



**Universidad de la República**  
Facultad de Ingeniería  
Instituto de Computación  
**Uruguay**

---

# **Implementación de Patrones de Microservicios**

Luis Ignacio Grondona  
Guillermo Aguirre  
Bruno González

---

Proyecto de Grado  
Ingeniería en Computación  
Universidad de la República

Montevideo, Uruguay, Octubre de 2021

Supervisores:      Dra. Ing. Laura González  
                                 Ing. Sebastián Vergara

# Resumen

En el tiempo reciente la arquitectura de microservicios ocupa un lugar cada vez más importante a la hora de definir e implementar sistemas de software. Surge como alternativa a otras arquitecturas, como por ejemplo la monolítica, intentando solucionar los problemas que éstas han presentado a lo largo de los años. Sin embargo, la arquitectura de microservicios introduce algunas nuevas problemáticas ya que resulta, en general, más compleja y no cuenta aún con la madurez suficiente en las prácticas definidas para guiar su adopción.

Como respuesta a esto surgen los patrones de microservicios que apuntan a definir soluciones estandarizadas a problemáticas comunes dentro de la arquitectura de microservicios. Si bien esto es un avance importante, aún existen problemas que son difíciles de resolver y otros que se derivan de la propia adopción de los patrones. En particular, se presentan desafíos en relación a cuándo es adecuado utilizar determinado patrón, cómo éste se relaciona con otros patrones, su compatibilidad con el sistema que se intenta diseñar y cómo realizar una implementación basada en dichos patrones utilizando diferentes tecnologías.

Este proyecto propone una plataforma que apunta a facilitar y disminuir la curva de aprendizaje necesaria para diseñar, implementar y poner en marcha sistemas de microservicios basados en patrones.

Primero, se hizo un análisis de requerimientos funcionales y no funcionales para la plataforma. También, se estudió el trabajo realizado en el área junto con los patrones existentes, extendiendo lo abordado en proyectos de grado anteriores.

Luego, en base a los requerimientos recabados se propuso una solución. Se definió una plataforma en la cual un usuario técnico con requerimientos sobre un sistema a construir puede obtener una recomendación de patrones de microservicios que abordan sus problemáticas y luego, en base a esa recomendación, un sistema ejecutable que implemente dichos patrones.

Finalmente, se implementó la plataforma definida. Esto involucró el desarrollo de un portal web para la interacción con el usuario final, un motor de recomendación de patrones de microservicios y un generador de un sistema de ejemplo ejecutable que contiene los patrones recomendados.

**Palabras clave:** arquitectura de microservicios, patrones de microservicios, implementación de microservicios, plataforma de microservicios, contenedores, orquestación de contenedores.

# Contenido

<b>1</b>	<b>Introducción</b>	<b>1</b>
1.1	Contexto y motivación . . . . .	1
1.2	Objetivos . . . . .	3
1.3	Aportes del proyecto . . . . .	3
1.4	Organización del documento . . . . .	4
<b>2</b>	<b>Marco Teórico</b>	<b>5</b>
2.1	Microservicios . . . . .	5
2.1.1	Descripción general . . . . .	5
2.1.2	Ventajas y Desventajas . . . . .	6
2.2	Patrones de microservicios . . . . .	7
2.2.1	Descripción general . . . . .	7
2.2.2	Relaciones entre patrones de microservicios . . . . .	8
2.2.3	Categorización de patrones de microservicios . . . . .	8
2.2.4	Descripción detallada de patrones de microservicios . . . . .	14
<b>3</b>	<b>Análisis</b>	<b>18</b>
3.1	Análisis de requerimientos . . . . .	18
3.1.1	Requerimientos funcionales . . . . .	18
3.1.2	Requerimientos no funcionales . . . . .	20
3.2	Trabajo existente . . . . .	20
3.2.1	Relevamiento de patrones . . . . .	20
3.2.2	Trabajo relacionado . . . . .	21
3.3	Resumen y conclusiones . . . . .	21
<b>4</b>	<b>Solución</b>	<b>23</b>
4.1	Descripción general . . . . .	23
4.1.1	Base de conocimiento . . . . .	24
4.1.2	Motor de recomendaciones . . . . .	27
4.1.3	Generación de solución ejecutable . . . . .	28
4.2	Arquitectura de la plataforma . . . . .	30
4.2.1	Vista de casos de uso . . . . .	30
4.2.2	Vista lógica . . . . .	32
4.2.3	Vista de desarrollo . . . . .	33
4.2.4	Vista de despliegue . . . . .	36

4.2.5	Vista de procesos . . . . .	37
4.3	Conclusiones . . . . .	41
<b>5</b>	<b>Implementación</b>	<b>42</b>
5.1	Portal Web . . . . .	42
5.1.1	Tecnologías . . . . .	42
5.1.2	Diagrama entidad relación del prototipo . . . . .	44
5.2	Base de conocimiento . . . . .	45
5.3	Recomendación de Patrones . . . . .	45
5.3.1	Generación de preguntas . . . . .	46
5.3.2	Procesamiento de respuestas . . . . .	46
5.4	Implementación de patrones . . . . .	47
5.4.1	Filtrado de patrones . . . . .	47
5.4.2	Implementación de patrones . . . . .	49
5.5	Extender la solución con nuevos patrones . . . . .	51
<b>6</b>	<b>Caso de estudio y evaluación</b>	<b>53</b>
6.1	Caso de estudio . . . . .	53
6.2	Juicio de expertos . . . . .	59
6.3	Evaluación de la solución generada . . . . .	60
<b>7</b>	<b>Conclusiones y trabajo futuro</b>	<b>62</b>
7.1	Resumen . . . . .	62
7.2	Conclusiones . . . . .	63
7.3	Trabajo futuro . . . . .	64
<b>A</b>	<b>Apéndice</b>	<b>66</b>
A.0.1	Comunicación . . . . .	66
A.0.2	Comunicación con el exterior . . . . .	67
A.0.3	Consulta de Datos . . . . .	68
A.0.4	Descomposición . . . . .	69
A.0.5	Descubrimiento de Servicios . . . . .	70
A.0.6	Despliegue . . . . .	73
A.0.7	Gestión de Datos . . . . .	75
A.0.8	Consistencia . . . . .	76
A.0.9	Implementación . . . . .	76
A.0.10	Interfaz de Usuario . . . . .	78
A.0.11	Mensajería Transaccional . . . . .	79
A.0.12	Migración hacia microservicios . . . . .	80
A.0.13	Monitoreo . . . . .	81
A.0.14	Preocupaciones Transversales . . . . .	83
A.0.15	Seguridad . . . . .	84
A.0.16	Testing . . . . .	84
A.0.17	Tolerancia a Fallos . . . . .	86
	<b>Referencias</b>	<b>87</b>

# 1 | Introducción

En este documento se presenta el proyecto de grado denominado “Implementación de Patrones de Microservicios” el cual está enmarcado en las áreas de trabajo del Laboratorio de Integración de Sistemas del Instituto de Computación.

En este capítulo se presenta la introducción al proyecto incluyendo contexto y motivación, los objetivos, los aportes que realiza y, finalmente, se expone una breve reseña sobre la organización del resto del documento.

## 1.1 Contexto y motivación

En los últimos años la ingeniería de software ha necesitado reinventarse para dar solución al creciente uso de internet. El incremento del ancho de banda, la presencia de miles de millones de usuarios activos en internet y las nuevas formas de comunicación basadas en imágenes y video de alta calidad, han obligado a diseñar arquitecturas de software resilientes y en constante evolución para soportar este volumen y garantizar la calidad del servicio [50].

Como parte de esta reinención, la arquitectura de microservicios ha cobrado mucha relevancia a la hora de diseñar sistemas escalables y robustos de mediano a gran porte. Surge como alternativa a la arquitectura monolítica y a la arquitectura orientada a servicios (SOA), intentando resolver problemas de estos esquemas.

Algunos de los inconvenientes principales que presentan las arquitecturas monolíticas son [4] [50]:

- Problemas de **escalabilidad** a la hora de atender varios miles o millones de usuarios concurrentes.
- **Modificabilidad** reducida por dependencias fuertes y propensión a errores cuando muchas personas trabajan sobre el mismo repositorio de código.
- Dificultades para entregar valor al usuario final y solucionar *bugs* con celeridad a causa de despliegues a producción poco frecuentes y riesgosos.
- Limitación para aprovechar la diversidad tecnológica y ventajas particulares que ofrece cada tecnología para solucionar problemas específicos.

- Porcentaje de *uptime* relativamente bajo debido a un único punto de falla y alto acoplamiento.

La arquitectura de microservicios aborda estos problemas enfocándose en la separación de responsabilidades y buscando desacoplar los componentes individuales de una aplicación en servicios independientes que interactúan entre sí [4]. Grandes plataformas tecnológicas como Netflix, Amazon y Google, entre otros, han estado a la vanguardia, innovando y poniendo a prueba esta arquitectura, generando así una tendencia en la industria.

Algunos requerimientos que tienen los sistemas hoy en día tales como el dinamismo y la flexibilidad son suplidos de gran manera por este tipo de arquitectura. Se facilita la independencia entre equipos en el desarrollo de los servicios. Al tener servicios más pequeños e independientes el desarrollo de éstos puede ser asignado a equipos específicos que terminan conociendo el servicio y/o producto de manera muy completa. A su vez, a diferencia de lo que sucede en arquitecturas monolíticas, al tener microservicios aislados se posibilita el uso de tecnologías distintas para su desarrollo alentando al uso de las que mejor se adapten al problema que se pretende resolver (o las que el equipo de desarrollo se sienta más cómodo utilizando) [4] [50].

A diferencia de la arquitectura monolítica que se caracteriza por su simplicidad, la arquitectura de microservicios resulta generalmente en plataformas de mayor complejidad [4]. Al tener mayor cantidad de servicios interactuando entre sí se puede tornar difícil diagnosticar dónde y por qué está ocurriendo determinado problema. Por otro lado, al basarse fuertemente en comunicaciones de red, se genera una dependencia fuerte con el funcionamiento de ésta. Además, se requieren diversas automatizaciones para poder administrar plataformas que tienden a volverse grandes. En el campo del manejo de datos en arquitecturas distribuidas se vuelve un desafío importante la transaccionalidad y propiedades como ACID (Atomicidad, Consistencia, Aislamiento, Durabilidad) que son normales en sistemas monolíticos.

A pesar de su gran utilización, la arquitectura de microservicios carece de una definición y de estándares aceptados por la comunidad. Esto se debe, en parte, porque su aparición es reciente y se está en aprendizaje y mejora continua. Por otro lado ya existen varias experiencias, soluciones a problemas comunes y buenas prácticas que vale la pena consolidar como patrones.

Los **patrones de microservicios** son soluciones probadas a problemas específicos relacionados con la arquitectura de microservicios. Son un gran avance en materia de especificación de prácticas y soluciones en el área donde referentes como Chris Richardson, Martin Fowler, entre otros, han seguido un enfoque para definir y plantear patrones a modo de guía, en algunos casos incluso ofreciendo código de ejemplo o diagramas que definen estas prácticas [18][46]. Si bien se ha hecho un esfuerzo en tratar de definir e identificar problemáticas típicas y brindar soluciones para ellas, sigue siendo un área poco desarrollada, generando una curva de aprendizaje pronunciada y una probabilidad alta de tomar malas decisiones.

Teniendo en cuenta los puntos detallados anteriormente es de interés la profundización en el tema, buscando proveer mecanismos que reduzcan la curva de aprendizaje que tiene la arquitectura de microservicios y que faciliten el diseño, implementación y puesta en marcha

de aplicaciones basadas en microservicios; en particular, mediante el uso de patrones.

En esta línea, en 2020 un Proyecto de Grado avanzó con la exploración y la construcción de un catálogo de los diversos patrones existentes. El Proyecto también brinda implementaciones de referencia para dos de estos patrones [1].

## 1.2 Objetivos

El objetivo general de este proyecto es proponer una plataforma que asista al usuario, mediante el uso de patrones, en el diseño, implementación y puesta en marcha de un sistema basado en una arquitectura de microservicios.

Para cumplir con esto, se plantean los siguientes objetivos específicos:

1. Relevar y actualizar conocimiento existente sobre patrones de microservicios aprovechando los avances logrados en el Proyecto de Grado de 2020.
2. Construir y ofrecer una base de conocimiento de patrones de microservicios a través de la cual el usuario pueda buscar, filtrar y acceder a información detallada de los patrones.
3. Desarrollar un motor de recomendaciones que, basado en las respuestas del usuario, proponga un conjunto de patrones de microservicios que aborden los requerimientos relevados.
4. Brindar implementaciones de referencia para patrones de microservicios.
5. Desarrollar mecanismos que permitan automatizar la generación dinámica de una solución ejecutable basada en los patrones recomendados.
6. Validar a través de encuestas, casos de estudio y juicios de expertos la utilidad de la plataforma ofrecida.

## 1.3 Aportes del proyecto

A continuación se destacan los principales aportes de este proyecto:

1. Base de conocimiento con descripciones de más de 45 patrones de microservicios categorizada según el área del problema que resuelve y con información complementaria como autores, tecnologías relacionadas, patrones alternativos, entre otros. Además, se provee de una interfaz gráfica administrativa que resulta amigable para la edición y agregado de nuevos patrones.
2. Cuestionario y motor de recomendaciones que a partir de un conjunto de respuestas y

una base de patrones provee una recomendación de qué patrones utilizar en un contexto específico.

3. Implementación de referencia para 17 patrones de microservicios.
4. Implementación de mecanismos que permiten la generación de una solución ejecutable que incluye los patrones recomendados e implementados.
5. Plataforma que brinda de forma integrada los elementos anteriores; la base de conocimiento, el cuestionario y motor de recomendaciones y los mecanismos para la generación de la solución ejecutable.

## 1.4 Organización del documento

El resto del documento se organiza de la siguiente forma:

En el Capítulo 2 se presenta el marco teórico donde se encuentran los principales conceptos que dan contexto y sirven como base para el entendimiento en la temática abordada.

En el Capítulo 3 se identifican las principales funcionalidades que una plataforma que asista en la implementación de la arquitectura de microservicios debería tener.

En el Capítulo 4 se presenta la solución propuesta.

En el Capítulo 5 se presenta la implementación de la plataforma en base a la solución propuesta.

En el Capítulo 6 se expone un caso de estudio y se presentan los resultados de la evaluación de la plataforma por parte de expertos en el área. Además se utiliza una herramienta para analizar calidad y buenas prácticas en soluciones generadas por la plataforma.

Finalmente, en el Capítulo 7 se presentan las conclusiones generales del proyecto así como lineamientos sobre los cuales continuar el estudio del tema a futuro.



## 2 | Marco Teórico

En este capítulo se presentan los conceptos necesarios para abordar y comprender el trabajo realizado.

### 2.1 Microservicios

En esta sección se describen los conceptos generales asociados a microservicios. Concretamente, se presenta una descripción general, ventajas y desventajas de utilizar una arquitectura de microservicios y aspectos de diseño relevantes para la misma. Por último, se presentan también patrones y buenas prácticas relacionadas a microservicios que son utilizadas en la plataforma propuesta.

#### 2.1.1 Descripción general

A pesar de su popularidad y uso, el término "microservicio" no tiene aún una definición formal. Según Chris Richardson un microservicio es "una aplicación o sistema de bajo porte o tamaño que implementa un conjunto bien específico de funcionalidades" [4].

Martin Fowler y James Lewis en su blog describen el término como "un enfoque para desarrollar una aplicación o sistema como un conjunto de pequeños servicios, cada uno corriendo en su propio proceso y comunicándose entre sí mediante mecanismos livianos". Esta definición pone foco en el tamaño y el comportamiento que deben tener los servicios. También se menciona un aspecto esencial en esta arquitectura que es la comunicación o cómo estos microservicios interactúan entre sí para cumplir los requisitos funcionales generales del sistema. Estos servicios son los que constituyen los sistemas basados en arquitecturas de microservicios [27].

El término "Arquitectura de microservicios" también carece de una definición estandarizada. Chris Richardson en su libro la define como "un estilo arquitectónico que descompone funcionalmente una aplicación en una colección de servicios". Richardson hace énfasis en que esta definición no hace mención al tamaño de los servicios, si no que lo que importa es que cada servicio tenga un conjunto específico y cohesivo de responsabilidades. También menciona que estos servicios deben ser altamente mantenibles, tener bajo acoplamiento entre sí,

así como la capacidad de ser liberados y escalar de manera independiente, entre otras características [4].

Aunque no se tenga una definición formal del término, en base a las definiciones de los distintos autores se pueden identificar una serie de características comunes sobre los servicios en este tipo de arquitectura:

- Deben cumplir funcionalidades específicas y bien definidas, normalmente orientadas a las capacidades del negocio apuntando a una alta cohesión.
- Deben comunicarse mediante mecanismos livianos y tener un bajo acoplamiento entre sí, haciendo un manejo descentralizado de los datos.
- Deben ser manejados por equipos pequeños e independientes.
- La liberación y actualizaciones de los servicios debería ser automatizada y frecuente.

### 2.1.2 Ventajas y Desventajas

La arquitectura de microservicios tiene varias ventajas dentro de las cuales se destacan las siguientes [4] [28]:

- Los servicios son pequeños y fáciles de mantener.
- Los servicios son desplegados de manera independiente.
- Los servicios pueden escalar de forma independiente, de acuerdo a la demanda concreta que tenga cada uno de ellos.
- Favorece la autonomía de los equipos de desarrollo.
- Promueve la heterogeneidad de tecnologías de desarrollo, permitiendo la adopción de tecnologías emergentes con mayor facilidad.
- Al ser un conjunto de servicios, no existe un único punto de falla.

A pesar de todas las ventajas que presentan las arquitecturas de microservicios, varios autores han afirmado que ésta no es una solución única aplicable a todos los requerimientos. Utilizar arquitecturas de microservicios presenta desafíos y problemas que deben ser tomados en consideración, lo cual implica que su selección para un sistema deba ser evaluada cuidadosamente [4] [28].

Según Martin Fowler una de las desventajas más importantes que presenta la arquitectura de microservicios es la complejidad operacional [28]. Al crecer la cantidad de servicios que deben ser gestionados y mantenidos, se genera la necesidad de la aplicación de DevOps [44] junto con la utilización de una serie de nuevas habilidades y herramientas, que presentan una curva de aprendizaje pronunciadas dado el cambio cultural que esto implica.

Otra de las desventajas que presentan las arquitecturas de microservicios según Fowler es la consistencia eventual. En primer lugar, dada la insistencia en gestión de datos descentralizada que tienen los enfoques basados en microservicios, se presentan problemas de coherencia por la demora en que se propagan las actualizaciones en todo el sistema. Por otro lado, introduce complejidad técnica dado que los desarrolladores deben tener en cuenta estos problemas a la hora de implementar sus soluciones [28].

Por último, Fowler destaca que las arquitecturas de microservicios también presentan problemas de distribución. El software distribuido tiene una gran desventaja, y es el simple hecho de ser distribuidos. Esto genera un sinnúmero de complejidades que deben ser analizadas cuidadosamente a la hora de implementar un sistema basado en arquitecturas de microservicios. Entre los problemas se destacan el rendimiento, la complejidad de implementación, y la confiabilidad ya que las llamadas remotas entre servicios pueden fallar en cualquier momento [28].

Richardson, por su lado, plantea las siguientes desventajas. En primer lugar, el autor plantea que encontrar el conjunto correcto de servicios es complejo. Utilizar muchos servicios puede resultar contraproducente y muy pocos puede devenir en un sistema con varios monolitos. Al igual que Fowler, menciona que la complejidad de tener un sistema distribuido hace que el desarrollo, la ejecución de pruebas y el despliegue sea más difícil. Por último, el autor afirma que desplegar funcionalidades que afectan a varios servicios requiere de una cuidadosa coordinación [4].

## 2.2 Patrones de microservicios

En esta sección se introduce el concepto de patrón de microservicios y se describen los patrones relevantes para el trabajo.

### 2.2.1 Descripción general

Un patrón es una solución reutilizable a un problema que ocurre en un contexto particular. Es una idea que tiene su origen en la arquitectura del mundo real y que ha demostrado ser útil en la arquitectura y el diseño de software. El patrón de diseño describe un problema que se presenta repetidamente durante el diseño del software, así como la solución [48].

La aplicación de un patrón de diseño permite a los desarrolladores reutilizarlo para resolver un problema de diseño específico. Además, ayudan a los diseñadores a comunicar conocimientos arquitectónicos, a las personas a aprender un nuevo paradigma de diseño y a los nuevos desarrolladores a evitar errores que tradicionalmente solo se han aprendido mediante experiencias costosas [48].

La arquitectura de microservicios, al igual que los demás enfoques arquitectónicos, tiene

muchas desventajas que se presentan regularmente [3] [17]. De esto se desprende la necesidad de crear soluciones estandarizadas, ya que éstas resuelven dichos problemas frecuentes, y evitan a los desarrolladores tener que invertir tiempo en implementar soluciones que ya fueron creadas y validadas.

El surgimiento de patrones de microservicios a su vez conlleva a que muchos referentes definan un lenguaje de patrones de arquitectura de microservicios, lo cual es una colección de patrones para aplicar la arquitectura de microservicios. Este lenguaje tiene dos objetivos principales, decidir si los microservicios son adecuados para su aplicación y utilizar la arquitectura de microservicios correctamente [3] [17].

Un patrón de microservicios no es más que una serie de conceptos que tienen como fin atacar un área puntual de una aplicación.

### 2.2.2 Relaciones entre patrones de microservicios

Como se comentó previamente, un patrón es una solución reutilizable a un problema. Puede ocurrir que más de un patrón resuelva el mismo problema, y a su vez dichas soluciones pueden resultar en problemas nuevos que deben ser abordados.

Estas relaciones proporcionan una guía valiosa cuando se usa un lenguaje de patrones. Se deben seleccionar patrones de forma continua y recursiva hasta llegar a patrones sin sucesor. Si dos o más patrones son alternativos, normalmente se debe optar por uno solo. El lenguaje de patrones de la arquitectura de microservicios define estas relaciones entre los patrones de la siguiente manera [4]:

- **Predecesor:** un patrón predecesor es un patrón que genera la necesidad de un nuevo patrón. Por ejemplo, el patrón de Arquitectura de microservicio es el predecesor del resto de los patrones en el lenguaje de patrones, excepto el patrón de arquitectura monolítica.
- **Sucesor:** un patrón sucesor es un patrón que resuelve un problema introducido por otro patrón. Por ejemplo, todos los patrones son sucesores del patrón Arquitectura de microservicio, excepto el patrón de arquitectura monolítica.
- **Alternativa:** un patrón alternativo es un patrón que resuelve el mismo problema que otro patrón de forma alternativa, lo que implica que estos patrones no se utilizan en simultáneo. Por ejemplo, el patrón de Arquitectura monolítica y el patrón de Arquitectura de microservicio son patrones alternativos.

### 2.2.3 Categorización de patrones de microservicios

Dado que muchos patrones ofrecen soluciones a problemas específicos y estos problemas muchas veces se encuentran en una misma área de aplicación, los referentes del área han

definido categorías que agrupan a los patrones de microservicios en estas áreas de aplicación.

Al igual que sucede con las definiciones de patrones y arquitecturas de microservicios, no existe una única definición para las categorías de patrones de microservicios. Varios autores han propuesto diferentes conjuntos de categorías con sus respectivas definiciones. En particular, este trabajo se basa en las categorías propuestas en el Proyecto de Grado 2020 [1]. La Tabla 2.1 presenta las categorías y patrones considerados en este trabajo.

A continuación se describen las categorías de patrones de microservicios.

### **Comunicación**

Dado que una aplicación creada con la arquitectura de microservicios es un sistema distribuido, la comunicación entre estos servicios del sistema se torna de vital importancia. Se deben tomar una variedad de decisiones arquitectónicas y de diseño sobre cómo se comunican sus servicios entre sí [4].

### **Consistencia**

En arquitecturas de microservicios con múltiples bases de datos, un problema recurrente es asegurar la consistencia de los datos. Esta categoría abarca patrones que intentan solucionar dicho problema.

### **Comunicación con el exterior**

Al igual que con la comunicación entre servicios, cuando uno implementa un sistema utilizando una arquitectura de microservicios debe tener especial cuidado al definir la comunicación con el exterior. Estos patrones definen mecanismos de comunicación a utilizar con los clientes externos al sistema.

### **Consulta de Datos**

En arquitecturas de microservicios, es normal que los datos estén distribuidos en cada servicio, por lo que es de interés gestionar consultas que requieran datos precedentes de varios servicios. Para esto, se requiere unir datos de los servicios necesarios de forma que satisfagan los requerimientos.

## **Descomposición**

Decidir cómo descomponer un sistema en un conjunto de servicios es en gran medida un arte. Es por esto que se definen una serie de estrategias que pueden ayudar a tomar la decisión que mejor se acople al sistema que se debe implementar

## **Descubrimiento de Servicios**

El descubrimiento de servicios consiste en resolver cómo determina un cliente de un servicio la dirección IP de otro servicio para que, por ejemplo, realice una solicitud HTTP. En una aplicación de microservicios moderna basada en la nube, las ubicaciones de las instancias de los servicios, como la dirección IP y puerto, suelen ser dinámicas. Además, el conjunto de las instancias de los servicios cambia dinámicamente debido al ajuste de escala automático, fallas y actualizaciones [6].

## **Despliegue**

Como cualquier sistema, las aplicaciones basadas en arquitecturas de microservicios requieren una cierta infraestructura donde ser desplegadas. Con el correr de los años, el proceso que realizaban los desarrolladores para desplegar el código en ambientes de producción se ha vuelto altamente automatizado. A su vez, cuando se utiliza una arquitectura de microservicios cada servicio es una aplicación pequeña, lo que significa que se tienen decenas o cientos de aplicaciones en producción. [15].

## **Gestión de Datos**

En arquitecturas de microservicios, al tener servicios distribuidos se debe decidir donde se almacenan los datos que utiliza cada servicio. A pesar de que en este tipo de arquitecturas es normal que los datos estén distribuidos en cada servicio, puede ocurrir que existan casos para los cuales sea mejor utilizar bases de datos compartidas entre los servicios [4].

## **Implementación**

Desarrollar una lógica empresarial compleja siempre es un desafío, y es aún más desafiante en una arquitectura de microservicio donde la lógica empresarial se distribuye entre varios servicios. Se deben abordar dos desafíos clave, evitar o eliminar referencias a objetos que atraviesen los límites del servicio y diseñar una lógica empresarial que funcione dentro de las limitaciones de la gestión de transacciones de una arquitectura de microservicios [8]:

### **Interfaz de Usuario**

Dado que en un sistema basado en arquitecturas de microservicios se tienen datos distribuidos dentro de los diferentes servicios, la responsabilidad de esta categoría radica en determinar cómo implementar una pantalla o página de interfaz de usuario que muestre datos de múltiples servicios .

### **Mensajería Transaccional**

Esta categoría se encuentra fuertemente relacionada a la categoría Comunicación. En este caso, la responsabilidad se enfoca en el manejo de mensajes de manera transaccional. Al implementar sistemas basados en arquitecturas de microservicios, un problema frecuente es la necesidad de publicar mensajes como parte de una transacción que actualiza la base de datos y evitar que las actualizaciones dejen al sistema en un estado inconsistente.

### **Migración hacia microservicios**

El uso de patrones dentro de esta categoría tiene sentido únicamente cuando se está partiendo de un sistema monolito. El proceso de transformación de una aplicación monolítica en microservicios es una forma de modernización de la aplicación [16]. La modernización de aplicaciones es el proceso de convertir una aplicación heredada en una que tiene una arquitectura y tecnología modernas.

### **Monitoreo**

Al tener múltiples servicios corriendo en simultáneo, surge la necesidad de monitorear a los diferentes servicios en pos de detectar anomalías en el funcionamiento y/o estado de los servicios. La supervisión y las alertas se tornan una parte clave del entorno de producción. Las métricas van desde métricas a nivel de infraestructura, como CPU, memoria y uso de disco, hasta métricas a nivel de aplicación, como latencia de solicitud de servicio y número de solicitudes ejecutadas.

### **Preocupaciones Transversales**

Esta categoría contiene a los patrones que resuelven problemas frecuentes que involucran múltiples áreas de un sistema basado en una arquitectura de microservicios.

## **Seguridad**

En una arquitectura de microservicio, los usuarios suelen ser autenticados por el API Gateway. Luego debe pasar información sobre el usuario, como identidad y roles, a los servicios que invoca. Cuando se implementa seguridad en sistemas con microservicios, no se puede mantener el contexto ni las sesiones en memoria ya que la información no es compartida entre los servicios.

## **Testing**

La arquitectura de microservicio hace que los servicios individuales sean más fáciles de probar ya que son mucho más pequeños que la aplicación monolítica. Sin embargo, en estos casos también es importante probar que los diferentes servicios funcionan correctamente en conjunto [12].

## **Tolerancia a Fallos**

En un sistema distribuido, siempre que un servicio realiza una solicitud síncrona a otro servicio, existe un riesgo constante de falla parcial. Debido a que el cliente y el servicio son procesos separados, es posible que un servicio no pueda responder de manera oportuna a la solicitud de un cliente. El servicio podría estar inactivo debido a una falla o mantenimiento, o puede estar sobrecargado y responder muy lentamente a las solicitudes. Debido a que el cliente está bloqueado esperando una respuesta, el peligro radica en que la falla pueda afectar a los clientes del cliente y así sucesivamente y causar una interrupción [6].

## **Resumen**

En la Tabla 2.1 se presenta la lista de patrones para cada categoría.



<b>Categoría</b>	<b>Patrón</b>
Comunicación	Mensajería Invocación a procedimientos remotos Orquestación Coreografía
Comunicación con el exterior	Backend for frontends API Gateway
Consistencia	Saga
Consulta de Datos	CQRS API Composition
Descomposición	Descomponer en base a las capacidades de negocio Descomponer en base a subdominios
Descubrimiento de Servicios	Descubrimiento de servicios del lado del servidor Descubrimiento de servicios del lado del cliente Registro con herramientas de terceros Registro Personal
Despliegue	Servicios en máquinas virtuales Servicio como Contenedor Service mesh Plataforma para el despliegue de servicios Despliegue sin servidores
Gestión de Datos	Una base de datos por servicio Una base de datos compartida
Implementación	Aggregates Domain Event Event Sourcing Transaction Script
Interfaz de Usuario	Client-side UI Composition Server-side page fragment composition
Mensajería Transaccional	Transactional outbox Polling Publisher Transaction Log Tailing
Migración hacia microservicios	Anti corruption layer Strangler application
Monitoreo	Exception Tracking (Seguimiento de excepciones) Seguimiento distribuido (Distributed tracing) Métricas de la aplicación Audit Logging Health Check API (API de estado de salud) Log aggregation (Acumulación de logs)
Preocupaciones Transversales	Configuración Externalizada Microservices Chasis
Seguridad	Access Token Consumer driven contract test Consumer-side contract test Service Component Test
Tolerancia de Errores	Circuit Breaker

Figure 2.1: Categorización de patrones

## 2.2.4 Descripción detallada de patrones de microservicios

A continuación se describen los patrones de microservicios de mayor relevancia para este trabajo. Para más información acerca de estos patrones o descripciones de los demás patrones relevados se puede consultar el apéndice A.

### Saga

Es un patrón que aborda el problema del manejo de transacciones en sistemas distribuidos. A diferencia del Two Phase Commit [43] el patrón Saga es una solución asíncrona a este problema. Es una secuencia de transacciones locales a cada sistema, basándose en que cada sistema puede asegurar las propiedades ACID [5].

### Una base de datos por servicio

En contraposición con el patrón Una base de datos compartida [A.0.7], aquí se propone la utilización de una base de datos por servicio. Cada base de datos se plantea independiente del resto y potencialmente con un modelo de datos específico por base de datos. A su vez, las transacciones de un servicio solo involucran su base de datos.

### Servicio como Contenedor

Este patrón propone desplegar los servicios como contenedores, donde cada instancia de un servicio se mapea como un contenedor, obteniendo así todas las ventajas que tienen los contenedores por sobre las máquinas virtuales [15] [21].

### Plataforma para el despliegue de servicios

Se propone desplegar los servicios en plataformas de orquestación de contenedores. Se encuentra fuertemente relacionado con el patrón Servicio como contenedor, y la diferencia radica en la utilización de plataformas que proveen de abstracción en lo que refiere a algunas cuestiones como por ejemplo la implementación del descubrimiento de servicios.

### Mensajería

Los clientes invocan a los servicios usando mensajería asíncrona. Cuando se usa mensajería, los servicios se comunican asíncronamente a través del intercambio de mensajes.

Una aplicación basada en mensajería generalmente utiliza los denominados canales de mensajería (brokers) aunque también puede ser asincrónica sin intermediarios [6].

### **Coreografía**

El patrón Coreografía consiste en hacer que la lógica de negocio se distribuya entre todos los servicios que se comunican entre sí, sin la presencia de un servicio que actúe como coordinador central [7].

### **API Composition**

El patrón API Composition surge de la necesidad de realizar consultas uniendo tablas de bases de datos en diferentes servicios utilizando el patrón Una base de datos por servicio. Para resolver esto, un servicio se encarga de generar una respuesta a la solicitud original agregando las diferentes respuestas de los servicios involucrados [10].

### **Descubrimiento de servicios del lado del servidor**

En el descubrimiento de servicios del lado del servidor, al realizar una solicitud a un servicio, el cliente realiza una solicitud a través de un enrutador (también conocido como balanceador de carga) que se ejecuta en una ubicación conocida. El enrutador consulta un registro de servicios, que podría estar integrado en el enrutador, y reenvía la solicitud a una instancia de servicio disponible [6].

### **Registro con herramientas de terceros**

Este patrón establece que un registrador externo es responsable de registrar y anular el registro de una instancia de servicio en el registro de servicios, ya sea cuando se inicia la instancia de servicio cuando se cierra la instancia del mismo [10].

### **Audit Logging**

Audit Logging propone loguear las acciones de los usuarios en un repositorio central dado que es útil saber qué acciones ha realizado un usuario recientemente.

### **Health Check API (API de estado de salud)**

El patrón Health Check API propone exponer un *endpoint* cuyo propósito sea notificar el estado actual del servicio. De esta manera se proporciona una herramienta para conocer en cualquier momento y en tiempo de ejecución el estado de los servicios [14].

### **Log Aggregation (Acumulación de logs)**

El objetivo que plantea este patrón es llevar un registro de la actividad de los servicios en un servidor que sea capaz de proveer alertas y capacidad de búsqueda [14].

### **Access Token**

El API Gateway autentica al usuario otorgando un token mediante el cual puede realizar invocaciones. Cuando el API Gateway recibe una invocación con un token autorizado reenvía dicho token a los servicios destino, logrando así un correcto manejo de la autenticación de los usuarios.

### **API Gateway**

Este patrón en concreto resuelve el acceso desde el cliente a las funcionalidades que se proveen a través de múltiples servicios independientes. Un API Gateway es el punto de entrada al sistema. Permite la interconexión más eficiente entre clientes y aplicaciones desarrolladas con la arquitectura de microservicios.

### **Transactional outbox**

En el contexto de que la aplicación esté utilizando una base de datos relacional, una forma sencilla de publicar mensajes de manera confiable es aplicar el patrón Transaccional Outbox. Este patrón utiliza una tabla de base de datos como una cola de mensajes temporal.

### **Transaction Log Tailing**

Este patrón sugiere publicar los cambios realizados en la base de datos siguiendo el registro de transacciones [6].

### **Configuración Externalizada**

Se establece proporcionar valores de propiedades de configuración, como las credenciales de la base de datos y la ubicación de la red, a un servicio en tiempo de ejecución.

### **Microservices Chasis**

Este patrón propone la utilización de distintos frameworks de manera conjunta teniendo como objetivo eliminar del alcance del desarrollador para no tener que encargarse de todas las preocupaciones transversales en cada microservicio.

## 3 | Análisis

Este capítulo analiza la propuesta de una plataforma que asista al usuario en el diseño, implementación y puesta en marcha de un sistema basado en una arquitectura de microservicios mediante el uso de patrones. También, se detallan los requisitos no funcionales con los que debe cumplir dicha plataforma. Además del análisis de la problemática, sus objetivos y requerimientos, se analiza el estado actual de la literatura respecto a patrones de microservicios y si existen en el mercado y/o la comunidad plataformas que provean una funcionalidad similar.

El capítulo se organiza de la forma siguiente: En la Sección 3.1 se analiza la problemática planteada, detallando los requerimientos funcionales en la Sección 3.1.1 y los requerimientos no funcionales en la Sección 3.1.2. En la Sección 3.2 se estudia el trabajo existente. En la Sección 3.2.1 se analiza la literatura disponible en busca de nuevos patrones, ejemplos de implementación y procesos de trabajo sobre arquitecturas de microservicios. En la Sección 3.2.2 se analiza el trabajo relacionado, viendo si existen plataformas que resuelvan problemas similares al planteado. Finalmente, en la Sección 3.3, se presenta el resumen y conclusiones del análisis.

### 3.1 Análisis de requerimientos

En esta sección se presenta el análisis de requerimientos sobre la propuesta realizada en el proyecto.

#### 3.1.1 Requerimientos funcionales

La propuesta inicial fue seguir el rumbo del Proyecto de Grado [1] de 2020 utilizando la información recabada en ese proyecto y ampliar su alcance. El objetivo fue diseñar e implementar una plataforma donde un usuario técnico con poco conocimiento sobre microservicios lograra, a partir de un conjunto de requisitos no funcionales, tener un sistema de ejemplo que siguiera una arquitectura de microservicios con patrones implementados que abordaran dichos requisitos no funcionales.

La clave debía ser la interacción del usuario, lograr que la plataforma devolviera algo que se ajustara a lo que éste necesitaba. La misma no solo debería permitir ajustar la recomendación

a sus requisitos, sino que también lograr que la solución ejecutable que se le devuelve al usuario sea extensible y variable tanto en tecnologías como en patrones.

De estos requisitos iniciales se pueden desprender tres funcionalidades importantes para la plataforma:

- Brindar un catálogo de patrones.
- Recomendar patrones.
- Generación de solución ejecutable de ejemplo.

El catálogo de patrones debe centralizar y categorizar el conocimiento sobre patrones de microservicios. Esto intenta proveer al usuario un conocimiento más específico de la utilidad de cada patrón y qué problema busca solucionar, con el objetivo de que se le informe cuáles son los patrones que constituirían su sistema de ejemplo. También se considera el carácter dinámico de los sistemas informáticos, con una base teórica fuerte es más probable que se tomen mejores decisiones en el futuro.

No basta solo con tener un catálogo de patrones, estos tienen que aplicarse de manera que aborden los requerimientos de cada usuario. Para esto es necesario que se le recomienden los patrones que apunten a solucionar dichos requerimientos de manera de incluirlos en el sistema de ejemplo.

También es de interés que el usuario final tenga la opción de elegir qué tipo de solución ejecutable de ejemplo quiere, seleccionando ya sea tecnología o filtrando qué patrones recomendados quiere incluir en la recomendación final. Esto permite adecuar en mayor medida la recomendación final a las necesidades del usuario, aprovechando la heterogeneidad de tecnologías que presentan las arquitecturas de microservicios.

La generación de una solución ejecutable de ejemplo es una funcionalidad clave de la plataforma, es donde se aplican las recomendaciones de patrones y se brinda un sistema que aborda sus problemáticas.

Además de brindar esta funcionalidad, se entiende que es de vital importancia permitir al usuario extender la solución de ejemplo de manera que se pueda utilizar de base para comenzar algún proyecto. Esto es debido a la complejidad que presentan las arquitecturas de microservicios, la relativa poca experiencia general en el área y que no se cuentan con prácticas ya establecidas lo cual hace que la curva de aprendizaje para comenzar a utilizar este tipo de patrones y arquitecturas sea bastante alta.

Estas tres funcionalidades fueron las que se priorizaron en el desarrollo de la plataforma.

### 3.1.2 Requerimientos no funcionales

Además de los requerimientos funcionales, la plataforma debía presentar otras características en relación a: mantenibilidad, adaptabilidad y portabilidad [55].

En cuanto a mantenibilidad, se quiso hacer énfasis principalmente en esta característica, de manera que el trabajo realizado pueda ser extendido por posibles futuros proyectos. Con respecto a adaptabilidad, la plataforma debe poder adaptarse a cambios en patrones, implementaciones y tecnologías. Finalmente su portabilidad, de manera que la plataforma pueda ser ejecutada en cualquier sistema, tanto local como en la nube, con mínimas configuraciones.

## 3.2 Trabajo existente

Esta sección detalla el análisis realizado sobre el trabajo existente sobre patrones y plataformas relacionadas.

### 3.2.1 Relevamiento de patrones

Para el relevamiento de patrones el primer paso fue analizar y recabar fuentes de información y de ellas investigar nuevos patrones que no estuvieran documentados en el Proyecto de Grado de 2020, que sirvió como guía para la lista de patrones. Para esto se consultaron fuentes y literatura tanto formal como informal, en forma de papers, libros y artículos.

Se utilizó una serie de portales de literatura científica como IEEE Xplorer<sup>1</sup>, Portal Timbó<sup>2</sup>, Scholar Google<sup>3</sup>, Springer Link<sup>4</sup>, Research Gate<sup>5</sup>. Para búsquedas menos formales se utilizó Google.

En el tiempo desde que comenzó el proyecto de grado hasta la fecha de escribir esta tesis, no se descubrieron patrones nuevos que hayan surgido que tuvieran relevancia a nivel de las fuentes. En la literatura informal se hace mención de algunos patrones nuevos, pero que son una combinación de patrones ya conocidos. Sí se encontraron patrones existentes que no fueron recabados con anterioridad, como el patrón Transaction Log Tailing, que está fuertemente relacionado al patrón Saga y ayuda en la transaccionalidad de los eventos.

---

<sup>1</sup><https://ieeexplore.ieee.org/Xplore/home.jsp>

<sup>2</sup><https://foco.timbo.org.uy/home>

<sup>3</sup><https://scholar.google.com/>

<sup>4</sup><https://link.springer.com/>

<sup>5</sup><https://www.researchgate.net/>



### 3.2.2 Trabajo relacionado

De igual manera que con la búsqueda de nuevos patrones, se buscaron plataformas que ofrecieran o tuvieran un comportamiento similar a lo que se quería lograr. Inicialmente no se encontraron otras soluciones similares. Esto cambió en Julio de 2021 cuando Chris Richardson lanzó una plataforma [19] que presenta un comportamiento similar.

Esta plataforma si bien intenta ayudar el seguimiento de buenas prácticas a la hora de diseñar una arquitectura o un microservicio, funciona de manera inversa a la plataforma propuesta en este Proyecto. Toma como entrada un microservicio ya diseñado y construido y en base a eso evalúa mediante preguntas que responde el usuario, asignando un puntaje, la aplicación de buenas prácticas y patrones que tiene el servicio. Retorna un puntaje con un resumen en forma de gráfica de cómo se distribuyen los mismos.

Si se toman en cuenta plataformas similares no asociadas directamente con microservicios, existe la plataforma de *Design Patterns* [47]. En esta plataforma se puede acceder a una gran base de conocimiento de patrones de diseño, los cuales son presentados en detalle explicando qué problemáticas resuelven, cuándo y cómo utilizarlos e incluso proveen pseudo código de referencia así como también código implementado en distintos lenguajes que el usuario puede elegir como guía. Esta plataforma es un buen ejemplo de cómo presentar el propósito e información sobre patrones proveyéndole al usuario con código que pueda usar.

## 3.3 Resumen y conclusiones

A partir del análisis de la propuesta y de productos existentes llegamos a la conclusión de que la plataforma debe constar de tres partes que persiguen diferentes funcionalidades.

Primero, tener un marco teórico o base de conocimiento sobre los patrones; qué problemática abordan, cómo funcionan, etc.

Segundo, un motor de recomendación de patrones de microservicios que, a partir de la interacción con el usuario, retorne un conjunto de patrones que conjugan buenas prácticas con respuestas a los requisitos no funcionales recabados.

Por último, generar para el usuario una solución ejecutable que implemente los patrones recomendados en base al conjunto de patrones provisto por la parte anterior.

De igual manera se definen requisitos no funcionales para la plataforma. Debe ser mantenible; debe definir e implementar las bases para que pueda ser extendida y mejorada por trabajos futuros. También es deseable que sea adaptable, de tal forma que se tenga en cuenta en el diseño la capacidad de la plataforma de adaptarse a cambios en patrones, implementaciones y tecnologías. Por otro lado, se busca la portabilidad: la plataforma debe poder ser ejecutada en cualquier sistema necesitando poca configuración.

---

En base a estos requerimientos, en el próximo capítulo se define y diseña una arquitectura en alto nivel que define y asienta las bases para la implementación del prototipo de manera que pueda ser extendido en un futuro. En particular, se define en qué consiste la interacción con el usuario y cómo se genera a través de ella una solución que aborde los requisitos no funcionales necesarios.

# 4 | Solución

En este capítulo se presenta la solución propuesta. Se define una plataforma que satisface los requerimientos detallados en el Capítulo 3. El capítulo se organiza de la siguiente forma: En la Sección 4.1 se describe de manera general la plataforma y sus partes. En la Sección 4.2 se detallan los modelos de arquitectura que describen la plataforma. A su vez, se presenta el proceso de generación y la estructura resultante de la solución ejecutable. Finalmente en la Sección 4.3 se presentan las conclusiones del capítulo.

## 4.1 Descripción general

Como fue descrito en el capítulo anterior se identificaron tres áreas de interés que la plataforma debe abordar: un marco teórico o base de conocimiento sobre los patrones, un motor de recomendación de patrones de microservicios y la generación de una solución ejecutable para el usuario que implemente un subconjunto de los patrones recomendados. A su vez se pretende que la plataforma definida cumpla con tres requisitos no funcionales: que sea mantenible, adaptable y portable.

La plataforma tiene dos partes que se diferencian por el rol del usuario: una apunta a administradores (portal administrativo) y la otra apunta a usuarios no administradores de la plataforma (portal de usuario).

### Portal administrativo

Se define un portal administrativo para la gestión de los datos de las diferentes entidades que maneja la plataforma. Este portal está orientado a administradores y no se permite el acceso a usuarios que no tengan este rol, con el objetivo de proteger el acceso a operaciones sensibles, como pueden ser la creación o borrado de los datos.

El portal permite editar, agregar, borrar información que provee la plataforma haciéndola extensible y adaptable. Se busca que la plataforma pueda evolucionar el contenido a lo largo del tiempo y en base a los requerimientos que se tengan, sin necesidad de recurrir a un perfil técnico de programación que realice los cambios.

A modo de ejemplificar qué tareas se pueden realizar en el portal administrativo, los administradores pueden agregar nuevos patrones a la base de conocimiento, como también agregar

referencias a algún patrón específico o modificar relaciones entre patrones. En caso de que se agreguen más implementaciones de patrones en un futuro, se pueden agregar las direcciones de los repositorios de código donde se almacenan dichas implementaciones.

Si bien el portal administrativo es una parte importante de la plataforma, el foco principal de la propuesta está en el portal de usuario (no administrativo).

### Portal de usuario

El portal de usuario es el foco principal de la plataforma. Esta parte de la misma es la que satisface los requerimientos identificados en el Capítulo 3.

Tiene tres subsistemas principales: la base de conocimiento, el motor de recomendación de patrones y el motor de generación de una solución ejecutable. Cada subsistema tiene como entrada la salida del subsistema anterior, entre otros componentes, como se ve en la Figura 4.1. Si bien están relacionadas, al depender de una entrada específica del subsistema anterior, mientras esa entrada sea la misma cada subsistema puede ser extendido de manera independiente.



Figure 4.1: Relaciones entre subsistemas

En las próximas subsecciones se describen estos tres subsistemas:

- Base de conocimiento
- Motor de recomendaciones
- Generación de solución ejecutable

#### 4.1.1 Base de conocimiento

Este subsistema es donde reside el conocimiento e información actual de patrones de microservicios. Los datos que se guardan siguen el modelo conceptual de la Figura 4.2 donde se pueden apreciar distintas entidades además de los patrones, que son la entidad principal

del subsistema. Los atributos que se describen en el diagrama son los principales de cada entidad. Dicho diagrama fue extendido de un diagrama de entidades del Proyecto de Grado de 2020 [1]. A continuación se brinda una breve descripción de cada entidad:

- Patrón: Es la entidad principal del subsistema.
- Autor: Distintos autores de patrones.
- Categoría: Agrupaciones de patrones con puntos en común.
- Pregunta: Una pregunta relacionada a una categoría.
- Respuesta: Respuesta a una pregunta dada, la cual se asocia a un patrón.
- Implementación: Un mismo patrón puede tener varias implementaciones.
- Tecnología: Son las distintas tecnologías asociadas a un patrón.
- Repositorio: Las implementaciones de un patrón en una tecnología se guardan en un repositorio.
- Solución: Conjunto de implementaciones de patrones de distinto repositorios.

Como se puede ver en la Figura 4.2 se reflejan algunas de las relaciones que tienen los patrones entre sí. La definición de estas relaciones se detalla en la Sección 2.2.2.

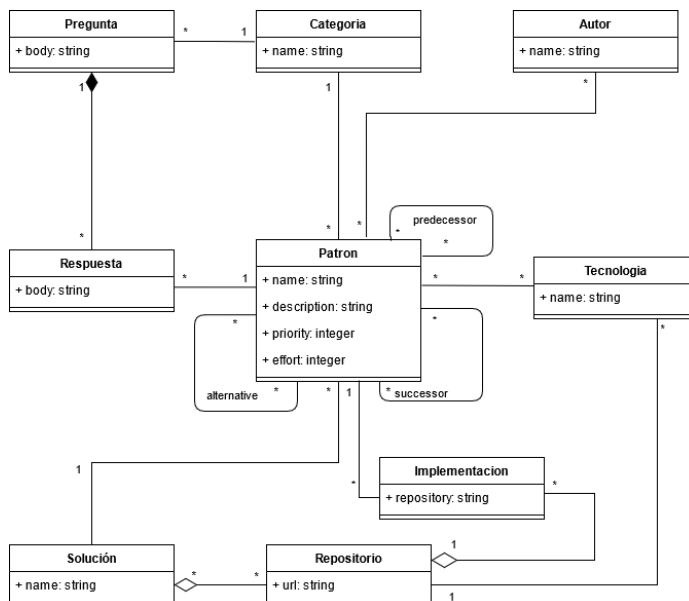


Figure 4.2: Diagrama conceptual de la estructura de datos (extendido de [1])

Otra parte importante del subsistema es la relación que tienen las preguntas y respuestas con los patrones. Estas entidades existen para poder tener una correspondencia entre las necesidades del usuario con patrones concretos. Las preguntas se asocian a una categoría, donde las respuestas solo pueden estar asociadas a patrones de esa misma categoría. Las mismas se apoyan en el conocimiento existente que se tiene sobre los patrones y su relacionamiento. Se describirá con más detalle este proceso en la siguiente sección.

Para tomar las decisiones sobre qué pregunta se asociaba a qué patrones se siguió un modelo de diagramas de decisión [2] y se crearon diagramas sobre cada categoría de patrones. A modo de ejemplo en las Figuras 4.3 y 4.4 se presentan algunos diagramas sobre las categorías de mayor importancia.

La Figura 4.3 muestra las distintas opciones y decisiones que llevan a recomendar un patrón u otro. En caso de querer o necesitar almacenar los datos de manera individual por servicio, se recomienda el patrón Una base de datos por microservicio. Esto puede presentar la necesidad de mantener la consistencia de los datos, por lo cual se recomienda el patrón Saga.

En el caso de la Figura 4.4 se detalla la dependencia entre determinados patrones. En caso de decidir ya sea por el patrón Mensajería o Invocación a procedimientos remotos es necesario contar con un patrón de coordinación. Esto es una ejemplificación de las relaciones definidas en la Sección 2.2.2.

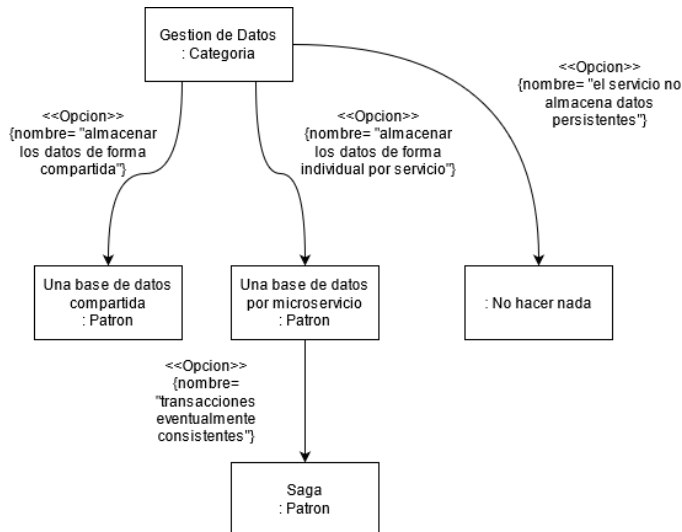


Figure 4.3: Diagrama de decisión sobre la gestión de datos

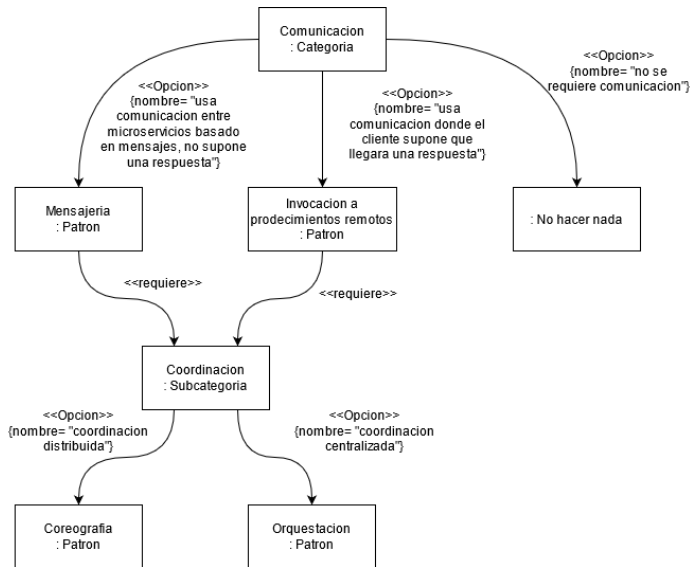


Figure 4.4: Diagrama de decisión sobre comunicación

Además del relacionamiento de los patrones estos tienen una implementación. Cada patrón puede tener implementaciones en distintas tecnologías, las que se almacenan en un repositorio de código. Estos conjuntos de repositorios son los que componen la solución ejecutable que el usuario puede utilizar.

#### 4.1.2 Motor de recomendaciones

El motor de recomendaciones de patrones se basa fuertemente en el relevamiento realizado para el subsistema de la base de conocimiento. De éste se desprende el relacionamiento entre los patrones y la asociación de preguntas y respuestas con patrones específicos.

Este subsistema tiene como entrada un conjunto de patrones, actualmente producto de la asociación de las respuestas con los patrones del subsistema anterior. Se recibe un conjunto de patrones, se realiza un procesamiento sobre ellos y se devuelve un conjunto nuevo de patrones el cual puede ser distinto al conjunto original.

El procesamiento principal que se hace sobre los patrones es el de identificar las relaciones que estos tienen entre sí y, en base a ellas, junto a una combinación entre la prioridad y esfuerzo asociado a implementar el patrón, se van agregando o quitando patrones del conjunto. Uno de los puntos fundamentales de este subsistema es que se encarga de evitar inconsistencias entre los patrones, es decir, que no haya patrones alternativos dentro de un mismo conjunto y que en caso de existir un patrón, los patrones predecesores a éste se encuentren

dentro del conjunto.

### 4.1.3 Generación de solución ejecutable

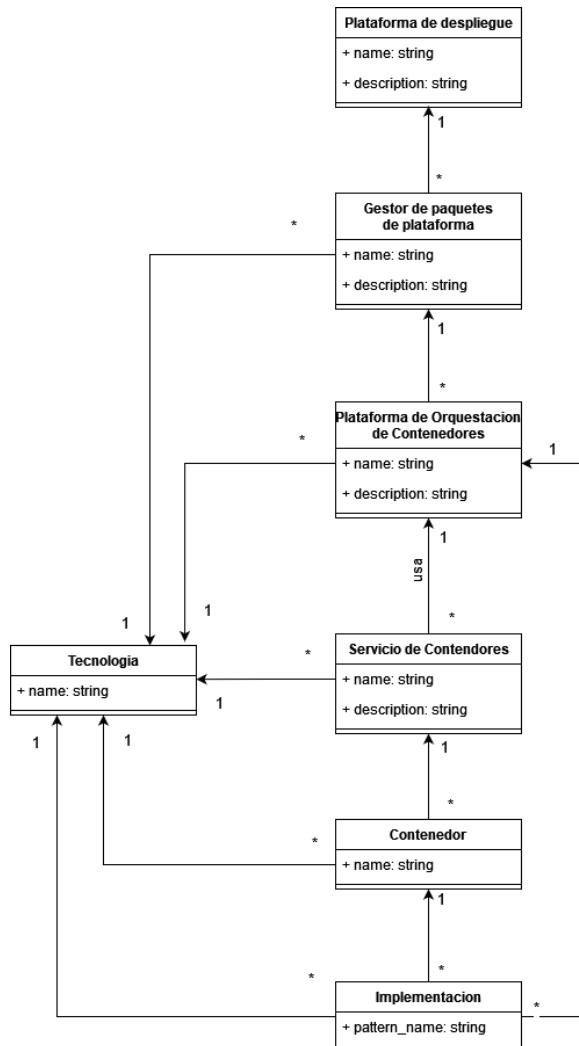


Figure 4.5: Diagrama de solución ejecutable

Este subsistema recibe un conjunto de patrones y retorna una solución ejecutable que incluye esos patrones implementados para que sea usada por el usuario. La solución es ejecutable



sobre cualquier sistema operativo independientemente de qué patrones hayan sido recomendados. Para esto se diseñó un subsistema que hace uso de contenedores [40] para alojar las implementaciones de manera aislada e independiente.

Como se puede ver en la Figura 4.5 estos contenedores son manejados y son capaces de interactuar entre sí mediante una plataforma de orquestación de contenedores [37]. Como existen patrones que se enfocan en la configuración y despliegue de las plataformas, es necesario tener un nivel más de abstracción. Para esto se utiliza el gestor de paquetes de plataforma y la plataforma de despliegue que permiten manejar y configurar de manera más variable distintos aspectos de la plataforma de orquestación de contenedores.

A modo de ejemplo, en la Figura 4.6 se detalla una instanciación del diagrama descrito, que es la que se utiliza para la implementación del prototipo (ver Capítulo 5). En esta instanciación, se utiliza Helmfile<sup>1</sup> como plataforma de despliegue, Helm<sup>2</sup> como gestor de paquetes de plataforma, Kubernetes<sup>3</sup> como plataforma de orquestación de contenedores y Docker<sup>4</sup> como servicio de contenedores. En una solución ya instanciada en cada contenedor se puede encontrar una aplicación web de ejemplo con algunos patrones implementados (dependiendo de la recomendación), bases de datos, servicios de colas de mensajes, entre otros.

Con este diseño la premisa es que se puede desplegar un sistema que implemente distintos tipos de patrones de microservicios, de manera que sea agnóstico a qué patrones sean implementados, cómo son implementados y sobre qué sistema deben ser ejecutados. Esto permitiría recibir un conjunto de patrones distinto cada vez y que se devuelva una solución específica para ese conjunto de patrones pero manteniendo la misma forma de despliegue.

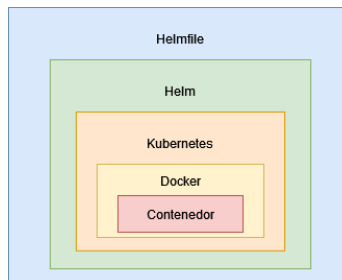


Figure 4.6: Diagrama de solución ejecutable

<sup>1</sup><https://github.com/roboll/helmfile>

<sup>2</sup><https://helm.sh/>

<sup>3</sup><https://kubernetes.io/>

<sup>4</sup><https://www.docker.com/>

## 4.2 Arquitectura de la plataforma

En esta sección se detalla la arquitectura que soporta la plataforma descrita en la sección anterior. Para hacerlo se utiliza el modelo de vistas 4 + 1 [49]. El objetivo es tomar lo descrito en la sección anterior sobre los subsistemas y plasmarlo de una forma menos abstracta, de manera que se mantenga el concepto de los tres subsistemas, pero que se desglose e instancie cada parte que conforma la plataforma de manera más concreta.

### 4.2.1 Vista de casos de uso

La vista de casos de uso detalla las funcionalidades que provee la plataforma, describe las secuencias de interacciones entre los objetos y los procesos del sistema. En la Figura 4.7 se presentan los casos más relevantes.

El primer caso de uso es el de *recomendar patrones*. Como se detalló en la sección anterior existe un motor dedicado a esto. Luego de responder el cuestionario, se procesan las respuestas y se devuelve una recomendación inicial de patrones.

El caso de uso de *responder cuestionario* consiste en una de las interacciones más importantes de la plataforma en la que un usuario no administrativo contesta una serie de preguntas con el objetivo de recabar los requerimientos que debe tener el sistema que se propone construir.

Luego viene el caso de uso de *filtrar recomendación*, donde el usuario puede filtrar los patrones que no desea tener en la recomendación final y quedarse con los que si desea.

El caso de uso siguiente en el flujo esperado para el usuario es el de *guardar recomendación*. Este caso de uso es la continuación del anterior, el usuario ya tiene una recomendación final de patrones y desea guardarla. Para esto es necesario que inicie sesión en la plataforma, lo que habilita que la recomendación final de patrones pueda ser guardada en el perfil del usuario.

Finalmente, se muestra el caso de uso de *descargar solución*. Este refleja la creación de la solución ejecutable, que sería el paso final en el flujo completo que provee la plataforma. Luego de guardar la solución, el usuario puede descargar una implementación de esa solución.

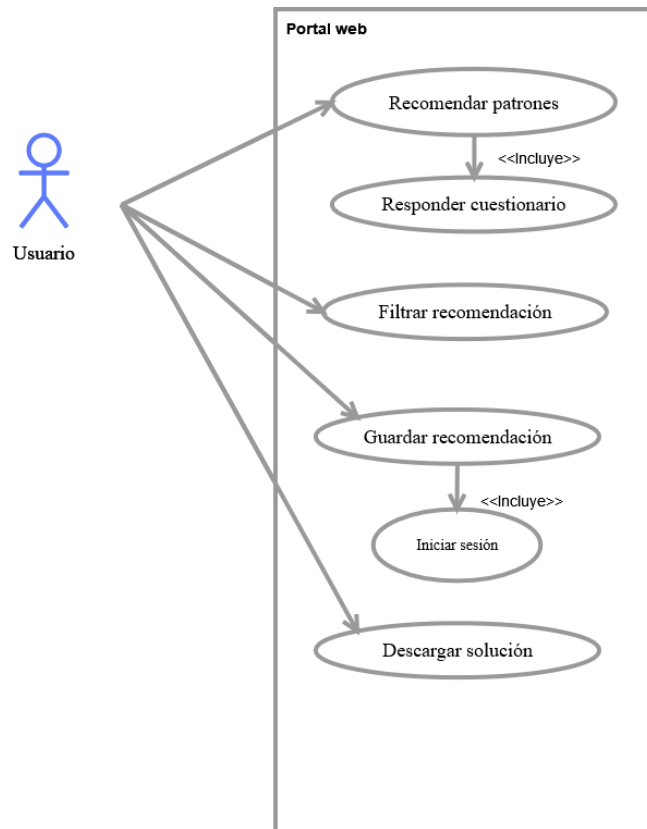


Figure 4.7: Vista de casos de uso

## 4.2.2 Vista lógica

La vista lógica es la que describe las entidades y componentes del sistema que soportan los casos de uso presentados en la sección anterior. Los componentes que se presentan en la Figura 4.8 son un desglose de las entidades de los subsistemas descritos previamente.

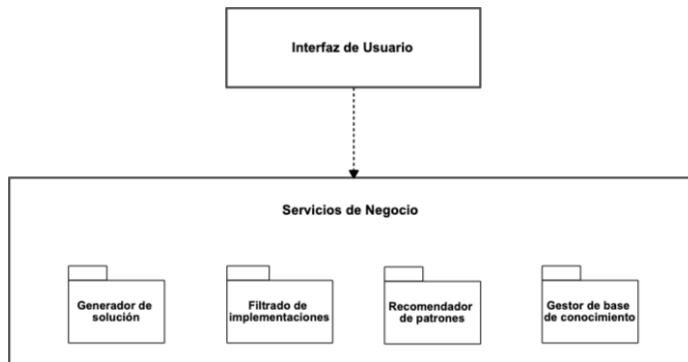


Figure 4.8: Vista lógica

Se puede ver que la *Interfaz de usuario* es lo que conecta los distintos componentes. Es a través de ésta que se interactúa con los datos, la recomendación de patrones y cómo se le presentan las salidas al usuario final.

El componente *Gestor de base de conocimiento* es el que se encarga de gestionar los datos, por ejemplo de persistir las reglas de relaciones entre patrones, entre otras cosas. Forma gran parte de la lógica del portal administrativo, el cual es la vía para hacer modificaciones a la base de conocimiento.

El componente *Recomendador de patrones* es el que toma como entrada la información referida a los patrones y el cuestionario, y aplica las reglas definidas para conseguir una recomendación de patrones consistente.

A partir de la recomendación de patrones es que se empieza a generar la solución. El componente *Filtrado de implementaciones* se encarga del primer paso de esto. Recibe una recomendación de patrones, la cuál utiliza para filtrar y dejar solo las implementaciones de patrones que se deseen en la solución final.

Finalmente el componente *Generador de solución* es el que se encarga de unir tanto las implementaciones de patrones, como las configuraciones y paquetes necesarios para retornar una solución ejecutable.

### 4.2.3 Vista de desarrollo

En esta sección se presentan las vistas de desarrollo, que detallan la separación en componentes y servicios de software de las entidades del sistema. En primer lugar, la Figura 4.9 muestra el modelo de la aplicación web que lleva a cabo las funcionalidades principales del sistema. Luego en la Figura 4.10 se presenta la estructura general de un microservicio con patrones implementados. Finalmente en la Figura 4.11 se presenta el diseño de implementación de las soluciones de microservicios generadas por la plataforma en base a las implementaciones de patrones, la cual incluye el diagrama de la Figura 4.10.

Se pueden observar dos tipos de subsistemas en la Figura 4.9, la *Aplicación web* y los *Repositorios de código*. El subsistema de la aplicación es el que se encarga de los casos de uso de *Recomendación de patrones* y *Guardar solución*, además del portal administrativo entre otras funcionalidades. Los subsistemas de repositorios de código son los encargados de alojar las distintas implementaciones de patrones y configuraciones necesarias para la generación de una solución ejecutable.

Se define el portal como una aplicación monolítica debido a que los requerimientos tanto funcionales como no funcionales se ajustan a este tipo de arquitectura. Por dentro está diseñado utilizando el patrón MVC [41] separando la lógica en *vistas*, *controladores* y *modelos*. Los servicios son los encargados del procesamiento de patrones y de generar la solución, de esta manera se separan las responsabilidades dentro del subsistema.

Las vistas componen la *Interfaz de Usuario*. El *Gestor de base de conocimiento* está compuesto por los modelos. El servicio *repository\_processing\_service* es el que engloba a los componentes de *Filtrado de implementaciones* y *Generador de solución*. El *Recomendador de patrones* es el servicio *pattern\_post\_processing\_service*.

Los subsistemas de *repositorio de código* son los que almacenan el código y permiten su descarga. Estos deben soportar la descarga del código de manera rápida y con una configuración simple, permitiendo agregar restricciones de seguridad como la autenticación de los pedidos y de quién o qué pueden ejecutar dicha funcionalidad.

En la Figura 4.10 se describe el diseño que tienen los templates de microservicios. Todos los microservicios que formen parte de la solución tienen el mismo diseño, la diferencia entre sí pueden ser los patrones implementados que los componen. Para algunas soluciones ejecutables algunos microservicios serán iguales entre sí y para otras no. Por más que tengan distintos patrones en la implementación todos siguen el esquema de patrones relacionados descrito en la figura.

Se utiliza un enfoque de abajo hacia arriba (bottom-up en inglés) para la implementación de patrones relacionados. Teniendo el conocimiento de las dependencias entre patrones se van construyendo, de manera incremental, partiendo del patrón base que no necesite de otros patrones para cumplir con su funcionalidad (que no tenga patrones predecesores). Agregando a la ejemplificación de la Sección 2.2.2 donde se define la relación entre patrones, primero se agregaría el patrón p1 y luego su patrón sucesor p0. Este diseño se apoya fuertemente en los

pasos anteriores de filtrado de patrones, donde se asegura que no van a existir inconsistencias entre los patrones, de manera que todos los patrones que componen una solución van a poder convivir en sincronía dentro de un mismo microservicio.

La Figura 4.11 muestra la estructura del diseño del despliegue de la solución ejecutable retornada al usuario. Se puede apreciar que se compone de diferentes microservicios que interactúan entre sí para implementar distintos patrones. Cada uno de estos microservicios lo podemos ver como una instancia del microservicio template de la Figura 4.10.

Por otro lado, existen determinados tipos de patrones que no forman parte de los microservicios de por sí, sino que actúan a un nivel más alto de abstracción como son las plataformas de orquestación y configuraciones que se presentan en el despliegue de este tipo de sistemas. El concepto y aplicación para este tipo de patrones es el mismo, se asegura que no hayan inconsistencias mediante el procesamiento previo y, en caso de haber patrones dependientes se utiliza el mismo procedimiento descrito anteriormente.

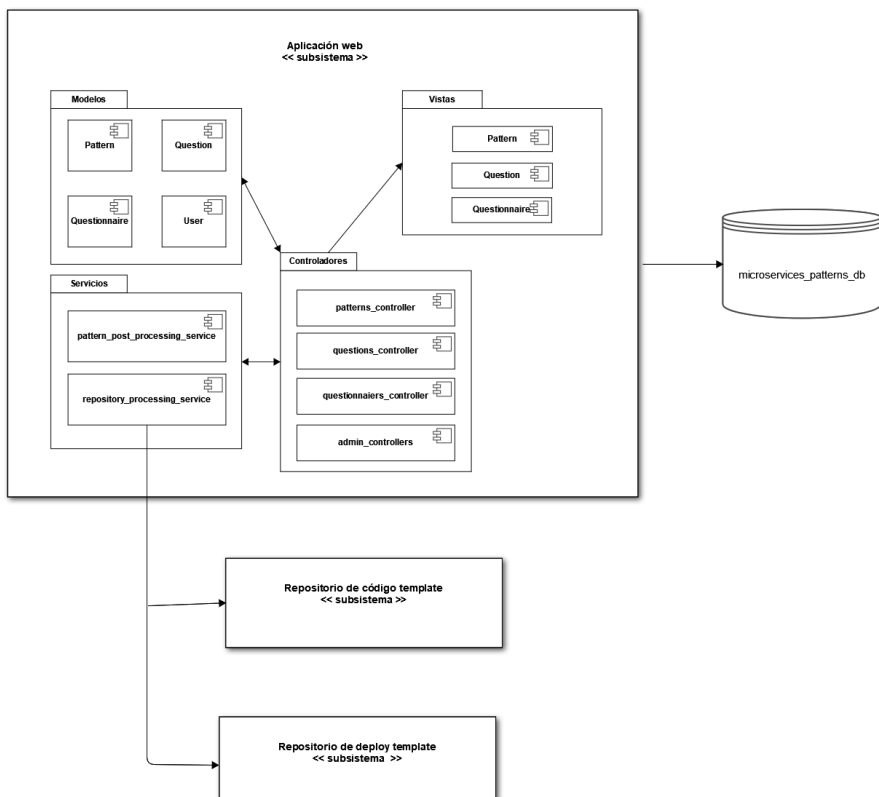


Figure 4.9: Vista de desarrollo: portal web

**pj son números naturales**

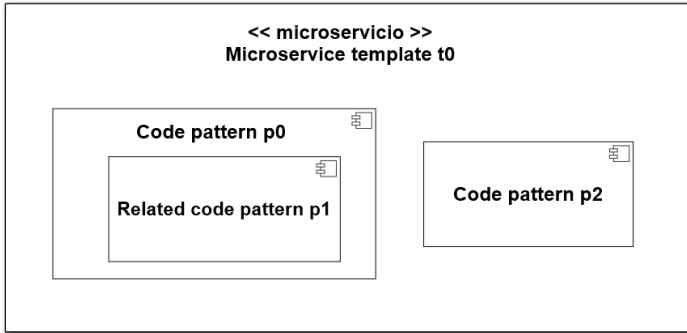


Figure 4.10: Vista de desarrollo: template de microservicio

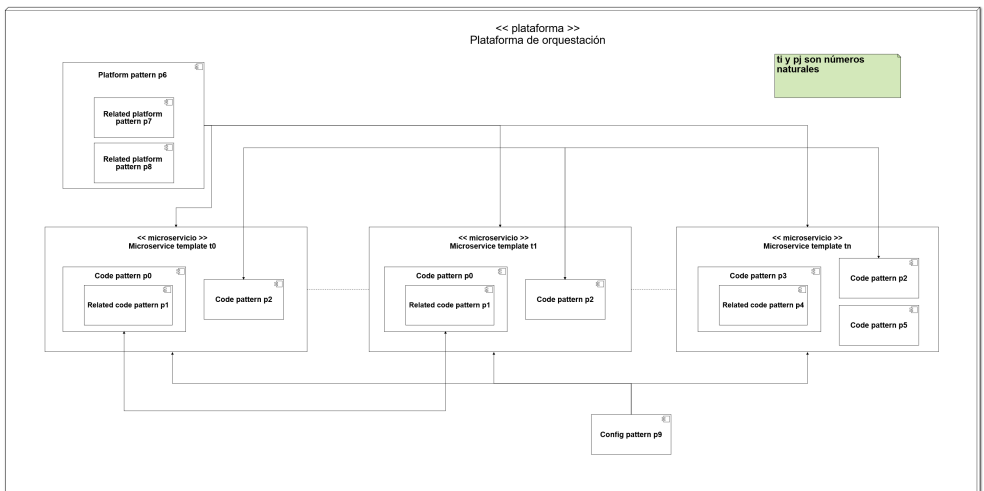


Figure 4.11: Vista de desarrollo: despliegue de solución ejecutable

#### 4.2.4 Vista de despliegue

La vista de despliegue ilustra cómo los diferentes componentes de *hardware* del sistema se relacionan entre ellos, marcando la relación entre los componentes de *hardware* y *software*. Se muestran dos diagramas, uno correspondiente al *portal web* y otro correspondiente al *despliegue de la solución*.

Como se detalla en la Figura 4.12 para desplegar el portal web se necesita un servidor expuesto a Internet. Esto se empaqueta en un contenedor definido en el mismo repositorio para hacer más sencillo el despliegue. El contenedor ejecuta la aplicación web y un cliente de base de datos.

El diagrama de la Figura 4.13 refleja el despliegue de la solución ejecutable generada. Se necesita una plataforma de orquestación de contenedores, ya que todos los microservicios están empaquetados en contenedores. Esto es para mayor facilidad a la hora de configurar y desplegar diferentes servicios que pueden ser muy heterogéneos. Los microservicios, dependiendo del patrón de gestión de datos que se utilice, se pueden comunicar con una base de datos individual o compartida.

Todo microservicio tiene una aplicación web que implementa las funcionalidades de los patrones que componen la solución y unas configuraciones que permiten desplegar dichos microservicios de manera dinámica. A su vez existen algunos patrones que están estrictamente relacionados a cómo es que se configuran dichos despliegues.

De igual manera que existen configuraciones a nivel de cada microservicio, existen configuraciones más generales a nivel de la plataforma de orquestación. Esto ayuda con el despliegue dinámico y variable de cada microservicio.

Todo el sistema está diseñado para ser desplegado de manera agnóstica a la plataforma de despliegue. Esto significa que puede ser desplegado tanto en una plataforma en la nube como en un servidor propio e incluso una computadora local. Todo esto es posible debido al uso de contenedores y plataformas de orquestación que independizan y aíslan cada servicio de manera que pueden ser desplegados, con las configuraciones pertinentes, en distintos tipos de plataformas.

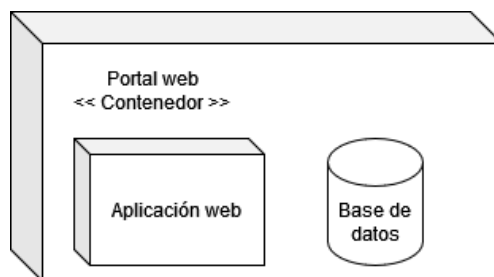


Figure 4.12: Vista de despliegue del portal web



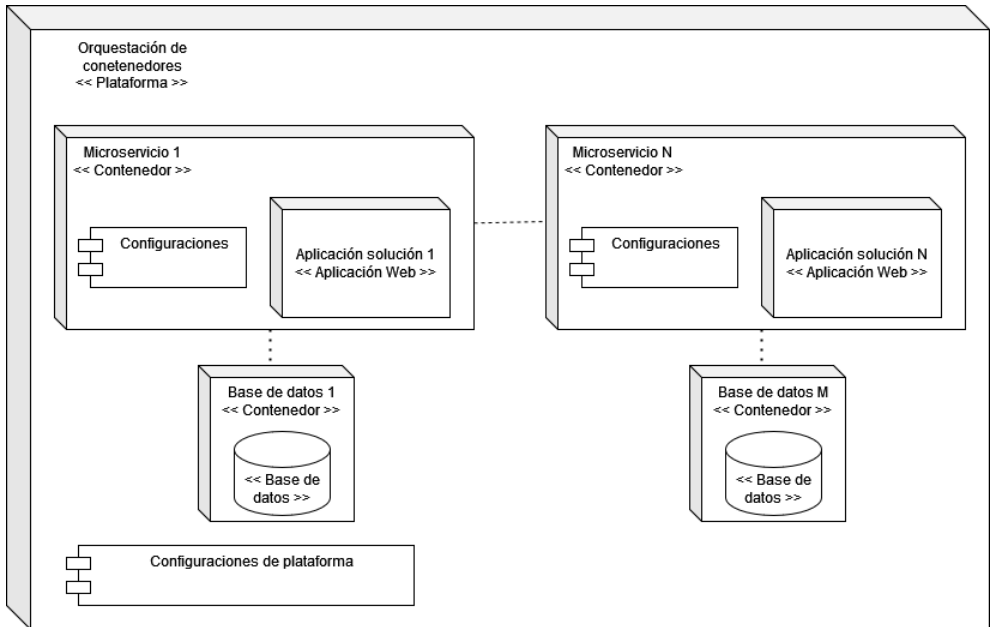


Figure 4.13: Vista de despliegue de la solución ejecutable

### 4.2.5 Vista de procesos

La vista de procesos representa los flujos paso a paso del sistema, de cómo se resuelve la comunicación y ejecución entre procesos y componentes. En esta sección se presentan la vista de procesos describiendo los dos procesos de mayor interés del sistema.

#### Flujo principal del sistema

En el diagrama de la Figura 4.14 se detalla el flujo principal del sistema, el cual va desde el completado del formulario hasta la devolución e instalación de una solución ejecutable. Lo componen las entidades y componentes previamente introducidos en secciones previas y aquí se puede ver cómo se conectan entre sí para producir el resultado esperado.

El flujo comienza cuando el usuario responde el cuestionario. A partir de esto, el *Portal web* se encarga de procesar las respuestas y generar una recomendación de patrones. Esta lista de patrones recomendados es presentada al usuario en la forma de una *Recomendación intermedia*, que el usuario puede guardar. Para hacerlo debe tener un usuario registrado; en caso de no tenerlo puede registrarse y luego iniciar sesión con el mismo. Si el usuario guarda o no la recomendación no afecta al paso siguiente, donde el usuario puede descartar patrones de la lista a su conveniencia o no, generando una recomendación final de patrones.

Si el usuario está registrado y decide guardar la recomendación entonces puede acceder a la opción de descargar una solución ejecutable que contenga los patrones de la recomendación final. Si el usuario no está registrado, la recomendación se perderá una vez que éste abandone el portal web.

Cuando el usuario decide descargar la solución ejecutable comienza el proceso de generación de la misma. Se detallará este proceso en más detalle con la Figura 4.15. El portal web procesa los repositorios de código necesarios, clonando el código fuente para luego filtrar los patrones que no están en la recomendación. Los repositorios de código contienen el código template de los microservicios y de la configuración para el despliegue de la plataforma, donde están implementados una variedad de patrones que pueden o no estar en una solución final. Luego del filtrado se genera un archivo comprimido con las carpetas y archivos necesarios y se le devuelve al usuario para que éste pueda descargarlo a su computadora localmente. El usuario entonces sigue los instructivos provistos para instalar dependencias, configurar y ejecutar la solución descargada en donde lo desee.

### **Generación de solución ejecutable**

Se describe con mayor detalle el proceso de la generación de la solución descrito previamente. Se puede observar en la Figura 4.13 que el generador está diseñado de manera que se puedan tener  $N$  microservicios distintos con patrones y configuraciones distintas. El proceso es dinámico y durante el comienzo se determina cuál es la cantidad necesaria de microservicios que se tienen que generar.

Se comienza generando la carpeta principal de la solución. Luego, clonando el código de la configuración para el despliegue. Una vez que se tiene esto, considerando los patrones que deben formar parte de la solución se determina el número de microservicios distintos que se necesitan; esto es el valor de  $N$ .

Cuando se define el número se comienza un proceso iterativo donde se van generando todas las carpetas y archivos de los microservicios necesarios. Este proceso es el mismo para todos, solo que aquí la recomendación de patrones se separa por microservicio, es decir que un subconjunto de patrones de la recomendación es el que compone el microservicio 1, otro subconjunto el microservicio 2 y así hasta llegar a los  $N$  microservicios. Esto no significa que vayan a faltar patrones en la solución final, si no que debido a las características de distintos patrones se necesitan más de un solo microservicio para que el patrón se considere implementado. Como fue mencionado antes, este proceso se repite  $N$  veces hasta que se tiene una carpeta final con todo el código, archivos y configuraciones necesarias para levantar todos estos microservicios y sus dependencias de manera local. Esta carpeta es comprimida y devuelta al usuario para su uso.

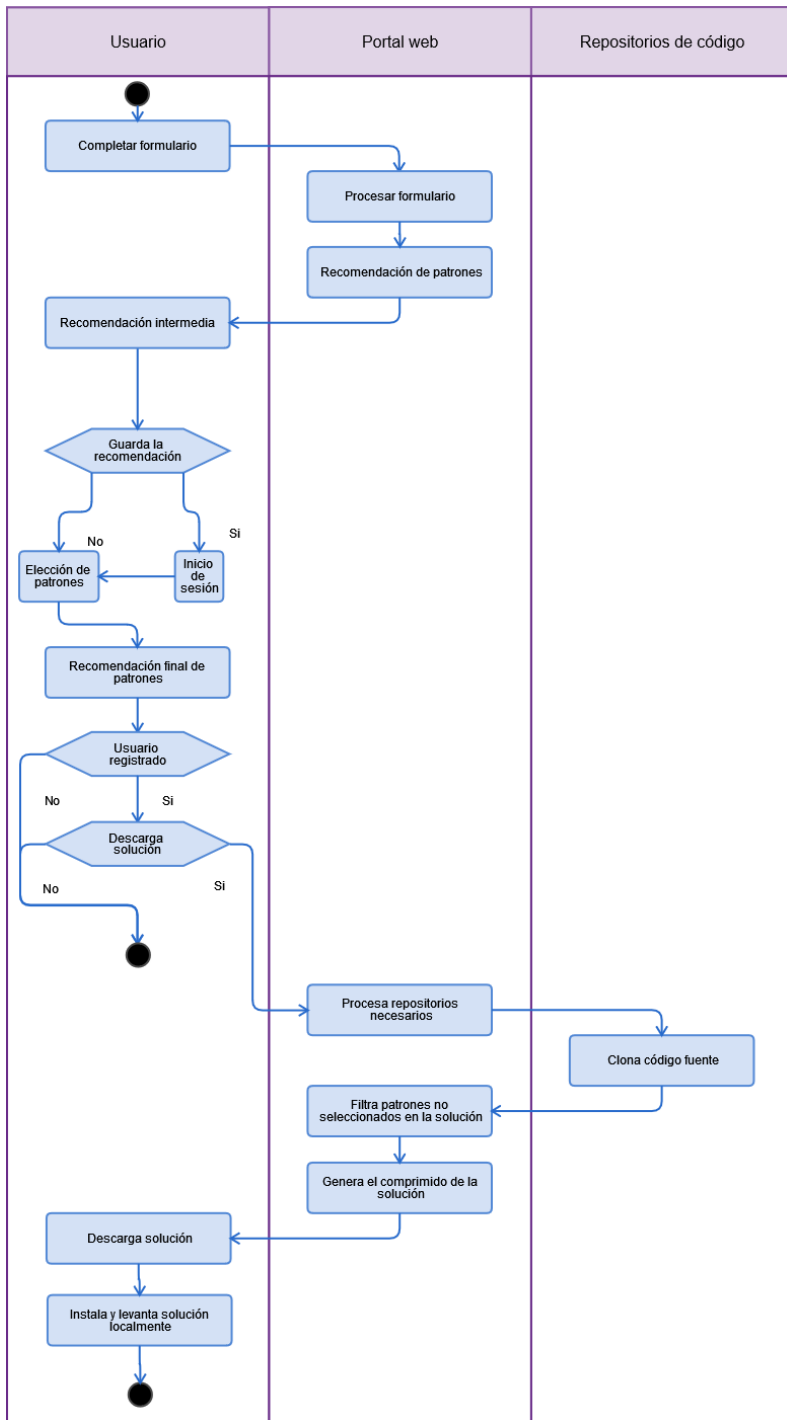


Figure 4.14: Vista de proceso de flujo principal

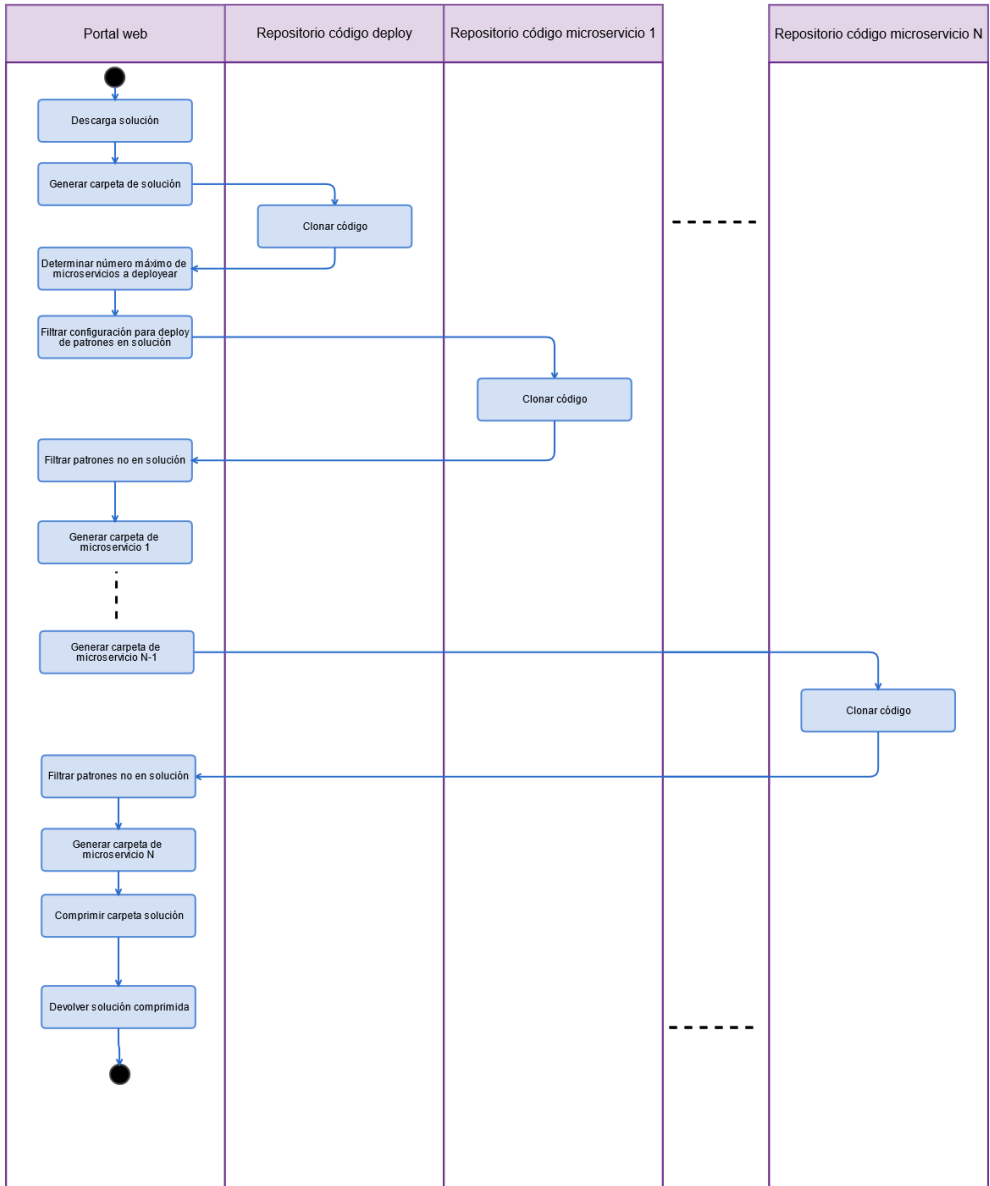


Figure 4.15: Vista de proceso de generación de la solución ejecutable

## 4.3 Conclusiones

En este capítulo se detallaron y describieron las propuestas de diseño y arquitectura que componen la plataforma.

Primero se hizo una descripción general del sistema y solución propuesta, describiendo en alto nivel los distintos subsistemas que componen la plataforma, cuál es su propósito y cómo estos interactúan entre sí. Se marca la diferencia entre los portales para distintos tipos de usuarios; por un lado un portal administrativo con fines de gestión de datos y por otro el portal no administrativo que es el que provee las funcionalidades principales de la plataforma. Dentro de la descripción de los tres subsistemas se hace referencia al comportamiento y funcionamiento esperado de la plataforma.

Haciendo uso del modelo 4+1 se hizo énfasis en distintos tipos de vistas que ayudan a describir la propuesta hecha pasando por distintos niveles de granularidad. Se introdujeron los conceptos del procesamiento y filtrado de patrones, el de la solución ejecutable, entre otros. Se detalló cómo estos componentes conforman la plataforma y cómo interactúan entre ellos para lograr las funcionalidades planteadas.

Se mostró un diseño y una arquitectura que cumple con los requerimientos funcionales detallados en el Capítulo 3 ya que define una base de conocimiento que alberga el marco teórico de los patrones y sus interacciones. A su vez plantea un motor de recomendación de patrones y también le brinda al usuario final una solución ejecutable, ya que puede descargar todos los archivos y código necesario para ejecutar localmente.

Se cumple también con los requerimientos no funcionales. La solución propuesta es mantenible, ya que cada subsistema y sus respectivos componentes pueden ser mejorados de manera individual y conjunta. A su vez se cumple con el requisito de tener una plataforma adaptable, de manera de poder soportar cambios en patrones, implementaciones y tecnologías ya que el diseño no está atado a ninguna instancia específica de esas entidades. También la plataforma es portable ya que dada la infraestructura y como se construye, la misma puede ser ejecutada en cualquier tipo de sistema.

# 5 | Implementación

Este capítulo brinda detalles del prototipo implementado de la solución propuesta. El prototipo cuenta con tres grandes componentes: la base de conocimiento, el motor de recomendación de patrones y, por último, el generador de una solución ejecutable con un conjunto de patrones recomendados implementados.

Estos componentes se presentan al usuario a través de un portal web.

## 5.1 Portal Web

En el portal, los usuarios interactuarán con la plataforma encontrando, al ingresar, una base de conocimiento de patrones de microservicios ordenada, categorizada y en la que se pueden realizar búsquedas. También se encuentran videos tutoriales de uso de la plataforma, tanto del cuestionario como de la solución.

Si un usuario está planificando la construcción de un sistema basado en microservicios, puede responder un cuestionario que recabará requerimientos no funcionales del sistema. Estos requerimientos tienen como objetivo servir de base para la recomendación de un conjunto de patrones de microservicios que aborden dichos requerimientos.

Por último, si el usuario lo desea, puede obtener una solución ejecutable que implementa algunos de los patrones recomendados en la etapa anterior.

Vale la pena destacar que la plataforma cuenta con dos tipos de usuarios: usuarios y administradores. Un usuario se puede registrar en cualquier momento mientras que los administradores deben ser creados por un administrador existente.

### 5.1.1 Tecnologías

El portal web se implementó en el lenguaje de programación Ruby. "Ruby es un lenguaje de programación dinámico y de código abierto enfocado en la simplicidad y productividad." <sup>1</sup>. La versión de Ruby utilizada fue la 2.7.1.

---

<sup>1</sup><https://www.ruby-lang.org/es/>

A su vez, se utilizó el framework web Rails, "un marco de desarrollo de aplicaciones web escrito en el lenguaje de programación Ruby. Este framework está diseñado para facilitar la programación de aplicaciones web al hacer suposiciones sobre lo que todo desarrollador necesita para programar"<sup>2</sup>. En particular, se utilizó la versión 6.1.3.1 de dicho software.

El portal web cuenta con una base de datos relacional, para la cual se utilizó el motor de base de datos PostgreSQL. "PostgreSQL es un potente sistema de base de datos relacional de objetos de código abierto con más de 30 años de desarrollo activo que le ha ganado una sólida reputación por su confiabilidad, solidez de funciones y rendimiento"<sup>3</sup>.

Por otro lado, para implementar el panel administrador se optó por usar la gema<sup>4</sup> Active Admin. "Active Admin es un *plugin* de Ruby on Rails para generar interfaces de estilo de administración. Extrae patrones de aplicaciones comerciales comunes para que los desarrolladores puedan implementar interfaces con muy poco esfuerzo"<sup>5</sup>.

Para utilizar el tiempo de manera eficiente teniendo en cuenta los objetivos del proyecto se decidió utilizar el template SB Admin 2<sup>6</sup> de Bootstrap<sup>7</sup> para la interfaz gráfica. Este template cuenta con recursos HTML y también de diseño en CSS.

Por último, también se incluyeron tests automatizados (unitarios y de integración) utilizando RSpec<sup>8</sup> y analizadores de código estático como Rubocop<sup>9</sup>, Reek<sup>10</sup>, Brakeman<sup>11</sup> y Rails Best Practices<sup>12</sup>.

RSpec es una herramienta de desarrollo basada en el comportamiento para programadores de Ruby. BDD es un enfoque al desarrollo de software que combina el desarrollo basado en pruebas (TDD, por sus siglas en inglés), el diseño basado en dominios, y planificación basada en pruebas de aceptación. RSpec se centra en los aspectos de documentación y diseño de TDD.

Se decidió optar por estas tecnologías debido a la experiencia previa con la que se contaba en el equipo haciendo uso de las mismas.

---

<sup>2</sup><https://guides.rubyonrails.org/>

<sup>3</sup><https://www.postgresql.org/about/>

<sup>4</sup><https://guides.rubygems.org/what-is-a-gem/>

<sup>5</sup><https://activeadmin.info/>

<sup>6</sup><https://startbootstrap.com/theme/sb-admin-2>

<sup>7</sup><https://getbootstrap.com/>

<sup>8</sup><https://relishapp.com/rspec/>

<sup>9</sup><https://github.com/rubocop/rubocop>

<sup>10</sup><https://github.com/troessner/reek>

<sup>11</sup><https://github.com/presidentbeef/brakeman>

<sup>12</sup>[https://github.com/flyerhzm/rails\\_best\\_practices](https://github.com/flyerhzm/rails_best_practices)

## 5.1.2 Diagrama entidad relación del prototipo

A continuación, en la Figura 5.1, se presenta el diagrama entidad-relación (ERD) del prototipo.

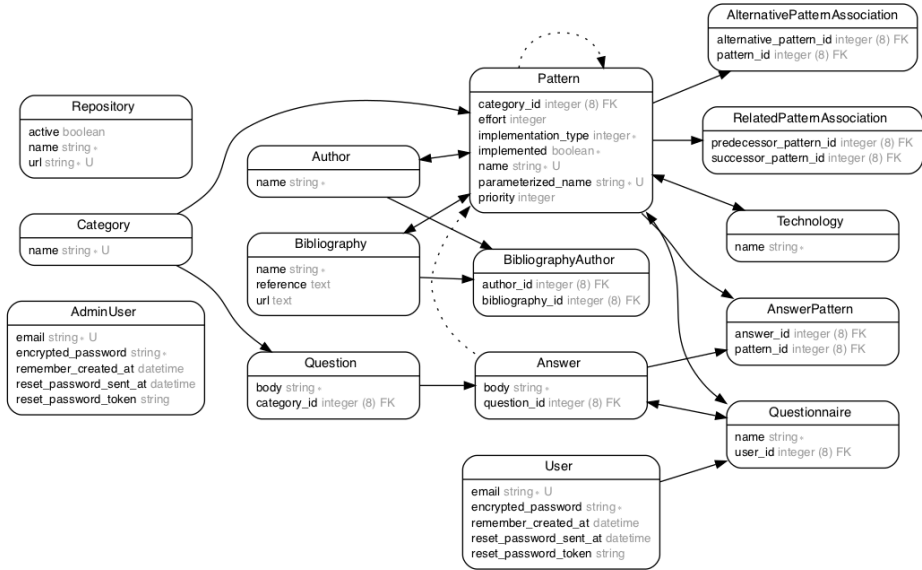


Figure 5.1: Diagrama entidad relación



## 5.2 Base de conocimiento

La base de conocimiento de patrones de microservicios contiene información sobre categorización, nombres, descripciones, relaciones, entre otros. Esta base se presenta al usuario a modo de listado, con funciones de filtrado y de búsqueda y es accesible por cualquier persona que ingrese al sitio web sin necesidad de registrarse.

La información de la base de conocimiento se encuentra en una base de datos y puede ser modificada por los usuarios con rol administrador en cualquier momento, pudiendo agregar/editar/eliminar categorías, patrones, autores, tecnologías, entre otros.

## 5.3 Recomendación de Patrones

El portal web también cuenta con un cuestionario de tipo *wizard* que todos los usuarios pueden completar y que tiene como resultado la recomendación/implementación de diferentes patrones. Las preguntas están fuertemente relacionadas a los requerimientos no funcionales del sistema que el usuario quiere implementar, por lo que se debió pensar cuidadosamente cuáles son las posibles combinaciones de preguntas y respuestas que brinden mayor información a la hora de recomendar los patrones.

Todas las posibles respuestas están asociadas a uno o mas patrones de microservicios, información que también forma parte del marco teórico de la solución. Por tanto, a partir de un conjunto de respuestas al cuestionario se obtiene un primer conjunto de patrones a recomendar al usuario.

Nuevamente, y al igual que con el contenido sobre patrones, el cuestionario es *customizable* por cualquier administrador, pudiendo estos agregar, editar o eliminar tanto preguntas y respuestas, como asociaciones entre respuestas y patrones.

Gran parte de la interacción que tiene el usuario con la plataforma se ve reflejada en el cuestionario. Consiste de una serie de preguntas múltiple opción que fueron formuladas luego de un análisis de la información disponible y nuestra base de conocimiento.

Las preguntas del cuestionario fueron hechas como guía principal para que el usuario se cuestionara algunos requerimientos no funcionales que su sistema tiene. Es la forma de interactuar y lograr recabar esa información de una manera didáctica. Cada pregunta tiene un conjunto de respuestas múltiple opción, donde cada respuesta está asociada a uno o más patrones. De esta forma se puede realizar una correspondencia de las respuestas del usuario a un conjunto de patrones a recomendar, que tiene como objetivo resolver problemas habituales en las arquitecturas de microservicios, teniendo en cuenta los requerimientos que tiene el usuario para su sistema.

### 5.3.1 Generación de preguntas

Como fue mencionado previamente las preguntas fueron formuladas en base al conocimiento que se obtuvo en la etapa de análisis. A su vez se hizo una profundización en el estudio de algunos patrones de manera de entender qué requisitos atacaban o qué problemas solucionaban para poder generar preguntas que fueran claras y tuvieran que ver con los requisitos posibles.

Una vez obtenida la información necesaria, se procedió a formular un conjunto de preguntas con dos respuestas asociadas a cada pregunta. Dichas respuestas están directamente asociadas a los patrones de microservicios, y todos los patrones tienen al menos una respuesta relacionada. Por lo tanto, a partir de las respuestas se puede obtener un subconjunto de patrones que se le recomendará al usuario.

### 5.3.2 Procesamiento de respuestas

Con el propósito de brindarle al usuario una solución lo más adecuada posible a partir de los requerimientos, se decidió realizar un postprocesamiento del conjunto de patrones sugerido, a modo de refinar/filtrar dicho conjunto y evitar realizar recomendaciones erróneas. El cuestionario devuelve una lista de respuestas donde cada una está relacionada directamente con un patrón, aunque esto no toma en cuenta el contexto o relación que tienen los patrones entre sí. Por ejemplo, en dicho procesamiento se busca la presencia de patrones alternativos en el conjunto para recomendar el patrón alternativo que mejor aplique.

A todos los patrones se les asignó valores numéricos para representar el esfuerzo de implementación y la prioridad. Estos valores son sumamente subjetivos y fueron asignados según nuestra perspectiva luego del análisis y estudio de los patrones que se hizo. En primer lugar, se dio una mayor prioridad a los patrones que se consideraron más relevantes a la hora de implementar y poder tener en la solución final. Para evitar fijar nuestro sesgo en este tipo de valores, los mismos se hicieron editables en la plataforma de administración.

Como no se toma en cuenta el contexto o la relación de patrones, la lista inicial de patrones recomendados podría no tener mucho sentido ya que se podrían recomendar patrones alternativos como una base de datos por servicio y una base de datos compartida. Si bien en un sistema real, estos patrones pueden coexistir de manera homogénea, para los efectos de este proyecto, teniendo como prioridad la simplicidad y facilidad para el usuario final, se decidió acotar las posibilidades de relacionamiento de patrones. Entonces, para la recomendación de patrones se implementó un filtrado que detecte patrones alternativos o que no se pueden relacionar directamente y quite a los que tienen menos prioridad de estos.

Una vez finalizado el procesamiento se le muestra al usuario una lista con los patrones recomendados, pudiendo éste filtrar dicha lista para quedarse únicamente con los patrones que le interesen. Las soluciones se pueden guardar aunque esto tiene como requisito que el usuario se encuentre registrado y *logueado*.

## 5.4 Implementación de patrones

Como se mencionó previamente, los patrones pueden tener distintos tipos de implementación. En los casos en que los patrones que se implementan utilizando código se decidió utilizar Python como lenguaje. "Python es un lenguaje de programación interpretado, orientado a objetos y de alto nivel con semántica dinámica" <sup>13</sup>.

Utilizando dicha tecnología, se procedió a construir una aplicación web template en el que se encuentran implementados todos los patrones (que están dentro del alcance de este proyecto).

Para la implementación de los patrones se debió tomar en cuenta que estos debían implementar de la forma más independiente posible, de modo que se pueda descargar la selección de patrones deseados y esto no afectase el funcionamiento de los mismos. A su vez, es importante que se pueda desplegar el sistema como un todo con facilidad.

### 5.4.1 Filtrado de patrones

El código para la implementación de cada patrón se encuentra englobado entre una serie de comentarios que los identifican, para poder quitar los patrones que deban ser removidos de forma dinámica. Esto se realiza mediante funciones que, utilizando expresiones regulares, remueven el código de los patrones no recomendados dentro de los archivos requeridos.

La aplicación web template cuenta con un archivo llamado *patterns\_locations.yml* para facilitar el filtrado de los patrones, que se realiza de forma dinámica. En el mismo, se encuentra información sobre qué archivos deben ser procesados y cómo remover el código no deseado utilizando expresiones regulares. A su vez, se detalla qué archivos deben ser eliminados.

El archivo cuenta con entradas para cada modelo y para cada patrón. Las entradas para los modelos cuentan con la siguiente estructura:

```
nombre_modelo:
  regex: <nombre del modelo en mayúscula>
  paths:
    - ruta al archivo donde se encuentra el código del modelo
      que debe ser sustituido
  files:
    - ruta a carpetas o archivos que tienen código del modelo
      que deben ser eliminados
```

Como se puede apreciar, se debe definir, para cada modelo, cuáles son los archivos que tienen código exclusivamente referente a dicho modelo que deben ser eliminados, y cuáles archivos tienen partes de código (como importaciones) que deben ser removidas utilizando expre-

---

<sup>13</sup><https://www.python.org/about/>

siones regulares. Para este segundo caso, el código debe estar encapsulado entre comentarios de la siguiente forma:

```
# PATTERN START: NOMBRE DEL MODELO
<codigo referente al modelo>
# PATTERN END: NOMBRE DEL MODELO
```

De esta forma, se pueden procesar los archivos que se encuentran dentro de *paths* utilizando expresiones regulares que quiten el código que se encuentra dentro de dichos comentarios, utilizando el nombre del modelo que se obtiene de *regex*.

Por otro lado, las entradas para los patrones cuentan con una estructura ligeramente diferente:

```
nombre_patron:
  min_ms_needed: <minimo numero de microservicios necesarios
    para poder ejecutar y testear el patron correctamente>
  # NOTA: las entradas "ms_n" pueden no ser necesarias en
    caso de que no se deban filtrar modelos en los
    microservicios
  ms_0:
    models:
      - <nombre del modelo que debe ser filtrado del
        microservicio 0>
    ...
  ms_N:
    models:
      - <nombre del modelo que debe ser filtrado del
        microservicio N>
  paths:
    - <ruta al archivo donde se encuentra el código del patrón
      que debe ser sustituido>
  nested_paths:
    - <ruta al archivo donde se encuentra el código del patrón
      que debe ser sustituido, que a su vez tiene la palabra
      clave NESTED>
  files:
    - <ruta a carpetas o archivos que tienen código del patrón
      que deben ser eliminados>
```

En este caso, el procesamiento es muy similar aunque cuenta con dos grandes diferencias. En primer lugar, se permite el procesamiento de patrones anidados, lo cual es útil cuando se tienen patrones sucesores y predecesores. En este caso, el patrón predecesor motiva la necesidad del patrón sucesor, así como el patrón sucesor resuelve algún problema introducido por el patrón predecesor. Esto quiere decir que, para utilizar el patrón sucesor y que este tenga sentido, es necesario también incluir el patrón predecesor. Suele ocurrir que el código

referente al patrón predecesor deba ser incluido al utilizar el patrón sucesor, por lo que en estos casos se utiliza el comentario de anidación y se incluye el código del patrón predecesor dentro de los comentarios del patrón sucesor.

Por otro lado, también se incluye el concepto de mínima cantidad de microservicios. Esta es la mínima cantidad de servicios necesarios para que los patrones se puedan testear correctamente. Por ejemplo, para el caso de API Composition, se utilizan tres servicios. Dos servicios retornan información que le compete a cada uno de ellos, mientras que el tercer servicio se encarga de consumir la información de estos dos servicios y devolverla en una única consulta.

### 5.4.2 Implementación de patrones

A continuación se presenta un listado de todos los patrones que fueron implementados:

- API Gateway
- Servicio como Contenedor
- Plataforma para el despliegue de servicios
- Log aggregation (Acumulación de logs)
- Health Check API (API de estado de salud)
- Descubrimiento de servicios del lado del servidor
- Registro con herramientas de terceros
- Access Token
- API Composition
- Una base de datos compartida
- Una base de datos por servicio
- Configuración Externalizada
- Saga
- Transactional Outbox
- Transaction Log Tailing
- Métricas de la aplicación
- Audit logging

En primer lugar, se procedió a implementar los patrones de despliegue *Servicio como Contenedor* y *Plataforma para el despliegue de servicios* debido a que estos patrones resuelven problemas relacionados a la ejecución de los microservicios. Una de las ventajas de tener una plataforma con arquitectura de microservicios es que se pueden usar diferentes lenguajes, frameworks y herramientas para resolver diferentes problemas. *Servicio como Contenedor* hace énfasis en la facilidad de configuración del ambiente así como también en la garantía de que los ambientes de desarrollo y productivos son iguales. Además se resuelve el problema de aislamiento y asignación de recursos a los microservicios que corren aunque sea en la misma máquina física (o virtual). Por otro lado, se facilita la construcción de automatizaciones que contribuyen a aumentar la velocidad de despliegue de la arquitectura y la construcción y ejecución de nuevos microservicios.

Todas estas características hacen que sea posible la automatización de diversas configuraciones para que el usuario descargue la solución implementada y, de forma simple y rápida, sea capaz de levantar la arquitectura propuesta en su sistema.

Para lograrlo, más precisamente, se utilizaron las tecnologías de Docker como servicio de containerización y Kubernetes como plataforma de despliegue.

Luego de nuestra investigación llegamos a la conclusión de que cualquier sistema que posea una arquitectura basada en microservicios necesita implementar patrones de monitoreo, observabilidad y resiliencia, para poder lograr un seguimiento de todas las acciones distribuidas que están ocurriendo, además de registrar posibles fallas y facilitar el debugging del sistema. Para esto se decidió implementar los patrones Log aggregation y Health Check API.

El primero se implementó con la plataforma ELK (Elasticsearch, Logstash y Kibana) de Elasticsearch recolectando la salida estándar de los microservicios y consolidándolo en una plataforma que permite buscar, diferenciar y correlacionar logs.

Por otro lado, para Health Check API se implementó un endpoint HTTP que retorna 200 (código HTTP) cuando el servicio está listo para recibir peticiones. Esto es esencial en una arquitectura de microservicios donde puede haber varias decenas de microservicios y cientos de contenedores ejecutando, mejorando la resiliencia de la plataforma. Plataformas de despliegue como Kubernetes (utilizado en este Proyecto), y otras que se utilizan en la industria, ofrecen a partir de la utilización de un endpoint de health check, la posibilidad de detectar cuándo un contenedor está fallando o respondiendo lentamente y reemplazarlo por otro igual de forma automática.

Los patrones de descubrimiento de servicios que se implementaron son Descubrimiento de servicios del lado del servidor y Registro con herramientas de terceros. Estos dos patrones están relacionados. Según Richardson [6] si se utiliza una única plataforma para el despliegue, se utiliza un descubrimiento de servicios basado en plataforma, que en nuestro caso será Kubernetes. Esta plataforma maneja un servicio de DNS interno que puede ser utilizado por los diferentes microservicios para invocar a otros.

Para los usuarios que deseen aumentar la seguridad de su solución, protegiendo las diferentes

APIs, se implementó el patrón Access Token. A su vez, se implementó el patrón API Composition para la consulta de datos. Este patrón es recomendado que se use siempre que sea posible [10].

Los patrones Una base de datos compartida y Una base de datos por servicio fueron ambos implementados, para proveer al usuario una variedad de opciones a la hora de almacenar datos. En general, para plataformas con varios microservicios (decenas) se recomienda que no utilicen una sola base de datos compartida entre todos. La principal razón de esto es que la indisponibilidad del motor de bases de datos generaría problemas en toda la plataforma.

Para proveer a los servicios de configuración, en vez de utilizar variables fijas y que estén escritas directamente en el código, se implementó el patrón Configuración Externalizada. De esta forma la configuración de los microservicios se realiza de manera dinámica y distribuida, haciendo que la solución esté menos acoplada.

## 5.5 Extender la solución con nuevos patrones

Para extender la solución con un nuevo patrón, se debe agregar el patrón al portal web a través de la interfaz administrativa, junto con la categoría a la que pertenece, el nombre, una descripción y otra información relevante. Una vez hecho esto, se debe agregar la implementación del patrón en cuestión. Para esto, primero se debe identificar cuál es el tipo de implementación de dicho patrón.

Si el tipo de implementación consiste en **documentación**, basta con incluir una buena documentación del mismo en el panel administrador. En el caso de que el patrón deba ser **implementado utilizando código**, se debe agregar la implementación del patrón al repositorio de la aplicación web template, incluyendo el código del mismo entre comentarios como se especifica en el archivo README de este proyecto. Por último, en caso de que el patrón se implemente agregando/editando archivos de configuración, se debe incluir la configuración del mismo en el proyecto Helmfile y ésta debe estar entre comentarios de la misma forma que se describe en el caso anterior.

El último paso en este proceso consiste en modificar el servicio de procesamiento de patrones, el cual es el encargado de filtrar los patrones en base a las respuestas introducidas por el usuario en el cuestionario, para que tome en cuenta el nuevo patrón.

A continuación se presentan los repositorios donde se encuentra almacenado el código de cada componente de la solución:

- Plataforma web:

[https://gitlab.fing.edu.uy/open-lins/ms-patterns-imp/microservices\\_patterns](https://gitlab.fing.edu.uy/open-lins/ms-patterns-imp/microservices_patterns)

- Aplicación web template con implementación de patrones:

<https://gitlab.fing.edu.uy/open-lins/ms-patterns-imp/web-server-templato>

- Charts con diferentes implementaciones:

<https://gitlab.fing.edu.uy/open-lins/ms-patterns-imp/helm-charts>



# 6 | Caso de estudio y evaluación

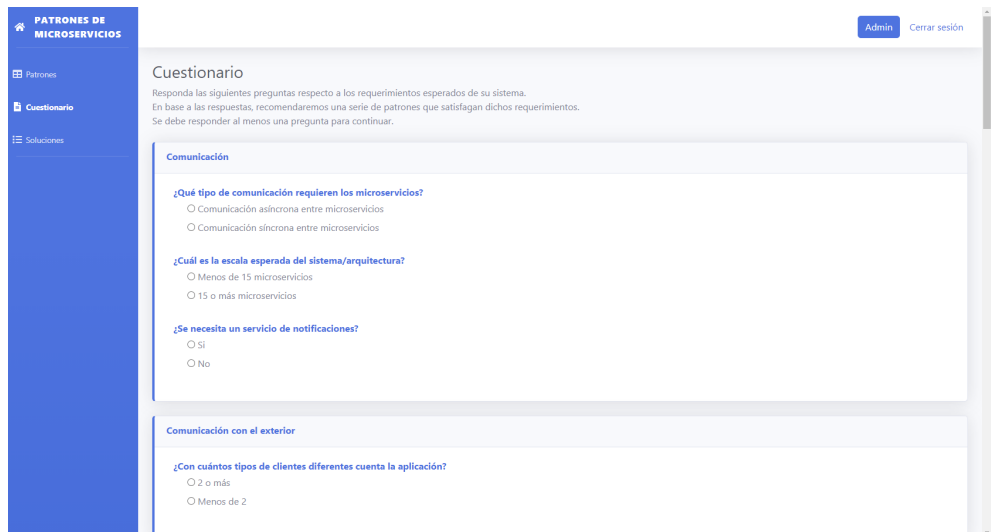
En este capítulo se muestra el uso de la plataforma para un caso hipotético y la evaluación de la misma a partir del juicio de diferentes expertos y plataformas.

El capítulo se organiza de la siguiente manera: en la Sección 6.1 se detalla paso a paso el uso de la plataforma mediante un caso de estudio. Luego en la Sección 6.2 se incorporan todas las respuestas al formulario de experiencia de uso para su análisis. Finalmente, se presentan los resultados de evaluar en la plataforma *Microservice Architecture Assessment Platform* [19] la solución ejecutable (en particular alguno de los microservicios retornados) generada en el caso de estudio.

## 6.1 Caso de estudio

El primer paso consiste en acceder al portal de usuario y completar el cuestionario. El usuario debe ingresar al portal y acceder a la sección de Cuestionario en el menú izquierdo como se muestra en la Figura 6.1. Una vez en la página del cuestionario, se procede a responder las preguntas que se consideran relevantes, y se debe responder al menos una. Para nuestro caso de estudio se respondieron las preguntas considerando un sistema hipotético que se desea implementar.

Una vez que se termina de responder el cuestionario se muestra un listado de los patrones recomendados y se da la opción de guardar la recomendación. El usuario debe estar registrado y autenticado.



**PATRONES DE MICROSERVICIOS**

Admin Cerrar sesión

### Cuestionario

Responda las siguientes preguntas respecto a los requerimientos esperados de su sistema. En base a las respuestas, recomendaremos una serie de patrones que satisfagan dichos requerimientos. Se debe responder al menos una pregunta para continuar.

#### Comunicación

¿Qué tipo de comunicación requieren los microservicios?

- Comunicación asíncrona entre microservicios
- Comunicación síncrona entre microservicios

¿Cuál es la escala esperada del sistema/arquitectura?

- Menos de 15 microservicios
- 15 o más microservicios

¿Se necesita un servicio de notificaciones?

- Sí
- No

#### Comunicación con el exterior

¿Con cuántos tipos de clientes diferentes cuenta la aplicación?

- 2 o más
- Menos de 2

Figure 6.1: Cuestionario

Para este caso, la lista de patrones recomendados es:

- Strangler application
- Transaction Script
- Service Mesh
- Descubrimiento de servicios del lado del servidor
- Polling publisher
- Mensajería
- Descomponer en base a capacidades de negocio
- Registro con herramientas de terceros
- Orquestación
- Saga
- API Gateway
- API Composition
- Access Token

- Circuit Breaker
- Exception Tracking (Seguimiento de excepciones)
- Seguimiento distribuido (Distributed tracing)
- Metricas de la aplicación
- Audit Logging
- Health Check API (API de estado de salud)
- Log Aggregation (Acumulación de logs)
- Microservices Chasis
- Configuración Externalizada
- Microservices Chasis
- Service Component Test
- Consumer-side contract test
- Consume driven contract test
- Registro Personal
- Domain Event
- Event sourcing
- Transaction Log Tailing
- Aggregates
- Transactional Outbox
- Una base de datos por servicio

Como podemos ver en la Figura 6.2 se obtiene una lista de patrones de microservicios con sus respectivas descripciones de los cuales algunos están implementados. Cada recomendación tiene un conjunto de patrones asociado, pero el usuario puede remover los patrones que no considera necesarios con el fin de descargar la solución que mejor se adapte a sus necesidades.

**PATRONES DE MICROSERVICIOS** Admin Cerrar sesión

### Patrones recomendados

Lista de patrones recomendados para el sistema

- Strangler application**

Marín Fowler describe este patrón como el proceso de ir creando un nuevo sistema gradualmente alrededor del viejo y que hacer que este crezca lentamente hasta que el nuevo sistema "estrangula" el viejo.

La visión de Richardson es de un enfoque incremental donde en el proceso se genera una aplicación denominada strangler (estranguladora) que consiste en microservicios que implementan nuevas funcionalidades que extienden a la aplicación original monolítica y la extracción de servicios desde el monolito se va implementando a medida que los requerimientos van mutando. Cuando un servicio que forma parte del monolito requiere un cambio en su funcionalidad que provoca que se realice mantenimiento, es seleccionado para ser trasladado a microservicio.

A medida que pasa el tiempo el tamaño del monolito va decreciendo teniendo como consecuencia una aplicación totalmente en microservicios.

Este patrón consiste de tres pasos: Transformar, coexistir y eliminar. Coexisten servicios implementados bajo los dos enfoques durante el transcurso de la transición.

**Beneficio:**

  - Demuestra valor real rápido y seguido: al implementar las nuevas funcionalidades como microservicios, se puede aprovechar para desarrollar con tecnologías modernas.
  - Minimizar cambios al monolito: al implementar los servicios separados se minimiza el trabajo de mantenimiento del monolito el cual necesita periódicamente refactorizar.
  - No se necesita una infraestructura de despliegue desde el principio: cómo se va migrando de a poco se puede comenzar con una configuración básica de la infraestructura de despliegue que permita irse adelantando en el mundo de los microservicios sin arriesgar en una inversión fuerte desde el principio.
- Transaction Script**
- Service mesh**
- Descubrimiento de servicios del lado del servidor** Implementado
- Polling Publisher**

Si la aplicación utiliza una base de datos relacional, una forma muy simple de publicar los mensajes insertados en la tabla OUTBOX es sondear periódicamente la tabla en busca de mensajes no publicados. Cuando se detectan mensajes no publicados estos se envían al broker de mensajería y se eliminan de la tabla OUTBOX.

Este es un enfoque simple que funciona razonablemente bien a baja escala. La gran desventaja es que consultar frecuentemente la base de datos puede ser costoso. Además, el hecho de que se pueda usar este enfoque con una base de datos NoSQL depende de sus capacidades de consulta. Esto sucede porque en lugar de consultar una tabla OUTBOX, la aplicación debe consultar las entidades en sí, y eso puede o no ser posible hacerlo de manera eficiente.

Figure 6.2: Patrones recomendados

Una vez que se ingresa al sistema y se guarda la recomendación, se debe acceder a la sección de Soluciones al igual que se muestra en la Figura 6.3.

**PATRONES DE MICROSERVICIOS** Admin Cerrar sesión

### Soluciones

Soluciones guardadas.

**Índice de soluciones**

Show 10 entries Search:

ID	Nombre	Fecha creado	Acciones
41	Caso de estudio	19/09/2021	Editar Eliminar Descargar

Showing 1 to 1 of 1 entries Previous 1 Next

Copyright © Proyecto de Grado 2020

Figure 6.3: Soluciones

En este caso se decide dejar todos los patrones recomendados y se procede a descargar la

solución, que es un archivo de tipo *zip*. Al descargar y descomprimir la solución, se obtiene un archivo con instrucciones de despliegue llamado *deployment\_instructions.html* y dos carpetas llamadas *helmfile* y *web-server-template*.

El archivo *deployment\_instructions.html* contiene una guía para correr la solución descargada: procedimiento, herramientas necesarias y links para descargas además de una forma de probar el funcionamiento de la solución.

La carpeta *helmfile* contiene todos los archivos en los que se definen los charts de helm<sup>1</sup> que serán recorridos y ejecutados por *helmfile*<sup>2</sup> en esta solución ejecutable. Además incluye una carpeta de valores con los valores que sobrescriben los charts template en este caso particular. Todos estos archivos se generan dinámicamente por el subsistema generador de la solución ejecutable.

La carpeta *web-server-template* contiene una carpeta por microservicio ejemplo que se utiliza en esta solución. Estos microservicios contienen el código con los patrones implementados y, a su vez, permiten probar la solución y el funcionamiento de los diferentes patrones. Además, se provee un script *build\_all.sh* que recorre las carpetas de los microservicios, genera la imagen de Docker para cada uno y la sube a un repositorio de imágenes de Docker que recibe como parámetro.

La Figura 6.4 muestra una captura de Lens<sup>3</sup> con la solución descargada en este caso ejecutando en un cluster de Kubernetes. En este caso se usó Minikube<sup>4</sup> para levantar un cluster de Kubernetes local.

---

<sup>1</sup><https://helm.sh/>

<sup>2</sup><https://github.com/roboll/helmfile>

<sup>3</sup><https://k8slens.dev/>

<sup>4</sup><https://minikube.sigs.k8s.io/docs/>

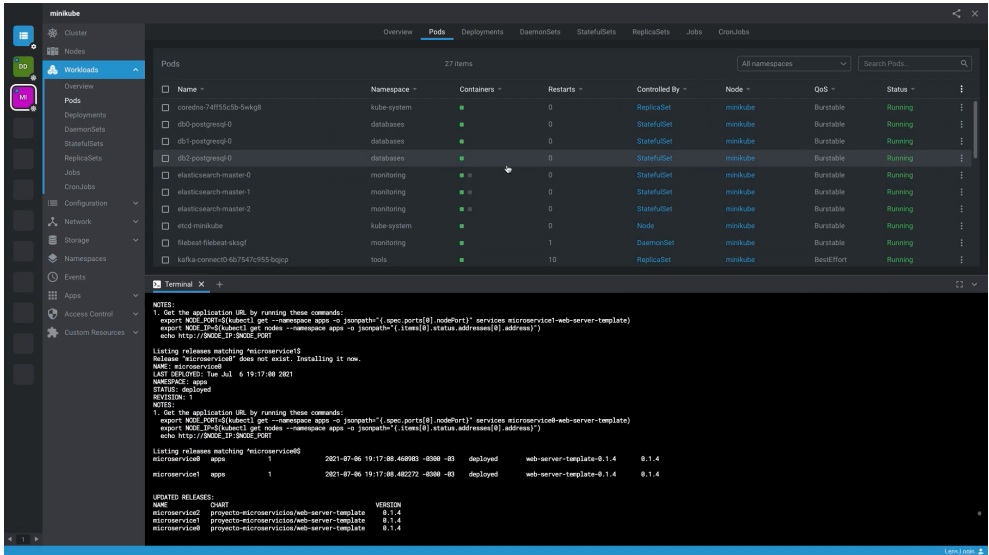


Figure 6.4: Solución ejecutando

La figura 6.5 muestra una captura de un navegador web en la que se observa la interfaz web que expone uno de los microservicios en ejecución.

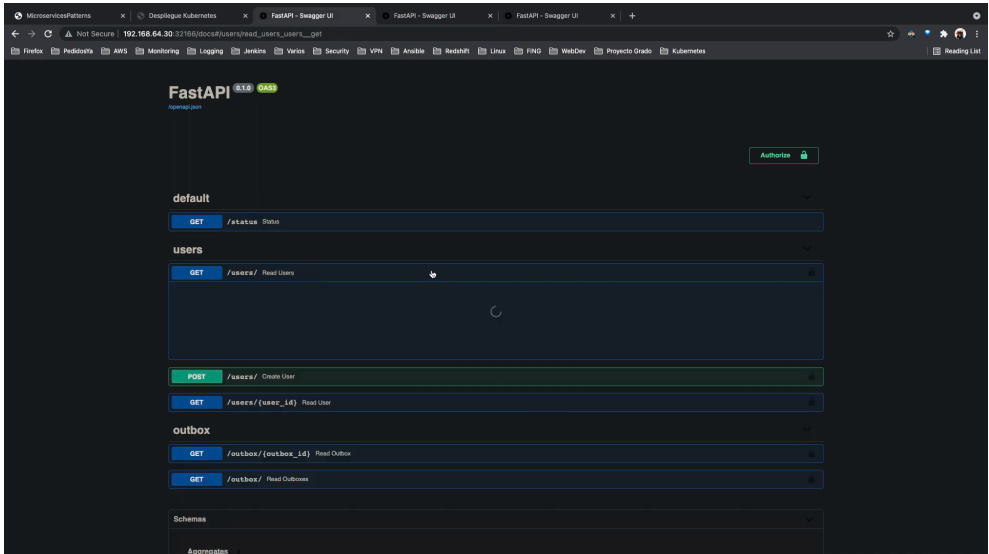


Figure 6.5: Microservicio web

## 6.2 Juicio de expertos

Con el fin de validar la plataforma se creó un formulario de satisfacción y se solicitó a personas con diferentes niveles de experiencia en el área de microservicios que prueben la herramienta y completen dicho formulario. Un total de ocho personas completaron el formulario y van desde desarrolladores y analistas de software hasta ingenieros y directores de ingeniería. A continuación se presenta la información y conclusiones obtenidas de las respuestas del formulario.

Para comenzar, del total de expertos sólo uno de ellos asegura conocer otra plataforma que tenga el mismo fin que la propuesta en este Proyecto. Esta plataforma es simplemente un catálogo de patrones de microservicios e información relacionada, por lo que la nuestra constituye una solución mas completa ya que también contiene un cuestionario y retorna una solución ejecutable. Esta información va en línea con nuestro análisis en cuanto a que actualmente no existe otra plataforma que resuelva los mismos casos de uso.

Por otro lado, en cuanto a la experiencia de usuario a la hora de usar la plataforma se obtuvieron respuestas positivas. Como se muestra en la Figura 6.6, de los ocho expertos que interactuaron con la plataforma cuatro destacan que tuvieron una muy buena experiencia mientras que los otros cuatro afirman que tuvieron una experiencia buena. Además, vale la pena destacar que ninguno de ellos tuvo inconvenientes a la hora de usar la plataforma.

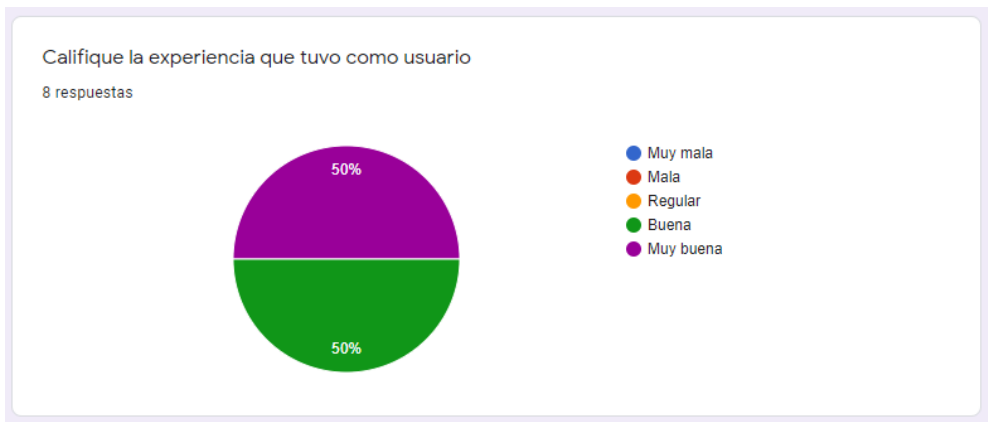


Figure 6.6: Experiencia como usuario

En cuanto a la utilidad de la plataforma, se observa en la Figura 6.7 que la mitad de los expertos consideran la que la misma es útil, mientras que 3 de ellos la consideran muy útil. Más aún, todos ellos aseguran que usarían una plataforma del estilo en el ámbito profesional.

Por último, varios de los expertos propusieron sugerencias para mejorar la plataforma, algunas de las cuales fueron implementadas. La primera de éstas fue agregar enlaces de documentación con el objetivo de respaldar la información que provee la plataforma. Esto se

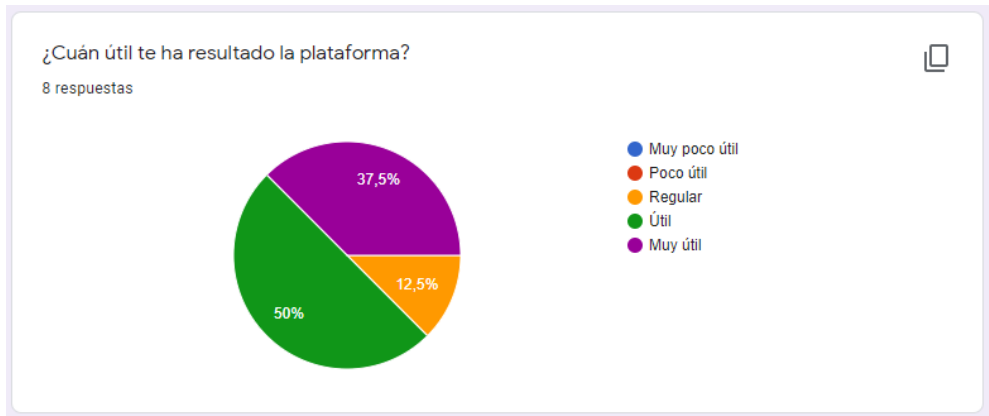


Figure 6.7: Utilidad de la plataforma

incluyó agregando un campo de referencias para cada patrón. Otra sugerencia fue agregar una introducción sobre las herramientas que se utilizan para correr la solución ejecutable. Dado que la solución descargable ya cuenta con un archivo de documentación sobre cómo ejecutar la solución donde se proveen los enlaces a la documentación de las herramientas y tecnologías que se utilizan, consideramos que esta información ya es accesible y no es necesario incluirla en otro lado. Finalmente, también se hicieron recomendaciones sobre como mostrar la información de cada patrón, separando la descripción de las ventajas y desventajas y los problemas que estos resuelven, pero debido a otras prioridades en el desarrollo del Proyecto decidimos dejar esto como trabajo a futuro.

En conclusión, a pesar de que la cantidad de usuarios que probaron la plataforma no es alta, se pudo observar que casi por unanimidad todos ellos consideran que el producto es bueno y que utilizarían una herramienta así. Por lo tanto, se considera que se cumplió con el objetivo de construir una herramienta que asista al usuario en la implementación de arquitecturas de microservicios basadas en patrones.

### 6.3 Evaluación de la solución generada

Además del juicio de expertos detallado en la sección anterior, se evaluó la solución generada en el caso de estudio utilizando la plataforma *Microservice Architecture Assessment Platform*[19] mencionada en el Capítulo 2. Dicha plataforma fue creada por Chris Richardson con el objetivo de evaluar sistemas basados en la arquitectura de microservicios, teniendo en cuenta las buenas prácticas y patrones definidos por él. Esta evaluación consiste en responder un cuestionario detallando, en sus distintas partes, si se implementan determinados patrones de microservicios, qué tan ágil se es para realizar liberaciones, de qué manera se manejan las fallas en los servicios, etc.



Los resultados de la evaluación de la solución generada en el caso de estudio se pueden ver en la Figura 6.8. Se puede observar que dentro de las categorías de *Arquitectura general*, *Configuración*, *Observabilidad* y *Comunicación entre servicios* se lograron puntajes muy altos. Esto es un resultado muy positivo y una confirmación de la hipótesis en la cual consideramos que los patrones que están relacionados a esas áreas siempre deben estar presentes en cualquier solución y por lo que se recomiendan en todos los casos. Además, estos patrones fueron prioritarios a la hora de decidir qué patrones se implementarían.

Existen también áreas con un puntaje bajo. Hay dos razones para esto: en el caso de *Deployment* las preguntas eran orientadas a un ambiente de producción real, por ejemplo, manejando preguntas cómo "¿Qué tan común es que se introduzcan fallas en el ambiente de producción?", "¿La infraestructura de liberaciones valida automáticamente el número deseado de instancias de servidores que están corriendo en producción?", entre otras. Estas preguntas no se aplican muy bien a nuestra solución ya que la misma no es un sistema que se encuentre en un ambiente de producción real. El segundo caso de bajo puntaje, como es el de *libraries* se debe a que en estas preguntas se hace énfasis en el aspecto de testeo y uso de librerías orientadas a esto. Aunque los patrones de testeo también son recomendados por defecto en cualquier solución, no fueron implementados, por lo cual la solución del caso de estudio no cuenta con este tipo de funcionalidades ni librerías.

Teniendo en cuenta los puntajes obtenidos, creemos que las decisiones tomadas, tanto a la hora de la recomendación de patrones como de la implementación, fueron las correctas contemplando el objetivo de proveer un sistema basado en microservicios que implementara patrones y siguiera buenas prácticas.

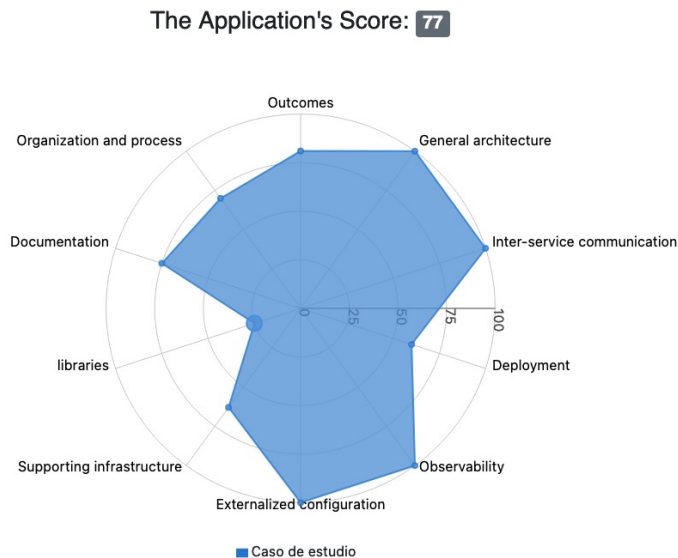


Figure 6.8: Resultado de evaluación del caso de estudio

# 7 | Conclusiones y trabajo futuro

En este capítulo se presentan las conclusiones del trabajo realizado y sugerencias de posibles agregados y mejoras que podrían considerarse para trabajo futuro.

## 7.1 Resumen

El objetivo general del Proyecto fue proponer una plataforma que asista al usuario en el diseño, implementación y puesta en marcha de un sistema basado en una arquitectura de microservicios mediante el uso de patrones.

Se comenzó realizando un relevamiento del conocimiento existente sobre los patrones de microservicios, tomando como base el conjunto de patrones presentado en el Proyecto de Grado de 2020. Luego, se procedió a realizar una investigación exhaustiva con el fin de actualizar y extender dicha base de conocimiento. Esto fue necesario para poder definir el alcance del trabajo a realizar.

Una vez que los objetivos se encontraban definidos, se procedió a investigar la plataforma implementada para el Proyecto de Grado de 2020, a modo de identificar las funcionalidades y determinar módulos a reutilizar. Sin embargo, se concluyó que para el presente trabajo era conveniente construir la plataforma desde cero utilizando tecnologías modernas, utilizadas en la industria y con las que el equipo estuviera familiarizado.

Luego de la etapa de implementación se logró llegar a una plataforma que cumpliera con la propuesta de solución presentada. Esta plataforma está compuesta por diferentes subsistemas que asisten al usuario en el proceso desde la concepción de un sistema basado en arquitecturas de microservicios hasta la implementación utilizando patrones y su puesta en marcha.

La plataforma fue implementada utilizando tecnologías modernas. Los tres componentes son:

1. Un portal web implementado en Ruby on Rails con el que interactúa el usuario final.
2. Un motor de recomendación de patrones que, a partir de un conjunto de respuestas del usuario final, sugiere un conjunto de patrones de microservicios que abordan los requerimientos relevados.

3. Un generador de una solución ejecutable que contiene la implementación de un conjunto de los patrones recomendados por el componente anterior.

## 7.2 Conclusiones

Se presentan en esta sección conclusiones y reflexiones del trabajo realizado.

La arquitectura de microservicios no es una solución milagrosa. No sólo pueden resultar complejas, sino que a la hora de utilizar esta arquitectura existen numerosos problemas que se deben abordar. Sumado a esto, estas arquitecturas todavía carecen de estandarización y buenas prácticas que lleven a solucionar dichos problemas.

Hoy en día existen varios autores que han intentado definir patrones de microservicios que asistan a los ingenieros y arquitectos de software a la hora de resolver los problemas que se generan al utilizar arquitecturas de microservicios. Sin embargo, al existir muchas fuentes, resulta difícil relevar esta información ya que es usual que existan diferentes patrones muy similares entre sí o incluso propuestas diferentes para el mismo patrón. Además, también suele ocurrir que las definiciones de algunos patrones sean vagas, demostrando, nuevamente, la falta de estandarización que hay en el tema.

A pesar de todo esto, el Proyecto de Grado previo realizó un buen trabajo relevando toda esta información, y en este proyecto también se logró extender y actualizar dicha base de conocimiento. Una vez recabada toda la información se procedió a diseñar la plataforma y a definir el alcance que tendría dicho trabajo.

El objetivo planteado fue implementar una plataforma que asistiera a los usuarios en la creación de arquitecturas de microservicios basados en patrones. A su vez, se decidió presentar una propuesta de solución que se centrara principalmente en completar el flujo desde que un usuario establece sus requerimientos hasta el momento en el que el usuario despliega su solución ejecutable a un ambiente de producción.

Por otro lado, para la implementación se decidió utilizar muchas tecnologías, priorizando aquellas con las cuales se tenía experiencia previa y tienen un uso extendido en la industria.

En términos generales, consideramos que se cumplieron con los objetivos planteados y se logró la correcta implementación de una herramienta que puede ser utilizada por los usuarios a la hora de trabajar con arquitecturas de microservicios usando patrones. Esto se ve respaldado por las respuestas obtenidas del formulario de satisfacción que completaron los usuarios que probaron la plataforma, como se detalla en la sección de Juicio de expertos.

Entre los requerimientos de la plataforma la mantenibilidad fue detectado como esencial. Creemos que esto se logró por dos razones principales: en primer lugar, el sitio web cuenta con un panel administrador donde los usuarios habilitados pueden modificar toda la información que se despliega en la web de forma simple y rápida. Por otro lado, a pesar de

que la arquitectura de la plataforma puede resultar compleja, la extensión de la misma se puede realizar de forma simple debido a el hecho de que la implementación de cada patrón se encuentra englobada dentro de comentarios que la identifican, y se incluyó documentación acerca de cómo extender el sistema añadiendo nuevos patrones o modificando los existentes.

## 7.3 Trabajo futuro

En esta sección se describen posibles oportunidades de mejora para la solución presentada.

En primer lugar, debido al alcance definido en base al tiempo estimado del proyecto, se dejó de lado la implementación de algunos de los patrones de microservicios conocidos. Por lo tanto, una primera forma de extender esta solución es implementar dichos patrones. A continuación se presenta una lista de patrones no implementados.

- Circuit Breaker
- Servicios en máquinas virtuales
- Despliegue sin servidores
- Registro Personal
- Descomponer en base a subdominios
- Mensajería
- Invocación a procedimientos remotos
- Orquestación
- Coreografía
- Descubrimiento de servicios del lado del cliente
- Exception Tracking (Seguimiento de excepciones)
- Seguimiento distribuido (Distributed tracing)
- Anti corruption layer
- Polling Publisher
- Domain Event
- Event Sourcing
- Transaction Script

- Service mesh
- Descomponer en base a las capacidades de negocio
- Microservices Chasis
- Client-side UI Composition
- Server-side page fragment composition
- CQRS
- Strangler application
- Consumer driven contract test
- Consumer-side contract test
- Service Component Test
- Backend for frontends
- Aggregates

Por otro lado, a modo de extender la solución, se podrían incluir nuevos patrones de microservicios a la base de conocimiento para que luego sean utilizados por el motor de recomendaciones. A pesar de haber realizado una investigación exhaustiva, no podemos afirmar que no existan ni vayan a existir nuevos patrones que no hemos tenido en cuenta.

En tercer lugar, para no salir del foco principal de este trabajo, se decidió implementar cada patrón en una única tecnología. Sin embargo, esto no quiere decir que no puedan ser implementados en otras tecnologías. Por lo tanto, otra extensión posible a la solución es la implementación de los patrones utilizando otros lenguajes de programación, herramientas, procedimientos, etc. En este caso debemos tener en cuenta que implementar los patrones en otra tecnología puede implicar el uso de nuevos repositorios de código, por lo que también habría que implementar la lógica para que un usuario pueda seleccionar en que tecnología descargar los patrones, o simplemente retornar los patrones implementados en todas las tecnologías en las que se hayan implementado.

Por otro lado, en las arquitecturas de microservicios uno de los problemas que se observa rápidamente con el aumento en la cantidad y diversidad de microservicios es en las automatizaciones para poner el código a ejecutar en producción. Desde testeo unitario, análisis de seguridad, generación de imágenes de Docker, despliegue en una plataforma de ejecución de contenedores, entre otras funcionalidades pueden ser atacadas con una plataforma de integración y delivery continuo (CI+CD). Sería interesante anexar a lo ya implementado una solución más robusta y escalable a la hora de poner a prueba, compilar y desplegar el código de los diferentes microservicios.

# A | Apéndice

Este este apéndice se presentan las definiciones de patrones de microservicios que fueron relevados durante el trabajo.

## A.0.1 Comunicación

Mensajería, Invocación a procedimientos remotos, Orquestación y Coreografía son patrones de comunicación de microservicios.

### Mensajería

El patrón Mensajería propone comunicar a los servicios intercambiando mensajes de forma asincrónica. Richardson define tres tipos de mensajes [6]:

- Documento: un mensaje genérico que contiene solo datos. El receptor decide cómo interpretarlo y la respuesta a un comando es un ejemplo de mensaje de documento.
- Comando: un mensaje donde se especifica la operación a invocar y sus parámetros.
- Evento: un mensaje que indica que ha ocurrido algo notable en el remitente. Un evento es a menudo un evento de dominio, que representa un cambio de estado de un objeto del dominio.

A su vez, generalmente se utilizan los siguientes canales de mensajería:

- Punto-a-punto: Se entrega el mensaje solo a uno de los receptores que lee desde ese canal.
- Publish/subscribe: Se entrega el mensaje a todos los consumidores que están suscritos al canal.

### Invocación a procedimientos remotos

La invocación a procedimientos remotos consiste en hacer que un cliente envíe una solicitud a un servicio, el servicio procese dicha solicitud y devuelva una respuesta. A pesar de que

---

algunos clientes pueden bloquear la espera de una respuesta mientras que otros pueden tener una arquitectura reactiva y sin bloqueo, se supone que la respuesta llegará al cliente de manera oportuna. Un ejemplo de invocación a procedimientos remotos es REST [6].

### **Orquestación**

El patrón orquestación es uno de los utilizados para implementar el patrón saga. Al utilizar este patrón, se define un coordinador central de transacciones, quien orquesta cada paso de la transacción y coordina las operaciones de compensación en caso de fallos [7].

El orquestador de la saga se comunica con los participantes mediante la interacción comando/respuesta asíncrona. Cada vez que un paso se completa, el microservicio le avisa al coordinador que ha terminado. Esta orquestación permite que los microservicios se mantengan desacoplados.

### **Coreografía**

Coreografía es otro de los patrones utilizados para implementar el patrón saga. En este caso, para que los diferentes microservicios puedan cumplir su función deben coordinarse entre sí. En este patrón no existe un coordinador central encargado de notificar a los servicios qué hacer, sino que la lógica de negocio se reparte entre estos.

Este patrón cuenta con beneficios como simplicidad ya que los servicios publican eventos cuando crean, actualizan o eliminan objetos comerciales, y bajo acoplamiento debido a que los participantes se suscriben a los eventos y no se conocen directamente [7].

Por otro lado, el patrón también presenta algunos inconvenientes como mayor complejidad ya que no hay un solo lugar en el código que defina la saga, dependencias cíclicas entre servicios y riesgo de alto acoplamiento ya que cada participante de la saga debe suscribirse a todos los eventos que lo afecten [7].

## **A.0.2 Comunicación con el exterior**

Dentro de esta categoría se destacan los patrones API Gateway y Backend for frontends.

### **API Gateway**

El patrón API Gateway propone implementar un servicio que proporcione un punto de entrada único ya sea para los microservicios de la aplicación o el mundo exterior. Puede tener otras

funcionalidades como ser la autenticación, monitoreo, balanceo de cargas, cache, entre otros [11].

El API Gateway cumple la función de un proxy inverso, enrutando las solicitudes de los clientes a los servicios. Se sitúa entre los clientes y los microservicios que componen a la aplicación. Por lo tanto, una de las principales ventajas es que encapsula la estructura interna de la aplicación [11].

Como desventajas, al ser un componente extra que el sistema debe manejar, se puede convertir en un cuello de botella ya que hay muchas complejidades que se deben gestionar al usarlo [11].

### **Backend for frontends**

El patrón Backend for frontends fue descrito por primera vez por Sam Newman. Se propone crear servicios de backend independientes para que los consuman aplicaciones o interfaces de frontend específicas [54].

Este patrón resulta útil cuando desea evitar realizar cambios personalizados para múltiples interfaces haciendo uso de un único backend. De este forma, se logra ajustar el comportamiento y el rendimiento de cada backend para que se adapte mejor a las necesidades del entorno de cada cliente, sin preocuparse por afectar a otros clientes que no dependan de éste [42].

Debido a que cada backend es específico de una interfaz, se puede optimizar para esa interfaz. Como resultado, será más pequeño, menos complejo y probablemente más rápido que un backend genérico que intenta satisfacer los requisitos de todas las interfaces. A su vez, esto permite utilizar diferentes tecnologías para cada backend, pudiendo elegir la que mejor se adapte en cada caso [42].

### **A.0.3 Consulta de Datos**

Existen dos patrones que resuelven este problema, CQRS y API Composition.

#### **CQRS**

La noción de este patrón, según Martin Fowler, es que se tiene un modelo de datos para actualizar la información y otro para consultarla. Aclara que en algunas situaciones esto es de mucha importancia pero implementar un sistema de estos añade riesgos complejos [30].

Se propone que los servicios mantengan una o más réplicas de datos que son propiedad de otros servicios pero que son de utilidad para consultas frecuentes. De este modo a la hora



de responder a una solicitud tiene acceso a datos que frecuentemente utiliza pero que no son propiedad del mismo, lo cual por un lado tiene la gran ventaja que no se requiere de la invocación a otros servicios para responder a algunas de las consultas [10].

Sin embargo, también se debe considerar realizar una cuidadosa implementación y diseño de las vistas ya que se requiere de la coordinación entre las mantenidas en servicios externos a donde residen los datos con la base de datos que los alberga. Por este motivo se debe encontrar una forma eficiente de diseñar las vistas de forma tal que los datos que allí residen sean la mínima cantidad posible con el máximo margen de utilidad para lograr que el balance costo beneficio que aporta la aplicación de este patrón sea sostenible para contemplarlo en la realidad.

Se pueden destacar tres problemas comunes que impulsan la utilización de este enfoque [30]:

- El uso del patrón API Composition para recuperar datos dispersos en múltiples servicios da como resultado uniones en memoria costosas e ineficientes.
- El servicio que posee los datos los almacena en un formulario o en una base de datos que no admite de manera eficiente la consulta requerida.
- La necesidad de separar las preocupaciones significa que el servicio que posee los datos no es el servicio que debería implementar la operación de consulta.

## **API Composition**

Este patrón surge de la necesidad de realizar consultas con datos provenientes de varios servicios. Se propone implementar una operación de consulta invocando los servicios que poseen los datos y combinando los resultados. El funcionamiento de este patrón se basa en dos conceptos principales, el API composer que implementa la operación de consulta consultando al proveedor, y el provider service, un servicio que posee algunos de los datos que la consulta retorna [10].

La principal desventaja de este patrón es que algunas consultas resultan ineficientes dado que se deben realizar uniones de conjuntos de datos grandes en memoria.

### **A.0.4 Descomposición**

Se destacan dos patrones de descomposición: Descomponer en base a capacidades de negocio, que organiza los servicios en torno a las capacidades empresariales, y Descomponer en base a subdominios, que organiza los servicios en torno a subdominios de diseño impulsado por dominios (DDD).

### **Descomponer en base a las capacidades de negocio**

Este patrón establece que la forma de descomponer en servicios debe estar alineada a las capacidades de negocios. Cada servicio tiene que estar enfocado en una funcionalidad específica de las que brinda la aplicación y no a cómo esta se implementa.

Una capacidad de negocio de una organización es algo que esta realiza para generar valor. Las capacidades de negocio son en general estables, en diferencia de la forma en que una organización lleva a cabo sus negocios que es dinámica en el tiempo [5].

### **Descomponer en base a subdominios**

Este patrón propone identificar los subdominios en los que se compone la organización y atacar los problemas dentro de un dominio particular. Cada subdominio particular hace referencia a una característica del negocio [5].

Richardson clasifica a los subdominios en tres [20]:

- Core: Diferenciador clave para el negocio y la parte más valiosa de la aplicación.
- Supporting: Relacionado con que hace el negocio, pero no es un diferenciador.
- Generic: No es específico del negocio y se implementa idealmente con software estándar.

La idea es definir varios modelos que muestren las distintas visiones de cada concepto del negocio, ya que si se utiliza el enfoque tradicional en donde se define un solo modelo para mapear la realidad se puede estar sobre-generalizando algunos aspectos, los subdominios se identifican de la misma manera que las capacidades.

## **A.0.5 Descubrimiento de Servicios**

A su vez, los patrones para el descubrimiento de servicios son:

- Descubrimiento de servicios del lado del servidor
- Descubrimiento de servicios del lado del cliente
- Registro con herramientas de terceros
- Registro Personal

---

## Descubrimiento de servicios del lado del servidor

Este patrón propone que el cliente realice las solicitudes a un enrutador, que es responsable del descubrimiento de servicios [6].

El descubrimiento de servicios del lado del servidor tiene una serie de beneficios:

- En comparación con el descubrimiento del lado del cliente, el código del cliente es más simple ya que no tiene que lidiar con el descubrimiento. En cambio, un cliente simplemente realiza una solicitud al enrutador
- Algunos entornos de nube proporcionan esta funcionalidad

También tiene los siguientes inconvenientes:

- A menos que sea parte del entorno de la nube, el enrutador debe ser otro componente del sistema que debe instalarse y configurarse. También será necesario replicarlo por disponibilidad y capacidad.
- El enrutador debe admitir los protocolos de comunicación necesarios (por ejemplo, HTTP, gRPC, Thrift, etc.) a menos que sea un enrutador basado en TCP
- Se requieren más saltos de red que cuando se usa la descubrimiento del lado del cliente

## Descubrimiento de servicios del lado del cliente

Cuando un cliente desea invocar a un servicio, consulta el registro del servicio para obtener una lista de las instancias de este, para finalmente realizar la solicitud a la instancia de servicio seleccionada [6].

Como medida para mejorar el rendimiento, el cliente puede almacenar en caché las instancias del servicio, para luego utiliza un algoritmo de balanceo de carga para seleccionar una instancia del servicio.

El beneficio principal del patrón descubrimiento del lado del cliente es que se tienen menos piezas móviles y saltos de red en comparación con el descubrimiento del lado del servidor.

Por otro lado, también tiene los siguientes inconvenientes:

- El patrón vincula al cliente con el Registro de servicios.
- Se debe implementar la lógica de descubrimiento de servicios del lado del cliente para cada lenguaje / marco de programación utilizado por su aplicación.

## Registro con herramientas de terceros

Este patrón establece que un registrador externo es responsable de registrar y anular el registro de una instancia de servicio en el registro de servicios. Cuando se inicia la instancia de servicio, el registrador registra la instancia de servicio en el registro de servicios. Cuando se cierra la instancia de servicio, el registrador anula el registro de la instancia de servicio del registro de servicios [6].

Los beneficios del patrón de registro de terceros incluyen:

- El código de servicio es menos complejo ya que no es responsable de registrarse.
- El registrador puede realizar verificaciones de estado en una instancia de servicio y registrar / anular el registro de la instancia según la verificación de estado.

También presenta algunos inconvenientes como:

- Es posible que el registrador de terceros solo tenga un conocimiento superficial del estado de la instancia de servicio y, por lo tanto, es posible que no sepa si puede manejar solicitudes.
- A menos que el registrador sea parte de la infraestructura, es otro componente que debe instalarse, configurarse y mantenerse. Además, dado que es un componente crítico del sistema, debe tener una alta disponibilidad.

## Registro Personal

Siguiendo este enfoque una instancia de servicio invoca la API del registro de servicio para registrar su ubicación de red. También puede proporcionar una URL que sirve para comprobar el estado del servicio, que es un endpoint el cual el registro de servicio invoca periódicamente para verificar que la instancia del servicio esté en buen estado y disponible para manejar las solicitudes. Por lo general, el cliente debe renovar periódicamente su registro para que el registro sepa que aún está activo [6].

Este patrón presenta los siguientes inconvenientes:

- Acopla el servicio al Registro de servicios
- Debe implementar la lógica de registro de servicios en cada lenguaje / marco de programación que utilice para escribir sus servicios.

## A.0.6 Despliegue

Se han identificado varios patrones que facilitan el proceso de despliegue, y son listados a continuación:

- Servicios en máquinas virtuales
- Servicio como Contenedor
- Service mesh
- Plataforma para el despliegue de servicios
- Despliegue sin servidores

### Servicios en máquinas virtuales

Este patrón propone desplegar cada servicio como una imagen en una máquina virtual (MV) [34].

Los beneficios de realizar esto son:

- Es fácil de escalar aumentando el número de instancias
- Las instancias de los servicios se encuentran aisladas unas de otras
- No hay recursos compartidos entre instancias de microservicios y se puede asignar una cantidad específica de recursos a cada instancia
- Se encapsulan los servicios junto con las tecnologías

Por otro lado, este patrón también presenta ciertas desventajas que se describen a continuación:

- Se deben crear y administrar máquinas virtuales
- La utilización de recursos es menos eficiente
- La creación de una imagen de máquina virtual requiere mucho tiempo

### Servicio como Contenedor

Este patrón propone desplegar cada servicio como una imagen en un contenedor [33].

Este enfoque tiene los siguientes beneficios:

- Es fácil de escalar aumentando el número de instancias.
- Las instancias de los servicios se encuentran aisladas unas de otras
- Cada instancia tiene una cantidad fija de recursos
- La creación de una imagen de contenedor es mucho más rápida que la creación de una imagen de máquina virtual

### **Service mesh**

Según Red Hat, service mesh es "una forma de controlar como diferentes partes de una aplicación comparten datos entre sí". Se trata de una capa de infraestructura dedicada construida por encima de la aplicación. Esta capa puede obtener información acerca de como interactúan los diferentes servicios en una arquitectura de microservicios, facilitando la optimización de esta comunicación [38].

Un service mesh garantiza que la comunicación entre los servicios sea rápida, confiable y segura implementando varios patrones tratados en el documento como descubrimiento de servicios, circuit breaker y seguimiento distribuido [45].

### **Plataforma para el despliegue de servicios**

Se propone desplegar los servicios en plataformas de orquestación de contenedores. Se encuentra fuertemente relacionado con el patrón "Servicio como contenedor". Entre sus beneficios se destacan agilidad a la hora del despliegue, facilidad al momento de escalar y una mayor tolerancia a los fallos.

### **Despliegue sin servidores**

El patrón Despliegue sin servidores propone utilizar una infraestructura de implementación que oculte cualquier concepto de servidores: hosts físicos o virtuales, o contenedores. La infraestructura toma el código de su servicio y lo ejecuta [15].

Cuando se utilizan infraestructuras de despliegue sin servidores, estas se encargan de la asignación dinámica de los recursos y se cobra únicamente por la cantidad de recursos utilizados para ejecutar el código. De esta forma, se aumenta agilidad e innovación y se elimina el tiempo en el aprovisionamiento de servidores y mantenimiento de sistemas [53].

## A.0.7 Gestión de Datos

Una base de datos compartida y Una base de datos por servicio son los dos patrones incluidos en esta categoría.

### Una base de datos compartida

Este patrón propone utilizar una única base de datos la cual será compartida por todos los servicios que componen a la aplicación. Este patrón cuenta con numerosos inconvenientes entre los que encontramos:

- Acoplamiento del tiempo de desarrollo: un desarrollador que trabaje en un servicio deberá coordinar los cambios de esquema con los desarrolladores de otros servicios que acceden a las mismas tablas.
- Acoplamiento en tiempo de ejecución: debido a que todos los servicios acceden a la misma base de datos, pueden potencialmente interferir entre sí.
- Es posible que una sola base de datos no satisfaga los requisitos de acceso y almacenamiento de datos de todos los servicios.

Si bien es posible de implementar, si se analizan muchas de las características y ventajas que tiene la arquitectura de microservicios se observa que claramente lo propuesto por este patrón atenta contra muchas de ellas, todo lo provechoso que resulta el descomponer en sistemas autónomos más chicos para simplificar las pruebas, el desarrollo e incrementar la velocidad de los desarrolladores se vería comprometida por el cuello de botella que significa que todos los servicios persistan datos en una única base [4].

### Una base de datos por servicio

Esta forma de gestionar las bases de datos es mucho más cercana con los lineamientos y ventajas que propone la arquitectura de microservicios, motivo por el cual es ampliamente utilizado cuando a sistemas de este estilo se refiere.

Tomar la decisión de seguir este enfoque puede ser muy provechosa para el desempeño de una aplicación, sin embargo, junto con los beneficios que provee también introduce una serie de desafíos en cuanto a la consistencia. Para mantener dicha consistencia, este patrón debe estar acompañado del patrón Saga [4].

Hay varias formas de mantener la privacidad de los datos persistentes de un servicio. No es necesario que proporcione un servidor de base de datos para cada servicio. Por ejemplo, si está utilizando una base de datos relacional, las opciones son:

- Tablas privadas por servicio: cada servicio posee un conjunto de tablas a las que solo debe acceder ese servicio.
- Esquema por servicio: cada servicio tiene un esquema de base de datos que es privado para ese servicio.
- Servidor de base de datos por servicio: cada servicio tiene su propio servidor de base de datos.

Las tablas privadas por servicio y el esquema por servicio tienen la sobrecarga más baja. El uso de un esquema por servicio es atractivo ya que aclara la propiedad. Algunos servicios de alto rendimiento pueden necesitar su propio servidor de base de datos.

### A.0.8 Consistencia

Dentro de la categoría Consistencia encontramos únicamente al patrón SAGA.

#### SAGA

El patrón SAGA surge ante la necesidad de tener transacciones en sistemas distribuidos. Es una alternativa asíncrona del 2PC [43]. Consiste en una secuencia de transacciones locales en cada sistema, basándose en cada sistema asegure las propiedades ACID [5].

Es necesario definir una saga para cada transacción que quiera actualizar datos de más de un servicio, y se coordina usando mensajería asíncrona. Cada transacción local actualiza la base de datos y publica un mensaje o evento para activar la siguiente transacción local de la saga. Si ocurre un error en alguno de los pasos intermedios (transacciones locales intermedias) la saga se recupera mediante transacciones compensatorias que se ejecutan en orden inverso a la secuencia original.

Existen 2 tipos de implementaciones:

- Basada en coreografía: No existe coordinador central ordenando a los sistemas. En cambio los sistemas se suscriben a los eventos de los demás y responden correspondientemente.
- Basada en orquestación: Existe un coordinador central que se encarga de diagramar los pasos a seguir de los sistemas para realizar la transacción.

### A.0.9 Implementación

Los patrones que resuelven estos problemas son: Aggregates, Domain Event, Event Sourcing y Transaction Script.



## Aggregates

Este patrón sugiere estructurar la lógica en un conjunto de aggregates. Richardson los define como un conjunto de objetos de dominio dentro de un límite que pueden ser tratados como una sola unidad [8]. Esto genera que toda manipulación sobre un objeto concreto deba realizarse sobre el aggregate que compone, teniendo como consecuencia directa el ser cuidadoso a la hora de definir los aggregates en los cuales se descompondrá la aplicación.

Hay dos razones por las que los aggregates son útiles al desarrollar lógica empresarial en una arquitectura de microservicios [8]:

- Evitan cualquier posibilidad de referencias a objetos que se extiendan por los límites del servicio, porque una referencia entre aggregates es un valor de clave principal en lugar de una referencia de objeto.
- Dado que una transacción solo puede crear o actualizar un único aggregate, estos se ajustan a las restricciones del modelo de transacción de microservicios.

Por otra parte, lo antes mencionado se basa en que el uso de la identidad en lugar de referencias a objetos significa que los aggregates están débilmente acoplados, asegurando que los límites de alcance entre aggregates estén bien definidos.

Finalmente, una transacción puede crear o actualizar un solo aggregate, lo cual tiene como gran beneficio que asegura que una transacción esté contenida dentro de un solo servicio. Como desventaja se destaca se hace más complicado implementar operaciones que necesitan crear o actualizar múltiples agregados.

## Domain Event

La definición a la que todos los autores hacen referencia sobre este concepto es a la de Eric Evans: Un enfoque, para construir aplicaciones de software complejas, centrado en el desarrollo de un modelo de dominio orientado a objetos [39].

Básicamente un modelo de dominio grande que contiene varios subdominios. Para cada servicio se define su propio modelo de dominio independiente del resto y juntos (los de todos los servicios) forman el modelo general de la aplicación.

Un evento es algo que ya ocurrió. Un evento de dominio es algo que ocurrió en el dominio y necesariamente otras partes del mismo dominio estén al tanto de lo que ocurrió. Un evento es un cambio de estado. En particular, este patrón se enfoca en que cada aggregate publique un evento al ser creado o enfrenta un cambio. Los eventos de dominio sirven cuando requerimos mantener la consistencia de los datos a través de los servicios, enviar notificaciones a diferentes servicios.

## Event Sourcing

En este enfoque se propone que la lógica se implemente centrada en el concepto de evento y propone incluir el concepto de aggregate para descomponer la lógica. En este contexto, los aggregates se persisten en la base de datos como una serie de eventos, donde cada evento representa un cambio de estado de ese aggregate [9].

Los eventos son persistidos en base de datos y en el momento que se requiera recuperar un aggregate se recuperan los eventos que lo mapean y se ejecutan secuencialmente hasta obtener el estado explicitado en el aggregate. Esto se puede lograr ya que cada componente cuando realiza un cambio crea un nuevo evento que mapea dicho cambio y lo persiste en la base de datos de eventos, por lo cual con la lista histórica de eventos se puede recrear cualquier estado anterior.

Al aplicar este patrón se obtiene un historial de cambios completo de cada aggregate existente, lo cual puede ser muy provechoso a la hora de implementar mecanismos de auditoría y seguridad. A su vez, el patrón aporta beneficios en lo que respecta a la potencial aparición de errores derivados de la lógica de persistencia ya que propone que esta no se mantenga embebida en la lógica de la aplicación.

## Transaction Script

Transaction Script es un patrón aplicable en aplicaciones pequeñas, donde la lógica es simple y aplicar un enfoque orientado a objetos puede derivar en complejizar demasiado la implementación. Consta en organizar la lógica a base de procedimientos que se encargan de las invocaciones de la capa de presentación y donde cada procedimiento se encarga de un único caso de uso. Estos procedimientos se encargan de toda la lógica: procesar datos de entrada, realizar invocaciones a la base de datos y devolver datos de respuesta [8] [32].

### A.0.10 Interfaz de Usuario

Existen dos patrones que resuelven este problema: Client-side UI Composition y Server-side Page Fragment Composition.

#### Client-side UI Composition

Cada equipo encargado de un servicio desarrolla el componente de interfaz de usuario que presenta datos del servicio en cuestión. Por otra parte, existe un equipo de UI general que es responsable de implementar los esqueletos de página utilizando los componentes desarrollados por los múltiples equipos de servicios [29] [22].

---

## Server-side page fragment composition

Cada equipo desarrolla una aplicación web que genera el fragmento HTML que implementa la región de la página principal que presenta información de su servicio. Existe un equipo de UI general responsable de desarrollar las plantillas que generan páginas mediante la agregación del lado del servidor de los fragmentos HTML específicos del servicio [23].

### A.0.11 Mensajería Transaccional

Transactional outbox, Polling Publisher y Transaction Log Tailing son los patrones que resuelven este problema.

#### Transactional outbox

El patrón Transaccional Outbox utiliza una tabla de base de datos como una cola de mensajes temporal. El servicio que envía mensajes tiene una tabla de base de datos OUTBOX. Como parte de la transacción de la base de datos que crea, actualiza y elimina objetos, el servicio envía mensajes insertando un nuevo registro en la tabla OUTBOX. La atomicidad está garantizada por definición de transacción local ya que son ACID [6].

Por otra parte, se puede utilizar un enfoque similar con algunas bases de datos NoSQL. Esto se puede lograr haciendo que cada entidad almacenada como un registro en la base de datos tenga un atributo y su valor sea una lista de mensajes que deben publicarse. Cuando un servicio actualiza un objeto en la base de datos, agrega un mensaje a esa lista. Esta acción es atómica porque se realiza con una sola operación de base de datos.

#### Polling Publisher

Si la aplicación utiliza una base de datos relacional, una forma muy simple de publicar los mensajes insertados en la tabla OUTBOX es sondear periódicamente la tabla en busca de mensajes no publicados. Cuando se detectan mensajes no publicados estos se envían al broker de mensajería y se eliminan de la tabla OUTBOX [6].

Este es un enfoque simple que funciona razonablemente bien a baja escala. La gran desventaja es que consultar frecuentemente la base de datos puede ser costoso. Además, el hecho de que se pueda usar este enfoque con una base de datos NoSQL depende de sus capacidades de consulta. Esto sucede porque en lugar de consultar una tabla OUTBOX, la aplicación debe consultar las entidades en sí, y eso puede o no ser posible hacerlo de manera eficiente.

## Transaction Log Tailing

Este patrón sugiere realizar un seguimiento del registro de transacciones de la base de datos y publicar cada mensaje / evento insertado en la bandeja de salida para el intermediario de mensajes siguiendo el log de transacciones [24].

### A.0.12 Migración hacia microservicios

Para realizar este proceso existen dos patrones que se pueden seguir: Strangler application y Anti Corruption Layer.

#### Strangler application

Martin Fowler describe este patrón como el proceso de ir creando un nuevo sistema gradualmente alrededor del viejo y que hacer que este crezca lentamente hasta que el nuevo sistema “estrangule” al viejo [31].

La visión de Richardson es de un enfoque incremental donde en el proceso se genera una aplicación denominada strangler (estranguladora) que consiste en microservicios que no solo implementan nuevas funcionalidades que extienden a la aplicación original monolítica, si no que también implementa funcionalidades del monolito que se extraen de éste con el fin de descomponerlo y eventualmente eliminarlo. Cuando un servicio que forma parte del monolito requiere un cambio en su funcionalidad que provoca que se le realice mantenimiento, es seleccionado para ser trasplantado a microservicios [16].

A medida que pasa el tiempo el tamaño del monolito va decreciendo teniendo como consecuencia una aplicación totalmente en microservicios.

Este patrón consiste de tres pasos: Transformar, coexistir y eliminar. Coexisten servicios implementados bajo los dos enfoques durante el transcurso de la transición.

Beneficios:

- Demuestra valor real rápido y seguido: al implementar las nuevas funcionalidades como microservicios, se puede aprovechar para desarrollar con tecnologías modernas.
- Minimizar cambios al monolito: al implementar los servicios separados se minimiza el trabajo de mantenimiento del monolito el cual necesita periódicamente refactorizar.
- No se necesita una infraestructura de despliegue desde el principio: cómo se va migrando de a poco se puede comenzar con una configuración básica de la infraestructura de despliegue que permita irse adentrando en el mundo de los microservicios sin arriesgar en una inversión fuerte desde el principio.

---

## Anti corruption layer

El patrón Anti corruption layer propone implementar una capa de software que traduce entre dos sistemas que contienen dos modelos de dominio diferentes: El sistema nuevo desarrollado en microservicios y un sistema legado [16].

Se utiliza una capa entre los dos sistemas para prevenir que el sistema nuevo sea “contaminado” por el sistema legado. Cuando se tiene un servicio ejecutando que tiene un esquema de datos distinto al monolito es probable que alguna solicitud involucre entidades que se mapean de forma diferente. Por este motivo es necesario traducir dichos conceptos que entran en conflicto para lograr un correcto funcionamiento.

De la misma manera, se utiliza para traducir eventos por el mismo motivo antes mencionado, por ejemplo, se reciben los eventos en el formato que el monolito los genera y se traducen al formato esperado.

Tiene las desventajas de que puede añadir latencia a las llamadas entre los dos sistemas y se agrega un componente extra al sistema que necesita mantenimiento.

### A.0.13 Monitoreo

Los patrones identificados dentro de esta categoría son:

- Exception Tracking (Seguimiento de excepciones)
- Seguimiento distribuido (Distributed tracing)
- Métricas de la aplicación
- Audit Logging
- Health Check API (API de estado de salud)
- Log aggregation (Acumulación de logs)

#### Exception Tracking (Seguimiento de excepciones)

El patrón Exception Tracking propone informar todas las excepciones a un servicio de seguimiento de excepciones centralizado que agrega y rastrea las excepciones y notifica a los desarrolladores [14].

### **Seguimiento distribuido (Distributed tracing)**

Seguimiento distribuido consiste en instrumentar servicios con código que asigna a cada solicitud externa un identificador único que se pasa entre servicios. De esta forma se proporciona información útil sobre el comportamiento del sistema, como las fuentes de latencia, y permite a los desarrolladores ver cómo se maneja cada solicitud individualmente [14].

### **Métricas de la aplicación**

Los servicios son los responsables de enviar sus propias métricas a un servidor de métricas central que recopila estadísticas sobre las operaciones. Concretamente, el equipo encargado del desarrollo del servicio es el responsable de recolectar y exponer las métricas de dicho servicio [14].

De esta manera se tiene centralizado un análisis primario de cada servicio lo cual permite una gran ventaja a la hora de aprovechar el uso de la información recolectada en lo que respecta a la dificultad de acceso a este tipo de datos.

### **Audit Logging**

Audit Logging sigue un enfoque similar a la mayoría de los patrones de monitoreo. Propone loguear las acciones de los usuarios en un repositorio central dado que es útil saber qué acciones ha realizado un usuario recientemente: atención al cliente, cumplimiento, seguridad, etc [14].

### **Health Check API (API de estado de salud)**

El patrón Health Check API propone exponer un endpoint con el propósito de notificar el estado actual del servicio. De esta manera se proporciona una herramienta para conocer en cualquier momento y en tiempo de ejecución el estado de los servicios [14].

Esta tarea tiene responsabilidad compartida entre el equipo de desarrollo quien se debe responsabilizar de la implementación del endpoint en concreto y del equipo de operaciones que periódicamente debe realizar invocaciones al mismo para verificar el estado del servicio.

### **Log Aggregation (Acumulación de logs)**

Este patrón propone utilizar un servicio de registro centralizado que agregue registros de la actividad de cada instancia de servicio, y permitir a los usuarios configurar alertas realizar

---

búsquedas en dichos registros. El mayor inconveniente que se presenta es que el manejo de un gran volumen de registros requiere una infraestructura sustancial [14].

### **A.0.14 Preocupaciones Transversales**

Esta categoría abarca los patrones Configuración Externalizada y Microservices Chasis.

#### **Configuración Externalizada**

Se propone sustituir el archivo de configuración tradicional que contiene, por ejemplo, las credenciales de la base de datos o la posición en la red de la entidad por un servicio que se consulte en tiempo de ejecución. Por lo cual la configuración se engloba en un componente independiente al cual acceden los servicios para obtener la configuración apropiada [14].

Se identifican dos variantes:

- Push model: La infraestructura de implementación pasa las propiedades de configuración a la instancia de servicio utilizando, por ejemplo, variables de entorno del sistema operativo o un archivo de configuración.
- Pull model: La instancia de servicio lee sus propiedades de configuración de un servidor de configuración.

#### **Microservices Chasis**

Al desarrollar varios servicios diferentes, se termina configurando y desarrollando varias veces lo mismo en varias de estas preocupaciones transversales. Lo único que genera esto es que se pierda más tiempo a la hora de desarrollar un servicio.

Este patrón propone la utilización de distintos frameworks de manera conjunta teniendo como objetivo eliminar del alcance del desarrollador de que tenga que configurar todas las preocupaciones transversales en cada microservicio [25].

Generalmente ese conjunto de herramientas que propone utilizar se conoce como un stack tecnológico y tiene como principal desventaja que por cada lenguaje que la aplicación maneje se debe manejar potencialmente un stack distinto por lo cual la complejidad tecnológica no es para nada despreciable.

La mayor ventaja que presenta esta patrón es una reducción en el costo de desarrollo del servicio y el desacoplamiento de la lógica con la infraestructura del sistema

## A.0.15 Seguridad

Para la categoría de seguridad se relevó un único patrón: Access Token.

### Access Token

El API Gateway autentica al usuario otorgando un token mediante el cual puede realizar invocaciones. Cuando el API Gateway recibe una invocación con un token autorizado reenvía dicho token a los servicios destino, logrando así un correcto manejo de la autenticación de los usuarios. Un servicio puede incluir el token de acceso en las solicitudes que realiza a otros servicios [14].

Entre los beneficios de este patrón se tiene que la identidad del solicitante se transmite de forma segura por el sistema y los servicios pueden verificar que el solicitante está autorizado para realizar una operación.

## A.0.16 Testing

A continuación se listan patrones para simplificar las pruebas probando los servicios de forma aislada:

- Consumer driven contract test
- Consumer-side contract test
- Service Component Test

### Consumer driven contract test

Como contexto para el desarrollo de este enfoque se utiliza una situación en la cual se tiene un servicio que provee un servicio y otro que lo consume. Como primer medida estos dos servicios que van a interactuar deben acordar los endpoints utilizados para la comunicación entre ambos.

En este enfoque se propone generar tests de integración para el proveedor que verifiquen que efectivamente la API que se expone sea la esperada por el consumidor. Surge para mitigar el problema de testear un servicio que tiene dependencias complejas, y a su vez evitar generar pruebas del estilo end-to-end por la complejidad que estos aportan [12].

Se introducen algunos conceptos como el contrato en donde se estipula el motivo del test y cómo se realizará. En este tipo de test el objetivo no es verificar a fondo la lógica del proveedor, esto se deriva a los test unitarios. Lo novedoso de este enfoque es que quien



---

desarrolla el test es el equipo que gestiona el servicio consumidor, ya que dicho servicio es quien consumirá los datos proveídos por el proveedor se delega la responsabilidad de que sea el equipo encargado de la implementación de dicho servicio el cual desarrolle test verificando que efectivamente la respuesta del proveedor es la esperada.

Concretamente, se crean tests y se los agrega mediante un mecanismo el cual debe definirse a un repositorio del proveedor en donde se alojarán todo el conjunto de tests implementados por los diferentes consumidores, posteriormente el equipo encargado del proveedor tendrá la responsabilidad de ejecutar dichas pruebas y en caso de fallo deberá corregir el servicio proveedor para satisfacer los requerimientos del consumidor o en su defecto comunicarse con el equipo que implementó dicho test.

### **Consumer-side contract test**

El objetivo de este patrón es verificar que el consumidor de un servicio (cliente) pueda comunicarse con el servicio. Para ello se propone utilizar contratos para testear también al consumidor, generando un servicio stub que testea la API del mismo [12].

A simple vista, puede parecer mejor definir la interacción entre dos servicios usando esquemas escritos, por ejemplo con OpenAPI o JSON. Sin embargo, resulta que los esquemas no son tan útiles al escribir tests ya que si bien se puede validar la respuesta utilizando el esquema, aún se necesita invocar al proveedor con una solicitud de ejemplo.

Además, surge la necesidad de tener respuestas de ejemplo. Esto se debe a que, aunque el objetivo de Consumer driven contract test es evaluar a un proveedor, los contratos también se utilizan para verificar que el consumidor cumple con el dicho contrato. Probar ambos lados de la interacción asegura que el consumidor y el proveedor estén de acuerdo con la API.

### **Service Component Test**

En el contexto de las aplicaciones de microservicios se tienen múltiples servicios que interactúan entre sí para implementar funcionalidades que la aplicación brinda. Una vez que cada servicio se probó unitariamente, es necesario comprobar que el comportamiento de los servicios es el adecuado en el ecosistema en donde coexisten.

Es por este motivo que surge la necesidad de probar a los servicios considerándolos cajas negras y verificar la interacción de los mismos con los demás servicios a través de sus APIs. Un enfoque podría ser escribir lo que se denominan esencialmente pruebas de extremo a extremo e implementar el servicio que se desea probar junto con todas sus dependencias transitivas, pero tiene la desventaja de ser lento, frágil y costoso de implementar para probar un servicio. Por lo cual, una variante más adecuada es utilizar este patrón que plantea el verificar el comportamiento de un servicio de forma aislada, reemplazando las dependencias del mismo con stubs que simulan su comportamiento [13].

## **A.0.17 Tolerancia a Fallos**

Para resolver estos problemas se suele utilizar el patrón Circuit Breaker.

### **Circuit Breaker**

Este patrón plantea llevar un registro de las invocaciones exitosas y fallidas. Si el ratio de fallas supera un umbral estipulado previamente el circuito “se abre”, para evitar que se propague la falla. Cuando se supera este umbral antes mencionado el Circuit Breaker hace que todas las próximas invocaciones fallen ante la vista del cliente y en realidad evita seguir sobrecargando el servicio caído no invocándolo por un tiempo definido. Después de un periodo de tiempo, el circuito al recibir una nueva solicitud intenta nuevamente invocar al servicio, si tiene éxito se cierra el circuito y se vuelve al funcionamiento normal, si no se sigue en el estado mencionado anteriormente [6].

# Referencias

- [1] Diego Verdier, Gonzalo Rodríguez. *Implementación de Patrones de Microservicios*, 2020.
- [2] Evangelos Ntontos, Uwe Zdun, Konstantinos Plakidas, Daniel Schall, Fei Li and Sebastian Meixner. *Supporting Architectural Decision Making on Data Management in Microservice Architectures*, 2019.
- [3] Chris Richardson. *Microservices Patterns With examples in Java*. Manning Publications, 2019.
- [4] Chris Richardson. *Microservices Patterns With examples in Java*. Manning Publications, chapter 1, 2019.
- [5] Chris Richardson. *Microservices Patterns With examples in Java*. Manning Publications, chapter 2, 2019.
- [6] Chris Richardson. *Microservices Patterns With examples in Java*. Manning Publications, chapter 3, 2019.
- [7] Chris Richardson. *Microservices Patterns With examples in Java*. Manning Publications, chapter 4, 2019.
- [8] Chris Richardson. *Microservices Patterns With examples in Java*. Manning Publications, chapter 5, 2019.
- [9] Chris Richardson. *Microservices Patterns With examples in Java*. Manning Publications, chapter 6, 2019.
- [10] Chris Richardson. *Microservices Patterns With examples in Java*. Manning Publications, chapter 7, 2019.
- [11] Chris Richardson. *Microservices Patterns With examples in Java*. Manning Publications, chapter 8, 2019.
- [12] Chris Richardson. *Microservices Patterns With examples in Java*. Manning Publications, chapter 9, 2019.
- [13] Chris Richardson. *Microservices Patterns With examples in Java*. Manning Publications, chapter 10, 2019.

- 
- [14] Chris Richardson. *Microservices Patterns With examples in Java. Manning Publications, chapter 11, 2019.*
- [15] Chris Richardson. *Microservices Patterns With examples in Java. Manning Publications, chapter 12, 2019.*
- [16] Chris Richardson. *Microservices Patterns With examples in Java. Manning Publications, chapter 13, 2019.*
- [17] Chris Richardson. *Blog.* [En línea] Disponible en: <https://microservices.io> [Accedido: 03-10-2021]
- [18] Chris Richardson. *A pattern language for microservices.* [En línea] Disponible en: <https://microservices.io/patterns/index.html> [Accedido: 03-10-2021]
- [19] Chris Richardson. *Microservices Assessment Platform.* [En línea] Disponible en: <https://microservices.io/platform/microservice-architecture-assessment.html> [Accedido: 17-11-2021]
- [20] Chris Richardson. *Pattern: Decompose by subdomain* [En línea] Disponible en: <https://microservices.io/patterns/decomposition/decompose-by-subdomain.html> [Accedido: 13-10-2021]
- [21] Chris Richardson. *Pattern: Service instance per container* [En línea] Disponible en: <https://microservices.io/patterns/deployment/service-per-container.html> [Accedido: 13-10-2021]
- [22] Chris Richardson. *Pattern: Client-side UI composition* [En línea] Disponible en: <https://microservices.io/patterns/ui/client-side-ui-composition.html> [Accedido: 13-10-2021]
- [23] Chris Richardson. *Pattern: Server-side page fragment composition* [En línea] Disponible en: <https://microservices.io/patterns/ui/server-side-page-fragment-composition.html> [Accedido: 14-10-2021]
- [24] Chris Richardson. *Pattern: Transaction log tailing* [En línea] Disponible en: <https://microservices.io/patterns/data/transaction-log-tailing.html> [Accedido: 14-10-2021]
- [25] Chris Richardson. *Pattern: Microservice chassis* [En línea] Disponible en: <https://microservices.io/patterns/microservice-chassis.html> [Accedido: 14-10-2021]
- [26] Martin Fowler. *Blog.* [En línea] Disponible en: <https://martinfowler.com>

- com [Accedido: 06-10-2021]
- [27] M. Fowler y J Lewis. *Microservices, 2014*. [En línea] Disponible en: <http://martinfowler.com/articles/microservices.html> [Accedido: 06-10-2021]
- [28] Martin Fowler - Microservice Trade-Offs *Microservice Trade-Offs, 2015*. [En línea] Disponible en: <https://martinfowler.com/articles/microservice-trade-offs.html> [Accedido: 06-10-2021]
- [29] Martin Fowler. *Micro Frontends, 2019*. [En línea] Disponible en: <https://martinfowler.com/articles/micro-frontends.html> [Accedido: 06-10-2021]
- [30] Martin Fowler. *CQRS, 2011*. [En línea] Disponible en: <https://martinfowler.com/bliki/CQRS.html> [Accedido: 07-10-2021]
- [31] Martin Fowler. *StranglerFigApplication, 2004*. [En línea] Disponible en: <https://martinfowler.com/bliki/StranglerFigApplication.html> [Accedido: 07-10-2021]
- [32] Martin Fowler. *Transaction Script* [En línea] Disponible en: <https://martinfowler.com/eaCatalog/transactionScript.html> [Accedido: 07-10-2021]
- [33] Microservices Deployment in a Glance, Rahul Shetty [En línea] Disponible en: <https://www.commonlounge.com/discussion/722433ee5c3c4664852a2d79d2bbe444#a-single-instance-of-microservice-per-container> [Accedido: 02-11-2021]
- [34] Microservices Deployment in a Glance, Rahul Shetty [En línea] Disponible en: <https://www.commonlounge.com/discussion/722433ee5c3c4664852a2d79d2bbe444#a-single-instance-of-a-microservice-per-vm> [Accedido: 02-11-2021]
- [35] Microservices Deployment in a Glance, Rahul Shetty [En línea] Disponible en: <https://www.commonlounge.com/discussion/722433ee5c3c4664852a2d79d2bbe444> [Accedido: 02-11-2021]
- [36] Red Hat, Inc. *What is CI/CD?* [En línea] Disponible en: <https://www.redhat.com/en/topics/devops/what-is-ci-cd> [Accedido: 23-11-2021]
- [37] Red Hat, Inc. *What is container orchestration?* [En línea] Disponible en: <https://www.redhat.com/en/topics/containers/>

`what-is-container-orchestration` [Accedido: 23-11-2021]

- [38] Red Hat, Inc. *What's a service mesh?* [En línea] Disponible en: <https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh> [Accedido: 23-11-2021]
- [39] Microsoft *Domain events: design and implementation* [En línea] Disponible en: <https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/domain-events-design-implementation> [Accedido: 6-12-2021]
- [40] Microsoft Corporation. *What is a container?* [En línea] Disponible en: <https://azure.microsoft.com/en-us/overview/what-is-a-container/#overview> [Accedido: 08-12-2021]
- [41] Microsoft Corporation. *Patrón MVC* [En línea] Disponible en: [https://docs.microsoft.com/es-es/aspnet/core/mvc/overview?WT.mc\\_id=dotnet-35129-website&view=aspnetcore-5.0#mvc-pattern](https://docs.microsoft.com/es-es/aspnet/core/mvc/overview?WT.mc_id=dotnet-35129-website&view=aspnetcore-5.0#mvc-pattern) [Accedido: 09-12-2021]
- [42] Microsoft Corporation. *Backends for Frontends pattern* [En línea] Disponible en: <https://docs.microsoft.com/en-us/azure/architecture/patterns/backends-for-frontends> [Accedido: 09-12-2021]
- [43] IBM Corporation. *Two-phase commit, 2014.* [En línea] Disponible en: <https://www.ibm.com/docs/en/zos/2.1.0?topic=transactions-two-phase-commit> [Accedido: 01-12-2021]
- [44] IBM Corporation. *Devops.* [En línea] Disponible en: <https://www.ibm.com/cloud/learn/devops-a-complete-guide> [Accedido: 01-12-2021]
- [45] NGINX *What Is a Service Mesh?* [En línea] Disponible en: <https://www.nginx.com/blog/what-is-a-service-mesh/> [Accedido: 01-12-2021]
- [46] Chris Richardson. *Eventuate.* [En línea] Disponible en: <https://eventuate.io/> [Accedido: 08-12-2021]
- [47] Refactoring Guru *Design Patterns* [En línea] Disponible en: <https://refactoring.guru/design-patterns> [Accedido: 17-11-2021]
- [48] H. Mu and S. Jiang. *Design patterns in software development. IEEE 2nd International Conference on Software Engineering and Service Science, Beijing, 2011.*

- 
- [49] Philippe Kruchten. *The 4+1 view model of architecture*. *IEEE Software* 12 (6), pages 42-50, 1995.
- [50] Yoandy Pérez Villazón. *Patrones para la implementación de una arquitectura basada en microservicios*, 2020.
- [51] Mark Richards. *Software Architecture Patterns: Understanding common architecture patterns and when to use them*, 2015.
- [52] Mati Paz Wasiuchnik *Bases de datos monolíticas en un mundo de microservicios* [En línea] Disponible en: <https://medium.com/@matipazw/bases-de-datos-monolíticas-en-un-mundo-de-microservicios-5e> [Accedido: 20-12-2021]
- [53] Serverless Stack *What is Serverless?* [En línea] Disponible en: <https://serverless-stack.com/chapters/what-is-serverless.html> [Accedido: 26-11-2021]
- [54] Sam Newman *Pattern: Backends For Frontends* [En línea] Disponible en: <https://samnewman.io/patterns/architectural/bff/> [Accedido: 26-11-2021]
- [55] International Organization for Standardization. *ISO/IEC 25010, 2011*