Universidad de Lima

Facultad de Ingeniería

Carrera de Ingeniería de Sistemas

ENFRENTANDO LA TEORÍA Y LA REALIDAD EN PROYECTOS DE SOFTWARE : ASPECTOS BACKEND DEL PROYECTO QEMPO

Trabajo de suficiencia profesional para optar el Título Profesional de Ingeniero de Sistemas

Alejandro Manuel Uladislao Arevalo La Fuente

Código 20133296

Asesor

Jorge Luis Irey Nuñez

Lima – Perú

Marzo de 2022

ENFRENTANDO LA TEORÍA Y LA REALIDAD EN PROYECTOS DE SOFTWARE : ASPECTOS BACKEND DEL PROYECTO QEMPO

RESUMEN

El presente informe explica los desafíos afrontados como nuevo profesional de Ingeniería de Sistemas sobre las diferentes etapas del desarrollo del proyecto Qempo Marketplace Backend; desde la propuesta inicial hasta la situación actual, lecciones aprendidas y mejoras a futuro. Este proyecto es parte de una propuesta tecnológica completa que parte desde el análisis de procesos existentes hasta la presentación de un producto completo. Qempo Marketplace es un proyecto con múltiples módulos y componentes, y en todos estos el que estuvo bajo mi responsabilidad es el componente backend. Durante este proyecto tuve que aprender, utilizando los sólidos fundamentos de la Ing. de Sistemas, a aplicar éstos en la práctica y crear un proyecto mantenible que sea conforme con las buenas prácticas del desarrollo de software. Los desafíos afrontados fueron transformar los procesos de negocio en una propuesta tecnológica, establecer los requerimientos de dicha propuesta y los plazos requeridos para llevarla a cabo e introducir este nuevo entorno de producción al negocio. Además de los requerimientos de negocio se tuvo que establecer las políticas para la protección de datos personales requeridas tanto por ley como para un negocio ético. La versión actual de la plataforma ha demostrado ser sólida, durable y flexible, sin embargo, hay recomendaciones y mejoras que se pueden dar para una versión a futuro.

PALABRAS CLAVES Systems Analysis and Design, IT Infrastructure, IS Project Management

ABSTRACT

This report will introduce the different steps in development for the Qempo Marketplace Backend project, starting from the initial proposal all the way to the current situation, lessons learned and future improvements. This project is part of a technological proposal that begins with analyzing the existing processes through the demonstration of a full technological product. The reason my report details only the backend component is because this was the area under my responsibility.

The business value proposition was originally bringing consumer goods from the United States with ease. The initial tech stack was a website powered by an ecommerce software as a service named Magento. The role of the tech team that I was part of was to create a new platform for a new value proposal: bringing consumer goods from the USA through travelers.

The new platform was designed from the ground up, and thus Software Engineering knowledge was essential throughout the project in order to achieve a robust, maintainable codebase. There were three versions of the software product which are correlated to a professional learning process based on acquired experience. Ordered, these three versions were: proof of concept, minimum viable product (MVP) and production server.

The challenges faced as a new professional in Systems Engineering were to transform the business processes into a technological proposal, establish the requirements for said proposal and the required time frames for development, and introducing this new product to the business production environment. Besides the business requirements, establishing policies for personal data protection were required both by law and by business ethics.

The final release of the new platform has demonstrated being solid, durable and flexible, however there are still recommendations for a future version.

KEYWORDS e-commerce, development, software

INTRODUCCIÓN

El presente informe aborda el proyecto desde el punto de vista de cada uno de los objetivos educacionales para el egresado de la carrera de Ingeniería de Sistemas. Mi enfoque introduce los elementos del proyecto “Qempo Marketplace Backend” y relaciona estos con cada uno de los objetivos educacionales.

Qempo es un negocio de plataforma que empezó como un *e-commerce* enfocado en las importaciones y evolucionó con el uso de tecnología en una plataforma que facilita el servicio de importación por medio de numerosos asociados logísticos, ya sean viajeros independientes o empresas de carga.

Al tratarse de una plataforma, la propuesta de valor del negocio se encontraba orientada a dos públicos objetivo: uno el mercado de consumidores que busca importar productos desde los Estados Unidos, y el otro, el mercado potencial de viajeros que busquen facilitar el servicio de importación a cambio de dinero. Este segundo segmento posteriormente se expandió a empresas de carga con capacidad logística pero poca inversión en tecnología.

Para realizar esta propuesta de valor se creó un equipo de tecnología del cual formé parte. Este fue evolucionando a medida que avanzó el proyecto, iniciando como un equipo de poca experiencia y terminando con un producto maduro y robusto.

A continuación, explico mi rol en el proyecto y como mis decisiones como profesional de Ingeniería de Sistemas ayudaron al equipo a lograr su propuesta de valor, mejorar sus KPIs, y atraer a la inversión; siguiendo los lineamientos establecidos por la disciplina de Ingeniería de Sistemas.

CAPACIDAD TÉCNICA

El negocio inició bajo el nombre de "Anson Import" el cual se encontraba dedicado al rubro de importaciones, pero en el año 2016 evolucionó hacia un modelo de negocio denominado "Qempo" cuya propuesta de valor era traer, de una manera fácil para el comprador, productos desde Estados Unidos.

¿Por qué considerar esto una evolución del modelo de negocio? En el año 2016 el enfoque de la industria tecnológica se encontraba en los nuevos gigantes basados en plataformas y la llamada "sharing economy". Uber, una plataforma basada en estos principios, alcanzaba una valuación de $70 mil millones de dólares americanos (The Economist, 2016); y el modelo de plataformas era promovido como la única solución para los negocios de tecnología (Choudary, Van Alstyne, & Parker, 2016).

En el modelo de negocio inicial el cliente iniciaba una solicitud para obtener un producto. Esta solicitud se ingresaba al software Magento que es una plataforma de *e-commerce* y ERP. Posteriormente un operador generaba una cotización y respondía por correo. Dicho flujo se encuentra diagramado en la Figura 1.

La transformación del modelo de negocio desde un *e-commerce* simple hacia un negocio de plataforma empieza en el año 2016. En este nuevo modelo se proponía dejar que los viajeros establezcan sus propias cotizaciones de producto. En base a los requerimientos establecidos por la gerencia de Qempo, se propuso un nuevo proceso de negocio ilustrado en la Figura 2, en el cual existen dos tipos de cliente: el comprador y el viajero. Mientras que el comprador busca obtener productos desde los Estados Unidos, el viajero busca obtener dinero a cambio del uso de su espacio de maleta en viajes desde los Estados Unidos.

Un modelo de negocio de plataforma tiene mayor capacidad de lograr una economía de escala, haciendo uso del efecto de red, en el cual el valor es generado por ambos: la tecnología y la comunidad facilitada por esta. Un negocio de plataforma busca alcanzar el crecimiento exponencial, escalando rápidamente sus operaciones. Según un artículo en Knowledge@Wharton (Knowledge@Wharton, 2016), los negocios de plataforma permiten mayor crecimiento, ganancias, ingresos, y valor.

Una vez elaborados los requerimientos del negocio, se solicitó realizar una prueba de concepto sólo para uso interno. Dado que no existía software para este propósito específico, la solución propuesta fue un desarrollo in-house. Dicha solución consistía en un proyecto monolítico, sin separación entre componentes internos ni los componentes *frontend* y *backend*. Considerando que se estaba preparando un prototipo, no se tuvo en cuenta requerimientos no funcionales de mantenibilidad ni se estructuró una arquitectura flexible. Esta versión estuvo en desarrollo desde noviembre de 2016, hasta fines de enero del 2017.

El prototipo estaba escrito en JavaScript (Node.js) utilizando Express para la gestión de HTTP y para el renderizado de plantillas se empleó EJS. Para la persistencia de datos se decidió utilizar un motor flexible que permita cambios en los esquemas, optándose por MongoDB. La prueba de concepto demostró la capacidad del equipo de abordar el problema de la construcción de una plataforma y permitió al equipo de gerencia demostrar a profundidad la propuesta de valor a inversionistas y otras partes interesadas. La decisión de usar estas tecnologías, en contraste a otras

como Python y Django o Ruby on Rails fue la familiaridad del equipo inicial con JavaScript. No obstante, hubo ciertos errores de diseño en esta versión del producto, dejando las siguientes lecciones para el equipo:

a) Un problema común durante las iteraciones de desarrollo en MongoDB era que los cambios en el esquema de datos esperados por la aplicación no necesariamente se encontraban reflejados en la información almacenada. Al realizarse un cambio en las expectativas de la aplicación hacía falta transformar la data almacenada para reflejar los mismos. Este problema ha sido extensamente descrito en el post de Olery Developer Portal (Olery Developer Portal, 2017).

b) El modelo de datos utilizado con Mongoose utilizaba constantemente documentos que referencian otros documentos. La solución utilizada en la prueba de concepto era un API de Mongoose llamada populate(), la cual permite realizar una proyección al seleccionar documentos ingresando información requerida desde otros documentos. El uso de este API era extenso en la prueba de concepto y generaba una fuerte carga innecesaria al servidor. Esto es debido a la información siendo de naturaleza relacional, representada en una base de datos no relacional como mencionado en un post por Sarah Mei (Mei, 2013).

c) Los errores inducidos por la falta de tipado estático en JavaScript. Esto significa que la interfaz de los objetos es implícita, lo cual puede inducir comportamientos inesperados en proyectos con múltiples desarrolladores. Este fenómeno es descrito por Gao, Bird y Barr (Gao, Bird, & Barr, 2017).

En base a las lecciones aprendidas de la prueba de concepto se incorporaron mejoras en el modelo de negocio y se empezaron a realizar las propuestas para lograr un auténtico producto mínimo viable dentro de Qempo. Esta solución debía ser lo suficientemente robusta para operar en un entorno de producción pasada la fase de pruebas. Por este motivo, se otorgó mayor prioridad a la toma de decisiones técnicas.

Para incorporar las buenas prácticas de desarrollo de software y aprovechar las lecciones aprendidas en la prueba de concepto, se tomaron las siguientes decisiones:

a) Se buscó una separación de áreas de interés, es decir *separation of concerns*, término acuñado por Dijkstra (Dijkstra, 1982). Los módulos separados fueron el *frontend* y el *backend* y la interacción entre ambos se gestionó mediante API's

b) Se buscó asegurar la consistencia de los datos mediante la utilización de una base de datos relacional. El modelo de consistencia eventual resultó poco adecuado para una materia puramente transaccional, ya que este deja la reconciliación de los cambios a una etapa final de la transacción. Esto puede resultar en contradicciones en los resultados, o peor aún, pérdida de datos. Se optó entonces por usar bases de datos transaccionales que cumplan completamente con la prueba ACID, descrita originalmente por Haerder y Reuter (Haerder & Reuter, 1983).

c) Respecto al tema del tipado estático existían dos soluciones: TypeScript, una herramienta de código abierto impulsada por Microsoft, la cual actúa como un preprocesador de código que verifica la integridad de tipos; y Flow, una opción que actúa de similar manera, pero

cuenta con tipado más estricto impulsada por Facebook. En el momento que se tomó la decisión, TypeScript ya era una plataforma mucho más madura, contando con una mayor cobertura de librerías clave y múltiples versiones completas. En contraste, Flow hasta el día de hoy sigue en una versión beta. Una comparación entre la cantidad de preguntas en StackOverflow para ambas herramientas, adjunta como Figura 3, sustenta esta decisión.

Como no se tenía tiempo ni presupuesto para contratar o reentrenar a los miembros del equipo, se optó por mantener el uso de JavaScript para facilitar la participación del equipo de *frontend* en el proceso de desarrollo de *backend* y mantener la flexibilidad que un lenguaje dinámico como JavaScript permite.

Se tomó la decisión técnica de basar la nueva versión en el modelo de Clean Architecture propuesto por Robert C. Martin (Martin, Clean Architecture and Design, 2014; Martin, Clean Architecture: A Craftsman's Guide to Software Structure and Design, 2017), también conocido como “Uncle Bob”. La organización de esta arquitectura requiere la separación de componentes en una serie de capas, que pueden ser como ejemplo:

- Entidades: Objetos de negocio que encapsulan las reglas globales de negocio.

- Casos de Uso: Reglas de negocio propias de la aplicación. Estos implementan el acceso a las entidades requeridas en la aplicación.

- Adaptadores de Interfaces: Lógica que transforma la información al formato requerido para objetos externos como la base de datos o la interfaz web.

Figura 1

Proceso de negocio original

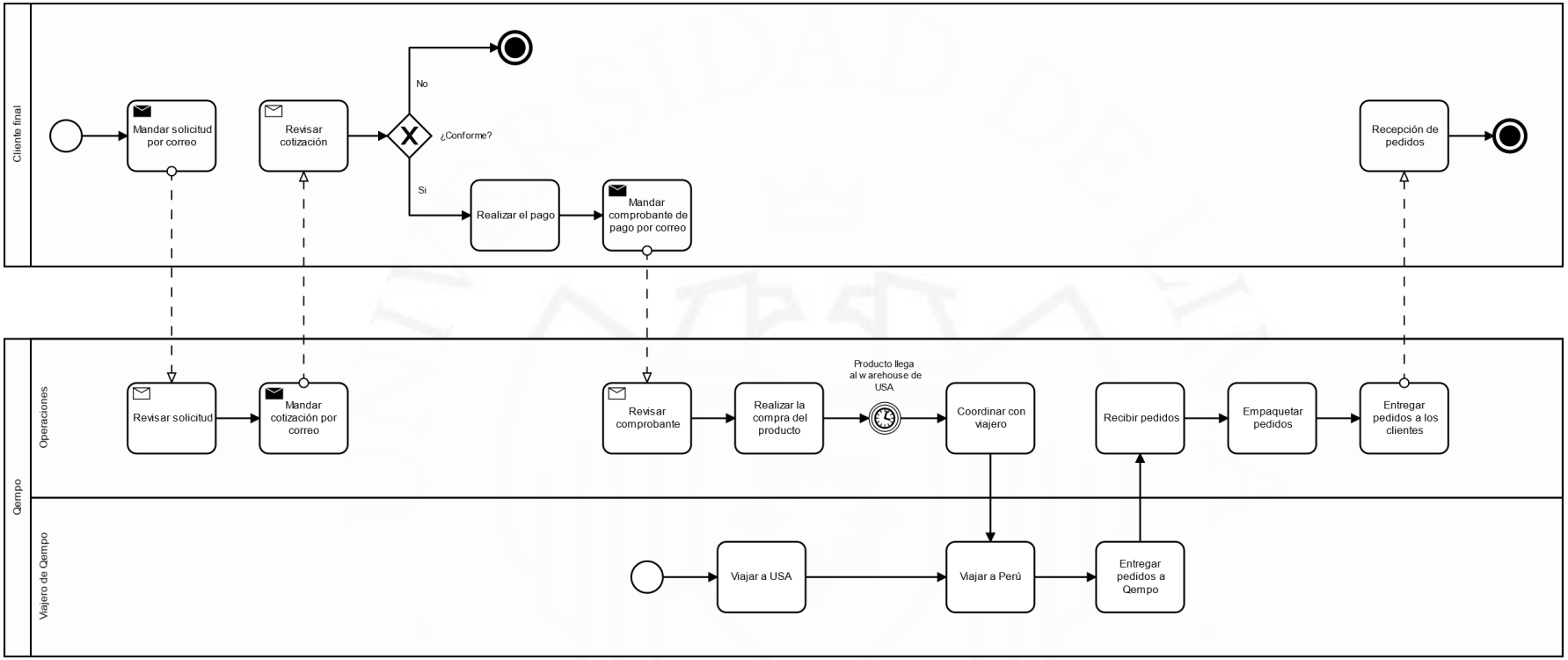
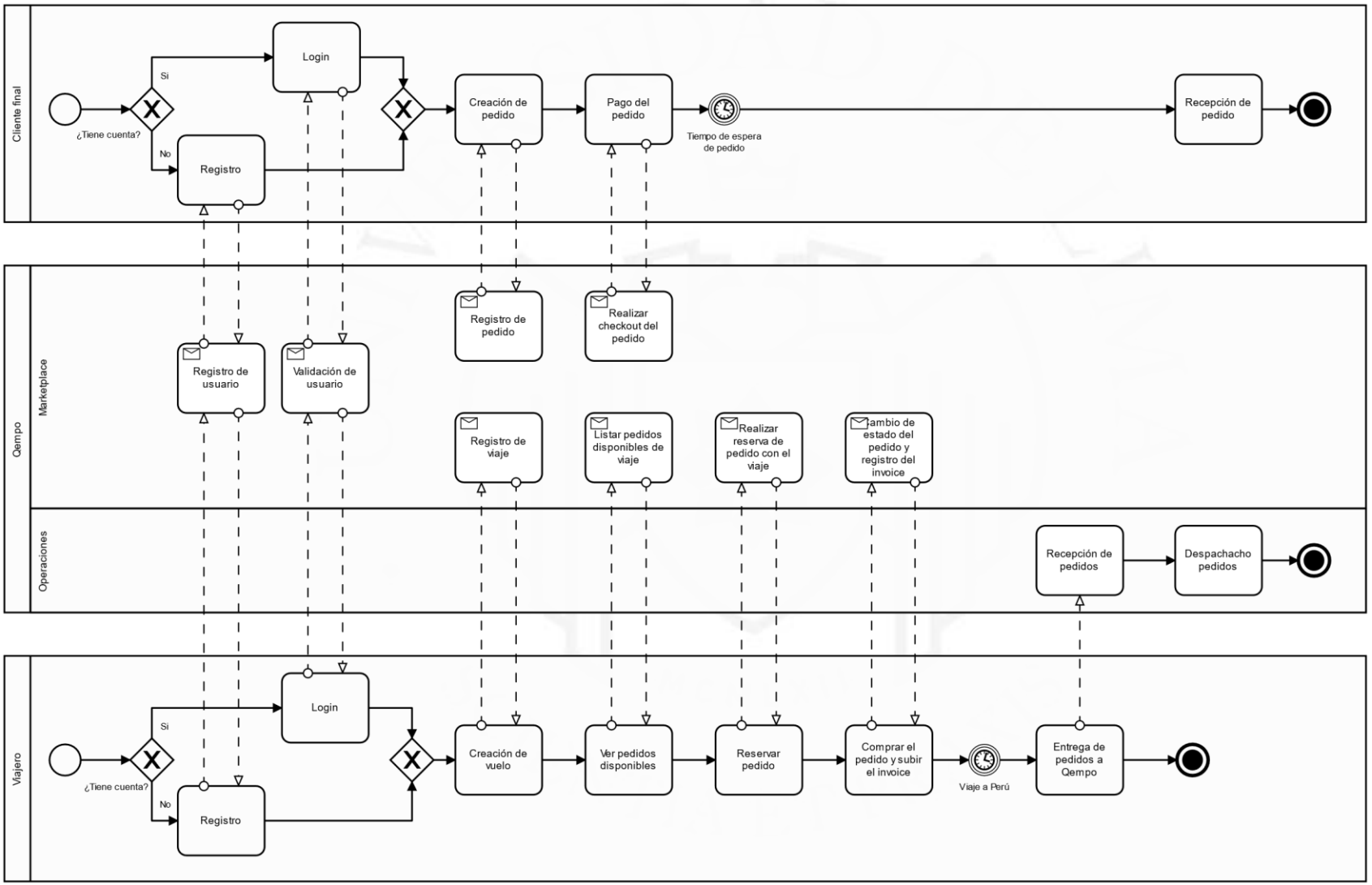


Figura 2

Proceso de negocio para la Prueba de Concepto



- Frameworks y Drivers: Librerías ya sean internas o externas, como la framework web o base de datos.

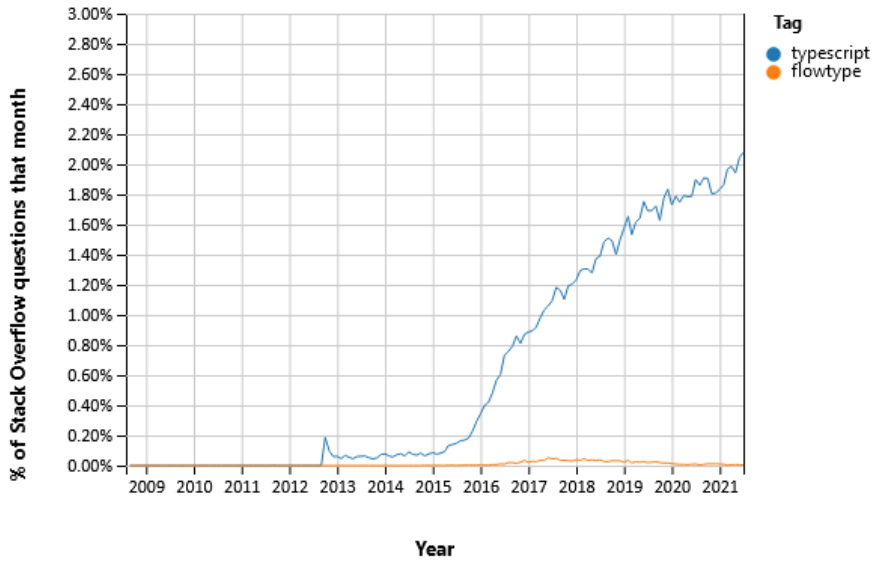
Para la Clean Architecture, la base de datos es un detalle de implementación, el cual debe ser propiamente abstraído por las capas superiores. El principio fundamental es que objetos en capas superiores no deben ser afectados por objetos en capas inferiores. Un cambio de base de datos no debe alterar el funcionamiento de las reglas de negocio o los casos de uso.

El desarrollo del producto mínimo viable o MVP por sus siglas en inglés, inició en abril del 2017, llegando a producción en febrero del 2018. Se utilizó “koa” como framework HTTP y Sequelize como framework de acceso a base de datos. El motivo de estas decisiones es que koa es una evolución de Express desarrollado por los mismos desarrolladores, mientras que Sequelize es un ORM ligero poco prescriptivo.

Otra decisión técnica importante para este lanzamiento fue la introducción del servidor web nginx al entorno. El entorno de pruebas para la versión prototipo no utilizaba un servidor web robusto, sino la misma solución se ejecutaba en el puerto 80 como servidor principal. Las implicaciones de esto para la seguridad de la información son catastróficas. La ejecución en el puerto 80 requiere privilegios administrativos, lo que significa que la aplicación requiere privilegios administrativos innecesarios para ejecutar. Al no tener el nivel de auditoría de un servidor web robusto, esto se hace una vulnerabilidad significativa.

Figura 3

Comparación entre la cantidad de preguntas en StackOverflow para TypeScript versus Flow type.



El lanzamiento del producto implicó un cambio total en los procesos de negocio de la empresa, el cual fue facilitado por el equipo de tecnología. Ya que Magento es un entorno alojado por un tercero, se exportó la base de datos de clientes de este entorno y se importó a la nueva solución desarrollada. El motivo de esto fue facilitar la migración de usuarios con la mayor transparencia posible.

Una vez realizado el lanzamiento del producto a fines de enero del 2018 esta versión fue mantenida hasta el mes de octubre de 2018. El impacto de las decisiones de negocio y técnicas se puede apreciar en las métricas obtenidas a lo largo de este periodo, las cuales serán descritas en breve.

Aspectos positivos de esta versión:

- El producto soportó satisfactoriamente la automatización del proceso de negocio: se hizo posible para los compradores solicitar pedidos y para los viajeros cotizar dichos pedidos.
- Durante sus 8 meses en producción, el producto generó ingresos brutos aproximados de $302,000 dólares distribuidos en aproximadamente 1500 órdenes, lo cual se traduce en aproximadamente $37,750 por mes y un *ticket* promedio de $201.

Sin embargo, surgieron algunos inconvenientes como:

- Errores de diseño interno a pesar de las buenas intenciones en el modelo de arquitectura: particularmente interfaces poco abstraídas y exceso de código “boilerplate”. En otras palabras, generaban mucho código innecesario y repetido.
- Poca flexibilidad para adecuarse a cambios mayores en el proceso de negocio. Una evaluación subjetiva de la capacidad del equipo puso el tiempo estimado para lograr pasar una funcionalidad compleja a producción en el rango de los 2 meses.

Después del pase a producción de la primera plataforma, la gerencia de Qempo realizó alteraciones significativas en la propuesta de valor de negocio. Un análisis realizado por el equipo administrativo durante este periodo operativo identificó el proceso de cotización como un “pain point” dentro del proceso de negocio. Al tener que esperar el resultado del proceso de cotización, se perdían clientes potenciales que esperan resultados inmediatos.

La respuesta a la problemática del proceso fue la introducción de un nuevo proceso de negocio en el que la responsabilidad de cotizar los productos pasaba a la plataforma. Esto implica un cambio radical en el comportamiento ya que el subproceso de cotizar desaparece del flujo de viajero, pasando a ser un elemento de la plataforma: esto requiere una mejor recopilación de la información de producto.

Del lado de tecnología, la nueva versión tendría que cumplir los siguientes requerimientos no funcionales:

- Flexibilidad al cambio. La estructura rígida del primer proyecto no facilitaba cambios mayores en la plataforma.
- Facilidad en la integración con base de datos. El primer proyecto realizaba queries usando el framework llamado Sequelize, el cual solo permite realizar operaciones simples de carga de entidades relacionadas. Dado que SQL es un lenguaje poderoso que permite hacer

consultas de mucho mayor complejidad, un framework de consultas más poderoso facilitaría la construcción de muchas consultas de mayor dificultad.

Se tomó la decisión técnica de emplear Python, usando Pyramid como framework web y SQLAlchemy como framework de acceso a la base de datos.

Esta decisión se sustentó en:

- En 2018, Python ya contaba con una herramienta de tipado estático dentro del mismo estándar del lenguaje, en lugar de herramientas de terceros. La implementación del PEP 484 se introdujo a Python en la versión 3.5 lanzada en 2015 *(Van Rossum et al., PEP 484 - Type Hints, 2014).*
- SQLAlchemy es más poderoso que Sequelize. A la vez que tiene mucho más tiempo en desarrollo y acaba siendo más maduro (Sequelize Team; SQLAlchemy authors and contributors, 2021). Una comparación en tendencias en StackOverflow demuestra mayor actividad sostenida en preguntas sobre SQLAlchemy, incluida en la Figura 4.
- Pyramid es ligero y poco prescriptivo. Este tipo de diseño es similar a koa, el framework previamente usado. Por su naturaleza similar se hace un reemplazo idóneo para koa para el equipo.
- Python es más estable en cuanto a las versiones: NodeJS tiene dos versiones mayores al año mientras Python sigue en el mismo número mayor de versión mayor desde el año 2008. Aunque un cambio de versión no implica potenciales cambios de sintaxis en NodeJS como en Python, sí se pueden dar cambios significativos en la forma en se cargan módulos, entre otros, lo cual puede acabar forzando el equipo a realizar cambios importantes en el software constantemente (OpenJS Foundation, 2021; Python Software Foundation, 2021).
- En el ecosistema de NodeJS hay un exceso de micro dependencias en proyectos de envergadura. En particular, en un incidente en el año 2016, un desarrollador removió un paquete con 11 líneas de JavaScript usado como micro dependencia de npm. Esto resultó en un sinnúmero de paquetes en el ecosistema fallando (Williams, 2016).
- Si bien NodeJS usa un modelo de programación asíncrono, la aplicación estaba extremadamente lejos de sufrir un cuello de botella por cambios de contexto para entrada y salida. En otras palabras, no era una optimización necesaria para el correcto funcionamiento de la aplicación. Aún para estos casos existen herramientas en Python como asyncio, Stackless o greenlet que permiten lograr un modelo de programación concurrente (Python Software Foundation, 2021; Rigo & Tismer, 2011; Stackless Team, 2021).

Otras decisiones de arquitectura que generaron gran impacto fueron:

- Cambiar el modelo de datos para alinearse a los cambios en el modelo de negocio. Ciertas nomenclaturas en este fueron alteradas para esta versión, pero la gestión del modelo continuó estando separada en scripts SQL.
- Aplicar la inversión de dependencias haciendo uso intensivo de estado interno en la aplicación.

- Durante el ciclo de vida de esta solución, se introdujo el proceso de integración continua. Este consiste en *builds* automatizados por cada versión subida al control de versiones (Fowler, Continuous Integration, 2006). Para facilitar este proceso se utilizó un repositorio de artefactos PyPI privado y Bitbucket Pipelines. El proceso consistía en la ejecución automática de la revisión de estilo de código y pruebas unitarias. En caso se ejecutase correctamente, la versión del artefacto se subía al repositorio interno de PyPI. El proceso se encuentra ilustrado en la Figura 5.
- Para hacer seguimiento a los errores en producción, se integró la plataforma Sentry (Functional Software, Inc) al producto backend. Esta captura automáticamente errores detectados durante la ejecución del software y los recopila en una plataforma dedicada al seguimiento de errores.
- Para hacer seguimiento al desempeño del servidor, se instaló la herramienta Cockpit (Zamot, 2020) en este. Este es un motor similar a un *task manager* que permite visibilizar el uso de recursos en el servidor, así como supervisar los servicios del sistema.

Esta versión estuvo 6 meses en producción desde octubre 2018, durante los cuales se generó ingresos brutos aproximados de $360,000 dólares distribuidos en aproximadamente 2000 órdenes, lo cual se traduce en aproximadamente $60,000 por mes y un *ticket* promedio de $180. El seguimiento de cambios más estricto impuesto en esta versión permitió identificar un promedio de 11 hotfixes al mes, consecuencia de los despliegues. La dificultad de desarrollo, que fue el principal factor motivando el cambio de lenguaje, también bajó considerablemente, con el equipo adquiriendo la capacidad de hacer al menos una funcionalidad difícil al mes, el doble del ritmo anterior.

Tanto durante el ciclo de vida de esta versión como de la anterior, el desarrollo de una plataforma intranet se mantuvo con baja prioridad debido a la priorización de funcionalidades que puedan atraer clientes o inversionistas. Esto causaba baja visibilidad del desempeño de la plataforma en el equipo de negocio.

Para resolver este problema se introdujo al entorno de producción una herramienta llamada pgAdmin (The pgAdmin Development Team). Esta es una interfaz gráfica que permite ejecutar y almacenar consultas SQL. Al equipo de operaciones y al equipo de negocio se le otorgó permisos de solo lectura con una serie de queries pre-escritas, las cuales extraían la información relevante para el desempeño de sus funciones.

Sin embargo, la versión también tuvo problemas. El uso extensivo de estado interno causaba que la aplicación entre en estado indeterminado si algún bug o error de memoria sobrescribía parte del estado interno. Esto resultaba en un colapso total de las respuestas otorgadas al usuario. Otro problema similar era la tendencia del servidor a sufrir fugas de memoria. Una fuga de memoria o *memory leak* es causada por la reserva de recursos en el servidor que nunca son liberados y se mantienen en la RAM. Se puede afirmar que el motivo de este colapso es que el diseño de los módulos era flexible pero la manera en que eran instanciados era poco flexible.

Otro problema fue la falta de un timeout para ciertas operaciones de entrada y salida. Al estar el servidor basado en un modelo de hilos, esto significa que todos los hilos podrían ser bloqueados y dejar la aplicación completamente no responsiva del lado de usuario.

La manera más rápida en que se arreglaban los problemas de fuga de memoria, bloqueo de recursos o estado indeterminado era reiniciando el servidor, pero a esto se le añadía que el servidor tenía un largo tiempo de inicialización debido a la cantidad de estado interno de la aplicación a inicializar.

Cuantificando el impacto de los problemas con las herramientas instaladas, el servidor sufrió un *uptime* de tan solo el 95%, y un promedio de 153 nuevos errores reportados en Sentry cada mes. De estos, algunos tenían miles de incidencias.

Ante la problemática y el ingreso de nuevos requerimientos funcionales mayores de negocio, se aprovechó para realizar un rediseño en la forma en que se gestiona la aplicación.

Los nuevos requerimientos fueron la adición de un carrito de compras y la recopilación de datos automática de los enlaces proveídos por el cliente. Para la solución, en primer lugar, la dependencia en SQLAlchemy dejó de tratarse como un paquete más y pasó a ser parte del núcleo de la aplicación, al ser una dependencia esencial para el funcionamiento de esta. Del otro lado, se aprovechó el bajo acoplamiento de Pyramid para que este sea reemplazado por Flask, al ser este último más flexible para la toma de decisiones arquitecturales del equipo. Otro criterio para el reemplazo es que Flask tiene mucha más actividad como comunidad y se encuentra en crecimiento, mientras Pyramid se encuentra en decrecimiento como se aprecia en la Figura 7.

En la primera versión Flask, la cual es la versión 2.0 del proyecto en Python, se aprovechó también para dividir la lógica de negocio en módulos. El scrapping de data de producto pasó a ser un módulo separado, el código boilerplate para la aplicación web pasó a ser parte del módulo común, entre otras. Al estar más fuertemente integrado SQLAlchemy, este pasó a ser usado como una implementación del patrón *repository* y el patrón *data mapper (Fowler, et al., 2002)*. Los modelos podían ser consultados con cualquier criterio arbitrario, otorgando la máxima flexibilidad ante los cambios. Un componente interno orquesta la transformación de estos modelos a modelos de presentación con una sola lista de atributos declarativa, facilitando la adición de atributos a consultas fácilmente.

La arquitectura resultante se encuentra ilustrada en la Figura 9. Los principios fundamentales de esta fueron los principios de Robert C. Martin pero manteniendo la flexibilidad y practicidad por delante. Cada componente representa un módulo Python, sin embargo, todos estos eran desplegados como un solo conjunto de binarias con diferentes formas de ejecución.

Esta versión salió a producción en abril de 2019. Durante sus 12 meses en producción, generó ingresos brutos aproximados de $931,000 dólares distribuidos en aproximadamente 3900 órdenes, lo cual se traduce en aproximadamente $77,583 por mes y un ticket promedio de $239. La cantidad de errores nuevos por mes reportados en Sentry se disminuyeron a 86. La cantidad de hotfixes promedio por mes se redujo a 7, y el tiempo de desarrollo estimado para una funcionalidad compleja se redujo a 2 semanas.

A finales del 2019, entró un nuevo paquete de funcionalidades requiriendo un cambio de versión mayor. El proceso de negocio alcanzó su forma final, la cual se encuentra adjunta en la Figura 8. Se introdujo un nuevo tipo de usuario llamado *courier*, el cual actúa como un nuevo proveedor de servicio, distinto al viajero. Para el desarrollo de esta versión, no se introdujeron cambios mayores en el *stack* tecnológico utilizado.

Esta versión ingresó al entorno de producción en abril del 2020, a inicios de la pandemia COVID-19. Aún bajo estas circunstancias, la nueva versión adquirió ingresos brutos aproximados de $1,278,000 dólares distribuidos en aproximadamente 4200 órdenes, lo cual se traduce en aproximadamente $106,500 por mes y un ticket promedio de $304. La cantidad de errores nuevos por mes reportados en Sentry se disminuyeron a 55. La cantidad de hotfixes promedio por mes se redujo a 3. Las métricas de velocidad del equipo no cambiaron.

Figura 4

Comparación entre la cantidad de preguntas en StackOverflow para TypeScript versus Flow type.

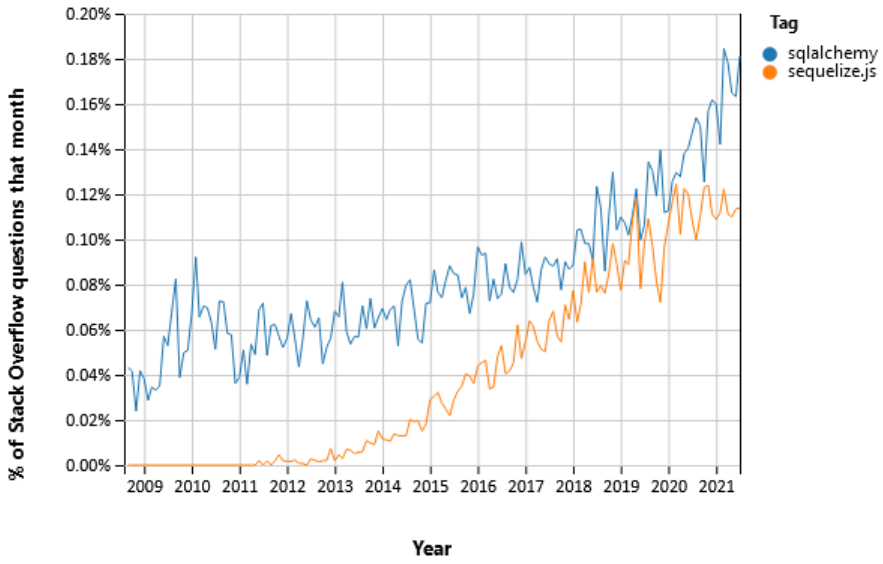


Figura 5

Proceso de integración continua

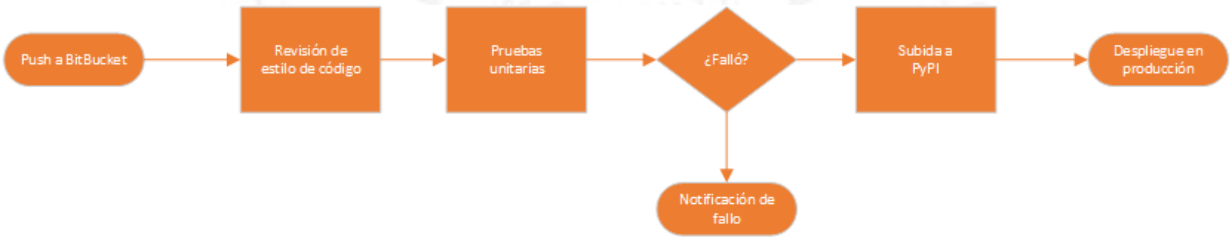


Figura 6

Línea de Tiempo de Qempo Backend

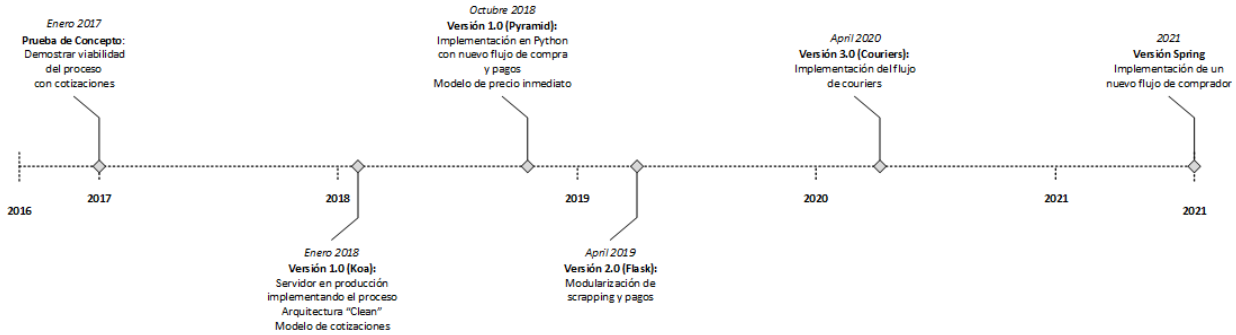


Figura 7

Comparación entre la cantidad de preguntas en StackOverflow para Flask versus Pyramid

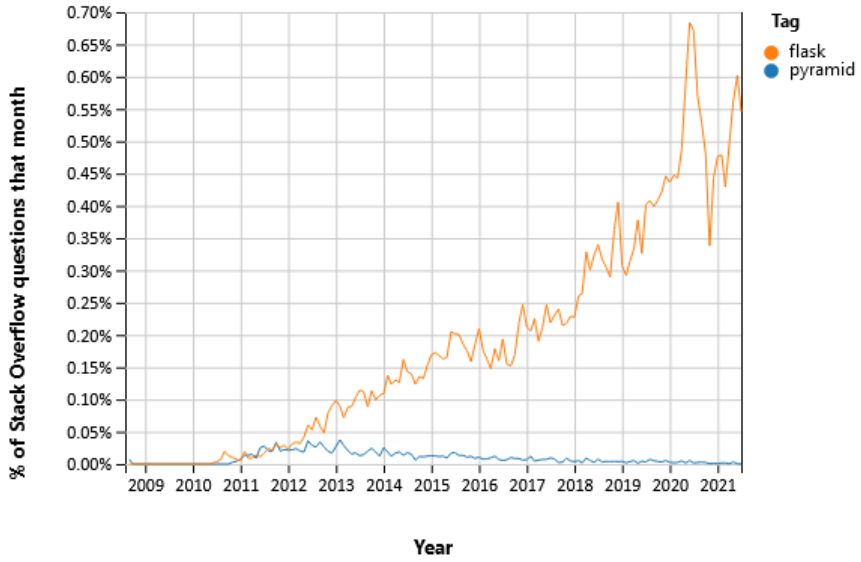


Figura 8

Proceso de negocio actual

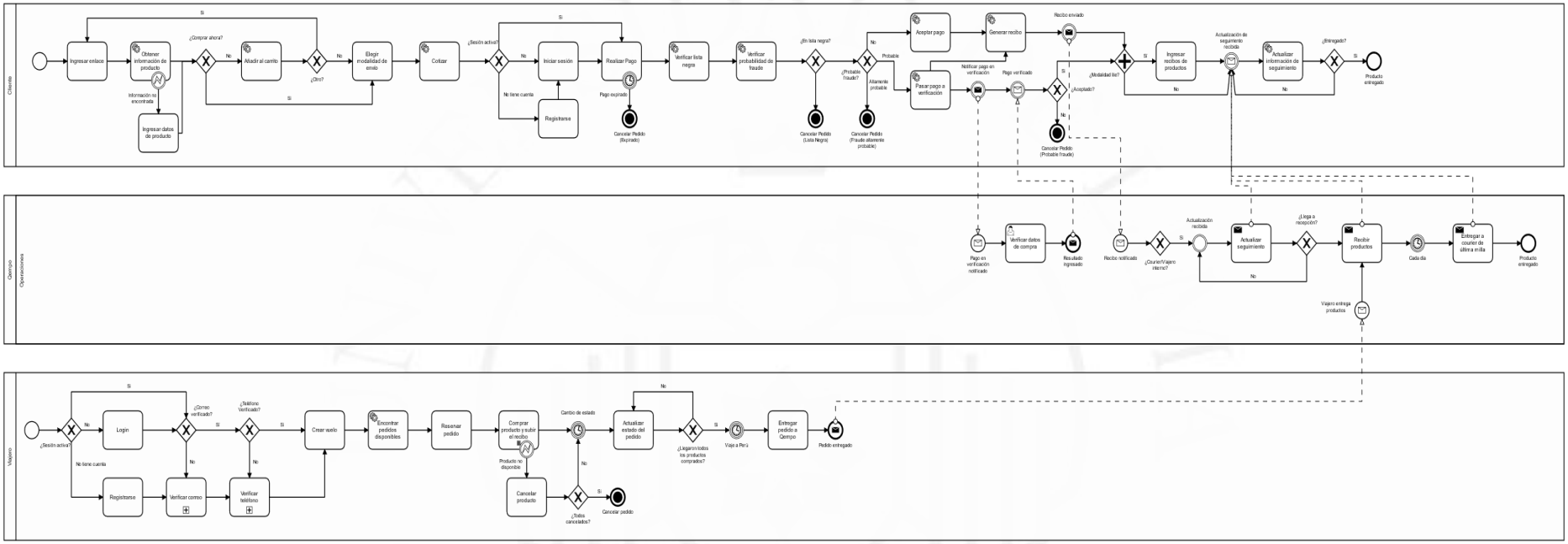
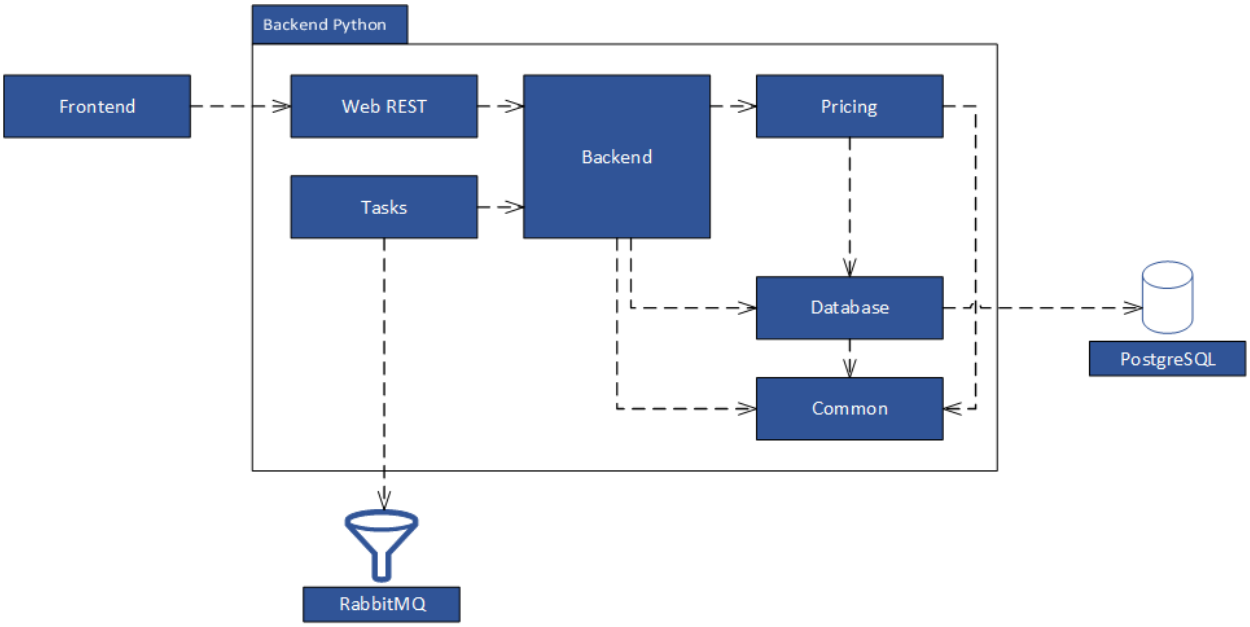


Figura 9

Arquitectura de la aplicación web



CAPACIDAD DE GESTIÓN

El proyecto inició con tan solo propuestas de proceso de negocio, así que fue responsabilidad del equipo convertir estos en requerimientos funcionales. Un proceso de negocio consiste en una serie de pasos o tareas para alcanzar un objetivo o resultado. Los requerimientos funcionales son componentes atómicos/unitarios que permiten desarrollar estas tareas. Un equipo de diseño, inicialmente externo y posteriormente integrado realizó los wireframes y diseños finales de las interfaces de usuario, recibiendo feedback del equipo de desarrollo sobre la viabilidad de diversos componentes.

Ya que el proyecto fue dividido en dos partes, frontend y backend, una de las más importantes responsabilidades a lo largo de este fue coordinar el diseño de APIs. Cada API siempre ha iniciado con una esquemática de datos sobre la cual los dos equipos deben estar de acuerdo, para luego pasar al desarrollo de la API que provee el objeto.

El desarrollo inició con una organización del trabajo "kanban", basada en tarjetas con tareas. El significado de kanban es letrero o señal visual, y lo que busca la metodología con este nombre es representar las tareas visualmente en un tablero para poder organizar las mismas. Con el tiempo el equipo fue integrando metodologías de Scrum. Esto implica la separación de tareas en sprints, los cuales deben entregar un incremento tangible del producto.

Para capacitar al equipo en Scrum, la empresa adquirió un asesor especialista en *Agile* que dió coaching al equipo en esta metodología. Las pautas más importantes de Scrum que se mantuvieron fueron las reuniones de coordinación diarias y la retrospectiva al finalizar la semana.

Inicialmente era el único integrante del equipo de *backend*, pero pasé a tener dos personas más colaborando en el proyecto durante el desarrollo de la primera versión en Python. Esta se desarrolló con mi supervisión sobre el código de producción. Al haber más manos en el código se establecieron pautas y controles para facilitar la cooperación. Los objetivos de estos controles son los siguientes:

1. Evitar que código roto o incompleto llegue a producción
2. Facilitar la legibilidad del código para los contribuidores
3. Reducir el impacto negativo de la deuda técnica. Ejemplos de esto son tiempos de respuesta largos o resultados inesperados.

El motivo por el cual existe tal énfasis en la generación de deuda técnica es que los problemas técnicos, como tiempos de respuesta largos o caída completa del servidor causan pérdida de clientes, algo que se puede apreciar en métricas como Google Analytics.

El primer paso para lograr estos objetivos fue estandarizar el estilo de código. Los conflictos en estilo de código suelen causar conflictos de control de versiones innecesarios, o incluso asignar autoría de un cambio a una versión errónea. Esto es un problema significativo al usar herramientas como git-blame o git-bisect (Chacon & Git community, Git - git-blame, 2021; Chacon & community, Git - git-bisect, 2021) que requieren el historial de versiones para identificar la versión errónea.

Se estableció el uso de una guía de estilo de código y el uso de un *linter* para determinar si el código entregado sigue el estilo recomendado. La herramienta escogida fue Black (Langa, et al.,

2021), al ser una herramienta de formato de código PEP 8 (Van Rossum, PEP 8 - Style Guide for Python Code, 2001) que puede corregir errores de formato automáticamente, ahorrando tiempo en correcciones de formato.

Durante el desarrollo de la versión 2.0 en Python, al extraerse componentes compartidos para uso en otras aplicaciones, se identificó estos como componentes en los que los errores son de alto riesgo, al afectar múltiples proyectos. Como una medida de control adicional a las guías de estilo, se añadieron pruebas unitarias a las funcionalidades compartidas. Para enforzar ambas políticas, tanto la revisión de estilo como la ejecución de pruebas unitarias se añadieron al proceso de integración continúa mencionado en la parte 1. Esto hace que la ejecución de ambos sea requerida como parte del ciclo de lanzamiento. Para recapitular, la integración continua consiste en *builds* automatizados del software por cada nueva versión (Fowler, Continuous Integration, 2006).

La medición del impacto de estas decisiones se da bajo criterios automatizados, como alcanzar el porcentaje de cobertura en pruebas unitarias requerido; así como en criterios medidos manualmente, como el tiempo de respuesta de la plataforma y el tiempo requerido para desarrollar una funcionalidad dentro del equipo, ambos mencionados en la parte 1.

La gestión del catálogo de funcionalidades pendientes se dio por medio de una metodología de priorización, basada en la estimación de impacto de nuevas funcionalidades en los diferentes KPI de negocio, comparada con la estimación de tiempo de desarrollo del equipo. Terminando cada ciclo de desarrollo se realizaba una medición simple del avance de cada funcionalidad con el fin de planificar la carga del siguiente ciclo de desarrollo. Para estimar los tiempos se asignaba un peso a la característica, fácil, medio o difícil, y luego se procedía a dar un estimado de 1 día, 2 días, 1 semana, 2 semanas o 1 mes.

En general, se puede dividir la medición del impacto negativo de la deuda técnica en producción en los siguientes grupos:

1. El desempeño de la plataforma en producción medido en tiempos de respuesta
2. cantidad de errores reportados
3. la dificultad percibida en el equipo para la implementación de nuevas funcionalidades.

Para medir estos dos primeros indicadores se utilizó los ya mencionados Sentry y Cockpit; y además Google Analytics, Google Lighthouse (Google Inc.; Google Inc.), GTmetrix (Carbon60) y Pingdom (SolarWinds Worldwide, LLC), plataformas que miden el desempeño de sitios web, ya sea en el ámbito técnico o en interacción con consumidores.

En cuanto a los objetivos de negocio, la medición de estos se dio por medio de las queries SQL para los KPI de negocio: algunos de estos como el ingreso bruto o ticket promedio fueron mencionados en la parte 1. Estas fueron diseñadas por el equipo de negocio en coordinación directa con el equipo de tecnología, de manera que toda data factible que ayude al negocio éste fácilmente disponible para consulta. Con esto, había visibilidad inmediata ante cambios en los KPI. Aún al introducirse la plataforma *backoffice*, estos continuaron en uso debido a que la herramienta pgAdmin cumplía el propósito.

Ya que la información en tablas de datos es poco visual, y no atractiva para un “pitch” a inversionistas, posteriormente se procedió a integrar Metabase (Metabase) como una herramienta

de visualización de KPIs. Esta permite generar dashboards con los indicadores de negocio y facilita la visualización de estos, así como proveer una interfaz para la exploración de los datos. La solución ideal sería un proceso ETL, pero no se procedió a esto ya que la base de datos no es lo suficientemente pesada como para que el tiempo de consulta sea nocivo al elaborar KPIs. Un ejemplo de una interfaz creada en Metabase se encuentra adjunto en la Figura 10.

Para la gestión de versiones del producto se introdujo el proceso Gitflow (Atlassian Corporation; Driessen, 2010). Este proceso involucra la creación de dos ramas principales. Una rama *master* que contiene el código de producción, y una rama *dev*, *develop* o *development* que contiene el código más reciente. En este modelo se crean ramas para el desarrollo de nuevas funcionalidades, las cuales inician con *feature/* dentro de la nomenclatura. Estas ramas se unen en la rama de desarrollo, la cual se une a la rama de producción una vez que esta se encuentra estable. Los desarrollos de carácter urgente, que tienen que ser desplegados directamente a la rama de producción, o *hotfixes*, tienen su propia categoría de ramas: *hotfix/*. Para diferenciar versiones en la rama de producción, se utilizan *tags* de git.

Para organizar los números de versiones, se adoptó el versionado semántico (Preston-Werner). Esta convención introduce un número de versión compuesto de tres partes con el formato *mayor.menor.parche*. Una revisión mayor involucra cambios con incompatibilidad de API, una menor añade funcionalidad sin romper APIs, y un parche arregla errores o bugs en revisiones mayores o menores.

En resumen:

1. Se gestionó el proyecto utilizando métodos ágiles,
2. Se establecieron métricas de desempeño del software, así como del desarrollo de este mediante control de versiones,
3. Se alinearon los objetivos del negocio con las métricas mediante KPIs y se generaron tableros de control (dashboards) para una mejor visualización de los resultados.

Una lista no exhaustiva de indicadores y la forma en que estos son adquiridos se muestra en la Tabla 1. Debido al uso poco estricto de SCRUM, no se pudieron adquirir indicadores de integrantes del equipo más allá de la velocidad de desarrollo. Una medición más objetiva requiere un mayor énfasis en la estimación de tiempos, algo que este desarrollo mantuvo en baja prioridad.

Figura 10

Interfaz de usuario de Metabase con un Dashboard de marketing.

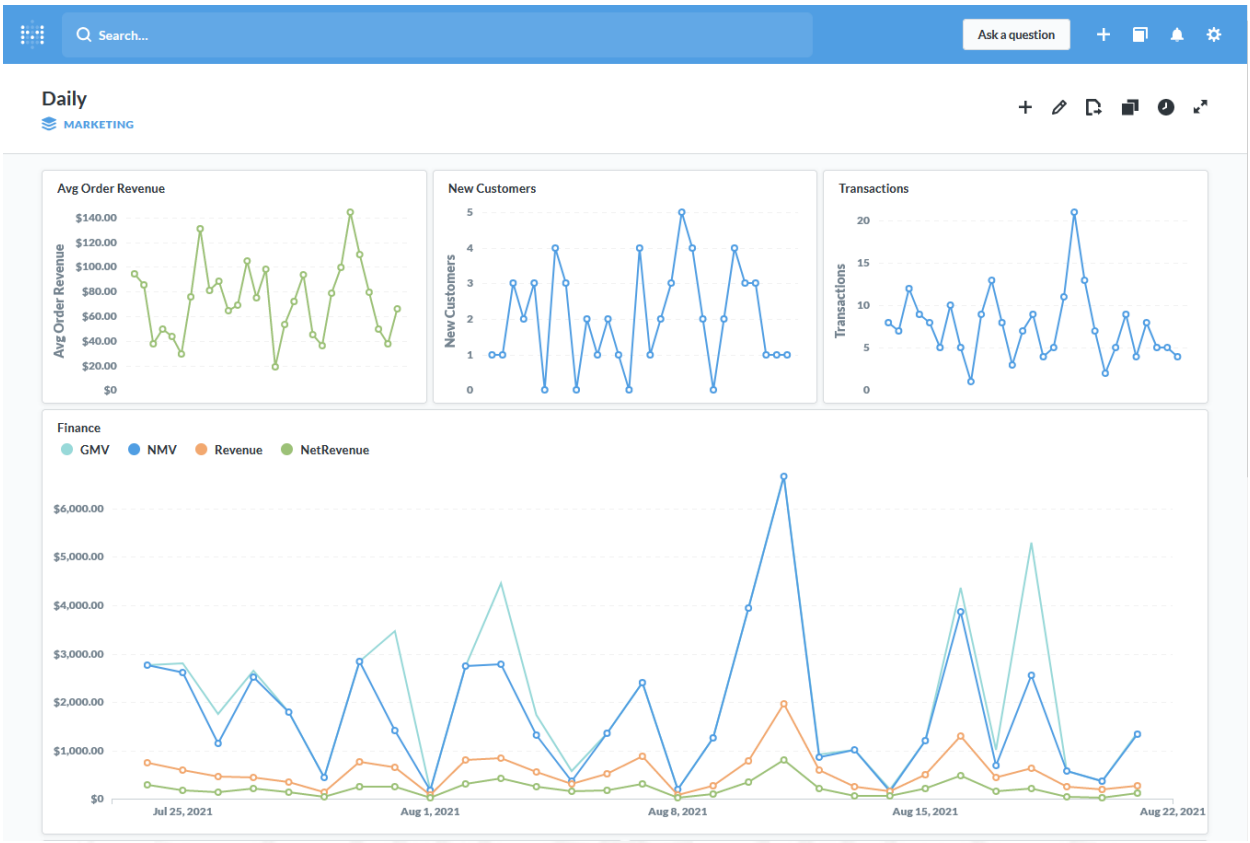


Tabla 1

Indicadores utilizados para la medición del desempeño

KPI	Forma de Medición
Número de Órdenes	Consultas SQL y Metabase
Ingreso Bruto	Consultas SQL y Metabase
Ticket promedio	Consultas SQL y Metabase
CAC (Costo de adquisición de consumidores)	Consultas SQL y plataformas de Marketing
LTV (Valor de vida de un consumidor)	Consultas SQL y plataformas de Marketing
Tiempo promedio de Entrega	Consultas SQL
Tiempo promedio de Aceptación	Consultas SQL
Porcentaje de Órdenes canceladas	Consultas SQL
Tiempo de Respuesta FCP (Contenido presentado)	Google Lighthouse y GT Metrix
Tiempo de Respuesta TTI (Tiempo a interacción)	Google Lighthouse y GT Metrix
Velocidad de Desarrollo	Priorización y Estimados de Tiempo
Cantidad de errores en producción	Sentry
Hotfixes por versión	Bitbucket e Historial de commits

APRENDIZAJE CONTINUO

El ingreso a este proyecto ocurrió mientras cursaba los últimos ciclos de la carrera. Como futuro profesional de Ingeniería de Sistemas, tenía un fuerte fundamento teórico pero poca experiencia práctica en grandes proyectos reales. Durante el desarrollo de la versión MVP se me asignó la responsabilidad de contribuir al desarrollo de este mismo, pero por mis cualidades como desarrollador fui asignado a una mayor responsabilidad sobre el componente *backend*.

Debido a la escasa experiencia en el desarrollo de software a gran escala y en aspectos de arquitectura de software tuve que capacitarme logrando con ello incorporar el método *clean architecture* ya mencionado en la parte 1. Las lecciones de Robert C. Martin tienen pocos detalles de implementación al ser lecciones de alto nivel para la programación en general. La primera versión acabó siendo una implementación al pie de la letra de los componentes mencionados en la propuesta de arquitectura: entidades u objetos de negocio, controladores, presentadores e interactores.

El principal fallo en esta primera versión fue el poco uso de otros patrones de diseño importantes. El código de acceso a los datos se encontraba separado en una capa, pero se utilizaban Data Access Objects (DAOs) para el acceso. Un DAO consiste en un objeto que implementa acceso al almacenamiento de datos. El patrón más similar a esta capa de acceso a los datos sería el patrón Table Data Gateway en Patterns of Enterprise Architecture (Fowler et al., 2002). Existen otros patrones como repositorio (Fowler et al., 2002), los cuales permiten tener una capa de acceso a los datos mucho más flexible. Esto es también una decisión que necesariamente debe tener en mente los frameworks utilizados. La implementación de objetos genéricos que implementan estos patrones flexibles puede ser costosa, y la solución más simple es que las librerías externas actúen como una implementación de estos.

En un principio tomé esto como una violación de los principios de la *clean architecture*, pero entendí posteriormente que no se trata de evitar las dependencias en el núcleo por completo, sino minimizar las dependencias a un conjunto que sea confiable. Un equipo pequeño no puede esperar implementar todas las funcionalidades que requiere, pero sí puede optar por escoger el producto más maduro para introducir este a su núcleo.

Al incorporarme al proyecto tampoco contaba con experiencia en desarrollo ágil o en el mantenimiento de aplicaciones de mayor tamaño durante un periodo de tiempo extendido y coincidentemente el equipo de desarrollo tampoco tenía la experiencia en proyectos de tal envergadura, así que el proceso de aprendizaje involucró a todo el personal. En consecuencia, se tuvo que implementar un plan de capacitación temporal a medida que surgían necesidades en el proyecto, tanto técnicas como de gestión.

Para aprender sobre desarrollo ágil, el equipo recibió asesoría sobre Scrum y Agile con un asesor durante un periodo de tiempo. Durante estas asesorías el equipo empezó a realizar reuniones semanales de retrospectiva, y reuniones diarias “daily stand-up”, como es prescrito en la metodología Scrum. Si bien con el tiempo algunas de estas prácticas prescriptivas quedaron en desuso, la experiencia con esta metodología mejoró la comunicación y planeación de objetivos en el desarrollo.

El crecimiento del proyecto de software que inició como un proyecto de un único miembro y evolucionó hacia un proyecto de equipo, introdujo conceptos como el control de versiones que no se utilizan frecuentemente en proyectos de menor escala, pero que son esenciales para proyectos de mayor envergadura. El uso de la metodología Gitflow (Atlassian Corporation; Driessen, 2010), mencionada en la parte 2 de este documento, tan solo empieza a verse atractivo durante el desarrollo de un proyecto de estas características. Existen otras metodologías para organizar un sistema de control de versiones como Git, pero como equipo aprendimos y usamos este.

Como parte del aprendizaje de Git también introducimos el uso de *rebase* (Chacon & Straub, Git Branching - Rebasing, 2014) al control de versiones. El uso de esta herramienta puede ser controversial (Mørken, 2017) debido a ser destructiva sobre el historial, así que después de probar distintos puntos en los cuales hacer uso de esta, se optó por solo hacer rebase sobre ramas que no sean centrales, como features o hotfixes.

Este fue el primer proyecto en el que tuve que mantener un API estable por un periodo tan prolongado de tiempo, lo cual enseña la importancia de determinar qué funcionalidades pueden ser introducidas como una revisión menor, y que funcionalidades requieren una versión mayor. El versionado semántico (Preston-Werner) tan solo empieza a tener sentido en el momento que un proyecto tiene un API estable. Si el código cliente deja de funcionar al introducirse una revisión menor, esta no es realmente una revisión menor.

Esto implica la creación de un "contrato de API" en cada versión mayor. El API debe respetar un formato establecido dentro del equipo antes de cada lanzamiento mayor, y una vez establecido este formato estable, este no puede cambiar hasta el próximo lanzamiento mayor.

El equipo se capacitó y aplicó estas metodologías en conjunto, ya que solo tienen sentido si se aplican a todo el equipo de desarrollo. Ambos líderes de desarrollo, tanto *frontend* y *backend* (en mi caso) éramos encargados de investigar las mejores prácticas de organización del proyecto de software y versionado, para luego traer éstas al ciclo de desarrollo de todo el equipo.

Con el historial limpio introducido por estas convenciones, se pudo hacer uso de la herramienta *git bisect* (Chacon & community, Git - git-bisect, 2021). En un proyecto de este tamaño el uso de esta facilita sustancialmente la búsqueda de revisiones problemáticas. La herramienta consiste en una búsqueda binaria a lo largo del historial de versiones para encontrar una revisión problemática en particular. Por ejemplo, existiendo 10 revisiones entre la versión 1.0.0 y 1.1.0, donde 1.0.0 no sufre de un bug mientras 1.1.0 sí; *git bisect* permite realizar una búsqueda binaria dentro de estas 10 versiones para determinar qué cambio unitario introdujo dicho bug.

Durante el desarrollo de las versiones 2.0 y 3.0 en Python, el equipo trajo una nueva metodología de priorización recomendada por los inversionistas. Esta consistía en la identificación de KPIs afectados por cada cambio en el catálogo de producto, y en base a pesos asignados a los KPI y estimaciones de tiempo, identificar las funcionalidades de mayor impacto a menor costo de tiempo.

El proceso de elaboración de requerimientos, sin embargo, se volvió suficientemente complicado como para requerir un equipo separado de producto. El equipo de producto estaría dedicado a reunir y discutir los mismos requerimientos con los equipos de gestión y tecnología. Durante mi interacción con este equipo y el equipo de negocio, aprendí sobre cómo enumerar los

requerimientos para una solicitud, y cuánto planeamiento es necesario y cuánto es excesivo. Por ejemplo, hubo múltiples circunstancias en las que se realizó un análisis de requerimientos demasiado detallado para una funcionalidad que no se encontraba en planes.

En retrospectiva, este proyecto se habría beneficiado significativamente por la presencia de miembros con mayor experiencia, pero este entorno lleno de graduados recientes creó un ambiente único donde se traían ideas desde afuera de la empresa y se probaban en la práctica, quedando con las mejores ideas. Es un modelo de aprendizaje y error, pero ayuda a todos los miembros a adquirir experiencia por medio de la experimentación.

Mi decisión como profesional para alcanzar mayor crecimiento al finalizar mi rol dentro de este proyecto ha sido buscar incorporarme a equipos con miembros de mayor nivel de *seniority*, para intercambiar ideas adquiridas por experiencias distintas en otros entornos.

CONDUCTA ÉTICA

El uso del cómputo y de las tecnologías de la información tiene un impacto en el entorno social en que vivimos, una consecuencia estudiada en múltiples ocasiones (Deb, 2014; Misa, 2007). Un clásico ejemplo del efecto que el código tiene sobre las vidas reales de las personas es el caso de la Therac-25 (Leveson & Turner, 1993). La Therac-25 era un equipo de radioterapia que sufría de un bug causado por una condición de carrera en el código. El resultado de este bug era una exposición a una dosis letal de radiación. Ya que el uso de las herramientas informáticas no se encuentra aislado de la sociedad, es necesario establecer un marco ético sobre el cual desarrollar nuestra profesión.

Una buena referencia son los lineamientos del código de ética y conducta profesional de la ACM (ACM Code 2018 Task Force, 2018), al ser esta una organización representativa del rubro. El código consiste en 25 puntos que un profesional en la computación debería respetar. A lo largo de esta sección se explica uno por uno por qué este proyecto fue desarrollado respetando las pautas establecidas por este código.

El punto 1.1: *contribuir a la sociedad y el bienestar humano, recalcando que todas las personas están interesadas en el cómputo,* no requiere una extensiva explicación, ya que este sistema fue desarrollado para facilitar una idea de negocio completamente legal.

Para elaborar en el punto 1.2: *evitar el daño,* durante el desempeño del proyecto mantuve una actitud tajante con respecto a cualquier conducta que pueda vulnerar los derechos de los consumidores. Hay componentes en el sistema que pueden ser abusados, como la base de datos de direcciones, por lo cual desde el primer despliegue a producción he hecho lo posible por resguardar la data de intrusión alguna.

Los puntos 1.3, 1.4 y 1.5: *ser honesto y confiable, ser justo y tomar acción para no discriminar*, y *respetar el trabajo requerido para producir nuevas ideas, inventos, trabajos creativos y artefactos de cómputo*; no requieren mayor elaboración al ser parte de una conducta profesional adecuada en general. Qempo siempre fue un centro de trabajo inclusivo abierto a todos los candidatos calificados. En cuanto a los puntos 1.6 y 1.7 de la guía de la ACM: *respetar la privacidad* y *honorar la confidencialidad*; esto requiere precauciones con respecto al tratado de datos personales, como ya mencionado antes. Al poner en una balanza la seguridad de los datos personales contra una meta de despliegue siempre he priorizado la seguridad de los datos.

Por ejemplo, las contraseñas de los usuarios usan tecnología moderna de hashing: *bcrypt* (Provos & Mazieres, 1999), la cual no se encuentra vulnerada de manera significativa de la misma forma que tecnologías más antiguas como *MD5* (Sotirov et al., 2008). La plataforma puede requerir fotografías de documentos sensibles de los clientes, las cuales son encriptadas usando AES y con una llave que se almacena por separado del repositorio binario con las imágenes. Ante solicitudes de revisión o supresión como previsto dentro del marco de la Ley de Protección de Datos Personales (Ley de Protección de Datos Personales, 2011), el equipo facilitó la ejecución de estos derechos.

Como profesional de Sistemas, hice hincapié en la necesidad de garantizar el derecho de rectificación de los datos personales, y demostré en la práctica la facilidad en otorgar al cliente este mismo. Si bien la remoción de datos causaba dificultades en la base de datos, ayudé a

proporcionar soluciones que no arruinaran la trazabilidad financiera pero que eliminen los datos personales de los clientes de la plataforma, si así era deseado.

Para resguardar la confidencialidad siempre se ha utilizado TLS para proteger la conexión del cliente durante el ingreso de estos datos sensibles. Dentro del mismo servidor, los usuarios de solo lectura solo recibían los privilegios necesarios. Por ejemplo, un usuario de operaciones en pgAdmin solo puede adquirir o escribir información por medio de procedimientos creados y previamente aprobados. No puede explorar el esquema de datos y la mayoría de los casos de uso ya han sido transferidos a *backoffice*. Si bien hay datos que inevitablemente deben ser visibles como las direcciones de entrega, se minimiza el acceso a estos al mínimo requerido. Ya que el negocio no cumple con todos los requisitos para aceptar pagos provistos por el SAQ D de PCI, no se aceptó información bancaria alguna que esté fuera de los márgenes del SAQ A-EP de PCI (PCI Security Standards Council, 2014; PCI Security Standards Council, 2014).

En cuanto a los puntos 2.1, 2.2 y 2.3: *buscar alcanzar una alta calidad en ambos los procesos y productos del trabajo profesional*, *mantener estándares elevados de competencia profesional, conducta y prácticas éticas*, y *conocer y respetar reglas existentes pertenecientes al trabajo profesional*; este proyecto fue realizado siguiendo los mejores estándares de calidad alcanzables por el equipo, y con el profesionalismo esperado. Siguiendo el punto 2.4, *aceptar y proveer evaluación profesional apropiada*, los miembros del equipo siempre hemos estado abiertos a críticas durante el desarrollo de una funcionalidad. Del lado de tecnología, los miembros encargados de investigar, o con mayor experiencia en un stack tecnológico, siempre han apoyado a miembros con menos experiencia en este, y evaluado la estructura del código. Así mismo, todo despliegue siempre recibía una validación de negocio para asegurar que resuelva los problemas correctamente.

El punto 2.5: *dar evaluaciones integrales y minuciosas de los sistemas de cómputo y sus impactos, incluyendo análisis de posibles riesgos*; puede ser elaborado. Debido a la ya mencionada información sensible almacenada en estos sistemas, se hizo especial énfasis en analizar riesgos que puedan resultar en brechas de datos. También, al momento de realizar análisis de los datos, se hizo hincapié en la necesidad de mantener estos reportes anonimizados o confidenciales. Muchas propuestas o solicitudes del lado operativo o de usuario, fueron rechazadas por no cumplir con las políticas de seguridad. Un ejemplo fue la constante incomodidad del equipo de negocio con contraseñas seguras y 2FA. Por medio de numerosos análisis de riesgos se logró que estos migren a gestores de contraseñas y autenticación de dos factores. Cabe resaltar que los componentes importantes desplegados por el equipo tecnológico siempre requieren contraseñas de alta complejidad, como por ejemplo el acceso a *backoffice* o el pgAdmin para usuarios de negocio.

El punto 2.6: *trabajar sólo en áreas donde se compete*; también puede ser elaborado. El equipo de tecnología es un equipo profesional en tecnología, por lo que todo análisis o decisión sobre información que no compete al área de tecnología fue enviado a las áreas respectivas. Por ejemplo, nunca se ofreció realizar *machine learning* u otra herramienta similar sobre data en la cual no se puede establecer un modelo apropiado.

Con respecto a los puntos 2.7, 2.8 y 2.9: *fomentar la conciencia pública y el entendimiento del cómputo, tecnologías relacionadas y sus consecuencias*, *acceder a los recursos de cómputo y comunicación solo cuando autorizado u obligado por el bien común*, y *diseñar e implementar*

sistemas robustos y usablemente seguros; siempre me he encargado de hacer entender al equipo de negocio cuales son las implicaciones y limitaciones del uso de la tecnología, sobre todo al realizar promesas a inversionistas potenciales. El acceso a los servidores siempre fue protegido y monitoreado para evitar acceso no autorizado. La plataforma en sí siempre fue revisada para evitar cualquier bug que pudiese resultar en la vulneración de datos personales. Todo dato ingresado es validado y ninguno interactúa directamente con la base de datos, evitando inyecciones de SQL que puedan vulnerar datos. El acceso de los usuarios a cada ruta es validado.

Los principios de liderazgo, de 3.1 a 3.5: *asegurarse que el bien común sea la preocupación principal durante todo el trabajo de cómputo profesional*, *articular, motivar la aceptación y evaluar el cumplimiento de las responsabilidades sociales por los miembros de la organización y el grupo, manejar personal y recursos para mejorar la calidad de la vida laboral*, *articular, aplicar y apoyar políticas y procesos que representan principios del código*, *crear oportunidades para que los miembros de la organización puedan crecer profesionalmente,* fueron aplicados en la gestión humana durante el proyecto. Durante los planeamientos siempre se solicitó estimados de velocidad al equipo para evitar fechas límite excesivamente cercanas, y que estas se ajusten a la velocidad del equipo. A solicitud del equipo de negocio, se otorgó a los miembros una cantidad de horas a la semana para que estos puedan trabajar en proyectos de su propia iniciativa que puedan contribuir al negocio. Muchos módulos que posteriormente pasaron a ser parte del stack tecnológico, como la primera versión en Python, bots para interacción con clientes o el módulo de categorización, iniciaron como iniciativas profesionales de un miembro. Al haber sido estos construidos por su promotor, estos pasaban a una posición de liderazgo dentro del módulo cuando éste llegaba a ser parte del proyecto.

Añadiendo el principio de liderazgo 3.6, *tener cuidado al modificar o retirar sistemas*, y *reconocer y tomar especial cuidado de sistemas que pasan a estar integrados en la infraestructura social*; este fue forzado a través del debido respeto a los sistemas de versionado. Al establecerse contratos internos de API, hubo especial cuidado al introducir modificaciones que puedan causar problemas a consumidores internos o terceros. El lanzamiento de una plataforma con una versión de API mayor implica el soporte a la misma. Un cambio en los resultados esperados pertenece a una siguiente versión mayor.

Debido a lo descrito, se puede afirmar que se cumplió con el punto 4.1 de la guía de la ACM, *defender, promover y respetar los principios establecidos por el código*. El punto 4.2, *tratar violaciones del código como inconsistentes a la membresía en la ACM*, no es aplicable al relacionarse a la membresía de la ACM.

LECCIONES APRENDIDAS

Inicié este proyecto como recién graduado en Ingeniería de Sistemas, armado con amplio conocimiento teórico listo para ser puesto en práctica. La amplia gama de responsabilidades que me fueron asignadas durante el desarrollo del proyecto me permitió recibir lecciones en la práctica y adquirir un gran bagaje de conocimientos prácticos.

Por ejemplo, durante el desarrollo de las numerosas versiones de Qempo Marketplace Backend, la arquitectura de la aplicación y los estándares de código variaron significativamente. ¿Por qué? El énfasis de la primera versión fue en estructura y código limpio. Si bien el código limpio es importante, es también importante que este sea fácil de modificar y práctico. Si se requieren rituales y/o boilerplate para añadir funcionalidades solo ralentiza el proceso de desarrollo innecesariamente. La primera versión requería largos rituales para añadir funcionalidades debido a su estructura rígida, pero "limpia": demostrando la importancia de también pensar en la flexibilidad del código.

También fue la primera vez que hice un proyecto con tantos cambios significativos. Para lidiar con el cambio, el uso de una política única de versionado dentro del entorno es extremadamente importante. El motivo es que esto permite a múltiples equipos identificar a simple vista cuando un contrato de API va a ser mantenido y cuando va a ser vulnerado. El uso de flujos de trabajo como Gitflow ayuda a mantener control del versionado y no perder el control de las funcionalidades. El uso de una nomenclatura estándar en este también es importante al no poder haber ambigüedades en los nombres de las ramas, lo cual facilita la organización del trabajo del equipo.

A lo largo del desarrollo del proyecto, la nomenclatura en la base de datos cambió al menos 3 veces. Esto debido al uso de nombres poco intuitivos o confusos, los cuales fueron reemplazados en versiones siguientes. La lección aprendida indica que, en el diseño de una base de datos para producción, se debe tener especial cuidado con la nomenclatura de los elementos, y se debe documentar los elementos de la misma para evitar que este conocimiento se pierda.

En este proyecto aprendimos a utilizar Scrum. Una metodología de este tipo no necesariamente tiene que ser seguida al pie de la letra, sino de la forma que mejor se adapte al equipo. El elemento más importante del desarrollo ágil con Scrum no son los rituales, sino la comunicación inducida por dichos rituales. Como ejemplo, el uso de *daily meetings* es un elemento de Scrum, pero no es la lección más importante de este.

Cómo lecciones por aprender quedan el uso de tecnología Cloud de más alto nivel como PaaS. A la fecha de mi graduación de la carrera, el entorno PaaS era todavía inmaduro y experimental, pero hoy en día con herramientas estandarizadas como Kubernetes, PaaS es un medio válido y robusto de *cloud* que ya no restringe al cliente a un solo proveedor de *cloud*. Una versión del proyecto hecha hoy en día probablemente haría valioso uso de esta tecnología.

También queda pendiente aplicar con mayor énfasis las pruebas unitarias dentro del proyecto. Muchas veces las pruebas quedaron de lado durante el desarrollo de este proyecto, debido a fechas de entrega o debido a la volatibilidad de los cambios. Esto es contrario a la metodología *test driven design (TDD)*, que pone las pruebas por delante del desarrollo.

A mi salida del proyecto se encontraba en planeamiento una nueva versión de este, desarrollada en Java, para conseguir un sistema más rápido y robusto. Esto probablemente no se habría

introducido a inicios del proyecto, sino que representa un paso más adelante en la evolución de este.

En lo personal, como mencioné al término de la parte 3, el siguiente paso es ingresar a equipos con mayor cantidad de miembros con diversos niveles de experiencia. Es evidente que muchos errores en el desarrollo de esta plataforma, los cuales resultaron en múltiples reescrituras de la misma podrían haber sido evitados por un fundamento más sólido propuesto por miembros con mayor experiencia.

GLOSARIO DE TERMINOS

- *e-commerce* – Actividad comercial sobre plataformas digitales, generalmente la venta de productos o servicios.
- *Ticket* – Valor bruto de un pago realizado por un consumidor.
- *Backend* – Software que se ejecuta en el servidor y gestiona el acceso a los datos y lógica de negocio.
- *Frontend* – Software que se ejecuta en el equipo del cliente y gestiona la presentación de la información.
- *Backoffice* – Plataforma de uso interno para personal del negocio.
- API – Interfaz para interacción entre aplicaciones. Son instrumentales al orquestar múltiples aplicaciones.
- Librería – Software con utilidades que puede ser enlazado a otro proyecto de software. Puede proporcionar funcionalidades comunes que son constantemente requeridas para proyectos.
- *Framework* – Una librería de mayor tamaño, que generalmente prescribe más detalles de implementación para el usuario de esta.
- *Hotfix* – Cambio desplegado directamente a la rama de producción después de pruebas, generalmente para arreglar un problema urgente en el entorno de producción.
- *Daily meetings* – Reuniones de 15 minutos donde se realiza un intercambio de qué se hizo, qué se hará y qué problemas hubo por cada miembro del equipo. Es parte del método SCRUM.
- *Lint* – Análisis de estilo de código fuente. Identifica código que no se adhiere a las convenciones o que potencialmente pueda adquirir errores de semántica por su forma.
- *Uptime* – Tiempo en servicio de una aplicación. El contrario de *uptime* es *downtime*, un periodo de tiempo sin servicio.
- DAO – Patrón de diseño que consiste en el uso de un objeto para mediar el acceso a los datos. Suele ser menos flexible que otros patrones más complejos.

REFERENCIAS

ACM Code 2018 Task Force. (2018). ACM Code of Ethics and Professional Conduct. *ACM Code of Ethics and Professional Conduct*. Retrieved July 10, 2021, from https://www.acm.org/code-of-ethics

Atlassian Corporation. (n.d.). Gitflow Workflow. *Gitflow Workflow*. Retrieved July 1, 2021, from https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

Carbon60. (n.d.). GTmetrix. *GTmetrix*. Retrieved July 1, 2021, from https://gtmetrix.com/

Chacon, S., & community, G. (2021). Git - git-bisect. *Git - git-bisect*. Retrieved July 1, 2021, from https://git-scm.com/docs/git-bisect

Chacon, S., & Git community. (2021). Git - git-blame. *Git - git-blame*. Retrieved June 1, 2020, from https://git-scm.com/docs/git-blame

Chacon, S., & Straub, B. (2014). Git Branching - Rebasing. *Git Branching - Rebasing*. Retrieved July 1, 2021, from https://git-scm.com/book/en/v2/Git-Branching-Rebasing

Choudary, S. P., Van Alstyne, M. W., & Parker, G. G. (2016). *Platform Revolution: How Networked Markets Are Transforming the Economy–And How to Make Them Work for You* (1st ed.). W. W. Norton & Company.

Deb, S. (2014). Information technology, its impact on society and its future. *Advances in Computing, 4*, 25–29.

Dijkstra, E. W. (1982). On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective* (pp. 60–66). Springer-Verlag.

Driessen, V. (2010). A successful Git branching model. *A successful Git branching model*. Retrieved July 1, 2021, from https://nvie.com/posts/a-successful-git-branching-model/

Fowler, M. (2006). Continuous Integration. *Continuous Integration*. Retrieved June 1, 2020, from https://martinfowler.com/articles/continuousIntegration.html

Fowler, M., Rice, D., Foemmel, M., Hieatt, E., Mee, R., & Stafford, R. (2002). *Patterns of Enterprise Application Architecture.* Addison-Wesley Professional.

Functional Software, Inc. (n.d.). Sentry. *Sentry*. Retrieved July 1, 2021, from https://sentry.io/

Gao, Z., Bird, C., & Barr, E. T. (2017). To Type or Not to Type: Quantifying Detectable Bugs in JavaScript. *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, (pp. 758–769). doi:10.1109/ICSE.2017.75

Google Inc. (n.d.). Google Analytics. *Google Analytics*. Retrieved July 1, 2021, from https://analytics.google.com

Google Inc. (n.d.). Google Lighthouse. *Google Lighthouse*. Retrieved Julio 1, 2021, from https://developers.google.com/web/tools/lighthouse

Haerder, T., & Reuter, A. (1983). Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys, 15*, 287–317.

Knowledge@Wharton. (2016). Network Revolution: Creating Value Through Platforms, People and Technology. *Network Revolution: Creating Value Through Platforms, People and Technology*. Retrieved July 1, 2021, from https://knowledge.wharton.upenn.edu/article/the-network-revolution-creating-value-through-platforms-people-and-digital-technology/

Langa, Ł., Willing, C., Meyer, C., Zijlstra, J., Naylor, M., Dollenstein, Z., . . . Si, R. (2021). Black - The Uncompromising Code Formatter. *Black - The Uncompromising Code Formatter*. Retrieved June 1, 2020, from https://github.com/psf/black

Leveson, N. G., & Turner, C. S. (1993). An investigation of the Therac-25 accidents. *Computer, 26*, 18–41. doi:10.1109/MC.1993.274940

Ley de Protección de Datos Personales. (2011). *Ley de Protección de Datos Personales*. Retrieved from https://diariooficial.elperuano.pe/pdf/0036/ley-proteccion-datos-personales.pdf

Martin, R. C. (2014). Clean Architecture and Design. *Clean Architecture and Design*. Retrieved June 1, 2020, from https://www.youtube.com/watch?v=Nsjsiz2A9mg

Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design* (1st ed.). USA: Prentice Hall Press.

Mei, S. (2013). Why You Should Never Use MongoDB. *Why You Should Never Use MongoDB*. Retrieved July 1, 2021, from http://www.sarahmei.com/blog/2013/11/11/why-you-should-never-use-mongodb/

Metabase. (n.d.). Metabase. *Metabase*. Retrieved July 1, 2021, from https://www.metabase.com/

Misa, T. J. (2007). Understanding'how computing has changed the world'. *IEEE Annals of the History of Computing*, *29*, 52–63.

Mørken, F. V. (2017). Why you should stop using Git rebase. *Why you should stop using Git rebase*. Retrieved July 1, 2021, from https://medium.com/@fredrikmorken/why-you-should-stop-using-git-rebase-5552bee4fed1

Olery Developer Portal. (2017). Goodbye MongoDB, Hello PostgreSQL. *Goodbye MongoDB, Hello PostgreSQL*. Retrieved July 1, 2021, from https://developer.olery.com/blog/goodbye-mongodb-hello-postgresql/

OpenJS Foundation. (2021). Node.js Previous Releases. *Node.js Previous Releases*. Retrieved July 1, 2021, from https://nodejs.org/en/download/releases/

PCI Security Standards Council. (2014). Payment Card Industry (PCI) Data Security Standard - Self-Assessment Questionnaire A-EP and Attestation of Compliance. *Payment Card Industry (PCI) Data Security Standard - Self-Assessment Questionnaire A-EP and Attestation of Compliance*. Retrieved July 1, 2021, from https://www.pcisecuritystandards.org/documents/SAQ_A-EP_v3.pdf

PCI Security Standards Council. (2014). Payment Card Industry (PCI) Data Security Standard - Self-Assessment Questionnaire D and Attestation of Compliance for Merchants. *Payment Card Industry (PCI) Data Security Standard - Self-Assessment Questionnaire D and Attestation of Compliance for Merchants*. Retrieved July 1, 2021, from https://www.pcisecuritystandards.org/documents/SAQ_D_v3_Merchant.pdf

Preston-Werner, T. (n.d.). Semantic Versioning 2.0.0. *Semantic Versioning 2.0.0*. Retrieved July 1, 2021, from https://semver.org/

Provos, N., & Mazieres, D. (1999). A Future-Adaptable Password Scheme. *USENIX Annual Technical Conference, FREENIX Track*, (pp. 81–91).

Python Software Foundation. (2021). Python Documentation by Version. *Python Documentation by Version*. Retrieved July 1, 2021, from https://www.python.org/doc/versions/

Python Software Foundation. (2021). The Python Standard Library - asyncio — Asynchronous I/O. *The Python Standard Library - asyncio — Asynchronous I/O*. Retrieved July 1, 2021, from https://docs.python.org/3/library/asyncio.html

Rigo, A., & Tismer, C. (2011). greenlet: Lightweight concurrent programming. *greenlet: Lightweight concurrent programming*. Retrieved July 1, 2021, from https://greenlet.readthedocs.io/en/latest/

Sequelize Team. (n.d.). Sequelize API Reference - Eager Loading. *Sequelize API Reference - Eager Loading*. Retrieved July 1, 2021, from https://sequelize.org/master/manual/eager-loading.html

SolarWinds Worldwide, LLC. (n.d.). Pingdom. *Pingdom*. Retrieved July 1, 2021, from https://tools.pingdom.com/

Sotirov, A., Stevens, M., Appelbaum, J., Lenstra, A. K., Molnar, D., Osvik, D. A., & de Weger, B. (2008). MD5 considered harmful today, creating a rogue CA certificate. *25th Annual Chaos Communication Congress.*

SQLAlchemy authors and contributors. (2021). SQLAlchemy 1.4 Documentation - Relationship Loading Techniques. *SQLAlchemy 1.4 Documentation - Relationship Loading Techniques*. Retrieved July 1, 2021, from https://docs.sqlalchemy.org/en/14/orm/loading_relationships.html

Stackless Team. (2021). About Stackless. *About Stackless*. Retrieved July 1, 2021, from https://github.com/stackless-dev/stackless/wiki

The Economist. (2016, September). From zero to seventy (billion). *The Economist*. Retrieved from https://www.economist.com/briefing/2016/09/03/from-zero-to-seventy-billion

The pgAdmin Development Team. (n.d.). pgAdmin. *pgAdmin*. Retrieved July 1, 2021, from https://www.pgadmin.org/

Van Rossum, G. (2001). PEP 8 - Style Guide for Python Code. *PEP 8 - Style Guide for Python Code*. Retrieved July 1, 2021, from https://www.python.org/dev/peps/pep-0008/

Van Rossum, G., Lehtosalo, J., & Langa, Ł. (2014). PEP 484 - Type Hints. *PEP 484 - Type Hints*. Retrieved July 1, 2021, from https://www.python.org/dev/peps/pep-0484/

Williams, C. (2016). How one developer just broke Node, Babel and thousands of projects in 11 lines of JavaScript. *How one developer just broke Node, Babel and thousands of projects in 11 lines of JavaScript*. Retrieved from https://www.theregister.com/2016/03/23/npm_left_pad_chaos/

Zamot, M. (2020). An introduction to Cockpit, a browser-based administration tool for Linux. *An introduction to Cockpit, a browser-based administration tool for Linux*. Retrieved July 1, 2021, from https://www.redhat.com/sysadmin/intro-cockpit

BIBLIOGRAFÍA

Choudary, S. P., Van Alstyne, M. W., & Parker, G. G. (2016). *Platform Revolution: How Networked Markets Are Transforming the Economy–And How to Make Them Work for You* (1st ed.). W. W. Norton & Company.

Deb, S. (2014). Information technology, its impact on society and its future. *Advances in Computing, 4*, 25–29.

Dijkstra, E. W. (1982). On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*. Springer-Verlag.

Fowler, M., Rice, D., Foemmel, M., Hieatt, E., Mee, R., & Stafford, R. (2002). *Patterns of Enterprise Application Architecture.* Addison-Wesley Professional.

Gamma E., Helm R., Johnson R. & Vlissides J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley Longman Publishing Co., Inc.

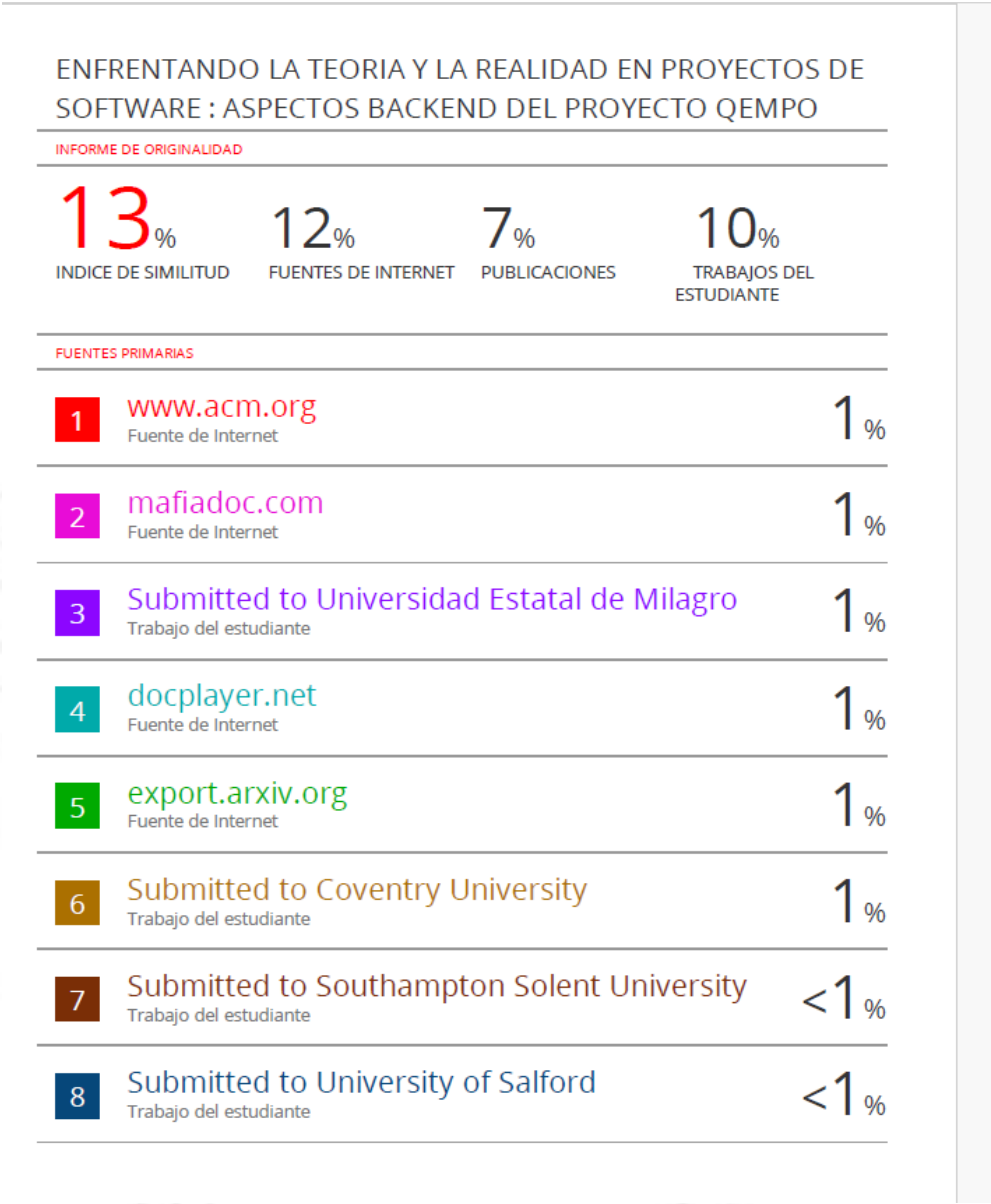
Gao, Z., Bird, C., & Barr, E. T. (2017). To Type or Not to Type: Quantifying Detectable Bugs in JavaScript. *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, (pp. 758–769). doi:10.1109/ICSE.2017.75

Haerder, T., & Reuter, A. (1983). Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys, 15*, 287–317.

Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design* (1st ed.). USA: Prentice Hall Press.

Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship* (1st ed.). USA: Prentice Hall PTR.

Martin, R. C. (2011). *The Clean Coder: A Code of Conduct for Professional Programmers* (1st ed.). USA: Prentice Hall Press.



ENFRENTANDO LA TEORIA Y LA REALIDAD EN PROYECTOS DE SOFTWARE : ASPECTOS BACKEND DEL PROYECTO QEMPO

INFORME DE ORIGINALIDAD

13%	12%	7%	10%
INDICE DE SIMILITUD	FUENTES DE INTERNET	PUBLICACIONES	TRABAJOS DEL ESTUDIANTE

FUENTES PRIMARIAS

1	www.acm.org Fuente de Internet	1%
2	mafiadoc.com Fuente de Internet	1%
3	Submitted to Universidad Estatal de Milagro Trabajo del estudiante	1%
4	docplayer.net Fuente de Internet	1%
5	export.arxiv.org Fuente de Internet	1%
6	Submitted to Coventry University Trabajo del estudiante	1%
7	Submitted to Southampton Solent University Trabajo del estudiante	<1%
8	Submitted to University of Salford Trabajo del estudiante	<1%