



UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

DIVISIÓN DE ESTUDIOS DE POSGRADO

POSGRADO EN CIENCIAS E INGENIERÍA DE LA COMPUTACIÓN

TESIS

Diseño de una metodología para la migración de aplicaciones web en Java a microservicios

QUE PARA OBTENER EL TÍTULO DE:

MAESTRO EN CIENCIA E INGENIERÍA EN COMPUTACIÓN

PRESENTA:

**ING. CARLOS EDUARDO ROMERO
CASANOVA**

DIRECTOR:

**M. C. GUSTAVO ARTURO MÁRQUEZ
FLORES**

Ciudad de México, 5 de marzo de 2020

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN, UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

Tesis para obtener el grado de Maestro en Ciencias e Ingeniería en Computación
Primera Edición, 5 de marzo de 2020

Prefacio

.....

Agradecimientos

A mis padres

A mis amigos

A la UNAM

A CONACYT

Índice general

1. Introducción	1
1.1. Objetivo	2
1.1.1. Objetivo principal	2
1.1.2. Objetivos específicos	2
1.2. Problema u Oportunidad	2
1.3. Hipótesis	3
1.4. Contribución y relevancia	3
2. Marco teórico	5
2.1. Aplicación web	5
2.1.1. Definición	5
2.1.2. Aplicación web con arquitectura monolítica	6
2.1.3. Funcionamiento	6
2.1.4. Evolución de las aplicaciones web	7
2.1.4.1. Arquitectura de un sólo nivel	9
2.1.4.2. Arquitectura de dos niveles	9
2.1.4.3. Arquitectura de tres niveles	10
2.1.4.4. Arquitectura de n niveles	11
2.1.4.5. Arquitectura de Java Enterprise Edition	12
2.2. Microservicios	13
2.2.1. ¿Qué son los microservicios?	13
2.2.2. Arquitectura y funcionamiento de los microservicios	14
2.2.3. Breve historia de los microservicios	15
2.2.4. Beneficios y características	16
2.2.5. Diferencias entre microservicios y SOA	18
2.2.6. Diferencias entre microservicios y arquitectura monolítica	18
2.2.7. Ventajas en relación a la arquitectura monolítica	19

2.2.8. Desventajas respecto a la arquitectura monolítica	20
3. Patrones y Antipatrones de diseño	23
3.1. Patrones de diseño	23
3.1.1. ¿Qué es un patrón de diseño?	23
3.1.2. Patrones comúnmente usados en Aplicaciones Web Java Enterprise Edition	24
3.1.2.1. Patrón Modelo Vista Controlador	24
3.1.2.2. Patrones de la capa de presentación	25
3.1.2.3. Patrones de la capa de negocios	30
3.1.2.4. Patrones de la capa de integración	33
3.1.3. Patrones en Microservicios	34
3.1.4. Patrón Strangler	36
3.1.5. Patrón de descomposición por habilidad de negocio	37
3.2. Antipatrones de diseño	37
3.2.1. ¿Qué es un antipatrón de diseño?	37
3.2.2. Antipatrones en Aplicaciones Web	38
4. Estado del Arte	47
4.1. Lenguajes de programación en construcción de software	47
4.2. Ecosistema JVM	48
4.3. Microservicios Java	55
4.3.1. Modelo <i>Bounded Context</i>	56
5. Diseño del método de migración	59
5.1. Método propuesto	59
5.2. Etapa 1: Verificación de elegibilidad de la aplicación web para la migración	60
5.2.1. Requisitos de la aplicación web	61
5.2.2. Consideraciones antes de la migración	63
5.3. Etapa 2: Transformación de la aplicación web a microservicios	63
5.3.1. Subetapa 1: Construcción del modelo Bounded Context	63
5.3.2. Subetapa 2: Elección del objetivo de negocio a migrar	64
5.3.3. Subetapa 3: Implementación del Patrón Strangler	65
5.3.3.1. Fase de transformación:	65
5.3.3.2. Fase de coexistencia:	67
5.3.3.3. Fase de eliminación:	67
5.4. Etapa 3: Verificación y validación de la aplicación resultante	68

6. Implementación del método propuesto	69
6.1. Método	69
7. Conclusiones y trabajo futuro	71
7.1. Conclusiones	71
7.2. Trabajo futuro	72
Glosario	73
Acrónimos	77

Índice de figuras

2.1. Aplicación web de arquitectura monolítica. Extraída y traducida de [46]	6
2.2. Componentes y funcionamiento de las aplicaciones web. Extraída y traducida de [1]	7
2.3. Arquitectura de 3-niveles. Extraída y traducida de [61]	8
2.4. Arquitectura de 1-nivel. Extraída y traducida de [48]	9
2.5. Arquitectura de dos niveles de tipo cliente pesado. Extraída y traducida de [48]	10
2.6. Arquitectura de tres niveles. Extraída y traducida de [48]	11
2.7. Aplicación de n niveles. Extraída y traducida de [48]	12
2.8. Arquitectura de Java EE. Extraída y traducida de [18]	13
2.9. Arquitectura de los Microservicios. Basado en [47, 60]	15
2.10. Arquitectura SOA. Extraído de [56]	17
2.11. Microservicios vs SOA. Extraído de [51]	19
2.12. Arquitectura monolítica y arquitectura de microservicios. Extraído de [47]	20
3.1. Modelo Vista Controlador. Extraído de [48]	25
3.2. Patrón View Helper. Extraído de [48]	26
3.3. Patrón Composite View.	28
3.4. Estructura del Front Controller. Extraída de [48]	29
3.5. Flujo de trabajo del Action Handler. Extraído de [48]	30
3.6. Flujo de trabajo del View Handler. Extraído de [48]	31
3.7. Estructura del Page Controller. Extraído de [30]	32
3.8. Diagrama de clases del Patrón Context Object. Extraído de [43]	33
3.9. Diagrama de clases del Patrón Business Delegate. Extraído de [20]	34
3.10. Diagrama de clases del Patrón Session Facade. Extraído de [23]	35
3.11. Diagrama de clases del Patrón Application Service. Extraído de [29] . .	36

3.12. Diagrama de clases del Patrón Business Interface. Extraído de [48]	37
3.13. Ejemplo del Patrón DAO. Extraído de [54]	38
3.14. Diagrama de clases del Patrón Procedure Access Object. Extraído de [37]	39
3.15. Patrón Service Activator. Extraído de [22]	40
3.16. Patrón API Gateway. Extraído de [10]	41
3.17. Diagrama de secuencias del Patrón Circuit Braker. Extraído de [2]	42
3.18. Database per service. Extraído de [35]	43
3.19. Patrón Strangler. Extraído de [34]	43
3.20. Implementación del patrón strangler. Extraído de [34]	44
3.21. UML	44
4.1. Proveedores JDK usados. Extraída de [53]	48
4.2. Versiones de Java SE utilizados en producción. Extraída de [53]	49
4.3. Versiones de Java EE utilizados en producción. Extraída de [53]	50
4.4. Herramientas de automatización usadas. Extraída de [53]	50
4.5. Herramientas de pruebas usadas. Extraída de [53]	51
4.6. Enfoques de la nube implementados. Extraída de [53]	52
4.7. Frameworks ORM. Extraída de [53]	52
4.8. Frameworks web más usados. Extraída de [53]	53
4.9. Bases de datos usadas. Extraída de [53]	54
4.10. Bases de datos usadas. Extraída de [53]	55
4.11. Patrones de diseño en Spring Cloud. Extraído de [67]	56
4.12. Modelo 4+1 de Arquitectura de Software. Extraído y traducido de [50]	57
4.13. Pasos descritos para la identificación y definición de microservicios. Extraído de [60]	58
5.1. Diagrama de flujo general del método de migración.	60
5.2. Proceso detallado del método de migración propuesto.	62

CAPÍTULO
1

Introducción

En la actualidad las aplicaciones web han ido incrementándose de tal manera que ya forma parte de las actividades cotidianas de la mayoría de las personas, como las compras en línea, pagos de servicios, consultas bancarias en internet, etc., de tal manera que el uso de éstas es casi fundamental para realizarlas. Por lo tanto, en la actualidad existen cientos de lenguajes de programación en los que las aplicaciones web se encuentran desarrolladas.

Sin embargo, aunque las aplicaciones web han ido incrementándose en número, éstos se están quedando obsoletos con el aumento de tecnologías emergentes, el *Big Data*, el *Internet de las Cosas* (IoT, por sus siglas en inglés) y la adopción a la *Nube* (Cloud, en inglés), éstos traen consigo nuevos requisitos que necesitan ser cubiertos y por lo tanto es imprescindible adaptarse al cambio, esto es que si la tecnología se vuelve rápida entonces los consumidores querrán las cosas todavía más rápidas, esto dando pie a lo que se denomina *Desarrollo y Operación* (DevOps, por sus siglas en inglés).[62] Los microservicios han surgido como medida a estos cambios, sin embargo, ahora las empresas se enfrentan a la problemática de adaptarse a esos cambios y de alguna manera adecuar a microservicios las aplicaciones web ya desarrolladas con el menor costo en tiempo, esfuerzo y dinero.

La arquitectura de microservicios es una técnica de desarrollo donde la aplicación entera es dividida en una colección de servicios débilmente acoplados, cada servicio se encarga de un dominio de negocio y puede ser desarrollado en el lenguaje que mejor se adecúe al objetivo de la tarea sin que eso suponga un problema en la comunicación entre ellos.[51, 27] Esta arquitectura trae diversos beneficios tanto al

1. INTRODUCCIÓN

cliente que solicita las aplicaciones en esta arquitectura como a la empresa que las desarrolla, éstos se detallan en el siguiente capítulo.

1.1 Objetivo

1.1.1 Objetivo principal

Diseñar una metodología que permita la migración de una aplicación web en Java que implementa el *Framework* de *Spring MVC* a Microservicios.

1.1.2 Objetivos específicos

- Conocer la situación de las aplicaciones web en cuanto a las necesidades que se presentan hoy en día.
- Determinar el impacto de los microservicios en la industria.
- Identificar los componentes de los microservicios y las aplicaciones web monolíticas.
- Identificar las tecnologías y Frameworks tanto de aplicaciones web monolíticas como de los microservicios más utilizados en la industria.
- Comparar las tecnologías que se utilizan en ambas arquitecturas.
- Determinar los puntos clave de las estructuras de ambas arquitecturas.
- Definir los pasos para la migración de la aplicación web a microservicios.

1.2 Problema u Oportunidad

Dado que en la mayoría de las organizaciones emplean aplicaciones web comunes o de arquitectura monolítica, a medida que el *software* va evolucionando y desarrollándose, como el ir añadiendo nuevas características o módulos, llega el momento en el que no es posible seguir escalando la aplicación y los desarrollos se vuelven lentos y difíciles ya que todos los equipos de desarrollo concentran en un solo repositorio de código para posteriormente realizar el despliegue una vez haya pasado las pruebas necesarias, todo ese trayecto entre el desarrollo y el despliegue se vuelve más tardado a medida que el software crece y como resultado hay pérdidas de tiempo y costos en el mantenimiento de este tipo de aplicaciones.

Por lo anterior, resulta necesario la adopción de una tecnología más flexible que soporte las crecientes demandas de la aplicación y que proporcione un crecimiento

1.3 Hipótesis

beneficioso a la empresa como la arquitectura de microservicios, sin embargo comenzar una aplicación desde cero es realmente costoso, además en la actualidad no existe una metodología para realizar una migración de aplicaciones web monolíticas en Java a microservicios.

De modo que la oportunidad que ofrece esta tesis es la de brindar una metodología que permita la migración de las aplicaciones web desarrolladas en Java a microservicios a fin de evitar la escritura de código desde cero a través de la implementación de la mayor parte de código de la aplicación web a migrar y de ese modo ahorrar tiempo y costos de desarrollo, obteniendo todos los beneficios que la arquitectura de microservicios ofrece como modularidad, flexibilidad y la implementación de DevOps.

1.3 Hipótesis

El presente trabajo planea resolver la siguiente hipótesis: "¿Es posible desarrollar una metodología lo suficientemente explícita que logre una migración exitosa de aplicaciones web monolíticas en Java a microservicios, es decir, que mantenga sus funcionalidades completas de cualquier aplicación web desarrollada en Java con el Framework de Spring MVC?"

1.4 Contribución y relevancia

A pesar de que la arquitectura de microservicios no es tecnología nueva, recientemente es que ha tomado mayor relevancia, sin embargo, en la actualidad no se cuentan con metodologías bien detalladas que resuelvan el problema de la mayoría de las organizaciones, lo que implica que éstas deben emprender el camino de desarrollar sus aplicaciones desde el inicio con la arquitectura de microservicios. Cabe mencionar que las técnicas y metodologías existentes sólo dan una visión general de las consideraciones a tomar en cuenta para realizar una migración pero no especifican el cómo hacerlo, tecnologías licenciadas en cambio sugieren el asesoramiento de expertos que incrementan los costos.

Esta metodología se enfocará en resolver ese problema para la mayor población posible de tecnologías de software libre y que no desean incrementar sus costos con la contratación de expertos en migraciones por lo que se emplearán las tecnologías que más se usan en el desarrollo de aplicaciones actualmente en ese rubro y de este

1. INTRODUCCIÓN

modo otorgarle a las organizaciones una manera de transformar sus aplicaciones monolíticas ya existentes a la arquitectura de microservicios sin tener que desarrollarlos desde cero y así aprovechar los beneficios que esta arquitectura.

Por lo que esta tesis es relevante para las organizaciones o empresas que deseen realizar cambios en sus aplicaciones web monolíticas para adaptarse mejor a las nuevas tecnologías y técnicas mencionadas anteriormente y que tienen aplicaciones web desarrolladas en el lenguaje de programación más utilizado, Java, y también emplean el Framework más usado en ese lenguaje, Spring MVC.[8, 59]

CAPÍTULO 2

Marco teórico

En este capítulo se presenta la teoría sobre las diferentes arquitecturas involucradas para el desarrollo de la metodología propuesta en el objetivo de la investigación. Los conceptos presentados abarcan desde el inicio de las aplicaciones web y los cambios a través del tiempo hasta lo que tenemos hoy en día sobre microservicios. Se presentarán además las ventajas y desventajas de la arquitectura monolítica y de microservicios, así como las comparaciones entre ambas. También se presentará cómo se encuentra estructurada una aplicación web en Java y cómo su estructura cambió la concepción de las aplicaciones web actualmente.

2.1 Aplicación web

2.1.1 Definición

Es una aplicación de cliente-servidor que funciona a través del *Protocolo de Control de Transmisión/Protocolo de Internet* (TCP/IP, por sus siglas en inglés), donde la aplicación web se despliega en un servidor y el navegador web actúa como el programa cliente que mantiene una conexión con el servidor por medio de peticiones *Protocolo de Transferencia de Hipertexto* (HTTP, por sus siglas en inglés)(GET/POST/PUT/HEAD, etc.), de acuerdo a los parámetros de la petición, la aplicación web responde con contenido dinámico, rastreadores, *cookies*, sesiones, y otras consideraciones de seguridad.[63]

2. MARCO TEÓRICO

2.1.2 Aplicación web con arquitectura monolítica

Podemos encontrar que la mayoría de las aplicaciones web constan de una arquitectura monolítica. Coloquialmente hablando el término monolítico se deriva de la palabra monolito que de acuerdo a la RAE[32] significa: "Monumento de piedra de una sola pieza".

Se dice que una aplicación posee una arquitectura monolítica cuando el software completo se encuentra empaquetado en un solo contenedor ejecutable o binario desplegable, además posee un código base simple dividido en múltiples módulos y la aplicación funciona como una sola unidad.[31, 64, 57]

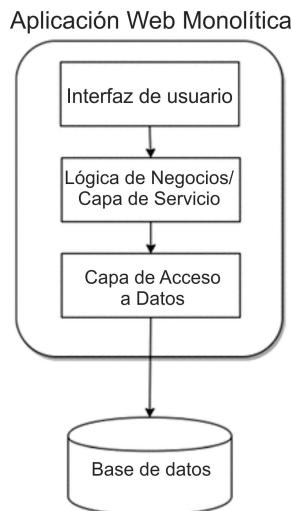


Figura 2.1: Aplicación web de arquitectura monolítica. Extraída y traducida de [46]

2.1.3 Funcionamiento

De acuerdo al sitio web de Adobe[1], una aplicación web que posee acceso a una base de datos como mínimo funciona de la siguiente manera:

1. El navegador web realiza una petición al servidor web
2. El servidor web localiza la página y la envía al servidor de aplicaciones web
3. El servidor de aplicaciones web busca instrucciones en la página, como consultas a bases de datos
4. El servidor de aplicaciones web envía la consulta al controlador de base de datos
5. El controlador de base de datos ejecuta la consulta en la base de datos

2.1 Aplicación web

6. El resultado de la consulta se devuelve al controlador como un conjunto de registros
7. El controlador envía el conjunto de registros de la operación al servidor de aplicaciones web
8. El servidor de aplicaciones web inserta los resultados en una página y la pasa al servidor web
9. El servidor web devuelve como respuesta a la petición la página *HTML* al navegador web solicitante

Podemos observar en la figura 2.2 lo antes mencionado.

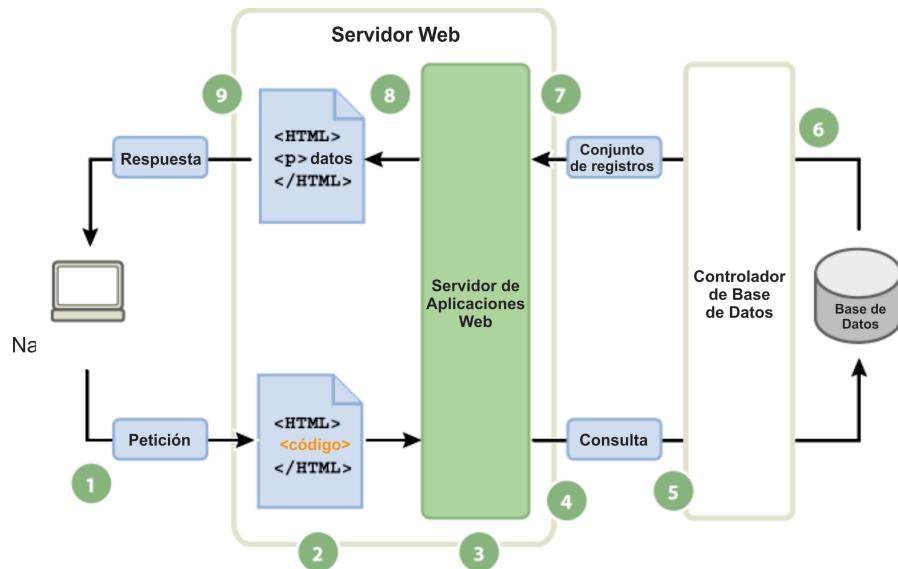


Figura 2.2: Componentes y funcionamiento de las aplicaciones web. Extraída y traducida de [1]

2.1.4 Evolución de las aplicaciones web

Las aplicaciones web han ido evolucionando con el paso del tiempo adecuándose a las nuevas necesidades. Al principio todo era bastante simple, sin embargo, con la invención de las redes todo se volvió complicado. Ahora, debemos desarrollar sistemas o aplicaciones que dependen de otros en ejecución en otras computadoras llamado cómputo distribuido.[36] Por ejemplo, en cómputo distribuido, una aplicación web es dividida físicamente en partes, *niveles*(tiers, en inglés), que se ejecutan simultáneamente en diferentes computadoras, y que se llama arquitectura de niveles. En

2. MARCO TEÓRICO

cada nivel proporciona un conjunto de servicios que pueden ser consumidos por la conexión o el nivel de cliente. Los niveles pueden además dividirse de manera lógica en *capas*(layers, en inglés), como la organización de código en módulos, para proveer funciones a nivel granular. Actualmente la mayoría de las aplicaciones web poseen arquitectura de tres niveles que coincide con el número de capas de la aplicación: [48, 61]

- **Nivel de presentación:** Se centra en la experiencia del usuario, las interfaces gráficas de usuario y/o la comunicación entre el usuario final y la aplicación
- **Nivel lógico o de aplicación:** Controla la lógica de negocio de la aplicación, también puede realizar llamadas a servicios y encargarse de las actividades de creación, lectura, actualización y eliminación (CRUD, por sus siglas en inglés) en el nivel de acceso a datos.
- **Nivel de acceso a datos:** Su función radica en el almacenamiento de los datos, es decir, el almacenamiento y devolución de los datos almacenados.

Podemos ver lo antes mencionado en la figura 2.3

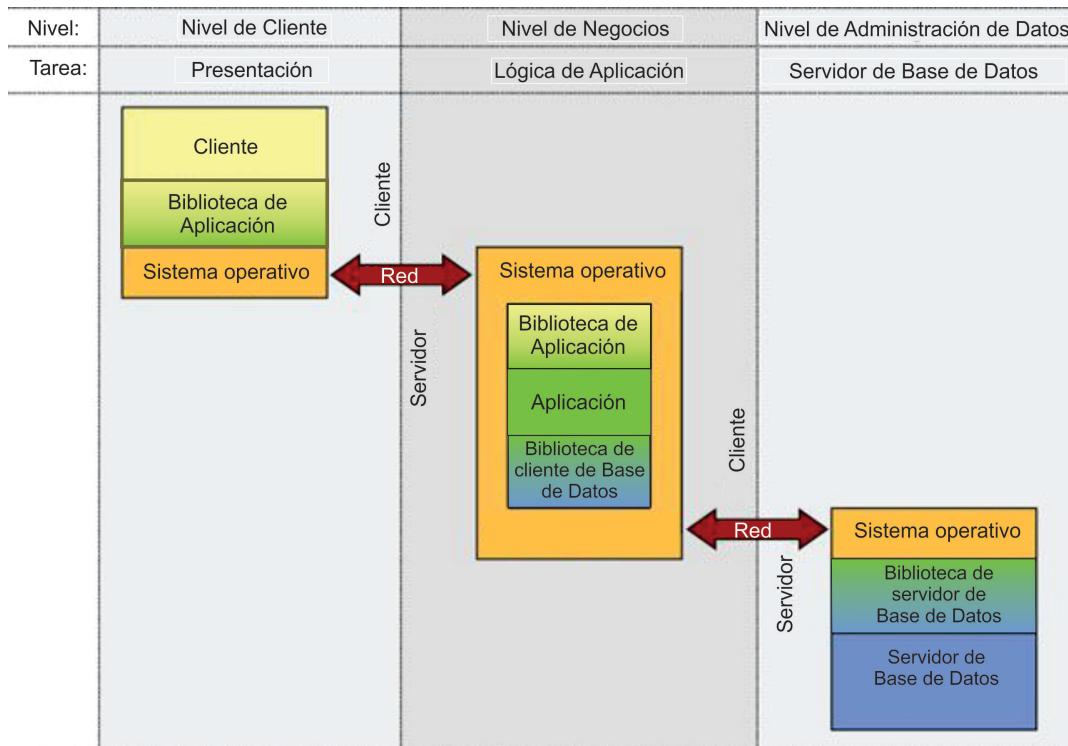


Figura 2.3: Arquitectura de 3-niveles. Extraída y traducida de [61]

2.1.4.1 Arquitectura de un sólo nivel

Este tipo de arquitectura se remonta a los días de los sistemas *mainframes* que se conectaban por terminales o consolas.[48] Una aplicación con este nivel de arquitectura es tan simple que no requiere acceso a la red mientras se ejecuta. La mayoría de las aplicaciones de escritorio, como procesadores de texto o compiladores, entran en ésta categoría. Las ventajas que esto supone varían: por una parte está la simplicidad, una aplicación de un solo nivel no necesita manipular ningún protocolo de internet, entonces no requiere de sincronización de los datos y por consecuencia no se tiene que lidiar con manejo de excepciones para cuando falle la red, o que un servidor ejecuta una versión diferente de un protocolo o programa; por otra parte puede tener un mayor rendimiento ya que los usuarios no requieren acceder a la red para realizar una petición a un servidor y que éste les responda.[36]

Como puede verse en la figura 2.4, la aplicación entera que comprende el conjunto de las capas lógicas como las interfaces de usuario, la lógica de negocios y los datos se hospedaban en el mismo equipo físico.

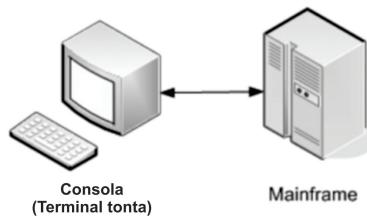


Figura 2.4: Arquitectura de 1-nivel. Extraída y traducida de [48]

2.1.4.2 Arquitectura de dos niveles

Con el incremento de las computadoras personales a principios de los años 80, debido a que se volvieron menos caras, poseían mejor capacidad de procesamiento comparado con las antiguas consolas o terminales además de que comenzaron a tener interfaces gráficas de usuario (GUIs, por sus siglas en inglés) se permitió al usuario introducir datos e interactuar con el servidor mainframe con una mejor visualización. Fue entonces que los mainframes contenían la capa de acceso a datos y la lógica de negocios y el equipo cliente sólo la interfaz gráfica de usuario, a este tipo de cliente-servidor se le llamó cliente ligero o liviano; más tarde se formó otro tipo de arquitectura de dos niveles, en donde el equipo cliente ahora poseía la lógica de

2. MARCO TEÓRICO

negocios y el mainframe el acceso a los datos, a este se llamó cliente gordo o pesado como se puede observar en la figura 2.5.[48]

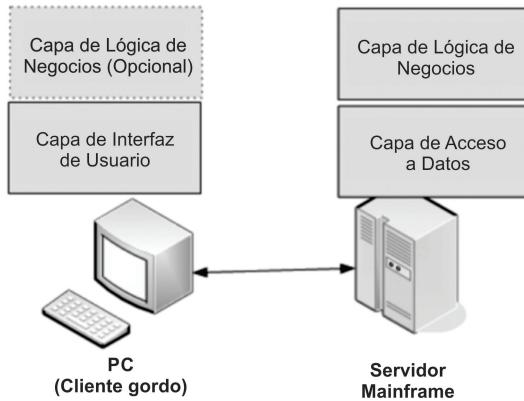


Figura 2.5: Arquitectura de dos niveles de tipo cliente pesado. Extraída y traducida de [48]

2.1.4.3 Arquitectura de tres niveles

La arquitectura de tres niveles se originó tiempo después de la de dos niveles por el crecimiento acelerado de la internet, reducciones en los costos de hardware e incremento en la capacidad de cómputo. En este tipo de arquitectura se caracteriza por poseer un cliente ligero como un navegador web para desplegar la interfaz gráfica de usuario que provenía del servidor, el servidor hospeda la capa de presentación, la lógica de negocios y la lógica de acceso a datos, y además, cuenta con un servidor de base de datos, como se observa en la figura 2.6.[48]

Con este tipo de arquitectura, se permite separar la lógica de negocios del acceso a datos, y por consecuencia proveer datos altamente optimizados, permitir las copias de seguridad de la información almacenada, manejo de concurrencia y redundancia, procedimientos de balanceadores de carga específica a las necesidades de los datos. Todo esto permite una mejor escalabilidad de la aplicación de manera que cada nivel se puede escalar de manera horizontal, así por ejemplo, colocar los datos en procesos dedicados o serie de equipos huéspedes.[36]

2.1 Aplicación web

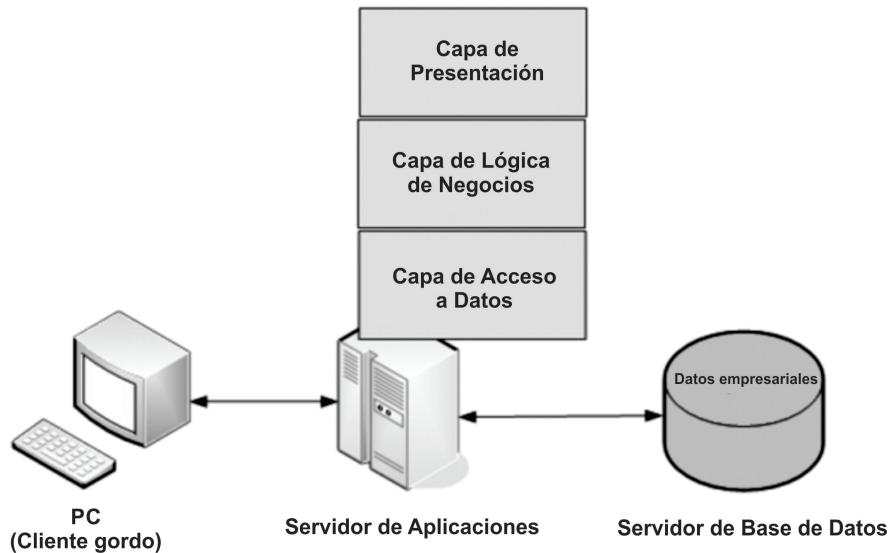


Figura 2.6: Arquitectura de tres niveles. Extraída y traducida de [48]

2.1.4.4 Arquitectura de n niveles

En la actualidad las empresas mantienen sus aplicaciones *web-enabled*, de modo que permiten un número ilimitado de programas ejecutarse simultáneamente, enviar información de uno a otro, a través de diferentes protocolos de comunicación e interactuar concurrentemente. Esto permite potenciar la aplicación y proveer muchos más servicios diferentes a muchos más clientes.[36] En este tipo de arquitectura las capas pueden comunicarse todas entre sí o simplemente una capa sólo puede comunicarse con la inmediata anterior lo que limita la dependencia entre ellas aunque con la posibilidad de generar tráfico innecesario en la red si una capa simplemente pasa una petición hacia la siguiente. [55]

En la figura 2.7, puede observarse la relación entre los niveles y la interacción entre ellos, en este caso, la aplicación de servidor ya no se encarga de la capa de presentación, sino que ahora está cargado a un servidor web que genera el contenido a presentar al cliente. Este contenido se transfiere al navegador del cliente y se encarga de construir la interfaz de usuario. La aplicación de servidor hospeda remotamente los componentes de negocio y son accedidos por el nivel de presentación a través de la red por medio de protocolos nativos.

2. MARCO TEÓRICO

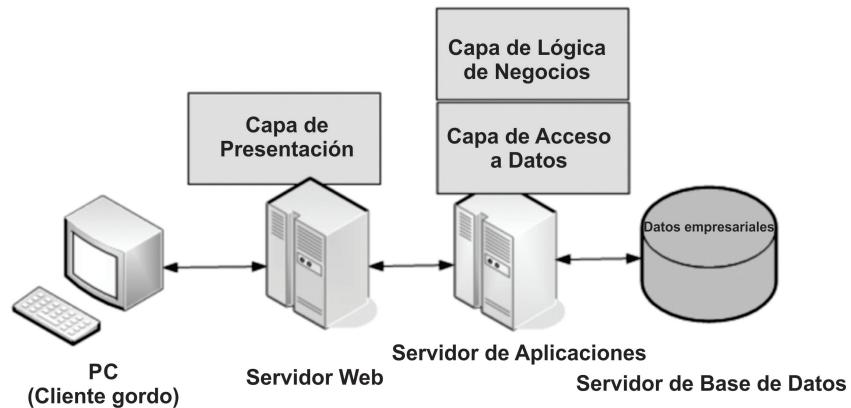


Figura 2.7: Aplicación de n niveles. Extraída y traducida de [48]

2.1.4.5 Arquitectura de Java Enterprise Edition

Aunque la distribución de tareas separadas por niveles permite asignar a expertos de esas áreas, por ejemplo, un experto en bases de datos que se encargue del nivel de datos creando procedimientos almacenados, funciones, etc., un experto desarrollador *front-end* del nivel de presentación, el desarrollo de aplicaciones de n niveles distribuidas es realmente complejo y demanda bastante esfuerzo.[65]

De cualquier manera, mantener conectados los servicios de los n niveles requiere de servidores *middleware*¹ que lleve a cabo las transacciones, seguridad y la agrupación de datos. Estas API's de los servidores middleware son necesarias para acceder a esos servicios, lo que conlleva a mezclar código propietario con código de terceros limitando el mantenimiento de este además de portabilidad y pérdida de tiempo de desarrollo. Debido a todas estas complicaciones *Sun Microsystems*, liberó la plataforma de *Java EE 2* en 1999 para el desarrollo de aplicaciones empresariales multiniveles con el concepto de desarrollar una vez y, desplegar y ejecutar donde sea.[48]

La plataforma de Java EE provee los servicios esenciales del sistema por medio de una arquitectura basada en contenedores.[48] La lógica de la aplicación se divide en componentes de acuerdo a su función, y estos componentes se instalan en diferentes máquinas dependiendo del nivel en el ambiente multinivel de Java EE al que pertenezca, como se puede observar en la figura 2.8, el nivel de cliente se ejecuta en la máquina del cliente, los componentes del nivel web y la lógica de negocios se ejecutan o ambos en un servidor Java EE o en diferentes servidores, y finalmente

¹Middleware: Software que se encarga de facilitar la comunicación entre servicios

2.2 Microservicios

el nivel de Sistema de Información Empresarial se ejecuta en el servidor de base de datos. [18]

Aunque las aplicaciones Java EE consisten de 3 o 4 niveles, las aplicaciones multiniveles Java EE generalmente son consideradas de 3 niveles porque se distribuyen en 3 ubicaciones diferentes, la maquinas clientes, el servidor Java EE y el servidor de base de datos o sistemas legados.

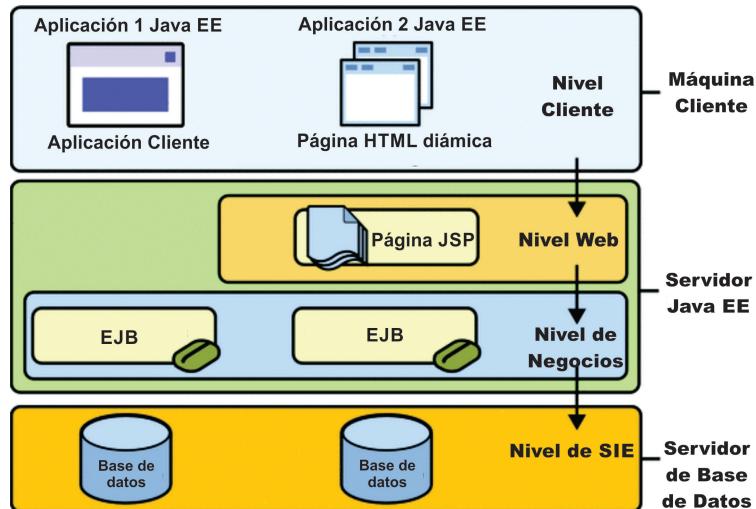


Figura 2.8: Arquitectura de Java EE. Extraída y traducida de [18]

2.2 Microservicios

2.2.1 ¿Qué son los microservicios?

Existen autores que definen a los microservicios como un método de desarrollo de servicios modulares, tal es el caso de Scott[62]:

A microservices architecture is a method of developing applications as a network of independently deployable, modular services in which each service runs a unique process and communicates through a well-defined, lightweight mechanism to serve a business goal. [Una arquitectura de microservicios es un método de desarrollo de aplicaciones como una red desplegable de manera independiente, servicios modulares en el que cada servicio se ejecuta como único proceso y se comunica a través de un mecanismo ligero y bien definido para servir a un objetivo de negocio.]

2. MARCO TEÓRICO

Otros como IBM[38] definen a los microservicios como un estilo de arquitectura para software complejo:

Microservices is an architecture style, in which large complex software applications are composed of one or more services. Microservice can be deployed independently of one another and are loosely coupled. Each of these microservices focuses on completing one task only and does that one task really well. In all cases, that one task represents a small business capability.[Microservicios es un estilo de arquitectura, en el que enormes y complejas aplicaciones de software están compuestas por uno o mas servicios. Los Microservicios pueden ser desplegados independientemente de entre uno y otro además de estar débilmente acoplados. Cada uno de estos microservicios se enfoca en completar una tarea y de realizarla completamente bien. En todos los casos, esa tarea representa una capacidad pequeña de negocio.]

Se entiende entonces que un microservicio es un tipo de enfoque que aplica la filosofía UNIX donde la aplicación se encuentra desarrollada como un conjunto de pequeños servicios independientes, y cada servicio se encarga de realizar una tarea específica (Búsquedas, Inicio de sesión, Verificación de correo, etc). Además cada servicio puede estar desarrollado en cualquier lenguaje de programación tiene la habilidad de comunicarse con los demás gracias a que aplican protocolos de invocación no propietarios como *REST* y Protocolo de Transferencia de Hipertexto.

2.2.2 Arquitectura y funcionamiento de los microservicios

La arquitectura de las aplicaciones importan cuando se ven afectados los requerimientos de calidad de servicio, comúnmente llamados requerimientos no funcionales. El término microservicio puede ser un problema a la hora de decidir qué lo define como tal, sin embargo poco tiene que ver con el tamaño, más bien se refiere a si el microservicio puede ser desarrollado por un equipo de desarrolladores pequeño con poca o nada interacción con otros equipos y en poco tiempo. [60] La arquitectura y funcionamiento general de los microservicios se muestran en la figura 2.9, adicionalmente puede o no tener una *Red de Distribución de Contenidos* (CDN, por sus siglas en inglés):

1. Un cliente realiza una petición a un servicio, por ejemplo Microservicio X, a través de un mecanismo como RestAPI
2. El API Gateway intercepta la petición y la pasa al Proveedor de identidad para validarla.

2.2 Microservicios

3. El Proveedor de identidad regresa la petición al API Gateway
4. El API Gateway identifica el microservicio hacia el cuál va la petición y la pasa hacia él por medio de RestAPI
5. El microservicio, en este caso Microservicio X, realiza peticiones a su base de datos y en caso de ser necesario pasa la petición a otros microservicios para realizar otras acciones
6. El microservicio regresa los resultados de la petición al cliente a través de RestAPI

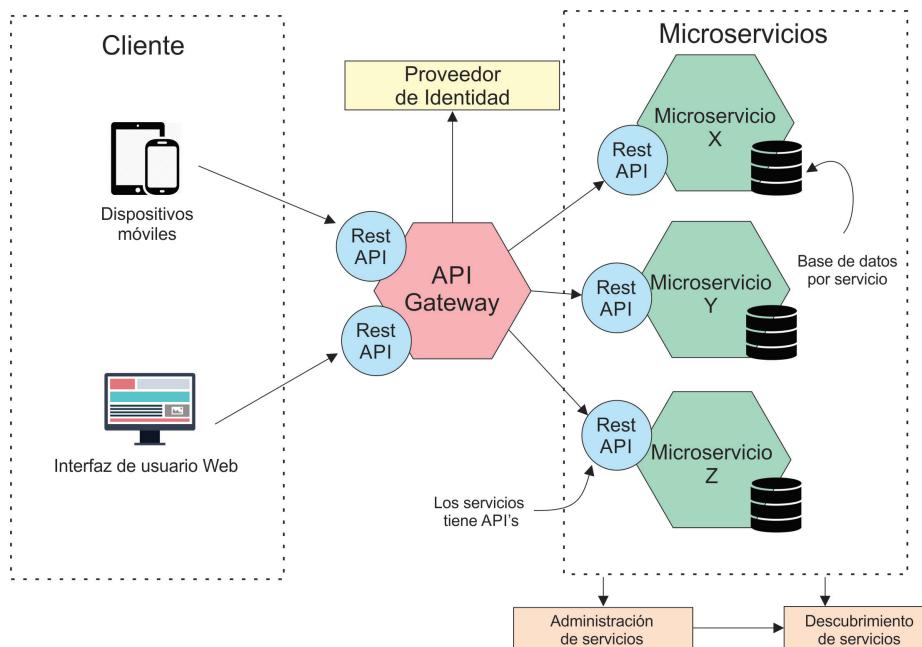


Figura 2.9: Arquitectura de los Microservicios. Basado en [47, 60]

2.2.3 Breve historia de los microservicios

La primera vez que se habló de un término parecido a microservicios fue cuando se dio origen a la llamada Arquitectura Orientada a Servicios (SOA, por sus siglas en inglés) definida por primera vez en los años 1980. Sin embargo ésta se hizo famosa hasta mediados de los años 90, cuando la compañía llamada *Gartner Group* reconoció que este estilo de arquitectura se estaba volviendo una tendencia por lo que decidieron adoptarlo y fue cuando se popularizó por todo el mundo.[52, 39] SOA

2. MARCO TEÓRICO

es un patrón de diseño de arquitectura de software en el que los componentes de las aplicaciones proveen servicios por medio de un protocolo de comunicaciones en la red a otros componentes de aplicaciones. La comunicación puede implicar el paso de datos simples o dos o más servicios de coordinación. Estos distintos servicios como *RESTful web services* llevan a cabo funciones pequeñas como la validación de pagos, validar una orden, activación de una cuenta, etc. [56]

A pesar de todas los beneficios que SOA prometía, su implementación no era tan sencilla, ya que todas las aplicaciones basadas en servicios requieren de otros servicios para administrarlos. El concepto de Middleware se basa en ésta idea, coordina grupos de servicios para asegurarse de que los datos fluyeran bien entre ellos y de que los servicios se encontraran disponibles cuando se les sea requeridos.[62]

Además del Middleware otra manera en la que se podía coordinar los servicios era con los llamados Bus de Servicio Empresarial (ESB, por sus siglas en inglés), que forma parte de la categoría de integración de un Middleware y básicamente es una capa de comunicación con la habilidad de mantener conexiones de punto a punto entre los dos roles principales que SOA tiene, la capa de proveedores y la capa de consumidores del servicio, como se observa en la figura 2.10. La capa de proveedores la conforman todos los servicios que se encuentra dentro de SOA y la capa de consumidores la conforman los usuarios, que pueden ser humanos, otros componentes de la aplicación o aplicaciones de terceros.[49, 56]

2.2.4 Beneficios y características

Los microservicios presentan ciertos beneficios y características que las aplicaciones monolíticas carecen y por las que son mejores que éstas.

- Cada microservicio que conforma la aplicación presenta total independencia entre los demás lo que permite tener una alta disponibilidad y aislar de manera más eficiente los fallos, por lo que de ocurrir uno en cualquiera de ellos éste puede ser desconectado y resuelto sin que ninguna otra parte de la aplicación se vea afectada, esta independencia facilita el desarrollo y entregas continuas de aplicaciones complejas.
- Cada microservicio funciona como una pequeña aplicación, ya que el almacenamiento puede ser exclusivamente para sí mismo en algunos casos y además cada uno de ellos puede desarrollarse en diferentes lenguajes de programación sin que el funcionamiento de la aplicación completa se vea afectado.
- La aplicación de microservicios es altamente escalable, es decir, cada servicio puede escalarse independientemente si así se requiere.
- Los despliegues de los microservicios son claramente más rápidos, dado que la

2.2 Microservicios

aplicación entera es un conjunto de microservicios, y cada microservicio es lo más pequeño posible en cuanto a su objetivo de negocio.

- Los equipos de desarrollo pueden organizarse en subgrupos donde cada uno de ellos se enfoque sólo en un módulo de la aplicación con el lenguaje de programación en lo que son expertos.
- Habilita la adopción a DevOps
- Permiten adopciones más fáciles de nuevas tecnologías

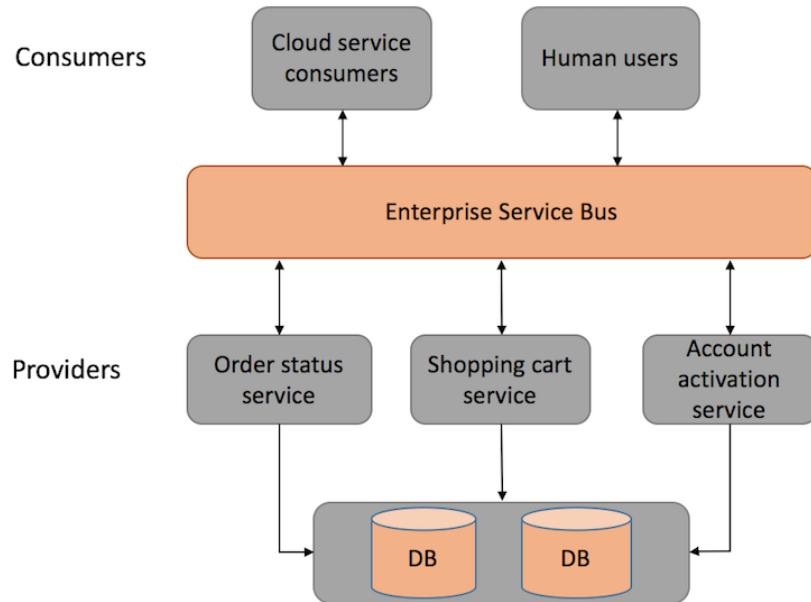


Figura 2.10: Arquitectura SOA. Extraído de [56]

Sin embargo, aunque la arquitectura de microservicios pareciera la *panacea* de los desarrollos también presentan ciertas complejidades con las que se deben lidiar, como la distribución, para el que se necesita nuevos componentes de soporte. Además la descomposición del sistema en pequeños servicios, monitorizarlos y administrarlos, son algunos de los factores importantes que se deben de tener en cuenta, por lo que la migración a los microservicios u algún otro tipo de servicio basado en la nube llega a ser un problema multidimensional.[33]

2. MARCO TEÓRICO

2.2.5 Diferencias entre microservicios y SOA

El término SOA que fue definido hace algunas décadas puede confundirse con lo que son los microservicios al compartir similitudes entre ellos, sin embargo una no tiene nada que ver con la otra. Las principales diferencias entre microservicios y SOA son: [51, 56]

- En SOA para la comunicación entre aplicaciones o servicios se emplea tecnologías pesadas de transporte como SOAP y estándares WS, además utilizan un ESB y en los Microservicios se utiliza un sistema simple de mensajes como REST o gRPC.
- En cuanto al desarrollo de los servicios, a pesar de que en ambos se puede realizar por diversos equipos de software, en SOA es requisito indispensable que todos sepan cuál será el mecanismo común de comunicación.
- En SOA todos los servicios comparten el mismo modelo de almacenamiento de datos y bases de datos compartidos. En los microservicios se tienen bases de datos independientes por cada servicio y cada servicio tiene su propio modelo de datos.
- Los microservicios son significativamente más pequeños que lo que SOA pudiera ser y sobre todo servicios desplegables independientemente. Por otro lado SOA puede ser tanto una aplicación monolítica como un conjunto de múltiples aplicaciones complejas monolíticas.

2.2.6 Diferencias entre microservicios y arquitectura monolítica

Las principales diferencias entre las arquitecturas de microservicios y arquitectura monolítica son:[51]

- En los microservicios el tiempo de respuesta es más rápido mientras que en la monolítica es más tardado.
- En microservicios todos los servicios están desacoplados y funcionan de forma independiente, así cuando un servicio falla la aplicación entera se mantiene, mientras que en la monolítica todos los servicios están acoplados fuertemente entre sí, si algo falla todo debe detenerse.
- En microservicios la escalabilidad es posible ya que la arquitectura lo permite, y en la monolítica aunque también es posible resulta más complicado y ocurre un desperdicio de recursos.
- Los microservicios permiten un desarrollo de liberaciones y entregas continuas por lo que están disponibles más rápido a diferencia de la monolítica donde es más difícil de realizar entregas continuas y los despliegues son más lentos.

2.2 Microservicios

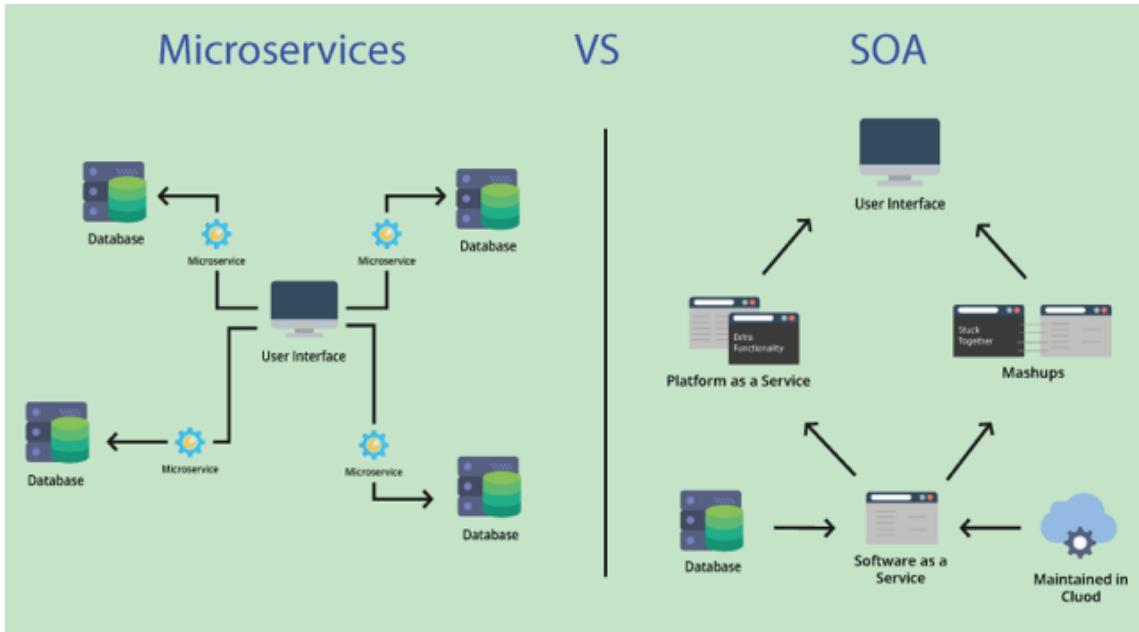


Figura 2.11: Microservicios vs SOA. Extraído de [51]

- Los microservicios están enfocados en productos mientras que en las monolíticas se enfocan en el proyecto entero.
- Tecnologías diferentes pueden usarse en cada servicio de la arquitectura de microservicios ya que ninguno depende directamente de otro, por lo que es más fácil actualizarse a nuevas tecnologías, mientras que la arquitectura monolítica todos dependen entre sí y no pueden usarse diferentes tecnologías tan fácilmente.

La figura 2.12 muestra de manera más clara los puntos mencionados con anterioridad.

2.2.7 Ventajas en relación a la arquitectura monolítica

Las ventajas más destacadas en la implementación de los microservicios en relación a la arquitectura monolítica destacan: [51]

- Actualizaciones menos riesgosas, facilita además la realización de desarrollo y entregas continuas.
- Mejor escalabilidad, lo que permite proporcionar recursos donde se necesite.

2. MARCO TEÓRICO

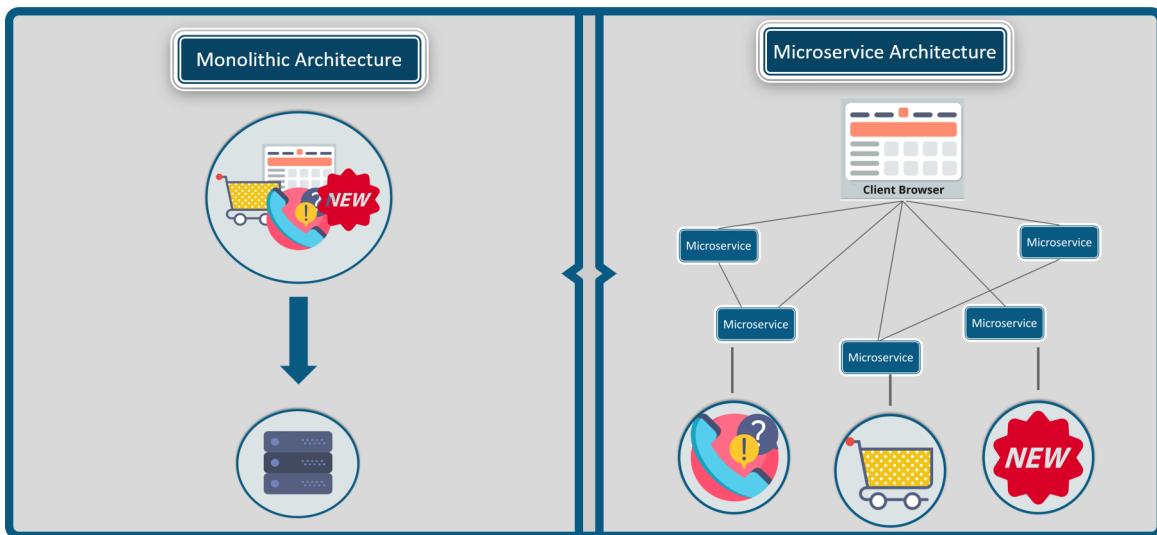


Figura 2.12: Arquitectura monolítica y arquitectura de microservicios. Extraído de [47]

- Cada microservicio se puede desarrollar en el lenguaje de programación que mejor se aadecue a la tarea que desempeñará, por lo que no hay limitaciones en cuanto al empleo de diferentes Frameworks en cada microservicio.
- Los desarrolladores pueden dividirse mejor el trabajo de manera que se enfocuen en un sólo objetivo de negocio
- Mejor mantenibilidad.
- Reducción en las fallas y caídas de la aplicación, ya que en caso de presentarse alguna sólo se afecta una parte de la aplicación y no la aplicación entera, de modo que aunque haya una caída no afectará a otros módulos de la aplicación.
- La comunicación puede llevarse a cabo sin esfuerzo ya que son *sin estado*, así las peticiones y respuestas son independientes.
- Permite la fácil adopción de que elimina la ineficiencia de los despliegues resultando en menos tiempo de puesta en producción.
- Hay una mejora en la calidad del código fuente gracias a su modularidad.

2.2.8 Desventajas respecto a la arquitectura monolítica

- Como la arquitectura de microservicios se enfoca en un sistema distribuido y servicios independientes las peticiones deben ser mantenidas cuidadosamente

2.2 Microservicios

entre los módulos, cosa que en la arquitectura monolítica es fácil de realizar ya que todas las peticiones son hacia el mismo contenedor.

- En la arquitectura de microservicios, en el caso en donde los módulos están desarrollados con su propio lenguaje de programación, éste depende de su propia API y plataforma, al momento de realizar un *volcado de pila* puede ser un proceso muy trabajoso, en cambio en la arquitectura monolítica el volcado de pila es relativamente más fácil ya que se encuentra en una sola pila.

2. MARCO TEÓRICO

CAPÍTULO
3

Patrones y Antipatrones de diseño

En este capítulo se mostrarán algunos conceptos generalizados sobre los patrones y antipatrones de diseño encontrados en el desarrollo de aplicaciones web y en los microservicios, además de proporcionar las respectivas referencias a cada uno de ellos para obtener información más detallada en caso de ser necesario. Se presenta además el patrón de migración de aplicaciones web a microservicios que se empleará en la metodología con tecnologías propuestas para su implementación en el siguiente capítulo.

3.1 Patrones de diseño

3.1.1 ¿Qué es un patrón de diseño?

Un patrón de diseño proporciona la solución a algún problema de diseño que ocurre una y otra vez en nuestro entorno, dicha solución puede aplicarse miles de veces sin que se haga de la misma forma dos veces.[28, 45]

De acuerdo a *Gang of four*[45], un patrón debe poseer cuatro características esenciales:

- El **nombre del patrón** que pueda describir de manera concreta el problema, la solución y consecuencias en tan sólo una o dos palabras, por lo que el nombrar un patrón de diseño puede llegar a ser realmente difícil.

3. PATRONES Y ANTIPATRONES DE DISEÑO

- El **problema** que describe cuando aplicar el patrón. A veces el problema incluye una lista de condiciones que se deben conocer antes de que se pueda aplicar el patrón.
- La **solución** que describe los elementos que conforman el diseño, sus relaciones, responsabilidades y colaboraciones. La solución no debe describir un diseño o implementación concreta ya que un patrón es como una plantilla que puede ser aplicada en diferentes situaciones por lo que debe ser una abstracción de un problema de diseño y de cómo una disposición general de elementos lo resuelven.
- Las **consecuencias** que son los resultados y logros obtenidos de aplicar el patrón en otras ocasiones. Esto es crítico para evaluar alternativas de diseño y para entender el costo beneficio de aplicar el patrón.

Entonces un patrón de diseño es una solución reusable expresado de forma abstracta para resolver un problema de diseño que se presenta de manera recurrente. Y la razón por la que son altamente valorados es porque describe un contexto en el que puede ser aplicado, así si la solución describiera una solución para un contexto particular podría no funcionar bien para todos los otros contextos.

3.1.2 Patrones comúnmente usados en Aplicaciones Web Java Enterprise Edition

En el desarrollo de aplicaciones web se ha estandarizado el patrón de diseño Modelo Vista Controlador para que los desarrollos sean más entendibles y fáciles de mantener. Con el surgimiento de la plataforma de Java Enterprise Edition se logró que los desarrollos de aplicaciones multíniveles sea mucho más fácil de desarrollar y mantener, es decir, además de soportar el Patrón MVC facilitó que cada parte de éste pueda estar en diferentes computadores o niveles de arquitectura. [48]

Aunque los siguientes patrones son los mayormente empleados por la plataforma Java Enterprise Edition que es el escenario a seguir, también se utilizan en otros lenguajes de programación ya que ayudan a optimizar el proceso de desarrollo y mantenimiento de las aplicaciones web. Cabe señalar que no significa que todos estos patrones deban ser empleados en una sola aplicación.

3.1.2.1 Patrón Modelo Vista Controlador

El patrón de diseño Modelo Vista Controlador (Model View Controller, en inglés) fue descrito por primera vez por Trygve Reenskaug en su trabajo *"Applications programming in Smalltalk-80™: How to use Model-View-Controller"* y primero se ideó como una estrategia para separar la lógica de la interfaz de usuario de la lógica de

3.1 Patrones de diseño

negocios. Además sugería el añadir una capa intermedia entre la capa de presentación y la de lógica de negocios que se llamaría Capa Controladora.[48]

Como se puede observar en la figura 3.1, cualquier acción provocada por el usuario es interceptada por el controlador y dependiendo de esa acción, el controlador invoca al modelo para ejecutar las reglas de negocio que modifiquen los datos de la aplicación. El controlador selecciona los componentes de las vistas para presentar los datos modificados de la aplicación al usuario final. En consecuencia, el patrón MVC provee una guía para una separación limpia de responsabilidades de la aplicación.

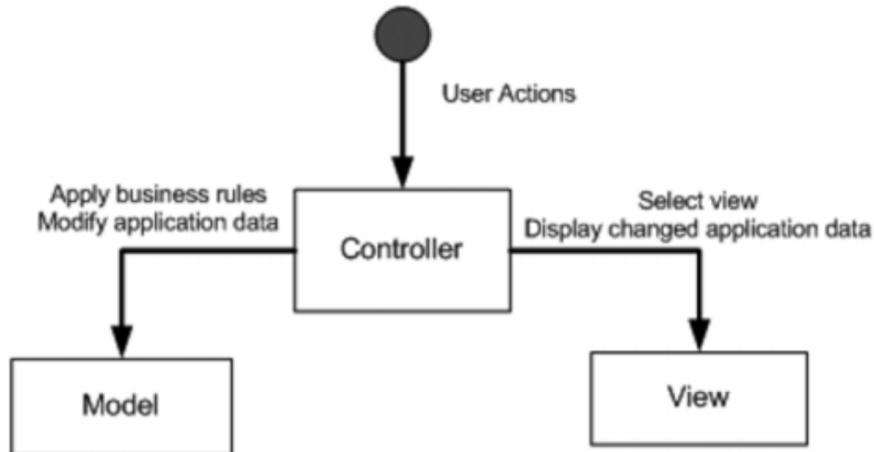


Figura 3.1: Modelo Vista Controlador. Extraído de [48]

3.1.2.2 Patrones de la capa de presentación

View Helper

Comúnmente las aplicaciones web en la capa de presentación se componen por archivos *JavaServer Pages*(JSP, por sus siglas en inglés), éstos archivos contienen código *HyperText Markup Language*(HTML, por sus siglas en inglés) e imágenes que se muestran al usuario, sin embargo, surge un problema cuando se quiere mostrar contenido dinámico almacenado en el modelo y se quiere entonces es evitar código java embebido en los JSP's para no perjudicar la legibilidad de estos y tener una buena separación de la capa de presentación con la lógica de negocios, a fin de lograr

3. PATRONES Y ANTIPATRONES DE DISEÑO

una buena división de tareas para los desarrolladores *back-end* y *front-end* y lograr crear componentes reutilizables de los modelos de datos que puedan ser usados entre las vistas.[58, 24, 48]

La solución a esos problemas radica en utilizar un patrón de diseño como el Patrón View Helper. La figura 3.2 muestra la estructura del patrón de diseño View helper de una manera más clara.

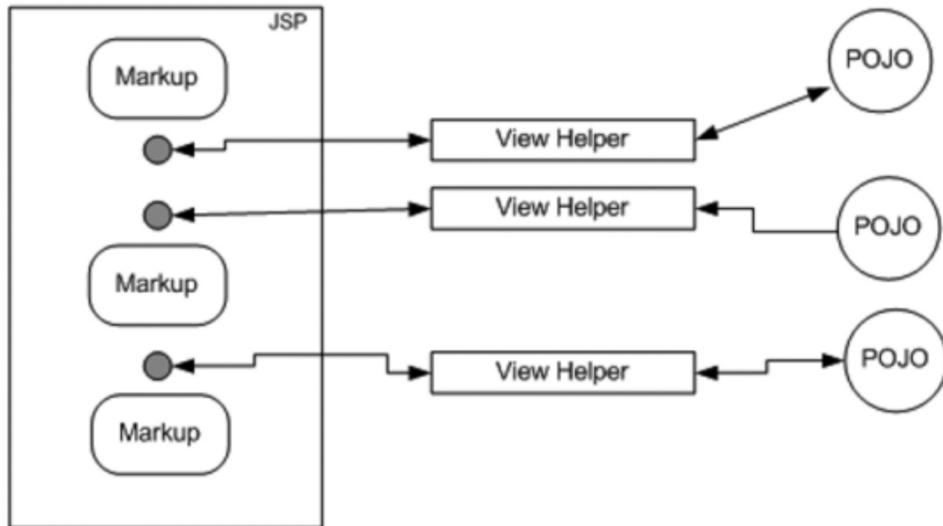


Figura 3.2: Patrón View Helper. Extraído de [48]

Los View Helpers separan perfectamente las vistas, JSP's, del procesado de los datos de la lógica de negocios, obteniendo las siguientes ventajas:[58]

- Los componentes en la capa de presentación se mantienen estandarizados
- El código java se mantiene abstracto para los diseñadores web, dándoles un conjunto de *helpers* para acceder al modelo
- Dada la estructura de los helpers, es fácil implementar contenido sin valor pero que represente información real si no existe aún un modelo definido para que los diseñadores web puedan realizar su trabajo satisfactoriamente.
- Los helpers funcionan como intermediarios entre la lógica de negocios y la vista logrando una separación clara y simple.

3.1 Patrones de diseño

Composite View

El desarrollo y mantenimiento de los componentes de las vistas no es tarea fácil, las aplicaciones muy sofisticadas y complejas requieren de la implementación de plantillas adaptables a datos dinámicos, otro punto importante es que la construcción de las vistas sea por conjuntos de subvistas más pequeñas de modo que se fomente la reutilización y sea más flexible ante los cambios, es decir, sea más fácil el mantenimiento. En las aplicaciones, cada vista está compuesta de 3 elementos principales: componentes, contenedores y la *disposición de componentes desplegados* (*layout*, en inglés) en un contenedor, sin embargo, en la práctica, se tiende a omitir la identificación de estos elementos, provocando que se vuelva compleja y de difícil mantenimiento.[4, 48]

La utilización del patrón de diseño Composite View satisface las necesidades antes mencionadas, de modo que ayuda a agrupar e insertar dinámicamente varias subvistas para construir vistas complejas con el layout adecuado. Este patrón es una combinación de dos patrones de diseño propuestos por de Gang of four: Composite y Strategy. [48, 42]

La figura 3.3 muestra un ejemplo de layout principal y la división de las subvistas que pueden conformar la página, de manera que cada sección: Encabezado, menú, submenú, cuerpo y pie de página son subvistas.

Front Controller

La sobrecarga de código en los JSP's, tanto de código Java embebido ya que mantiene el controlador en el JSP, como HTML son un grave problema para el mantenimiento y resolución de errores de las aplicaciones web. Además de los redireccinamientos de las páginas, realizando a través de los JSP's acciones de validaciones antes de mostrar su contenido, lo que ocasiona duplicación de código y una mala administración de las acciones que realiza la petición.

Este patrón de diseño soluciona el problema de la interceptación de peticiones, facilitando el redireccionamiento de páginas en el desarrollo, ya que recibe todas las peticiones que realice el cliente por un sólo *gestor de peticiones* que se encarga de despachar esas peticiones al control indicado. El Front Controller puede jugar múltiples roles, como delegar la petición a un controlador o a un helper.[21, 25]

En la figura 3.4 podemos observar la estructura de una aplicación que utiliza el Patrón Front Controller.

Los beneficios obtenidos con este patrón de diseño incluyen la obtención de un control centralizado de manera que existe un punto de entrada consolidado y con-

3. PATRONES Y ANTIPATRONES DE DISEÑO

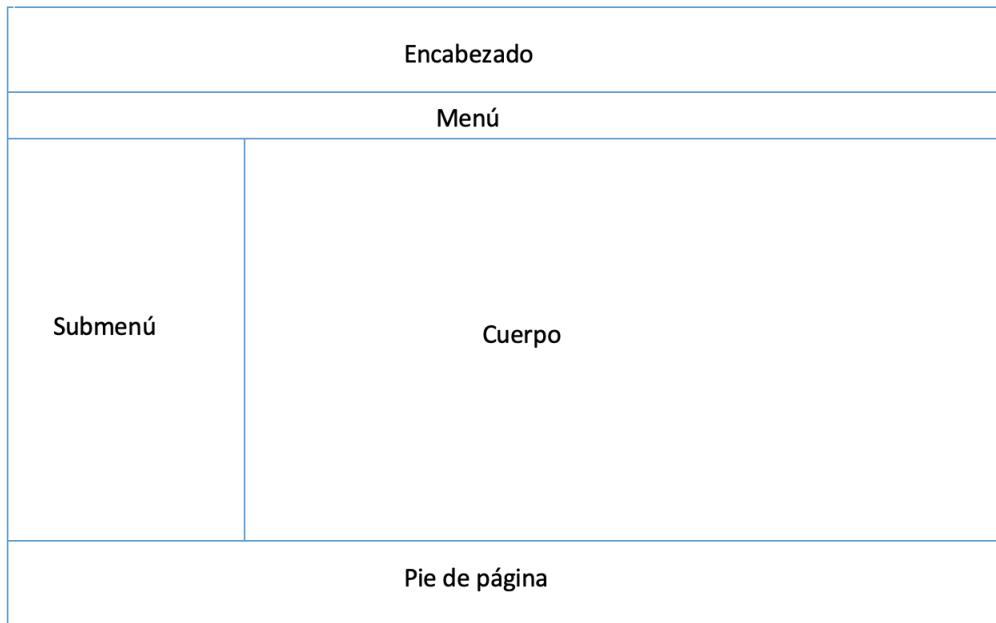


Figura 3.3: Patrón Composite View.

trolar la afluencia de peticiones en la aplicación, lo que hace más fácil manipular la aplicación.[48]

Application Controller

El patrón de diseño Front Controller soluciona el problema de la interceptación de peticiones del cliente, sin embargo, no trata nada de la lógica de negocios por lo que se mantiene controlador basado en JSP, es decir, una mala práctica que puede derivar en el uso del antipatrón conocido como Magic Servlet. El patrón Application Controller mejora los beneficios del Front Controller ya que realiza las invocaciones a la lógica de negocios, la identificación y redireccionado a otras vistas quedan separadas para lograr una mejor legibilidad y mantenimiento de estos.[48, 41]

El funcionamiento del patrón involucra dos componentes esenciales, que están conectados al Front Controller:

- *Action Handler:* Que delega las peticiones al controlador adecuado, figura 3.5
- *View Handler:* Que localiza la vista, une el modelo devuelto por el controlador y prepara la respuesta al cliente, figura 3.6.

3.1 Patrones de diseño

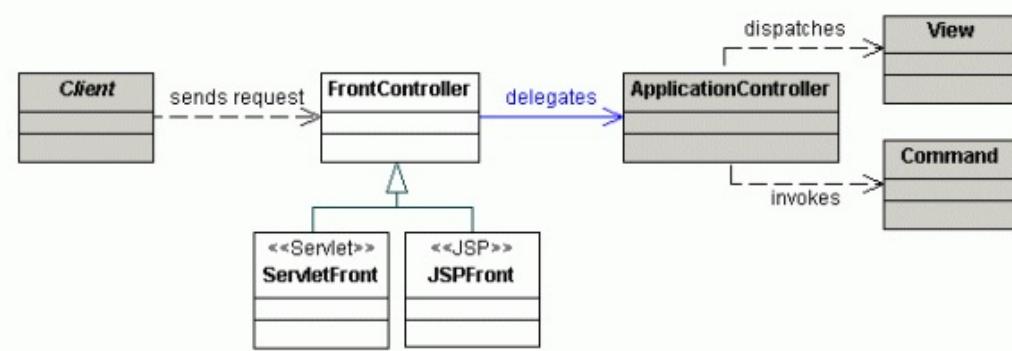


Figura 3.4: Estructura del Front Controller. Extraída de [48]

Page Controller

Lo que eventualmente soluciona este patrón de diseño es la eliminación del controlador basado en JSP's, de manera que se remueve código que realiza invocaciones a la lógica de negocios como respuesta a las acciones del usuario en el JSP y así generar componentes reutilizables y desplegarlos por acción de usuario. Por lo tanto este patrón consolida el procesado de acciones del usuario realizadas en las peticiones en la lógica de negocios, y determina la vista correcta de la página resultante, como se muestra en la figura 3.7.[48, 30]

Intercepting filters

Este patrón de diseño mejora la verificación de peticiones antes y después, como autorizaciones, por lo que este proceso es reutilizable y transparente, además facilita la realización de configuraciones como restricciones en ciertas acciones del usuario sin modificar el código en los controladores.

Context Object

Permite encapsular el estado de un objeto en cualquier protocolo particular, lo que permite completa independencia para poder usarse en cualquier parte de la aplicación. Esta separación de dependencia de protocolos permite dar soporte a diferentes clientes con el mismo código base, además brinda una forma más fácil de probar los Page Controllers ya que pueden probarse sin utilizar objetos de tipo *servlet* en la

3. PATRONES Y ANTIPATRONES DE DISEÑO

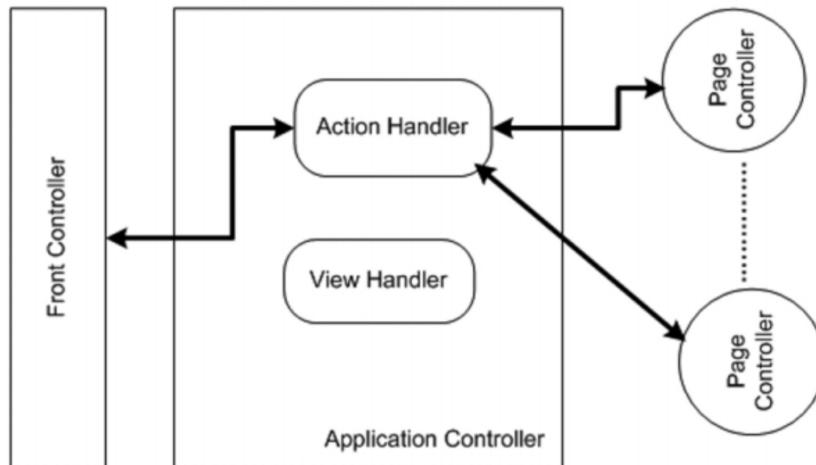


Figura 3.5: Flujo de trabajo del Action Handler. Extraído de [48]

ejecución. [43, 48] La figura 3.8 muestra el diagrama de clases de este patrón.

3.1.2.3 Patrones de la capa de negocios

Service Locator

La función principal del patrón Service Locator es encapsular los objetos de la *Interfaz de Nombrado y Directorio Java* (JNDI, por sus siglas en inglés) y eliminar cualquier sobrecarga en el rendimiento, devolviendo instancias de servicios por petición. Básicamente la idea principal del patrón es tener un objeto que sabe como permanecer en contacto con todos los servicios que la aplicación pueda necesitar.[48, 44]

Los beneficios principales obtenidos al usar este patrón son:

- Abstracción del mecanismo de búsquedas complejas asociadas al servicio de objetos, así se añade flexibilidad porque el servicio a clientes se libera del código de búsquedas
- Es fácil exteriorizar la configuración del Service Locator

A pesar de ser bastante fácil de implementar, se complica a la hora de realizar pruebas unitarias, además para software más complejo donde se haga uso de clases por múltiples aplicaciones mejor se recomienda utilizar la *Inyección de Dependencias*.

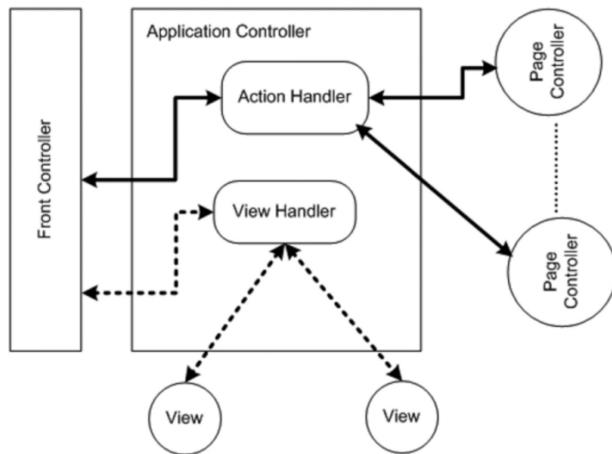


Figura 3.6: Flujo de trabajo del View Handler. Extraído de [48]

Business Delegate

Este patrón funciona como un adaptador para invocar objetos de negocio de la capa de presentación, es decir, se utiliza para encapsular el acceso a servicios de negocio ocultando los mecanismos de acceso a estos, como se muestra en la figura 3.9. Lo que brinda beneficios como el desacoplamiento de capas entre la de negocios y la de presentación, dando como resultado una mantenible y flexible capa de presentación.

Session facade

Este patrón de diseño está implementado como un nivel de componente más alto, como los *Enterprise JavaBeans* (EJB) de sesión, pero manteniendo las interacciones entre los componentes de bajo nivel, como las entidades EJB, además provee una interfaz sencilla para la funcionalidad de una aplicación o parte de ella, permite aprovechar de manera efectiva los servicios de administración de contenedores tales como transacciones y seguridad además de ayudar a esclarecer las responsabilidades de los diferentes componentes de la aplicación. La figura 3.10 muestra el diagrama de clases de este patrón.

A pesar de todo una desventaja de este patrón es su empinada curva de aprendizaje con conceptos complejos y además de las diferentes clases e interfaces hay una basta información de configuración requerida que añade una sobrecarga en el

3. PATRONES Y ANTIPATRONES DE DISEÑO

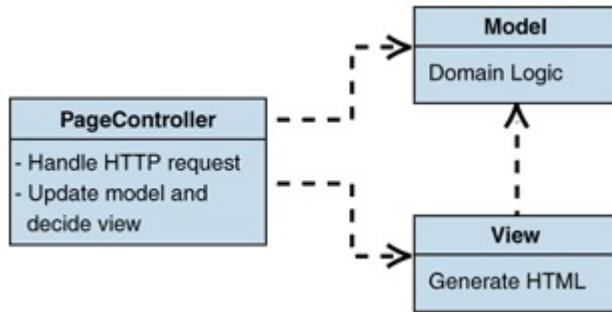


Figura 3.7: Estructura del Page Controller. Extraido de [30]

mantenimiento.[48, 23]

Application Service

Este patrón centraliza la lógica de negocios a través de diversos componentes y servicios de la capa de negocios por medio de clases POJO (Plain Old Java Object, acrónimo en inglés) y aunque es posible poner código de la lógica de negocios en la sesión, no es la mejor opción.[48, 29] La figura 3.11 muestra el diagrama de clases a detalle.

Los beneficios que se logran con la implementación del patrón son:

- Una lógica de negocios encapsulada en simples componentes POJO.
- Los componentes POJO hacen la aplicación mucho más sencilla de probar y ejecutar fuera del contenedor.
- Se obtiene un mejor rendimiento porque la sesión recae en los POJO.

Business Interface

Este patrón de diseño consolida los métodos de la lógica de negocio en una interfaz común, figura 3.12, a modo de que pueda reforzar las verificaciones en tiempo de compilación y prevenir anomalías entre la interfaz remota y la implementación del bean. Esta implementación de business interface asegura consistencia y el código o la capa redundante asociada con business delegate se remueve porque el controlador usa el proxy de las implementaciones del business interface.

3.1 Patrones de diseño

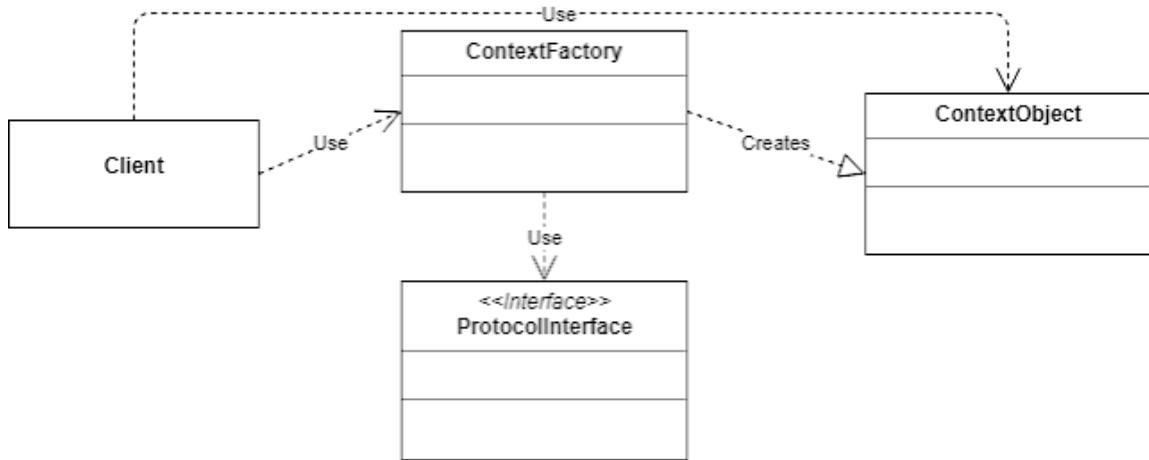


Figura 3.8: Diagrama de clases del Patrón Context Object. Extraído de [43]

3.1.2.4 Patrones de la capa de integración

Objeto de Acceso a Datos

La implementación del patrón de Objeto de Acceso a Datos (DAO, por sus siglas en inglés) ayuda a encapsular la lógica de acceso a datos y provee una interfaz consistente para los componentes de la capa de negocios.[48] En los DAO se tienen los siguientes componentes, como se puede ver en la figura 3.13:

Procedure Access Object

Este patrón sirve para invocar procedimientos almacenados en las bases de datos sin interactuar directamente a bajo nivel con la API de JDBC por lo que promueve la orientación a objetos, minimiza la redundancia de código y se encarga de manejar el código de bajo nivel y administrar los recursos. La figura 3.14 muestra el diagrama de clases de este patrón. [37, 48]

Service Activator

Este patrón de diseño ayuda a recibir y llevar a cabo peticiones de servicio asíncronas, de modo que pueda soportar casos de uso de larga ejecución como al generar reportes y que los usuarios no queden bloqueados durante ese proceso, la figura 3.15 muestra el funcionamiento. [48]

3. PATRONES Y ANTIPATRONES DE DISEÑO

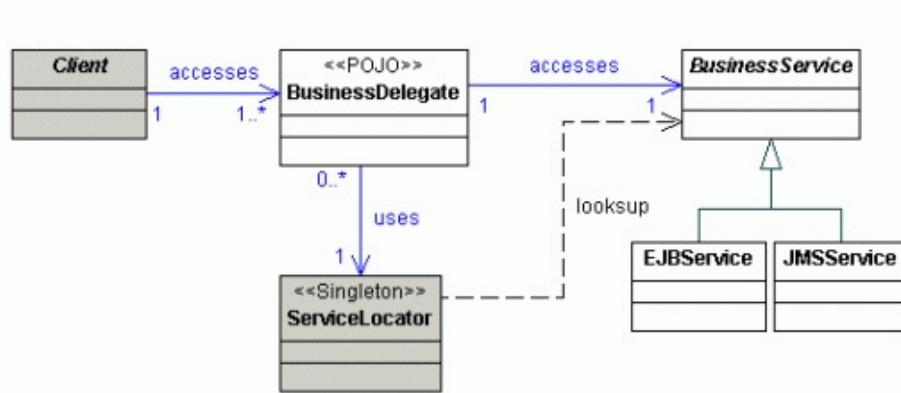


Figura 3.9: Diagrama de clases del Patrón Business Delegate. Extraído de [20]

Web Service Broker

Permite proporcionar servicios de negocio a clientes externos basados en un protocolo web estándar y *XML*. De este modo se logran diversos beneficios como la fácil exposición de servicios de POJO's existentes en una variedad de opciones remotas, sin importar la tecnología o plataforma el acceso a los servicios puede ser lograda a través estos servicios web.[48]

3.1.3 Patrones en Microservicios

API Gateway

Este patrón de diseño, figura 3.16, permite acceder a los diferentes microservicios ya que funciona como el punto de entrada de todas las peticiones de los clientes, de este modo si el resultado debe ser la unificación de información de consultas en varios microservicios se puede lograr por medio de este patrón.[10]

Remote Procedure Invocation

Por medio de este patrón, un cliente realiza una invocación a un servicio usando un síncrono protocolo de invocación a procedimientos remotos, como REST, de modo que exista intercomunicación con otros servicios ya que en ocasiones los servicios

3.1 Patrones de diseño

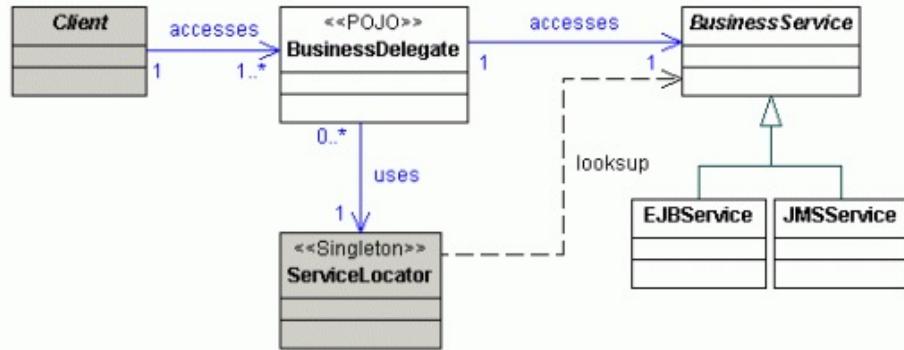


Figura 3.10: Diagrama de clases del Patrón Session Facade. Extraído de [23]

deberán colaborar entre sí para manejar las peticiones.[60]

Circuit breaker

Este patrón soluciona el problema del tratamiento de fallas entre comunicaciones de microservicios, de manera que cuando un servicio realiza una petición a otro y este se no se encuentra disponible ya sea por alta latencia u otro fallo se evite el consumo de recursos por la cascada de errores provocados, es decir, se mantiene monitoreando el estatus de los microservicios y cuando se cumple un margen de fallas entre los microservicios con este patrón se cierra el paso de peticiones desde donde ocurrió el primer error y hasta que funcionen con normalidad permite el paso nuevamente.[11, 2] La figura 3.17 muestra el diagrama de secuencia.

Service discovery

Este patrón permite la identificación y ubicación de los microservicios que conforman todo el ecosistema sin necesidad de especificar exactamente dónde se encuentran y que sean estos microservicios quien se registre automáticamente ante una entidad Service Registry, de manera que cuando se requiera consumir un microservicio, primero se consulte al Service Registry quien conoce dónde se ubican todas las instancias disponibles. [12]

3. PATRONES Y ANTIPATRONES DE DISEÑO

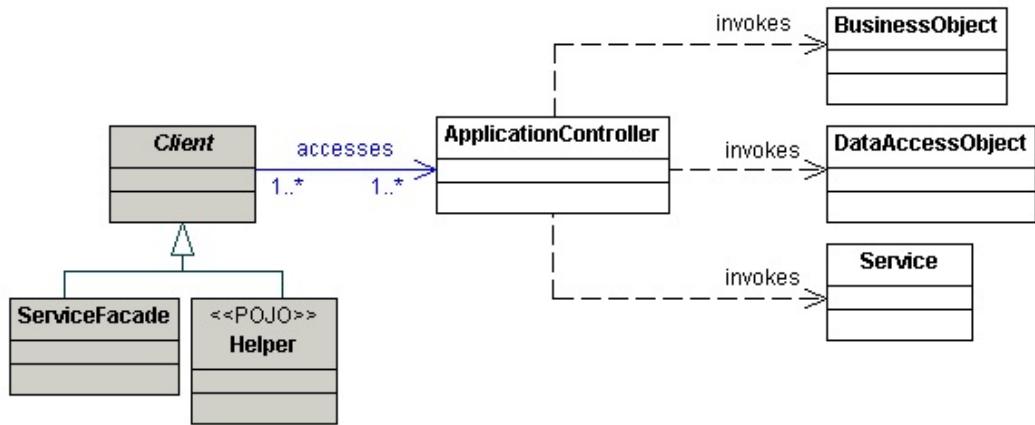


Figura 3.11: Diagrama de clases del Patrón Application Service. Extraído de [29]

Database per service

Este patrón de diseño permite la división modular de la base de datos, de modo que cada servicio que cumple un objetivo de negocio posee su propia base de datos, ésta también es pequeña de manera que solo almacena lo que el microservicio requiera, como se observa en la figura 3.18.

3.1.4 Patrón Strangler

Este patrón de diseño permite realizar migraciones de sistemas legados reemplazando funcionalidades particulares con nuevos servicios o aplicaciones, así cuando el nuevo servicio está listo éste se pone en producción y la funcionalidad antigua en la aplicación es eliminada. De esta manera ante la necesidad de nuevos desarrollos se realizan como parte de nuevos servicios y no de la aplicación que se va migrando. Esto conlleva grandes beneficios como reducir la necesidad de esperar a que todo el código de la aplicación sea reescrito desde cero en la arquitectura de microservicios, en la figura 3.19 se aprecia la implementación del patrón. [34]

Para implementar este patrón de diseño se requiere de básicamente tres pasos, como podemos ver en la figura 3.20: [34]

1. **Transformación:** Primero todo el tráfico se enruta hacia la aplicación web monolítica, mientras se crea un entorno paralelo para el nuevo desarrollo.

3.2 Antipatrones de diseño

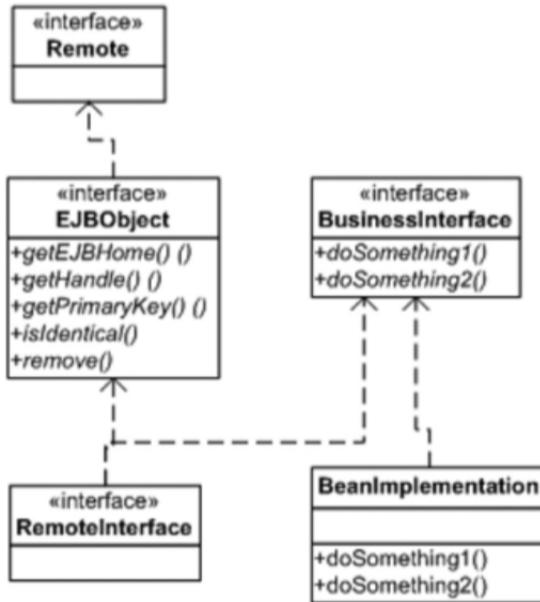


Figura 3.12: Diagrama de clases del Patrón Business Interface. Extraído de [48]

2. **Coexistencia:** Cuando el nuevo microservicio sea desarrollado se prueba en paralelo a la aplicación web.
3. **Eliminación:** Una vez que el nuevo microservicio funcione del modo esperado y haya pasado las pruebas de software satisfactoriamente se procede a eliminar el componente de la aplicación web monolítica.

3.1.5 Patrón de descomposición por habilidad de negocio

3.2 Antipatrones de diseño

3.2.1 ¿Qué es un antipatrón de diseño?

Como su contra-parté, el patrón de diseño, define una supuesta solución a un problema, sin embargo dicha solución es deficiente que termina generando consecuencias negativas, esto sucede cuando un administrador de proyectos o desarrollador de software no posee suficiente experiencia o conocimientos en resolver cierto problema en particular, así como el aplicar un patrón de diseño de una mala manera. Estos antipa-

3. PATRONES Y ANTIPATRONES DE DISEÑO

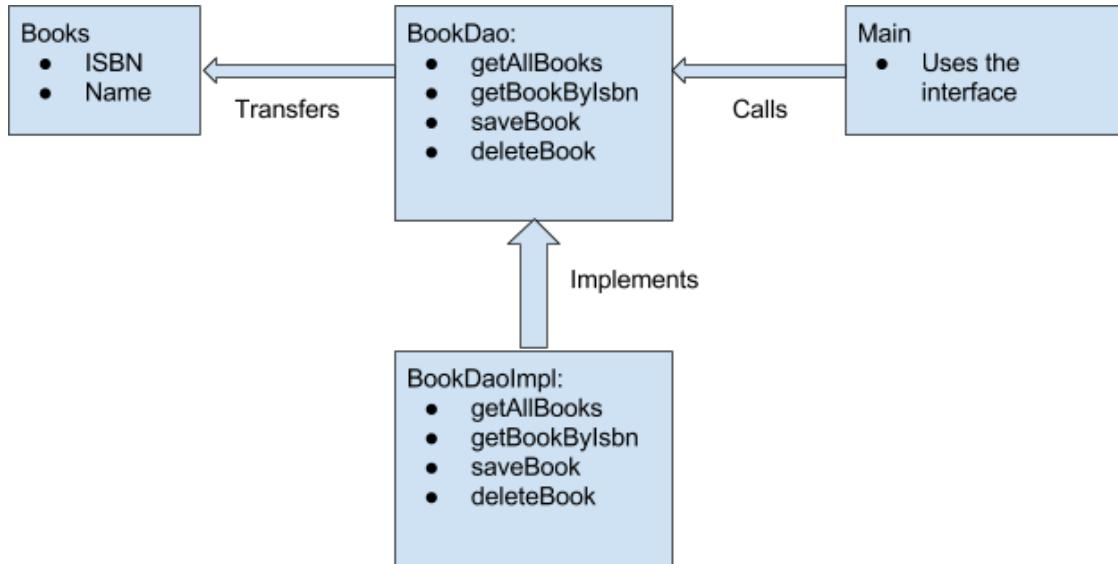


Figura 3.13: Ejemplo del Patrón DAO. Extraido de [54]

trones resaltan los problemas más comunes en la industria y proveen ciertos criterios que ayudan a identificarlos y determinar sus causas.[7]

3.2.2 Antipatrones en Aplicaciones Web

Aunque pareciera que los antipatrones rara vez ocurren, es más común de lo que suponemos y se deben eliminar antes de que tenga graves consecuencias. Los siguientes antipatrones son lo más comunes en el desarrollo de software.

Spaghetti Code

Es el antipatrón de diseño más clásico y famoso, ha existido desde el origen de los lenguajes de programación, es más recurrente en los lenguajes no orientados a objetos, éste define una estructura desorganizada del código de algún software que trae como consecuencia legibilidad difícil, mal entendimiento, imposible la reutilización de código, entre otras cosas, incluso llega a ser incomprensible para el desarrollador original si éste se ausenta por algún tiempo del software.[14]

La solución a este antipatrón es la refactorización del código, que es básicamente reestructurar el código de forma ordenada para facilitar tanto legibilidad como nuevos requerimientos no anticipados, es además una tarea muy laboriosa que debe ser

3.2 Antipatrones de diseño

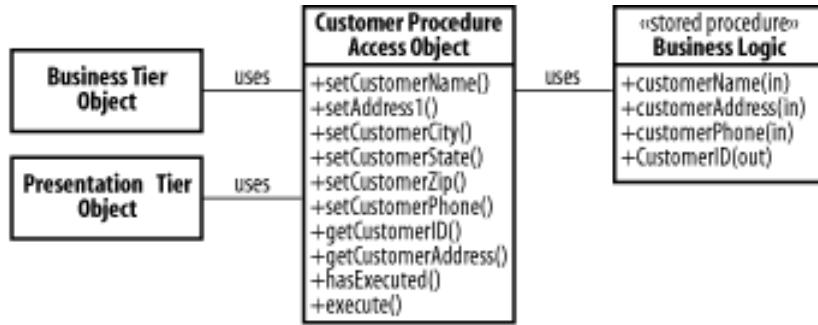


Figura 3.14: Diagrama de clases del Patrón Procedure Access Object. Extraído de [37]

prevenida antes de que suceda.[14]

Magic Servlet

En este antipatrón lo que lo identifica como Magic Servlet es que un simple servlet se encarga de realizar todas tareas del modelo, vista y controlador, esto es, conectarse directamente a la base de datos o recursos externos, sirve salidas de HTML directamente además de ser demasiado grandes y complicados de entender. La solución a este antipatrón es la refactorización en MVC, esto es, separar todas las salidas en JSP o separarlos en más servlets, también separar las llamadas a la base de datos en acciones reutilizables y coordinar el procesado de entradas, acciones y vistas con servlets o controladores.[37]

OpenSessionInView

Es uno de los antipatrones más comunes en el desarrollo Java, que sucede en la capa de persistencia de datos, como al usar *Hibernate* o la API de Persistencia de Java (JPA, siglas en inglés), prácticamente el problema sucede cuando en las relaciones de Hibernate o JPA son de uno a muchos o de muchos a muchos y Hibernate permite definir cómo obtener esas entidades, tomando como ejemplo la figura 3.21 si al momento de pedir un curso desde la vista también obtener la relación con Profesor, o simplemente esa entidad por sí sola, de ser lo primero, ocurre que Hibernate siempre estará pidiendo la asociación cuando se pida un curso, de manera que aunque no se utilice en la vista o cualquier otro lado sólo se estará cargando en memoria y saturando la Máquina Virtual de Java (*JVM* del inglés Java Virtual Machine); si es

3. PATRONES Y ANTIPATRONES DE DISEÑO

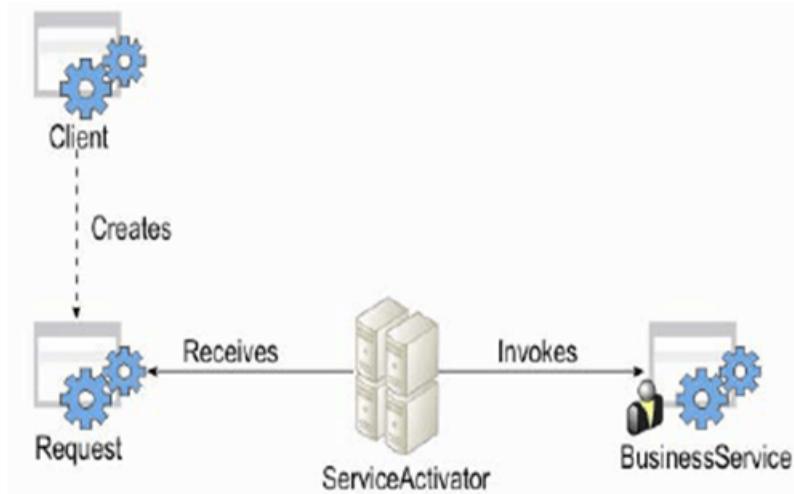


Figura 3.15: Patrón Service Activator. Extraído de [22]

lo segundo, Hibernate pone las demás entidades como *proxies* para poder pedirlas posteriormente si es que se hace un llamado a estos por medio de un *getter* en la misma petición sin embargo si la conexión se cierra antes de pedir más datos se presenta un error llamado *LazyInitializationException*. [19]

Por lo que optar por emplear el antipatrón *OpenSessionInView* resultaría como la supuesta solución que permitiría a la sesión de Hibernate o el entity manager de JPA abierto hasta que se resuelve la vista, sin embargo, esto genera un problema de $n + 1$ *queries*, ya que por cada elemento se realiza una consulta, por ejemplo, un listado que imprima los Profesores y sus Cursos, realizará:

- 1 consulta para listar a todos los profesores
- n consultas para obtener los cursos de cada profesor

Así que debe evitarse su uso a toda costa para no perjudicar el rendimiento, la solución es utilizar *Join fetch*, que está disponible tanto en la clase *Criteria* como en *HQL* y usarlo sólo donde se requiera para evitar dejar objetos sin uso en la JVM.

Obsolecencia continua

Con las liberaciones de nuevas versiones de productos debido al rápido crecimiento de la tecnología, los desarrolladores se están enfrentando constantemente a ponerse al día en cuanto al producto que manejan para construir las aplicaciones, por ejemplo, Java es bien conocido por presentar este inconveniente, cuando el libro de Java 8.X

3.2 Antipatrones de diseño

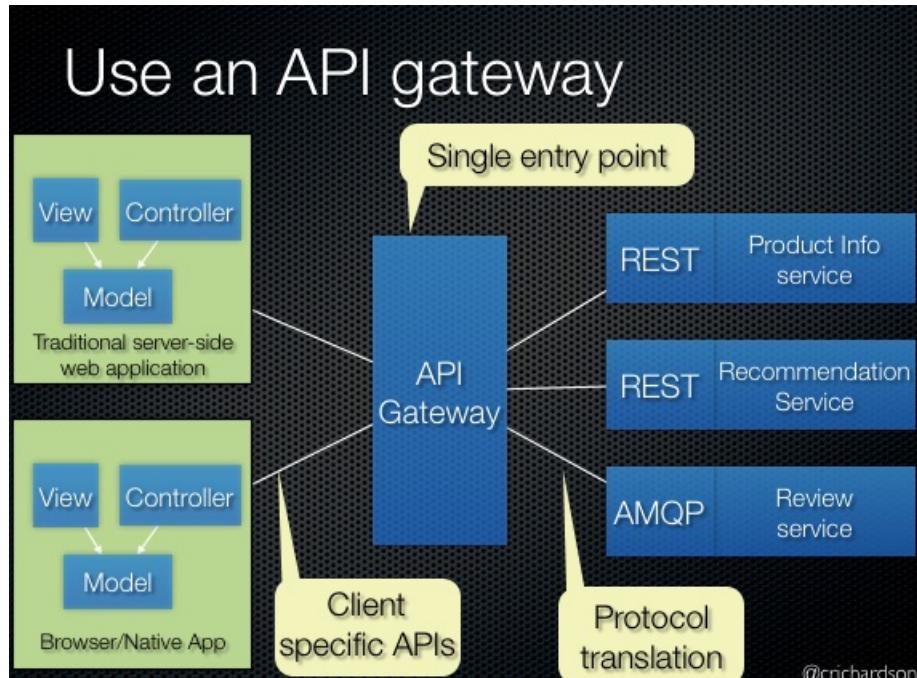


Figura 3.16: Patrón API Gateway. Extraído de [10]

sale, un nuevo Kit de Desarrollo Java (JDK, siglas del inglés Java Development Kit) que vuelve obsoleta la información. Aunque existen muchos otras tecnologías que participan en la continua obsolescencia.[3]

Una posible solución a este antipatrón es convertir el *software como servicio* (SaaS, del inglés Software as a Service). De tal manera que las actualizaciones se harán tan frecuentes como el vendedor desee, así los costos recaen en el proveedor en vez del equipo de desarrollo. Otra manera es definir y mantener una versión del entorno de desarrollo, pero a la larga podría dificultarse la compatibilidad de los sistemas, lo que hace a este antipatrón uno de los más temidos y de los que es fácil caer en él.[3, 26]

Copiar y pegar código

Este antipatrón es muy común en el campo del desarrollo, la premisa es que si hay software similar por desarrollar en dos o más proyectos por qué no usar las partes del código que hacen lo mismo en vez de hacerlo desde cero, aunque la mayoría de las veces es bueno eso, la técnica puede emplearse mal y tener consecuencias negativas. Por ejemplo, suponiendo que hay un software ya probado que funciona correcta-

3. PATRONES Y ANTIPATRONES DE DISEÑO

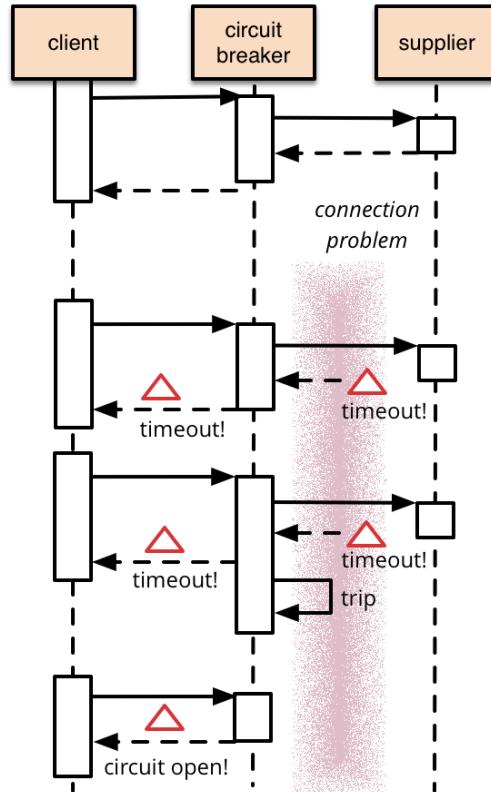


Figura 3.17: Diagrama de secuencias del Patrón Circuit Braker. Extraído de [2]

mente, pero ahora tiene nuevos requerimientos por añadir, sin embargo lo que debe realizarse tiene acciones similares a otro requerimiento ya puesto en marcha, lo que podría ocurrir es que en vez de hacer código reutilizable para ambos requerimientos, surja la idea de copiar y pegar la sección de código del anterior requerimiento, lo que provoca duplicidad e incremento en las líneas de código que a final de cuentas hace más difícil el mantenimiento si se diera el caso de que existe un *bug* en el código copiado, hay que arreglar también el código de donde se obtuvo.[6]

La solución a este antipatrón es la refactorización de código, eliminando código duplicado y transformar el software en pequeñas bibliotecas o componentes que faciliten la reutilización de código, este proceso puede ser muy largo y costoso si el proyecto está conformado en su mayoría por el antipatrón. Este proceso de refactorización involucra tres etapas: *Code mining*, refactorización y administración de la

3.2 Antipatrones de diseño

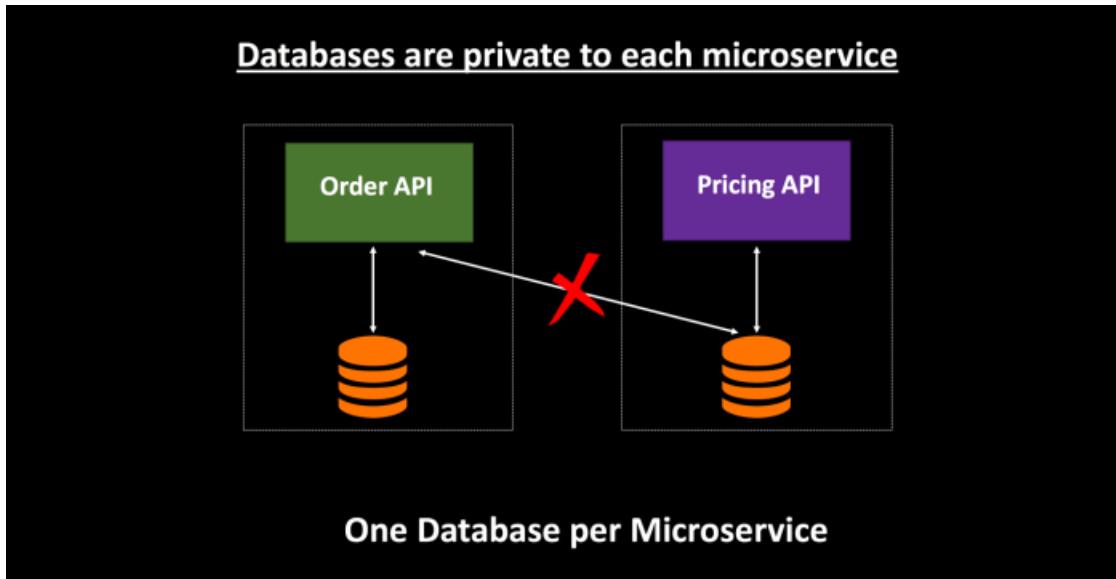


Figura 3.18: Database per service. Extraído de [35]

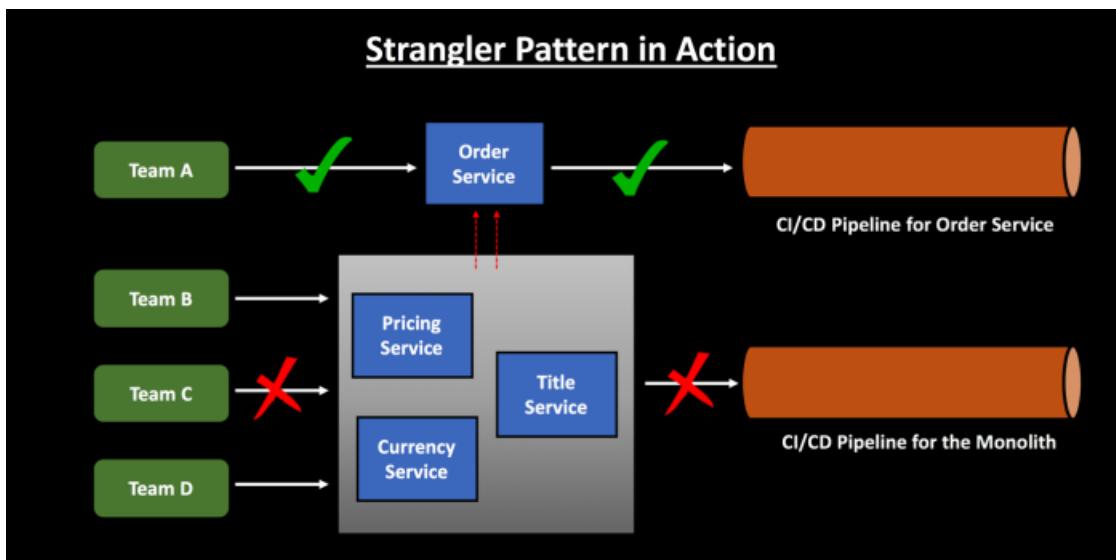


Figura 3.19: Patrón Strangler. Extraído de [34]

configuración.[6]

3. PATRONES Y ANTIPATRONES DE DISEÑO

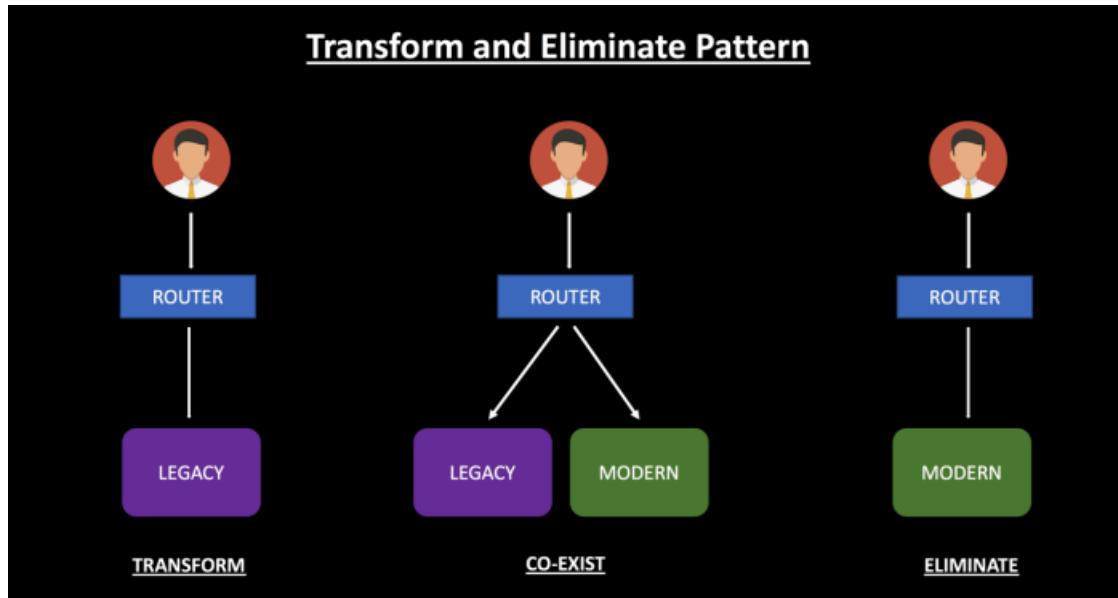


Figura 3.20: Implementación del patrón strangler. Extraído de [34]

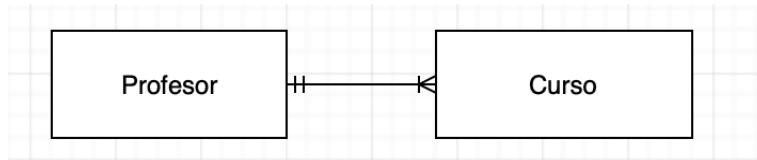


Figura 3.21: UML

Reinventar la rueda

Este antipatrón es muy común en los desarrollos de software que se realizan de manera remota, donde generalmente los desarrolladores no tienen una interacción con los demás miembros que trabajan en el mismo proyecto. Sin embargo, éste se encuentra presente también en desarrollos presenciales, de 32 proyectos de software orientados a objetos, no se tiene evidencia de que se haga una reutilización de código exitosa.[13]

Este antipatrón sólo es conveniente en un ambiente de investigación así como la intención de reducción de costos en coordinación donde diferentes desarrolladores

3.2 Antipatrones de diseño

con diferentes habilidades trabajan de manera remota. Sin embargo su uso es demasiado extendido y común entre las organizaciones ya que la mayoría de los métodos de desarrollo de software consideran que cada proyecto está aislado de otros, lo que conlleva a los desarrollos desde cero, y por ende se construyen sistemas aislados.[13]

Navaja Suiza

Este antipatrón de diseño trata sobre el uso excesivo de clases o interfaces de un software, en donde el desarrollador intenta cubrir todos los posibles casos de uso que podrían existir o necesitar el usuario final, pero en el proceso sólo deja sobrecargado de código las clases. Esto es realmente problemático porque se ignora la complejidad que genera el comprender cómo se supone la clase debe de funcionar incluso en los casos sencillos, además de la dificultad para depurar, documentar y dar mantenimiento.[16]

Como a menudo se encuentra interfaces y estándares muy complejos en los desarrollos es sumamente importante tener definidas las convenciones para este tipo de tecnologías para que la administración de la arquitectura de las aplicaciones complejas no se vea comprometida, este documento es llamado Perfil, y se aplica para definir las formas correctas de utilizar tecnologías complejas. Y es conveniente también que además del Perfil, se incluya la descripción y especificación de las secuencias de ejecución, llamadas a métodos y *manejo de excepciones*.[16]

3. PATRONES Y ANTIPATRONES DE DISEÑO

CAPÍTULO
4

Estado del Arte

Este capítulo trata sobre la actualidad de las tecnologías que se utilizan en la industria respecto a aplicaciones web y microservicios, además de las estrategias de migración de monolítico a microservicio.

4.1 Lenguajes de programación en construcción de software

De acuerdo al Índice TIOBE[8], los 5 lenguajes de programación que más se utilizan en el desarrollo de aplicaciones se muestran en la tabla 4.1.

Posición	Lenguaje de programación
1	Java
2	C
3	C++
4	Python
5	Visual Basic .NET

Tabla 4.1: Top 5 de lenguajes de programación más usados

Tomando como base que el lenguaje Java es el más utilizado de acuerdo al TIOBE, y aprovechando que es uno de los de software libre que es uno de los principales propósitos de la tesis, queda entonces por identificar el ecosistema de la Máquina Virtual de Java (JVM, por sus siglas en inglés).

4. ESTADO DEL ARTE

4.2 Ecosistema JVM

De acuerdo al reporte del 2018 publicado en el sitio web snyk en el cual los datos fueron obtenidos de más de 10,200 cuestionarios, los desarrolladores se mantienen en ciertas versiones de software para el desarrollo de aplicaciones y se presentan a continuación los resultados:

Java Development Kit

El proveedor JDK que la mayoría de los desarrolladores utilizan para la construcción de software se encuentran OpenJDK y Oracle JDK, con un 21% y 70% respectivamente, se puede ver en la figura 4.1 los demás resultados obtenidos y dejando claro que 7 de cada 10 desarrolladores usan Oracle JDK.

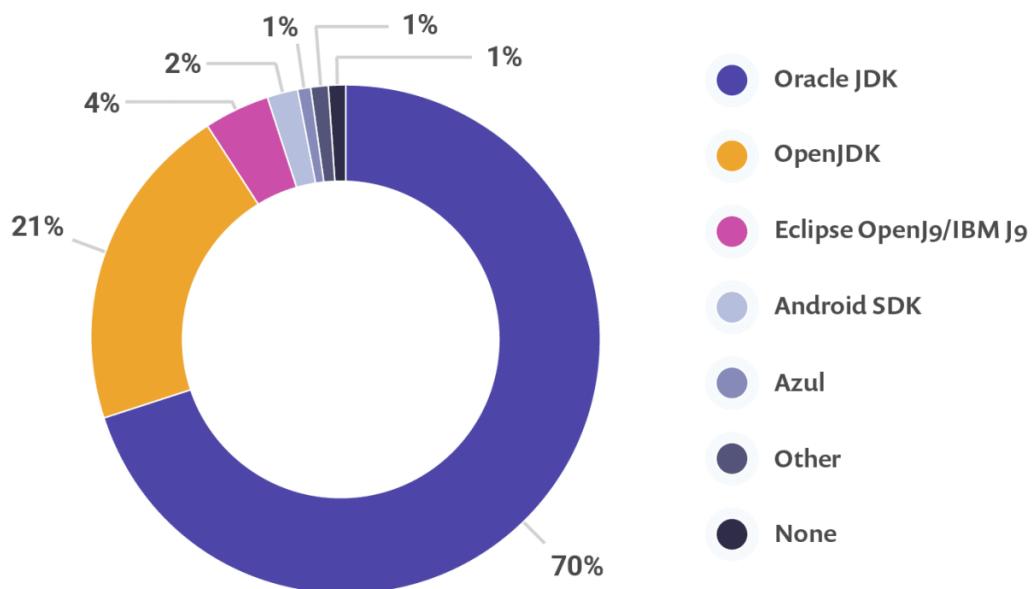


Figura 4.1: Proveedores JDK usados. Extraída de [53]

4.2 Ecosistema JVM

Java Standard Edition

De los resultados sobre la versión de Java más usada se obtuvieron que aún Java 8 sigue siendo el más usado en producción, a pesar de haber versiones más recientes desde la versión 9 hubieron cambios significativos que provocaron que no todos adoptaran esa versión y decidieron mantenerse en la 8 como se puede observar en la figura 4.2.

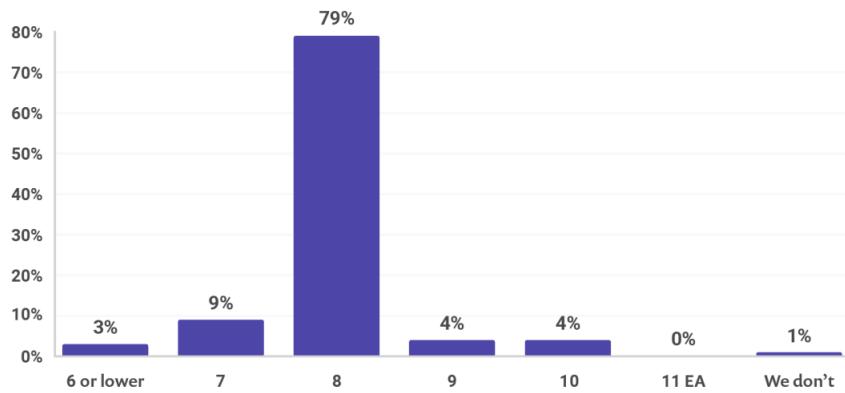


Figura 4.2: Versiones de Java SE utilizados en producción. Extraída de [53]

Java Enterprise Edition

De acuerdo a la figura 4.3, un 40 % no usa ninguna versión de JEE, y del 60 % restante usan una o más versiones de JEE.

Herramientas de automatización de construcción de código

En cuanto a construcción automatizada del software, se ha obtenido que Maven domina con 60 % y Gradle con 19 % se posiciona en segundo lugar en cuanto a utilización por los desarrolladores se refiere y puede observarse en la figura 4.4.

Herramientas para pruebas

Los datos arrojados de los cuestionarios involucran la utilización de una o más de una tecnología. Se puede observar 4.5 que la más utilizada es JUnit.

Enfoque en la Nube

Hablando de la nube, menos de la mitad de desarrolladores han adaptado a este concepto, sin embargo de acuerdo a la figura 4.6 se puede ver que el 43 % del total al

4. ESTADO DEL ARTE

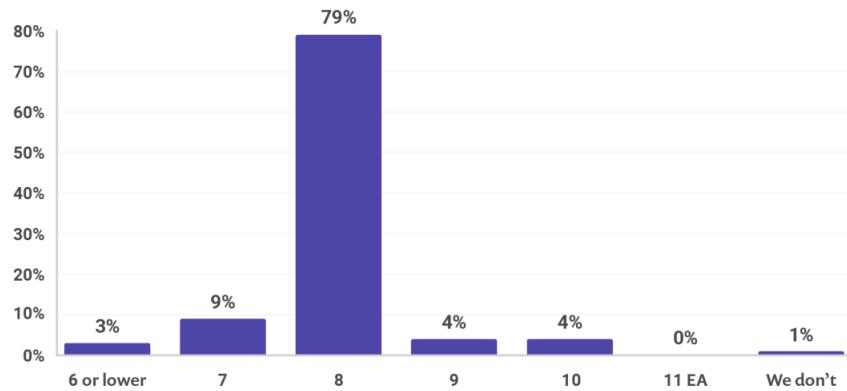


Figura 4.3: Versiones de Java EE utilizados en producción. Extraída de [53]

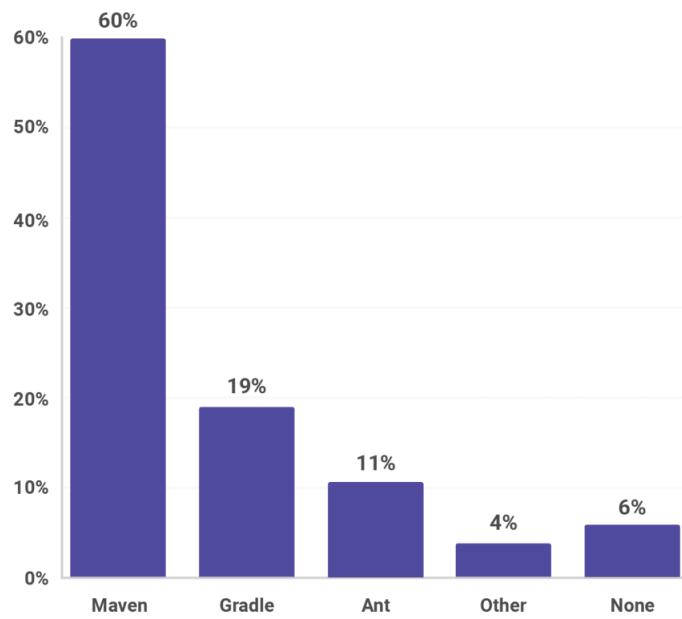


Figura 4.4: Herramientas de automatización usadas. Extraída de [53]

4.2 Ecosistema JVM

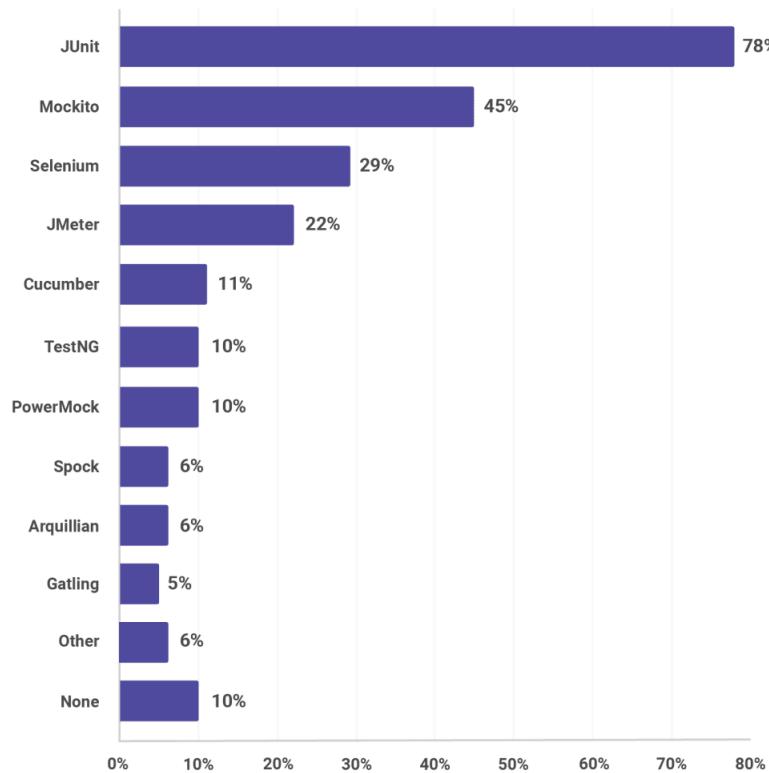


Figura 4.5: Herramientas de pruebas usadas. Extraída de [53]

menos usa un enfoque en la nube basado en contenedores, un 33 % usa máquinas virtuales y 33 % no usa ninguna, también indican que pueden utilizar más de un enfoque.

Framework Object Relational Mapping

De acuerdo a la figura 4.7, 1 de cada 5 desarrolladores no usan ningún Framework ORM, y del 80 % restante usan uno o más frameworks. 54 % utiliza Hibernate, y 23 % Plain ODBC al igual que Spring JDBC Template.

Frameworks Web

En cuanto a los Frameworks para web que se utilizan, podemos ver en la figura 4.8 que los que dominan son Spring Boot y Spring MVC, con 40 % y 38 % respectivamente.

4. ESTADO DEL ARTE

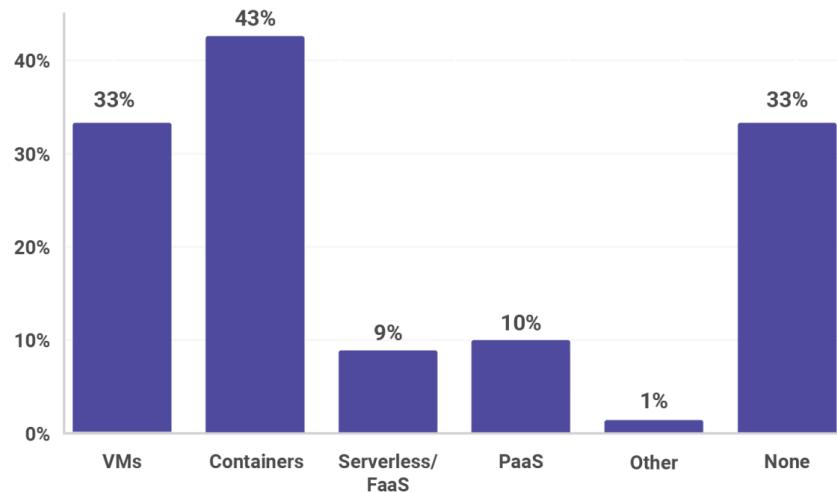


Figura 4.6: Enfoques de la nube implementados. Extraída de [53]

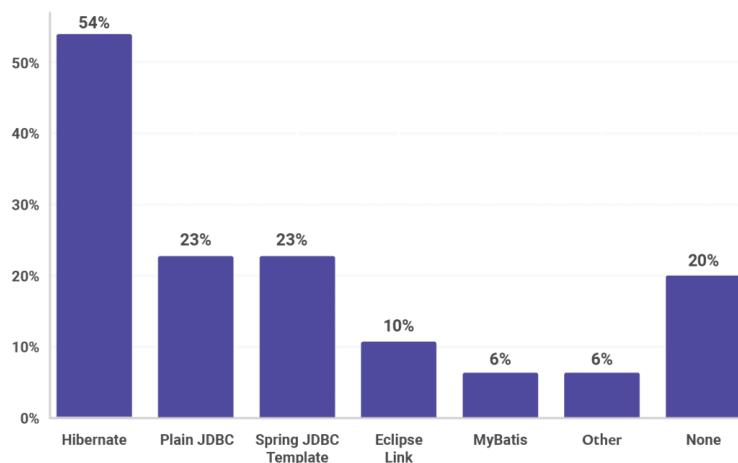


Figura 4.7: Frameworks ORM. Extraída de [53]

4.2 Ecosistema JVM

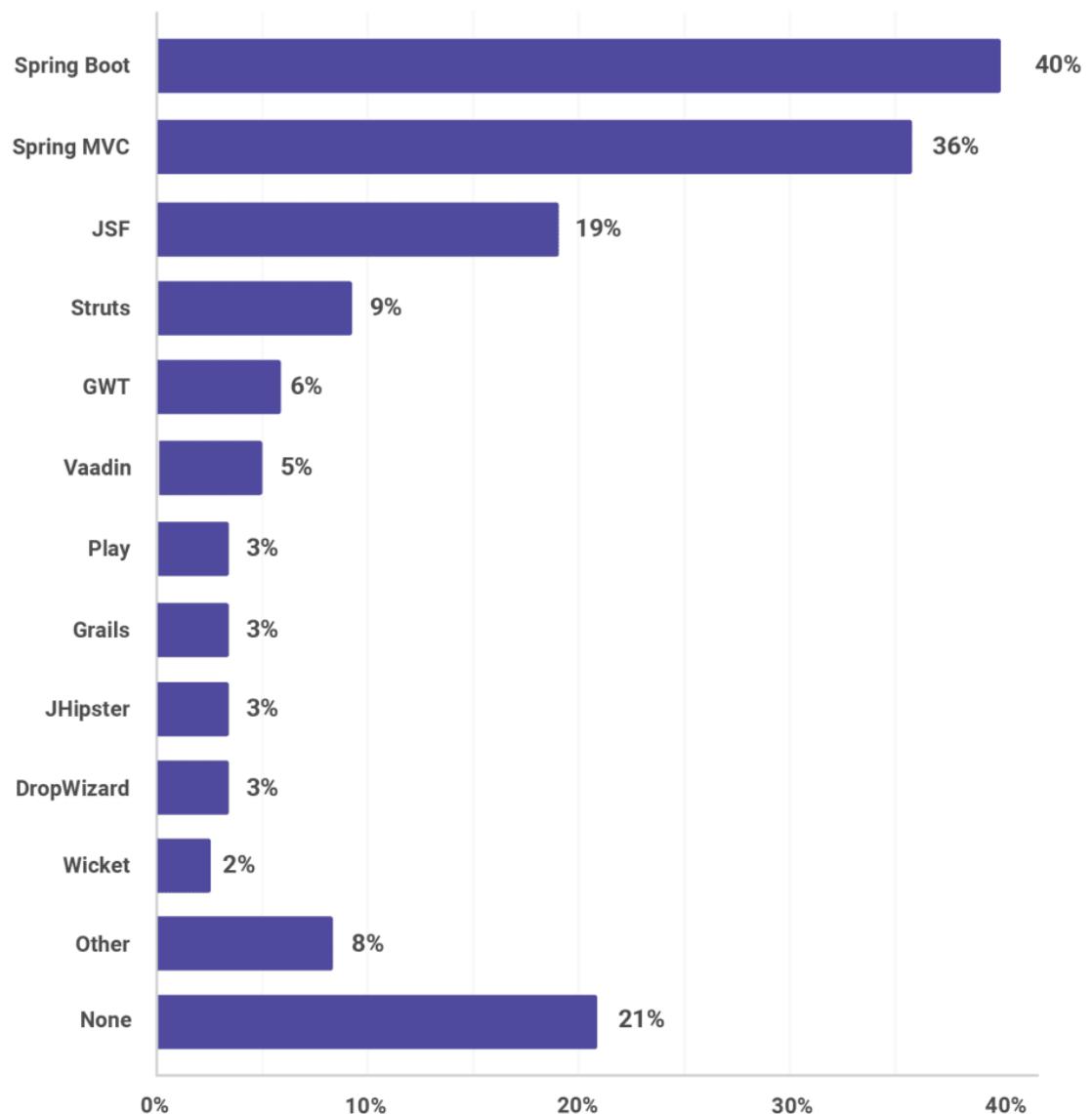


Figura 4.8: Frameworks web más usados. Extraída de [53]

Bases de datos

La figura 4.9 muestra que Oracle domina en un 6% sobre MySQL. MongoDB es la base de datos NoSQL más usada con un 5%.

4. ESTADO DEL ARTE

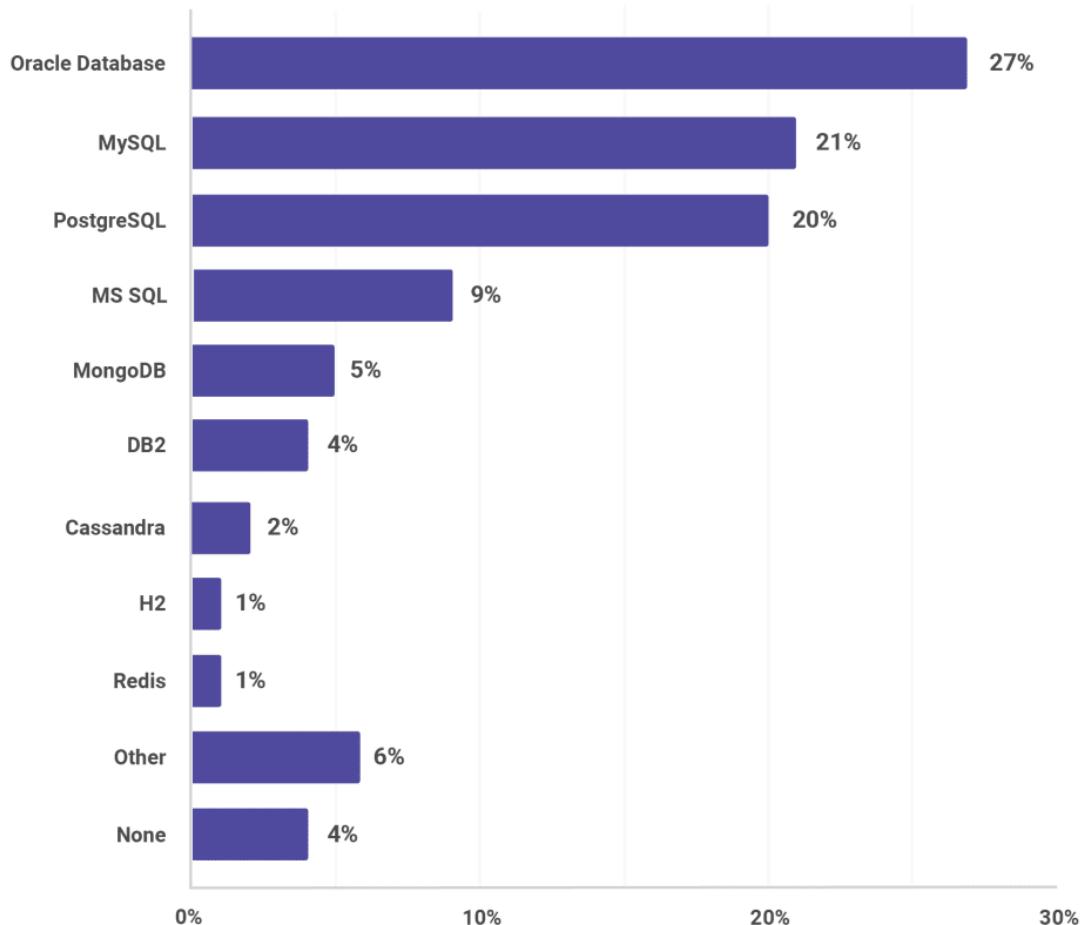


Figura 4.9: Bases de datos usadas. Extraída de [53]

Servidores de aplicaciones

En cuanto a servidores de aplicaciones web, en la figura 4.10 se puede observar que más de 4 de cada 10 encuestados usa Apache Tomcat, y casi 2 de cada 10 usa JBoss/Wildfly, le sigue con un 15 % JBoss/Wildfly y en tercera posición se encuentra Jetty con 9 %.

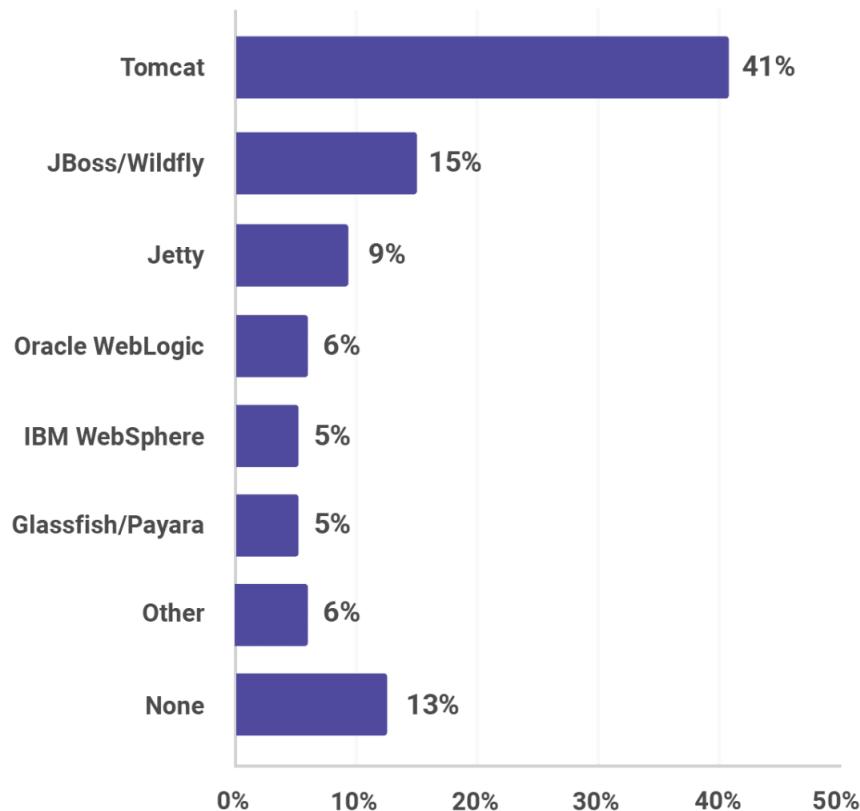


Figura 4.10: Bases de datos usadas. Extraída de [53]

4.3 Microservicios Java

La construcción de microservicios en Java se facilita gracias a Spring Boot que trae consigo varios subproyectos como Spring Cloud que contiene una serie de características y patrones de diseño que logran que el desarrollo de microservicios sean simples configuraciones y cuya finalidad es que los desarrolladores se concentren solamente en las habilidades de negocios. La figura 4.11 muestra de manera general los patrones de diseño que el proyecto Spring Cloud integra.

4. ESTADO DEL ARTE

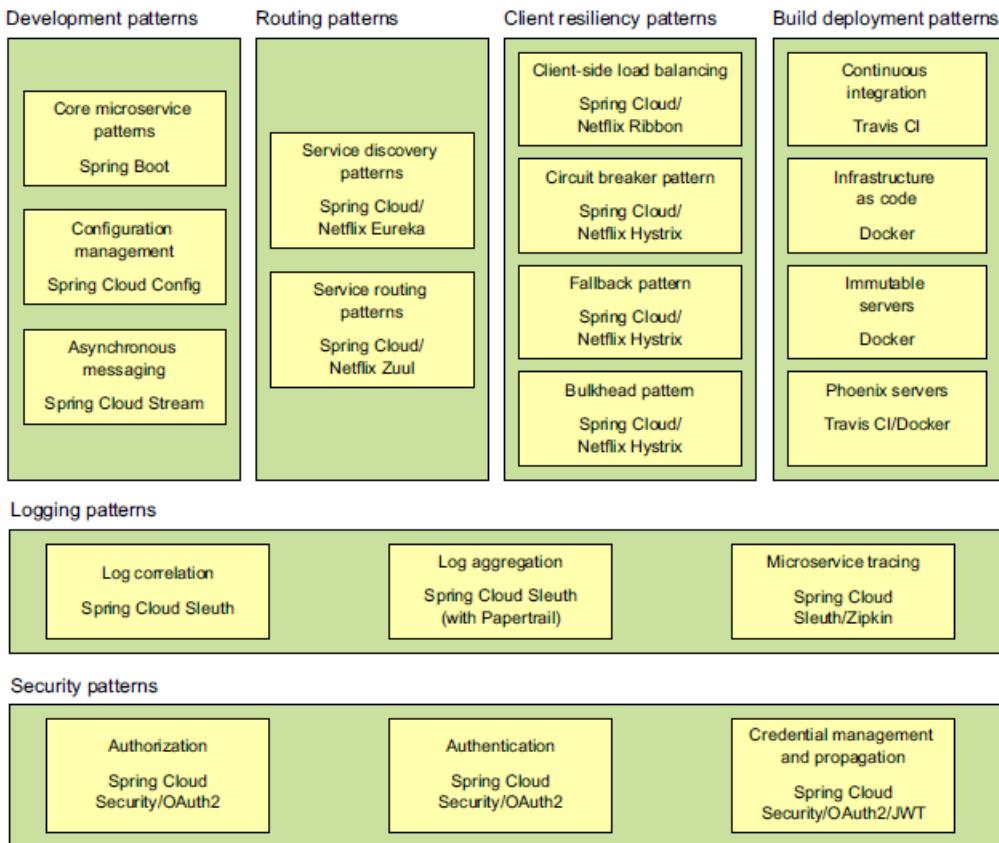


Figura 4.11: Patrones de diseño en Spring Cloud. Extraído de [67]

4.3.1 Modelo *Bounded Context*

Los Requerimientos funcionales y Requerimientos no funcionales son elementos primordiales en el desarrollo del modelo *Bounded Context* del Diseño Guiado por el Dominio (DDD, por sus siglas en inglés), que se detalla a profundidad en el libro “Domain-driven design: tackling complexity in the heart of software” del autor Eric Evans. [40].

Es importante destacar que la arquitectura de la aplicación web puede verse desde múltiples perspectivas que se deben tomar en cuenta para la migración. El artículo escrito por Phillip Krutchen “Architectural Blueprints—The “4+1” View Model of Software Architecture” muestra las cuatro diferentes vistas de una arquitectura de software, describe los aspectos particulares ésta y las relaciones entre el conjunto de

4.3 Microservicios Java

elementos del software, como puede verse en la figura 4.12. [50]

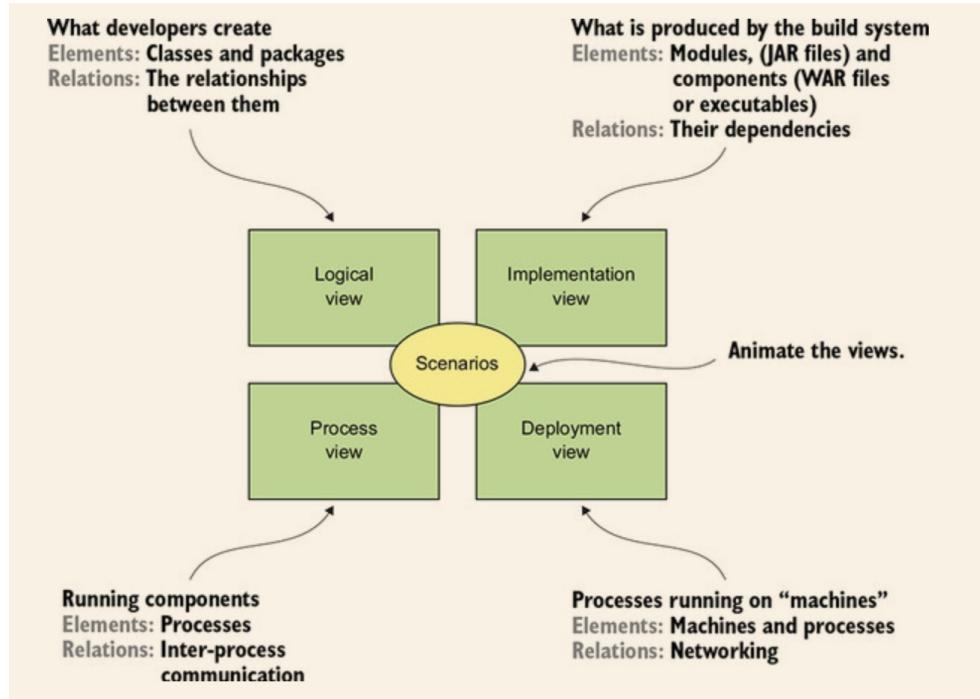


Figura 4.12: Modelo 4+1 de Arquitectura de Software. Extraído y traducido de [50]

En donde la vista lógica son los elementos creados por los desarrolladores (clases, objetos, etc.) y las relaciones entre ellas (herencia, polimorfismo, asociaciones, etc.), la vista de implementación consiste en los módulos representados como paquetes de código y componentes ejecutables (WAR, EAR, JAR) y la relación que existe entre estos componentes, la vista de proceso son los componentes en ejecución, cada elemento es un proceso y la relación entre ellos representa un proceso de intercomunicación, y finalmente la vista de despliegue muestra cómo la aplicación está organizada en las máquinas y la relación entre ellas es a través de la red, y la vista extra o “+1” en el modelo muestra los escenarios, donde cada escenario describe cómo las diversos componentes arquitectónicos en una vista en particular colabora para manejar las peticiones. Por ejemplo la vista de implementación muestra cómo los componentes ejecutables colaboran. [60]

Es fundamental que se tenga en cuenta lo mencionado anteriormente ya que las vistas muestran un aspecto importante y los escenarios revelan los elementos que co-

4. ESTADO DEL ARTE

laboran entre las vistas, además, una aplicación está definida por los *Requerimientos funcionales* y los *Requerimientos no funcionales*.

Una de las formas de desarrollar el modelo Bounded Context es siguiendo los 3 pasos descritos por Charles Richardson en “Microservices Patterns”, como se muestra en la figura 4.13. Cabe señalar que no es un proceso que deba seguirse en orden y es probable que sea iterativo y requiera de un buen criterio.

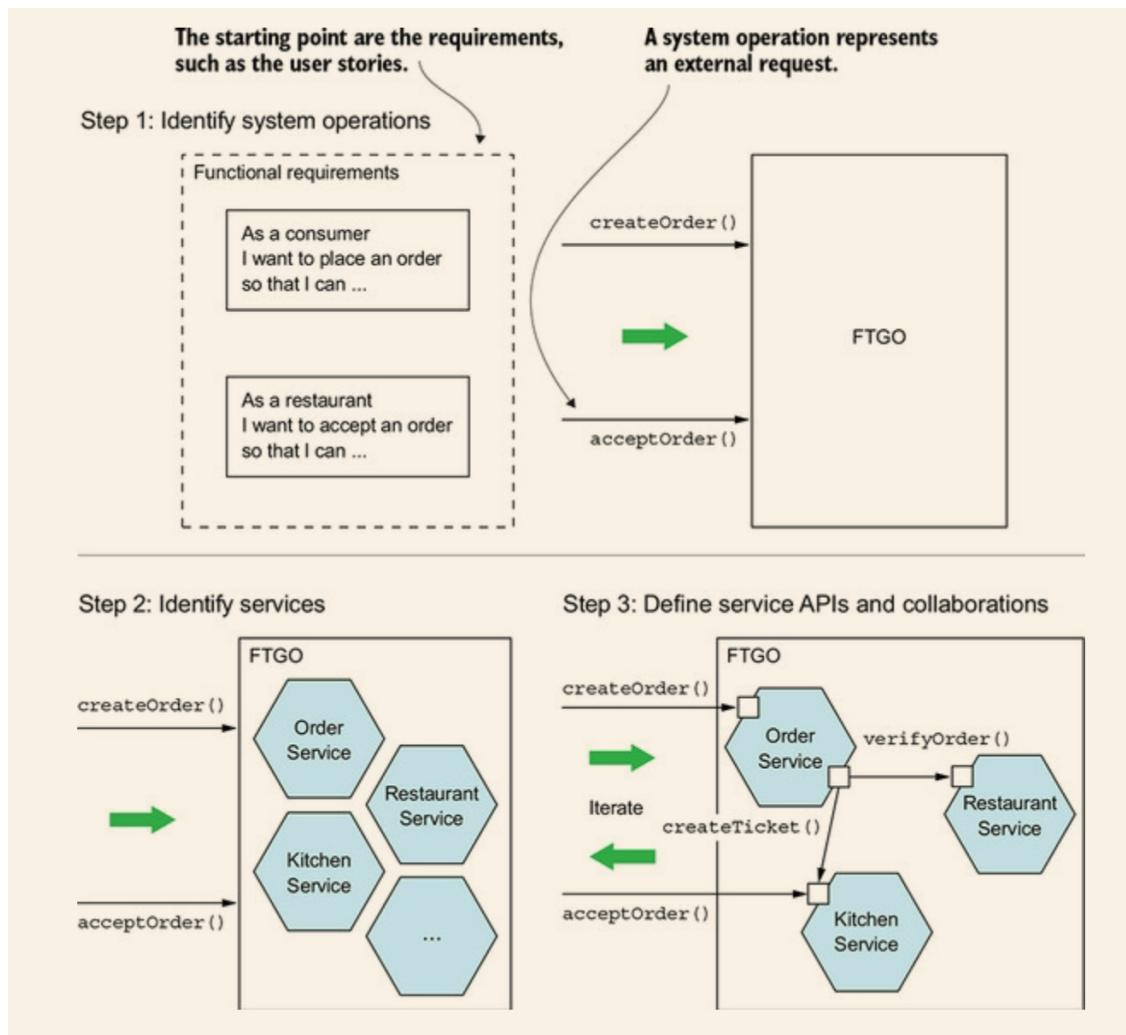


Figura 4.13: Pasos descritos para la identificación y definición de microservicios. Extraído de [60]

CAPÍTULO
5

Diseño del método de migración

Este capítulo trata de la propuesta del método para la migración de aplicaciones web monolíticas a la arquitectura de microservicios. Se plantea la estructura del método y las etapas a seguir para lograr una migración exitosa.

5.1 Método propuesto

De acuerdo a fuentes encontradas de trabajos relacionados, se han identificado las etapas de desarrollo del método para la migración de aplicaciones web a microservicios. Debido a la vasta diversidad en los tipos de desarrollo con requerimientos diferentes, así como a las habilidades de los equipos de desarrollo en distintos escenarios y empresas, un método rígido y único que se ajuste a todo tipo de desarrollo no sería posible, por lo que seguir el Método Situacional de Ingeniería (SME, por sus siglas en inglés) se adecúa mejor para esta propuesta.

El método que se propone para efectuar la migración de aplicaciones web en Java a microservicios consta de 3 etapas principales:

- Etapa 1: Verificación de elegibilidad de la aplicación web para la migración
- Etapa 2: Transformación de la aplicación web a microservicios
- Etapa 3: Verificación y validación de la aplicación resultante

La figura 5.1 muestra el diagrama de flujo general que sigue la metodología, y la figura 5.2 muestra el proceso detallado del método propuesto en la página 62:

5. DISEÑO DEL MÉTODO DE MIGRACIÓN

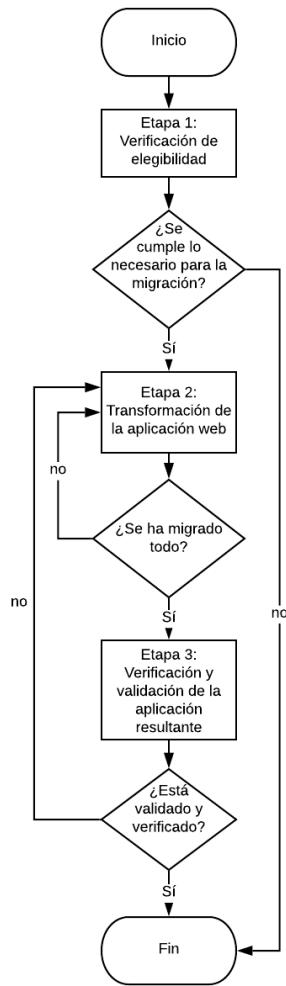


Figura 5.1: Diagrama de flujo general del método de migración.

5.2 Etapa 1: Verificación de elegibilidad de la aplicación web para la migración

Aunque pareciera que todas las aplicaciones deberían migrarse a microservicios, ésta acción podría ser contraproducente afectando tiempo y costos de inversión desperdiados, por tal motivo y para garantizar que la migración a microservicios brinde los beneficios esperados se proporcionan a continuación una lista de los casos en los

5.2 Etapa 1: Verificación de elegibilidad de la aplicación web para la migración

que conviene realizar una migración:

- La aplicación web es utilizada por muchos usuarios.
- La aplicación web ha tenido que replicarse en otros servidores.
- El desarrollo y/o mantenimiento de la aplicación web involucra más de un equipo de desarrolladores.
- Debido a peticiones del cliente por liberaciones más rápidas, la aplicación web mantiene liberaciones constantes en pocas semanas, ya sea para la resolución de errores o adición o eliminación de características.
- La aplicación web posee requerimientos ampliamente relacionados a escalabilidad, fiabilidad y disponibilidad.
- La aplicación web se ha vuelto demasiado grande que afecta el tiempo de despliegue por cada liberación.
- La tendencia de la aplicación web es la adopción de la nube.
- La empresa de desarrollo implementa o está por implementar DevOps.

La decisión de continuar con la migración de la aplicación web queda a consideración después de analizar los puntos anteriores e interiorizar si se debe continuar por lo que es necesario que se analice cada punto para no realizar migraciones innecesarias y causar pérdidas en vez de ganancias.

5.2.1 Requisitos de la aplicación web

La aplicación web monolítica en Java que se desea migrar a microservicios debe cumplir con los siguientes requisitos, por lo que de no cumplir con alguno se debe adecuar la aplicación web en donde no se cumpla:

1. Estar basado en web o API monolítica.
2. Implementar el Patrón Modelo Vista Controlador (Subsección 3.1.2.1) a través del Framework Spring MVC o Spring Boot 2.
3. Manejar una estructura estandarizada de URL's.
4. Aplicar patrones de diseño en cada capa de la aplicación (Subsección 3.1.2).
5. Usar Maven o Gradle como herramienta de automatización
6. Contar con los casos de uso y/o historias de usuario utilizados en la aplicación web monolítica
7. Obtener el histórico de cambios realizados en los diferentes módulos de la aplicación web.

5. DISEÑO DEL MÉTODO DE MIGRACIÓN

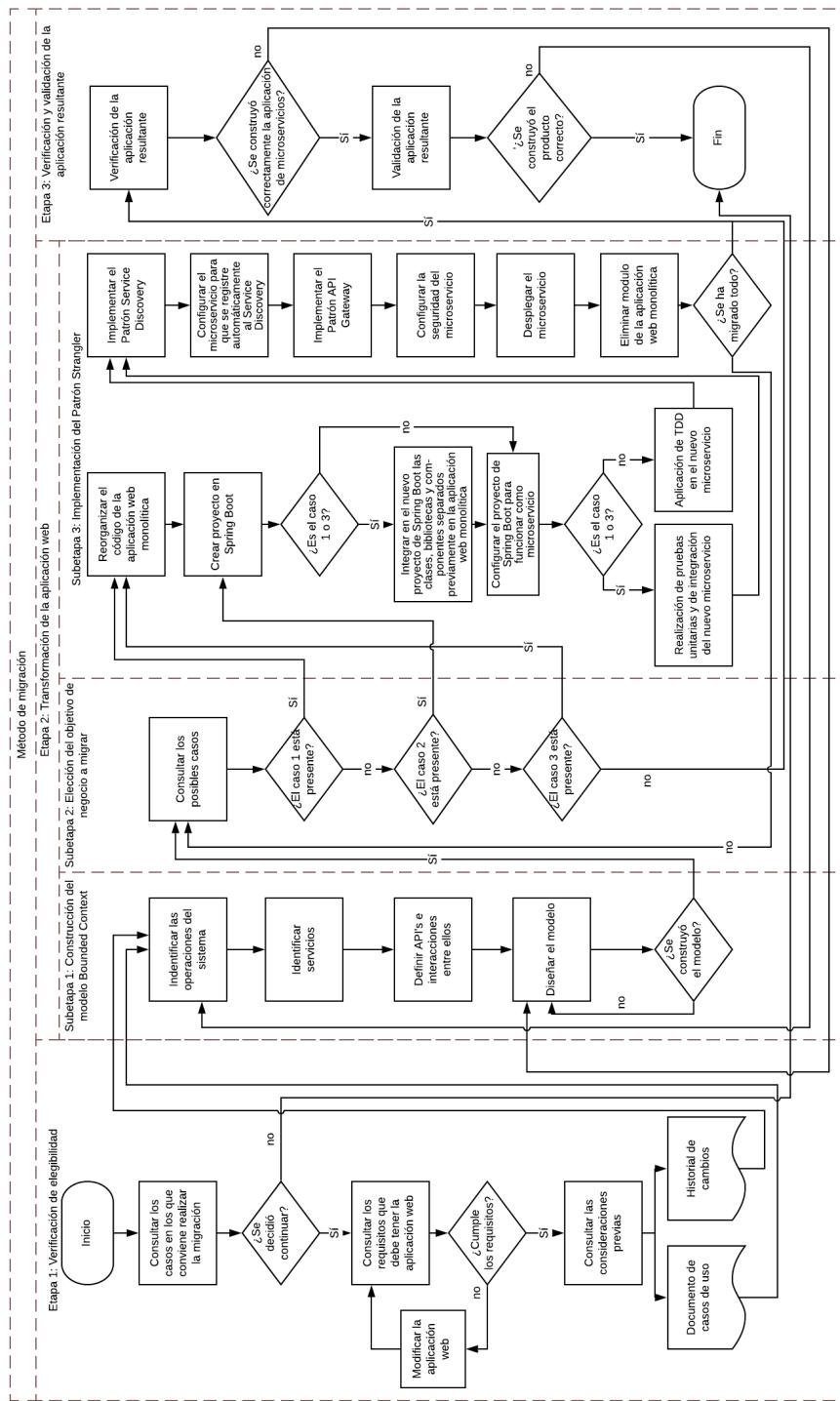


Figura 5.2: Proceso detallado del método de migración propuesto.

5.3 Etapa 2: Transformación de la aplicación web a microservicios

5.2.2 Consideraciones antes de la migración

Antes de comenzar a reestructurar la aplicación web es importante que se tengan en cuenta los siguientes puntos:

- Es recomendable reorganizar los equipos de desarrollo para que cada uno de estos se dedique a un objetivo de negocio.
- Cada equipo encargado de un objetivo de negocio y asegurar que cada microservicio se encuentre bien desarrollado, desplegado y mantenido independientemente.
- Siempre que surja una nueva funcionalidad en los requerimientos éste deberá ser un nuevo microservicio y no un módulo nuevo en la aplicación web monolítica a migrar a fin de minimizar el trabajo de la migración.
- Tener buenas prácticas de desarrollo de software por lo que se debe asegurar que la aplicación web monolítica está libre de antipatrones de diseño (Sección 3.2), y de ser el caso, corregirse antes de comenzar con la migración.

5.3 Etapa 2: Transformación de la aplicación web a microservicios

En esta etapa de la metodología propuesta se propone proporcionar casos posibles que pueden presentarse en la migración y otorgar los pasos a seguir en cada uno de ellos con el objetivo de lograr la migración satisfactoriamente. Por lo tanto, para esta etapa consta de 3 subetapas, las cuales se realizarán iterativamente hasta concluir con una arquitectura de microservicios completamente funcional:

- Subetapa 1: Construcción del modelo Bounded Context
- Subetapa 2: Elección del objetivo de negocio a migrar
- Subetapa 3: Implementación del Patrón Strangler

5.3.1 Subetapa 1: Construcción del modelo Bounded Context

Para la construcción del modelo, será necesario que se cuente con la documentación requerida previamente, para esta subetapa solo será necesario contar con los casos de uso o requerimientos funcionales y el código de la aplicación web.

Se debe realizar un análisis de la arquitectura de la aplicación con base en el Modelo 4+1, estableciendo una organización de todo lo que implica el funcionamiento de ella.

5. DISEÑO DEL MÉTODO DE MIGRACIÓN

5.3.2 Subetapa 2: Elección del objetivo de negocio a migrar

La elección del objetivo de negocio no es una tarea fácil y aunque pareciera que comenzar el proceso de la migración con la parte más pequeña del modelo Bounded Context o al azar es la mejor idea, lo cierto es que probablemente se incurra en una migración insatisfactoria. Por lo que se propone realizarlo de manera iterativa aplicando la estrategia Divide y vencerás, y se debe tener en cuenta que se pueden presentar 3 casos:

- Caso 1: La aplicación web requiere autenticación para todas o algunas transacciones.
- Caso 2: Característica nueva aún no desarrollada.
- Caso 3: Característica con mayores cambios registrado en el histórico de la aplicación.

De los casos presentados previamente, la primera opción que se debe realizar es el caso 1 y cuando ya no se presente más pasar al caso 2 y así sucesivamente hasta terminar la migración.

Caso 1: La aplicación web requiere autenticación para todas o algunas transacciones

Si la aplicación web tiene un módulo de Inicio de sesión conviene comenzar con ese para garantizar la autenticación y autorización de las transacciones que realiza el usuario de manera que se garantice que el usuario que realiza la petición a algún servicio está autenticado. Éste caso es primordial y de presentarse se debe realizar primero, para que en la implementación del Patrón Strangler no haya inconvenientes.

Caso 2: Característica nueva aún no desarrollada

Una característica aún no desarrollada puede construirse desde su concepción como un microservicio y no desarrollarla en la aplicación web monolítica que se quiere migrar para evitar la pérdida de tiempo, dinero y esfuerzo que implicaría su migración inminente.

Caso 3: Característica con mayores cambios registrado en el histórico de la aplicación

Si el caso 2 no está presente o ya se ha realizado entonces la elección de la característica a migrar no es tan trivial y conviene que sea el de mayores cambios porque sabemos que dichos cambios representan un problema de disponibilidad durante los

5.3 Etapa 2: Transformación de la aplicación web a microservicios

despliegues y se quiere garantizar alta disponibilidad. Entonces para determinar la característica a migrar se sugiere realizar una revisión del versionado de código y los casos de uso de las características que el cliente ha solicitado a través del tiempo para elegir el módulo que ha sufrido mayores cambios, aunque si la aplicación web monolítica es demasiado grande y ha sufrido demasiados cambios durante varios años, esto podría determinarse con el análisis de forma incremental año por año en un lapso considerable, por decir 5 años máximo hasta determinar el módulo de mayores cambios.

5.3.3 Subetapa 3: Implementación del Patrón Strangler

Para esta subetapa se propone la implementación del Patrón Strangler (Subsección 3.1.4) con la ayuda del modelo Bounded Context.

5.3.3.1 Fase de transformación:

Para esta fase del patrón se propone una serie de pasos:

1. Reorganización de código en la aplicación web monolítica.
2. Creación del proyecto en Spring Boot.
3. Integración en el nuevo proyecto de Spring Boot las clases, bibliotecas y componentes separados previamente en la aplicación web monolítica.
4. Configuraciones en el proyecto de Spring Boot para funcionar como microservicio.
5. Pruebas unitarias y de integración del nuevo microservicio.
6. Aplicación de TDD en el nuevo microservicio.

Donde la secuencia a seguir va de acuerdo a los casos de la Subsección 5.3.2:

- Si es el caso 1 o 3: Se realizan (1), (2), (3), (4) y (5).
- Si es el caso 2: Se realizan (2), (4) y (6).

1. Reorganización de código en la aplicación web monolítica

En este paso, se debe realizar una reestructuración del código del backend de la aplicación con ayuda del modelo Bounded Context. De manera que se agrupen en paquetes que identifiquen al objetivo de negocio elegido a migrar en el modelo. Además en esta subfase se deben crear clases de ser necesarios. Por ejemplo si la clase controladora posee acciones para diversos objetivos de negocios entonces se debe crear una nueva para manipular solamente el que se eligió previamente, lo mismo para las capas de servicio y datos.

5. DISEÑO DEL MÉTODO DE MIGRACIÓN

2. Creación del proyecto en Spring Boot

Como se quiere migrar a microservicios, es necesario que se cree un nuevo proyecto pero con las características adecuadas, es decir, debe ser un proyecto de Spring Boot. Que ha sido desarrollado para vivir en la nube y posee todas las funcionalidades y características de la arquitectura de microservicios.

3. Integración en el nuevo proyecto de Spring Boot las clases, bibliotecas y componentes separados previamente en la aplicación web monolítica

Para este paso es necesario la creación de un proyecto en Spring Boot 2, en el que se añadirán las clases y bibliotecas previamente separadas de manera interna en la aplicación web y estructurándola con las mismas capas de backend, es decir, capa de servicio, de acceso a datos y controlador.

4. Configuraciones en el proyecto de Spring Boot para funcionar como microservicio

El proyecto creado en Spring Boot requerirá de configuraciones adicionales para que funcione como un microservicio, sin embargo dichas configuraciones son más sencillas a diferencia de Spring MVC, esto se logra con añadir en ocasiones algunas anotaciones de Spring Boot en algunas clases, estas anotaciones permiten a Spring configurar automáticamente los elementos anotados aunque de ser necesario permite la configuración avanzada de forma manual.

Además de las propias configuraciones para el funcionamiento de la aplicación en Spring Boot, las anotaciones permiten la implementación de diversos patrones de diseño para cumplir con los requisitos de la arquitectura de microservicios:

- a) Implementación del Patrón Circuit Breaker.
- b) Implementación del Patrón Client-side load balancing.
- c) Implementación del Patrón Fallback.
- d) Implementación del Patrón Bulkhead.

5. Pruebas unitarias y de integración del nuevo microservicio

5.3 Etapa 2: Transformación de la aplicación web a microservicios

La subfase final consta de las pruebas del microservicio creado, como la mayor parte del código ha sido creado conviene realizar pruebas de la manera tradicional.

6. Aplicación de TDD en el nuevo microservicio

Este paso que se aplica solamente en donde se realiza un microservicio desde cero, es decir, cuando la característica que el cliente desea no existía en la aplicación web por lo que se adecua perfectamente al ciclo de vida del microservicio.

5.3.3.2 Fase de coexistencia:

Una vez construido el microservicio y que se han realizado las pruebas necesarias, se procede a integrarlo al entorno de producción sin que se elimine aún de la aplicación web monolítica. Para este paso es indispensable que haya una capa superior de software que sirva de ruter o proxy entre el nuevo microservicio y la aplicación web monolítica. Por eso se propone aplicar el subproyecto de Spring Cloud como sigue:

1. Implementar el Patrón Service Discovery.
2. Configurar el microservicio para que se registre automáticamente al Service Discovery.
3. Implementar el Patrón API Gateway.
4. Configurar la seguridad del microservicio.
5. Desplegar el microservicio.

De los pasos anteriores, se aplican siguiendo lo siguiente:

- Si es el primer microservicio de la migración: Se realizan (1), (2), (3), (4) y (5).
- Si no: (2), (4) y (5).

5.3.3.3 Fase de eliminación:

En esta fase del Patrón Strangler se debe eliminar el objetivo de negocio que se migró a microservicio, aunque solo aplique para los casos 1 y 3 de la Subsección 5.3.2. Para efectuar este paso a través del enrutamiento de Spring Cloud se puede desplegar la aplicación web monolítica sin el objetivo de negocio que se migró mientras la otra sigue ejecutándose, y al terminar el despliegue se puede cambiar el enruteamiento hacia éste. Cada que se va eliminando un objetivo de negocio de la aplicación web monolítica, el entorno de microservicios va evolucionando y va adquiriendo los beneficios de esta arquitectura.

5. DISEÑO DEL MÉTODO DE MIGRACIÓN

5.4 Etapa 3: Verificación y validación de la aplicación resultante

En la verificación y validación se asegura que la aplicación cumple con las características de una arquitectura de microservicios y además que cumple las expectativas iniciales, es por eso que se proporcionan los siguientes valúaciones en las tablas 5.1 y 5.2 :

Verification
¿Cada microservicio cumple con un objetivo de negocio?
¿Funcionan de forma independiente?
¿Cada microservicio se puede escalar rápidamente?
¿Se implementan los patrones de Microservicios?
¿Interactúan correctamente los microservicios?

Tabla 5.1: Verificación de la aplicación resultante

Es importante que todos los puntos de la tabla 5.1 respondan a un Sí, y en el caso de que alguno de ellos sea No, se debe regresar a la subsección 5.3.1 en la subfase del Diseño del modelo.

Validación
¿La orquestación de microservicios mantiene las funcionalidades de la aplicación web?
¿Todo el ambiente de microservicios se adecúa a los casos de uso?

Tabla 5.2: Validación de la aplicación resultante

Para los resultados de la tabla 5.2 deben responder también a un Sí, en caso contrario se debe analizar desde la subsección 5.3.1 en la subfase de identificar las operaciones del sistema.

Si todo lo anterior resulta en satisfactorio entonces se garantiza que la aplicación funciona como debería y además aprovecha las ventajas de la arquitectura de microservicios.

CAPÍTULO
6

Implementación del método propuesto

Este capítulo trata sobre la implementación del método propuesto para la migración de aplicaciones web en Java a la arquitectura de microservicios para probar la corrección. Por lo que se proporciona una aplicación desarrollada en Java con el Framework de Spring MVC realizada por un tercero para evitar errores de sesgo.

6.1 Método

El software que se utilizará como prueba del método fue construido siguiendo las especificaciones del sitio web o7planning [5] y cumple con todos los requisitos de una aplicación web, esta se llama Online Shop y es un carrito de compras.

Para comenzar con el método se va a suponer que la aplicación anteriormente mencionada se está ejecutando en 3 servidores, además ahora tiene una afluencia mucho mayor que desde sus inicios, es decir, la aplicación web ha tenido éxito vendiendo sus productos que se ha ido reconociendo en otras partes del mundo lo que significa que con ese crecimiento hayan constantes fallas de disponibilidad a causa de caídas del sistema por la alta carga de usuarios conectados al mismo. Además de lo antes mencionado, la empresa de desarrollo se está capacitando en DevOps porque no solo el cliente dueño de Online Shop sino también los dueños de otros sistemas de software también han tenido un buen crecimiento y constantemente solicitan cambios en sus sistemas, cambios que abarcan desde adición de características como la

6. IMPLEMENTACIÓN DEL MÉTODO PROPUESTO

eliminación de otras y piden que esos cambios realizados se desplieguen en producción lo más rápido posible, aunado a eso, como en todo ciclo de vida del software se realizan correcciones de errores en el código. Todos estos cambios impactan en las ganancias de las empresas dueñas de los software por el tiempo que pasan sin otorgar su servicio a los usuarios.

CAPÍTULO
7

Conclusiones y trabajo futuro

En este capítulo se presentan las conclusiones de la tesis y el trabajo futuro que conlleva.

7.1 Conclusiones

El objetivo fundamental de esta tesis era abordar el problema de los desarrollos de aplicaciones web en arquitectura monolítica que necesitan adaptarse a las nuevas técnicas de desarrollo como DevOps, necesitan alta escalabilidad, disponibilidad o su objetivo principal es la adopción de la Nube pero la inversión para realizar el cambio es alta debido a que debe hacerse desde cero o aventurarse a migrarlo como sea pudiendo tener migraciones insatisfactorias con pérdidas de tiempo, dinero y esfuerzo.

Así pues, la aportación principal de esta tesis consiste en el diseño de un método que proporcione una visión general de lo que se debe cubrir para lograr una migración satisfactoria para cualquier aplicación web monolítica con la menor pérdida en disponibilidad durante el proceso de migración aplicando el Patrón Strangler (Subsección 3.1.4). Se ha elegido éste patrón de migración, porque lo que se quiere lograr es que exista la menor pérdida posible en todos los aspectos, y éste patrón soluciona un problema de disponibilidad durante la migración, es decir, se asegura que la aplicación web no se desconecte completamente ya que un sistema caído no genera ganancias.

7. CONCLUSIONES Y TRABAJO FUTURO

7.2 Trabajo futuro

El presente trabajo deja abierto un amplio campo de investigación que por el poco tiempo y recursos no se pudo abarcar:

- Implementación completa de la metodología propuesta en otros proyectos de aplicaciones web monolíticas empresariales que cumplan con los requisitos para confirmar su efectividad.
- Integración de Microfrontends en la metodología.
- Agregar a la metodología de migración cómo migrar explícitamente bases de datos SQL a NoSQL.
- Añadir configuraciones avanzadas en las diferentes implementaciones de los patrones de diseño.

Glosario

B

Big Data

Término utilizado para definir el procesamiento de grandes volúmenes de datos.. 1

Bounded Context

Define o delimita el contexto en donde un modelo puede aplicar, es decir, establece los límites en términos de organización, uso con partes específicas de la aplicación, y manifestaciones físicas como código base y esquemas de bases de datos.[40]. 56, 58, 63–65

D

Desarrollo y Operación

Es una filosofía para realizar entregas, despliegues y supervisión continua de las aplicaciones.. 1, 20

E

Enterprise JavaBeans

API de Java Enterprise Edition que se utiliza para simplificar el desarrollo de aplicaciones empresariales funciona encapsulando la lógica de negocios.. 31

F

Glosario

Framework

Entorno de trabajo que define una estructura en concreto para facilitar las tareas.. 2–4, 20, 51, 53, 61, 69

I

Interfaz de Nombrado y Directorio Java

Es una API de Java para servicios de directorio. [9]. 30

Internet de las Cosas

Es un término que se refiere a que todas las cosas estén conectados a internet.. 1

M

Máquina Virtual de Java

Es un componente especial que se encuentra entre el nivel del sistema operativo y el código compilado o Byte Code y se encarga de ejecutarlo.. 47

N

Nube

Es un término utilizado para definir una red de servidores, cada uno con una función única y supuesto a operar como un ecosistema sencillo, diseñados para almacenar datos, ejecutar aplicaciones o entregar contenido, todo accesible desde cualquier dispositivo con acceso a internet.[17]. 1, 49, 71

P

panacea

Remedio de todos los males.. 17

pila

Es una estructura de datos que permite almacenar y recuperar datos de modo que el último en entrar es el primero en salir. 21, 76

Protocolo de Control de Transmisión/Protocolo de Internet

Es un modelo de referencia que se basa en una pila de protocolos independientes distribuidos en cuatro capas, la de aplicación, transporte, interred y enlace. [66].
5

Protocolo de Tranferencia de Hipertexto

Es el protocolo de comunicación que permite las transferencias de información en la World Wide Web. 5, 14

proxy

Es un equipo dentro de la red, entre un servidor y un equipo cliente cuya función es almacenar los objetos recientes solicitados por el cliente para mostrarlo más rápido cuando se requiera nuevamente.. 32

R

Requerimientos funcionales

Son las acciones que el software debe hacer y generalmente se encuentran en forma de casos de uso o historias de usuario.. 56, 58

Requerimientos no funcionales

Son los requerimientos que tienen que ver con la calidad del servicio proporcionado por la aplicación web como escalabilidad, disponibilidad, etc.. 56, 58

S

software

Conjunto de programas o instrucciones en una computadora que se ejecutan para realizar tareas.. 2

Spring MVC

Es un módulo del framework Spring que facilita el desarrollo de aplicaciones web proporcionando un diseño Modelo-Vista-Controlador.[15]. 2-4, 61, 69

V

Glosario

volcado de pila

Es una lista de llamadas de métodos que esa aplicación realizó en cierto punto del tiempo durante su ejecución y reporta los errores que han ocurrido, advertencias u otra información relevante, esto en forma de pila. 21

Acrónimos

C

CRUD

Create, Read, Update, Delete. 8

D

DAO

Data Access Object. 33, 38

DDD

Domain-Driven Design. 56

DevOps

Development and Operation. 1, 3, 17, 61, 69, 71

E

EJB

Enterprise JavaBeans. 31

ESB

Enterprise Service Bus. 16, 18

H

HTTP

HyperText Transfer Protocol. 5

Acrónimos

I

IoT

Internet of Things. 1

J

JNDI

Java Naming and Directory Interface. 30

JVM

Java Virtual Machine. 47

S

SME

Situational Method Engineering. 59

SOA

Service Oriented Architecture. 15–19

T

TCP/IP

Transport Control Protocol/Internet Protocol. 5

TDD

Test-Driven Development. 65, 67

Bibliografía

- [1] Aspectos básicos de las aplicaciones web. <https://helpx.adobe.com/mx/dreamweaver/using/web-applications.html>.
- [2] bliki: Circuitbreaker. <https://martinfowler.com/bliki/CircuitBreaker.html>.
- [3] Continuos obsolescence. <https://sourcemaking.com/antipatterns/continuous-obsolescence>.
- [4] Core j2ee patterns - composite view. <https://www.oracle.com/technetwork/java/compositeview-137722.html>.
- [5] Create a java shopping cart web application using spring mvc and hibernate.
- [6] Cut-and-paste programming. <https://sourcemaking.com/antipatterns/cut-and-paste-programming>.
- [7] Design patterns and refactoring. <https://sourcemaking.com/antipatterns>.
- [8] Latest news. <https://www.tiobe.com/tiobe-index>. Acceso: Nov. 18, 2018.
- [9] Lesson: Overview of jndi. <https://docs.oracle.com/javase/tutorial/jndi/overview/index.html>.
- [10] Microservices pattern: Api gateway pattern. <https://microservices.io/patterns/apigateway.html>.
- [11] Microservices pattern: Circuit breaker. <https://microservices.io/patterns/reliability/circuit-breaker.html>.
- [12] Microservices pattern: Server-side service discovery pattern. <https://microservices.io/patterns/server-side-discovery.html>.

BIBLIOGRAFÍA

- [13] Reinvent the wheel. <https://sourcemaking.com/antipatterns/reinvent-the-wheel>.
- [14] Spaghetti code. <https://sourcemaking.com/antipatterns/spaghetti-code>.
- [15] Spring mvc tutorial - javatpoint. <https://www.javatpoint.com/spring-mvc-tutorial>.
- [16] Swiss army knife. <https://sourcemaking.com/antipatterns/swiss-army-knife>.
- [17] What is the cloud - definition: Microsoft azure. <https://azure.microsoft.com/en-us/overview/what-is-the-cloud/>.
- [18] The java ee 5 tutorial. <https://docs.oracle.com/javaee/5/tutorial/doc/bnaay.html>, Sep 2007.
- [19] The opensessioninview antipattern. <https://blog.frankel.ch/the-opensessioninview-antipattern/>, Nov 2010.
- [20] Business delegate pattern - core j2ee patterns. <https://www.dineshonjava.com/business-delegate/>, Jan 2018.
- [21] Front controller design pattern - core j2ee patterns. <https://www.dineshonjava.com/front-controller-design-pattern/>, Jan 2018.
- [22] J2ee design pattern : Integration tier patterns : Service activator design pattern. <https://www.javaskool.com/service-activator-design-pattern/>, Jan 2018.
- [23] Session facade pattern - core j2ee patterns. <https://www.dineshonjava.com/session-facade/>, Jan 2018.
- [24] View helper design pattern - core j2ee patterns. <https://www.dineshonjava.com/view-helper-design-pattern/>, Jan 2018.
- [25] Front controller design pattern. <https://www.geeksforgeeks.org/front-controller-design-pattern/>, Apr 2019.
- [26] A BETTER, D. B. The continuous obsolescence anti-pattern. https://gb.ivoox.com/en/the-continuous-obsolescence-anti-pattern-audios-mp3_rf_35893837_1.html?autoplay=true, May 2019.

BIBLIOGRAFÍA

- [27] AGARWAL, R. Why use microservices? - dzone microservices. <https://dzone.com/articles/why-microservices>, Aug 2018.
- [28] ALEXANDER, C., ISHIKAWA, S., AND SILVERSTEIN, M. *A pattern language: towns, buildings, construction*. Oxford Univ. Pr., 1977.
- [29] ALUR, D., CRUPI, J., MALKS, D., BOOCH, G., AND FOWLER, M. *Core J2EE patterns: best practices and design strategies*. Prentice Hall PTR, 2007.
- [30] ARCHIVEDDOCS. Page controller. [https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff649595\(v=pandp.10\)](https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff649595(v=pandp.10)).
- [31] ARDALIS. Arquitecturas de aplicaciones web comunes. <https://docs.microsoft.com/es-es/dotnet/standard/modern-web-apps-azure-architecture/common-web-application-architectures>.
- [32] ASALE, R. monolito. <https://dle.rae.es/?id=PgcPa01>.
- [33] BALALAIE, A., HEYDARNOORI, A., JAMSHIDI, P., TAMBURRI, D. A., AND LYNN, T. Microservices migration patterns. *Software: Practice and Experience* (2018).
- [34] BEHARA, S. Monolith to microservices using the strangler pattern - dzone microservices. <https://dzone.com/articles/monolith-to-microservices-using-the-strangler-patt>, Dec 2018.
- [35] BEHARA, S. Breaking the monolithic database in your microservices architecture - dzone microservices. <https://dzone.com/articles/breaking-the-monolithic-database-in-your-microserv>, Jun 2019.
- [36] CHAFFEE, A., AND CHAFFEE, A. Counting tiers. <https://thesismicroservices19.page.link/xhKP>, Jan 2000.
- [37] CRAWFORD, W., AND KAPLAN, J. *J2EE design patterns*. OReilly, 2003.
- [38] DAYA, S., VAN DUY, N., EATI, K., FERREIRA, C. M., GLOZIC, D., GUCCER, V., AND GUPTA, M. *Microservices from Theory to Practice Creating Applications in IBM Bluemix Using the Microservices Approach*, 1st ed. IBM Corporation, 2015.

BIBLIOGRAFÍA

- [39] DESPODOVSKI, R. Microservices vs. soa – is there any difference at all? - dzone integration. <https://dzone.com/articles/microservices-vs-soa-is-there-any-difference-at-al>, Nov 2017.
- [40] EVANS, E. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley, 2003.
- [41] FADATARE, R. Application controller design pattern in java. <https://www.javaguides.net/2018/08/application-controller-design-pattern-in-java.html>, Nov 2018.
- [42] FADATARE, R. Composite view design pattern in java. <https://www.javaguides.net/2018/08/composite-view-design-pattern-in-java.html>, Nov 2018.
- [43] FADATARE, R. Context object design pattern in java. <https://www.javaguides.net/2018/08/context-object-design-pattern-in-java.html>, Nov 2018.
- [44] FOWLER, M. Inversion of control containers and the dependency injection pattern. <https://martinfowler.com/articles/injection.html#UsingAServiceLocator>.
- [45] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design patterns elements of reusable object oriented software*. Addison Wesley, 1995.
- [46] KALSKE, M., MÄKITALO, N., AND MIKKONEN, T. Challenges when moving from monolith to microservice architecture. In *Current Trends in Web Engineering* (Cham, 2018), I. Garrigós and M. Wimmer, Eds., Springer International Publishing, pp. 32–47.
- [47] KAPPAGANTULA, S. Microservice architecture - learn, build, and deploy applications - dzone microservices. <https://dzone.com/articles/microservice-architecture-learn-build-and-deploy-a>, Nov 2018.
- [48] KAYAL, D. *Pro Java EE Spring Patterns: best practices and design strategies implementing Java EE patterns with spring framework*. Apress, 2008.
- [49] KRESS, J., MAIER, B., NORMANN, H., SCHMEIDEL, D., SCHMUTZ, G., TROPS, B., UTSCHIG-UTSCHIG, C., AND WINTERBERG, T. Enterprise service bus. <https://www.oracle.com/technetwork/articles/soaind-soa-esb-1967705.html>, Jul 2013.

BIBLIOGRAFÍA

- [50] KRUCHTEN, P. Architectural blueprints—the “4 1” view model of software architecture. <https://www.cs.ubc.ca/~gregor/teaching/papers/41view-architecture.pdf>, Nov 1995.
- [51] KUMAR, A. Microservice architecture: is it right for your software development. <https://dzone.com/articles/microservice-architecture-is-it-right-for-your-soft>, Aug 2018.
- [52] LEWIS, J., AND FOWLER, M. Microservices. <https://martinfowler.com/articles/microservices.html>, Mar 2014.
- [53] MAPPLE, S., AND BINSTOCK, A. Jvm ecosystem report 2018, Oct 2018.
- [54] MEDY, SAMEER, JANE, PANKAJ, KUMAR, H., AND CHAKRAVARTHI. Dao design pattern. <https://www.journaldev.com/16813/dao-design-pattern>, Mar 2019.
- [55] MIKEWASSON. N-tier architecture style - azure application architecture guide. <https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/n-tier>.
- [56] MIRI, I. Microservices vs. soa - dzone microservices. <https://dzone.com/articles/microservices-vs-soa-2>, Jan 2017.
- [57] PARMAR, K. Monolithic vs microservice architecture - dzone integration. <https://dzone.com/articles/monolithic-vs-microservice-architecture>, May 2016.
- [58] PATZER, A. Foundations of jsp design patterns: The view helper pattern. <https://thesismicroservices19.page.link/qLo3>, Nov 2004.
- [59] POWER, V., AND SKOWRONSKI, J. Most popular java web frameworks - dzone java. <https://dzone.com/articles/most-popular-java-web-frameworks>, Feb 2018.
- [60] RICHARDSON, C. *Microservice Patterns: with examples in Java*. Manning, 2019.
- [61] SCHULDT, H. *Multi-Tier Architecture*. Springer US, Boston, MA, 2009, pp. 1862–1865.

BIBLIOGRAFÍA

- [62] SCOTT, J. A. *A Practical Guide to Microservices and Containers: Mastering the Cloud, Data, and Digital Transformation*. Compliments of Mars Data Technologies, 2017.
- [63] SHKLAR, L., AND ROSEN, R. *Web application architecture: principles, protocols, and practices*. Wiley, 2006.
- [64] SMITH, S., WENZEL, M., A., A., JENKS, A., AND JAKOBSEN, H. Common web application architectures. <https://docs.microsoft.com/en-us/dotnet/standard/modern-web-apps-azure-architecture/common-web-application-architectures>, Jan 2019.
- [65] STRINGFELLOW, A. What is n-tier architecture? - dzone devops. <https://dzone.com/articles/what-is-n-tier-architecture>, May 2017.
- [66] TANENBAUM, A. S., AND WETHERALL, D. J. *Redes de computadoras*. Pearson Educación, 2012.
- [67] WALLS, C. *Spring in Action*. Manning Publications Co., 2019.