Universidad de Oviedo
Universidá d'Uviéu
University of Oviedo

Escuela de Ingeniería Informática
Universidad de Oviedo

Trabajo de Desarrollo

Trabajo De Fin De Grado: Definición y prototipado de pipelines CI/CD en jenkins (AutoFlow)

Grado en Ingeniería Informática del Software

Trabajo de Fin de Grado

AUTOR

Pedro Zahonero Mangas

TUTOR

Pablo Javier Tuya

Junio 2025

Versión 1.95 (15/07/2020). Modelo Métrica 3

Teaching Innovation Project*: PINN-19-A-029 (University of Oviedo)*

This work has been published in [2]

Este documento ha sido creado basándose en la plantilla elaborada por JOSÉ MANUEL REDONDO LÓPEZ. [1] [2]

Agradecimientos

Agradezco a todas las personas que me han apoyado y dado ánimo durante el desarrollo del TFG y a las que me han enseñado todo lo que sé y me han ayudado a ser la persona que soy hoy en día.

Índice de contenido

Capitulo 1 ¿Qué es este trabajo?

1.1 Resumen

Este trabajo se centra en la definición y prototipado de pipelines CI/CD en Jenkins para automatizar el desarrollo y despliegue de distintas aplicaciones. Se utiliza un entorno Docker donde Jenkins y otros servicios (SonarQube, PostgreSQL, Selenoid) se integran para dar soporte a la pipeline. La pipeline está diseñado para tres proyectos de diferentes tecnologías: Spring Java, Node.js y .NET.

El proyecto contiene agentes Jenkins que ejecutan las pipelines para cada una de las plataformas indicadas anteriormente. Se utiliza SonarQube para realizar análisis de calidad del código junto a una base de datos PostgreSQL para almacenar los resultados y Selenoid para ejecutar pruebas frontend en navegadores simulados. También se ha desarrollado un plugin personalizado para el envío de notificaciones por slack y una librería común que permite reutilizar funciones comunes en las pipelines y así optimizar la configuración para distintos proyectos.

El objetivo es proporcionar un sistema completo para la definición y prototipado de pipelines CI/CD en Jenkins de manera que pueda servir como guía a desarrolladores para tener su propia configuración.

1.2 Palabras clave

Infraestructura, pipelines, Jenkins, SonarQube, Spring, Node.js, .NET

1.3 Abstract

This work focuses on defining and prototyping CI/CD pipelines in Jenkins to automate the development and deployment of various applications. A Docker-based environment is used, in which Jenkins and other services (SonarQube, PostgreSQL, Selenoid) are integrated to support the pipeline. The pipeline is designed for three projects using different technologies: Spring Java, Node.js, and .NET.

The project includes Jenkins agents that execute the pipelines for each of the aforementioned platforms. SonarQube is used to perform code-quality analysis alongside a PostgreSQL database to store the results, and Selenoid to run frontend tests in simulated browsers. Additionally, a custom plugin has been developed for sending notifications via Slack, as well as a shared library that enables reuse of common functions across pipelines, thereby optimizing configuration for different projects.

The objective is to provide a comprehensive system for defining and prototyping CI/CD pipelines in Jenkins that can serve as a guide for developers to create their own configurations.

1.4 Keywords

Infrastructure, pipelines, Jenkins, SonarQube, Spring, Node.js, .NET

Capitulo 2 Estudio de Viabilidad del Sistema

2.1 Identificación del alcance del sistema

El objetivo de este proyecto es proporcionar un entorno completo para la definición y prototipado de pipelines CI/CD en Jenkins. El propósito es facilitar a los desarrolladores la automatización del ciclo de vida del software mediante un conjunto de ejemplos que permitan la integración, prueba y despliegue de aplicaciones en entornos reales.

El sistema abarca la gestión de pipelines para múltiples tecnologías (Spring, Node.js y .NET), asegurando su integración con herramientas clave como Docker, SonarQube, Selenoid y Azure DevOps. Además, se proporciona una librería común para la gestión de pipelines facilitando la configuración y reutilización de procesos automatizados y un plugin para la notificación en Slack.

Objetivos Estratégicos del Sistema

Implementación de una Infraestructura de CI/CD Escalable y Modular: Diseñar y desarrollar un sistema basado en Jenkins y contenedores Docker que pueda adaptarse a distintos lenguajes y frameworks.

Facilitar la Implementación de Pipelines CI/CD: Proporcionar un conjunto de ejemplos prácticos, librerías y configuraciones predefinidas para que los desarrolladores puedan integrar rápidamente pipelines CI/CD en sus proyectos.

2.2 Requisitos de usuario

Requisitos Funcionales (RF)

1. **Gestión de pipelines CI/CD:**
    - RF1.1: El sistema debe implementar pipelines independientes para proyectos Java, Node.js y .NET.
    - RF1.2: Cada pipeline debe incluir varias etapas.
        - RF1.2.1: Instalación de dependencias.
        - RF1.2.2: Compilación
        - RF1.2.3: Ejecución de pruebas
        - RF1.2.4: Análisis estático de código:
        - RF1.2.5: Despliegue en Azure.
    - RF1.3: El sistema debe permitir la creación automática de contenedores Docker para empaquetar las aplicaciones.
    - RF1.4: Las pipelines deben incluir pruebas de frontend utilizando Selenoid.
    - RF1.5: Las pipelines deben reportar el estado de las ejecuciones mediante logs y métricas detalladas.
    - RF1.6: Las pipelines deben incluir análisis estático mediante SonarQube
2. **Configuración de Jenkins y su ecosistema:**
    - RF2.1: El sistema debe permitir la configuración de Jenkins en un contenedor Docker.
    - RF2.2: Debe ser posible registrar tres agentes en Jenkins
        - RF2.2.1: Un agente para el proyecto Spring
        - RF2.2.2: Un agente para el proyecto Node.js
        - RF2.2.3: Un agente especializado para proyectos .NET con .NET SDK instalado para poder ejecutar comandos específicos.
    - RF2.3: El sistema debe integrar SonarQube para el análisis estático del código.
    - RF2.4: PostgreSQL debe estar disponible como base de dataos de SonarQube.
3. **Desarrollo de componentes reutilizables:**
    - RF3.1: El sistema debe incluir una librería común con configuraciones y scripts compartidos entre los pipelines.

- RF3.2: Se debe desarrollar un plugin para Jenkins que envíe notificaciones de los pipelines a un canal de Slack.
    - RF3.2.1: El plugin debe permitir configurar mensajes personalizados según el estado del pipeline
        - RF3.2.1.1: Éxito.
        - RF3.2.1.2: Fallo: Debe notificarse en qué etapa se reportó el fallo.
        - RF3.2.1.3: Advertencia.

4. **Pruebas y validación:**
    - RF4.1: El sistema debe permitir ejecutar pruebas automáticas de integración y unitarias en todos los pipelines.
    - RF4.2: Las pruebas de frontend deben ejecutarse mediante un contenedor Selenoid que soporte navegadores automatizados.
    - RF4.3: El sistema debe generar reportes de las pruebas.

Requisitos No Funcionales (RNF)

1. **Rendimiento:**
    - RNF1.1: Los pipelines deben completarse en un tiempo razonable, con un máximo de 5 minutos para proyectos pequeños y 15 minutos para proyectos más complejos.
    - RNF1.2: Las pruebas de frontend deben ejecutarse en menos de 5 minutos por conjunto de pruebas estándar.
2. **Escalabilidad:**
    - RNF2.1: El sistema debe permitir la adición de nuevos agentes a Jenkins sin necesidad de reconfiguración completa.
    - RNF2.2: Los pipelines deben ser adaptables a nuevos proyectos y lenguajes de programación.
3. **Mantenibilidad:**
    - RNF3.1: La librería común y el plugin de Slack deben estar documentados para facilitar su mantenimiento y actualización.
    - RNF3.2: Las configuraciones de Jenkins y los pipelines deben versionarse en un repositorio de control de versiones.
    - RNF3.3: Toda la infraestructura deberá estar documentada. Esta documentación deberá contar con diagramas, versiones de componentes y

procedimientos de instalación/configuración, garantizando la fácil comprensión y mantenimiento del sistema

4. **Fiabilidad:**
    - RNF4.1: La ejecución de los pipelines no debe ser interrumpida por fallos en los tests automatizados a no ser que sean fallos criticos. En caso de fallos no criticos, la pipeline debe continuar su ejecución y marcar el estado como "inestable" (amarillo) en Jenkins, permitiendo la visibilidad de los errores sin afectar el proceso global.
        - RNF4.1.1: Fallos criticos:
            - RNF4.1.1.1: Error de compilación
            - RNF4.1.1.2: Error de despliegue
        - RNF4.1.2: Fallos no criticos:
            - RNF4.1.2.1: Warnings en la aplicación
            - RNF4.2.2.2: Error en la QualityGate
            - RNF4.2.1.3: Las pruebas de la aplicación fallan
    - RNF4.2: Los resultados de las pruebas deben ser almacenados y accesibles para su análisis posterior, generando reportes detallados en SonarQube y Jenkins para facilitar la identificación de fallos y tendencias en la calidad del código.
5. **Seguridad:**
    - RNF5.1: Las credenciales y configuraciones sensibles (e.g., tokens de Slack, credenciales de Azure) deben almacenarse de forma segura en Jenkins (credenciales cifradas).
    - RNF5.2: El acceso a Jenkins y sus agentes debe estar restringido a usuarios autorizados mediante autenticación y control de roles.
6. **Interoperabilidad:**
    - RNF6.1: El sistema debe ser compatible con las versiones actuales de las herramientas involucradas (Jenkins, SonarQube, Docker, Azure).
    - RNF6.2: Las configuraciones deben seguir estándares que permitan su despliegue en diferentes entornos.
        - RNF6.2.1: Las herramientas deben estar configuradas en un mismo docker compose de manera que sea facil su instalación en otro entornos.
        - RNF6.2.2: Los contenedores deben estra conectados entre sí mediante la misma network

7. **Usabilidad:**
    - RNF7.1: El plugin de Slack debe proporcionar mensajes comprensibles y con enlaces directos a los logs de Jenkins.

2.3 EVS 4, 5 y 6: Estudio y Valoración de Alternativas de Solución y Selección de Alternativa Final

2.3.1 Alternativas sistemas CI/CD

A pesar de ser un proyecto sobre Jenkins es bueno saber sobre otras alternativas, así como como las ventajas y desventajas de otros sistemas. En este caso se analizará y comparará Jenkins, GitLab CI/CD, CircleCI y Azure Pipelines

2.3.1.1 Sistema 1 – Jenkins

Jenkins es un servidor open source desarrollado en Java que automatiza tareas en procesos de la integración continua.

Jenkins presenta las siguientes características:

- Es multiplataforma y accesible mediante interfaz web.
- Requiere ser instalado y configurado lo que puede presentar una curva de aprendizaje más alta que otras plataformas.
- Jenkins ofrece plugins lo que hace que esta herramienta sea personalizable.
- Documentación extensa y una comunidad que aporta plugins y soporte.

2.3.1.2 Sistema 2 – GitLab CI/CD y GitHub Actions

GitLab y GitHub tienen su propio sistema de automatización CI/CD, ambos en formato yaml. Ambos dan la facilidad de escalar proyectos y mantienen una documentación detallada y una comunidad activa, pero las dos cuentan con características diferentes.

GitLab

- Runners con aprovisionamiento dinámico.
- Integración nativa con el resto del ecosistema GitLab.
- Plan gratuito y planes de suscripción basados en minutos de CI y soporte.
- Escaneo de vulnerabilidades.

- Versión de código abierto que permite a las organizaciones alojar y modificar la plataforma según sus necesidades.

GitHub

- Actions reutilizables desde GitHub Marketplace.
- Ejecución en Linux, macOS y Windows.
- Límite de minutos gratuitos que se amplía mediante planes de pago.
- Plataforma propietaria con actualizaciones y mantenimiento centralizados.
- Integración nativa con repositorios, Dependabot y GitHub Security.

2.3.1.3 Sistema 3 – CircleCI

Es una herramienta basada en la nube, aunque también se puede instalar localmente. Sus características más destacadas son:

- Facilidad para integrarse con una alta variedad de servicios comunes utilizados en tu rutina diaria, como AWS, Slack o Atlassian.
- Seguridad excepcional y estando certificado con la acreditación FedRAMP, que confirma su alto estándar de seguridad.
- Interfaz web moderna
- Entornos de ejecución en Linux, macOS y Windows
- Integraciones nativas y extensibilidad mediante “Orbs”.
- Escalabilidad automática en la nube
- Plan gratuito limitado con opciones de pago según necesidades.
- Medidas de seguridad integradas y actualizaciones gestionadas por la propia plataforma.
- La configuración correcta de los flujos de trabajo y las pruebas iniciales en CircleCI puede llevar tiempo, especialmente si se trata de un proyecto complejo o con múltiples dependencias.

2.3.1.4 Sistema 4 – Azure Pipelines

Herramienta proporcionada por Microsoft, fácil de integrar con distintas herramientas y servicios de terceros. Sus características principales son:

- Es compatible con varios lenguajes de programación, plataformas y proveedores de nube.
- Al ser un producto de Microsoft la seguridad que proporciona es muy grande teniendo varias certificaciones de cumplimiento que le han permitido convertirse en un lideren

seguridad IaaS. No solo está protegido Azure, sino que también el usuario final está cubierto por este.

- Documentación completa y soporte a través de la comunidad de Azure DevOps
- Proporciona soporte completo para Windows, Linux y macOS.
- Tiene un límite de minutos gratuitos que pueden ampliarse con planes de pago.

2.3.1.4.1 Ventajas y desventajas de cada alternativa al sistema

Factor	*Jenkins*	*GitLab CI/CD*	*GitHub Action*	*CircleCI*	*Azure Pipelines*
Fácil instalación/configuración	+	-	+	-	+
Open source	+	+	+	-	-
Documentación y comunidad	+ (limitada en áreas específicas)	+	+	+	+
Curva de aprendizaje	-	-	+	+	-
Modularidad/personalización	+	+	+	+	+
Integración con herramientas	+	-	+	+	+
Escalabilidad	-	+	+	+	+
Costos asociados	+	-	-	-	-
Seguridad	-	+	+	+	+

2.3.2 Alternativas de herramientas de análisis de código estático

En este apartado compararemos distintas herramientas de análisis de código estático, entre ellas se encuentran. SonarQube, ESLint, PMD, CodeClimate y FindBugs.

2.3.2.1 SonarQube

SonarQube es una plataforma de código abierto utilizada para la inspección continua de la calidad del código. Mediante análisis estático, Sonarqube identifica errores simples, vulnerabilidades de seguridad y violaciones de buenas prácticas de programación. Algunas de estas buenas prácticas son: escribir código limpio y legible, evitar duplicación de código, tamaño de funciones y métodos, cobertura de código...

Características:

- Soporta más de 20 lenguajes de programación.
- Compatibilidad con múltiples entornos
- Reglas y métricas personalizables

2.3.2.2 ESLint

Centrado principalmente en JavaScript, ESLint es un linter ampliamente adoptado que identifica y reporta errores, violaciones de estándares de codificación y posibles errores.

Características:

- Ofrece amplias opciones de configuración e integraciones con varios IDE, lo que lo hace altamente personalizable y flexible.
- Es extensible gracias a los plugins lo que lo hace personalizable.
- Es gratuita y open source

2.3.2.3 PMD

PMD es un robusto analizador de código fuente con soporte para múltiples lenguajes que detecta fallos comunes de programación, problemas de estilo y posibles problemas de rendimiento. Viene con un conjunto de reglas predefinidas, pero también permite a los usuarios crear reglas personalizadas según sus necesidades.

Características:

- Configuración algo técnica
- Herramienta gratuita

2.3.2.4 CodeClimate

CodeClimate analiza repositorios de código y proporciona comentarios detallados sobre problemas relacionados con la complejidad del código, duplicación, vulnerabilidades y mantenibilidad. Ofrece integración con sistemas populares de control de versiones y facilita revisiones de código automatizadas.

Caracteristicas:

- Soporte de lenguajes limitado
- Tiene pocas opciones de personalización
- Para acceder a todas sus funciones hay que pagar

2.3.2.5 FindBugs

Diseñado para aplicaciones Java, FindBugs es una herramienta de análisis estático que identifica errores de programación comunes, posibles fallos e ineficiencias. Proporciona un conjunto de reglas personalizable para cumplir con los requisitos específicos del proyecto e integra bien con varios sistemas de compilación.

Características:

- El análisis de código es más básico que otras herramientas
- Tiene una comunidad pequeña
- Es gratuita y open source

2.3.2.5.1 Ventajas y desventajas

Factor	*SonarQube*	*ESLint*	*PMD*	*CodeClimate*	*FindBugs*
Soporte de lenguajes	+	-	+	-	-
Facilidad de configuración	+	+	-	+	+
Integración CI/CD	+	+	-	+	+
Personalización	+	+	+	-	+
Análisis de calidad	+	+	+	+	-
Costos	+	+	+	-	+

2.3.3 Alternativas sistemas de ejecución de navegadores online

En este apartado compararemos distintas herramientas que nos permite ejecutar pruebas frontend en navegadores de forma online. Compararemos Selenoid, BrowserStack TurboScale, Lambdatest HyperExecute, Gridlastic, Zalenium.

2.3.3.1 Selenium Remote Server

Es un componente de Selenium que actúa como un servidor intermedio entre el código de prueba y el navegador, permitiendo la ejecución de comandos en aplicaciones web de manera remota.

2.3.3.2 Selenium Grid

Es una herramienta que permite ejecutar pruebas automatizadas en múltiples navegadores y sistemas operativos simultáneamente. Tiene una infraestructura distribuida donde un “hub” coordina las pruebas en varios “Nodos”, optimizando el tiempo de ejecución.

2.3.3.3 Selenoid

Selenoid es una alternativa a Selenium Grid de código abierto, que ejecuta navegadores y emuladores de Android dentro de contenedores Docker. Se distribuye con un conjunto de imágenes Docker listas para usar que corresponden a la mayoría de las versiones populares de navegadores.

2.3.3.4 BrowserStack TurboScale

BrowserStack TurboScale ofrece una solución de cuadrícula de automatización de navegadores autogestionada y escalable la nube, diseñada para simplificar la gestión y optimizar los recursos. Permite a los equipos crear una cuadrícula de automatización de navegadores basada en Docker utilizando Kubernetes, que se puede configurar con un proveedor de nube preferido con una dependencia mínima de operaciones. Esto lo hace altamente adaptable y fácil de integrar en los flujos de trabajo existentes.

2.3.3.5 Lambdatest HyperExecute

LambdaTest HyperExecute es una plataforma inteligente de orquestación de pruebas para pruebas continuas. Está diseñada para optimizar los tiempos de ejecución de pruebas y admite una amplia gama de entornos y configuraciones. HyperExecute se integra perfectamente con herramientas populares de CI/CD, lo que permite flujos de trabajo simplificados y ciclos de entrega más rápidos.

2.3.3.6 Gridlastic

Gridlastic es una solución en línea de Selenium Grid que facilita la creación y gestión de infraestructuras de sistemas de prueba sofisticados. Ofrece un método detallado para realizar pruebas con soporte para diferentes versiones de navegadores y sistemas operativos. Además, Gridlastic permite implementar la automatización de pruebas en navegadores para mejorar la efectividad y la eficiencia de las pruebas.

2.3.3.7 Zalenium

Zalenium te permite desarrollar marcos de trabajo flexibles y extensibles para la automatización de pruebas. Lo que lo hace destacar es su objetivo de proporcionar una infraestructura desechable y flexible de Selenium Grid. También ofrece un contenedor escalable con un panel de control, vista previa en vivo y grabación de video.

2.3.3.7.1 Ventajas y desventajas de las diferentes herramientas

Factor	*Selenium Remote Server*	*Selenium Grid*	*Selenoid*	*BrowserStack TurboScale*	*Lambdatest HyperExecute*	*Gridlastic*	*Zalenium*
Ejecución en múltiples navegadores/SO	-	+	+	+	+	+	+
Facilidad de configuración	-	-	+	-	-	+	+
Soporte multiplataforma	+	+	+	+	+	+	-
Escalabilidad	-	+	+	+	+	+	+
Documentación y comunidad	+	+	-	-	-	-	+
Costo	+	+	+	-	-	-	+
Integración CI/CD	-	+	+	+	+	-	+

Capitulo 3 Planificación y Gestión del TFG

3.1 Planificación del proyecto

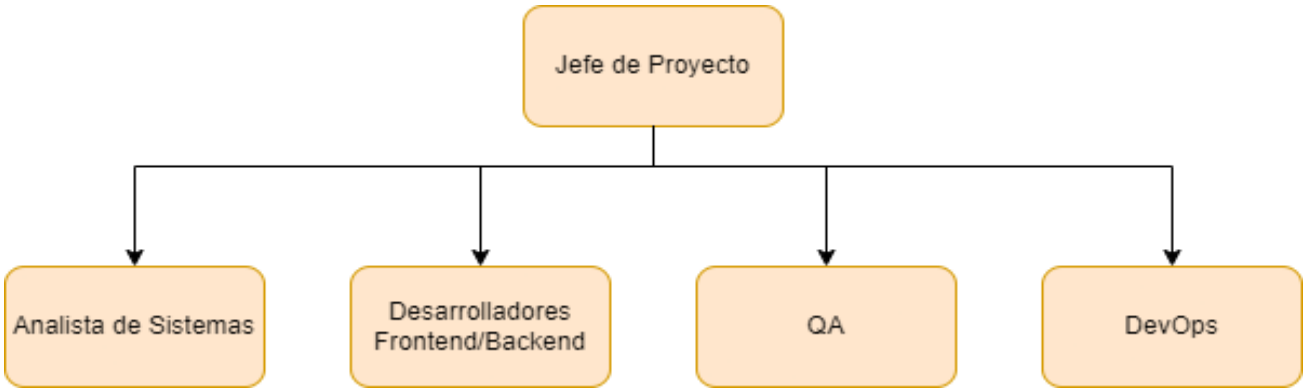
3.1.1 Identificación de Interesados

- Equipo de desarrollo: Son los encargados en desarrollar las aplicaciones, en este caso los proyectos Spring, Node.js y .NET.
- Operadores de Sistemas: Es el equipo encargado de gestionar la infraestructura técnica, servidores de despliegue y los contenedores.
- Equipo de QA: Se encargan de validar el código y la funcionalidad de las aplicaciones.
- Jefe del proyecto: En un sistema empresarial, sería el responsable de aprobar e implementar el sistema para garantizar con los objetivos del sistema.
- Ingeniero DevOps: Es el encargado de diseñar, configurar e implementar las pipelines CI/CD

3.1.2 OBS y PBS

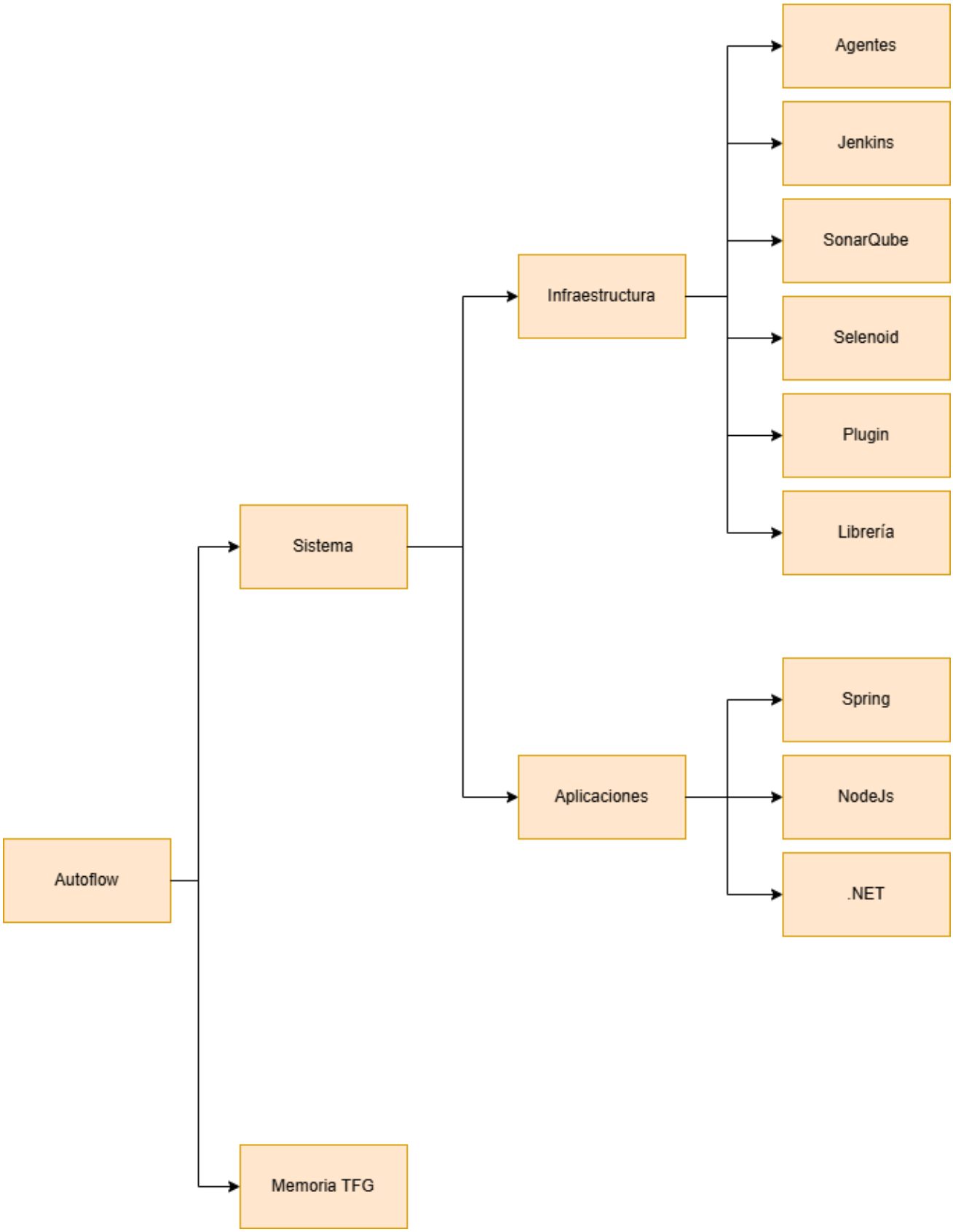
3.1.2.1 OBS

A continuación, se presenta un OBS del proyecto. A pesar de que este proyecto fue desarrollado por el alumno, se ha creado un escenario ficticio en el que una empresa está desarrollando una aplicación y está usando nuestro sistema.



3.1.2.2 PBS

A continuación, se presenta el PBS del proyecto:


Agentes
Jenkins
SonarQube
Infraestructura
Selenoid
Plugin
Librería
Sistema
Spring
Aplicaciones
NodeJs
Autoflow
.NET
Memoria TFG

3.1.3 Planificación Inicial. WBS

En esta sección se presenta el WBS. El inicio del proyecto se establece que es el 13/3/2024. A continuación, se presenta un diagrama de Gantt del proyecto:

EDT	Nombre de tarea	Duración	Duracion (horas)	Comienzo	Fin	Predecesoras
1	**DEFINICIÓN Y PROTOTIPADO DE PIPELINES CI/CD EN JENKINS**	**385 días?**	**327**	**lun 27/11/23**	**sáb 17/05/25**	
1.1	**Analisis del proyecto**	**99 días**	**43**	**vie 01/12/23**	**mié 17/04/24**	
1.1.1	Búsqueda de información acerca de la infraestructura del sistema de pipelines	17 días	24	vie 01/12/23	lun 25/12/23	
1.1.2	Analisis de los requisitos funcionales	5 días	8	mar 12/03/24	sáb 16/03/24	
1.1.3	Creación del Backlog	19 días	8	lun 18/03/24	jue 11/04/24	4
1.1.4	Creación de story mapping	4 días	3	vie 12/04/24	mié 17/04/24	5
1.2	**Organización**	**77 días**	**3**	**lun 27/11/23**	**mar 12/03/24**	
1.2.1	Reunión con el tutor del TFG	1 día	1	lun 27/11/23	lun 27/11/23	
1.2.2	Reunión con Indra y el tutor	1 día	1	lun 15/01/24	lun 15/01/24	
1.2.3	Reunión con el tutor del TFG	1 día	1	mar 12/03/24	mar 12/03/24	
1.3	**Desarrollo**	**340 días**	**210**	**lun 15/01/24**	**vie 02/05/25**	
1.3.1	Creación de la infraestrucutura Jenkins y SonarQube	30 días	35	lun 15/01/24	vie 23/02/24	
1.3.2	**Desarrollo de aplicaciones**	**76 días**	**74**	**lun 26/02/24**	**lun 10/06/24**	
1.3.2.1	Desarrollo aplicación Spring	12 días	12	lun 26/02/24	mar 12/03/24	12
1.3.2.2	Desarrollo de tests a	6 días	12	mié 13/03/24	mié 20/03/24	14
1.3.2.3	Desarrollo de aplicación NodeJs	11 días	12	vie 26/04/24	vie 10/05/24	
1.3.2.4	Desarrollo de tests aplicación NodeJs	7 días	12	lun 13/05/24	mar 21/05/24	16
1.3.2.5	Desarrollo de aplicación .NET	9 días	14	lun 20/05/24	jue 30/05/24	

1.3.2.6	Desarrollo de tests aplicación .NET	7 días	12	vie 31/05/24	lun 10/06/24	18
1.3.3	Configurar reporting	1 día	7	lun 24/06/24	lun 24/06/24	
1.3.4	**Configuración de Selenoid**	**9 días**	**20**	**mar 11/06/24**	**vie 21/06/24**	
1.3.4.1	Añadir al docker-compose Selenoid	4 días	4	mar 11/06/24	vie 14/06/24	
1.3.4.2	Configuración de tests e2e para el uso de Selenoid	5 días	16	lun 17/06/24	vie 21/06/24	22
1.3.5	**Configuración Azure**	**8 días**	**29**	**jue 20/06/24**	**lun 01/07/24**	
1.3.5.1	Entorno de despliegue Azure	3 días	12	jue 20/06/24	sáb 22/06/24	
1.3.5.2	Entorno de despliegue Azure	2 días	9	lun 24/06/24	mar 25/06/24	25
1.3.5.3	Entorno de despliegue Azure	4 días	8	mié 26/06/24	lun 01/07/24	26
1.3.6	Desarrollo de plugin	7 días	16	jue 19/09/24	vie 27/09/24	
1.3.7	Desarrollo de librería	1 día	4	sáb 15/06/24	sáb 15/06/24	
1.3.8	**Configuración del repositorios Git**	**110 días**	**3**	**mar 23/01/24**	**lun 24/06/24**	
1.3.8.1	Creacion de repositorio GitHub	1 día	0,5	mar 23/01/24	mar 23/01/24	
1.3.8.2	Conexión repositorio y Jenkins	1 día	0,5	mar 23/01/24	mar 23/01/24	
1.3.8.3	Creacion de repositorio GitLab	1 día	0,5	sáb 22/06/24	sáb 22/06/24	
1.3.8.4	Conexión repositorio y Jenkins	1 día	0,5	sáb 22/06/24	sáb 22/06/24	
1.3.8.5	Creacion de repositorio Bitucket	1 día	0,5	lun 24/06/24	lun 24/06/24	
1.3.8.6	Conexión repositorio y Jenkins	1 día	0,5	lun 24/06/24	lun 24/06/24	
1.3.9	**Configuración de pipelines**	**14 días**	**17**	**mar 04/06/24**	**vie 21/06/24**	
1.3.9.1	Pipeline basica NodeJs	3 días	4	mar 04/06/24	jue 06/06/24	
1.3.9.2	Añadir a la pipeline pruebas e2e con selenoid	3 días	3	vie 07/06/24	mar 11/06/24	38
1.3.9.3	Pipeline basica .NET	3 días	4	lun 17/06/24	mié 19/06/24	
1.3.9.4	Añadir a la pipeline pruebas e2e con selenoid	2 días	1	jue 20/06/24	vie 21/06/24	40
1.3.9.5	Pipeline basica Spring	3 días	4	mar 04/06/24	jue 06/06/24	
1.3.9.6	Añadir a la pipeline pruebas e2e con selenoid	2 días	1	vie 07/06/24	lun 10/06/24	42

1.3.10	**Instalación y configuración del sistema pipeline en un entorno Linux**	**2 días**	**5**	**jue 01/05/25**	**vie 02/05/25**	
1.3.10.1	Configuración Maquina Virtual	1 día	1	jue 01/05/25	jue 01/05/25	
1.3.10.2	Instalación y configuración del proyecto	1 día	4	vie 02/05/25	vie 02/05/25	45
1.4	**Memoria del TFG**	**150 días?**	**71**	**lun 21/10/24**	**sáb 17/05/25**	
1.4.1	Estructura basica	1 día	4	lun 21/10/24	lun 21/10/24	
1.4.2	**Estudio y viabilidad del sistema**	**63 días?**	**15**	**lun 23/12/24**	**mié 19/03/25**	
1.4.2.1	Estudio del alternativas	5 días	8	lun 23/12/24	vie 27/12/24	
1.4.2.2	Plan de pruebas	52 días	7	mar 07/01/25	mié 19/03/25	
1.4.3	**Diseño del sistema de Información**	**138 días?**	**9**	**lun 21/10/24**	**mié 30/04/25**	
1.4.3.1	Diseño de clases	3 días	2	lun 21/10/24	mié 23/10/24	
1.4.3.2	Diseño de pruebas	9 días	7	jue 20/03/25	mar 01/04/25	51
1.4.4	**Construcción del sistema de información**	**105 días?**	**40**	**lun 23/12/24**	**sáb 17/05/25**	
1.4.4.1	Ejecución y anotación de los resultados de las pruebas	7 días	8	mié 02/04/25	jue 10/04/25	54
1.4.4.2	Manual de creación y configuración del sistema	2 días	6	vie 16/05/25	sáb 17/05/25	
1.4.4.3	Manual de usuario	93 días?	24	lun 23/12/24	mié 30/04/25	
1.4.4.4	Manual de instalación	2 días	2	lun 05/05/25	mar 06/05/25	
1.4.5	**Planificación y Gestion del TFG**	**3 días**	**3**	**lun 05/05/25**	**mié 07/05/25**	
1.4.5.1	Planificación de riesgos	3 días	3	lun 05/05/25	mié 07/05/25	

3.1.4 Riesgos

En este apartado se presentará un plan de gestión de riesgos con el objetivo de identificar los riesgos, su impacto en el sistema y qué hacer ante ellos.

3.1.4.1 Identificación y análisis de riesgos

A continuación, se muestra una lista de riesgos con el análisis de riesgo. El análisis de riesgos nos ayuda evaluar la probabilidad y el impacto de los riesgos identificados, lo que nos facilita la toma de decisiones sobre cómo gestionarlos.

Para cada riesgo identificado, se ha asignado:

- **Probabilidad (P)** según la escala de cinco niveles (Muy Bajo = 10 %, Bajo = 30 %, Medio = 50 %, Alto = 70 %, Muy Alto = 90 %).
- **Impacto (I)** sobre los objetivos del proyecto en una escala de 1 a 3 (1 = Bajo, 2 = Medio, 3 = Alto).
- **Importancia (P*I)** como producto de Probabilidad x Impacto.

Los riesgos con **Importancia ≥ 1,5** se consideran **alta prioridad** y deberán abordarse con estrategias de mitigación o evasión en la planificación de la respuesta.

ID	Nombre	Descripción	P	I	P*I
1	Fallo de arranque de contenedores	Los contenedores de Jenkins o agentes no arrancan por configuración errónea de Docker Compose o límites de recursos incorrectos en el host.	10%	2	0,20
2	Agentes saturados	Un agente se queda bloqueado, generando cola y retrasos en el pipeline.	50%	3	1,50
3	Inestabilidad de Selenoid	Contenedor Selenoid falla o no responde, impidiendo ejecutar tests de front.	30%	2	0,60
4	Token/Credenciales inválidas	Credenciales de SonarQube, GitHub, Azure o Slack mal configuradas o expiradas, causando fallos en el uso de estas herramientas.	10%	3	0,30
5	Costes imprevistos en Azure	Uso excesivo de recursos cloud sin monitorización, disparando la factura más allá del presupuesto.	50%	2	1,00
6	Integración de librería común inconsistente	Cambios en la librería compartida generan inconsistencia en uno o varios proyectos	50%	3	1,50
7	Versión de Jenkins o plugins incompatible	Actualización automática o manual de Jenkins/plugins que rompe la compatibilidad con los Jenkinsfile	30%	2	0,60
8	Caída del servicio SonarQube	SonarQube deja de responder por fallo de la base de datos o del contenedor, impidiendo análisis de calidad y bloqueando despliegues automáticos.	30%	3	0,90
9	Escalabilidad limitada de la VM	La VM de Ubuntu 22.04 no aguanta la carga de múltiples builds concurrentes, provocando timeouts o errores de memoria.	50%	3	1,50

3.1.4.2 Estrategia de respuesta

A continuación, se presentan estrategias de respuesta para los riesgos identificados ordenado por prioridad.

ID	Riesgo	Acciones
2	Agentes saturados	- Monitorizar uso de CPU/memoria de agentes. - Establecer autoescalado horizontal de agentes Docker. - Priorizar colas y limitar concurrencia de pipelines.
6	Integración de librería común inconsistente	- Versionar librería con semver y publicar en repositorio (Artifactory/Nexus). - Ejecutar pipelines de integración de la librería antes de cada release.
9	Escalabilidad limitada de la VM	- Dimensionar VM según carga estimada; habilitar swap o memoria dinámica. - Evaluar migración a Kubernetes o añadir nodos adicionales
3	Inestabilidad de Selenoid	- Fijar versiones compatibles de navegadores en Selenoid. - Desplegar health checks y restart policies en Docker.
5	Costes imprevistos en Azure	- Configurar alertas de coste en Azure Cost Management. - Revisar uso de recursos semanalmente y ajustar tamaños o apagar entornos inactivos.
7	Versión de Jenkins o plugins incompatible	- Estar al tanto de actualizaciones y mantener versiones actualizadas.
8	Caída del servicio SonarQube	- Configurar backup y réplica de la base de datos PostgreSQL. - Desplegar SonarQube en modo cluster o con redundancia.
4	Token/Credenciales inválidas	- Automatizar rotación y validación de credenciales (GitHub, SonarQube, Azure, Slack). - Configurar comprobaciones periódicas en Jenkins.
1	Fallo de arranque de contenedores	- Validar docker-compose.yml con linters y tests de arranque automáticos. - Crear snapshots de configuración para rollback rápido.

3.1.5 Presupuesto de proyecto

En este apartado se calculará el coste del proyecto.

3.1.5.1 Coste del personal

A continuación, se calculará el coste del personal para posteriormente calcular el coste de las tareas del proyecto.

Perfil profesional	N.º personas	Salario medio bruto anual por persona (€)	Salario medio bruto anual (€)	Coste Trabajador (€/año)	Coste por hora (€)
Desarrollador	3	30.000	90.000	120.060,00	20,31
Ingeniero DevOps	1	52.728	52.728	70.339,15	36,63
Operador de Sistemas	1	32.400	32.400	43.221,60	22,51
QA	2	47.676	95.352	127.199,56	33,12
Jefe de Proyecto	1	46.200	46.200	61.630,80	32,10
Total				422.451,12	

El cálculo del coste del trabajador se ha seguido la siguiente formula:

Coste de un trabajador = Salario bruto + Seguridad Social

Seguridad Social = (Sueldo bruto * 0,236) + (Sueldo bruto * 0,055) + (Sueldo bruto * 0,035) + (Sueldo bruto * 0,006) + (Sueldo bruto * 0,002)

El valor de la Seguridad Social se determina de la siguiente manera (Se toma como referencia los valores para un contrato indefinido):

- Contingencias comunes: 23,6 % del sueldo bruto
- Prestaciones por desempleo: 5,5 % del sueldo bruto. Este valor aumenta en los contratos temporales.
- Contingencias profesionales: 3,5 % del sueldo bruto. (En realidad, este porcentaje depende de la actividad, pero tomaremos este valor como referencia).
- Formación: 0,6 % del sueldo bruto.
- FOGASA: 0,2 % del sueldo bruto.

3.1.5.2 Coste del WBS

Ahora se calcula el coste de realizar cada tarea, para ello se asignarán los responsables para cada proyecto y se calculará el coste de la tarea en función de las horas y lo que cuesta el tiempo del responsable.

WBS	Tarea	Responsable	Coste €/h	Horas	Coste	Coste total (€)
1.1.1	Búsqueda de información acerca de la infraestructura pipelines					1419,36
		Ingeniero DevOps	36,63	24	879,12	

		Operador de Sistemas	22,51	24	540,24	
1.1.2	Análisis de los requisitos funcionales	Jefe de Proyecto	32,10	8		256,80
1.1.3	Elaboración del backlog	Jefe de Proyecto	32,10	8		256,80
1.1.4	Story mapping					156,93
		Jefe de proyecto	32,10	3	96,30	
		Desarrollador	20,21	3	60,63	
1.2	Organización del proyecto	Jefe de proyecto	32,10	3		96,30
1.3.1	Infraestructura Jenkins y SonarQube					2069,9
		Ingeniero DevOps	36,63	35	1282,05	
		Operador de sistemas	22,51	35	787,85	
1.3.2.1	Desarrollo aplicación Spring	Desarrollador	20,84	12		250,12
1.3.2.2	Test Spring	Desarrollador	20,84	12		250,12
1.3.2.3	Desarrollo aplicación Node.js	Desarrollador	20,84	12		250,12
1.3.2.4	Test Node.js	Desarrollador	20,84	12		250,12
1.3.2.1	Desarrollo aplicación .NET	Desarrollador	20,84	14		291,76
1.3.2.2	Test .NET	Desarrollador	20,84	12		250,12
1.3.3	Reporting	Ingeniero DevOps	36,63	7		256,41
1.3.4	Configuración y pruebas Selenoid	Ingeniero DevOps	36,63	20		732,6
1.3.5	Despliegue en Azure	Ingeniero DevOps	36,63	29		1062,41
1.3.6	Desarrollo del plugin Slack	Desarrollador	20,84	16		333,50
1.3.7	Implementación de la librería común	Desarrollador	20,84	4		83,38
1.3.8	Configuración repositorios Git	Desarrollador	20,84	3		62,53
1.3.9	Creación de pipelines básica y E2E					977,14
		Ingeniero DevOps	36,63	17		622,71
		Desarrollador	20,84	17		354,28

1.3.10	Instalación y pruebas en Linux					287,35
		Ingeniero DevOps	36,63	5	183,15	
		Desarrollador	20,84	5	104,20	
1.4.1	Estructura básica	Jefe de Proyecto	32,10	4		128,4
1.4.2.1	Estudio de alternativas	Jefe de Proyecto	32,10	8		256,80
1.4.2.2	Plan de pruebas	Tester QA	33,12	8		264,96
1.4.3.1	Diseño de pruebas	Tester QA	33,12	7		231,84
1.4.3.2	Diseño de clases	Jefe de Proyecto	32,10	2		64,20
1.4.4.1	Ejecución y anotación de los resultados de las pruebas	Tester QA	33,12	8		264,96
1.4.4.2	Manual de usuario	Jefe de Proyecto	32,10	24		770,40
1.4.4.3	Generación del Código de los Componentes y Procedimientos	Jefe de Proyecto	32,10	6		192,6
1.4.4.4	Manual de instalación	Jefe de Proyecto	32,10	2		64,20
1.4.5.1	Planificación de riesgos	Jefe de Proyecto	32,10	3		96,30

Sumando todo se obtiene que el coste de realizar este proyecto es de 12.905,42€.

3.2 Ejecución del Proyecto

3.2.1 Bitácora de Incidencias del Proyecto

El proyecto empezó con un problema y es que inicialmente arrancó siendo una colaboración con la empresa INDRA, pero como estaba de prácticas en otra empresa, por tema de protección de datos tuve que hacer el trabajo sin ellos, convirtiendo un proyecto de una pipeline para un proyecto ADA en un manual de una infraestructura Jenkins sobre distintas pipelines para distintos proyectos de diferentes características. Aun así el objetivo del proyecto sigue siendo el mismo pero para tres plataformas diferentes en vez de uno.

Durante el proyecto me he encontrado con varios problemas que han ralentizado el desarrollo de este.

- **Incompatibilidad de versiones**: No se estaba generando los reportes de la cobertura de código y resultó ser un problema de incompatibilidad de versiones entre el plugin y el proyecto.

- **No se ejecutaban los test con Jest en la pipeline del proyecto de Node.js**: Esto se debe a que se estaba instalando una versión de Jest distinta a la que usaba en el proyecto por lo que generaba incompatibilidades. Esto fue porque la versión de Node.js que se estaba instalando en Jenkins era la 20.14 y la del ordenador la 17.9. Se decidió actualizar la versión de Node.js del ordenador a la 20.14 para que sea la misma.
- **No encontraba Selenoid el archivo browser.json**: Selenoid no encontraba el archivo browser.json que contiene los navegadores con sus distintas opciones. Esto fue que el volumen del contenedor no estaba bien montado y fue algo que no llevó mucho tiempo solucionar.
- **Los test de selenium no estaban funcionando en selenoid, pero si en el navegador local**: Se encontró que esto se debe a que los test de selenium según la versión que se usa son compatibles para algunas versiones de los navegadores o no. En este caso se estaba usando una versión compatible con la versión del ordenador de desarrollo (Chrome 80), pero el navegador que se estaba usando en Selenoid era más nuevo (Chrome 119). Se tuvieron que cambiar los test de nuevo ya que los métodos de Selenium que se estaban usando quedaron obsoletos con la nueva versión.
- **Uso de Docker desde los agentes de Jenkins**: No se podía usar docker en los agentes de Jenkins porque los agentes no tienen los permisos para ejecutarlo. Se probaron muchas opciones para que estos tuvieran los permisos y la única solución que se encontró fue ejecutar en los contenedores de los agentes un comando para darle los permisos cada vez que estos se inicien. Esto también llevó tiempo encontrar una solución.
- **Problemas con los plugins de Jenkins**: Durante el desarrollo de repente dejo de funcionar las pipelines, y al entrar a configuración me salían muchas advertencias de actualizaciones de plugins, así que probé a actualizar todo a ver si se arreglaba. Durante el proceso de actualizar me daba muchos errores para actualizar los distintos plugins, pero descubrí que si reiniciabas Jenkins a veces si te instalaban así que reiniciaba Jenkins hasta que ya se actualizaron todos y ya se me ejecutaban correctamente las pipelines.
- **Problema de análisis de código en el proyecto .NET**: Hubo un problema al hacer el análisis de código con SonarQube en el que no se mostraba apenas datos, solo los de la carpeta de los test. No mostraba ningún dato de análisis, pero se podía ver que ficheros había analizado. Después de mucha investigación vi que el problema era que tenía una dependencia xUnit en la solución de la aplicación web haciendo que SonarQube detectara que esa solución era la de los tests y no la otra solución donde sí que están. Así que quité esa dependencia y ya hacía el análisis correctamente.

Capítulo 4 Análisis del sistema de información

4.1 Requisitos del sistema

RS1: La infraestructura será desplegada mediante un docker-compose.yml.

RS1.1: El servicio Jenkins como servidor de automatización.

RS1.1.1: Se usará la imagen *jenkins/jenkins:lts*.

RS1.1.2: Se montará un volumen local de persistencia de datos *./jenkins_home:/var/jenkins_home*

RS1.1.3: Se montará el socket Docker */var/run/docker.sock:/var/run/docker.sock*.

RS1.1.4: Se montará el binario Docker */usr/bin/docker:/usr/bin/docker:ro* para que el contenedor Jenkins pueda invocar Docker del host.

RS1.1.5: Se expondrá el puerto 8080 para el acceso web.

RS1.2: Los tres agentes tendrán su propia configuración

RS1.2.1: El agente para el proyecto Spring tendrá instalado Docker y Azure CLI.

RS1.2.2: El agente para el proyecto Node.js tendrá instalado Docker, Azure CLI y libcurl4.

RS1.2.3: El agente para el proyecto .NET tendrá instalado .NET 8 SDK, dotnet-sonarscanner, Docker y Azure CLI.

RS1.3: El contenedor SonarQube se desplegará con:

RS1.3.1: Volumen *sonarqube_data:/var/lib/sonarqube/data* (persistencia)

RS1.3.2: Volumen *sonarqube_extensions:/opt/sonarqube/extensions*.

RS1.3.3: Exposición del puerto 9000 para acceso web.

RS1.3.4: Conexión a PostgreSQL ≥ 12 a través de la red Docker interna

RS1.4: El contenedor PostgreSQL se desplegará con:

RS1.4.1: Volumen *postgres_data:/var/lib/postgresql/data* (persistencia).

RS1.4.2: Red interna compartida con SonarQube.

RS2: Se usará docker-compose-selenoid.yml para desplegar Selenoid y su interfaz de usuario

RS2.1: El contenedor de Selenoid se desplegará con:

RS2.1.1: Volumen *selenoid_config:/etc/selenoid* que contenga browsers.json con imágenes de navegador (Chrome 119 y Firefox latest)

RS2.1.2: Exposición del puerto 4444 para la API de WebDriver

RS3: Cada proyecto (Spring, Node.js, .NET) tendrá su propio Jenkinsfile.

RS3.1: El Jenkinsfile de Java utilizará Maven

RS3.2: El Jenkinsfile de Node.js utilizará Node (npm)

RS3.3: El Jenkinsfile de .NET utilizará dotnet

RS4: Cada etapa de cada pipeline tendrá una forma distinta de ejecutarse

RS4.1: En la etapa "Instalación de dependencias", el agente de Jenkins debe ejecutar:

- *mvn install* para proyecto Spring.
- *npm install* para Node.js.
- *dotnet restore* para .NET.

RS4.2: En la etapa "Compilación", el agente debe invocar:

- *mvn compile* (Spring).
- *npx eslint* .(Node.js).
- *dotnet build* (.NET)

RS4.3: En la etapa "Ejecución de pruebas unitarias", el agente debe ejecutar:

- JUnit para Spring.
- Mocha para Node.js.
- xUnit para .NET.

RS4.4: En la etapa "Análisis estático de código", el agente debe invocar el plugin SonarQube Scanner en Jenkins, apuntando a la instancia de SonarQube (URL y token configurados) y se ejecutará:

RS4.4.1: *mvn sonar:sonar* (Spring).

RS4.4.2: *npm run sonar* (Node.js)

RS4.4.3: dotnet SonarScanner.NetCore (.NET)

RS4.5: En la etapa "Reporting" se creará un report de los resultados de las pruebas de la aplicación mediante Allure.

RS4.6: En la etapa "Despliegue en Azure", se usará el Azure CLI (plugin Azure para Jenkins) con un Service Principal almacenado como credencial en Jenkins

RS4.6.1: Se realizará el despliegue a Azure App Service.

RS5: Jenkins tendrá instalados y configurados los siguientes plugins:

- Pipeline: Shared Groovy Libraries
- Maven Integration Plugin
- Allure Jenkins Plugin

- JaCoCo Plugin
- Docker Pipeline
- SonarQube Scanner for Jenkins
- Azure CLI Plugin
- NodeJS Plugin
- Badge
- Pipeline Stage View Plugin
- HTML Publisher Plugin
- Pipeline: Declarative, Pipeline: Git, GitHub Branch Source
- Credentials Binding Plugin

RS6: El plugin Slack Notification enviará mensajes a un canal configurado mediante un incoming Webhook URL

RS6.1: Los mensajes incluirán:

- Estado de la pipeline
- Nombre de la etapa donde ocurra un fallo

4.2 Plan de Pruebas

A continuación, se presenta el plan de pruebas, el cual trataremos los objetivos y alcance, descripción del entorno y la estrategia de pruebas.

4.2.1 Objetivos y alcance del plan de pruebas

- Validar el correcto comportamiento de la pipeline.
- Verificar la integración de las distintas herramientas (Jenkins, Docker, SonarQUbe, Azure, Selenoid).
- Comprobar que el proceso de despliegue se realice correctamente.
- Asegurar la correcta notificación y registro de errores generando reportes.
- Asegurar el manejo adecuado de incidencias críticas:
  - Si los tests fallan, la pipeline continúa en estado "unstable" para permitir análisis y feedback sin bloquear el flujo.
  - Si falla la construcción de la aplicación, la pipeline se detiene y se marca como "failed", evitando el despliegue de versiones defectuosas.

4.2.2 Descripción del Entorno

- **Jenkins**: Ejecutado en un contenedor Docker, orquestando la ejecución de pipelines.
- **Agentes**: Tres contenedores independientes para cada uno de los proyectos (Spring, Node.js y .NET).
- **Servicios Auxiliares**:
  - **SonarQube**: Análisis de calidad de código.
  - **Selenoid**: Ejecución de pruebas de front.

- **Integraciones Externas**:
  - **Azure**: Entorno de despliegue final.
  - **Slack**: Plugin para notificaciones.

4.2.3 Estrategia de pruebas

- **Pruebas Unitarias**:
  - Se verificará el funcionamiento del plugin de Slack mediante pruebas automatizadas, utilizando los frameworks Junit y Mockito.
  - Se verificará el funcionamiento de las aplicaciones de ejemplo mediante test automatizados. Para cada aplicación se usarán distintos frameworks:
    - Spring: Junit
    - Node.js: Mocha
    - .NET: XUnit, Moq
- **Pruebas de Integración**:
  - Se asegurará la correcta comunicación entre Jenkins, los agentes y otros sistemas externos, como SonarQube y Selenoid. Se diseñará un script para que de forma automática se ejecuten comandos que comprueben la conexión entre contenedores.
  - En cada aplicación de ejemplo se verificará la interacción de sus componentes mediante pruebas automáticas. Cada aplicación usará distintos frameworks para los test.
    - Spring: Junit, Spring MVC
    - Node.js: Mocha
    - .NET: XUnit, Moq
- **Pruebas de la Interfaz de Usuario**: Se comprobará la funcionalidad, usabilidad y comportamiento de las interfaces de usuario de cada aplicación de ejemplo. En los tres proyectos se usará el framework Selenium WebDriver.
- **Pruebas de Sistema**: Se evaluará el flujo completo de cada pipeline, asegurando la correcta ejecución de todas las etapas, desde el build hasta el despliegue de la aplicación. Estas pruebas se llevarán a cabo de forma manual en distintos escenarios.

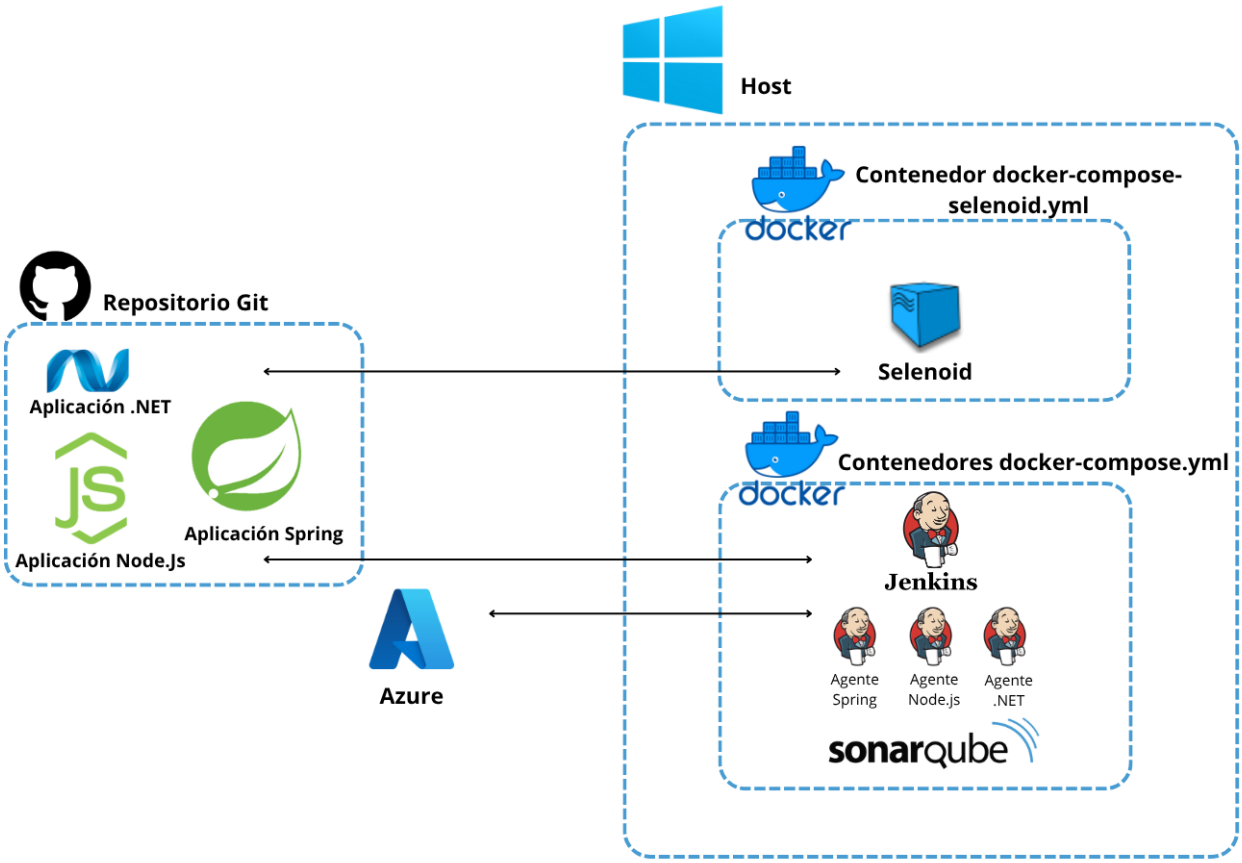
Capítulo 5 Diseño del Sistema de Información

5.1 DSI 1: Diseño de la Arquitectura de Módulos del Sistema

5.1.1 DSI 1.1 Diseño de Módulos del Sistema

Los módulos están estructurados de la siguiente manera, Jenkins y sus agentes, Sonarqube, PostgreSQL y Selenoid son contenedores de Docker que están conectados por la misma red. Las aplicaciones de Spring, Node.js y .NET están en un repositorio de GitHub. Jenkins está conectado mediante un Webhook con el repositorio de GitHub para detectar cada cambio en el repositorio y lanzar la pipeline.

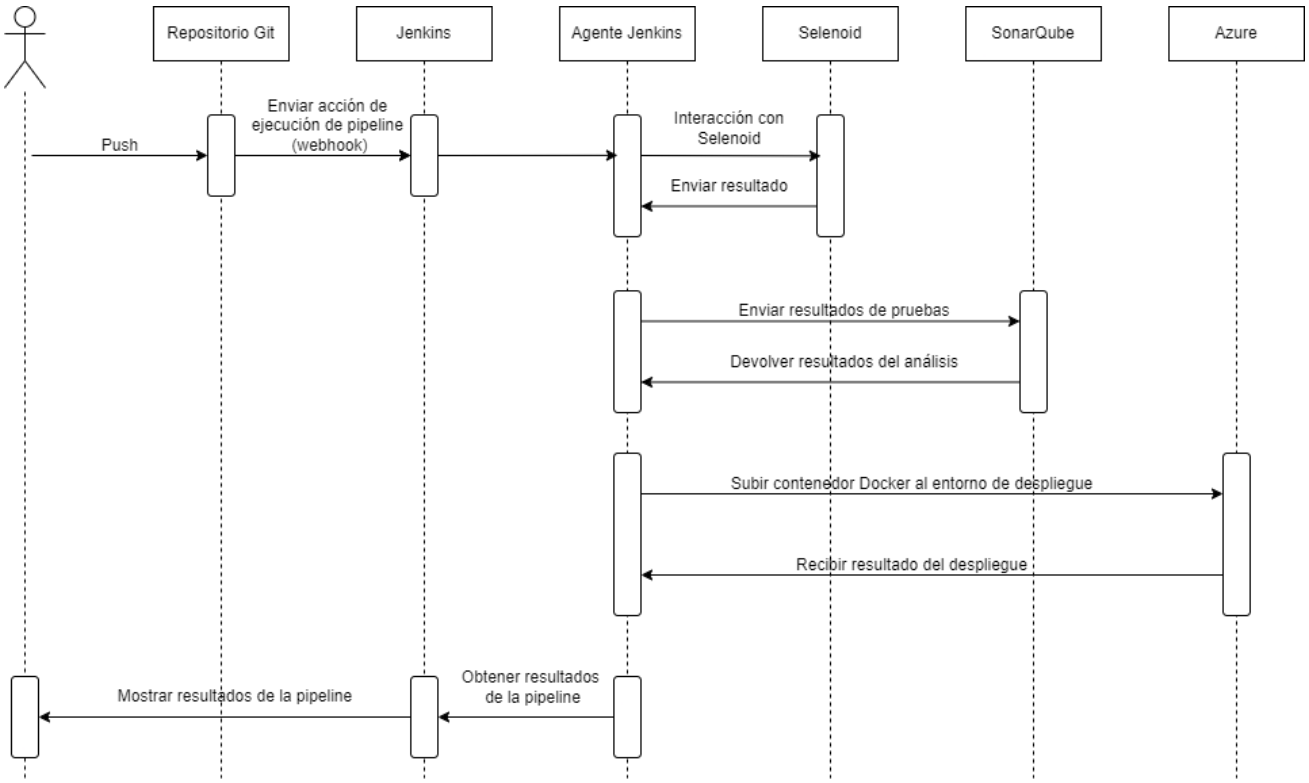
A continuación, se expone un diagrama con la vista general del sistema:



5.1.2 DSI 1.2: Diseño de Comunicaciones entre Módulos

En nuestro sistema, el repositorio Git una vez recibe un nuevo cambio, lanza a través de un Webhook una petición a Jenkins para que inicie la pipeline. El nodo principal una vez recibe la petición de ejecutar la pipeline llama a un agente asignado (si lo hay) para que la ejecute. Este agente ejecuta los test de los distintos proyectos, se comunica con el contenedor de Selenoid para ejecutar las pruebas del front, llama a SonarQube para analizar estáticamente el código y guardar en PostgreSQL los datos del análisis. Finalmente se sube la aplicación a un servidor de Azure y se envía el estado de la pipeline al repositorio de GitHub.

A continuación, se muestra un diagrama de secuencia de nuestro sistema:



5.2 DSI 2: Diseño de Casos de Uso Reales

En esta sección se presenta el caso de uso que proporciona una visión general del comportamiento del sistema. A través de este diagrama se ilustran los distintos pasos que componen la ejecución de una pipeline típica.

5.2.1 Caso de Uso

5.3 DSI 3: Diseño de Clases

En este apartado se hablará del diseño del plugin hecho para nuestro sistema y de las aplicaciones creadas para la pipeline.

5.3.1 SlackNotificationPlugin

Este proyecto está compuesto por dos clases, una es la que contiene los métodos y atributos que usa Jenkins para ejecutar en la pipeline (NotificationBuilder) y la otra clase es la que contiene la lógica del plugin (NotificationAction).

La clase NotificationBuilder necesita un constructor con unos parámetros si es necesario ya que así es como obtendremos la información necesaria de la pipeline. Un método perfom que se ejecuta en la pipeline y contendrá el código necesario para ejecutar la lógica de nuestro plugin. Por último, dentro habrá una clase estática que hereda de BuildStepDescriptor<Builder> y que contendrá dos métodos, isApplicable() que nos devuelve si se puede usar el plugin o no, y getDisplayName que es el nombre del Plugin.

La clase NotificationAction tiene un constructor con unos parámetros para obtener la información necesaria de la clase NotificationBuilder, los métodos get de cada atributo y un método sendNotification con la lógica del plugin.

5.3.2 Proyectos Spring, Node.js, .NET

Para este proyecto se ha desarrollado tres proyectos básicos con distintas tecnologías con el fin de probar nuestro sistema en entornos distintos. Una de las aplicaciones es una aplicación Spring, otra Node.js y finalmente una aplicación .NET donde todos los proyectos siguen el patrón de diseño MVC (Modelo Vista Controlador).

- La aplicación Spring es un inventario de productos.
- La aplicación Node.js es una lista de tareas.
- La aplicación .NET es un catalogo de películas.

Aunque las temáticas son diversas, las funcionalidades básicas son similares en las tres aplicaciones, ya que todas permiten crear, eliminar, editar y visualizar registros (películas, productos o tareas, según corresponda).

La estructura de cada proyecto se organiza en tres carpetas, correspondientes a las capas del patrón MVC:

- **Modelo**: Contiene la definición de los objetos y, en proyectos más complejos, incluiría la lógica de negocio. En este caso, dada la simplicidad de los proyectos, la lógica implementada es mínima.
- **Controlador**: Reúne los métodos que conectan el modelo con la vista, gestionando las operaciones de creación, actualización, eliminación y consulta de los registros.
- **Vista**: Agrupa los archivos encargados de la presentación, tales como los formularios para crear, actualizar y visualizar la información.

5.4 DSI 4: Diseño Físico de Datos

Para este proyecto lo que tenemos es una base de datos que guarda la información de los resultados del análisis en SonarQube.

5.4.1 Descripción del SGBD Usado

SonarQube necesita una base de datos donde almacenar toda la información pero no todas son compatibles con SonarQube. Las bases de datos compatibles con SonarQube incluyen Microsoft SQL Server, Oracle y PostgreSQL.

En este proyecto se ha optado por PostgreSQL por la familiaridad con esta tecnología.

5.4.2 Integración del SGBD en Nuestro Sistema

En este proyecto PostgreSQL se encuentra en un contenedor Docker que se levanta junto con SonarQube mediante un archivo Docker Compose. Para integrar ambos servicios es necesario:

- Asignar un nombre a la base de datos, un usuario y una contraseña. Esto se define utilizando la etiqueta environment donde, especificaremos estos datos en POSTGRES_DB como nombre de la base de datos, POSTRGRES_USER como el nombre de usuario Y POSTGRES_PASSWORD como la contraseña.
- Proporcionar la url de la base de datos y el usuario y contraseña a SonarQube. Esto se hace a través de la etiqueta environment donde SONAR_JDBC_URL es la url de la base de datos, SONAR_JDBC_USERNAME es el nombre de usuario y SONAR_JDBC_PASSWORD es la contraseña.

```
sonarqube:
  image: sonarqube:9.9.3-community
  depends_on:
   - db
  ports:
   - "9000:9000"
  networks:
   - backend
  environment:
   SONAR_JDBC_URL: jdbc:postgresql://db:5432/sonarqube
   SONAR_JDBC_USERNAME: sonar
   SONAR_JDBC_PASSWORD: sonar
  deploy:
   resources:
    limits:
     cpus: "1.0"
     memory: "1GB"
  volumes:
   - sonarqube_data:/opt/sonarqube/data
   - sonarqube_extensions:/opt/sonarqube/extensions
   - sonarqube_logs:/opt/sonarqube/logs
   - sonarqube_temp:/opt/sonarqube/temp
  restart: on-failure
  container_name: sonarqube
 db:
```

```
image: postgres:12-bullseye
networks:
 - backend
environment:
 POSTGRES_DB: "sonarqube"
 POSTGRES_USER: sonar
 POSTGRES_PASSWORD: sonar
deploy:
 resources:
  limits:
   cpus: "1.0"
   memory: "256MB"
volumes:
 - postgresql:/var/lib/postgresql
 - postgresql_data:/var/lib/postgresql/data
restart: on-failure
container_name: postgresql
```

5.5 DSI 5: Especificación Técnica del Plan de Pruebas

Este apartado contendrá el diseño e implementación de las pruebas que comprueben el funcionamiento del sistema de acuerdo con el plan de pruebas. Debido a la complejidad de las pruebas, el diseño y la implementación no estarán separadas.

Anotación:

- *Los links que hay a continuación son accesos rápidos al archivo o directorio mencionado en el repositorio Git del proyecto.*

5.5.1 Pruebas Unitarias

Pruebas Unitarias Plugin Slack

ID	Caso de Prueba	Entrada	Resultado Esperado
1	Crear el objeto builder	URL, Channel, Message	Objeto NotifcationBuilder creado correctamente
2	Enviar mensaje por Slack	URL válido, Channel válido, Message	Mensaje enviado correctamente

(Codigo fuente de las pruebas)

Pruebas Unitarias en el proyecto Spring

ID	Caso de Prueba	Entrada	Resultado Esperado
1	Crear producto con un ID especifico	ID = 125	Producto creado con ID 125
2	Crear producto con un nombre	Titulo = “Producto de Prueba”	Producto creado con el nombre “Producto de Prueba”

3	Crear producto con un precio	Precio = 10.5	Producto creado con un precio de 10.5

(Código fuente de las pruebas)

Pruebas Unitarias en el proyecto .NET

ID	Caso de Prueba	Entrada	Resultado Esperado
1	Crear una pelicula	ID = 1, titulo = “Action Movie”, género = “Action”	Película creada con id 1, nombre “Action Movie” y género “Action”
2	Obtener película por ID	ID = 1	Devuelve una instancia de la película con el ID 1
3	Editar el titulo de una película	ID = 1, titulo = “Updated title”, genero = “Action”	Devuelve película editada con el título “Updated title”
4	Borrar película por ID	ID = 1	No está la película dentro de la lista de películas.

(Código fuente de las pruebas)

5.5.2 Pruebas de integración

Pruebas de integración - Conexión entre contenedores

ID	Caso de Prueba	Procedimiento de prueba	Resultado Esperado
1	Validación de comunicación entre contenedores en estado normal	Se ejecutará un script que verifique la conexión de los contenedores SonarQube, Agentes y Selenoid con Jenkins	Respuesta obtenida desde el contendor Jenkins por parte de las demás herramientas: SonarQube, Agentes y Selenoid.
2	Detección de contenedores caídos	Se ejecutara un script en el muestre los contenedores apagados, en esta prueba se apagará el de SonarQube	Error de conexión o timeout.
3	Recuperación de un contenedor caído	Se ejecutará un script que encienda los contenedores apagados, en este caso el de SonarQube.	Respuesta obtenida por el contenedor de SonarQube

(Código fuente de las pruebas)

Pruebas de Integración en el proyecto de Spring

ID	Caso de Prueba	Entrada	Resultado Esperado
1	Actualizar un producto	Nombre = “Producto Actualizado”, precio = 25.0	La película se actualiza y se devuelve un código 200

2	Borrar un producto	ID del primer producto en la lista	La película se borra y se devuelve un código 200

(Código fuente de las pruebas)

Pruebas de Integración en el proyecto de Node.js

ID	Caso de Prueba	Entrada	Resultado Esperado
1	Crear una tarea	Tarea = “Task 1”	La tarea se crea y devuelve estado 200
2	Borrar una tarea	ID de la tarea	Mensaje “Task eliminado correctamente” y estado 200
3	Borrar una tarea no existente	ID no existente	Error 404

(Código fuente de las pruebas)

Pruebas de Integración en el proyecto de .NET

ID	Caso de Prueba	Entrada	Resultado Esperado
1	Acceder al inventario de películas		Estado 200
2	Detalles de la película con ID 1	ID = 1	Titulo de la pelicula es “Test Movie 1” y estado 200
3	Editar película con ID 1	ID = 1, Title = "Updated Test Movie 1", ReleaseDate = DateTime.Now, Genre = "RW", Price = 10.99M, Rating = "PG"	Película editada y estado 200

(Código fuente de las pruebas)

5.5.3 Prueba de Interfaz de Usuario

Pruebas Interfaz de Usuario en el proyecto de Spring

ID	Caso de Prueba	Entrada	Resultado Esperado
1	Crear producto	Botón crear producto, Nombre = “NuevoProducto”, precio = 20	El producto aparece en la lista de productos
2	Editar producto	Botón editar, Nombre = “ProductoAELiminar”, precio = 15	El producto aparece actualizado en la lista
3	Eliminar producto	Botón eliminar	El producto ha sido eliminado correctamente

(Código fuente de las pruebas)

Pruebas Interfaz de Usuario en el proyecto de Node.js

ID	Caso de Prueba	Entrada	Resultado Esperado
1	Crear tarea	Botón crear tarea, Tarea = “New Task”	La tarea aparece en el listado de tareas
2	Borrar tarea	Botón eliminar	La tarea ha sido eliminada correctamente

(Código fuente de las pruebas)

Pruebas Interfaz de Usuario en el proyecto de .NET

ID	Caso de Prueba	Entrada	Resultado Esperado
1	Navegar a la ventana principal		Encontrar el titulo “Index”, el botón “Create New” y el filtro de películas
2	Crear película	Botón crear pelicula, Title = “Test Movie”, ReleaseDate= “01/02/2024”, Genre = “Test Genre”, Price = 10, Rating = PG”	
3	Ver detalles de la primera película	Botón detalles	EL titulo de la ventana es “Details”.
4	Editar película	Botón editar película, Title = “Updated Test Movie”	La película aparece en el listado editada.
5	Borrar película	Botón borrar	La película ha sido borrada correctamente

(Código fuente de las pruebas)

Para la ejecución de los test de la interfaz de usuario de forma local es necesario cambiar la url de la conexión remota a Selenoid de http://host.docker.internal:4444/wd/hub por http://localhost:4444/wd/hub. El primer URL se necesita para que los test que se están ejecutando desde un contenedor Docker (agente Jenkins) se puedan conectar al contenedor Selenoid.

5.5.4 Pruebas de sistema

Ahora se describirá los casos de prueba del sistema con el procedimiento que se ha de seguir para realizarse. Cada caso de prueba se replicará para las tres aplicaciones (Spring, Node.js y .NET)

Anotación:

- *Después de cada procedimiento hay que:*
  1. *Subir los cambios al repositorio Git*
  2. *Ejecutar pipeline*
  3. *Una vez finalizado el test revertir el commit usado para la prueba*

5.5.4.1 Ejecución correcta de las tres pipelines

Caso de prueba: Ejecución correcta de la pipeline
Procedimiento: Lanzar la ejecución de la pipeline de cada proyecto con todos los servicios activados (SonarQube, Selenoid y Azure).
Resultado esperado: Ejecución correcta de la pipeline y despliegue de la aplicación
(Link al Jenkinsfile del proyecto Spring)
(Link al Jenkinsfile del proyecto Node.js)
(Link al Jenkinsfile del proyecto .NET)

5.5.4.2 Test falla

Caso de prueba: Los test de la aplicación fallan
Procedimientos:

Proyecto	Pasos
Spring	1. Provocar fallo en el test de integración, cambiar: Matchers.*hasProperty*("precio", Matchers.*is*(25.0)) por Matchers.*hasProperty*("precio", Matchers.*is*(20.0)) (Link del archivo que se cambió) 2. Apagar contenedor de Selenoid para que el test de UI no se pueda conectar.
.NET	1. Provocar fallo en el test de integración, cambiar: Assert.Equal("Updated Test Movie 1", movieInDb.Title); por Assert.Equal("Updated Test Movie 2", movieInDb.Title); (Link del archivo que se cambió) 2. Provocar fallo en el test de UI, cambiar: Assert.IsTrue(driver.PageSource.Contains("Test Movie")); por Assert.IsTrue(driver.PageSource.Contains("Test Moviess")); (Link del archivo que se cambió)
Node.js	1. Provocar fallo en el test de integración, cambiar: expect(res.text).to.contain('Task 1'); por expect(res.text).to.contain('asdjklnsajkdsa'); (Link del archivo que se cambió)

Resultado esperado:

1. El estado de la pipeline es “unstable”
2. Se han ejecutado todas la etapas
3. El report se ha generado.

5.5.4.3 No pasar las reglas de sonar

Caso de prueba: Fallo de la Quality Gate

Procedimientos:

Proyecto	Pasos
Spring	1. Añadir al código **n=-2** para introducir un code smell 2. Comentar algunos tests para reducir la cobertura de código

.NET	1. Eliminar la validación de un modelo:
	`if (!ModelState.IsValid)`
	`{`
	`return View(movie);`
	`}`
	(Link del archivo que se cambió)

Node.js	1. Cambiar *const* o *let* por *var* en el código: const {ObjectId} = require("mongodb"); por var {ObjectId} = require("mongodb"); (Link del archivo que se cambió)

Resultado esperado:

1. El estado de la pipeline es “unstable”
2. Se han ejecutado todas las etapas
3. El Quality Gate está en estado “Failed”
4. SonarQube ha reportado las issues esperadas

Sobre las reglas de SonarQube y el Quality Gate

SonarQube tiene unas reglas por defecto que nos dice si nuestro código cumple con la metodología Clean Code y por ende el Quality Gate. El umbral por defecto que viene definido en SonarQube, es el que se usará en las pruebas y es el siguiente:

- No hay ninguna issue que sea critico o bloqueante. Las issues pueden ser:
  - Code Smell: Los Code Smell son indicadores de problemas potenciales que nos pueden dificultar el mantenimiento del código en un futuro.
  - Bug: Error o defecto en el software que hace que un programa funcione incorrectamente.
  - Vulnerability: Un agujero de seguridad que puede usarse para atacar nuestro software.
- Que el code coverage de las pruebas sea mayor al 80%.
- El número de líneas duplicadas es menor al 3%
- La deuda técnica es menor de un 5%
- Que no haya ningún Security Hotspot.

5.5.4.4 Fallo en el Build

Caso de Prueba	Procedimiento	Resultado esperado

Fallo en la construcción de la aplicación **Spring**	Introducir un error de compilación (eliminar “;”)	Ejecución incompleta de la pipeline con estado “failed”
Fallo en la construcción de la aplicación **.NET**	Eliminar una sentencia return en un método que espera devolver un valor	Ejecución incompleta de la pipeline con estado “failed”
Fallo en la construcción de la aplicación **Node**	Quitar un paréntesis de cierre “)” en el código	Ejecución incompleta de la pipeline con estado “failed”

5.5.4.5 Fallo en el despliegue

Caso de Prueba	**Procedimiento**	**Resultado esperado**
Fallo en el despliegue de la aplicación Spring	Modificar tenant con un valor incorrecta	Fallo en el despliegue de la aplicación
Fallo en el despliegue de la aplicación .NET	Modificar el subsciption ID con un valor incorrecto	Fallo en el despliegue de la aplicación
Fallo en el despliegue de la aplicación Node.js	Modificar el subsciption ID con un valor incorrecto	Fallo en el despliegue de la aplicación

Capítulo 6 Construcción del Sistema de Información

6.1 CSI 1: Preparación del Entorno de Generación y Construcción

6.1.1 Lenguajes de programación y tecnologías

En este apartado se especificará sobre los lenguajes y tecnologías usadas en el desarrollo del sistema

6.1.1.1 Sistema de pipeline

Nombre	Versión	Clasificación
YAML	-	Lenguaje
Jenkins	latest	Tecnología
SonarQube	25.1.0.102122-community	Tecnología
Selenoid	latest-release	Tecnología
PostgesSQL	12-bullseye	Tecnología
Docker	28.0.1	Tecnología
Visual Studio Code	1.100	Herramienta

Para la construcción de la infraestructura de la aplicación se ha creado un Docker compose escrito en YAML donde indicaremos los contenedores Docker a construir (Jenkins, SonarQube, Selenoid, PostgreSQL).

6.1.1.2 Aplicaciones y plugin

Nombre	Versión	Clasificación
Java	17	Lenguaje
JavaScript	-	Lenguaje
C#	-	Lenguaje
Node.js	20.14.0	Tecnología
Spring	-	Tecnología

.NET	8.0	Tecnología
IntelliJ	Ultimate Edition 2023.2.5	Herramienta
Visual Studio	2022	Herramienta

Para este proyecto se han desarrollado tres aplicaciones simples, un proyecto Spring que usa Java, un proyecto Node.js desarrollado en JavaScript y un proyecto .NET desarrollado en C#. El plugin fue desarrollado en Java. Para los proyectos de Node.js, Spring y el plugin se ha usado IntelliJ como IDE para el proyecto .NET se ha usado Visual Studio Community ya que este IDE está muy bien integrado con esta tecnología ya que ambos pertenecen a Microsoft.

6.1.1.3 Documentación y control de versiones

Documentación	**Microsoft Word, Microsoft Project, Draw.io**
Control de versiones	OneDrive, Git, GitHub

Para la redacción de la memoria y la documentación se ha usado Microsoft Word. El WBS de la planificación se usó Microsoft Proyect y para los diagramas de la memoria se usó draw.io.

Se usó GitHub como control de versiones de las aplicaciones y de la infraestructura. Mientras que OneDrive se usó como control de versiones de la memoria.

6.2 CSI 2: Generación del Código de los Componentes y Procedimientos

En este apartado se mostrará la estructura del proyecto, así como de las clases más relevantes del proyecto.

6.2.1 Estructura del proyecto

A continuación, se muestra una representación de la estructura del proyecto:

AutoFlow/ (Link del repositorio)

- **TiendaNET/**
  - TiendaOnline (Project)
  - TestTiendaOnline (Tests)
- **SpringPMA/**
  - src/main (Project)
  - src/test (Tests)
- **TodoListNodeJs/**
  - routes, views, models (Project)

  - test (Tests)
- **SlackNotificationPlugin/**
  - src (Project)
  - test (Tests)
- **DockerContainers/**
  - docker-compose.yml
  - agentNode
  - agentSpring
  - agentNet

jenkins_lib/ (Link del repositorio)

- **vars/**

A continuación, se presentará las clases más relevantes de cada aplicación así como sus funciones. También se detallará infraestructura de los contenedores de Jenkins y SonarQube y la librería.

6.2.2 TiendaNET

Esta carpeta contiene la aplicación web desarrollado en .NET con los test. Esta carpeta contiene dos soluciones, una es la aplicación web y otra los tests. La aplicación sigue el patrón MVC teniendo una estructura clara de separación de responsabilidades entre modelos, vistas y controladores. Por la simpleza de la aplicación solo hay tres archivos para los test, uno por cada separación de responsabilidades. Los frameworks usados para las pruebas son: xUnit, Moq y Selenium WebDriver y para el reporting se usa Allure. Al ser un proyecto pequeño de ejemplo para probar nuestra infraestructura no hay muchos ficheros ni carpetas. A continuación, se muestra la estructura del proyecto:

- **TiendaOnline/ -** (Aplicación principal)
  - **Controllers/**
    - **MoviesController.cs** - Controlador principal para operaciones CRUD de películas
  - **Models/**
    - **Movie.cs** - Modelo de datos con propiedades Id, Title, Genre, ReleaseDate, Price, Rating
    - **ErrorViewModel.cs** - Modelo para manejo de errores
  - **Data/**
    - **MvcMovieContext.cs** - Contexto de Entity Framework Core para acceso a base de datos
  - **Views/**
    - **Movies/**
      - **Index.cshtml** - Vista principal con lista de películas y filtros
      - **Create.cshtml** - Formulario para crear películas
      - **Edit.cshtml** - Formulario para editar películas
      - **Details.cshtml** - Vista de detalles de película

        - **Delete.cshtml** - Confirmación de eliminación
    - **wwwroot/** (Archivos estáticos)
        - **css/** - Hojas de estilo
        - **js/** - Scripts JavaScript
        - **lib/** - Librerías de terceros (Bootstrap, jQuery)
    - **Program.cs** - Punto de entrada y configuración de la aplicación
    - **TiendaOnline.csproj** - Archivo de proyecto con dependencias y configuración
- **TestTiendaOnline/** (Proyecto de pruebas)
    - **UnitTest.cs** - Pruebas unitarias del controlador con Moq y Allure
    - **FrontTest.cs** - Pruebas de interfaz con Selenium WebDriver
    - **MoviesControllerTests.cs** - Pruebas de integración del controlador
    - **TestTiendaOnline.csproj** - Configuración del proyecto de pruebas con dependencias de testing
    - **TestResults/** - Directorio para resultados de pruebas y cobertura
- **TiendaOnline.sln** - Archivo de solución que incluye ambos proyectos
- **Jenkinsfile** - Pipeline de CI/CD para compilación, pruebas y despliegue

Se detalla a continuación algunas de clases del proyecto.

6.2.2.1 Clase MoviesController

Esta es la clase del controlador que maneja las operaciones CRUD para películas.

Atributos:

- **_context**: Es una instancia de **MvcMovieContext** que proporciona acceso a la base de datos.

Métodos principales:

- **Index(string searchString, string movieGenre)**: Método que redirige a la vista donde se muestra la lista de películas que puede estar filtradas por título y género.
- **Details(int? id)**: Redirige a la vista donde se muestran los detalles de una película específica.
- **Create()**: Muestra el formulario para crear una nueva película.
- **Create(Movie movie)**: Hace el procesa de la creación de una nueva película.
- **Edit(int? id)**: Muestra el formulario para editar una película existente.
- **Edit(int id, Movie movie)**: Procesa los datos para editar una película existente.
- **Delete(int? id)**: Muestra por pantalla la confirmación para eliminar una película.
- **DeleteConfirmed(int id)**: Método que elimina una película.
- **MovieExists(int id)**: Verifica si una película existe en la base de datos.

6.2.2.2 Clase MvcMovieContext

Esta clase es el contexto de bases de datos de Enity Framework Core que define la conexión con la base de datos y los conjuntos de entidades.

6.2.3 SpringPMA (Proyecto Spring)

Esta carpeta contiene la aplicación web desarrollada en Java con Spring Boot para la gestión de productos, junto con sus pruebas. La aplicación sigue el patrón MVC de Spring Boot con una estructura clara de separación de responsabilidades entre modelos, controladores y vistas. Los frameworks usados para las pruebas son JUnit, Selenium y para el reporting se usa Allure con JaCoCo para cobertura de código. Al ser un proyecto pequeño de ejemplo para probar nuestra infraestructura no hay muchos ficheros ni carpetas

A continuación, se muestra la estructura del proyecto:

SpringPMA/- (Aplicación principal)

- **src/main/java/com/example/SpringPMA/**
  - **SpringPMAApplication.java** - Punto de entrada principal de la aplicación Spring Boot
  - **model/**
    - **Producto.java** - Modelo de datos con propiedades id, nombre, precio
  - **controller/** - Controladores para manejar las operaciones CRUD de productos
  - **service/** - Lógica de negocio de la aplicación
- **src/main/resources/**
  - **templates/productos/** - Plantillas Thymeleaf para las vistas
    - **nuevo.html** - Formulario para crear nuevos productos
  - **static/** - Archivos estáticos (CSS, JS)
- **src/test/java/com/example/SpringPMA/**
  - **controller/**
    - **ProductoControllerTest.java** - Pruebas de integración del controlador
  - **view/**
    - **ProductoViewTest.java** - Pruebas de interfaz con Selenium WebDriver
  - **SpringPMAApplicationTests.java** - Pruebas de la aplicación principal
- **pom.xml** - Archivo de configuración de Maven con dependencias y plugins.

Como se pude observar no hay test para el modelo, esto se decidió así por la simpleza del modelo ya que solo habría que testear getters y setters de los atributos.

Se detalla a continuación algunas de clases del proyecto.

6.2.3.1 Clase ProductoController

Esta es la clase del controlador en el que se maneja operaciones CRUD para gestionar productos.

Métodos principales:

- **obtenerProductos(Model model):** Redirige a la vista donde se muestran los productos.

- **obtenerProductoPorId (@PathVariable Long id, Model model):** Redirige a la pantalla donde se muestran los detalles de un producto.
- **mostrarFormularioNuevoProducto(Model model):** Muestra un formulario para crear un producto.
- **crearProducto(@ModelAttribute Producto nuevoProducto):** Método POST que procesa al creación de un producto.
- **mostrarFormularioEditarProducto(@PathVariable Long id, Model model):** Muestra el formulario para editar un producto.
- **actualizarProducto(@PathVariable Long id, Producto productoActualizado):** Se procesan los datos para edita un producto.
- **eliminarProducto(@PathVariable Long id):** Elimina un producto y redirige a la pantalla de productos.

6.2.4 TodoListNodeJs

Esta carpeta contiene la aplicación web desarrollada en Node.js con Express para la gestión de listas de tareas, junto con sus pruebas. La aplicación sigue el patrón MVC de Express con una estructura clara de separación de responsabilidades entre rutas (controladores), vistas y modelos de datos. Los frameworks usados para las pruebas son Jest, WebdriverIO para pruebas de interfaz y para el reporting se usa Allure. Al ser un proyecto pequeño de ejemplo para probar nuestra infraestructura no hay muchos ficheros ni carpetas

A continuación, se muestra la estructura del proyecto:

TodoListNodeJs/- (Aplicación principal)

- **app.js** - Punto de entrada principal de la aplicación Express
- **routes/**
  - **todo.js** - Controlador principal para operaciones CRUD de tareas
- **views/** - Plantillas Twig para las vistas
  - **todo.twig** - Vista principal con lista de tareas y formulario de creación
  - **index.twig** - Vista alternativa para la lista de tareas
- **public/** - Archivos estáticos
  - **stylesheets/**
    - **style.css** - Hojas de estilo personalizadas
- **tests/** - Proyecto de pruebas
  - **front.test.js** - Pruebas de interfaz con WebdriverIO
  - **task.test.js** - Pruebas unitarias del controlador con Supertest
- **allure-results/** - Directorio para resultados de pruebas con Allure
- **jest.config.js** - Configuración de Jest para pruebas y reporting
- **Jenkinsfile** - Pipeline de CI/CD para compilación, pruebas y despliegue

Se detalla a continuación algunas de clases del proyecto.

6.2.4.1 Todo.js

Archivo donde se definen las rutas y controladores para las operaciones CRUD de tareas.

Rutas principales:

- **GET /todo:** Obtiene y muestra todas las tareas
- **POST /todo:** Crea una nueva tarea
- **GET /todo/delete/:id**: Elimina una tarea especifica

6.2.5 SlackNotificationPlugin

Esta carpeta contiene el plugin personalizado desarrollado en Java para Jenkins que permite la integración con Slack, junto con sus pruebas. El plugin sigue la arquitectura estándar de Jenkins con una estructura clara de separación de responsabilidades entre las clases de acción, constructores y descriptores. Los frameworks usados para las pruebas son JUnit y Mockito para el testing unitario.

A continuación, se muestra la estructura del proyecto:

SlackNotificationPlugin/- (Plugin principal)

- **src/main/java/io/jenkins/plugins/sample/**
  - **NotificationAction.java** - Clase de acción que implementa el envío de notificaciones a Slack
  - **NotificationBuilder.java** - Constructor del paso de pipeline para Jenkins
- **src/test/java/io/jenkins/plugins/sample/**
  - **NotificationTest.java** - Pruebas unitarias del plugin con JUnit y Mockito
- **target/** - Directorio de compilación y resultados
  - **hpi/** - Archivo del plugin empaquetado
- **pom.xml** - Archivo de configuración de Maven con dependencias específicas de Jenkins

Se detalla a continuación algunas de clases del proyecto.

6.2.5.1 Clase NotificationBuilder

Esta clase extiende de Builder y es la clase a la que Jenkins “accede” para enviar los parámetros que le hemos dado. También ejecuta la clase que contiene la lógica.

Atributos:

- **webhookUrl**: URL del webhook de Slack para enviar las notificaciones.
- **channel**: Canal de Slack al que se enviarán las notificaciones.
- **message**: Mensaje que se enviará a Slack.

Métodos principales:

- **NotificationBuilder(String webhookUrl, String channel, String message)**: Constructor que inicializa los atributos.

- **getWebhookUrl()**, **getChannel()**, **getMessage()**: Métodos getter para los atributos.
- **perform(Run<?, ?> run, FilePath workspace, EnvVars env, Launcher launcher, TaskListener listener)**: Método que ejecuta la acción de notificación.

Dentro de esta clase también hay una clase con la anotación @Extension que extiende de BuildStepDescriptor<Builder>. Esta clase es la clase que se muestra en la configuración del plugin en Jenkins. La anotación @Extension es importante para que Jenkins la reconozca.

Está anotada con **@Symbol("simpleSlackNotifier")** para permitir su uso en scripts de pipeline

6.2.5.2 NotificationAction

Esta es la clase que ejecuta la lógica del envío de mensajes por Slack.

Atributos:

- **name**: Nombre de la acción.
- **webhookUrl**: URL del webhook de Slack para enviar las notificaciones.
- **channel**: Canal de Slack al que se enviarán las notificaciones.
- **message**: Mensaje que se enviará a Slack.

Métodos principales:

- **NotificationAction(String webhookUrl, String channel, String message)**: Constructor que inicializa los atributos y envía la notificación.
- **sendNotification(String webhookUrl, String channel, String message)**: Método privado que realiza la conexión HTTP con Slack para enviar la notificación.

6.2.6 DockerContainers

Esta carpeta contiene la infraestructura de contenedores Docker desarrollada para soportar el ecosistema completo de CI/CD, junto con scripts de pruebas y configuraciones. La infraestructura sigue una arquitectura de microservicios con una clara separación de responsabilidades entre servicios de automatización, análisis de calidad y agentes especializados. Los frameworks usados para las pruebas son scripts de Python para verificación de conectividad y estado de contenedores.

A continuación, se muestra la estructura del proyecto:

DockerContainers/- (Infraestructura principal)

- **docker-compose.yml** - Archivo principal que define todos los servicios de la infraestructura CI/CD
- **docker-compse-selenoid.yml** – Archivo que contiene los servicios de selenoid y selenoid-ui
- **Dockerfile** - Imagen base para Jenkins con configuraciones personalizadas
- **config/** - Configuraciones específicas de servicios
  - **browsers.json** - Configuración de navegadores para Selenoid
- **agenteNET/** - Agente especializado para aplicaciones .NET

- **Dockerfile** - Configuración del agente .NET con SDK 8.0, Docker y SonarScanner
- **agentSpring/** - Agente especializado para aplicaciones Java/Spring
  - **Dockerfile** - Configuración del agente Spring con JDK y herramientas Maven
- **agentNode/** - Agente especializado para aplicaciones Node.js
  - **Dockerfile** - Configuración del agente Node.js con npm y herramientas de testing
- **Docker-Tests.py** - Script de Python para pruebas de conectividad y estado de contenedores

Se detalla a continuación algunas de clases del proyecto.

6.2.6.1 Archivo docker-compose.yml

El archivo principal es **docker-compose.yml**, que define todos los servicios que componen la infraestructura:

6.2.6.1.1 Jenkins

La imagen de Jenkins se obtiene mediante un Dockerfile en el cual se instala Docker Compose ya que se usa en las pipelines para levantar contenedores de las aplicaciones para la ejecución de los test de front. (Link del archivo Dockerfile)

Lo más destacado de la configuración de Jenkins en el docker-compose son los volúmenes en el que se define el directorio donde se almacenarán todos los datos de Jenkins como lo es la configuración, pipelines... Esto es importante ya que, en caso de eliminar el contenedor pero no el volumen, Jenkins se mantendrá tal y como lo teníamos.

- **.\jenkins_home:/var/jenkins_home:** Aquí se monta la carpeta local jenkins_home en la ruta /var/Jenkins_home que es donde se guarda toda la configuración, trabajos, credenciales... Este volumen persiste los datos fuera del contenedor, de modo que no se pierden si el contenedor se detiene o se recrea.
- **/var/run/docker.sock:/var/run/docker.sock:** Monta el socket de docker del host dentro del contenedor lo que permite que Jenkins pueda ejecutar comandos de Docker del host.
- **/usr/bin/docker:/usr/bin/docker:ro:** Monta el binario docker de host dentro del contenedor en modo solo lectura. Junto con el socket anterior, permite usar Docker de forma completa desde dentro del contenedor

6.2.6.1.2 Agentes Especializados:

El sistema incluye tres agentes especializados para diferentes tecnologías:

- **agentNode**: Para aplicaciones Node.js
- **agentSpring**: Para aplicaciones Java/Spring
- **agentNet**: Para aplicaciones .NET

Cada agente tiene su propio Dockerfile ya cada uno requiere de distintas librerías para que se ejecuten correctamente las aplicaciones.
Dentro del docker-compose todos los agentes están configurados con:

- Acceso privilegiado

- Claves SSH para la comunicación con Jenkins
- Acceso al socket de Docker
- Límites de recursos (6 CPUs y 6GB de memoria)

6.2.6.1.3 SonarQube

Para este servicio no se usa ningún Dockerfile, se accede a la imagen directamente.

Configuración destacada:

- **Environment**: configura la conexión de SonarQube con la base de datos
- **Volúmenes**: configura la persistencia de datos:
    - **sonarqube_data**: Almacena la base de datos embebida (en caso de no utilizar PostgreSQL).
    - **sonarqube_extensions**: Contiene los plugins y extensiones de SonarQube.
    - **sonarqube_logs**: Guarda los registros de ejecución.
    - **sonarqube_temp**: Almacena los archivos temporales.

6.2.6.2 Archivo docker-compose-selenoid.yml

Este archivo define los servicios de selenoid y selenoid-iu.

En el servicio selenoid se mapean cuatro volúmenes:

- **"/var/run/docker.sock:/var/run/docker.sock":** Permite que el contenedor se comunique con el Daemon de Docker de host para gestionar otros contenedores.
- **"./config:/etc/selenoid":** Monta la carpeta local ./config en /etc/selenoid dentro del contenedor, esta carpeta contiene el archivo browsers.json donde se configura los contenedores de los navegadores que puede usar Selenoid.
- **"./target:/output" y "./target:/opt/selenoid/video":** Estos dos volúmenes permiten guardar la salida y los videos generados de la ejecución de los tests

6.2.7 Jenkins_lib

Esta carpeta define la librería de ejemplo que si hizo para el proyecto. Esta librería está en un repositorio distinto para poder usarlo desde Jenkins.

Generalmente una librería compartida en Jenkins se organiza siguiendo una estructura de directorios que permita separar distintas partes del código.

- **root/**
    - **vars/**
        - buildApp.groovy (Script global para definir pasos comunes)
        - deployApp.groovy
    - **src/**

- **com/**
  - **ejemplo/**
    - Utilidades.groovy (Clases y métodos utilitarios)
    - Configuracion.groovy
- **resources/**
  - **templates/**
    - notificacionEmail.txt (Archivos estáticos o plantillas)

Vars: Es el directorio que contiene los scripts que se cargan de forma global en las pipelines. Cada archivo contiene una función que se puede invocar en el Jenkinsfile.

Src: Se utiliza para incluir código orientado a objetos o utilidades que pueden ser importadas en cualquier script.

Resources: Directorio para almacenar archivos de soporte, plantillas o configuraciones adicionales que pueden ser necesario en la ejecución de la pipeline.

En este proyecto solo se usará la carpeta vars ya que se definirán funciones básicas. La estructura de la librería queda tal que así:

- **root/**
  - **vars/**
    - **cleanUp.groovy** – Limpia el workspace y apaga el contenedor de la aplicación desplegada
    - **generateReports.groovy** – Genera y publica el coverage y los reports de Allure
    - **sonarAnalysis.groovy** – Ejecuta el análisis estático de SonarQube

6.3 Ejecución de pruebas

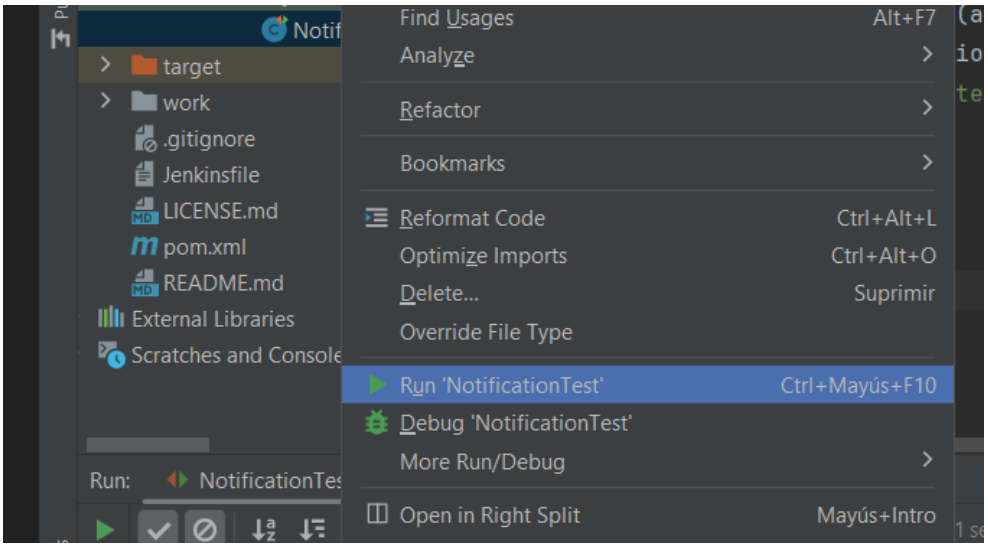
En este apartado se ejecutarán todas las pruebas definidas anteriormente y se mostrará el resultado de ellas.

6.3.1 Ejecución de pruebas de las aplicaciones

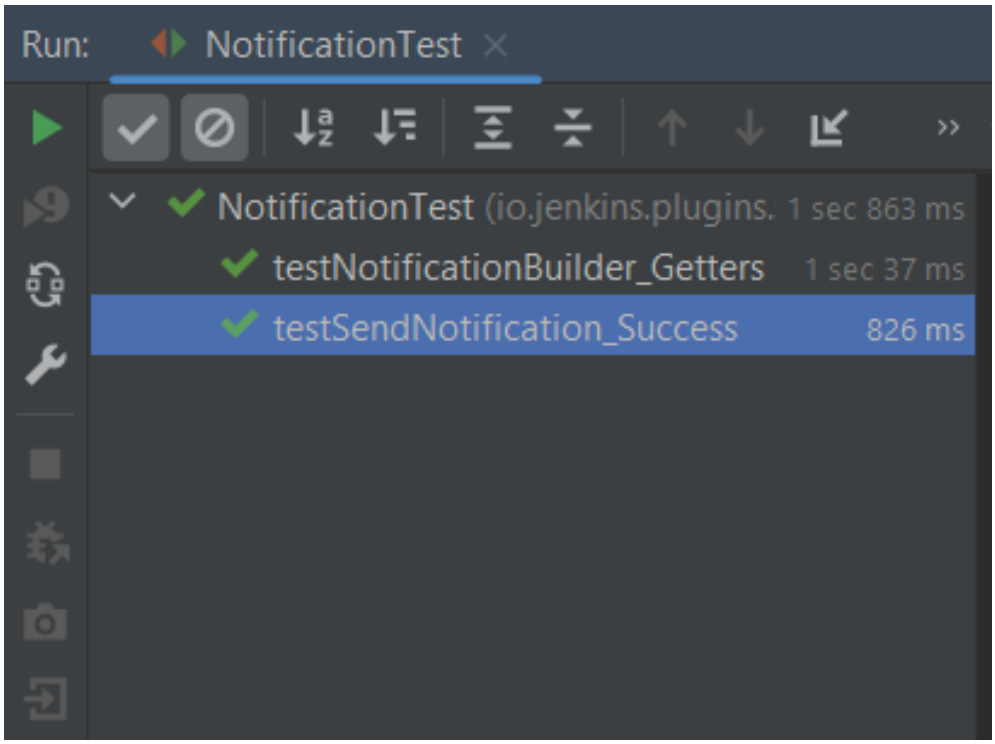
6.3.1.1 Pruebas unitarias

Pruebas Unitarias Plugin Slack

Puesto que el plugin cuenta con Injected Tests creados por maven-hpi-plugin vamos a ejecutar los test creados desde la UI del IDE IntelliJ.



Resultado:



Pruebas unitarias de la aplicación Spring

Comando para ejecutar el test:

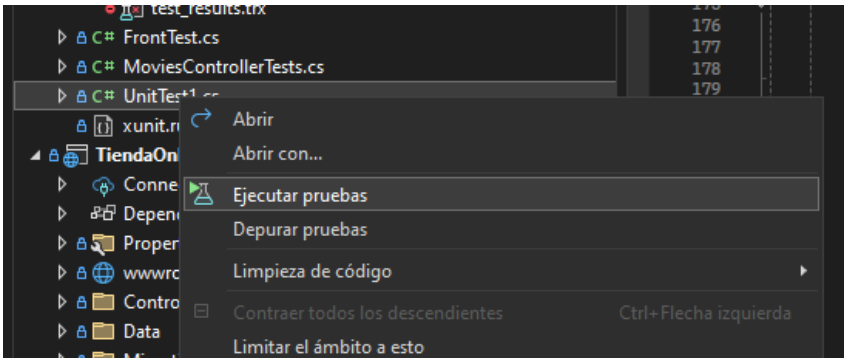
```
mvn -Dtest=ProductoTest test
```

Resultado:

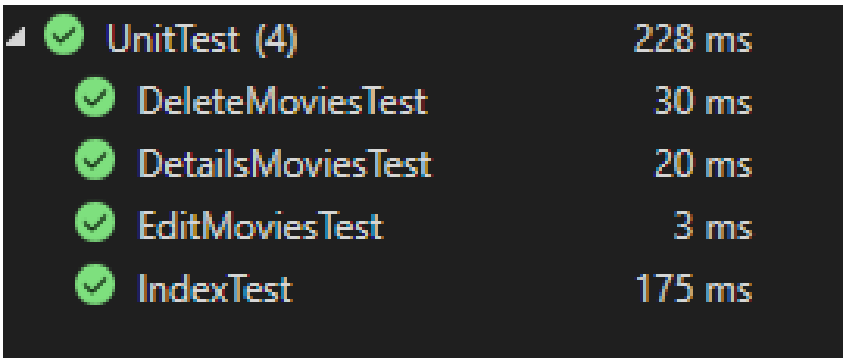
```
[INFO] Results:
[INFO]
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
[INFO]
```

Pruebas unitarias de la aplicación .NET

Para la ejecución del test se ha usado la UI de Visual Studio Community:



Resultado:



6.3.1.2 Pruebas de Integración

Pruebas de integración del sistema

```
Caso de Prueba 1: Validación de comunicación entre contenedores
Comunicación exitosa entre jenkins y sonarqube
Comunicación exitosa entre jenkins y agentNet
Comunicación exitosa entre jenkins y agentNode
Comunicación exitosa entre jenkins y agentSpring
Comunicación exitosa entre jenkins y dockercontainers-selenoid-1

Apagando SonarQube para el segundo test...
SonarQube apagado exitosamente.

Caso de Prueba 2: Detección de contenedores caídos
Contenedores inactivos detectados: tiendanet-tiendaonline-1, tfg_pipeline_nodejs-app-1, tfg_pipeline_nodejs-mongo-1,
 sonarqube

Caso de Prueba 3: Recuperación de un contenedor caído
sonarqube reiniciado exitosamente
```

Pruebas de Integración de la aplicación Spring

Comando para ejecutar el test:

```
mvn -Dtest=ProductoControllerTest test
```

Resultado:

```
[INFO] Results:
[INFO]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
[INFO]
```

Pruebas de Integración de la aplicación Node.js

Comando para ejecutar el test:

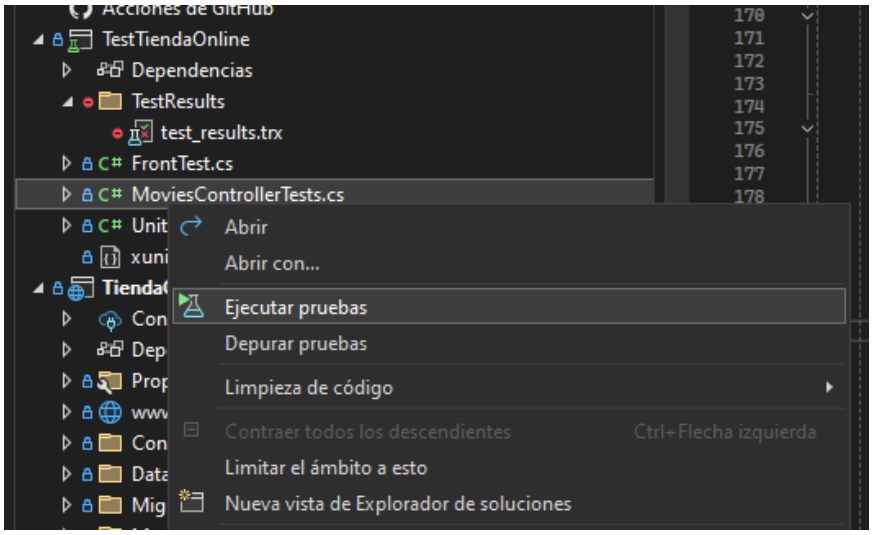
```
npx nyc mocha --no-timeouts --exit ./tests/task.test.js
```

Resultado:

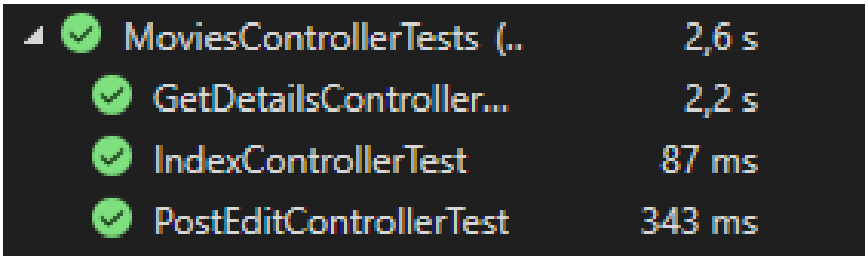
```
  Controller Tests
    GET /tasks
      √ should get all tasks (706ms)
    DELETE /tasks/:id
      √ should delete a task (86ms)
DELETE /todo/delete/67cc261462e8a3335d530771 404 8.713 ms - 10274
      √ should return 404 if task does not exist


  3 passing (3s)
```

Pruebas de Integración de la aplicación .NET

Para la ejecución del test se ha usado la UI de Visual Studio Community:



Resultado:



6.3.1.3 Pruebas de interfaz de usuario

Pruebas de interfaz de usuario de la aplicación Spring

Comando para ejecutar el test:

```
mvn -Dtest=ProductoViewTest test
```

Resultado:

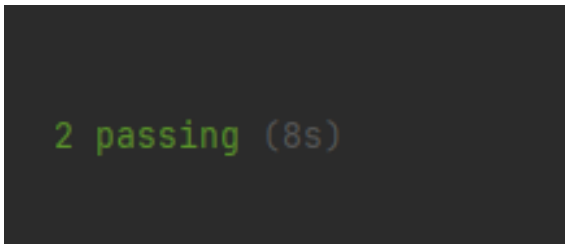
```
[INFO] Results:
[INFO]
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
[INFO]
```

Pruebas de interfaz de usuario de la aplicación Node.js

Comando para ejecutar el test:

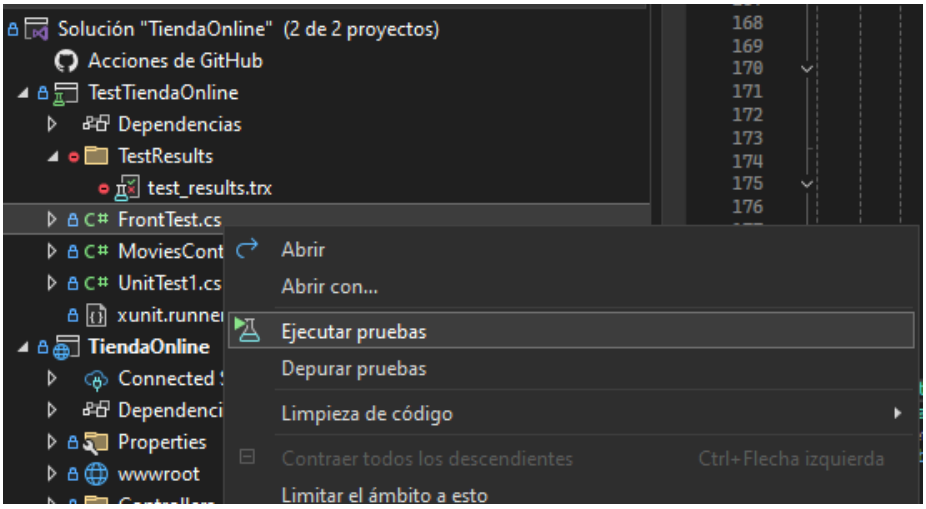
```
npx nyc mocha --reporter mocha-allure-reporter --no-timeouts --exit ./tests/front.test.js
```

Resultado:

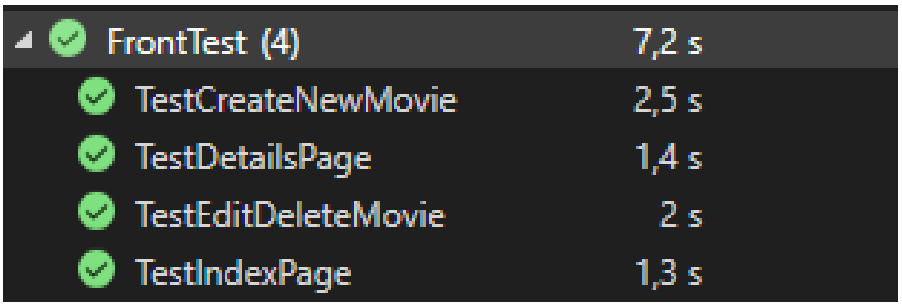


Pruebas de interfaz de usuario de la aplicación .NET

Para la ejecución del test se ha usado la UI de Visual Studio Community:



Resultado:



6.3.2 Ejecución de pruebas del sistema

6.3.2.1 Ejecución correcta de la pipeline

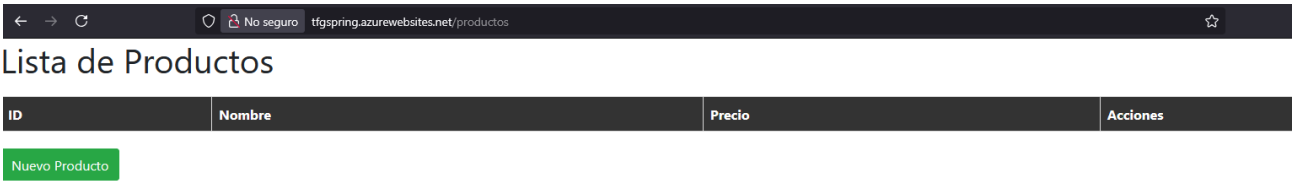
Ejecución correcta de la pipeline - Spring

Ejecución de la pipeline:

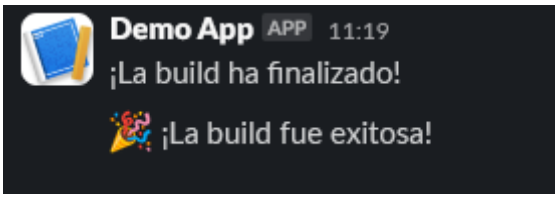
	Declarative: Checkout SCM	Declarative: Tool Install	Build	Dockerizar app
Average stage times: (Average full run time: ~5min 18s)	4s	2s	14s	17s
#161 abr 05 11:13 1 commit	3s	3s	16s	7s

Test	Reporting	Sonar	Quality gate	Deploy to Azure	Declarative: Post Actions
52s	19s	22s	14s	52s	3s
1min 32s	23s	29s	15s	2min 2s	4s

Aplicación desplegada en Azure:



Mensaje de slack:



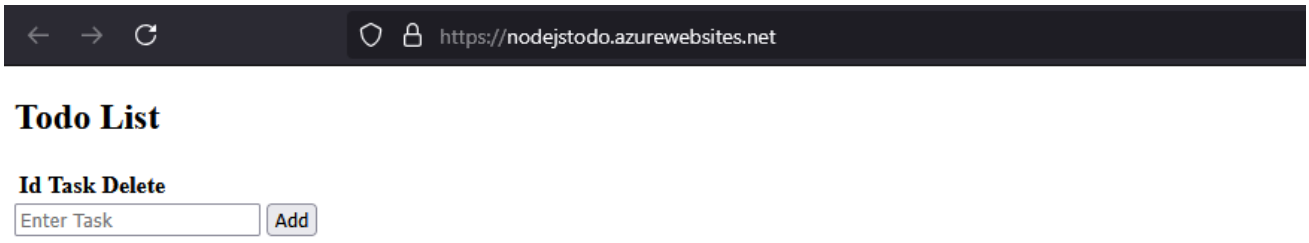
Ejecución correcta de la pipeline – Node

Ejecución de la pipeline:

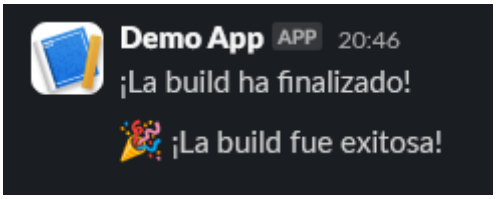
	Declarative: Checkout SCM	Declarative: Tool Install	Install Dependencies	Build	Dockerizar app
Average stage times: (Average full run time: ~7min 52s)	14s	6s	1min 40s	8s	1min 22s
#322 abr 08 20:38 1 commit	14s	1s	1min 30s	7s	41s

Test	Sonar	Quality gate	Generate ZIP	Deploy to Azure	Declarative: Post Actions
46s	1min 23s	32s	1min 21s	4min 44s	1min 0s
59s	1min 0s	41s	2min 13s	4min 56s	54s

Aplicación desplegada en Azure:



Mensaje de slack:



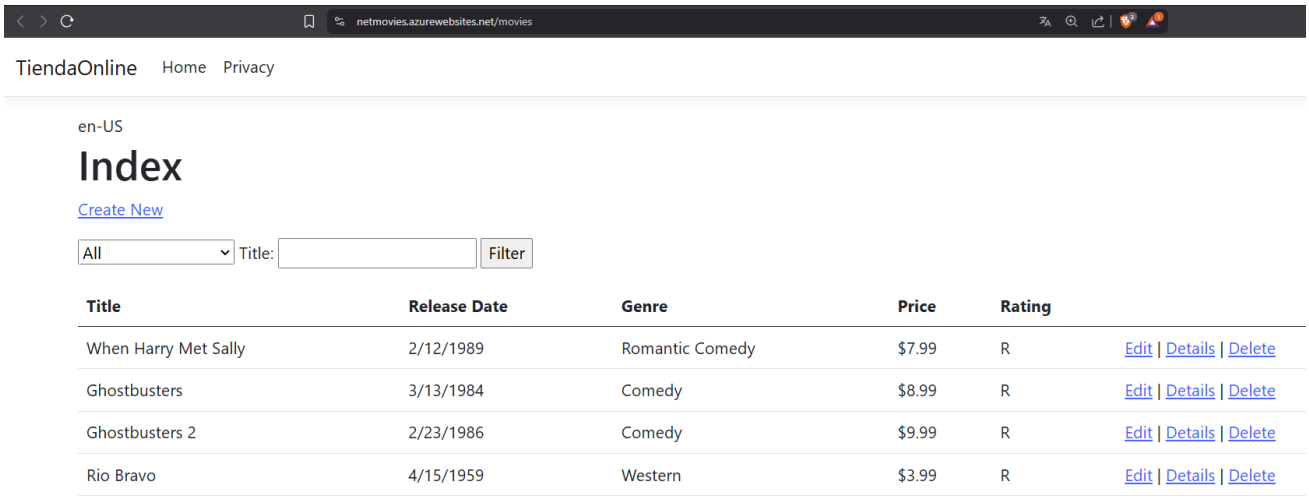
Ejecución correcta de la pipeline – .NET

Ejecución de la pipeline:

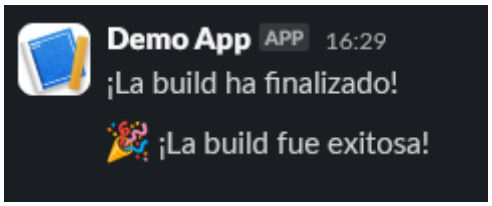
	Declarative: Checkout SCM	Declarative: Tool Install	Restore Dependencies	Install Tools	Build	Dockerize App and Create Selenoid Containers
Average stage times: (Average full run time: ~11min 57s)	5s	6s	21s	9s	25s	1min 39s
#237 abr 06 16:18 1 commit	3s	3s	14s	10s	34s	1min 51s

Test	Sonar	Quality Gate	Pack NuGet Package	Publish to NuGet	Zip proyect	Deploy	Declarative: Post Actions
1min 5s	2min 38s	19s	12s	12s	18s	3min 0s	9s
1min 16s	1min 26s	22s	14s	16s	30s	4min 22s	11s

Aplicación desplegada en Azure



Mensaje de Slack:

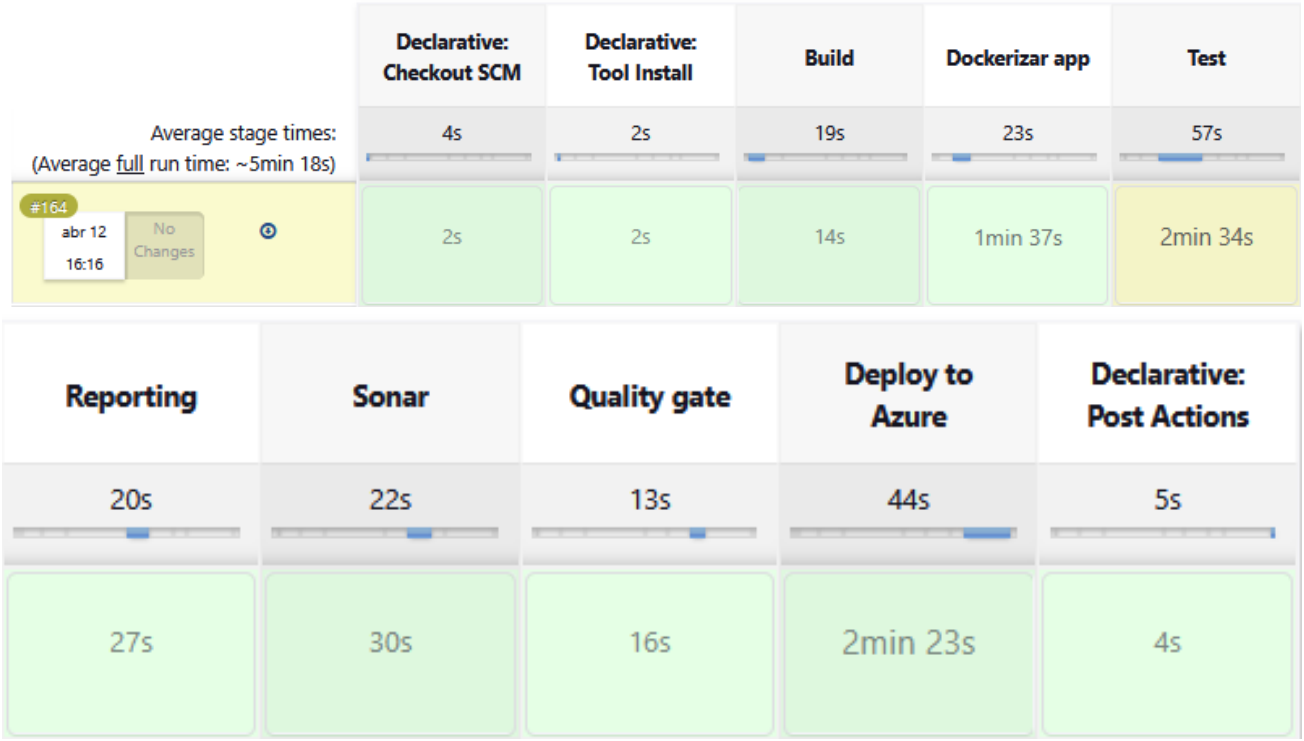


6.3.2.2 Test falla

En esta primera prueba se trata de ver que es lo que sucede en el caso de que un test falla.

Test falla – Proyecto de spring

Se ejecuta la pipeline:



	Declarative: Checkout SCM	Declarative: Tool Install	Build	Dockerizar app	Test
Average stage times: (Average full run time: ~5min 18s)	4s	2s	19s	23s	57s
#164 abr 12 16:16 No Changes	2s	2s	14s	1min 37s	2min 34s

Reporting	Sonar	Quality gate	Deploy to Azure	Declarative: Post Actions
20s	22s	13s	44s	5s
27s	30s	16s	2min 23s	4s

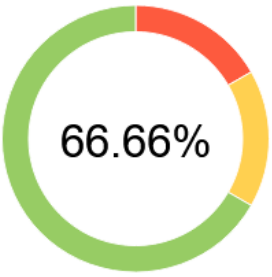
Como se puede ver la pipeline termina con estado “unstable” y se han ejecutado todas las etapas.

Report generado después de la ejecución de la pipeline:

ALLUREREPORT 11/27/2024

19:43:29 - 19:43:42 (13s 397ms)

6
test cases

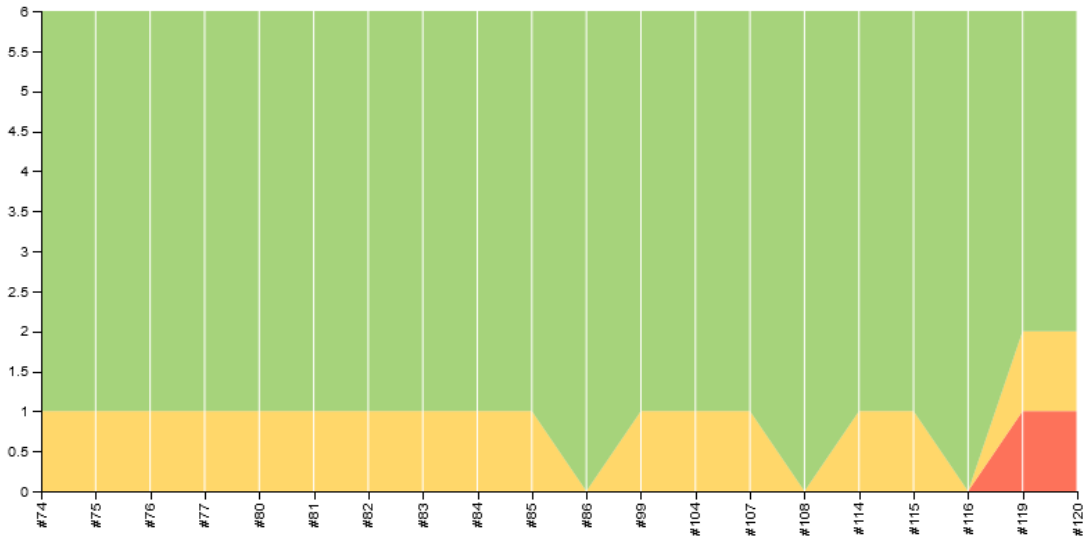


SUITES 3 items total

com.example.TFG_Pipeline.controller.ProductoControllerTest	1 / 1
com.example.TFG_Pipeline.view.ProductoViewTest	1
com.example.TFG_Pipeline.model.ProductoTest	3

Show all

TREND



CATEGORIES 2 items total

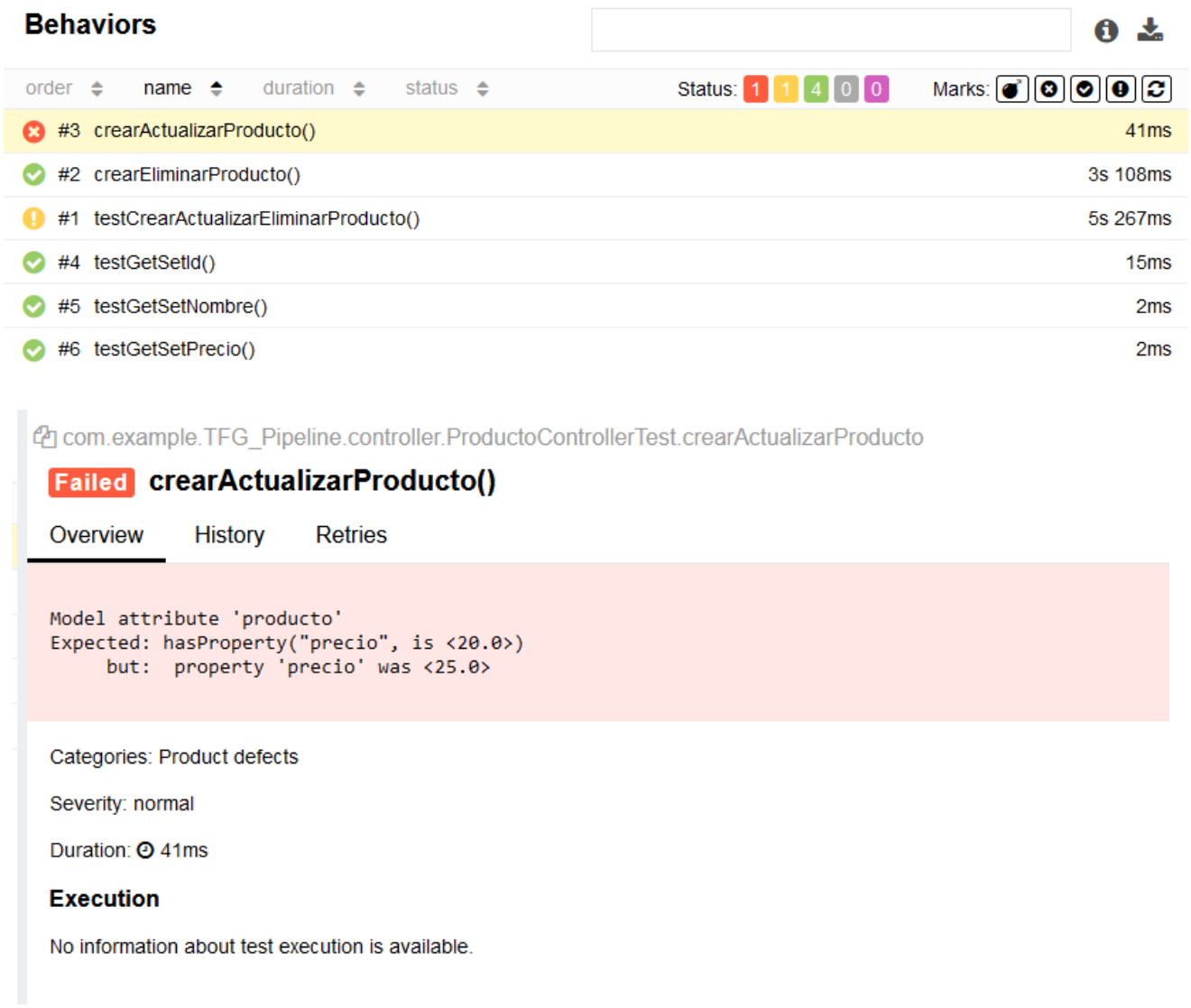
Product defects	1
Test defects	1

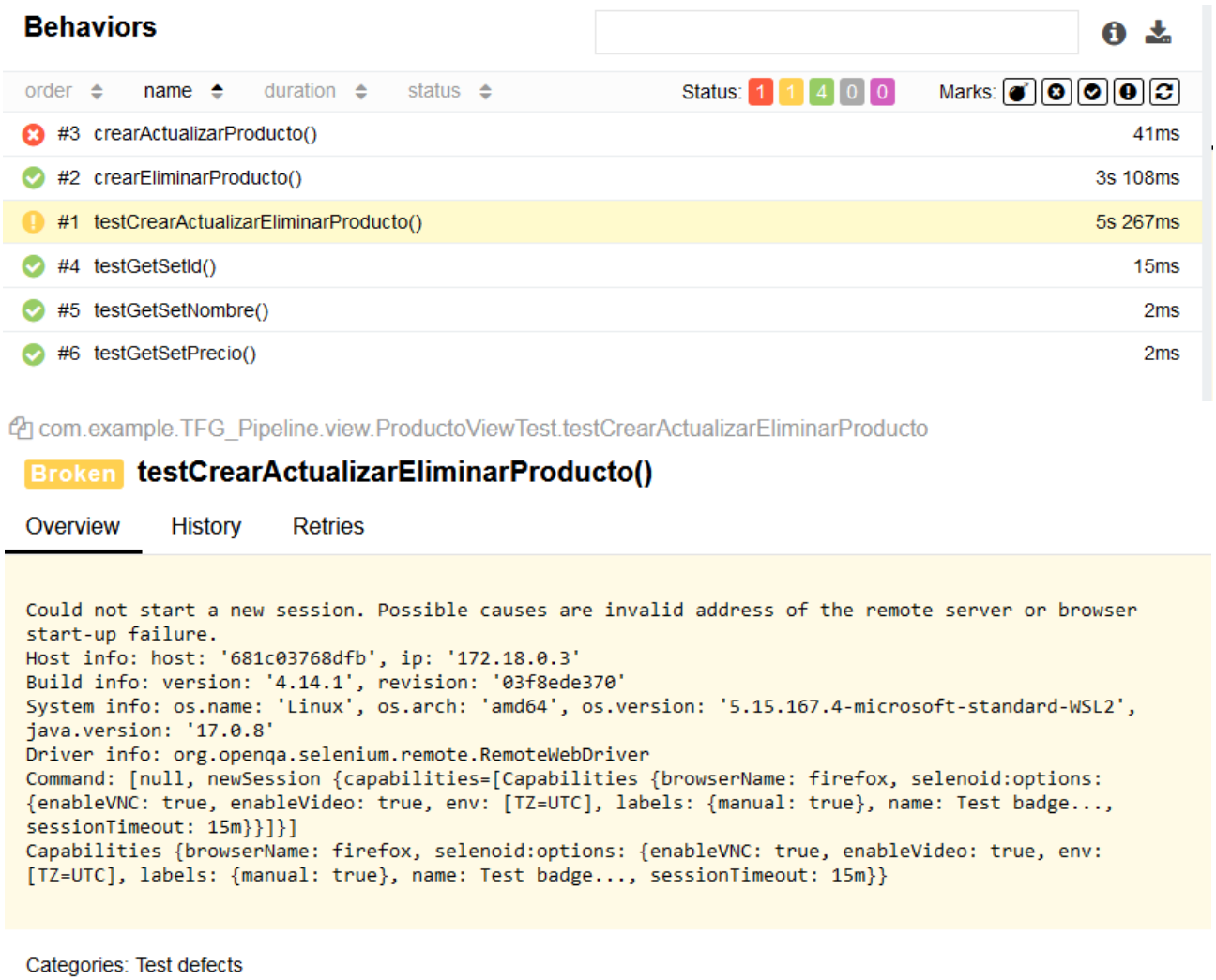
Show all

EXECUTORS

Jenkins — TFG_Spring » main #120

Como se puede ver en el report generado, tal y como se esperaba dos pruebas han fallado. Si se clica en los fallos se puede ver estos con más detalle:





Test falla – Proyecto .NET

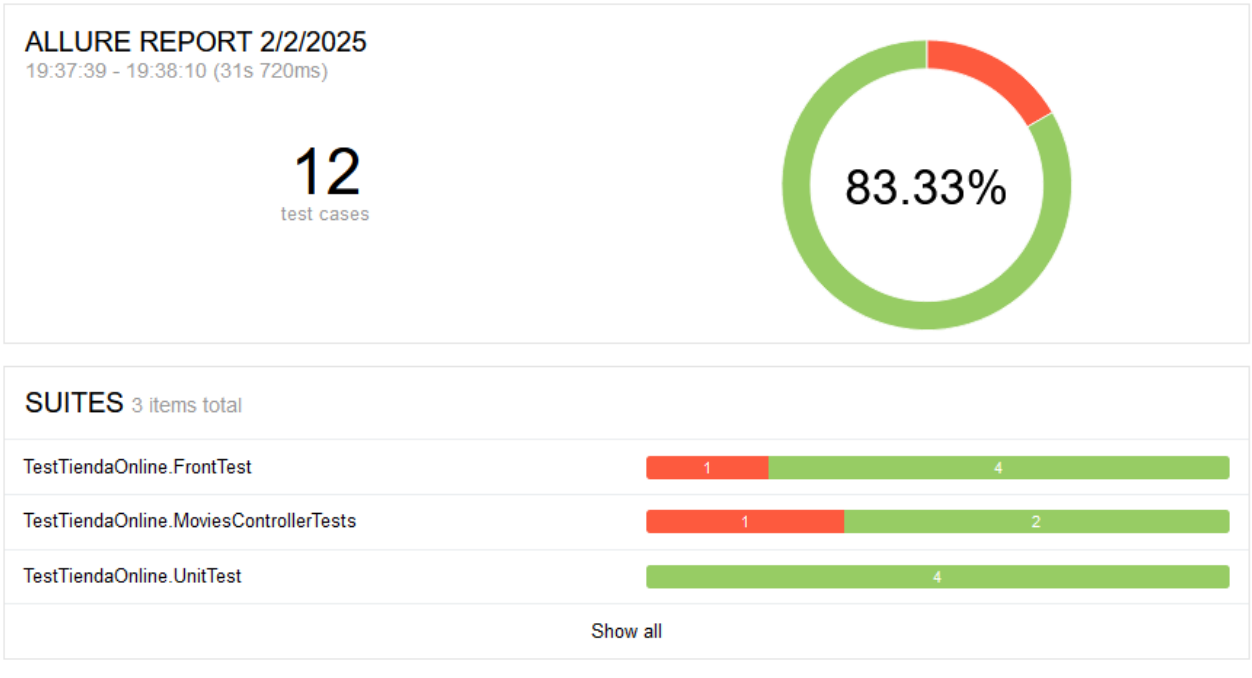
Ejecución de la pipeline:

	Declarative: Checkout SCM	Declarative: Tool Install	Restore Dependencies	Install Tools	Build	Dockerize App and Create Selenoid Containers
Average stage times:	2s	3s	9s	7s	12s	2min 5s
#242 abr 13 14:23 No Changes	2s	3s	9s	7s	12s	2min 5s

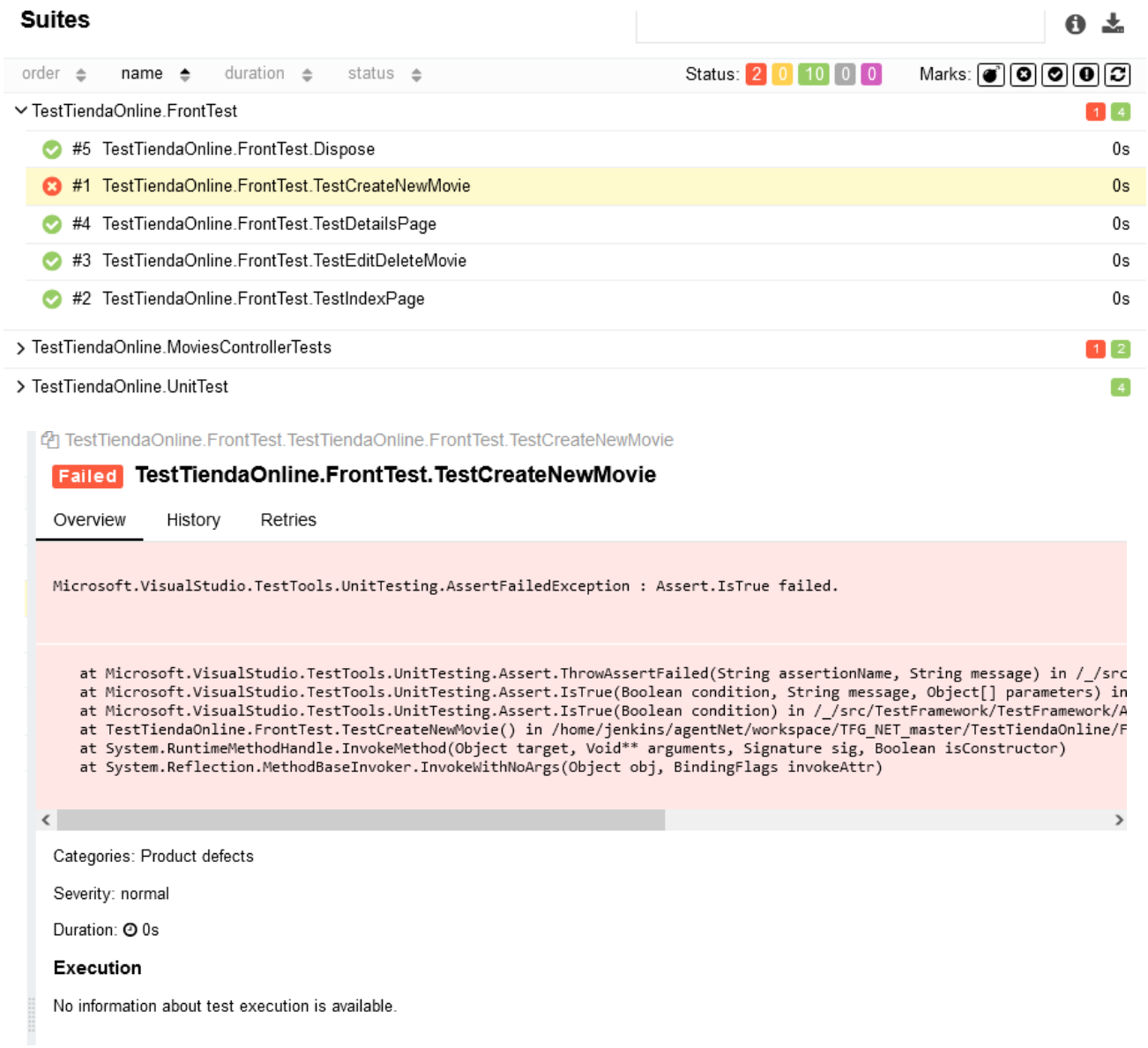
Test	Sonar	Quality Gate	Pack NuGet Package	Publish to NuGet	Zip proyect	Deploy	Declarative: Post Actions
35s	46s	15s	8s	9s	18s	3min 45s	8s
35s	46s	15s	8s	9s	18s	3min 45s	8s

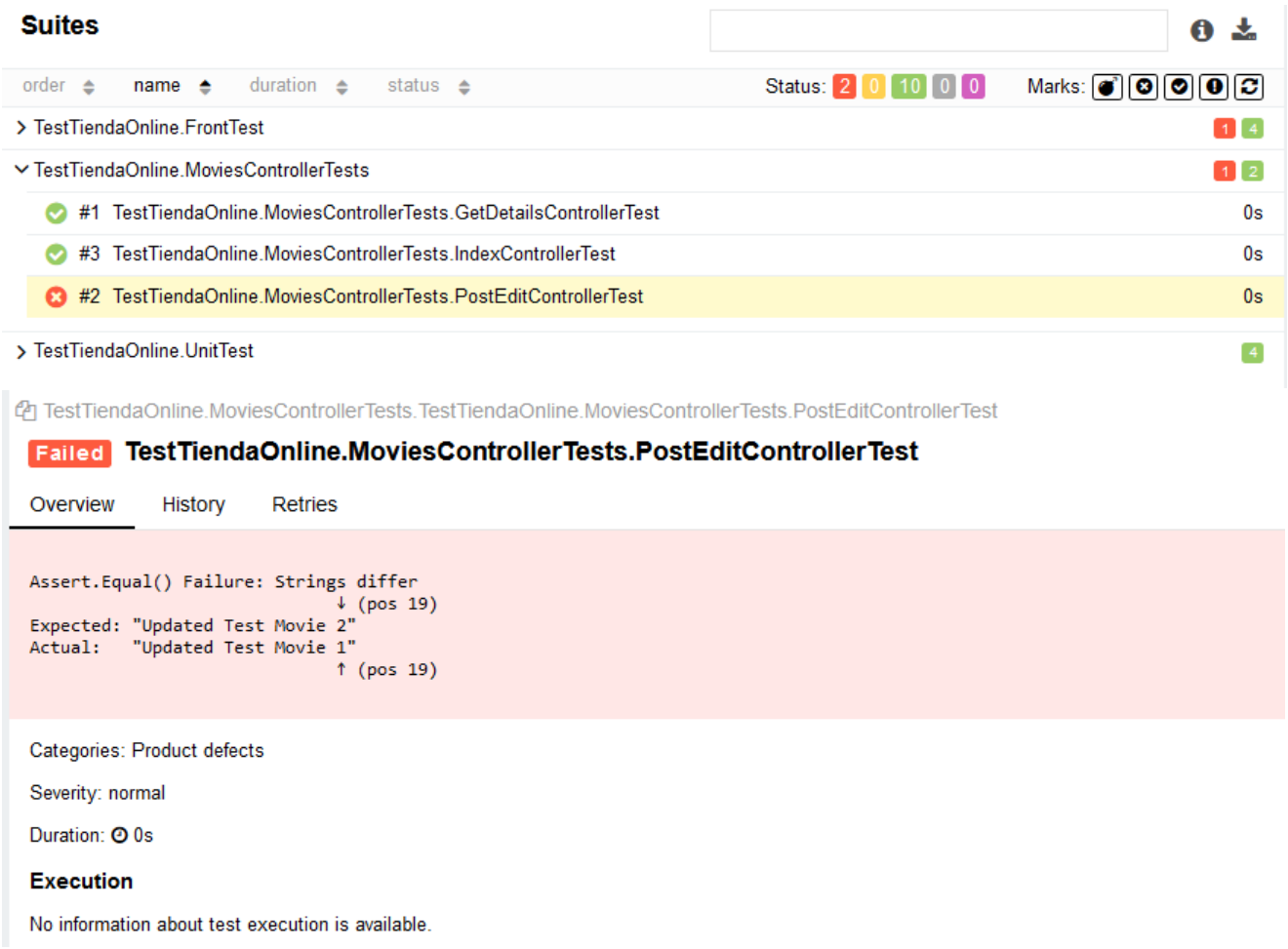
El resultado de la pipeline es que el estado de la pipeline es “unstable” y se han ejecutado todas las etapas.

Report de los tests generado en la pipeline:



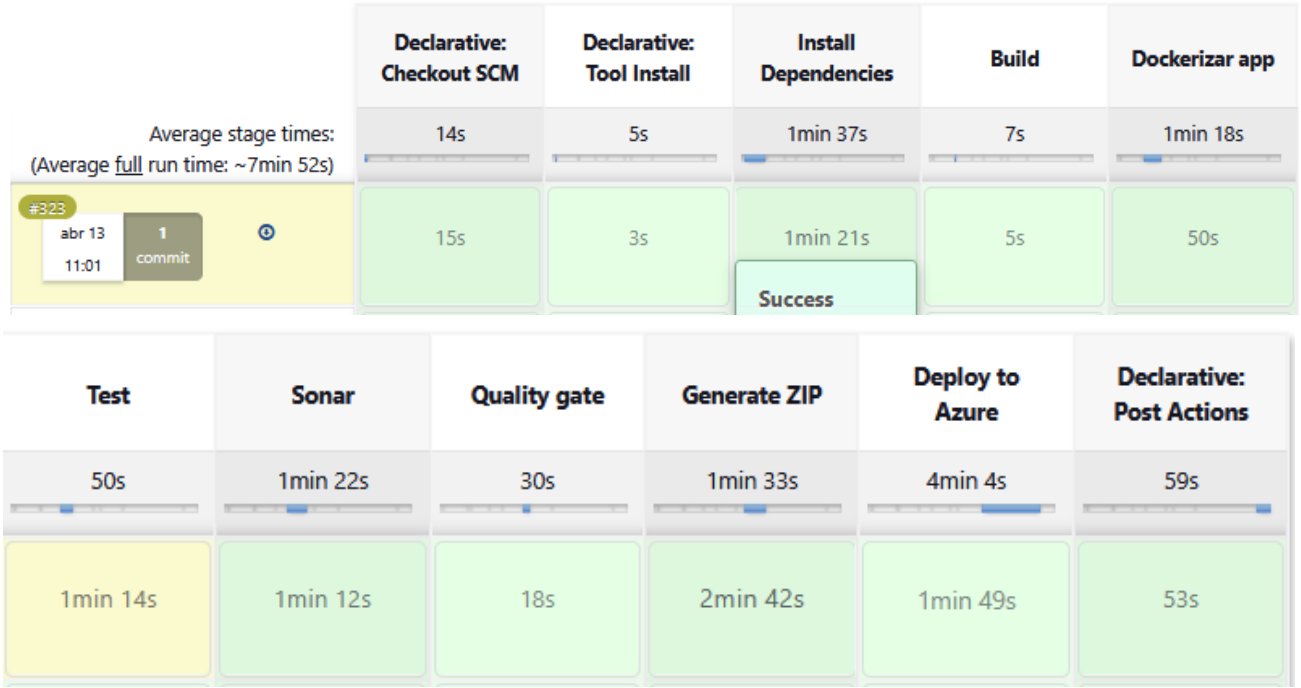
Como era de esperar dos test han fallado, si se clica en ellos podemos ver con mas detalle los fallos:





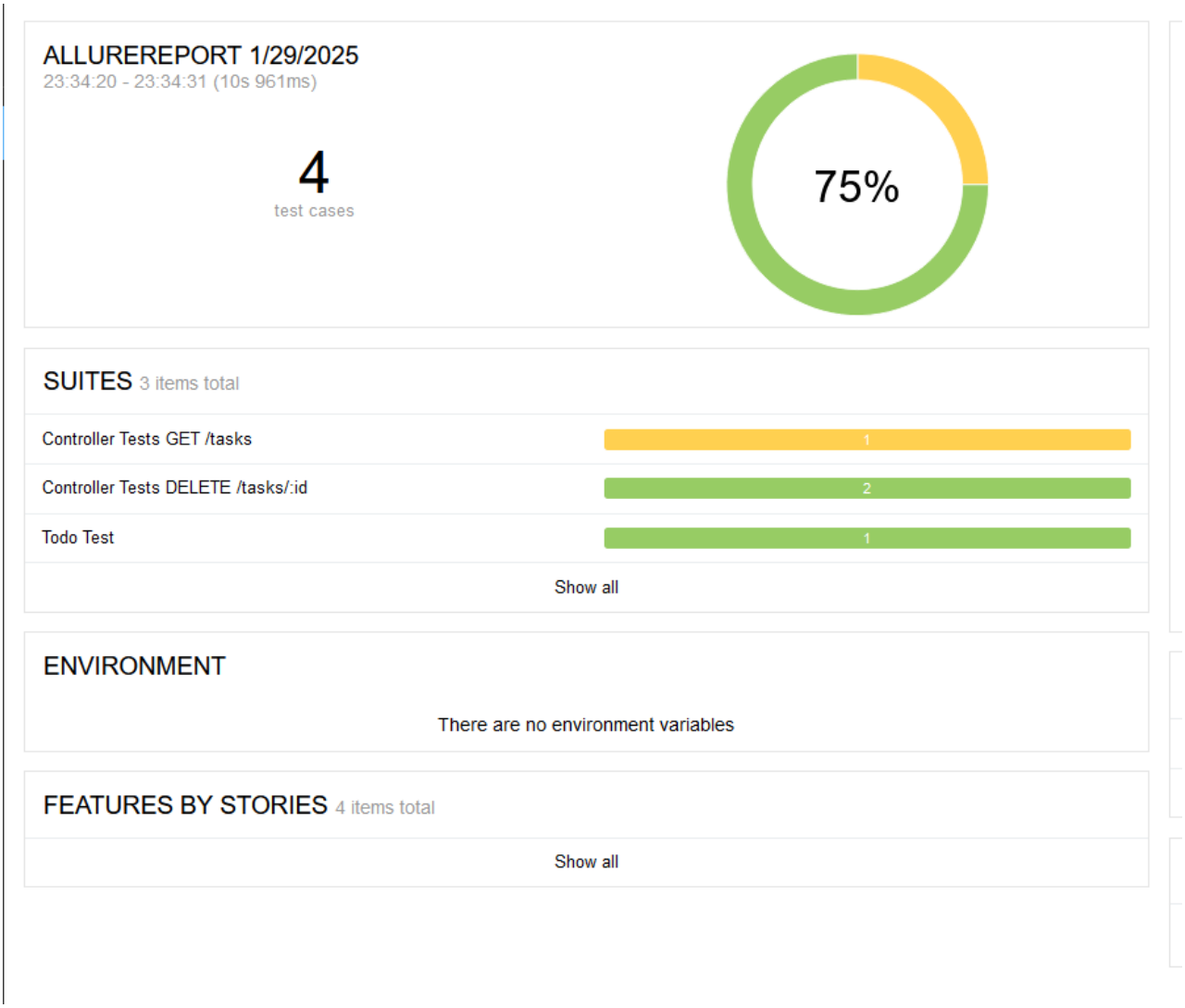
Test falla – Proyecto Node.js

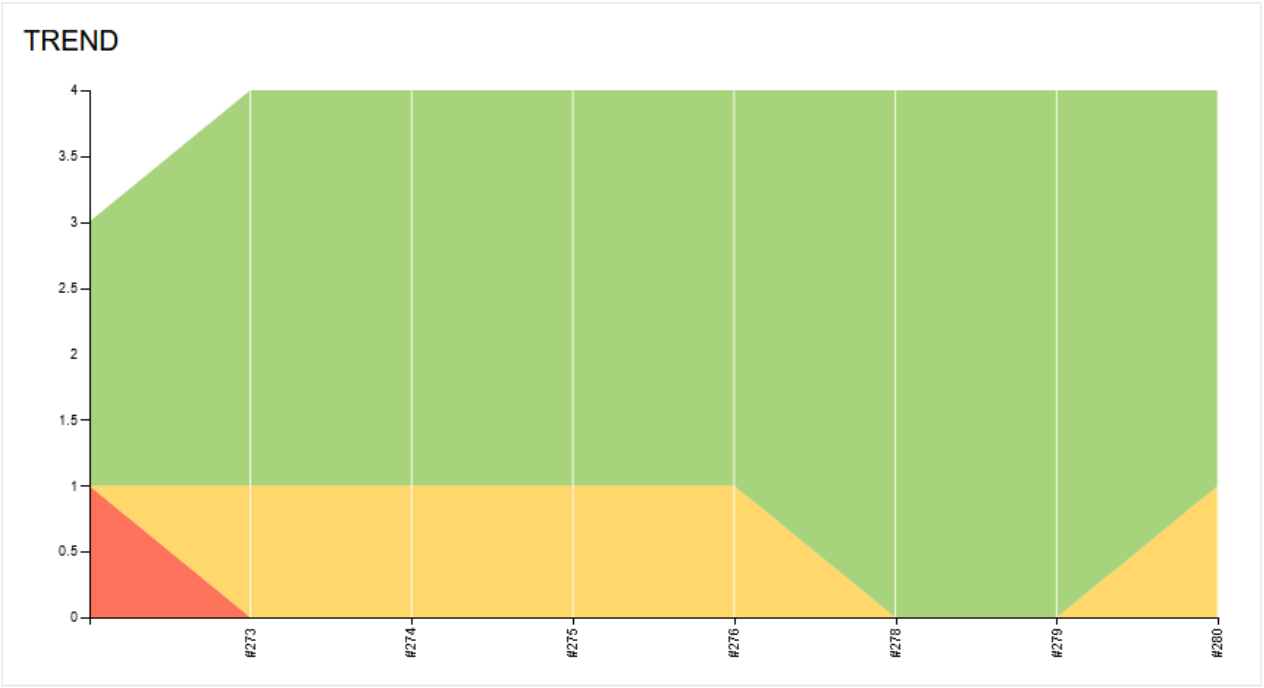
Ejecución de la pipeline



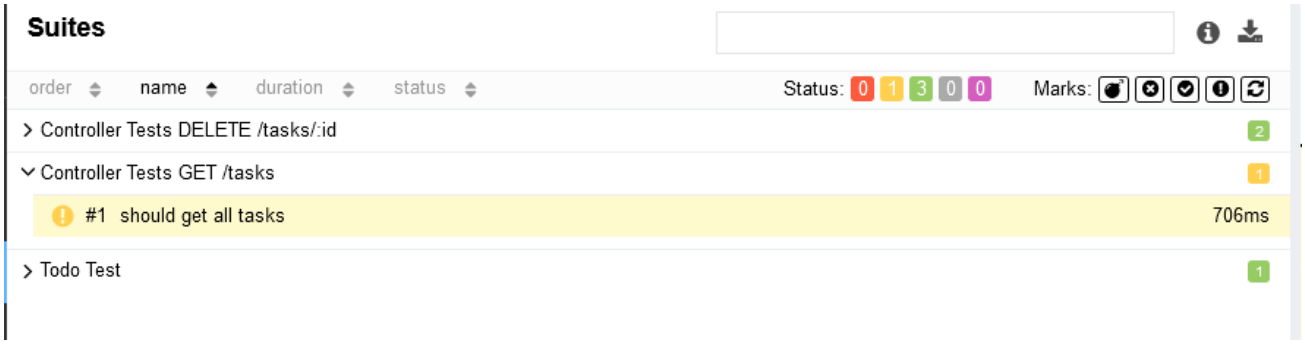
El resultado es el esperado, la pipeline tiene el estado “unstable” y se han ejecutado todas las etapas.

Report de los tests:





Se observa que solo un test ha fallado, si se observa con mas detalle podemos ver el fallo que hemos provocado:



Controller Tests GET /tasks.should get all tasks

Broken **should get all tasks**

Overview History Retries

```
expected '<!DOCTYPE html>\n<html>\n<head>\n    <title>Todo List</title>\n    <link rel="stylesheet" href="https://stackpath
```

```
Error: expected '<!DOCTYPE html>\n<html>\n<head>\n    <title>Todo List</title>\n    <link rel="stylesheet" href="https://st
                    at Assertion.assert (node_modules/expect.js/index.js:96:13)
                    at Assertion.string.Assertion.contain (node_modules/expect.js/index.js:410:12)
                    at Context.<anonymous> (tests/task.test.js:43:29)
                    at process.processTicksAndRejections (node:internal/process/task_queues:95:5)
```

Categories: Test defects

Severity: normal

Duration: 706ms

Execution

No information about test execution is available.

6.3.2.3 No pasar las reglas de sonar

No pasar las reglas de sonar - Spring

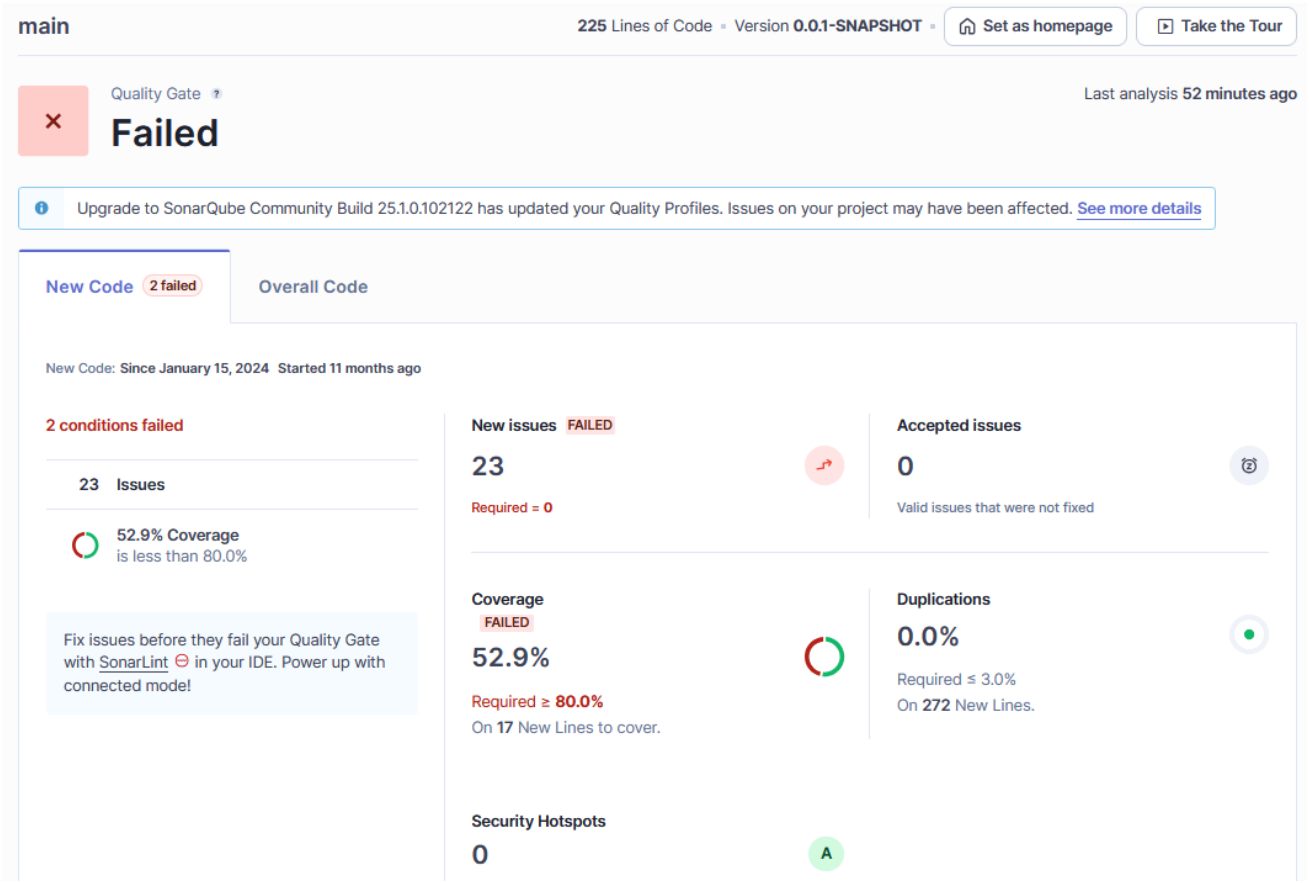
Ejecución de la pipeline:

	Declarative: Checkout SCM	Declarative: Tool Install	Build	Dockerizar app
Average stage times: (Average full run time: ~5min 18s)	6s	3s	20s	17s
#165 abr 13 10:48 No Changes	14s	2s	16s	10s

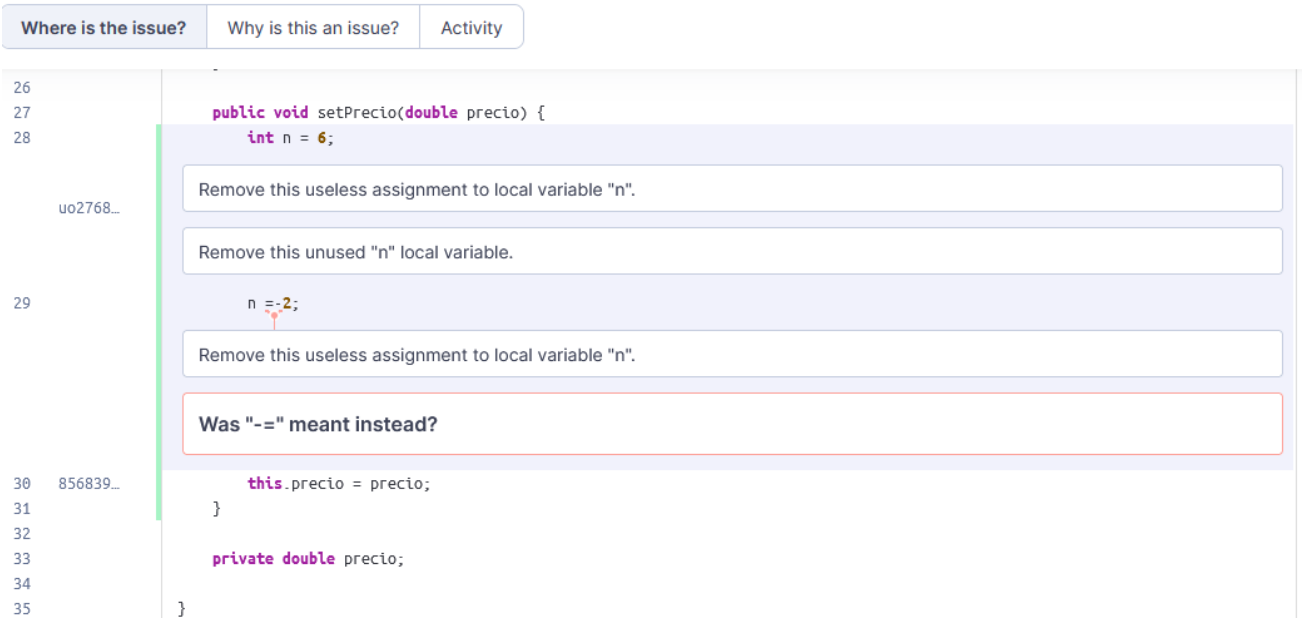
Test	Reporting	Sonar	Quality gate	Deploy to Azure	Declarative: Post Actions
1min 1s	21s	24s	13s	43s	5s
1min 25s	25s	29s	15s	1min 49s	5s

Como se puede observar la pipeline está marcada como “unstable” ya que falló la quality gate. La pipeline es “unstable” por lo que se han ejecutado todas las etapas como era de esperar.

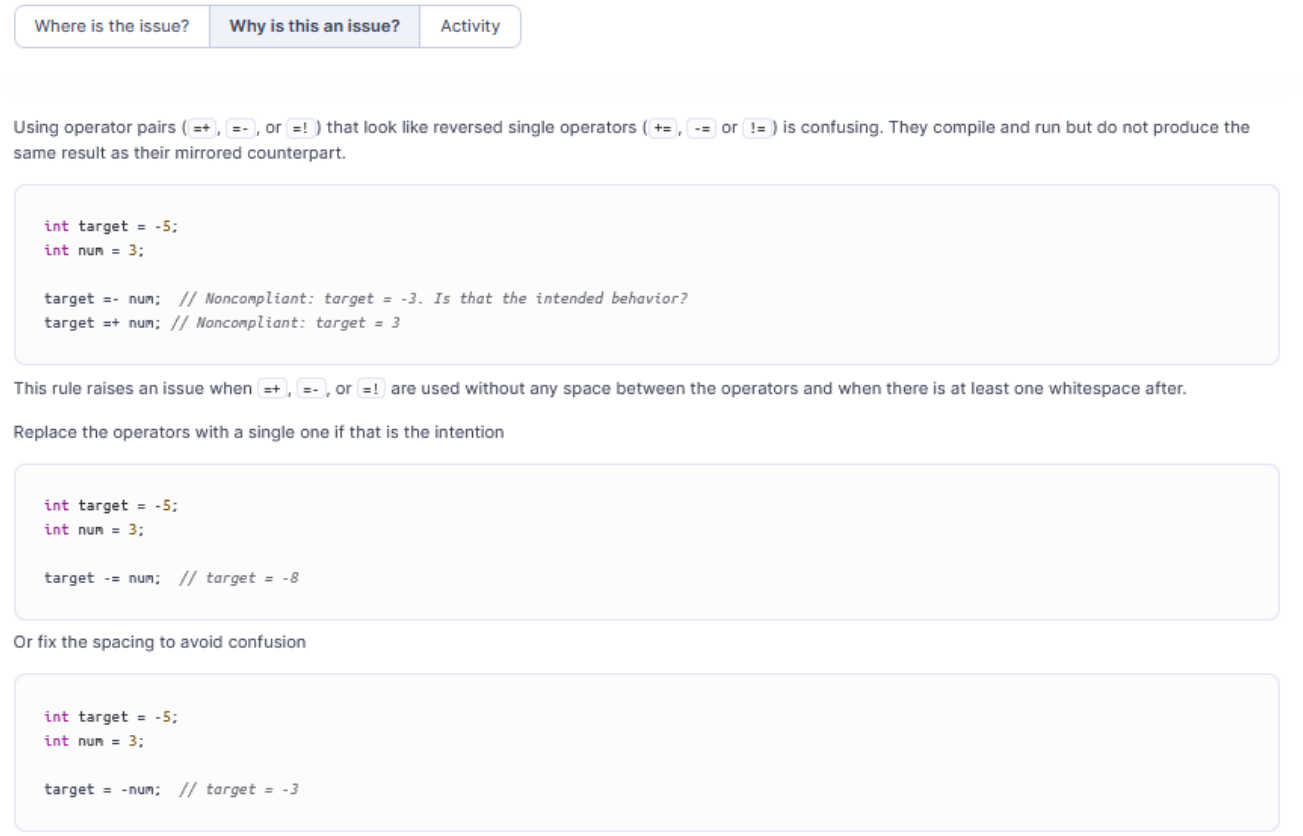
SonarQube:



Como se puede observar han fallado las dos condiciones que se mencionó anteriormente y sale que la quality gate ha fallado. Se observa que hay nuevas code smells y el tiempo de la deuda técnica es mayor. Si se mira con más detalle en los bugs se pueden ver varios de ellos.



Se busca el bug que hemos creado anteriormente y se puede ver que Sonar lo ha detectado y se mira con detalle para ver cuál es el problema. Sonar menciona que al estar el "=" y el "-" juntos lo que igual se pretendía no era asignar un -2 a "n" sino restarle 2 a "n" usando el operador "-=" por lo que se nos pregunta si pretendíamos una resta.



Where is the issue? | **Why is this an issue?** | Activity

Using operator pairs (`=+` , `=-` , or `=!`) that look like reversed single operators (`+=` , `-=` or `!=`) is confusing. They compile and run but do not produce the same result as their mirrored counterpart.

```
int target = -5;
int num = 3;

target =- num;  // Noncompliant: target = -3. Is that the intended behavior?
target =+ num; // Noncompliant: target = 3
```

This rule raises an issue when `=+` , `=-` , or `=!` are used without any space between the operators and when there is at least one whitespace after.

Replace the operators with a single one if that is the intention

```
int target = -5;
int num = 3;

target -= num;  // target = -8
```

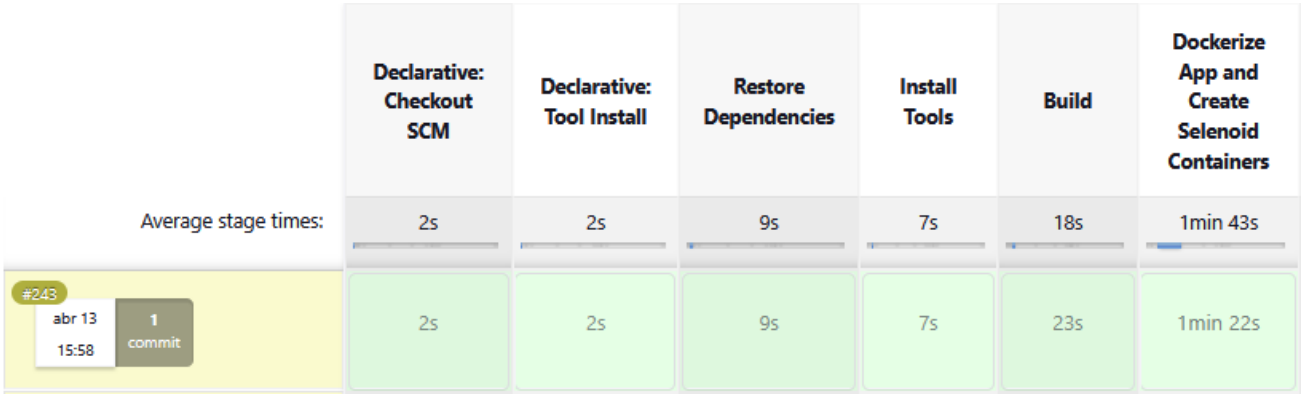
Or fix the spacing to avoid confusion

```
int target = -5;
int num = 3;

target = -num;  // target = -3
```

También SonarQube tiene un apartado en el que se nos dice como se puede arreglar este bus. Todo esto se aplica también a muchas otras issues como code smells.

No pasar las reglas de sonar - .Net

Ejecución de la pipeline:

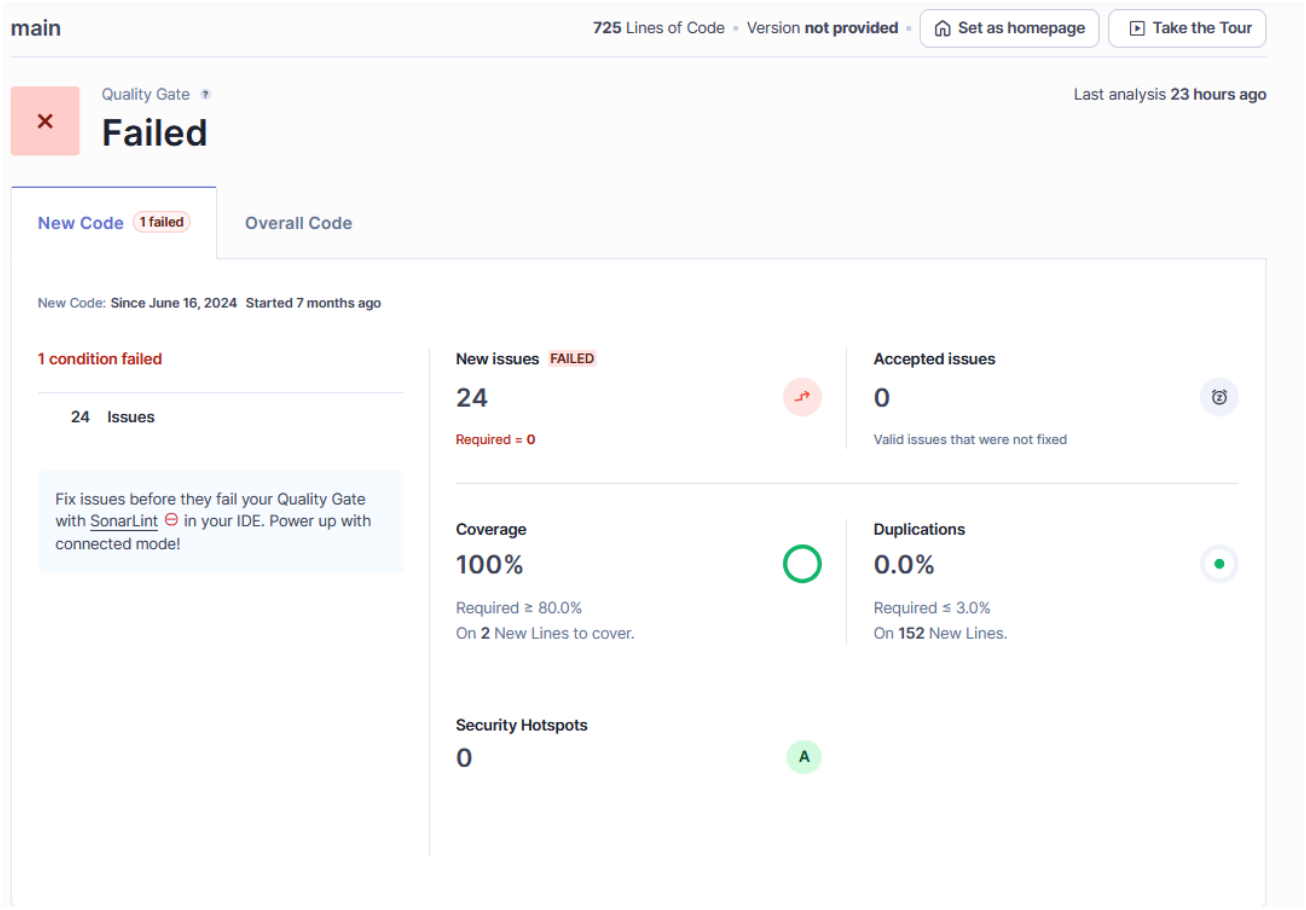


	Declarative: Checkout SCM	Declarative: Tool Install	Restore Dependencies	Install Tools	Build	Dockerize App and Create Selenoid Containers
Average stage times:	2s	2s	9s	7s	18s	1min 43s
#243 abr 13 15:58 — 1 commit	2s	2s	9s	7s	23s	1min 22s

Test	Sonar	Quality Gate	Pack NuGet Package	Publish to NuGet	Zip proyect	Deploy	Declarative: Post Actions
1min 30s	56s	17s	9s	10s	18s	3min 46s	8s
2min 24s	1min 7s	19s	10s	11s	18s	3min 47s	8s

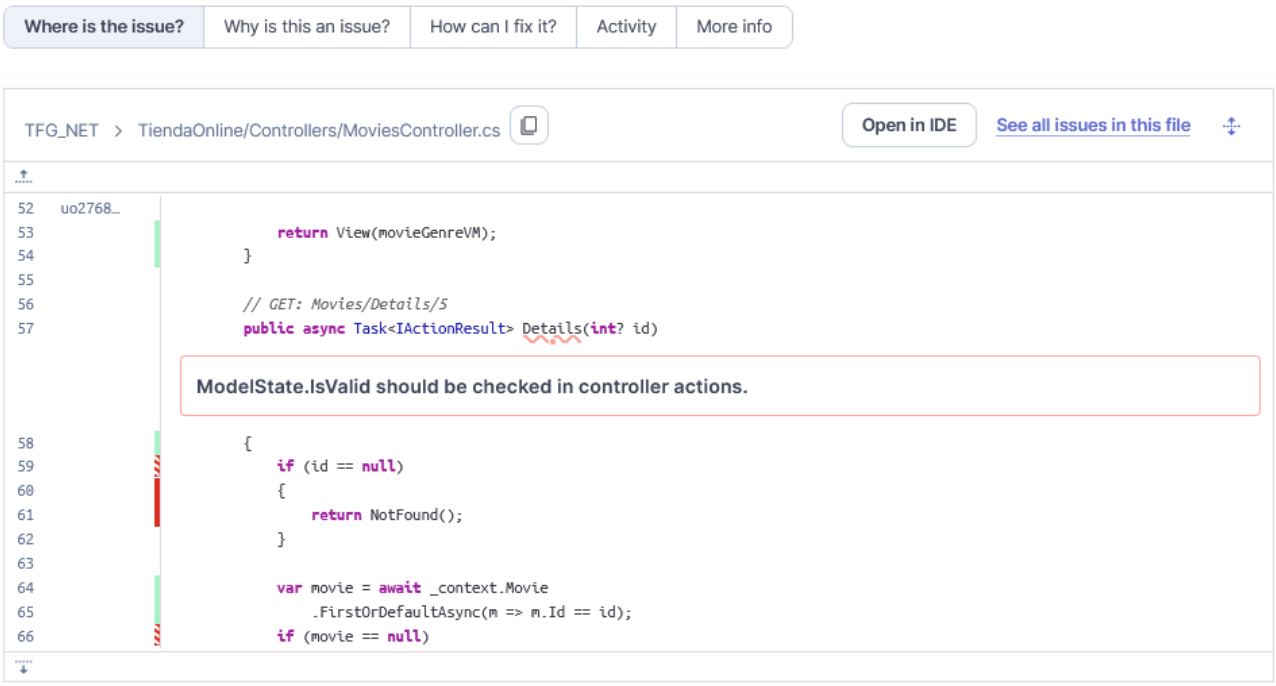
Se observa que la pipeline tiene termina con estado “unstable” y con todas las etapas ejecutadas.

SonarQube:



Como se ve en el dashboard del proyecto de SonarQube hay 24 issues, por lo que el Quality Gate fallará, ya que una de las condiciones es que haya 0 issues.

Se escoge una de las issues para ve cual es el problema y como solucionarlo.



If `ModelState.IsValid` returns true, it means that the data is valid and the process can continue. If it returns false, it means that the validation failed, indicating that the data is not in the expected format or is missing required information.

In such cases, the controller action should handle this by returning an appropriate response, such as re-displaying the form with error messages. This helps maintain the integrity of the data and provides feedback to the user, enhancing the overall user experience and security of your application.

Noncompliant code example

```
public async Task<IActionResult> Create(Movie movie) // Noncompliant: model validity check is missing
{
    _context.Movies.Add(movie);
    await _context.SaveChangesAsync();

    return RedirectToAction(nameof(Index));
}
```

Compliant solution

```
public async Task<IActionResult> Create(Movie movie)
{
    if (!ModelState.IsValid)
    {
        return View(movie);
    }

    _context.Movies.Add(movie);
    await _context.SaveChangesAsync();

    return RedirectToAction(nameof(Index));
```

En este ejemplo se observa que hay una issue en MoviesControler en el método Details porque falta una comprobación de que ModelState es válida.

No pasar las reglas de sonar - Node.js

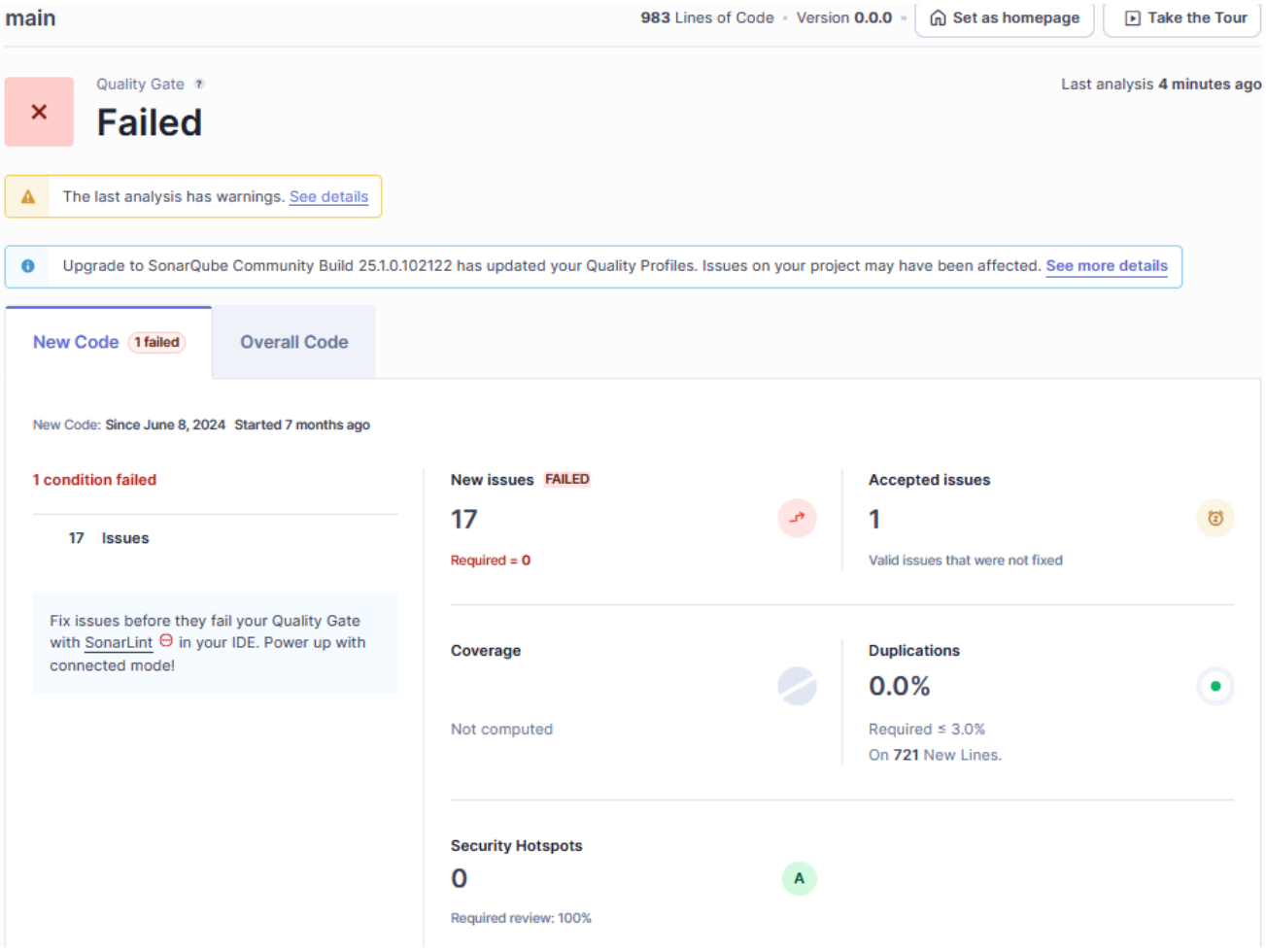
Ejecución de la pipeline:

	Declarative: Checkout SCM	Declarative: Tool Install	Install Dependencies	Build	Dockerizar app
Average stage times: (Average full run time: ~7min 52s)	14s	5s	1min 36s	7s	1min 13s
#324 abr 13 11:15 1 commit	13s	2s	1min 27s	4s	43s

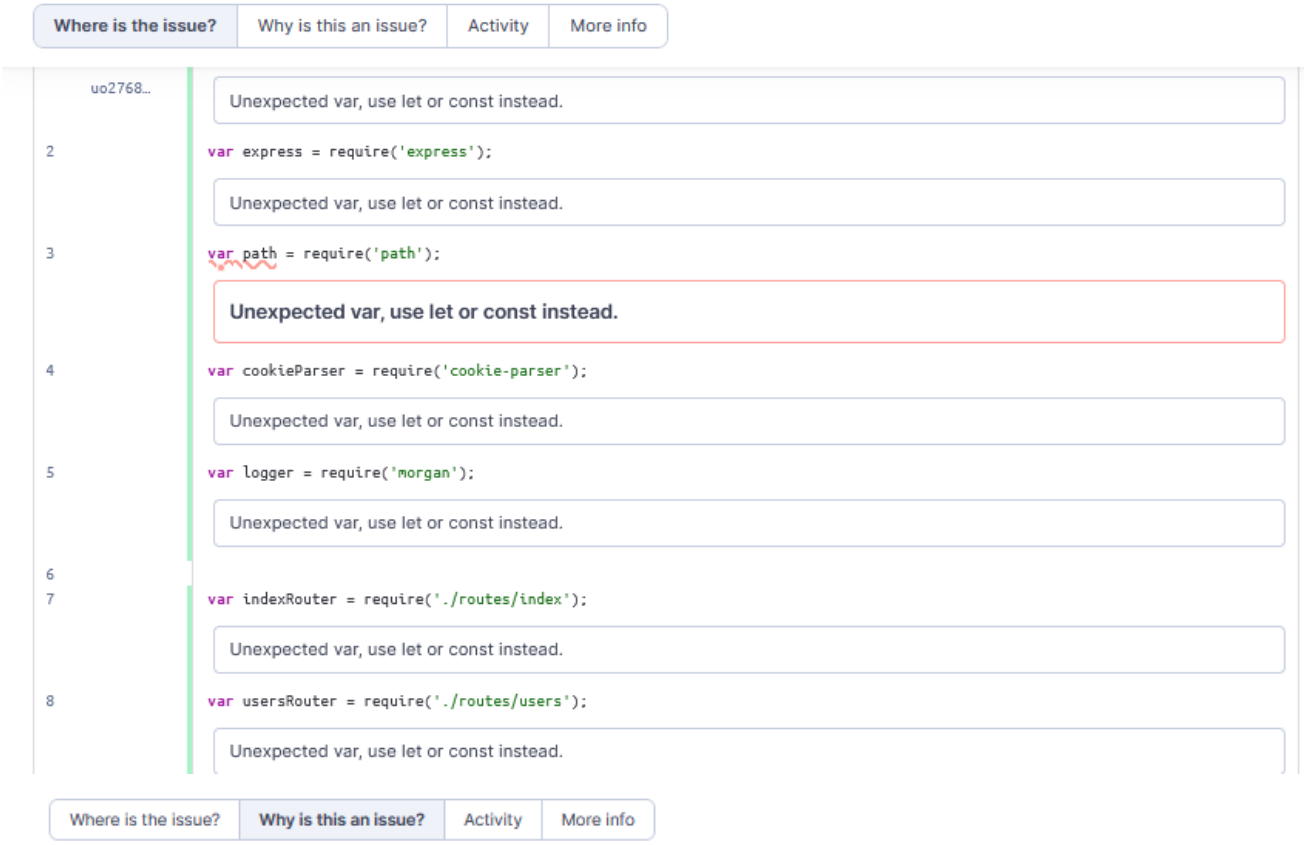
Test	Sonar	Quality gate	Generate ZIP	Deploy to Azure	Declarative: Post Actions
51s	1min 16s	29s	1min 40s	3min 34s	58s
53s	38s	18s	2min 27s	2min 0s	50s

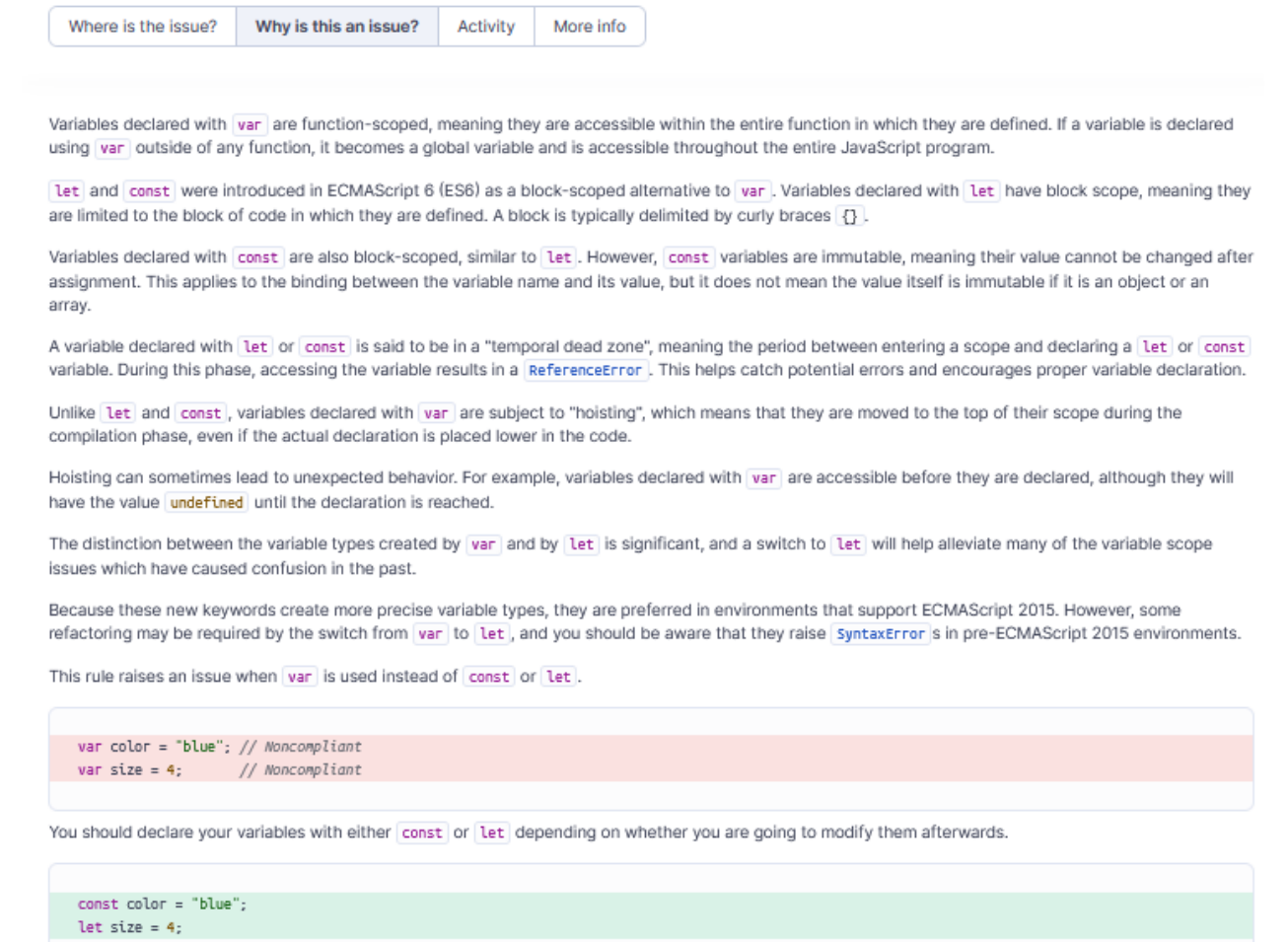
Como se esperaba la pipeline tiene estado “unstable” y todas las etapas se han ejecutado.

SonarQube:



Como se puede observar hay 17 issues lo que hace que falle la condición de que haya 0 issues y por tanto que la Quality Gate falle.





Como vemos Sonar nos señala varias de las issues que tenemos que arreglar y al igual que en los otros proyectos podemos saber por qué y cómo solucionarlo. En este caso nos dice que cambiemos la declaración de las variables var por const o let.

6.3.2.4 Fallo en el Build

Fallo en el Build - Spring

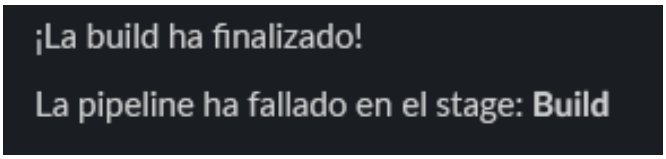
Ejecución de la pipeline:

En esta prueba la pipeline falla y no se ejecuta toda la pipeline. Si se observa el log de la consola se puede ver que error hay y donde se produce, lo que facilita el arreglo. En este caso muestra un error de compilación como se esperaba.

Log de la consola:

```
[ERROR] COMPILATION ERROR :
[INFO] -------------------------------------------------------------
[ERROR] /home/jenkins/agent/workspace/TFG_Spring_main/src/main/java/com/example/TFG_Pipeline/controller/ProductoController.java:[20,37] ';' expected
[INFO] 1 error
[INFO] -------------------------------------------------------------
[INFO] ------------------------------------------------------------------------
[INFO] BUILD FAILURE
```

Mensaje de Slack;



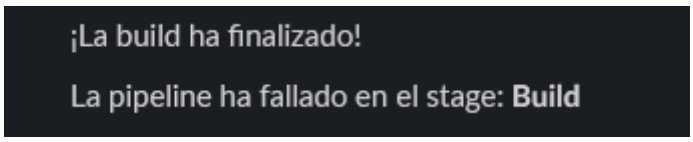
Fallo en el Build - .NET

Ejecución de la pipeline:

La pipeline falla y en el log de Jenkins se puede ver el error y donde se encuentra.

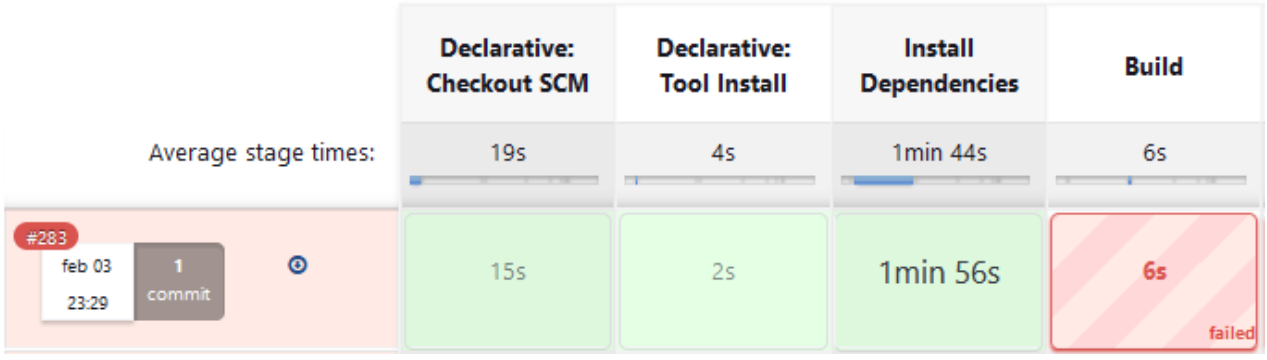
```
/home/jenkins/agentNet/workspace/TFG_NET_master/TiendaOnline/Controllers/MoviesController.cs(186,42): error CS0161: 'MoviesController.DeleteConfirmed(int)': not all code paths return a value [/home/jenkins/agentNet/
workspace/TFG_NET_master/TiendaOnline/TiendaOnline.csproj]
    9 Warning(s)
    1 Error(s)
```

Mensaje de Slack:



Fallo en el Build - Node.js

Ejecución de la pipeline:

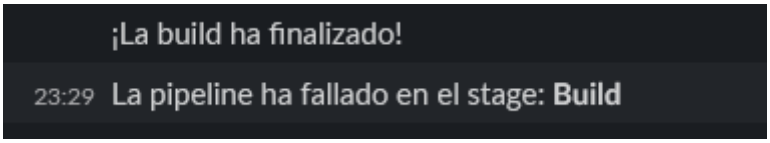


Dockerizar app	Test	Sonar	Quality gate	Generate ZIP	Deploy to Azure	Declarative: Post Actions
1min 12s	42s	18s	7s	1s	1s	53s
961ms failed	886ms failed	876ms failed	999ms failed	1s failed	922ms failed	13s

Falla la pipeline y no se ejecutan las siguientes etapas, el log muestra el problema:

```
/home/jenkins/agent/workspace/TFG_MultiBranchPipeline_Rebuild/routes/todo.js
  11:18  error  Parsing error: Unexpected token ;

✖ 1 problem (1 error, 0 warnings)
```

Mensaje de Slack:



6.3.2.5 Fallo en el despliegue

Fallo en el despliegue - Spring

Ejecución de la pipeline:

	Declarative: Checkout SCM	Declarative: Tool Install	Build	Dockerizar app
Average stage times: (Average full run time: ~5min 18s)	4s	3s	20s	16s
#162 abr 08 21:13 1 commit	3s	8s	1min 15s	13s

Test	Reporting	Sonar	Quality gate	Deploy to Azure	Declarative: Post Actions
52s	22s	24s	14s	48s	5s
1min 0s	49s	43s	19s (paused for 1min 52s)	12s failed	19s

Como era de esperar la pipeline falla y por tanto no se despliega la nueva versión de la aplicación.

```
Masking supported pattern matches of $AZURE_CLIENT_SECRET
[Pipeline] {
[Pipeline] sh
+ az login --service-principal -u d1315aa9-ed60-40bf-bcc5-2e42a6a4d5e7 -p **** -t 05ea74a3-92c5-4c31-978a-925c3c799cd
ERROR: Failed to resolve tenant '05ea74a3-92c5-4c31-978a-925c3c799cd'.
```

Error detail: {"error":"invalid_tenant","error_description":"AADSTS90002: Tenant '05ea74a3-92c5-4c31-978a-925c3c799cd' not found. Check to make sure you have the correct tenant ID and are signing into the correct cloud. Check with your subscription administrator, this may happen if there are no active subscriptions for the tenant. Trace ID: bc1d9a0d-12f8-4a91-8399-a96b5219cc00 Correlation ID: 78576956-9146-47a3-b28c-226c2ecde3ec Timestamp: 2024-12-20 23:28:05Z","error_codes":[90002],"timestamp":"2024-12-20 23:28:05Z","trace_id":"bc1d9a0d-12f8-4a91-8399-a96b5219cc00","correlation_id":"78576956-9146-47a3-b28c-226c2ecde3ec","error_uri":"https://login.microsoftonline.com/error?code=90002"}

Como se puede observar en la imagen la consola muestra que el “tenant” es incorrecto y que ha fallado el despliegue, por lo que en un caso como este habría que revisar la credencial.

Fallo en el despliegue - .NET

Ejecución de la pipeline:

La pipeline falla una vez detectado el error y dentro del log de la pipeline se observa que el id de suscripción no existe.

```
+ az account set -s 7f2d989c-2cfa-45c8-944c-328deee9629aº
ERROR: The subscription of '7f2d989c-2cfa-45c8-944c-328deee9629aº' doesn't exist in cloud 'AzureCloud'.
```

Fallo en el despliegue - Node.js

Ejecución de la pipeline:

	Declarative: Checkout SCM	Declarative: Tool Install	Install Dependencies	Build	Dockerizar app
Average stage times:	17s	4s	2h 6min	7s	1min 41s
#287 feb 04 23:49 No Changes	19s	4s	2min 32s	11s	1min 36s

Test	Sonar	Quality gate	Generate ZIP	Deploy to Azure	Declarative: Post Actions
1min 2s	29s	8s	11s	3s	54s
53s	1min 58s	15s	1min 42s	22s failed	30s

Como se ve la pipeline falla y se obtiene como se esperaba el mismo error que en la prueba anterior.

```
+ az account set -s 7f2d989c-2cfa-45c8-944c-328deee9629a@º
ERROR: The subscription of '7f2d989c-2cfa-45c8-944c-328deee9629a@º' doesn't exist in cloud 'AzureCloud'.
```

6.4 CSI 6: Elaboración de los Manuales de Usuario

6.4.1 Manual de Usuario

A continuación, se describe un manual de usuario en el que se enseña cómo se crea y se configura nuestro sistema desde cero.

Anotación:

- *Los links que hay a continuación son accesos rápidos al archivo o directorio mencionado en el repositorio Git del proyecto.*

6.4.1.1 Infraestructura del sistema pipeline

En este apartado se describirá como crear y configurar el entorno de Jenkins, SonarQube y Selenoid. Para ello primero se instalará Docker en nuestro ordenador ya que nuestra infraestructura utiliza contenedores Docker, en este caso se usa Windows por lo que se descarga Docker desktop.

6.4.1.1.1 Jenkins

Dado a que el proyecto involucra varios contenedores conectados entre sí, se utilizará Docker compose. Esto es un archivo YAML en el que podemos definir, configurar y ejecutar varios contenedores de forma coordinada.

La imagen de Jenkins se obtendrá mediante un Dockerfile en el cual se instalará Docker Compose ya que posteriormente se integrará en nuestras pipelines.

```
FROM jenkins/jenkins:lts

USER root

# Instalar Docker Compose
RUN apt-get update && \
  apt-get install -y curl && \
  curl -L "https://github.com/docker/compose/releases/download/v2.19.1/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose && \
  chmod +x /usr/local/bin/docker-compose

# Verificar la instalación
RUN docker-compose --version
```

En el directorio de trabajo se crea un archivo llamado docker-compose.yml en que se definirá el servicio Jenkins.

```
version: '3.8'

services:
 jenkins:
  build: .
  container_name: jenkins
  privileged: true
  user: root
  volumes:
   - .\jenkins_home:/var/jenkins_home
   - /var/run/docker.sock:/var/run/docker.sock
   - /usr/bin/docker:/usr/bin/docker:ro
  ports:
   - "9080:8080"
  networks:
   - backend
  deploy:
   resources:
    limits:
     cpus: "4.0"
     memory: "4GB"
```

(Mas información acerca del docker-compose)

Para levantar el contenedor se debe abrir la terminal en la ruta donde se encuentra el archivo docker-compose.yml y ejecutar el comando:

```
docker-compose up -d
```

Con el contenedor funcionando, se accede a Jenkins a través de la URL http://localhost:<9080>. Al ingresar, se mostrará una pantalla en la que se solicita una contraseña para desbloquear Jenkins. Esta contraseña se puede obtener consultando los registros del contendor mediante el comando:

```
docker logs -f <nombre del contenedor>
```

o, alternativamente, a través de la interfaz de Docker Desktop, haciendo clic en el contenedor de Jenkins.

Una vez ingresada la contraseña, Jenkins ofrece la opción de instalar los plugins recomendados o seleccionar otros de manera manual. En este caso, se opta seleccionar los recomendados ya que instala muchos de los que se necesitan.

Posteriormente, se solicitará la creación de un usuario administrador, por el que se pedirá proporcionar los datos necesarios. Finalmente, se procede a la configuración de la URL de Jenkins. En este ejemplo se ha optado por utilizar una URL basada en DNS. Para ello se debe editar el archivo C:\Windows\System32\drivers\etc\hosts y agregar la dirección ip junto con un nombre, en este caso docker.jenkins.

```
192.168.1.8 docker.jenkins
```

De este modo, se modifica la URL sustituyendo localhost por docker.jenkins quedando la URL así: http://docker.jenkins:<9080>/

6.4.1.1.2 Agentes de Jenkins

Una vez ya está configurado y corriendo Jenkins, se configurarán agentes que ejecuten las pipelines de las aplicaciones.

El primer paso es crear una imagen en un Dockerfile del agente en el que se le instale las herramientas necesarias para ejecutar algunas de las pipelines.

```
FROM jenkins/ssh-agent:bullseye-jdk17

USER root

RUN chmod g+rw /var/run/docker.sock
RUN groupadd docker || true && usermod -aG docker jenkins
RUN apt-get update && \
  apt-get install -y apt-transport-https ca-certificates curl gnupg2 software-properties-
common && \
  curl -fsSL https://download.docker.com/linux/debian/gpg | apt-key add - && \
  add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/debian
$(lsb_release -cs) stable" && \
```

```
  apt-get update && \
  apt-get install -y docker-ce docker-ce-cli docker-buildx-plugin docker-compose-plugin &&
\
  apt-get install --yes libcurl4 && \
  curl -L "https://github.com/docker/compose/releases/download/v2.19.1/docker-
compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose && \
  chmod +x /usr/local/bin/docker-compose && \
  curl -sL https://packages.microsoft.com/keys/microsoft.asc | gpg --dearmor -o
/etc/apt/trusted.gpg.d/microsoft.gpg && \
  echo "deb [arch=amd64] https://packages.microsoft.com/repos/azure-cli/ bullseye main"
> /etc/apt/sources.list.d/azure-cli.list && \
  apt-get update && \
  apt-get install -y azure-cli

USER jenkins
```

Esta imagen pertenece al agente de Node.js en el que se le instala:

- Docker: Para la creación de un contenedor con la aplicación Node.js.
- libcurl4: Librería necesaria para ejecutar las pruebas de Node.js.
- Docker-compose: Para levantar el contenedor de Selenoid
- Azure-cli: Para ejecutar comandos para el despliegue en Azure

En el docker-compose definimos nuestro servicio para el agente:

```
agent:
image: dockeragent
privileged: true
user: root
container_name: agent
networks:
        backend
expose:
        22
environment:
        JENKINS_AGENT_SSH_PUBKEY=ssh-rsa AAAAB3...
volumes:
        //var/run/docker.sock:/var/run/docker.sock
deploy:
resources:
limits:
cpus: "6.0"
memory: "6GB"
```

En este archivo Docker compose usamos la directiva **network** para que el agente se conecte a la misma red que el servicio Jenkins (la configuración de la network se detallará más adelante).

En la sección **environment**, se añade una clave pública SSH para permitir que el agente se autentique y acceda a Jenkins.

La directiva **volumes** monta el socket de Docker del host, permitiendo que el agente ejecute y controle contenedores Docker.

Configuración de clave publica:

- Se ejecuta ssh_keygen -f <nombre_cualquiera> en una terminal y se obtendrá dos ficheros, nombre_cualquiera y nombre_cualquiera.pub
- Para crear una credencial en Jenkins hay que ir a Administrar Jenkins > Credenciales > Global > Add Credential.
    - **Kind**: SSH Username with private key
    - **Id**: Jenkins
    - **Private Key**: Clic en el check y en añadir y se pega la public key generada al principio.
    - **Passphrase**: Vacio
- En el Docker compose se pega la key generada al principio.

Esta clave será también guardada como credencial en Jenkins.

Configuración del agente en Jenkins

Como el docker-compose.yml ya tiene todos los servicios de la infraestructura definidos solo tenemos que levantarlo una vez como se menciono antes, pero es importante buildear todas las imágenes de los agentes con los nombres que cada agente. Para ello hay que estar en el directorio de cada agente y ejecutar:

```
docker build -t dockerspringagent .
docker build -t dockernodeagent .
docker build -t dockernetagent .
```

Ahora toca configurar en Jenkins la presencia de un agente. Para ello iremos a **Administrar Jenkins > Nodos > Nuevo nodo,** le damos un nombre, seleccionamos nodo permanente y aceptamos.

Ahora en la pantalla de configuración del nodo toca configurar el agente en Jenkins.

- En el directorio raíz: /home/jenkins/<nombre_ del_agente>
- Usar: Utilizar este nodo tanto como sea posible
- Método de ejecución: Arrancar agentes remotos en máquinas tanto como sea posible
- Nombre de la maquina: nombre del agente
- Credentials: La creada previamente

6.4.1.1.3 SonarQube

En este caso no hay un Dockerfile con la imagen de SonarQube así que se especifica la imagen directamente en el docker compose. Además de SonarQube se crea la base de datos que usará SonarQube.

```
sonarqube:
  image: sonarqube:25.1.0.102122-community
```

```
  depends_on:
   - db
  ports:
   - "9000:9000"
  networks:
   - backend
  environment:
   SONAR_JDBC_URL: jdbc:postgresql://db:5432/sonarqube
   SONAR_JDBC_USERNAME: sonar
   SONAR_JDBC_PASSWORD: sonar
  deploy:
   resources:
    limits:
     cpus: "1.0"
     memory: "1GB"
  volumes:
   - sonarqube_data:/opt/sonarqube/data
   - sonarqube_extensions:/opt/sonarqube/extensions
   - sonarqube_logs:/opt/sonarqube/logs
   - sonarqube_temp:/opt/sonarqube/temp
  restart: on-failure
  container_name: sonarqube
 db:
  image: postgres:12-bullseye
  networks:
   - backend
  environment:
   POSTGRES_DB: "sonarqube"
   POSTGRES_USER: sonar
   POSTGRES_PASSWORD: sonar
  deploy:
   resources:
    limits:
     cpus: "1.0"
     memory: "256MB"
  volumes:
   - postgresql:/var/lib/postgresql
   - postgresql_data:/var/lib/postgresql/data
  restart: on-failure
  container_name: postgresql
```

(Mas información acerca del docker-compose)

La directiva **depends_on** establece la dependencia de SonarQube con la base de datos, denominada **db**.

La directiva **networks** define la red compartida con los demás contenedores, permitiendo que estén interconectados (la configuración de la network se verá más adelante). Por otro lado, la

sección **environment** se emplea para configurar la conexión de SonarQube con la base de datos.

Se levanta el contenedor y una vez esté funcionando entramos a la url http://localhost:<9000>/ o http://docker.sonarqube:<9000>/ (si tenemos configurado el dns como con Jenkins).

Al ingresar, se solicitará iniciar sesión. Dado que aún no se ha configurado ningún usuario, se utilizan las credenciales predeterminadas: **admin** tanto como nombre de usuario como contraseña. Posteriormente, SonarQube requerirá actualizar la contraseña, por lo que se deberá introducir una nueva contraseña de su elección. Finalmente, ya estaríamos en el dashboard de SonarQube listo para configurarlo.

6.4.1.1.4 Conexión entre Jenkins y SonarQube

Para establecer una conexión entre Jenkins y SonarQube primero hay que conectar los contenedores. Esto se hace creando una network en el docker compose.

```
networks:
 backend:
  name: network_jenkins
 host:
  external: true
```

De esta manera se crea una network llamada network_jenkins que poniendo external puede acceder otros contenedores que no estén en la misma docker compose.

Una vez conectados se crea un token y un webhook en SonarQube. El token sirve para que Jenkins tenga acceso a SonarQube, mientras que el webhook sirve para lanzar las acciones de análisis.

Crear token

Para crear el token hay que dirigirse a **Administration > Security > Users > Tokens**. Se abrirá una ventana emergente en la que se deberá asignar un nombre al token (por ejemplo, "Jenkins") y hacer clic en "Generar". El sistema creará un token que, posteriormente, deberá almacenarse en las credenciales de Jenkins. Para ello, se accede a **Administrar Jenkins > Manage Credentials > Jenkins > Global Credentials > Add Credential**, se selecciona el tipo de credencial **secret text**, se introduce el token en el campo correspondiente y se asigna el identificador **SonarQube**.

Crear webhook

Para configurar el webhook, se va a **Administration > Configuration > Webhooks > Create**. Se asigna un nombre de preferencia y en el campo URL se ingresa: http://jenkins:<puerto_jenkins>/sonarqube-webhook. Se utiliza “jenkins” porque es el nombre del servicio asignado a Jenkins en el archivo Docker Compose. Esto es posible ya que anteriormente se conectó ambos servicios mediante la network creada. Finalmente se clica en crear.

Agregar el Plugin de SonarQube en Jenkins

Ahora falta agregar a Jenkins el plugin de SonarQube. Esto se hace en **Administrar Jenkins > Administrar Plugins** y se busca sonarqube. Se instala y a continuación se va a **Administrar**

Jenkins > Configurar el Sistema y se localiza la sección SonarQube servers. Se añade una nueva instalación y se agrega un nombre (por ejemplo, "sonarqube") y una URL http://sonarqube:<puerto_sonarqube>. Aquí, "sonarqube" corresponde al nombre del servicio configurado para el contenedor de SonarQube. Por último, se introduce el token creado anteriormente.

6.4.1.1.5 Selenoid

Para este parte de la infraestructura se definirán dos servicios, el de Selenoid y Selenoid-ui, este último es la interfaz de Selenoid donde se puede ver cómo se ejecutan los test.

```
services:
 selenoid:
  image: aerokube/selenoid:latest-release
  ports:
   - "4444:4444"
  volumes:
   - "/var/run/docker.sock:/var/run/docker.sock"
   - "./config:/etc/selenoid"
   - "./target:/output"
   - "./target:/opt/selenoid/video"
  command: ["-service-startup-timeout", "4m", "-session-attempt-timeout", "4m", "-video-
output-dir", "/opt/selenoid/video", "-container-network", "network_jenkins"]
  networks:
   - selenoid
 selenoid-ui:
  image: aerokube/selenoid-ui:latest-release
  links:
   - selenoid
  ports:
   - "8070:8080"
  command: ["--selenoid-uri", "http://selenoid:4444"]
  networks:
   - selenoid
networks:
 selenoid:
  name: network_jenkins
  external: true
```

(Mas información acerca del servicio de selenoid)

Para el servicio ui de Selenoid se usa la imagen aerokube/selenoid-ui:latest-release y se usa links para establecer una conexión con el contenedor de Selenoid.

Se mapea el puerto 8070 del host al puerto 8080 del contenedor, de modo que se pueda acceder a la interfaz usando el puerto 8070. Esto se hace ya que otro servicio ya tiene ocupado el puerto 8080.

Con la directiva command se establece el parámetro –selenoid-uri http://selenoid:4444 para indicar a la interfaz dónde se encuentra el servicio de Selenoid.

Ambos servicios se conectan a la red selenoid la cual hace referencia a la red externa network_jenkins definida anteriormente. Se usa external:true para indicar que esta red ya existe en Docker, por lo que no intentará crearla.

Configuración browser.json

El archivo browsers.json es un archivo donde se definen las imágenes Docker que corresponden a cada navegador y versión, de manera que cuando el test pida un Chrome 119.0, Selenoid sepa que contenedor tiene que levantar.

Estructura básica:

```
{
 "chrome": {
   "default": "119.0_VNC",
   "versions": {
    "80.0": {
     "image": "selenoid/chrome:80.0",
     "port": "4444",
     "path": "/"
    },
    "80.0_VNC": {
     "image": "selenoid/vnc:chrome_80.0",
     "port": "4444",
     "path": "/"
    },
    "128.0_VNC": {
     "image": "selenoid/vnc_chrome:128.0",
     "port": "4444",
     "path": "/",
     "tmpfs": {"/tmp": "size=512m"}
    },
    "119.0_VNC": {
     "image": "selenoid/vnc_chrome:119.0",
     "port": "4444",
     "path": "/"
    }
   }
 }
}
```

(Link de la carpeta con toda la configuración de la infraestructura)

Es importante descargarse la imagen de los navegadores para que se puedan utilizar, esto se hace ejecutando el siguiente comando para distintos navegadores:

```
docker pull selenoid/vnc_chrome:119.0
docker pull selenoid/firefox:latest
```

6.4.1.1.6 Conexión entre un repositorio Git y Jenkins para lanzar pipelines

Cuando se desea ejecutar pipelines desde un repositorio Git, es necesario configurar un webhook que envíe eventos a Jenkins a través de una URL pública. En este proyecto, Jenkins se ejecuta en un contenedor local por lo que la URL localhost no es accesible desde el exterior.

Para solucionar este problema, se puede usar una herramienta que exponga el puerto que usa el contenedor Jenkins y genere una URL pública accesible desde Internet. Este proyecto utiliza Ngrok.

Configuración de Ngrok

Lo primero que hay que hacer es descargar e instalar la herramienta. Una vez instalada se ejecuta la herramienta, la cual abrirá una terminal y se ejecuta el siguiente comando:

```
ngrok http <puerto_contenedor_jenkins>
```

Al ejecutar este comando, se generará una url publica que se usará para configuración del webhook.

Configuración del webhook en GitHub

Se accede a la configuración del repositorio y se selecciona en el menú “Webhooks” y se hace clic en “Add webhook”.

Ahora en el campo “Payload URL” se introduce la URL pública generada anteriormente y se añade al final “/github-webhook/” u otro nombre. En eventos se elige “Just the push event” para que se ejecute la pipeline al hacer push al repositorio y le damos a “add webhook” para terminar la configuración.

Ahora ya se pueden lanzar pipelines cada vez que se hace un push al repositorio.

Configuración del webhook en GitLab

Se accede a la sección Setting > Webhooks, se introduce la URL pública generada anteriormente y se añade al final “/gitlab-webhook/” u otro nombre. Se elige “Push events” para que se ejecute la pipeline al hacer push al repositorio y ya bastaría para tener nuestro webhook en GitLab configurado.

Configuración del webhook en Bitbucket

En Bitbucket hay que ir a la sección **Repository settings > Webhooks > Add webhook**. Se escribe un nombre, la URL creada anteriormente junto con “/bitbucket-hook/” u otro nombre y se selecciona la opción “Push” en eventos y se guarda.

6.4.1.2 Creación de una pipeline

Con la infraestructura de Jenkins y SonarQube ya establecida, a continuación, se describe el proceso para crear una pipeline.

1. Desde la pantalla de inicio de Jenkins, se selecciona la opción para crear una nueva tarea.
2. Se ingresa un nombre para la tarea, se elige la opción Multibranch Pipeline y se hace clic en OK
3. En la sección Branch Sources de la configuración, se añade el repositorio de la aplicación. Si el reposirtorio es privado, será necesario generar un token de acceso para que Jenkins pueda conectarse correctamente. Dicho token se almacenerá como credencial en Jenkins.

Ahora que Jenkins ya tiene acceso al repositorio, hay que crear el flujo de la pipeline en un Jenkinsfile. Este Jenkisfile puede ser escrito usando dos sintaxis diferentes:

- Pipeline Declarativo:
  - Utiliza una sintaxis más estructurada, lo que facilita la legibilidad y el mantenimiento.
  - Se define mediante la palabra clave pipeline y suele incluir bloques agent, stages y steps
- Pipeline Scripted
  - Se basa en groovy y permite mayor flexibilidad, ya que se escribe de forma imperativa.
  - Se utiliza bloques como node
  - Es útil cuando se requieren flujos de trabajo o lógica que no encajan bien en la sintaxis.

Ahora toca elegir que sintaxis usar y las etapas que van a haber en la pipeline. En este proyecto se ha optado por usar la sintaxis del Pipeline Declarativo. Cada aplicación va a tener distintas etapas:

- **Spring**: Build -> Dockerizar App -> Test -> Reporting -> Sonar -> Quality Gate -> Deploy to Azure
- **Node.js**: Build -> Dockerizar App -> Test -> Reporting -> Sonar -> Quality Gate -> Generate ZIP -> Deploy to Azure
- **.NET**: Restore dependencies -> Install tools -> Build -> Dockerizar App -> Test -> Reporting -> Sonar -> Quality Gate -> Pack Nuget Package -> Publish to Nuget -> Zip project -> Deploy to Azure

A continuación, se describirá que hace cada etapa:

- **Build**: Etapa donde se compila el código fuente y se detectan posibles errores de compilación.
- **Dockerizar App**: Se empaqueta la aplicación en una imagen Docker, permitiendo ser ejecutado posteriormente y facilitar que los test de interfaz de usuario puedan interactuar con ella.

- **Test**: Se ejecutan todas las pruebas definidas para la aplicación.
- **Reporting**: Se generan reportes detallados de los resultados obtenidos de las pruebas
- **Sonar**: Se realiza un análisis de la calidad del código utilizando SonarQube
- **Quality Gate**: Se comprueba que el código cumple con los estándares mínimos de calidad establecidos en SonarQube.
- **Deploy to Azure**: Se despliega la aplicación en la infraestructura de Azure. En la pipeline de la aplicación de Spring se usa el contenedor Docker para el despliegue, mientras que en las demás se usan archivos ZIP que contienen el código de la aplicación.
- **Generate ZIP/ZIP Project**: Se empaqueta la aplicación y sus artefactos en un archivo ZIP
- **Pack Nuget Package**: Se empaqueta la aplicación en un paquete NuGet.
- **Publish to Nuget**: Se publica el paquete NuGet generado en un repositorio, lo que posibilita la reutilización del proyecto en otros desarrollos.

6.4.1.2.1 Configuración pipeline del proyecto Spring

Aquí se describe la configuración necesaria para la pipeline del proyecto de Spring

```
@Library('mi-libreria') _
pipeline{
  agent {label 'jenkins_agent_spring'}
  environment {
  AZURE_CREDENTIALS = credentials('Azure')
  }
  tools {
       // Install the Maven version configured as "M3" and add it to the path.
       maven "MAVEN_HOME"
       allure "allure"
    }
```

- @Library importa una librería llamada mi-librería. Está librería será explicada mas adelante.
- Con la directiva agent se especifica el agente que va a ejecutar la pipeline con la etiqueta ‘jenkins_agent_spring’
- La directiva environment sirve para definir variables de entorno, en este caso se definen las credenciales de Azure que se usarán posteriormente para el despliegue de la aplicación.
- En tools se especifican herramientas que se necesitan durante la ejecución de la pipeline. En este caso se instala Maven

6.4.1.2.2 Configuración pipeline del proyecto Node.js

Aquí se describe la configuración necesaria para la pipeline del proyecto de Node.js

```
@Library('mi-libreria') _
pipeline {
  agent { label 'jenkins_agent_node'}
  tools {
```

```
    nodejs "nodejs"
  }
```

- @Library importa una librería llamada mi-librería.
- Se se especifica el agente que va a ejecutar la pipeline con la etiqueta 'jenkins_agent_node' (El agente de Node.js)
- Para esta pipeline vamos a usar la herramienta nodejs para que se pueda ejecutar el proyecto de Node.js.

6.4.1.2.3 Configuración pipeline del proyecto .NET

Aquí se describe la configuración necesaria para la pipeline del proyecto de .NET

```
pipeline {
  agent { label 'jenkins_agent_net' }
  tools {      allure "allure"    }
  environment {
    PATH="$PATH:/root/.dotnet/tools"
    SONAR_TOKEN = credentials('Sonarqube')
    NUGET_API_KEY = credentials('NuGet')
    AZURE_APP_NAME = 'netmovies'
    RESOURCE_GROUP = 'TFG'
  }
```

- @Library importa una librería llamada mi-librería.
- Con la directiva agent se especifica el agente que va a ejecutar la pipeline con la etiqueta 'jenkins_agent_net'. Este es el agente para ejecutar proyectos .NET.
- En este caso se definien las credenciales de SonarQube y Nuget. También se asigna al path donde esta dotnet para ejecutar comandos para proyectos .NET.
- En tools se especifican herramientas que se necesitan durante la ejecución de la pipeline. En este caso se instala allure para la creación de reportes

6.4.1.3 Configuración del despliegue de una aplicación en Azure

A continuación, se presentará la configuración necesaria para desplegar una aplicación en Azure. Hay dos maneras en las que se puede configurar un entorno en Azure, a través de la interfaz de Azure o a través de la CLI que nos proporciona. En este proyecto se ha optado por usar la CLI de Azure.

6.4.1.3.1 Configuración en Azure

En esta explicación se usará de ejemplo la configuración de spring.

Creación de la App Service

Para la creación de la App Service se usarán los siguientes comandos en la CLI de Azure:

Creación del grupo de recursos

```
az group create \
```

```
  --name TFG \
  --location westeurope
```

Creación del plan gratuito de App Service

```
az appservice plan create \
  --name springservicetfg \
  --resource-group TFG \
  --sku F1 \
  --is-linux
```

Creacion de la Web App

```
az webapp create \
  --resource-group TFG \
  --plan springservicetfg \
  --name tfgspring \
  --runtime "JAVA|17-java17" \
  --deployment-local-git
```

Para los proyectos de Node.js y .NET se cambiaria la configuración runtime por “NODE|18-lts” y "DOTNETCORE|8.0" respectivamente.

6.4.1.3.2 Configuración en Jenkins para despliegue

Credenciales

Primero hay que crear unas credenciales en Jenkins con el clientId, el clientSecret y el tenantId. Para generarlo se usa:

```
az ad sp create-for-rbac --sdk-auth
```

Esta credencial se configurará en **Jenkins > Manage Jenkins > Credentials > Global** y se pueden guardar de dos maneras:

- Azure Service Principal: Se añade una de tipo Azure Service Principal y se rellenan los campos con los datos obtenidos anteriormente.
- Username y Password: Aquí se guarda el clientId como Username y el clientSecret como contraseña.

Configuracion de la pipeline

El primer paso es autenticarse en Azure, para ello se usarán las crendeciales creadas anteriormente.

En caso de haber creado las credenciales de la primera manera se usará la siguiente manera:

```
withCredentials([azureServicePrincipal('AZURE_CREDENTIALS')]) {
            sh '''
              az login --service-principal \
                -u $AZURE_CLIENT_ID \
                -p $AZURE_CLIENT_SECRET \
                --tenant $AZURE_TENANT_ID
```

```
            '''
        }
```

Si se ha usado la segunda se usará la siguiente manera:

```
withCredentials([usernamePassword(credentialsId: 'Azure', passwordVariable:
'AZURE_CLIENT_SECRET', usernameVariable: 'AZURE_CLIENT_ID')]) {
                    sh '''
                      az login --service-principal -u $AZURE_CLIENT_ID -p
$AZURE_CLIENT_SECRET -t $AZURE_TENANT_ID
                      az account set -s $AZURE_SUBSCRIPTION_ID
                    '''
                  }
```

En este caso la tenant se guardaría en una variable en la propia pipeline o se crearía otra credencial.

Una vez autenticado se procederá con el despliegue que se hará usando el siguiente comando:

```
sh """
  az webapp deploy \
  --resource-group $resourceGroup \
  --name $webAppName \
  --type jar \
  --src-path $pathToJAR
 """
```

En caso de los proyectos Node.js y .NET se usará zip en vez de jar.

Una vez ya está desplegado se cerrará sesión usando:

```
sh 'az logout'
```

6.4.1.4 Configuración de una librería en Jenkins

En la sección 6.2.7 Jenkins_lib se habló de la estructura de una librería y el contenido que había en él, una vez está creada esta librería en un repositorio Git hay que configurarla:

Para configurar la librería en Jenkins se va a **Administrar Jenkins** > **Configuración del sistema** a la sección “Global Pipeline Libraries” (Importante tener el plugin instalado).

En esta sección se escribe un nombre, la url del repositorio de Git (https://github.com/UO276840/jenkins_lib/tree/main), la Branch a la que se quiere acceder y las credenciales para acceder al repositorio.

Para usar una función en el Jenkinfile hay que incluir la declaración **@Library(‘nombreLibreria’)** _ al inicio del Jenkinsfile.

(Link de Jenkinsfile usando una librería)

6.4.1.5 Creación de un plugin para Jenkins

6.4.1.5.1 Requisitos previos

Para creación del plugin para Jenkins es necesario tener instaladas y configuradas las siguientes herramientas:

- JDK (Java 8)
- Apache Maven: Es la herramienta usada para la construcción y empaquetado del plugin

6.4.1.5.2 Creación del proyecto del plugin

Para la creación del plugin se usará un comando proporcionado por la comunidad de Jenkins que generará la base del proyecto:

```
mvn -U archetype:generate -Dfilter=io.jenkins.archetypes:
```

Este comando listará varios arquetipos disponibles. En este caso se escoge el arquetipo "plugin".

Ahora se solicitarán varios datos:

- GroupId
- ArtifactId
- Version
- Package

Rellenado estos datos se generará la estructura del proyecto.

6.4.1.5.3 Desarrollo del plugin

Para el desarrollo del plugin se siguió la estructura mencionada en la seccion 5.2.5 pero aunque no se ha usado para este proyecto, está la posibilidad de configurar los parámetros desde la interfaz gráfica a través de un archivo config.jelly.

```
<?jelly escape-by-default='true'?>
<j:jelly xmlns:j="jelly:core" xmlns:st="jelly:stapler" xmlns:d="jelly:define"
xmlns:l="/lib/layout" xmlns:t="/lib/hudson" xmlns:f="/lib/form">
  <f:entry title="Webhook URL" field="webhookUrl">
      <f:textbox />
    </f:entry>
    <f:entry title="Channel" field="channel">
      <f:textbox />
    </f:entry>
    <f:entry title="Message" field="message">
      <f:textarea />
    </f:entry>
</j:jelly>
```

El nombre del field hace referencia al nombre puesto a los parámetros del constructor.

6.4.1.5.4 Integración y uso en Jenkins

Para integrarlo con Jenkins hay que obtener un archivo .hpi e instalarlo en Jenkins y para ello hay que compilar el plugin:

```
mvn clean package
```

Esto empaquetará el plugin en un archivo .hpi.

Antes de instalarlo en Jenkins se puede probar haciendo uso de:

```
mvn hpi:run
```

Esto iniciará una instancia de Jenkins en http://localhost:8080/ con el plugin “instalado” y así probar el plugin.

Para instalarlo en Jenkins hay que acceder a **Manage Jenkins** > **Manage Plugins** e ir a la pestaña Advanced. Desde ahí en el apartado Upload Plugin se carga el archivo .hpi generado y se reinicia Jenkins.

Ahora ya instalado solo queda usar el plugin en nuestra pipeline, en este proyecto se usa para notificar cuando a finalizado la pipeline, el estado y en caso de fallar en que etapa.

Uso del plugin en un Jenkinsfile:

```
simpleSlackNotifier webhookUrl: 'https://hooks.slack.com/services/T07JQ3MBK4P...,
                    channel: '#tfg',
                    message: ' ¡La build fue exitosa!'
```

(Link de un Jenkinsfile usando el plugin)

6.4.1.6 Configuración de reporting

6.4.1.6.1 Instalación del plugin

En **Administrar Jenkins > Gestión de Plugins > Disponible** se busca e instala **Allure Jenkins Plugin**

Después en la sección **Administrar Jenkins > Gestión de Herramientas Globales > Allure Commandline** se añade una entrada con nombre Allure y una versión, en este caso 2.19.0.

De este modo, en pipelines podrás invocar tool name: 'Allure'

6.4.1.6.2 Configuración de cada proyecto

En este apartado se explicará como configurar cada proyecto para que el plugin recoja los resultados de las pruebas.

Configuración proyecto Spring

Para este proyecto se necesita instalar el plugin en el pom.xml, una vez instalado cuando ejecutemos los test se creará una carpeta target/allure-results en la que se guardará los resultados de los tests.

En la pipeline se ejecutará el comando allure que crea la interfaz donde poder visualizar los resultados de todos los tests.

```
allure([
        includeProperties: false,
        jdk: '',
        properties: [],
        reportBuildPolicy: 'ALWAYS',
        results: [[path: target/allure-results]]
    ])
```

Configuración proyecto Node.js

Para este proyecto se necesita instalar mocha-allure-reporter para que mocha pueda generar los resultados de los tests en allure-results. Para ejecutar los tests se usará:

```
npx nyc mocha --reporter mocha-allure-reporter --no-timeouts --exit ./tests/*
```

En la pipeline se ejecutará el comando allure para generar la interfaz:

```
allure([
        includeProperties: false,
        jdk: '',
        properties: [],
        reportBuildPolicy: 'ALWAYS',
        results: [[path: allure-results]]
    ])
```

Configuración proyecto .NET

Para el proyecto .NET hay que instalar los paquetes Allure.Commons y Allure.Xunit. Posteriormente crearemos los resultados usando:

```
dotnet test ./TestTiendaOnline --configuration Release --logger
"trx;LogFileName=test_results.trx"
```

Para crear la interfaz:

```
allure([
                results: [[path: "./TestTiendaOnline/TestResults"]],
                reportBuildPolicy: 'ALWAYS'
                ])
```

6.4.2 Manual de Instalación en Linux

A pesar de que el proyecto ha sido configurado en Windows, también se puede usar la misma configuración en una maquina Linux. Para probar esto se ha instalado y configurado Jenkins en una máquina virtual con Ubuntu 22.04. La única diferencia es la instalación de Docker en Linux pero se expone brevemente la instalación completa hasta la ejecución de la pipeline.

6.4.2.1 Instalación de Docker

Pasos:

1. **Actualizar sistema**

```
sudo apt update && sudo apt upgrade -y
```

2. **Instalación de dependencias**

```
sudo apt install -y ca-certificates curl gnupg lsb-release
```

3. **Añadir la clave GPG oficial de Docker**

```
sudo mkdir -p /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
```

4. **Configurar el repositorio de Docker**

```
echo \
"deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] \
https://download.docker.com/linux/ubuntu \
$(lsb_release -cs) stable" | \
sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

5. **Instalar Docker Engine y Docker Compose Plugin**

```
sudo apt install -y docker-ce docker-ce-cli containerd.io docker-compose-plugin
```

6.4.2.2 Descarga del Proyecto

```
git clone https://github.com/UO276840/AutoFlow.git
```

6.4.2.3 Ajuste de límites de recursos

En el fichero docker-compose.yml, modifica los límites para ajustarlo a la maquina donde se ejecuten los servicios.

6.4.2.4 Arranque de contenedores

Dentro de la carpeta DockerContainers buildeamos todas las imágenes de los agentes

```
sudo docker build -t ./agenteNet
sudo docker build -t ./agenteNode
sudo docker build -t ./agenteSpring
```

Hacemos pull de las imagenes de los navegadores que usa selenoid

```
sudo docker pull selenoid/firefox:latest
```

Arrancamos contenedores

```
sudo docker compose up -d
```

6.4.2.5 Configuración inicial de Jenkins

1. Se accede a http://localhost:9080/
2. Obtenemos la contraseña inicial para introducirla en Jenkins:

   ```
   sudo docker exec Jenkins car /var/Jenkins_home/secrets/initialAdminPassword
   ```

3. Instalación de plugins
   - Plugins recomendados
   - Pipeline: Shared Groovy Libraries
   - Maven Integration Plugin
   - Allure Jenkins Plugin
   - JaCoCo Plugin
   - Docker Pipeline
   - SonarQube Scanner for Jenkins
   - Azure CLI Plugin
   - Simple Slack Notifier Plugin
4. Se crea un usuario administrador
5. Se configuran agentes

6.4.2.6 Configuración de Sonarqube

1. Se accede a http://localhost:9000
2. Usuario y clave por defecto: admin/admin
3. Se cambia la contraseña inicial
4. Se genera un token en My Account > Security y guarda la credencial en Jenkins.
5. Se configura el instalador de SonarQube en Jenkins

6.4.2.7 Configuración de credenciales y entornos

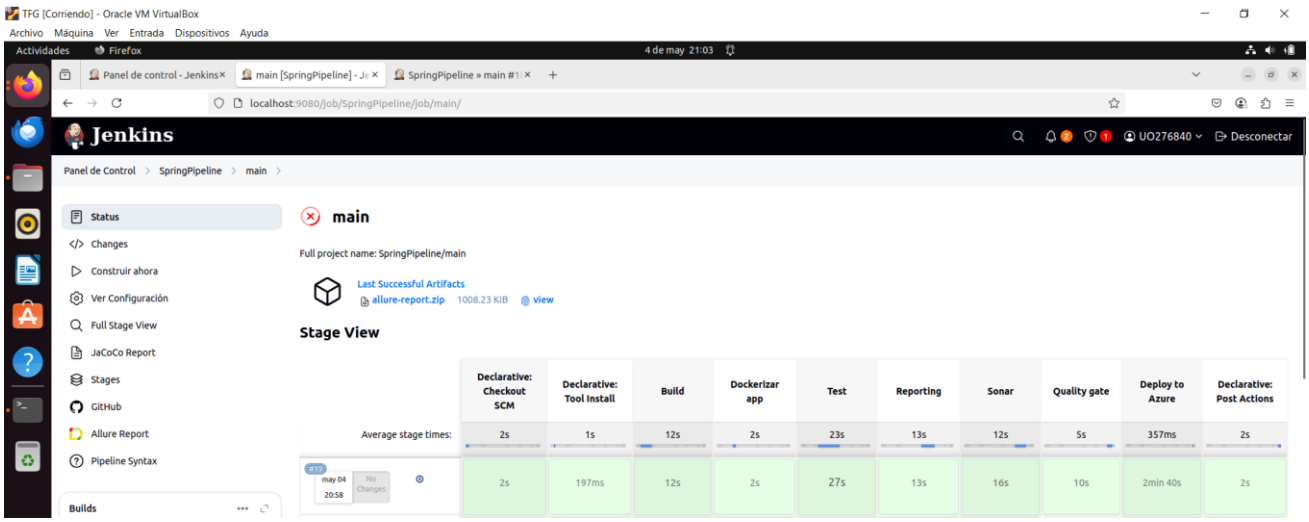
Se añaden las credenciales de Azure y Git y todas las instalaciones necesarias (Maven, Librería, Allure)

6.4.2.8 Creación de la pipeline

1. En Jenkins: New Item > Pipeline
2. Se define el repositorio Git y el Branch

3. Se guarda y ejecuta con Build Now.

Hecho todo esto se probó a ejecutar la pipeline de una de las aplicaciones, en este caso fue la del proyecto Spring, y comprobamos el correcto comportamiento:



6.4.3 Manual de Ejecución

Para ejecutar el sistema pipeline es necesario tener todos los contenedores corriendo y para que funcione docker en los agentes necesitamos ejecutar los siguientes comandos en la terminal de los agentes:

```
chown :docker /var/run/docker.sock
chmod g+rw /var/run/docker.sock
```

Esto sirve para darle permiso a los agentes para que puedan usar docker.

Plugins que hay que instalar que no están en los recomendados:

- Pipeline: Shared Groovy Libraries
- Maven Integration Plugin
- Allure Jenkins Plugin
- JaCoCo Plugin
- Docker Pipeline
- SonarQube Scanner for Jenkins
- Azure CLI Plugin
- NodeJS Plugin
- Badge
- Pipeline Stage View Plugin
- HTML Publisher Plugin

Capítulo 7 Conclusiones y Ampliaciones

7.1 Conclusiones

El proyecto se ha completado satisfactoriamente, dando como resultado una infraestructura CI/CD funcional, adaptable y reutilizable que automatiza el ciclo de integración, prueba y despliegue de distintos entornos tecnológicos (Spring Boot, Node.js y .NET). El principal valor de este proyecto radica en la integración de herramientas como Jenkins, SonarQube, Selenoid y Azure dentro de un flujo de desarrollo para que sirva como guía y punto de partida para otros desarrolladores que quieran implementar su propia infraestructura.

Este sistema permite validar la calidad del código, realizar pruebas, analizar resultados y realizar despliegues en la nube. Gracias a la librería común se favorece a la reutilización de lógica entre los distintos proyectos, lo que simplifica su mantenimiento y una extensión futura.

Durante el desarrollo de este proyecto, se consolidaron y ampliaron conocimientos técnicos relacionados con prácticas DevOps, integración y entrega continua CI/CD y el uso de contenedores Docker. También he podido aplicar conceptos aprendidos en el grado. En gran parte la asignatura de Calidad, Validación y Verificación Software para la creación de la infraestructura de pipelines, análisis de código estático, plan de pruebas y su ejecución. También Dirección y Planificación de Proyectos Informáticos en la planificación del proyecto, así como la documentación. Sistemas Distribuidos e Internet para el proyecto de Node.js y Spring Boot,

A nivel personal, este trabajo se ha desarrollado en paralelo con el trabajo laboral, lo cual ha supuesto un reto importante de organización y ha conllevado a retrasar el desarrollo del trabajo. A pesar de todo esto, el sistema es totalmente funcional y cumple con todos los requisitos establecidos.

En términos generales, este proyecto me ha permitido aplicar de forma práctica todo lo que aprendí durante la carrera y ampliar mi conocimiento, mejorando habilidades técnicas que me servirán en el mundo laboral. La solución desarrollada puede servir como referencia para otras personas que estén interesadas en la automatización de procesos de desarrollo y despliegue.

7.2 Ampliaciones

Este sistema aún puede crecer y ser ampliado, una posible ampliación sería la de trasladar el servidor Jenkins como sus agentes de compilación dentro de un clúster Kubernetes de manera que los agentes aparezcan de forma dinámica como pods y el clúster ajuste automáticamente el número de nodos en función de la carga de trabajo. Esto pude mejorar el rendimiento y la

eficiencia del sistema lo que reduciría tiempos de espera en la ejecución de pipelines y facilitaría la experiencia en entornos con múltiples proyectos o builds concurrentes.

Otra ampliación puede ser la ampliar el plugin para enviar mensajes a otros servidores y recibir comandos del chat que ejecuten pipelines. Esto mejoraría la productividad de los desarrolladores al permitirles interactuar con el sistema CI/CD directamente desde su entorno de trabajo diario, agilizando acciones comunes como puede ser desplegar.

Referencias Bibliográficas

[1] J. M. Redondo, «Documentos-modelo para Trabajos de Fin de Grado/Master de la Escuela de Informática de Oviedo,» 17 6 2019. [En línea]. Available: https://www.researchgate.net/publication/327882831_Plantilla_de_Proyectos_de_Fin_de_Carrera_de_la_Escuela_de_Informatica_de_Oviedo.

[2] J. Redondo, «Creación y evaluación de plantillas para trabajos de fin de grado como buena práctica docente.,» Revista de Innovación y Buenas Prácticas Docentes, p. pp, 2020.

[3] Robin Arellano, 2. Ejecutar Jenkins en Docker, (10 de septiembre de 2022). [En línea Video]. Disponible en: https://www.youtube.com/watch?v=Xc6lGvPZ2uA

[4] A. A. community Ivan Krutov, Kirill Merkushev and the Aerokube, «Aerokube Selenoid | A cross browser Selenium solution for Docker». [En línea]. Disponible en: https://aerokube.com/selenoid/latest

[5] S. Kumar, «Answer to “permission denied when executing the jenkins sh pipeline step”», Stack Overflow. [En línea]. Disponible en: https://stackoverflow.com/a/71815690

[6] dbradish-microsoft, «az spring app». [En línea]. Disponible en: https://learn.microsoft.com/en-us/cli/azure/spring/app?view=azure-cli-latest

[7] «Browser navigation», Selenium. [En línea]. Disponible en: https://www.selenium.dev/documentation/webdriver/interactions/navigation/

[8] O. Olayemi, «Browsers in containers — executing robot framework tests with selenoid (Part 2)», Medium. [En línea]. Disponible en: https://sejuba.medium.com/browsers-in-containers-executing-robot-framework-tests-with-selenoid-part-2-2ca76059d24e

[9] «Build and Run the Plugin», Build and Run the Plugin. [En línea]. Disponible en: https://www.jenkins.io/doc/developer/tutorial/run/

[10] abadraja, «Cannot connect to the Docker daemon at unix:///var/run/docker.sock. Is the docker daemon running? inside a Dockerfile», Stack Overflow. [En línea]. Disponible en: https://stackoverflow.com/q/51857634

[11] Chanafot, «Cannot execute SonarScanner for .NET project on Jenkins because “dotnet-sonarscanner: not found”», Stack Overflow. [En línea]. Disponible en: https://stackoverflow.com/q/64017095

[12] «Cannot find module “internal/modules/cjs/loader.js” [Solved] | bobbyhadz». [En línea]. Disponible en: https://bobbyhadz.com/blog/javascript-cannot-find-module-loader-js

[13] Sentrio, «CircleCI: Una herramienta para la integración continua», Sentrio. [En línea]. Disponible en: https://sentrio.io/blog/circleci-herramienta-integracion-continua/

[14] «Como crear un usuario en Debian y darle permisos de SUDO». [En línea]. Disponible en: https://laboratoriolinux.es/index.php/-noticias-mundo-linux-/distribuciones/24452-como-crear-un-usuario-en-debian-y-darle-permisos-de-sudo.html

[15] «Cómo crear una estructura de equipo de desarrollo de software | Innowise». [En línea]. Disponible en: https://innowise.com/es/blog/how-to-build-software-development-team-structure/

[16] craigloewen-msft, «Configuración avanzada en WSL». [En línea]. Disponible en: https://learn.microsoft.com/es-es/windows/wsl/wsl-config

[17] «Contenerización de aplicaciones en Docker», No Country for Geeks. [En línea]. Disponible en: https://www.nocountryforgeeks.com/contenerizacion-de-aplicaciones-en-docker/

[18] «Declarative vs Scripted Pipeline – Key differences - DS Stream Blog». [En línea]. Disponible en: https://www.dsstream.com/post/declarative-vs-scripted-pipeline-key-differences

[19] M. de Assis, «Deploy a .NET 7 application on a Ubuntu VPS with Nginx and Jenkins», Medium. [En línea]. Disponible en: https://medium.com/@assis4002/deploy-a-net-7-application-on-a-ubuntu-vps-with-nginx-and-jenkins-e56fc3e914a0

[20] «Difference between Selenium Remote Webdriver and Selenium Webdriver», GeeksforGeeks. [En línea]. Disponible en: https://www.geeksforgeeks.org/difference-between-selenium-remote-webdriver-and-selenium-webdriver/

[21] «docker compose up», Docker Documentation. [En línea]. Disponible en: https://docs.docker.com/reference/cli/docker/compose/up/

[22] «Dockerizar aplicación Spring Boot - DeveloProgramming- Docker Java», DeveloProgramming. [En línea]. Disponible en: https://developrogramming.com/dockerizar-aplicacion-spring-boot/

[23] V. Pecanac, «Dockerizing ASP.NET Core Application With Dockerfiles», Code Maze. [En línea]. Disponible en: https://35.226.11.130/aspnetcore-app-dockerfiles/

[24] «Dotnet SonarScanner for MSBuild begin error in Jenkins (agent dockerfile) - SonarQube», Sonar Community. [En línea]. Disponible en: https://community.sonarsource.com/t/dotnet-sonarscanner-for-msbuild-begin-error-in-jenkins-agent-dockerfile/78809

[25] «ECMAScript Modules · Jest». [En línea]. Disponible en: https://jestjs.io/docs/ecmascript-modules

[26] Borja, «Ejemplo de alcance del sistema según ISO 9001:2015», Arrizabalaga Consulting 4.0 Agile. [En línea]. Disponible en: https://arrizabalagauriarte.com/en/ejemplo-alcance-del-sistema-segun-iso-90012015/

[27] «ERROR webdriver: RequestError: connect ECONNREFUSED 127.0.0.1:???? · webdriverio/webdriverio · Discussion #12661», GitHub. [En línea]. Disponible en: https://github.com/webdriverio/webdriverio/discussions/12661

[28] «Extending with Shared Libraries», Extending with Shared Libraries. [En línea]. Disponible en: https://www.jenkins.io/doc/book/pipeline/shared-libraries/

[29] C. A. Allen, «Generating Code Coverage Metrics for .NET Framework Applications», Coding with Calvin - Calvin Allen. [En línea]. Disponible en: https://www.codingwithcalvin.net/generating-code-coverage-metrics-for-net-framework-applications/

[30] «GitLab CI/CD vs. GitHub Actions», Graphite.dev. [En línea]. Disponible en: https://graphite.dev/guides/gitlab-cicd--vs-github-actions

[31] «Guía completa de integración de C# con Selenium para automatizar pruebas web». [En línea]. Disponible en: https://www.gyata.ai/es/c-sharp/c-with-selenium/

[32] M. Greenwood, «How to exclude files from coverage in SonarQube», HatchJS.com. [En línea]. Disponible en: https://hatchjs.com/sonarqube-exclude-files-from-coverage/

[33] «How to Use the Host Network in Docker Compose». [En línea]. Disponible en: https://www.squash.io/tutorial-host-network-in-docker-compose/

[34] KarlErickson, «Implementación de aplicaciones en Spring Boot desde la CLI de Azure». [En línea]. Disponible en: https://learn.microsoft.com/es-es/azure/spring-apps/basic-standard/how-to-launch-from-source

[35] «initializing Docker API Proxy: open \\.\pipe\docker_engine: Access is denied. · Issue #13663 · docker/for-win», GitHub. [En línea]. Disponible en: https://github.com/docker/for-win/issues/13663

[36] «Install Compose standalone», Docker Documentation. [En línea]. Disponible en: https://docs.docker.com/compose/install/standalone/

[37] Sentrio, «Introducción a Jenkins: ¿qué es, para qué sirve y cómo funciona?», Sentrio. [En línea]. Disponible en: https://sentrio.io/blog/que-es-jenkins/

[38] J. D. Muñoz, «Introducción a las redes en docker. Enlazando contenedores docker», PLEDIN 3.0. [En línea]. Disponible en: https://www.josedomingo.org/pledin/2020/02/redes-en-docker/

[39] «Jenkins agents build jobs temporary files permissions - Using Jenkins / Ask a question», Jenkins. [En línea]. Disponible en: https://community.jenkins.io/t/jenkins-agents-build-jobs-temporary-files-permissions/11066

[40] X. Mallón, «Jenkins: Conoce qué es y para qué sirve». [En línea]. Disponible en: https://keepcoding.io/blog/que-es-jenkins-y-para-que-sirve/

[41] «Jest testing with Selenium WebDriver», TestingBot. [En línea]. Disponible en: https://testingbot.com/support/getting-started/jest.html

[42] J. Aristigueta, «Jesus Aristigueta | Instalación de Docker y Docker Compose en Debian». [En línea]. Disponible en: https://jesusaristigueta.com/blog/docker-compose-debian/

[43] «Managing Plugins», Managing Plugins. [En línea]. Disponible en: https://www.jenkins.io/doc/book/managing/plugins/

[44] «MockMvc – Introducción a Spring MVC testing framework: Probando endpoints - Marc Nuri», www.marcnuri.com. [En línea]. Disponible en: https://blog.marcnuri.com/mockmvc-introduccion-a-spring-mvc-testing

[45] «NET SDK Support», .NET SDK Support. [En línea]. Disponible en: https://plugins.jenkins.io/dotnet-sdk

[46] «Network containers», Docker Documentation. [En línea]. Disponible en: https://docs.docker.com/engine/tutorials/networkingcontainers/

[47] «Networking in Compose», Docker Documentation. [En línea]. Disponible en: https://docs.docker.com/compose/networking/

[48] philipp, «Node v13 / Jest / ES6 — native support for modules without babel or esm», Stack Overflow. [En línea]. Disponible en: https://stackoverflow.com/q/60372790

[49] T. B. Achour, «permission denied when executing the jenkins sh pipeline step», Stack Overflow. [En línea]. Disponible en: https://stackoverflow.com/q/43372035

[50] tdykstra, «Pruebas de integración en ASP.NET Core». [En línea]. Disponible en: https://learn.microsoft.com/es-es/aspnet/core/test/integration-tests?view=aspnetcore-9.0

[51] Sentrio, «¿Qué es Azure DevOps?», Sentrio. [En línea]. Disponible en: https://sentrio.io/blog/que-es-azure-devops/

[52] «Qué es Docker Compose y Cómo Usarlo | Tutorial Completo». [En línea]. Disponible en: https://imaginaformacion.com/tutoriales/que-es-docker-compose

[53] X. Mallón, «¿Qué es GitLab CI/CD? | KeepCoding Bootcamps». [En línea]. Disponible en: https://keepcoding.io/blog/que-es-gitlab-ci-cd/

[54] KarlErickson, «Quickstart - Build and deploy apps to Azure Spring Apps». [En línea]. Disponible en: https://learn.microsoft.com/en-us/azure/spring-apps/basic-standard/quickstart-deploy-apps

[55] KarlErickson, «Quickstart - Deploy your first application to Azure Spring Apps». [En línea]. Disponible en: https://learn.microsoft.com/en-us/azure/spring-apps/enterprise/quickstart

[56] A. B. Collier, «Selenium Crawler #2: Docker Bridge Network». [En línea]. Disponible en: https://datawookie.dev/blog/2021/04/selenium-crawler-2-docker-bridge-network/

[57] A. Popp, «Selenium Java Maven Cross-Browser tests in Docker using Selenoid», Testautomation Popp. [En línea]. Disponible en: https://testautomation-popp.de/selenium-cross-browser-tests-in-docker-using-selenoid/

[58] «Selenium with C# : How to start running Automated Tests», BrowserStack. [En línea]. Disponible en: https://browserstack.wpengine.com/guide/selenium-with-c-sharp-for-automated-test/

[59] «seleniumhq.github.io/examples/java/src/test/java/dev/selenium/drivers/RemoteWebDriverTest.java at trunk · SeleniumHQ/seleniumhq.github.io», GitHub. [En línea]. Disponible en: https://github.com/SeleniumHQ/seleniumhq.github.io/blob/trunk/examples/java/src/test/java/dev/selenium/drivers/RemoteWebDriverTest.java

[60] «Selenoid». [En línea]. Disponible en: https://aerokube.com/selenoid/1.6.0/#_selenoid_with_docker_compose

[61] S. Thomas, «Selenoid Alternatives: Explore the Top Cloud Automation Grids», Medium. [En línea]. Disponible en: https://medium.com/@sarah.thoma.456/selenoid-alternatives-explore-the-top-cloud-automation-grids-978b0878d0df

[62] «Selenoid UI». [En línea]. Disponible en: http://aerokube.com/selenoid-ui/latest/#_quick_start_guide

[63] R. Gainanov, «Selenoid. Selenium in Docker», GAINANOV.PRO. [En línea]. Disponible en: https://gainanov.pro/eng-blog/devops/selenium-in-docker-with-selenoid/

[64] «[SERVICE_STARTUP_FAILED] [wait: http://172.18.0.4:4444/wd/hub does not respond in 30s] · Issue #1021 · aerokube/selenoid», GitHub. [En línea]. Disponible en: https://github.com/aerokube/selenoid/issues/1021

[65] «SonarScanner CLI | SonarQube Server Documentation». [En línea]. Disponible en: https://docs.sonarsource.com/sonarqube-server/latest/analyzing-source-code/scanners/sonarscanner/#windows

[66] «SonarScanner for Jenkins». [En línea]. Disponible en: https://docs.sonarsource.com/sonarqube/8.9/analyzing-source-code/scanners/sonarscanner-for-jenkins/

[67] A. Angelov, «Test Automation Reporting with Allure in .NET Projects», Automate The Planet. [En línea]. Disponible en: https://www.automatetheplanet.com/test-automation-reporting-allure/

[68] Rolique, «Test Automation with Selenide: Run Selenoid with Docker Compose file», Medium. [En línea]. Disponible en: https://rolique.medium.com/test-automation-with-selenide-run-selenoid-with-docker-compose-file-2045621fbd33

[69] «Testeando JavaScript con Mocha y Chai». [En línea]. Disponible en: https://www.paradigmadigital.com/dev/testeando-javascript-mocha-chai/

[70] «Testing Node.js/Express app + MongoDB with jest and supertest», DEV Community. [En línea]. Disponible en: https://dev.to/shyamajp/testing-nodejsexpress-app-mongodb-with-jest-and-supertest-56ce

[71] «Testing web apps in JavaScript using Selenium WebDriver and Jest | Applitools Tutorials». [En línea]. Disponible en: https://applitools.com/tutorials/sdks/selenium-js/quickstart

[72] «Ubuntu», Docker Documentation. [En línea]. Disponible en: https://docs.docker.com/engine/install/ubuntu/

[73] chris, «Ventajas y desventajas de Microsoft Azure - Servicio en la nube para empresas». [En línea]. Disponible en: https://blog.hensongroup.com/es/pros-and-cons-of-microsoft-azure-cloud-service-for-businesses/

[74] «vm importModuleDynamically option in Node 20.10 requires --experimental-vm-modules flag and 20.9 does not · Issue #51153 · nodejs/node», GitHub. [En línea]. Disponible en: https://github.com/nodejs/node/issues/51153

[75] «WebdriverIO - WebDriver bindings for Node.js». [En línea]. Disponible en: http://webdriver.io

[76] R. Jain, «WebdriverIO Tutorial: Browser Commands for Selenium Testing», Medium. [En línea]. Disponible en: https://medium.com/@jainrahul098/webdriverio-tutorial-browser-commands-for-selenium-testing-c5ca04ce07f6

[77] «Windows», Docker Documentation. [En línea]. Disponible en: https://docs.docker.com/desktop/setup/install/windows-install/

[78] «Writing end-to-end tests in Node.js using Selenium», Reflect. [En línea]. Disponible en: https://reflect.run/articles/writing-end-to-end-tests-in-nodejs-using-selenium/

[79] «xUnit.net». [En línea]. Disponible en: https://allurereport.org/docs/xunit/

[80] riddle_me_this, «Answer to “Cannot access templates running Spring Boot with JAR”», Stack Overflow. [En línea]. Disponible en: https://stackoverflow.com/a/48969220

[81] Alexandre987, «Answer to “Cannot connect to the Docker daemon at unix:///var/run/docker.sock. Is the docker daemon running? inside a Dockerfile”», Stack Overflow. [En línea]. Disponible en: https://stackoverflow.com/a/78234034

[82] M. Villanueva, «Answer to “Cannot execute SonarScanner for .NET project on Jenkins because «dotnet-sonarscanner: not found»”», Stack Overflow. [En línea]. Disponible en: https://stackoverflow.com/a/69079281

[83] akauppi, «Answer to “Node v13 / Jest / ES6 — native support for modules without babel or esm”», Stack Overflow. [En línea]. Disponible en: https://stackoverflow.com/a/63281463