Implementación de Pruebas de Integración Automatizadas en la Capa Backend de los Aplicativos Web del Proyecto RSI

Adriana Villamizar Vera, Miguel Felipe Espinel Botero

Trabajo de Grado para optar al título de Ingenieros de Sistemas

Director

Fernando Antonio Rojas Morales

Magíster en Ciencias de la Computación

Codirector

Danny Felipe Vergel Paba

Especialista en Gerencia de Proyectos

Universidad Industrial de Santander

Facultad de Ingenierías Fisicomecánicas

Escuela de Ingeniería de Sistemas e Informática

Ingeniería de Sistemas

Bucaramanga

2024

Dedicatoria

A Dios quien nos dio la habilidad y fuerzas para culminar esta meta.

A nuestras familias, por su apoyo incondicional.

Agradecimientos

A nuestros familiares y amigos cercanos que nos apoyaron y acompañaron durante toda la etapa de nuestra formación académica. A nuestro tutor, el Ingeniero Danny Felipe Vergel por su paciencia, disposición, apoyo y compromiso que nos brindó en el desarrollo del trabajo de grado. Al profesor Fernando Rojas, quien en su rol de director nos orientó y apoyó para lograr el mejor resultado posible en este proyecto de grado. A todo el equipo de líderes y compañeros de RSI que pusieron a disposición tiempo y conocimiento para brindarnos apoyo cuando lo necesitamos. A cada docente y profesional que a lo largo de nuestro proceso de formación como ingenieros de sistemas y profesionales nos enriquecieron con sus enseñanzas y su ejemplo.

Tabla de Contenido

Pág.

Introducción 12

1. Planteamiento y Justificación 14

2. Objetivos 16

3. Marco de Referencia 17

3.1. Pruebas de software 17

3.1.1. Tipos de pruebas de software 17

3.2. Pruebas unitarias 18

3.3. Pruebas de integración 19

3.4. Automatización de pruebas de software 19

3.4.1. Importancia de la automatización de pruebas en un esquema de CD 20

3.5. Metodología TDD 20

3.6. Metodología BDD 20

3.7. Arquitectura de microservicios 21

3.8. Arquitectura del proyecto RSI 22

3.8.1 Capa de base de datos 23

3.8.2 Capa backend 23

3.8.2.1 Arquitectura hexagonal 25

3.8.2.2 Implementación de pruebas automatizadas en la capa backend 26

3.8.2.3 Comunicación entre microservicios 27

3.8.3 Capa frontend.....27

4. Marco Metodológico.....28

4.1 Marco de trabajo Scrum.....28

4.2 Control de versiones.....29

4.3 Integración continua.....29

4.4 Flujo de trabajo.....30

5. Implementación.....31

5.1. Investigar sobre herramientas para implementar pruebas de integración automatizadas en la capa backend del proyecto RSI.....32

5.1.1 Objetivo.....32

5.1.2 Criterios de aceptación.....33

5.1.3 Ejecución.....33

5.1.3.1 Pruebas de integración.....34

5.1.3.2 Pruebas de integración con SpringBootTest.....34

5.1.3.2.1 Pruebas de integración con SpringBootTest y MockMVC.....42

5.1.3.3 Pruebas de integración con TestContainers.....45

5.1.3.4 Evaluación de herramientas.....52

5.1.3.5 Pruebas de integración automatizadas aplicadas con CI.....56

5.1.3.6 Reporte de cobertura por pruebas de integración.....59

5.1.3.7 Esquema general de planteamiento para implementación de pruebas de integración automatizadas para la capa backend en el proyecto RSI.....62

5.2. Investigar sobre herramientas para implementar pruebas de integración automatizadas en casos con comunicación entre microservicios. 63

5.2.1 Objetivo...63

5.2.2 Criterios de aceptación...64

5.2.3 Ejecución..64

5.2.3.1 Pruebas por contrato empleando mock de la respuesta del productor 69

5.2.3.2 Pruebas por contrato empleando stubs para la respuesta del productor............................72

5.3. Investigar sobre metodología TDD y posibles aportes al proyecto. 79

5.3.1 Objetivo...79

5.3.2 Criterios de aceptación...79

5.3.3 Ejecución..80

5.4. Indagar sobre estrategias de aplicación de pruebas de software e identificación de necesidad de pruebas de integración...81

5.4.1 Objetivo...81

5.4.2 Criterios de aceptación...82

5.4.3 Ejecución..82

5.4.3.1 Estrategias de aplicación de pruebas de integración y su adopción en el proyecto RSI....83

5.4.3.2 Evaluación de desarrollos para la aplicación de pruebas de integración 87

5.5. Implementar pruebas de integración en principales funcionalidades del módulo Core que lo requieran. 88

5.5.1 Objetivo...88

5.5.2 Criterios de aceptación...88

5.5.3 Ejecución..89

5.6. Documentar el proceso de implementación y ejecución de pruebas de integración automatizadas dentro de RSI. 92

5.6.1 Objetivo..92

5.6.2 Criterios de aceptación..93

5.6.3 Ejecución..93

5.7. Brindar capacitaciones a cada equipo del proyecto RSI sobre la implementación de pruebas de integración automatizadas en la capa backend..94

5.7.1 Objetivo..94

5.7.2 Criterios de aceptación..94

5.7.3 Ejecución..95

6. Resultados..95

7. Conclusiones..96

8. Trabajo Futuro..97

Referencias Bibliográficas..99

Lista de Tablas

Pág.

Tabla 1. Comparación entre herramientas para implementación de pruebas de integración.........53

Tabla 2. Consideraciones de evaluación de herramientas para implementación de pruebas de integración..54

Tabla 3. Ventajas y desventajas de las distintas estrategias de implementación de pruebas de integración..84

Tabla 4. Diagrama de cobertura por test de integración con SpringBootTest...............................90

Tabla 5. Esquema de pruebas de integración con MockMvc 90

Tabla 6. Esquema del flujo de pruebas de integración con TestContainers. 91

Lista de Figuras

Pág.

Figura 1. Arquitectura de software del proyecto RSI 22
Figura 2. Estructura de la arquitectura hexagonal 25
Figura 3. Flujo de trabajo de RSI para desarrolladores 30
Figura 4. Diagrama de cobertura por test de integración con SpringBootTest....................36
Figura 5. Esquema de pruebas de integración con MockMvc 43
Figura 6. Esquema del flujo de pruebas de integración con TestContainers. 47
Figura 7. Ejecución automática y resultados de pruebas de integración en despliegue de ambiente predev. 58
Figura 8. Visualización de porcentaje de cobertura por pruebas automatizadas. 60
Figura 9. Informe general sobre coverage por test. 60
Figura 10. Reporte de porcentaje de coverage por cada paquete del proyecto backend 61
Figura 11. Visualización por colores de líneas cubiertas (verde) y no cubiertas (rojo) en una clase 61
Figura 12. Diagrama global generalizado de implementación de pruebas de integración en microservicios backend de RSI....................63
Figura 13. Pirámide de pruebas de software de Cohn 66
Figura 14. Flujo de prueba por contrato haciendo mocks de la respuesta del proveedor 69
Figura 15. Esquema del flujo de prueba por contrato usando stubs de verificación 72

Resumen

Título: Implementación de pruebas de integración automatizadas en la capa backend de los aplicativos web del proyecto RSI. *

Autor: Adriana Villamizar Vera, Miguel Felipe Espinel Botero **

Palabras clave: Pruebas unitarias, pruebas de integración, automatización, software.

Descripción: Dentro del proceso de desarrollo de un software, una de las diferentes etapas cruciales para garantizar la calidad es la realización de pruebas de las funcionalidades implementadas. Dependiendo del alcance que se le quieran dar a dichas pruebas se aplica el tipo respectivo; por ejemplo, se pueden probar unidades de código aisladas mediante pruebas unitarias, o la interacción de varios componentes del sistema haciendo uso de pruebas de integración.

No obstante, aprovechar herramientas que faciliten el proceso de ejecución de pruebas resulta ser de vital importancia para optimizar recursos temporales y humanos, a esto se le conoce como automatización de pruebas de software.

El enfoque principal de este proyecto es diseñar e implementar una solución para la necesidad de usar pruebas automatizadas en la capa backend del proyecto RSI, el cual es un proyecto desarrollado con el fin de renovar los sistemas de información de la Universidad Industrial de Santander, poniendo en uso nuevas tecnologías.

* Trabajo de Grado

** Facultad de Ingenierías Físico-mecánicas, Escuela de Ingeniería de Sistemas e Informática, Ingeniería de Sistemas. Director: Fernando Antonio Rojas Morales. Magíster en Ciencias de la Computación. Codirector: Danny Felipe Vergel. Especialista en Gerencia de Proyectos.

Abstract

Title: Implementation of automated integration tests in the backend layer of RSI project web applications

Author: Adriana Villamizar Vera, Miguel Felipe Espinel Botero *

Keywords: Unit test, integration test, automation, software. **

Description: Within a software development process, one of the crucial steps to ensure quality is the testing of the implemented functionalities. Depending on the scope of such tests, the respective type is applied; for example, isolated code units can be tested by means of unit tests, or the interaction of several system components using integration tests.

However, taking advantage of tools that facilitate the test execution process is of vital importance to optimize time and human resources, this is known as software test automation.

The main focus of this project is to design and implement a solution regarding the need of using automated tests in the backend layer of the RSI project, which is developed in order to renew the information systems of the Universidad Industrial de Santander, using new technologies.

* Degree Work

** Physicomechanical Engineering Faculty, School of Systems Engineering and Informatics, Systems Engineering. Director: Fernando Antonio Rojas Morales. Computational Science Master. Codirector: Danny Felipe Vergel. Project Management Specialist.

Introducción

Los procesos de negocio de una organización pueden ser llevados a un software con el fin de facilitar y agilizar su ejecución; a su vez, el desarrollo de dicho software debe hacerse de tal forma que se garantice la calidad de este, esto es, elaborar un producto flexible, eficiente, confiable, mantenible, portable, seguro e íntegro. Para lograr un producto de software de calidad es necesario someterlo a pruebas y así verificar que funciona de la manera que se espera; los beneficios que se obtienen son la prevención de errores, la reducción de costos de desarrollo y la mejora del rendimiento. Existen diferentes tipos de pruebas, entre las cuales se encuentran: pruebas unitarias, pruebas de integración, de aceptación, de rendimiento, de regresión, pruebas e2e (end-to-end) y de usabilidad, las cuales se aplican dependiendo el ámbito y alcance que se desea abarcar del software.

Dado que el objetivo de una arquitectura de software es ser eficiente, es decir, realizar la tarea para la cual fue diseñada a la vez que se minimicen en la mayor medida posible los recursos humanos y de tiempo requeridos para construir y mantener el sistema, una estrategia muy común para mejorar la capacidad de probar un sistema es automatizar la realización de los tipos de pruebas que pueden llevarse a cabo bajo esta técnica, estas son las unitarias, de integración, de extremo a extremo, de regresión y de rendimiento. Aunque es cierto que implementar pruebas automatizadas aumenta el tiempo de desarrollo, también lo es el hecho de que se reduce el tiempo que tomaría volver a probar el software tras realizar cambios, así como el de solucionar errores que pueden identificarse plenamente con las pruebas, en otras palabras, el producto de software se hace lo suficientemente flexible como para minimizar el coste del cambio, garantizando su funcionalidad, y facilitando su mantenimiento y ampliación.

Existen prácticas llamadas "primero las pruebas" en las que se plantea que la automatización de las pruebas debe abordarse desde los primeros pasos del concepto del sistema, es decir, que las pruebas deben escribirse antes que el código, así se garantiza que cualquier parte que se añada al sistema irá acompañada de pruebas automatizadas y el sistema mantiene su condición de comprobable; una de estas prácticas es TDD (Test Driven Development)[1].

El proyecto RSI, un desarrollo enfocado en renovar los Sistemas de Información de la Universidad Industrial de Santander actualmente cuenta con la ejecución de pruebas manuales por parte de desarrolladores, equipo de QA e incluso pruebas de aceptación con clientes. Adicionalmente, ya se tienen implementadas pruebas unitarias automatizadas a nivel de servicios backend; sin embargo, aún falta automatizar el proceso de pruebas de mayor cobertura como las de integración.

El propósito de este proyecto es el diseño e implementación de pruebas automatizadas de integración en los diferentes microservicios de la capa backend del proyecto RSI, abarcando tanto servicios antiguos que ya fueron construidos sin pruebas como nuevos. Con el fin de lograr este objetivo, se ha realizado una labor de investigación sobre tecnologías disponibles para implementación de pruebas automatizadas que se ajusten a las características de la arquitectura de RSI, y la posterior proposición y aplicación de una forma óptima de llevar a cabo dicha implementación.

[1] Santacroce, F. (2022, 9 junio). Test-Driven Development: A Time-Tested Recipe for Quality Software. Semaphore. https://semaphoreci.com/blog/test-driven-development

1. Planteamiento y Justificación

Los sistemas de software utilizados por las diferentes áreas de la Universidad Industrial de Santander, debido a que su creación se realizó hace varios años con las tecnologías que se contaban en su momento, actualmente carecen de actualización, soporte y documentación, por ello, surgió el proyecto RSI (Renovación de los Sistemas de Información) que busca desarrollar nuevamente esos antiguos sistemas actualizados con tecnologías más modernas que cuentan con suficiente documentación y facilitarán el soporte y actualización de los sistemas en el futuro. RSI ha venido desarrollando los distintos módulos que componen los sistemas de información, y como ocurre en todo software, surgen cambios y se agregan nuevos componentes a medida que avanza el proceso de desarrollo; es allí donde nace la necesidad de garantizar la flexibilidad, mantenibilidad, y funcionalidad del software optimizando recursos de tiempo y esfuerzo humano.

El poder garantizar la funcionalidad de un software se hace mediante las pruebas que se aplican sobre el mismo, pero a medida que este crece y se hacen cambios en algunas partes, un proceso de pruebas manual resulta tedioso y costoso en cuanto al tiempo, es por eso que se requiere un buen diseño e implementación de pruebas automatizadas sobre los servicios que se desarrollan y así brindarle flexibilidad al software, esto a su vez reduce costos en el proyecto, ya que no se ocupará el tiempo del equipo de calidad realizando esas pruebas y podrá enfocarse en temas más relevantes, así también, se libera tiempo por parte de los desarrolladores en la aplicación de pruebas, identificación y corrección de errores. No obstante, no basta con simplemente implementar pruebas automatizadas, es necesario comparar las diferentes alternativas de técnicas existentes para su aplicación y así poder elegir las que resulten óptimas en relación con los requisitos de los sistemas desarrollados.

Como parte del equipo de desarrollo, la labor a llevar a cabo estará centrada en estudiar y evaluar las tecnologías mediante las cuales se pueden implementar pruebas automatizadas de integración a nivel de backend para identificar la que más se adapte a las necesidades del proyecto RSI y llevar a cabo la implementación de dichas pruebas en un módulo central del sistema.

Pregunta de investigación: ¿Bajo qué tecnología y cómo implementar pruebas de integración óptimas a los microservicios de la capa backend que se desarrollan en el proyecto RSI para garantizar la calidad del software a la vez que se optimiza el proceso de desarrollo?

2. Objetivos

2.1 Objetivo general

Implementar un proceso de pruebas de integración automatizadas en la capa backend de las aplicaciones WEB del proyecto RSI, que permitan medir la calidad y optimicen el proceso de desarrollo del sistema de software.

2.2 Objetivos específicos

Realizar una indagación sobre la arquitectura y tecnologías usadas actualmente para el desarrollo backend en el proyecto RSI y aquellas que están a la vanguardia en el mercado para el manejo de pruebas de integración automatizadas para aplicar la que más se acople a los requerimientos del proyecto.

Hacer revisión de la metodología TDD para tomar de ella aspectos que puedan facilitar el proceso de desarrollo de software del proyecto RSI enfocado en pruebas de integración.

Realizar configuración para garantizar que las pruebas automatizadas no alterarán las bases de datos que se conectan en cada microservicio probado.

Identificar requerimientos de desarrollos nuevos y existentes en la capa backend del módulo Core, el cual es un módulo central en la funcionalidad general de las aplicaciones web del proyecto RSI, para crear y codificar casos de prueba siguiendo los criterios de aceptación del desarrollo. Verificar los fallos de las pruebas, implementar la lógica de negocio, ejecutar pruebas automatizadas y refactorizar el código escrito de nuevas funcionalidades.

3. Marco de Referencia

Con el fin de comprender el contexto en el que se encuentra este proyecto, a continuación, se presentan los principales temas relacionados y la arquitectura de RSI sobre la cual se lleva a cabo el desarrollo de lo contemplado como objetivos del presente trabajo académico.

3.1. Pruebas de software

Una prueba de software consiste en evaluar y verificar que un artefacto de software hace aquello que se supone debería hacer y lo hace de forma correcta. Realizar un correcto proceso de ejecución de pruebas conlleva beneficios como la prevención de errores, reducción del costo de desarrollo y mejora de rendimiento.

Aunque las pruebas cuestan dinero, también garantizan ahorrarlo en desarrollo y soporte si se cuenta con buenas técnicas de pruebas y procesos de control de calidad. Cuando durante el desarrollo se dedica un amplio espacio para las pruebas, mejora la confiabilidad que se tiene del software y se entregan productos de calidad con pocos errores. A nivel general, las pruebas de software suelen seguir un proceso común: la definición de un entorno de prueba, el desarrollo de casos de prueba, el análisis de los resultados de las pruebas y el envío de informes de errores. Ejecutar todos estos pasos puede llevar mucho tiempo, más aún si son pruebas manuales; sin embargo, para sistemas más grandes, con frecuencia se utilizan herramientas que automatizan este tipo de tareas. Las pruebas automatizadas ayudan a implementar diferentes escenarios y obtener retroalimentación sobre lo que funciona y lo que no de forma óptima.

3.1.1 Tipos de pruebas de software

Dependiendo del objetivo que se quiere lograr con la ejecución de pruebas, existen distintos tipos:

- **Pruebas de aceptación:** validan que el software responde a los requisitos del usuario y funciona correctamente en su entorno.
- **Pruebas de integración:** evalúan la comunicación e interoperabilidad entre los diferentes componentes o módulos que componen un software.
- **Pruebas unitarias:** verifican que cada unidad funcione correctamente de manera aislada antes de integrarse con otras partes del sistema.
- **Pruebas funcionales:** validan que un software cumpla con las especificaciones de funcionamiento establecidas, un ejemplo de este tipo es la prueba de caja negra.
- **Pruebas de rendimiento:** evalúan la calidad del software en términos de escalabilidad, velocidad y estabilidad bajo diferentes escenarios y condiciones de carga.
- **Pruebas de regresión:** se centran en asegurar la estabilidad y la integridad del sistema después de realizar modificaciones.
- **Pruebas de usabilidad:** evalúan la facilidad de uso, la experiencia y satisfacción general del usuario en su interacción con el software.

Dos de los tipos básicos de pruebas, las cuales se encuentran muy relacionadas con el desarrollo de este proyecto, son las unitarias y las de integración, seguidamente se ahondará en ellas.

3.2. Pruebas unitarias

Las pruebas unitarias o unit test[2] consisten en aislar una unidad de código, por ejemplo, una clase o un método, para comprobar que funciona de forma correcta. Generalmente, este tipo de pruebas se realizan durante la fase de desarrollo de las aplicaciones de software y son llevadas a cabo por desarrolladores; no obstante, también las pueden realizar miembros del equipo de QA.

[2] Yeeply. (2022). ¿Qué son las pruebas unitarias y cómo llevar una a cabo? Yeeply. https://www.yeeply.com/blog/que-son-pruebas-unitarias/

Las pruebas unitarias permiten detectar errores que, de no ser percibidos en ese punto, no podrían descubrirse hasta fases de prueba mucho más avanzadas como las de integración o regresión, y por ello es por lo que la realización de las pruebas unitarias supone un ahorro de tiempo y recursos durante el proceso global de desarrollo. También cabe resaltar que este tipo de pruebas son la base dentro del conglomerado de pruebas de software que existen.

3.3. Pruebas de integración

Las pruebas de integración[3] se encargan de validar los puntos de interacción entre funciones o componentes individuales, es decir, buscan asegurar que los componentes funcionan bien al juntarlos. Con este tipo de pruebas, al igual que con las unitarias, se logran disminuir gastos de recursos y tiempo gracias a que los errores son detectados oportunamente y se les puede dar solución sin que causen un gran impacto como el que provocarían si se detectaran en una etapa más avanzada.

Con el fin de facilitar la realización de pruebas y optimizar recursos en desarrollo, existen herramientas que permiten automatizar este proceso de control de calidad.

3.4. Automatización de pruebas de software

Automatizar las pruebas consiste en aplicar herramientas de software que reemplacen los procesos manuales de revisión y validación de un producto de software. Para la metodología de entrega continua, CD[4], que consiste en publicar periódicamente nuevas versiones de software, la realización de pruebas de forma automática es de gran ayuda, ya que cada vez que surge algo nuevo, el proceso de comprobar que no se ha afectado la funcionalidad resulta corto y preciso.

---

[3] IBM Documentation. (s. f.). https://www.ibm.com/docs/es/rtw/9.0.1?topic=phases-integration-testings/

[4] La integración y la distribución continuas (CI/CD). (s. f.). https://www.redhat.com/es/topics/devops/what-is-ci-cd

3.4.1 Importancia de la automatización de pruebas en un esquema de CD

Automatizar las pruebas resulta fundamental para alcanzar el objetivo de publicar versiones de código nuevas de forma eficiente para los clientes. La CD forma parte de un proceso de implementación mayor, esta es la integración continua (CI) y depende de ella. La CI debe ejecutar pruebas automatizadas ante cualquier actualización del software y verificar que dichos cambios no afectan a la integridad de las funciones previas ni generan errores. Esta relación entre las pruebas automatizadas, la CI y la CD aporta numerosas ventajas a los equipos de software que trabajan para hacer entregas constantes.

En busca de hacer del proceso de aplicación de pruebas lo óptimo posible, existen metodologías que plantean unas pautas con las cuales se lograría un mejor desempeño en un proyecto de desarrollo de software, dos de ellas son TDD y BDD.

3.5. Metodología TDD

El desarrollo orientado a pruebas o Test-Driven Development (TDD), se basa en hacer que los desarrolladores se enfoquen en los requerimientos del producto primero, escriban los casos de prueba que implica el cumplimiento del requerimiento y luego se pase a la codificación necesaria para satisfacer los casos de prueba planteados. Con esta metodología se busca lograr un código más claro, sencillo y libre de errores y además tiene como beneficios el aumento de la calidad del producto de software y de la productividad de los equipos de desarrollo.

3.6. Metodología BDD

El desarrollo orientado por comportamiento, BDD (Behavior Driven Development) es una metodología que promueve el desarrollo de software apegado a un entendimiento compartido entre los miembros técnicos y no técnicos del equipo, es decir, entre desarrolladores, QAs y clientes. Este enfoque abarca distintos niveles de pruebas: pruebas unitarias, de integración y de aceptación,

su principal objetivo es verificar el comportamiento y la interacción de los distintos componentes del sistema.

Dependiendo de la arquitectura en la que esté basado un proyecto, la implementación de las anteriores metodologías puede resultar más o menos compleja y conveniente.

3.7. Arquitectura de microservicios

Una arquitectura basada en microservicios consiste en dividir una aplicación en un conjunto de servicios que se implementan de forma independiente y se comunican entre sí mediante interfaces de programación de aplicaciones, APIs. Este enfoque hace que un sistema sea moderno, escalable y altamente distribuido, así como también facilita la entrega frecuente de aplicaciones grandes y de alta complejidad, y simplifica la identificación y solución de fallos y errores en cada servicio por separado, lo que conlleva a una reducción de los costos asociados al mantenimiento. Contrario a lo que sería una aplicación monolítica, la arquitectura de microservicios proporciona a los equipos de desarrollo la capacidad de desplegar nuevas funcionalidades y realizar modificaciones de manera ágil, sin la necesidad de reescribir extensas porciones del código previamente desarrollado. Ya mencionados los beneficios de la arquitectura de microservicios, cabe indicar también algunos inconvenientes que se pueden presentar dentro de esta estrategia de trabajo; uno de ellos es que se aumenta el consumo de recursos de CPU y memoria dado que es necesario asignarlos de forma específica para cada microservicio diferente en el que se encuentra distribuida la aplicación, por otro lado, se requiere una mayor inversión de tiempo gastado en análisis y diseño para plantear de forma correcta el reparto de la aplicación y la complejidad para realizar pruebas globales, que aumenta al tratarse de funcionalidades separadas que requieren interactuar entre sí para establecer un funcionamiento integral.

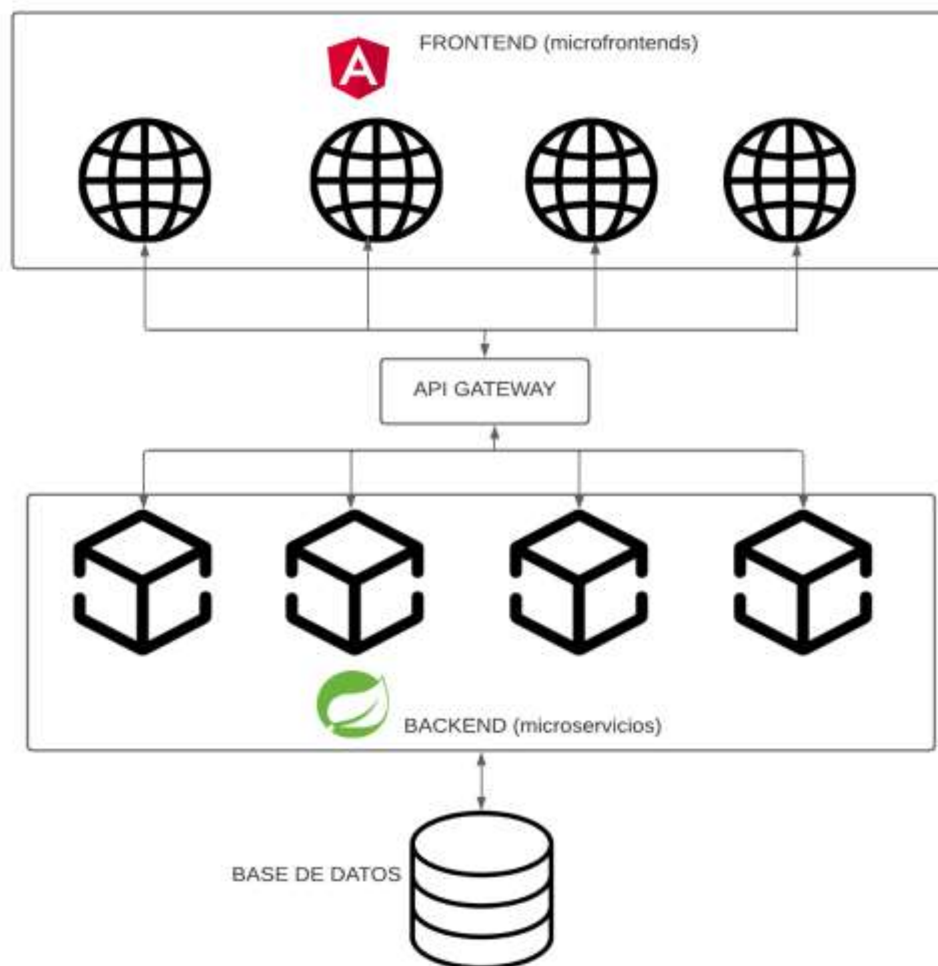
El proyecto RSI, dentro del cual se desenvuelve este trabajo de investigación, hace uso de la arquitectura de microservicios, a continuación, se presenta una vista generalizada de la estructura de dicho proyecto.

3.8. Arquitectura del proyecto RSI

En la Figura 1 se observa a nivel macro parte de la arquitectura que se emplea actualmente en el proyecto RSI, cada uno de sus componentes técnicos usados para hacer posible el funcionamiento del proyecto se detalla a continuación.

Figura 1

Arquitectura de software del proyecto RSI



3.8.1 Capa de base de datos

La persistencia de datos para su posterior gestión en RSI se hace a través de base de datos SQL, este sistema de gestión de base de datos es uno de los más utilizados a nivel organizacional debido a que soporta y facilita el manejo de grandes volúmenes de datos; permite la ejecución eficiente de consultas complejas, lo cual lo hace un sistema de alto rendimiento y escalable, es compatible con estándares como JDBC, admite diversos tipos de datos y gracias a su diseño de tipo objeto-relacional, hace posible almacenar y manipular tanto datos estructurados como no estructurados.

3.8.2 Capa backend

El backend del proyecto RSI se desarrolla bajo arquitectura de microservicios, esto permite que cada microservicio pueda ser desarrollado en un lenguaje, framework y con tecnologías diferentes dependiendo de las características de cada uno, además aporta independencia entre ellos; sin embargo, dentro del desarrollo, la tecnología predominante en la cual se implementan los microservicios es Spring Boot, la cual hace uso del lenguaje Java.

Cada uno de los microservicios se encuentra alojado dentro de un servidor de aplicaciones que permite gestionar su despliegue, así como los recursos utilizados por las aplicaciones.

Para el desarrollo de las aplicaciones se hace uso del framework Spring[5], una popular tecnología de código abierto para crear aplicaciones independientes que se ejecutan en la máquina virtual de Java (JVM). Algunas de las características por las cuales esta herramienta cuenta con una alta preferencia por la comunidad del software son la excelente documentación y soporte que tiene, la mejora constante y adaptación al ritmo del ecosistema que le compete, la facilidad que ofrece a los desarrolladores al solventar la implementación de funcionalidades típicas que una

[5] Johnson, R., Hoeller, J., Donald, K., Sampaleanu, C., Harrop, R., Risberg, T., ... & Webb, P. (2004). The spring framework-reference documentation. interface, 21, 27.

aplicación necesita realizar, desde las de bajo nivel hasta otras más complejas y tediosas; gracias a este soporte integrado, el desarrollador puede enfocarse en la lógica de negocio mayormente. Este framework también ofrece la función de inyección de dependencias, esto hace posible crear aplicaciones modulares de componentes poco acoplados ideales para el enfoque de microservicios y otros tipos.

Aunque Spring Framework es muy completo y ofrece numerosas funcionalidades, la configuración, instalación e implementación de aplicaciones Spring requieren un tiempo considerable y un conocimiento suficiente. Spring Boot facilita este proceso, volviéndolo más fácil y rápido al incorporar tres características clave: configuración automática, enfoque destinado para agregar y configurar dependencias de inicialización, y aplicaciones independientes.

En otras palabras, Spring Boot es una extensión de Spring y los beneficios adicionales que aporta son:

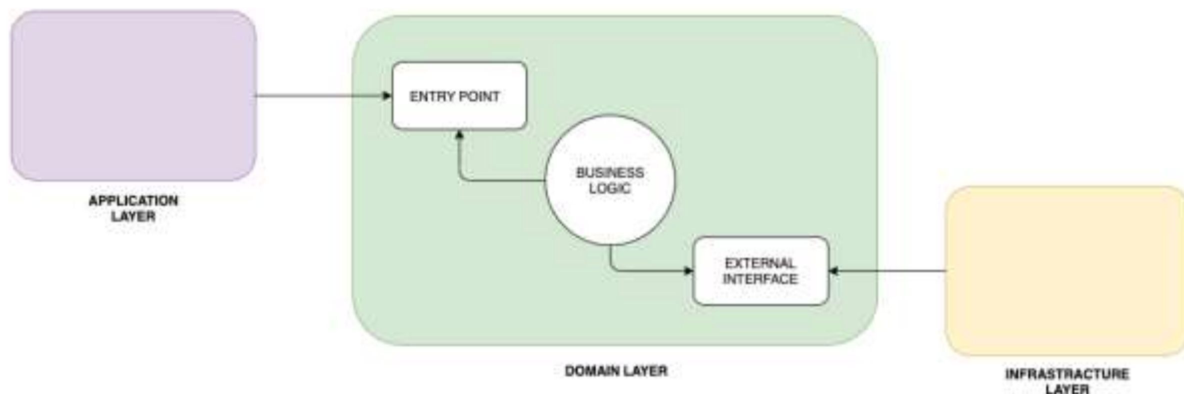
- Inicialización de aplicaciones con dependencias preestablecidas que no tienen que configurarse manualmente.
- Permite agregar dependencias, conocidas como *"starters"* las cuales se seleccionan al definir las necesidades del proyecto durante el proceso de inicialización, por ejemplo existen para pruebas, seguridad, aplicación web, entre otros.
- Permite la creación de aplicaciones independientes que son ejecutadas por sí mismas, es decir, sin la necesidad de un servidor web externo, basta con integrar un servidor embebido como *Tomcat* en la aplicación para usar durante el proceso de ejecución.

Spring Boot posibilita el uso de la arquitectura hexagonal en la construcción de microservicios, la cual también es empleada dentro de RSI como patrón de desarrollo.

3.8.2.1 Arquitectura hexagonal. La Arquitectura hexagonal[6] es un modelo de diseño de software donde la lógica de dominio se especifica en un núcleo de negocio, el acceso a dicha lógica del dominio desde el exterior se realiza a través de puertos y adaptadores. En la Figura 2 se pueden observar las tres capas principales que constituyen esta arquitectura:

Figura 2

Estructura de la arquitectura hexagonal



Nota. Tomado de *Hexagonal Architecture, DDD, and Spring*, por Łukasz Ryś, 2024, Baeldung (https://www.baeldung.com/hexagonal-architecture-ddd-spring)

- **Capa de aplicación:** mediante esta capa, el usuario o cualquier otro programa interactúa con la aplicación. Contiene el código que permite estas interacciones, por ejemplo, el código de la interfaz de usuario, las rutas HTTP para una API, o las serializaciones en JSON para los programas que consumen la aplicación.
- **Capa de dominio:** contiene el código que afecta e implementa la lógica de negocio. Este es el núcleo de la aplicación, por tanto, debe estar aislada tanto de la parte de aplicación

[6] Ryś Ł, Ryś Ł. Hexagonal Architecture, DDD, and Spring | Baeldung. Baeldung. https://www.baeldung.com/hexagonal-architecture-ddd-spring. Published 8 de enero de 2024.

como de la parte de infraestructura. Además, también debe contener interfaces que definan la API para comunicarse con partes externas.

- **Capa de infraestructura:** es la parte que contiene todo lo que la aplicación necesita para funcionar, como la configuración de la base de datos.

3.8.2.2 Implementación de pruebas automatizadas en la capa backend. Actualmente, las únicas pruebas automatizadas en la capa de backend con las que cuenta el proyecto RSI son unitarias, a nivel de la capa de dominio. Su implementación se hace con las herramientas que se tienen a disposición mediante dependencias como:

- **JUnit**[7]: framework estándar para la ejecución de pruebas unitarias en java.
- **Mockito**[8]: framework para imitar el comportamiento de dependencias externas, ayuda a simplificar el desarrollo de las pruebas.
- **AssertJ**[9]: librería que provee un conjunto de métodos usados para comprobaciones en las pruebas.

Dado que las pruebas unitarias deben considerarse como pruebas aisladas, y en este caso se realizan sobre las clases de servicio (lógica de negocio), y estas tienen dependencias de los repositorios mediante los cuales se comunica con base de datos o incluso, de otros servicios externos, el comportamiento de dichas interacciones con agentes externos a la clase en prueba se imita mediante Mockito, de esta forma también se evita el tener que levantar el contexto de la aplicación en su totalidad, ahorrando así recursos en la ejecución de las pruebas.

A la hora de implementar pruebas de mayor alcance, como de integración, es necesario tomar en cuenta la comunicación que realizan algunos microservicios con otros.

---

[7] Brannen, S et al., (n.d.). JUnit 5 user guide. https://junit.org/junit5/docs/current/user-guide/#overview-what-is-junit-5
[8] Mockito Framework site. https://site.mockito.org/.
[9] Baeldung. (2022, July 4). Introduction to AssertJ. Baeldung. https://www.baeldung.com/introduction-to-assertj

3.8.2.3 Comunicación entre microservicios. Dentro de una arquitectura basada en microservicios, donde cada uno es independiente del otro, resulta de gran relevancia la forma en que dos de ellos se puedan comunicar entre sí para compartir información.

Una herramienta utilizada para establecer dicha comunicación entre servicios en estos casos es *Feign*[10], una librería que forma parte del stack de *Spring Cloud*[11], para implementar clientes de servicios REST de forma declarativa. Algunas de las características principales de *Feign* son:

- Es altamente configurable, lo cual hace posible el uso de diversos codificadores y decodificadores para formatear la información tanto de la petición como de la respuesta de cada invocación de un servicio a otro.
- Soporta anotaciones de *JAX-RS* y *Spring MVC* para la declaración de los servicios REST.
- Se puede integrar perfectamente con los demás componentes del stack de *Spring Cloud*.

3.8.3 Capa frontend

Como último componente de la arquitectura del proyecto RSI se encuentra la capa de frontend, en la cual se lleva a cabo el desarrollo y despliegue de las interfaces de usuario, de forma similar al backend, está organizada en módulos separados (microfrontends) y escalables orquestados entre sí para construir la interfaz de usuario completa.

Los distintos microfrontends[12] se encuentran alojados y desplegados en un servidor web que cuenta con características de alto rendimiento y capacidad para soportar una gran cantidad de solicitudes, además, es de código abierto.

---

[10] Spring Cloud OpenFeign. (s. f.). Spring Cloud OpenFeign. https://spring.io/projects/spring-cloud-openfeign
[11] Spring Cloud. Spring Cloud. https://spring.io/projects/spring-cloud/.
[12] Geers, M. (2017, August 28). Micro frontends. Micro Frontends. https://micro-frontends.org/

Para la codificación se emplean lenguajes como Typescript dentro del framework de diseño de aplicaciones frontend *Angular*, un framework enfocado en desarrollo de aplicaciones de una sola página eficaces y sofisticadas, estas herramientas aportan robustez, facilidad de mantenimiento, alta documentación, soporte disponible y escalabilidad al proyecto.

4. Marco Metodológico

4.1 Marco de trabajo Scrum

El desarrollo de este proyecto se lleva a cabo dentro del marco de la metodología ágil *Scrum*, la cual es seguida dentro del proyecto RSI. Esta metodología permite abordar proyectos que se adapten de manera flexible en entornos complejos, dinámicos y cambiantes. Se centra en entregas parciales y regulares del producto final, priorizando el valor que proporcionan a los clientes. Además, Scrum facilita la mejora del trabajo colaborativo entre equipos, al promover equipos autodirigidos y auto-organizados. No solo brinda estructura al proceso, sino que también fomenta el aprendizaje y la organización basándose en experiencias; aborda desafíos y favorece la reflexión sobre éxitos y fracasos. Todo esto se logra a través de un conjunto de herramientas y recursos que permiten a los equipos organizarse ágilmente.

Scrum define artefactos, roles y eventos específicos para facilitar la gestión del proyecto y los desarrolladores se guían por historias de usuario para llevar a cabo sus implementaciones. Una *historia de usuario* es una descripción general de una función del software escrita desde la perspectiva del cliente, en ella se dan especificaciones de los requisitos del sistema y se establecen los criterios que se deben cumplir para aceptar el desarrollo que se le dé, los desarrolladores se encargan de revisarlas, puntuarlas en función de su complejidad y de desarrollarlas, luego se

someten a una serie de pruebas para finalmente darse por completadas y poder presentar su producto como un incremento.

4.2 Control de versiones

Para que cada desarrollador pueda tener acceso al código fuente de los proyectos que maneja, es necesario alojar dicho código en repositorios remotos con la posibilidad de ser clonados y modificados de forma local, así como de integrarse los nuevos cambios que se vayan agregando a lo largo del proceso de desarrollo, y dado que el trabajo de desarrolladores es colaborativo, es decir, más de una persona puede requerir hacer modificaciones sobre el mismo proyecto al tiempo, es necesario contar con una herramienta que controle esta integración de nuevas versiones de código, optimizando la gestión de cambios y dejando trazabilidad del proceso. Actualmente, la tecnología más popular y apropiada para esta función es *Git*[13], este sistema de control de versiones cuenta con una arquitectura distribuida, es decir que la copia de trabajo del código de cada desarrollador es a la vez un repositorio que puede almacenar el historial de todos los cambios. Existe una herramienta web basada en *Git*, llamada *GitLab*, la cual se complementa con *DevOps* y *CI/CD*, permite la gestión, administración, creación y conexión de los repositorios con diferentes aplicaciones y hacer todo tipo de integraciones facilitando así el software colaborativo.

4.3 Integración continua

Con el fin de ir integrando en el software nuevas funcionalidades o ajustes de las ya existentes, se utiliza la Integración continua junto con la entrega continua *CI/CD*, con esto se puede automatizar este paso de despliegues recurrentes. Esta herramienta permite implementar *pipelines*[14] en los cuales se establecen los comandos que se deban ejecutar sobre el código justo

[13] Atlassian. (n.d.). What is Git: Atlassian Git Tutorial. https://www.atlassian.com/git/tutorials/what-is-git

[14] GitLab. (2023, April 13). What is Ci/CD? https://about.gitlab.com/topics/ci-cd/#what-are-ci-cd-pipelines

antes de ser desplegado, por ejemplo, su compilación, revisión de métricas de calidad, ejecución de pruebas automatizadas, entre otras.

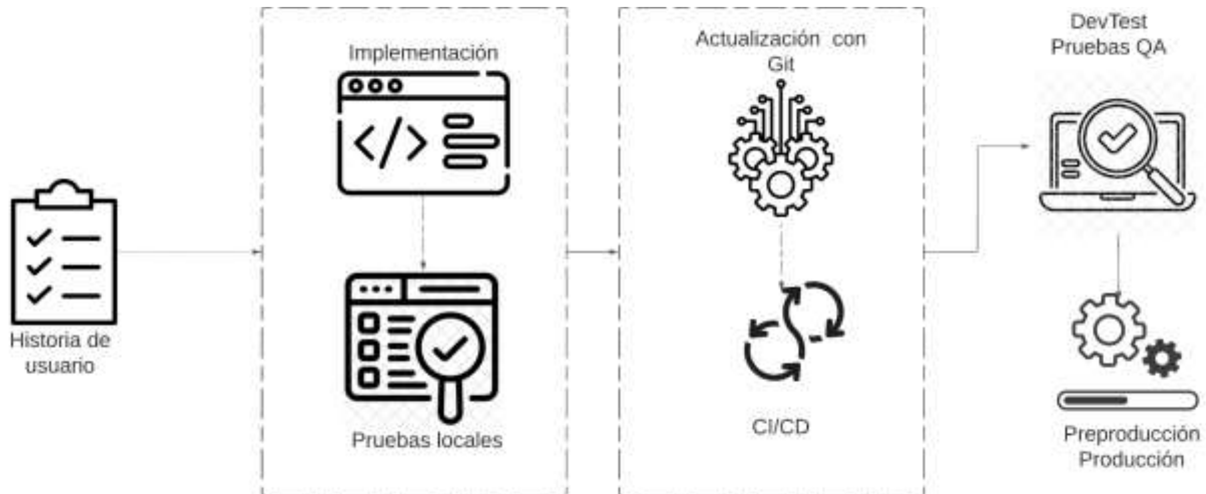
4.4 Flujo de trabajo

En síntesis, la metodología que se sigue para llevar a cabo los desarrollos en RSI está basada en el marco de trabajo Scrum, dirigido por historias de usuario que los desarrolladores ejecutan, realizan su proceso de pruebas locales y posteriormente, haciendo uso del controlador de versiones y la integración con CI/CD se agregan los cambios realizados al software ya existente y funcional; una vez se ha subido el desarrollo a este primer ambiente de despliegue, se lleva a cabo un proceso de *DevTes*t[15] entre desarrolladores mediante la ejecución de pruebas manuales y revisión de calidad de código para garantizar que todo esté correcto, una vez superada esta fase de pruebas se hace el despliegue a los demás ambientes de desarrollo donde el equipo de QA se encarga de realizar sus pruebas, garantizar que fueron cumplidos los criterios de aceptación de la historia de usuario y darla por completada para finalmente realizar el despliegue a ambientes de producción. Este flujo de trabajo se resume a grandes rasgos en el diagrama de la Figura 3.

Figura 3

Flujo de trabajo de RSI para desarrolladores

[15] What is DevTest? (s. f.). https://www.oracle.com/application-development/what-is-devtest/



Nota. Versión generalizada del flujo de trabajo en desarrollos.

5. Implementación

Para el desarrollo de este proyecto se realizó una primera fase de investigación en la cual se indagó sobre herramientas disponibles para la automatización de pruebas de integración sobre software con la arquitectura manejada por RSI, posteriormente se hicieron implementaciones de prueba para cada una de las tecnologías halladas que, mediante resultados mensurables, permitieran evaluar y definir la opción más conveniente para aplicar dichas pruebas dentro del contexto del proyecto, una vez planteada la propuesta, se escribieron pruebas de integración en uno de los microservicios centrales para ciertas funcionalidades seleccionadas tras haberles realizado un análisis que las clasificó como aptas para aplicarles pruebas de integración automatizadas, por último, se consolidó en la wiki del equipo RSI la documentación con el conocimiento necesario para los demás desarrolladores y analistas, de esta forma queda claro el proceso que seguirán para continuar implementando pruebas de integración en los nuevos

artefactos de software que se vayan creando y las consideraciones a tener en cuanto a la implementación de nuevas tecnologías para el uso de pruebas de software.

Todos los pasos anteriormente mencionados se desarrollaron bajo el marco de la metodología Scrum, la cual permitió ir evaluando los entregables de cada fase y hacer los ajustes necesarios que fuesen surgiendo. En esta sección se detalla la ejecución de cada fase del proyecto a través de historias de usuario, con sus respectivos objetivos, criterios de aceptación y desarrollo de estas.

5.1. Investigar sobre herramientas para implementar pruebas de integración automatizadas en la capa backend del proyecto RSI

5.1.1 Objetivo

El objetivo de esta historia de usuario es investigar y evaluar las herramientas disponibles actualmente para implementar pruebas de integración automatizadas en la capa backend, adecuadas para un software que cuente con una arquitectura como la del proyecto RSI. Cada alternativa que se contemple debe implementarse a manera de prueba para validar su comportamiento en el sistema y extraer resultados medibles que, dado el caso, se puedan poner en comparación con los de las demás alternativas que puedan existir y de esta forma definir cuál es la más conveniente.

Como punto de partida para el cumplimiento de este objetivo debe tomarse en cuenta que en el proyecto RSI ya se cuenta con pruebas unitarias automatizadas a nivel de backend, la tecnología con la que se desarrollan y el nuevo alcance que se le debe dar a las de integración.

La propuesta final que se seleccione como la forma adecuada de ejecutar pruebas de integración automatizadas debe cumplir con el propósito de mejorar la calidad del software y

optimizar los recursos empleados en pruebas integrales de funcionalidades específicas que así lo requieran.

5.1.2 Criterios de aceptación

- Hacer una revisión de la estructura y tecnologías empleadas en la capa backend del proyecto RSI que permita tener un punto de referencia a la hora de buscar herramientas para la aplicación de pruebas de integración.
- Seleccionar distintas alternativas de herramientas disponibles para la aplicación de pruebas de integración automatizadas que cumplan con los requisitos mínimos de compatibilidad con el sistema e investigar a fondo sobre cada una de ellas.
- Generar métricas de comparación entre las herramientas seleccionadas y a partir de ellas definir una propuesta final que corresponda a la más adecuada según los requerimientos del proyecto.
- La propuesta que se defina para la implementación de las pruebas de integración debe garantizar la integridad de las bases de datos relevantes del sistema, como, por ejemplo, ambientes de desarrollo, preproducción y producción.

5.1.3 Ejecución

Para llevar a cabo la definición de la forma más adecuada de implementar pruebas de integración en los microservicios del backend del proyecto RSI, inicialmente se revisó la arquitectura y tecnologías actuales que se manejan dentro del mismo, como fue mencionado anteriormente, el proyecto cuenta con una arquitectura de microservicios hexagonal usando el framework de desarrollo SpringBoot; la comunicación a cada servicio se realiza a través de las peticiones que se hacen por las APIs a la capa controladora, allí se deserializa la solicitud y se hace el llamado a la capa de servicio donde se sitúa la lógica de negocio. A partir de esta, se establece

la comunicación con la capa de repositorio para las tareas de persistencia sobre las bases de datos cuando sea necesario.

Una vez conocida la estructura del backend de RSI, se hizo una investigación exhaustiva de las tecnologías *SpringBootTest* y *TestContainers* que fueron las que se encontraron más compatibles con las necesidades del proyecto; antes de la puesta a prueba de cada una de estas herramientas, se realizó una revisión general de lo que son pruebas de integración.

5.1.3.1 Pruebas de integración. A diferencia de las pruebas unitarias, que se encargan de probar el funcionamiento de una unidad de código, llámese método o componente, las pruebas de integración se encargan de validar el funcionamiento de varias unidades en conjunto, es decir, prueba la interacción entre dos o más partes cohesivas.

Una prueba de integración puede definirse como una prueba que cubre todo el recorrido de la aplicación, es decir, desde que se envía una petición a la aplicación, pasa por la capa de lógica de negocio, hasta que haya cambiado el estado de la base de datos de acuerdo a las expectativas y responda correctamente; el objetivo de este tipo de pruebas es ver y analizar los posibles defectos y fallos en la interacción de las diferentes capas que componen una aplicación y ver cómo se integran entre ellas.

5.1.3.2 Pruebas de integración con SpringBootTest. Dentro del framework *SpringBoot* existe un paquete llamado *SpringBootTest* que contiene herramientas dedicadas a realización de pruebas; provee una anotación con su mismo nombre, @*SpringBootTest*, la cual es útil cuando se necesita levantar todo el contexto de la aplicación. Esto es lo que se busca a través de las pruebas de integración, ya que se pretende probar la interacción de todas las partes sin simular el comportamiento de ninguna de ellas. La anotación funciona creando el *ApplicationContext*[16] en

---

[16] ApplicationContext (Spring Framework 6.1.5 API). (s. f.). https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/ApplicationContext.html

un contenedor, que se utilizará en todas las pruebas que se tengan implementadas. Mediante un atributo llamado *webEnvironment* de *@SpringBootTest* se puede configurar el entorno de ejecución; por ejemplo, para que el contenedor funcione en un entorno simulado se puede hacer uso de WebEnvironment.MOCK. También es posible configurar un archivo de propiedades de configuración especial con el cual se levante el contexto de la aplicación para las pruebas, esto es útil cuando se desea establecer una conexión a una base de datos diferente para la ejecución de las pruebas o alguna otra característica debe ser diferente para su realización, de ello se encarga la anotación *@TestPropertySource*.

En síntesis, este paquete de pruebas de *SpringBoot* permite levantar un contexto de la aplicación con una configuración específica a utilizar en el proceso de pruebas y ejecutarlas de forma que validen el comportamiento real de las clases que intervienen dentro de cada funcionalidad que se esté probando.

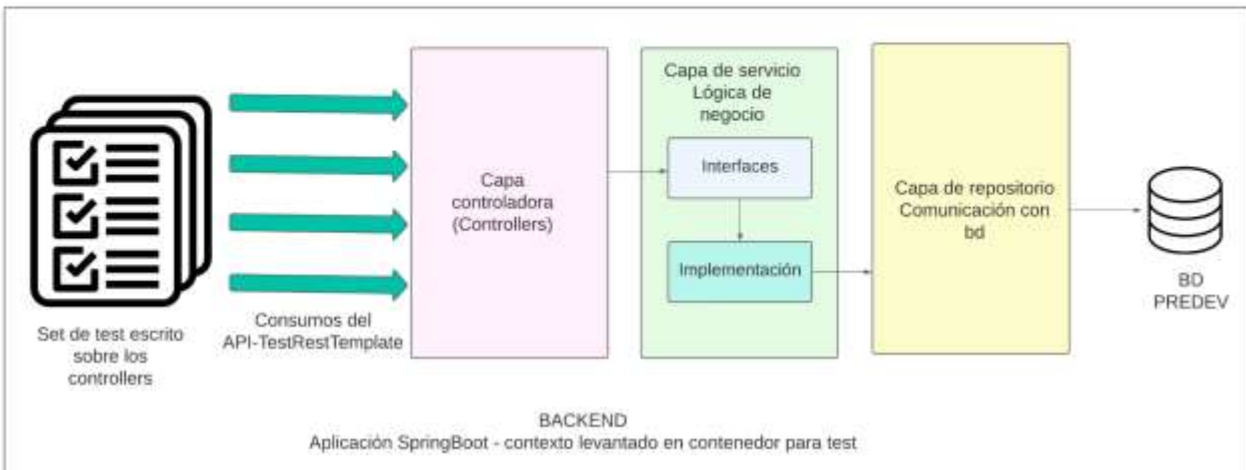
Para entender mejor el alcance de las pruebas de integración con esta tecnología se presenta el diagrama de la Figura 4, donde se puede observar cómo a partir de las clases de prueba escritas sobre las clases de la capa controladora, es decir, aquellas anotadas con *@Controller*, se envían las peticiones a través de *TestRestTemplate*[17] hacía los endpoints que se deseen probar del API del servicio; una vez generado el consumo, se sigue el flujo normal que seguiría una solicitud real desde un cliente, pasando de la capa controladora a comunicarse mediante la interfaz y ejecutar la capa de lógica de negocio (*@Service*), desde allí, si es necesario, se establece la interacción con la base de datos a través de los repositorios, en este caso, se tiene una conexión con una base de datos real del ambiente de desarrollo "*Predev*", la cual está designada para la realización de pruebas por parte de desarrolladores. De esta forma se logra probar el comportamiento al integrar, desde que

[17] baeldung. (2024, January 23). Exploring the Spring Boot Testresttemplate. Baeldung. https://www.baeldung.com/spring-boot-testresttemplate#overview

un cliente envía una petición, pasando por todas las capas de la aplicación, hasta llegar al elemento de persistencia.

Figura 4

Diagrama de cobertura por pruebas de integración con SpringBootTest



Las principales anotaciones que se utilizan para probar con este paquete se listan a continuación:

- **@SpringBootTest**: carga el contexto completo de la aplicación Spring y proporciona un entorno web simulado, además permite establecer que se ejecute la aplicación en un puerto aleatorio de la máquina host cada vez que se ejecutan las pruebas.
- **@LocalServerPort:** vincula el puerto aleatorio a la URL de entrada a la API, con esto luego se construye la URL en un método separado que puede ser reutilizado en toda la clase de prueba a ejecutar.
- **@TestRestTemplate:** permite crear un llamado al endpoint de la funcionalidad que se quiere probar.

- **@Autowired:** Inyecta la dependencia del servicio o repositorio con el cual se interactúa; el principal propósito de esto es hacer validaciones una vez se obtiene la respuesta del endpoint.
- **@Test:** esta anotación acompaña cada método de prueba que se implemente para que sea identificado como una prueba individual que se debe ejecutar en el proceso general.
- **@TestMethodOrder:** anotación a nivel global de la clase de prueba que habilita el uso de la definición del orden en el que se deben ejecutar los métodos pertenecientes a dicha clase.
- **@Order:** permite definir en qué orden se ejecutarán cada uno de los métodos de una clase de prueba, por ejemplo, si primero se requiere ejecutar un método que active el guardado de registros y luego uno que active la eliminación de estos, al primero se le define un orden de 1 y al segundo un orden de 2.
- **@BeforeAll, @BeforeEach, @AfterAll, @AfterEach:** anotaciones dentro de las cuales se pueden definir sentencias que se requieren ejecutar antes/después de todos o cada método de prueba, por ejemplo, inicializar un objeto o una variable.
- **AssertThat, AssertEquals, AssertNotNull, AssertNull, AssertTrue, AssertFalse:** este tipo de métodos pertenece a librerías como *AssertJ* o *JUnit* y se utilizan para hacer las validaciones de lo que se espera en la respuesta del endpoint asociado a la funcionalidad que se esté probando.

Para implementar las pruebas con esta tecnología, lo que se hizo fue adecuar el entorno con las configuraciones necesarias, para posteriormente poder escribir y ejecutar las pruebas.

- En principio se agregó la dependencia de spring-boot-starter-test con alcance de *test*, que habilita la realización de pruebas:

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-test</artifactId>
   <scope>test</scope>
</dependency>
```

- Así mismo, un plugin que habilita la ejecución de las pruebas con maven (requerido si se desea hacer la ejecución mediante comandos, por ejemplo dentro de la ejecución de un pipeline de despliegue):

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>3.0.0-M6</version>
  <dependencies>
     <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <version>5.6.2</version>
     </dependency>
  </dependencies>
</plugin>
```

- Mediante el uso de las anotaciones anteriormente mencionadas y conociendo la lógica de cada método que iba a ser testeado, se escribieron pruebas de integración para ciertas funcionalidades, un ejemplo de como se ve una de estas se muestra a continuación:

```
@SpringBootTest(webEnvironment=SpringBootTest.WebEnvironment.RANDOM_
PORT)
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
@Slf4j
public class RolControllerIntegrationTest {

   @Value("${app.url.base.test}")
   private String urlBase;

   @LocalServerPort
   private int port;

   private static HttpHeaders headers;
   private static HttpEntity<String> entity;
   private static RolDTO rolDTO;

   @Autowired
   private TestRestTemplate testRestTemplate;

   @Autowired
   private ITipoRolRepository iTipoRolRepository;

   @BeforeAll
   public static void init() {
       headers = new HttpHeaders();
       headers.setContentType(MediaType.APPLICATION_JSON);
       entity = new HttpEntity<>(null, headers);
       rolDTO = new RolDTO();
```

```
        rolDTO.setNombre("Prueba test integración - rol");
        rolDTO.setCodigo("TI_ROL_TEST");
        rolDTO.setDescripcion("Rol    creado    desde    prueba    de
integración");
        rolDTO.setHasPermits(true);
        rolDTO.setActivo(1L);
        rolDTO.setPermitsType("Asignados");
    }

    @Test
    @Order(1)
    void saveRolTest(){
        var idTipoRol = iTipoRolRepository.findTopTipoRol().getId();
        rolDTO.setIdTipoRol(idTipoRol);
        HttpEntity<RolDTO> entity =  new HttpEntity<>(rolDTO, headers);
        ResponseEntity<RolDTO> response = testRestTemplate.exchange(
                createURLWithPort("rol"), HttpMethod.POST, entity,
                RolDTO.class);
        log.info(response.toString());
        assertThat(response.getStatusCodeValue()).isEqualTo(200);
        assertNotNull(response.getBody());

assertThat(response.getBody().getNombre().equals(rolDTO.getNombre())
);
        assertNotNull(response.getBody().getId());
        rolDTO.setId(response.getBody().getId());
    }
    private String createURLWithPort(String urlModulo) {
        return urlBase + port + "/core/api/" + urlModulo;
```

```
    }
}
```

- Se realizaron ejecuciones de las pruebas implementadas para distintos casos de uso, prestando atención a su comportamiento respecto a integridad de base de datos, orden de ejecución de cada uno de los métodos y tiempo que tomaba en llevarse a cabo.

Tras haber hecho uso de esta herramienta para implementar pruebas, se pudieron extraer algunas observaciones importantes a tener en cuenta:

- Se deben escribir las pruebas de manera que los cambios que se hagan en base de datos, si los hay, sean exitosos, pero no persistan una vez terminada la ejecución de una prueba.
- Se puede utilizar herencia creando una clase principal de la cual extiendan las demás clases de prueba para que se comparta la configuración del ambiente de pruebas entre todas las clases de prueba a ejecutar; de esta forma se evita que se tenga que hacer uno por cada clase, eliminando duplicidad de código.
- Al usar la base de datos real, ya existen registros previos que pueden ser usados como insumos para escribir las pruebas, o tener registros previamente cargados para hacer uso de estos.

Partiendo de las observaciones anteriores que se pudieron establecer tras haber experimentado con esta herramienta, se buscó una opción que se pudiera usar en los casos que se deben deshacer los cambios hechos a nivel de bases de datos, por ejemplo, inserciones, eliminaciones o actualizaciones, y se identificó a *MockMvc*[18] como la solución para dichos casos.

[18] MockMvc. MockMvc :: Spring Framework. (n.d.). https://docs.spring.io/spring-framework/reference/testing/spring-mvc-test-framework.html

5.1.3.2.1 *Pruebas de integración con SprignBootTest y MockMVC.* *MockMvc* es un framework habilitado para ejecución de pruebas de integración en *Spring Boot*, que realiza la gestión completa de las solicitudes de Spring MVC, pero a través de objetos de solicitud y respuesta simulados en lugar de un servidor en ejecución, esta es su diferencia con el uso de *SpringBootTest* (con puerto definido) y *TestRestTemplate*. En comparación con un entorno real de servlets, *MockMvc* elimina la necesidad de arrancar el contenedor embebido que contendrá la aplicación (por ejemplo, Tomcat), de esta forma, no se ocupa ningún puerto, sino que se utiliza la instancia de *MockMvc* para interactuar con el entorno simulado y no se inicia una comunicación *HTTP* real. Sin embargo, este entorno de servlets simulado sigue la semántica *HTTP*, y permite modificar las cabeceras y el cuerpo de la petición (*MockMvcRequest*) e inspeccionar la cabecera, el estado y el cuerpo de la respuesta.

Para arrancar el entorno de pruebas con esta herramienta se puede utilizar la versión auto configurada de Spring Boot, la cual se implementa con la anotación *@AutoConfigureMockMvc.*

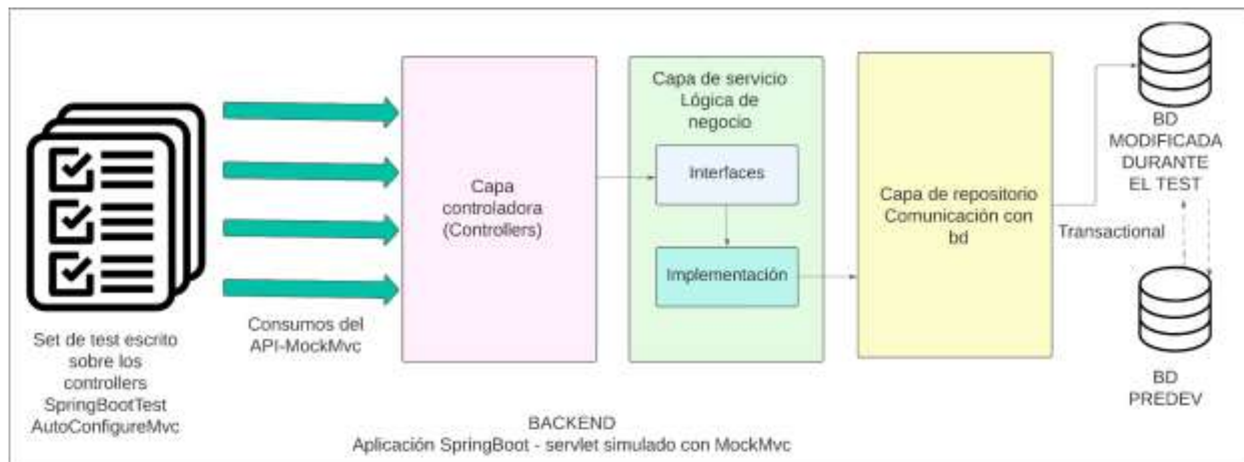
El hecho de que las peticiones *HTTP* no se hagan desde un servlet real corriendo en un puerto, sino desde uno simulado, hace posible el uso de *@Transactional,* una anotación que al ser usada se encarga de revertir los cambios que se hayan hecho en la base de datos una vez termine el método de prueba sobre el cual se haya puesto. De esta forma se solventa el inconveniente de dejar datos de prueba en la base de datos que podría causar problemas posteriormente, incluso corriendo las mismas pruebas en una segunda ocasión, por validaciones como campos con valores repetidos, o eliminación de registros relevantes que ya existiesen.

El flujo y alcance de las pruebas sigue siendo el mismo de las implementadas con *SpringBootTest* levantadas en un puerto específico con un servlet real, la única diferencia es que

el servlet en este caso será simulado y las peticiones se definirán con *MockMvc,* en la Figura 5 se muestra el esquema de las pruebas con esta tecnología.

Figura 5

Esquema de pruebas de integración con MockMvc



Adicional a las anotaciones que se mencionaron en el apartado 5.1.3.2, esta herramienta tiene algunas anotaciones extras que se describen a continuación:

- **@AutoConfigureMockMvc:** configura automáticamente la herramienta *MockMvc* para las pruebas.
- **@Transactional:** hace posible que una vez el método de prueba que se haya anotado con este decorador se ejecute, se haga rollback de los cambios que se hayan realizado en la base de datos, por ejemplo, inserción de un objeto, eliminación o actualización.

Por otra parte, la instancia de *MockMvc* que se inyecta para emitir las peticiones *HTTP* simuladas, tiene su estructura para este fin:

- **perform:** método que encierra todo lo que es la petición que se va a simular, dentro de él se define el tipo de método (GET, POST, PUT, etc), la url del endpoint a probar a partir del controller, es decir, sin incluir dirección de host ni puerto, el body o parámetros, los headers y contentType de la petición.

- **andExpects:** permite definir lo que se espera de la respuesta del endpoint, se puede repetir este método cuantas veces como validaciones se requieran implementar, permite evaluar desde el estado de la respuesta hasta campos específicos de los datos de la misma.
- **andDo:** con este método se pueden definir acciones que se quieran hacer una vez obtenida la respuesta del endpoint, usualmente se utiliza para imprimir el resultado.
- **andReturn:** si se requiere asignar la respuesta de la petición en un objeto aparte para luego manipularlo y hacer algunas otras validaciones u operaciones con él, este método lo hace posible.

Con esta tecnología también se implementaron pruebas, haciendo los cambios necesarios a nivel de anotaciones y forma de escritura de cada uno de las pruebas. Se tuvieron en cuenta los mismos parámetros de análisis del punto anterior y se evidenciaron mejoras respecto a la primera implementación.

A continuación, se tiene un ejemplo de una clase de pruebas implementada de esta manera:

```
@Transactional
@Slf4j
@SpringBootTest
@AutoConfigureMockMvc
@TestPropertySource(value = "classpath:application-test.properties")
public class GenericDataTypeControllerIntegrationTest {

    private ObjectMapper OBJECT_MAPPER = new ObjectMapper();

    @Autowired
    private MockMvc mockMvc;

    @Test
    void saveDataTypeTest() throws Exception {
```

```
        DataTypeDTO dataTypeDTO = new DataTypeDTO();
        dataTypeDTO.setName("Test name");

        var res = mockMvc.perform(post("/api/dato/dataTypes")
                        .header("X-Forwarded-For", "test")
                        .header("HTTP_CLIENT_IP", "test")
                        .header("User-Agent", "test")

.content(OBJECT_MAPPER.writeValueAsString(dataTypeDTO))
                        .contentType(MediaType.APPLICATION_JSON))
                .andExpect(status().isOk())
                .andReturn();

       var                              b                                =
OBJECT_MAPPER.readValue(res.getResponse().getContentAsString(),
Boolean.class);

        assertTrue(b);
    }
}
```

Tras realizar implementación y ejecución de pruebas de integración con este enfoque, se obtiene como conclusión que es el que mejor se acopla a las necesidades de los aplicativos actuales del backend de RSI, ya que permite usar la base de datos real destinada para pruebas, pero además ayuda a mantener la integridad de la misma al poder hacer rollback de las modificaciones que se realizan durante la aplicación de las pruebas y sigue cumpliendo el rol de probar la interacción entre las distintas capas del backend y persistencia.

5.1.3.3 Pruebas de integración con TestContainers. Testcontainers[19] es una librería que proporciona APIs sencillas y livianas para iniciar el desarrollo local y probar dependencias con

[19] Getting started. (s. f.). Testcontainers. https://testcontainers.com/getting-started/

servicios reales incluidos en contenedores[20] Docker. Con esto, es posible la adaptación del ambiente creado para ejecución de pruebas automatizadas hacia un tipo de ambiente completamente aislado donde se pueda hacer una evaluación de estas de manera más detallada y limpia, manteniendo tal ambiente lo más parecido a los demás donde se tienen alojados los proyectos, como producción. Haciendo uso de contenedores con dependencias como bases de datos o sistemas de mensajería, es posible replicar los servicios existentes con instancias reales en un estado igual al que estas se encuentran funcionando.

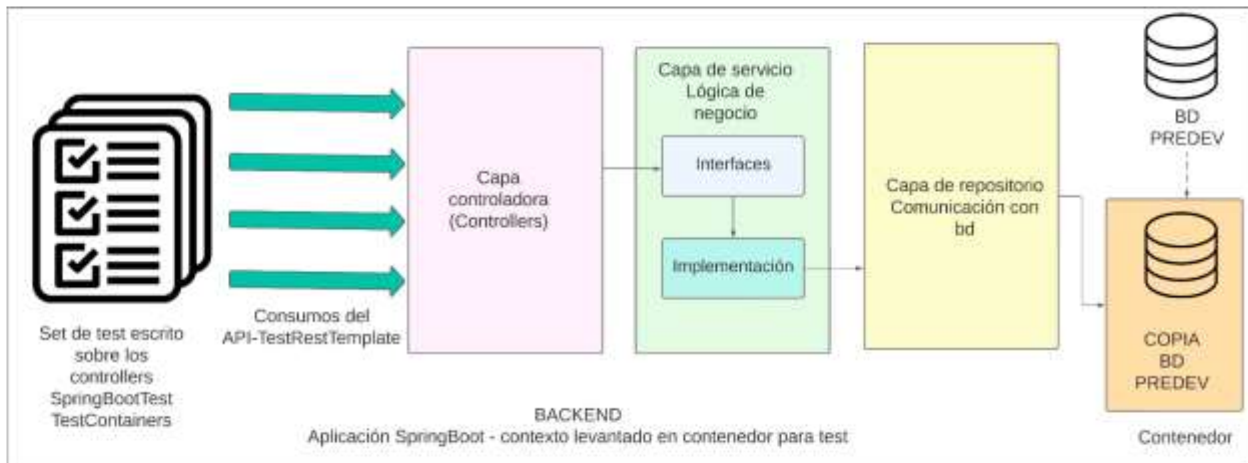
La principal diferencia obtenida al usar esta herramienta resulta ser que, al momento de levantarse el contexto de aplicación para la ejecución de pruebas de integración, la conexión a servicios externos se establece con los que se definen con *TestContainers* (bases de datos, servicios de mensajería, etc.).

En el caso de estudio de los proyectos backend usados en RSI, la ventaja identificada para el uso de esta herramienta fue el poder replicar la base de datos usada en el ambiente de pruebas *Predev*, de manera que se tuviese una instancia de esta exclusivamente para la ejecución de pruebas de integración creada y usada únicamente cuando se realicen despliegues de la aplicación y se ejecuten las correspondientes pruebas. Lo descrito anteriormente es un uso común que se le puede dar a la librería cuando se trata de la adaptación de un ambiente para ejecutar pruebas de integración y otros tipos que lo requieran; el beneficio otorgado por esto, es el conocimiento previo a la ejecución de las pruebas sobre el estado real de las dependencias en el instante, en este caso una base de datos. El esquema del flujo de las pruebas de integración con esta herramienta se muestra en la Figura 6.

[20] What are containers?. IBM. (n.d.). https://www.ibm.com/topics/containers

Figura 6

Esquema del flujo de pruebas de integración con TestContainers



Ahora, las especificaciones de cómo implementar el uso de la herramienta para pruebas de integración pueden variar dependiendo del tipo de servicio a usar; siendo este el escenario de una base de datos, se realizó una configuración sencilla para suplir la necesidad de uso de la herramienta y ver cómo se comportaba tomando una base de datos de Oracle:

- Para usar *TestContainers* en *Spring Boot* fue necesario agregar sus correspondientes dependencias en el proyecto (en este caso, con una base de datos Oracle, la configuración es específica).

```
<dependency>
   <groupId>org.testcontainers</groupId>
   <artifactId>testcontainers</artifactId>
   <version>1.16.2</version>
   <scope>test</scope>
</dependency>
<dependency>
   <groupId>org.testcontainers</groupId>
   <artifactId>junit-jupiter</artifactId>
```

```
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>oracle-xe</artifactId>
    <scope>test</scope>
</dependency>
```

- Teniendo en cuenta la eficiencia en cuanto al uso de recursos y tiempo de ejecución de pruebas, el uso de los contenedores necesarios puede administrarse de manera que solo se inicie un contenedor con el servicio a usar para todas las pruebas que apunten a usar *TestContainers* (aquellas clases anotadas con @*Testcontainers*). Esto es, un solo contenedor para todas las clases de pruebas que lo necesiten, evitando así iniciar un contenedor por cada clase de prueba que se ejecute o incluso por cada método de prueba que se ejecute en una clase; la ventaja de esto recae en tiempos de ejecución de pruebas optimizados, ya que el tiempo necesario para iniciar una instancia de un contenedor con una base de datos en un estado específico necesario para ejecutar pruebas puede tomar algunos minutos (lo que genera costos en tiempo teniendo en cuenta que este proceso se realiza cada vez que hay nuevas adiciones al software). Una implementación que pudo suplir esta idea fue haciendo el uso de un patrón de diseño *Singleton*[21] sencillo para administrar el contenedor del servicio a usar creado en una clase dedicada; esta se usó en la implementación de prueba para estudiar la viabilidad del uso de *TestContainers* en el caso de estudio de RSI:

```
@Slf4j
```

[21] Refactoring.Guru. (2024, 1 enero). Singleton. https://refactoring.guru/design-patterns/singleton

```
public class CommonTestContainer extends OracleContainer {
   private static final String IMAGE_VERSION = "gvenzl/oracle-xe";
   private static CommonTestContainer container;

   private CommonTestContainer() {
       super(IMAGE_VERSION);
   }

   public static CommonTestContainer getInstance() {
       if (container == null) {
           container = new CommonTestContainer();
       }
       return container;
   }

   @Override
   public void start() {
       super.withStartupTimeout(Duration.ofMinutes(9));
       super.start();
       log.info("Username: " + container.getUsername());
       log.info("Password: " + container.getPassword());
       System.setProperty("spring.datasource.url",
container.getJdbcUrl());
       System.setProperty("spring.datasource.username",
container.getUsername());
       System.setProperty("spring.datasource.password",
container.getPassword());
   }
```

```
    @Override
    public void stop() {
        //stopping container
    }
}
```

- Teniendo ya una configuración general de la instancia del contenedor a usar, lo único que se necesitó para poder usar la herramienta fue definir el objeto del contenedor a usar en las clases de pruebas de integración:

```
@Testcontainers
@SpringBootTest(webEnvironment =
SpringBootTest.WebEnvironment.RANDOM_PORT)
public class RutaNavegacionIntegrationTestContainer {

    @LocalServerPort
    private Long port;

    private static HttpHeaders headers;

    private static HttpEntity<String> entity;

    private static RutaNavegacion rutaNavegacion;

    @Autowired
    private TestRestTemplate restTemplate;

```

```
    @Container
    public static OracleContainer container =
CommonTestContainer.getInstance();

    @BeforeAll
    public static void init() {
        headers = new HttpHeaders();
        headers.setContentType(MediaType.APPLICATION_JSON);
        entity = new HttpEntity<>(null, headers);
    }

    @Test
    void contextLoads() {

    }
}
```

- En cuanto al uso de *Testcontainers* para iniciar una instancia de base de datos, es importante destacar el conocimiento previo del estado de esta que se necesita para ejecutar las pruebas (una base de datos vacía o con un esquema creado). Para el caso en el cual se necesita que la base de datos se inicie en un estado específico como puede ser la existencia de esquemas, tablas y vistas, es necesario añadirlo a la configuración como mejor se adapte; para esto se pueden adoptar ciertas medidas de resolución de la necesidad como el uso de archivos con sentencias definidas (SQL) para crear una base de datos haciendo uso de las herramientas propias del framework, un ORM o de

terceros como herramientas de migraciones de bases de datos como Flyway ó Liquibase, este último siendo el más común identificado para el caso de uso.

En los proyectos de backend de RSI no se cuenta con la implementación de una herramienta como Liquibase, por lo cual no es posible acceder a archivos propios de creación de una base de datos que se puedan usar para levantar un contenedor que replique la base de datos en un estado específico (el más actual); sin embargo, en la ejecución de esta historia de usuario se realizó una prueba mínima para poder identificar cómo podría ser su uso y así darle paso a la posible implementación de *TestContainers*.

Se realizaron copias del estado de la estructura de la base de datos en cierto momento haciendo uso de Liquibase y se generaron archivos *.sql* con la descripción de tal estructura para alimentar a la base de datos del contenedor como archivos de iniciación de esta, y así, poder tener el servicio disponible para ejecutar pruebas de integración.

El beneficio obtenido al integrar esto en la implementación de pruebas fue, más allá de buscar optimizar ciertas propiedades de estas (como los tiempos de ejecución, por ejemplo), el poder tener un ambiente de pruebas más robusto, aislado y controlado; como beneficio, esto permite que se pueda tener mayor conocimiento y confianza sobre el estado del sistema en el momento que presente fallas y se deba revisar detalladamente apoyándose en la ejecución de pruebas.

5.1.3.4 Evaluación de herramientas. Tras haber realizado implementaciones y pruebas con las dos herramientas mencionadas en los items 5.1.3.2 y 5.1.3.3, se tomaron como criterios de comparación el tiempo de ejecución, la cantidad de código que se requiere para escribir las pruebas junto con otras consideraciones extraídas de la experimentación con cada método. A continuación,

se presenta en la Tabla 1 el paralelo entre ambas opciones, lo cual fue base para definir el planteamiento de implementación de pruebas de integración automatizadas en el backend de RSI.

Tabla 1

Comparación entre herramientas para implementación de pruebas de integración

Criterio	**Herramientas**	
-	SpringBootTest y Base de datos de ambiente *predev*	TestContainers
Tiempo de ejecución	757 ms / método de prueba aproximadamente	~7 min (levantando contenedor) + 433 ms / método de prueba aproximadamente
Densidad de código (para un caso específico)	126 líneas	160 líneas
Comunidad y soporte disponibles	Sí	Sí (pero en menor proporción respecto a la otra alternativa)
Costos por licencia	No	No

Los resultados anteriores se extrajeron evaluando ambas opciones bajo las mismas condiciones, es decir, con la misma cantidad de métodos de prueba, que en total fueron 24 y para las mismas clases, de forma que requirieron la misma lógica. Evidentemente, resulta más sencillo y menos costoso hacerlo mediante la primera alternativa. Adicional a estos datos numéricos, en la

Tabla 2 se presentan otras observaciones que se tienen en cuenta a la hora de decantarse hacia alguna de las alternativas.

Tabla 2

Consideraciones de evaluación de herramientas para implementación de pruebas de integración

SpringBootTest y base de datos de ambiente *predev*	**TestContainers**
No requiere configuraciones adicionales ni código con detalles de implementación para habilitar la base de datos.	Es necesario configurar la copia de la base de datos real en el contenedor, lo cual puede demorar un tiempo exagerado (2 horas aproximadamente) si no se tiene una copia previamente lista.
Si se realizan cambios en la estructura de las entidades de base de datos, ya están disponibles cuando se vayan a ejecutar las pruebas.	Se requiere un manejo de versiones de la base de datos, lo cual implica tecnologías y análisis adicionales, así como más tiempo de ejecución en preparación del ambiente para luego pasar a ejecutar pruebas.

Al estar usando una base de datos existente (*predev*) se cuenta con datos base en los casos que se requieren entidades de las cuales hay dependencia, por ejemplo para una prueba que involucra la entidad *Rol*, ya existe otra de *TipoRol* que puede ser usada.	Si no se especifican datos para poblar la base de datos en el contenedor, no hay datos que se puedan utilizar en las pruebas, es necesario crear todos los registros necesarios correspondientes a entidades que involucren usarse en una prueba.
Para evitar dejar los datos generados con las pruebas en la base de datos, hay que eliminar lo que se haya creado al final de cada una. Esto puede optimizarse implementando *Transactional* y *MockMvc* juntos.	No es necesario eliminar los datos que quedan después de pruebas, ya que cuando se termina el proceso, la instancia del contenedor (y la base de datos) también termina su ejecución y es eliminada.

A partir de los resultados y consideraciones anteriores, se definió que la forma más óptima para implementar pruebas de integración para el backend de RSI es utilizando SpringBootTest en conjunto con MockMvc y la conexión a base de datos del ambiente de desarrollo *predev*, la cual es usada para pruebas por parte de desarrolladores. Esta consideración está basada en las pruebas que se realizaron del uso de las herramientas mencionadas; es importante tener en cuenta que pueden existir ciertos casos de prueba para funcionalidades específicas donde sea más conveniente el uso de alguna configuración adicional o diferente, lo cual se debe evaluar al momento de definir cómo realizar tales pruebas. Cabe mencionar que todo el instructivo y pasos a seguir con especificaciones técnicas sobre cómo realizar la implementación de pruebas de integración con el planteamiento definido se dejó documentado dentro de la wiki del proyecto RSI para que los demás integrantes puedan tener acceso a esta guía.

Dado que en la metodología manejada dentro del proyecto se cuenta con *CI/CD*, fue necesario integrar la ejecución de estas pruebas dentro de este proceso, teniendo en cuenta que la integridad de bases de datos de ambientes especiales, como son *desarrollo* y *producción*, debe quedar garantizada.

5.1.3.5 Pruebas de integración automatizadas aplicadas con CI. Para que cada vez que haya cambios en el código de algún microservicio se realice una verificación del cumplimiento de las pruebas de integración, fue necesario que en el *pipeline* de CI se agregara un paso que se encargue de esta labor y no permitiera el despliegue si no se superan las pruebas con éxito. Este nuevo paso solo se lleva a cabo en el momento en el que el desarrollador desee integrar sus cambios al ambiente de *predev*, ya que es allí donde se tiene la base de datos dispuesta para ejecutar estas pruebas. Para hallar la forma de implementar este elemento, se hizo un análisis de la configuración con la que contaba el proceso de CI para los despliegues del backend en RSI, posteriormente, se agregó el *job* de ejecución de pruebas de integración a medida que se realizaban experimentos y con base en el comportamiento obtenido se hicieron correcciones y complementos a la configuración hasta obtener la ejecución y reporte como se esperaba. De forma resumida, la disposición del job como finalmente se definió, se presenta a continuación:

- Cada proyecto tiene una configuración de los *jobs* pertenecientes a los *pipelines* que se deben ejecutar en cada despliegue, para esto se agregó el *job* de ejecución de pruebas de integración haciendo uso de las directivas necesarias para automatizar este proceso:

```
integrationTest:
  stage: verify
  rules:
    - if: '$CI_PIPELINE_SOURCE == "push" &&
           $CI_COMMIT_BRANCH == "predev"'
```

```
allow_failure: false
cache:
  - key: "${CI_COMMIT_REF_NAME}"
    paths:
      - .m2/repository
      - target
    policy: pull
variables:
 …
script:
       - chmod +x mvnw
    -./mvnw
    -Dtest="co/edu/uis/**/integration/**" test
artifacts:
  when: always
  reports:
    junit:
      - target/surefire-reports/TEST-*integration*.xml
```

Allí se define que el *job* pertenece al paso de verificación de pruebas, se ejecutará solo cuando se hacen actualizaciones al ambiente *predev*, no se permitirá continuar el despliegue si hay fallas en las pruebas, se toman las variables necesarias para la configuración de la aplicación desde las variables de entorno almacenadas en GitLab, se define el comando de ejecución mediante un comando de ejecución de pruebas de *maven*[22] y se define que lo que se ejecutará corresponde a pruebas de integración, por último se configuró la generación del reporte de resultados de las pruebas.

[22] Porter, B., Zyl, J. van, & Lamy, O. (2024, April 12). Welcome to Apache Maven. Maven. https://maven.apache.org/

- Con lo anterior, se logró una configuración práctica, limpia y útil para que se ejecuten las pruebas de integración automáticamente cada vez que se incluyen cambios al repositorio cada proyecto y se pueda visualizar el reporte de resultados de dichas pruebas en la misma plataforma de GitLab. Los reportes incluyen información puntual y de valor respecto a la ejecución de pruebas, como los tiempos usados por cada prueba, cuáles fallaron y detalles de las fallas en caso de que haya, esto se observa en la Figura 7.

Figura 7

Ejecución automática y resultados de pruebas de integración en despliegue de ambiente predev

Nota. Capturas de ambiente CI/CD en la ejecución y resultados de pruebas automatizadas.

5.1.3.6 Reporte de cobertura por pruebas de integración. Dentro del proyecto RSI se cuenta con la herramienta SonarQube, una plataforma para evaluar código fuente que permite encontrar problemas de legibilidad o de seguridad y además genera métricas que pueden ayudar a mejorar la calidad del código y otros aspectos de un software. Esta herramienta entra a cumplir su rol al momento en que los desarrolladores hacen sus despliegues a los ambientes de desarrollo; se analiza cada pieza nueva o modificada de código que se integre y se generan los reportes de análisis sobre esto. Sin embargo, no se tenía información sobre la cobertura de código monitoreado por parte de pruebas automatizadas, por tanto, como parte de este proyecto se planteó la forma de implementar este tipo de reporte para que complemente la integración de ejecución de las pruebas en los jobs de despliegue a predev y los reportes de resultados de pruebas que se incluyeron anteriormente.

Para dar desarrollo a este punto de la historia de usuario, se realizó una revisión de la documentación de *SonarQube* y sus funcionalidades teniendo mayor énfasis en los reportes de pruebas automatizadas y posteriormente se procedió a configurar cada dependencia y parámetro necesario tanto a nivel de proyecto como de *pipeline* en GitLab para obtener el reporte previsto.

- Se implementó el uso de *jacoco,* herramienta que se encarga de generar reportes en formato xml al compilar el proyecto backend y validar las pruebas automatizadas que tenga escritas, dichos reportes posteriormente son leídos y mostrados por la interfaz de SonarQube relacionada con el proyecto en cuestión.
- En la configuración del *pipeline* de CI para despliegue en GitLab hacia el ambiente de *predev* se habilitó en el *job* de compilación la ejecución de pruebas para que de esta forma, por acción del plugin dedicado, se generase el reporte de cobertura de código por parte de las pruebas implementadas.

- Con la configuración lista, se logró que cada vez que se ejecute un *pipeline* de despliegue a *predev*, se genere el reporte de cobertura que se puede visualizar de la forma que se observa en las Figuras 8-11; dentro de este se puede indagar la cantidad de líneas de código monitoreadas respecto al total, así como la cantidad de condicionales, el porcentaje alcanzado por paquete y por clase, además, en cada clase del código fuente se puede ver señalizado lo que está cubierto y lo que no por las pruebas automatizadas.

Figura 8

Visualización de porcentaje de cobertura por pruebas automatizadas

Figura 9

Informe general sobre cobertura de código

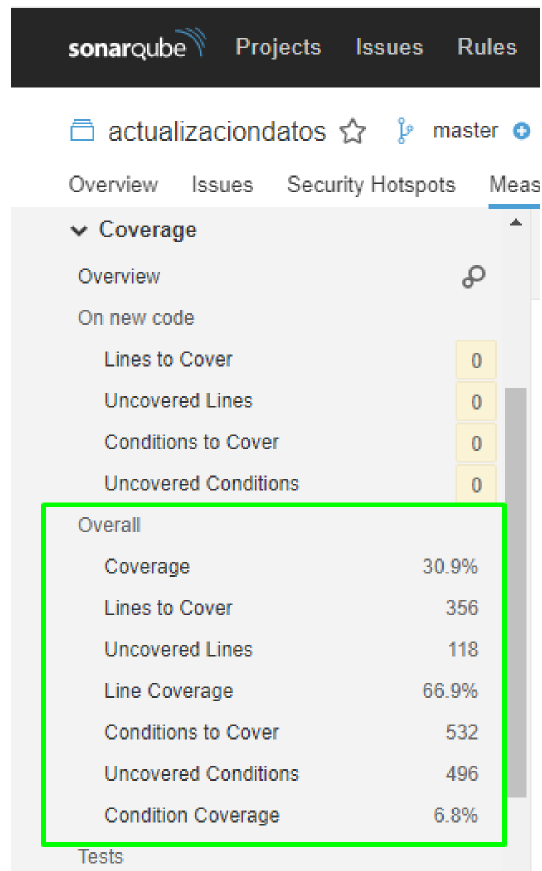


Figura 10

Reporte de porcentaje de cobertura para cada paquete de un proyecto backend

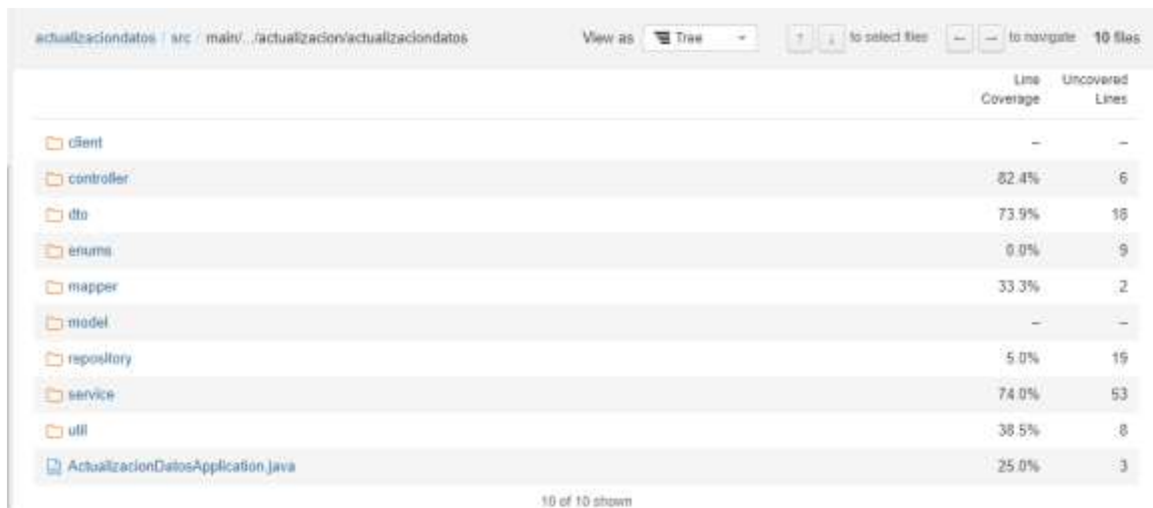


Figura 11

Visualización por colores de líneas cubiertas (verde) y no cubiertas (rojo) en una clase

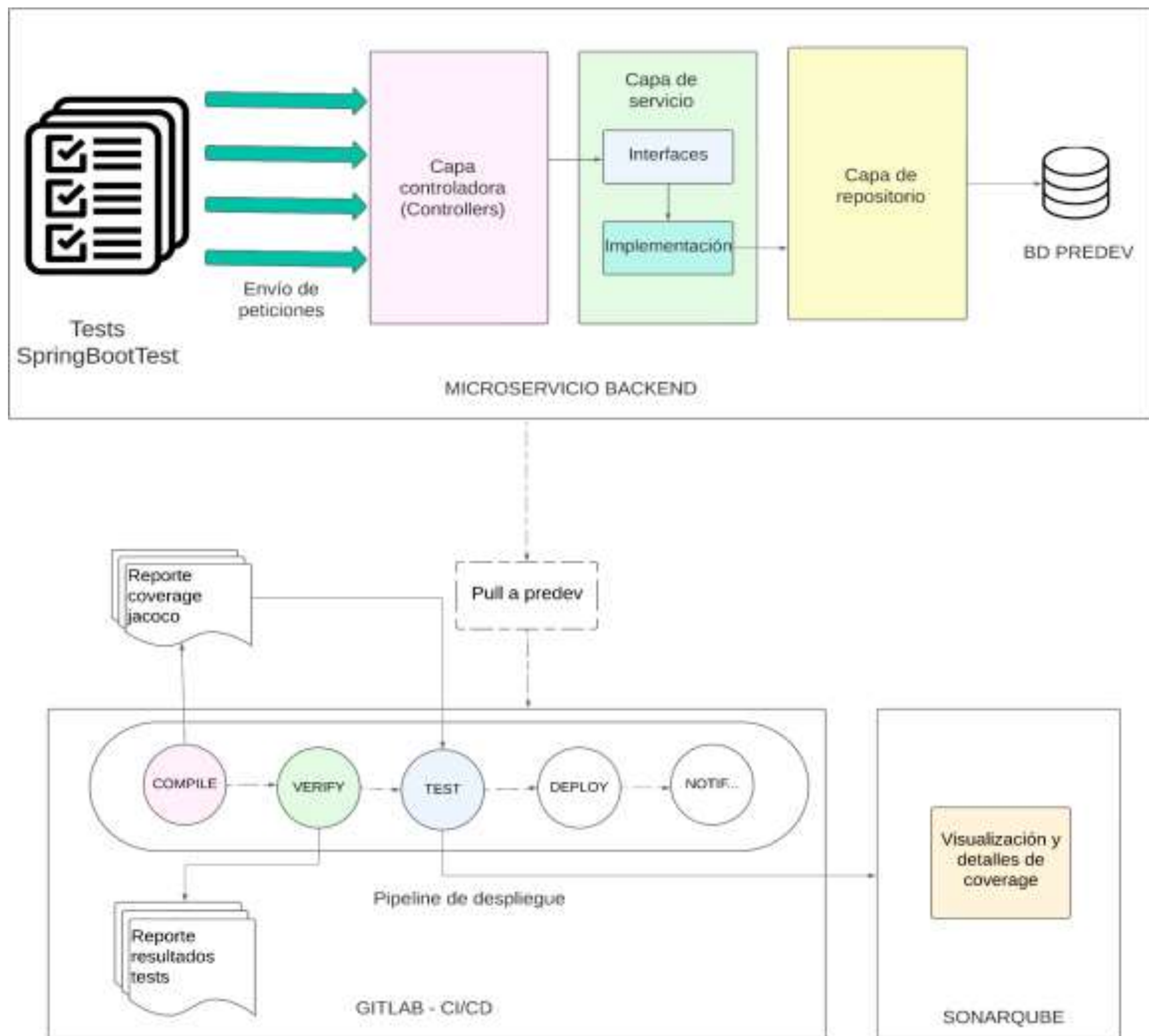
actualizaciondatos / src / main/.../actualizacion/actualizaciondatos / controller / GenericDataTypeController.java

```
28  migue…    @GetMapping("/dataTypes")
29            public ResponseEntity<List<DataTypeDTO>> getDataTypes() {
30                var dataTypes = this.genericDataService.getDataTypes();
31                return new ResponseEntity<>(dataTypes, HttpStatus.OK);
32            }
33
34  migue…    @PostMapping("/dataTypes")
35  migue…    public ResponseEntity<Boolean> saveDataType(@RequestBody DataTypeDTO dataType) {
36  migue…        var response = this.dataTypeService.saveDataType(dataType);
37                return new ResponseEntity<>(response, HttpStatus.OK);
38            }
39
40            @PutMapping("/dataTypes")
41  migue…    public ResponseEntity<Boolean> updateDataType(@RequestBody DataTypeDTO dataType) {
42  migue…        var response = this.dataTypeService.updateDataType(dataType);
43                return new ResponseEntity<>(response, HttpStatus.OK);
44            }
45
46            @DeleteMapping("/dataTypes")
47            public ResponseEntity<Boolean> deleteDataType(@RequestParam Long id) {
48                var response = this.dataTypeService.deleteDataType(id);
49                return new ResponseEntity<>(response, HttpStatus.OK);
50            }
```

5.1.3.7 Esquema general de planteamiento para implementación de pruebas de integración automatizadas para la capa backend en el proyecto RSI. En síntesis, la forma en que se plantea la implementación de pruebas de integración automatizadas para los microservicios del backend de RSI se ilustra en la Figura 12, donde se contempla desde la implementación de las pruebas con la herramienta SpringBootTest haciendo conexión a la base de datos real de ambiente predev; su posterior ejecución dentro del pipeline de despliegue de CI en GitLab que a su vez genera los reportes detallados de las pruebas ejecutadas y por último la visualización del reporte de cobertura de código en SonarQube.

Figura 12

Diagrama global generalizado de implementación de pruebas de integración en microservicios backend de RSI



5.2. Investigar sobre herramientas para implementar pruebas de integración automatizadas en casos con comunicación entre microservicios.

5.2.1 Objetivo

Dado que existen funcionalidades dentro del sistema que requieren la comunicación entre distintos microservicios y contemplando este caso como un tipo especial de interacción entre componentes del sistema, el objetivo de esta historia de usuario es indagar acerca de cómo escribir

y ejecutar pruebas de integración sobre servicios que involucren el consumo de servicios externos, siendo estos otros microservicios del proyecto RSI o de terceros. La idea de esto es encontrar una solución a la necesidad de aplicar pruebas de integración a los servicios que recaen en este caso de uso teniendo en cuenta que al consumir servicios externos pueden presentarse casos como la no disponibilidad de estos en cierto momento, los cambios que pueden tener en su funcionamiento y otras consideraciones que puedan resultar, de manera que se pueda conservar el correcto funcionamiento de las pruebas implementadas y de no ser así, los resultados de las pruebas automatizadas notifiquen del error inducido en caso de que se presenten problemas.

5.2.2 Criterios de aceptación

- Revisar la tecnología con la que se realiza la comunicación entre microservicios dentro del backend de RSI para tener un punto de partida en la búsqueda de las herramientas necesarias.
- Realizar un proceso de búsqueda y evaluación de tecnologías disponibles para implementar las pruebas de integración automatizadas cuando hay comunicación entre servicios separados (microservicios de RSI o de terceros).
- Implementar y probar las alternativas encontradas para filtrar la opción que mejor satisfaga los requerimientos y se acople a la estructura existente del backend de RSI.
- Realizar pruebas para garantizar que el proceso definido para implementar este tipo de pruebas no impacte la metodología de CI/CD en ambiente *predev*.

5.2.3 Ejecución

Para desarrollar esta historia de usuario, inicialmente se hizo una revisión de la metodología actual seguida en RSI para establecer comunicación entre microservicios de la capa backend. Esto se hace mediante la herramienta *Feign*, que como se mencionó anteriormente,

permite crear un cliente de servicios para hacer consumos *REST* y hace más ligero y sencillo la implementación de dichas peticiones entre distintos componentes. *Feign*, al ser una librería perteneciente al stack de Spring Cloud, es compatible con el resto de elementos de este mismo conjunto, por ejemplo, *Spring Cloud Contract.*

Una vez conocida esta especificación, se pasó a una etapa de investigación sobre herramientas destinadas a probar funcionalidades que consuman servicios ajenos al microservicio en el que se encuentran implementadas.

Para solventar este tema, existe un enfoque denominado Contract Tests[23], o traducido, pruebas de contrato, que se refiere a una técnica de pruebas en el desarrollo de software donde se verifica que los servicios, productores y consumidores, se comporten según las especificaciones definidas en los contratos. Para un mejor entendimiento, un productor se refiere al servicio que será llamado desde el cliente o consumidor a través de *Feign*, en el caso específico del backend de RSI; un contrato describe cómo debe comportarse un servicio en función de las solicitudes que recibe y las respuestas que devuelve. Estos contratos actúan como interfaces claras y se utilizan para asegurar que las partes involucradas se comuniquen correctamente, incluso si están desarrollados por equipos diferentes.

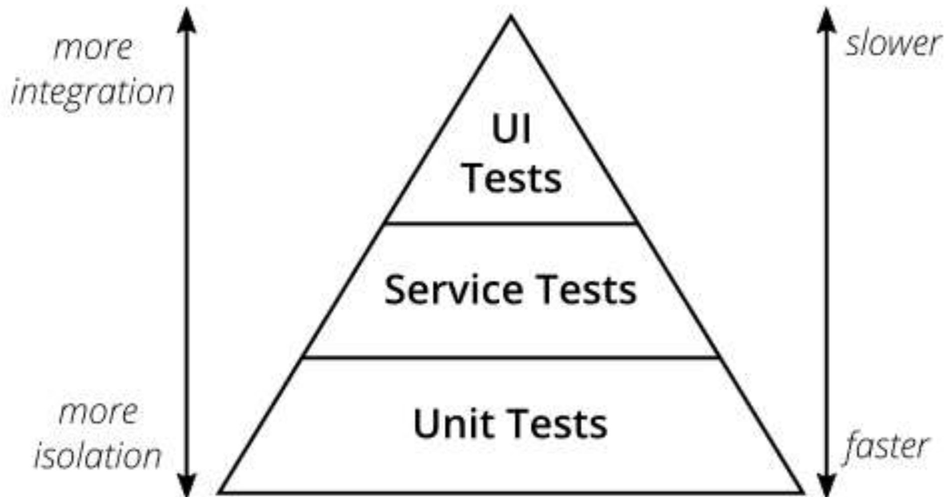
En literatura variada, este enfoque se trata como otro tipo de pruebas de software, incluyéndose dentro de la *pirámide de pruebas*; sin embargo, de acuerdo al alcance que se le quiera dar, puede ubicarse bajo o sobre las pruebas de integración, más adelante se argumenta ambos puntos de vista.

[23] Foreman, A. (2023, January 4). Contract testing vs integration testing: Pactflow. Pactflow Contract Testing Platform. https://pactflow.io/blog/contract-testing-vs-integration-testing/

Mike Cohn en su libro *Succeeding with Agile* presentó la pirámide de pruebas básica, que consta de tres capas e indica cómo deberían verse las pruebas:

Figura 13

Pirámide de pruebas de software de Cohn



Nota. Adaptado de The Practical Test Pyramid, por Ham Vocke, 2018, martinFowler.com (https://martinfowler.com/articles/practical-test-pyramid.html)

Debido a su simplicidad, la esencia de la *pirámide de pruebas* sirve como una buena regla general a la hora de establecer un conjunto de pruebas propio, teniendo en cuenta dos premisas de la pirámide original de Cohn:

- Escribir pruebas con diferente granularidad.
- Cuanto más alto sea el nivel, menos pruebas deberá tener, ya que aumenta la complejidad.

Basándose en esta pirámide se pueden crear variaciones según las necesidades de cada sistema; en RSI se cuenta con pruebas unitarias y ya se han involucrado las pruebas de integración,

ahora se analiza cómo incluir pruebas por contrato según el enfoque que más se ajuste. Existen dos formas posibles para ubicar las pruebas por contrato dentro de la pirámide: una en la que se encuentran entre las pruebas unitarias y las de integración y otra donde se encuentran un nivel más arriba de las pruebas de integración, a continuación, se presentan más detalles sobre cada una de las aproximaciones al tema:

- Pruebas por contrato ubicadas entre pruebas unitarias y de integración: en este enfoque se consideran las pruebas por contrato como pruebas menos complejas que las de integración, ya que solo se ocupan de probar la interacción entre una petición desde un servicio a la API de otro, verificando que la respuesta de uno encaje en lo que espera el otro, pero no se contempla el sistema completo, entendiendo por sistema la lógica de negocio, interacción con componente de persistencia (base de datos) y otros. Además, considera que las pruebas por contrato son más rápidas que las de integración, sin embargo, siguiendo la premisa de la pirámide original de Cohn, al ubicarse en este nivel las pruebas por contrato deberían ser más en cantidad respecto a las pruebas de integración, lo cual resultaría no ser tan cierto en la realidad (al menos en el presente caso de estudio), ya que son más los casos de procesos complejos que requieran pruebas de integración que los casos en los que existe comunicación entre microservicios. Por otro lado, en lugar de ejecutar una respuesta real del productor, se simula dicha respuesta, por tanto, se da lugar a errores que no serán detectados eficazmente aunque existan las pruebas.
- Pruebas por contrato ubicadas sobre las pruebas de integración: desde esta perspectiva se considera que las pruebas por contrato son menores en cantidad respecto a las de integración, y que además se tiene acceso tanto al productor como al consumidor,

pudiendo así escribir el contrato también desde el lado del productor; además se contempla una verificación del sistema de forma más completa, tanto en sus interacciones entre las diferentes capas a través de puertos y adaptadores, como en sus interacciones con microservicios de otros componentes, no obstante, esto toma más tiempo.

Para efectos del proyecto, se consideraron ambos enfoques, dado que existe la posibilidad de que en algún punto se deba establecer comunicación desde un microservicio con un proveedor externo, es decir, al cual no se puede acceder para escribir pruebas sobre él, será útil saber cómo simular la respuesta que deba generar ese endpoint del cual se tiene dependencia. Sin embargo, la metodología principal que debe seguirse para el resto de los casos es el segundo, donde se prueba la interacción completa entre ambas partes sin imitar comportamientos.

Existe un paquete llamado *Spring Cloud Contract*[24] diseñado para la implementación de pruebas por contrato en Spring Boot, su función es validar el contrato (pacto de interacción) entre un productor y un consumidor en un sistema de servicios distribuidos, para ello cuenta con Spring *Cloud Contract Verfier*[25], una herramienta que permite el desarrollo de *Consumer Driven Contract (CDC)*[26] en aplicaciones basadas en *JVM*[27]; CDC es una metodología de pruebas de software que se utiliza para probar los componentes de un sistema de forma aislada, garantizando al mismo tiempo que los componentes del proveedor son compatibles con las expectativas que los componentes del consumidor tienen de ellos. Este componente se presenta con un *lenguaje de*

[24] Spring Cloud Contract 4.1.2. Spring Cloud Contract. (n.d.). https://spring.io/projects/spring-cloud-contract#overview

[25] Spring Cloud Contract. 2. spring cloud contract verifier introduction. (n.d.). https://cloud.spring.io/spring-cloud-contract/2.1.x/multi/multi__spring_cloud_contract_verifier_introduction.html

[26] ISE, M. (2023, May 15). Consumer-Driven Contract Testing (CDC). Consumer-driven Contract Testing (CDC) - Engineering Fundamentals Playbook. https://microsoft.github.io/code-with-engineering-playbook/automated-testing/cdc-testing/

[27] Java Virtual Machine. (n.d.). https://www.ibm.com/docs/en/i/7.3?topic=platform-java-virtual-machine

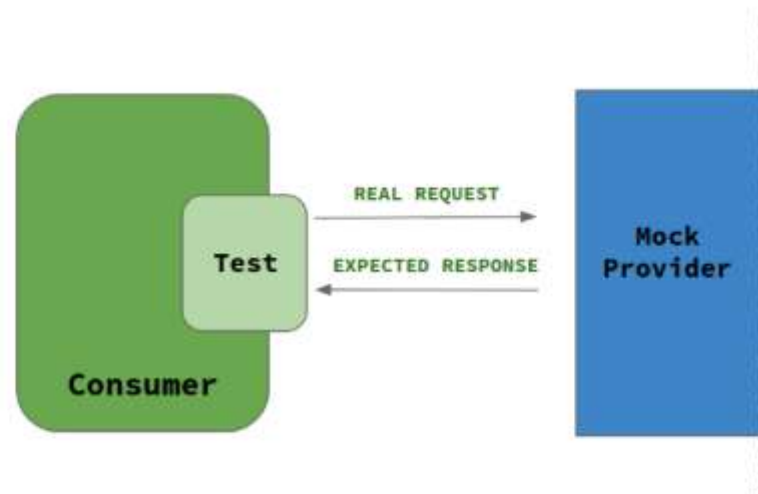
definición de contratos que puede ser escrito en Groovy[28] o YAML[29]. Las definiciones de contrato se emplean para generar varios recursos, incluyendo definiciones predeterminadas de *stubs* en formato *JSON* destinadas a *WireMock (HTTP Server Stub)* para su uso durante las pruebas de integración en el código del cliente. Es importante señalar que, si bien el código con los datos de prueba aún debe ser creado manualmente, *Spring Cloud Contract Verifier* se encarga de generar las pruebas completas.

Como se mencionó, ambos enfoques serán presentados como opciones de implementación de este tipo de pruebas, a continuación, se detalla cada uno de ellos.

5.2.3.1 Pruebas por contrato empleando mock de la respuesta del productor

Figura 14

Flujo de prueba por contrato haciendo mocks de la respuesta del proveedor



[28] Documentation. The Apache Groovy programming language - Documentation. (n.d.). https://groovy-lang.org/documentation.html
[29] YAML syntax. YAML Syntax - Ansible Community Documentation. (2024, April 12). https://docs.ansible.com/ansible/latest/reference_appendices/YAMLSyntax.html

Como se observa en la Figura 14, en este caso se escribe la prueba desde el servicio consumidor; durante la ejecución de este, en algún punto se envía la petición real al endpoint que consume, y se le da una respuesta esperada mediante un mock[30] (entidad creada para imitar el comportamiento de un objeto o componente real dentro de un sistema de software) del servicio proveedor.

Con esta tecnología también se llevó a cabo la respectiva implementación de prueba, mediante la cual se iba experimentando el comportamiento generado y haciendo los ajustes respectivos respecto a herramientas utilizadas, validaciones establecidas y detalles en la escritura de contratos hasta obtener la versión funcional óptima del planteamiento.

- De forma similar a como se hizo la implementación de pruebas de integración entre las diferentes capas de un microservicio, en este punto, también se agregaron las correspondientes dependencias que permiten usar *WireMock* para generar el mock de la respuesta del proveedor,
- Para hacer efectiva la simulación de las respuestas de los servicios consumidos, se configuró la clase de pruebas con la implementación de la herramienta para el caso de uso y se terminó de escribir el resto de contenido de las pruebas, las cuales se hacen de la misma forma que una prueba de integración vista en el punto 5.1.3.1, se llevó a cabo su ejecución y análisis de los resultados obtenidos, entendiéndose por resultados a los tiempos de ejecución empleados y criterios cualitativos sobre esta herramienta. En la siguiente sección de código se presenta un ejemplo de implementación señalando en color azul la principal característica de este tipo de prueba:

```
@AutoConfigureWireMock
```

[30] Mock testing. Split. (2024, March 28). https://www.split.io/glossary/mock-testing/

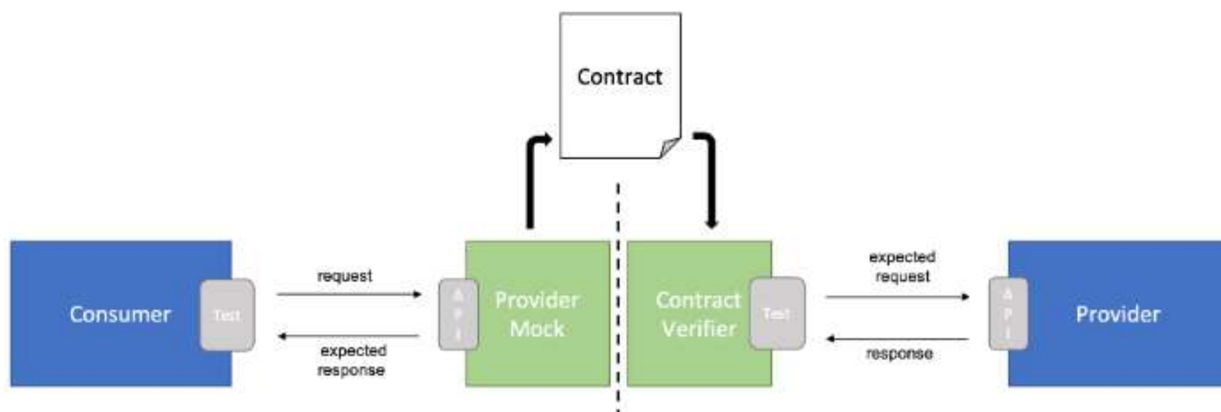
```
public class UsuarioControllerIntegrationTest extends
MainIntegrationTestConfig {

    private static HttpHeaders headers;
    private static HttpEntity<String> entity;
    @Autowired
    private TestRestTemplate testRestTemplate;
   ...
    @BeforeAll
    public static void init() {
        headers = new HttpHeaders();
        headers.setContentType(MediaType.APPLICATION_JSON);
        entity = new HttpEntity<>(null, headers);
        ...
    }
    @Test
    void saveUsersTest() {
       //Given
      WireMock.stubFor(WireMock.put(WireMock.urlEqualTo("/integratio
n/api/user/checkUsers"))
                .willReturn(WireMock.aResponse().withStatus(200)));
        HttpEntity<AddUsersDTO> entity = new
HttpEntity<>(addUsersDTO, headers);
        ResponseEntity<String> response = testRestTemplate.exchange(
                (createURLWithPort("usuario") + "/saveAll"),
HttpMethod.POST, entity, String.class);
        assertThat(response.getStatusCodeValue()).isEqualTo(200);
        assertNotNull(response.getBody());
        assertThat(response.getBody()).isEqualTo("true");
```

```
        }
}
```

5.2.3.2 Pruebas por contrato empleando stubs para la respuesta del productor. En este caso, en lugar de imitar la respuesta del proveedor, lo que se hace, en términos generales es, a partir de un contrato generado desde el proveedor, almacenar la respuesta definida dentro de un *stub*[31] que luego será tomado por el consumidor y utilizado cuando se haga el llamado a ese servicio desde la ejecución de la clase de pruebas; para entender mejor, un stub es un doble de prueba simple que proporciona respuestas predefinidas o codificadas a las llamadas realizadas por el código que se está probando. En la Figura 15 se observa el esquema de la forma en que funcionan los contract test dándoles este alcance.

Figura 15

Esquema del flujo de prueba por contrato usando stubs de verificación



Nota. Demostración gráfica del funcionamiento de pruebas por contrato.

[31] Syer, D. (2007, January 15). Spring Blog. Unit Testing with Stubs and Mocks. https://spring.io/blog/2007/01/15/unit-testing-with-stubs-and-mocks

Para implementar las pruebas de esta forma, se configuraron las dependencias requeridas y se escribieron, desde cada parte implicada, productor y consumidor, los respectivos contratos y pruebas respectivamente, adicionalmente, para hacer efectivo este tipo de test dentro del ciclo de integración continua, se agregaron configuraciones necesarias para ejecutar estas pruebas dentro del pipeline de despliegue de nuevo código en ambiente de pruebas, de esta forma, cada vez que un desarrollador desee integrar cambios a un microservicio, se hará la validación de que estos test de contrato sean exitosos, de lo contrario no se le permitirá desplegar esa nueva versión, hasta que sea corregida la falla y los test vuelvan a ser exitosos; específicamente, se creó y configuró un repositorio remoto en GitLab en el cual poder alojar los contratos generados al desplegar el proyecto que cumple el rol de productor con lo cual se logra hacer la integración con el proceso de CI/CD .

Para el consumidor:

- Se agregaron las dependencias que habilitan el uso de Contract Test:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-contract-wiremock</artifactId>
    <version>2.1.1.RELEASE</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-contract-stub-runner</artifactId>
    <version>2.1.1.RELEASE</version>
    <scope>test</scope>
</dependency>
```

- Dado que debe indicarse el stub desde el cual se tomará la respuesta del productor que haga match con la petición del consumidor, se agregó la dependencia del repositorio donde el proveedor dejaría alojado dichos contratos.

En el ejemplo que se muestra a continuación, los contratos definidos se encuentran en la carpeta stubs del repositorio local generado tras haber ejecutado los test en el proyecto proveedor, el cual fue uno de los microservicios empleados dentro de la etapa de indagación de este enfoque.

```
<dependency>
            <groupId>co.edu.uis</groupId>
            <artifactId>notifproj</artifactId>
            <classifier>stubs</classifier>
            <version>0.0.1-SNAPSHOT</version>
            <scope>test</scope>
 </dependency>
```

- Para cada caso con el que se realizaron pruebas, se configuraron los parámetros específicos como el puerto donde se emularía el recibimiento de los consumos hacia el microservicio secundario, el identificador de ese proyecto productor, y la anotación que activa el uso de los contratos a través de un stub. En la siguiente sección se relaciona un ejemplo de los casos implementados.

```
@AutoConfigureStubRunner(ids = { "co.edu.uis:notifproj:+:stubs:8090"
},stubsMode = REMOTE,
repositoryRoot= "url_repositorio",
properties = "git.branch=nombre_rama")
```

- Tras la ejecución del set de pruebas piloto, se comprobó que en caso de que la respuesta del contrato no sea la que el consumidor esperaba, se muestra el error, del mismo modo, si no encuentra un contrato definido para la petición que debe lanzar, o no cumple todas las condiciones, dentro de los logs se obtiene información del elemento que falla.

Hasta este punto se configuró el contract test del lado del servicio que consume el endpoint de otro servicio, ahora, lo que se hizo para agregar el contrato del lado del productor fue:

- Se añadió la siguiente dependencia que permite autogenerar la clase de test y verificar que los endpoints que se quieren probar cumplan lo definido en el contrato y la configuración para alojar el stub con los contratos generados en el repositorio remoto:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-contract-verifier</artifactId>
    <version>2.1.1.RELEASE</version>
    <scope>test</scope>
</dependency>
<configuration>
...
        <contractsRepositoryUrl>ruta_repositorio_remoto
        </contractsRepositoryUrl>
        <contractsProperties>
            <git.branch>rama_repositorio_remoto</git.branch>
        </contractsProperties>
</configuration>
<executions>
        <execution>
```

```
            <phase>package</phase>
            <goals>
                <goal>pushStubsToScm</goal>
            </goals>
        </execution>
</executions>
```

- De igual forma, se incluyó un plugin que habilita la ejecución de los contract test mediante maven. Además, se creó la clase principal de configuración que se relaciona dentro de la etiqueta *BaseClassForTests*, en ella se indica lo necesario para cargar el contexto de Spring Boot para la ejecución de las pruebas.

```
<plugin>
      <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-contract-maven-plugin</artifactId>
      <version>2.1.1.RELEASE</version>
   <extensions>true</extensions>
   <configuration>
         <baseClassForTests>
                        co.edu.uis.notifproj.BaseClass
         </baseClassForTests>
      </configuration>
</plugin>
```

- Llegados a este punto, se escribieron los contratos, estos pueden ser escritos en Groovy, Yaml o incluso otros lenguajes, sin embargo, para este caso, Yaml fue el seleccionado para escribir los contratos debido a que su sintaxis es más cómoda y sencilla que

Groovy. Tras haber definido una estructura sobre la ubicación de los contratos dentro del directorio del proyecto, se hizo la respectiva implementación haciendo uso de los distintos elementos parametrizables y de documentación ofrecidos por la herramienta, como un nombre para el test, una descripción, definición del método, la url, los valores del request y los valores de la respuesta, así mismo, se probó el uso de matchers con expresiones regulares que habilitan distintos valores posibles entrantes desde una petición y no se ligará a un valor en específico. A continuación se presenta un ejemplo de esta estructura:

```
description: "contract test enviar correo privado de mensaje predefinido"
request:
  method: POST
  url: "/api/email/send/msg"
  headers:
    Content-Type: application/json
    #idPerTo: "1"
    idUsrTo: "1"
    rolUsrTo: "1"
  body:
    identificador: "MENSAJE_HELLO"
    tipoCorreo: "ambos"
    sincrono: false
    idsDesTo: [1]
    aspTo: false
    tipoCorreoTo: null
  matchers:
    headers:
```

```
      #- key: idPerTo
        #regex: "^[0-9]+$"
      - key: idUsrTo
        regex: "^[0-9]+$"
      - key: rolUsrTo
        regex: "^[0-9]+$"
    body:
      - path: $.identificador
        type: by_regex
        predefined: non_blank
      response:
  headers:
    Content-Type: application/json
  body: "true"
  status: 200
```

- Una vez completados los pasos anteriores, se pasó a la ejecución y análisis de su comportamiento, validando que se hubiese hecho el proceso completo de generación de contratos, cargue al repositorio remoto de los mismos y verificación de cumplimiento de los mismos.
- A partir de las ejecuciones de pruebas con el set de experimentación que se planteó, se obtuvieron diferentes casos de test funcionales y superados; sin embargo, se pudo observar que no es tan amplia la documentación respecto a notación y sintaxis de los contratos, lo cual hace más demorado el tiempo de escritura de los test, aumentando la complejidad de la línea de aprendizaje de esta tecnología.

- Como beneficio, se obtiene un mayor control de que cada pieza nueva o modificación de código que se integre al software cumpla con las pruebas funcionales establecidas mediante los contratos.

5.3. Investigar sobre metodología TDD y posibles aportes al proyecto.

5.3.1 Objetivo

La metodología TDD es una de las más populares actualmente debido a los beneficios que aporta al desarrollo de software, al tener un proceso guiado por las pruebas desde su diseño, gran cantidad de recursos que se gastarían en identificación de hallazgos y soporte pueden ser optimizados. Aunque el proyecto de RSI ya cuenta con muchos componentes y funcionalidades desarrolladas, las cuales no se hicieron bajo el marco de esta metodología, vale la pena contemplar la existencia de esta teniendo en cuenta la adopción de la implementación de pruebas automatizadas en el ciclo de desarrollo de software para una posible adaptación parcial o completa en el desarrollo de funcionalidades futuras, por tanto, el objetivo de esta historia de usuario es realizar una revisión acerca de los principios y técnicas empleadas con este enfoque y hacer una reflexión del uso de esta metodología de desarrollo para su aplicación en el proyecto RSI.

5.3.2 Criterios de aceptación

- Hacer un análisis sobre las pautas de la metodología TDD que se puedan aplicar al desarrollo en el proyecto RSI, teniendo en cuenta las condiciones actuales de este.
- Plantear fundamentos relacionados con TDD que sirvan como base de la implementación de esta metodología dentro del proyecto y socializarlos con el equipo.
- Socializar los principios de TDD de los cuales se haga adopción dentro del proceso de desarrollo del proyecto.

5.3.3 Ejecución

Para desarrollar este punto, se investigó acerca de la metodología *Test-Driven Development* o TDD. En síntesis, el principio por el cual se guía TDD es que las pruebas sean planteadas desde el diseño del software, es decir, desde la escritura de las historias de usuario y que antes de pasar a implementar el código requerido para las nuevas funcionalidades, se escriban las pruebas que deberán ser superadas por dicho código, en caso de que no se pase alguna de las pruebas se procede a refactorizar hasta que estas sean exitosas. Sin embargo, adoptar esta metodología en su totalidad por un equipo de desarrollo tiene varios retos, como el hecho de que es complejo su aprendizaje, ya que puede reducir la productividad durante el período que se demore en superar la curva de aprendizaje.

Como aspecto relevante de TDD que se puede acoger en el planteamiento de las pruebas de integración en RSI que se hace en este proyecto es el hecho de que desde la escritura de las historias de usuario se contemplen las pruebas:

- Cuando el analista escriba una historia de usuario, debe identificar la complejidad de la funcionalidad que se esté solicitando y basándose en ello definir si se requiere aplicar pruebas unitarias, pruebas de integración o pruebas por contrato; esto puede realizarse de la mano de los desarrolladores asignados o del equipo en general en las sesiones de planeación que se realizan cada cierto tiempo.
- Hacer uso de las facilidades brindadas por el software de seguimiento de tareas en cuanto a la implementación de pruebas automatizadas en los desarrollos que se tengan pendientes por realizar, si se requiere o no, diferenciando entre los distintos tipos de pruebas que se deberían escribir dependiendo de las consideraciones presentadas en la evaluación de las historias de usuario.

Estas sugerencias para iniciar con el acercamiento a TDD dentro del proyecto RSI se dieron a conocer dentro de las capacitaciones realizadas al equipo, de esta forma ya todos los subequipos de desarrollo pueden tener una guía y de manera conjunta ir avanzando hacia mejores prácticas. Es importante tener en cuenta que la adopción de una nueva metodología de desarrollo debe ponerse a prueba en un caso como este, donde se ha seguido un flujo de trabajo guiado por SCRUM y ciertas prácticas que se han usado desde el inicio del proyecto o a medida que este ha avanzado; el cómo hacerlo es decisión del equipo general de RSI y las consideraciones que se tengan, de igual manera esto puede presentarse mediante diferentes alternativas como la designación de un equipo de desarrollo para adoptar la metodología y evaluar los resultados después de cierto tiempo (un sprint de desarrollo), o la adopción parcial de la TDD por parte de todos los equipos a la vez mientras se obtienen y evalúan resultados por parte de líderes y los equipos mismos teniendo como objetivo un incremento en el uso de la metodología.

5.4 Indagar sobre estrategias de aplicación de pruebas de software e identificación de necesidad de pruebas de integración.

5.4.1 Objetivo

Según los fundamentos teóricos sobre la aplicación de pruebas de software, las unitarias deben cubrir mayor cantidad de casos, ya que son más ligeras tanto en su escritura como en su ejecución, entre otras consideraciones; a medida que se escala hacia tipos de pruebas superiores como las de integración, se considera que deben ir disminuyendo en cuanto a cantidad, puesto que representan mayor complejidad en distintos aspectos (tiempos de ejecución, alcance y otros). Partiendo de lo anterior, al solicitar una nueva funcionalidad o módulo dentro de un software, lo ideal es someterse a un análisis para definir si dados sus casos de uso y características específicas requiere que le sean aplicadas pruebas de integración automatizadas, o, contrario a lo que se busca,

terminaría en un costo innecesario. El objetivo de esta historia de usuario es revisar qué estrategia puede tomarse y cómo se suele hacer la evaluación del software a la hora de decidir cómo aplicar pruebas automatizadas, tal que sirva de guía a la hora de determinar lo que precisa pruebas de integración y lo que no, teniendo presente que el carácter del software es cambiante en cada situación y que deben analizarse los requerimientos y particularidades de cada caso.

5.4.2 Criterios de aceptación

- Buscar estrategias relevantes que puedan tomarse como guía para identificar si una nueva funcionalidad requiere que le sean aplicadas pruebas de integración.
- A forma de documento informativo, hacer accesible esta información a todos los equipos de desarrollo del proyecto RSI para que puedan empezar a aplicarlo.
- Evitar el uso de métricas cuantitativas dentro de las pautas generadas, con el fin de prevenir limitaciones a las particularidades de cada funcionalidad o módulo que se evalúe con ayuda de esta guía.

5.4.3 Ejecución

Para esta historia de usuario se indagó sobre las diferentes técnicas que existen cuando se está en un proceso de implementación de pruebas automatizadas sobre un desarrollo nuevo, a futuro o ya realizado; adicional a esto, también se tuvo presente hacer una revisión sobre cómo suele evaluarse (desde la perspectiva individual de un desarrollador) una funcionalidad en cuanto a la necesidad de aplicar pruebas de integración u otro tipo respecto a lo que se tenga; esto último se realizó mediante la socialización de experiencias encontradas propias de otros programadores dentro del campo del desarrollo de software donde se incluya el uso de pruebas automatizadas con el fin de tener una mayor idea de cómo aproximar inicialmente la evaluación de las funcionalidades de los proyectos de RSI respecto a la implementación de pruebas.

5.4.3.1 Estrategias de aplicación de pruebas de integración y su adopción en el proyecto RSI. Al hablar de pruebas de integración es importante tener en cuenta al menos una definición general como un tipo de pruebas que verifican el correcto funcionamiento entre varios componentes de un sistema a la vez; a partir de esto, al consultar sobre estrategias para implementar este tipo de pruebas se pueden encontrar cuatro tipos comúnmente usados dependiendo de cómo se adapten mejor respecto a la complejidad del sistema o la etapa de desarrollo en que se encuentran los componentes del software a probar.

- **Big bang testing:** es un enfoque de implementación de pruebas de integración donde se toman todos los componentes ya desarrollados y probados individualmente procediendo a probarlos como uno solo; este método es usado usualmente cuando los componentes tienen cierto nivel de independencia y pueden probarse individualmente.
- **Bottom-up testing:** es una estrategia en la cual la aplicación de pruebas se realiza desde los componentes más independientes del sistema hacia los más dependientes; en una arquitectura por capas, se empieza a aplicar pruebas desde la capa inferior moviéndose gradualmente hasta la superior haciendo uso de *Drivers*; aquí se suele usar una combinación de técnicas de pruebas unitarias y pruebas de integración a medida que se evoluciona en el desarrollo.
- **Top-down testing:** a diferencia del enfoque Bottom-up, esta estrategia busca la implementación de pruebas desde los componentes de mayor nivel incrementando hasta llegar a los de menor nivel; usualmente el enfoque es guiado por el uso de *Stubs* para simular el comportamiento de otros componentes que no se han desarrollado o incluído en el alcance de las pruebas todavía.

- **Hybrid testing:** con esta estrategia se busca hacer uso de las técnicas de prueba incrementales como Top-down y Bottom-up de manera que se pruebe el sistema haciendo una combinación de estas guiado por el uso de *stubs* y *drivers* para simular el comportamiento de los componentes que no han sido desarrollados todavía.

El uso de una u otra estrategia de aplicación de pruebas de integración es una decisión que depende de los requerimientos del proyecto en el momento de implementar estas pruebas; cada una de estas estrategias tiene ventajas y desventajas dependiendo de la necesidad que se presente para evaluarlas, en la Tabla 3 se detallan algunas.

Tabla 3

Ventajas y desventajas de las distintas estrategias de implementación de pruebas de integración

Estrategia	Ventajas	Desventajas
Big bang testing	• Conveniente para sistemas pequeños. • Fácil de implementar, ya que se cuenta con todos los componentes disponibles. • Ahorra recursos al no realizar pruebas por cada componente.	• Mayor nivel de complejidad para identificar el origen de fallas. • Puede causar retrasos en las entregas al necesitar todos los componentes disponibles.
Bottom-up testing	• Mayor facilidad de localizar orígen de fallas. • No hay tiempos de espera	• La integración de componentes de menor nivel con los de mayor nivel puede

	para que se desarrollen otros componentes. ● No hay necesidad de simular otros componentes o saber detalles sobre el diseño estructural.	representar cierta complejidad y requiere una coordinación cuidadosa. ● Probar funcionalidades críticas que dependen de componentes de mayor nivel puede retrasarse por no estar listos.
Top-down testing	● Se puede hacer enfoque temprano en funcionalidades críticas que dependan de componentes de mayor nivel. ● Posibilidad de obtener un prototipo anticipado del sistema. ● Facilidad de localizar orígen de fallas.	● El uso de muchos *stubs* puede incrementar la complejidad del proceso de pruebas. ● Componentes de niveles más bajos pueden llegar a ser probados inadecuadamente.
Hybrid testing	● Conveniente para proyectos grandes con mucha dependencia de módulos. ● Optimiza recursos al	● Complejidad aumentada al tener muchos componentes o arquitectura de gran escala. ● Combinar diferentes

	balancear los esfuerzos de diferentes estrategias, probando funcionalidades críticas mientras se verifica una integración comprensiva del sistema. ● Representa mayor flexibilidad a la hora de probar los distintos componentes.	estrategias requiere un aumento en el esfuerzo de evaluación de componentes.

Para el caso de la arquitectura de los servicios backend del proyecto RSI y el estado en que se encuentran en cuanto a la implementación de pruebas de integración, inicialmente la mayoría de estas podrían ser guiadas por el enfoque *Big Bang Testing* teniendo en cuenta algunas consideraciones:

- Ya existe una cantidad considerable de servicios desarrollados y en funcionamiento.
- Se tiene soporte de pruebas unitarias en algunos de los servicios existentes.
- Los servicios existentes son sistemas pequeños al evaluarlos individualmente.
- Usar este enfoque para la aplicación de pruebas a nivel de funcionalidad individual y tomando como componentes las distintas capas que componen una funcionalidad (controlador, servicio, repositorio).

Para los casos en los que se presenta el consumo de otros servicios ya se cuenta con el soporte de las pruebas por contrato, lo cual supone una inclinación al enfoque *Top-down Testing*

en cuanto al uso de *stubs* y la simulación de respuestas obtenidas por la comunicación con estos servicios externos existentes o por desarrollarse. En concreto, puede hacerse un uso de la combinación de lo que ofrecen estas dos estrategias mencionadas, de manera que se pueda cumplir con las expectativas de dar un buen soporte a la verificación del software mediante pruebas de integración.

5.4.3.2 Evaluación de desarrollos para la aplicación de pruebas de integración. Independientemente de la estrategia de implementación de pruebas a usar, se debe tener en cuenta la evaluación individual de las funcionalidades a probar en cuanto a las características que se puedan identificar como la complejidad, tiempo estimado de desarrollo, servicios implicados y qué tan críticas pueden ser respecto al sistema en general. Luego de indagar sobre cómo aplicar de alguna manera puntual la implementación de pruebas de integración y otros tipos como unitarias respecto a una funcionalidad que se quiera evaluar, se encontró que no existe alguna serie de pautas establecidas o generalizadas, sino que estas suelen ser establecidas por los mismos desarrolladores o analistas de calidad a la hora de la evaluación.

Respecto a pruebas de integración como tal, existen distintos puntos de vista sobre cómo aplicarlas y aspectos como la cantidad que se suelen aplicar, si se hace siempre si se recurre a solo pruebas unitarias en algunos casos, lo cual está ligado a la percepción de lo que se considera como una prueba de cada tipo (algo que en algunos casos puede resultar muy ambiguo); al final la decisión de cómo abordar esta práctica puede terminar en manos del desarrollador designado y su experiencia y conocimientos teóricos-prácticos sobre la implementación de pruebas de software.

Teniendo en cuenta el estado actual de la implementación de pruebas de integración en los servicios de RSI, lo más ideal resulta ser que inicialmente la evaluación de los desarrollos en curso y nuevos se haga en conjunto entre líderes, desarrolladores y analistas de calidad por equipos

haciendo énfasis en los criterios que se consideren relevantes para cada funcionalidad y la forma que mejor se perciba una prueba de integración. Para desarrollos existentes que aún no cuenten con pruebas, lo ideal es recurrir a la documentación propia de cada uno de estos y realizar la evaluación basándose principalmente por los detalles incluidos como requerimientos, criterios de aceptación y fallas ya identificadas para determinar la necesidad de implementar pruebas de integración u otro tipo.

5.5. Implementar pruebas de integración en principales funcionalidades del módulo Core que lo requieran.

5.5.1 Objetivo

Dentro de los microservicios con los que cuenta el backend del proyecto RSI existe el módulo Core en el cual se encuentran varias funcionalidades principales y transversales al sistema en general, por esta razón, la implementación de pruebas integrales automatizadas tendrá inicio por las funcionalidades más relevantes de este módulo, que tras un debido análisis se identifique que requieren ser probadas con este tipo de pruebas. Dado que son funcionalidades que pueden llegar a afectar al sistema en general, es esencial tener la mayor calidad posible y de esta forma disminuir los errores, reprocesos, y mantenimiento generado por esos errores. El objetivo de esta historia de usuario es que las principales y más complejas funcionalidades del módulo Core sean probadas de forma integral y automatizada, garantizando siempre la no afectación a las bases de datos que no están dispuestas a alteraciones por este tipo de pruebas.

5.5.2 Criterios de aceptación

- Identificar las funcionalidades pertenecientes al microservicio Core que requieren pruebas de integración, haciendo su respectivo análisis con ayuda de las consideraciones expuestas en la historia de usuario inmediatamente anterior.

- Escribir las pruebas de integración necesarias, empleando la tecnología y forma definida tras realizar el proceso de investigación y evaluación de alternativas.
- Incluir la ejecución de las pruebas de integración automatizadas dentro del proceso CI/CD bajo el cual se agrega toda nueva unidad de código a cada uno de los ambientes del proyecto.
- Realizar pruebas de *dev test* por el par desarrollador para validar que no haya sido alterado el comportamiento del microservicio y que la ejecución de las pruebas en el ambiente de desarrollo *predev* se lleve a cabo de forma correcta.

5.5.3 Ejecución

Para llevar a cabo el desarrollo de esta historia de usuario, inicialmente se hizo una revisión general de los distintos módulos y funcionalidades que contiene el servicio de Core, posteriormente, siguiendo como referencia las consideraciones ya expuestas para aplicar pruebas de integración, se identificaron aquellos endpoints o procesos conformados por conjuntos de endpoints que clasificaron como candidatos a ser probados, adicionalmente se determinaron aquellos casos que contenían consumos de un microservicio externo, esto con el fin de aplicar también las correspondientes pruebas por contrato.

Cabe destacar también que, aunque dentro del proyecto RSI ya se contaba con la estrategia de implementación de pruebas unitarias, varias de las clases de servicio de este proyecto carecían de estas, por tanto, por buenas prácticas de desarrollo y siguiendo la premisa de la jerarquía de pruebas, antes de pasar a la implementación de las pruebas de integración y por contrato en las funcionalidades identificadas, se aplicaron varias pruebas unitarias de las que estaban faltantes a nivel de capa de servicio.

Seguidamente, se hizo la implementación de las pruebas de integración en las funcionalidades identificadas, las cuales corresponden a aquellas consideradas de mayor relevancia y criticidad dentro de este componente transversal partiendo de las revisiones de historias de usuario asociadas y fallos documentados luego de haber terminado los desarrollos, algunos de estos son el manejo y administración de usuarios, rutas de navegación, accesos y permisos a los distintos módulos del sistema, carga y descarga de documentos en distintos formatos, entre otros. En las Tablas 4 y 5 se muestran los resultados obtenidos en cuanto al soporte total brindado para la implementación de pruebas y el cubrimiento que se logró con estas pruebas por cada uno de los paquetes del microservicio, teniendo en general un total alcanzado de 19%.

Tabla 4

Cantidad total de funcionalidades probadas y pruebas de integración.

	Cantidad total
Funcionalidades probadas	18
Pruebas implementadas	37

Tabla 5

Resultados obtenidos de cobertura de código con implementación de pruebas de integración automatizadas en funcionalidades específicas de microservicio de Core

Paquete	**% Cobertura**
Controller	22 %
Service	29 %
Repository	8 %
Model	14 %
DTO	18 %

Mapper	11 %

Por último, dado que las pruebas por contrato involucran también al microservicio productor, se escribieron los contratos en los proyectos backend de los servicios correspondientes, ya que desde Core se consumen algunos, de esta forma se completaron las pruebas por contrato para la interacción entre estos microservicios, donde se hace uso de funcionalidades como envío de correos, verificación de información con otros sistemas y demás operaciones. Para poder hacer la inclusión de estas pruebas por contrato en el proceso de CI, como se documentó en el apartado **5.2.3.2**, se requería un repositorio remoto en el cual almacenar los recursos necesarios, por tanto, se creó un nuevo repositorio en GitLab con acceso privado solo para el equipo de RSI y el almacenamiento de pruebas por contrato, allí se cargan los *stubs* con los contratos generados por los microservicios productores y pueden ser accedidos por los microservicios consumidores que los requieran. En la Tabla 6 se presentan los datos específicos de cantidad de pruebas de contrato escritas y funcionalidades cubiertas.

Tabla 6

Cantidad de pruebas por contrato implementadas

Criterio	**Cantidad total**
Pruebas por contrato en Core	6
Funcionalidades con pruebas por contrato	5
Contratos escritos en servicios de RSI	7
Servicios de RSI consumidos por Core	3

Cada set de pruebas escrito fue ejecutado y se validó que se superaran exitosamente, en algunos casos fue necesario realizar refactorización de código de algunas funcionalidades en las que se identificaron fallas durante el proceso de pruebas para que finalmente pudieran dar éxito en

sus pruebas automatizadas. Todo esto se realizó siempre bajo las consideraciones tomadas sobre cómo implementar las pruebas, garantizando la integridad de las bases de datos de ambientes especiales.

5.6. Documentar el proceso de implementación y ejecución de pruebas de integración automatizadas dentro de RSI.

5.6.1 Objetivo

Cada proceso, estructura, patrón o modelo que se haya definido para ser seguido como norma dentro de un proyecto de desarrollo de software debe ser documentando conscientemente para que pueda prevalecer a lo largo del tiempo, o pueda ser entendido a futuro y realizar sobre ello los ajustes que se ameriten, esto hace parte de las buenas prácticas del desarrollo. Por tal motivo, al haberse definido una metodología con ciertas especificaciones y haciendo uso de determinadas tecnologías para implementar y ejecutar las pruebas de integración automatizadas sobre los microservicios del proyecto RSI, es necesario que sea documentada de forma clara, detallada y entendible para los demás integrantes del equipo dicha forma de realizar este tipo de pruebas.

El propósito final de esta historia de usuario es que cualquier integrante del proyecto RSI que se vea involucrado en el proceso de aplicación y ejecución de pruebas, pueda recurrir a la documentación de las herramientas disponibles y la manera establecida de cómo llevar a cabo esa implementación dentro de los microservicios del proyecto, para que realice ese procedimiento según las consideraciones definidas y el trabajo realizado en este proyecto a manera de ejemplo y punto de partida, para almacenar este tipo de documentación se cuenta con la Wiki del proyecto RSI, donde cada integrante tiene acceso y podrá consultar la información en este sitio cuando le sea necesario.

5.6.2 Criterios de aceptación

- Escribir en la wiki la documentación sobre pruebas de integración automatizadas para microservicios del proyecto RSI.
- Mencionar dentro de la documentación las especificaciones técnicas y el paso a paso de cómo se puede realizar el proceso de implementación de estas pruebas.
- Dejar claras consideraciones y puntos relevantes que no deberían pasarse por alto a la hora de escribir y ejecutar las pruebas.
- Considerar tanto pruebas de integración básicas como aquellas que hacen uso de herramientas externas o que involucran la comunicación entre microservicios.

5.6.3 Ejecución

Para llevar a cabo esta historia de usuario, se documentó dentro de la Wiki todo lo relacionado con pruebas de integración, pruebas por contrato, instrucciones para implementar pruebas con herramientas como SpringBootTest y Test Containers, paso a paso para implementar pruebas por contrato con Spring Cloud Contract, y la guía o consideraciones que ayuden como referencia para la definición a nivel de historia de usuario sobre temas como qué amerita pruebas y qué no o cómo enfocar la implementación de estas.

En cada uno de los temas documentados se especificaron anotaciones y dependencias empleadas, se dejaron ejemplos de implementaciones completas y evidencias de cómo deberían verse los resultados, todo esto con el fin de facilitar a los desarrolladores el entendimiento de lo planteado, y reducir la complejidad en la escritura de las pruebas al dejar claros y definidos los puntos principales.

En total se produjeron tres artículos de documentación, a los cuales cualquier integrante del proyecto RSI que cuente con los permisos de acceso la Wiki podrá ingresar y revisar en cualquier momento:

1. Documentación sobre test de integración y guía para su implementación en Spring Boot.
2. Documentación sobre contract test y guía para su implementación en Spring Boot.
3. Documentación de la guía de referencia y consideraciones para determinar la evaluación de funcionalidades en cuanto la aplicación de pruebas de integración.

5.7. Brindar capacitaciones a cada equipo del proyecto RSI sobre la implementación de pruebas de integración automatizadas en la capa backend.

5.7.1 Objetivo

Es importante que quede clara la metodología que se propone emplear para la escritura y ejecución de pruebas de integración automatizadas entre todos los involucrados del proyecto RSI dado que en principio la mayor parte de estos integrantes no tienen este conocimiento; lo anterior implica la necesidad de que sean capacitados al respecto, por tanto, el objetivo de esta historia de usuario es brindar dicha capacitación a cada equipo de desarrollo del proyecto a manera de introducción a la implementación de pruebas automatizadas, resolver dudas y de esta forma garantizar la adopción de una nueva práctica para pruebas de software.

5.7.2 Criterios de aceptación

- Preparar material didáctico para presentar en las capacitaciones, que facilite el entendimiento del tema.
- Brindar las capacitaciones a los equipos del proyecto RSI involucrados con el tema.

- Dejar el material, tanto presentaciones como capacitaciones, a disposición de todos los integrantes del proyecto para que puedan ser consultados en cualquier momento que se requiera.

5.7.3 Ejecución

Para llevar a cabo el objetivo de esta historia de usuario, se creó material explicativo, expositivo e instructivo sobre pruebas de integración automatizados en microservicios backend con Spring Boot y se realizó una sesión de capacitación para líderes analistas de equipos y desarrolladores backend del proyecto RSI, en la cual se dio a conocer el tema y el planteamiento de la metodología sobre implementación de este tipo de pruebas; también se compartió la experiencia que se tuvo en la realización de este proyecto en cuanto a nuevos aprendizajes a la hora de trabajar con pruebas automatizadas y las consideraciones personales desde la perspectiva de desarrolladores de software al realizar estas tareas, se aclararon dudas y se compartió así mismo el material con la documentación pertinente de la wiki del proyecto RSI.

6. Resultados

Tras superar la etapa de investigación sobre fundamentos y herramientas para pruebas de integración, se generó el planteamiento de la forma de implementar dichas pruebas automatizadas en la capa backend de los aplicativos desarrollados por el proyecto RSI, así mismo como las consideraciones para identificar la necesidad de aplicar pruebas en una funcionalidad a nivel de historia de usuario. A partir de lo anterior, se analizaron las funcionalidades del módulo central Core, se identificaron las que requerían pruebas de integración y se logró implementar en total 37 pruebas automatizadas, con las cuales se logró el cubrimiento de 13 funcionalidades del módulo

como se presentó en las Tablas 4 y 5 de este documento; también se logró implementar el uso de pruebas de integración guiadas por contratos respecto a los diferentes servicios de RSI que se consumen desde el módulo Core como se sintetizó en la Tabla 6. Como se esperaba, el tiempo de ejecución de los *pipelines* para despliegue en ambiente *predev* aumentó al tener que ejecutar las pruebas de integración y generar los reportes de resultados, dicho tiempo adicional inducido por las pruebas implementados fue de aproximadamente 90 segundos lo cual representa un buen uso de los recursos y configuración del servicio a la hora de la ejecución de las pruebas para evitar que se obtengan tiempos muy prolongados y generen retrasos a la hora de realizar nuevos despliegues del servicio.

7. Conclusiones

Al implementar pruebas de integración dentro de los aplicativos backend del proyecto RSI e integrarlo con el proceso de integración continua (CI/CD) se logró tener un mayor control sobre la calidad del software desarrollado, ya que siempre que se realice un incremento se garantizará la aprobación exitosa de las pruebas automatizadas, de lo contrario no se permitirá realizar nuevos despliegues; de esta forma se optimizan también tiempos dado que no será necesario que un integrante de QA realice pruebas de regresión manualmente en todo momento para poder observar que todas las funcionalidades se mantengan de forma correcta, además, si llegase a surgir un error en las pruebas, inmediatamente se detecta y se genera el reporte con detalles sobre las causas para darle así pronta solución, lo cual evita el costo que generaría luego de un período de haber realizado el despliegue la detección de error, identificación de causas y gestión del mantenimiento. Adicionalmente, el hecho de tener a disposición una herramienta visual que permita llevar el

seguimiento de la cobertura por parte de las pruebas en conjunto con los reportes de pruebas ejecutadas y superadas con éxito facilita el reconocimiento de qué tan asegurado está el software en estos términos e incluso permite establecer metas sobre las métricas esperadas que se deberían cumplir en cada uno de los módulos de software desarrollados.

La etapa de investigación y evaluación de las distintas herramientas disponibles para implementar este tipo de pruebas automatizadas fue crucial, ya que de esta forma se pudo generar el planteamiento óptimo y que se adecua a las características de la arquitectura del sistema desarrollado por RSI.

Por otro lado, en cuanto a la metodología TDD se puede decir que esta resulta ser de gran utilidad al aplicarla dentro de un equipo de desarrollo de software, sin embargo, dado su alto costo de aprendizaje y adopción, lo ideal dentro de RSI es paulatinamente ir acogiendo algunos de sus aspectos para, como se consideró en este proyecto, acompañar el diseño de historias de usuario con análisis sobre la necesidad de pruebas automatizadas.

En síntesis, tener un proceso adecuado sobre aplicación de pruebas de integración automatizadas, como el que fue resultado de este proyecto, aporta confiabilidad, calidad, mantenibilidad y facilidad de soporte al software que se elabora, y, aunque tome un poco más de recursos implementando estas pruebas, lo compensa los que se optimizan en labores de análisis de calidad, monitoreo y solución de errores generados en la ausencia de dicho proceso.

8. Trabajo Futuro

El hecho de que un proceso de desarrollo de software se encuentre respaldado con pruebas automatizadas le aporta beneficios significativos, aún más, cuando se trata de pruebas de

integración; a lo largo de este proyecto se pudo establecer una metodología para realizar la implementación de estas pruebas a nivel de backend, no obstante, aún se puede reforzar y extender, por ejemplo, incluyendo herramientas de versionamiento de base de datos que faciliten el uso de TestContainers y de esa forma tener un entorno de pruebas totalmente aislado y mejor controlado. Por otro lado, teniendo en cuenta la arquitectura general del proyecto RSI donde el backend está conectado con una capa de frontend, vale la pena considerar la posibilidad de implementar pruebas de integración también desde esta última e incluso darle un alcance hasta nivel de interfaz de usuario, de esta forma se tiene un mejor control y el equipo de QA podría automatizar aún más parte de sus labores; todo esto, se refleja en mayor calidad del software desarrollado y optimización de recursos.

Referencias Bibliográficas

Cohn, M. (2010). *Succeeding with agile: software development using Scrum*. Pearson Education.

Fernández Carrasco, Oscar M, García León, Delba, Beltrán Benavides, Alfa. (1995). Un enfoque actual sobre la calidad del software. ACIMED, 3(3), 40-42. Recuperado en 12 de abril de 2024, de http://scielo.sld.cu/scielo.php?script=sci_arttext&pid=S1024-94351995000300005&lng=es&tlng=es.

Martin, R. C. (2017). Clean Architecture: A Craftsman's Guide to Software Structure and Design, page 28. Prentice Hall.

Moe, M. M. (2019, May 13). *Comparative study of test-driven development (TDD), behavior-driven development (BDD) and acceptance test–driven development (ATDD)*. International Journal of Trend in Scientific Research and Development. https://www.ijtsrd.com/computer-science/other/23698/comparative-study-of-test-driven-development-tdd-behavior-driven-development-bdd-and--acceptance-test%E2%80%93driven-development-atdd/myint-myint-moe.

TechMagic. (2018, 21 junio). Get started with behavior driven development - TechMagic - Medium. Medium. https://medium.com/@TechMagic/get-started-with-behavior-driven-development-ecdca40e827b.

Vocke, H. (2018, February 26). The practical test pyramid. martinfowler.com. https://martinfowler.com/articles/practical-test-pyramid.html#TheTestPyramid.

Wolff, E. (2016). Microservices: flexible software architecture. Addison-Wesley Professional.

Baeldung. (2024, January 8). Testing in Spring Boot. Baeldung. https://www.baeldung.com/spring-boot-testing.

Zilberfeld, G. (2012, January 12). Design for testability – the true story. InfoQ. https://www.infoq.com/articles/Testability/.