Universitat Oberta de Catalunya

uoc.edu

www.GymBosses.com

Gabriel Paradiso
Grado de Ingeniería Informática
Desarrollo web

Gregorio Robles Martínez
Santi Caballe Llobet

12/06/2020

FICHA DEL TRABAJO FINAL

Título del trabajo:	*www.GymBosses.com*
Nombre del autor:	*Gabriel Paradiso Silva*
Nombre del consultor/a:	Gregorio Robles Martínez
Nombre del PRA:	*Santi Caballe Llobet*
Fecha de entrega (mm/aaaa):	06/2020
Titulación::	*Grado de Ingeniería Informática*
Área del Trabajo Final:	*Desarrollo Web*
Idioma del trabajo:	Castellano
Palabras clave	*Gimnasios, Desarrollo Web*

Resumen del Trabajo (máximo 250 palabras)

GymBosses es una aplicación web de gestión de gimnasios que dependía del soporte de un administrador para la gestión de cuentas, dificultado la expansión en el mercado. El objetivo de este proyecto fue automatizar la creación y pago de las suscripciones de la plataforma, lograr autonomía por parte del cliente y mejorar la experiencia del usuario. Para llevarlo a cabo, se utilizaron principios de metodologías ágiles para la planificación y desarrollo de las tareas. Se implementó las funcionalidades necesarias para darle autonomía al usuario, a través de una plataforma simple e intuitiva, reduciendo considerablemente el tiempo dedicado a soporte por parte del administrador.

Abstract (in English, 250 words or less):

GymBosses is a Web application for gym administration that depended on the support of an operator for account management, making it difficult to expand in the market. The goal of this project was to automate the creation and payment of platform subscriptions, achieve customer autonomy and improve the user experience. To carry out this project, principles of agile methodologies were used for the planning and development of the tasks. The necessary functionalities were implemented to give autonomy to the user, through a simple and intuitive platform, reducing considerably the time dedicated to support by the administrator.

Índice

Lista de figuras

1. Introducción

1.1 Contexto y justificación del Trabajo

GymBosses es un proyecto de expansión de la actual aplicación http://www.ClubAppUy.com, cuyo objetivo principal es gestionar clientes, controlar acceso y pagos de mensualidad en gimnasios. ClubAppUy fue lanzada en 2015 y desde entonces es utilizada por las tres sedes del “Gimnasio Forma” en Uruguay (Se puede encontrar una demo de este producto en https://www.youtube.com/watch?v=ybmug1Sa0ZU).

Este proyecto se desarrolló utilizando python/django, postgres y django templates, siguiendo el patrón MVT(model-view-template, similar a MVC) desplegado en Heroku.

Para acceder al gimnasio los clientes del mismo debe hacer “check-in” en una terminal (raspberry-pi + monitor + teclado) ingresando su DNI. Paralelamente, el responsable del gimnasio corrobora el estatus del cliente (si se encuentra al **día** o es **deudor**).

Los clientes a los que está dirigida la aplicación son pequeños/medianos gimnasios que quieran mejorar su gestión administrativa a través de una solución “user friendly” y económica. Actualmente ClubAppUy es rentable, pero las ganancias son mínimas dado que solo lo utiliza un cliente.

La creación de cuentas en la plataforma ClubAppUy, el pago de la suscripción al servicio y la creación/actualización de planes de pago, se realizan de forma manual por el administrador de la aplicación. Esto significa que la inversión de tiempo en mantenimiento de la plataforma es proporcional a la cantidad de gimnasios que utilizan la aplicación, limitando el crecimiento.

Por otra parte, a nivel técnico, el framework utilizado permitió tener un MVP de forma rápida, sin embargo, el alto acoplamiento al mismo tiene como resultado que realizar cambios o agregar nuevas funcionalidades se torne muy complejo.
GymBosses nace con el objetivo de solucionar las limitantes que hoy en día ClubAppUy presenta, mediante la utilización de un stack más apropiado y una arquitectura desacoplada que permita acelerar el desarrollo de nuevas funcionalidades.

Actualmente GymBosses está en pleno desarrollo, permitiendo:
La creación de cuentas sin necesidad de intervención del administrador de la plataforma y autenticación mediante JWT.

- La autogestión de planes de pagos (Creación de planes)
- Funcionalidades básicas:
    - Crear/listar/buscar clientes
    - Crear/listar pagos
    - Check-in de clientes al gym

1.2 Objetivos del Trabajo

El objetivo principal es lograr una versión estable de GymBosses que sea "Feature Parity" con ClubAppUy y que permita **expandirse** a más gimnasios. Para ello, es clave permitir la **autogestión** de los gimnasios y el **pago de la suscripción automatizado** para reducir el tiempo invertido en mantenimiento.
Se utilizará el framework MoSCoW con el que se asume el compromiso de terminar las tareas con el nivel **Must Have**, pero que idealmente se alcanzará el resto de los niveles. Se definen entonces los siguientes milestones:

Milestone 1 - Gimnasios autogestionables V1 🚀

El primer objetivo es ser "Feature Parity" con la aplicación ClubAppUy y agregar las funcionalidades que faltan para que cada gimnasio pueda autogestionarse sin necesidad de la interacción del administrador de la plataforma.

Must have:

- ✓ Definir set de datos "final" para la versión 1 que se van a almacenar para Clientes, Pagos, Ingresos y cómo se van a almacenar.
- ✓ Gestionar static files en S3.
- ✓ Autorefresh de la página de inicio para reflejar cada vez que un cliente nuevo hace check-in.

Should have:

- ✓ Trabajar en la homogeneidad de la interfaz.

Could have:

- ✓ Gestionar más de un gimnasio. El cliente actual tiene 3 gimnasios, por lo que debería poder gestionarlos usando una misma cuenta.
- ❑ Implementar Update/Delete de planes de pago.
- ❑ Incrementar cobertura de tests

Won't have:

- ❑ Pagos automatizados

Milestone 2 - Pagos automatizados 💰

El objetivo de este milestone es que el pago de la suscripción a la plataforma se haga a través de una pasarela de pagos y que el acceso a la aplicación esté condicionado por el mismo (un cliente que no paga su mensualidad no puede acceder a la aplicación).

Must have:

- ✓ Definir precio de la mensualidad
- ✓ Definir pasarela de pago a utilizar (Stripe, Paypal, Alipay)
- ✓ Realizar one time payment
- ✓ Controlar acceso a la aplicación dependiendo del estado del pago

Should have:

- ✓ Realizar pagos recurrentes. El cliente ingresa su tarjeta y mensualmente se le descuenta la cuota.
- ✓ Cancelar subscripción

Could have:

- ❑ Implementar precios diferenciados dependiendo de la cantidad de clientes

Won't have:

- ❑ Pagos de mensualidad de clientes del gimnasio

Milestone 3 - Transformar datos en Información 💡

Actualmente se tienen datos que no se están explotando, los datos son desplegados en tablas sin aportar información al dueño/administrador del gimnasio.

¿Quién es mi cliente típico?, ¿cuáles son las horas más concurridas?, ¿tengo más o menos clientes?, ¿estoy generando más dinero?, ¿cuanto tiempo en promedio dura la suscripción de mis clientes?, ¿cuáles son las características de los clientes que cancelan su suscripción?. Este milestone busca responder alguna de estas preguntas para ayudar a la toma de decisiones.

Must have:

- ✓ Definir 3 preguntas claves que ayuden a la toma de decisiones
- ✓ Implementar una solución para esas preguntas

Could have:

- ❑ Prever comportamientos como identificar clientes que estén por cancelar su suscripción y ayudar a la retención de los mismos.

1.3 Enfoque y método seguido

Previo a la presentación de la tesis se estudió la posibilidad de continuar el proyecto ClubAppUy, dado que es un producto estable, genera valor para el cliente y es utilizado a diario desde 2015.
Para poder alcanzar los objetivos planteados era necesario implementar nuevas funcionalidades como integrar pasarela de pago, autogestión de cuentas, etc.
Se comenzó realizando una refactorización del proyecto que cumpliera con las "best practices" de django sugeridas en el libro "Two Scoops of Django" [1], pero la lógica de la aplicación continuaba demasiado acoplada al framework y la implementación de nuevas funcionalidades seguía siendo lenta.

Fue entonces que se decidió comenzar un proyecto de cero, separando la aplicación en 3 capas: Frontend, Backend, Datos y utilizando tecnologías que permitieran acelerar el desarrollo sin sacrificar una buena arquitectura.

Esta estrategia resultó ser la más apropiada para el proyecto dado que al utilizar un lenguaje más familiar, hizo que el desarrollo del nuevo producto sea más robusto, mejor estructurado y más divertido de implementar.

1.4 Planificación del Trabajo

Se trabajará bajo los principios de metodologías ágiles, trabajando en sprints de 2 semanas. Dado que el tiempo real de trabajo son 3 meses, (Marzo - Mayo) se plantearon 3 milestones mensuales en los que se dedicarán 2 sprints a cada uno.
Al final de cada mes idealmente se podrá realizar una demo del milestone logrado.

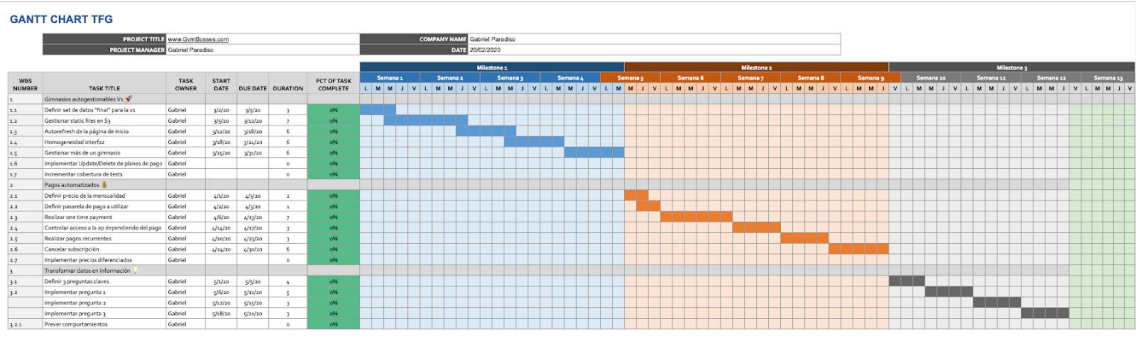

Figura 1. Gantt Chart

Como se puede apreciar en el Gantt chart todos los milestones tienen tareas sin tiempo asignado, esto definido por el MoSCoW son tareas que están dentro de la categoría **COULD HAVE** y que se implementarán en caso de disponer del tiempo. Se deja la última semana libre para poder pulir, terminar imprevistos o trabajar en tareas que no tenían tiempo asignado

1.5 Evaluación de riesgos

El milestone número 2 presenta el mayor nivel de incertidumbre dada la falta de experiencia trabajando con pasarelas de pagos. Para mitigar este riesgo las estimaciones en los milestone 1 y 3 se espera que tomen menos tiempo del asignado con el fin de empezar antes o tener tiempo extra al final.

Dada la poca experiencia en el área de diseño de producto y frontend, la tarea de mejora de interfaz presenta también un posible riesgo.

Resistencia al cambio por parte del cliente. El personal del gimnasio que utilizan la aplicación día a día pueden presentar cierta fricción al tener que utilizar un software nuevo. Para mitigar este riesgo se diseña la interfaz pensando en el usuario final y en facilitar su trabajo.

1.6 Breve sumario de productos obtenidos

Tal como fue planteado en la planificación se lograron los siguientes objetivos:

Gimnasios autogestionables V1 🚀:
Los gimnasios no necesitan del administrador de la plataforma para la creación de cuentas, gestión de planes de pagos, etc.

Pagos automatizados 💰:
Se logró integrar pagos recurrentes con Stripe, lo que permite cobrar automáticamente a los gimnasios que utilicen la aplicación. Esto a su vez permite expandir a más clientes que quieran utilizar la aplicación sin incrementar el tiempo de mantenimiento.

Transformar datos en Información 💡:
Se lograron responder las siguientes preguntas claves a la hora de gestionar un gimnasio:

- ¿Cuántos clientes tenemos?, ¿estamos creciendo o tenemos menos clientes?, ¿Hay un patrón con las suscripciones de mis clientes?.
- ¿Cuánto dinero ingresa? ¿Ingresa más dinero o menos dinero que el mes/año pasado?.
- ¿Cuántas personas han ingresado en la última hora? ¿Cuántas personas aproximadamente hay en la sala actualmente?

1.7 Breve descripción de los otros capítulos de la memoria

En los siguientes capítulos se presentan las tecnologías utilizadas, la arquitectura de la aplicación, el diseño de la base de datos y se explica en detalle las tareas realizadas.

2. Tecnologías utilizadas

El backend consta actualmente de un único servicio escrito en Golang.
Golang o simplemente GO tal como lo define la documentación oficial [2] y wikipedia [3] es un Lenguaje de programación open source, compilado y concurrente diseñado en google por Robert Griesemer, Rob Pike y Ken Thompson en 2007 y lanzado en 2009. El objetivo de google era poder crear un lenguaje que trajera las ventajas de lenguajes dinámicos como python pero con la performance de C o C++.
Es por esto que sus características principales son:

- **Performance:** La performance se compara con C o C++ dado que su compilador inicialmente construido en C y luego en GO, transforma el código a código máquina.
- **Tipado estático:** Si bien el tipado es estático, también introduce algunos conceptos de lenguajes dinámicos, en javascript por ejemplo es posible declarar una variable sin especificar su tipo y luego cambiar su tipo. Esto puede resultar problemático cuando tu aplicación comienza a crecer y resultar en comportamientos erróneos o inesperados. En Go es posible crear una variable sin especificar su tipo ej:

  uoc := *“Universidad UOC”*

  En este ejemplo declaramos la variable ouc sin especificar su tipo, pero el compilador al ver el valor asume que es un string, pero a diferencia de los lenguajes dinámicos no se puede cambiar el tipo una vez la variable es creada. Es por esto que es estático con algunos conceptos de dinámico.
- **Simple de escribir y de leer:** Este es uno de sus puntos más fuertes, se trata de un lenguaje simple, conciso y con una comunidad que refuerza el uso de buenas prácticas.
- **Concurrente:** parte de la performance también viene de que es un lenguaje concurrente, lo que permite ejecutar varias tareas al mismo tiempo en un mismo procesador. Cabe destacar que tal como lo explica Rob Pike en su conferencia “Concurrency Is Not Parallelism” [4] el concepto de concurrencia es distinto al de paralelismo. Para entender este concepto veremos un claro ejemplo planteado por EDteam en su video “¿Que es Go (Golang)?” [5] donde cada tarea es representada por un gopher (mascota de GO), cada procesador es representado por una pista y el objetivo final es cruzar la meta.

  En un lenguaje en el que las tareas se ejecutan de forma serial, es necesario esperar a que una tarea termine para iniciar la siguiente tarea, en este caso que el gopher llegue a la meta para poder empezar el siguiente gopher.

Figura 2. Ejecución serial

En un lenguaje en el que las tareas se ejecutan de forma paralela, hay varios gophers que se encuentran corriendo por diferentes pistas al mismo tiempo, pero de haber más gophers esperando, deben esperar a que los actuales crucen la meta para comenzar.

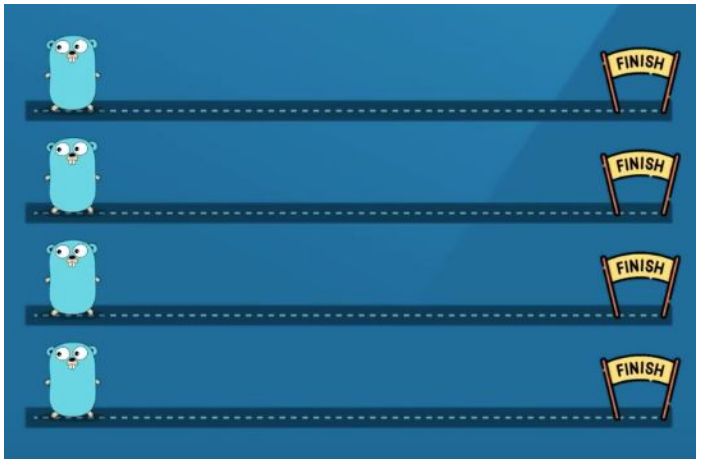

Figura 3. Ejecución paralela

En un lenguaje concurrente como GO, la diferencia es que no es necesario esperar a que la tarea termine, sino que el procesador va ejecutando de a trozos cada tarea dando la sensación de que fueran en paralelo, pero sobre un mismo procesador. En el caso de los gophers estos comenzarán antes de que el de adelante llegue a la meta.

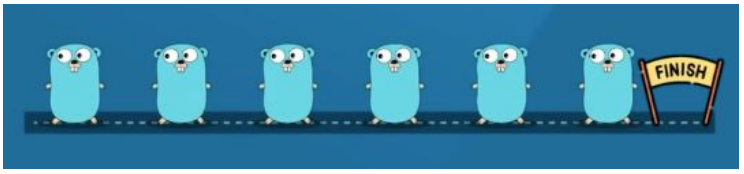

Figura 4. Ejecución concurrente

Ahora sí como la mayoría de los ordenadores poseen varios procesadores, esto significa que que hay gophers corriendo de forma concurrente en cada procesador y al mismo tiempo de forma paralela entre procesadores.

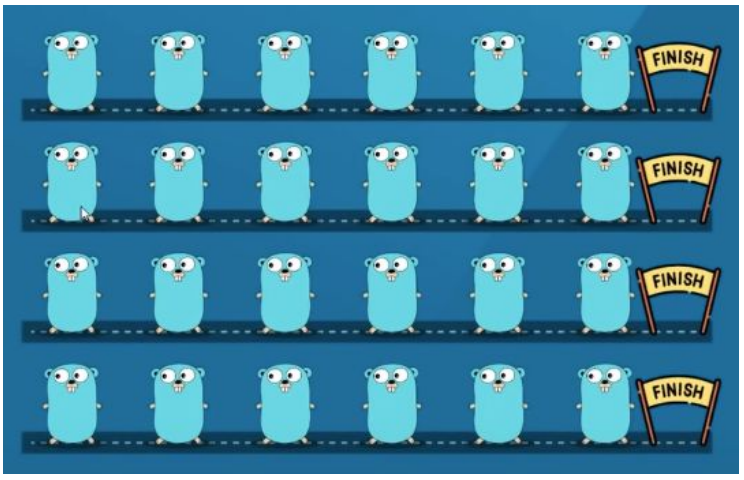

Figura 5. Ejecución concurrente en paralelo

Esto impacta directamente la performance dado que se pueden ejecutar más tareas, en menos tiempo aprovechando mejor los recursos.

- **MultiParadigma:** Permite el paradigma de programación orientada a objetos pero no tiene herencia de tipos ni palabras claves. Esto resulta en una forma familiar de programar, sin caer en las típicas malas prácticas de herencia. A pesar de no ser un lenguaje funcional, también permite aplicar conceptos de programación funcional o otros paradigmas.
- **Librería estándar**. La ausencia de popularidad de frameworks en Golang es en gran parte debido a que la librería estándar es tanto completa como performante, reduciendo la necesidad de importar paquetes externos, o uso de frameworks. Si bien existen frameworks y librerías externas, por lo general se intenta que estas sean lo más minimalistas posible y apegarse a la librería estándar por defecto.

El motivo por el cual elegí este lenguaje es porque además de ajustarse perfectamente al caso de uso, he trabajado con él desde hace 4 años, lo que permite que la velocidad de desarrollo sea superior a utilizar un lenguaje/framework desconocido.

React + Redux

El frontend se encuentra desarrollado en Javascript utilizando ReactJS y Redux.
React es como lo define wikipedia [6] una librería de JS open source creada por Facebook en 2013 con el fin de crear interfaces de usuario y se ha convertido en el estándar en Single Page Applications.
La idea y necesidad detrás de la creación de react es poder dividir la interfaz de usuario en varios componentes independientes que puedan ser re-utilizados y gestionen su propio estado. Al tener su propio estado encapsulado, React

mediante el uso de su propio DOM virtual actualiza sólo los componentes que han cambiado su estado, sin necesidad de actualizar todos los componentes.
Tal como lo indica Tutorialspoint [7] las características principales de React son:

- **Virtual DOM:** Tal como se mencionó para incrementar la performance, react utiliza un DOM virtual que le permite comparar de forma más rápida que utilizando el DOM normal si un componente ha cambiado o no.
- **Propiedades:** Los componentes tienen props o propiedades que son atributos de configuración del componente, con esto se puede generalizar un componente y dependiendo de la propiedad pasada por parametro desde un componente padre, podemos alterar el comportamiento del componente hijo. Ejemplo planteado en la web oficial de react [8]:

```
class HelloMessage extends React.Component {
  render() {
    return (
      <div>
        Hola {this.props.name}
      </div>
    );
  }
}
```

Figura 6. Ejemplo componente React

En este caso el componente “HelloMessage” está recibiendo el nombre de la persona que debe saludar mediante la propiedad “name”
Si al utilizar este componente lo declaramos como:

<HelloMessage name=”UOC” />
El componente al ejecutarse mostrará “Hola UOC”.

Redux tal como se indica en wikipedia [9] es también una librería de JS que se utiliza para almacenar el estado de la aplicación, creado por Dan Abramov y Andrew Clark en 2015 inspirados en FLUX.
La idea detrás de Redux es tener un store que almacena el estado global de la aplicación, y los componentes (en nuestro caso de react) obtienen acceso a ese store mediante acciones. Si bien los componentes tienen su propio estado, vimos que la interfaz es un conjunto de componentes y compartir estado entre componentes es complicado, especialmente si son componentes hermanos en donde no es posible pasar la información como si tuvieran una relación padre-hijo, es ahí donde entra Redux. En el tutorial de la web oficial de Redux [10] se definen como los componentes principales:

- **Actions:** Las acciones son eventos y son la única forma de enviar información desde la aplicación al store.
- **Reducers:** Los reducers son funciones puras que en respuesta a una acción, toman el estado actual de la aplicación y retornan el nuevo estado.

- **Store:** El store como vimos es el encargado de almacenar el estado general de la aplicación.

En cuanto al diseño de la UI se utilizó Material-UI, un framework de UI basado en Material-Design que permite utilizar componentes con un diseño atractivo para el usuario. Se pueden construir sistemas de diseño sobre material-UI y el beneficio principal es el desarrollo rápido.

La decisión de utilizar esta combinación de tecnologías se basó en la comunidad detrás de React + Redux, la posibilidad de crear aplicaciones móviles reutilizando componentes con React Native y la curiosidad de aprender un nuevo framework.

A nivel de base de datos se utilizó PostgreSQL, uno de los sistemas de gestión de bases de datos relacional más utilizados, acorde a su documentación oficial [11] es conocido principalmente por su arquitectura, fiabilidad, integridad de datos, set de funcionalidades, así como por su comunidad. Su desarrollo comenzó en 1986 en la universidad de California en Berkeley bajo el proyecto POSTGRES y fue lanzado en 1996.

Las características principales de PostgreSQL son:

- **Gratis y de código abierto:** Posiblemente una de las características por las cuales es tán popular y de acuerdo al ranking de DB-Engines [12] ocupa el puesto número 4

Rank			DBMS	Database Model	Score		
Jun 2020	May 2020	Jun 2019			Jun 2020	May 2020	Jun 2019
1.	1.	1.	Oracle	Relational, Multi-model	1343.59	-1.85	+44.37
2.	2.	2.	MySQL	Relational, Multi-model	1277.89	-4.75	+54.26
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	1067.31	-10.99	-20.45
4.	4.	4.	PostgreSQL	Relational, Multi-model	522.99	+8.19	+46.36

Figura 7. Ranking uso bases de dato

- **Bajo mantenimiento:** Dado que tiene más de 30 años de desarrollo activo, se trata de un sistema maduro y muy estable, por lo que el tiempo dedicado a su mantenimiento es muy bajo.
- **Cumple con los requisitos ACID:** Desde 2001 PostgreSQL cumple con los requisitos ACID (Atomicidad, Consistencia, Aislamiento y Durabilidad) características que determinan la robustez de un SGBD, asegurando que los datos no serán corrompidos y que perduraran en el tiempo.
- **Multiplataforma:** Se puede utilizar en la mayoría de sistemas operativos (Linux, macOS, BSD, Windows, Solaris).
- **Estándar SQL:** Cumple en su mayoría el estándar SQL, por lo que la sintaxis es familiar.

La decisión de utilizar PostgreSQL se fundamenta principalmente por la confianza y conocimiento de este SGBD. Otro punto por el cual se decidió utilizar PostgreSQL es la facilidad de integración con GO y de crear en Heroku.

Para el almacenamiento de las imágenes de los clientes se utiliza Amazon Simple Storage Service o simplemente AWS S3, un servicio de almacenamiento de objetos en la nube.

S3 ofrece a través de una API o varios SDK acceso web a un almacenamiento de objetos que apunta a proveer escalabilidad, alta disponibilidad (entre 99.95% to 99.99%) y baja latencia. El diseño de este servicio de almacenamiento no es de conocimiento público, por lo que no se sabe exactamente cómo es su arquitectura. La organización de su almacenamiento se basa en “buckets” (o baldes en castellano) donde cada archivo tiene una key única. Para subir un archivo de hasta 5 terabytes se puede utilizar la REST API, cualquiera de sus SDK o su interfaz gráfica. Para acceder a los archivos se puede descargar utilizando su API, SDK, interfaz gráfica o el protocolo BitTorrent.

Algunas de las alternativas a S3 son Azure Blob storage, Google cloud storage, minIO. La decisión de utilizar S3 se basa en que posee un SDK para Golang simple y potente, así como la intención de en un futuro mover todos los servicios al ecosistema AWS.

Para la autenticación de clientes se utiliza Auth0, un servicio de Autenticación que permite simplificar el control de acceso a la app. Auth0 se encarga de almacenar los usuarios y contraseñas, cuando un usuario se logea a la aplicación, Auth0 válida usuario y password y retorna un JWT token que incluye el email del usuario de forma encriptada. Como alternativa hay otros sistemas de autenticación como OKTA o simplemente se podría haber almacenado usuario y contraseña encriptada en nuestra base de datos. Controlar los usuarios y contraseñas es una opción que compromete la seguridad y la información a la que se puede acceder es bastante sensible. Se decidió utilizó Auth0 sobre otras opciones por el hecho de que su documentación es muy buena y tienen un plan gratis de hasta 7000 usuarios activos.

Para el control de pagos se utiliza Stripe, una pasarela de pago que gestiona suscripciones de forma simple y segura. En la sección de objetivos del trabajo, en el objetivo 2: Pagos automatizados, se detallan las características, cuáles eran las alternativas y porqué se seleccionó esta.

En cuanto a la infraestructura, actualmente el backend y la base de datos se encuentran desplegados en Heroku.

Heroku tal como lo define Wikipedia [13] es una plataforma como servicio de computación en la nube. La idea detrás de heroku como lo mencionan en su web [14], es permitir que el desarrollador se enfoque en la aplicación sin tener

que pensar en las complicaciones de desplegar, monitorizar, logs, escalar instancias, etc.

Las características principales de heroku son:

- **Elasticidad y crecimiento:** Se puede cambiar en cualquier momento la cantidad de Dynos asignados. Los Dynos son la unidad de poder de computación, están basados en contenedores de Linux y podemos relacionarlos con "instancias" de un servicio. Hay distintos tipos de Dynos, comenzando por el plan gratuito que ofrece unidades de procesamiento más que suficientes para probar un MVP.
- **Routing y balanceo de carga:** Internamente heroku sabe cómo manejar la carga y distribuirla entre los diferentes dynos.
- **Resiliencia:** En caso de que uno de los dynos muera, heroku sabrá crear un nuevo dyno que sustituya el fallido.
- **Facilidad de despliegue:** mediante el uso de un archivo .procfile que es simplemente un archivo de texto indicando qué servicios necesitan ser ejecutados.
- **Extensiones:** Cuenta con muchos "Add-ons" que permiten extender la funcionalidad, facilitando el uso de bases de datos, recolección de métricas, logging, etc.

Dado su facilidad a la hora de desplegar aplicaciones en Go y de crear la base de datos PostgreSQL además de su plan gratuito inicial, se decidió utilizar Heroku en vez de sus posibles alternativas: AWS o Google cloud platform. Posiblemente en un futuro cuando sea necesario tener un control más detallado del backend, este sea trasladado a AWS.

El frontend se encuentra desplegado en AWS Amplify console un servicio de hosting y despliegue continuo especialmente para clientes web y mobile o serverless backends. Amplify console forma parte del framework AWS Amplify que intenta resolver problemas comunes a la hora de desarrollar una nueva aplicación proporcionando servicios como: despliegue continuo, datastores, serverless, servicios de machine learning, etc.

Las características de AWS Amplify Console que motivaron a utilizarlo:

- **Despliegue simple:** el despliegue continuo se realiza conectando el repositorio git con Amplify, este detecta cuando hay cambios en el repositorio y mediante un archivo de configuración le podemos indicar como hacer build y desplegar nuestra aplicación. Es un equivalente de lo que logramos con heroku pero en el frontend.
- **Bajo costo:** consta de un plan gratuito limitado por la cantidad de despliegues, espacio utilizado y tráfico que luego pasa a pagarse según uso.
- **Certificados SSL:** ofrece certificados SSL de forma "gratuita" permitiendo utilizar HTTPS necesario para realizar pagos con Stripe.
- **End-to-end Testing:** ofrece simples integraciones de end-to-end con Cypress.

Cypress.io es un framework opensource para javascript que permite desarrollar tests automatizados End-To-End. Hasta el momento, Selenium ha sido la opción por defecto a la hora de realizar tests automatizados, las principales características y razones por las cuales se decidió utilizar Cypress en lugar de Selenium son:

- **Instalación sin configuración:** La instalación es simple y rápida sin necesidad de realizar configuraciones. Al ser una librería de javascript las dependencias se almacenan en el package-lock.json
- **Simple de usar:** La librería es intuitiva y crear tests es rápido y simple, permitiendo poner en práctica metodologías como TDD o BDD.
- **Integración con AWS Amplify Console:** Amplify Console ofrece facilidades para integrar tests realizados con cypress al pipeline de despliegue.
- **Espera automática:** Cypress maneja las esperas de forma automática, este es uno de los problemas más grandes al desarrollar tests en Selenium, generando tests inconsistentes.

3. Arquitectura de la aplicación

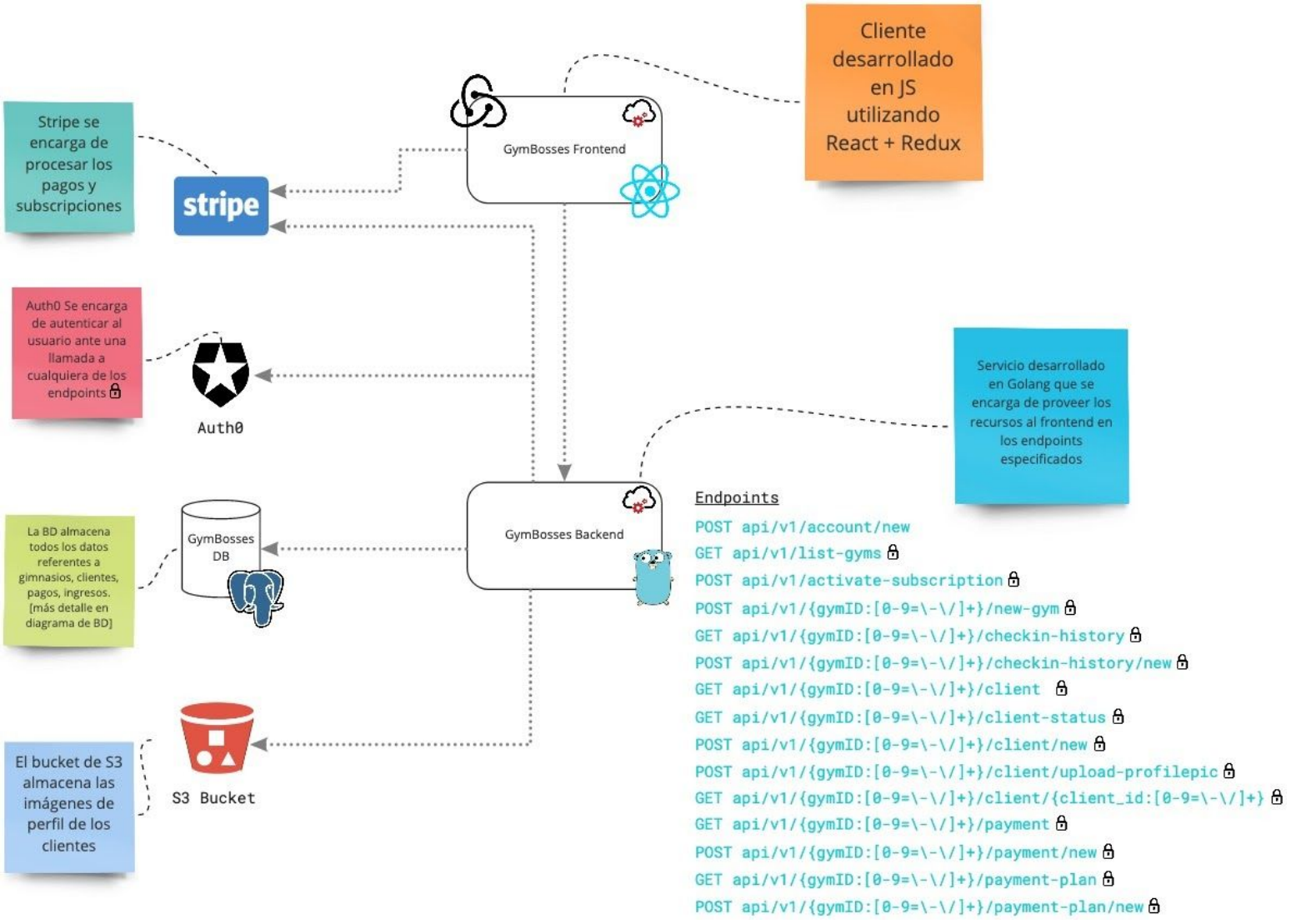

Figura 8. Arquitectura de la aplicación

Se plantea una arquitectura en capas [15] de 3 capas, separando claramente la capa de presentación (frontend), capa de negocio + persistencia (backend), y la capa de bases de datos. Esta arquitectura nos da la flexibilidad que se necesita, sin incrementar la complejidad innecesariamente para la etapa en la que se encuentra el proyecto.

El Backend a pesar de ser un único servicio, se encuentra estructurado siguiendo el concepto de domain driven design [16], en el que cada unidad de negocio está separada en diferentes paquetes. La ventaja de estructurar el código de esta forma, desde el punto de vista de arquitectura, es que llegado el momento en el que sea necesario crecer, moverse a una arquitectura de microservicios sería relativamente simple.

Para garantizar la seguridad de la información, los endpoints que contienen el icono 🔒 en el diagrama indican que necesitan de un JWT token [17] que identifique que es un usuario con permisos para obtener la información que solicita. Este token es generado por Auth0 cuando hacemos login en la aplicación y dentro tiene codificado el correo del cliente, por lo tanto en nuestra base de datos solo tenemos que almacenar la correspondencia entre el email y los gimnasios a los que tiene acceso, evitando tener que almacenar usuarios y contraseñas.

El frontend por su parte intenta ser una capa fina que simplemente despliega información, dado que el backend realiza las transformaciones de datos necesarias y entrega la información lista para ser renderizada.

Tanto el código del backend como el del frontend se encuentran en un mono repositorio de acceso privado en gitlab: https://gitlab.com/gaboparadiso/gymbosses.

4. Estrategia de Testing

La estrategia de tests se basa en el principio de “La pirámide de testing” la cual generalmente se estructura en 3 niveles de tests: Unitarios, integración, aceptación.
En nuestro caso solo implementamos los niveles de Unitario y Aceptación dado que la cobertura proporcionada por estos niveles es suficiente ya que el único cliente del frontend es el navegador. Los tests unitarios se realizaron utilizando la librería estándar de Go y librerías para generar mocks de dependencias externas como base de datos.
En el caso de los test de aceptación se definió la estrategia, tecnología a utilizar y se implementaron los primeros tests de aceptación utilizando el framework Cypress.io. Este nivel aún necesita agregar más cobertura y agregar al pipeline de despliegue en AWS Amplify.

5. Diseño de la base de datos

Figura 9. Diseño de la base de datos

En cuanto a el diseño de la base de datos, tal como se ve en el diagrama entidad relación hay 2 tipos de tablas, las que se crean a nivel global y las que se crean por gimnasio (nombre-tabla_GymID). La razón de esta decisión es la clara separación de la información por gimnasio que tiene como beneficios:

- Simplificar el control de acceso a información (se otorgan permisos por tablas)
- El control de crecimiento de id de cada tabla, permitiendo hacer “horizontal partitioning” de la forma más natural posible. Por ejemplo: la

tabla que almacena cada vez que un cliente ingresa al gimnasio crece considerablemente más que el resto de las tablas, si el ingreso de todos los gimnasios se almacena en una sola tabla, en poco tiempo se agotaran los índices.

- En caso de que algún gimnasio deje de utilizar la aplicación es simple remover toda su información. (cumpliendo normativas GDPR)

6. Descripción de tareas llevadas a cabo

6.1 Resumen

Acorde a la planificación, se realizaron todas las tareas a las cuales se les había asignado tiempo y se intentó cubrir de forma secundaría la tarea “Incrementar la cobertura de test” a pesar de que era una tarea calificada como **COULD HAVE** y no tenía tiempo asignado.
Tal como estaba previsto en los riesgos, la tarea relacionada a la mejora de la interfaz fue la que más tiempo llevó, pero se pudo terminar en tiempo y forma.

Milestone: Gimnasios autogestionables V1 🚀

Definir set de datos “final” para la v1	Gabriel	3/2/20	3/5/20	3	100%
Gestionar static files en S3	Gabriel	3/5/20	3/12/20	7	100%
Autorefresh de la página de inicio	Gabriel	3/12/20	3/18/20	6	100%
Homogeneidad interfaz	Gabriel	3/18/20	3/24/20	6	100%
Gestionar más de un gimnasio	Gabriel	3/25/20	3/31/20	6	100%
Implementar Update/Delete de planes de pago	Gabriel	Sin tiempo asignado		0	0%
Incrementar cobertura de tests	Gabriel	Sin tiempo asignado		0	20%

Milestone: Pagos automatizados 💰

Pagos automatizados 💰					
Definir precio de la mensualidad	Gabriel	4/1/20	4/3/20	2	100%
Definir pasarela de pago a utilizar	Gabriel	4/2/20	4/3/20	1	100%
Realizar one time payment	Gabriel	4/6/20	4/13/20	7	100%
Controlar acceso dependiendo del pago	Gabriel	4/14/20	4/17/20	3	100%
Realizar pagos recurrentes	Gabriel	4/20/20	4/23/20	3	100%
Cancelar subscripción	Gabriel	4/24/20	4/30/20	6	100%
Implementar precios diferenciados	Gabriel			0	0%

Milestone:Transformar datos en Información 💡

Transformar datos en Información 💡					
Definir 3 preguntas claves	Gabriel	5/1/20	5/5/20	4	100%
Implementar pregunta 1	Gabriel	5/6/20	5/11/20	5	100%
Implementar pregunta 2	Gabriel	5/12/20	5/15/20	3	100%
Implementar pregunta 3	Gabriel	5/18/20	5/21/20	3	100%
Prever comportamientos	Gabriel			0	0%

6.2 Análisis detallado

Gestionar static files en S3

Esta tarea se enfoca en almacenar y desplegar correctamente la imagen de perfil de cada cliente de los gimnasios. Para realizarla se analizaron 2 alternativas posibles:

- Gestionar la subida de archivos desde el Frontend y solo enviar la URL al backend para almacenarla en la bd.
- Gestionar la subida de archivos desde el Backend

La única ventaja de subir los archivos desde el frontend es la simplicidad y rapidez con la cual se puede gestionar, pero presenta desventajas a nivel de seguridad, ya que es más simple dar acceso a personas no deseadas a obtener las imágenes o a utilizar nuestro bucket como almacenamiento en la nube.

Luego de analizar las opciones se decidió ir por la segunda e implementar un endpoint en el backend al que se puede acceder solo si se tiene un JWT correcto. Este endpoint recibe la imagen, valida su tamaño, anonimiza el nombre del archivo y lo sube a AWS S3. El frontend por su parte realiza la funcionalidad de seleccionar la imagen, recortarla y controla también el peso de la imagen.

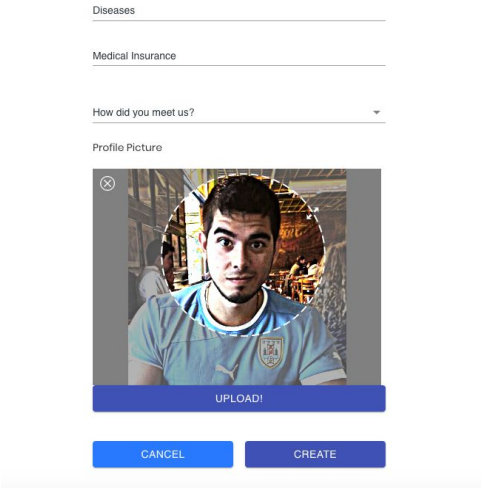

Figura 10. Subir imágen de perfil

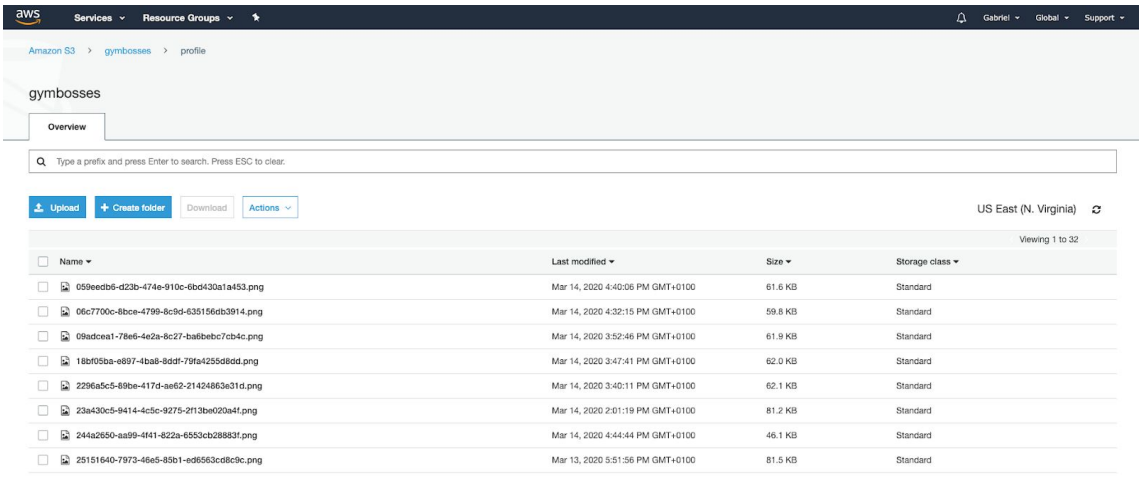

Figura 11. Bucket en AWS S3

Autorefresh de la página de inicio

Esta tarea apunta a actualizar el componente que muestra cuando un cliente ingresa al gimnasio, de forma que el administrador pueda controlar si ha pagado la cuota o no.

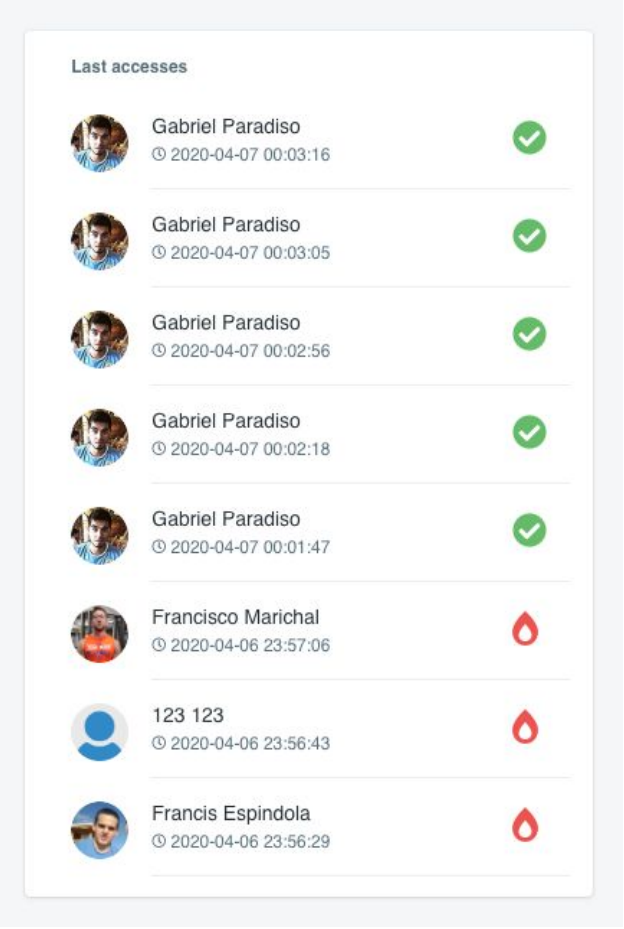

Figura 12. Componente de Historial de ingresos

Para realizarla se estudiaron las diferentes técnicas:

- Client polling (polling, long polling)
- Server pushing (web sockets)

Web sockets: La comunicación la inicia el servidor una vez que tiene información para enviarle al cliente. El problema con esta técnica es que el servidor tiene que ser capaz de reconocer cada cliente y poder gestionar la información como eventos para saber cuándo enviarle la información. La ventaja que presenta es que es lo más cercano a tiempo real, pero la complejidad introducida + complejidad de implementación no justifica el caso de uso.

Long polling: La comunicación la inicia el cliente y el servidor retiene la request hasta que tiene una respuesta para el cliente y se la envía. La ventaja es que es relativamente más simple de implementar que web-sockets, pero de todas formas se necesita modelar la información como eventos para saber cuando notificar al cliente si se quiere implementar correctamente. Por lo que de momento el beneficio no justifica el costo.

Polling: La comunicación la inicia el cliente realizando request al backend cada cierta cantidad de tiempo. La ventaja de esta técnica es la simplicidad al implementarlo, mientras que la clara desventaja es la cantidad de request innecesarias que se realizan al backend.
Dado que de momento la aplicación va a ser utilizado por solo 4 gimnasios, se decide ir por esta opción y realizar request cada 3 segundos, ya que el backend puede soportar x1000 esta carga.

Homogeneidad interfaz

Esta tarea apunta a mejorar la interfaz de forma que sea consistente, atractiva y fácil de utilizar. Si bien ya se estaba utilizando Material-UI [18] que es la implementación para react del ‘design system’ de google ‘Material Design’, la experiencia no era del todo placentera. Los componentes custom desarrollados no seguían ninguna guia en términos de: tipo de letra, tamaño de letra, colores, alineación, etc. Además de estos problemas generales se detectaron problemas en pantallas puntuales:

Pantalla de inicio

Solo aporta la información de los clientes que ingresan y contiene una gráfica que aún no tiene la funcionalidad implementada.
Solución:

- Se implementa el endpoint y componente que muestra los contadores de clientes activos, inactivos y deudores.
- Se implementan los endpoints y componente de gráfica para que muestre el flujo de clientes en una ventana de tiempo de 3 meses
- Se trabaja sobre el estado de los componentes para que muestren el nombre del gimnasio

Previo a cambios de diseño:

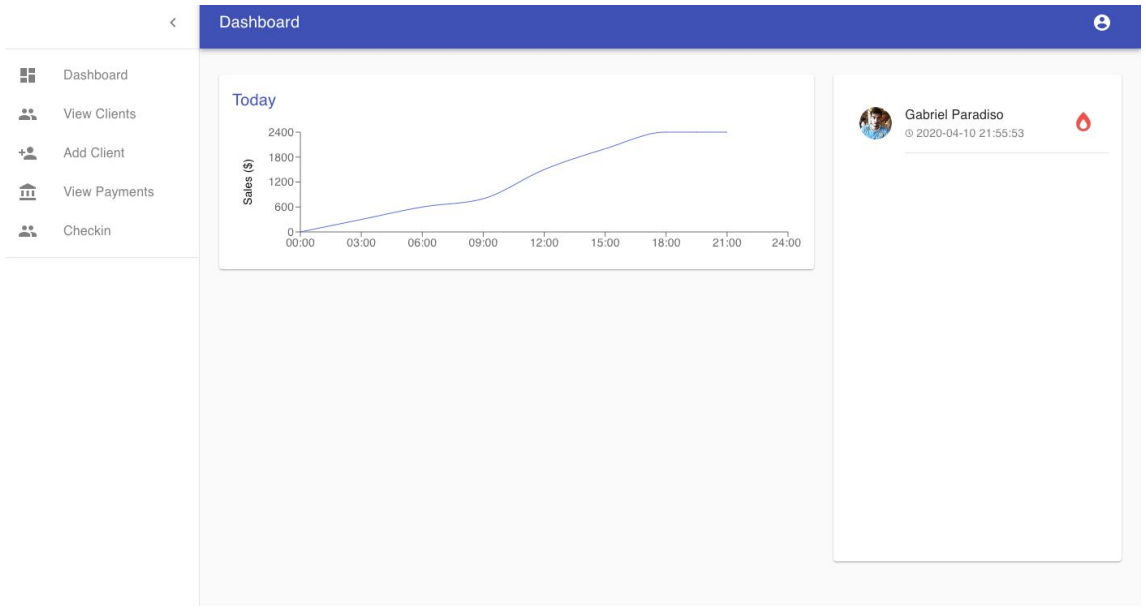

Figura 13. Dashboard previo cambios de diseño

Luego de cambios de diseño:

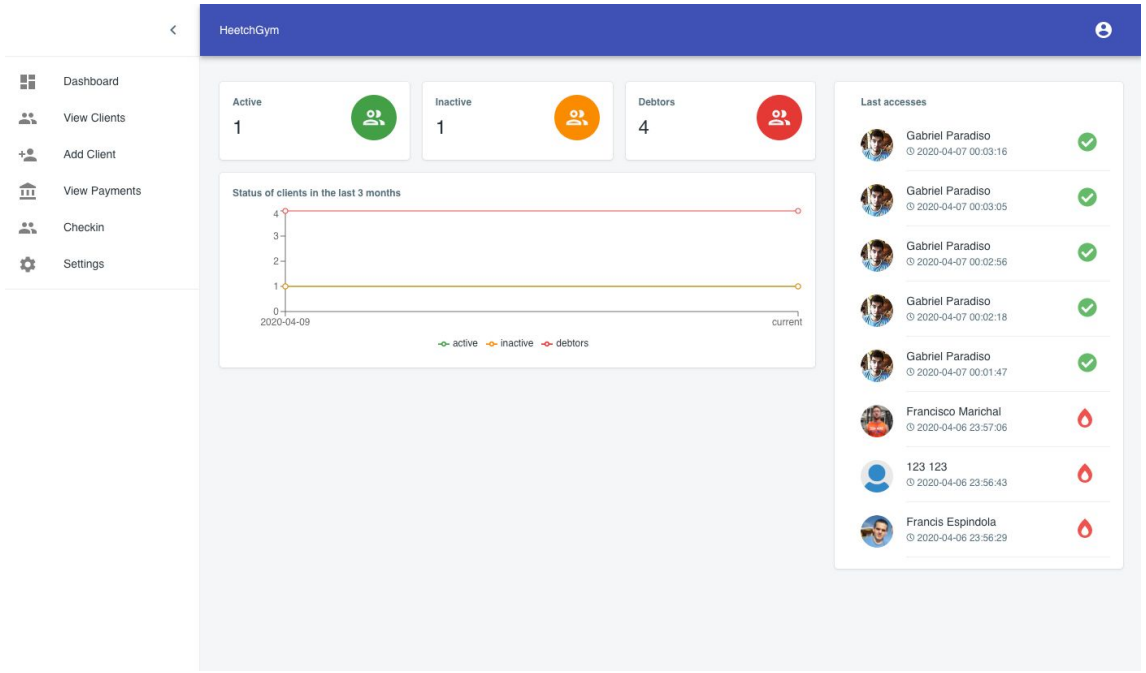

Figura 14. Dashboard luego de cambios de diseño

Pantalla Perfil del cliente

Esta pantalla no muestra toda la información y carece totalmente de estilos consistentes.

Solución:

- Se re-implementa el componente del header.
- Se redistribuye el orden de los componentes.
- Se le asignan tamaños fijos independientemente del contenido.

Previo a cambios de diseño:

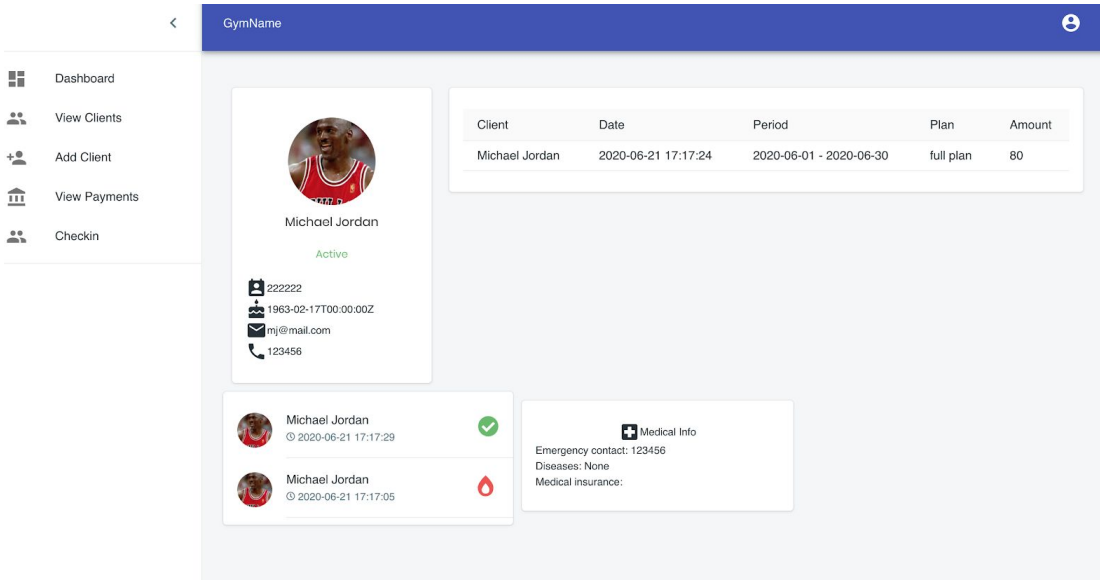

Figura 15. Perfil del cliente previo a cambios de diseño

Luego de cambios de diseño:

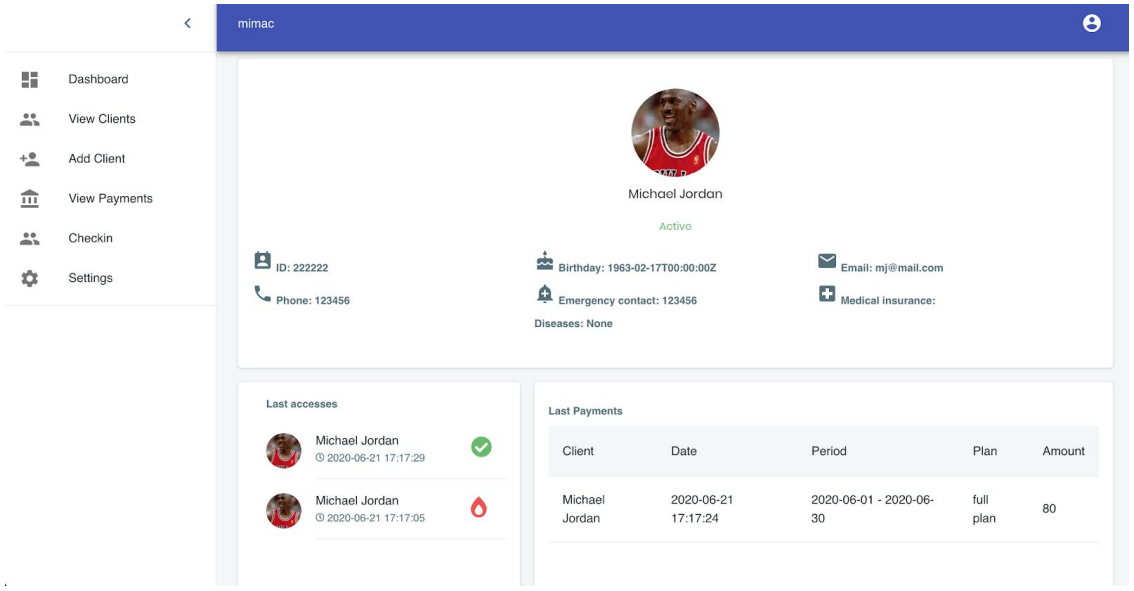

Figura 16. Perfil del cliente luego de cambios de diseño

Gestionar más de un gimnasio

Esta tarea apunta a poder satisfacer el caso de uso donde un cliente posee más de un gimnasio y quiere manejarlos con una única cuenta.
Solución:

- Se implementa el endpoint correspondiente que crea la asociación cuenta gimnasio requerida.
- Se implementa el componente ListGyms que aparece una vez el usuario se loguea en caso de tener más de un gimnasio, si solo tiene un gimnasio es re-dirigido al dashboard.
- Se implementa el componente para crear un nuevo gimnasio dentro de la nueva sección “Settings” que también reutiliza ListGyms.

Componente ListGyms:

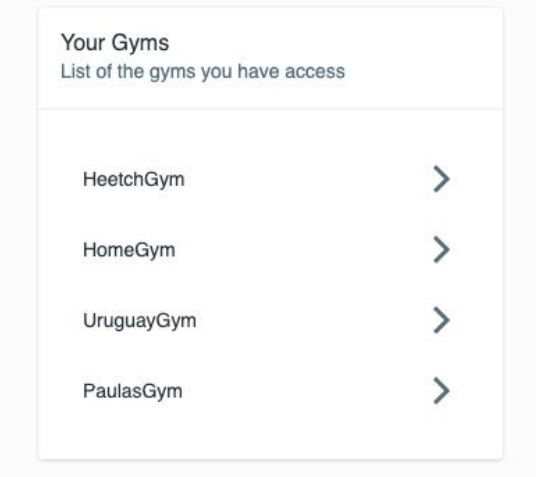

Figura 17. Componente ListGyms

Componente NewGym:

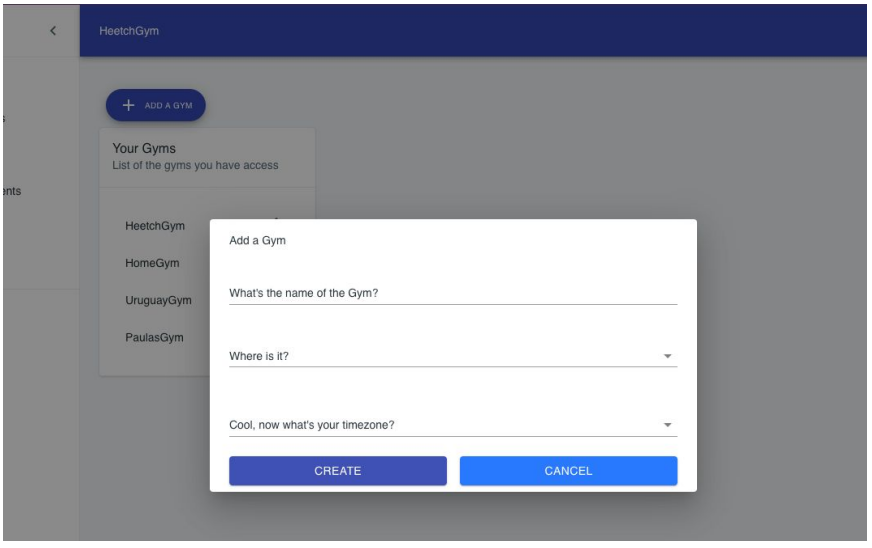

Figura 18. Componente NewGym

Incrementar cobertura de tests

Esta tarea apunta a incrementar la cobertura de tests, tal como se especifica en el apartado estrategia de test, se logra el objetivo de cubrir las funcionalidades principales a nivel unitario e iniciar a cubrir funcionalidades a nivel de aceptación.

Definir precio de la mensualidad

El precio se estima basado en los siguientes criterios:

- Precio de los competidores
- Capacidad de flexibilidad en precios
- Experiencia con cliente actual
- Costos de infraestructura

En cuanto a los precios de competidores, se evalúa GoTeamUP [19] y VirtuaGym [20].
VirtuaGym es el líder actual del mercado en gestión de gimnasios, si bien la lista de pricing no es pública, el precio del servicio ronda los 100 dólares por mes por gimnasio para manejar aprox 1000 clientes mensuales. Esta información se obtuvo del cliente actual ya que quiso implementar esta solución el año pasado y volvió a ClubAppUy luego de ver que era demasiado complejo de utilizar además de caro.
GoTeamUP ofrece diferentes precios de acuerdo a la cantidad de clientes que tiene el gimnasio.

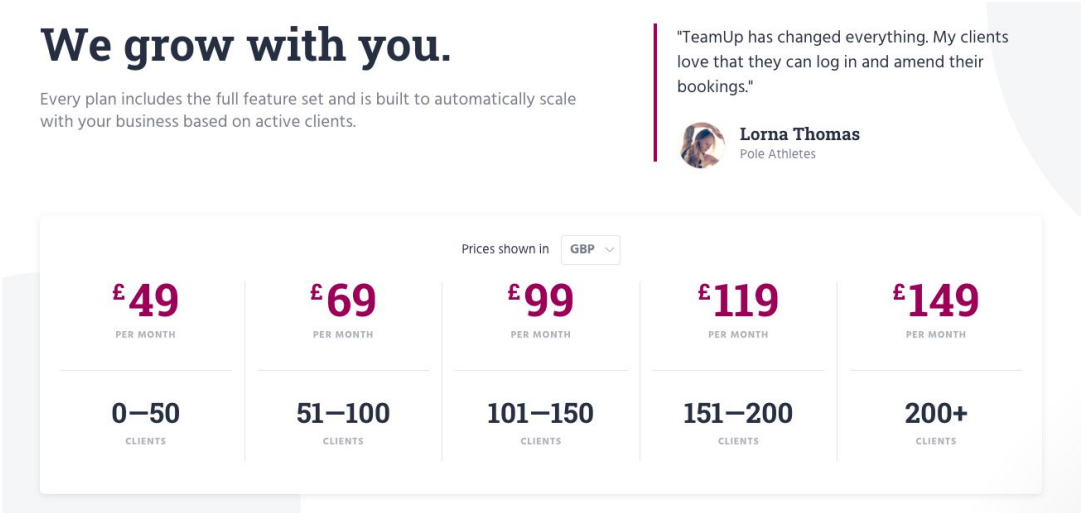

Figura 19. Precios GoTeamUP

Los costos actuales de infraestructura (Heroku + AWS) son inferiores a 20 dólares mensuales y la cotización actual de ClubAppUy es la siguiente:

Figura 20. Costos ClubAppUy

Conclusión

Se decide mantener el esquema de precios actual de ClubAppUy dado que:

- Cubre los costos de infraestructura con 1 solo cliente.
- Los precios son ⅓ del precio de sus competidores y el balance cantidad de features/precio lo justifica.
- Permite crecer según el tamaño del gimnasio.

Además de estos planes se agrega un plan de hasta 100 clientes para apuntar a gimnasios más pequeños o incluso personal trainers.

Planes:
Starter Hasta 100 clientes EUR 30
Basic Hasta 300 clientes EUR 55
Pro Hasta 600 clientes EUR 75
Unlimited EUR 90

Definir pasarela de pago a utilizar

Esta tarea apunta a definir mediante qué pasarela de pago se va a recibir los pagos de los dueños de los gimnasios. La idea final es realizar pagos recurrentes y que el acceso a la plataforma esté limitado por estos.

Se analizan las siguientes pasarelas de pago:

- Stripe
  Fundada en 2011 y tiene aprox 20% del mercado. El costo de utilizar el servicio es de 2.9% + $0.30 por transacción.
  Si bien la disponibilidad en 26 países es importante para muchos negocios, en nuestro caso no es relevante ya que solo recibiremos pagos en una cuenta española.
  En cuanto al soporte disponen de soporte técnico 24/7.
  Dispone de pagos recurrentes, funcionalidad que nos interesa para cobrar mensualmente a los clientes.
  La documentación es clara y parece no ser tan complicado implementarlo.

- PayPal
  Fundada en 1998, tiene cerca del 60% del mercado. El costo es de 2.9% + $0.30 por transacción.
  Tiene disponibilidad en 200 países, pero como se menciona anteriormente esto no es relevante para nosotros.
  También dispone de soporte técnico 24/7 por varios medios.
  Dispone de pagos recurrentes llamado “Suscripciones”, pero la documentación de esta funcionalidad no es del todo clara.

Conclusión

Dado que el costo es el mismo, se decide utilizar Stripe ya que la documentación es más clara y la implementación de suscripciones (pagos recurrentes) es más simple.

Realizar pagos recurrentes

Esta tarea agrega la funcionalidad de pagar por el servicio GymBosses de forma automática. Cubre la tarea “Realizar one time payment” que se planificó como un paso intermedio.
Los planes se crean desde la UI de stripe especificando nombre, descripción, precio y en caso de tener periodo de prueba la cantidad de días.

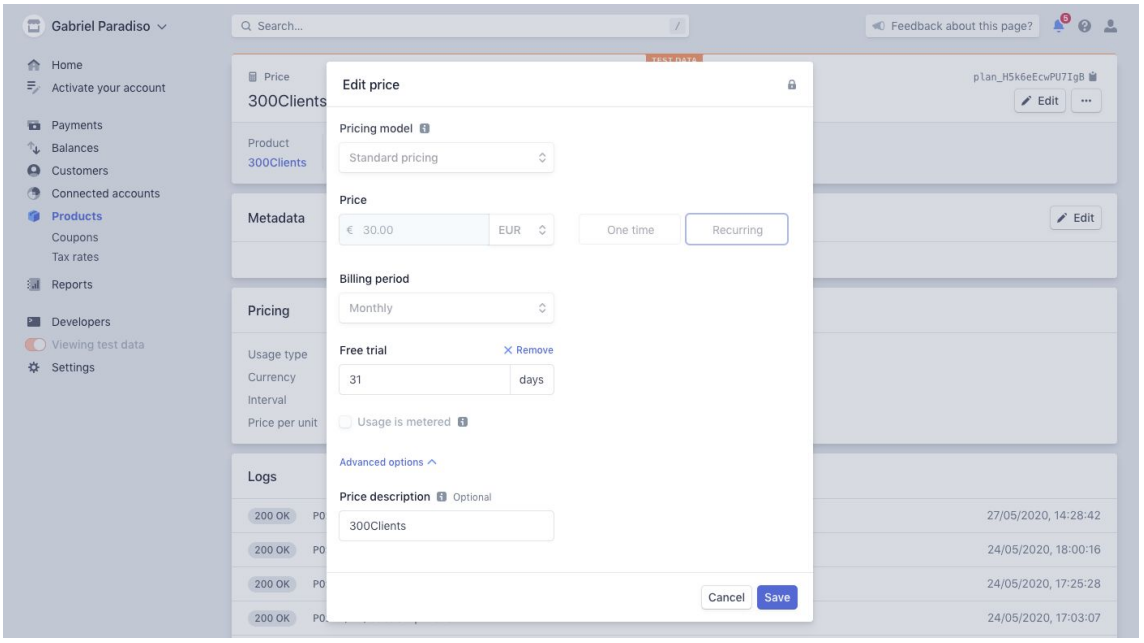

Figura 21. Creación de plan desde Stripe

Al momento de crear una cuenta en Gymbosses el backend crea a través de la API una cuenta en stripe y la suscripción al plan correspondiente.

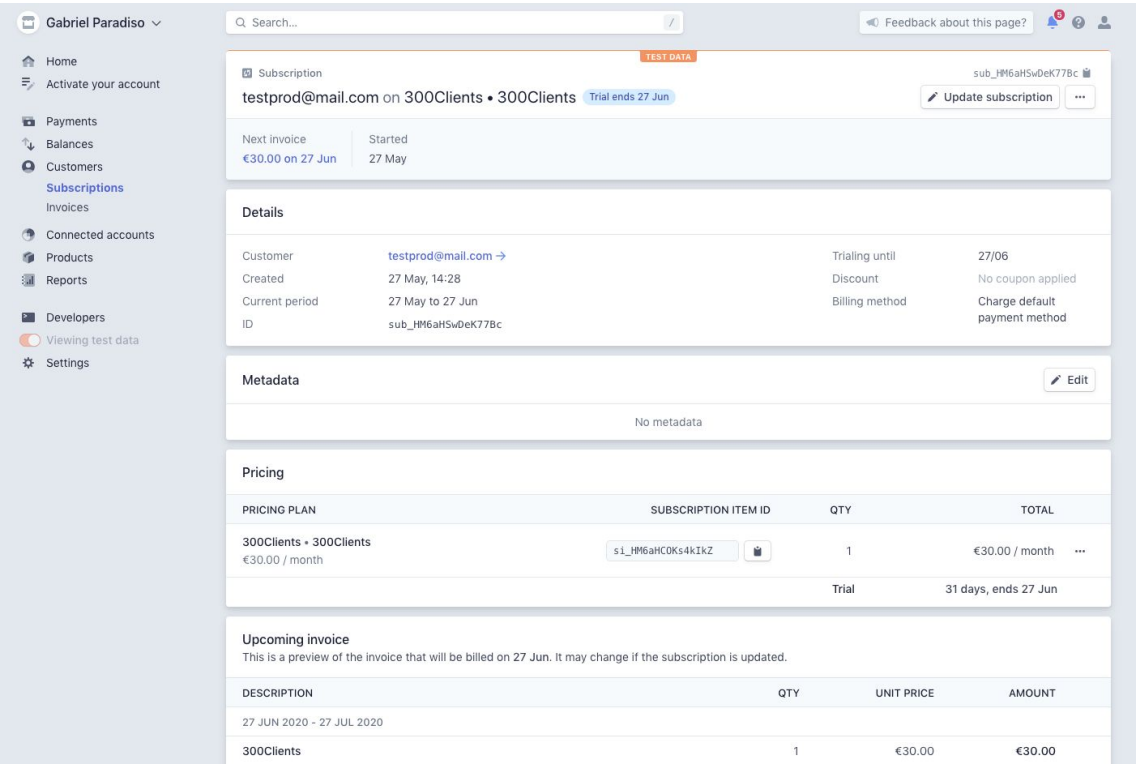

Figura 22. Suscripción en Stripe

Como se ve en la imagen anterior la suscripción se crea con el estado de “trialing” y es la mejor forma de que el cliente pueda probar la aplicación durante 1 mes sin necesidad de que ingrese su tarjeta, ya que ingresar la tarjeta sin conocer el producto genera una fricción que impacta drásticamente en la adquisición de nuevos usuarios.
Al finalizar el periodo de prueba Stripe intentará cobrar el primer mes, al no haber ingresado ninguna tarjeta el estado de la suscripción pasará a “Past due”. Una vez se ingrese una tarjeta válida Stripe intentará automáticamente cobrar la mensualidad y actualizará el estado. Lo más relevante de utilizar Stripe como pasarela de pago es que provee un cliente para react [21] que envía de forma segura los datos de pago a stripe desde el frontend y los datos de la tarjeta no pasan por el backend ni son almacenado en nuestra base de datos.

Actualmente el modo de prueba está activado en Stripe, lo que permite testear la integración utilizando la tarjeta de prueba número 4242 4242 4242 4242 sin incurrir en ningún costo. Una vez activo el modo “Live” Stripe requiere que la comunicación se haga bajo el protocolo HTTPS, es por eso que como parte de esta tarea se decidió desplegar el frontend en AWS Amplify [22] ya que ofrece certificados SSL “gratuitos”.

Controlar acceso dependiendo del pago

Para controlar el acceso a la aplicación, el backend consulta mediante la API de stripe el estado de la suscripción, en caso de que la misma sea distinta de “Active” o “Trialing” retorna al frontend un error 402 Payment Required. El frontend identifica este statuscode y redirecciona al usuario a la página www.gymbosses.com/subscription para que ingrese su tarjeta y se realice el pago.

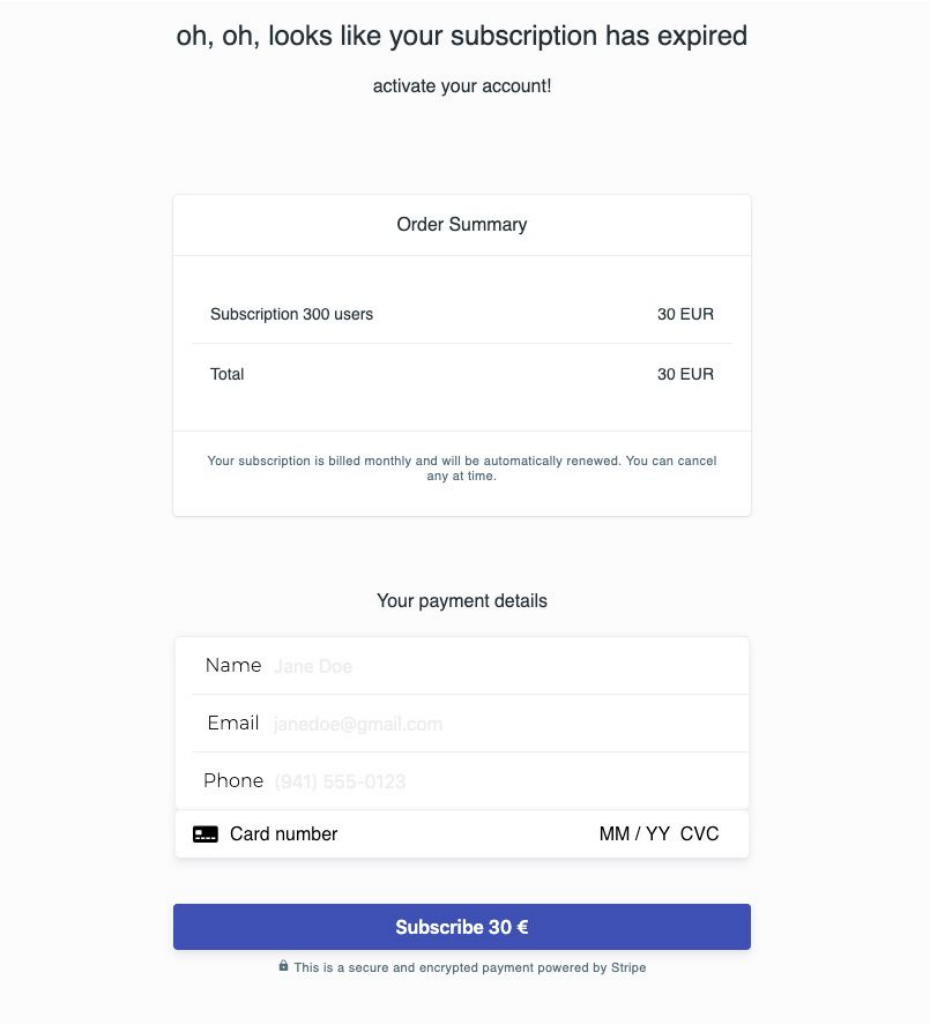

Figura 23. Ingresar detalles de pago

Una vez ingresa los datos de la tarjeta, stripe realizará el cobro y actualizará el estado a “active” por lo que redireccionamos al usuario al dashboard. Esta validación de status de la suscripción se realiza tanto al hacer login como en todos los endpoints que crean nueva información (nuevo cliente, nuevo pago, check in, etc) por lo que sin el pago correspondiente la aplicación es inutilizable.

Cancelar Suscripción

Esta tarea implementa la funcionalidad de cancelar la suscripción, para esto es necesario poder visualizar el estado y detalles de la misma, por lo que se crea el siguiente componente:

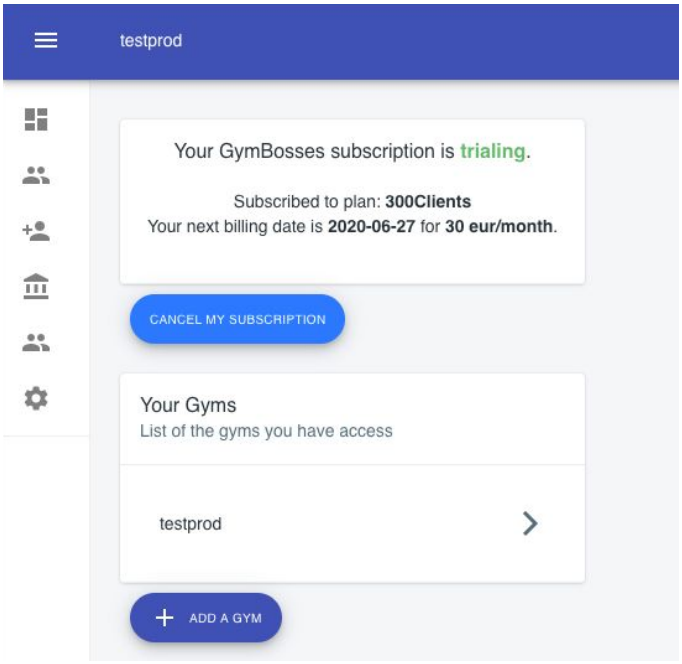

Figura 24. Cancelar suscripción

Mediante el botón de “Cancel my subscription” el usuario recibe una pantalla de confirmación y tiene la posibilidad de cancelar su suscripción al finalizar el periodo de prueba o el mes pago.

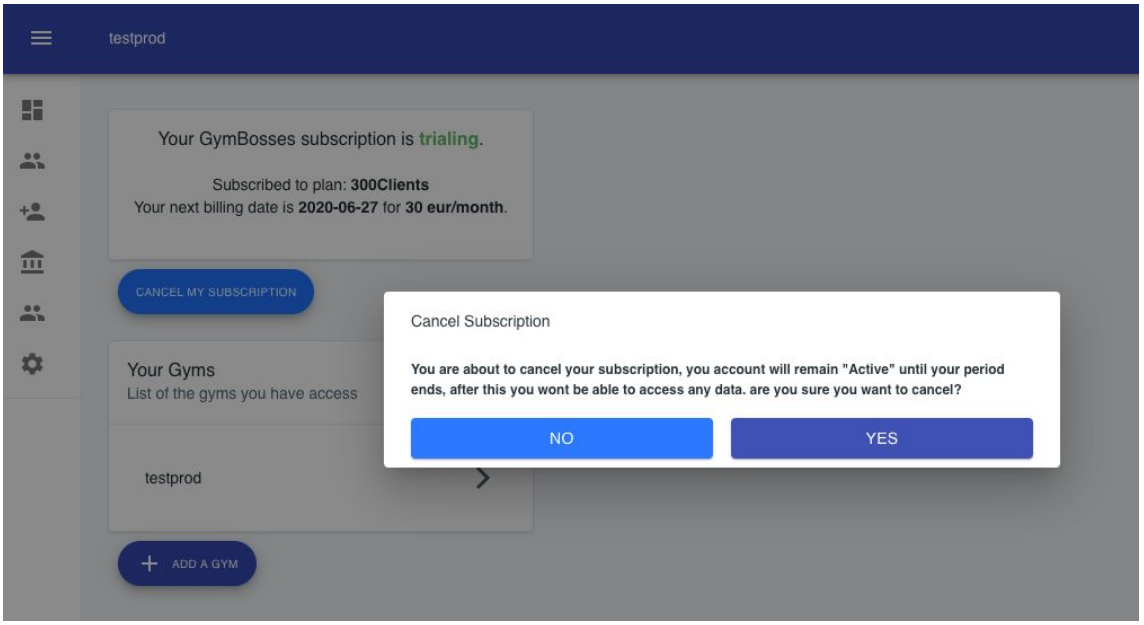

Figura 25. Confirmar cancelación de suscripción

Esto envía una request al backend para finalizar la suscripción y el backend es quien interactúa con la API de stripe.

Transformar datos en información

Se decide responder las siguientes preguntas:

- ¿Cuántos clientes tenemos?, ¿estamos creciendo o tenemos menos clientes?, ¿Hay un patrón con las suscripciones de mis clientes?.
- ¿Cuánto dinero ingresa? ¿Ingresa más dinero o menos dinero que el mes/año pasado?.
- ¿Cuántas personas han ingresado en la última hora? ¿Cuántas personas aproximadamente hay en la sala actualmente?

Para el primer grupo de preguntas se decide crear un endpoint que retorne el estado de los clientes en tiempo para que el frontend pueda consumir y mostrarlos en una gráfica.
Para lograr esto se crea una tarea que corre a diario, calcula la cantidad de clientes y los almacena en la tabla *client_status_statistics_X.*

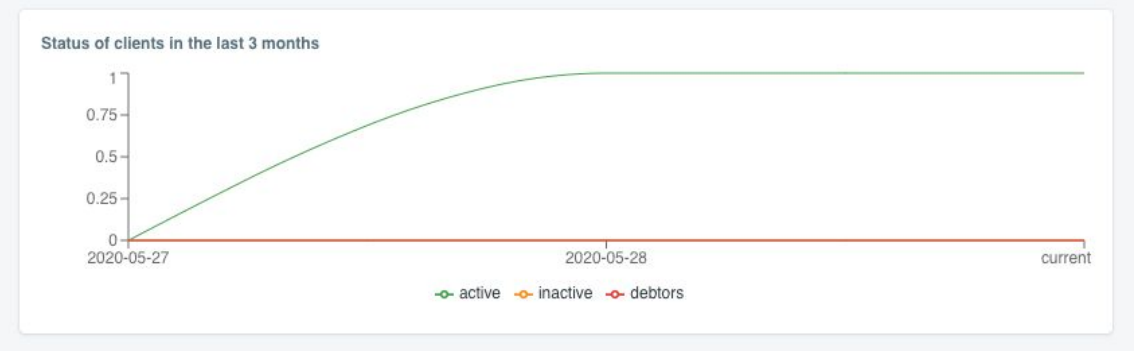

Figura 26. Estadística del estado de los clientes en los últimos 3 meses

Para responder el segundo grupo de preguntas la información ya se encuentra en el formato correcto, por lo que simplemente se crea un endpoint que en el backed realiza una SQL query agrupando por fechas y exporta el total de ingresos por mes al frontend que lo presenta como una gráfica.

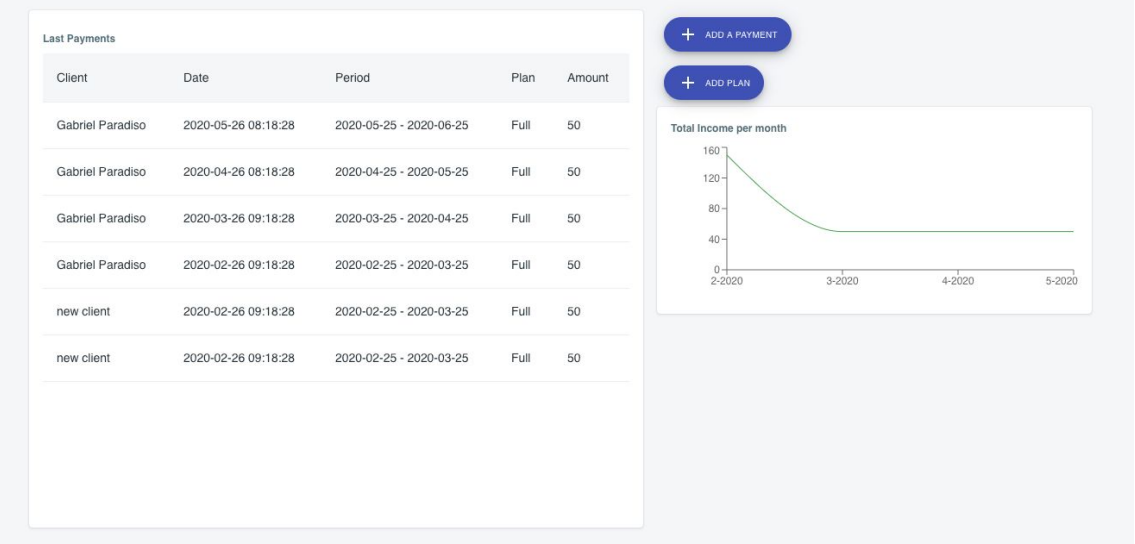

Figura 27. Estadística del ingreso de dinero agrupado por mes

El tercer grupo de preguntas cobró especial importancia a partir de la situación excepcional que se está viviendo. De momento en la mayor parte del mundo las actividades físicas en espacios cerrados no está permitida, pero se sabe que pronto se podrá retomar siguiendo pautas específicas en cuanto a la cantidad de personas que pueden ingresar al mismo tiempo. Para obtener el número exacto de personas en un momento específico es necesario cambiar el flow en la que los clientes interactúan, solicitando que hagan check-in y check-out. Es por eso que como alternativa se decide contar la cantidad de personas agrupadas por hora. Esto permite ver la distribución de ingresos al gimnasio en el tiempo y ofrece un estimado de las personas que actualmente se encuentran ahí, dado que en promedio una sesión de entrenamiento dura entre 40 minutos y 1 hora.

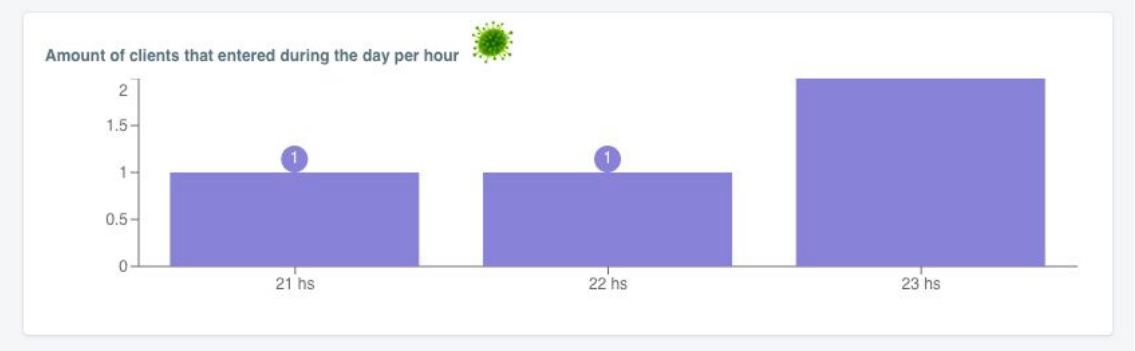

Figura 28. Estadística de ingresos de clientes al gimnasio por hora

7. Conclusiones

Anteriormente, Gymbosses carecía de funcionalidades que ClubAppUy ya proporcionaba, además, dependía en gran medida del administrador para permitir que los gimnasios operansen efectivamente, impidiendo ofrecer el servicio a una cartera más amplia de clientes.

Gracias a la ejecución de este proyecto, GymBosses ha logrado alcanzar la igualdad de funcionalidades con ClubAppUy permitiendo a los clientes utilizar la nueva versión. Así mismo, se logró reducir la inversión de tiempo de mantenimiento por parte del administrador y se incrementó la autonomía de los gimnasios al gestionar sus cuentas.

Por otra parte, el pago de la suscripción a la plataforma se automatizó, disminuyendo la fricción entre el administrador y el cliente (estableciendo fechas y ejecuciones de pago, así como la suspensión de suscripciones en caso de inclumpir con los mismos). Al reducir el trabajo manual de control de pagos, se hace posible captar más clientes.

GymBosses acumulaba datos que no estaban siendo aprovechados por el cliente, por lo que, se decidió transformar esos datos en información relevante para la toma de decisiones. Es así como, ahora se puede acceder a gráficas y estadísticas que permitan entender el comportamiento de los clientes y la salud financiera del gimnasio.

Los objetivos inicialmente planteados fueron alcanzados satisfactoriamente. Tanto la planificación como el modelo MoSCoW utilizado para priorizar las tareas, permitieron definir objetivos realistas para los tiempos pautados.

Durante la implementación de los pagos automatizados, se detectó un problema de planificación ya que no se tomó en cuenta el requisito de certificados SSL para aceptar pagos. Como consecuencia el tiempo estimado para esta tarea se extendió 3 días. Dada la flexibilidad del modelo de planificación utilizado, esto se pudo solventar sin inconvenientes.

En cuanto a lecciones aprendidas durante el TFG, definitivamente adquirí habilidades de frontend y de diseño intentando ofrecer al usuario una experiencia atractiva y de fácil uso. Se puso en práctica conocimientos adquiridos en las materias “Análisis y diseño con patrones” y “Sistemas distribuidos” a la hora de diseñar la arquitectura. Se obtuvo una planificación acertada basándose en experiencia profesional, experiencia académica durante el transcurso de la materia “Proyecto de desarrollo de software” y teórica obtenida en la materia “Gestión de proyectos”. También, se despertó la iniciativa de explotar de forma más eficiente los datos del gimnasio, proporcionando soporte a la toma de decisiones a partir de lo trabajado en la materia “Minería de datos”.

Como próximos objetivos del proyecto, se plantea:

- Incrementar la cobertura de tests a nivel de Aceptación e integrarlos al pipeline de despliegue.
- Migrar los datos de 1 de los gimnasios actuales y permitir al usuario que comience a utilizar la aplicación. A partir de aquí surgirán bugs y mejoras a implementar.
- Invertir tiempo en marketing para atraer nuevos clientes.
- Investigar cuales son las necesidades no cubiertas hasta el momento que mejoren la experiencia de los usuarios y plantear un roadmap nuevo.

8. Glosario

- **Administrador:** Hace referencia a el desarrollador de la plataforma que proporciona soporte a los clientes (Gimnasios).
- **Cliente:** Dependiendo del contexto se entiende por cliente los gimnasios que utilizan la plataforma, en caso de que se hable de los clientes del gimnasio se hace referencia a las personas que utilizan las facilidades del gimnasio.
- **ClubAppUy:** Nombre de la aplicación predecesora de GymBosses.
- **Control de Acceso:** Se refiere al acceso físico de los clientes del gimnasio a las facilidades del gimnasio.
- **Cuenta/s:** Hace referencia a la cuenta que se crean los gimnasios en la plataforma GymBosses.
- **Feature Parity:** Se denomina feature parity a una aplicación A que contiene todas las funcionalidades que una aplicación B.
- **Gimnasios:** Centro de entrenamiento en general, aplica para gimnasios convencionales, como de disciplinas específicas.
- **Gopher:** Mascota del lenguaje de programación Golang.
- **GymBosses:** Nombre de la aplicación de gestión de gimnasios www.gymbosses.com
- **Planes de pago:** Representan los planes que cada gimnasio ofrece a sus clientes. Ejemplo: “Pase libre”, “Clases de MuayThai”, etc.
- **Stack:** hace referencia al conjunto de tecnologías utilizadas.
- **Suscripciones:** Hace referencia a la suscripción que tienen los gimnasios con la plataforma GymBosses.

9. Bibliografía

[1] Two Scoops of Django. Daniel Roy Greenfeld, Audrey Roy Greenfeld, Audrey Roy. Two Scoops Press. Edición 3, ilustrada 2015.

[2] Documentación oficial de go. golang.org. [En línea] Junio 2020. https://golang.org/doc/

[3] Go. Wikipedia. [En línea] Junio 2020. https://es.wikipedia.org/wiki/Go_(lenguaje_de_programaci%C3%B3n)

[4] Concurrency Is Not Parallelism. Rob Pike. [En línea] Junio 2020. https://www.youtube.com/watch?v=cN_DpYBzKso

[5] ¿Que es Go (Golang)?. EDteam. [En línea] Junio 2020. https://www.youtube.com/watch?v=mXpZDQ2Au8U

[6] React. Wikipedia. [En línea] Junio 2020. https://es.wikipedia.org/wiki/React

[7] ReactJS - Overview. Tutorialspoint. [En línea] Junio 2020. https://www.tutorialspoint.com/reactjs/reactjs_overview.htm

[8] React. reactjs.org. [En línea] Junio 2020. https://es.reactjs.org/

[9] Redux. Wikipedia. [En línea] Junio 2020. https://es.wikipedia.org/wiki/Redux_(JavaScript)

[10] Redux basic tutorial. redux.js.org . [En línea] Junio 2020. https://redux.js.org/basics/basic-tutorial

[11] Sobre PostgreSQL. postgresql.org . [En línea] Junio 2020. https://www.postgresql.org/about/

[12] DB-Engines Ranking. DB-Engines. [En línea] Junio 2020. https://db-engines.com/en/ranking

[13] Heroku. Wikipedia. [En línea] Junio 2020. https://es.wikipedia.org/wiki/Heroku

[14] What is Heroku. Heroku. [En línea] Junio 2020. https://www.heroku.com/what

[15] Layered Architecture. O'reilly. [En línea] Junio 2020. https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html

[16] Domain-Driven Design: Tackling Complexity in the Heart of Software. Eric Evans. Addison-Wesley Professional. August 2003.

[17] Json Web Token. Wikipedia. [En línea] Junio 2020. https://en.wikipedia.org/wiki/JSON_Web_Token

[18] Material-UI. Material-UI. [En línea] Junio 2020. https://material-ui.com

[19] GoTeamUP Pricing. GoTeamUp. [En línea] Junio 2020. https://goteamup.com/en-gb/pricing

[20] VirtuaGym Pricing. VirtuaGym. [En línea] Junio 2020. https://business.virtuagym.com/gym-software

[21] Cliente Stripe React. Stripe. [En línea] Junio 2020. https://stripe.com/docs/stripe-js/react

[22] AWS Amplify. AWS. [En línea] Junio 2020. https://aws.amazon.com/es/amplify