UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

Modernización de aplicación bancaria móvil

TESIS

Que para obtener el título de

Ingeniera en Computación

P R E S E N T A

Laura Fabiola Aguilar Plascencia

DIRECTOR DE TESIS

Ing. Alberto Templos Carbajal

Ciudad Universitaria, Cd. Mx., 2024

Índice

Índice de figuras

1. Objetivo

Migración de una aplicación bancaria móvil para dispositivos *Android* y *iOS* desde una arquitectura monolítica a una arquitectura de microservicios

2. Introducción

Anteriormente las filas de las instituciones bancarias solían ser extensas, había que pasar más de media hora esperando un turno para realizar algún pago, retirar dinero, solicitar atención a cliente, etc.

"*Largas filas en los bancos, transacciones lentas y difícil acceso a un crédito o cuenta bancaria han sido por muchos años los principales dolores de cabeza de los ciudadanos del mundo cuando se acercan al sector financiero tradicional." [Semana]*[1]

Todo ese tiempo invertido ahora puede ser aprovechado por los usuarios ya que actualmente estos procesos son más sencillos y seguros de realizar haciendo uso de una banca móvil.

Una banca móvil es una aplicación desarrollada para instituciones bancarias con el fin de proporcionar los servicios que el banco ya ofrece en sus sucursales como pagos de servicios, transferencias bancarias, sistemas de inversión, entre otros movimientos que el banco permita ya que cada uno puede llegar a tener servicios distintos, sin embargo, todos tienen la ventaja de brindar mayor accesibilidad a la información de cada usuario sin consumir demasiado tiempo como lo era anteriormente ya que este recurso es totalmente digital, además su uso puede llegar a ser más seguro si se utiliza de forma responsable y consciente.

[1] Semana. (2020, October 20). ¿Cómo funcionan las apps bancarias y qué ventajas tienen para su economía? Revista Semana. https://www.semana.com/economia/articulo/como-funcionan-las-apps-bancarias-y-que-ventajas-tienen-para-su-economia/202003/

Dentro del desarrollo de cualquier aplicación se establece un ciclo con distintas etapas de acuerdo a la forma de trabajo, como etapas generales se tiene el análisis de requerimientos, diseño de la arquitectura, diseño de la apariencia de la aplicación, desarrollo, pruebas de funcionamiento y salida a producción. Para llevar a cabo estas etapas se requiere de profesionales encargados de la parte del diseño, análisis, desarrollo *frontend*, desarrollo *backend*, realización de pruebas, etc. Es importante mencionar que estas etapas pueden repetirse más de una vez hasta llegar al resultado esperado

El proyecto de migración corresponde al desarrollo de *backend* que es el encargado de implementar la lógica de la aplicación del lado del servidor, este desarrollo es lo que hace que la aplicación funcione y realice los movimientos y transacciones correctamente ya que desde el *backend* se realizan validaciones de los datos recopilados del usuario u obtenidos dentro de consultas a servicios externos o internos mediante *API´s*, también se tiene acceso a gestores de bases de datos donde se almacena y se realizan transacciones de datos, entre otras actividades que no son visibles al usuario.

“El desarrollador backend es el profesional responsable de diseñar, implementar y mantener la lógica del lado del servidor de una aplicación. Trabajan detrás de escena y garantizan que los datos, servicios y funciones del servidor se entreguen al usuario de manera eficiente y segura .” [Hireline][2]

[2] Perfil de Desarrollador Backend. (s/f). Hireline. Recuperado el 15 de marzo de 2024, de https://hireline.io/mx/enciclopedia-de-perfiles-de-tecnologia/desarrollador-backend

3. Marco Teórico

Para poder comprender mejor los términos que se usarán a lo largo del texto se dará una explicación de los temas necesarios para entender mejor las actividades realizadas.

3.1 Diagrama de secuencia

"Los diagramas de secuencia son una solución de modelado dinámico popular en UML porque se centran específicamente en líneas de vida o en los procesos y objetos que coexisten simultáneamente, y los mensajes intercambiados entre ellos para ejecutar una función antes de que la línea de vida termine." [Lucidchart][3]

Un ejemplo sería una conexión a la base de datos, para el caso de la aplicación móvil representamos también en qué momento se ejecuta un servicio, consultas a servicios externos, interacción del usuario, etc.

Esta representación nos ayuda a identificar flujos de operaciones, fallas en la lógica de implementación y también funciona como método de consulta cuando se tiene un código heredado.

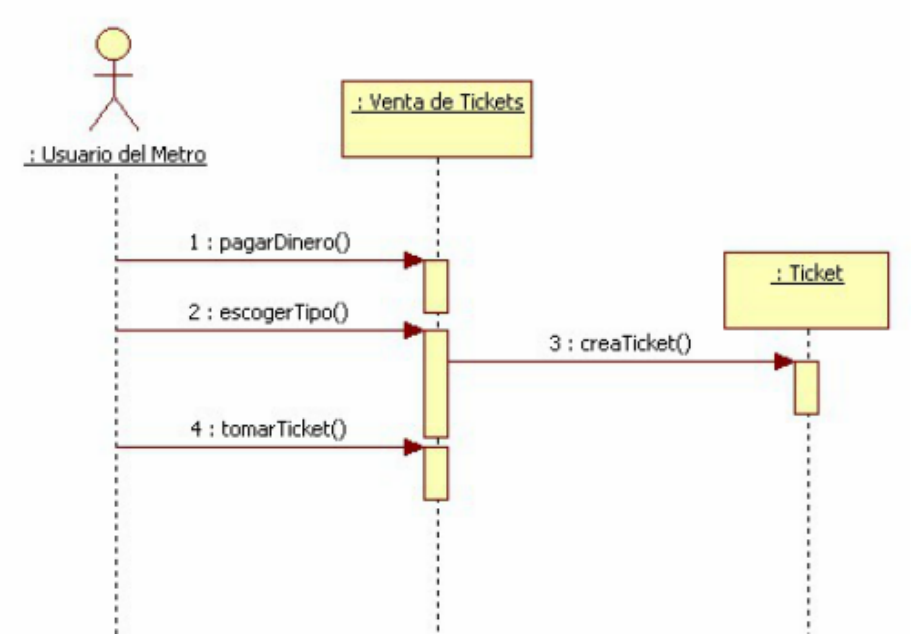
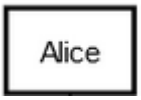
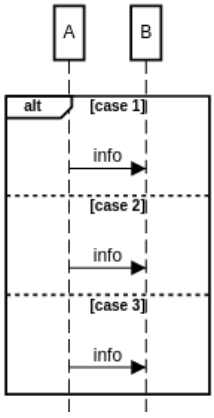


Figura 1 - Simbología básica de un diagrama de secuencia

[3] Tutorial de diagrama de secuencia UML. (n.d.). Lucidchart. Retrieved January 14, 2023, from https://www.lucidchart.com/pages/es/diagrama-de-secuencia

Símbolo	Descripción
	Representan las entidades que interactúan dentro de un proceso.
	Representa la interacción del usuario con la aplicación.
	Indica que se va a realizar una petición o habrá alguna acción a realizar sobre la entidad destino
	Indica que hay una respuesta por parte de la entidad
	Son bloques que indican el tiempo en que una entidad está activa
	De esta manera se representan los ciclos indicando cada decisión y las acciones que se realizan en cada uno de ellos.

3.2 Programación orientada a objetos

“La programación orientada a objetos es un modelo de programación en el que el diseño de software se organiza alrededor de datos u objetos, en vez de usar funciones

y lógica. Se enfoca en los objetos que los programadores necesitan manipular, en lugar de centrarse en la lógica necesaria para esa manipulación. " [Universidad Europea][4]

El paradigma de la programación orientada a objetos se basa en cuatro pilares fundamentales: abstracción, herencia, encapsulamiento y polimorfismo.

3.2.1 Abstracción: Es el nivel de detalle que se elige representar mediante objetos definidos en clases.

Por ejemplo si se quisiera representar una carta se puede identificar que la carta contiene un destinatario, un remitente y un texto. Pero también contiene acciones como por ejemplo: enviar, entregar, recibir, etc. La definición de acciones y elementos de la carta es lo que llamamos abstracción, y cada elemento se puede detallar según se requiera.

Tomando de ejemplo el destinatario se puede dividir en nombre, dirección, código postal,etc de esta forma se realiza un mayor nivel de abstracción.

3.2.2 Herencia: La herencia hace referencia a la creación de una clase basada en otra ya existente.
Por ejemplo se tiene una clase llamada Automóvil, esta clase tiene como métodos: *avanzar()*, *frenar()*, *reversa()*. Ahora se necesita también representar un automóvil deportivo por lo que se necesita generar una clase llamada Deportivo que haga las mismas acciones que automóvil y además tenga un método que indique que abre el descapotado. Para estos casos donde una clase puede implementar métodos que están en otra clase utilizamos el concepto de herencia para reutilizar el código que ya está escrito y únicamente usar los métodos y atributos que necesitamos, en java se utiliza la palabra reservada *extends* para indicar este concepto.

[4] Programación orientada a objetos. (2022, August 24). Universidad Europea. https://universidadeuropea.com/blog/programacion-orientada-objetos/

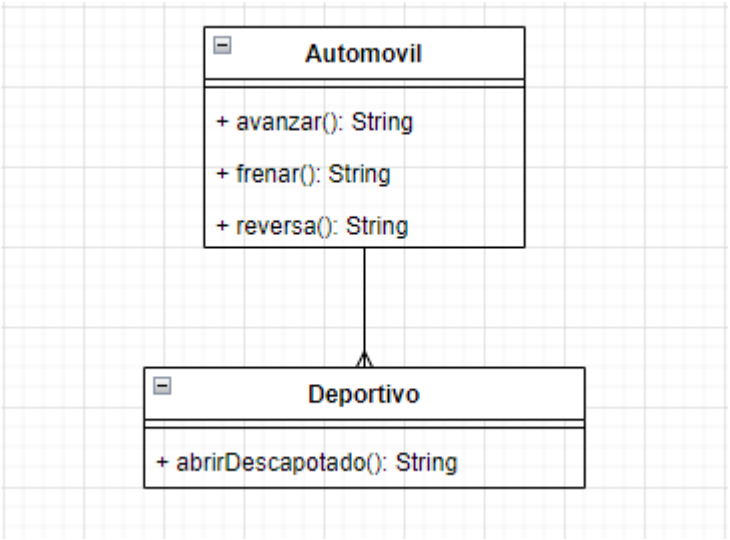


Figura 2 - Representación de la herencia de clases

3.2.3 Encapsulamiento: Se refiere al nivel de acceso que hay en una clase para poder consultar y/o modificar un atributo o método.

En java existen modificadores de acceso que definen desde dónde se puede acceder a los métodos o atributos de una clase, se utilizan para brindar seguridad y representar mejor cada objeto.

Nivel de acceso	*Private*	*Default*	*Protected*	*Public*
Misma clase	Si	Si	Si	Si
Subclase en el mismo paquete	No	Si	Si	Si
Clase en el mismo paquete	No	Si	Si	Si
Subclase en distinto paquete	No	No	Si	Si
Clase en distinto paquete	No	No	No	Si

En la tabla anterior se muestra el nivel de acceso de cada palabra reservada, por ejemplo al definir la clase *Persona* con los atributos: *nombre*, *edad*, *fecha de nacimiento*, *color de cabello*, etc hay atributos que por nada del mundo pueden modificarse como por ejemplo la *edad* o la *fecha de nacimiento* por lo que se les tendría que colocar el modificador de acceso *private* como una restricción, sin embargo

el *color de cabello* si puede ser cambiado por un estilista, un familiar, un amigo e incluso él mismo entonces a este atributo se le asignaría otro modificador como puede ser *protected.*

Cuando definimos una restricción a los atributos o métodos estamos encapsulando ese elemento para tener un mayor control de los objetos y cómo interactúan entre sí para que sea coherente y ordenado.

3.2.4 Polimorfismo: Es la capacidad de respuesta a un método que está implementado de varias maneras, esto también puede llamarse *sobrecarga de métodos* ya que un mismo método puede definirse con un número distinto de parámetros, distinto orden o distinto tipo de parámetros.
Por ejemplo, En una clase llamada Calculadora se pueden definir los siguientes métodos:

1. *private int* suma(int a, int b)
2. *private double* suma(double a, double b)

Como se puede ver ambos métodos se llaman igual y ambos reciben dos parámetros, sin embargo, cambia el tipo de parámetros y de retorno. La manera en la que se define cuál método usar dependerá del tipo de parámetros que se le manden al método sumar() y al tener más de una implementación tendrá mayor capacidad de respuesta.

La programación orientada a objetos brinda distintas ventajas como la reutilización de código por medio de clases heredadas, si se analizan bien los objetos que se requieran representar y los que ya están definidos es posible optimizar su desarrollo.

Al ser desarrollado en clases es más fácil realizar modificaciones, agregar o borrar elementos, también favorece la detección de errores y optimizar tiempo ya que los objetos pueden irse trabajando de forma paralela sin que existan errores.

3.3 SQL

“El lenguaje de consulta estructurado es un lenguaje para poder almacenar y consultar información en una base de datos.” [*Amazon*][5] La manera en la se organiza la información es dentro de tablas que están conectadas entre sí de manera lógica, los registros dentro de cada tabla son identificados por una *primary key* que es única, que al hacer uso de esta *key* se pueden realizar búsquedas específicas de la información almacenada en la base de datos.

“Las sentencias SQL se dividen en dos categorías: lenguaje de definición de datos (DDL) y lenguaje de manipulación de datos (DML). Las sentencias DDL se utilizan para describir una base de datos, para definir su estructura, para crear sus objetos y para crear los subobjetos de la tabla.” [IBM][6]

Dentro de sus sentencias están la siguientes:

- *CREATE*: Permite la creación de tablas.
- *ALTER*: Permite modificar la estructura de una tabla ya sea añadir o eliminar columnas etc.
- *DROP*: Eliminación de objetos dentro de la base de datos.

“Las sentencias DML se utilizan para controlar la información contenida en la base de datos. ” [*IBM*]

A continuación se describen algunas sentencias DML:

- *SELECT*: muestra información sobre los datos almacenados en la base de datos.
- *INSERT*: Inserta filas en una tabla.
- *UPDATE*: Actualiza información de una tabla.
- *DELETE*: Borra filas de una tabla.

[5] (N.d.). Amazon.com. Retrieved January 14, 2023, from https://aws.amazon.com/es/what-is/sql/
[6] IBM Documentation. (2023, abril 25). Ibm.com. https://www.ibm.com/docs/es/idr/11.3.3?topic=console-replicating-data-definition-language-ddl-changes

3.4 Ambientes de desarrollo

El ambiente de desarrollo es donde se despliegan los servicios, funcionalidades y diseños que se encuentran en constante modificación y que no han sido probados en su totalidad, dentro de este ambiente se realizan pruebas de lógica de servicios y funcionalidad por los mismos desarrolladores para realizar correcciones y mejoras que se consideren convenientes.

Este ambiente no suele ser muy estable ya que normalmente se realizan múltiples cambios, esta situación es insegura para el proyecto ya que no se garantiza un óptimo funcionamiento por lo que se necesitan otros ambientes dedicados a actividades específicas para brindar seguridad de la información y estabilidad del proyecto.

3.5 Ambientes de prueba *QA*

Al ambiente de pruebas se le denomina *QA* que significa *Quality Assurance* en este entorno se realizan pruebas que garantizan que el *software* previamente elaborado cumple con los requerimientos de seguridad, disponibilidad, rendimiento y además de que funcione conforme la documentación previa.

“Las pruebas son escritas desde la perspectiva del usuario y deben probar el sistema de una forma lo más cercana posible a producción. Las pruebas deben estar basadas alrededor de escenarios predefinidos típicos del proceso de negocio.” [necastro.blogspot] [7]

Para comenzar a probar en el entorno de *QA* el *software* debió haber sido probado en el ambiente de desarrollo varias veces para después realizar el despliegue en *QA* donde se realizan pruebas dedicadas a la calidad y además se revisa exhaustivamente el funcionamiento.

Este ambiente es más estable que el de desarrollo ya que en él se encuentra únicamente código que se estima que funciona correctamente, sin embargo, es posible que se encuentren con algunos errores y esto es normal ya que anteriormente no se

[7] Castro, N. (2010, November 22). QAT: Quality assurance test / UAT: User acceptance testing. Blogspot.com. http://necastro.blogspot.com/2010/11/qat-quality-assurance-test-uat-user.html

realizan tantas pruebas como en este entorno dedicado a eso. El *software* puede ser revisado más de una vez hasta tener esa seguridad de que cumple con todos los parámetros de calidad.

Cuando se termina de probar una funcionalidad en *QA* entonces se entiende que el desarrollo está listo para ser liberado a producción y ponerse en marcha.

3.6 Ambiente productivo

El ambiente productivo es aquel que se da a conocer a los usuarios para su uso, sería el producto final de un requerimiento. Aquí se realiza un seguimiento del *software* para garantizar una buena experiencia de usuario, es decir, si se encuentra una falla durante el monitoreo esta se reporta para que sea solucionada lo más pronto posible.

Este ambiente debe ser totalmente estable para que no existan momentos en los que un usuario encuentre al sistema caído, sin embargo, esto no siempre se cumple en su totalidad ya que puede ser afectado por diversos factores como por ejemplo recibir múltiples peticiones puede saturar al servidor y causar retardo en las respuesta del servidor.

3.7 Arquitectura monolítica

Este tipo de arquitectura es un modelo de desarrollo que se encuentra encapsulado en una sola capa, por lo que se suele asociar con una capa grande ya que contiene toda la lógica en un mismo lugar.
Dentro de sus características se tiene que:

- Las funcionalidades comparten los mismos recursos y la memoria.
- Se tiene un solo servidor.
- Todo está escrito en un mismo código.
- Se utiliza una sola base de datos para todo el código.

Figura 3 - Arquitectura monolítica

Los beneficios que tiene el uso de está arquitectura son:

- Es simple y autónomo
- Fácil de testear
- Diseño confiable
- Fácil actualización y mantenimiento

Pero también tiene desventajas, entre ellas están las siguientes:

- Está anclado a una sola tecnología
- Tiene complicaciones para escalarse
- Muy complejo el trabajo colaborativo
- Un cambio representa un cambio en la versión global

3.8 Arquitectura de microservicios

La arquitectura de microservicios consiste en crear pequeños componentes de software que realizan una sola tarea, la hacen bien y son totalmente autosuficientes.

De entre las características más relevantes se encuentran:

- Tienen una única responsabilidad.
- Están totalmente encapsulados.
- Son autónomos.

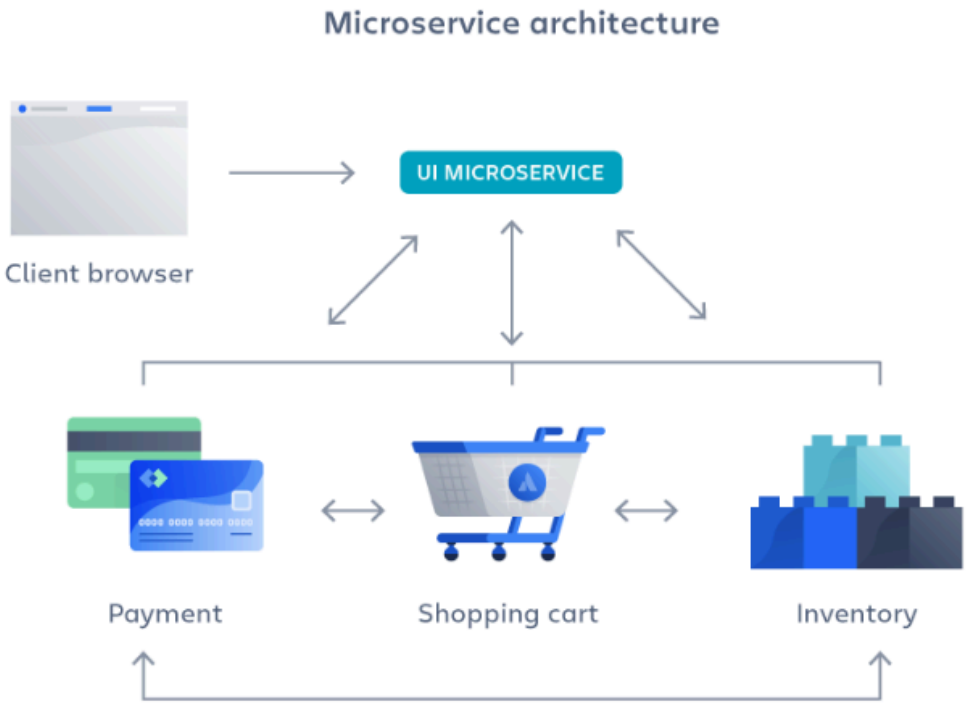


Figura 4 - Arquitectura de microservicios

Este tipo de arquitectura brinda los siguientes beneficios:

- Son más ágiles
- Tienen escalado flexible
- Se tiene libertad tecnológica
- Se puede tener código reutilizable

Mientras que tiene algunas desventajas como:

- Es complejo de monitorear
- Se requiere madurez por parte del equipo de desarrollo

4. Definición del problema

Se busca realizar una modernización de la aplicación móvil de una institución bancaria con el fin de brindar una mejor experiencia a los usuarios, mejorando visualmente la organización de las funciones en la *app* de tal forma que sea amigable y moderno haciéndolo más atractivo al usuario, así como también se busca agilizar los procesos que se llevan a cabo dentro de la aplicación como son: transferencias, pago de servicios, pago de tarjetas, recargas celulares, sistema de inversiones, entre otras.

Para llevar a cabo la implementación de esta modernización se propone realizar una migración de la aplicación a tecnologías más recientes. Para la parte del *frontend* se realizará un análisis de cómo se encuentra organizada la aplicación actualmente y se rediseñara cada nueva pantalla de la aplicación, teniendo elementos que actualmente se pueden ver en otros bancos y añadiendo nuevos diseños que sean más intuitivos para facilitar el uso de la aplicación.

Además de la migración de elementos visuales, se estimó migrar la parte que contiene el desarrollo de los servicios, la arquitectura que se tiene actualmente es monolítica, es decir, que todo el *backend* se encuentra encapsulado en una sola capa lo cual tiene ciertas ventajas pero también ciertas deficiencias que previamente fueron mencionadas.

Entonces la tarea principal es modificar la arquitectura con la que se accede a los servicios, ahora se requiere utilizar una arquitectura de microservicios, donde se encapsulan servicios dependiendo de la función que realicen para que de esta manera ante cualquier inconveniente que la aplicación llegue a presentar sea corregida con mayor facilidad y sobre todo que no se vea afectada toda la aplicación entre otras ventajas que brinda este tipo de arquitectura que de igual forma se detalla en el apartado de marco teórico.

5. Análisis y metodología empleada

“Actualmente los negocios operan en un entorno global que cambia rápidamente. Tienen que responder a nuevas oportunidades y mercados, condiciones económicas

cambiantes y la aparición de productos competidores. El software es parte de casi todas las operaciones de negocio, por lo que es fundamental que el software nuevo se desarrolle rápidamente para aprovechar nuevas oportunidades y responder a la presión competitiva" [Sommerville] [8]

Al ser una institución bancaria que está sujeta a regulaciones constantes el desarrollo debe ser entregado en periodos cortos de tiempo ya que de lo contrario la institución podría sufrir multas, por ello se estableció una metodología de desarrollo ágil. La forma en la que se lleva a cabo la organización del proyecto es mediante la metodología *SCRUM*.

Esta metodología permite el trabajo colaborativo, y garantiza que sea ágil ya esto se debe a que la carga de trabajo total es dividida en tareas más pequeñas de las cuales se determina qué tareas serán resueltas en un tiempo determinado al que se le llama *sprint*.

Figura 5 - Ciclo de un sprint

A continuación se brinda una descripción de las actividades que se realizan en cada reunión.

[8] Sommerville, I. (2005). Ingenieria del software (M. I. A. Galipienso, A. Botia Martinez, F. Mora Lizan, & J. P. Trigueros Jover, Trads.; 7a ed.). Pearson Educacion.

5.1 *Sprint planning*: En esta reunión se determina que tareas se trabajarán durante el *sprint*, también se determina un puntaje para cada tarea que significa la dificultad que normalmente se refiere a los días que conlleva realizar la actividad, esto puede depender de distintos factores como puede ser la experiencia o dificultad de una actividad ya que de eso depende la rapidez con la que resuelve un problema.

5.2 *Daily scrum*: Para que el desarrollo se lleve de manera ágil se convocan reuniones diarias llamadas *dailys* donde se mencionan de forma breve los avances que se tienen o bien se externan dudas, además se informa los bloqueantes dentro de las actividades asignadas ya que esto puede modificar la dificultad de dicha actividad.

El objetivo de esta reunión diaria es conocer el progreso diario y buscar pronta solución a los problemas que se presenten y así poder completar el objetivo del *sprint*.

5.3 *Sprint review*: Cuando un *sprint* está por finalizar se realiza una reunión donde se realiza una demostración mediante una presentación con el detalle de las actividades en progreso, lo que se comenzará a trabajar en los siguientes *sprints*, los bloqueantes y las razones por las que se consideran de esa manera.

También se llega a mostrar alguna imagen o video que sirva como evidencia del funcionamiento de algún servicio o de la aplicación para que el cliente vea el progreso mostrado como algo funcional.

En este momento se genera un espacio de preguntas y respuestas sobre las dudas que surjan sobre la presentación que no resulten claras para el cliente, también se habla de los bloqueantes y se debate para dar una solución lo más pronto posible y así seguir avanzando con las tareas pendientes.

5.4 *Sprint retro*: Una vez que se finaliza el *sprint*, se realiza una reunión de retroalimentación, en esta normalmente se realiza una dinámica para mencionar aquellas situaciones que se viven durante el *sprint* como son:

1. Objetivos: las propuestas que se comprometieron alcanzar para el *sprint*.

2. Fortalezas: las acciones o situaciones que sucedieron en el *sprint* que finaliza y que se deberían mantener para los siguientes *sprints*.
3. Riesgos: situaciones que se deben prevenir para que no se conviertan en impedimentos durante el siguiente *sprint*.
4. Impedimentos: Se mencionan las acciones que se convirtieron en un bloqueante y se proponen acciones para buscar una pronta solución cuando se llegué a presentar una situación similar más adelante.

Una vez mencionado lo anterior se debate sobre acciones para que los riesgos se reduzcan o se tenga un plan de acción para cuando sucedan, en caso de existir se mencionan nuevas estrategias para seguir agilizando el desarrollo y tener una mejora continua.

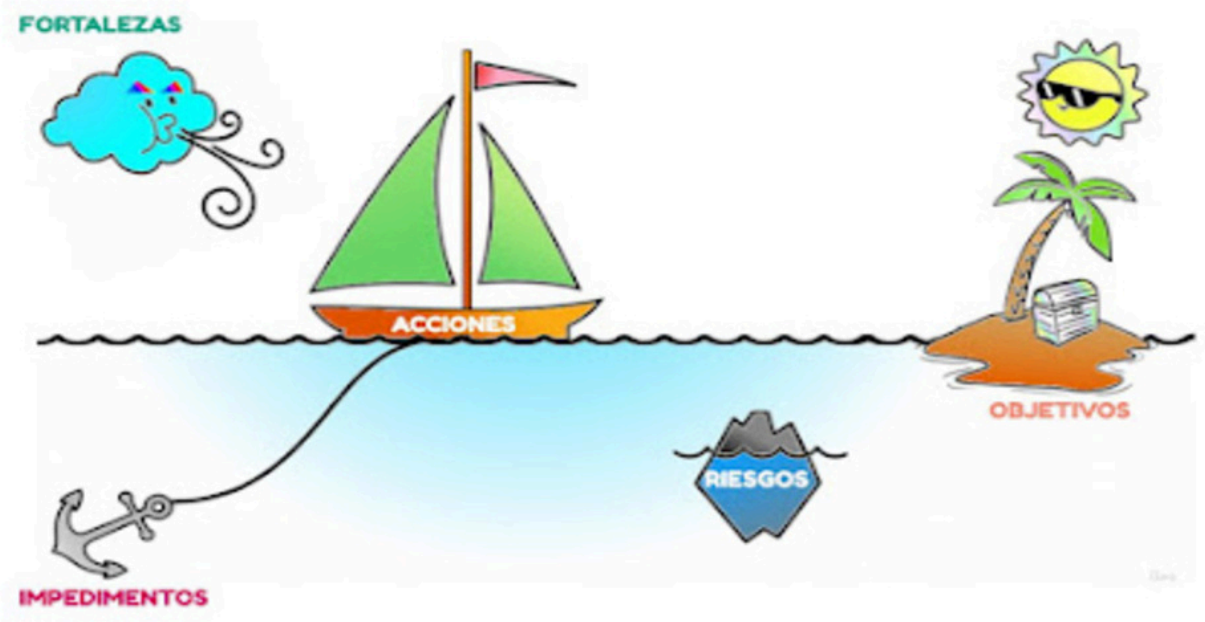


Figura 6 - Ejemplo de sprint retro

6. Desarrollo del proyecto

Durante todo el proyecto se trabajó mediante la metodología *scrum* para llevar a cabo la migración de la aplicación bancaria a microservicios con ayuda de la herramienta *Jira* para facilitar la organización y poder visualizar las tareas asignadas en cada *sprint*.

El proyecto se enfocó tanto en *backend* como en *frontend* y también un poco en pruebas manuales y automáticas.

Cuando se determinaba qué funcionalidad se iba a migrar en el *sprint* se hacía la identificación de los servicios necesarios para dicha función. Algunos ejemplos son :

Funcionalidad	Detalle de Producto	
Servicios	*getInfoUser()*	Obtiene información del usuario como: id, teléfono, dirección, dispositivo de conexión, etc.
	listProduct()	Obtiene una lista con el número de cuenta o tarjeta que tiene el cliente además de tipo de moneda, fecha de apertura.

Funcionalidad	Fondos de inversión	
Servicios	*fundContent()*	Consulta y genera un archivo *json* con el nombre de los fondos activos en el banco y su código que lo identifica, adicional una url de un archivo que contiene gráficas sobre cada fondo. Esta información la almacena en una base de datos.
	fundPerformance()	Obtiene el rendimiento de cada fondo así como su proyección a 30, 90, 180 y 360 días, su tipo de moneda y el código que le corresponde a cada fondo.

	compositionFund()	Lee el archivo que se genera en *fundContent()* para ser mostrado al momento que el usuario ingrese al apartado de invertir.
	newInversor()	Se contrata una nueva inversión a un fondo de inversión dependiendo del riesgo que el cliente está dispuesto a tener para una inversión a corto, mediano y largo plazo.

Funcionalidad	Servicio de pago	
Servicios	*validateDate()*	Este servicio verifica que la fecha en la que se realiza un pago de servicio no sea día festivo o fin de semana.
	getEntrprises()	Obtiene un listado de empresas que se pueden elegir para realizar un pago de servicio, una recarga celular o de transporte.
	listProduct()	Obtiene una lista con el número de cuenta o tarjeta que tiene el cliente además de tipo de moneda, fecha de apertura.
	servicePayment()	A través de este servicio se pueden realizar pagos a servicios, pagos a tarjetas y recargas. Este servicio consulta las respuestas de los 3 microservicios anteriores para

		recopilar la información necesaria del pago.
	validatePayment()	Para confirmar el pago este servicio envía un *token* por *sms* o correo cuando el pago es realizado por primera vez para evitar fraudes.

De acuerdo a lo anterior se añadía a *jira* una *epic*[9] que representaba la funcionalidad, dentro de la épica se añadía una historia de usuario[10] por cada servicio y estas historias se dividían en parte *front* y parte *back* con el fin de publicar en producción la funcionalidad completa.

Durante la *planning* se debate sobre las actividades que conlleva realizar la migración de cada servicio para poder puntuar la historia. Por ejemplo:

Servicio	*listProduct(idUser)*
Detalle	Consulta el *idUser* que proviene de la respuesta de *getInfoUser()*
	Consulta el servicio de *middleware productsList()*

Servicio	*fundContent()*
Detalle	Consulta 2 servicios de *middleware* que son *fundsGateway()* y *carteraGateway()*

[9] Epic es un conjunto de trabajo de grán tamaño que puede dividirse en tareas específicas denominadas “historias de usuario” en función de las necesidades o solicitudes de los clientes o usuarios finales. Los epics son una práctica importante para los equipos ágiles y de DevOps. Consultado en https://www.atlassian.com/es/agile/project-management/epics el 04 de julio del 2024.
[10] Una historia de usuario es la unidad de trabajo más pequeña en un marco ágil. Es un objetivo final, no una función, expresado desde la perspectiva del usuario del software. Consultado en .https://www.atlassian.com/es/agile/project-management/user-stories el 04 de julio del 2024

	Filtra las respuestas de *middleware* para obtener únicamente los fondos activos.
	Consulta de la base de datos el archivo que contiene los fondos activos y los compara con el que se generó actualmente.
	Si el archivo es distinto entonces guarda el actual en la base de datos reemplazando el *json* anterior.

Servicio	*servicePayment()*
Detalle	Consulta la respuesta del servicio *validateDate()* para validar que el pago puede proceder
	Consulta la respuesta del servicio *getEnterprises()* para obtener el número que identifica al servicio que se pagará o la compañía telefónica en caso de realizar recarga.
	Si se pagará una tarjeta entonces se verifica que tipo de tarjeta es (*VISA* o *MASTERCARD*)
	Obtiene de *listProduct()* el número de cuenta/tarjeta con la cual el usuario indicó realizar el pago
	Envía el *request* con la información recopilada a un api externa de *Prisma* [11]para procesar el pago.
	Si el pago se realizó con éxito entonces se muestra la confirmación al *front*, de lo contrario se asigna un código de error pre-establecido.
	El pago es guardado en la base de datos con el estado correspondiente (*SUCCESS, ERROR, PENDING*).

[11] Prisma se focaliza en procesamientos de pagos online y presenciales, de todas las marcas de tarjetas y entidades financieras. Consultado en https://www.prismamediosdepago.com/ el 05 de julio del 2024

Algo que se debe tener en cuenta para cada historia es el análisis previo que consiste en revisar cuáles son sus parámetros de entrada y salida del servicio actual para replicarlo de la misma manera en la migración. Por ejemplo al servicio de *“listProduct”* se le asignaron 3 puntos tomando en cuenta la documentación que se genera antes y después de cada desarrollo (la cual detallaré más adelante), estos puntos representan una duración de 3 días aproximadamente para finalizar únicamente esta historia.

Comúnmente los servicios se puntúan entre 5 y 8 como por ejemplo el *“fundContent”* debido a que realizan conexiones a bases de datos o una mayor cantidad de consultas a otros servicios. Hubo excepciones de algunos servicios que resultaban muy cortos como el ejemplo anterior de *“listProduct”*, el cual únicamente consultaba 2 servicios internamente sin realizar un cambio en los formatos de la información. También hubo servicios extensos que tenían 13 o más puntos de historia como *“servicePayment”,* que en este caso se generaron subtareas para que no tuviera afectación en las gráficas de rendimiento al finalizar el *sprint*.

Dentro del *back-end* las tareas eran asignadas conforme sus puntos de historia, y estos puntos dependían de la complejidad. Una persona debería tener alrededor de 14 a 18 puntos dentro de un *sprint* que tenía la duración de 3 semanas, considerando que cada punto equivale aproximadamente a un día hábil de trabajo.

Para tener una idea general de las actividades que se desarrollaron para cada funcionalidad se muestra el siguiente diagrama:

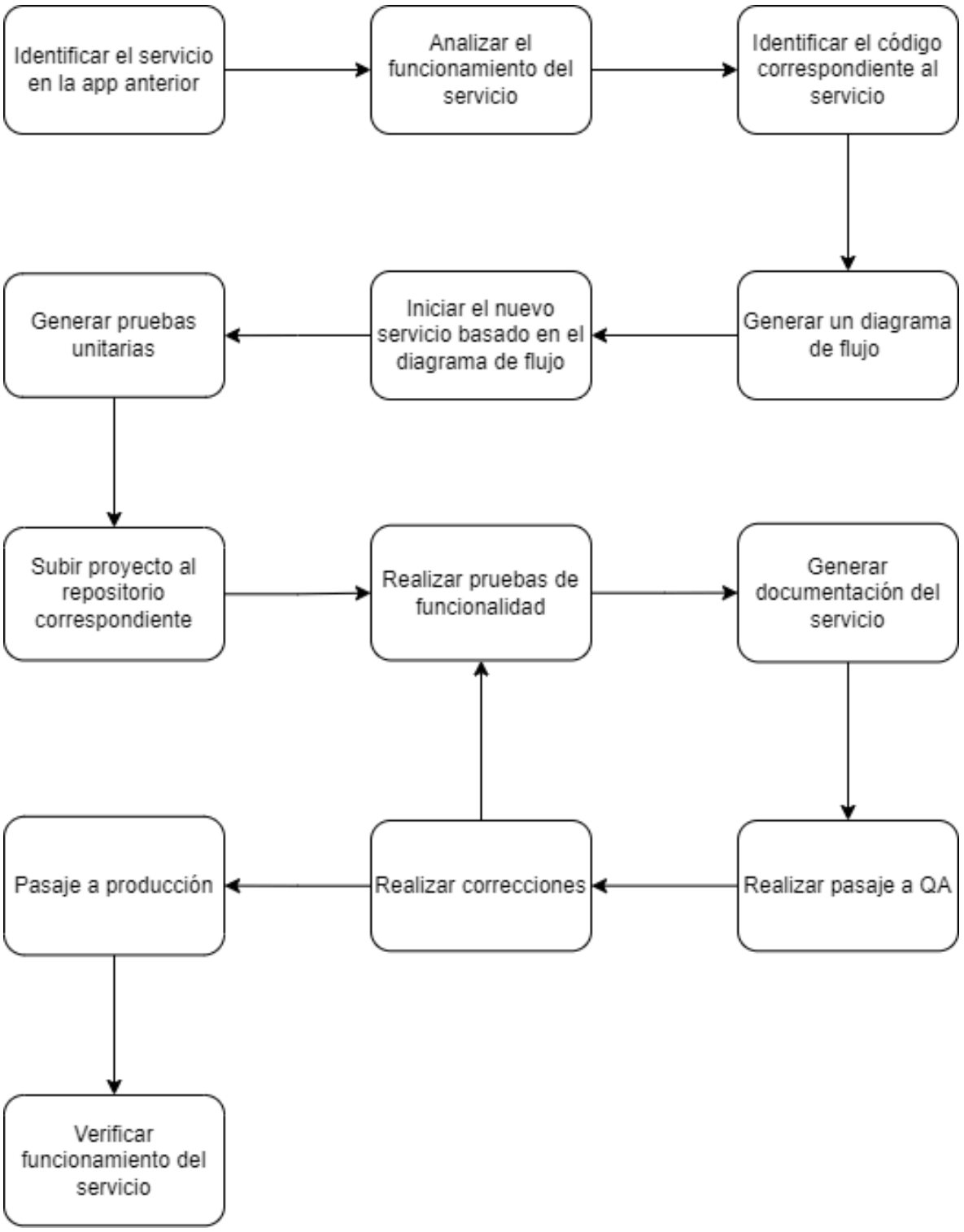


Figura 7 - Diagrama de actividades como desarrolladora java.

6.1 Identificar el servicio en la app anterior

Una vez comenzado el *sprint* y que los tickets fueron asignados se comienza con el análisis de los servicios. Para ello es necesario conocer en qué parte de la aplicación anterior se utiliza la operación para así poder identificar qué parámetros se envían a esta y cuáles devuelve, así como el formato en el que lo establece, es decir, si es un formato *JSON* o *XML* en su respuesta.

Por ejemplo, para el ticket de *"servicePayment"* primero se debe realizar una búsqueda dentro de la *app*, por lo que se siguieron los siguientes pasos para encontrar el llamado a este servicio.

Paso 1: Ingresar a la aplicación y seleccionar el apartado de "Pagos".

Figura 8 - Pantalla principal de la app a migrar.

Paso 2: Aparece una lista con los servicios a los que se puede realizar un pago, seleccionamos "*Visa*" para realizar un pago a una tarjeta.

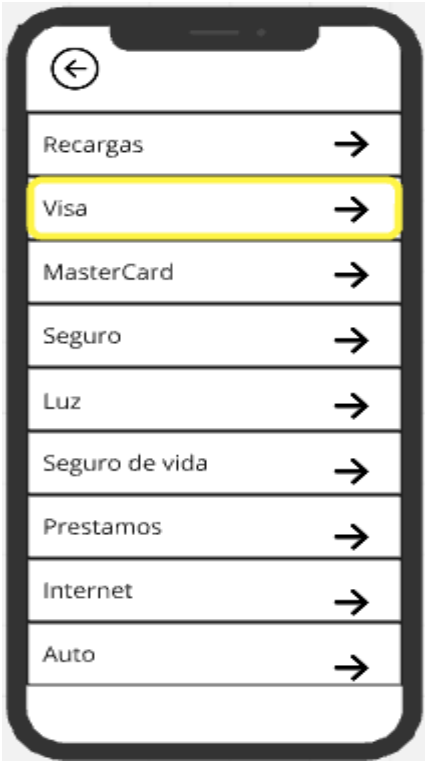


Figura 9 - Listado de servicios disponibles a pagar.

Paso 3: Se muestra una pantalla para ingresar el número de la tarjeta que se requiere pagar y otro apartado para ingresar el importe. Se ingresan los datos y seleccionamos “continuar".

Figura 10 - Pantalla para realizar un pago a una tarjeta.

Paso 4: Después aparece un listado con las cuentas que el cliente tiene en el banco. Se selecciona la cuenta con la que se va a realizar el pago y se selecciona “continuar”.

Figura 11 - Pantalla que muestra la cuenta para realizar un pago. (Datos ilustrativos)

Paso 5: Finalmente se muestra la confirmación del pago con un resumen de los datos seleccionados previamente y es en esta pantalla que se ejecuta *“servicePayment”*

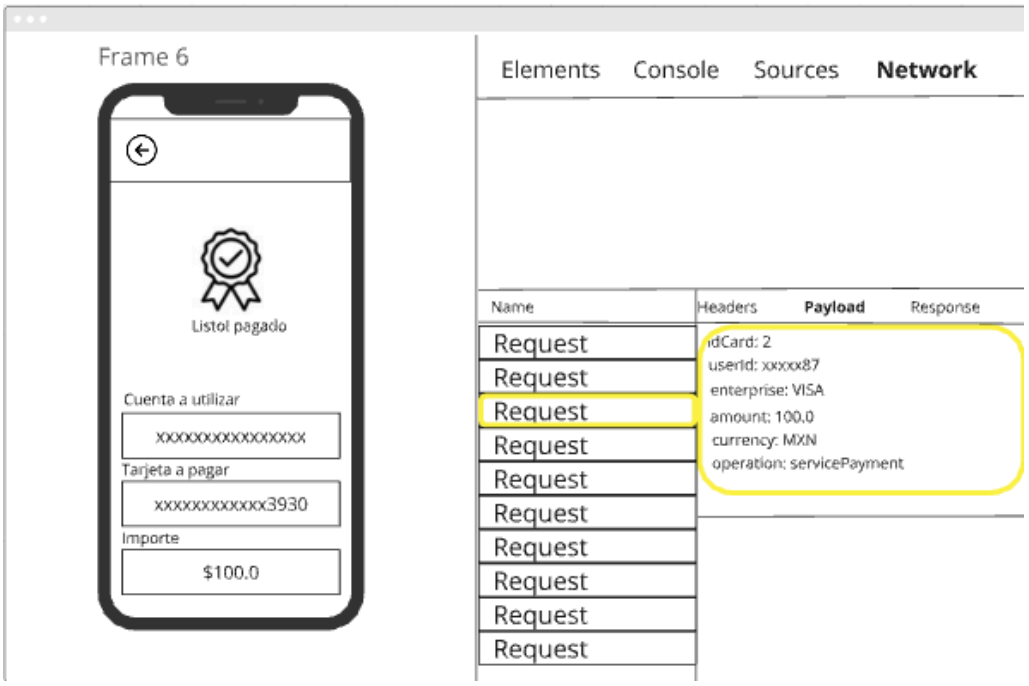


Figura 12 - Representación de la visualización de un servicio en la app a migrar en modo desarrollador.

Es importante mencionar que no existe documentación que se pueda consultar donde indique en qué parte es que se ejecuta el servicio, el paso a paso se va descubriendo conforme se van seleccionando los distintos botones hasta encontrar la funcionalidad que se solicita en el *ticket*.

Esta información se almacena en un documento llamado “Ficha previa” ya que es fundamental conocer esta información para mantener el mismo formato en la nueva versión y que servirá al momento del análisis e implementación del código para garantizar que la migración sea lo más transparente posible y no se altere ninguna funcionalidad ya definida.

Al finalizar este análisis se añade una leyenda como comentario en el *ticket* que indique que esta etapa está completada, con el fin de ir documentando la tarea y conocer el estatus de la asignación.

6.2 Analizar el funcionamiento del servicio

Una vez identificado el servicio en la app podemos darnos cuenta el origen de los datos que se envían como *request* a la funcionalidad, es decir, si los recibe de la respuesta de otro servicio o son valores que se guardan en la sesión del usuario.

Detallando la información encontrada en el “paso 5” de la identificación de la operación se encontró que, algunos de sus parámetros provienen de otras operaciones que se ejecutan en servicios anteriores que se mencionan en la siguiente tabla:

Parámetro	Origen
idCard	Es el índice dentro de la respuesta de *“listProduct”*, indicando cual es la cuenta con la que se paga.
userId	Se obtiene de la respuesta de *“getInfoUser”* que representa el identificador del usuario.
enterprise	Se obtiene de la selección que se realiza dentro del listado que devuelve la consulta de *“getEntrprises”* al momento de indicar el tipo de tarjeta a pagar.
amount	Cantidad a pagar que ingresa el usuario.
currency	Tipo de moneda en la que el pago se realiza.
operation	Nombre del servicio a ejecutar.

Tabla con el detalle del origen de los parámetros del request para la operación.

La definición del origen de estos parámetros se realiza observando las ejecuciones de las operaciones dentro de las pantallas previas al paso 5 y determinar cuales están relacionados con el servicio que se está revisando.

Además también se obtiene el *url* y el *response* para determinar cómo se debe devolver la información dentro del microservicio que se va a generar. Los datos se obtienen del siguiente apartado:

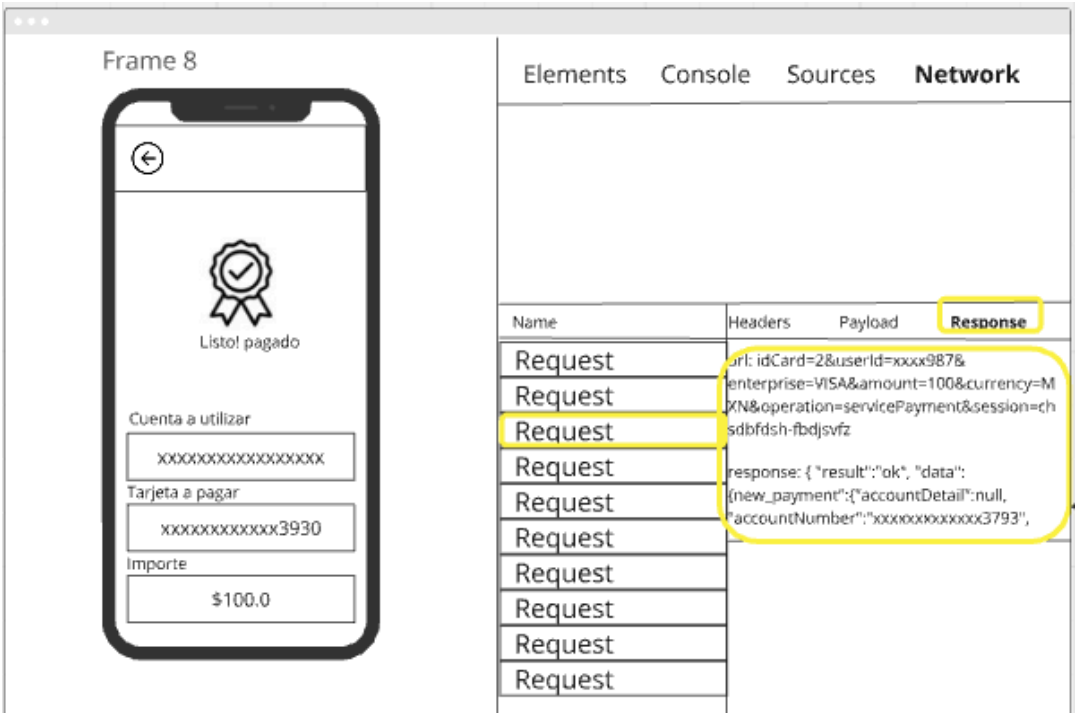


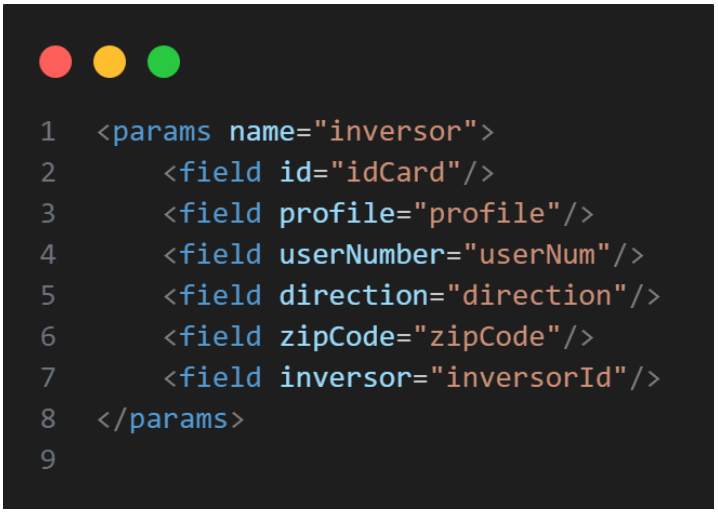
Figura 13 - Imagen con el url y response del servicio buscado en la app.

Esta información se agrega a la “Ficha previa” para que más adelante sirva como referencia al momento de realizar pruebas.

6.3 Análisis del código correspondiente al servicio

Primero se identifica el archivo que corresponde al servicio a analizar, estos archivos son del tipo *XML* y su nombre corresponde al de la operación que define.

Dentro del archivo se encuentra un apartado para los parámetros de entrada y salida de la funcionalidad como se muestra en la siguiente imagen:



```
<params name="inversor">
    <field id="idCard"/>
    <field profile="profile"/>
    <field userNumber="userNum"/>
    <field direction="direction"/>
    <field zipCode="zipCode"/>
    <field inversor="inversorId"/>
</params>

```

Figura 14 - Definición de parámetros de entrada y salida para la operación newInversor.

Posteriormente se encuentra la definición para el o los servicios de *middleware* que esta operación consume.

```
<service name=inversor class="com.operation.BaseOp">
    <ref name="sendReq" id="ReqInversor"/>
    <ref name="Receiver" id="RespInversor"/>
</service>
```

Figura 15 - Definición de servicio middleware al que se conecta la operación.

"ReqInversor" hace referencia al objeto que contiene el nombre de los valores de entrada al servicio y *"RespInversor"* se refiere a los valores que devuelve en su respuesta. Estos objetos se encuentran en otra clase especialmente para *middleware*, por lo que se deben buscar estos identificadores para encontrar el detalle del contenido de estos datos. Por ejemplo:

```
<params name="ReqInversor">
    <field option="option"/>
    <field perfil="perfil"/>
    <field numcliente="numcliente"/>
    <field cta="ctaCliente"/>
    <field codePost="codePost"/>
    <field number="number"/>
</params>
```

Figura 16 - Definición de los parámetros de entrada para el servicio middlewere de newInvestor.

```
<params name="RespInversor">
    <field resultado="resultado"/>
    <field opcion="opcion"/>
    <field inversor="inversor"/>
    <field numeroDoc="numDoc"/>
    <field perfil="perfil"/>
    <field inversor="inversorId"/>
    <field cuenta="cuenta"/>
    <field localidad="localidad"/>
    <field nombre="nombre"/>
</params>
```

Figura 17 - Definición de los parámetros de salida para el servicio middlewere de newInvestor.

Posteriormente se encuentran instrucciones que indican cómo va a actuar el servicio y las validaciones que se realizan de manera interna. Esta parte es de suma importancia ya que, a partir de estas líneas se va generando un diagrama de flujo que será la base para generar la migración correctamente.

```
<instruction name="IsEmpty" param="profile" do="continue" or="SetError" />
<instruction name="Compare" param="profile" value="A" do="SetUno" or="continue" />
<instruction name="Compare" param="profile" value="B" do="SetDos" or="continue" />
<instruction name="Compare" param="profile" value="C" do="SetTres" or="continue" />
<instruction name="Compare" param="profile" value="D" do="SetCinco" or="SetError2" />
```

Figura 18 - Pasos que indican las validaciones del parámetro de entrada “profile”.

Explicando el ejemplo anterior se tiene la primera instrucción con el nombre de *“IsEmpty”* este paso verifica que el parámetro profile sea diferente de nulo o vacio, si esta condición se cumple, entonces se continúa a la siguiente línea pero en caso de que no se cumpla la validación, entonces se debe asignar un error y dar por finalizada la ejecución. La instrucción *“Compare”* valida que el parámetro *profile* sea igual al *“value”*, si hay coincidencia realiza un flujo correspondiente. Este análisis se realiza para las instrucciones siguientes del archivo.

Este análisis también se agrega a la ficha previa con la información recopilada donde la estructura base del documento es la siguiente:

Ficha previa de <NombreDelServicio>

- Seguimiento de pantallas paso a paso donde se consume el servicio
- Manera en la que se consume desde la *app* anterior *(request, url y response)*
- Servicios *middleware* que consuma
    - NombreServicios (*request*, *response,url)*
- Análisis técnico (diagrama de flujo)

El diagrama de flujo correspondiente a *newInversor* resultado del análisis anterior sería:

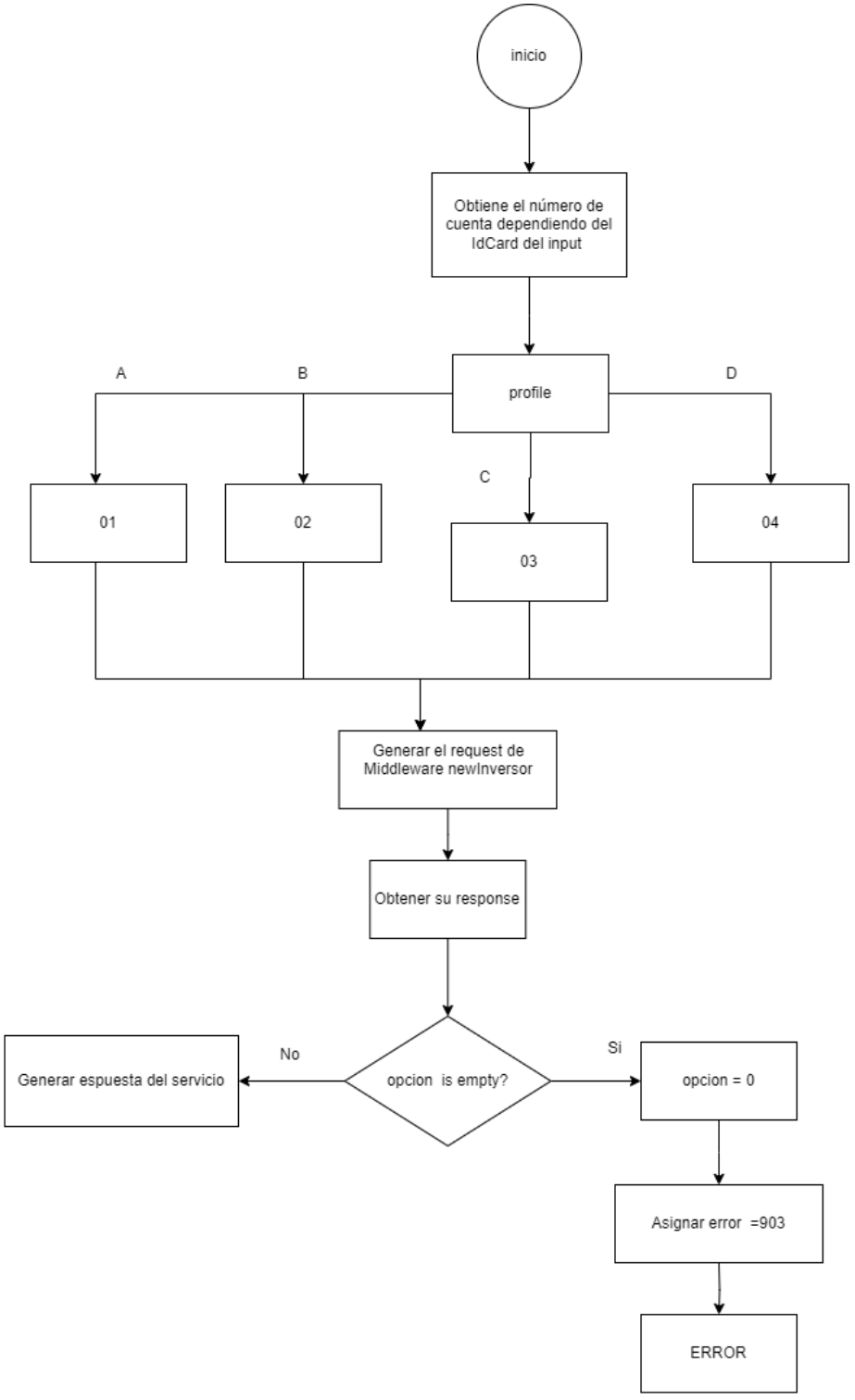


Figura 19 - Ejemplo de diagrama de flujo de un servicio de inversión

6.4 Iniciar un nuevo servicio

Existe una herramienta donde se da de alta un nuevo servicio que a su vez al guardarlo se genera un proyecto base con el nombre que se indica en la herramienta, ejemplo:

Figura 20 - Herramienta donde se genera un nuevo proyecto para el microservicio.

Después de generar el microservicio se deben mapear los parámetros de entrada y salida indicando su tipo de dato *(date, int, string*, etc) y si se trata de un dato sensible como por ejemplo un número de cuenta, entonces se indica la manera en la que esté dato será protegido (ofuscado, cifrado, etc), además se da una breve descripción de lo que el valor representa y de ser posible un ejemplo.

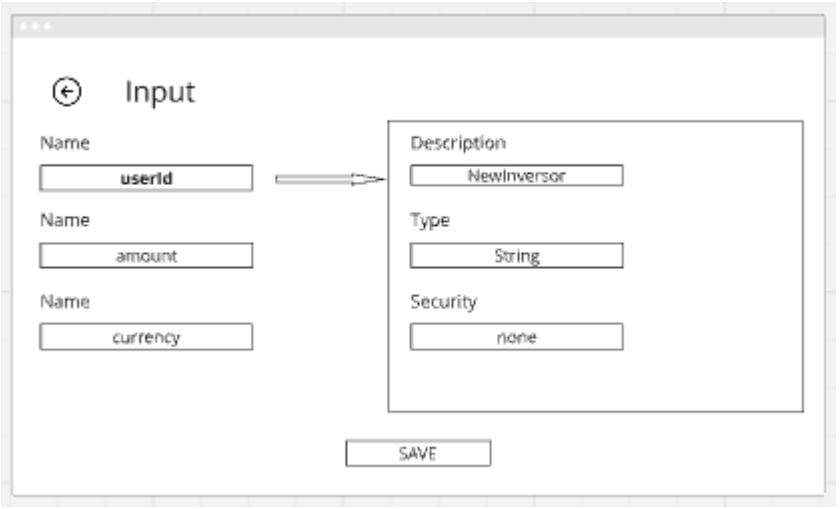


Figura 21 - Pantalla donde se dan de alta los parámetros de entrada y salida.

Cuando se guardan estos parámetros automáticamente se agregan al proyecto *DTO´s* que corresponderá a la entrada y la salida que se agregó previamente.

El proyecto se desarrolla en lenguaje java donde también se utiliza *maven* que es un gestor de dependencias que nos ayuda al momento de compilar y empaquetar el proyecto para poder ejecutarlo después.

Con ayuda del diagrama de flujo se puede desarrollar un microservicio en el cual se conserva la lógica de lo que el servicio realiza, sin embargo, se utilizan instrucciones pertenecientes a una versión de java más reciente y organizando mejor los archivos, es decir, se genera un archivo para manejo de excepciones donde se encuentran códigos de error para tener una respuesta controlada en caso de una situación inesperada, también existe un archivo para los llamados a base de datos que se realizan mediante *stored procedures*, un archivo por el desarrollo de la lógica del servicio y en caso de ser necesario un archivo utils donde se colocan métodos generales que sean de utilidad para más de un servicio haciendo el código reutilizable.

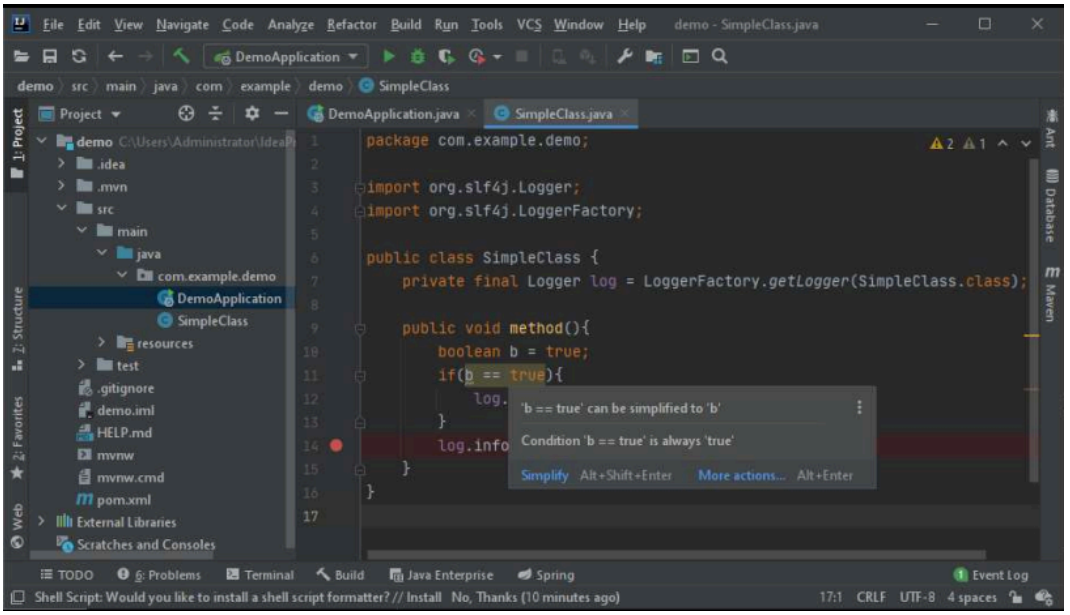


Figura 22 - Ejemplo de estructura de un nuevo proyecto

6.5 Generar pruebas unitarias

La prueba unitaria está diseñada para verificar que el bloque de código se ejecuta según lo esperado, de acuerdo con la lógica teórica del desarrollador. La prueba unitaria sólo interactúa con el bloque de código a través de entradas y salidas que son comparadas con nuestra validación.

Sabiendo lo anterior se generan pruebas unitarias tanto para respuestas esperadas como para códigos de error, también se debe verificar que las validaciones plasmadas en el diagrama de flujo sean correctas al igual que el tipo de formato que se le da a los datos como fecha, importes, etc. Por ejemplo:

```java
@Test
    public void validateErrorNotUserTest() {
        Mockito.when(getUserInfo(anyString))
        .thenThrow(new IllegalArgumentException("El id del usuario no se encuentra registrado"));

        Response result = newInversor.postInversor(2, 500.0, "MXN");

        assertEquals("El id del usuario no se encuentra registrado", result.header.message);


    }
```

Figura 23 - Prueba unitaria básica para validar el mensaje de error cuando la información de un usuario no se encuentra.

No existe una cantidad exacta de pruebas unitarias que se deban realizar pues eso más bien depende de cada servicio ya que algunos son más complejos que otros o contienen un número de validaciones mayor.

6.6 Subir proyecto al repositorio

Finalizado el microservicio junto con las pruebas unitarias se compila y se genera el archivo *.jar* con ayuda de *maven* para verificar que no existen errores de código.
Se utiliza la herramienta de *git* para poder subir el proyecto al repositorio que le corresponda, de está forma se tienen controladas las versiones y modificaciones que se realicen.

Al momento que el código se encuentra en el repositorio pasa por algunas validaciones más las cuales se llaman *pipelines*, estos son varios pasos donde se configura, compila y se ejecutan nuevamente las pruebas unitarias, además de realizar un análisis de la calidad del código.

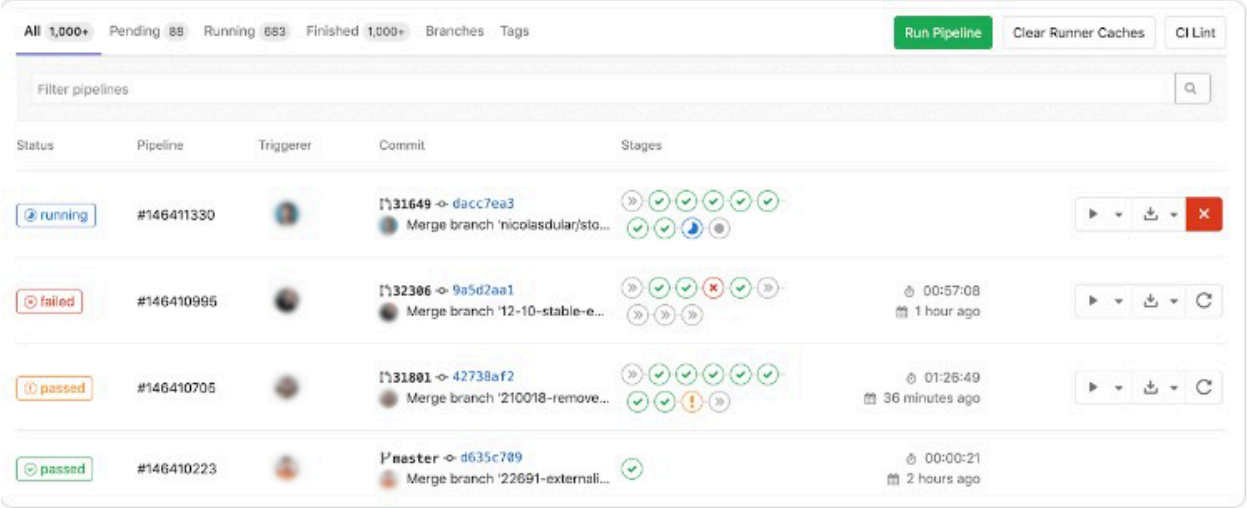


Figura 24 - Ejemplo de ejecución de pipelines para validación de código

El último paso que corresponde al análisis de calidad de código se realiza mediante la herramienta *SonarQube* que analiza totalmente el código para generar métricas que puedan ayudar a mejorar la calidad del desarrollo. Dentro de los errores que detecta son: vulnerabilidades, código repetido, o bien pruebas unitarias que no cubren correctamente lo que se desarrolló. Si las métricas son menores al 75% entonces marcará un error y no permitirá subir el código hasta que se alcance la calidad mínima requerida, sin embargo, se espera que el código tenga un *coverage* mayor a ese porcentaje. A continuación se muestra un ejemplo:

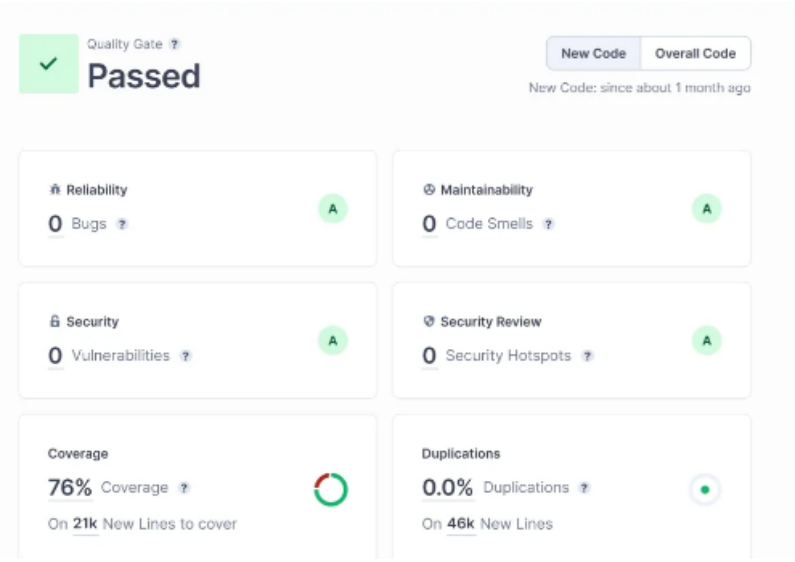


Figura 25 - Análisis que realiza SonarQube

6.7 Realizar pruebas de funcionalidad

Cuando el código está disponible en el repositorio se tiene que verificar la disponibilidad del servicio por el método *http* correspondiente, esta prueba se realiza con ayuda de la herramienta *Postman*, en esta etapa se busca replicar todos los casos que se colocaron en las pruebas unitarias para asegurarse que funciona correctamente, en especial que la respuesta contenga el mismo formato que la aplicación antigua y que su información sea correcta.

En ocasiones suele pasar que se encuentran nuevos casos para los que no se tiene una respuesta controlada y que debe ser corregido o manejado mediante un nuevo código de error.

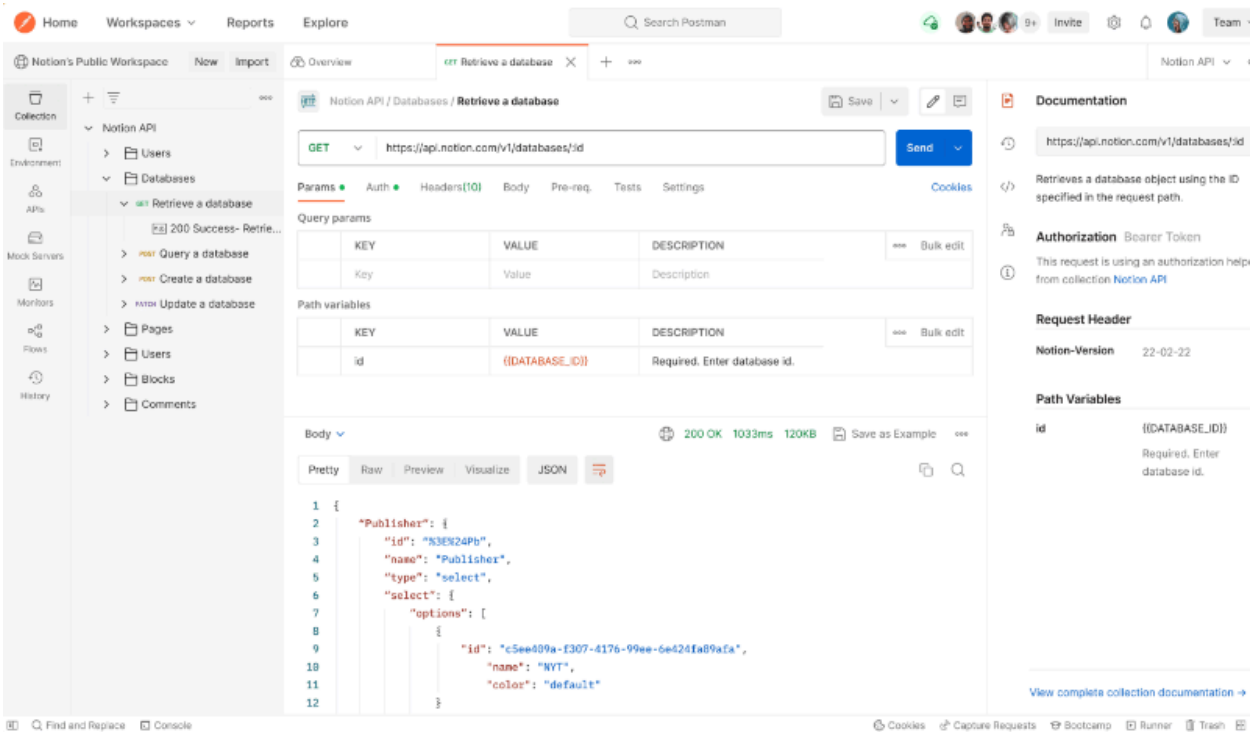


Figura 26 - Ejemplo de prueba de servicio en postman

6.8 Generar documentación del servicio

Cuando se finaliza la migración de un servicio se debe realizar un documento en el que se explique lo más detallado posible cuál es la funcionalidad del servicio, después se añaden capturas de pantalla que demuestren la secuencia que hay que seguir desde el inicio de la app hasta la pantalla donde se ubica el servicio que se está utilizando, luego se realiza un diagrama de secuencia que demuestra como es el flujo completo

desde que se loguea el usuario en la aplicación hasta que consume la funcionalidad migrada, indicando todas las interacciones que esta llegue a tener, estas pueden ser bases de datos, servicios *middleware*, entre otros. Por ejemplo:

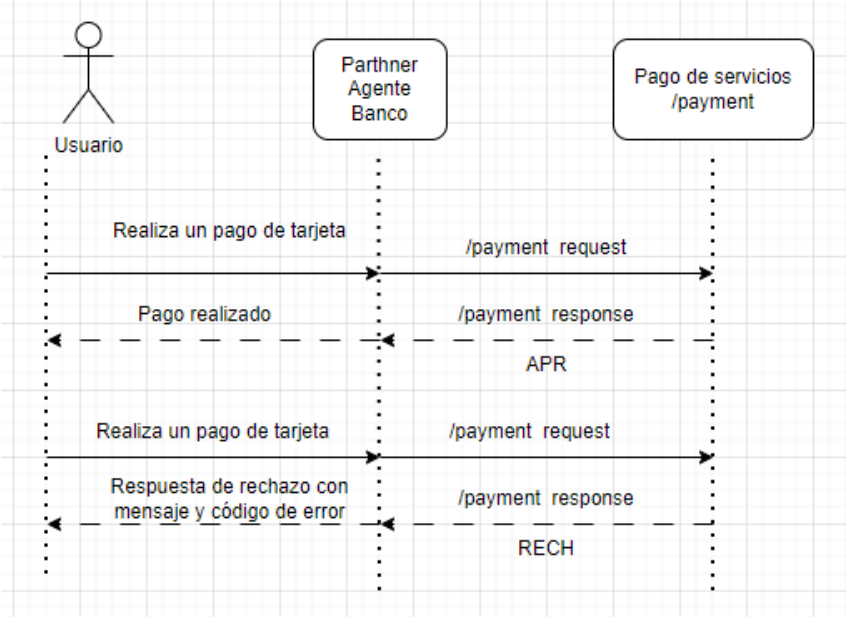


Figura 27 - Diagrama de secuencia básico

También se coloca un ejemplo de cómo se utiliza el servicio donde se incluya la *url* para su ejecución, mencionar los parámetros del *request* que se mandan en formato *JSON* y las formas en las que puede responder, las cuales describen el caso exitoso y en caso de existir códigos de error esperados se deben colocar también dentro de la documentación.

```
{
    "header": {
        "messageCode": "C0D148",
        "messageDescription": "BdCommunicationException",
        "resultCode": "fail",
        "transactionId": "8a33ff49",
        "errorDetail": [
            {
                "code": "C0D148",
                "description": "Detalle: CODIGO DE TABLA DEBE INFORMARSE"
            },
            {
                "code": "C0D148",
                "description": "BdCommunicationException"
            }
        ]
    }
}
```

Figura 28 - Ejemplo de response en caso de error en la base de datos

Si el servicio tiene conexión a una base de datos se indica la siguiente información:

- nombre de la instancia

- puerto
- ambiente
- tablas que modifica y/o consulta

Finalmente se enlistan los *links* del repositorio donde se encuentra actualmente el código fuente del microservicio y dónde se almacenan los *logs* cada que el servicio es ejecutado para su monitoreo y así poder detectar errores.

Hasta este punto se finaliza un *ticket* de una migración. Una vez realizado este cambio en el *ticket* se puede comenzar con uno nuevo.

6.9 Realizar pasaje a *QA*

Para realizar este proceso se necesita tener asignado un *ticket* con la descripción y fecha en la que se hará este cambio de ambiente.

Para comenzar a desarrollar el servicio este debe funcionar con normalidad, dentro del ambiente de desarrollo se deben realizar varios pasos previos ya que las funcionalidades deben dar seguimiento a los requerimientos y así se emplee la integración continua.

Como segundo paso una vez aprobado el pasaje a *QA* se verifica que la documentación del servicio contenga las rutas donde se encuentra el repositorio y archivos de configuración, diagramas de secuencia que se realizaron previamente, etc.

La estructura que se genera para proseguir con el pasaje del microservicio al ambiente de calidad es la siguiente:

- Documentación
  - CONTROL DE CAMBIOS
  - DIAGRAMA DE ARQUITECTURA (SRO)
  - CERTIFICACIÓN (Tech Leader/un Dev)
    - Servicio
    - Endpoints
    - URL para verificar el servicio
      - Endpoint1
  - ESQUEMA DE SECUENCIA (Tech Leader/un Dev)
    - Endpoint1
  - INVOCACIONES Y DEPENDENCIAS (Tech Leader/un Dev)
    - Endpoint1
      - Request
      - Response
    - Repositorio del servicio en Gitlab
  - CERTIFICADOS UTILIZADOS (SRO/Sec Champion)
  - BASES DE DATOS (SRO)
  - USUARIOS UTILIZADOS (SRO)
  - INFORMACIÓN ADICIONAL PARA EL MONITOREO EN APM (Tech Leader)
  - ELASTIC (Tech Leader)
- OK de PO
  - Ejemplo
- Aprobación de Testing
  - Ejemplo

Figura 29 - Estructura de la documentación para realizar deploys

QA es un ambiente controlado donde se realiza una ambientación de datos de tal forma que sea casi una copia de producción, esto es con el fin de que al realizar pruebas de calidad de código las respuestas de los servicios probados sean como se espera en producción y en caso de que no se cumpla con los requerimientos sea reportado a tiempo. En este ambiente también se realizan pruebas de rendimiento en donde se prueba que el tiempo de respuesta de los nuevos microservicios sean menor al actual.

Cuando se tiene completa la documentación y se valida que el servicio funciona como se indica de forma correcta, se debe ejecutar un *pipeline* dentro del repositorio que realiza un *merge* entre la rama de desarrollo del servicio y la de *QA*. Finalizado este proceso de forma exitosa se puede dar inicio a la fase de pruebas.

De acuerdo a los requerimientos se definen matrices de pruebas, en ellas se enlistan distintos casos de uso con la información que se va a verificar para validar la respuesta esperada. La estructura sería la siguiente:

- Caso de uso

    - Aquí se coloca el identificador del caso de uso que se va a ejemplificar.
- Tipo de prueba
    - Se coloca uno de los siguientes tipos: *happy path*, regla de negocio, caso de error.
- Datos a probar
    - En este apartado se coloca el *json* con los parámetros del *request* de prueba.
- Respuesta
    - Se coloca el *json* con el response obtenido de la ejecución de la prueba.
- Resultado
    - Se indica si la respuesta fue la esperada de acuerdo a los datos de entrada, el flujo que se siguió y los tiempos de respuesta esperados.
- Observación
    - Si es resultado es satisfactorio entonces este espacio se queda en blanco, en caso de que el resultado no fuera el esperado aquí se indican las diferencias que existen o la información que se espera recibir.

Después de cada prueba se documentan todos sus casos y en caso de ser necesario se realiza alguna corrección o se marca como aprobado.

Normalmente cuando finaliza el *sprint* se tiene una *sprint review*, donde se muestran las actividades que se desarrollaron durante el *sprint*, además se documentan los bloqueantes que se tuvieron durante el tiempo que duró ese *sprint* para contemplarlo en futuras estimaciones.

6.10 Realizar correcciones

Para esta etapa se generan nuevos *tickets* que se asocian a funcionalidades y entonces se puedan corregir las fallas o realizar las mejoras que se detectaron en la etapa de prueba, estos cambios puede ser: diferencia en el nombre de uno o varios parámetros, el tipo de dato de la información o la estructura del *json* que responde el servicio, también puede ser que el tiempo de respuesta del servicio sea demasiado largo para el tipo de operación, en este caso se intenta mejorar la estructura del código para reducir ese tiempo de ejecución.

La etapa de prueba da la información a modificar y entonces se comienza el desarrollo nuevamente hasta que los cambios están hechos y se realizan nuevas pruebas unitarias o se modifican las existentes según corresponda, después se repite el mismo proceso para subir los cambios al repositorio y seguir realizando una integración

continua, además se actualiza la documentación si es necesario y se procede a pasar al ambiente *QA* como se mencionó anteriormente.

6.11 Pasaje a producción

Para esta tarea normalmente se tienen *tickets* de acuerdo a la cantidad de servicio terminados y probados en *QA* del *sprint* anterior.

Esta es una etapa muy importante ya que se exponen los nuevos desarrollos a los usuarios, el proceso es similar al despliegue en *QA* por lo que se debe actualizar la documentación para que la información del ambiente de desarrollo ahora corresponda al ambiente *QA*, además se requiere que en las pruebas se dé como aprobado el microservicio.

Es importante que al momento de integrar las nuevas funcionalidades se pueda validar constantemente el funcionamiento y evitar posibles fallas al momento de hacer el despliegue. En caso de algún *bug* se puede realizar un *rollback* y trabajar sobre el problema para volver a intentar el despliegue o bien realizar la modificación de manera inmediata y se despliega como un cambio emergente con las correcciones. Los cambios emergentes son aquellos que no necesitan tener una documentación lista sino que al ser un cambio de urgencia se pasa el *pipeline* de manera más rápida.

Dentro de las sesiones de *sprint retro* se habla sobre las problemáticas del *sprint* y cómo se pueden prevenir para próximas ocasiones, también se mencionan las iniciativas que pueden favorecer al proyecto. A partir de esta reunión se generan estrategías que se deben implementar en el siguiente *sprint*.

6.12 Verificar funcionamiento del servicio

Si el servicio se encuentra productivo se realiza un monitoreo durante varios días e incluso semanas de ser posible para localizar comportamientos anormales del servicio. Esta tarea puede volverse complicada cuando existen múltiples peticiones ya que en ocasiones por la velocidad no se alcanzan a registrar todos los llamados al servicio, sin

embargo, si no se reporta ningún inconveniente se da por exitoso el servicio aunque igual se mantiene el monitoreo pero es menos constante.

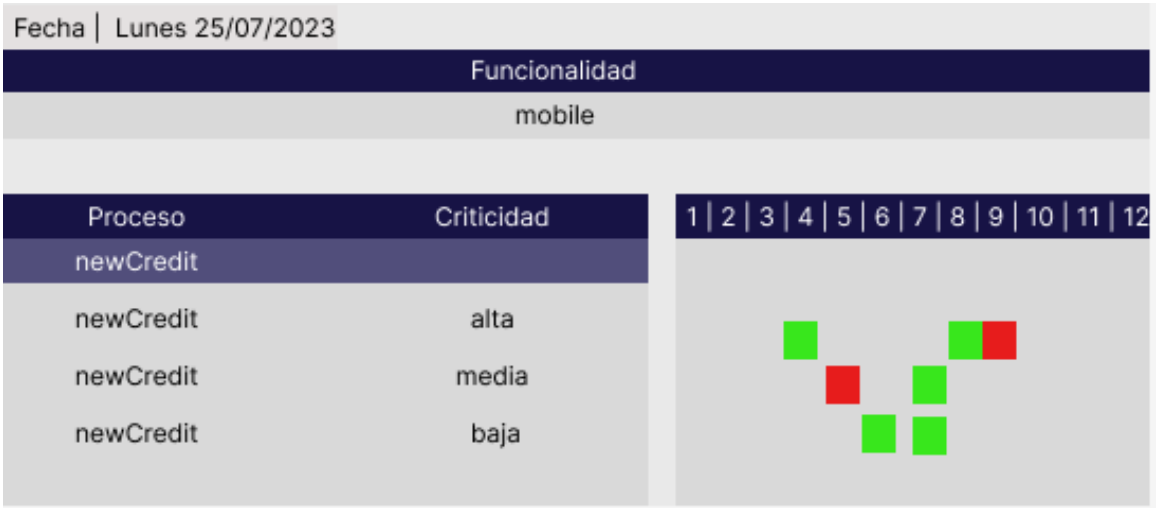


Figura 30 - Pantalla de monitoreo

En el ejemplo anterior se muestra el monitoreo del microservicio *newCredit* que muestra desde dónde se está ejecutando el servicio que en este caso es *mobile* que indica que es desde la aplicación móvil, además se muestran unos recuadros verdes y rojos que indican que esa petición se realizó correctamente o bien que la petición falló y no se obtuvo un resultado exitoso.
Aquellos que estan de color rojo se ve con mayor detalle dandole *clic* para conocer el *id* de esa petición y posteriormente se hace una búsqueda de ese *id* en *Kibana* para ver el detalle del error, si se debe a un error en *front, back* o base de datos. Es muy útil realizar esta búsqueda ya que en ocasiones te dice la línea de código en la que el fallo es detectado y facilita su resolución además que la búsqueda se puede hacer lo más específica y eficiente posible con filtros de fecha, servicio, rango de hora, etc.

7. *Stack* Tecnológico

En el contexto de la migración de una aplicación bancaria de una arquitectura monolítica a microservicios, la selección del *stack* tecnológico se basa en varios criterios clave. Primero, se considera la capacidad de cada tecnología para soportar la modularidad y la independencia de servicios, que son fundamentales en una arquitectura de microservicios. *Angular* fue elegido para el *frontend* por su robustez en la creación de interfaces de usuario interactivas y su capacidad para manejar la

complejidad a través de componentes modulares. En el *backend*, *Java* se seleccionó por su madurez, robustez y soporte para concurrencia, lo cual es crucial para manejar transacciones bancarias seguras y de alto rendimiento. *SQL Oracle* se utilizó debido a su capacidad para manejar grandes volúmenes de datos con alta integridad y sus características avanzadas en términos de seguridad y transacciones. *Maven* se eligió como herramienta de gestión de dependencias y construcción por su amplio uso en proyectos *Java*, lo que facilita la gestión del ciclo de vida del *software*.

Jenkins es una herramienta de integración continua *(CI)* y entrega continua *(CD)* que facilita la automatización de los procesos de construcción, pruebas y despliegue del *software*. En el contexto de una arquitectura de microservicios, *Jenkins* resulta particularmente valioso para gestionar y automatizar las tareas de integración y despliegue de cada microservicio de manera independiente y coordinada.

La inclusión de *Jenkins* en el *stack* tecnológico permite establecer *pipelines* de *CI/CD* que automatizan la ejecución de pruebas unitarias y de integración, así como la construcción y despliegue de aplicaciones. Esto asegura que los cambios en el código se integren de manera continua y se desplieguen de forma eficiente en entornos de producción o *staging*. *Jenkins* también se integra fácilmente con otras herramientas de pruebas y análisis de calidad de código, lo que ayuda a mantener altos estándares de calidad y eficiencia en el desarrollo de la aplicación bancaria.

8. Seguridad

En el proyecto de migración, la seguridad de la información y la privacidad de los datos son de suma importancia, especialmente dado que se trata de una aplicación bancaria que maneja datos sensibles. Uno de los aspectos clave en la seguridad es la ofuscación de datos sensibles en los logs de producción. Esta práctica ayuda a prevenir la exposición accidental de información crítica en los registros, minimizando el riesgo de que datos sensibles sean accesibles a través de los logs. Adicionalmente, la arquitectura de microservicios permite implementar controles de seguridad específicos para cada servicio, facilitando la segregación de datos y la aplicación de políticas de seguridad más estrictas.

9. Resultados obtenidos

Durante un año de trabajo se migraron más de 50 servicios de los cuales la mayoría ya se encuentran en producción.

Sin embargo en las estadísticas y análisis que se realizan para saber si los cambios realizados en la aplicación han fomentado el registro de nuevos usuarios a partir de que se comenzó con este proyecto de modernización han registrado que en efecto hay un 15% de nuevos usuarios que se relacionan con este cambio en la *app*, además también se analiza la cantidad de transacciones realizadas en los servicios migrados a comparación de los que ya estaban en monolito para determinar si se han utilizado con más frecuencia.

La modernización tenía como objetivo generar nuevos clientes al banco, brindar mayor y mejor experiencia de usuario para que pudieran adecuarse a la nueva tecnología con el fin de conservar y atraer nuevos clientes que tengan preferencia por los servicios que el banco ofrece.

Estas mejoras demuestran que el desarrollo que se hizo día a día tuvo buenos resultados y que sobre todo sí cumplen los objetivos de agilizar procesos, se alcanzó un aumento del 35% en el uso de transferencias.

10. Conclusiones

El desarrollo del proyecto me ha enseñado muchas cosas tanto técnicas como en el desarrollo personal, me ha ayudado a mejorar mis habilidades blandas como la comunicación, liderazgo, expresión escrita entre otras.

Fue un reto al inicio ya que desconocía completamente las arquitecturas que aquí se plantearon, sin embargo, con la práctica fui comprendiendo mejor y logré aplicarla al migrar los servicios. Con la práctica he mejorado mi desarrollo técnico simplificando código y optimizando recursos para que la ejecución de un servicio no consuma demasiado tiempo, cosa que ha ayudado a cumplir los objetivos generales del proyecto.

He aprendido de experiencias que también me han aportado en lo académico y profesional y que me han ayudado a involucrarme más en la calidad del desarrollo que realizo día a día.

Con la metodología de trabajo que describí previamente si en un futuro se decidiera añadir nuevas funcionalidades a la aplicación esto sería posible ya que la arquitectura de microservicios lo permite de forma más sencilla, para ello se agregaría está funcionalidad a un servicio existente por ejemplo si fuera un nuevo método de pago se agregaría al servicio de pago pero, si se tratara de una función totalmente nueva se generaría un nuevo servicio; una ventaja sería toda la documentación que se está generando ya que es muy detallada.

Además de la documentación el código que fuí desarrollando lo hice de tal manera que fuera descriptivo para que en caso de que en algún momento otro desarrollador lo analizara por alguna modificación o una nueva migración fuera entendible por lo que considero que es de fácil mantenimiento a futuro.

11. Referencias

- Perfil de Desarrollador Backend. (s/f). Hireline. Recuperado el 15 de marzo de 2024, de https://hireline.io/mx/enciclopedia-de-perfiles-de-tecnologia/desarrollador-backend

- ¿Qué es el desarrollo de aplicaciones móviles? (n.d.). Microsoft.com. Retrieved January 13, 2023, from https://azure.microsoft.com/es-es/resources/cloud-computing-dictionary/what-is-mobile-app-development/

- ¿Qué es la migración de aplicaciones? (n.d.). Microsoft.com. Retrieved January 13, 2023, from https://azure.microsoft.com/es-es/resources/cloud-computing-dictionary/what-is-application-migration/

- Semana. (2020, October 20). ¿Cómo funcionan las apps bancarias y qué ventajas tienen para su economía? Revista Semana. https://www.semana.com/economia/articulo/como-funcionan-las-apps-bancarias-y-que-ventajas-tienen-para-su-economia/202003/

- Atlassian. (n.d.). Comparación entre la arquitectura monolítica y la arquitectura de microservicios. Atlassian. Retrieved January 14, 2023, from https://www.atlassian.com/es/microservices/microservices-architecture/microservices-vs-monolith

- Castro, N. (2010, November 22). QAT: Quality assurance test / UAT: User acceptance testing. Blogspot.com. http://necastro.blogspot.com/2010/11/qat-quality-assurance-test-uat-user.html

- Programación orientada a objetos. (2022, August 24). Universidad Europea. https://universidadeuropea.com/blog/programacion-orientada-objetos/

- Sequencediagram.org - UML sequence diagram online tool. (n.d.). Sequencediagram.org. Retrieved January 14, 2023, from https://sequencediagram.org/

- Tutorial de diagrama de secuencia UML. (n.d.). Lucidchart. Retrieved January 14, 2023, from https://www.lucidchart.com/pages/es/diagrama-de-secuencia

- (N.d.). Amazon.com. Retrieved January 14, 2023, from https://aws.amazon.com/es/what-is/sql/

- IBM Documentation. (2023, abril 25). Ibm.com. https://www.ibm.com/docs/es/idr/11.3.3?topic=console-replicating-data-definition-language-ddl-changes
- CI/CD pipelines. (s/f). Gitlab.com. Recuperado el 17 de marzo de 2024, de https://docs.gitlab.com/ee/ci/pipelines/
- Code quality tool & secure analysis with SonarQube. (s/f). Sonarsource.com. Recuperado el 17 de marzo de 2024, de https://www.sonarsource.com/products/sonarqube/
- Sommerville, I. (2005). Ingenieria del software (M. I. A. Galipienso, A. Botia Martinez, F. Mora Lizan, & J. P. Trigueros Jover, Trads.; 7a ed.). Pearson Educacion.