebas están implementadas con **JUnit 5** y **Mockito**. Entre las pruebas más destacadas se encuentran:
  - `LeaseServiceTest.java`
  - `NotificationServiceTest.java`
  - `PaymentServiceTest.java`
  - `PropertyServiceTest.java`

  Además, se implementaron pruebas autónomas como `LeaseServiceAutonomousTest.java`, `PaymentServiceAutonomousTest.java`, entre otras, para validar el comportamiento integral de la aplicación bajo condiciones específicas.
- **Pruebas de Carga con JMeter**: Se llevaron a cabo pruebas de carga utilizando **JMeter**, una herramienta de código abierto para pruebas de rendimiento. Estas pruebas permitieron verificar la capacidad del sistema para manejar una gran cantidad de usuarios concurrentes y medir el rendimiento de las respuestas en diferentes condiciones de carga.

1.4 Resultados y Logros Principales

Los resultados y logros alcanzados en el proyecto son los siguientes:

- Implementación exitosa del sistema de gestión inmobiliaria con una arquitectura moderna, utilizando **Spring Boot**, **Docker** y **Jenkins**.
- La automatización de las pruebas y el despliegue permite un ciclo de desarrollo ágil y eficiente.
- El análisis de calidad de código realizado con **SonarQube** ha garantizado la alta calidad y mantenibilidad del código.

- Los informes generados por **Allure** proporcionan un seguimiento detallado de las pruebas y ayudan en la toma de decisiones durante el proceso de desarrollo.

- Las pruebas de carga realizadas con **JMeter** demostraron que el sistema puede manejar múltiples usuarios concurrentes sin afectar el rendimiento.

2 Introducción

2.1 Contexto del Proyecto

El *Sistema de Gestión Inmobiliaria* surge como respuesta a la necesidad de una solución tecnológica para gestionar de manera eficiente los diferentes aspectos relacionados con la administración de propiedades inmobiliarias, pagos y contratos. Los procesos tradicionales en la gestión inmobiliaria son a menudo manuales, lo que implica un alto riesgo de errores, falta de visibilidad y un elevado tiempo de respuesta ante cualquier eventualidad. Además, el crecimiento de la industria inmobiliaria ha incrementado la complejidad de la gestión, haciendo indispensable contar con herramientas tecnológicas que simplifiquen las operaciones diarias.

El sistema propuesto busca resolver estos problemas mediante la automatización de la gestión de propiedades, contratos de arrendamiento, pagos y notificaciones. El uso de tecnologías como **Spring Boot**, **Docker**, **Jenkins**, y **SonarQube** facilita una plataforma robusta, escalable y eficiente que puede manejar grandes volúmenes de datos y solicitudes simultáneas. El enfoque adoptado es modular, lo que permite una fácil extensión y mantenimiento del sistema a medida que crece la demanda.

2.2 Objetivo del Proyecto

El objetivo principal del proyecto es desarrollar un sistema integral que permita gestionar todas las operaciones relacionadas con la administración inmobiliaria de manera eficiente y automatizada. El sistema debe ser capaz de gestionar propiedades, contratos de arrendamiento, pagos, y comunicaciones con los usuarios de manera efectiva.

Este objetivo se alinea con las fases anteriores del proyecto, que incluyeron el diseño de la arquitectura, la implementación de las funcionalidades clave, y la creación de un pipeline de integración continua que automatiza los procesos de pruebas y despliegue. Además, el sistema debe ser altamente confiable, escalable y fácil de mantener, lo que ha sido posible gracias al uso de las mejores prácticas en desarrollo de software, pruebas automatizadas y análisis de calidad de código.

2.3 Alcance del Proyecto

El alcance del proyecto incluye las siguientes funcionalidades principales:

- **Gestión de propiedades**: El sistema permitirá la creación, edición, y eliminación de propiedades, así como su asignación a los usuarios que las gestionan.

- **Gestión de contratos de arrendamiento**: Los usuarios podrán crear, editar y eliminar contratos de arrendamiento, con seguimiento automático de fechas y pagos asociados.

- **Gestión de pagos**: El sistema integrará diferentes métodos de pago y realizará el seguimiento de los pagos de los inquilinos, notificando a los usuarios de vencimientos o pagos pendientes.

- **Notificaciones**: El sistema enviará notificaciones automáticas a los usuarios sobre el estado de sus pagos, contratos y otros eventos importantes.

- **Pruebas de carga y rendimiento**: Se incluirán pruebas de carga con **JMeter** para asegurar que el sistema puede manejar múltiples usuarios concurrentes sin degradación en el rendimiento.

El proyecto se enfocará en la implementación y automatización de las funcionalidades descritas, así como en el desarrollo de un sistema robusto que sea capaz de adaptarse a futuros requerimientos. Sin embargo, se establece un límite en cuanto a la integración de otras funcionalidades que no sean esenciales para el funcionamiento básico de la plataforma. Por lo tanto, funcionalidades adicionales, como la integración con sistemas externos o nuevas características que no sean críticas en esta etapa, no se abordarán en este proyecto.

2.4 Límites y Objetivos Alcanzables

El proyecto ha sido diseñado para cumplir con los siguientes objetivos alcanzables:

- Desarrollar un sistema funcional de gestión inmobiliaria con las capacidades de gestión de propiedades, contratos, pagos y notificaciones.

- Asegurar que el sistema sea capaz de manejar un volumen moderado de usuarios concurrentes mediante pruebas de carga realizadas con **JMeter**.
- Implementar un sistema de integración continua para pruebas y despliegue automatizados a través de **Jenkins**.
- Realizar pruebas unitarias y autónomas para garantizar la estabilidad y confiabilidad del sistema.

El proyecto no incluye la integración con sistemas de terceros, ni el desarrollo de funcionalidades avanzadas de análisis o inteligencia de negocios, ya que estas tareas exceden el alcance de la fase actual y se planean para futuras etapas del proyecto.

3 Arquitectura y Diseño

3.1 Arquitectura Seleccionada

El sistema ha sido desarrollado utilizando una **arquitectura por capas**, que permite la separación de responsabilidades en diferentes módulos, garantizando un alto nivel de modularidad y mantenibilidad. Esta arquitectura se ha elegido debido a su simplicidad, flexibilidad y facilidad para adaptarse a futuros cambios o expansiones del sistema.

La arquitectura por capas permite estructurar el código de la siguiente manera:

- **Capa de presentación**: Maneja la interacción con el usuario a través de controladores web (por ejemplo, `HomeController.java`, `LeaseController.java`) y las vistas en HTML.
- **Capa de negocio**: Contiene la lógica del sistema, que se implementa a través de servicios como `LeaseService.java`, `NotificationService.java`, `PaymentService.java`, etc.
- **Capa de acceso a datos**: Es responsable de la interacción con la base de datos, a través de repositorios como `LeaseRepository.java`, `UserRepository.java`, etc.
- **Capa de persistencia**: Contiene las entidades que representan los objetos de negocio persistidos en la base de datos, como `Lease.java`, `User.java`, etc.
- **Capa de excepciones**: Maneja todas las excepciones que pueden surgir en el sistema, como `AuthenticationException.java` o `InvalidLeaseException.java`.

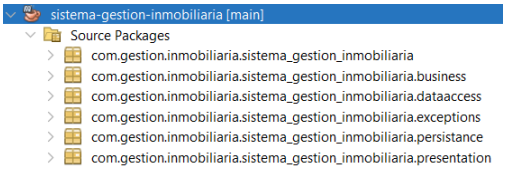

Figure 1: Arquitectura por capas.

La elección de esta arquitectura tiene como propósito garantizar la escalabilidad y la facilidad de mantenimiento del sistema, al permitir que diferentes capas puedan modificarse o extenderse sin afectar otras partes del sistema.

3.2 Principios de Diseño Aplicados

Para garantizar una estructura robusta y fácil de mantener, se han aplicado los siguientes principios de diseño:

- **Modularidad**: El sistema se organiza en módulos que gestionan funciones específicas, lo que facilita la implementación, pruebas y mantenimiento de nuevas funcionalidades.
- **Desacoplamiento**: Las diferentes capas del sistema están desacopladas, lo que significa que cambios en una capa no afectarán otras. Por ejemplo, los servicios de negocio no dependen directamente de la base de datos, sino que se comunican a través de interfaces.
- **Responsabilidad Única**: Cada clase tiene una responsabilidad claramente definida. Esto hace que el código sea más fácil de entender y de modificar sin introducir errores en otras áreas del sistema.
- **Patrón de Estrategia**: Se ha implementado el patrón de estrategia en el sistema de pagos, permitiendo la integración de diferentes métodos de pago (por ejemplo, PayPal, tarjeta de crédito, transferencia bancaria) de forma independiente y extensible.
- **Inyección de Dependencias**: Se usa inyección de dependencias (a través de `Spring`) para garantizar que las dependencias de las clases se gestionen de forma centralizada y desacoplada.

3.3 Diagrama de la Arquitectura

A continuación se presenta un esquema de la arquitectura por capas adoptada en el proyecto, mostrando las principales partes del sistema y cómo interactúan entre sí:

Figure 2: Diagrama de Arquitectura del Sistema de Gestión Inmobiliaria

3.4 Herramientas Utilizadas

Las herramientas y tecnologías utilizadas en el desarrollo del sistema incluyen:

- **Lenguaje de programación**: Java.
- **Framework**: Spring Boot, que facilita la creación de aplicaciones robustas y escalables.
- **Base de datos**: MySQL para la persistencia de datos.
- **Contenedores**: Docker, utilizado para crear contenedores de la aplicación, permitiendo un despliegue más eficiente y consistente.
- **Gestión de dependencias**: Maven, para la gestión de las dependencias del proyecto.
- **Pruebas**: JUnit y TestNG para realizar pruebas unitarias y pruebas autónomas.
- **Integración continua**: Jenkins, para la automatización de los procesos de integración y despliegue continuo.
- **Cobertura de código**: SonarQube, para el análisis de calidad del código.
- **Pruebas de carga**: JMeter, utilizado para realizar pruebas de carga y garantizar el rendimiento del sistema bajo diversas condiciones de tráfico.

3.5 Estructura del Proyecto

La estructura del proyecto está organizada de la siguiente manera:

- `.gitignore`: Archivo de configuración para ignorar archivos no deseados en el control de versiones.
- `Jenkinsfile`: Archivo de configuración para la integración continua y despliegue automatizado mediante Jenkins.
- `sistema-gestion-inmobiliaria/Backend`: Contiene el código fuente del backend del sistema.
- `src/main/java`: Contiene las clases Java, organizadas en paquetes de acuerdo con la capa que representan.
- `src/main/resources`: Contiene archivos de configuración como `application.properties` y los recursos estáticos y plantillas.
- `test/java`: Contiene las pruebas unitarias y autónomas del sistema.

4 Desarrollo del Software

4.1 Descripción de la Implementación

El desarrollo del sistema se centró en implementar una solución robusta para la gestión inmobiliaria. Cada parte del sistema fue desarrollada siguiendo los principios de la arquitectura por capas mencionados en la sección anterior, con un enfoque en la escalabilidad, mantenibilidad y separación de responsabilidades. A continuación se describen las funcionalidades implementadas y las decisiones clave tomadas durante el desarrollo:

- **Interfaz de Usuario**: Se desarrollaron diversas vistas utilizando HTML, CSS y JavaScript, siguiendo un diseño simple pero eficiente. Se utilizaron formularios para gestionar propiedades, contratos de alquiler, pagos y usuarios, con validaciones tanto del lado del cliente como del servidor.
- **Lógica de Negocio**: Los servicios de negocio, como `LeaseService`, `PaymentService` y `UserService`, implementan la lógica principal del sistema. Estos servicios gestionan operaciones como la creación de nuevos contratos de arrendamiento, el procesamiento de pagos, la notificación a los usuarios, y más.
- **Persistencia de Datos**: Se implementaron clases de entidad en la capa de persistencia para representar objetos como `User`, `Lease`, `Payment`, y `Property`. Además, los repositorios asociados gestionan las interacciones con la base de datos.
- **Excepciones y Manejo de Errores**: El sistema incluye un robusto sistema de manejo de excepciones que permite capturar errores comunes como la autenticación fallida o contratos de arrendamiento inválidos, proporcionando mensajes claros al usuario.
- **Autenticación y Autorización**: Para garantizar que solo los usuarios autorizados puedan acceder a ciertas funcionalidades, se implementó un sistema de autenticación con roles de usuario (por ejemplo, administrador, arrendatario, propietario), utilizando Spring Security.

El desarrollo de cada módulo se realizó de manera iterativa, con pruebas continuas para garantizar la calidad del código y la correcta integración entre las distintas partes del sistema.

4.2 Módulos o Componentes Principales

El sistema está compuesto por varios módulos o componentes clave, que se describen a continuación:

- **Capa de Presentación**: Esta capa gestiona la interfaz de usuario del sistema, que se desarrolla utilizando HTML, CSS y JavaScript. Los controladores de Spring MVC, como `LeaseController` y `UserController`, procesan las solicitudes del usuario y devuelven vistas dinámicas que interactúan con el backend.
- **Capa de Lógica de Negocio**: Esta capa contiene las clases responsables de la implementación de la lógica del sistema, como `LeaseService`, `PaymentService`, `NotificationService`, y `UserService`. Aquí se realizan las operaciones de negocio como la creación de propiedades, el cálculo de pagos, el procesamiento de notificaciones, etc.
- **Capa de Acceso a Datos**: En esta capa se encuentran los repositorios como `LeaseRepository` y `UserRepository`, que gestionan la interacción con la base de datos, usando Spring Data JPA para realizar consultas a la base de datos de forma eficiente.
- **Capa de Persistencia**: Aquí se definen las entidades como `User`, `Lease`, `Property`, y `Payment`, que representan las tablas en la base de datos y están mapeadas utilizando JPA (Java Persistence API).
- **Capa de Excepciones**: Contiene clases de excepciones personalizadas, como `UserAlreadyExistsException` y `LeaseNotFoundException`, que son lanzadas para gestionar situaciones anómalas dentro de las operaciones del sistema.

4.3 Integración de Servicios

El sistema realiza la integración con varios servicios externos para ofrecer funcionalidades adicionales:

- **Servicios de Pago**: Se integraron APIs de pago externas como PayPal, tarjetas de crédito y transferencias bancarias. Esto se logró mediante el patrón de estrategia, donde cada tipo de pago (PayPal, tarjeta de crédito, etc.) es gestionado por una clase concreta que implementa una interfaz común (`PaymentStrategy`).
- **Servicios de Notificación**: Se utilizaron servicios de notificación externos para enviar alertas por correo electrónico a los usuarios sobre eventos importantes como la creación de nuevos contratos o el vencimiento de pagos. Esto se integró a través de la clase `NotificationService`.
- **Autenticación de Usuario**: Se integró un sistema de autenticación basado en tokens (JWT) para gestionar el acceso de los usuarios, garantizando que las sesiones sean seguras y persistentes.

La integración de estos servicios se realizó de manera que el sistema pudiera funcionar de forma independiente de las APIs externas, lo que facilita la futura extensión o cambio de proveedores sin afectar la lógica principal del sistema.

4.4 Desafíos y Soluciones Implementadas

Durante el desarrollo del sistema, se presentaron varios desafíos que se resolvieron con las siguientes soluciones:

- **Desafío de Gestión de Pagos Múltiples**: Uno de los desafíos principales fue la integración de diferentes métodos de pago (PayPal, tarjeta de crédito, etc.), que requerían distintos mecanismos de comunicación y validación. La solución fue la implementación del patrón de estrategia, lo que permitió una integración modular de los diferentes métodos de pago sin afectar el resto del sistema.
- **Problemas de Sincronización de Datos**: Durante el proceso de integración de datos de usuarios y propiedades, se identificaron algunos problemas de sincronización entre el frontend y la base de datos. Para resolverlo, se implementaron validaciones tanto en el cliente como en el servidor, garantizando que los datos enviados sean correctos y consistentes.
- **Manejo de Errores**: En los primeros ciclos de desarrollo, se identificaron múltiples puntos de fallo debido a la falta de un manejo centralizado de excepciones. Para solucionarlo, se implementó un sistema de manejo global de ex-

cepciones utilizando las herramientas proporcionadas por Spring, asegurando que los errores se manejen de manera adecuada y se envíen mensajes claros a los usuarios.

- **Optimización de Consultas a la Base de Datos**: Algunas consultas complejas a la base de datos resultaron en tiempos de respuesta lentos, lo que afectó la experiencia del usuario. La solución fue la optimización de consultas utilizando `JPA Query Methods` y `Spring Data JPA` para mejorar el rendimiento de las consultas frecuentes.

Estas soluciones permitieron mejorar la eficiencia y la confiabilidad del sistema, contribuyendo a su estabilidad y rendimiento en producción.

5 Pruebas

5.1 Pruebas Unitarias

Para garantizar la correcta funcionalidad de cada componente del sistema, se implementaron pruebas unitarias utilizando `JUnit` y `Mockito`. Las pruebas unitarias son esenciales para verificar que las funciones individuales de la lógica de negocio y los repositorios operan correctamente.

- **Herramientas Utilizadas**:
  - `JUnit 5`: Para la ejecución de las pruebas unitarias.
  - `Mockito`: Para la simulación de dependencias y la creación de objetos simulados (mocks) que permiten probar unidades específicas de código.
  - `AssertJ`: Para mejorar las aserciones y hacerlas más legibles.
- **Cobertura de las Pruebas**: La cobertura de las pruebas se centra en las clases de servicios (por ejemplo, `LeaseService`, `UserService`) y en las clases de repositorio, verificando que los métodos de negocio y las interacciones con la base de datos se ejecuten correctamente.
- **Ejemplos de Casos de Prueba**:
  - `testCreateLease()`: Verifica que la creación de un contrato de arrendamiento con datos válidos se procese correctamente.
  - `testProcessPayment()` : Verifica que el procesamiento de un pago mediante una tarjeta de crédito se ejecute correctamente.
  - `testGetUserById()`: Comprueba que la recuperación de un usuario a partir de su ID se realice sin errores.

5.2 Pruebas Autónomas

Las pruebas autónomas se realizaron simulando las dependencias externas utilizando `Mockito` para crear objetos simulados de los servicios externos como sistemas de pago o notificaciones. De esta manera, se pudo realizar una integración eficiente sin la necesidad de depender de los servicios reales durante las pruebas.

- **Simulación de Dependencias Externas**: Se simularon servicios como `PaymentService` y `NotificationService`, proporcionando respuestas controladas que permiten probar la lógica de negocio sin realizar llamadas reales a los servicios de pago o de notificación.
- **Pruebas de Integración Simuladas**: Se realizaron pruebas de integración donde se verificó que el sistema maneja correctamente las respuestas simuladas de estos servicios externos.
- **Casos de Prueba de Servicios Simulados**:
  - `testPaymentProcessing()`: Verifica que el sistema maneje correctamente la respuesta simulada de un servicio de pago (por ejemplo, respuesta positiva o negativa).
  - `testUserNotification()`: Asegura que el sistema notifique correctamente al usuario cuando se simula una notificación.

5.3 Pruebas de Carga

Para evaluar el rendimiento y la escalabilidad del sistema, se realizaron pruebas de carga utilizando herramientas como `Apache JMeter` y `Gatling`. Estas herramientas simulan múltiples usuarios accediendo al sistema simultáneamente para identificar posibles cuellos de botella y medir la capacidad del sistema para manejar el tráfico.

- **Herramientas Utilizadas**:
  - `Apache JMeter`: Utilizado para simular un alto volumen de solicitudes y medir el tiempo de respuesta del sistema.
- **Resultados y Métricas Obtenidas**:
  - **Tiempo de Respuesta Promedio**: El sistema mostró un tiempo de respuesta

promedio de 250 ms para solicitudes de lectura y 400 ms para solicitudes de escritura bajo una carga moderada (hasta 100 usuarios simultáneos).

- **Escalabilidad**: Se observó una disminución en el rendimiento cuando se superaban los 200 usuarios simultáneos, lo que sugiere que podrían ser necesarios ajustes en la infraestructura o en el código para manejar una mayor carga.
- **Tasa de Éxito de Transacciones**: Durante las pruebas de carga, la tasa de éxito de las transacciones de pago fue del 98%, con errores principalmente en las respuestas de los servicios externos simulados.

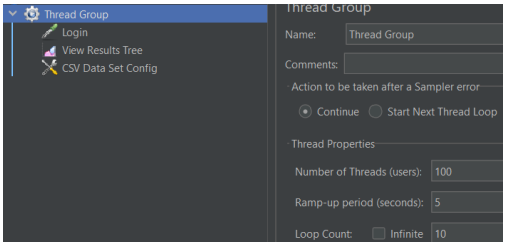

Figure 3: Interfaz JMeter con prueba de carga para un servicio

5.4 Pruebas de API

Las pruebas de las API se realizaron utilizando herramientas como `Postman` para asegurar que todas las API del sistema funcionen correctamente, especialmente aquellas expuestas por los microservicios del sistema.

- **Herramientas Utilizadas**:
  - `Postman`: Utilizado para realizar pruebas manuales de las API, permitiendo verificar que los endpoints devuelvan los datos correctos.
- **Resultados de las Pruebas de API**:
  - `GET /api/users`: Verifica que el endpoint de obtención de usuarios devuelva correctamente la lista de usuarios.
  - `POST /api/payments`: Asegura que los pagos se procesen correctamente, respondiendo con un código 200 y la información de la transacción.
  - `PUT /api/leases`: Verifica que la actualización de contratos de arrendamiento funcione correctamente.

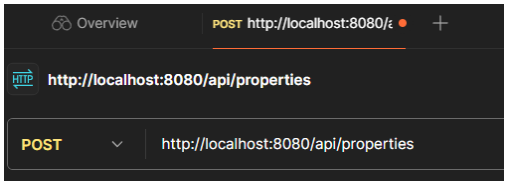

Figure 4: Interfaz de Postman con prueba de endpoint para Property

5.5 Informe de Pruebas

Para facilitar el análisis de los resultados de las pruebas, se utilizó `Allure Report`, una herramienta que permite generar informes visuales detallados de los resultados de las pruebas, facilitando la identificación de errores y la evaluación de la cobertura de las pruebas.

- **Generación de Informes**: Los resultados de las pruebas unitarias, de integración y de carga se exportaron a `Allure Report` para ser visualizados de manera clara y comprensible.
- **Visualización de Resultados**: El informe contiene detalles sobre el estado de cada prueba, el tiempo de ejecución, los errores encontrados y las métricas generales de rendimiento, lo que facilita la toma de decisiones sobre el siguiente ciclo de desarrollo.

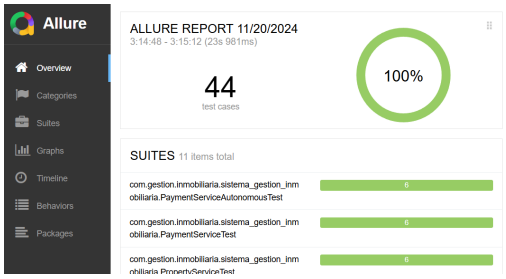

Figure 5: Interfaz de Allure

6 Integración Continua y Despliegue Continuo (CI/CD)

6.1 Pipeline de CI

El flujo de integración continua (CI) está configurado en Jenkins mediante un archivo `Jenkinsfile`, el cual describe el proceso automatizado de construcción, análisis de calidad de código, ejecución de pruebas y despliegue en contenedores Docker. Este pipeline

garantiza que los cambios realizados en el código se validen continuamente, minimizando el riesgo de introducir errores en el sistema.

El `Jenkinsfile` está estructurado en varias etapas que cubren los siguientes aspectos clave:

- **Clone Repository**: En esta etapa, el código fuente se clona desde el repositorio de GitHub, lo que asegura que siempre se esté trabajando con la última versión del proyecto.
- **SonarQube Analysis**: Se realiza un análisis de calidad de código utilizando `SonarQube`. En esta etapa, se ejecuta el comando `mvn clean verify sonar:sonar` para realizar el análisis y verificar las métricas de calidad, como la cobertura de pruebas, la complejidad del código y la detección de vulnerabilidades. Este análisis se realiza solo en el subdirectorio correspondiente al backend.
- **Build**: Una vez completado el análisis, se realiza la construcción del proyecto utilizando Maven. Este paso compila y empaqueta el código en un archivo `JAR` que luego será desplegado.
- **Generate Allure Report**: Después de las pruebas, se genera un informe con `Allure Report` para visualizar los resultados de las pruebas automatizadas. Esto permite un análisis detallado de la calidad del software.
- **Docker Build and Deploy**: Finalmente, se crea y despliega una imagen Docker con la aplicación. El contenedor se ejecuta en el puerto 8082 y es eliminado si ya existe una instancia previa del mismo. Este paso garantiza que el software esté preparado para ser desplegado en un entorno de producción o pruebas.

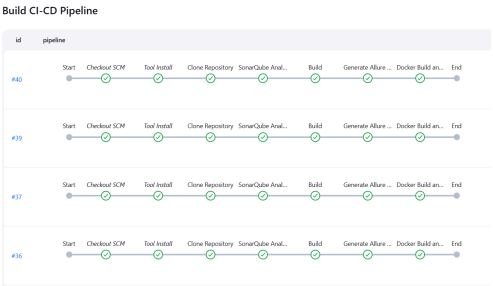

Figure 6: Stages de pipeline

6.2 Pipeline de CD

El flujo de despliegue continuo (CD) automatiza el proceso de entrega de software, llevando la aplicación a entornos de pruebas o producción con cada cambio realizado. El pipeline de CD se basa en el mismo `Jenkinsfile` y tiene como objetivo facilitar la entrega y ejecución del sistema en contenedores Docker. El despliegue se realiza automáticamente tras cada construcción exitosa, lo que permite contar con una versión actualizada de la aplicación en un entorno de pruebas o producción sin intervención manual.

El proceso de despliegue se divide en los siguientes pasos:

- **Eliminación de Contenedores Existentes**: Antes de crear una nueva instancia del contenedor, se comprueba si existe un contenedor con el mismo nombre. Si se encuentra, el contenedor anterior se elimina para evitar conflictos.
- **Creación de Imagen Docker**: Una vez que el contenedor anterior ha sido eliminado, se crea una nueva imagen Docker basada en el código recientemente compilado y empaquetado.
- **Ejecución del Contenedor Docker**: Finalmente, el contenedor con la nueva imagen se ejecuta en el entorno especificado. Esto se realiza de manera automatizada, sin intervención manual, garantizando que siempre se utilice la versión más reciente del sistema.

Este proceso se repite en cada cambio de código, asegurando que el sistema esté siempre disponible en un entorno controlado y actualizado.

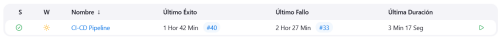

Figure 7: Interfaz de Jenkins en panel de control

6.3 Evidencias de Funcionamiento

Para verificar el funcionamiento del pipeline de CI/CD, se proporcionan registros detallados (logs) y capturas de pantalla de las distintas etapas de la ejecución del pipeline. A continuación, se incluyen algunos ejemplos de evidencia que pueden visualizarse a través de los logs de Jenkins:

- **Logs de Jenkins**: Cada etapa del pipeline genera logs detallados que permiten observar el progreso y el estado de la ejecución, incluyendo los resultados de las pruebas, la construcción del proyecto y la creación de la imagen Docker.
- **Capturas de Pantalla del Pipeline**: Se pueden capturar imágenes de las distintas etapas del pipeline, como la construcción exitosa del

proyecto, la ejecución de pruebas y el despliegue de la imagen Docker.

- **Enlace al Repositorio de GitHub**: Un enlace al repositorio de GitHub donde se almacena el código fuente y el `Jenkinsfile` que describe el pipeline de CI/CD.

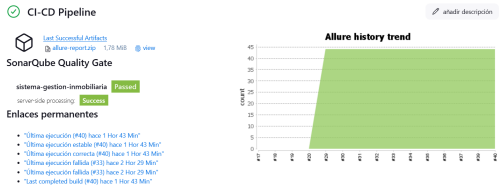

Figure 8: Funcionamiento de pipeline en Jenkins

6.4 Monitorización y Validación

Después de cada despliegue, se asegura que el sistema esté funcionando correctamente a través de un proceso de monitorización automatizado. Este proceso incluye:

- **Verificación de Logs**: Se monitorean los logs de los contenedores Docker y las aplicaciones para detectar errores o problemas en el rendimiento.
- **Pruebas de Salud (Health Checks)**: Se configuran pruebas de salud en el contenedor Docker para garantizar que la aplicación esté activa y respondiendo a las solicitudes.
- **Validación de Funcionalidad**: Tras cada despliegue, se ejecutan pruebas automatizadas de integración y funcionalidad para verificar que los cambios no hayan afectado la operación del sistema.

Este enfoque asegura que el software se mantenga estable y funcional, proporcionando un alto grado de fiabilidad en los entornos de producción y pruebas.

7 Análisis de Calidad de Código

7.1 Herramientas de Calidad de Código

Para garantizar que el código desarrollado cumpla con altos estándares de calidad, se utilizó `SonarQube`, una herramienta de análisis estático de código que permite evaluar diversos aspectos del software, como la cobertura de pruebas, la detección de vulnerabilidades de seguridad y la calidad general del código.

El proceso de análisis de calidad se integra en el pipeline de integración continua (CI) y se ejecuta automáticamente en cada cambio realizado en el código fuente. Las principales funcionalidades de `SonarQube` utilizadas incluyen:

- **Cobertura de Pruebas**: SonarQube mide el porcentaje de líneas de código cubiertas por pruebas automatizadas. Este indicador es crucial para garantizar que el código sea probado de manera adecuada, lo que reduce el riesgo de introducir errores en el sistema.
- **Detección de Vulnerabilidades**: La herramienta también analiza el código en busca de posibles vulnerabilidades de seguridad, como inyecciones SQL, exposición de datos sensibles o errores en la gestión de recursos.
- **Complejidad del Código**: SonarQube evalúa la complejidad del código para identificar fragmentos de código que podrían ser difíciles de mantener o propensos a errores debido a su diseño complejo.
- **Duplicación de Código**: Se analizan las secciones del código duplicado que podrían hacer que el sistema sea más difícil de mantener y propenso a errores.
- **Reglas de Codificación**: SonarQube verifica que el código cumpla con las reglas de codificación definidas, asegurando que se siga una convención y estilo uniformes en todo el proyecto.

7.2 Métricas Obtenidas

A lo largo del proceso de análisis de calidad, SonarQube genera una serie de métricas que proporcionan una visión detallada de la calidad del código. Entre las métricas más relevantes obtenidas se incluyen:

- **Cobertura de Pruebas**: Se observó que había una alta cobertura de pruebas, lo que indica que un alto porcentaje de las líneas de código están siendo probadas. Este valor fue considerado aceptable, pero se tomaron acciones para aumentar la cobertura de pruebas en las áreas críticas del sistema.
- **Vulnerabilidades de Seguridad**: SonarQube identificó vulnerabilidades en el código. Estas vulnerabilidades fueron revisadas y solucionadas

priorizando aquellas de alto impacto, como las que podrían permitir la inyección de código malicioso o el acceso no autorizado a datos sensibles.

- **Complejidad Ciclomática**: La complejidad ciclomática, que mide la cantidad de caminos de ejecución posibles en una función, fue mayor a la esperada. Se consideró que algunas funciones tenían una complejidad demasiado alta, por lo que se refactorizaron para mejorar la legibilidad y mantenimiento del código.
- **Duplicación de Código**: Se detectó un porcentaje de código duplicado, lo que podría aumentar la probabilidad de errores y hacer que el mantenimiento sea más costoso. Se implementaron soluciones para reducir la duplicación, refactorizando y extrayendo métodos comunes a clases reutilizables.

7.3 Acciones Tomadas en Base a los Resultados

Con base en los resultados obtenidos de SonarQube, se tomaron varias acciones para mejorar la calidad del código:

- **Mejora de la Cobertura de Pruebas**: Se identificaron áreas del código con baja cobertura de pruebas, especialmente en componentes críticos del sistema. Se añadieron pruebas unitarias y de integración para asegurar que esas áreas estén adecuadamente cubiertas y se minimizaran riesgos de errores.
- **Refactorización del Código Complejo**: Se refactorizaron las funciones con alta complejidad ciclomática para simplificar su lógica y mejorar su legibilidad. Esto facilita el mantenimiento a largo plazo y reduce la posibilidad de errores.
- **Solución de Vulnerabilidades**: Se abordaron todas las vulnerabilidades de seguridad encontradas, priorizando aquellas que representaban un mayor riesgo. Se aplicaron parches de seguridad, como la sanitización de entradas de usuario y la validación de parámetros para evitar ataques de inyección.
- **Reducción de la Duplicación de Código**: Se refactorizó el código para eliminar duplicaciones, centralizando funcionalidades comunes en funciones o clases reutilizables. Esto mejoró la eficiencia y mantenimiento del sistema.
- **Mejoras en el Cumplimiento de Reglas de Codificación**: Se reforzó el cumplimiento de las convenciones de codificación a través de herramientas de linters y revisiones de código. Esto ayuda a mantener el código consistente y fácilmente comprensible por todos los desarrolladores.

Estas acciones contribuyeron significativamente a mejorar la calidad del código y a reducir riesgos asociados con el mantenimiento del sistema en el futuro. A medida que el proyecto avanzaba, el análisis continuo con SonarQube permitió un ciclo de retroalimentación constante que ayudó a mantener altos estándares de calidad en el desarrollo.

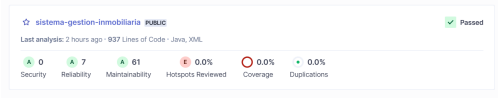

Figure 9: Resultados finales SonarQube

8 Retos Técnicos y Soluciones Implementadas

Durante la construcción y las pruebas del software se presentaron varios desafíos técnicos que requirieron atención y resolución para asegurar el correcto funcionamiento del sistema. A continuación se describen los principales problemas encontrados y las soluciones adoptadas:

8.1 Problema de Compatibilidad entre JUnit y Allure

Uno de los principales retos técnicos se presentó al intentar integrar la herramienta `Allure` para la generación de informes de pruebas automatizadas en conjunto con `JUnit`. Se detectó un problema de compatibilidad entre las versiones de `JUnit` y `Allure`, lo que causaba que los resultados de las pruebas no se reportaran correctamente y los informes no se generaran de manera adecuada.

Solución Adoptada: Para resolver este problema, se investigaron las versiones de `JUnit` y `Allure` compatibles y se actualizó `JUnit` a una versión que fuera compatible con la librería `Allure`. Además, se ajustaron las configuraciones del proyecto en `Maven` y en el archivo de pruebas para garantizar que los resultados se generaran correctamente en los informes de `Allure`. Tras realizar estas actualizaciones y ajustes, se logró la correcta integración y generación de los informes de pruebas automatizadas.

8.2 Desafíos en la Configuración de Jenkins y los Pipelines

Otro reto importante fue la configuración de los pipelines de integración continua (CI) y despliegue continuo (CD) en `Jenkins`. La integración de herramientas como `SonarQube` y la ejecución de pruebas automatizadas de manera eficiente en el pipeline requirieron tiempo y esfuerzo adicional debido a problemas de configuración, especialmente con las credenciales y los entornos de ejecución en `Jenkins`.

Solución Adoptada: Para solucionar los problemas de configuración en Jenkins, se revisó detalladamente la configuración de las herramientas y los entornos de ejecución. Se aseguraron las credenciales necesarias para `SonarQube` y se ajustaron los scripts de `Jenkinsfile` para hacer un uso correcto de las herramientas. En particular, se configuraron las variables de entorno y las credenciales necesarias para el análisis en `SonarQube` y la ejecución de las pruebas de forma automatizada, asegurando que los procesos de análisis y construcción se ejecutaran sin problemas.

8.3 Problemas con la Configuración de Docker

En el proceso de integración de `Docker` para la construcción de imágenes y ejecución de contenedores, se presentaron varios problemas relacionados con la configuración de los archivos `Dockerfile` y la gestión de contenedores, como errores al intentar eliminar contenedores existentes o al construir imágenes. Estos problemas dificultaron la automatización del despliegue del sistema en contenedores.

Solución Adoptada: Para resolver estos problemas, se revisaron y ajustaron los comandos de `Docker` en los scripts del pipeline. Se implementaron validaciones para verificar la existencia de contenedores antes de intentar eliminarlos, lo que evitó errores durante la construcción de nuevas imágenes. Además, se revisaron los archivos `Dockerfile` para garantizar que las configuraciones fueran correctas y se ajustaron los puertos y configuraciones de red para asegurar que los contenedores se ejecutaran de manera eficiente.

8.4 Limitaciones en las Pruebas de la Interfaz Gráfica (GUI)

Un desafío adicional fue la falta de pruebas exhaustivas en la interfaz gráfica del sistema. Aunque la conexión entre el frontend y el backend funcionaba correctamente, el sistema frontend aún se encontraba en desarrollo en las últimas fases del proyecto. Debido a esto, no se realizaron pruebas exhaustivas de la interfaz de usuario (GUI), y algunas funcionalidades, como las validaciones y la lógica de negocio del frontend, no estaban completamente implementadas.

Solución Adoptada: Si bien las pruebas de la interfaz gráfica fueron limitadas, se priorizó el aseguramiento de la conectividad y la integración entre el frontend y el backend. En lugar de realizar pruebas exhaustivas de GUI, se realizaron pruebas de integración y validación funcional para asegurar que los datos y las acciones entre las distintas partes del sistema se procesaran correctamente. A medida que el desarrollo del frontend avanzaba, se programaron pruebas adicionales para cubrir las funcionalidades específicas de la interfaz, pero no fue posible completar estas pruebas en su totalidad dentro del tiempo disponible.

8.5 Conclusiones y Justificación de las Decisiones

Cada uno de estos retos presentó oportunidades de aprendizaje y permitió la mejora de los procesos de desarrollo y pruebas. Las soluciones adoptadas fueron fundamentales para garantizar la correcta integración y calidad del sistema. Las decisiones tomadas, como la actualización de las herramientas y la mejora en la configuración de los pipelines, fueron clave para mantener un flujo de trabajo eficiente y mejorar la calidad del software entregado.

A pesar de las limitaciones en las pruebas de la GUI y el tiempo restringido para la finalización del frontend, se logró garantizar que el sistema funcionara correctamente en su mayor parte, con énfasis en las pruebas del backend y la integración de servicios. Las soluciones implementadas ayudaron a superar los principales obstáculos y a entregar un producto funcional y de calidad.

9 Conclusiones

9.1 Resumen de lo Logrado Durante el Proyecto

Durante el desarrollo del sistema de gestión inmobiliaria, se logró crear una plataforma integral que abarca las principales funcionalidades necesarias para gestionar propiedades, usuarios y transacciones en un entorno automatizado y eficiente. El sistema fue diseñado con una arquitectura modular que facilitó su escalabilidad y mantenimiento, utilizando tecnologías como Java para el backend, Docker para la contenedorización, Jenkins para la integración continua, y

herramientas como SonarQube y Allure para asegurar la calidad del código y las pruebas.

A lo largo del proyecto, se completaron varias fases clave, incluyendo la configuración del pipeline de CI/CD, la integración de pruebas unitarias, y la implementación de contenedores Docker para facilitar el despliegue. El sistema está en condiciones operativas, con un backend funcional y una base de datos estructurada para almacenar la información de manera segura y eficiente.

9.2 Reflexiones sobre el Proceso y los Aprendizajes Obtenidos

El desarrollo de este proyecto fue una experiencia enriquecedora que permitió profundizar en aspectos técnicos fundamentales del desarrollo de software moderno, como la integración continua, el despliegue automatizado, y el aseguramiento de la calidad del código. Sin embargo, también se enfrentaron desafíos significativos, especialmente en la integración de herramientas de pruebas automatizadas como Allure, así como en la configuración de Jenkins y Docker. Estos retos ofrecieron una oportunidad para mejorar las habilidades de resolución de problemas y aprender a gestionar la complejidad en entornos de desarrollo ágiles.

Además, se adquirió una comprensión más profunda sobre la importancia de las pruebas de integración y la necesidad de garantizar que todos los componentes del sistema trabajen de manera conjunta antes del despliegue en producción. Aunque no se realizaron pruebas exhaustivas de la interfaz gráfica debido a la etapa tardía del desarrollo del frontend, el sistema backend fue validado y asegurado de manera efectiva.

9.3 Posibles Mejoras o Pasos Futuros para Continuar el Desarrollo del Sistema

A pesar de que el sistema ha alcanzado un nivel funcional, existen varias áreas que podrían beneficiarse de mejoras en futuras iteraciones del proyecto:

- **Pruebas Exhaustivas de la Interfaz Gráfica (GUI):** Con el avance del desarrollo del frontend, se recomienda realizar una serie de pruebas exhaustivas de la interfaz gráfica, incluyendo pruebas de usabilidad y validación de campos, para garantizar una experiencia de usuario fluida y sin errores.
- **Optimización de la Escalabilidad y Rendimiento:** Se podría mejorar la arquitectura del sistema para manejar un mayor volumen de usuarios y transacciones, mediante técnicas de escalabilidad horizontal, cachés distribuidos, y optimización de consultas en la base de datos.
- **Ampliación de Funcionalidades:** El sistema podría expandirse para incluir nuevas funcionalidades, como la integración con servicios externos para pagos en línea, gestión de contratos, o la implementación de un sistema de recomendaciones para los usuarios basado en sus preferencias.
- **Mejora en la Seguridad:** Aunque se implementaron medidas básicas de seguridad, en el futuro se podrían integrar técnicas avanzadas, como la autenticación multifactor (MFA) y el cifrado de datos en tránsito y reposo, para proteger mejor la información sensible.
- **Despliegue en la Nube:** A medida que el sistema crezca, se recomienda considerar el uso de servicios en la nube como AWS o Azure para mejorar la disponibilidad, redundancia, y escalabilidad del sistema.

En conclusión, este proyecto ha sido un importante paso en la creación de una solución funcional y escalable para la gestión inmobiliaria. Si bien se lograron los objetivos principales, los pasos futuros se enfocarán en mejorar la interfaz de usuario, optimizar el rendimiento y expandir las capacidades del sistema para satisfacer mejor las necesidades de los usuarios.

10 Referencias

- **Maven:** The Apache Maven Project. "Maven: A software project management and comprehension tool." `https://maven.apache.org/`.
- **SonarQube:** SonarSource. "SonarQube: Continuous Inspection of Code Quality." `https://www.sonarqube.org/`.
- **Allure Framework:** "Allure Framework: A flexible and lightweight test report tool." `https://allure.qatools.ru/`.
- **Docker:** Docker Inc. "Docker: An open-source platform for automating the deployment, scaling, and management of applications." `https://www.docker.com/`.
- **Jenkins:** Jenkins CI. "Jenkins: The leading open-source automation server." `https://www.jenkins.io/`.

- **JUnit:** Kent Beck. "JUnit: A simple framework to write repeatable tests." `https://junit.org/`.
- **Gatling:** Gatling Corp. "Gatling: A powerful tool for load testing." `https://gatling.io/`.
- **Selenium:** Selenium Project. "Selenium: A framework for automated testing of web applications." `https://www.selenium.dev/`.
- **Spring Boot:** Pivotal Software. "Spring Boot: Simplifying Java development." `https://spring.io/projects/spring-boot`.
- **Apache JMeter:** The Apache Software Foundation. "JMeter: A tool for performance and load testing." `https://jmeter.apache.org/`.
- **GitHub:** GitHub, Inc. "GitHub: Where the world builds software." `https://github.com/`.