UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

ACTIVIDADES PROFESIONALES EN MIMONI (CASHAHORA S.A. DE C.V.)

INFORME DE ACTIVIDADES PROFESIONALES

Que para obtener el título de

Ingeniería Mecatrónica

P R E S E N T A

Alfonso Lizárraga Santos

ASESOR DE INFORME

M. en I. Billy Arturo Flores Medero Navarro

Ciudad Universitaria, Cd. Mx., 2016

Contenido

Introducción

El presente reporte establece las actividades por mí realizadas como parte de la compañía Mimoni (formalmente Cashahora, S.A de C.V) durante el período del 20 de Abril del 2015 al 1° de Abril del 2016. El título del puesto desempeñado durante este periodo es "Lead Architect", traducido como "Arquitecto Líder de Software" dentro del área de Desarrollo y Sistemas. El foco de mis funciones es dirigir y coordinar el mantenimiento del sistema de administración interna actual de la compañía, considerado un sistema obsoleto, y a la vez diseñar y definir la arquitectura del nuevo sistema que reemplazará al actual.

Este nuevo sistema utilizará tecnologías innovadoras y ágiles, culminando en un sistema empresarial flexible, moderno, altamente configurable y de fácil ampliación. En el margen de dichas responsabilidades se llevaron a cabo una gran cantidad de proyectos paralelos, mismos que están descritos en el presente reporte.

La estructura del documento comienza estableciendo la historia y características de la compañía, con enfoque en el departamento donde me desempeñé profesionalmente. Después, se introducen varios de los términos utilizados a lo largo del reporte, principalmente los nombres y descripciones de las tecnologías evaluadas y utilizadas para la construcción del nuevo sistema núcleo de la compañía como los demás proyectos relacionados, así como las herramientas y servicios utilizados para la puesta en marcha y configuración del mismo.

A continuación se menciona el primero de los proyectos de mayor envergadura: el desarrollo del plan general de infraestructura en la nube para todos los proyectos de software internos de la empresa. Éste define las estrategias y configuraciones necesarias de Amazon Web Services para desplegar servicios web escalables, resistentes a fallos y de alta disponibilidad. Para que un servicio web alcance esta denominación, debe poder crecer fácilmente su infraestructura para responder a aumentos fuertes de peticiones de servicio (escalabilidad), debe poder seguir funcionando a pesar de que alguno de los nodos de los que se compone el sistema falle o se vea interrumpido (resistencia a fallos), y finalmente que pueda reaccionar automáticamente a cambios o fallas dentro del sistema de tal manera que siempre esté disponible para procesar solicitudes (alta disponibilidad).

Acto seguido, se expone el segundo proyecto: las integraciones de software realizadas con socios externos a la compañía, que nos permiten realizar consultas de historiales crediticios o bien realizar procesos de dispersión monetaria a los clientes. En esta sección se omiten los detalles de implementación y algunos detalles de configuraciones de equipos y de funcionamiento de los sistemas externos, puesto que no es posible divulgar tal información sin el consentimiento de las compañías involucradas debido a que son considerados datos sensibles.

Para concluir se presenta el tercer y más grande proyecto: el diseño de la arquitectura, el desarrollo y lanzamiento del sistema nuevo de administración interna de la empresa. Este sistema consumió una gran cantidad de tiempo dentro de mi estancia en la empresa y significó un gran cambio para todas las áreas

cuyas funciones fueron trasladadas a esta nueva plataforma. Por estas razones, este proyecto puede considerarse como el más importante de todos los realizados como parte de mis actividades profesionales dentro de la empresa.

Descripción de la empresa

Historia de la empresa

Mimoni es una empresa mexicana, creada a partir de las lecciones aprendidas durante su predecesora, una empresa llamada Micel (http://www.micel.mx/nosotros.html) que empezó en Mayo del 2008 proveyendo préstamos a compradores de teléfonos celulares. En un inicio los préstamos estaban pensados como una ayuda para que las personas sin teléfono celular pudieran adquirir uno, y poco a poco fue cambiando esta percepción. Eventualmente, gracias a los estudios de mercado realizados y a los primeros clientes con los que contó la compañía, vieron que muchos de ellos no estaban tan interesados en poseer el teléfono físico, sino que buscaban la manera de poderse comunicar de una forma más accesible. Así es como la empresa giró su mercado objetivo y su propuesta de valor y comenzó a ofrecer préstamos de celulares con un paquete de telefonía bastante innovador para el momento (300 minutos, 60 SMS, 150 MB), con lo que los clientes rápidamente vieron el valor de la propuesta y prefirieron tener un teléfono en modalidad de renta con plan incluido que ahorrar el dinero necesario para comprar un Smartphone de las características que se tenían.

Eventualmente, en Agosto del 2013 se tomó la decisión de transicionar a préstamos ya no de celulares, sino de dinero. Las condiciones del mercado de telefonía y comunicación móvil habían cambiado mucho en esos años, y la propuesta de valor de la empresa ya no correspondía con algo que el mercado pidiera o necesitara, además de ya encontrarse por debajo de lo que las empresas ya establecidas de telefonía podían ofrecerle a sus clientes. Fue en este momento que se suspendieron todas las actividades de promoción y marketing de Micel, para comenzar las actividades propias de Mimoni.

Hasta la fecha, la estrategia de cambio de oferta ha funcionado bastante bien, atrayendo inversiones fuertes de diversos grupos financieros, quienes se han visto atraídos por el modelo de negocio de los préstamos de dinero en línea y de respuesta rápida: un cliente de Mimoni no debe esperar más de 1 día para obtener al menos la aprobación de su préstamo, sin necesidad de acudir a ninguna sucursal o tener comprobantes de ingreso o cuenta bancaria. Todos los datos necesarios para decidir si se extiende el préstamo se adquiere en el momento en el que el cliente completa los datos de su solicitud. Posteriormente se consulta su historial crediticio y todo el historial y análisis previo de características de los clientes que pagan en tiempo y forma, datos que fueron adquiridos desde que inició Micel, hace 8 años.

Misión

Mimoni es la empresa de préstamos por internet más grande del país enfocada en facilitar a sus miembros el acceso a los beneficios del sistema financiero formal.

Visión

Crear un proceso principalmente estadístico y tecnológico que pueda, de manera instantánea y precisa, predecir el riesgo crediticio para el 60% de la población latinoamericana con ingresos estables pero sin acceso fácil a créditos personales o de consumo.

Filosofía y Valores

- **Integridad:** Actuamos con ética, transparencia, honestidad y lealtad.
- **Seguimos las reglas.** Somos responsables de nuestras acciones.
- **Respeto:** Valoramos las contribuciones de los demás. Tomamos decisiones basados en los hechos, no en la jerarquía. No discriminamos.
- **Pasión:** somos positivos. Estamos comprometidos con la ejecución y los resultados. Somos innovadores y creativos. Somos entusiastas.

Organigrama

En la ilustración 1 se describe el organigrama de la empresa, enfocado principalmente en el área de desarrollo y análisis, que es donde se desarrollaron mis actividades profesionales. Debido al tamaño de la empresa (300 empleados, aproximadamente), es poco práctico entrar a profundidad en las demás áreas.

Glosario de términos

ERP[1]: Siglas de *Enterprise Resource Planning*, en español descritos como Sistemas de Planeación de Recursos Empresariales, se refieren al software de información que permiten a las empresas manejar e integrar los procesos de negocio propios de las compañías, en específico integrando los servicios de planeación de productos, ventas, manufactura, envíos, mercadotecnia, inventarios, pagos, facturación, etc.

[1] Wikipedia (2001-2016). "Enterprise Resource Planning" [En línea]. Disponible en https://en.wikipedia.org/wiki/Enterprise_resource_planning [2016, Abril]

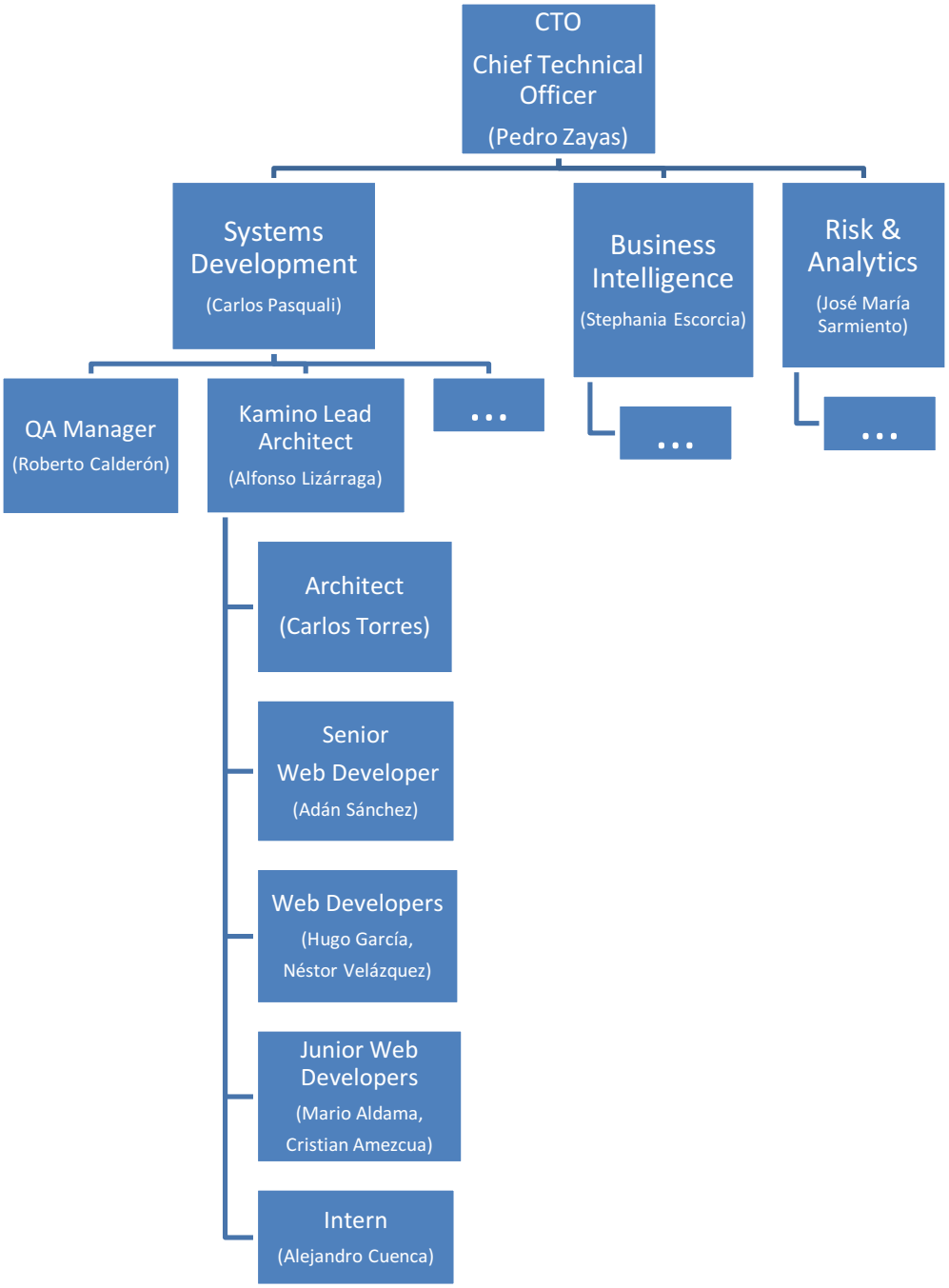


Ilustración 1: Organigrama

CRM[2]: Siglas de Customer *Relationship Management*, o Administración de la Relación con los Clientes, se refiere al software utilizado por empesas para gestionar el contacto con los clientes, desde las ventas, atención al cliente, seguimiento de compra, evaluación del servicio, etc.

AWS[3]: Siglas de *Amazon Web Services*, un servicio de Infraestructura bajo demanda (IaaS, *o Infrastructure as a Service*) que provee Amazon para que otras empresas o individuos renten espacio/procesamiento/servidores dentro de la infraestructura tecnológica existente de Amazon, ofreciendo cobro bajo demanda y asignación dinámica de equipo de cómputo según requieran sus clientes.

REST[4]: siglas de *REpresentational State Transfer*, o Transferencia de Estado Representacional, es una arquitectura para desarrollo de sistemas distribuidos, principalmente de Internet.

SOAP[5]: siglas de *Simple Object Access Protocol*, o Protocolo de Acceso Simple de Objetos, es un protocolo de intercambio de objetos entre dos sistemas distribuidos.

.NET Framework[6]: Es una colección de librerías de código desarrollado por Microsoft, ejecutadas principalmente en ambientes Windows. Entre sus componentes están una enorme librería de clases, conocida como el *Framework Class Library* (FCL), y un ambiente de ejecución común (CLR), que permite que código desarrollado en uno de los lenguajes de programación soportados (C#, Visual Basic, F#), pueda ejecutar código desarrollado en otro lenguaje soportado.

Python[7]: Es un lenguaje de programación interpretado de alto nivel, dinámicamente tipado y multiplataforma, que permite el desarrollo de acuerdo de diversos paradigmas de programación (orientado a objetos, funcional, imperativa). Se distribuye bajo una licencia compatible con la Licencia Pública de GNU (Código abierto), y apareció en 1991.

Django[8]: Es una colección de librerías y rutinas que permite realizar desarrollos web de manera sencilla y rápida con el lenguaje Python. Posee un sistema de plugins que permite que otros desarrolladores ofrezcan paquetes que den más funcionalidad al sistema base, y es una de las librerías más utilizadas para el desarrollo web en Python.

---

[2] Wikipedia (2001-2016). "Customer Relationship Management" [En línea]. Disponible en https://en.wikipedia.org/wiki/Customer_relationship_management [2016, Abril]
[3] Amazon (2016). *"Amazon Web Services"* [En línea]. Disponible en https://aws.amazon.com/es/ [2016, Abril]
[4] Fielding, Roy (2000). "*Architectural Styles and the Design of Network-based Software Architectures"* [En línea]. Disponible en https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm [2016, Abril]
[5] Daigneau, R. (2012). *Service Design Patterns.* Massachusetts: Addison-Wesley.
[6] Microsoft (2016). ".NET Framework" [En línea]. Disponible en https://www.microsoft.com/net/default.aspx [2016, Abril]
[7] The Python Software Foundation (2016). "Python" [En línea]. Disponible en https://www.python.org/ [2016, Abril]
[8]Django Software Foundation (2016). "Django: The web framework for profesionists with deadlines" [En línea]. Disponible en https://www.djangoproject.com/ [2016, Abril]

AngularJS[9]: Es una colección de librerías de código abierto desarrollada por Google y mantenida por diferentes empresas e individuos para el lenguaje de programación Javascript. Su objetivo principal es hacer más sencillo el desarrollo de aplicaciones de una sola página (SPA), ejecutando la lógica detrás del sitio en el navegador antes que en el servidor remoto.

Descripción del puesto de trabajo

El título del puesto en el que se desarrollaron mis actividades profesionales es "*Lead Software Architect*". La función principal de un arquitecto de software consiste en diseñar a alto nivel el software a construir, desde la selección de lenguajes, librerías, patrones de diseño y estructura de los proyectos, hasta módulos de conexión con sistemas existentes del negocio. Por esto, es necesario conocer a fondo los requerimientos para extender cualquiera de los sistemas existentes, cómo funciona cada uno y cómo funcionan en conjunto, proponer planes de contingencia, mantenimiento, desarrollo, mejora continua, infraestructura, metodologías de desarrollo y de lanzamiento.

Además de estas funciones, a lo largo de mi desempeño en la empresa realicé actividades complementarias, como:

- Desarrollo de proyectos: desde obtener los requerimientos de los clientes internos.
- Hacer estimaciones de tiempos, esfuerzos y recursos.
- Planear los diferentes proyectos y etapas de desarrollo de acuerdo a las metodologías de planeación de proyectos con enfoque en los KPI (*Key Performance Indicator*, o Indicadores Clave de Rendimiento) de la empresa, así como definir con los demás equipos de desarrollo la conexión con sus respectivos sistemas, definiendo claramente las dependencias entre plataformas y equipos.
- Detalles de autenticación y autorización, es decir, quién y desde qué sistema puede realizar qué operaciones, así como monitoreo y registro de actividad de cada usuario.
- Adiciones a reportes institucionales existentes en cada cambio fuerte: todos los datos nuevos que se agreguen a un proceso deben poderse visualizar en los reportes gerenciales, permitiendo así cuantificar la diferencia entre el proceso o estrategia anterior y la nueva, para la toma de decisiones de negocio de manera informada y precisa.

En el caso particular de esta empresa, mis funciones se centraron en dos sistemas: un sistema obsoleto y monolítico, construido hacía más de 5 años para Micel, y que después fue adaptado para uso de Mimoni, de nombre "Kamino", y "Korriban", el sistema nuevo en cuya arquitectura y desarrollo fui el principal responsable, y cuyo objetivo es reemplazar Kamino y su tecnología obsoleta (fue creado con las librerías

[9] Google (2016). "AngularJS" [En línea]. Disponible en https://angularjs.org/ [2016, Abril]

.NET del 2008 hacia atrás) con un diseño modular, innovador, y que se apegara lo más posible a la arquitectura de microservicios, con interfaces *RESTful* para exponer su información a través de servicios web, y construido para ser fácil de mantener y optimizar, puesto que al estar dividido en módulos se puede cambiar uno de ellos a la vez sin afectar al resto.

Estas funciones no eran realizadas de manera individual, sino que contaba con un equipo especializado en cada sistema y en lenguajes de programación diferentes, de tal manera que ellos son quienes ejecutan el plan trazado.

El equipo consistía de:

- Carlos Torres: Arquitecto de Kamino (.NET)
- Mario Aldama: Desarrollador Web (.NET), IPN
- Cristian Amezcua: Desarrollador Web (.NET) Jr, IPN
- Víctor García: Desarrollador Web (.NET), BUAP
- Adán Sánchez: Desarrollador Web Full-Stack (Python y AngularJS) Sr, UV
- Néstor Velázquez: Desarrollador Web Full-Stack (Python y AngularJS), IPN
- Alejandro Cuenca: Becario de desarrollo Web (Python), IPN

Dado que las fortalezas del equipo estaban orientadas principalmente al desarrollo de software, mi rol dentro de él fue uno de liderazgo y de PTM (*Project Technical Manager*), antes que de desarrollo.

Introducción a las actividades realizadas

Para una mejor comprensión de las actividades realizadas, se presenta una breve descripción de las principales tecnologías, arquitecturas, configuraciones, alternativas, entre otros temas que ayudarán a una plena comprensión de los proyectos realizados.

SOA (*Service Oriented Architecture*)

El término de Arquitectura Orientada a Servicios es muy recurrente en la literatura y en las recomendaciones de arquitectos de software que crean aplicaciones para grandes corporativos y clientes *enterprise*: es una arquitectura abierta, extensible, federada y con posibilidad de composición que promueve orientación a servicios y se compone de servicios autónomos, con soporte para cambios en calidad de servicio (QoS), de diferentes proveedores, interoperables, descubribles y potencialmente reutilizables, implementados como servicios web. En los grandes corporativos, es normal que a lo largo de la vida de la empresa se vaya comprando software para una tarea específica, y que estos sistemas se mantengan en operación por un largo periodo de tiempo. Esto significa que, además de ser productos heterogéneos, muchos de ellos fueron realizados con herramientas obsoletas, lenguajes obsoletos, o que exigen que la conexión a ellos sea mediante protocolos obsoletos.

Un servicio es una funcionalidad de interés, reutilizable y atómica, ofrecida por un sistema para su uso por otros sistemas que requieran dicha funcionalidad sin la necesidad de saber los detalles internos de su implementación. En ocasiones, un servicio o conjunto de ellos expuestos por un mismo sistema se puede llamar API (*Application Programming Interface*), dado que es la puerta de acceso del mundo a dicho sistema.

Como herramienta para salvar todos estos obstáculos, se creó el término de Arquitectura Basada en Servicios, que establece que todo sistema debe exponer al mundo de una manera estándar los servicios que provee, mismos que después pueden ser utilizados por cualquier otro sistema. La ventaja de una arquitectura así radica en que cada sistema encapsula sus procesos internos, de tal forma que no es posible para un sistema externo saber exactamente cómo es que se realiza una operación, pero sí solicitarla. Esto significa que se pueden realizar todas las modificaciones necesarias a los procesos internos de un sistema y, siempre y cuando no se modifique la descripción del servicio, los demás sistemas podrán interactuar sin ningún cambio con él. Esto garantiza que se puedan construir grandes flujos de procesos de negocio, donde un sistema se conecta con otro y le manda el resultado a un tercero, y así sucesivamente para completar un proceso de negocio. La siguiente figura ilustra un ejemplo de esta arquitectura:

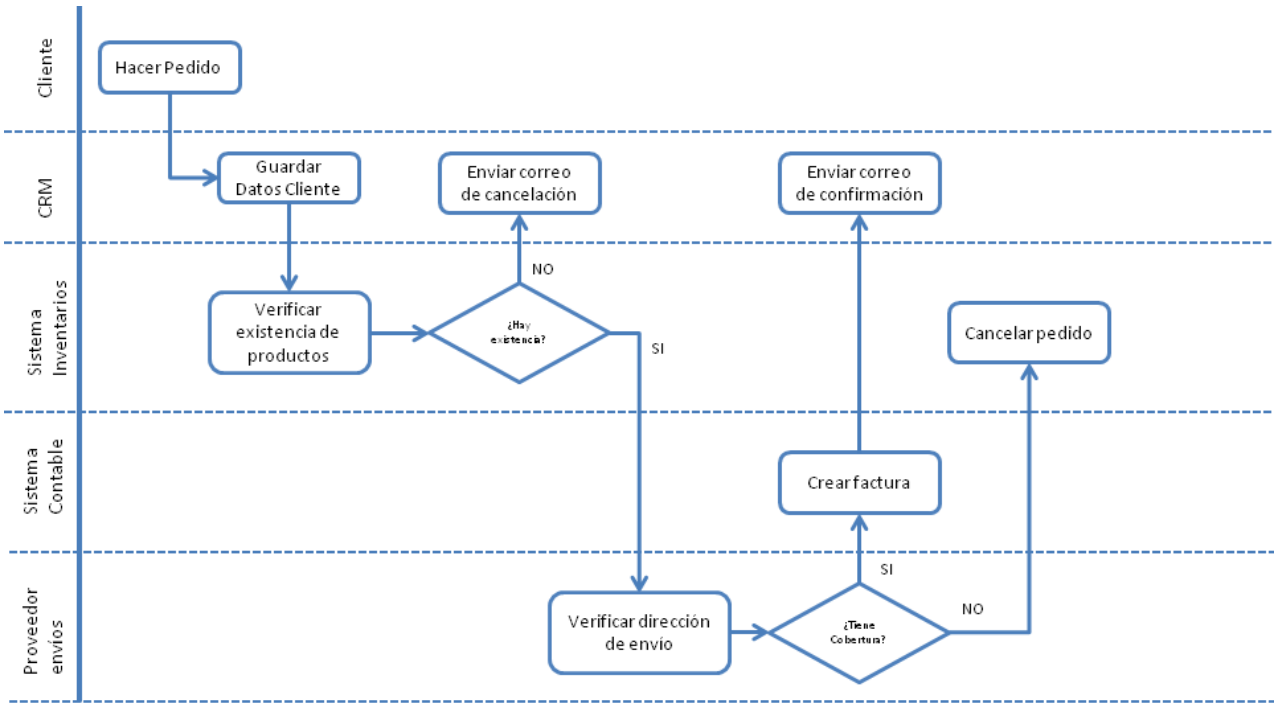


Ilustración 2: Diagrama BPMN

Esta figura de procesos de negocio muestra la interacción entre sistemas y personas que llevan a concluir una acción, en este caso, completar una venta. Como se puede apreciar, el sistema de CRM debe poder

recibir los datos del cliente y actualizarlos en su base de datos, además de poder enviar correos con mensajes predefinidos, el sistema de inventarios debe permitir consultar existencias de los productos del pedido, el sistema contable debe permitir crear facturas del pedido, y por último el proveedor externo de envío debe permitirle a los sistemas internos de la empresa consultar si la dirección introducida por el cliente se encuentra dentro de alguna de sus áreas de cobertura, de otra forma el pedido no podría ser completado. Estos son los servicios que cada sistema expone para su consumo por parte de sistemas externos.

La notación utilizada en la figura es conocida como BPMN (*Bussiness Process Management Notation*), es una notación estándar utilizada por analistas y gerentes de productos y de operaciones, que indica los pasos a seguir para terminar un proceso de negocio, y quién o qué es la entidad responsable de realizar cada paso. Gracias a esta notación se pueden definir criterios para medir la eficiencia del proceso, identificar cuellos de botella, y las reglas operacionales de la empresa, que en ocasiones son opacas para el resto de los empleados, y conduce a errores logísticos y de operación dentro de la empresa.

Para poder llevar a cabo todos los objetivos de una Arquitectura Orientada a Servicios, se recomienda implementar sistemas que lleven a cabo las siguientes funciones:

1. **BPM**: *Bussiness Process Manager*, es un sistema que se encarga de coordinar las llamadas a los servicios expuestos por cada sistema conectado, en el orden correcto y tomando las precauciones y control de errores necesarios para realizar un proceso de negocio. Básicamente es el que sabe qué se debe realizar, con qué valores, y en qué orden para poder realizar una acción completa de negocio, como recibir un pedido.
2. **BRE**: *Bussiness Rule Engine*, es un sistema que define las reglas de negocio que rigen sobre la operación de los procesos. Por ejemplo, si una regla de negocio establece que las personas que realicen pedidos desde su celular no podrán ser procesados y deberán cancelarse, esta regla estará en la base de datos de reglas del BRE, y el BPM, al consultar la siguiente acción a tomar en el proceso de negocio, obtendrá el siguiente paso, ya sea seguir con el pedido o cancelarlo.
3. **BAM**: *Bussiness Activity Monitor*, es un sistema que monitorea y registra las actividades de la empresa. Estas actividades son los bloques definidos en el diagrama BPMN, por lo que permite guardar información práctica respecto a la operación real de la empresa: tiempo de procesamiento, medidas de eficiencia operativa respecto a la cantidad de pedidos cancelados, enviados, pendientes, etc.
4. **ESB**: *Enterprise Service Bus*, es el sistema base, la espina dorsal de la compañía. A este sistema se conectan todos los demás sistemas presentes en la empresa, y cumple 4 funciones principales:
    a. Mediación: todos los sistemas se comunican con el ESB, que incluye conectores listos para usarse que permiten que cualquier sistema se conecte con él, independientemente del puerto, protocolo, formato, etc. De esta manera los sistemas ya no se hablan entre sí, sino que siempre pasa a través del ESB. Esto significa que el BAM puede monitorear directamente los mensajes que pasan por el ESB para obtener la información y estadísticas que reporta.

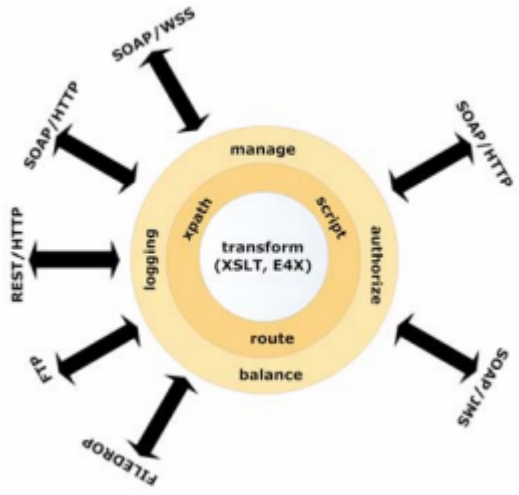


Ilustración 3: Mediación de servicios

b. **Ruteo de mensajes**: El ESB tiene un registro de cada sistema conectado a él, por lo que puede enrutar una petición realizada por otro sistema a su destino.
c. **Transformación y enriquecimiento de mensajes**: Este es un caso muy usual en empresas grandes: en muchas ocasiones, los sistemas con los que se cuenta son obsoletos, desarrollados en diferentes plataformas o lenguajes, por lo que los servicios que exponen pueden estar esperando protocolos muy diferentes para comunicarse con ellos. La función de un ESB en este caso es poder transformar un mensaje de un protocolo que entiende el sistema origen a un protocolo que entiende el sistema destino, sin hacer un desarrollo extra en cualquiera de ambos sistemas para poderse entender.
d. **Composibilidad de servicios**: En ocasiones, los servicios expuestos por los sistemas internos de la empresa son de muy bajo nivel, y para ser útiles se requiere unir varios servicios en uno solo. El ESB permite crear servicios compuestos.

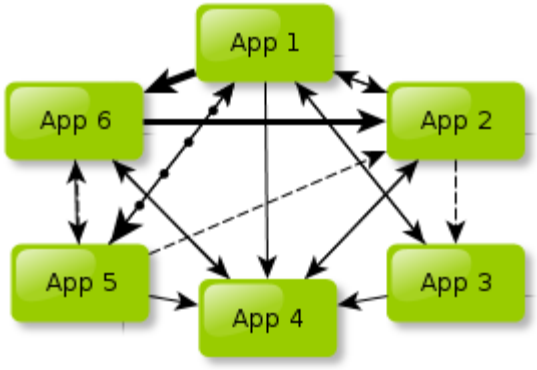


Ilustración 4: Arquitectura de Sistemas sin ESB

El objetivo de utilizar un ESB es centralizar todas las comunicaciones entre sistemas, de tal forma que una aplicación no dependa de manera directa de otra, sino que estén desacopladas y unidas mediante un intermediario que permite realizar funciones más avanzadas, como guardar mensajes en caso de que uno de los sistemas no responda o esté en mantenimiento, y enviarlos en cuanto vuelva a estar disponible.

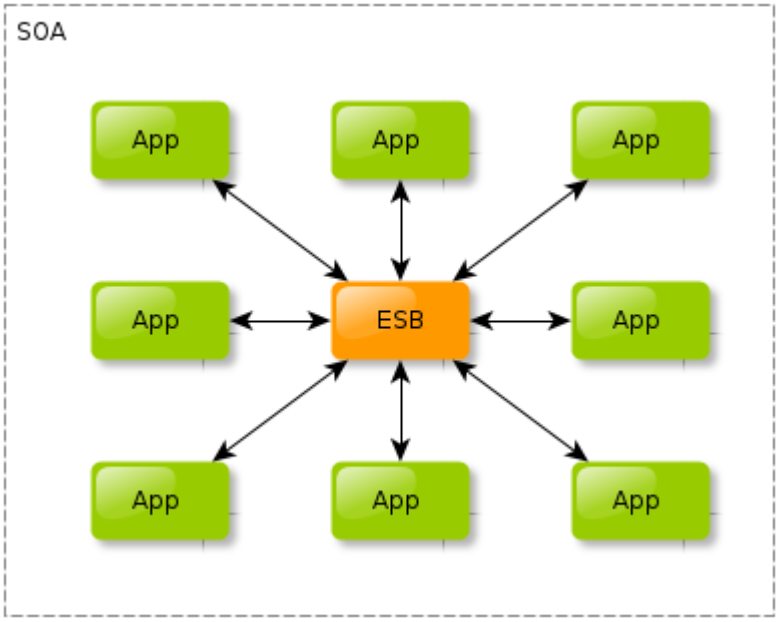


Ilustración 5: Organización de sistemas internos con un ESB

5. **API Manager**: es un sistema que permite reunir en un solo lugar todos los servicios que estarán expuestos al público, ya sea como parte de la página web de la compañía, o para consumo desde una app o un tercero previamente aprobado para ello. Esta capa se encarga de manejar los controles de uso (cuántas peticiones puede realizar cada cliente por segundo), detalles de autenticación de clientes (cómo identificar quién tiene acceso a cada recurso) y de funcionar como la capa de entrada de las peticiones externas a los sistemas internos de la compañía.
6. **Identity Server**: es un sistema que reúne en un solo lugar los detalles de cada usuario, ya sea interno (empleados de la compañía) o externos (clientes), y que realiza las funciones de autenticación y autorización para los demás sistemas de la empresa, incluyendo las funciones de SSO (*Single Sign On*): una vez que inicias sesión en cualquiera de las páginas del dominio de la empresa, no es necesario volver a iniciar sesión al querer acceder a otra página.
7. **Service Registry**: Todos los sistemas de negocio que exponen servicios deben estar en un registro central, de tal forma que sea posible para un nuevo sistema descubrir todos los servicios internos que puede utilizar, sin necesidad de conocer a detalle cada uno de los sistemas existentes.

Otra característica importante de SOA es que utiliza XML y sus múltiples estándares WS-*, como WS-BPEL, WS-Security, WS-Coordination, WS-AtomicTransaction, WS-BusinessActivity, WS-CDL, etc. Todas estas extensiones son utilizadas para crear y componer flujos de trabajo complejos entre los diferentes sistemas exponiendo servicios.

Para concluir, una Arquitectura Orientada a Servicios es el objetivo final de todo el desarrollo realizado durante mis actividades profesionales. Rara vez se puede comenzar con un diseño 100% SOA, debido a la complejidad de reconfigurar todos los sistemas actuales para adaptarse a esta arquitectura, sino que por lo general se van dando pasos incrementales hacia una arquitectura final que tenga todas estas características deseables para las empresas.

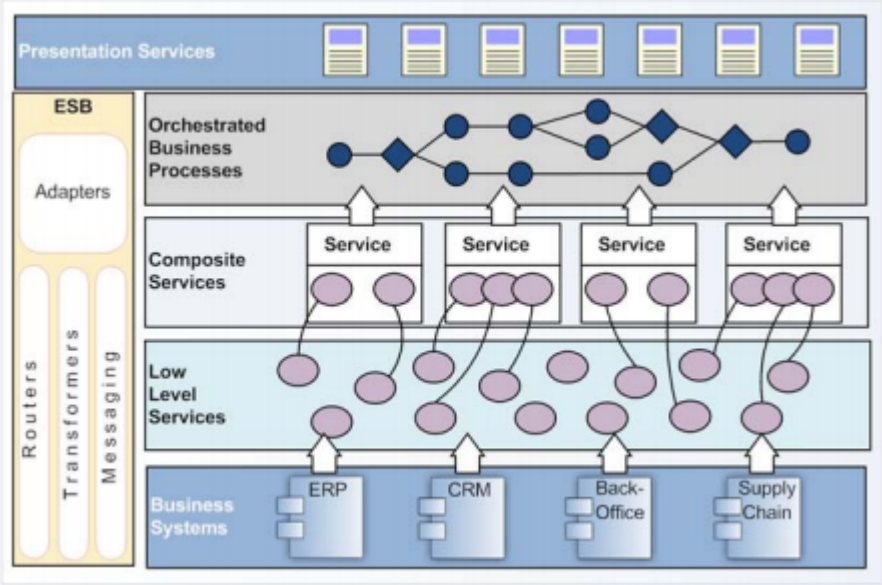


Ilustración 6: Arquitectura SOA Final

REST(REpresentational State Transfer)

La web que todos conocemos es el mejor ejemplo de una arquitectura distribuida: se basa en miles de computadoras conectadas entre sí a través de cables y equipos complementarios, y permite que todas ellas se comuniquen entre sí, si así lo desean. No es azar que haya surgido de esa manera, ni que haya crecido tanto después de su invención: existe una arquitectura de sistemas distribuidos que define las diferentes capas y estructuras mediante las cuales la web puede existir, y sobre ella, todos los servicios, arquitecturas de sistemas, y plataformas que construyen tanto gobierno como entidades privadas.

Cuando Tim Berners-Lee realizó la primera comunicación entre un servidor y un cliente mediante el protocolo HTTP en 1991, se dice que inició la web. Desde entonces, este protocolo ha ido evolucionando, desde la versión 1.0 en 1996, hasta la más actual, 2.0 en el 2015, bajo la supervisión de varios organismos internacionales, entre ellos el *World Wide Web Consortium* (W3C) y el *Internet Engineering Task Force* (IETF). Sin embargo, no fue hasta el año 2000, con la publicación de su tesis de doctorado, que Roy Fielding introdujo una arquitectura formal de sistemas distribuidos basados en el protocolo HTTP, REST (*REpresentational State Transfer*). Esta publicación revolucionó en gran medida el estilo y la construcción de servicios web, al grado de que la mayor parte de las grandes empresas tecnológicas hoy en día exponen sus servicios internos mediante una API del tipo REST, de aquí en adelante referida como RESTful.

Antes de ser introducida formalmente esta arquitectura, se utilizaban muchos métodos para la conexión entre sistemas distribuidos, siendo las principales CORBA, COM(Microsoft) RPC (predecesor de SOAP) y SOAP. Muchas empresas habían establecido sus propios sistemas y protocolos para comunicación entre sistemas desarrollados con los mismos lenguajes o del mismo proveedor, y poco a poco se estaba estableciendo SOAP como el protocolo por defecto de comunicación entre sistemas. En el mundo financiero y de grandes corporativos, SOAP sigue siendo el protocolo por defecto, debido a la gran cantidad de sistemas en operación que tienen, muchos de los cuales tienen ya alguna interfaz SOAP. Ahora bien, para todos los sistemas nuevos que expondrán una API al mundo, lo más común es realizarla para que sea RESTful, mientras que, si sólo expondrán una API para otros sistemas construidos hace tiempo dentro de la misma empresa, es más probable que se utilice SOAP.

Lo importante de la arquitectura REST es que no sólo funciona para APIs que implementen por completo sus instrucciones, sino que también define varias capas del funcionamiento actual de SOAP, puesto que en muchas ocasiones, este protocolo utiliza HTTP por abajo para funcionar.

A continuación se establecen las principales ideas detrás de la arquitectura:

- **Cliente-Servidor**: establece que los sistemas o plataformas que se creen para comunicarse deben funcionar mediante este paradigma, de tal manera que haya una separación de funcionalidades: la interfaz gráfica no requiere saber exactamente cómo ni dónde se guarda la información. Además, esto permite que cada parte evolucione de manera independiente.

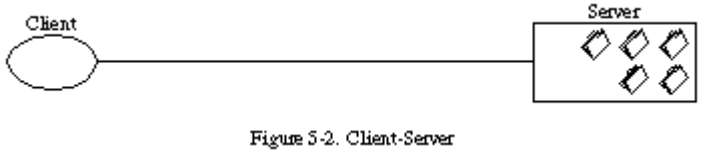


Figure 5-2. Client-Server

Ilustración 7: Arquitectura Cliente-Servidor

- **Independiente del estado (*stateless*)**: Cada petición que se realice del cliente al servidor debe ser independiente del estado de la aplicación del servidor, es decir, cada petición debe contener toda

la información necesaria para que el servidor la entienda, sin hacer uso de ningún tipo de contexto o datos adicionales almacenados en el servidor para dicho cliente.

Las ventajas de esta restricción son las siguientes:

- **Visibilidad**: significa que un sistema de monitoreo sólo requiere ver una petición para poder entender qué es lo que estaba sucediendo en el momento.
- **Confiabilidad**: recuperarse de los errores es mucho más sencillo, puesto que con mandar de nuevo las peticiones realizadas al servidor se obtiene la respuesta esperada.
- **Escalabilidad**: Dado que el servidor no guarda ningún tipo de información de estado, puede procesar más rápido cada solicitud, y su implementación es mucho más sencilla al no tener que administrar dichos estados.

Por otro lado, esto crea los siguientes problemas:

- Disminuye el rendimiento de red, dado que en cada petición se tienen que enviar datos repetitivos de estado para que el servidor pueda procesarla.
- El responsable de mantener el estado de la aplicación es el cliente, lo que reduce la capacidad del servidor de mantener un comportamiento consistente de la aplicación.

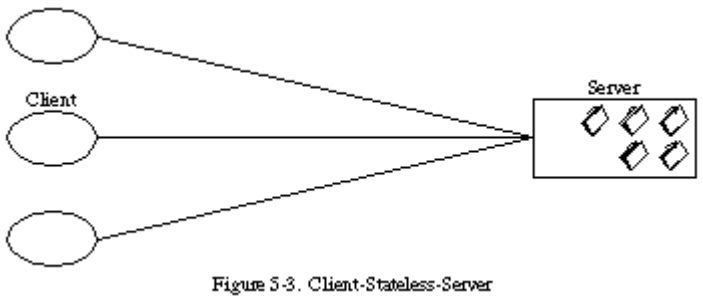


Figure 5-3. Client-Stateless-Server

Ilustración 8:Arquitectura Cliente-Servidor independiente del estado

- **Caché**: establece que cada recurso que se requiera del servidor debe estar marcado, dependiendo de si puede guardarse un tiempo en el cliente (cacheable), o si, por el contrario, cada vez que se requiera esa información debe solicitarse de nuevo al servidor. Esto permite disminuir significativamente la cantidad de peticiones que el servidor debe procesar, puesto que los recursos requeridos ya se encuentran en el caché del cliente, y aún no han expirado. Esto abre la puerta a que existan recursos que hayan sido modificados en el tiempo que se pidieron por primera vez y cuando se están utilizando directo del caché. Las decisiones sobre cuánto tiempo los recursos pueden considerarse “frescos” debe tomarla el servidor indicando una fecha de expiración, puesto que se considera que él es quien sabe cuán frecuentemente se modifican dichos recursos. Un ejemplo de esto son los cachés de los navegadores web actuales, que piden todos los recursos al cargar por primera vez una página, pero las subsecuentes solicitudes sólo piden aquellos recursos que no son comunes entre ellas, acelerando en gran medida la carga de las páginas.

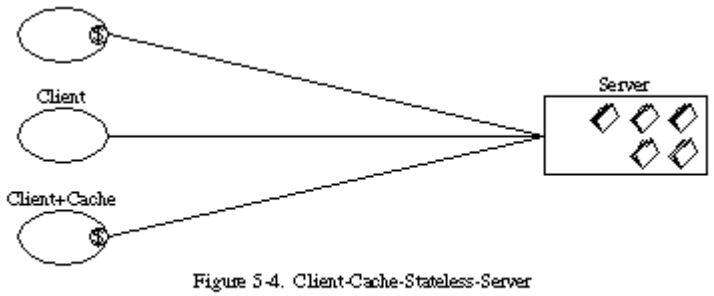


Figure 5-4. Client-Cache-Stateless-Server

Ilustración 9: Cliente-Servidor con caché en el cliente

- **Interfaz Uniforme**: toda la comunicación entre componentes debe realizarse mediante una interfaz uniforme, de tal manera que las implementaciones de servicios sean independientes de la interfaz, permitiendo así evolución independiente de ambas cosas. La desventaja de esto es que, al no haber interfaces específicas por servicio, se corre el riesgo de transmitir mucha información innecesaria en una petición. Ejemplos de interfaces son JSON y XML, ambos formatos estandarizados para transferencia de información.

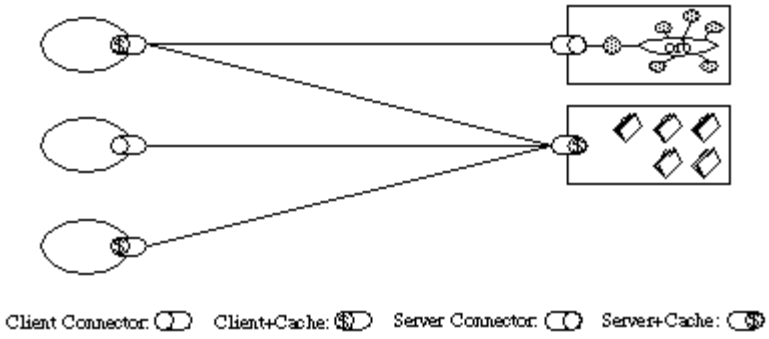


Figure 5-6. Uniform-Client-Cache-Stateless-Server

Ilustración 10: Cliente-Servidor con caché, independiente del estado y con interfaz uniforme

- Sistema en capas: Al añadirle capas a un sistema se permite crear una jerarquía de niveles, restringiendo la operación de cada componente y su interacción con los demás, haciendo que cada componente sólo pueda "ver" la capa inmediata con la que está interactuando. Esto permite encapsular los detalles de implementación de los sistemas o la existencia de sistemas obsoletos, el lugar de almacenamiento de los recursos, etc, bajando así la complejidad completa del sistema.

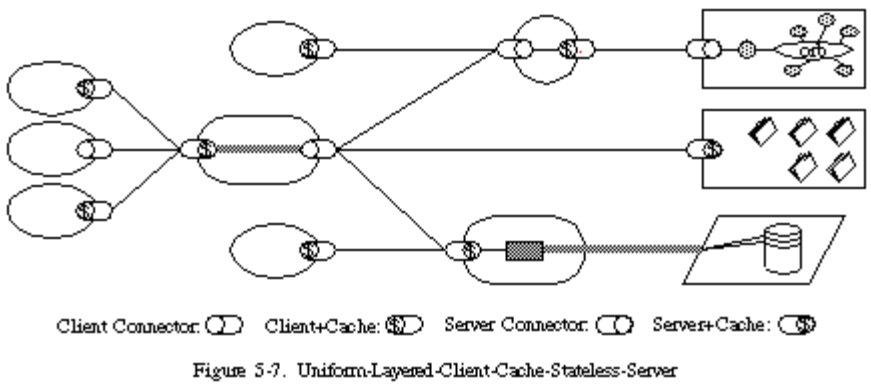


Figure 5-7. Uniform-Layered-Client-Cache-Stateless-Server

Ilustración 11: Cliente-Servidor con caché, interfaz uniforme y en capas

- Código bajo demanda: la arquitectura REST también permite ampliar las capacidades o funcionalidades de los clientes mediante la descarga de código (scripts) o applets bajo demanda. Esto aumenta la extensibilidad de los clientes, pero reduce en gran medida la visibilidad, por lo que esta característica está marcada como opcional.

La arquitectura REST además define elementos de datos, mismos que podemos resumir de la siguiente manera.

Elemento	Descripción	Ejemplo
Recurso (resource)	Un recurso es una abstracción de la información, cualquier conjunto de datos puede ser considerado un recurso: un documento, una imagen, una colección de recursos, etc. El recurso es la base de la arquitectura REST.	Detalle de un artículo en un sitio de compras, una imagen, etc.
Identificador de recurso	Cada recurso requiere ser identificado de manera única, de tal forma que si un cliente quiere acceder al recurso en un tiempo t posterior a su primer acceso, aún pueda hacerlo si tiene guardado su identificador.	URL, URN, URI
Representación	Consiste en la información per-se, es lo que se obtiene al solicitar un recurso al servidor.	Documento HTML, imagen JPEG.
Metadatos de representación	Indican qué tipo de medio es el recurso que se ha solicitado, así como datos generales sobre él.	Last-modified time, media type
Metadatos de recurso	Datos sobre el recurso, por ejemplo, dónde encontrarlo, alternativas en caso de que el servidor original no esté disponible, etc.	Source link, alternates, vary
Datos de control	Definen datos especiales que sólo hacen sentido para el cliente, e indican cómo proceder al querer solicitar o al recibir ciertos recursos.	If-modified-since, cache-control

Tabla 1: Elementos REST

Todos estos elementos y características de la arquitectura REST, a pesar de haber sido descritas de una manera muy abstracta, influyeron en una gran medida en los sistemas diseñados hoy en día.

Un claro ejemplo se puede observar en la implementación del API de Twitter, donde ellos mismos hacen referencia a que fue construida siguiendo las ideas de REST:

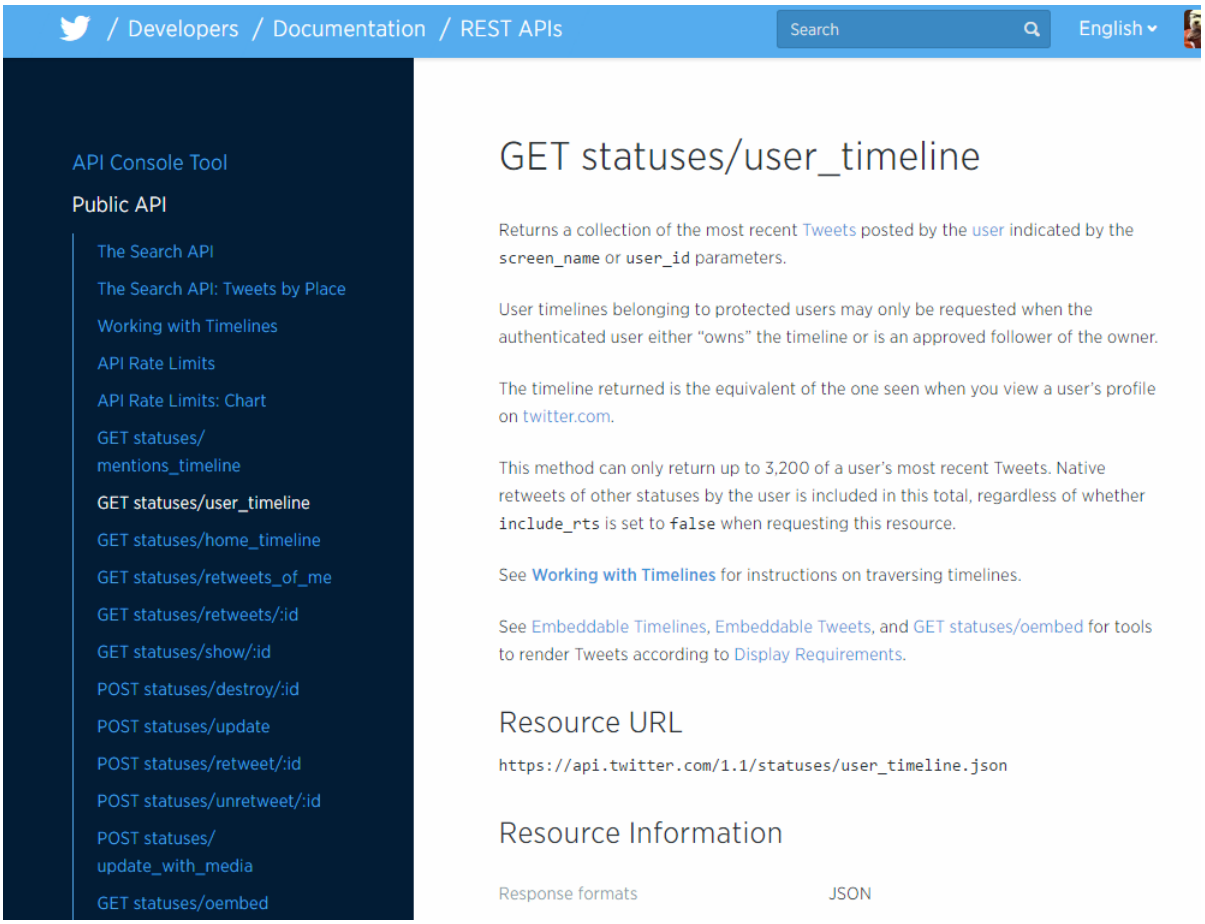


Ilustración 11: Twitter API

Esta arquitectura se presta mucho más que otras para la construcción de este tipo de API debido a que la base del producto se puede representar de manera muy sencilla mediante un recurso. En el caso de Twitter, los recursos pueden ser:

- Status, equivalente a un Twit,
- Direct Messages,
- Friends,
- Followers,
- Account,
- Users,
- Favorites,

- Lists,
- Y muchos más

Al trabajar con estos recursos, solamente se requiere definir los detalles de representación. La mayor parte de las API actuales prefieren utilizar HTTP como el medio de transporte y JSON como la representación por default, pero permiten que los clientes indiquen la representación que prefieren al hacer la petición. Incluso algunas API permiten otro tipo de medio de transporte, como TCP o directamente a un puerto IP. Las representaciones más comunes son JSON, XML, CSV, HTML y texto plano.

Cada uno de estos recursos tiene sus características de caché particulares, y las acciones que se pueden realizar con ellos. En una API RESTful, se utilizan los métodos (verbos) definidos como parte del protocolo HTTP para indicar la acción a realizar con el recurso, siendo los más comunes:

Método	Acción	Ejemplo
GET	Regresa la información sobre el recurso especificado por el identificador	Al escribir www.google.com, el navegador hace una petición GET hacia el servidor con dicho identificador.
POST	Permite modificar información de un recurso anteriormente guardado en el servidor, ya sea para sobrescribir información o para agregar información nueva al recurso.	Al enviar un formulario, comúnmente se hace mediante una petición POST
PUT	Solicita que la información enviada se guarde en el servidor. Si la petición incluye un identificador, se permite que se considere una modificación al recurso original, en caso contrario se maneja como un nuevo recurso.	Rara vez se usa en los navegadores web, pero en servicios web RESTful suele utilizarse para creación de recursos.
DELETE	Solicita que el servidor elimine el recurso identificado en la petición.	Es más utilizado en API RESTful que en clientes como navegadores web.
HEAD	Hace lo mismo que el método GET, solo que no regresa la información del recurso, sino sólo los datos de cabeceras. Usualmente se utiliza para verificar la existencia del recurso (sin necesitar la información actual).	Al validar links a otros recursos, es bastante utilizado.
OPTIONS	Representa una solicitud de información sobre las opciones de comunicación disponibles para el cliente, así como las capacidades y los métodos HTTP aceptados en una petición a ese recurso.	En muchos clientes se acostumbra hacer una petición OPTIONS antes de un GET para garantizar que sí se puede realizar la acción.

Tabla 2: Métodos HTTP en servicios RESTful

Gracias a todas estas características, es posible construir un servicio web, o API, que mediante las capacidades existentes en el protocolo HTTP permita realizar acciones sobre recursos contenidos en el sistema que provee dicho servicio. Contrastado con la complejidad de los servicios SOAP, esta simplicidad

de construcción y estandarización de funcionamiento (los programadores esperan que al hacer una petición GET a cualquier API, regrese los detalles de un recurso) han hecho que se vuelva mucho más popular construir servicios web utilizando la arquitectura REST.

Microservicios

Una vez definida la arquitectura a alto nivel del sistema (qué servicios debe exponer, cuáles serán sus responsabilidades, qué lógica de negocio tendrá, etc), se deberá tomar la decisión de qué forma tomarán los servicios que deben construirse. Una propuesta interesante dentro del ámbito de SOA son los microservicios, una arquitectura que promueve las siguientes características:

- **Minimizar el costo de realizar cambios**: todos los productos de software inevitablemente requieren adiciones o cambios a funcionalidades previamente implementadas. En sistemas incorrectamente diseñados, un cambio que a primera vista parece ser cuestión de minutos, puede convertirse rápidamente en un trabajo arduo debido a referencias múltiples en otros archivos, dependencias no obvias entre módulos, y otras cosas que requieran también cambios y se vuelva una reacción en cadena. En el caso de los microservicios, una de las principales propuestas que tienen es permitir escalar en funcionalidad, siguiendo las siguientes ideas:
    - Responsabilidades distintas se colocan en servicios distintos.
    - Cada servicio es responsable de su propio repositorio, no se comparten.
    - Cada instancia de un microservicio se ejecuta como un proceso dedicado e independiente.
    - La comunicación entre servicios sólo es permitida a través de la red, es decir, los servicios deben interactuar entre sí mediante sus API, jamás directamente hacia archivos o la base de datos de uno de ellos.
    - El protocolo de comunicación debe ser universal, cualquier lenguaje de programación debe poderlo hablar.
    - Dado que no todos los problemas de negocio son iguales, los microservicios promueven el uso de tecnologías y plataformas específicas para cada problema, permitiendo así que cada equipo desarrolle sus servicios y librerías sobre lenguajes separados, si así escogen. Esto permite que cada equipo tenga sus tiempos de entrega, no necesariamente ligados a los de los demás equipos, y que sea libre para elegir entre diferentes alternativas de desarrollo, por ejemplo, escogiendo flexibilidad sobre rapidez de desarrollo, o facilidad de mantenimiento sobre flexibilidad, dependiendo del problema de negocio en cuestión a resolver.
    - Coreografía debe ser preferida sobre orquestación. Cuando una arquitectura orientada a servicios utiliza un patrón de orquestación para comunicarse, hay llamadas y conexiones punto a punto entre los servicios, creando así una teleraña de llamadas que vuelve muy difícil de mantener, cambiar o reemplazar los servicios individuales. En cambio, cuando se aplica un patrón de coreografía, un servicio no se conecta directamente con otro para

realizar una acción, sino que todos ellos están conectados a un bus de eventos y solamente reaccionan de acuerdo a los canales de eventos a los que están suscritos. Esto significa que los canales de eventos deben ser lo más sencillos posibles, y que la lógica detrás del consumo de los mensajes del canal debe estar en cada servicio, prefiriendo así simples protocolos REST o de mensajería sobre complejidades SOAP como WS-Choreography u orquestaciones por medio de BPEL (*Business Process Expression Language*, el lenguaje en el que se definen los procesos de negocio en un BPM).

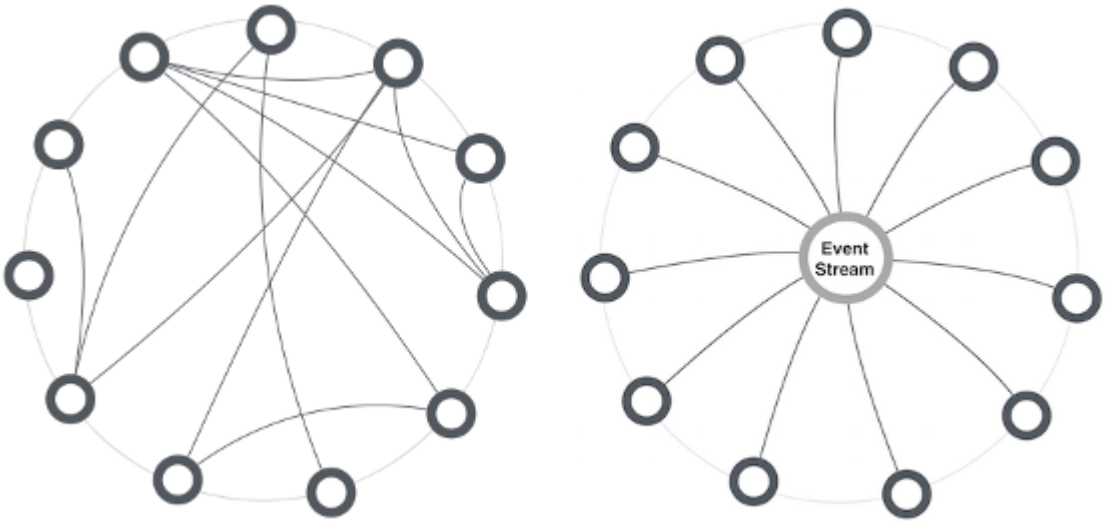


Ilustración 12: Orquestación vs Coreografía

- **Organización de servicios de acuerdo a las capacidades y actividades del negocio**: en la mayor parte de las organizaciones, cuando se intenta separar un componente monolítico de software en partes, usualmente se hace de acuerdo a la tecnología implementada, por ejemplo UI, Backend, Bases de datos, etc. Esto hace que haya lógica de negocio implementada en todas partes, y se vuelve muy difícil de mantener. Esta estructura es el mejor ejemplo de la Ley de Conway: Toda organización que diseña un sistema producirá un diseño cuya estructura es una copia de la estructura de comunicación de la organización. Por ejemplo:

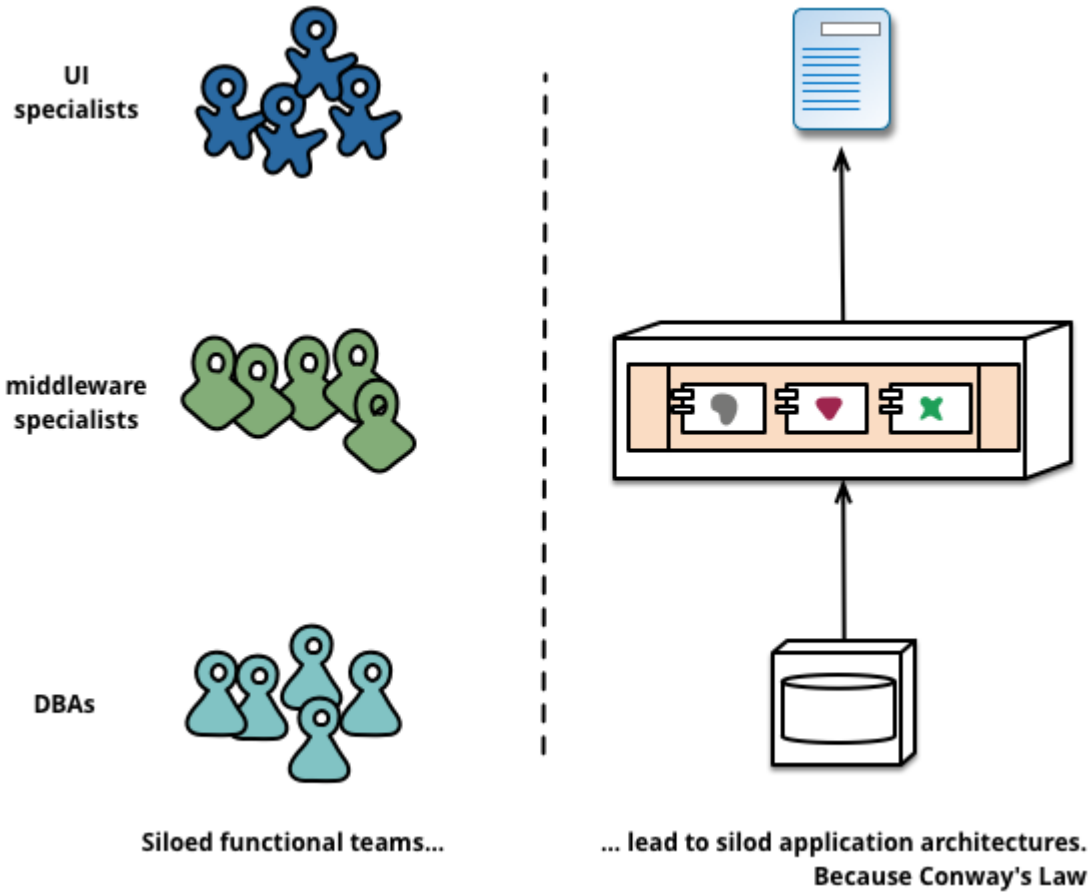


Ilustración 13: Ejemplo de Ley de Conway

Una arquitectura basada en microservicios maneja una división diferente de responsabilidades: organiza y divide los servicios de acuerdo a las capacidades o actividades principales de la organización, creando equipos multidisciplinarios, donde cada equipo tiene todas las habilidades técnicas necesarias para el desarrollo: administración, experiencia de usuario, bases de datos, y desarrollo.

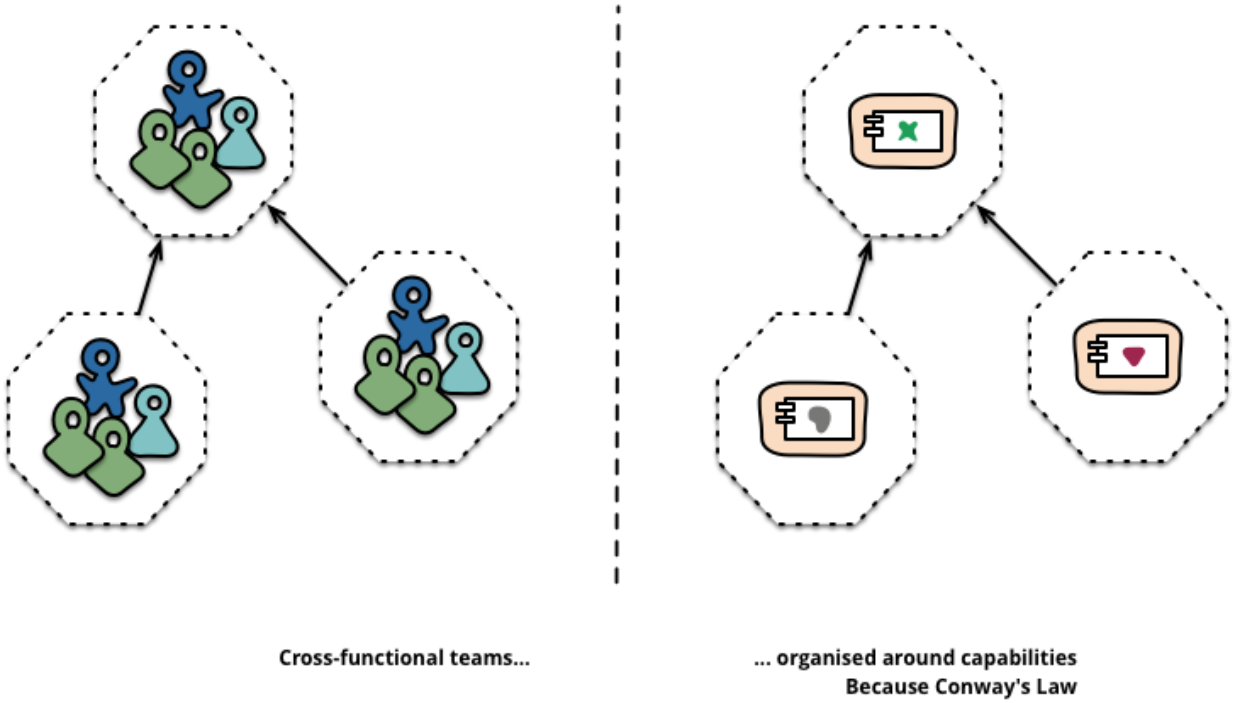


Ilustración 14: Organización de acuerdo a microservicios

- **Administración decentralizada de datos**: los sistemas monolíticos y los que siguen por completo la arquitectura SOA tienden a favorecer la persistencia de datos en un solo lugar y en un solo producto. Esto permite que todas las operaciones de datos sean íntegras y consistentes, es decir, que no se pierdan ni se cambien datos y por tanto quede incoherente la información guardada. Sin embargo, esto propicia que, si se debe actualizar un dato en una tabla compartida entre diferentes secciones del código (cosa bastante común, por ejemplo un Contrato que debe poderse visualizar desde la sección contable, de clientes, de proceso, etc), se introduce un acoplamiento temporal en ese recurso: mientras uno de ellos lo está utilizando, los demás no podrán hacer uso de él sin violar la coherencia de datos. A este tipo de procedimientos se les conoce como transacciones, y son la base del funcionamiento de las bases de datos para responder al problema del acceso o modificación concurrente de la información.
  La arquitectura de microservicios prefiere distribuir la información utilizando una base de datos por cada servicio, asegurando así que solamente ese servicio tiene acceso a ella y que nadie más podrá estar haciendo uso de manera concurrente. Al trabajar así se permite que cada aplicación elija la solución de persistencia que mejor funcione para su carga estimada de trabajo, y se eliminan las transacciones distribuidas. La desventaja es que no se puede garantizar una consistencia fuerte, como en las transacciones de los sistemas centralizados, y debe emplearse un monitoreo y corrección de datos estricto para mantener la coherencia y consistencia de los datos ahí guardados.

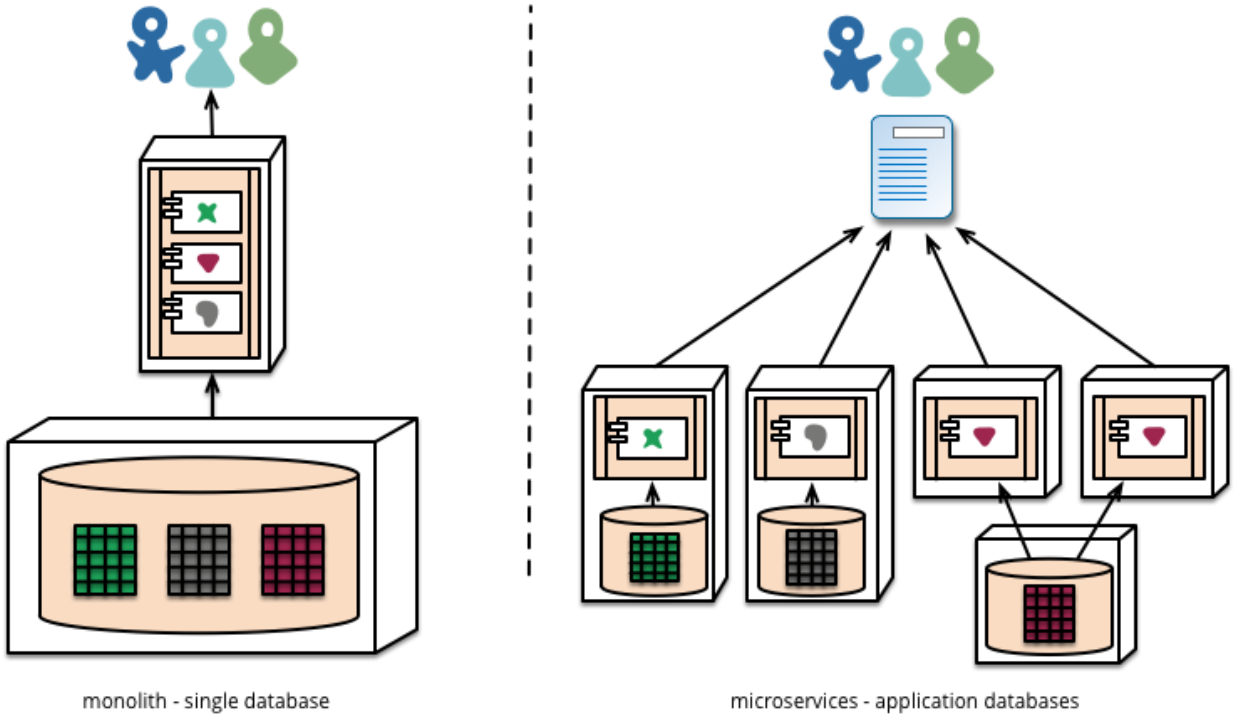


Ilustración 15: Decentralización de datos

- **Automatización de la infraestructura**: conforme los equipos de desarrollo y las herramientas han avanzado, se ha podido automatizar cada vez más los procesos de compilación, pruebas de usuario, integración, funcionales y de rendimiento, así como el poder distribuir el código nuevo a todos los servidores donde se ejecuta la aplicación.

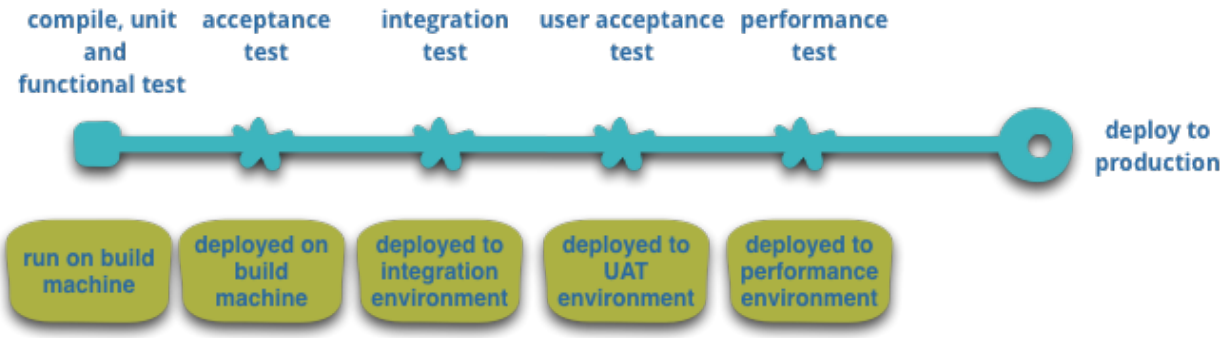


Ilustración 16: Diagrama de proceso de despliegue de código

La idea de automatización del proceso de despliegue de software hace posible se conoce como *Continuous Delivery*, que significa “Entrega Continua”. Esto significa que, al automatizar todos los pasos por los que debe pasar el código antes de poder ser entregado al usuario final, el proceso

de despliegue se vuelve muy sencillo, y no requiere mayor intervención humana, minimizando así los errores durante las pruebas y durante el proceso de despliegue. Gracias a esta automatización, la entrega continua puede ser de tantos microservicios como se desee, sin que aumente la necesidad de un equipo dedicado a realizarlos.

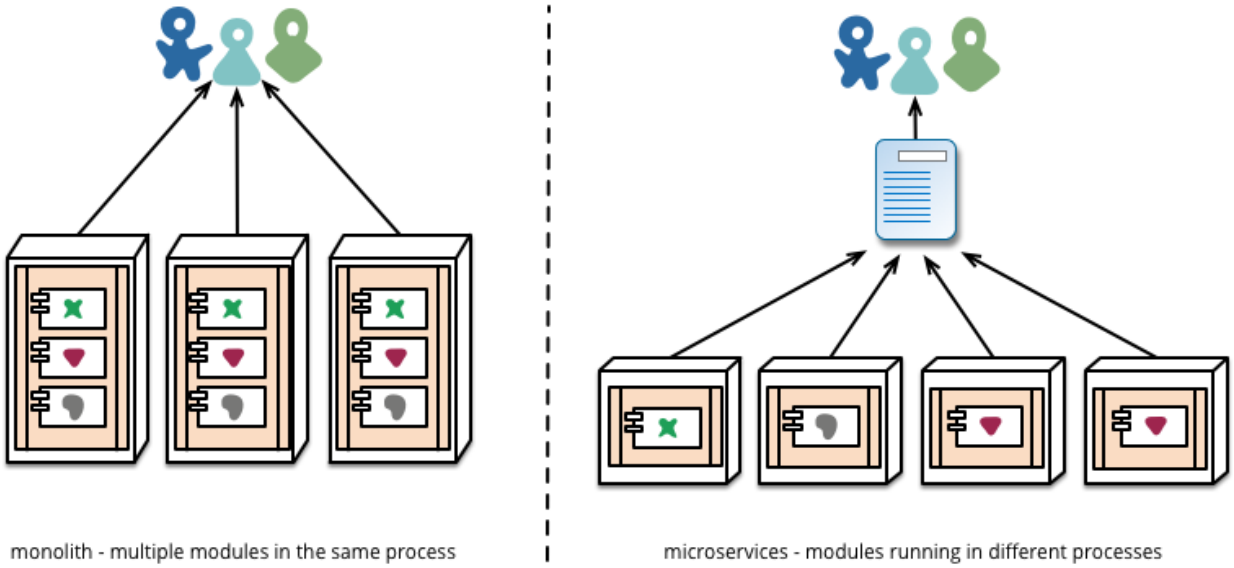


Ilustración 17: Despliegue de sistemas monolíticos vs microservicios

- **Diseñados para fallar**: dado que esta arquitectura requiere que sean varios los servicios que estén operando 24/7, la probabilidad de que el sistema en conjunto falle es proporcional a la probabilidad de que alguno de los servicios de los que está compuesto falle. La conclusión final de esto es que un sistema compuesto por varios subsistemas será mucho más propenso a fallas debido a la mayor cantidad de partes que tiene. Debido a esto se ha vuelto especialmente relevante la lógica introducida en cada servicio que permite tolerar cierto nivel de fallos de los demás. En cualquier momento una llamada a un servicio puede fallar, haciendo que quien había solicitado dicho servicio requiera responder de una manera elegante, sin tirar por completo el sistema. Uno de los sistemas más conocidos en la industria de esto es el *Simian Army* [10]de Netflix, un sistema específicamente diseñado para hacer fallar todos los servicios de los que se compone el servicio de video por streaming de Netflix. Con esta herramienta, que se ejecuta todos los días en horas de trabajo, se simulan caídas inesperadas de servicios, y con ello los ingenieros de Netflix pueden observar cómo se comportan los demás servicios cuando alguno falla y qué se requiere para mantener un nivel de servicio aceptable aún habiendo fallos.
  Otra consecuencia de esto es que todos los servicios deben estar transparentemente monitoreados y sus salidas constantemente examinadas, a fin de poder establecer cuándo ocurren fallos y reaccionar lo antes posible ante ellos.

[10] https://github.com/Netflix/SimianArmy

- **Diseñados para evolucionar**: la tendencia en equipos que desarrollan microservicios es pensar en ellos como un producto y no como un proyecto, entendiendo que su trabajo nunca está terminado, puesto que siempre hay cambios, optimizaciones, correcciones y modificaciones que hacer en cada uno de los servicios. Un proyecto, en cambio, da la idea de que una vez terminada cierta etapa, acaba ya la responsabilidad de los creadores. Amazon en particular es famoso por su ideología de "You build it, you run it", significando que el mismo equipo que creó el servicio es responsable de asegurarse de que se esté ejecutando como es debido, haciendo así evidente la buena o mala calidad del código producido en la cantidad y frecuencia de fallos que presenta. Otra parte de esto es cómo puede un sistema monolítico evolucionar a uno basado en microservicios. El periódico inglés The Guardian ha demostrado como ambos patrones pueden existir a la vez, exponiendo APIs a las diferentes secciones del núcleo monolítico y utilizando lenguajes de prototipado rápido para extender dichas funcionalidades y crear servicios ad-hoc a las noticias o la cobertura necesaria para cierto evento, y desechando el código especializado cuando la cobertura termina. Poco a poco, el código generado que es común para varias coberturas se va consolidando y proveyendo los servicios que antes proporcionaba el núcleo monolítico, y así se va logrando una separación de responsabilidades mediante la construcción de microservicios.

La arquitectura basada en microservicios se puede considerar un caso de arquitectura SOA, incluso muchos ingenieros de peso en la industria, como Martin Fowler y los ingenieros de Netflix argumentan que los microservicios son "SOA realizado como debe ser". En general, podemos ver que se obtienen muchas ventajas (y sus correspondientes problemas) al utilizar una arquitectura de este tipo, pero la elección final debe tomarla el arquitecto de software con argumentos sólidos y después de un análisis de la organización, la experiencia acumulada del área de desarrollo, la existencia del personal con las habilidades para lograr cada etapa, la facilidad de comunicación interorganizacional, y el panorama futuro y las proyecciones tecnológicas y económicas de la compañía. Una vez tomada la decisión, lo común es que deban pasar varios años antes de poder vislumbrar la eficacia de la arquitectura diseñada, por lo que no es posible realizar mediciones y correcciones a corto plazo, una elección usualmente determina cómo se verán los sistemas de la empresa por años en el futuro.

Es por esto que los arquitectos de software basan mucho sus diseños en la experiencia y trayectoria profesional propia y del equipo de la organización con la que están trabajando. Sólo así es posible proponer una arquitectura y un plan de trabajo factible, con costos y riesgos aceptables, y que verdaderamente logre una mejora en la forma de trabajo respecto a cómo se hacía anteriormente.

Proyecto 1: Desarrollo de plan general de Infraestructura en la nube

Definición del proyecto

En un proyecto de software existen diferentes etapas, desde la planeación, evaluación de requerimientos, hasta la ejecución y la puesta en marcha del software desarrollado. Para este último paso, existen diferentes maneras de obtener y configurar la infraestructura necesaria para el lanzamiento.

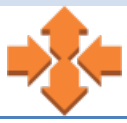
Una de las maneras más populares es mediante infraestructura en la nube (*Cloud Computing*), bajo demanda, y muchos proveedores han creado servicios de IaaS, entre ellos Microsoft (Plataforma Azure), Google (*Google Cloud Computing*) y Amazon (*Amazon Web Services*). Todos ellos ofrecen un servicio de infraestructura bajo demanda, en el cual el cliente puede elegir cuánta capacidad de cómputo desea rentar, y pagar solamente por las horas de uso. Esto permite una enorme flexibilidad a la hora de crear proyectos o asignar presupuestos: dado que no hay costos fuertes de inversión iniciales, los proyectos se pueden suspender, cambiar, aumentar o disminuir el alcance del mismo sin comprometer recursos ya invertidos.

Esto permite tener una rápida respuesta a los requerimientos cambiantes del proyecto, en términos de recursos necesarios de procesamiento, memoria o almacenamiento: cuando se requiera aumentar o disminuir dichos recursos, no se requieren recursos especializados en hardware o en administración de sistemas. Mediante la interfaz web de AWS, se pueden crear los recursos de manera sencilla y en cuestión de minutos, sin necesidad de compras, facturaciones, envíos, logística, instalaciones, garantías, etc.

Debido a todas estas ventajas respecto a la infraestructura propia, se decidió hacer uso de Amazon Web Services (AWS) para toda la infraestructura tecnológica de la empresa, salvo por una conexión con un proveedor externo, conexión en la que se profundiza en el siguiente capítulo.

AWS ofrece decenas de servicios en la nube, muchos de ellos siendo automatizaciones a escala de funcionalidades que, si bien es sencillo configurar para pocos servidores, es muy complicado crecer dicha configuración a más, y caro en el sentido de que se requiere siempre un administrador de sistemas con excelentes conocimientos de las tecnologías y configuraciones realizadas, además del monitoreo y procesos de recuperación necesarios.

A continuación se describen los diferentes servicios que ofrece AWS.

Logo	Nombre	Descripción
	Amazon EC2 (*Elastic Cloud Compute*)	Servicio que permite crear servidores virtuales bajo demanda y utilizarlos como si fueran servidores físicos. Al crear una nueva instancia, se elige el sistema operativo, el tamaño de almacenamiento y el tipo de instancia (procesadores, memoria, etc).
	Auto Scaling	Servicio que permite que un grupo de instancias se comporten de cierta forma de acuerdo al tráfico, % de uso de CPU, etc. Para usarlo se selecciona una imagen de disco, y las acciones a tomar cuando se cumplan las

		condiciones establecidas.
	Elastic Load Balancer	Servicio que permite balancear las peticiones que recibe un sitio web. Al crearlo, se elige qué puertos estarán abiertos, las instancias EC2 a las cuales se redirigirá el tráfico, además de un *health check*, que revisa que las instancias detrás del balanceador funcionan correctamente.
	Amazon VPC (Virtual Private Cloud)	Servicio que permite crear secciones de la nube lógicamente aisladas de las demás, para usarse como un centro de datos privado. Esto permite controlar mediante reglas ACL los puertos, origen y destino del tráfico que entra a la red interna.
	Amazon Route 53	Servicio de registro de DNS de Amazon. Aquí se puede configurar subdominios, redirecciones, e incluso balanceo de tráfico mediante diferentes métodos: localización geográfica, carga, porcentaje, etc.
	Amazon CloudWatch	Servicio que monitorea y dispara alarmas cuando algún evento o condición se cumple. Por lo general se utiliza para monitorear las instancias EC2, y tomar acciones cuando su uso sube por arriba del promedio. Es parte esencial de los grupos de Auto Scaling.
	AWS IAM (*Identity Access Management*)	Servicio que administra todas las cuestiones relacionadas con seguridad, accesos, usuarios, roles y grupos. Con esta herramienta se pueden configurar accesos a la consola web de AWS, otorgar permisos, restringir acciones, entre otras cosas.
	Amazon CloudFront	Servicio de CDN (*Content Delivery Network*), consiste en una serie de servidores colocados en regiones estratégicas, de tal manera que una petición de un recurso estático pueda ser atendida por el servidor más cercano al cliente, en vez de ir hacia el origen real de los recursos.
	Amazon S3 (*Simple Storage Service*)	Servicio de almacenamiento de objetos, comúnmente se utiliza para guardar archivos, sitios web estáticos, imágenes, entre otros. Básicamente se comporta como un Dropbox o Google Drive de tamaño casi-ilimitado y con excelente disponibilidad y durabilidad de datos (>99.99%).
	Amazon SES (*Simple Email Service*)	Servicio de envío de correo masivo. Permite enviar correos utilizando la infraestructura de Amazon a nombre de una cuenta de correo, y proporciona algunas estadísticas de rebotes, aperturas, etc.
	Amazon SQS (*Simple Queue Service*)	Servicio de colas de mensajes escalable. Permite enviar mensajes de hasta 256 Kb, garantizando la recepción de todos los mensajes enviados a las colas creadas, y permitiendo mantenerlos hasta 14 días, con un número ilimitado de lecturas.
	Amazon RDS (*Relational Database Service*)	Servicio de Bases de Datos Relacionales, permite crear y configurar fácilmente una instancia exclusiva para bases de datos relacionales, como MySQL, PostgreSQL, SQL Server, Oracle y MariaDB. Gestiona automáticamente licencias, respaldos, actualizaciones, réplicas, etc.
	Amazon ElastiCache	Servicio que ofrece un sistema de caché en memoria escalable. Actualmente, permite elegir entre una configuración en cluster de Memcache o Redis, ambos siendo sistemas de almacenamiento de llave-valor en memoria, por lo que son mucho más rápidos que una BD normal.

	Amazon DynamoDB	Servicio que ofrece una base de datos no relacional (NoSQL) de baja latencia y alta velocidad de respuesta a cualquier escala, con mínimo 6 réplicas de los datos distribuidos en 3 zonas de disponibilidad.

Tabla 3: Servicios comunes de AWS

Debido a que todos estos servicios tienen distintos enfoques, y por tanto distinto modelo de costos, era necesario desarrollar un plan que hiciera el uso más eficaz y eficiente posible de las funcionalidades que provee AWS, teniendo todos los esquemas de réplicas y alta disponibilidad necesarios para el negocio con el menor costo posible.

Este plan requería contemplar todos los recursos requeridos por el área de sistemas, para todos los proyectos. Esto significa que se contemplaron cerca de 80 instancias EC2, 20+ bases de datos en RDS, y varias configuraciones de réplicas para Bases de Datos, Balanceadores de Cargas con ELB y configuraciones de Auto Escalamiento con Auto Scaling Groups.

Metodología de desarrollo

Para iniciar el proyecto, primero se realizó un análisis de la infraestructura actual, y de su nivel de alta disponibilidad/tolerancia a fallos. Por razones de confidencialidad no es posible divulgar la configuración exacta de la infraestructura, ni el análisis realizado, pero sí es posible mostrar un resumen de los resultados. En general se encontró lo siguiente:

- Bases de datos únicas por proyecto: significa que se tenía un único punto de falla para cada uno de los proyectos, en caso de pérdida o corrupción de datos no se contaba con réplicas listas para su uso (*hot-swap*), y ningún método de conmutación a una instancia sana en caso de error. En cuanto a respaldos de las bases, se encontraron resultados mixtos: mientras que en algunos proyectos los respaldos se realizaban de manera diaria, manteniendo un historial de transacciones que permitiría restaurar una base al momento exacto en el que falló, hasta aquellos cuyos respaldos se hacían diario a una hora preestablecida, sin guardar el historial de transacciones, por lo que la restauración podía verse comprometida. En el mejor de los casos, podría restaurarse hasta 1 minuto antes de la pérdida, en el peor, se perdía todo un día de trabajo.
- Bases de datos generadas en servidores de uso múltiple en vez de servidores dedicados: una instancia de base de datos corría en un servidor de uso múltiple, donde se tenían alojados más sistemas, lo cual contribuía a una baja eficiencia en uso de recursos. En este caso, se tenía un servidor corriendo en una instancia EC2 en vez de RDS, que sería lo ideal para ese tipo de base, por la versión y solución de base de datos utilizada, y porque así es mucho más sencillo configurar réplicas y esclavos de sólo lectura.
- En cuanto a servidores de aplicación, se encontró que todos los sistemas salvo 1 tenían una configuración apropiada de alta disponibilidad (al menos contaban con 2 servidores, detrás de un balanceador de cargas), sin embargo, la mayor parte del tiempo sólo se utilizaba un 5% del CPU de las instancias, aproximadamente. Gracias a esto, se podía lograr un ahorro significativo del gasto

en instancias de servidores de aplicación disminuyendo el tamaño y poder de procesamiento de las instancias, en general, y sólo creando instancias extra cuando la demanda así lo requiriera.
- Una gran cantidad de instancias de servidores de aplicación (mayor al 50% de ellas) corría sobre hardware de generaciones pasadas, puesto que cuando se crearon era lo más actual. No existe ningún riesgo ni hay una necesidad inmediata de migrarlas, pero se puede conseguir un aumento en capacidad de procesamiento y memoria al hacerlo, además de bajar el costo por hora de la instancia, ya que las instancias equivalentes de última generación son más baratas.

Etapas del proyecto

Este proyecto consistió en 3 etapas principales:

- Recopilación de información sobre infraestructura existente: se realizó un compendio de todas las instancias, bases de datos, balanceadores de cargas, dominios, y demás usos de servicios de Amazon Web Services que se estaban realizando en el momento. Para hacer más sencilla la recopilación de datos, se utilizó la función de exportación disponible desde la consola de AWS.
- Análisis de infraestructura existente: Con la información recopilada se procedió a armar la estructura de costos y compararla contra los recibos mensuales recibidos, apreciándose un aumento constante de los costos de operación. Por lo tanto, se decidió analizar cuál sería la infraestructura más eficiente, estable y tolerante a fallos que se puede conseguir al menor costo.
- Presentación de resultados y recomendaciones: Los resultados del análisis se dieron a conocer en una reunión especial de infraestructura y visión a futuro de la empresa. En esta misma reunión se dieron las recomendaciones y se comenzó la planeación de migraciones de infraestructura.

Resultados

Con estos resultados preliminares se decidió hacer la propuesta de mejora en los siguientes aspectos:

- Bases de datos colocadas en el servicio adecuado para ellas, RDS, configuradas con Multi-AZ (*Multiple Availability Zones*, significa que Amazon mantiene en operación en todo momento una segunda instancia de base de datos igual a la original en una zona de disponibilidad distinta, lista para funcionar como la instancia original en caso de que algo le suceda a la primera), con *Provisioned IOPS* (Operaciones de Entrada-Salida Aprovisionadas por segundo, que permite tener una velocidad muy alta y estable de lectura y escritura a disco), con ventanas de mantenimiento semanales y respaldos automáticos una vez al día con historial de transacciones al segundo. Para dos bases de datos se configuró además una réplica de sólo lectura, una copia fiel de la base de datos principal, replicada de manera asíncrona, a la cual se pueden hacer consultas de forma independiente de la principal. Esto significa que la principal puede mantenerse para la operación del sistema, y que todas aquellas áreas que requieran un acceso de casi tiempo real a la base de datos principal pueden realizar sus consultas en la réplica, ejerciendo así mucha menos carga a la base de datos principal.
- Hacer uso de los demás servicios de AWS para bases de datos no relacionales, por ejemplo, usando DynamoDB y ElasticCache. DynamoDB es una base de datos no relacional (NoSQL) diseñada por Amazon que ofrece muy bajo tiempo de respuesta (milisegundos) a cualquier consulta, y a casi cualquier escala. Esta base de datos podría utilizarse en vez de otras soluciones como Redis o MongoDB, utilizadas en algunos de los sistemas. Dado que DynamoDB ya provee

alta disponibilidad y réplicas, del lado de infraestructura no se requeriría mayor esfuerzo para garantizar el buen funcionamiento de las aplicaciones que requieran este tipo de bases de datos. ElastiCache es otro servicio de Amazon que permite crear clusters de cachés de memoria, encargándose de escalarlos, mantenerlos y gestionarlos. Un caché de memoria se utiliza principalmente para mejorar la respuesta de los sistemas, haciendo que los resultados de consultas normalmente pesadas se guarden en memoria, de tal forma que cuando se desee repetirlos, se extraiga el valor del caché en vez de hacer el cálculo de nuevo. De esta forma, se puede aumentar el tráfico de un servicio web de manera importante sin requerir más servidores de aplicación. Además, varios sistemas requerían de tener un servidor de caché siempre disponible, como se verá en la arquitectura final del sistema.

- Se estableció un tamaño base de instancia (t2.small – t2.medium, dependiendo del sistema) para servidores de aplicación (ver tabla 1), a partir de las observaciones de desempeño actuales de los servidores de cada sistema, con la consideración de que se mantendrán siempre entre un 10% y 30% de porcentaje de uso de CPU, la cual fue considerada como la métrica base para decidir si se requieren más instancias para servidores de aplicaciones, o si se requieren menos.
- Colocar las instancias de servidores de aplicación en configuraciones de auto-escalamiento, con un mínimo de 2 instancias siempre en servicio y un máximo de 5. El aumento de instancias será cuando se supere el 30% de porcentaje de uso de CPU por más de 5 minutos, y se aumentará en una instancia, hasta llegar a las 5. Esta configuración permite dar abasto en caso de rápidos aumentos, puesto que el límite superior de uso es bajo, y también permite ignorar fluctuaciones rápidas de flujo, con los 5 minutos como tiempo mínimo en el cual se supere el límite superior de uso de CPU.
- Migrar instancias de generaciones anteriores al actual (instancias corriendo en hardware menos actual que las actuales) a la última generación. Por lo general, el hacer esto trae un aumento de potencia de CPU y de memoria RAM, con una reducción en el costo por hora de las instancias.
- Todas las instancias actuales se están consumiendo bajo demanda, lo cual tiene un costo más alto del necesario. Dado que una gran cantidad de instancias deben estar siempre funcionando, Amazon permite reservar esas instancias con un pago inicial, cobrando mucho menos por hora de uso. Si estas instancias se usarán más de 6 meses, es lo más conveniente para bajar los costos. Lo mismo aplica para las instancias de base de datos.

La arquitectura final propuesta para bases de datos se puede resumir en la siguiente figura:

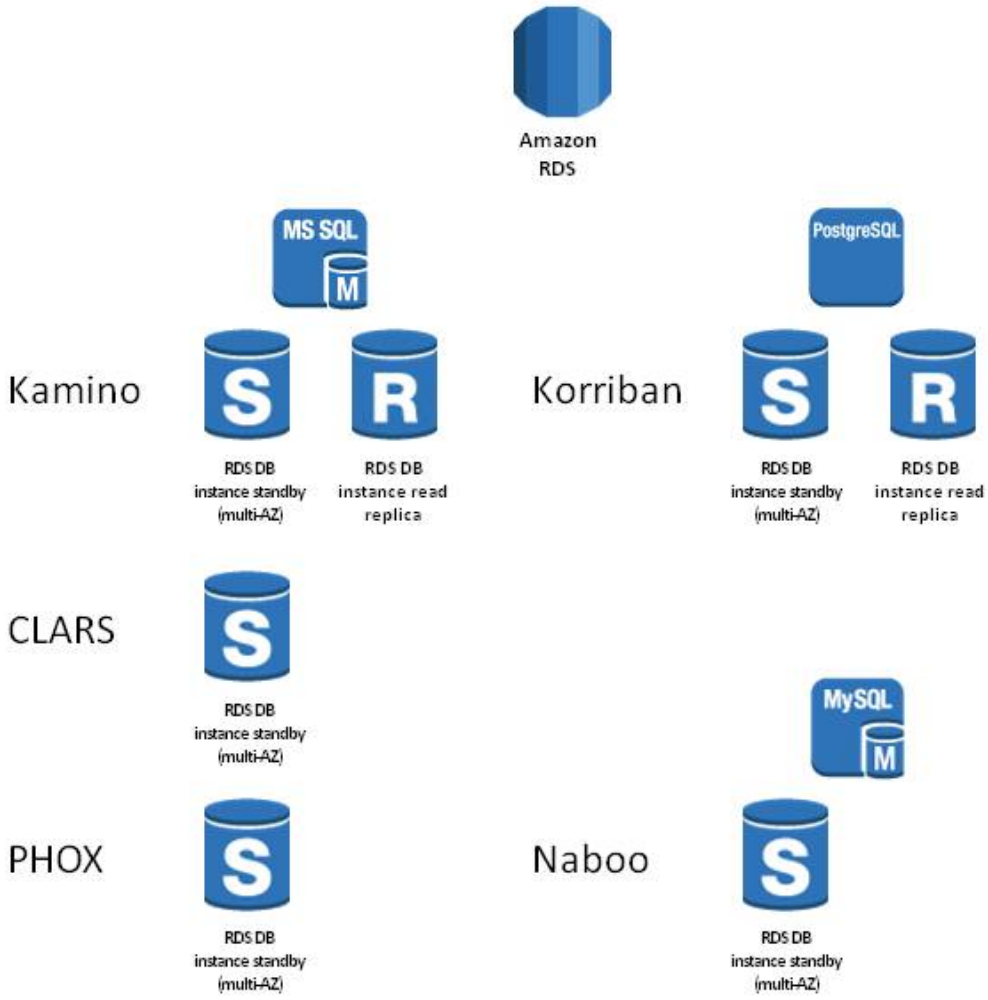


Ilustración 18: Arquitectura para bases de datos

La arquitectura final propuesta para los servidores de aplicación fue la siguiente:

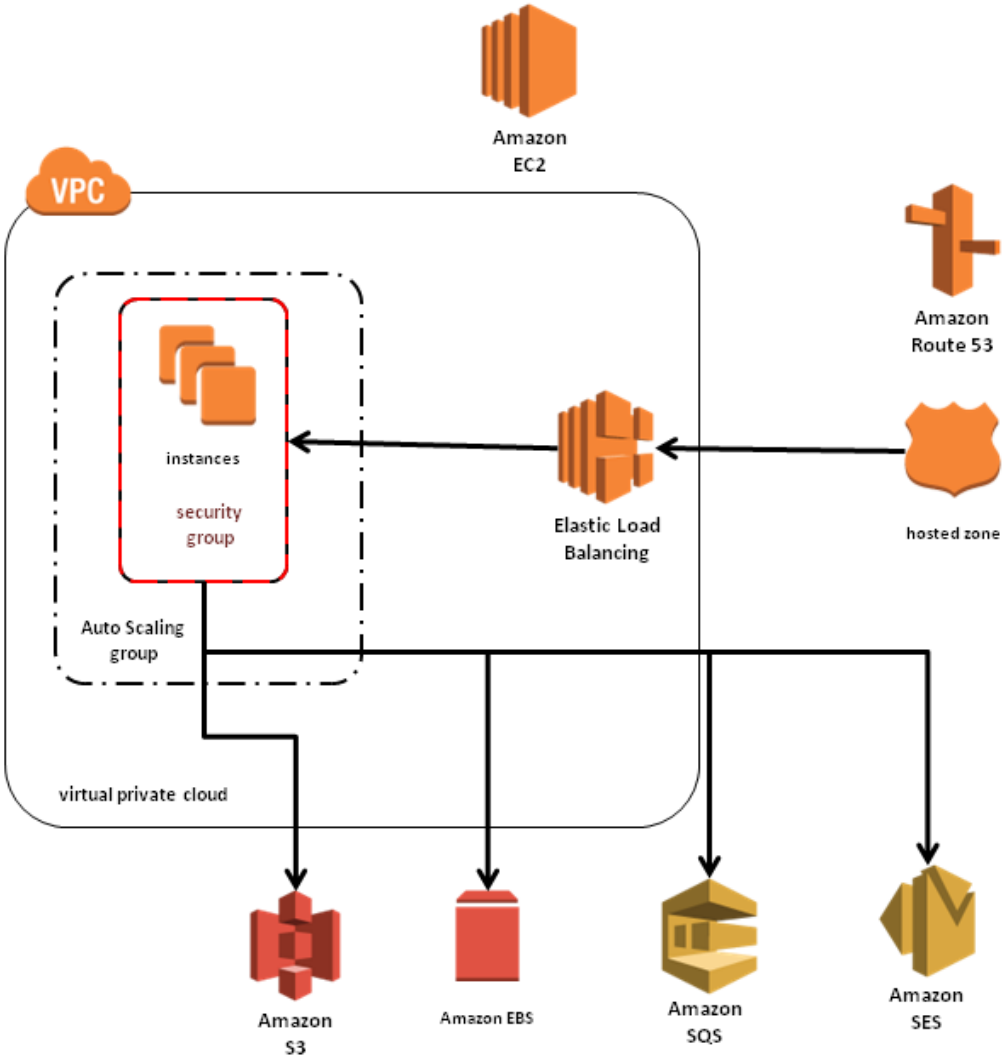


Ilustración 19: Arquitectura general de servidores de aplicación

Proyecto 2: Conexiones con servicios externos: Intermex

Definición del proyecto

Uno de los procesos que más logística requieren en el flujo de negocio es la entrega del préstamo al cliente. En la experiencia de la empresa, el método por el cual se entrega ha sido uno de los factores que más impacto tiene sobre el repago, es decir, sobre el porcentaje del préstamo que se logra cobrar al cliente.

Por estas dos razones se comenzó a explorar vías alternas de dispersión, es decir, la entrega del préstamo al cliente, y se determinó que el servicio de dispersión ofrecido por la empresa Intermex era de particular interés para el negocio, debido a que lo único que se requería, desde el punto de vista logístico, era conectar nuestra plataforma de ERP, "Kamino", al sistema propio de Intermex mediante el cual se podía realizar una orden de dispersión indicando el nombre de la persona, el monto, y la sucursal de Intermex a la que acudiría la persona a recoger el préstamo.

Intermex es una empresa que se dedica principalmente a comercializar transferencias y remesas, por lo que tiene una red de sucursales bastante amplia a lo largo del país, principalmente en pueblos y lugares a los cuales la cobertura bancaria aún es rudimentaria, por lo que es una excelente opción de dispersión para aquellas zonas donde no hay otras formas de hacerle llegar el dinero a los clientes.

Dado que este método de dispersión requiere mínima intervención humana, se optó por automatizar todo el proceso de envío de orden de dispersión, y monitoreo posterior para confirmar el retiro por parte del cliente, cancelaciones, o rectificaciones de parte de Intermex.

Metodología de desarrollo

Este proyecto consistió en gran medida de desarrollo de un módulo nuevo de software, por lo que se siguió la metodología estándar de desarrollo de producto, realizando de manera paralela dos funciones:

- Análisis de documentación técnica, desarrollo y pruebas iniciales del servicio web expuesto por Intermex para realizar las dispersiones.
- Interfaz de usuario mediante la cual se podían realizar consultas de órdenes enviadas, buscando conocer el estado de la dispersión, historial de movimientos, etc. Esta interfaz debía ser un módulo adicional a Kamino, y mostrar la información de los préstamos listos para ser dispersados.

Para el desarrollo de ambas secciones se conformó un equipo de trabajo, conformado por Christian Amezcua y Mario Aldama como desarrolladores, mientras que yo fungía el rol de líder de proyecto.

Etapas del proyecto

El proyecto inició con una reunión multidisciplinaria entre los contactos técnicos de Intermex y el equipo de trabajo, definiendo las características principales de los servicios web expuestos por su sistema, tomando nota de las recomendaciones, tecnologías, protocolos, terminologías y procesos que realiza su plataforma.

En términos generales, se encontraron las siguientes características del sistema de Intermex:

- Servicios web expuestos mediante el protocolo SOAP, con una conexión SSL necesaria entre nuestro cliente y su servidor,
- Dos ambientes: uno de desarrollo, para el cual no se requiere utilizar una VPN, y el de producción, para el cual se requiere utilizar una VPN de un proveedor externo que les provee esas soluciones a su empresa,
- El contenido de cada mensaje se envía utilizando el formato JSON en vez de XML,
- Se cuenta con servicios de dos tipos: aquellos que crean una nueva dispersión o modifican alguna existente, y aquellos que se usan para consulta del estado de las solicitudes realizadas anteriormente.

Con estos requerimientos técnicos, la primera decisión que se tomó fue acerca de cómo sería la arquitectura general del sistema. Se tenían dos posibilidades:

1. Construir un módulo nuevo dentro del sistema actual de CRM, Kamino. Esto implicaría un desarrollo en la plataforma .NET en la versión 2.5. Al momento de escribir este reporte, la versión actual y con soporte es la 4, con esto se tiene una idea de lo obsoleto del sistema actual, y el considerable esfuerzo de desarrollo necesario para crear este módulo. Dentro de los pros, todo el equipo tiene al menos un poco de conocimiento de la plataforma .NET, y ya habían trabajado anteriormente con Kamino.
2. Construir un sistema completo que se encargue de la conexión VPN con Intermex, y encapsule los detalles de los servicios web, exponiendo solamente unos servicios web lo más simples posibles y orientados por completo a la creación de dispersiones. Esto tiene las ventajas de que cuando se termine el desarrollo del nuevo sistema CRM, Korriban, este módulo podrá utilizarse también con el nuevo sin necesidad de desarrollos extras y permite encapsular en los servidores de este sistema las configuraciones de conexión de la VPN, sin embargo, requeriría sus propias bases de datos y servidores, lo cual aumentaría el costo mensual del servicio.

Dado que la construcción del nuevo sistema CRM/ERP, Korriban, ya estaba en proceso, se decidió construir el sistema completo de conexión con Intermex, ya que el costo mensual de los servidores en AWS sería menos de 50 USD mensuales, costo que palidece con los costos mensuales de toda la demás infraestructura (por cuestiones de seguridad, es imposible divulgar los detalles exactos de facturación, pero esa cantidad está por debajo de la media en cualquier implementación de alta disponibilidad).

A continuación se establecieron los detalles técnicos del proyecto, mismos que se especifican en la siguiente tabla:

Rubro	Descripción	Elección Final	Razonamiento
Lenguaje de desarrollo	El lenguaje de programación en el que se desarrollaría el sistema completo. Se requería un lenguaje sencillo, con soporte para consumir servicios SOAP y exponer	Python 2.7	Python es un lenguaje fácil de aprender y de desarrollar, maduro, con amplias colecciones de

	servicios REST de manera sencilla		librerías para realizar casi cualquier función.
Framework de desarrollo web y de servicios	Es una colección de librerías que realizan tareas comunes de servicios web, como autenticación, formularios, protecciones contra varios tipos de ataques, conexión con bases de datos, conexión con sistemas de caché, etc.	Django 1.8 + Django REST Framework	Django es un framework web maduro y sencillo de aprender y utilizar, y librerías como Django REST Framework permiten generar serializadores, modelos y servicios web de una manera muy rápida y sencilla.
Software de servidor de aplicaciones para el lenguaje de programación	Se refiere al sistema que ejecuta el código desarrollado de manera nativa.	Gunicorn	Gunicorn es el servidor de aplicaciones más sencillo para Python, y aunque no gana en rendimiento, sí gana en sencillez de configuración.
Software de servidor de contenido estático	Es el software que recibe las peticiones y decide qué debe hacerse con ellas. En caso de ser una petición a un sitio web, redirige la petición al servidor de aplicaciones, en caso contrario, regresa el archivo solicitado.	Nginx	Nginx es un servidor de contenidos muy robusto y extremadamente rápido.
Tipo de instancias EC2 de AWS	Se refiere al tipo de servidor adecuado para ejecutar el código del sistema.	t2.small	Estas instancias tienen 1 procesador virtual y 2 GB de RAM. Es más que suficiente para correr el código.
DBMS (*Database Management System*)	Es la capa de persistencia de datos de todo sistema: básicamente el lugar donde se guardan y de donde se consultan.	PostgreSQL 9.5	PostgreSQL es la base de datos de código abierto más recomendada por sus características, estabilidad y robustez.
Estrategia de despliegue de código	Esta estrategia se refiere a la forma en la que se actualiza el código en los servidores de producción cuando se hacen cambios o se añaden características al sistema.	Fabric + Git	Fabric es una librería que permite ejecutar comandos remotamente en uno o varios servidores a través de Secure Shell, o SSH.
Sistema de control de versiones	Se refiere al sistema que mantiene una historia completa de los archivos del proyecto, incluyendo los cambios realizados por cualquier miembro del equipo a cualquiera de los archivos.	Git	Ya se cuenta con un sistema que permite utilizar git como SCV, es robusto y distribuido, además de ligero en recursos y fácil de configurar y usar.

Pruebas	Para verificar el correcto funcionamiento del programa, se requiere realizar pruebas automáticas. Cada vez que se hace un cambio, se verifica que no rompa nada corriendo estas pruebas.	Django Unit Test	Las pruebas unitarias permiten definir escenarios de pruebas y así confirmar la exactitud del código escrito.

Tabla 4: Detalles técnicos del proyecto

Una vez establecidos los anteriores detalles técnicos, se procedió a codificar el sistema, constando de la conexión SOAP con los servidores de Intermex en el ambiente de pruebas. Una vez se codificaron todas las llamadas y sus respectivas pruebas, y se comprobó que se reflejaban las solicitudes correctamente del lado de Intermex, se desarrollaron los servicios internos que consumirían tanto Kamino como Korriban. Debido a la naturaleza interna de este desarrollo, no es posible especificar las firmas exactas de los servicios internos, ni las firmas de los servicios de Intermex, pero podemos describir superficialmente los tipos de servicios web construidos:

- Crear una dispersión: genera una orden de dispersión en el sistema de Intermex, requiere un monto y datos de identificación de la orden y de la persona a quien será entregada.
- Cambiar datos de la persona que recibirá la orden.
- Consultar nuevas órdenes entregadas.
- Confirmar que se registró correctamente la entrega de dichas órdenes en nuestro sistema.
- Cancelar una orden enviada previamente, antes de haber sido entregada.
- Consultar órdenes que fueron canceladas por causas de fuerza mayor, de parte de Intermex. Esto significa que ya se había marcado como pagada, pero ocurrió un error humano y no se debió haber marcado como tal.
- Confirmar que se registró correctamente la el cambio de pagada a en proceso debido al proceso anterior.

Resultados

Los resultados de este proyecto se vieron reflejados principalmente en un aumento de clientes nuevos a los que por primera vez se pudo entregarles un préstamo, puesto que previamente, por la zona geográfica en la que habitaban, no se contaba con los medios para hacérselo llegar.

Se realizó una prueba piloto con sucursales Intermex en la Ciudad de México, como comprobación de la eficiencia de entrega que ofrecían, y cuando fue satisfactorio, se aumentó la cantidad de préstamos entregados por este medio, aunque no se convirtió en el medio por defecto de entrega.

Debido a la arquitectura elegida para su construcción, cuando los primeros módulos de Korriban se terminaron de construir, integrar este método de dispersión junto a los otros fue muy sencillo y rápido, por lo que se consideró que las consideraciones y decisiones iniciales de arquitectura fueron correctamente realizadas.

Proyecto 3: Arquitectura, análisis, desarrollo y lanzamiento del nuevo sistema de manejo de clientes, contratos, pagos y solicitudes.

Definición del proyecto

Este es, sin duda, el más complejo y completo de los proyectos realizados hasta ahora. Consiste en varios servicios independientes, realizando cada uno una sola función, y que en conjunto permiten realizar todas las funciones de autenticación interna de usuarios, manejo de clientes, contratos, agendar entregas de préstamos, dispersiones y seguimiento de entregas. En total, las aplicaciones por diseñar fueron las siguientes:

- SSO (*Single Sign On*) con una implementación de OAuth2 para autenticación y autorización de usuarios internos, con soporte para búsquedas de usuarios, grupos y permisos por aplicación y por servicio.
- Adaptación y configuración de un API Manager para exponer los servicios o API externos de las aplicaciones desarrolladas.
- Capa de traducción de Kamino, el sistema actual monolítico realizado en .NET, que permite acceder a su información y los métodos expuestos mediante peticiones REST.
- Aplicación para agendar entregas y dispersiones, construida por completo por el equipo, de nombre código Korriban.
- Plataforma web como SPA (*Single Page App*) que une todos los servicios en un portal administrativo completo para los usuarios internos.

En el caso del SSO y de la capa de traducción de Kamino, se requería lograr una compatibilidad completa con Kamino, un sistema monolítico realizado en .NET de hace 10 años, y que exponía servicios de una forma exclusiva de la plataforma .NET, con *Windows Communication Platform* (WCF), es decir, no era ni SOAP, REST o algún formato o protocolo estándar. Por ello, se planteó la capa de traducción de Kamino a REST, que al realizarse en .NET podría comunicarse de manera transparente con los servicios ya expuestos, y a su vez exponerlos como servicios REST.

Para definir por completo los sistemas y las funcionalidades necesarias, se realizaron una serie de Diagramas de Modelado de Proceso de Negocio (BPMN), con los que se plasmó el funcionamiento actual de Kamino y el funcionamiento nuevo que se pensaba integrar, incluyendo toda la logística de entrega de préstamos que previamente funcionaba de manera muy básica, sin control de inventarios en forma. En esta etapa de definición de proyecto se decidió incluir varios cambios al proceso de negocio de entrega de préstamos. De acuerdo a varias pruebas de diferentes métodos de entrega vs porcentaje de repago realizadas anteriormente, se determinó que la manera en que se entregaban podría mejorarse, y para ello se realizó una alianza con una empresa de mensajería que, al tener sucursales en una gran cantidad de ciudades del país, permitía hacer entregas personales a los clientes. Esta etapa del proyecto fue la que más tiempo tomó para definir, puesto que aún no estaban claros los nuevos procesos de negocio, lo que provocó que hubiera muchos cambios durante el desarrollo, y que dichos riesgos fueran incluidos dentro del tiempo de desarrollo y los recursos necesarios para lograrlo del plan de proyecto.

Metodología de desarrollo

El proyecto se llevó a cabo en diversas fases: la primera fue la definición de los requerimientos funcionales de Korriban, que sería la parte que más lógica de negocio nueva implementaría. El desarrollo se comenzó una vez sabiendo cómo funcionaban esos módulos en el sistema anterior, Kamino, y se pospuso la planeación y la definición de los módulos nuevos hasta que estuvieran mejor definidos, dado que cualquier propuesta de desarrollo que pudiéramos hacer en ese momento sufriría grandes cambios al final.

Mientras tanto, otro miembro del equipo tomó la dirección del desarrollo del sistema de SSO, que no dependía de requerimientos nuevos o de módulos distintos, mientras que otro dirigía la construcción de la plataforma web que sería la cara del nuevo sistema a los usuarios. En todo este proceso, mi función principal fue asesorar y guiar a los miembros del equipo para lograr el mejor resultado posible, así como definir la arquitectura, lenguajes, protocolos de comunicación entre nuestros sistemas internos así como la comunicación necesaria con otros sistemas y otros equipos de desarrollo.

El equipo completo estaba formado por las siguientes personas:

- Adán Sánchez, Web Developer Sr. y líder de desarrollo de Korriban y la plataforma web.
- Alejandro Cuenca, becario y Web Developer Jr, líder de desarrollo del SSO.
- Néstor Velázquez, Web Developer y líder de desarrollo del API Manager.
- Alfonso Lizárraga (yo), Lead Architect, Project Manager y líder de desarrollo de capa de traducción de Kamino.

Construcción del proyecto

Una vez se distribuyeron las responsabilidades y se inició formalmente el proyecto, se acordó utilizar la metodología Ágil de desarrollo (Agile development), que puede constar de una metodología SCRUM o similares. La idea principal de estas metodologías es aceptar que no se conoce por completo la definición de un proyecto o producto, y que en vez de buscar definir todo desde un inicio, se maximiza la capacidad de cada equipo de entregar resultados en tiempos cortos, de tal forma que poco a poco se vaya definiendo y construyendo el producto a la vez. Además de esto, buscan maximizar la comunicación entre miembros del equipo, poniendo mucho énfasis en la solución de problemas en conjunto y en minimizar las distracciones como juntas innecesarias, siempre dando indicaciones de en qué problema está trabajando cada persona, así como las complicaciones o facilidades que encuentran al trabajar en él.

Gracias a estas metodologías, se comenzó el desarrollo de las partes funcionales, proceso que se detalla a continuación:

SSO (Single Sign On)

El *Single Sign On* se refiere al software que permite centralizar la autenticación (quién y con qué credenciales puede acceder) y la autorización (a qué recursos puede acceder cada quién) en un servidor

central, donde se pueden realizar todos los cambios, asignar usuarios a grupos y los permisos tanto específicos de usuarios como de grupos.

Además de todo esto, permiten que al iniciar sesión en uno de los sistemas interno de la compañía, al entrar a un segundo sistema interno, a pesar de que no compartan servidores físicos, memoria, códigos o procesamiento, no sea necesario iniciar sesión de nuevo, sino que la sesión creada al acceder al primer sistema interno sea reutilizada para el segundo sistema.

Para poder realizar todas estas funciones, la industria ha generado una gran variedad de protocolos de autenticación, entre ellos OpenID, OAuth, OAuth2, SAML, y extensiones específicas de XML para SOAP como WS-Trust y WS-Federation. Gracias a toda esta adopción por parte de la industria, se encuentran disponibles muchas soluciones comerciales de Servidores de Identidad que, entre otras funciones, ofrecen funcionalidades de SSO. Es por esto que el primer paso en el desarrollo de este sistema fue evaluar las soluciones existentes en la industria, y se evaluaron principalmente los siguientes sistemas:

- **WSO2 Identity Server**[11]: Este sistema de Gestión de Identidad forma parte de una suite de software empresarial pensado para organizaciones que siguen la arquitectura SOA, por lo que ofrece una gran cantidad de funcionalidad desde un inicio, y tiene la ventaja de ser software Open Source basado en Java. Entre la funcionalidad que ofrece está:
    - API para integrar con cualquier aplicación
    - Autenticación de múltiples pasos
    - SSO vía OpenID, SAML2 y OpenID Connect
    - Puenteo de SSO entre sistemas internos y aplicaciones en la nube.
    - Mapeo de credenciales a través de diferentes protocolos.
    - Auditoría vía XDAS.
    - Delegación de autenticación vía OAuth 1.0a, OAuth 2.0 y WS-Trust.
    - Federación de identidad vía OpenID, SAML2, OpenID Connect y Passive STS.
    - Integración con Microsoft SharePoint.
    - Páginas de inicio de sesión personalizable.
    - Soporte para múltiples tipos de proveedores de usuarios, desde un LDAP interno o externo, Microsoft Active Directory o cualquier base de datos con conector JDBC.
    - Soporte para más de un proveedor de usuario
    - Políticas de contraseñas y bloqueo de cuentas.
- **Gluu**[12]: Este servidor de identidad, también basado en Java, ofrece un portafolio menor de funcionalidad pero tiene documentación mucho más accesible. Entre las funcionalidades que ofrece están:
    - SSO configurable mediante SAML 2.0 y OpenID.
    - Administración de acceso mediante UMA y OAuth 2.0 para asegurar API y centralizar políticas de autorización.

[11] https://docs.wso2.com/display/IS510/Single+Sign-On
[12] https://www.gluu.org/

- Autenticación de múltiples pasos (*Multi-factor authentication*).
- Integración con fuentes de usuarios como Active Directory y servidores LDAP.
- Interfaz de administración de usuarios y grupos.
- Procesos de registro para nuevos usuarios.

Dadas la robustez y abundancia de funcionalidad ofrecida por las soluciones anteriores, se evaluaron a profundidad para conocer las limitantes que ambas plataformas ofrecen, y cuál de ellas, si es que alguna, era la mejor opción a largo plazo para la empresa.

La evaluación del WSO2 Identity Server nos llevó a la conclusión de que es un software muy robusto y útil principalmente para empresas grandes con requerimientos muy específicos de autenticación y autorización. Sin embargo, para una empresa relativamente pequeña como la nuestra, tenía muchísimas más funcionalidades de las necesarias, y la documentación para poder utilizar aquellas que sí eran necesarias era deficiente cuando mucho. Parecía estar diseñada para que se requiriera uno o más talleres de uso del software, que se consideraron demasiado caros para el beneficio añadido. Por estas razones se desechó la idea de utilizar esta solución.

En el caso de Gluu, se determinó que, a pesar de tener mejor documentación, no todo funcionaba tan sencillamente como se mencionaba, y los planes de soporte de 70 mil dólares anuales dejaban claro que, en caso de tener problemas con el servidor, no era un asunto sencillo restablecerlo. Debido a esto, también fue desechado como posible solución.

Esto significa que al final, la solución más viable económica y técnicamente para la empresa era realizar nuestro propio servidor de identidad. Este servidor sólo implementaría uno de tantos protocolos, OAuth 2.0, y el API para consulta de usuarios, grupos y permisos sería diseñado por nosotros, así como su implementación. El nombre código interno para este proyecto fue *Saleucami*.

El desarrollo se realizó utilizando las siguientes herramientas:

Rubro	Descripción	Elección Final	Razonamiento
Lenguaje de desarrollo	El lenguaje de programación en el que se desarrollaría el sistema completo. Se requería un lenguaje sencillo, con soporte para exponer servicios REST de manera sencilla.	Python 2.7	
Framework de desarrollo web y de servicios		Django 1.9 + Django REST Framework	Alejandro Cuenca, el líder de desarrollo de este proyecto, tenía mucha más experiencia utilizando este framework y lenguaje de programación que cualquier otro.
Librería de autenticación	La librería que permite realizar tareas de autenticación	Django-oauth2-	Esta librería ya tiene integrados los

		provider	procedimientos técnicos y de seguridad (encriptación, etc) necesarios para poder realizar autenticación de usuarios mediante OAuth2
Servidor de aplicaciones	Se refiere al sistema que ejecuta el código desarrollado de manera nativa.	Gunicorn	
Software de servidor de contenido estático		Nginx	
Tipo de instancias EC2 de AWS		t2.micro	Estas instancias tienen 1 procesador virtual y 1 GB de RAM. Es más que suficiente para correr el código.
DBMS		PostgreSQL 9.5	
Estrategia de despliegue de código		Fabric + Git	
Sistema de control de versiones		Git	
Pruebas		Django Unit Test	

Tabla 5: Detalles técnicos de Saleucami

Además de estas características, dado que la información del usuario, como nombre y apellido, correo electrónico institucional y permisos rara vez cambia, se agregó una capa de caché para volver lo menos frecuentes posibles las lecturas a la base de datos. Para este caché se eligió utilizar Memcached, y posteriormente Redis dadas las facilidades que da AWS con el servicio de ElastiCache, que genera un cluster de instancias de Redis de manera transparente para nuestra aplicación.

API Manager

El API Manager se refiere al software que controla el acceso a las API (servicios) de las aplicaciones internas de la empresa. Dado que la interfaz de usuario sería una aplicación de una sola página, accesible mediante un navegador, era particularmente importante mantener seguras de accesos no autorizados a las aplicaciones que exponían la información sensible de clientes o usuarios internos.

El control de acceso se realiza de acuerdo a tokens de autenticación y revisión de permisos de usuarios, además de restringir el uso de los APIs registrados a cierta frecuencia de consulta por minuto u hora, asegurando así que nadie pueda utilizar estos servicios de manera excesiva, causando así cargas muy altas de trabajo. Todos los accesos realizados a cada API quedan permanentemente guardados y permite monitorear todas estas solicitudes.

En este caso también existen diferentes soluciones de código abierto que tienen esta funcionalidad, entre las soluciones evaluadas están:

- **WSO2 API Manager**[13]: Este sistema de administración de servicios es uno de los más completos que se encontraron. Cuenta con las siguientes funcionalidades:
    - Diseño y prototipado de APIs, desde la interfaz de publicación como desde una definición de API de Swagger 2.0.
    - Permite crear API de prueba (mock), que siempre regresan datos predefinidos, pero sirve para hacer un prototipado rápido de los clientes que consumen estos servicios.
    - Soporta publicar APIs SOAP y REST con formatos XML y JSON.
    - Soporta todos los tipos de autenticación de usuarios de API definidos por OAuth 2 (implícito, código de autorización, cliente, SAML, IWA Grant Type).
    - Portal de administración de API, con consola de pruebas.
    - Soporta transformación de protocolo, de datos y composición de API.
    - Mapea entre diferentes protocolos, como HTTPS y JMS o escritura directa a sistema de archivos.
    - Ofrece límite de uso de API y políticas por usuario y por API.
    - Soporte listo para WSO2 Data Analytics Server y Google Analytics.

- **Kong**[14]: Este servidor está construido sobre Nginx y Apache Cassandra, dos proyectos maduros y muy utilizados en la industria, por Mashape para el funcionamiento de cerca de 15,000 de sus APIs internos. Dentro de sus funcionalidades, están las siguientes:
    - API REST para su administración.
    - Soporte para Plugins: éstos hacen la mayor parte del trabajo, y contienen plugins desde Autenticación Básica, por llaves, OAuth2, logeo y monitoreo de tráfico, así como límite de peticiones por usuario y API.
- **AWS API Gateway**[15]: Este es un servicio de Amazon que ofrece los mismos servicios que las soluciones anteriores, sólo que es completamente manejado por Amazon en cuanto a escala.
- **Tyk**[16]: Ofrece dos soluciones: un API Gateway, con las funciones previamente expuestas, y un API Dashboard con su correspondiente API REST para realizar todas las funciones de administración de usuarios, API's, edición de API e importación utilizando Swagger, y autoconfiguración de APIs con soporte para CORS y Caché.

En este caso, las ofertas comerciales tenían todas las funcionalidades necesarias para este proyecto, por lo que se decidió utilizar alguna de ellas en vez de desarrollarlo internamente. Por un lado, gracias a la experiencia adquirida al probar el primer API Manager de WSO2, se decidió desecharlo debido a la alta complejidad de uso y deficiente documentación técnica. En el caso de Kong, el hecho de tener que instalar

[13] http://wso2.com/products/api-manager/
[14] https://getkong.org/about/
[15] https://aws.amazon.com/es/api-gateway/
[16] https://tyk.io/

plugins para realizar todas las tareas le daba una sensación de flexibilidad, pero también de mayor riesgo al poder tener problemas de compatibilidad entre versiones de plugin, etc. Además, la falta de una interfaz gráfica de la cual obtener reportes para los directores fue una razón fuerte para desecharlo. En cuanto al API Gateway de AWS, la experiencia de uso era mucho menos que intuitiva: se tenía que dar de alta cada servicio de cada aplicación, ¡manualmente!

Esto, y el hecho de que Tyk fuera un proyecto de código abierto con impecable documentación, nos llevó a elegirlo como la solución para API Management. Tyk es un proyecto realizado en el lenguaje Go[17], un lenguaje pensado para ser altamente eficiente en ambientes multiprocesos y multihilos, que es justo el caso de uso de un API Manager. Ello y su interfaz amigable y sencilla hicieron mucho más rápida su adopción dentro de la empresa.

Capa de traducción de Kamino

Kamino es el nombre del sistema monolítico y obsoleto en el que se ejecutaban la mayor parte de procesos de negocio posteriores a que el cliente solicitara su préstamo. En él estaban integradas las funcionalidades de CRM y ERP, puesto que ahí se podía consultar y modificar la información de los clientes, de las solicitudes y de los contratos, así como realizar dispersiones y administrar los pagos y recargos.

El principal problema que presenta Kamino es la dificultad de desarrollo y de mantenimiento: no existía una sola persona que tuviera un conocimiento profundo del código, y el hecho de que tuviera cerca de 5 años de vida y originalmente hubiera estado diseñado para un fin distinto volvía aún más difícil hacer cambios de manera segura. Además de esto, la tecnología con la que fue construido databa de aún más de 5 años, por lo que exponer servicios mediante otra manera que no fuera la actual requería un considerable esfuerzo de lectura de documentación, que ya no se podía encontrar en internet debido a la antigüedad de la plataforma.

Debido a esto, y a que Kamino exponía varias funciones de negocio como Activar Contratos, Marcar Préstamo como entregado, entre otros, se eligió mantenerlo como el sistema núcleo, bajo todos los demás. Los servicios expuestos actualmente se modificarían y simplificarían, a fin de utilizarlo solamente como un registro y resguardo de actividad de negocio, quitándole lógica de dispersiones y de agendaciones y reagendaciones de entregas. Kamino entonces sólo sabría cuando un préstamo se agenda, cuando entrega y por qué método, y cuando debe activar un contrato.

Otro de los requerimientos 100% necesarios era mantener la compatibilidad con los reportes y sistemas de análisis existentes construidos sobre la base de datos de Kamino, lo que significa que no era posible realizar cambios drásticos en su estructura o eliminarlo por completo del proceso. Había que encontrar una manera de retroalimentar su base de datos mediante los servicios expuestos, mientras que la lógica de negocios se iba pasando poco a poco a Korriban. Dado que este desarrollo serviría como puente entre Kamino y Korriban, se bautizó con el nombre código *Korrino*.

[17] https://golang.org/

Dadas estas limitantes y requerimientos, se decidió construir con las siguientes herramientas:

Rubro	Elección Final	Razonamiento
Lenguaje de desarrollo	.NET Framework 4.5	Es la última versión de la plataforma .NET, la única que tiene soporte innato para WCF, el protocolo de comunicaciones expuesto por Kamino.
Framework de desarrollo web y de servicios	ASP .NET Web API 2	Este framework permitía exponer fácilmente funciones mediante llamadas REST, haciendo que fuera trivial escribir los métodos necesarios para exponer información de la base de datos o traducir los métodos de Kamino de WCF a REST.
Software de servidor de contenido estático	IIS (*Internet Information Services*)	Este servidor es el equivalente de Microsoft para Nginx ó Apache en el mundo de código abierto y Linux. Además, es el único que puede ejecutar código de .NET de manera nativa, y sólo está disponible en Windows.
Tipo de instancias EC2 de AWS	t2.small	Estas instancias tienen 1 procesador virtual y 2 GB de RAM. La plataforma .NET tiene requerimientos más fuertes de procesamiento, puesto que requiere de todo un ambiente Windows para funcionar.
DBMS	SQL Server	La misma base de datos que ocupa Kamino, puesto que de ahí se extraerá la información para exponerla mediante los servicios REST.
Estrategia de despliegue de código	AWS Elastic Beanstalk	Este es un servicio de Amazon que permite crear ambientes completos para ciertas configuraciones de software: se elige la plataforma, la versión, y el tipo de ambiente que se desea crear, y automáticamente genera un balanceador de cargas y varios servidores de aplicación, y permite subir nuevas versiones del código compilado y hacer despliegues de manera ordenada, de tal forma que no se pierde ninguna petición ni conexión al sistema aunque el despliegue se haga en ese momento.
Sistema de control de versiones	Git	
Pruebas	Integration Testing	Dado que no se contaba con la experiencia con una plataforma de pruebas fácil de utilizar para .NET, se desarrollaron pruebas de integración con los demás sistemas que permitían realizar pruebas también a Korrino.

Tabla 6: Detalles técnicos de Korrino

Korrino entonces es la capa de servicio que permite a cualquier otra aplicación consumir información de Kamino de manera coherente, tomando en cuenta todas las posibles malformaciones de datos debido al tiempo que ha estado en operación y la obsolescencia de sus componentes, y sobre todo, exponiendo una cara íntegra de la información central al resto de los sistemas.

Korriban

La estrella de este proyecto, Korriban, es el sistema que realizaría varias de las funciones críticas de la compañía, como entrega de préstamos, logística y seguimiento, además de tener un diseño modular que permitiera ir modificando o sustituyendo el código responsable de un servicio por uno nuevo. Este módulo se diseñó y construyó pensando en la arquitectura de Microservicios, manteniendo responsabilidades semejantes en la misma aplicación, y creando tantas aplicaciones como responsabilidades separadas había.

Debido a detalles de confidencialidad, no me es posible detallar por completo el funcionamiento interno de este sistema, pero puede construirse un flujo básico de proceso con el que se ilustre qué partes del negocio tocaba:

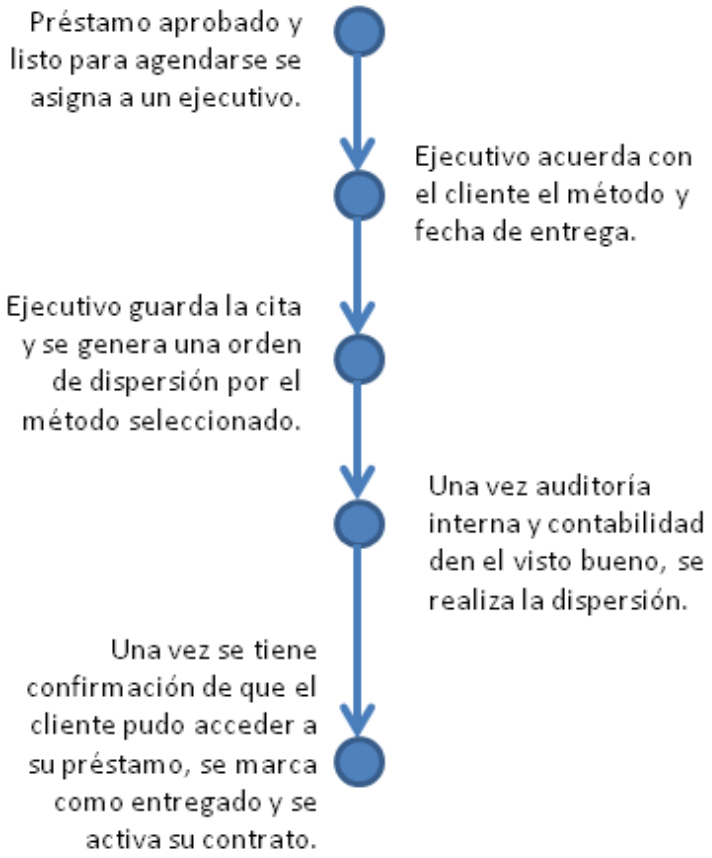


Ilustración 20: Proceso de Negocio de Korriban

Sin entrar en detalles, existe lógica de negocio que indica qué métodos de dispersión están disponibles para cada cliente al agendar, así como los cambios posibles entre métodos de dispersión de acuerdo a las

pruebas realizadas para mejorar el repago en la empresa. Además, entre los métodos de dispersión disponibles, cada uno requiere un proceso radicalmente distinto a los demás. Uno de ellos es el anteriormente expuesto Intermex, que al ofrecer una API de conexión es el más amigable para desarrollo.

Las tecnologías elegidas para este desarrollo fueron las siguientes:

Rubro	Elección Final	Razonamiento
Lenguaje de desarrollo	Python 2.7	Se requería un lenguaje sencillo, con soporte para exponer servicios REST de manera sencilla, con soporte para exportaciones en diferentes formatos como Excel, CSV, etc.
Framework de desarrollo web y de servicios	Django 1.9 + Django REST Framework	Se eligió por la experiencia del equipo en este lenguaje y framework, su soporte para operaciones asíncronas (como consultas de estado de dispersiones con sistemas relacionados), y su facilidad de creación de filtros, búsquedas y exponer servicios altamente personalizados.
Aplicación de ejecución de tareas asíncronas	Celery + Flower + AWS SQS	Celery es un software que permite compartir código con Django, permitiendo así escribir módulos de tareas asíncronas que puedan acceder a la base de datos y demás lógica de negocio contenida en los servicios y hacer uso de ella para procesos asíncronos de negocio. Flower es un software que permite monitorear los procesos de trabajo de Celery, que es donde se ejecutan las tareas. Amazon SQS es un servicio de mensajería de colas, donde se envían mensajes a un servidor central y cada cliente se encarga de consultarlo para saber si hay mensajes nuevos en los canales a los que están suscritos, y de ser así, procesarlos. Celery utiliza esta solución de mensajería para enviar los mensajes de tareas asíncronas a realizar entre el código síncrono web y la nube de procesos de trabajo.
Servidor de aplicaciones	uwsgi	uWSGI es el servidor de aplicaciones más robusto para código en Python, su configuración es mucho más flexible pero complicada, y soporta un reinicio sin necesidad de perder peticiones o estar no disponible.
Software de servidor de contenido estático	Nginx	
Tipo de instancias EC2 de AWS	t2.small	Estas instancias tienen 1 procesador virtual y 2 GB de RAM. El código de los servicios requiere menos recursos, pero Celery necesita mucha RAM para su funcionamiento óptimo (muchos procesos paralelos en vez de pocos).
DBMS	PostgreSQL 9.5 en AWS RDS con Multi-AZ Deployment y un esclavo de sólo lectura para reporteo.	

Estrategia de despliegue de código	Fabric + Git
Sistema de control de versiones	Git
Pruebas	Django Unit Test

Tabla 7: Detalles técnicos de Korriban

Plataforma web como SPA

Hasta el momento se contaba con una colección de servicios: autenticación, dispersiones, agendación, actualización de contratos, administración de clientes, administración de usuarios, grupos y permisos, entre otros. Sin embargo, hacía falta una plataforma en la que todos estos servicios se conjuntaran para crear un portal interno que los ejecutivos de cada área pudieran acceder y utilizar.

Para este fin se diseñó una plataforma web, al estilo de SPA (*Single Page App*). Estas páginas tienen la particularidad de que están construidas por completo en Javascript y se ejecutan por completo en el navegador. Esto significa que una vez cargada la aplicación, ésta se encarga de mantener el estado interno, controlando cuándo y qué tipo de datos necesita cargar en cada sección. Un cambio entre páginas, por ejemplo, solamente requeriría una consulta de la plantilla de la página a cargar, sin datos, y luego una posterior a alguno de los servicios expuestos que permitiría "rellenar" la plantilla con los datos obtenidos del servicio. Este tipo de aplicaciones ha sido muy recurrido en los últimos 6-8 años por las grandes empresas, puesto que permite construir funcionalidad modular y una interfaz de usuario mucho más agradable que antes, puesto que permite una interacción mayor al no haber cargas completas de páginas entre secciones de la aplicación.

Por lo común, este tipo de aplicaciones son desarrolladas específicamente por ingenieros dedicados al área de Front-End, sin embargo en nuestro caso, al no contar con un experto en esta área, todos los miembros del equipo elegimos y aprendimos un framework Javascript para realizar este tipo de aplicaciones. El nombre código para este desarrollo fue Korriban Front-End.

Las tecnologías elegidas para este desarrollo fueron las siguientes:

Rubro	Elección Final	Razonamiento
Lenguaje de desarrollo	Javascript, con ES6	Para construir código que se ejecute en un navegador, realmente no hay una alternativa a Javascript. Este lenguaje, formalmente conocido como ECMAScrip 6, es el único que se ejecuta de manera nativa en el navegador.
Framework de desarrollo web y de servicios	AngularJS 1.4	Angular JS es un framework que hace más sencillo construir SPA con Javascript, desarrollado por Google y con una gran cantidad de documentación y tutoriales. Es uno de los proyectos más recomendados para este tipo de aplicaciones, puesto que incluye módulos para todas las funcionalidades comunes de una SPA, como ruteo, historial, ataduras de doble vía (*two-way binding*), entendido como que si se modifica un valor en memoria, se modifica en la

		plantilla, y viceversa, así como facilidades para consumo de servicios REST y para pruebas unitarias y de integración.
Software de servidor de contenido estático	AWS S3	Amazon ofrece un servicio llamado Simple Storage Service que, además de fungir como almacenamiento para archivos, también permite hostear un sitio estático desde él. En el caso de las SPA, el código debe ser descargado por completo al navegador del cliente, por lo que es 100% estático y por tanto, mucho más conveniente de distribuir por este medio, al no necesitar servidores o contenedores especiales.
Estrategia de despliegue de código	SFTP	S3 permite varios métodos de carga de archivos, pero el más sencillo es utilizar SFTP (*Secure File Transfer Protocol*).
Sistema de control de versiones	Git	
Pruebas	Karma + Protractor	Karma es una herramienta que permite automatizar la ejecución de pruebas unitarias, mientras que Protractor permite realizar pruebas de Inicio a Fin (*End2End Testing*), garantizando que toda la aplicación sea consistente, desde la página de inicio de sesión hasta la última acción posible a realizar.

Tabla 8: Detalles técnicos de Korriban Front-End

Resultados

Este proyecto fue el más grande de los que se presentan en este reporte. Su diseño, construcción, arquitectura, pruebas y despliegue ocupó un equipo completo la mayor parte de 5 meses de trabajo y comunicación constante con las demás áreas, para procurar requerimientos, verificar conexiones, protocolos, y flujos de negocio. La influencia que tuvo sobre la empresa es difícil de minimizar, puesto que para el área de desarrollo y sistemas fue el primer paso hacia una verdadera organización tecnológica basada en microservicios y SOA, y para el área operativa una gran mejora y estandarización de procesos de negocio que antes estaban sin un correcto monitoreo y rastreabilidad, gracias a todas las verificaciones de datos y flujos de cambios de estado de dispersiones y contratos que se codificaron.

En el caso del servidor de identidad, o SSO, posteriormente se le agregó un módulo bastante grande para poderlo utilizar para autenticar no sólo a los usuarios internos de la compañía, sino también a los clientes a través del sitio web y de una aplicación móvil Android desarrollada por otro equipo. Realizando mediciones de rendimiento, se encontró que con el caché añadido se lograron tiempos de respuesta de 20-30ms constantemente, haciendo que la experiencia de inicio de sesión fuera impresionantemente rápida.

En cuanto al API Manager, el hecho de construir todos nuestros servicios utilizando REST simplificó en tal medida la necesidad de transformación y enriquecimiento entre servicios que influyó de manera significativa en la decisión posterior de no utilizar un ESB interno para comunicarnos entre servicios. Eventualmente, se planea que todos los servicios de las demás aplicaciones (sitio web de clientes, aplicaciones móviles, etc) se agreguen al API Manager.

La capa de servicios entre Kamino y Korriban fue la decisión más controversial, dado que había una presión muy fuerte de parte de la gerencia para eliminar la presencia de Kamino en esa etapa del proceso. Sin embargo, dado que las métricas necesarias por la dirección para conocer el estado de la empresa dependían al 100% en los datos y la estructura de Kamino, era prácticamente imposible quitar la dependencia con Kamino sin comprometer gravemente la estructura de reporteo y analíticos ya construida. Se acordó realizar un plan de migración para las áreas de Riesgo y Business Intelligence, para que pudieran actualizar sus fuentes de datos a Korriban en vez de Kamino en un futuro.

Finalmente, se tuvieron varios problemas fuertes con la implementación de Korriban: dado que se habían agregado candados de seguridad en cada paso del proceso de negocio, los procesos que antes se realizaban de manera básica en Kamino y que pasarían a Korriban requerían tener los datos históricos para su correcto funcionamiento. Sin embargo, cerca del final del desarrollo, salió a la luz que muchos de esos datos históricos no tenían suficiente información ni contexto para poder transformarlos de manera coherente en la nueva estructura de Korriban, por lo que hubo necesidad de realizar una gran cantidad de revisiones manuales y, al final, incluso aceptar que no se tendrán los datos completamente fiables y que será necesario intervención manual en ciertos casos exóticos, donde los estados de los clientes no concuerden con los registros históricos disponibles. Fuera de este problema, se logró mejorar en gran medida la eficiencia de las áreas que ahora utilizan Korriban, haciendo mucho más rápido el proceso de agendación y mucho más agradable y directo el proceso de realizar las dispersiones.

La nueva plataforma web incluyó muchas mejoras de UI/UX (User Interface/User eXperience) con la asesoría del departamento de UX de la compañía, por lo que mejoró de manera increíble la facilidad y eficiencia con la que los ejecutivos realizaban los procesos. Además, esta interfaz fue construida de manera modular, haciendo muy sencillo agregar otros procesos de negocio y sus correspondientes servicios a futuras versiones, marcando así un camino muy claro hacia la total desaparición de Kamino y una arquitectura de sistemas basados en microservicios.

Conclusiones

En conclusión, esta experiencia fue muy importante para mi desarrollo profesional debido a la gran cantidad de retos y de situaciones nuevas a las que me enfrenté, además de la escala de las mismas. En empleos anteriores los retos habían sido de aprendizaje, sobre todo: cómo y con qué herramientas construir la base tecnológica de la empresa, aprender a utilizarlas y construir una base bien diseñada y fácil de mantener para soportar las operaciones por un periodo largo de tiempo. En este caso, la experiencia consistió mucho más en evaluar los sistemas existentes, proponer mejoras, diseñar los nuevos sistemas y dirigir su implementación, por lo que me permitió crecer en muchas más dimensiones que solamente la dimensión técnica.

Uno de los principales retos a los que me enfrenté fue el problema de cómo estimular la innovación en una organización grande, con una cultura y procedimientos ya establecidos (y no precisamente siguiendo un plan ordenado, sino de acuerdo a las circunstancias). La cultura de una empresa no es algo fácil de abandonar: el cómo se hacen las cosas se convierte en una rutina, en tradición, y al momento de querer cambiar la manera de pensar, un sistema, o algún componente, las personas se lanzan a defender el modelo anterior y sólo a regañadientes y tras mucha labor de convencimiento logran aceptar las novedades. Para lograr una verdadera innovación, un cambio en el proceso o en el modelo de negocio, aprendí dos lecciones que parecen ser contradictorias:

- Buscar que la solución propuesta encaje bien en los procesos existentes, y permita que todas las demás áreas sigan funcionando con cambios mínimos. Este enfoque tiene la ventaja de que requiere muy poco esfuerzo y cambio de las demás áreas para funcionar, pero también significa que es el más limitado desde el punto de vista de la innovación, puesto que desde un inicio se está atado a lo que se tiene actualmente, y todo nuevo proceso se debe "esconder" tras una máscara que permita que las personas de la organización sigan trabajando utilizando la capacitación previa que han recibido, y manteniendo la nueva eficiencia alcanzada tras bambalinas. Este fue el enfoque que se siguió en el tercer proyecto, creando una capa de traducción entre el sistema nuevo y el antiguo, y manteniendo en operación ambos a la vez.
- Si la solución requiere un cambio significativo en los procesos o métodos ya establecidos de la empresa, lo más probable es que sea difícil de vender a los directores y de mantener a largo plazo, especialmente si es una solución que tomará varios meses (ni se diga años) para implementar. En casos de estudio de éxito realizados por la escuela de negocios de Harvard (ver bibliografía), lo que mejor probabilidades tiene de concretarse y generar un cambio es comenzar desde cero, con la misión específica de construir una base nueva, bien diseñada y duradera para ese mercado ya identificado.

En retrospectiva, concluyo que la mejor manera para que la empresa pueda mantenerse como líder en el campo es mediante la segunda estrategia que mencioné, dejando atrás todo el bagaje que tiene construido "como se pueda" por un área de sistemas organizada y altamente eficiente. Esta visión fue lo que más trabajé por mostrar a mis compañeros y jefes, y aunque no haya sido aplicada al 100%, creo que

lo construido por mi equipo durante mi tiempo en Mimoni ha seguido correctamente una visión modular y de mejora, aunque no un reemplazo total, sí un reemplazo selectivo y acumulativo con una dirección coherente y un diseño dirigido a un fácil mantenimiento y un cambio mucho más ágil del que se tenía previamente.

Bibliografía

Amazon Web Services, I. (2016). *Amazon Web Services*. Recuperado el 17 de Abril de 2016, de Amazon Web Services: https://aws.amazon.com/es/

Brewer, E. A. (2000). *Towards Robust Distributed Systems*. Obtenido de http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf

Brewer, E. (2012). *CAP Twelve Years Later: How the 'Rules' Have Changed*. Obtenido de http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed

Christensen, C. M. (1997). *The Innovator's Dilemma: When New Technologies Cause Great Firms to Fail.* Boston: Harvard Business Review Press.

Daigneau, R. (2012). *Service Design Patterns.* Massachusetts: Addison-Wesley.

Erl, T. (2005). *Service-Oriented Architecture, Concepts, Technology and Design.* Prentice Hall.

Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley Professional.

Fielding, R. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. Obtenido de http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

Fielding, R. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. Recuperado el 17 de Abril de 2016, de University of California, Irvine: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

Foundation, D. S. (2016). *Django: The web framework for profesionists with deadlines*. Recuperado el 17 de Abril de 2016, de Django: The web framework for profesionists with deadlines: https://www.djangoproject.com/

Foundation, P. S. (2016). *Python*. Recuperado el 17 de Abril de 2016, de Python: https://www.python.org/

George, F. (2012). *YOW Conferences Australia*. Obtenido de YOW!: https://yow.eventer.com/yow-2012-1012/micro-services-architecture-by-fred-george-1286

Google. (2016). *AngularJS*. Recuperado el 17 de Abril de 2016, de AngularJS: https://angularjs.org/

Hohpe, G. W. (2003). Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. En *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Addison-Wesley Professional.

Humble, J. F. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Addison-Wesley Professional.

JavaZone. (2014). *Micro-Services - Java, the UNIX way*. Obtenido de Vimeo: https://vimeo.com/74452550

Lamport, L. (1978). The Implementation of Reliable Distributed Multiprocess Systems. *Microsoft Research*
.

Microsoft. (2016). .*NET Framework*. Recuperado el 17 de Abril de 2016, de .NET Framework: https://www.microsoft.com/net/default.aspx

North, D. (2012). *Patterns of Effective Delivery*. Obtenido de Vimeo: https://vimeo.com/43659070

Anexos