Universidad de Lima

Facultad de Ingeniería

Carrera de Ingeniería de Sistemas

APRENDIZAJE CONTINUO PARA EL APOYO Y ASISTENCIA A NECESIDADES COMERCIALES DINÁMICAS: EL CASO QEMPO

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

Christian Armando Llanos Mercado

Código 20110669

Asesor

Jorge Luis Irey Nuñez

Lima – Perú

Mayo de 2023

UNIVERSIDAD DE LIMA
MCMLXIV
SCIENTIA ET PRAXIS

CONTINUOUS LEARNING FOR SUPPORT AND ASSISTANCE TO DYNAMIC BUSINESS NEEDS: THE QEMPO CASE

TABLA DE CONTENIDO

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

RESUMEN

Una de las cualidades que distingue a un bachiller egresado de la carrera de Ingeniería de Sistemas de la Universidad de Lima es la capacidad de identificar, planificar y aplicar soluciones con un enfoque en tecnologías de la información de manera efectiva de forma tal que la solución satisface las necesidades del negocio, y en consecuencia, las del mercado cuyo amplio abanico puede ir desde emprendimientos hasta mercados más grandes y estables como los de bienes raíces. Inicié mi carrera profesional como desarrollador de aplicaciones móviles en el negocio de un amigo siendo partícipe de la fundación y creación del mismo, logrando ver el crecimiento y progreso del mismo a la par del avance en mi carrera profesional. Posteriormente participé en un equipo de ingenieros desarrolladores que me permitió interactuar en diferentes proyectos donde pude aplicar los conocimientos adquiridos para contribuir con mejoras en la estructura de las mismas, así como en la integración y entrega continua del software. Actualmente, soy parte de en un equipo encargado de volver a desarrollar una aplicación que es parte de un sistema bastante complejo, desde la búsqueda y compra de casas hasta la renta y mantenimiento de las mismas y en donde me encuentro tomando algunas decisiones para encaminar el proyecto, mantener la comunicación con el sistema existente y contribuir en la automatización de las etapas del despliegue e integración de software. Mi siguiente reto será ser líder técnico para supervisar, mejorar y tomar decisiones de infraestructura y arquitectura con un equipo a mi cargo.

Palabras clave: Emprendimiento, tecnología, desarrollo continuo, integración continua, aplicaciones, bienes raíces, sistemas heredados.

ABSTRACT

One of the qualities that distinguishes me as a Bachelor of Systems Engineering degree from the University of Lima is the ability to identify, plan and apply solutions with a focus on information technology effectively in such a way that the solution satisfies the needs of the business, and consequently, those of the market whose wide range can from startups up to larger and more stable markets such as real estate. I started my professional career as a mobile application developer in a friend's business, being a participant in its foundation and creation, managing to see its growth and progress along with the advancement in my professional career. Later I participated in a team of developer engineers that allowed me to interact in different projects where I was able to apply the knowledge acquired to contribute to improvements in their structure as well as in the integration and continuous delivery of the software. Lately, I am part of a team in charge of redeveloping an application that is part of a fairly complex system, from the search and purchase of houses to renting and maintaining them, and where I find myself making some decisions to direct the project, maintain communication with the existing system and contribute to the automation of the software deployment and integration stages. My next challenge will be to be a technical leader to supervise, improve and make decisions on infrastructure and architecture with a team in my charge.

Keywords: Entrepreneurship, technology, continuous development, continuous integration, applications, real estate, legacy systems.

INTRODUCCIÓN

A lo largo de mi carrera profesional en la empresa Curiosity Startup, que luego se convertiría en Qempo, además de mi breve paso por el laboratorio de investigación de la Universidad de Lima, he podido aprovechar las oportunidades de aprendizaje en múltiples áreas dentro de la creación e implementación de soluciones tecnológicas.

Una de las primeras oportunidades que se aprovecharon fue la libertad de aprender a crear una aplicativo Android con solo las bases del lenguaje Java que la carrera universitaria me dio. Aunque poco desarrollada, la habilidad de aprender por uno mismo incrementó considerablemente. Así, como programador y autodidacta, los siguientes retos en el proyecto más largo de mi carrera fueron superados con apoyo de mis compañeros.

Luego de encontrar una estructura de archivos y un modelo de arquitectura que logre un proyecto desacoplado y escalable para aplicaciones Android, entendí que ser parte de un negocio en crecimiento es ser cambiaste y aceptarlo para mejorar. Esto último a razón de que el proyecto Android fue cancelado para inmediatamente enfocar los esfuerzos en la solución web de Qempo. Una de las consecuencias de esta cultura de cambio es que aprendí un lenguaje nuevo y terminé mis labores en la empresa aprendiendo a crear bases de proyectos backend en Python.

Qempo, una plataforma e-commerce de productos en el extranjero, busca unir a compradores con viajeros y agencias de importación para que el primer grupo pueda acceder a un mercado extranjero y el segundo generar ganancias de un viaje o tener un nuevo canal de clientes. Este negocio necesitó de una gestión de proyecto a través de herramientas como KanBan y Scrum, de integración continua para mejorar la entrega de software y muchos ciclos de aprendizaje.

La mejora continua como parte de la cultura organizacional de la empresa llevó al equipo a integrar la gestión de proyectos con la creación de software seguro y escalable. Para hacerlo seguro se aprendió mucho de las vulnerabilidades y riesgos del sistema, para hacerlo escalable se desacopló múltiples veces el proyecto. Así mismo, la adición de KPI's, *Key Performance Indicator*, por sus siglas en inglés, incrementó la importancia de cuantificar,

medir y analizar el comportamiento de los usuarios y de nuestros procesos. Todo esto a través de ciclos de desarrollo llamados *sprints*.

A continuación, se darán detalles del proceso de aprendizaje, la toma de decisiones y las consecuencias de ellas a través de los primeros proyectos profesionales en los que estuve involucrado y los hitos que en ellas tuvieron lugar.

1. CAPACIDAD TÉCNICA

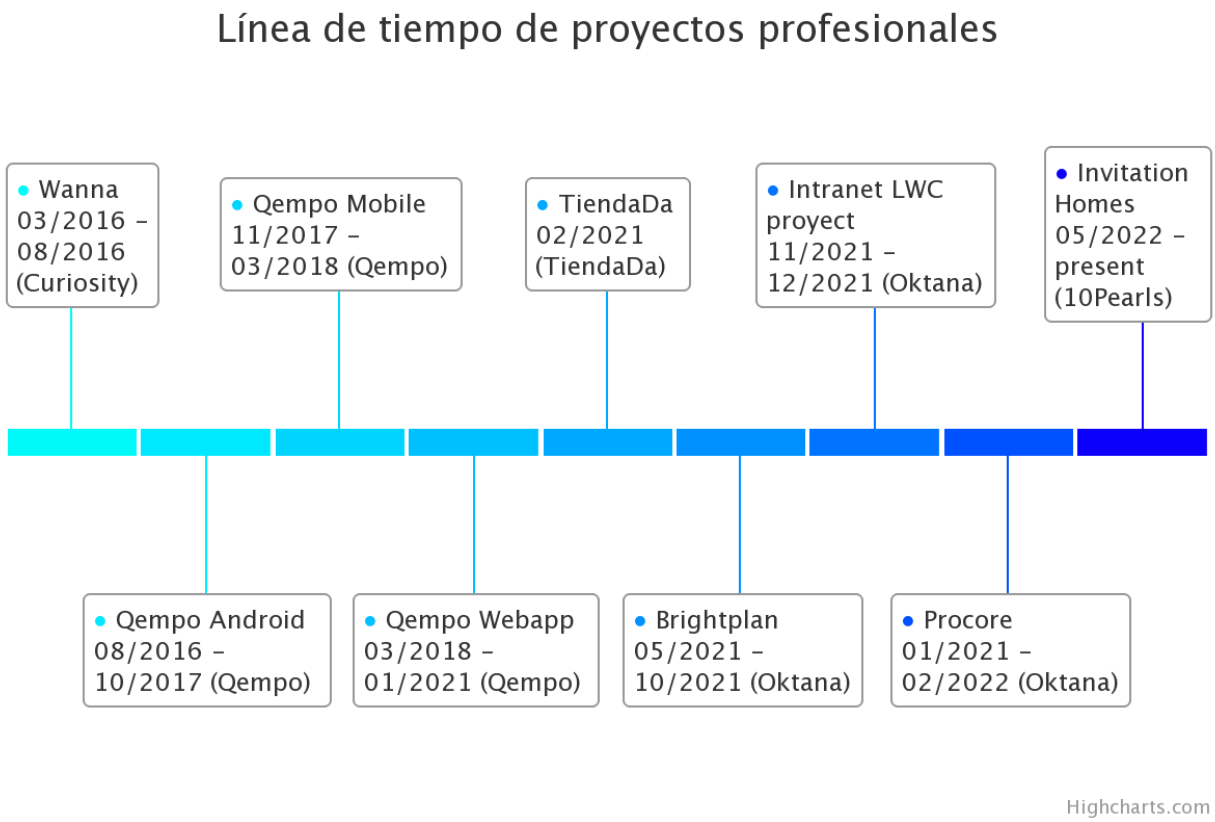
Durante mi desarrollo profesional he participado en diferentes proyectos cuyo espacio de tiempo se puede observar en la Figura 1.1.

Qempo, TiendaDa y Wanna son tres proyectos que se desarrollaron durante el tiempo que trabajé en la empresa Qempo LLC, antes llamada Curiosity Startup. La primera es la más longeva, por lo que tuvo muchas iteraciones y cambios en el camino. La segunda, aunque fue breve mi aporte, fue una aplicación de todos mis conocimientos adquiridos a lo largo de mi carrera. Y la última, un aplicativo Android, el primer proyecto profesional que tuve.

A continuación, en la Figura 1.1 se describen los principales problemas que tuvimos con las áreas de negocio y tecnología, así como las decisiones en tecnologías de la información y desarrollo de software.

Figura 1.1

Línea de tiempo de proyectos profesionales



a) Wanna Android app

La empresa Curiosity Startup se creó con el propósito de crear y materializar ideas de negocio, poder generar aplicaciones, software que satisfaga la idea y lanzarlo al mercado con el fin de tener un crecimiento acelerado. Wanna fue la primera idea de negocio que se tomó, el objetivo era facilitar la coordinación de eventos sociales entre personas para lo cual se plantearon como principales requerimientos:

- Coordinar los horarios de los participantes para poder ofrecer un horario libre en común.
- Poder monetizar a través de publicidad acorde a la categoría del evento además de recibir notificaciones push.
- Registrar a los usuarios a través de su número de celular, así como permitir una conversación entre ellos.

Como toda idea de negocio nueva o innovadora se requería una prueba de concepto (PoC por sus siglas en inglés) que permitiera verificar o comprobar si el mercado objetivo aceptaba el planteamiento del negocio. Para cumplir con este producto se organizó el proyecto denominado Wanna Android App en el cual se tuvieron que tomar algunas decisiones de arquitectura y diseño:

- Utilizar librerías para gestionar las notificaciones, envío y recepción de mensajes privados, y llamadas al servidor.
- Utilizar el patrón repositorio para la abstracción de datos con inyección de dependencias como patrón de diseño de software.

La librería usada para gestionar las notificaciones fue PushBots (https://pushbots.com), un gestor de notificaciones para enviar, recibir y/o interactuar con el usuario, así como producir análisis de usuarios y su interacción con el aplicativo. Socket-io (https://socket.io) fue la librería para crear nuestra mensajería, brinda una comunicación de baja latencia, bidireccional y basada en eventos entre el cliente, los usuarios, y un servidor, el aplicativo. Retrofit (https://square.github.io/retrofit), para llamados HTTP, es un cliente HTTP con anotación para Android y Java.

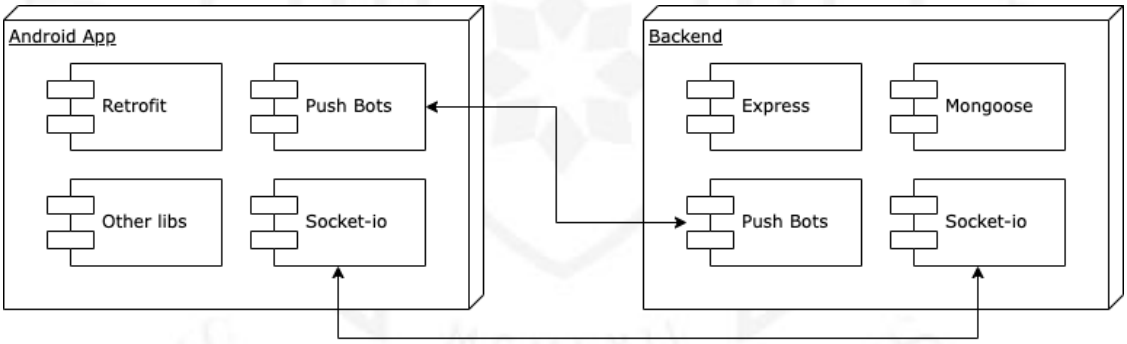
Entre otras librerías, se usó Butter Knife (https://jakewharton.github.io/butterknife/) para enlazar la vista con los campos y métodos de la Actividad. Picasso

(https://square.github.io/picasso), para descargar y almacenar en caché imágenes. Ask (https://github.com/00ec454/Ask), para manejar la verificación de permisos en tiempo de ejecución. Y finalmente, una librería de EverythingMe (https://github.com/EverythingMe/easy-content-providers) para poder extraer los contactos y calendarios del usuario.

Como se puede observar en la Figura 1.2, la aplicación estaba conformada por dos frentes: frontend y backend. El primero es el dispositivo móvil Android donde se instala la aplicación con las librerías mencionadas. El segundo es una aplicación escrita en el lenguaje de programación Javascript con el marco de trabajo web Express (https://expressjs.com) para la interacción entre los usuarios y los llamados HTTP. Para la base de datos en MongoDB (https://www.mongodb.com) se usó el modelador de objetos Mongoose (https://mongoosejs.com). De forma similar al frontend, se usaron las librerías PushBots y Socket-io para las notificaciones push y la mensajería.

Figura 1.2

Diagrama de componentes del proyecto Wanna



Esta prueba de concepto fue revisada por los *stakeholders* (personas o partes interesadas). Su decisión fue priorizar otros proyectos en la cartera de la empresa por la falta de recursos de programación y el mayor interés por sacar adelante dichos proyectos. Debido a esto, la aplicación Android no fue publicada en el Play Store ni el backend desplegado en algún servidor. Una vez tomada la decisión de suspender el proyecto, continué con el siguiente proyecto de mayor interés, la aplicación Android de su negocio más rentable: Qempo.

b) Qempo Android app

El proyecto Qempo fue la primera idea que tuvieron los dos fundadores de Curiosity Startup mucho antes de constituir la empresa como tal. En sus inicios, bajo el nombre de "Anson Import", se dedicaban a importar productos de Estados Unidos a Perú a través de mensajes directos en sus correos electrónicos o cuentas de foros peruanos. De forma empírica entendieron que había un mercado insatisfecho que podía crecer, una oportunidad de negocio. En el inicio era un modelo de negocio *B2C* (*Business to consumer* por sus siglas en inglés). Todo el proceso de compra y venta, así como la adquisición de clientes, era realizado por los mismos actores.

En esta etapa, se detectaron dos problemas claves: complicaciones para llevar registro de los pedidos y dificultad para hacer entender a los clientes el giro del negocio. Con esto en mente, se trazó la meta de mejorar el uso de los recursos operativos y crear una página web más amigable al usuario. Como solución a estos problemas y para cumplir con las metas, se levantaron los requerimientos funcionales y no funcionales en la Tabla 1.

Tabla 1.1

Requerimientos de ERP para Qempo B2C

Requerimientos funcionales	Requerimientos no funcionales
● Agregar estados a los pedidos: pendiente, pagado, entregado. ● Administrar usuarios. ● Crear una web informativa.	● Sencillo de implementar y mantener. ● Preservar un costo mensual no mayor a cien dólares.

La solución fue usar Magento (http://www.magento.com), una plataforma de comercio electrónico. Aquí se pudo crear una web que explica a los usuarios los pasos para usar los servicios y también mejorar la gestión de pedidos a través de su panel administrativo. Con el éxito de la solución, el problema de la gestión de órdenes volvió a surgir por ser solo una persona la que manejaba la herramienta del lado del negocio y la cantidad de nuevos clientes adquiridos.

Con la nueva herramienta ERP funcionando, los problemas a notar fueron la falta de recursos operativos (OPEX), el modelo de negocio impedía escalar por requerir una proporcionalidad entre sus clientes y trabajadores, y, por último, la plataforma tenía limitaciones para configurar a las necesidades de la empresa. De forma similar a la solución

de Magento, los requerimientos funcionales y no funcionales fueron identificados y se muestran en la Tabla 1.2.

Tabla 1.2

Requerimientos de página web para Qempo y Magento

Requerimientos funcionales	Requerimientos no funcionales
● Registrar usuarios con cuenta de Facebook o correo electrónico. ● Crear pedido de importación. ● Crear registro de viaje. ● Cotizar un pedido de importación para llevar en un viaje. ● Visualizar órdenes. Como comprador, ver los pedidos cotizados por viajeros, y como viajero, dichas cotizaciones. ● Cancelar reserva (cotización de pedido) por viajero o comprador. ● Nuevos estados a las órdenes: comprado, en Perú, entregado y confirmado. ● Delegar la transición de los nuevos estados a los viajeros y compradores.	● Infraestructura de software pequeña. ● Consumir pocos recursos computacionales. ● Usar marcos de trabajo completos en vez de micro marcos de trabajo. ● Reducir el tiempo de entrega de nuevas funcionalidades.

Para este punto en el tiempo, con la decisión de enfocar los recursos al proyecto Qempo, se propuso crear un aplicativo Android para complementar el proyecto web que nacería con los requerimientos previamente mencionados. Dicho aplicativo tuvo dos iteraciones, el primero fue una PoC y el segundo, aunque quedó en desarrollo, se hizo con intención de entregar a los viajeros acceso beta, es decir, un acceso temprano y exclusivo a un grupo seleccionado de usuario para que usen, comenten, reseñen y opinen sobre las funcionalidades. La primera versión del proyecto tuvo los mismos requerimientos funcionales y no funcionales a los de su contraparte web, pero, además, contó con los descritos en la Tabla 1.3.

Tabla 1.3

Requerimientos de aplicación PoC en Android

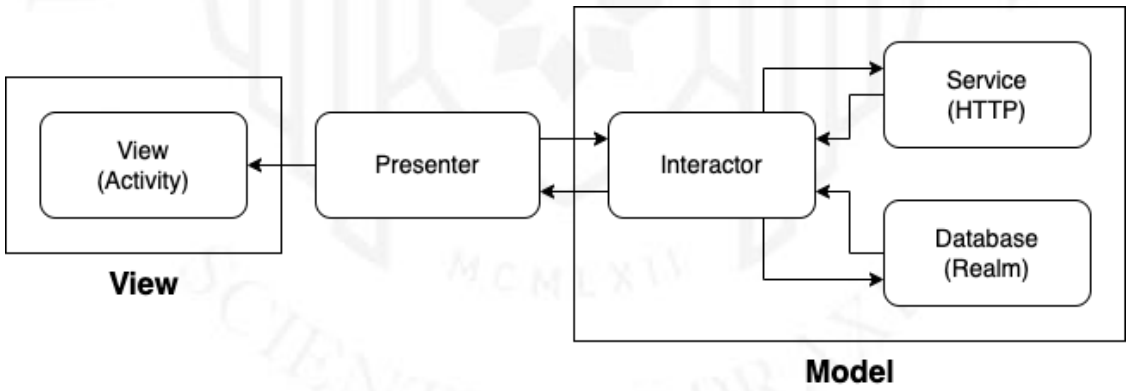
Requerimientos funcionales	Requerimientos no funcionales
● Mensajería inmediata entre viajero y comprador cuando exista una reserva. ● Clasificación de una a cinco estrellas entre viajeros y compradores.	● Persistir información de viajes, pedidos y órdenes. ● Monitorear y registrar problemas de estabilidad en el aplicativo.

El modelo de arquitectura que se usó y se muestra en la Figura 1.3 es una variación del Model-View-Presenter (MVP) para construir las interfaces de usuario y tener la lógica de presentación separada y definida. Nuestra vista era la actividad (Activity) que extendía una interfaz, dicha interfaz era usada por el presentador (Presenter) por medio de la creación de una instancia para ejecutar funciones en la *activity*. De forma similar, el *presenter*, a cargo de comunicar la actividad con el interactuador (Interactor), extendía una interfaz para ser usado por el mencionado *interactor*. Este último era el encargado de hacer los llamados HTTP a través de un servicio (Service) y llamados a la base de datos no relacional a través de un singleton que la misma librería de base de datos expone. Una vez que recibe una respuesta, el *interactor* procede enviar los datos a la actividad por medio del presentador. Su función no solo es transmitir información sino también realizar cambios de o en la vista.

En otras palabras, cuando el usuario interactúa con la aplicación, la actividad ejecuta una acción que pasa por el *presenter* al interactor. Luego, la respuesta que vuelve es transmitida desde el *interactor* hacía la vista a través del presentador.

Figura 1.3

Modelo de arquitectura de Qempo Android app, PoC



Como parte de la solución requería mantener información *offline*, se recurrió a una librería llamada Realm (https://realm.io/) que nos sirvió como base de datos no relacional orientada a objetos. Por otro lado, para mantener una conversación síncrona entre los usuarios, la solución fue la misma que en Wanna, Socket.io, porque nos permitió mantener un canal abierto por dónde enviar los mensajes.

Para obtener reportes comprensivos de nuestros *bugs* y fallas se implementó el uso de la librería Crashlytics (https://firebase.google.com/products/crashlytics), con esto se

podía tener los registros de los momentos en que el aplicativo fallaba u ocurría algo inesperado. Tener acceso a estos registros permitió que se tuviera un mejor control sobre los *bugs* y errores en el aplicativo. Gran parte de los registros eran advertencias sobre algunas dependencias, mientras que una parte eran errores que no se habían contemplado en los casos de uso y otra parte eran varios tipos de errores como compatibilidad, llamados al servidor u optimización de la misma app.

La PoC se presentó a los *stakeholders* y la respuesta fue positiva. Decidieron seguir con el siguiente paso: desarrollar una nueva app con los mismos requerimientos funcionales. Dicha versión de Qempo Android se realizaría con nuevo diseño y una estructura de proyecto diferente, pero con el mismo modelo de arquitectura. Adicionalmente, se usó inyección de dependencias para separar la lógica del negocio del código pertinente a Android. Así, se obtuvieron algunos requerimientos adicionales para el aplicativo Android que se muestran en la Tabla 1.4.

Tabla 1.4

Requerimientos de primera versión de aplicación Android

Requerimientos no funcionales
● Prescindir del monitoreo y registro de accidentes en la aplicación. ● Pruebas unitarias. ● Uso de librerías: RxJava, Dagger y Lombok.

Las pruebas unitarias se realizaron con la librería Robolectric (http://robolectric.org/) y Mockito (https://site.mockito.org/). La primera ejecuta las pruebas en un entorno de Android simulado dentro de una JVM, sin el peso y las irregularidades de un emulador. La segunda, es un marco de trabajo estándar en Android para *mockear* objetos, es decir simular objetos que imitan el comportamiento de objetos reales de forma controlada.

Como parte de la mejora de mis habilidades de desarrollo, se decidió agregar el uso de RxJava (https://github.com/ReactiveX/RxJava) una implementación Java VM (*Virtual Machine* por sus siglas en inglés) de ReactiveX (https://reactivex.io/), una librería para componer programas asincrónicos y basados en eventos usando el patrón observador. Específicamente, para tener un mayor con control de los llamados al servidor, control como un mejor manejo de errores y efectos secundarios que los *callback* de Retrofit no permitía. De forma similar, Lombok (https://projectlombok.org/) para minimizar y/o eliminar código

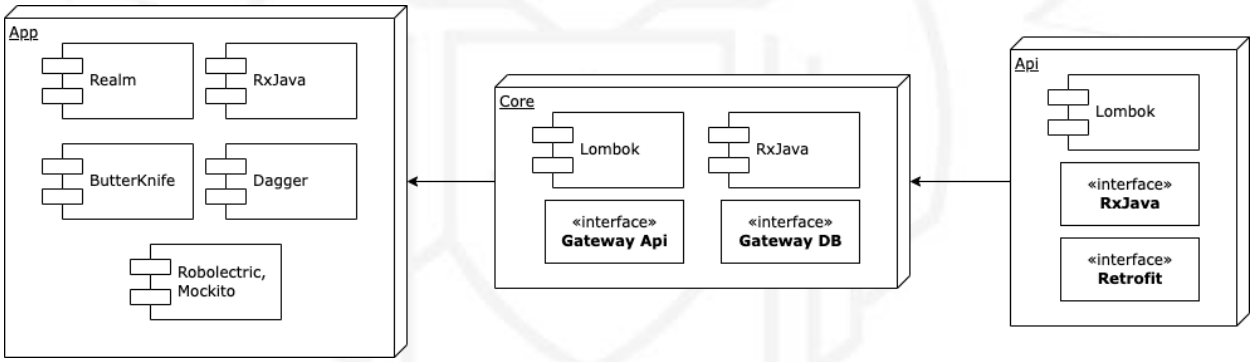
repetitivo. La forma en que funciona es al conectarse al proceso de compilación y generar automáticamente Java bytecode en los archivos *.class* según la serie de anotaciones que se introducen en el código.

Por último, Dagger (https://dagger.dev/), como librería de inyección de dependencias. Un patrón de diseño en el que un objeto o función recibe otros objetos o funciones de los que depende. Una forma de inversión de control, o IoC por sus siglas en inglés, la inyección de dependencias tiene como objetivo separar las molestias de construir objetos y usarlos, lo que resulta en programas con bajo acoplamiento.

El uso de estas librerías, como se mencionó anteriormente, permitió la separación de la lógica del negocio, la capa de presentación y los servicios externos. En el proyecto se dividieron en App, Core y Api, respectivamente, como se aprecia en la Figura 1.4.

Figura 1.4

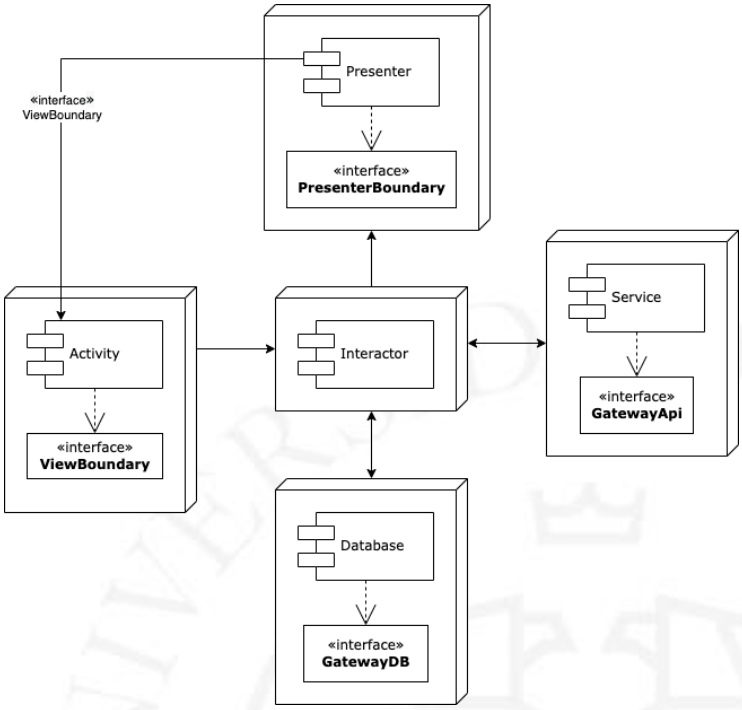
Diagrama de componentes de Qempo Android App



El modelo de arquitectura para la nueva versión siguió basado en MVP. Sin embargo, como se puede apreciar en la Figura 1.5 gracias al uso de Dagger la comunicación entre la vista, el presentador, el interactor, el servicio API y la base de datos, se realizaba a través de interfaces. *ViewBoundary*, la interfaz para comunicar acciones a realizar en la vista perteneciente a la App o capa de presentación. *PresenterBoundary*, la interfaz para comunicar las acciones de la lógica de negocio o Core hacia la App. *GatewayDB* y *GatewayApi*, ambas interfaces para comunicarnos con los servicios externos como el servicio API y la base de datos. Por último, el *interactor* se proveía de las instancias necesarias y ejecutaba las acciones necesarias entre los distintos componentes.

Figura 1.5

Modelo de arquitectura de Qempo Android app, segunda versión



En medio del desarrollo de la segunda versión, los *stakeholders* decidieron cambiar sus objetivos para darle mayor enfoque al proyecto Qempo. La empresa Curiosity Startup cesó de existir y Qempo se convirtió en el nuevo negocio y única prioridad. Esto significó, aparte de cancelar el desarrollo de otros proyectos de la matriz, cancelar el desarrollo del aplicativo Qempo Android y empezar un nuevo proyecto que abarque a los usuarios Apple y Android: Qempo Mobile, con el marco de trabajo de React Native.

c) Qempo Mobile (React Native App)

Con la meta de alcanzar un público más amplio, se buscaron alternativas de aplicaciones híbridas que pudieran darnos la oportunidad de desarrollar para ambos públicos: Android y iOS.

El equipo conocía por proyectos anteriores acerca de la existencia de *frameworks* como Phonegap (ahora conocido como Cordova) y Ionic. Así mismo, estaba React Native, un *framework* creado por Facebook en base a la librería React.

Un punto en común de todas estas tecnologías de desarrollo híbrido fue que se pueden escribir con JavaScript, el lenguaje de programación con el que la mayoría del grupo de programadores estaba familiarizado.

En este punto del tiempo, la aplicación web de Qempo estaba siendo desarrollada en Angular, un *framework* web, que por sus constantes cambios de versiones hizo que el equipo considerase un cambio de marco de trabajo. Es así que se decidió probar React Native y aprovechar el desarrollo como una prueba segura y de reconocimiento que más adelante reforzaría la migración de Angular a React en el frente web.

Por otro lado, los objetivos y requerimientos de este nuevo proyecto difieren de su predecesor al acercarse más a la mencionada aplicación web que se trabajaba en paralelo a las aplicaciones móviles.

El objetivo principal de la aplicación era poder ofrecer a los usuarios una experiencia complementaria a la que obtenían de la parte web. Las funcionalidades debían existir en la aplicación web, pero adaptadas a la experiencia móvil. Con esto en mente, los requerimientos funcionales y no funcionales fueron los mismos que la versión Android salvo algunas diferencias que se mencionan en la Tabla 1.5.

Tabla 1.5

Requerimientos de primera version de aplicación React Native

Requerimientos funcionales	Requerimientos no funcionales
• Usar herramienta para extraer información de un producto a través de su URL. (*Scrapper*) • Restablecer contraseña con validación de código único.	• Recibir metadata de la base de datos. Información de *couriers*, categorías de productos.

Las librerías o dependencias usadas para el proyecto fueron las siguientes:

- immutability-helper: Mutar la copia de una data sin cambiar la fuente original. Es decir, prevenir los efectos secundarios de Javascript.
- moment y react-moment: Analizar, validar, manipular y mostrar fechas y horas en Javascript.
- react-redux con redux-observable: Un contenedor de estados predecible para aplicaciones Javascript y un intermediario (*middleware*) basado en RxJS para Redux. Ambas librerías crean un árbol de estados para conocer el estado, valga la redundancia, de la aplicación en todo momento por todos sus componentes, así

mismo, componer y cancelar acciones asíncronas para crear efectos secundarios y más.

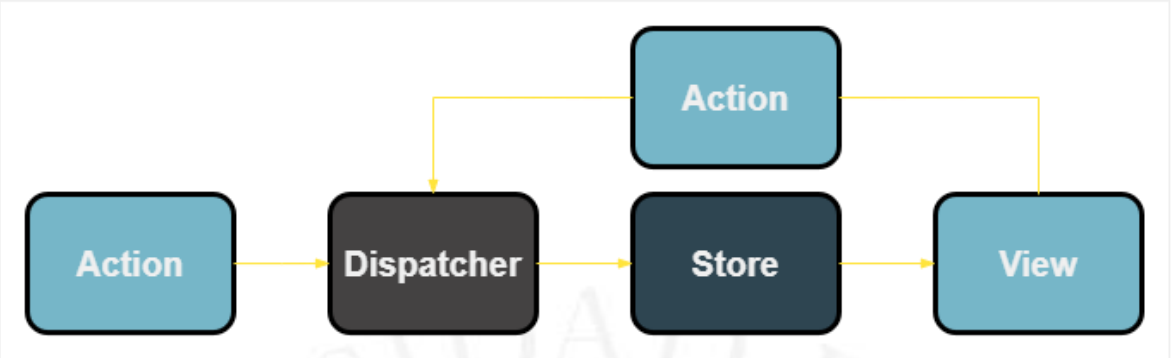
- flow: Librería de tipado estático para Javascript. Ayuda a comprobar los tipos de datos (tipificar) durante la compilación y no durante la ejecución.

La arquitectura que de la Figura 1.6 está basada en un flujo de datos unidireccional propuesta por Facebook llamada Flux, es más un patrón que un marco de trabajo formal (*In-Depth Overview | Flux*, 2022). Esta arquitectura de aplicación consta de 4 elementos y conceptos para soportar el flujo de datos.

1. *store*: Estructura de datos que almacena el estado de la aplicación o parte de la aplicación. Similar a un modelo de MVC (Model-View-Controller), en lugar de representar un único registro de datos representa el estado de muchos objetos. Actualiza la estructura de datos cuando recibe un *action* a través del *dispatcher.*
2. *action*: Un objeto Javascript con dos propiedades, el tipo y la carga (*payload*). El tipo sirve para que los *stores* interpreten apropiadamente el cambio a realizar mientras que el *payload* es lo que usará el *store* para actualizar los estados pertinentes. Son generados por los creadores de acciones o *action creators* en inglés, método qué transmite el *action* al *dispatcher*.
3. *view*: La interfaz de usuario que puede usar *actions creators* y se encarga, principalmente, de heredar el estado que obtiene de los *stores* a través de su jerarquía de vistas hijas.
4. *dispatcher*: Administra todo el flujo de datos de la aplicación, un mecanismo simple para distribuir los *actions* a los *stores*. Cada vez que un *action creator* proporciona al *dispatcher* una nueva acción, los *stores* lo reciben por estar registrados en este administrador.

Figura 1.6

Arquitectura Flux propuesta por Facebook



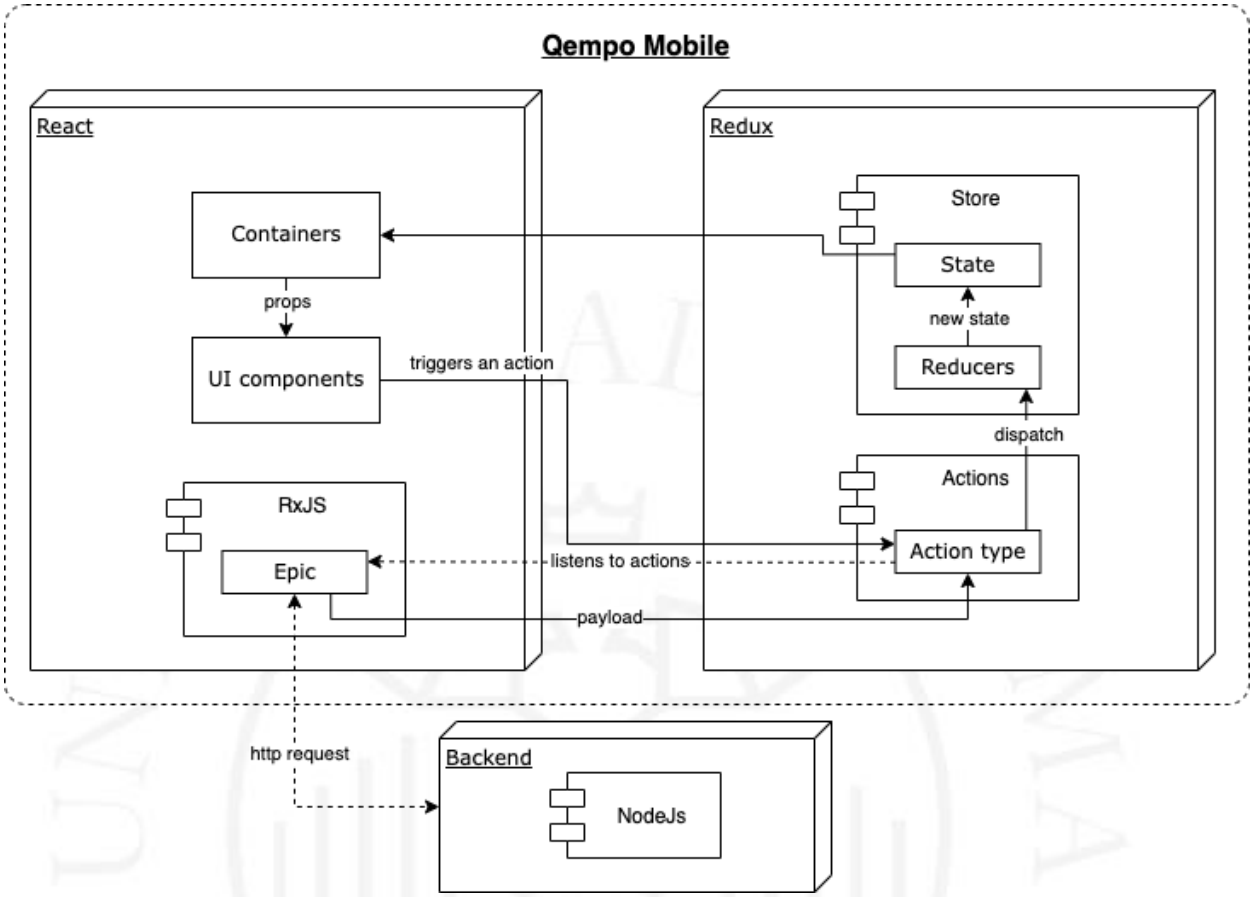
Nota: Flujo de datos en Flux con datos que se originan en las interacciones del usuario, 2016, https://facebook.github.io/flux/docs/in-depth-overview

Con este patrón unidireccional y la librería que react-redux, la estructura del proyecto tomó una forma basada en módulos o funcionalidades. Cada módulo tiene una estructura en común como en la Figura 1.7 y descrita a continuación.

- *components*: Archivos .jsx con el nombre del módulo y representa la vista principal la funcionalidad. La vista que recibe el estado de los *stores* y los *action creators* desde el *container*.
- *containers*: Archivos .js con el nombre del módulo y el sufijo "container" que nos ayudan a pasar el estado de nuestro *store* junto a los *action creators* que necesitamos e inyectarlos a nuestro componente a través de los *props* (propiedades).
- *sub-module*: Parte atómica de la funcionalidad que también tiene la misma estructura que un módulo. No es requerida, sirve para desacoplar y aligerar la carga en el módulo.
- api.js: Un archivo para listar y exportar todas las rutas usadas para los llamados al internet y también las funciones para hacer llamados http.
- duck.js: Un archivo con el fin de agrupar algunos elementos de Flux, Redux y efectos secundarios de RxJS. Sigue la propuesta ducks-modular-redux (https://github.com/erikras/ducks-modular-redux) para aislar estos elementos.

Figura 1.7

Diagrama de arquitectura de Qempo Mobile



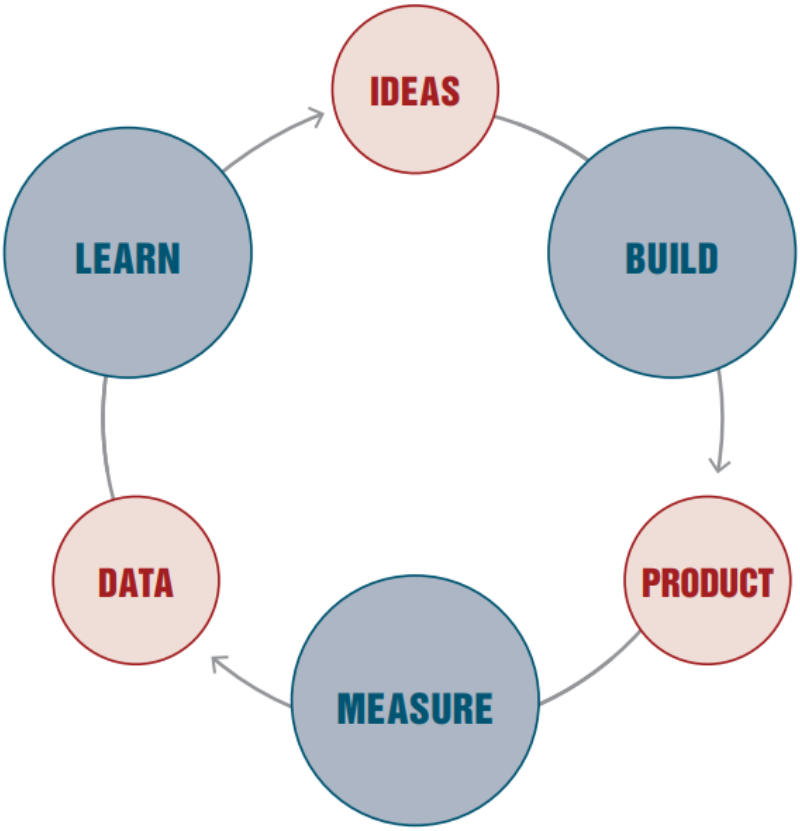
d) Qempo Webapp

El proyecto Qempo cuenta con múltiples iteraciones, desde su concepción antes del proyecto Wanna, hasta después de la versión híbrida en React Native.

En la segunda iteración del negocio, se optó el modelo de negocio *C2C* (*Consumer to Consumer* por sus siglas en inglés) y por una solución *SaaS* (*Software as a Service* por sus siglas en inglés). El primer cambio fue para poder crecer sin la necesidad de aumentar los OPEX, mientras que el segundo fue para tener mayor control sobre la plataforma de cara al cliente. Para estar seguros que la solución era lo que se buscaba, se optó por un desarrollo de software con metodología *Lean Start-up,* ver Figura 1.8. Se trata de reducir la frecuencia del ciclo de desarrollo de un producto o software y descubrir si un modelo de negocio es viable. Para esto, propone un ciclo de retroalimentación de ideas que son validadas a través de un producto y la validación de los usuarios al usarla; con esta data, aprender para generar nuevas ideas.

Figura 1.8

Ciclo de retroalimentación de la metodología Lean Start-up



Nota. Gráfico resumen de la metodología Lean Start-up. Adaptado de *The lean startup: How today's entrepreneurs use continuous innovation to create radically successful businesses* (p.73), por Eric Ries, 2011, Crown Business

Las tecnologías a usar para esta iteración fueron las siguientes:

- Node.js para el backend porque funciona en un hilo de procesamiento y consume pocos recursos.
- Plantillas EJS con jQuery para el frontend porque se quería usar el mismo proyecto del backend que podía servir dichas plantillas.
- MongoDB para una base de datos NoSQL y ejecutar migraciones rápidas.
- PM2 para ejecutar el backend y el paquete *mongodb-org* disponible en el repositorio *yum* MongoDB para manejar la base de datos.

En esta nueva iteración se logró validar el modelo y la solución; sin embargo, la experiencia de usuario necesitaba mejorar mucho porque los usuarios no entendían con

claridad el proceso de importación a través de la web. Adicional a esto, el alto acoplamiento del frontend y el backend por usar las plantillas EJS elevaba la dificultad en el desarrollo de nuevas funcionalidades o la mejora de alguna existente.

- **Versión 1.0**

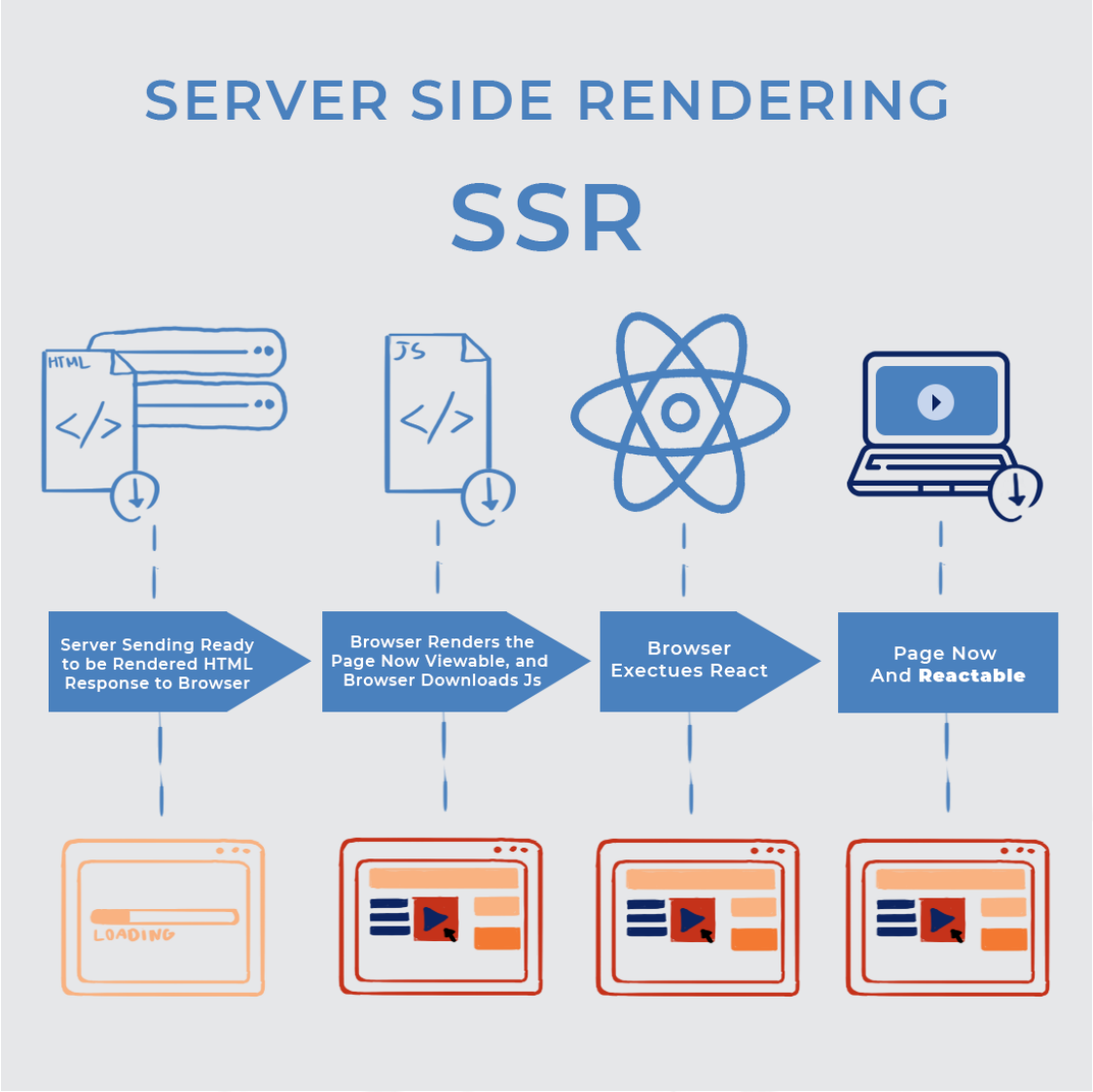
Con esta información, el primer objetivo de la siguiente iteración fue mejorar el diseño de la plataforma con un nuevo tema y componentes más dinámicos; el segundo, reducir el tiempo de los ciclos de aprendizaje de la metodología usada y así entregar funcionalidades con más frecuencia. Los requerimientos funcionales con respecto a la versión anterior fueron los mismos bajo la salvedad de que tenían que realizarse con los nuevos diseños. Por el lado de los requerimientos no funcionales, fue utilizar un marco de trabajo completo para el frontend y desacoplar el frontend a través de un proyecto propio. Dicho proyecto se realizó con Angular versión 2 (https://angular.io/).

El resultado fue la primera versión estable del sitio web con la capa de presentación, o frontend, y la capa de datos, o backend, desacopladas. Lo que nos llevó a una nueva iteración de Qempo donde se enfocó en reducir el tiempo inicial de carga de la página web, mejor posicionamiento en los motores de búsqueda y obtener más usuarios.

Para los dos primeros objetivos se utilizó un método llamado SSR (Server Side Rendering por sus siglas en inglés), que, según la infografía de la Figura 1.9 se refiere a la capacidad de una aplicación para mostrar la página web en el servidor en lugar de mostrarla en el navegador. Cuando el Javascript de la página web se procesa en el servidor web, se envía una página completamente procesada al cliente. Además, los motores de búsqueda pueden rastrear e indexar el contenido antes de entregar la página al cliente y tener un mejor posicionamiento (*What Is Server-Side Rendering? Definition and FAQs*, s.f.).

Figura 1.9

Infografía de SSR, server-side rendering



Nota. Explicación gráfica de cómo funciona el método SSR, server-side rendering. Adaptado de *SSR vs CSR- which is the right choice for your Progressive Web App (PWA)?*, por Claudia Söhlemann, (https://kruschecompany.com/ssr-or-csr-for-progressive-web-app/)

La librería Angular Universal (https://angular.io/guide/universal) sirvió para procesar la aplicación en el lado del servidor web hecho con la librería Koa (https://koajs.com/). En esta iteración había un alto acoplamiento entre el manejo del estado y el manejo de la vista, lo que impedía una correcta implementación del SSR. La solución fue usar la arquitectura Flux mencionada en el proyecto Qempo Mobile a través de la librería @ngrx/store. Cabe aclarar que este punto del desarrollo de la página web es anterior al inicio de la aplicación híbrida.

La siguiente iteración se basó en integración continua, una práctica de desarrollo de software en la que los miembros de un equipo integran su trabajo con frecuencia y cada integración se verifica mediante una compilación automatizada para detectar errores de integración lo más rápido posible (Fowler, 2006), porque había problemas de rendimiento en el servidor de producción cada vez que había un nuevo pase a producción debido a que en el mismo ambiente se ejecutaba la compilación del código. El desacoplamiento del servidor de producción y la compilación automática del código requería de un repositorio de artefactos, almacenamiento de artefactos de compilación creados mediante integración continua, dichos artefactos son los archivos creados por el proceso de compilación, tales como paquetes de distribución, archivos, registros e informes (*¿Qué Es Un Repositorio De Artefactos? | Guía De CI/CD De TeamCity*, s.f.). Dicho repositorio fue configurado en un servidor usando Verdaccio, un repositorio de código abierto de npm donde se pueda almacenar, versionar y distribuir de manera automática los artefactos.

El encargado de generar el artefacto fue Bitbucket, es una herramienta de colaboración y alojamiento de código basada en Git, gracias a su servicio integrado de CI/CD llamado *pipeline*. Este le permite compilar, probar e incluso implementar automáticamente su código en función de un archivo de configuración en su repositorio. Una vez generado el artefacto se sube automáticamente a Verdaccio. Es así como la integración continua mejoró los pases a producción y el tiempo de despliegue entre nuevas versiones.

- **Versión 2.0**

Es aquí donde empecé a trabajar en el proyecto Qempo Webapp, en la nueva versión del proyecto frontend con el marco de trabajo React ya que tenía experiencia con React Native por Qempo Mobile. Es de notar que la versión anterior, como se mencionó en la sección anterior, estuvo hecha con Angular y algunas de las razones para el cambio de marco de trabajo fueron las siguientes:

  - Solucionar problemas o errores concernientes a Angular era laborioso por falta de documentación en la comunidad que lo usaba.
  - Las librerías disponibles para Angular no solían tener un buen nivel de madurez, es decir, eran propensos a vulnerabilidades y errores.
  - React era más estable, la cantidad de *breaking changes* (cambios en el software que puedan generar fallos en otros componentes del mismo) era

menor y la comunidad ya era lo suficientemente grande como para tener una documentación sustancial sobre diversos problemas y soluciones.

- Existían librerías que nos permitieron continuar usando la arquitectura Flux con la librería Redux.js y el método SSR.

Luego de que la migración se llevase a cabo con éxito, el tiempo de desarrollo de nuevas funcionalidades y solución de errores disminuyó como se esperaba.

Qempo como negocio ganó un concurso de startups convocado por "The Venture City", una empresa de inversionistas que brinda asesorías a startups. Gracias a estas asesorías se pudo constatar que nuestra plataforma no registraba cómo el usuario interactuaba con ella y en consecuencia tomar decisiones de negocio y sobre la plataforma subjetivamente. Es así que el objetivo de la siguiente iteración fue el de cuantificar y calificar el comportamiento del usuario en la plataforma.

En las asesorías nos dieron a conocer varias herramientas de visualización de datos en base a eventos en la plataforma. Nuestra ya web usaba múltiples plataformas que capturaban dichos eventos como Google Analytics, Amplitude y Facebook Pixel. Sin embargo, el control de estas era limitado porque no contábamos con una herramienta que se encargue de enrutar eventos de distintos orígenes, como el click de un botón en el frontend o la creación de una orden en el backend, hacía las plataformas mencionadas. Segment.io (https://segment.com/) fue la solución para dirigir las más de veinte fuentes de datos y entender mejor cómo es que nuestra web era usada.

2. CAPACIDAD DE GESTIÓN

Cuando empecé a trabajar en la empresa Curiosity Startup, era el único desarrollador frontend para los tres primeros proyectos mientras que el backend era codificado por otros desarrolladores en la empresa. Debido a que estos proyectos fueron pruebas de concepto o no se llegaron a lanzar al mercado de usuarios, el backend era principalmente un servicio API. Es así que gran parte de la gestión del desarrollo de las aplicaciones estaba a mi cargo. Mis tareas eran muy variadas, pero sin un marco de trabajo específico:

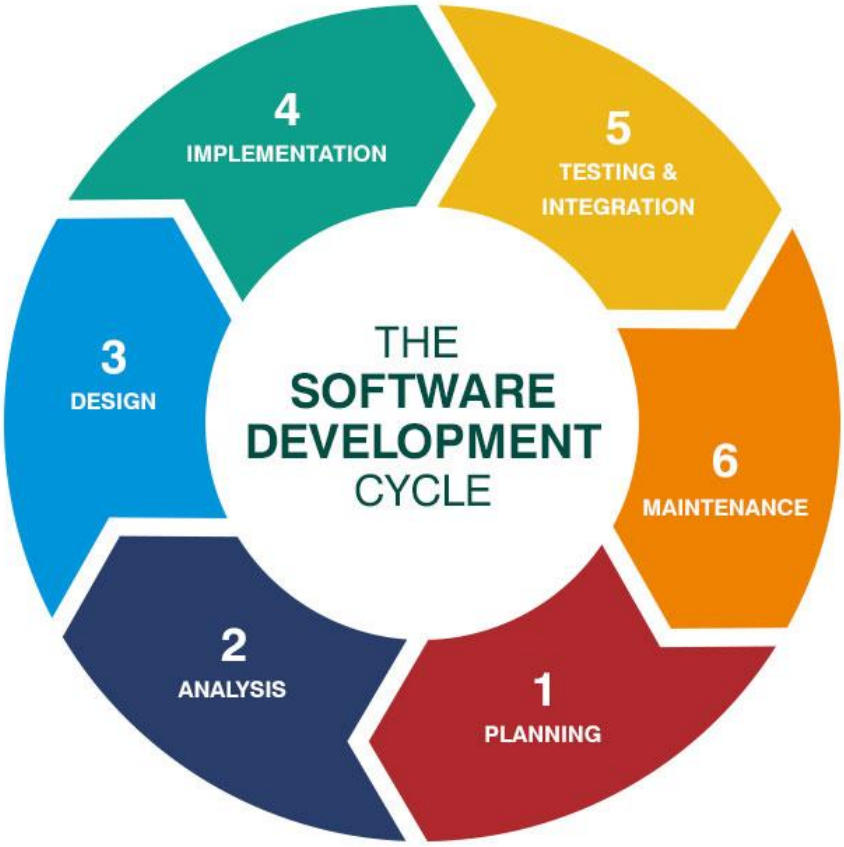
- Levantar requerimientos funcionales y no funcionales.
- Diseñar la interfaz de usuario, así como la experiencia de usuario.
- Crear la estructura y arquitectura del proyecto.
- Codificar, probar, compilar y mejorar la aplicación.
- Deliberar y consensuar con backend cómo consumen las API.

a) Software Development Life Cycle

Por ser nuevo en el desarrollo de software y el negocio una microempresa, gran parte de estas tareas las llevaba a cabo a través de preguntas directas o reuniones breves con el stakeholder (mi jefe directo) o con los otros desarrolladores que me daban consejos y/o retroalimentación en base a sus experiencias. Este ciclo de planificación, desarrollo y preguntas era una forma bastante básica del ciclo de vida del desarrollo de software, también conocido como SDLC por sus siglas en inglés y graficado en la Figura 2.1. Este ciclo es una metodología con base o marco de trabajo para estructurar, planificar, y ejecutar tareas relacionas con el desarrollo de un sistema de información. (*The softwar The software development lif elopment life cycle and its application*, 2018).

Figura 2.1

Las seis etapas más comunes de un SDLC



Nota. Representación gráfica de las seis etapas más comunes en un típico ciclo de vida de desarrollo de software. Adaptado de *Software Development Life Cycle (SDLC)*, por Big Water Consulting, 2019, (https://bigwater.consulting/2019/04/08/software-development-life-cycle-sdlc/)

Los tres primeros pasos se podían traducir en las reuniones donde se exponía la idea del negocio, los que necesitaba hacer la app y una idea básica de cómo debía verse. A veces se hacían bosquejos de las vistas, otras veces quedaba a mi criterio. Esto me ayudó en gran parte a desarrollar un sentido de *ownership*, una cultura o ambiente donde los trabajadores toman iniciativa, resuelven problemas y demuestran liderazgo (*Ownership Culture*, s.f.). Era aquí que decidía qué vistas o funcionalidades debían ser construidas primero y qué librerías usar por lo que ofrecía sin mucha configuración o porque era sencilla y no añadía código innecesario al proyecto.

El cuarto y quinto paso era donde pasaba la mayor parte del tiempo, no solo porque era codificar sino porque probar aquello que programaba tomaba un tiempo no despreciable. Las pruebas de usabilidad eran manuales a pesar de tener pruebas unitarias ya que estas solo cubrían cierta parte de la aplicación. Además de eso, por tener el requerimiento de que la

aplicación pueda ser usada por dispositivos antiguos, algunas librerías presentaban errores de compatibilidad o se comportaban de manera distinta entre las versiones más actuales.

El último paso, mantenimiento, suele también ser dividido en dos: despliegue y mantenimiento. La primera parte, el despliegue, en estos proyectos solía ser la instalación de la aplicación en algún o varios dispositivos móviles para mostrar el avance a mi jefe directo; la siguiente, el mantenimiento, era arreglar algún bug inesperado o modificar la experiencia de usuario por recomendación de él mismo o sugerencia mía. Es así que cuando se iba a presentar la PoC la aplicación era subida al PlayStore como parte del despliegue. Cabe resaltar que la limitación de hacerlo para el ecosistema Android se debía al presupuesto que tenía la empresa ya que se necesitaba comprar una licencia de Apple para desarrollar y subir la aplicación al App Store.

Parte de la cartera de negocios de Curiosity Startup incluía, además de los proyectos donde inicié, la página web Qempo. Durante su iteración con el marco de trabajo en Angular se tuvo la necesidad de mantener un control sobre el rendimiento del equipo de trabajo. La solución a escoger tendría que implementar métricas con las tareas y sus estados en el tiempo. Así mismo, todo el equipo de desarrollo estaba familiarizado con metodologías de desarrollo de software. En consecuencia, se tomó la decisión unánime de usar *Agile*. El desarrollo ágil de software es una metodología o filosofía, una forma de pensar que ayuda a organizaciones y equipos a innovar, reducir riesgos y responder a cambios de requerimientos. Agile se adaptó bastante bien a nuestro flujo de trabajo porque se enfoca en el cambio rápido de las necesidades del negocio a través del desarrollo de software incremental y la iteración continua de desarrollo en el SDLC. Bajo esta filosofía una tarea podía ser una funcionalidad, una actividad o un error.

Como se ha mencionado, el marco de trabajo elegido empezó a ser un problema para la continuidad de desarrollo de nuevas funcionalidades, así como el mantenimiento de la página web. Así mismo, la naturaleza del proyecto, ser una startup, no permitió que se tuvieran buenas métricas ni trazabilidad de la productividad con respecto a Angular. Sin embargo, a partir de la lista de razones para cambiar de *framework* mencionada en el capítulo anterior, sección D, se pudieron obtener algunas mediciones.

- Aproximadamente tres horas de labor a la semana para solucionar errores.
- Entre dos a tres clientes reportaban no poder interactuar correctamente con la página web la semana del lanzamiento a producción de una nueva versión.

- Buscar una librería que funcione correctamente con la versión de Angular y que tenga una comunidad mínima de usuarios tomaba entre cinco a seis horas por actualización mayor.

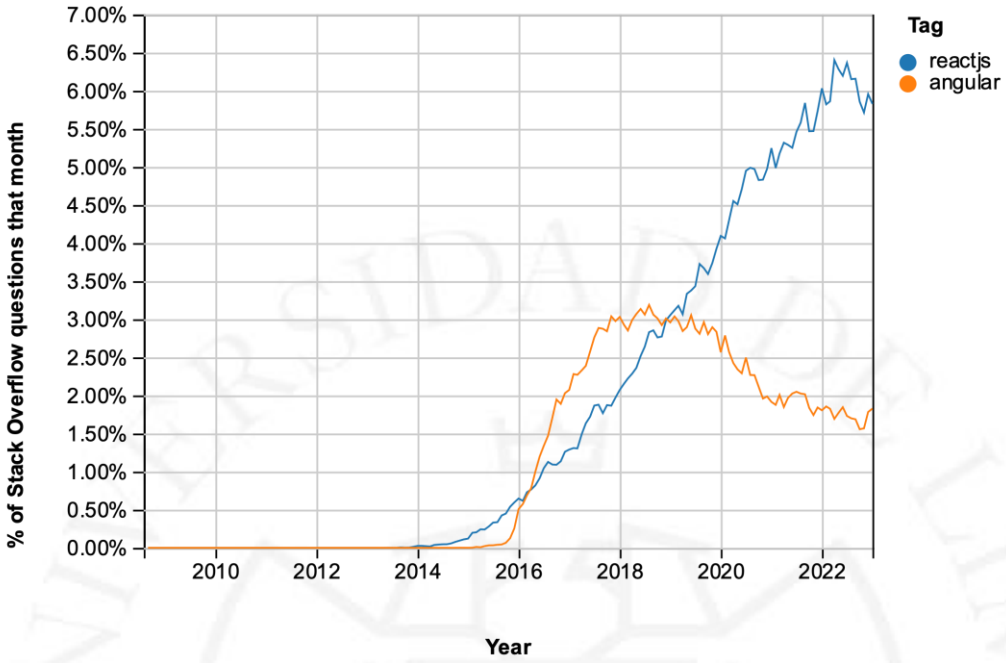
El primer punto ocurría durante un mes luego de cada actualización mayor, tanto como para la actualización a la versión 4 como a la versión 5. En el mismo periodo de tiempo, las librerías que ya usaban el software a veces eran obsoletas, por lo tanto, el equipo tenía que buscar una nueva. Mientras tanto, el segundo punto ocurría la primera semana y se solucionaba lo más rápido posible para impedir el flujo del negocio.

En otras palabras, los constantes cambios entre versiones, la actualización de librerías y la solución de errores en las mismas ralentizaban la velocidad del equipo de frontend y backend en al menos 3 días de recursos por actualización.

Adicionalmente, otra razón por la que se empezó a considerar React como el marco de trabajo sustituto fue el crecimiento de la comunidad en Stack Overflow. Como se observa en la Figura 2.2, las preguntas relacionadas a React crecieron de un 2% a un 3% mientras que las preguntas con la etiqueta de Angular se mantuvieron en un constante 3%.

Figura 2.2

Tendencias de Stack Overflow entre React y Angular



Por esto, el líder técnico tomó la decisión de migrar a otro marco de trabajo con mejor estabilidad y mayor madurez, que no tenga cambio de versiones tan aceleradas y/o produzcan tanta incompatibilidad con versiones anteriores.

b) Kanban

La primera herramienta que decidimos usar para aplicar la metodología ágil fue Kanban, también conocido como tablero Kanban. Es una herramienta ágil de gestión de proyectos diseñada para ayudar a visualizar el trabajo, limitar el trabajo en proceso (WIP, *Work In Progress*, por sus siglas en inglés) y maximizar la eficiencia del equipo (Rehkopf, s.f.). Un tablero puede ser dividido en cinco elementos clave como se muestra en la Figura 2.3:

- Señales visuales: Post-it's, tarjetas con el ítem a trabajar.
- Columnas: Cada columna representa una actividad y juntas generan el flujo de trabajo.
- Límite de WIP: Cantidad máxima de cartas o tarjetas por columna.
- Punto de compromiso: Momento en el que una idea empieza el flujo de trabajo.
- Punto de entrega: Momento en el que una idea termina el flujo de trabajo.

Con estos elementos definidos, se establecieron las siguientes métricas:

- Porcentaje de cumplimiento de las tareas.
- Velocidad de desarrollo del equipo.
- Número de funcionalidades en proceso de desarrollo.
- Número de funcionalidades en proceso de ser probadas.

Sobre las métricas de funcionalidades, conocer cuántas estaban en proceso de desarrollo permitió al equipo conocer mejor la eficiencia del tiempo invertido en desarrollar software, por ejemplo, si es que existe una buena distribución de tareas con respecto a las horas trabajadas o no. El rango de tareas semanales fue de seis a diez actividades, la amplitud se debió a la complejidad de la tarea a realizar. Luego, conocer la cantidad de funcionalidades a ser probadas reveló que se tenían funcionalidades estancadas o atrasadas a ser pasadas a producción. Eran entre una a dos tareas por semana que los *stakeholders* y desarrolladores probaban, un número bajo en comparación a las tareas en desarrollo.

La herramienta de donde levantamos la data para las mediciones vino por recomendación de nuestro jefe, así que empezamos a usar Trello, una aplicación web estilo Kanban subsidiaria de Atlassian. El elemento columna fue lo más característico del uso que le dimos a la herramienta porque pudimos tener claridad del estado en el que estaba alguna tarea o trabajo. Las columnas de la Figura 2.3 son las recomendadas por Atlassian, no obstante, las que teníamos eran las siguientes:

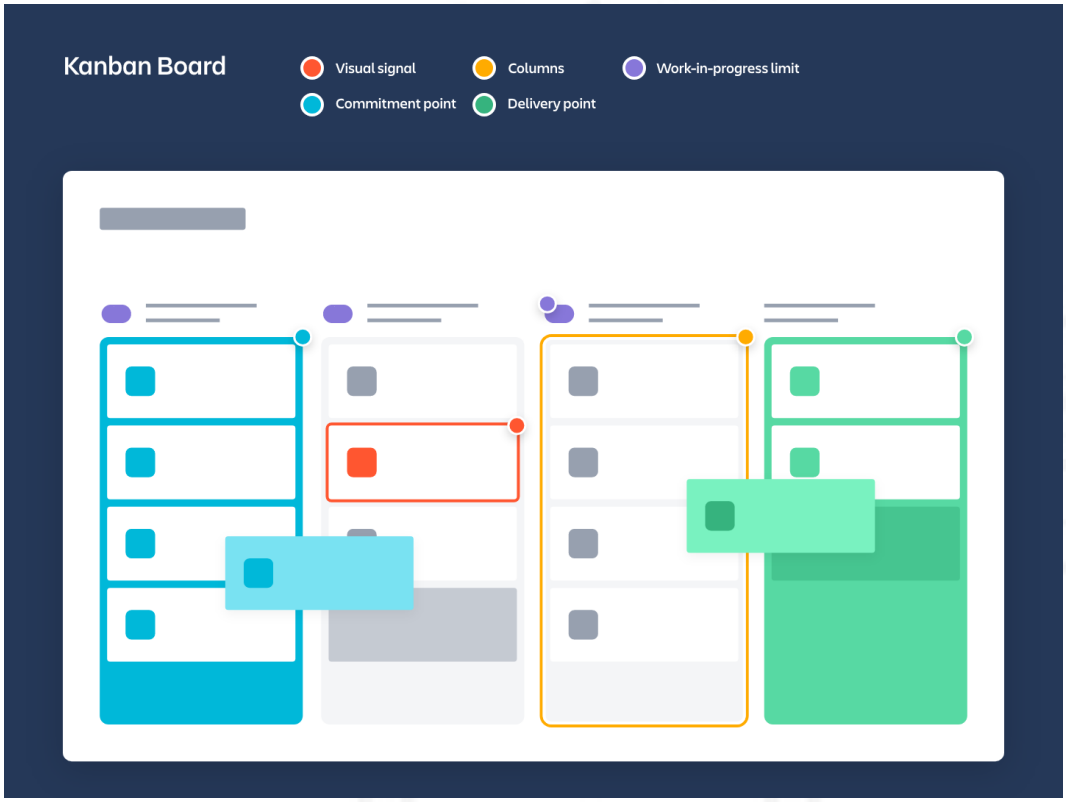
- To-do: Las tareas próximas a trabajar, las que tenían la prioridad suficiente para entrar al flujo.
- Frontend: Las tareas pertenecientes al frontend que estaban en desarrollo.
- Backend: Las tareas pertenecientes al backend que estaban en desarrollo.
- Testing: Las tareas de frontend o backend que terminaron de desarrollarse y que debían ser probadas.
- Done: Las tareas que se terminaron de desarrollar completamente y estaban listas para pasar a producción.
- Backlog: Todas las ideas o tareas esperando alcanzar la prioridad suficiente para entrar al flujo. Esta columna solía estar en el tablero, pero por ser muy extensa se decidió mover a otro tablero.

Así mismo, aumentó nuestra comunicación ya que las tarjetas, nuestras señales visuales, tenían la capacidad de agregar comentarios y con ellos archivos relevantes para el

contexto de la tarea. A pesar de no tener un límite por columna, nos limitamos a tener dos items por persona. Esta limitación se creó en la práctica ya que a veces teníamos tareas que necesitaban de otras para poder seguir su desarrollo. De forma similar, el momento en el que una tarea entraba o salía del flujo no tenía parámetros o restricciones salvo la urgencia de necesitarla en la página web.

Figura 2.3

Elementos de un tablero Kanban



Nota. Los 5 elementos principales de un tablero Kanban definido por David Anderson. Adaptado de *What is a kanban board?* por Max Rehkopf. (https://www.atlassian.com/agile/kanban/boards)

Con la implementación de Kanban a la migración de Angular a React, en un plazo de 7 sprints (ciclo o iteración) de una semana pudimos observar que el equipo de backend lograba realizar entre 5 a 6 tareas por en un espacio de 1 semana mientras que el equipo de frontend entre 4 a 5 tareas. Además, el tiempo en que una tarea era aprobada para pasar a producción disminuyó de 3 días a menos de 2 días ya que el *stakeholder* pasó de preguntar o esperar cuándo podría revisar una tarjeta a observar el tablero primero y pregunta si aún tenía dudas. Por otro lado, cada desarrollador empezó a tener más claridad sobre el estado del trabajo del resto del equipo porque con el pasar de las semanas pudimos observar un aumento de comentarios e información extra en las tarjetas de 2 a 3 interacciones en promedio por tarjeta, siendo esta interacción el reemplazo de una sincronización directa.

Otra consecuencia de la migración fue la mejora en la velocidad del equipo como se muestra en la Tabla 2.1:

Tabla 2.1

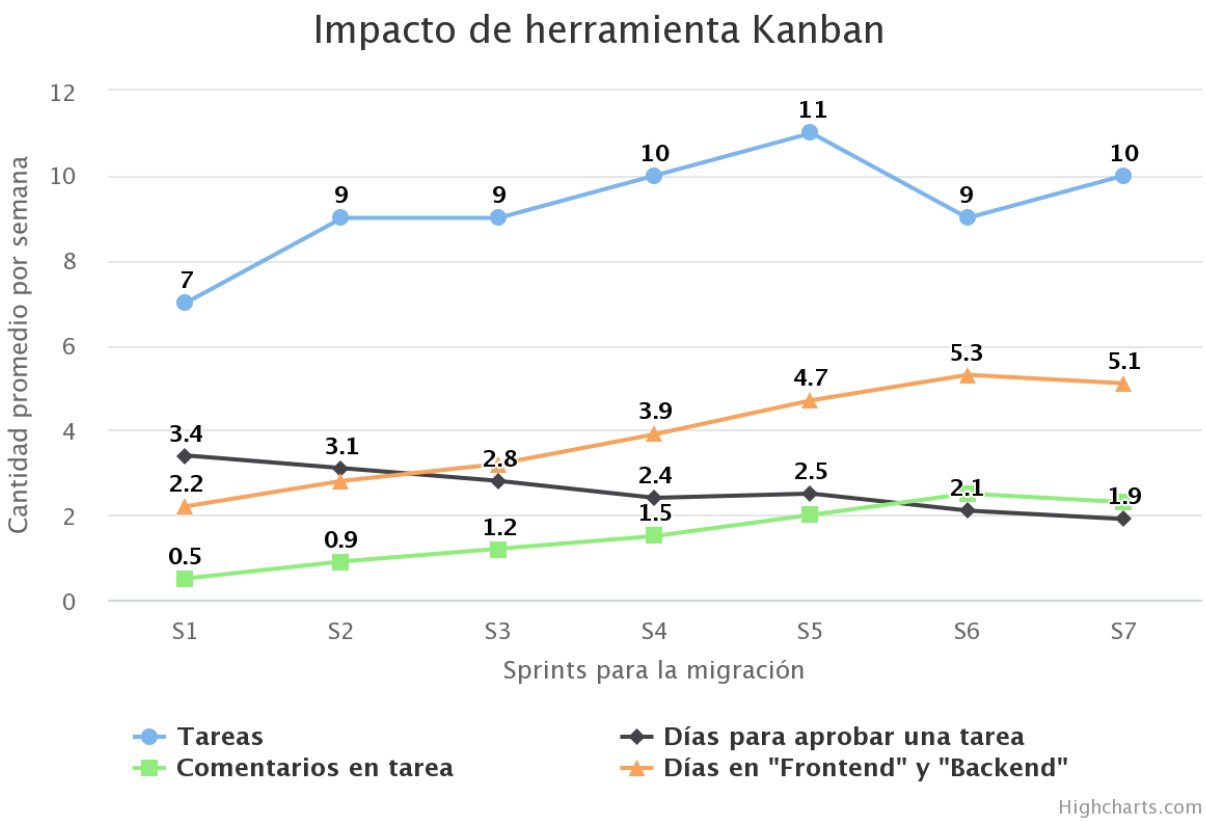
Cantidad de tareas mensuales por dificultad y marco de trabajo.

Tareas mensuales	**Angular**	**React**
Difícil dificultad	1	3
Media dificultad	9	16
Baja dificultad	17	25

No solo casi duplicó la cantidad de tareas en dificultad media y baja, sino que triplicó la cantidad de tareas difíciles que el equipo entregaba en producción. Este incremento se debió a que ya no se perdía tiempo lidiando con errores del framework en sí y reduciendo considerablemente las actualizaciones que rompían la compatibilidad con la versión anterior.

Figura 2.4

Impacto de la herramienta Kanban en la migración de frontend

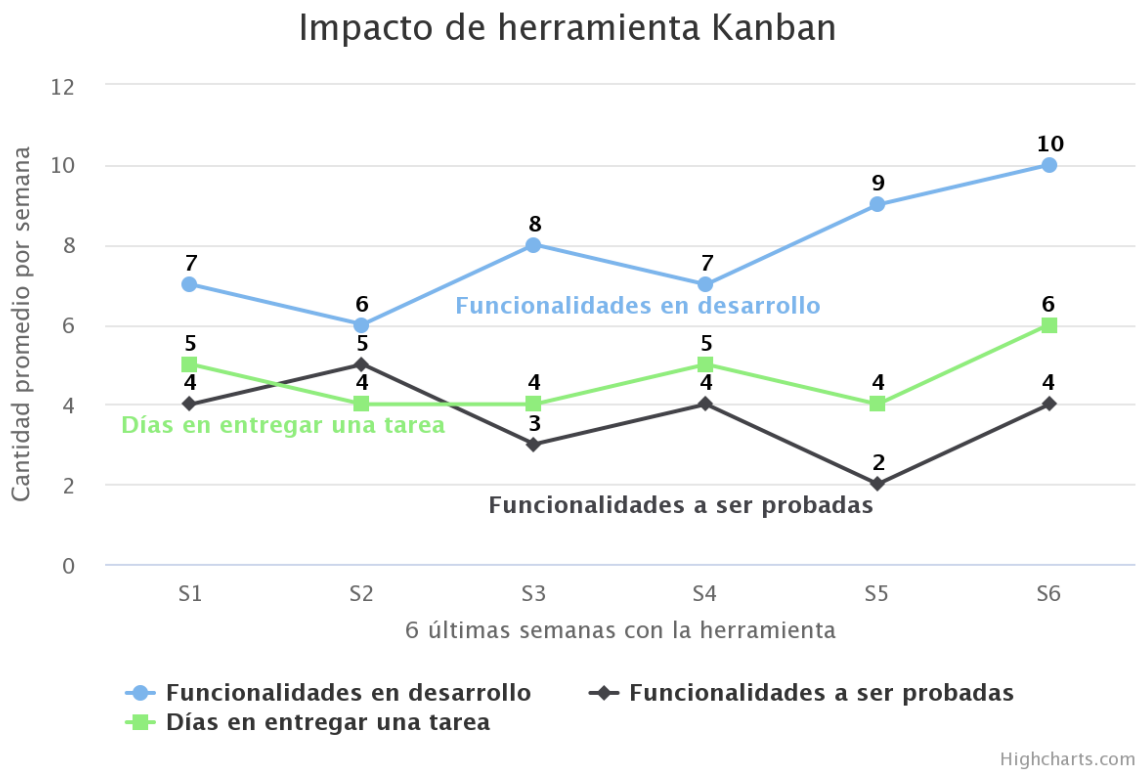


Como se puede ver en la Figura 2.4, la herramienta se continuó usando por un periodo de 6 meses más. Para este punto el equipo y los *stakeholders* notaron ciertos problemas y limitaciones que teníamos. Entre estos, estaba que una tarjeta podía estar en una columna por mucho más tiempo del que normalmente estaría. Podía pasar cinco a más días porque en medio del desarrollo el programador se percataba de que necesitaba alguna funcionalidad o información previa. Por ejemplo, cuando se empezó a realizar la tarea de estimación de costos de envíos no se había tomado en cuenta que se requería saber si el maquillaje estaba grabado de impuesto, así como otros datos necesarios para el cálculo. Así ocurrió para varios artículos que necesitábamos y debíamos tener en nuestro sistema, y por lo tanto la tarea tomaba mucho más tiempo de lo que se planeó o de lo que se estimó originalmente. Estos nuevos requerimientos se convertían en una tarea más a pasar por el flujo de desarrollo antes que la tarea ya iniciada y dejada en espera.

Para poder reflejar mejor el efecto de Kanban en el equipo de programadores se muestra a continuación un periódo de las 6 últimas semanas de uso de la herramienta ya con el nuevo marco de trabajo React implementado.

Figura 2.5

Impacto de la herramienta Kanban en sus últimas 6 semanas de uso



En la Figura 2.5 se puede observar cómo es que el número de días en entregar una tarea varía con respecto a las funcionalidades probadas por los *stakeholders* y en proceso de desarrollo. Por ejemplo, en la semana 3 solo se pudieron revisar 3 funcionalidades, al mismo tiempo se aumentó el desarrollo con 2 tareas más, así que en la siguiente semana aumentaron los días en entregar una tarea. Ya en este tramo final del uso de la herramienta, las tareas nuevas que se tenían que realizar como parte de una funcionalidad existente eran tomadas como una misma tarea para evitar la constante creación de nuevas tareas. Sin embargo, a veces el objetivo de una funcionalidad podía ser cortado y terminado. mientras que la parte y/u objetivo removido era pasado al backlog como tarea y/o funcionalidad a futuro.

El incremento de tiempo que pasaban las tareas en el flujo también aumentaba por uno de los elementos de Kanban: el límite de WIP. Lo que se traducía en que muchas ideas de cómo mejorar el negocio quedaban estancadas en la columna *backlog*, problema mencionado al describir dicha columna y más notorio al terminar de usar la herramienta ágil. Según la metodología *Lean Start-up*, debíamos reducir la frecuencia con la que estas tareas salían a producción y así descubrir si la idea es exitosa.

Luego de 6 meses de usar Kanban, con el *backlog* convertido en un nuevo tablero por su larga lista y la versión 2.0 del backend cerca de ser lanzada, el equipo decidió intentar trabajar con otra herramienta ágil. A pesar de que Kanban nos dio un orden que aumentó la eficiencia de nuestro tiempo también ralentizó el proceso de inicio a fin de una tarea. En contraste, Scrum, un marco de desarrollo ágil heurístico, ponía más enfoque en probar e iterar con regularidad. Además, era una herramienta ágil con la que el equipo ya estaba familiarizado. Por lo tanto, se decidió cambiar a Scrum.

c) Scrum

Scrum se basa en el aprendizaje continuo y la optimización ante elementos cambiantes. Admite que el equipo desconoce el todo al inicio de un proyecto y evolucionará a través de la experiencia (Drumond, s.f.). Con una etapa de priorización en el proceso y ciclos de lanzamiento relativamente cortos, la estructura de Scrum ayudó al equipo a adaptarse a los constantes cambios de requerimiento que podían venir por parte de los *stakeholders* o los usuarios de la web.

De los artefactos o herramientas que Scrum provee, el *product backlog* (lista de producto) fue lo que mejor se acopló a nuestro estilo de desarrollo proveniente de Kanban. Dejamos de tener un tablero específico para nuestro *backlog* para empezar a utilizar esta lista mantenida por el *stakeholder* y los miembros más *senior* (experimentados) entre los desarrolladores, todos fungiendo los roles de *product owner* (dueño del producto) y/o de *product manager* (gerente del producto).

El *sprint backlog* (lista de iteración) presentó al equipo un problema: errores emergentes durante el *sprint*. Debido a la falta de experiencia con la herramienta, fue difícil entender que esta podía ser flexible y en consecuencia añadir o cambiar tareas en la misma iteración. Luego de usar más la herramienta, se comprendió que la meta o el objetivo del *sprint* no debía ser afectado por estos cambios, es decir, el *sprint goal* no podía ser comprometido.

Parte de Scrum son los eventos o ceremonias que el equipo realiza con cierta frecuencia. Nuestro equipo empezó con las ceremonias más representativas:

- **Organizar el backlog**: El *product owner* se encarga de esta tarea, mantener la lista actualizada para que esté siempre lista para el trabajo. Pero en la práctica la

ceremonia fue la misma que en Kanban, el *stakeholder* y los miembros más *senior* suplían este evento en lo mejor de sus habilidades.

- **Planificación de Sprint**: Una reunión de todo el equipo de desarrollo para planificar el trabajo en la siguiente iteración. Ceremonia que terminó sin agregar valor y pasó a ser un evento igual al primero mencionado.
- ***Sprint***: Un intervalo de tiempo, típicamente dos, donde el equipo trabaja por un entregable o presentable. El equipo de Qempo iteró entre dos semanas y una semana hasta decidir que dos semanas era el mejor tiempo para terminar tareas.
- ***Daily meeting* o *stand up***: La ceremonia más productiva que pudo mantener el equipo a través de los sprints. Una reunión diaria a la misma hora para alinear al equipo a la meta del sprint con las tres preguntas recomendadas: ¿qué hice ayer?, ¿qué haré hoy? y ¿tengo impedimentos? Llegamos, con el pasar de los meses y el crecimiento de la empresa a ser aproximadamente diez personas y con una reunión agotadora y más de 20 minutos.
- **Revisión del sprint**: Una reunión más con el objetivo de mostrar lo trabajado en el ciclo; normalmente para aceptar o no dicho trabajo final. Aunque usada al inicio, rápidamente fue una tarea más de planificación y organización.
- **Retrospectiva**: Un tiempo donde el equipo se vuelve a reunir para comentar sobre lo que fue bueno y lo que fue malo a fin de continuar con lo bueno y evitar lo malo. Una mejora general para el ciclo de desarrollo.

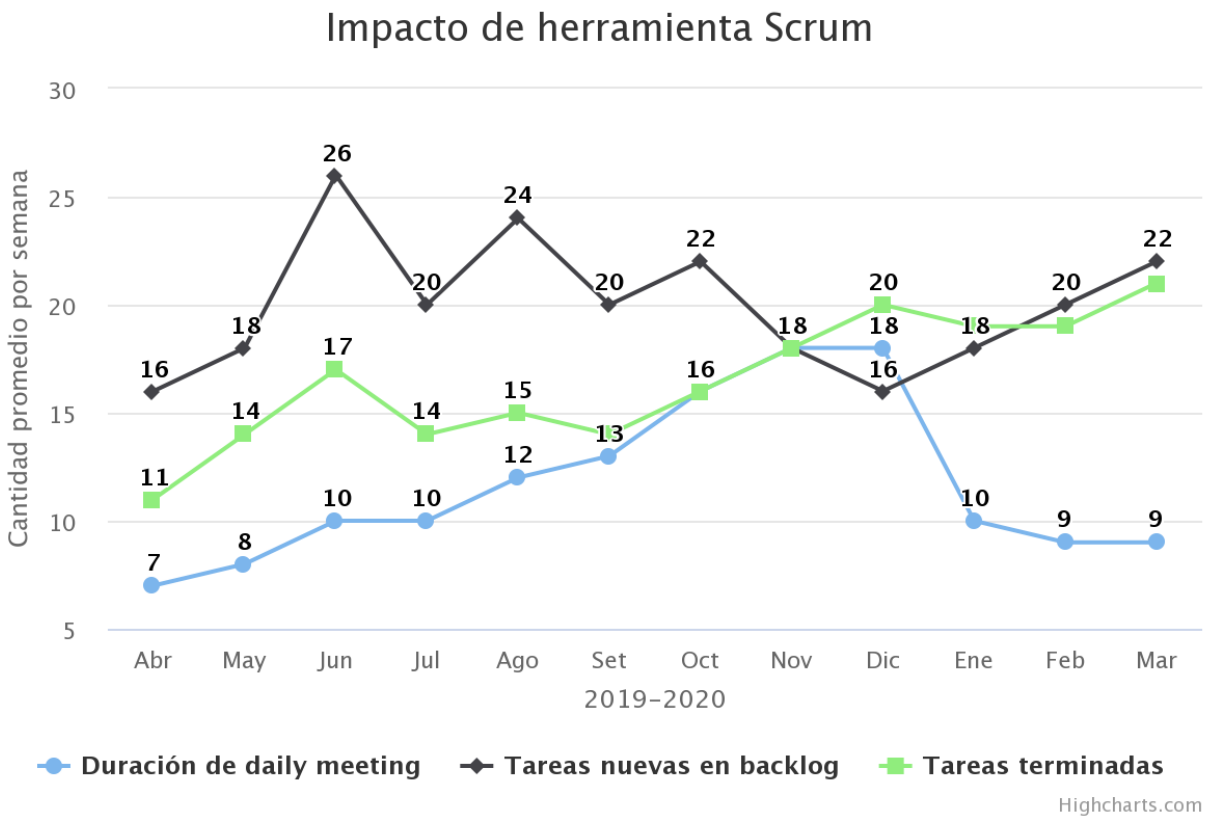
Para el equipo de desarrollo y demás áreas de la empresa quedó claro que el tiempo de un *daily meeting* era demasiado largo y tedioso por ser la mezcla de varios equipos, desde operaciones con el trato al cliente y entrega de pedidos, hasta las campañas en redes y estrategias de adquisición. Pasó de ser un evento de 7 minutos a uno de hasta 12 o 17 minutos por la cantidad de personas. Aunque esta ceremonia también fue eventualmente dejada atrás, se volvió a tomar por insistencia mía ya que quise implementar dichas ceremonias con un grupo más reducido y dar más responsabilidades al equipo como no depender del Scrum Master, persona encargada de facilitar la implementación de Scrum, para las ceremonias y que sea un rol compartido entre los desarrolladores.

Es así que un daily meeting volvió a durar entre 9 y 10 minutos, pero para un número más reducido, 6 personas. Todo esto mientras que los demás eventos cesaron por completo

o fueron absorbidos como tareas más para un desarrollador y un *stakeholder*, la misma persona con el que se inició a implementar la herramienta ágil Kanban.

Figura 2.6

Impacto de la herramienta Scrum a lo largo de un año



Como indica la Figura 2.6, a lo largo del uso Scrum la cantidad de tareas terminadas, es decir, las tareas que llegaban a pasar a producción, fue incrementando en relación a la cantidad de tareas que había en el *backlog*. Al inicio tuvimos un incremento de tareas nuevas en el backlog porque las tareas estaban llegando a la meta con mayor rapidez. Anteriormente, generar más tareas era innecesario porque solían estancarse en el backlog y priorizarse múltiples veces, después, con la planificación del *sprint* y la organización del *backlog*, las tareas nuevas se volvieron más atómicas y claras. Así que llenar el backlog era necesario para continuar probando e iterando ideas de negocio. Por último, la cantidad de tareas nuevas que se agregaron al *backlog* se mantuvo más constante en los siguientes meses luego de un ajuste en la velocidad del equipo.

Por otro lado, la duración del *daily meeting* vio su pico más alto durante los meses de Noviembre y Diciembre porque son los meses festivos y con grandes ofertas de productos

como Black Friday y Cyber Monday. Ambos eventos con grandes descuentos en tiendas norteamericanas a las que el público objetivo de Qempo quiere obtener. Al realizar estas reuniones estos meses con un equipo de operaciones bastante ocupado y un equipo de desarrollo, el daily se dividió para solo tener al equipo de desarrollo porque el equipo de operaciones no encontró valor en la ceremonia. Es por esto que los últimos meses de este ciclo anual fueron tan bajos.

Dos resultados importantes que se notaron fueron el porcentaje de tareas del *backlog* terminado y el efecto y frecuencia de errores en producción. La primera mitad del año trabajando con Scrum, el backlog terminado estaba debajo del 40%, luego se estabilizó en aproximadamente 50%. La excepción fue el mes de Diciembre por ser el mes posterior a las festividades mencionadas. El segundo impacto fue el control de errores, con una mayor velocidad de entrega y una reducción en el tamaño de las tareas, más errores llegaban a producción, sin embargo, eran errores de bajo o mediano impacto. En contraste, al inicio del uso de la herramienta las tareas por ser más grandes y si ocurría algún error en producción su impacto era más crítico para el negocio.

3. APRENDIZAJE CONTINUO

Empecé a trabajar en ITLAB o Laboratorio de Aprendizaje en Tecnologías de Información de la Universidad de Lima como parte de mis prácticas pre-profesionales. Allí aprendí a programar en Java para Android y configurar el directorio activo de Windows para el sistema operativo Windows Server. Además, también participé de actividades que el laboratorio realizaba en ferias o talleres de aprendizaje.

Entre mis responsabilidades estaban las siguientes:

- Aprender a desarrollar un aplicativo Android a través del lenguaje Java. El proyecto en cuestión fue crear una aplicación para la cafetería donde el estudiante pueda ver el menú de la semana.
- Dar charlas del uso básico de un ERP open-source (código abierto en inglés) a los estudiantes del colegio de Ingeniería. La herramienta era OpenBravo (https://www.openbravo.com/) y la charla, sobre el proceso de una orden de abastecimiento hasta su completa entrega.
- Configurar y desplegar un Active Directory en el nuevo sistema de Windows Server del laboratorio. Hubo un cambio de ambiente de laboratorio y se decidió aprovechar el momento para renovar el sistema del servidor.
- Mantener canales de comunicación actualizados. Canales de la universidad a los estudiantes como el menú de la cafetería en un sitio web, y canales del laboratorio a la comunidad universitaria como la página de Facebook.

Aprender Java para programar aplicaciones Android fue la experiencia que dio inicio a la carrera de programador que seguiría más adelante. Era el único trabajador del laboratorio encargado de esta tarea, así que el aprendizaje autodidacta fue la solución para poder cumplir con el desarrollo de la aplicación. De forma similar, la asignación no contaba con un diseño visual ni representativo, por lo que tuve que valerme de un bosquejo creado por mi jefe directo para tener un punto de partida. Estos meses me ayudaron a descubrir la capacidad autodidacta que tenía y que más adelante me servirían no solo para aprender nuevos lenguajes de programación sino también cómo desarrollar software de mejor calidad.

a) Wanna Android app

La recomendación de un amigo me llevó a trabajar en Curiosity Startup por mi experiencia en ITLAB. El proyecto Wanna fue mi primer aplicativo Android que llegó a la Play Store aunque haya sido una prueba cerrada para probar el flujo de despliegue a los usuarios.

Como encargado de la parte de interfaz de usuario y con una corta experiencia previa, los primeros resultados del código fueron poco sostenibles y escalables por falta de orden tanto en el código, en la arquitectura y en la estructura del proyecto, ver Figura 3.1. Una actividad podía contener los llamados REST y allí mismo modificar los datos que se podían presentar en otra vista. Es por esto que se rehizo parcialmente el proyecto para acomodar un mejor código que permitiese un crecimiento más ordenado. El resultado fue la PoC con las siguientes características:

- Falta de un modelo de arquitectura.
- Patrón repositorio e inyección de dependencias.
- Separación de carpetas por vistas, APIs y utilitarios.

La continua falta de un modelo de arquitectura influenció en la estructura del proyecto. Lo más natural que resultó al momento de empezar a generar código fue crear una carpeta principal llamada *main* donde tuviese otras carpetas con el nombre de la vista o pantalla funcional. Por ejemplo, *home* tenía todas las actividades para un CRU de un evento, o *login* con actividades para iniciar sesión, verificar cuenta y registro. Al mismo tiempo se crearon otras carpetas para el patrón repositorio, la inyección de dependencias, los modelos de los objetos y respuestas API, y una última llamada *utils* con clases reutilizables en todo el proyecto, p. ej. para interceptar SMS's o transformar una imagen a 64 bits.

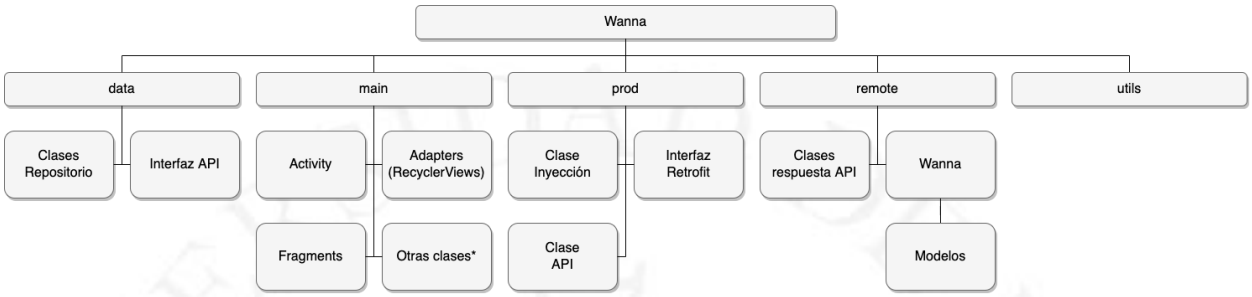
La razón por la que se decidió usar el patrón repositorio junto a la inyección de dependencias fue para ahorrar la creación de múltiples archivos por vista o funcionalidad y tener unos pocos archivos con todas las funciones necesarias para el flujo de datos a través de interfaces. Todas estas actividades tenían inyectadas un *presenter* entregado por el patrón repositorio y un patrón singleton. El inyector obtiene la instancia de la clase correspondiente y regresa la implementación correspondiente del repositorio con los métodos de la interfaz Retrofit y API.

A pesar de que el nombre de presentador pertenece al modelo MVP, su función en el proyecto fue llamar a los servicios REST y entregar las respuestas a la vista correspondiente

sin necesidad de cambiar un modelo ya que la vista almacenaba los datos a mostrar. Es así que las actividades solo requerían extender una interfaz y los métodos sobrecargados eran ejecutados por el *presenter* cuando la respuesta se retornaba.

Figura 3.1

Estructura del proyecto Wanna



* Otras clases necesarias para la actividad, p. ej. Clases objeto para animaciones o para manejar un calendario.

b) Qempo Android app

Para este proyecto se decidió usar la arquitectura MVP y dejar de usar el patrón repositorio con inyección de dependencias. Usar el mismo patrón que se usó en Wanna, como ya se mencionó, requería tener todas las funciones para el flujo del negocio en pocos archivos, lo que haría el proyecto más difícil de escalar con estos archivos extremadamente extensos. Por otro lado, la arquitectura permitió un desacoplamiento entre la vista y el *presenter* para dar paso al modelo y al *interactor*, este último como variación del modelo para centralizar la interacción entre la interfaz de usuario y el manejo de datos. Como se explicó en el primer capítulo, el flujo de la información en esta aplicación era una comunicación bidireccional secuencial porque el flujo inicia con un evento de usuario, pasa por el *presenter*, el *interactor* y este llama quien necesite para devolver la información al usuario, una vez más, a través del *presenter*.

El proyecto se dividió diferentes carpetas, de las cuales las principales fueron las siguientes:

- UI (*User Interface* por sus siglas en inglés): No solo tenía las actividades sino también interfaces, fragmentos y *RecyclerViews*.
- Remote: Con un singleton para los llamados REST, interfaces para los llamados de cada funcionalidad o vista y los modelos de datos.

- Presenter e Interactor: Cada carpeta con las clases correspondientes a cada segmento visual de la aplicación, actividad y/o fragmento.

Los fragmentos o *fragments* son una parte reutilizable de la interfaz de usuario (*Fragments*, 2021) que permitieron manejar las vistas con mayor fluidez por consumir menos recursos de procesamiento que una actividad. Por esto es que se usaban para cambiar la interfaz de usuario dentro de una misma actividad en lugar de intercambiar o transicionar entre *activities*.

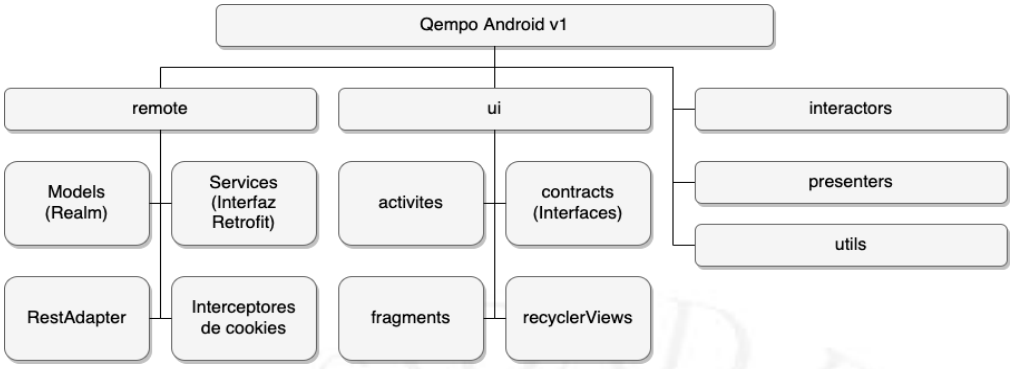
En cuanto a los *RecyclerViews*, estas clases son adaptadores que optimizan la renderización de listas. Provistos por Android mismo, aumentan el rendimiento de la aplicación al "reciclar" los contenidos de la lista y evitar procesamiento innecesario, así como almacenamiento en memoria. La implementación de estos adaptadores fue el primer intento en mejorar el procesamiento visual de la app, es decir, una mejor experiencia de usuario.

Otra parte importante de la aplicación fue el uso de la librería Retrofit en el archivo *RestAdapter* para los llamados REST mediante un singleton y las interfaces que sirvieron como intermediarios para ejecutar los llamados y devolver las respuestas correspondientes. Además, los modelos o clases que extendían *RealmObject*, parte de la librería Realm, servían para mantener los datos almacenados en la memoria del dispositivo y ser usados sin conexión al servidor de la aplicación.

La lógica del negocio estuvo separada en distintos archivos por módulo o funcionalidad bajo la carpeta UI. Como se puede ver en la Figura 3.2, dentro de UI está la estructura de cada funcionalidad con las actividades y los contratos como requeridos mientras los otros dos, opcionales. Es por esto que dentro de cada carpeta se pueden encontrar nombres muy similares cuyo sufijo pertenece a la carpeta que lo alberga. Por ejemplo, la funcionalidad Login tuvo LoginView, interfaz de la actividad, LoginService, interfaz para llamados REST, LoginInteractor, clase encargada de conectar la vista con los datos y así otras clases Java más.

Figura 3.2

Estructura del proyecto Qempo Android v1

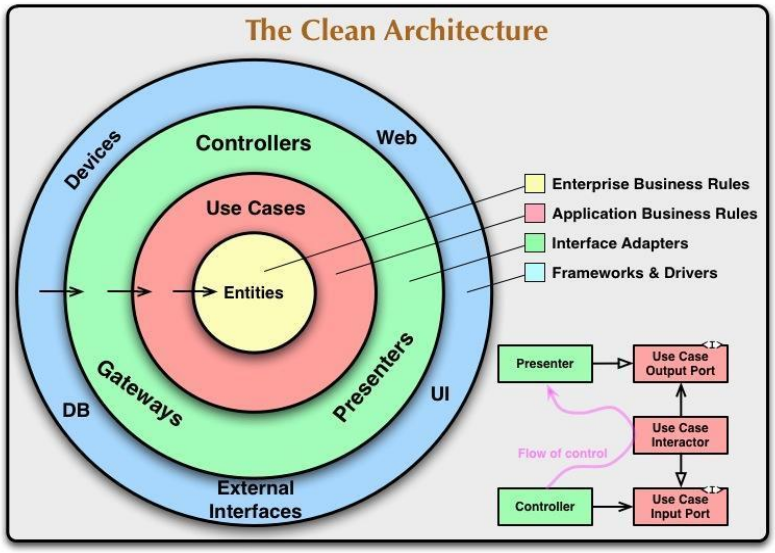


Aunque usar MVP permitió tener un mejor orden al momento de conectar las partes del código, la implementación para este proyecto también llevó a que se generen muchas clases con un mismo prefijo y un sufijo dependiente de la carpeta como se explicó recientemente. De forma similar, la separación de las carpetas hacía poco por desacoplar las reglas del negocio de la lógica del aplicativo e incentivaba que se mezclaran en cada actividad, interfaz o fragmento.

El problema de alto acoplamiento también lo sufría el equipo de desarrollo web, es por ello que se decidió investigar una solución de arquitectura. El conocimiento de la arquitectura MVP nos llevó a encontrar la arquitectura de software *Clean Architecture*, propuesta por Robert Martin en su blog *The Clean Code Blog.*

Figura 3.3

Gráfico de capas y flujo de Clean Architecture



Nota. Gráfico de capas y flujo de Clean Architecture. Adaptado de *Clean Architecture: A Craftsman's Guide to Software Structure and Design* (p.203), por Robert C. Martin, 2017, Prentice Hall

Según el autor, al aplicar reglas universales de arquitectura de software, puede mejorar drásticamente la productividad del desarrollador a lo largo de la vida útil de cualquier sistema de software (Martin, 2012). Para el autor, todos los sistemas de arquitectura tienen un mismo objetivo, que es la separación de responsabilidades. Estas arquitecturas logran esta separación dividiendo el software en capas. Cada una tiene al menos una capa para las reglas del negocio y otra para interfaces. De la Figura 3.3 se entiende que cada anillo representa una capa de abstracción, una separación de capas en forma de cebolla con una regla de dependencia: las capas internas no deben saber nada sobre las capas externas. Las capas internas contienen lógica del negocio, las capas externas contienen la implementación de las tecnologías y la capa intermedia contiene los adaptadores de interfaz.

Clean Architecture es mejor entendida como una "meta arquitectura", una guía de alto nivel para crear arquitecturas en capas y lograr el objetivo de desacoplamiento, separar responsabilidades. Las reglas son generales y no específicas, por ejemplo, en MVP, la vista puede conocer al presentador, pero el presentador es independiente de la vista por medio de una interfaz. Cualquier diseño de interfaz de usuario en el que las "capas internas" desconozcan sobre la interfaz de usuario, y en el que esta siga también las otras reglas descritas en el artículo de Bob Martin, pueden ser consideradas como parte de esta arquitectura.

Como consecuencia, la siguiente versión de la aplicación, y también el proyecto web, tuvieron cambios en su implementación. Por la parte Android, los cambios para reflejar la separación en capas ocurrieron en la estructura del proyecto:

- Separar la lógica del negocio y la lógica del aplicativo en dos carpetas: *core* y *app*.
- Usar la librería Dagger para volver a usar inyección de dependencias, pero sin el patrón repositorio.
- Separar las funcionalidades por carpetas en lugar de archivos.

Dentro de las carpetas *app* y *core* estaban las carpetas de las funcionalidades. Para la primera carpeta, cada funcionalidad contenía las actividades y fragmentos de las vistas correspondientes, así como sus interfaces mediante los cuales los *presenters* se comunicaban. También estaban las clases de los dos servicios de datos: API y base de datos bajo los nombres de RealmService y RetrofitService respectivamente como se puede ver en la figura 3.4. Estos nombres y sufijos de las clases llevan el nombre de la librería por ser justamente parte de la lógica de la aplicación. Además de estos archivos, se tenía una carpeta *di* (*dependency inyection* por sus siglas en inglés) con las interfaces y clases abstractas para la inyección de dependencias en cada vista. Por último, otra carpeta adicional con los adaptadores de ser necesarios.

Con respecto a la segunda carpeta, *core*, cada funcionalidad tenía la interfaz del presentador, el interactor, y, de ser necesario, clases Java de respuesta y petición. Similar a la primera carpeta, esta contenía dos subcarpetas, una para los modelos de los objetos y otra para las interfaces de API y base datos. Esta última llevaba el nombre de *gateway* por ser el intermediario del flujo de datos entre el negocio y la aplicación.

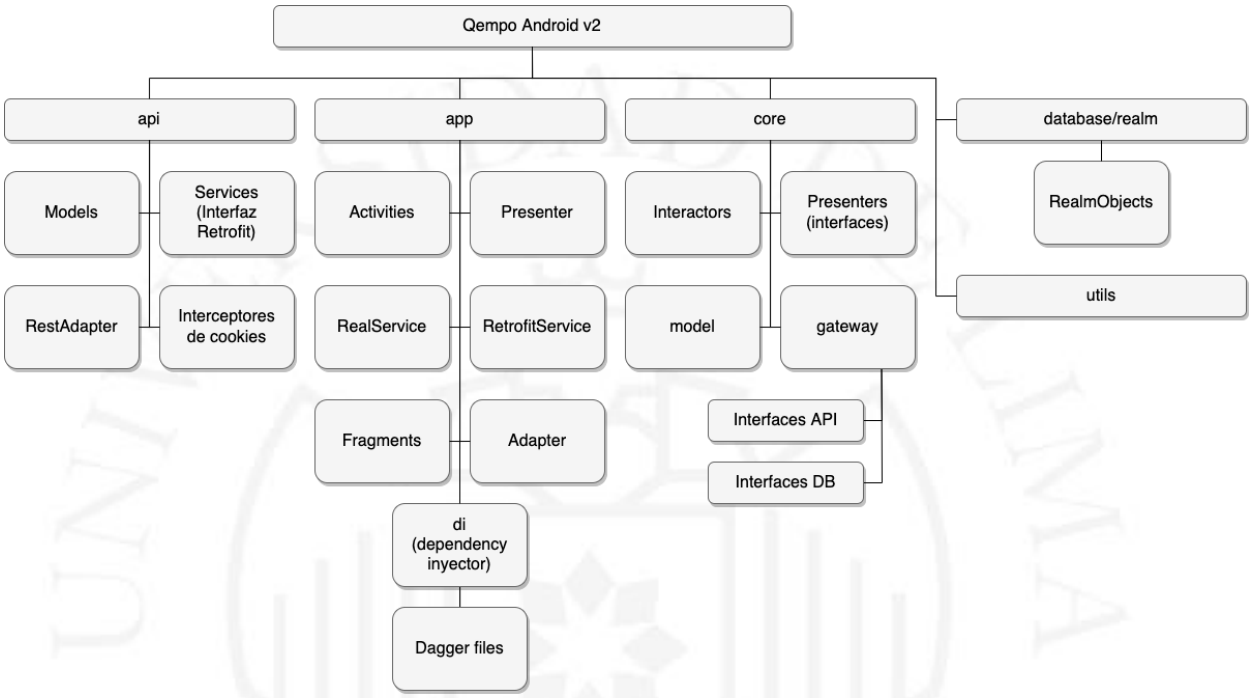
Así como en los dos proyectos anteriores, este tuvo una carpeta utilitaria, pero no solo eso sino que también tuvo una carpeta llamada *api* para separar todo lo relacionado a la librería Retrofit. Aquí estaban los modelos de las respuestas recibidas por el servidor, las interfaces con los llamados REST, el singleton de la librería y clases para modificar las cookies.

Esta última estructura del proyecto, como se ve en la Figura 3.4, a pesar de ser el resultado de correcciones e iteraciones, no pudo ser refinada porque, como se mencionó en otro capítulo, el proyecto fue cancelado para dar prioridad a su parte web. Se aprendió la importancia de la separación de las reglas del negocio y la aplicación, en otras palabras, si en algún momento se decide cambiar la librería Retrofit a otra, la refactorización debería

afectar en los más mínimo al código dentro de la carpeta *core*, de igual manera, si tuviéramos que prescindir de viajeros, el código pertinente dentro de *app* solo tendría que ser removido. Desacoplar ambas partes genera un código modular y más mantenible.

Figura 3.4

Estructura del proyecto Qempo Android v2



c) Qempo Mobile (React Native App) y Qempo Webapp

Con este proyecto, mis paradigmas de programación tuvieron que cambiar porque se requería aprender Javascript, un lenguaje de programación liviano y usado comúnmente como parte de páginas web. Empecé con ES6, conocido también como ECMAScript 2015 o ECMAScript 6, el estándar del lenguaje introducido el 2015. Aquí me enfrenté con algunos desafíos por ser el segundo lenguaje de programación en mi repertorio.

Tuve que entender la diferencia entre un lenguaje interpretado y uno compilado. Los archivos Javascript son textos que son interpretados por el cliente, por ejemplo, un navegador, mientras que en Java se compilan a bytecode, un lenguaje intermedio, para luego ser leído en un ordenador que lo ejecute.

También aprender entre un lenguaje orientado a objetos y un lenguaje de scripting basado en objetos con capacidad de emular programación orientada a objetos; javascript este

último y Java el primero. Así mismo, entender la diferencia entre una variable mutable e inmutable y cómo manejar la mutabilidad y sus efectos secundarios en Javascript.

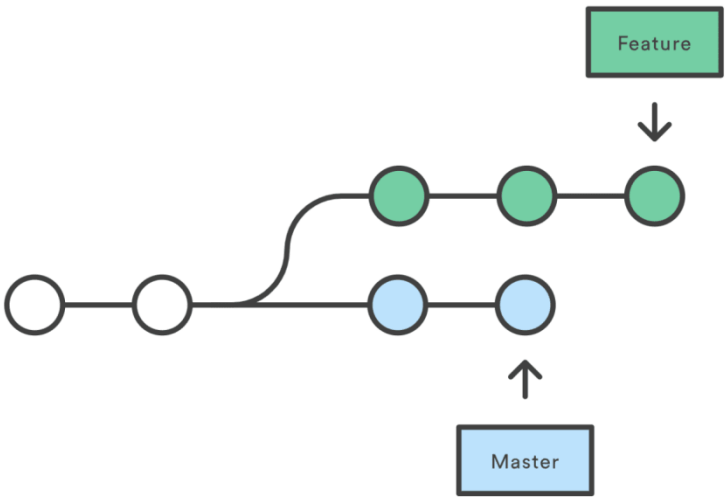
Por otro lado, aprendí a usar una librería de tipado para Javascript, Flow, porque el lenguaje está débilmente tipado. Mientras que una variable puede ser un objeto, número, texto o arreglo, en Java todas las variables tienen un tipo determinado, y, una vez definidas, no se pueden cambiar. Aunque pueda haber ventajas en la capacidad de representar diferentes tipos de datos, los errores pueden pasar desapercibidos por ser un lenguaje interpretado y no compilado.

Para el desarrollo de todos los proyectos anteriores se usó un sistema de control de versiones distribuido gratuito y de código abierto llamado Git. La razón fue su fácil aprendizaje y uso más allá de todas las capacidades que puede brindar en proyectos más grandes. Tenía lo necesario para guardar nuestro código. Dicho de otra manera, no era necesario entender y conocer todas sus funcionalidades para usarlo. Es por esto que durante el proyecto Wanna se trabajó en la misma rama (*branch*) *main*. Luego, para las dos versiones del proyecto Qempo Android se usaron ramas distintas a la principal con la intención de tener esta rama separada del desarrollo y posibles errores.

Si bien se tenía la noción de que una rama debía usarse para desarrollar una funcionalidad, no fue hasta luego de leer el artículo "Git Feature Branch Workflow" (*Git Feature Branch Workflow*, s.f.) escrito en Atlassian, la suite de múltiples herramientas mencionadas, y entender el gráfico de la Figura 3.5 que el uso de ramas se incrementó para el proyecto de React Native. A pesar de que este flujo de trabajo facilita que varios desarrolladores trabajen en simultáneo sin alterar el código principal el único desarrollador era mi persona. Esto llevó a que las ramas no tuvieran una estructura definida, tanto por el nombre, tipo de rama (funcionalidad o arreglo) o una combinación de ambas. Otra consecuencia de trabajar solo fue que los *Pull Request*, o solicitudes de unión de una rama a otra, a la rama principal eran creados y aprobados por mí mismo.

Figura 3.5

Flujo de trabajo por ramas de funcionalidad



Nota. Adaptado de *An Overview of Git Branching Strategies (Git Workflow)*, por Amit Kumar, (https://www.blog.techtious.com/tag/feature-branch-workflow/)

Por otra parte, el mensaje de los *commits*, cambios de código guardados en el repositorio, empezó a mejorar y tener más claridad sobre los cambios respectivos pues el mensaje debía comunicar de forma clara y concisa qué es lo que cambió. Esta mejora ayudó a tener revisiones de código más rápidas pues la necesidad de ver los cambios se redujo a leer el mensaje del *commit*.

En cuanto a la estructura del proyecto y expandiendo lo mencionado en el primer capítulo, la carpeta de los modelos contenía el tipado de los objetos y la carpeta *utils* siguió existiendo con el mismo propósito de servir como un centro de funciones utilitarias para el código. Adicionalmente, había otras carpetas más como parte de la arquitectura Flux, el marco de trabajo React Native y lo aprendido:

- Navigator: Dentro, un archivo con la estructura simple de las pantallas y configuración de la navegación.
- Store: Dentro, un archivo con la configuración del *store* y sus *middlewares*.
- Http: Archivos para manejar el envío de peticiones a internet.

Años más adelante, con el proyecto web en marcha y la aplicación híbrida cancelada como se mencionó en el primer capítulo, la empresa ganó un concurso de startups de una incubadora e inversionista llamada "The Venture City". Durante los días que duró la asesoría presencial nos enseñaron sobre una metodología usada por Google desde 1999 e introducida

por John Doerr llamada *Objectives and Key Results* (Objetivos y resultados clave en inglés). Una poderosa metodología de seguimiento y establecimiento de objetivos. Los OKR ayudan a alinear al equipo y se enfocan en lo que es importante para la organización. El establecimiento de objetivos, de acuerdo con los OKR, aumenta la autonomía del equipo, el compromiso de los trabajadores y genera muchos otros beneficios más.

En la práctica, a diferencia de otras técnicas de establecimiento de objetivos los OKRs apuntan a establecer metas muy ambiciosas. Cuando se usan de esta manera, los OKR pueden permitir que los equipos se concentren en los grandes logros y alcancen más de lo que el equipo pensó que era posible, incluso si no cumplen por completo el objetivo establecido. (*Re:Work - Guide: Set Goals With OKRs*, s.f.). Parte de la metodología consiste en la revisión de dichos objetivos, así que tomando ejemplo de Google decidimos hacerlo trimestralmente. De igual forma, tomamos el rango de porcentaje de satisfacción recomendado y se decidió considerar un objetivo cumplido con el 70% de resultados satisfactorios. Luego de haberse establecidos los objetivos del negocio se elaboraron también OKRs alineados a estos mismos para el equipo de tecnología.

Por ejemplo, si un objetivo del negocio es mejorar la experiencia en el proceso de cotización, entonces un OKR para el área de tecnología se vería como en la Tabla 3.1

Tabla 3.1

Ejemplo de un OKR

Objetivo	Resultados
Mejorar el algoritmo de la cotización y los pasos para completarla	1. Encontrar la cotización correcta para el producto elegido 4 veces de cada 5 intentos 2. Mantener actualizadas las reglas de cotización cada 2 semanas 3. Reducir en 15% la cantidad de personas que abandonan el flujo.

A razón de lo descrito en este capítulo, las materias y conocimientos aprendidos durante la carrera universitaria fueron complementados con habilidades autodidactas y enseñanzas tanto de cada proyecto como de los compañeros de equipo para cumplir con los diferentes proyectos y responsabilidades en Curiosity Startup y, posteriormente, Qempo. Inicialmente, el aprendizaje dentro de la función como desarrollador frontend estaba relacionado principalmente con la tecnología y aplicativo Android. Más adelante en el

proyecto web de Qempo como parte de un equipo de desarrollo, se aprendió la dinámica de programación colaborativa, gestión de proyectos y objetivos de negocio.

En resumen, como parte del aprendizaje continuo se logró alcanzar un sólido conocimiento en las siguientes áreas:

Área técnica:

- Java y Javascript (Lenguajes de programación)
- Patrón repositorio, inyección de dependencias, singleton (Patrones de diseño)
- Model-view-presenter (Patrón de arquitectura)
- Estructura de proyectos (Arquitectura de software)
- *Clean Architecture* (Arquitectura de software)
- Flujo de trabajo *Feature Branch* (Gestión del código)

Área de gestión:

- Metodología ágil (Gestión de proyectos)
- OKR (Estrategias de negocio)

En este sentido, mi siguiente paso en el ámbito profesional estará definitivamente orientado a mejorar mis habilidades gerenciales como la gestión de proyectos y gestión de varios equipos de desarrollo. También estará orientado al mantenimiento y cambios de la arquitectura de servidores para poder soportar gran carga de datos y acomodar los recursos necesarios de TI para poder satisfacer las necesidades transaccionales del negocio.

4. CONDUCTA ÉTICA

Las decisiones tomadas a lo largo de mi carrera profesional en los diferentes proyectos y giros de negocio como programador frontend han tenido un impacto apreciable en el desarrollo de estas empresas. Con cada solución o iniciativa siempre existía un grado de responsabilidad moral para actuar con ética además de estar siempre alineadas con la visión y misión del negocio. Es decir, desde apuntar a tener un código cada vez más mantenible hasta generar OKRs para el negocio, el crecimiento del negocio ha estado también gobernado por conductas éticas.

Los lineamientos a continuación explicados y detallados algunos de los puntos que conforman el Código de Ética y Conducta Profesional de ACM (Association for Computing Machinery, 2018).

Dado que el modelo de negocio de Qempo implica registrar datos de clientes y adquisiciones de productos siempre ha sido prioritario manejar la información dentro de los parámetros de ética, confiabilidad y sobre todo garantizar el uso adecuado de los mismos.

a) Principios éticos generales

- **Contribuir a la sociedad y al bienestar humano, reconociendo que todas las personas son partes interesadas en la Informática:**

  Miembros del laboratorio de investigación y tecnología, ITLAB, de la Universidad de Lima impartieron talleres a los alumnos de la universidad con el propósito de compartir el conocimiento generado producto de la investigación como herramienta tecnológica y darles una herramienta más en su carrera profesional.

- **Evitar el daño:**

  Con la aplicación Wanna, compartir los horarios podía significar que un usuario sea sujeto a vigilancia malintencionada por alguno de los contactos con los que comparte la aplicación. Para aminorar esta posibilidad, los espacios ocupados debían aparecer sin mostrar información de a quién le pertenecía ese tiempo. No obstante, de tener un evento de solo dos personas

sería evidente conocer estos espacios tanto del organizador como del participante.

- **Ser honesto y confiable:**

En Qempo se ofrecen diez dólares por invitar a un amigo y que realice su primera compra. El descuento es del 30% en los costos de procesamiento y la recompensa monetaria es sobre la compra total. Lo ofrecido puede ser a veces malinterpretado como 30% sobre la compra total. Para evitar esto, me encargué de crear los componentes necesarios para mostrar los términos y condiciones de esta promoción donde se explica a qué se aplica el descuento y bajo qué circunstancias. De forma similar, cuando salen promociones o descuentos, los detalles están disponibles al usuario a través de un link o en comunicación con el servicio al cliente.

Así mismo, los artículos periodísticos y testimonios de clientes fueron creados como parte de mi tarea de crear secciones en la página web como parte de una estrategia para generar confiabilidad en los clientes. En estos segmentos se recopila información verificable de clientes reales que han brindado una reseña y links a los diferentes artículos digitales donde Qempo es mencionado.

Como empresa de importación, la confianza en que entreguemos el producto al cliente final es un pilar muy importante para el éxito del negocio. Sin embargo, existen múltiples razones por las cuales los clientes pueden desconfiar del servicio, entre las más comunes están:

- El producto demora en llegar.
- El servicio al cliente tarda en responder.
- El costo es elevado para productos de especialidad.
- Una reseña negativa en algún lado de la web.
- Un retraso en los correos de actualización de estado del pedido.

Para abordar estas inquietudes se hizo parte integral del negocio cumplir con márgenes de satisfacción y brindar beneficios a los clientes insatisfechos. De ser necesario, yo era el encargado de crear cupones para los clientes en el sistema y asignarlos al usuario correspondiente.

- **Respetar el trabajo necesario para producir nuevas ideas, inventos, trabajos creativos y artefactos informáticos:**

  Tanto en la aplicación de ITLAB como en el aplicativo Wanna, la oportunidad de crear la interfaz de usuario fue dada a mi persona. Si bien es cierto que ambos proyectos carecían de un diseñador que se encargue de esta tarea, el objetivo de encargar la tarea al único desarrollador fue el de promover interacciones de usuario con la perspectiva de un programador.

  Una extensión para el navegador Chrome fue propuesta por el equipo de Qempo, pero llevada a cabo por mi persona ya que me ofrecí a concretar la idea. La iniciativa consistió en crear una herramienta que permita a sus usuarios generar una orden de compra con mayor facilidad y rapidez. En otras palabras, desde una página de Amazon (https://www.amazon.com) poder abrir la extensión, generar la cotización y empezar el flujo de compra en una nueva pestaña.

- **Respetar la privacidad:**

  Con el crecimiento a través de los años, la empresa Qempo ha alcanzado a almacenar gran cantidad de datos no solo de la información de sus clientes sino también de las transacciones que realizan. Es por eso que se tiene bastante resguardo para el almacenamiento de toda la data.

  Un ejemplo de los esfuerzos por proteger al usuario y a su información es que el backend no guarda información sensible como contraseñas en texto plano como sí lo hacía con el nombre o dirección del cliente. Lo que se escribe en la base de datos es el resultado de una función *hash*, un algoritmo matemático que transforma cualquier bloque arbitrario de datos en una nueva serie de caracteres con una longitud fija (Donohue, 2014). Este algoritmo es una encriptación irreversible, es decir, si es que un atacante obtuviera *hash*, este no podría extraer la contraseña original. No obstante, para verificar la validez de la contraseña cuando el usuario la ingresa, esta es encriptada con la misma función e igualada a los caracteres almacenados.

Por otro lado, en el frontend, la forma en cómo se realiza el análisis del comportamiento de los usuarios es de especial consideración para su privacidad. Las herramientas terceras como Google Analytics, Segment.io o Amplitud no deben conocer quién es el usuario pues no se debe distribuir datos sensibles como su correo electrónico o número de teléfono. En cambio, el equipo de desarrollo sugirió que genere un código único por usuario de forma interna para ser identificado ante dichas herramientas externas y así lograr examinar patrones de uso en la página web.

- **Respetar la confidencialidad**

Para el procesamiento de los pagos, el usuario debe ingresar la información bancaria o de su tarjeta. Esta información es una de las más confidenciales que el usuario puede dar, seguido de la foto de su documento de identidad. El desarrollo de este paso en la compra del producto tuvo un gran debate entre el equipo de tecnología y el equipo de desarrollo, siempre con la presencia de los *stakeholders*. Hubo dos opciones:

  - Un formulario de pago alineado a nuestro diseño de marca.
  - Un formulario estándar provisto por el procesador de pagos.

La primera opción fue la que el equipo de diseño quiso porque respetaba la estética de la página web. Sin embargo, la segunda opción promovida por el equipo de desarrollo al que pertenecí fue la escogida. La razón principal fue mantener la confidencialidad de los datos del medio de pago. Con un diseño propio, los datos tenían que pasar por nuestro servidor, en consecuencia, se debía certificar nuestro servicio bajo el estándar de seguridad de datos PCI (Payment Card Industry por sus siglas en inglés). Mientras que en la otra opción no se necesitaba la certificación, también nos libraba de la vulnerabilidad de exponer estos datos ante algún ataque. Es así que mantener una experiencia de diseño a través de toda la página no justificó el costo de la certificación tanto económica como en recursos horas/hombre.

b) Responsabilidades profesionales

- **Esforzarse por lograr una alta calidad tanto en los procesos como en los productos del trabajo profesional:**

Tener un nivel de calidad adecuado en el equipo de tecnología fue un aspecto que fue creciendo con el pasar del tiempo. Con cada nueva versión que era lanzada al público y los errores que traía consigo, se aprendió a ser más cuidadosos con el producto final, el software.

Uno de los esfuerzos para este control de calidad fue la adopción del modelo *Feature Branch* en el sistema de control de versiones. Este flujo de trabajo daba lugar a generar *Pull Requests* para que el código a unirse a nuestra rama maestra sea revisado por el líder técnico. De esta forma se tiene un paso más en el control de calidad del código. Una persona más que verifica que el código se ajuste a los estándares de calidad del equipo y así evitar errores en el código o código que pueda llevar en un mediano o largo a errores o generar deuda técnica.

Por otro lado, sobre la metodología ágil Scrum, algunos de los rituales que se usaron para mantener la calidad de las *stories* y el *backlog* fueron la planificación y revisión del sprint, organización del backlog y la retrospectiva. Todas estas ceremonias se enfocan de una forma u otra en la calidad de lo que se trabaja. Sin embargo, como se detalló en el capítulo segundo, estos rituales fueron relegados a ser parte de la labor de planificación y organización de un *stakeholder* junto al *product owner*. Esto ocurrió a razón del tiempo que se tenía que dedicar por cada ritual. Los beneficios eran muy bajos comparados con los resultados que se podían obtener al continuar con el desarrollo del producto además de ser la opinión del equipo de desarrollo respecto a estos eventos.

- **Aceptar y proporcionar una revisión profesional adecuada:**

  En el flujo del ciclo de vida de desarrollo que tiene el equipo, existen dos pasos importantes finales antes de la entrega del producto al usuario final. Tanto la parte tecnológica como la parte del producto hacen una revisión antes de continuar con el proceso de entrega.

  Revisión de código: Antes mencionado como *Pull Request*, estas peticiones validan que el código sea compacto, siga mejores prácticas de desarrollo, aplique patrones de diseño, tenga poca complejidad ciclomática, etc. Esto

aseguraba al equipo que el software es mantenible y está suficientemente desacoplado.

Revisión del producto: Una vez el código ha pasado la revisión de código y ha sido desplegado al servidor de desarrollo, el equipo de producto, es decir, los *stakeholders*, empiezan a probar las funcionalidades añadidas y/o a verificar la solución a un error. Dichas pruebas o verificaciones son realizadas en la página web simulando a un usuario cualquiera.

- **Acceder a los recursos informáticos y de comunicación sólo cuando esté autorizado, o cuando sea necesario para proteger el bien público:**

  Como parte del respeto por los derechos ARCO (Acceso, Rectificación, Cancelación y Oposición), los usuarios que ya no deseaban ser parte de Qempo podían comunicarse con el equipo de atención al cliente y pedir que sus datos sean removidos de nuestros servidores. Para poder validar el pedido del cliente, se le pedía que demuestre que efectivamente el usuario que deseaba borrar era de su propiedad a través de información sensible como el número de su documento de identidad, una de sus direcciones de envío, un código de verificación o una combinación de estos u otros datos. Una vez verificada la identidad del usuario y la pertenencia del mismo, el equipo de tecnología procedía a borrar su información.

c) Principios de liderazgo profesional

- **Asegurar que el bien público sea la preocupación central en el trabajo profesional:**

  Una de las dudas que los clientes presentaban antes de usar los servicios de Qempo fue acerca del proceso de importación y cómo es que la conexión a través de viajeros era permitida por Aduanas. Para responder esta duda y otras similares, me encargué de crear componentes y actualizar páginas donde se informa al usuario que el precio del servicio incluye los impuestos que el viajero tenía que pagar a Aduanas para cubrir el proceso de importación. Este costo fue considerado desde el inicio de las operaciones porque de lo contrario se estaría infringiendo las leyes y perjudicando al cliente con procesos legales.

- **Crear oportunidades para que los miembros de la organización o el grupo crezcan como profesionales:**

  Dentro del área de tecnología, más específicamente el frontend, las oportunidades son escasas en proyectos de lenguaje Javascript con un *framework* como React. Qempo, al estar bajo ese marco descrito, pudo generar pérdida de motivación en los miembros de su equipo y evitar la mejora continua en el desarrollo del software. Es por esto que un acuerdo entre ellos y el CEO se llevó a cabo para promover proyectos pequeños de interés. Se trató de un tiempo destinado a que el desarrollador conozca y aprenda temas de interés sin necesidad de que esté relacionado al negocio. Es decir, que tenga la libertad de explorar nuevas tecnologías a través de un proyecto personal para evitar la fatiga de estar restringido a un solo ambiente tecnológico poco variable.

  Un claro ejemplo fruto de esta iniciativa fue un *bot*, un programa autónomo en Internet u otra red que puede interactuar con sistemas o usuarios. Creado por mi persona, el bot existía en el servicio de mensajería Telegram como una conversación con el usuario. Fue creado en el lenguaje Python y se encargaba de envíar, recibir y tabular encuestas semanales a los trabajadores, dichas encuestas eran auto evaluaciones de rendimiento semanal.

- **Tener cuidado al modificar o retirar sistemas:**

  El proceso de despliegue de una nueva versión de la página web constituye múltiples pasos donde el error humano está presente en cada uno de ellos. Algunos de estos errores podrían ser:

  - La página está caída y no se puede acceder a ella.
  - La página presenta problemas que no permiten su uso regular.
  - La base de datos presenta errores y las respuestas a la web causan errores.
  - El servidor presenta errores en el código y causa problemas en la página.

Todos estos errores se traducen en menos ventas, insatisfacción en los clientes, sobrecarga de consultas en el área del servicio al cliente. A razón de esto es que se creó un script de bash alojado en el servidor con las tareas a realizar. De esta forma y con las

actualizaciones necesarias que realicé al archivo, el error humano es reducido en cada despliegue y ejecutado siempre de la misma forma.

5. LECCIONES APRENDIDAS

A pesar de haber mantenido mi profesión en el área de frontend, no fue impedimento para desenvolver el trabajo a otras áreas como diseño de interfaz de usuario, gestión de proyectos o análisis de información. Tener una perspectiva informada de interfaz y experiencia de usuario puede mejorar el planeamiento y desarrollo de un proyecto porque la estructura del proyecto y las herramientas a utilizar pueden ser escogidas con mayor efectividad. Es decir, la sinergia de todos estos puntos resultará en un mejor software ya que tendrá bases sólidas en las que construirse.

Un ejemplo de lo mencionado son el proyecto Qempo Android y Qempo Mobile. La última versión del primer proyecto y la única del segundo fueron el resultado de múltiples iteraciones para alcanzar una estructura de proyecto adecuada a través de diferentes modelos de arquitectura, patrones de arquitectura y librerías externas. A pesar de no haber salido al público, ambos proyectos estaban preparados para más funcionalidades y la adición de pruebas unitarias.

Las decisiones de negocio deben ser respaldadas por información extraída de los datos almacenados en las bases de datos, es decir, métricas objetivas. Para escoger una idea sobre otra, la hipótesis de que la primera va a dar mejores resultados que la otra debe ser soportada por el conocimiento generado y el nivel de cumplimiento de los KPIs que produce. Por ejemplo, el ratio de la cantidad de órdenes pagadas versus la cantidad de productos cotizados como métrica ayuda a conocer qué porcentaje de éxito tiene la página de inicio a fin. Si una nueva página de inicio mejora este ratio, entonces la decisión de usar esa nueva página es la correcta.

Como parte de un análisis más detallado, el ratio mencionado cobra mayor significado cuando cada paso del proceso es medido con herramientas de analítica de usuario y podemos medir un conjunto de nuevas ideas como un cambio de UI en el paso de escoger un método de entrega. Si este cambio añadido a la nueva página de inicio reduce el ratio de compra completada, entonces la antigua interfaz debe permanecer.

La deuda técnica que se puede generar en un proyecto con tantas iteraciones como las que tuvo Qempo es considerable. Entender que la estructura de un proyecto, así como las tecnologías a usar y el enfoque de las herramientas pueden cambiar en un corto o mediano plazo es importante para saldar esta deuda. Cuando la necesidad de cambiar el rumbo del

negocio supera las consecuencias de tener un código de calidad deseado es que la deuda técnica se incrementa. Es así que en los ciclos de cambio el equipo de tecnología y los *stakeholders* deben mantener un balance entre implementar mejoras en el software o implementar una funcionalidad para mejorar el negocio. Para los inicios de Qempo, la intención de un marketplace para compradores y viajeros tuvo que ser desplazada por un servicio de importación con múltiples opciones. Los viajeros ya no serían la única manera de importar las compras sino también diferentes *couriers* con los que la empresa contrató, así como otros métodos importación de acuerdo a las necesidades del cliente.

Es así que en la experiencia del graduado lo más importante para alcanzar un relativo éxito en una startup es la constante y frecuente implementación de ideas que puedan mover el negocio hacia adelante. No obstante, para medir el éxito de la idea sobre cualquier otra se debe analizar y extraer la información generada a través de la recolección de datos. Una vez comprobado que la decisión es la adecuada porque cumple con los KPIs y otros indicadores, se puede integrar la idea al producto final considerando la posible deuda técnica que pueda generar

6. GLOSARIO DE TÉRMINOS

Acoplamiento: Nivel de interdependencia entre módulos de software que permite medir la relación y conexión entre estos. (ISO/IEC/IEEE International Standard - Systems and software engineering--Vocabulary, 2017, p. 107)

Anotación: En Java, una característica que le permite añadir información suplementaria a un archivo que es usada por diferentes herramientas durante y después del desarrollo. (Anotaciones en Java, 2021)

API: Interfaz de programación de aplicaciones. Mecanismos que permiten que dos componentes de software se comuniquen entre sí mediante un conjunto de definiciones y protocolos. (Saha, H. S., 2019)

B2C: Modelo de negocio donde una transacción tiene lugar entre una empresa y un individuo como cliente final.

Backend: Parte de una aplicación de computadora o el código de un programa que le permiten operar y a la que un usuario no puede acceder. (L. L. C., 2023)

C2C: Modelo de negocio en el que los clientes compran bienes de otros clientes a través de una plataforma o negocio de terceros. (Leung, W. K., Shi, S., & Chow, W. S., 2020)

CEO: El ejecutivo de más alto rango en una empresa cuyas responsabilidades principales incluyen la toma de decisiones corporativas importantes, la gestión de las operaciones y los recursos generales de una empresa. (What is a CEO?, 2016)

CI: Integración Continua es la práctica de automatizar la integración de los cambios de código de varios contribuidores en un único proyecto de software. (Rehkopf, What is continuous integration?, s.f.)

Columnas: Cada columna representa una actividad y juntas generan el flujo de trabajo.

Courier: Una persona o empresa que lleva mensajes, cartas o paquetes de una persona o lugar a otro. (Cambridge University Press, s.f.)

ERP: Planificación de recursos empresariales es un software que utilizan las compañías para gestionar distintas actividades empresariales diarias, como la contabilidad, la gestión de proyectos, la gestión de riesgos y las operaciones de la cadena de suministro. (Oracle, s.f.)

Frontend: La capa superior al backend que incluye todo el software o hardware que forma parte de una interfaz de usuario. (L. L. C., 2023)

Git: Sistema de control de versiones distribuido, gratuito y de código abierto diseñado para manejar desde proyectos pequeños hasta proyectos muy grandes, con rapidez y eficiencia. (Git, s.f.)

Hash: Un algoritmo matemático que convierte cualquier conjunto de datos arbitrario en una nueva secuencia de caracteres de longitud constante.

JVM: Máquina virtual Java que administra la memoria de la aplicación y proporciona un entorno de ejecución portátil para aplicaciones basadas en Java. (Tyson, 2022)

Librería: Un conjunto de archivos de código para desarrollar software. Se usan a menudo para reducir la cantidad de código escrito en funciones o clases reutilizables. (The difference between libraries and frameworks, 2023)

Límite de WIP: Cantidad máxima de cartas o tarjetas por columna.

Marco de trabajo: Un conjunto de bibliotecas o herramientas que estructuran la creación de aplicaciones. Incluyen librerías, plantillas y guías de cómo desarrollas estas aplicaciones (The difference between libraries and frameworks, 2023)

Mock: Un objeto simulado en el sistema que decide si una prueba unitaria ha pasado o no. (Reese, 2022)

MVC: Patrón de diseño de software comúnmente utilizado para implementar interfaces de usuario, datos y lógica de control. Separa la lógica del negocio, el software y la interfaz. (Sarcar, V., 2018)

MVP: Patrón de presentación de interfaz de usuario basado en los conceptos del patrón MVC que dicta cómo estructurar la vista. (Qureshi, M. R. J., & Sabir, F., 2014)

NPM: El registro de software más grande del mundo. Usado por desarrolladores de código abierto para compartir y usar librerías, así como por organizaciones para código privado. (Hafner, A., Mur, A., & Bernard, J., 2021)

Patrón de diseño: Solución general repetible sobre cómo resolver un problema común en el diseño de software. (Design Patterns, s.f.)

Patrón de arquitectura: Descripción del diseño y colección de componentes en sistemas que conforman los bloques de construcción del software. (Butani, 2020)

PCI: Un conjunto de requisitos que apuntan a garantizar que todas las empresas que procesan, almacenan o transmiten información de tarjetas de crédito mantengan un ambiente seguro de su sistema informático. (Harran, M., & Mckelvey, N., 2013)

PM2: Un administrador de procesos daemon que ayuda a administrar y mantener una aplicación en línea para Node.js. (Single Page Doc, s.f.)

PoC: Prueba de concepto de un proyecto que se ejecuta para demostrar que una idea de producto, plan de negocio o plan de proyecto es factible. (What Is Proof of Concept (POC)?, s.f.)

Punto de compromiso: Momento en el que una idea empieza el flujo de trabajo.

Punto de entrega: Momento en el que una idea termina el flujo de trabajo.

Refactorizar: Reestructurar y cambiar la estructura interna de código fuente sin cambiar o agregar a su comportamiento y funcionalidad original. (Fowler, Refactoring, s.f.)

SaaS: Servicio como software, una forma de acceder a aplicaciones a través del internet sin instalar o mantener software. (Power your business with the best of the web, s.f.)

Señales visuales: Post-it's, tarjetas con el ítem a trabajar.

SSR: La capacidad de una aplicación de procesar una página web en el servidor y devolverla al navegador para acelerar la velocidad en que la página carga. (Server-Side Rendering, s.f.)

Stakeholder: Un empleado, inversionista, cliente, etc. que participa o compra en una empresa y tiene interés en su éxito. (Cambridge University Press, s.f.)

Startup: Empresa generalmente autofinanciada, típicamente en las primeras etapas de su desarrollo, enfocada en capitalizar una demanda de mercado no satisfecha mediante el desarrollo de un producto, servicio o plataforma viable. (What Is a Startup Company, Anyway?, 2022)

React: Una librería en Javascript de frontend para construir interfaces de usuario. (React, s.f.)

REFERENCIAS

Drumond, C. (s.f.). *Scrum - what it is, how it works, and why it's awesome*. Atlassian. https://www.atlassian.com/agile/scrum

Fowler, M. (2006). *Continuous Integration*. Martin Fowler. https://martinfowler.com/articles/continuousIntegration.html

Git Feature Branch Workflow. (s.f.). *Atlassian.* https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow

Ownership Culture. (s.f.). Project Equity. https://project-equity.org/ownership-culture/

¿Qué es un repositorio de artefactos? | Guía de CI/CD de TeamCity. (s.f.). JetBrains. https://www.jetbrains.com/es-es/teamcity/ci-cd-guide/concepts/artifact-repository/

Rehkopf, M. (s.f.). *What is a Kanban Board?* Atlassian. https://www.atlassian.com/agile/kanban/boards

re:Work - Guide: Set goals with OKRs. (s.f.). *re:Work.* https://rework.withgoogle.com/guides/set-goals-with-okrs/steps/introduction/

What is Server-Side Rendering? Definition and FAQs. (s.f.). *HEAVY.AI.* https://www.heavy.ai/technical-glossary/server-side-rendering

Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Boston, MA: Prentice Hall. ISBN: 978-0-13-449416-6

Andreeva, E., Mennink, B., & Yasuda, K. (2011). *Cryptographic Hash Functions: Recent Design Trends and Security Notions*. Cryptology ePrint Archive, Paper 2011/565. https://eprint.iacr.org/2011/565

Association for Computing Machinery. (2018, 28 de Junio). *Código de Ética y Conducta Profesional de ACM*. https://www.acm.org/about-acm/code-of-ethics-in-spanish

Lemke, Gillian (2018). *The software development life cycle and its application.* Senior Honors Theses & Projects. 589.

Fragmentos. (2021, 26 de Octubre). Android Developers. https://developer.android.com/guide/fragments

In-Depth Overview | Flux. (2022, 3 de junio). Meta Open Source. https://facebook.github.io/flux/docs/in-depth-overview/

PCI Security Standard Council. (2022, agosto). *PCI DSS v4.0 Quick Reference Guide* (4). PCI Security Standards. https://docs-

prv.pcisecuritystandards.org/PCI%20DSS/Supporting%20Document/PCI_DSS-QRG-v4_0.pdf

ISO/IEC/IEEE International Standard - Systems and software engineering--Vocabulary. (2017). ISO/IEC/IEEE 24765:2017(E), 1-541.

Anotaciones en Java. (2021). TOKIO IO. https://www.tokioschool.com/noticias/anotaciones-en-java

Saha, H. S. (2019). *What is an API?*. Newstex.

Deustua Aris, M. Percepción de valor del branding en las startups B2C en el perú.

L. L. C. (2023, Apr 19). Frontend and backend, two key concepts in technology. *CE Noticias Financieras*.

Leung, W. K., Shi, S., & Chow, W. S. (2020). Impacts of user interactions on trust development in C2C social commerce: The central role of reciprocity. [C2C s-commerce] *Internet Research*, 30(1), 335-356. https://doi.org/10.1108/INTR-09-2018-0413

What is a CEO? (2016, Jun 28). *Progressive Digital Media Technology News.*

Cambridge University Press. (s.f.). *Cambridge Dictionary*.

Oracle. (s.f.). *What is ERP?* Oracle. https://www.oracle.com/erp/what-is-erp

Tyson, M. (2022, 28 de Octubre). *What is the JVM? Introducing the Java virtual machine*. Info World. https://www.infoworld.com/article/3272244/what-is-the-jvm-introducing-the-java-virtual-machine.html

The difference between libraries and frameworks. (2023). https://www.red-gate.com/simple-talk/development/other-development/the-difference-between-libraries-and-frameworks/

Reese, J. (2022, 4 de Noviembre). *Unit testing best practices with .NET Core and .NET Standard*. Learn Microsoft. https://learn.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices

Sarcar, V. (2018). MVC pattern. *Java design patterns* (pp. 437-457). Apress. https://doi.org/10.1007/978-1-4842-4078-6_26

Qureshi, M. R. J., & Sabir, F. (2014). A comparison of model view controller and model view presenter. https://doi.org/10.48550/arxiv.1408.5786

Hafner, A., Mur, A., & Bernard, J. (2021). Node package manager's dependency network robustness. https://doi.org/10.48550/arxiv.2110.11695

Butani, A. (2020, 16 de Diciembre). *5 essential patterns of software architecture*. Redhat. https://www.redhat.com/architect/5-essential-patterns-software-architecture

Harran, M., & Mckelvey, N. (2013). *PCI compliance – no excuses, please.* International Journal of Information and Network Security, 2(2), 118-123. https://doi.org/10.11591/ijins.v2i2.1940

Single Page Doc. (s.f.). PM2. https://pm2.keymetrics.io/docs/usage/pm2-doc-single-page

What Is Proof of Concept (POC)? (s.f.). TechTarget. https://www.techtarget.com/searchcio/definition/proof-of-concept-POC

Fowler, M. (s.f.). Refactoring. https://refactoring.com

Power your business with the best of the web. (s.f.). Salesforce. https://www.salesforce.com/in/saas

What Is a Startup Company, Anyway? (2022, 7 de Abril). Startups. https://www.startups.com/library/expert-advice/what-is-a-startup-company

React. (s.f.). Getting Started. https://reactjs.org/docs/getting-started.html

Git. (s.f.). Git. https://git-scm.com

Cordova (s.f). Apache Cordova. https://cordova.apache.org/

PhoneGap (s.f). Github. https://github.com/phonegap

What is Angular? (s.f). Angular. https://v5.angular.io/docs

BIBLIOGRAFÍA

Thomas, D.R., Beresford, A.R., Coudray, T., Sutcliffe, T., Taylor, A. (2015). *The Lifetime of Android API Vulnerabilities: Case Study on the JavaScript-to-Java Interface*. Security Protocols 2015. Lecture Notes in Computer Science. Vol 9379. Springer, Cham. https://doi.org/10.1007/978-3-319-26096-9_13

von Solms, R., & van Niekerk, J. (2013). *From information security to cyber security*. Computers & Security. Vol. 38, 97-102. https://doi.org/10.1016/j.cose.2013.04.004

Niven, P. R., & Lamorte, B. (2016). *Objectives and key results: Driving focus, alignment, and engagement with OKRs*. John Wiley & Sons.

Cervone, F. (2017, 27 de febrero). *Model-View-Presenter: Android guidelines | by Francesco Cervone*. Medium. https://medium.com/@cervonefrancesco/model-view-presenter-android-guidelines-94970b430ddf

Söhlemann, C. (2022, 31 de Marzo). *SSR or CSR - what is better for Progressive Web App?* Krusche & Company. https://kruschecompany.com/ssr-or-csr-for-progressive-web-app/

Object prototypes - Learn web development | MDN. (2022, 28 de septiembre). MDN Web Docs. https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes

TSP 20110669_SUSTENTACIÓN

INFORME DE ORIGINALIDAD

5%	4%	0%	4%
INDICE DE SIMILITUD	FUENTES DE INTERNET	PUBLICACIONES	TRABAJOS DEL ESTUDIANTE

FUENTES PRIMARIAS

1	Submitted to Universidad de Lima Trabajo del estudiante	1%
2	equipo3245.blogspot.com Fuente de Internet	<1%
3	Submitted to Universitat Politècnica de València Trabajo del estudiante	<1%
4	www.atlassian.com Fuente de Internet	<1%
5	Submitted to Aliat Universidades Trabajo del estudiante	<1%
6	Submitted to Universidad de Nebrija Trabajo del estudiante	<1%
7	Submitted to Universidad Nacional de San Cristóbal de Huamanga Trabajo del estudiante	<1%
8	exonegocios.com Fuente de Internet	<1%