UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Programa de Apoyo al Egreso de Profesionales en Actividad

TÍTULO: Plataforma de desarrollo y publicación de aplicativos para análisis de datos sobre salud

AUTOR: Jorge Octavio Condomí

DIRECTOR ACADÉMICO: Prof. Lic. Sebastián Dapoto

DIRECTOR PROFESIONAL: Lic. Nicolas García

CARRERA: Licenciatura en informática

Resumen

Gracias a los avances tecnológicos, el acceso a internet y la adopción masiva de dispositivos móviles se han empujado los límites ya establecidos sobre cómo las personas (pacientes) acceden a sus propios datos de salud y bienestar. La base de datos que posee la empresa de salud para la cual se realizará el desarrollo tiene un gran volumen de datos estadísticos provenientes de encuestas y estudios clínicos. Se analiza y documenta la creación de una herramienta web que permita la creación facilitada de aplicativos que tengan acceso a esos datos almacenados para poder ofrecer al usuario nuevas maneras de visualizar su información. Se analizan las características y particularidades de un proyecto de este tipo alcanzado por la normativa que protege los datos de salud HIPAA.

Palabras Clave

Widgets, HIPAA, Código como servicio, IDE, MySQL, SQLite, MeteorJs, Tiempo real, ReactJs, React Native, Babel, Sass.

Conclusiones

Se concluye que implementar una plataforma de desarrollo enriquecida completamente orientada para facilitar el acceso a información confidencial es posible. Se deja constancia sobre las limitaciones técnicas encontradas durante el proyecto, y cómo el equipo ha encontrado soluciones para poder sortear estos obstáculos siempre bajo un marco regido por las reglas impuestas por la normativa de HIPAA, y teniendo en cuenta las decisiones de negocio que la empresa ha tomado en pos de lograr un producto comercialmente atractivo.

Trabajos Realizados

Se realizó un desarrollo web basado en MeteorJs y ReactJs. Se definió e implementó un sistema de creación y publicación de aplicaciones donde usuarios editores programan mediante el uso de un IDE web. Se definió un mecanismo para facilitar el acceso a información de salud protegida por la ley HIPAA y que estas aplicaciones pueda hacer uso de esa información de manera aislada y protegida. También se definió una aplicación móvil en React Native que muestra estas aplicaciones.

Trabajos Futuros

Dentro de los trabajos propuestos se debe hablar sobre actualizaciones de versiones de librerías y frameworks utilizados, la implementación de un IDE web mas robusto y la modificación de los estilos para que la plataforma de creación sea accesible mediante móvil. También documentar cómo fue desarrollado el cliente móvil para iOS detallando particularidades de la plataforma. Adema se puede extender el trabajo actual programando una aplicación nueva utilizando el IDE web desarrollado en esta tesina y documentando el proceso desde el punto de vista del usuario editor.

Fecha de la presentación: Diciembre 2022

TESINA DE LICENCIATURA

Plataforma de desarrollo y publicación de aplicativos para análisis de datos sobre salud

Facultad de informática
Universidad Nacional de La Plata

Jorge Otavio Condomí
Licenciatura en informática
7454/2

Director Académico: Prof. Lic. Sebastián Dapoto
Director Profesional: Lic. Nicolas García
Diciembre 2022

Índice general

1. Introducción **1**

2. Objetivo **3**

3. Contexto y regulaciones **4**
3.1. Contexto 4
3.1.1. Científicos ciudadanos 4
3.1.2. El teléfono celular como herramienta de diagnóstico . . 5
3.1.2.1. Hardware 5
3.1.2.2. Ubicuidad 6
3.1.2.3. Ecosistemas de aplicaciones 6
3.1.3. La empresa, el equipo e invitados 7
3.1.3.1. Pruebas de usabilidad y feedback interno . . 8
3.2. HIPAA 9
3.2.1. ¿Que es PHI? 9
3.2.2. HIPAA en el desarrollo de software 10
3.2.2.1. Políticas de Privacidad 11
3.2.2.2. Políticas de Seguridad 11
3.2.2.3. Políticas de Notificación de incidente 13

4. Trabajos relacionados **14**
4.1. CodeSandbox 15
4.1.1. CodeSandbox - SandPack 15

5. Análisis de tecnologías **17**
5.1. Tecnologías de front-end 17
5.1.1. ReactJs 17
5.1.1.1. Desarrollo guiado por componentes 18
5.1.1.2. Lodash 19
5.1.1.3. MomentJs 21
5.2. Tecnologías para el desarrollo móvil 21

5.2.1. React Native 21
5.3. Lenguaje 23
5.3.1. JavaScript 23
5.4. Tecnologías de backend 24
5.4.1. Node.Js 24
5.4.2. Express 25
5.4.3. MeteorJs 26
5.4.3.1. Colecciones 27
5.4.3.2. Publicación y suscripción 27
5.4.3.3. Paquetes MeteorJs 28
5.4.3.4. Métodos Meteor 29
5.4.4. Babel 29
5.4.5. Bases de datos 31
5.4.5.1. MySQL 31
5.4.5.2. MongoDB 32
5.4.5.3. SQLite 33
5.5. Desarrollo y características del *IDE Web* 36
5.5.1. Requerimientos 36
5.5.1.1. Actores 36
5.5.1.2. Lista de casos de uso iniciales 37
5.6. Prueba de concepto inicial sobre la implementación 38
5.6.1. Código como servicio 38
5.6.2. Previsualización inmediata del widget 39
5.6.3. Consultas SQL dentro de los widgets 40
5.6.3.1. Segmentación y aislamiento de información del usuario final 41
5.7. Arquitectura del sistema e interconexiones 43
5.7.1. Recursos compartidos: Base de Datos MySQL 43
5.7.2. Servicios internos: Login Service 44
5.7.3. Servicios internos: CMS Service 44
5.7.4. Login y manejo de sesión 47
5.7.5. Ciclo de vida de un Widget 50
5.7.5.1. Composición y anidamiento 50
5.7.5.2. Posibles estados 51
5.7.5.3. Pantalla de inicio 52
5.7.5.4. Creación y edición 53
5.7.5.5. Características heredadas 55
5.8. Desarrollo de un *widget* 58
5.8.1. Estilos y su procesamiento 60
5.8.1.1. Previsualización y renderizado 61
5.8.2. Usando datos de prueba 61

5.9. Consultas a la base de datos 62
5.9.1. Creación e inicialización de la base de datos 62
5.9.1.1. Precarga en el cliente 64
5.9.1.2. Precarga en el servidor 65
5.9.1.3. Inicialización de la base de datos SQLite . . . 66
5.9.1.4. Tabla metadata 67
5.9.2. Precarga de tablas 68
5.9.2.1. Versionado de tablas 68
5.9.2.2. Funciones de precarga para cada tabla 69
5.9.3. Limitaciones de la implementación 72
5.10. Desarrollos secundarios 73
5.10.1. Plataforma móvil 73
5.10.1.1. React Native y la estrategia general 74
5.10.1.2. Webview 74
5.10.1.3. Interceptando eventos 75

6. Conclusiones y trabajos futuros **76**
6.1. Conclusiones 76
6.2. Trabajos futuros 77

Glosario **79**

Índice de figuras **85**

Bibliografía **87**

Capítulo 1

Introducción

En los últimos años, gracias a los avances tecnológicos, el acceso a internet y la adopción masiva de dispositivos móviles, la industria ha sufrido cambios que marcan un nuevo horizonte donde se ubica al usuario en el epicentro del ecosistema dándole poder de decisión y poder de información. En los sistemas relacionados con la salud, esta situación ha empujado los límites ya establecidos sobre como las personas - *pacientes* - acceden a sus datos de salud y bienestar.

En el caso particular que se estudia en este trabajo, la base de datos que posee la empresa de salud para la cual se realizará el desarrollo tiene un conjunto de información muy amplio y rico en datos estadísticos provenientes de encuestas y estudios clínicos realizados por diversas entidades de investigación. Sin embargo, gran parte de esa información no se está utilizando en este momento.

Actualmente, la web de la empresa muestra varias visualizaciones e interpretaciones de la información del usuario como tablas, gráficos complejos y distintas representaciones, muchas veces basadas únicamente en datos de un usuario, y otras realizando una comparación de algún aspecto en particular contra información agregada de distintos universos.

Con el objetivo de hacer un mejor uso del volumen de información con el que se cuenta, el equipo de *Investigación y Desarrollo* ha generado una gran cantidad de ideas. Estas ideas proponen productos completamente nuevos o productos que plantean una nueva manera de presentación de la información, para hacer más fácil y sencilla su comprensión. Una vez finalizados, estos desarrollos deberían ser probados por usuarios reales. Sin embargo, el costo y tiempo de desarrollo que implican estos proyectos ha sido un obstáculo.

En el pasado, el departamento de software ha llevado a cabo pruebas de concepto con ilustraciones y prototipos no funcionales pero el *feedback* resultante de un pequeño grupo de usuarios de prueba o voluntarios no siempre suele coincidir con el éxito o fracaso del producto probado. Además, estas pruebas son costosas, implican una gran inversión de tiempo y necesitan de un considerable número de personas para poder realizarse.

Por todo lo expuesto, la empresa se vio impulsada a buscar una forma alternativa de llevar a cabo estas ideas y convertirlas en algo que esté al alcance de los usuarios en el menor tiempo posible. Así es que surge la idea de contar con una plataforma de desarrollo ágil que provea un gran nivel de abstracción de todas las cuestiones de desarrollo como la configuración, la conexión a la base de datos, manejo de usuarios, los servidores y despliegues, entre otros.

En un principio, la plataforma fue pensada sólo para uso interno, pero luego surgió la idea de nutrirla con herramientas para hacer más fácil el proceso de desarrollo, la publicación y la distribución del software resultante. Llevando a cabo esta idea, es posible dar acceso a investigadores y desarrolladores externos para que hagan uso de los datos, y a la vez puedan aumentar el valor de esa información mediante nuevas integraciones o aportando ideas innovadoras a la hora de manipular los datos preexistentes. Esto, a su vez, podría generar la apertura de nuevos frentes de comercialización y flujos de ingresos para la empresa.

Además, una cuestión importante a tener en cuenta es que debido a la naturaleza de los datos y a las regulaciones de privacidad existentes (HIPAA), el desarrollo de la plataforma tiene particularidades especiales al momento de utilizar y dar acceso a la información personal de los usuarios finales.

Capítulo 2

Objetivo

Se plantea como objetivo principal de esta tesina documentar el desarrollo de una plataforma de programación web, en adelante *IDE Web*, que permita a programadores externos a la empresa programar aplicativos con fácil acceso a diferentes recursos como bases de datos, información de prueba, documentación y otras herramientas de desarrollo. Estos aplicativos, luego de ser auditados, se publicarán para que usuarios finales puedan utilizarlos en la plataforma web de la empresa o mediante la aplicación móvil existente.

Se verá en detalle las particularidades que resultaron desafiantes de este proyecto y el contexto, especialmente se hará foco en el *IDE Web* y se mencionarán aspectos generales de la aplicación móvil que se creó simultáneamente. El desarrollo de estos aplicativos o *widgets*, si bien serán creados usando *IDE Web*, queda por fuera del alcance de este documento.

Capítulo 3

Contexto y regulaciones

3.1. Contexto

La tesina se realiza en un contexto definido por la industria de salud y la tecnología. Si bien el desarrollo especifico no está vinculado estrictamente a la salud todo el contexto subyacente de la motivación y el objetivo final sí lo están.

3.1.1. Científicos ciudadanos

El termino *ciencia ciudadana*[1] se refiere a cualquier investigación o experimento científico que cuenta con la intervención de personal no especializado junto con científicos y profesionales.

Especialmente los entusiastas del bienestar e interesados en conocer su propio cuerpo y cómo mejorar algún aspecto particular de éste, son los que promueven y empujan a las empresas que ofrecen y/o facilitan el acceso a este tipo de información. Estas empresas suelen guiar y advertir a los pacientes/usuarios para que los datos provistos no se malinterpreten, dado que la información de salud en el ámbito de ciencia ciudadana no debe ser tomada como recomendación médica.

3.1.2. El teléfono celular como herramienta de diagnóstico

Los teléfonos celulares inteligentes han facilitado y potenciado la participación de científicos ciudadanos en innumerables iniciativas[1], como por ejemplo exploración espacial, monitoreo de aves e incluso obtención de información de salud.[2]

Los teléfonos celulares inteligentes poseen tres características que los convierten en una interesante herramienta de evaluación y diagnostico sobre salud y bienestar[2] :

- *Hardware*
- *Ubicuidad*
- *Ecosistemas de aplicaciones*

3.1.2.1. Hardware

Actualmente existen miles de modelos y muchas marcas de fabricantes lo cual hace imposible establecer una especificación de hardware que represente a la población total. Es posible tomar como punto de referencia a las dos grandes empresas fabricantes que lideran el mercado como **Apple** y **Samsung**, con un *market share* reportado de 22 % y 18 % respectivamente en el ultimo trimestre del 2021 [3]. Sus modelos de teléfono mas populares tienen procesadores capaces de realizar un promedio de 22 billones ($22 * 10^{12}$) de operaciones por segundo [4] lo cual lo ponen a la altura de computadoras personales.

Otro aspecto a destacar sobre los teléfono inteligentes actuales es su capacidad de conectividad y la amplia variedad de opciones que ofrecen. Por ejemplo, un *iPhone 13* lista las siguientes capacidades inalámbricas dentro de sus especificaciones[3]:

---

[1] People-powered research [Investigación impulsada por personas] `https://www.zooniverse.org/`

[2] 'Citizen science' [Ciencia ciudadana], Wikipedia `https://en.wikipedia.org/wiki/Citizen_science#Smartphone`

[3] 'iPhone 13', Apple `https://www.apple.com/iphone-13/specs/`

- *5G (sub-6 GHz and mmWave)*
- *Gigabit LTE with 4x4 MIMO and LAA7*
- *Wi-Fi 6 (802.11ax) with 2x2 MIMO*
- *Bluetooth 5.0 wireless technology*
- *Ultra Wideband chip for spatial awareness8*
- *NFC with reader mode*
- *Express Cards with power reserve*

Este completo repertorio de ondas de radio permiten al *smartphone* conectarse a una amplia variedad de otros dispositivos, ya sean *access points*, parlantes *bluethooth* o lectores de NFC para hacer un pago sin contacto. Pero lo realmente interesante en el marco de salud y bienestar son los dispositivos que, mediante la conexión con un *smartphone* permiten obtener datos sobre el usuario y/o su condición física.

3.1.2.2. Ubicuidad

Los *smartphones* son el dispositivo móvil por excelencia. se estima que más de un 83 % de la población mundial posee uno actualmente[4].

3.1.2.3. Ecosistemas de aplicaciones

Gran parte del éxito de los teléfonos inteligentes se debe a la capacidad de personalizar las capacidades del dispositivo usando aplicaciones, en adelante *Apps*, descargadas mediante los distintos canales de distribución que cada sistema operativo posee, siendo en el caso puntual de *iOS* el *Appstore* y en los dispositivos *Android*, *Google Play*. No es un detalle menor que este sistema de distribución produzca un margen de ganancia tan masivo que en 2021, combinando las dos plataformas mencionadas llegó a ser de USD $113000M[5].

Cada uno de los sistemas operativos anteriormente mencionados ponen a disposición de los desarrolladores interesados, una serie de APIs mediante

[4]'HOW MANY SMARTPHONES ARE IN THE WORLD?' [¿Cuantos celulares inteligentes hay en el mundo?], Bankmycell
`https://www.bankmycell.com/blog/how-many-phones-are-in-the-world`

[5]Mansoor Iqbal, 'App Revenue Data (2022)', Business Of Apps `https://www.businessofapps.com/data/app-revenues/#AppRevenues`

las cuales pueden hacer uso de las capacidades del hardware del dispositivo tales como la cámara, el micrófono, y la red celular, entre otros. También permiten el uso de servicios que abstraen al programador de tener que idear funcionalidades desde cero, servicios complejos como reconocimiento de voz, geolocalización, análisis de imágenes y reconocimiento facial. Estas plataformas de distribución ofrecen, muchas veces de manera gratuita, las aplicaciones y permiten categorizarlas por regiones, idioma y modelo de dispositivo.

El concepto de permitir a desarrolladores utilizar las capacidades del dispositivo mediante el uso de interfaces de acceso y luego distribuir el resultado de ese desarrollo para ofrecerle a los usuarios la posibilidad de ampliar o enriquecer su experiencia al utilizar su dispositivo, es algo que define la motivación final del desarrollo de esta tesina.

3.1.3. La empresa, el equipo e invitados

La empresa posee un área de desarrollo grande, dividida en grupos donde cada grupo tiene uno o varios proyectos dentro de sus responsabilidades. Generalmente un proyecto equivale a un producto, y el equipo responsable tiene la obligación de mantener los sistemas actuales que soporten ese producto. A su vez, deben desarrollar en paralelo las subsiguientes versiones aún no lanzadas.

El equipo de desarrollo a cargo de este proyecto consta de tres programadores, un diseñador y un ingeniero de QA, y si bien el equipo funcionó en aislamiento del resto de la empresa durante las primeras etapas donde sólo se enfocó en resolver los problemas técnicos identificados desde el nacimiento de la idea, en etapas posteriores se invitaron a varios desarrolladores de otros proyectos a participar de un hackaton para que probaran el sistema y pudieran trabajar en equipos desarrollando *widgets* de prueba. Con esto se buscó observar el comportamiento de la plataforma y obtener opiniones sobre la *UI* y posibles mejoras en la experiencia del usuario. Los desarrolladores luego dieron su feedback sobre las cuestiones a mejorar.

A diferencia de otros sistemas creados en la empresa, esta plataforma arranca sin un objetivo completamente definido, y, como se fue manifestando a lo largo del proceso, su naturaleza fue cambiando para ajustarse a las nuevas directivas y decisiones de negocio que se iban tomando conforme la visión del sistema como un producto comercializable se hacía mas clara. Esta incertidumbre se conocía desde el inicio y fue en gran parte una motivación extra para el grupo de desarrollo. El proyecto se encaró como un desafío, con

un espíritu emprendedor de aventura e investigación, lo que nos brindó la oportunidad de aportar muchas ideas propias al resultado final.

3.1.3.1. Pruebas de usabilidad y feedback interno

Una vez que las bases del sistema estuvieron establecidas se involucró a más personas en calidad de consultores, principalmente a biotecnólogos del sector de laboratorio. Mediante una serie de pruebas de usabilidad, donde se les fue pidiendo que realizaran tareas simples y se los observó intentar completar las tareas sin conocer previamente el sistema, ellos pudieron brindar un *feedback* inicial sobre el *IDE Web*. Más allá del resultado de las pruebas, resultó muy útil su opinión sobre ciertas mejoras a realizar, e incluso nos mostraron el software que usaban actualmente y cómo estaban acostumbrados a trabajar.

3.2. HIPAA

HIPAA, traducido como Ley de Transferencia y Responsabilidad de Seguro Médico, es una legislación de EUA que provee un marco de seguridad y privacidad para datos médicos. Esta ley cobra vital importancia en los últimos años con el incremento de ataques cibernéticos a entidades de salud y el filtrado de datos médicos de los proveedores y aseguradoras. Al ser una ley federal protege a todas las personas en el territorio norteamericano.

HIPAA provee una serie de regulaciones que todas las entidades relacionadas con datos de salud identificables, llamados PHI, tienen que cumplir. Existen básicamente dos tipos de organizaciones que están alcanzados por la normativa: las entidades cubiertas (CE, por sus siglas en ingles) y las empresas asociadas (BA).

Las CE están compuestas por prestadores de seguro de salud, hospitales y entidades de cuidados. Mientras que los BA son cualquier entidad o empresa que este en contacto con PHI. Un ejemplo de una BA son las empresas de destrucción de documentación, los proveedores de Hosting e incluso abogados que tengan acceso a los datos médicos de sus representados.

Las empresas y proveedores definidos como BA tienen que, entre otras consideraciones, entrenar a su personal, crear planes de contingencia y análisis de riesgos para el tratamiento de PHI.

3.2.1. ¿Que es PHI?

Se define como PHI a cualquier dato o conjunto de datos que identifique a una persona con información de salud. Algunos ejemplos de datos que son considerados PHI[5] son:

1. Nombres
2. Fecha, excepto años
3. Número de teléfono
4. Información geográfica
5. Número de FAX
6. Número de seguridad social

7. Dirección de email
8. Número de registro médico
9. Número de cuentas
10. Número de certificados
11. Patentes de vehículos
12. URLs web
13. Número de serie y identificadores de dispositivos
14. Direcciones IP
15. Foto completa de rostro e imágenes comparables
16. Identificadores biométricos (registro de retina, huellas dactilares, etc.)
17. Cualquier código, número o dato que identifique a un individuo

Esta información no necesariamente tiene que ser actual, un hacker puede por ejemplo relacionar un número de teléfono fuera de servicio con su dueño anterior.

PHI se puede definir como la intersección de un dato personal y un dato de salud. Existen varios formatos de PHI; se habla de ePHI cuando la información es creada o almacenada de forma electrónica. También puede ser verbal o escrita.

Cualquier entidad alcanzada por la ley HIPAA puede ser denunciada por filtración de PHI, los individuos puede ser multados con valores que van desde USD $100 hasta USD $25000 y pueden enfrentar encarcelamiento de uno a diez años dependiendo de la gravedad y volumen de la información filtrada.

3.2.2. HIPAA en el desarrollo de software

El desarrollo de una aplicación que maneje datos médicos alcanzados por HIPAA tiene algunas particularidades ya que la ley regula ciertos aspectos de la forma en que el dato PHI debe ser almacenado, transmitido y mostrado. Estas tres acciones básicamente abarcan todo el viaje que se realiza desde una base de datos, pasando por un backend, siendo transmitido y mostrado

en un frontend, incluso alcanza a pasos intermedios entre ambos, como por ejemplo si existiese una capa de caché.[6]

Es posible agrupar las distintas regulaciones en tres grandes grupos que se verán a continuación.

3.2.2.1. Políticas de Privacidad

Las políticas de privacidad de HIPAA resguardan los registros electrónicos relacionados con información de salud y cualquier otro tipo de información médica confidencial. Mientras que hoy día prevalece la información médica en algún formato electrónico (también conocido como ePHI), el PHI también debe ser protegido en todos sus formatos, ya sea escrito, filmado, o incluso hablado.

Para poder proveer la protección correcta a estos datos, las organizaciones reguladas bajo la ley HIPAA deben tomar a cabo una serie de medidas. Inicialmente deben configurar medidas de protección técnicas para resguardar el ePHI, y posteriormente deben establecer controles para prevenir cualquier uso indebido y la filtración de los datos protegidos.

Esta política también establece derechos que puede ejercer el paciente por sobre sus datos, por ejemplo habla sobre que el paciente puede pedir y recibir cualquier dato PHI y solicitar su remoción o rectificación en caso de errores.

3.2.2.2. Políticas de Seguridad

El objetivo de estas políticas es el de establecer una guía para las entidades sobre como garantizar la disponibilidad, integridad y confidencialidad del dato PHI que sea trasmitido, almacenado, recibido o creado por una entidad cubierta o empresa asociada.

Las reglas de seguridad hacen necesario para cada organización que maneje ePHI levantar tres tipos de defensas: protecciones técnicas, físicas y administrativas.

Protecciones Técnicas

Control de acceso Control de acceso sobre la visualización y manipulación de PHI, esto incluye implementar una política de claves robustas

y preferentemente reforzada por mecanismos de *Autentificación multifactor*

Control de auditoría Se refiere a proveer claridad en los movimientos de ePHI dentro y fuera de la organización. Esto incluye llevar registro de todos los intentos de acceso y transmisión de información para facilitar la detección de filtrados de PHI.

Control de integridad Estas reglas de control de integridad gobiernan la disponibilidad y precisión de ePHI y las medidas tomadas para mantener los datos seguros de amenazas como la modificación maliciosa o el borrado accidental. Generalmente esto requiere mantener varias copias de seguridad redundantes en sistemas independientes.

Control de trasmisión La información es muchas veces vulnerada al ser transmitida, especialmente en la actualidad, donde muchos trabajadores acceden de manera remota a entornos de trabajo mediantes redes inalámbricas inseguras. Para prevenir que los atacantes puedan interceptar la información en tránsito, se debe utilizar una encriptación *end-to-end* en cada comunicación.

Protecciones físicas

Acceso a instalaciones Incluye cuestiones como el bloqueo de puertas y control de acceso con tarjetas de identificación para los cuartos de los servidores. También los proveedores de soluciones en la nube debe proveer detalles de como las instalaciones son aseguradas.

Acceso a dispositivos Es común que el acceso a información protegida se haga mediante el uso de teléfonos celulares, ya sean provistos por la compañía o personales. Estos dispositivos tienen que estar asegurados y, de ser posible, se tiene que evitar directamente el guardado de información sensible en los dispositivos.

Protecciones administrativas

Estas recomendaciones se refieren a varias políticas y procedimientos implementados para asegurar la seguridad y privacidad de ePHI. Requieren supervisión de un oficial de seguridad y privacidad dedicado, quien es también responsable de reforzar la correcta conducta del personal.

Manejo de riesgo Una de las primeras tareas a la hora de certificarse como ajustado a la ley HIPAA es la de analizar e identificar los

posibles riesgos existentes que amenazan el ePHI. Esto incluye, pero no se limita, crear un inventario completo de cada área y sistemas utilizados para almacenar o transmitir PHI antes de tratar de determinar todas aquellas potenciales maneras en las que se puede producir una filtración de datos. Este paso es fundamental y es la base de la estrategia de resarcimiento.

Personal de seguridad HIPAA requiere que cada entidad cubierta y cada empresa asociada tengan designados un oficial de seguridad y un oficial de privacidad, pudiendo ser la misma persona que ocupa ambos roles.

Manejo de la información Se refiere al manejo responsable de ePHI. Este proceso involucra administrar el dato durante todo su ciclo de vida, desde el momento que es creado hasta cuando es eliminado de los sistemas. Los administradores deben también restringir acceso de terceros, clasificar la información de acuerdo a niveles de confidencialidad y mantener control completo sobre los mismos.

Entrenamiento sobre seguridad La mayoría de los ataques explotan la ignorancia y la falta de preparación técnica y no las vulnerabilidades técnicas. El entrenamiento del personal es fundamental para poder concientizar sobre las políticas y prácticas que se requieren en la ley HIPAA.

Evaluaciones periódicas Los administradores deben llevar a cabo evaluaciones de riesgo periódicas para asegurarse que sus documentos de certificación están al día y que sus protecciones de seguridad, administrativas y físicas siguen siendo relevantes y suficientes. Idealmente, todas las entidades deben realizar una revalidación una vez por año.

3.2.2.3. Políticas de Notificación de incidente

Ante un evento de filtración de información las entidades están obligadas a notificar a los pacientes cuyos datos fueron comprometidos, pero si el numero de registros comprometidos supera los 500, las notificaciones también deben ser expedidas hacia el HHS (*U.S. Department of Health and Human Services*) y hacia los medios.

Capítulo 4

Trabajos relacionados

Desde el inicio fue complicado encontrar un software que fuera comparable con el objetivo de nuestro trabajo. Existen muchas propuestas que basan su premisa en la idea de programar en el navegador, pero todas las opciones analizadas son incompletas o tienen un enfoque completamente distinto al imaginado para nuestro *IDE Web*. Uno de los aspectos más desafiantes es el tema de la privacidad de los datos, ya que hacer uso de cualquier plataforma externa requería un análisis exhaustivo y un compromiso, desde el punto de vista de negocio, hasta incluso legal, que la empresa no estaba dispuesta a realizar.

Aunque el objetivo era claro, los medios para lograrlo no lo eran tanto; el *IDE Web* surge como una idea simple que fue madurando durante su desarrollo. Actualmente, mas allá de que existen varios aplicativos publicados, aún persiste la sensación de que el potencial del *IDE Web* no ha sido alcanzado por completo.

A continuación se mencionan herramientas o servicios que ofrecen algún aspecto que ha ayudado a visualizar el *IDE Web* como objetivo a alcanzar. Ninguna de estas herramientas o servicios podría reemplazar al *IDE Web* pero han servido de comparación y contraste a la hora de idear el proyecto.

4.1. CodeSandbox

CodeSandbox[1] se define como un editor web de código fuente para el prototipado rápido. Entre sus características fundamentales se destacan:

- Soporte para proyectos de múltiples archivos
- Soporte para más de 40 lenguajes o dialectos de desarrollo web como ReactJs, Vue, Angular, etc.
- Permite el desarrollo cooperativo en tiempo real
- Previsualización inmediata
- Integraciones con herramientas esenciales de desarrollo web como GitHub[2] y NPM[3]

4.1.1. CodeSandbox - SandPack

Recientemente la plataforma *CodeSandbox* lanzó un proyecto open source llamado **SandPack**.

Si bien este proyecto es posterior con respecto al desarrollo del *IDE Web*, es posible tenerlo en cuenta a la hora de pensar en una futura versión.

Sandpack se define como un ecosistema de componentes y utilidades abierto que permite compilar y correr *frameworks* modernos en el nevagador[7]. Básicamente permite embeber las capacidades de *CodeSandbox*, es decir un editor de código con previsualización en vivo y otras herramientas relacionadas con un IDE, dentro de un proyecto ReactJs exponiendo varios componentes que permiten de manera declarativa controlar cómo se comportan y cómo se ven estas herramientas dentro de un proyecto.

El ejemplo básico de la implementación de *SandPack* es el visualizado en el bloque de código 1 el cual genera un editor de código con la previsualización en tiempo real correspondiente como se puede apreciar en la figura 5.2.

---

[1] `https://codesandbox.io/`

[2] Github `https://github.com/`

[3] npm `https://www.npmjs.com/`

```
1  import { Sandpack } from "@codesandbox/sandpack-react";
2
3  <Sandpack template="react" />;
```

Bloque de código 1: Implementación mínima de un editor de código Sandpack

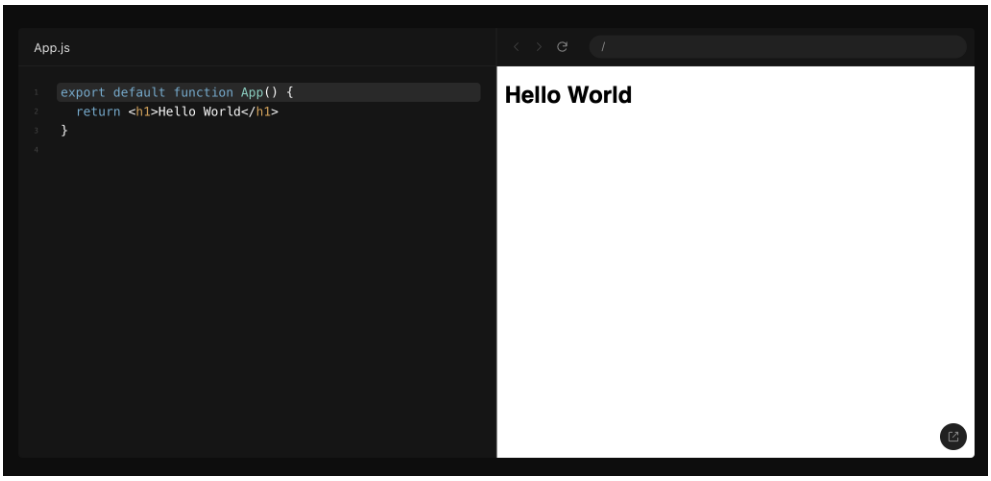

Figura 4.1: Componente SandPack renderizado con opciones por defecto

Otra de las características interesantes de este proyecto es la posibilidad de configurar *Providers* y *Custom Hooks* para que los componentes de ReactJs tengan acceso a funcionalidad extendida. Esta es una característica que, como se explica más adelante en este documento, juega un papel crucial para poder integrar los *widgets* con el ecosistema de servicios existentes.

Capítulo 5

Análisis de tecnologías

En este capítulo veremos las distintas tecnologías seleccionadas para el desarrollo propuesto, sus características principales y alternativas que se analizaron junto con los motivos de la selección.

5.1. Tecnologías de front-end

JavaScript ha tenido un sinfín de *frameworks* que han sido creados para facilitar la vida de los desarrolladores. Muchos de estos *frameworks* quedaban en el olvido conforme pasaban las versiones de ECMAScript y la escena de desarrollo front-end adoptaba una nueva moda. Otros se han ido actualizando y su evolución ha permitido que se mantengan vigentes por muchos años.

5.1.1. ReactJs

ReactJs, o simplemente React, es una librería de JavaScript muy popular usada para crear interfaces. React fue lanzada y desarrollada por *Facebook* en el año 2013. Son ellos los encargados de mantenerla junto a una comunidad de desarrolladores independientes y otras compañías. Su diseño se enfocó en facilitar muchas de las cuestiones complicadas asociadas al front-end de sitios grandes. Inicialmente la comunidad se mostró escéptica debido a los cambios de convenciones que presentaba React con respecto a otros *frameworks* contemporáneos [8].

Aunque React sea una librería muy compacta, que no trae todo lo que un

desarrollador puede llegar a requerir, o que el código que uno escribe parece HTML dentro de JavaScript, React ha sabido mantenerse como *framework* de front-end líder para crear Single Page Applications[9].

5.1.1.1. Desarrollo guiado por componentes

Component Driven Development (CDD) es la práctica de desarrollo y diseño de crear interfaces de usuario con componentes modulares. La UI se diseño siguiendo el principio bottom-up empezando por los componentes básicos y progresivamente combinándolos para construir pantallas.

Las interfaces de usuario modernas cada día crecen en complejidad conforme la tecnología avanza y las aplicaciones crecen, convirtiéndose en estructuras acopladas delicadas y difíciles de depurar. Dividirlas de manera modular permite que sea más fácil crear interfaces robustas pero sin sacrificar flexibilidad.

Los cuatro pasos de CDD se resumen en:

Construir componentes de a uno Construir los componentes en aislamiento y definir sus estados relevantes.

Combinar componentes Utilizar varios componentes menores para agrupar funcionalidad y mantener coherencia.

Ensamblar páginas Construir páginas combinando componentes compuestos.

Integrar las páginas en el proyecto Agregar las páginas conectando el estado y la lógica de negocios.

Dentro de los beneficios podemos nombrar que al desarrollar los componentes en aislamiento, prácticas saludables como el testeo unitario son mas accesibles y permiten al creador enfocarse en el componente y su comportamiento sin preocuparse por el entorno.

Otro beneficio es la posibilidad de crear librerías o muestrarios de componentes. Existen proyectos open source muy interesantes para la creación de exploradores de componentes donde se es posible visualizarlos que ya están creados e incluso interactuar y ver cómo reaccionan a cambios de estados;

entre los más populares se encuentra *React Storybook*[1].

5.1.1.2. Lodash

Lodash[2] es una librería de Javascript ampliamente utilizada ya que aporta muchas utilidades que permiten facilitar el manejo de arreglos y colecciones. Tiene una gran influencia de lenguajes de programación funcionales, lo cual permite encadenar operaciones e incluso crear funciones compuestas o currificar funciones normales.

Dentro de JavaScript se puede importar lodash y utilizar un guión bajo (un '*Low dash*') _ para acceder a las funciones expuestas.

A continuación se listas algunas de las funciones que utilizamos en este proyecto, mas allá de que **lodash** posee más de 200 funciones de ayuda.

_.uniq(array)

Uniq[3] Retorna un arreglo libre de valores duplicados.

```
1 _.uniq([2, 1, 2]);
2 // => [2, 1]
```

_.flatten(array)

flatten[4] toma un arreglo como parámetro y devuelve otro que posee solo un nivel de valores. En JavaScript los arreglos pueden contener arreglos; esta función *aplana* la estructura.

```
1 _.flatten([1, [2, [3, [4]], 5]]);
2 // => [1, 2, [3, [4]], 5]
```

---

[1] React storybook - Build bulletproof UI components faster `https://github.com/storybookjs/storybook`

[2] `https://lodash.com/`

[3] `https://lodash.com/docs/4.17.15#uniq`

[4] `https://lodash.com/docs/4.17.15#flatten`

_.difference(array, [values])

La función *difference*[5] retorna un arreglo conteniendo los valores que existen en el primer argumento pero no en los arreglos sucesivos.

```
_.difference([2, 1], [2, 3], [3]);
// => [1]
```

_.keys(object)

Esta función[6] devuelve un arreglo con los nombres de las propiedades existentes en un objeto, solo las propiedades propias y no las heredadas.

```
function Foo() {
  this.a = 1;
  this.b = 2;
}

Foo.prototype.c = 3;

_.keys(new Foo);
// => ['a', 'b'] (iteration order is not guaranteed)

_.keys('hi');
// => ['0', '1']
```

.some(collection, [predicate=.identity])

La función *some*[7] recibe un arreglo o un objeto y una función predicado, retorna un valor booleano que depende de si al menos un valor de esa colección satisface la condición. También se puede usar sin función predicado y **lodash** actuara de manera acorde dependiendo de que estructura se use.

```
_.some([null, 0, 'yes', false], Boolean);
// => true

```

[5] https://lodash.com/docs/4.17.15#difference

[6] https://lodash.com/docs/4.17.15#keys

[7] https://lodash.com/docs/4.17.15#some

```
var users = [
  { 'user': 'barney', 'active': true },
  { 'user': 'fred',   'active': false }
];

// The `_.matches` iteratee shorthand.
_.some(users, { 'user': 'barney', 'active': false });
// => false

// The `_.matchesProperty` iteratee shorthand.
_.some(users, ['active', false]);
// => true

// The `_.property` iteratee shorthand.
_.some(users, 'active');
// => true
```

5.1.1.3. MomentJs

MomentJs[8] es una librería especializada en interpretar, manipular y mostrar fechas y mediciones temporales en JavaScript y permite abstraerse de muchas cuestiones técnicas que se tienen que tener en cuenta a la hora de manipular y mostrar unidades temporales en el proyecto.

La utilizaremos principalmente para hacer cálculos sobre los tiempos que llevan los procesos en ejecutarse, para obtener la fecha actual y principalmente para mostrar fechas en la UI.

5.2. Tecnologías para el desarrollo móvil

5.2.1. React Native

React Native es un framework open source para desarrollar aplicaciones para Android e iOS utilizando ReactJs y las propiedades que obtendríamos al desarrollarlas de forma nativa.

Habitualmente en Android se desarrollan las *apps* en Java o Kotlin, en iOS se hace lo mismo con Swift y Objective-C.

[8] `https://momentjs.com/`

Con React Native utilizaremos componentes ReactJs que luego en tiempo de ejecución renderizarán la vista correspondiente a Android o iOS, como muestra el ejemplo del cuadro 5.1.

Con un único código en JavaScript podremos ejecutar tanto en Android como en iOS.

React Native	**Componente android**	**Componente iOS**	**Descripción**
`<View>`	`<ViewGroup>`	`<UIView>`	Contenedor genérico que soporta eventos de toque y estilos
`<Text>`	`<TextView>`	`<UITextView>`	Contenedor que muestra lineas de texto
`<Image>`	`<ImageView>`	`<UIImageView>`	Muestra diferentes tipos de imágenes

Cuadro 5.1: Componente de React Native y su contraparte nativa para ambas plataformas

5.3. Lenguaje

5.3.1. JavaScript

Como lenguaje base para el desarrollo del *IDE Web* como así también de los clientes web y *mobile* se decidió utilizar JavaScript, lenguaje de programación encargado de dotar de mayor interactividad y dinamismo a las páginas web. Cuando este se ejecuta en el navegador, no necesita de un compilador. El navegador lee directamente el código, sin necesidad de terceros. Por lo tanto, se le reconoce como uno de los tres lenguajes nativos de la web junto a HTML (contenido y su estructura) y a CSS (diseño del contenido y su estructura).

JavaScript fue creado a mediados de los noventas por **Brendan Eich**, quien trabajaba para *Netscape*. Aunque inicialmente era considerado un lenguaje poco serio y sin mucho futuro, que solo servía para ser usado por diseñadores y no-programadores, fue ganando popularidad. En 1997 debido a la gran velocidad con la que JavaScript crecía, Netscape delegó el trabajo de crear la especificación del lenguaje y su futuro mantenimiento a la European Computer Manufacturers Association [9]. La especificación de ECMA se publicó bajo el nombre *ECMA-262* y un lenguaje ECMAScript que incluía JavaScript, JScripts y ActionScript. Actualmente ECMAScript y JavaScript suelen referirse a lo mismo, pero muchas veces se hace la distinción de que JavaScript es el lenguaje y su implementación, mientras que ECMAScript es el estándar del lenguaje y la versión del lenguaje. El año 2005 fue un punto de inflexión gracias a la publicación del *paper Ajax: A New Approach to Web Applications [Ajax: un nuevo acercamiento para las Aplicaciones Web][10]* de Jesse James Garrett donde se describe una nueva manera de obtener datos asincrónicamente, es decir, permitiendo interacciones mucho más complejas. Este *paper* sobre AJAX resultaría revolucionario por las repercusiones y se considera uno de los pilares de la internet como la conocemos hoy día.

En el año 2016 se lanzó la sexta versión de ECMAScript conteniendo muchas mejoras significativas al lenguaje y en la cual se adoptó el nombre *ECMAScript 2015*; desde entonces ha tenido una versión nueva cada año [10].

Actualmente JavaScript mantiene su reinado y en el 2022 marcó su

[9] ECMA, Asociación Europea de Fabricantes de Computadoras

[10] ECMAScript Versions [Versiones de ECMAScript] `https://en.wikipedia.org/wiki/ECMAScript#Versions`

décimo año consecutivo como el lenguaje de programación más usado.[11]

5.4. Tecnologías de backend

5.4.1. Node.Js

Node.Js es un entorno de ejecución (*runtime*) para JavaScript construido con **V8**, un motor de JavaScript de Chrome de código abierto escrito en C++ que implementa ECMAScript [12] y permite la ejecución de código JavaScript en un ambiente de línea de comando.

Node.js es comúnmente utilizado para el desarrollo back-end y gracias a su naturaleza asincrónica es capaz de crear aplicaciones de servicios y servidores web. También es muy popular para crear herramientas de línea de comando.

Una aplicación Node.js se ejecuta sobre un solo proceso, sin crear nuevos hilos puede atender concurrentemente muchas peticiones. Node.js provee un conjunto de primitivas asincrónicas para I/O en su librería estándar que previenen operaciones bloqueantes; casi ninguna función en Node.js realiza I/O directamente, por lo que el proceso nunca se bloquea [11]. Por ello, es muy propicio desarrollar sistemas escalables en Node.js. El hecho de que Node.js esté diseñado para trabajar sin hilos no significa que no pueda aprovechar múltiples núcleos en su entorno. Es posible generar subprocesos o procesos hijos utilizando una API especial.

Una diferencia importante entre el desarrollo con JavaScript en un navegador y Node.js es que en el navegador la versión del interprete depende de la versión del navegador instalado que tenga el usuario, mientras que en Node.js está dictaminada por la versión que este instalada en el servidor. Por lo tanto, actualizar o adoptar las nuevas versiones de ECMAscript es mucho más sencillo, e incluso se puede ejecutar Node.js con una serie de banderas determinadas para habilitar características experimentales que todavía no son parte del estándar.

---

[11] 2022 Developer Survey, StackOverflow `https://survey.stackoverflow.co/2022/#most-popular-technologies-language`

[12] V8 Documentation [V8, documentación] `https://v8.dev/docs`

Node.js posee un número impresionante de librerías creadas por la comunidad a disposición de los desarrolladores a través de la herramienta NPM.

NPM permite tener a disposición mas de 350000 paquetes publicados. Mediante la herramienta de línea de comando el desarrollador puede iniciar un proyecto NPM simplemente ejecutando `npm init`[12], donde después de una serie de preguntas, se termina inicializando un archivo *package.json* donde esta escrita toda la información del nuevo proyecto, incluyendo versión, dependencias, y *scripts* que están disponibles para poder ejecutar el proyecto o correr los casos de prueba configurados. En el bloque de código 2 se puede observar el contenido de un archivo *package.json* describiendo un proyecto de ejemplo.

```
> npm init --yes
Wrote to /home/monatheoctocat/my_package/package.json:
{
  "name": "my_package",
  "description": "",
  "version": "1.0.0",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "repository": {
    "type": "git",
    "url": "https://github.com/monatheoctocat/my_package.git"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "bugs": {
    "url": "https://github.com/monatheoctocat/my_package/issues"
  },
  "homepage": "https://github.com/monatheoctocat/my_package"
}
```

Bloque de código 2: Ejemplo de archivo package.json con la información del proyecto npm

5.4.2. Express

Node.js es una plataforma de bajo nivel, para hacer más simple e interesante el desarrollo existen muchas librerías que han sido adoptadas como frameworks para determinadas situaciones. Usar **Express** es una de las maneras más populares y poderosas de crear servidores web. Se define como un

framework web para Node.js, mínimo y flexible, que proporciona un conjunto sólido del características para las aplicaciones web y móviles. En el ejemplo del bloque de código 3 se puede ver que con tan solo una decena de líneas de código se puede tener un servidor web corriendo.

```
const express = require('express')
const app = express()
const port = 3000

app.get('/', (req, res) => {
  res.send('Hola Mundo!')
})

app.listen(port, () => {
  console.log(`Ejemplo de servidor escuchando en el puerto ${port}`)
})
```

Bloque de código 3: Ejemplo de servidor web diciendo 'hola mundo' escrito en Node.js usando Express

5.4.3. MeteorJs

MeteorJs se define como una plataforma de desarrollo *fullstack* para JavaScript. Incluye un conjunto específico de herramientas para facilitar el desarrollo de aplicaciones, herramientas de construcción y paquetes seleccionados de la comunidad de JavaScipt y NodeJs.

Dentro de las cualidades que MeteorJs posee se puede destacar que permite el uso de un solo lenguaje, ya sea en el *frontend* como en el *backend* y que provee medios para facilitar la *reactividad*, permitiendo que la UI refleje el estado actual de la aplicación con el mínimo esfuerzo.

MeteorJs nos provee de varias soluciones ya resueltas para distintas partes de una aplicación. Nosotros usaremos solo un subconjunto de esas soluciones pero quedan muchas más sin aprovechar; incluso con el uso de paquetes de la comunidad se puede ampliar considerablemente las capacidades de la plataforma.

A continuación se listan y describen algunos de los aspectos de MeteorJs que durante el desarrollo del *IDE Web* han sido de gran ayuda.

5.4.3.1. Colecciones

MeteorJs usa las colecciones para almacenar datos. Las colecciones son estructuras especiales que se encargan de almacenar la información de manera permanente en la base de datos MongoDb del server y sincronizarla con cada uno de los navegadores de los usuarios conectados. Al utilizar MongoDb5.4.5.2 las colecciones están formadas por documentos (en un formato EJSON) y la plataforma ofrece un alto nivel de abstracción para poder facilitar la operatoria de almacenamiento, búsqueda y modificación de datos.

En el ejemplo del bloque de código 4 se puede observar como se crea una colección, y al especificarle un nombre le estamos diciendo a MeteorJs que esta colección se tiene que persistir en la DB y que todos los usuarios pueden acceder a los datos contenidos en la colección. Por supuesto hay mecanismos para facilitar el control de acceso a estos datos pero no los veremos en este momento.

```
1    // Creando una coleccion
2    const Messages = new Mongo.Collection('messages');
3
4    // Creando un muevo mensaje
5    Messages.insert({ text: 'Hello, world!' });
```

Bloque de código 4: Ejemplo de creación de colección en MeteorJs

5.4.3.2. Publicación y suscripción

Este es un mecanismo por el cual MeteorJs controla qué conjunto de datos son expuestos y cómo los clientes se subscriben a ellos. Publicación y suscripción (*pub/sub*)[13] no es un concepto exclusivo de MeteorJs, es un patrón de diseño conocido y probado que brinda varias ventajas. El patrón *pub/sub* desacopla la lógica de comunicación y la lógica de negocio, permitiendo aislamiento de componentes, conservando el sistema robusto pero mantenible. Este bajo acoplamiento permite un crecimiento elástico entre las distintas partes del sistema. Desde el punto de vista del desarrollo, permite modularizar e incluso permite la comunicación entre sistemas de distintos orígenes o creados con distintos lenguajes.[14]

En el bloque de código 5 podemos observar como el servidor expone una colección a los clientes, pero solo con un subconjunto de los datos totales, filtrados en este caso por la propiedad *userId*. El cliente al suscribirse

```
// En el servidor
const Widgets = new Mongo.Collection('widgets');
Meteor.publish('widgets', function () {
    return Widgets.find({userId: this.userId});
});


// En el cliente
const Widgets = new Mongo.Collection('widgets');
const subscription = Meteor.subscribe('widgets');
const loading = !subscription.ready();
let myWidgets = {}
if (!loading && widgetId) {
  myWidgets = Widgets.find({});
}
return myWidgets;
```

Bloque de código 5: Ejemplo básico de publicación y suscripción

solo recibe esos documentos. También se puede dar el caso en que solo se sincronicen documentos que van a ser mostrados inmediatamente, es decir, si tuviésemos una lista y se fuera mostrando por páginas, solo las páginas que se están visualizando contendrían datos mientras que las páginas siguientes se sincronizarían bajo demanda.

5.4.3.3. Paquetes MeteorJs

Unas de las facilidades de MeteorJs es la posibilidad de utilizar "*paquetes*"*Open Source* (*código abierto*) que la comunidad publica. Estos módulos se encargan de agregar nueva funcionalidad o modificar las capacidades del *framework*. Cada uno de estos paquetes fueron siendo instalados a medida que el desarrollo lo requería, a continuación se detallan algunos paquetes que jugaron un rol fundamental en el desarrollo del *IDE Web*.

Astronomy

Astronomy [15] permite definir el modelo de la información almacenada en las colecciones de MeteorJs. Nos provee una manera fácil de poder tener control sobre las propiedades y validaciones que los objetos tienen que cumplir y que originalmente, dada la naturaleza *schemeless* (*libre de esquema*) del motor de base de datos MongoDB, no es posible.

Este módulo es uno de los más importantes dado que toda la capa de modelo esta definida usando los esquemas de Astronomy.

5.4.3.4. Métodos Meteor

MeteorJs permite la creación de funciones o métodos universales que funcionan tanto en el front-end como en el back-end. Esto permite desarrollar funcionalidad sin tener en consideración en donde se esta ejecutando.

En el *IDE Web* contamos con muchos *métodos MeteorJs*, en especial aquellos que hacen referencia al procesamiento de datos. En el ejemplo mostrado en el bloque de código 6 se demuestra un método MeteorJs especializado para buscar el perfil de un usuario a partir del nombre se usuario.

```
Meteor.methods({
  'findUser': function(username) {
    return Meteor.users.findOne({
      username: username
    }, {
      fields: { 'username': 1 }
    });
  }
});
```

Bloque de código 6: Ejemplo básico de un método de meteor.

5.4.4. Babel

Babel es la herramienta que nos permite manipular el código fuente y utilizar las últimas adiciones de JavaScript pero seguir siendo compatible con navegadores y ambientes que no soportan esas características. Es decir, Babel traduce el código escrito, el termino técnico en inglés es *transpile*, y produce un código que dependiendo de la sintaxis utilizada puede contener agregados que lo hacen retro-compatible. En el bloque de código 7 se puede observar un ejemplo simplificado de la transformación que se logra. El código agregado tiene el nombre de *polyfill*.

Si bien el ejemplo es simple se ve claramente como el código resultante esta traducido para que cualquier ambiente compatible con *ES5* lo pueda ejecutar. Otra característica interesante de Babel es que se le pueden agregar

plugins para ampliar su funcionalidad, ya sea para agregar nuevos *polyfills* o hacer nuevas transformaciones al código inicial.

```
// Babel Input: ES2015 arrow function
[1, 2, 3].map(n => n + 1);

// Babel Output: ES5 equivalent
[1, 2, 3].map(function(n) {
  return n + 1;
});
```

Bloque de código 7: Ejemplo de como Babel transforma código escrito en ES2015 en código compatible con navegadores antiguos.

5.4.5. Bases de datos

En nuestro proyecto se hacen uso de varios motores de bases de datos distintos y variados. La elección de cada uno esta basada en el problema a resolver junto con la experiencia previa del equipo de desarrollo.

5.4.5.1. MySQL

MySQL es un sistema de gestión de bases de datos relacional desarrollado bajo licencia dual: Licencia pública general/Licencia comercial por Oracle Corporation y está considerada como la base de datos de código abierto más popular del mundo[13], y una de las más populares en general junto a Oracle y Microsoft SQL Server, todo para entornos de desarrollo web.

MySQL presenta algunas ventajas que lo hacen muy interesante para los desarrolladores. La más evidente es que trabaja con bases de datos relacionales, es decir, utiliza múltiples tablas que se interconectan entre sí para almacenar la información y organizarla correctamente.

Al ser basada en código abierto es fácilmente accesible y la inmensa mayoría de programadores que trabajan en desarrollo web han utilizado MySQL en alguno de sus proyectos. MySQL esta ampliamente extendido y cuenta además con una ingente comunidad que ofrece soporte a otros usuarios. Pero estas no son las únicas características como veremos a continuación:

Arquitectura Cliente y Servidor MySQL basa su funcionamiento en un modelo cliente y servidor. Es decir, clientes y servidores se comunican entre sí de manera diferenciada para un mejor rendimiento. Cada cliente puede hacer consultas a través del sistema de registro para obtener datos, modificarlos, guardar estos cambios o establecer nuevas tablas de registros, por ejemplo.

Compatibilidad con SQL SQL es un lenguaje generalizado dentro de la industria. Al ser un estándar, MySQL ofrece plena compatibilidad, por lo que un desarrollador que ha trabajado con otro motor de bases de datos relacional no tendrá problemas en migrar a MySQL.

[13]DB-Engines Ranking [Ranking de Bases de Datos] `https://db-engines.com/en/ranking`

Vistas Desde la versión 5.0 de MySQL se ofrece compatibilidad para poder configurar vistas personalizadas del mismo modo que podemos hacerlo en otras bases de datos SQL. En bases de datos de gran tamaño las vistas se hacen un recurso imprescindible.

Procedimientos almacenados MySQL posee la característica de no procesar las tablas directamente sino que a través de procedimientos almacenados es posible incrementar la eficacia de nuestra implementación.

Disparadores (*triggers*) MySQL permite además poder automatizar ciertas tareas dentro de nuestra base de datos. En el momento que se produce un evento otro es lanzado para actualizar registros o optimizar su funcionalidad.

Transacciones Una transacción representa la actuación de diversas operaciones en la base de datos como un dispositivo. El sistema de base de registros avala que todos los procedimientos se establezcan correctamente o ninguna de ellas. En caso por ejemplo de una falla de energía, cuando el monitor falla u ocurre algún otro inconveniente, el sistema opta por preservar la integridad de la base de datos resguardando la información.

5.4.5.2. MongoDB

MongoDB es un sistema de base de datos NoSQL, orientado a documentos y de código abierto. Una de sus principales características es que nos permite guardar nuestras estructuras o documentos en formato BSON (una especificación similar a JSON [14]).

MongoDB es Schemaless lo que permite que los documentos tengan estructuras diferentes sin afectar su funcionamiento, algo que no podemos hacer con las tablas de las bases de datos relacionales. [16]

En MongoDB, las colecciones son el equivalente a las tablas en una base de datos relacional. Las colecciones agrupan documentos, que serían correspondientes a las filas en el modelo relacional. Los documentos se escriben en formato BSON. Las claves representan columnas y los valores el contenido de la celda. Los valores pueden ser: numéricos, booleanos, *strings*, *arrays* u otros documentos BSON.

[14]JSON vs BSON `https://www.mongodb.com/json-and-bson#json-vs-bson`

5.4.5.3. SQLite

SQLite es una librería software que posibilita la gestión de bases de datos relacionales. [17] A diferencia de otros gestores de base de datos cliente-servidor, no está implementado de manera independiente al programa con el que establece comunicación, más bien forma parte de él, integrándose en su estructura, formando lo que se denomina un gestor de base de datos embebido o empotrado. Por lo tanto, todas las operaciones de base de datos se manejan dentro de la aplicación mediante llamadas y funciones contenidas en la librería SQLite.

Almacena toda una base de datos en un solo archivo que contiene las tablas e índices. Organiza las tablas en Arboles B+ y los índices en Arboles B[18].

Principales características

Cero configuración A diferencia de otros motores de base de datos no se necesita ninguna configuración externa, instalación de software o archivo de configuración para empezar a usar una base de datos SQLite.

Empotrable (o embebible) No se necesita otro proceso servidor dedicado para SQLite. La librería de SQLite se acopla o empotra en el código fuente de la aplicación cliente. No hay protocolos de comunicación externos requeridos para poder usar estas bases de datos.

API Al estar creado usando **C**, la comunicación entre el cliente y el motor sucede de manera interna y no requiere consideraciones especiales a la hora de compilar.

Soporte para transacciones Soporta ACID y no se requiere ningún tipo de proceso especial para recuperar una base de datos que no logró completar una transacción de manera exitosa. En la operación de lectura el motor SQLite lleva a cabo todas las acciones necesarias para realizar un rollback (retroceso) al estado inmediato anterior.

Seguridad de hilos (Thread-safe) Muchos hilos pueden acceder a la misma base de datos concurrentemente.

Liviano La librería completa tiene un peso de cerca de 320Kb cuando todas las características de SQLite están habilitadas, bajando hasta 190Kb si se reducen las características en tiempo de compilación al mínimo.

Personalización Provee un marco robusto para la creación de funciones, funciones de agregación y agrupación personalizadas.

Soporta UTF-8 y UTF-16 .

Multi-plataforma Funciona en todos los sistemas de escritorio como así también en dispositivos empotrados móviles, como teléfonos celulares e incluso en sistemas diminutos con capacidades mínimas de cómputo y memoria. Las bases de datos son compatibles entre todas las plataformas, sin importar su arquitectura u origen.

Un solo archivo Una base de datos por archivo, junto con los índices y *metadata* necesaria. Este enfoque hace que sea sumamente fácil mover, copiar y respaldar estas bases de datos.

SQLite soporta un subconjunto de herramientas para la manipulación y defensión de datos de ANSI SQL-92 [15] muy extenso y algunos comandos exclusivos de SQLite. Algunas características de SQL soportadas:

Lenguaje de definición de datos, *DDL*

- Creación y borrado de tablas, índices, vistas y *triggers* (disparadores)
- Soporte parcial para `ALTER TABLE` (renombrar tabla y agregar columnas)
- Restricciones (constraints) `UNIQUE`, `NOT NULL` y `CHECK`
- Restricciones de clave foránea
- Autoincremento
- Resolución de conflicto

Lenguaje de manipulación de datos, *DML*

- `INSERT`, `DELETE`, `UPDATE` y `SELECT`
- Subconsultas y pasaje de parámetros de subconsultas exteriores

[15] `https://es.wikipedia.org/wiki/SQL-92`

- `GROUP BY`, `ORDER BY` y limites.
- `INNER JOIN`, `LEFT OUTER JOIN` y `NATURAL JOIN`
- `UNION`, `UNION ALL`, `INTERSECT` y `EXCEPT`

Comando transaccionales

- `BEGIN`
- `COMMIT`
- `ROLLBACK`
- `SAVEPOINT`
- `ROLLLBACK TO`
- `RELEASE`

5.5. Desarrollo y características del *IDE Web*

5.5.1. Requerimientos

Basándonos en lo expuesto anteriormente se elabora una lista de Requerimientos funcionales que definen los aspectos fundamentales del sistema. Muchos de estos requerimientos surgen en la concepción de la idea del desarrollo y otros surgen en momentos posteriores refinando funcionalidad existente o agregando nuevas capacidades al sistema.

5.5.1.1. Actores

Para poder entender el comportamiento de la aplicación tenemos que poner en claro los distintos roles de usuario que intervienen y sus relaciones. En muchos casos los actores tienen su contraparte equivalente como entidad en el modelo de la base de datos

Usuarios editores Los editores son los principales usuarios del *IDE Web* y son los desarrolladores que programan *widgets*. Utilizaremos el término editores para referirnos a ellos y así evitar la confusión con los propios desarrolladores del *IDE Web*. En cuestión de relaciones, los editores pueden pertenecer a uno o varios *grupos*; los grupos son una manera de poder discriminar conjuntos de usuarios de manera simple y efectiva. Si llevamos este concepto de grupo al ámbito del negocio podríamos decir que una empresa interesada en desarrollar sus *widget* sería un grupo, y todos los usuarios editores que dicha empresa registre serían editores.

Usuarios *reviewer* Los usuarios con rol de *reviewer* pueden realizar las mismas acciones que los editores pero además tienen acceso a los controles dedicados a la auditoría de los *widgets*. Los *reviewer* cumplen un papel importante y son intrínsecamente empleados de la empresa desarrolladora del *IDE Web*, dedicados a auditar los *widgets* en busca de vulnerabilidades de seguridad y problemas de *performance*. Además, están encargados de verificar aspectos relacionados al *marketing* y cuestiones legales.

Usuarios Administradores Los administradores son usuarios todopoderosos que funcionalmente tienen las mismas capacidades que los editores y los *reviewer*, pero además tienen acceso a controles de configura-

ción de la aplicación únicos para administradores que más adelante se detallarán.

Usuarios Finales Si bien no tienen un rol asignado a nivel código, es importante hacer la distinción. Los usuarios finales son aquellos que poseen datos en el sistema y mediante el uso de los *widget* acceden a su visualización. Es decir, son los consumidores de esta herramienta.

5.5.1.2. Lista de casos de uso iniciales

- Como editor puedo ingresar al sistema de publicación *IDE Web* usando mi usuario y contraseña.
- Como editor puedo crear un *widget* e ingresar su información básica.
- Como editor puedo escribir el código fuente de un *widget* y ver de manera inmediata como se ejecuta.
- Como editor tengo acceso a herramientas de desarrollo que faciliten la programación de un *widget*.
- Como editor puedo ver y modificar todos los *widgets* creados por mí y los de otros editores pertenecientes a los mismos grupos que yo.
- Como editor puedo enviar un *widget* a revisión.
- Como editor puedo marcar un *widget* aprobado como publicado.
- Como editor puedo derivar una nueva versión de un *widget* publicado.
- Como editor puedo pertenecer a varios *grupos*, los *widgets* pueden pertenecer a un solo grupo a la vez.

Durante las numerosas sesiones de brainstorming donde se idearon estos casos de uso también surgieron ideas para ampliar el repertorio de características y mejorar la experiencia de los editores y de los usuarios finales. Algunas de estas ideas se discuten en la sección de desarrollos futuros (ver *Trabajos futuros*). Se priorizaron los casos de uso que permitían obtener un sistema completamente funcional para poder luego enfocarnos en construir funcionalidad específica teniendo usuarios que hayan validado el esfuerzo inicial.

5.6. Prueba de concepto inicial sobre la implementación

Basándonos en los requerimientos iniciales y luego de varias sesiones donde el equipo proponía soluciones y se analizaba su factibilidad teórica, se llegó a la conclusión que necesitábamos realizar una o varias pruebas de concepto para validar las siguientes hipótesis sobre ciertos aspectos que *a priori* parecían desafiantes de resolver:

1. Para poder distribuir el *widget* necesitamos almacenarlo de tal manera que sea agnóstico a la plataforma, es decir, que sea posible su ejecución en varios ambientes (web o *mobile*).

2. Deberíamos proveer al editor de una manera de visualizar lo que está programando.

3. Un editor debería poder escribir consultas SQL para acceder a los datos del usuario que está ejecutando el *widget* (los llamados *usuarios finales*, Ver Actores) pero no debería poder acceder a los datos de otros usuarios. Esta premisa era de vital importancia para poder cumplir con las especificaciones de HIPAA sobre acceso a datos personales.

5.6.1. Código como servicio

Para el problema número **1** la prueba de concepto se basó en verificar si podíamos hacer que Babel funcionara siendo instanciado desde JavaScript y así poder pasarle argumentos que nos permitiera alimentarlo con el código del *widget* escrito por el editor.

El resultado fue muy prometedor ya que encontramos la herramienta **@babel/core**[16] que nos permitía utilizar todas las capacidades de Babel a gusto, ejecutándolo desde un *método de MeteorJs* asincrónicamente (ver bloque de código 8) y, usando el código del *widget* y otros parámetros de configuración, podíamos hacer que nos entregue una sola cadena de caracteres que representara el *widget* listo para ser almacenado. En adelante llamaremos

[16] `https://babeljs.io/docs/en/babel-core`

código *transpilado*[17] a este código resultante, para evitar confusión con el código que el editor escribió, el cual fue la materia prima de este proceso. Modificando la lista de opciones que le pasábamos a Babel también podíamos agregarle varios *polyfills* [18] para que ese código pudiera ser ejecutado en *mobile* y en web. Esta solución nos pareció adecuada, dado que mantener el *código transpilado*| en la base de datos en forma de *string* nos permitía poder distribuirlo mucho más fácilmente que si fueran archivos.

```
Meteor.methods({
  'babelify.js'({ rawCode }) {
    return new Promise((resolve, reject) => {
      try {
        const { code, map } = Babel.transform(rawCode, {
          presets: [BabelEs2015, BabelReact],
          plugins: [BabelSpread]
        });
        resolve({code, map});
      } catch (error) {
        throw new Meteor.Error(500, error.message, error)
      }
    })
  });
```

Bloque de código 8: Declaración del método de MeteorJs que se encarga de convertir el código escrito en *código transpilado*

5.6.2. Previsualización inmediata del widget

Ya teniendo el *código transpilado* del *widget* podíamos ejecutarlo en el mismo navegador para obtener una previsualización casi inmediata de lo que se estaba programando, y así quedaba resuelto el problema número **2**. Se ideó un diseño dividido en la UI del *IDE Web* para que el código fuente se visualice al mismo tiempo que se ejecuta la previsualización del *código transpilado*. En

[17]Transpilador es un tipo especial de compilador que traduce de un lenguaje fuente a otro fuente también de un nivel de abstracción parecido. Se diferencia de los compiladores tradicionales en que estos reciben como entrada ficheros conteniendo código fuente y generan código máquina del más bajo nivel.

[18]Un polyfill es un fragmento de código (generalmente JavaScript en la Web) que se utiliza para proporcionar una funcionalidad moderna en navegadores antiguos que no lo admiten de forma nativa.

la Figura 5.1 se muestra una simplificación de las zonas que se planearon para la UI.

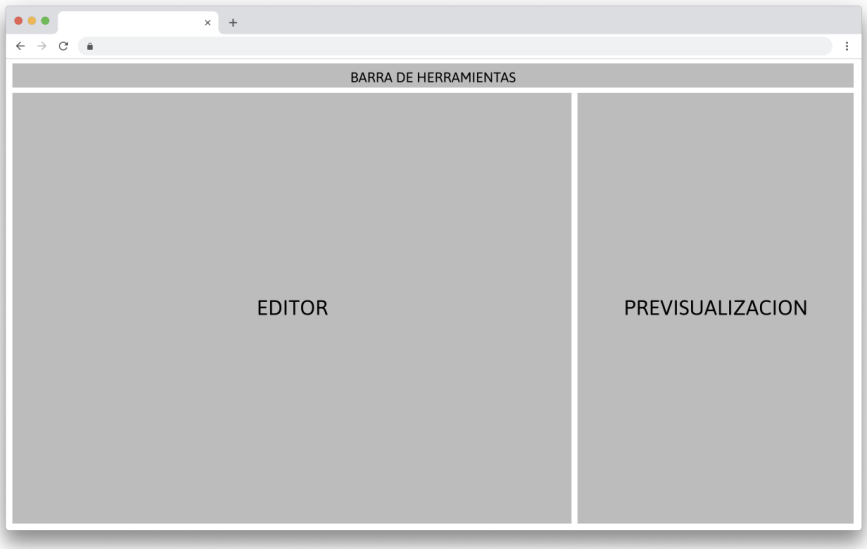

Figura 5.1: Diseño general de la UI propuesta

En etapas posteriores del desarrollo se introducirán elementos y detalles que complementan este planeamiento inicial como por ejemplo que el ancho de la columna de previsualización sea el mismo ancho que tienen los celulares [19] para poder visualizar el diseño del *widget* con la mayor fidelidad al producto que el usuario final verá en su pantalla.

5.6.3. Consultas SQL dentro de los widgets

Para el punto número **3** es importante entender la motivación y necesidad de que se pueda escribir SQL pudiendo tener otras alternativas para acceder a esa información como llamadas a una API REST o incluso accediendo a estructuras de datos ya declaradas en el contexto del *widget*. Los profesionales biotecnólogos, título que tienen la gran mayoría de los empleados del sector del laboratorio, junto con los data Scientist son los encargados de darle sentido

[19] Se tomo como referencia el modelo de iPhone mas popular entre la población *target* que se determino que serian potenciales usuarios de esta plataforma mediante estadísticas demográficas conocidas dentro de la empresa de antemano.

a los datos almacenados, encontrar relaciones entre sí y probar las hipótesis que luego deberán ser presentadas a los consumidores finales de alguna manera amigable a través de distintos *widgets*. Algo que se ha visto reflejado en otros desarrollos anteriores en la empresa, es que estos profesionales están acostumbrados desde su formación a realizar pruebas y relacionar datos utilizando alguna variante de SQL. Este requerimiento es entonces una manera de facilitarle el trabajo al editor de los *widgets* y permitirle que pueda insertar el código SQL para consultar datos que haya escrito algún biotecnólogo, de manera directa y sin mayores modificaciones. La experiencia nos ha demostrado que estas consultas suelen ser muy extensas y generalmente están escritas de una manera poco optimizada, con muchas subconsultas y técnicas de agrupamiento y cálculo que, para una persona con formación académica en ciencias de la computación, resultan poco elegantes. Este estilo de consultas y la cantidad masiva de información que suelen contener las tablas con las que se trabajan, han empujado al equipo de desarrollo del *IDE Web* a explorar maneras de mejorar la performance en las consultas sin alterarlas de manera directa.

5.6.3.1. Segmentación y aislamiento de información del usuario final

Para poder garantizar que el editor tenga la libertad de escribir SQL sin restricciones pero que únicamente tenga acceso a los datos del usuario final, se decidió aplicar una técnica llamada **segmentación de datos** donde mediante mecanismos de sincronización se toman los datos del usuario final y se copian a una base de datos exclusiva para ejecutar en ese entorno las consultas SQL requeridas. Esta técnica, aplicada en nuestro contexto, nos brinda como ventaja inmediata que no importa qué comandos SQL contenga la consulta, los datos ajenos al usuario final simplemente **no están** en esa base de datos, mitigando el riesgo de filtrado de información. Aunque esto es efectivo, nos impone un nuevo requerimiento y es que esas bases de datos aisladas por usuario tienen que mantenerse actualizadas con los últimos datos que puedan existir en la base de datos comunitaria.

Podemos diferenciar claramente tres operaciones que se van a realizar sobre estas nuevas bases de datos aisladas.

- Creación y sincronización inicial
- Consultas

- Operaciones de sincronización

Esta funcionalidad de SQL en los *widgets* se tornó una pieza fundamental del desarrollo, por lo que se brindan detalles al respecto posteriormente con mayor detenimiento.

5.7. Arquitectura del sistema e interconexiones

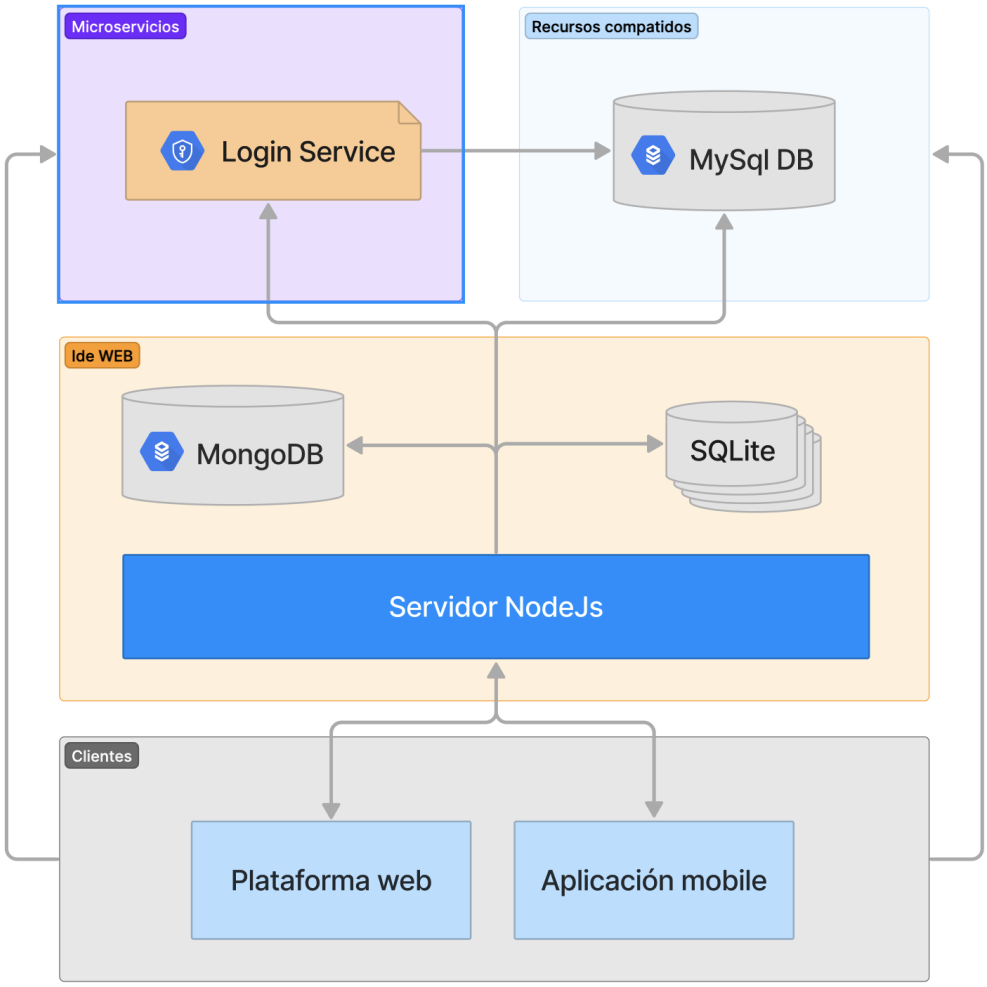

Figura 5.2: Interconexiones del Ide Web con otras partes del ecosistema

5.7.1. Recursos compartidos: Base de Datos MySQL

La empresa maneja toda la información de pacientes y clientes en una base de datos MySQL. Esta base de datos inicialmente tenía el único propósito de almacenar determinados datos de pacientes referidos a un estudio en particular pero con el paso del tiempo y el crecimiento de la empresa fue tomando un

carácter generalista y hoy posee información de variada índole y muchas tablas de gran tamaño.

El *IDE Web* tiene como dependencia esta base datos por varios motivos. Primero por el control de usuarios, la información de los editores del *IDE Web* y usuarios finales (pacientes/clientes) se encuentra almacenada en esta base de datos. También es aquí donde el *IDE Web* va a ir a buscar la información sobre estudios que potencialmente los editores usen para presentar al usuario; si bien el acceso no es directo, esta base de datos es la fuente de información primaria. Queda fuera del alcance de este documento la estructura total de la base datos, pero ciertas tablas serán presentadas oportunamente cuando sea necesario.

5.7.2. Servicios internos: Login Service

El *IDE Web* consume una API interna para el manejo de la autenticación de usuarios. Específicamente se implementa OAuth2.0 [19]. El equipo que mantiene la API distribuye un paquete de meteor mediante NPM que expone interfaz para hacer uso de los servicios de autenticación, creación y modificación de usuarios y los perfiles de los mismos. Al agregar ese módulo como dependencia, en nuestro proyecto podemos utilizar los métodos listados en el bloque de código 9.

```
1  Auth.prototype.getUser(userId, token);
2  Auth.prototype.getProfile(id, token);
3  Auth.prototype.createUser(user, profile);
4  Auth.prototype.updateUser(userId, username, password, profileAttributes, token)
5  Auth.prototype.updateProfile(userId, profileAttributes, token);
6  Auth.prototype.searchUsers(filter, token);
7  Auth.prototype.getUsers(token);
8  Auth.prototype.login(username, password);
9  Auth.prototype.resetPassword(token, userId, passwor);
10 Auth.prototype.changePassword(email, template, token);
11 Auth.prototype.emailAvailability(email);
```

Bloque de código 9: Métodos del cliente del Login Service disponibles

5.7.3. Servicios internos: CMS Service

La empresa cuenta con un microservicio dedicado al manejo de contenido CMS, el cual mediante una API Rest permite el acceso a texto localizado y

contenido estático. Una de las principales ventajas de esto es la posibilidad de realizar cambios en un texto, por ejemplo, sin la necesidad de desplegar nuevamente las aplicaciones que muestran dicho texto.

El acceso al contenido del *CMS Service* se realiza mediante *Métodos de meteor*, como se ven en el bloque de código 10, que abstraen las llamadas a la API y pueden ser utilizados, no solo por el frontend (Widgets o el *IDE Web* mismo) sino también por el *backend*, como por ejemplo el proceso que transforma los widgets en código *transpilado* para almacenarlo.

```
import logger from './logger';

const CMS_APP_ID = '...';
const CMS_API_URL = 'https://.../api/v1/localize';
const CMS_COMPONENT = 'external-partners';
const TTL_CMS_CACHE = 12;

class CmsManager {

  buildQueryString(component, locale) {
    return `${CMS_API_URL}?clientId=${CMS_APP_ID}&locale=${locale}&component=${component}`
  }

  async getKeysFor(component, locale = 'en-US') {
    return new Promise((resolve, reject) => {
      let url = this.buildQueryString(component, locale);
      logger.info({ type: 'cms-fetch-keys', userId: Meteor.userId() }, 'cms');
      HTTP.call('GET', url, {},
          (err, result) => {
            if(err){
              logger.info({ type: 'cms-query-error', error: JSON.stringify(err) });
              return reject(err)
            }
            resolve(result.data);
          })
    });
  }
}

Meteor.methods({
  async getTranslations(component = '', locale) {
    if (!Meteor.userId()) {
      return null;
    }

    const cms = new CmsManager();
    let apiResponse = await cms.getKeysFor(component, locale);
    return apiResponse.translations[CMS_COMPONENT] || [];
  }
})
```

Bloque de código 10: Clase y método de Meteor para acceder al contenido localizado del CMS

5.7.4. Login y manejo de sesión

La pantalla de login, si bien es simple como se ve en la figura 5.3, contiene todos los elementos necesarios para este tipo de secciones. Además del formulario de login, se encuentran presentes opciones como *Recuperar contraseña* y *Crear nuevo usuario*. Al momento de la escritura de esta tesina la funcionalidad para crear una nueva cuenta se encuentra inhabilitada, como se explica más adelante, ya que los usuarios se sincronizan desde la base de datos general de la empresa hacia la aplicación MeteorJs. Si bien esto es suficiente por el momento, está planeado proveer un formulario estándar de creación de usuario para desacoplar ambas bases de datos y poder tener un control de acceso más granular.

Durante la etapa inicial del servidor, MeteorJs define una función manejadora de login (ver bloque de código 11). Se realiza la captura de credenciales mediante el formulario de la pantalla de login.

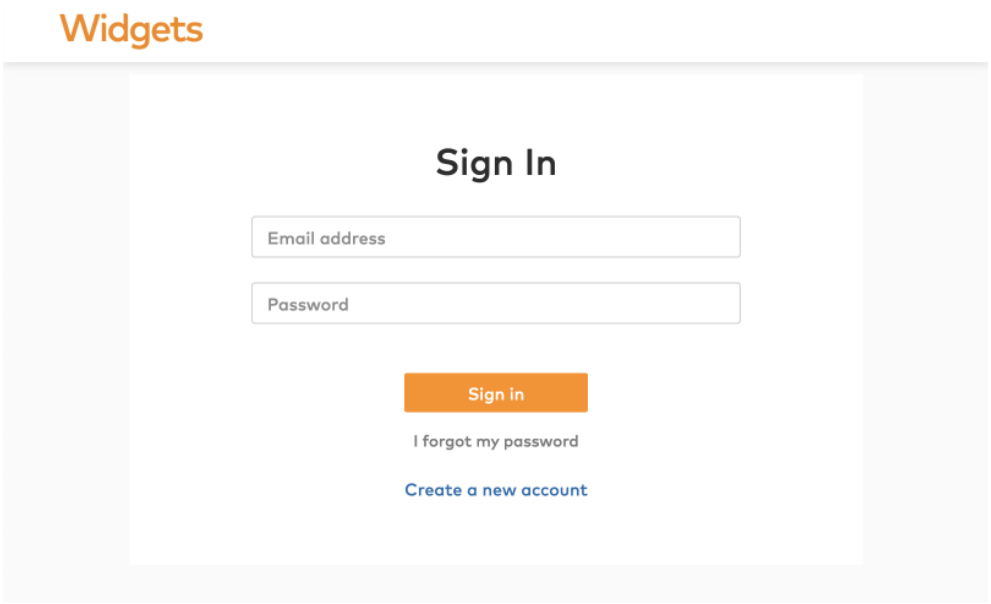

Figura 5.3: Pantalla de login

Se hace uso de la API del *Servicio de Login (5.7.2)*, la cual ante una respuesta positiva el servidor de Login extiende un *token* y se desencadena una llamada al módulo *Accounts* de MeteorJs para que se lleven a cabo las verificaciones necesarias.

```
1  const tokenLoginHandler = ({token, email}) => {
2    let _id = null;
3    let userId = null;
4    try {
5      const ref = jwt.verify(token, process.env.AUTH_SECRET)
6      userId = ref.user_id;
7      const attributesId = ref.document_id;
8      const accessToken = ref.access_token;
9      const mysqlUser = MysqlClient.syncQuery(
10         `SELECT id FROM users WHERE user_id = "${userId}"`
11     )[0]['id'];
12     const user = Meteor.users.findOne({ 'username': userId });
13     if (user) { // Usuario ya existe en la coleccion interna
14       _id = user._id;
15       Meteor.users.update({ _id }, {
16         $set: {
17           'profile.accessToken': accessToken,
18           'profile.authToken': token,
19           'createdAt': user.createdAt || moment.utc().format(),
20         }
21       });
22       logger.info({type: 'login', userId}, 'auth');
23     } else { // Usuario nunca antes visto en el ide, se agrega
24       const localUser = {
25         username: userId,
26         profile: { userId, mysqlUser, attributesId, accessToken, authToken: token }
27       };
28       logger.info({type: 'new', userId}, 'auth');
29       _id = Meteor.users.insert(localUser);
30     }
31     return { userId: _id };
32   } catch (error) {
33     logger.error({
34         type: 'failed',
35         userId,
36         error: JSON.stringify(error, Object.getOwnPropertyNames(error))}, 'auth'
37     );
38     return null;
39   }
40 }
```

Bloque de código 11: Función de verificación de token y sincronización del usuario entre bases de datos

Se puede observar en el bloque de código 11 que no solamente se verifica el *token* sino que también se consulta a la base de datos compartida MySQL (linea 9) sobre el usuario. Si el usuario en cuestión todavía no existe en la

colección de usuarios de MeteorJs (línea 12) entonces se crea (linea 48); en caso de ya encontrarse en la colección, sus datos son actualizados para reflejar su sesión actual (línea 34). El resto del código se encarga de registrar en el *log* los distintos eventos para poder tener visibilidad en caso de errores y también para estar en línea con las regulaciones de log que impone HIPAA.

5.7.5. Ciclo de vida de un Widget

5.7.5.1. Composición y anidamiento

Durante las primeras presentaciones a los gerentes técnicos de la empresa sobre el avance del proyecto surgió un requerimiento que inicialmente había sido descartado porque se consideró que su funcionalidad superaba el alcance del MVP que se estaba realizando, pero la gerencia decidió que era necesario para probar los límites del producto y poder ampliar las estrategias de marketing previstas.

Los cambios se refieren a cómo los *widgets* se denominarían ahora *Stories* y pasarían a formar parte de otros *widgets* llamados *Apps*. Es decir que las *App* serían la unidad *publicable*, y las *Stories* conformarían la unidad ejecutable, pudiendo haber muchas *Stories* en una *App*. La mejor analogía para esta estructura es imaginar una aplicación móvil con distintas *tabs*, en donde cada *tab* seria una *Story* de la *App*. Este cambio nos permite por ejemplo tener una *App* relacionada con algún aspecto médico en particular, como diabetes u otra enfermedad crónica, y tener dentro de esa *App* un numero variables de *Stories* donde en cada uno se trata o se muestra una temática especifica de dicha enfermedad.

Podemos entonces agregar a la lista de requerimientos iniciales los siguientes:

- Como editor puedo crear *Apps*
- Como editor puedo definir la lista de *Stories* que conforman una *App*
- Como editor puedo modificar las características de una *App* como el título, icono, descripción, etc.

5.7.5.2. Posibles estados

Los posibles estados de un *widget* marcan distintas etapas de revisión y publicación.

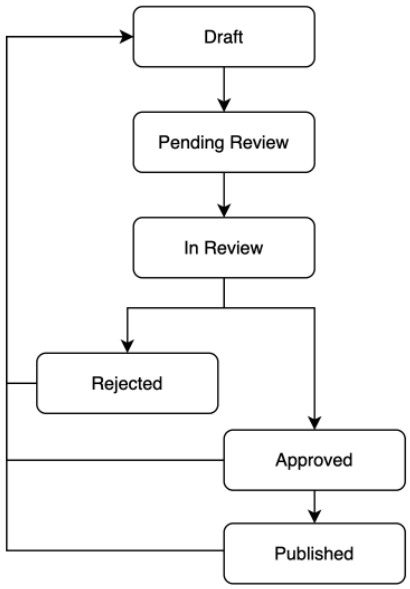

Figura 5.4: Diagrama de estados posibles de un widget

Como se muestra en el diagrama de estados posibles (Figura 5.4) se utiliza el estado **draft** como punto de partida pero también es el único estado del *widget* que permite modificar el código escrito, es decir, si la *Story* está en proceso de revisión (**In review**) y el *auditor* pide que se realice un cambio, al ingresar a modificar el código el *IDE Web* advertirá al editor que la *Story* automáticamente se moverá al estado **draft** si se modifica alguna de las propiedades de la *Story*, como el título o el código, etc.

5.7.5.3. Pantalla de inicio

Luego del login el usuario ingresa finalmente al sistema, en la Figura 5.5 se pueden observar tres grupos de elementos.

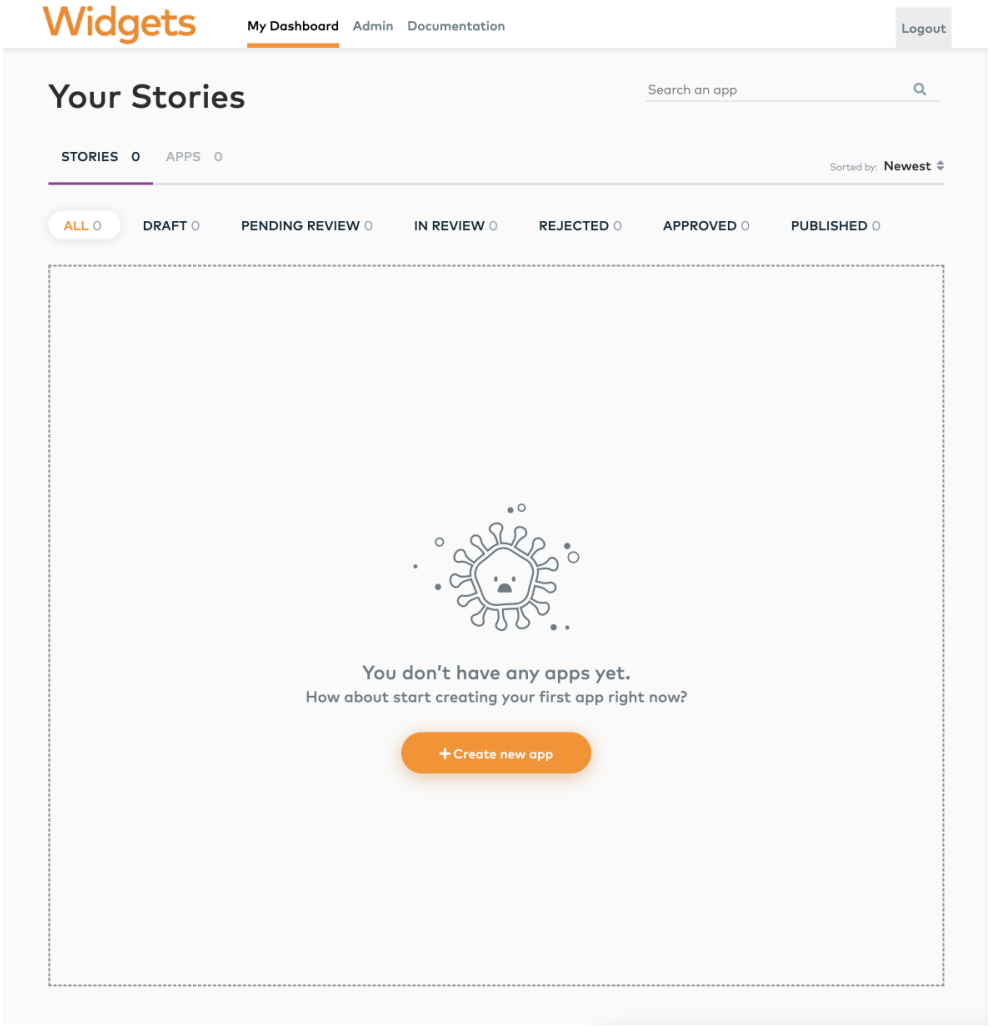

Figura 5.5: Pantalla de inicio

Cabecera Mostrando el logo y enlaces hacia el panel de administración y la documentación. Ambas secciones se detallarán más adelante.

Barra de navegación Donde se presenta un buscador, un filtro rápido para visualizar solo *Stories* o *Apps* y un selector para elegir mediante que criterio se ordenaran los ítems a mostrar. Se puede observar también

una serie de botones que permiten filtrar aún más los ítems mostrados, dependiendo del estado que éstos tengan.

Contenido Inicialmente el espacio está vacío, invitando al usuario a tomar la iniciativa y crear la primera *app*. Conforme se vayan creando nuevos elementos se irán mostrando en esta área.

La pantalla de inicio está pensada para permitir al usuario acceder de manera inmediata a funcionalidad frecuente como puede ser editar una *app* y/o filtrar la lista mostrada para facilitar la identificación. Cada *app* puede tener varios *stories* y cada *story* esta representada con una tarjeta (figura 5.6) en la pantalla de inicio. El número de tarjetas puede crecer fácilmente al administrar más de una *app*, por lo que el filtrado tiene un papel importante en la experiencia del usuario al utilizar esta página en particular.

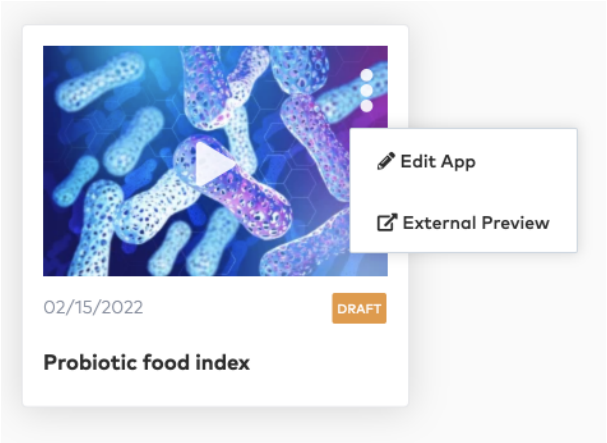

Figura 5.6: Tarjeta de un *widget* en la pantalla de inicio junto con el menú contextual desplegado

5.7.5.4. Creación y edición

La creación y edición se realizan utilizando el mismo formulario (Figura 5.7), el cual permite al usuario definir una serie de propiedades necesarias para poder satisfacer varios aspectos del negocio. Las propiedades básicas son **nombre** y **descripción** pero existen otras que son específicamente requeridas para la presentación u organización del *widget*.

A continuación se describen las propiedades que definen al *widget*:

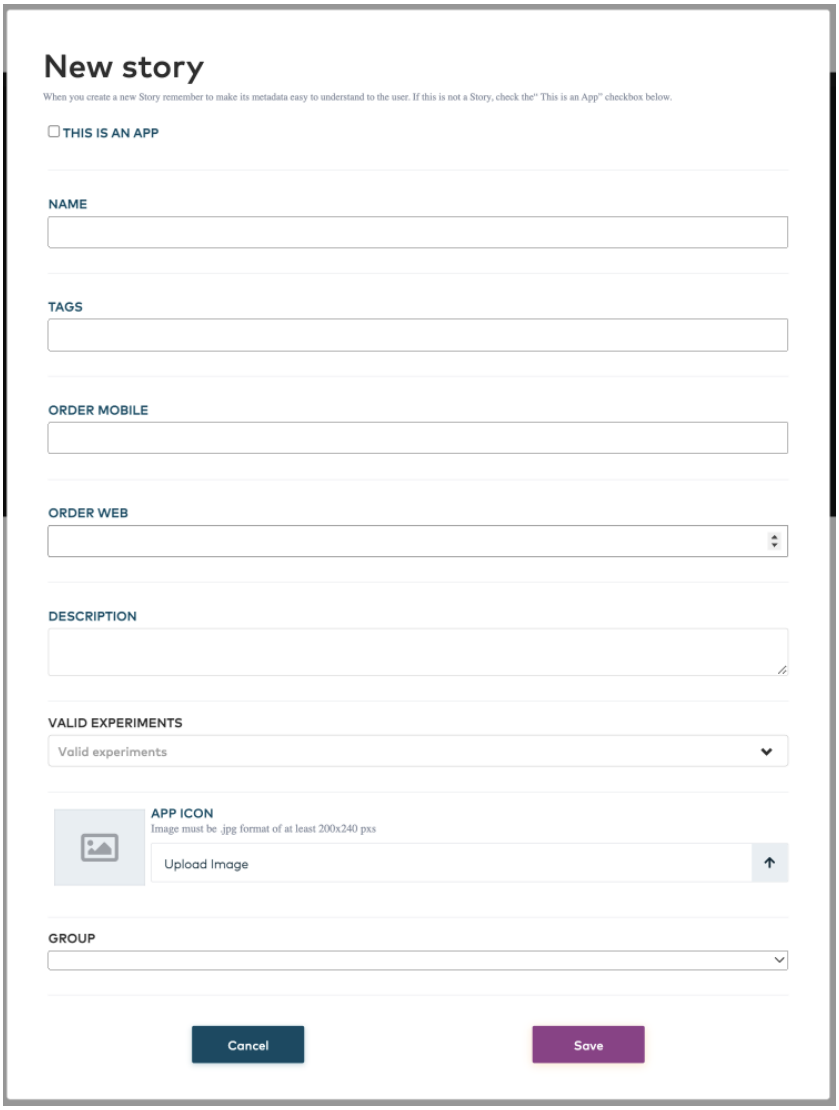

Figura 5.7: Modal de creación de widget

Tags Los *tags* o etiquetas en español permiten agrupar lógicamente *widgets* bajo la misma etiqueta. En la figura 5.7 se observa que existe un único *input*, en donde al escribir se buscan los tags existentes dentro del grupo correspondiente al editor y permite seleccionar dichos tags desde una lista, o escribir nuevos tags que serán creados al enviar el formulario.

Order Mobile Este campo exclusivamente numérico representa el *peso* de la aplicación en los listados que se muestran en el cliente *mobile*; mientras más grande es el número representando el peso, más abajo en el listado

estará.

Order Web El mismo concepto pero para los listados presentados en el cliente web.

Valid experiment Presenta un listado con los experimentos que están registrados en la base de datos para poder relacionar las aplicaciones que usan esos experimentos y así publicitarlos a los usuarios finales.

App Icon Es una imagen que se puede subir para representar el *widget* dentro de la plataforma web.

Group Para los editores que están registrados en varios grupos, este selector permite indicar a cual pertenecerá este nuevo *widget*.

5.7.5.5. Características heredadas

El *IDE Web* provee algunas herramientas a los editores para facilitar determinadas operaciones. Una de las herramientas más complejas desarrolladas es el acceso a las bases de datos compartidas.

Se utilizó la jerarquía de clases de ReactJs para poder nutrir a los *widgets* de funcionalidad extendida.

```
export default class WidgetComponent extends Component {
  constructor(props) {
    super(props);
    this.user = Meteor.user();
    this.api = {
    ... // Manejadores de acceso a distintas API's
    }
  }

  query(sql) {
    return new Promise((resolve, reject) => {
      const currentTime = moment.utc();
      const { title: identifier } = this.props;
      try {
        Meteor.call('runQuery', sql, (error, result) => {
          if (error) {
            return reject(error);
          } else {
            durationLogger({
              type: 'frontend-runQuery',
              identifier,
              duration: moment.duration(moment.utc() - currentTime).asMilliseconds(),
              query: sql
            });
            return resolve(result);
          }
        });
      } catch (error) {
        reject(error);
      }
    });
  }
}
```

Bloque de código 12: Método agregado a la jerarquía de clases de componentes ReactJs para facilitar acceso a la base de datos

```
/**
 * The metabolism related members.
 * @type {Array}
 */
const METABOLISMS = ['carbohydrates', 'lipids', 'protein'];

/**
 * Renders a page that shows insight about body weight for the user.
 */
export default class Metabolism extends WidgetComponent {
  constructor(props) {
    super(props);
    this.state = {
      selectedSamplesResults = [],
    }
  }

  async componentDidMount() {
    try {
       const selectedSamplesResults =  await Promise.all(this.getSelectedSamplesResults());
       this.setState({ selectedSamplesResults });
    } catch (e) {
      console.error('Err', e);
    }
  }

  /**
   * Fetch the abundance of carbohydrates, lipids and amino acids related bacteria
   * for the group of Selected Samples.
   * @return {Array<Promise>}
   */
  getSelectedSamplesResults() {
    return METABOLISMS.map(metabolism =>
      this.query(
        `SELECT COALESCE(avg_percent, 0.0) AS '${metabolism}'
        FROM condition_reference
        WHERE classification = '${metabolism}'
        AND condition_name = 'selected_samples';`,
      ),
    );
  }
}
```

Bloque de código 13: Ejemplo de uso del método query heredado de WidgetComponent

En el cuerpo de un *widget* se debe extender de la clase *WidgetComponent* (bloque de código 12 donde se muestra la definición del método *query*) en lugar de hacerlo de la clase ReactJs *Component*. Eso es suficiente para tener acceso a los métodos y propiedades de clase especiales. En el ejemplo 13 se observa como se puede realizar una consulta asíncrona para conseguir los valores deseados.

5.8. Desarrollo de un *widget*

La pantalla principal donde se desarrollan los *widgets* tiene un formato particular, en la figura 5.8 se puede observar la división entre cuatro áreas muy bien definidas. En la parte superior se muestra información básica del *widget*, tales como el título y botones para manejar el estado del mismo.

La parte central está completamente dedicada a la escritura del código del *widget*. Dos paneles opcionales completan la disposición principal. El panel de previsualización se extiende a lo largo por la parte derecha y tiene un ancho fijo para que el *widget* se muestre con la mismas capacidades de espacio que en un dispositivo móvil. El otro panel opcional es en el que se escriben los estilos.

Figura 5.8: Pantalla de edición de un *widget*

5.8.1. Estilos y su procesamiento

Como toda aplicación web, los *widgets* constan de dos partes fundamentales: el comportamiento y la presentación. El comportamiento está gobernado por el código React que escribe el editor pero la presentación está principalmente regida por el estilo, es decir el código CSS que dice cómo se tienen que ver los componentes en pantalla.

Los *widgets* como entidad en la base de datos tienen una propiedad dedicada a almacenar el código CSS. Al momento de *renderizar* un *widget* se agrega su CSS a los estilos de la página para que sea posible utilizar las propiedades allí definidas.

Una de las características que se decidió implementar desde el inicio del proyecto fue darle al editor la libertad de escribir el *widget* utilizando el lenguaje que quisiera. El mismo principio se utilizó al idear como los editores iban a escribir los estilos. Si bien CSS es la herramienta universal, existen dos alternativas que agregan a CSS muchas herramientas y capacidades que nativamente no son posibles o son tediosas de implementar. SassSyntactically Awesome Style Sheets [Hoja de estilos sintácticamente asombrosas] `https://sass-lang.com/` y Less[20] tiene un gran número de seguidores y el equipo de desarrollo del *IDE Web* posee amplia experiencia utilizando sass en otro proyectos, incluso en el *IDE Web* mismo.

En términos de UX darle al editor la facilidad de escribir CSS puro o alguna de sus variantes más poderosas, simplemente se tradujo en agregar un selector en la pestaña de los estilos.

El cliente envía el código de estilos al backend Mediante la misma llamada que se realiza al intentar almacenar un *widget*.

El backend procesa el estilo Del mismo modo que el código ReactJs es *transpilado* en javascript, el código sass o less es compilado en CSS. Este proceso se realiza mediante un par de librerías de node que hacen de interfaz con ejecutables que procesan el código fuente y retornan CSS puro. Si el editor eligió usar CSS desde el selector en la UI entonces no se realiza ninguna transformación.

[20] It's CSS, with just a little more. [Es CSS, con algo mas] `https://lesscss.org/`

5.8.1.1. Previsualización y renderizado

Como todo el esfuerzo de procesamiento de javascript y CSS involucraba una llamada al backend para obtener código *usable* por el navegador, la previsualización del *widget* siempre sucedía luego de que este se guardara en la base de datos.

5.8.2. Usando datos de prueba

Otro de los complementos para el editor que se pensó de antemano fue la capacidad de tener conjuntos de datos a disposición del editor. De esta forma, es posible probar el *widget* desde el punto de vista de algún usuario con determinadas características en su información presente en la DB.

Partiendo desde ese requerimiento se decidió implementar una versión reducida que sirva como prueba de concepto y luego, en una fase posterior, desarrollar esta característica en su totalidad.

La primera implementación implicaba tener un botón en el *IDE Web* que marcara al usuario actual con una bandera de prueba y en el backend *Meteor* se encargara de precargar la base de datos *SQLite*. Contra esta base de datos se corrían las consultas, usando varios archivos *.json* seleccionados previamente. Estos archivos fueron generados recolectando distintos conjuntos de información anónima de varios usuarios de prueba que la empresa posee y se han usado para el desarrollo de otros productos.

El punto negativo que tenía esta implementación es que tiene una naturaleza binaria (se carga todo o no se carga nada), por lo que el perfil del usuario no representaba un caso de uso real ya que podía tener datos que naturalmente son mutuamente excluyentes.

En la revisión siguiente se espera que ese simple botón en la UI se sustituya con un selector más complejo que permita listar y elegir entre varias condiciones y características identificadas y luego el backend se encargaría de importar esos conjuntos de datos en la base de datos personal del usuario logueado.

5.9. Consultas a la base de datos

Dentro de los desafíos iniciales 5.6.3.1 se habló sobre la necesidad de poder realizar consultas SQL de manera directa y cómo la técnica de aislamiento de datos nos facilitaba no solo poder darle al editor la libertad de escribir las consultas libremente, sino que también garantizaba la privacidad de los datos.

Para probar en la práctica que esta técnica efectivamente sería válida se decidió hacer una prueba de concepto (POC). Para esta prueba se utilizó SQLite debido a que es sumamente fácil programáticamente crear una DB nueva y, al estar estas expresadas en archivos discretos, se logra satisfacer el requerimiento de aislamiento sin esfuerzo.

La estrategia consiste en términos generales en obtener la información que existe sobre un usuario en particular y crear una nueva base datos con solo esta información, respetando la estructura de tablas existente. Conforme el equipo de desarrollo avanzaba con la implementación se hizo evidente que esta estrategia resultaba incompleta y planteaba muchas particularidades, que tuvieron que ser manejadas en iteraciones posteriores.

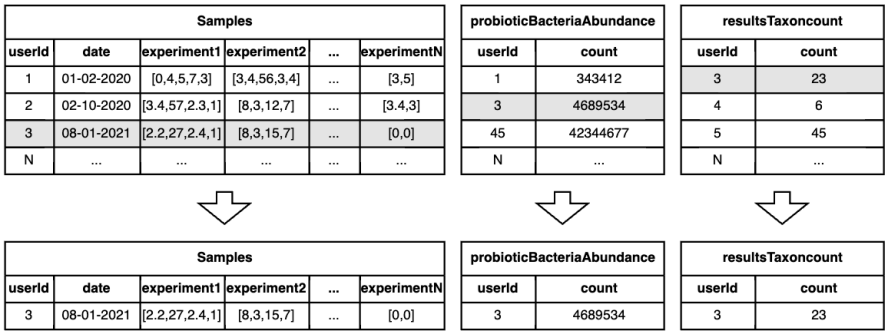

Figura 5.9: Ejemplo de segmentación y aislamiento de la información de un usuario en particular

5.9.1. Creación e inicialización de la base de datos

La primera versión de este sistema de sincronización entre la base de datos real y la basada en archivos SQLite fue lo suficientemente sofisticada como para demostrar la viabilidad del concepto, pero bastante rudimentaria en términos de optimización. La implementación consistía en:

- Cuando el usuario *widget* requería una consulta SQL se procedía de

manera secuencial a consultar cada tabla de la base de datos por la información del usuario y a crear la contraparte de la misma en una base de datos SQLite.

- Al terminar todo el procesamiento anteriormente descrito se continuaba con la ejecución de la consulta sobre esta nueva base de datos.
- Los datos eran enviados al cliente para que éste hiciera mas procesamiento si era necesario, y finalmente se mostraban al usuario final.

Esta primera implementación dejó en evidencia una serie de falencias.

1. La idea de copiar **toda** la información con la que se cuenta de un usuario resultó inviable, ya que podía tardar varios segundos y en casos de usuarios con muchos registros, llegaba a tardar varios minutos en completarse.
2. Que el proceso se ejecutara cuando el usuario requería información de la base de datos no daba oportunidad al sistema de poder prepararse de antemano.
3. Con cada ejecución del *widget* se requerían los mismos datos y generalmente estas consultas no solo extraen datos sino que hacen cálculos estadísticos y agrupaciones complejas que agregan tiempo de procesamiento (ver *Consultas SQL dentro de los widgets*).

Como solución al punto 1 y 2 se decidió implementar en la clase *WidgetComponent* (la clase padre de todos los *widgets*) un método que permita al editor nombrar que tablas específicamente se van a considerar necesarias para que el usuario final pueda acceder a la información requerida. Esta declaración explícita de las dependencias de un *widget* se debía hacer en el momento previo al 'montaje' del componente ReactJs, entonces permitía que todo proceso de sincronización inicial sucediese al instante que el usuario final ejecutase el *widget*, sin necesidad de estar vinculado a la acción específica de realizar una consulta. En muchos casos, el momento de montaje del *widget* y el momento en el que la consulta era requerida difería, porque esta última estaba generalmente asociada a una acción del usuario final, como seleccionar un ítem de una lista o apretar un botón, entonces la *performance* percibida era incluso mayor haciendo todo el proceso de carga de datos 'transparente'.

5.9.1.1. Precarga en el cliente

Como se muestra en el bloque de código 14, uno de los métodos que heredan los componentes es una función asíncrona que permite controlar la carga de las tablas que el *widget* haya listado como dependencias.

```
export default class WidgetComponent extends Component {
  ...
  preloadTables(tables) {
    const currentTime = moment.utc();
    const { title: identifier } = this.props;
    return new Promise((resolve, reject) => {
      try {
        Meteor.call('preloadTables', tables, (error, result) => {
          if (error) {
            return reject(error);
          }
          durationLogger({
            type: 'frontend-preloadTables',
            identifier,
            duration: moment.duration(moment.utc() - currentTime).asMilliseconds(),
            tables
          });
          return resolve(result);
        });
      } catch (error) {
        reject(error);
      }
    });
  }}
}

// Ejemplo de uso en un widget

const REQUIRED_MYSQL_TABLES = ['samples', 'resultsTaxoncount'];

export default class AlcoholMetabolism extends WidgetComponent {
  ...
  componentWillMount() {
    async () => await this.preloadTables(REQUIRED_MYSQL_TABLES);
    this.setState({ loading: false });
  }

}
```

Bloque de código 14: Método del cliente para la precarga de tablas

5.9.1.2. Precarga en el servidor

Para continuar con la explicación de la forma en que se precargan las tablas debemos hablar sobre el Método de meteor*preloadTables* (Bloque de código 15). El mismo se ejecuta en el contexto del servidor y se encarga de inicializar la base de datos SQLite que contendrá toda la información segmentada para que se usen en los *widgets*. Esta información se irá cargando con los datos a medida que los *widgets* que requieren información sean ejecutados.

```
async preloadTables(tables = []) {
  if (!Meteor.userId()) {
    return null;
  }
  const username = Meteor.user() && Meteor.user().username;
  const dbManager = new LocalUserDb(Meteor.user().profile.mysqlUser);
  try {
    await dbManager.initialize();
    return await dbManager.preloadTables(tables);
  } catch (err) {
    let error = err.message
      ? err.message.replace('SQLITE_ERROR', 'DB QUERY ERROR')
      : '';
    logger.info({
      type: 'db-preload-error',
      error: JSON.stringify(error, Object.getOwnPropertyNames(error)),
      username
    });
    throw new Meteor.Error('db-preload-error', error);
  } finally {
    dbManager.close();
  }
 }
```

Bloque de código 15: Método de meteor del lado del servidor que precarga las tablas

5.9.1.3. Inicialización de la base de datos SQLite

El *Método de meteor* que se encarga de preparar la información para que pueda ser consultada tiene como primer paso la creación de la base de datos SQLite que contendrá las tablas. El archivo que representa la base de datos lleva como nombre el resultado de una función Hash md5 cuyo valor de entrada es el *id* del usuario.

Este método verifica que exista el archivo y, en caso de no existir, lo crea y le agrega la tabla *metadata* como se puede ver en el bloque de código 16.

```
/**
 * Initialize the tables and populate them with current data if needed
 */
async initialize(onDemand = false) {
  let shasum = crypto.createHash('md5');
  shasum.update(`db_${this.userIdMysql}`);
  let filename = shasum.digest('hex');
  let filepath = `${process.env.STORAGE_USER_DB_PATH}${filename}.db`;
  let shouldInitialize = !fs.existsSync(filepath);
  this.db = null;
  await new Promise((resolve, reject) => {
    this.db = new sqlite.Database(
      filepath,
      sqlite.OPEN_READWRITE | sqlite.OPEN_CREATE,
      err => {
        if (err) {
          reject(err);
        } else {
          resolve();
        }
      },
    );
  });
  this.db.serialize();
  if (shouldInitialize) {
    await this._setMetadata();
  }
  this.db.parallelize();
}
```

Bloque de código 16: Método de creación de la base de datos SQLite por cada usuario

5.9.1.4. Tabla metadata

Dentro de cada base de datos por usuario, además de las tablas equivalentes a las de la base de datos original también se agrega una tabla llamada **metadata** para registrar distintos valores que serán útiles para mantener los datos actualizados. La tabla tiene una estructura muy sencilla de *clave-valor* que permite guardar información variada sobre la base de datos, la última actualización y también otros datos que necesitemos luego.

La tabla es creada mediantes las consultas correspondientes (ver bloque de código 17) en el momento de crear la base de datos por usuarios.

```
1  async _setMetadata() {
2    let db = this.db;
3    await this._run(
4      `CREATE TABLE IF NOT EXISTS metadata
5          (name TEXT PRIMARY KEY, value TEXT NOT NULL)`,
6    );
7    await this._run(
8      'INSERT INTO metadata VALUES ("updated", ?)',
9      moment().format('MM-DD-YYYY'),
10   );
11 }
```

Bloque de código 17: Método de creación de la tabla metadata

5.9.2. Precarga de tablas

5.9.2.1. Versionado de tablas

Se puede dar el caso en que se quiera modificar la estructura de una tabla dentro de las bases de datos SQLite que ya existen. Por esto, antes de ejecutar una consulta que requiera una tabla desactualizada, hay que eliminarla y volver a crearla utilizando la nueva estructura, para que quede acorde a los últimos cambios. Para mitigar este caso particular, se decidió mantener un listado de tablas y versiones dentro del código y así llevar un registro de los cambios de estructura. La versión es simplemente un número que se va incrementando a medida que se requiera actualizar las tablas ya cargadas.

```
1 export const TABLES_VERSIONS = {
2   'samples': 1,
3   'functions': 2,
4   'lactoseIntoleranceMicroorganisms': 1,
5   'probioticFoodsUserAbundance': 2,
6   ...
7 }
```

Bloque de código 18: Lista de tablas y versiones

Siguiendo con el flujo de ejecución del método de MeteorJs, el siguiente punto es la precarga de tablas, que a su vez implica varios pasos:

El primer punto en el bloque de código 19 es la carga de los nombres de todas las tablas que existen en la base de datos; para esto se consulta la tabla sqlite_master y mediante una condición simple se excluye a la tabla metadata. El nombre de la tabla se transforma para evitar caracteres prohibidos y poder respetar un formato común. Luego se calcula la diferencia entre la lista de tablas que requerimos tener cargadas y las que ya están cargadas para evitar duplicar esfuerzos. También se obtiene la lista de tablas requeridas que están desactualizadas; esto se logra consultando el nombre de todas las tablas y comparando la versión registrada en *metadata* para la tabla involucrada. Si la versión difiere de la que existe en la constante TABLES_VERSIONS (ver bloque de código 18) se considera como desactualizada y se agrega a la lista de tablas a borrar. Una vez eliminadas, se agregan los nombres de esas tablas a la lista de tablas a precargar, para que vuelvan a crearse con sus datos y su estructura actualizadas.

El siguiente paso es obtener por cada tabla las consultas que se necesitan llevar a cabo y luego de manera secuencial ir ejecutándolas para poder dejar

```
async preloadTables(tables) {
  const currenTables = await this._getAllLoadedTables();
  const normalizedCurrentNames = currenTables.map(sanitizeTableNames);
  const outdatedTables = await this._getAllOutdatedTables();
  const normalizedOutdatedTables = outdatedTables.map(sanitizeTableNames);
  const tablesToRefresh = _.uniq(_.difference(tables, normalizedOutdatedTables));
  let tablesToLoad = _.uniq(_.difference(tables, normalizedCurrentNames));

  if (tablesToRefresh.length) {
    await this._deleteTableContent([...tablesToRefresh]);
    // Add the recently deleted tables to the list
    tablesToLoad = [...tablesToLoad, ...tablesToRefresh];
  }

  if (tablesToLoad.length == 0) return Promise.resolve(); // Nothing to do

  // Get all the queries to run
  const jobs = this._syncTables(tablesToLoad);

  return Promise.all(jobs).then(async (allTheQueries) => {
    try {
      return await this._runSync(_.flatten(allTheQueries));
    } catch (error) {
      logger.info({ type: 'db-preload-error', error: JSON.stringify(error) });
      throw new Meteor.Error('db-preload-error', error.message);
    } finally {
      this.db.close();
    }
  })
}
```

Bloque de código 19: Función principal de precarga

la base de datos SQLite actualizada.

5.9.2.2. Funciones de precarga para cada tabla

Para poder precargar la información de una tabla en particular se decidió crear una estrategia que permitía mantener las consultas que debían de ejecutarse para los distintos pasos de **creación**, **consulta** y **borrado** de la tabla.

La etapa de creación simplemente consiste en ejecutar la consulta que define las columnas e índices de la tabla. La etapa de borrado únicamente realiza una consulta del tipo:

`DROP TABLE IF EXISTS`.

Según se puede apreciar en la figura 5.9 la segmentación de datos es sumamente importante para mantener la privacidad de los usuarios, entonces es en la etapa de consulta donde se seleccionan exclusivamente los datos del usuario, y es en este momento donde se puede realizar cálculos y funciones estadísticas para exponer esa información en la nueva tabla para futuras referencias. Por ejemplo, por cuestiones de HIPAA no podemos mostrar datos de otros usuarios que no sea el que está ejecutando el *widget* en este momento, pero sí podemos aportar datos sobre la forma en que se compara la información del usuario con otros que tenemos en el sistema. Esta métrica comparativa puede ser simplemente un percentil que permita al usuario evaluar un determinado dato y tener una idea de qué tan común es dentro de la población.

Una función de *precarga* como la de la figura 20 tiene como resultado un arreglo de consultas. Se puede apreciar también que se hace uso de *helpers*[21] para registrar el tiempo que toma la consulta a la base de datos original, para luego identificar posibles áreas de optimización.

[21]Funciones que encapsulan algún procesamiento o dato especifico

```
 async sweetenersBacteriaUsersAbundance(userId) {
  const { escape, query } = MysqlClient;
  const tableName = "sweeteners_bacteria_users_abundance";
  const beginingTime = mysqlSyncLogger.loading(tableName);

  const sql = `
    SELECT
      ssr,
      (SUM(r.abundance) * 100) / 5.167 AS abundance
    FROM (
      SELECT
        rt.count_norm / 10000 as abundance,
        rt.ssr
      FROM results_taxoncount AS rt
      INNER JOIN samples AS s ON s.sequencing_revision = rt.ssr
      AND rt.taxon IN (543, 28221, 201174)
      AND s.user = ${userId}
    ) AS r
    GROUP BY ssr;
  `;
  const result = await query(sql);
  const purge = `DROP TABLE IF EXISTS ${tableName}`;
  const creation = `CREATE TABLE ${tableName} (
    id INTEGER PRIMARY KEY,
    ssr INTEGER,
    abundance REAL
  )`;

  if (!result.length) return [purge, creation];

  const inserts = result.map( row => `( NULL, ${escape(row.ssr)}, ${escape(row.abundance)}) `)

  mysqlSyncLogger.loaded(tableName, beginingTime);

  return [
    purge,
    creation,
    `INSERT INTO ${tableName} VALUES ${inserts.join(", ")}`
  ];
},
```

Bloque de código 20: Funciones de creación, consulta y borrado para una tabla de ejemplo

5.9.3. Limitaciones de la implementación

Si bien la estrategia de precargar las tablas bajo demanda funcionaba muy bien, se identificó una limitación clave que provocaba que la experiencia del usuario sea inferior a la aceptable.

Las consultas complejas, y muchas veces mal optimizadas, tardaban un tiempo considerable y en ocasiones se calculaban varias veces los mismos valores durante la sesión del usuario. Esto sucedía generalmente en las consultas que se realizaban al cargar alguna pantalla del *widget*. Al mostrarse la consulta en pantalla, si el usuario continuaba navegando y luego volvía, el pedido de datos se volvía a disparar y nuevamente se calculaba el resultado. Este cálculo no solo consistía en recuperar información de las tablas de la base de datos sino que generalmente contenía cálculos estadísticos o subconsultas.

La solución se implementó de la siguiente manera: cada vez que se intentara ejecutar una consulta, la cadena de caracteres que representaba la consulta se enviaba a una función de hash (md5) para que se le asigne un valor relacionado a esos caracteres, en ese orden en particular. De esta forma, todas las consultas que fueran exactamente las mismas, darían como resultado de esa función el mismo valor. Este valor se utilizaba como índice para hacer una consulta a un servidor de Redis, si la consulta ya existía se recuperaban y se devolvían los datos del resultado guardado. En el caso que no existiese ese índice, se procedía a realizar la precarga (de ser necesaria) y a ejecutar la consulta propiamente dicha; antes de devolver el control al usuario se insertaba el valor del resultado en Redis y se le asignaba un tiempo de vida en el orden de la hora para que luego de ese periodo de tiempo el *caché* se invalide automáticamente y así forzar una actualización que mantenga los datos actualizados con el siguiente pedido de ejecución para la misma consulta.

Las pruebas realizadas mostraron un cambio substancial en la velocidad percibida de los *widgets*. Si bien algunas consultas era 'pesadas', muchas de ellas se revisaron en búsqueda de malas prácticas y se optimizaron.

5.10. Desarrollos secundarios

5.10.1. Plataforma móvil

El *cliente mobile* (para plataformas móviles) ha sido siempre considerado el cliente principal de los *widgets*. Se puede encontrar evidencia de este dogma en varias decisiones tomadas a lo largo del desarrollo del *IDE Web*. Si bien no es un proyecto secundario desde el punto de vista del producto, sí lo es ante la visión de esta tesina y es por eso es que se decidió referirse a él solamente de manera superficial.

Las tecnologías resultan muy similares entre ambos proyectos, lo que facilita la comprensión del funcionamiento y la mecánica de las interacciones con los usuarios finales.

Ya resueltos los desafíos iniciales y los principales problemas que se presentaron al desarrollar el *IDE Web*, el esfuerzo de encarar la contraparte para celulares fue enfocado desde el punto de vista de la experiencia del usuario, es decir, que el objetivo estaba mucho más claro al momento de plantear la UI y los diseñadores pudieron dar prioridad a las interacciones.

Esta plataforma móvil, además de ser cliente del *IDE Web* iba a permitir a la empresa publicitar y comercializar otros productos ya existentes.

Dentro de los requerimientos que se relevaron inicialmente se encontraban:

- La experiencia entre la web y *mobile* al utilizar los *widgets* debería ser lo más homogénea posible.
- Cualquier solución debe ser compatible con las últimas versiones de iOS. Android se dejaría como un esfuerzo secundario. Esta decisión estaba basada en la información estadística demográfica con la que se contaba sobre el usuario apuntado. Muchas veces iOS y Android requieren implementar soluciones específicas para cada plataforma, incluso a veces teniendo que escribir código nativo en su respectivo lenguaje, por lo que la perspectiva de tener que utilizar una sola de las plataformas resultó una buena noticia.
- Los usuarios tienen que poder visualizar la lista completa de *widgets* disponibles y al ejecutarlo por primera vez tiene que presentarse una ventana modal con la descripción del mismo y una leyenda que explique cómo se usan los datos previamente obtenidos y un botón para dar consentimiento y continuar.

Los desafíos técnicos estaban más concentrados en aprovechar al máximo las capacidades existentes del *IDE Web* y poder lograr una amalgama coherente entre ambas plataformas, teniendo siempre en cuenta que los *widgets* que ya estaban escritos y funcionando para la plataforma web deberían de sufrir la menor cantidad posible de alteraciones y así poder ejecutarse de una manera similar en este nuevo cliente. Al momento de diseñar la idea original ya se habían tenido en cuenta muchas consideraciones para tratar de garantizar esta compatibilidad. En especial, se hizo mucho hincapié en trazar un plan para integrar el cliente *mobile* a otros aplicativos de terceros, dispositivos Wearables y accesorios de salud.

5.10.1.1. React Native y la estrategia general

Al inicio del proyecto se tenía una idea muy superficial de la forma en la que se iban a integrar ambas plataformas. En principio la solución que se tenía en mente era hacer que los componentes React del *widget* se renderizaran dentro del contexto de React Native. Esta idea no estaba tan mal, ya que el grado de compatibilidad entre los dos dialectos es casi perfecto, pero el problema era cómo se manejaban los casos donde la esta compatibilidad no existía.

Se decidió aprovechar y hacer uso del mismo previsualizador que se usa en el *IDE Web*. Es decir, un documento HTML con todo lo necesario para que el *widget* se ejecute con los estilos correspondientes. Entonces, desde el punto de vista del *desarrollo mobile*, solo necesitaríamos una ventana que permita cargar una URL conocida y mostrarla.

5.10.1.2. Webview

Webview es un tipo de componente nativo de iOS, representado en el lenguaje *swift* como un objeto *WKWebView*[22]. React Native tiene una abstracción que permite hacer uso de las capacidades de un webview y mostrarlo en pantalla. Si tuviéramos que buscar un paralelismo entre la plataforma web tradicional y la mobile, un Webview seria un *iframe*, una

[22]WKWebView, An object that displays interactive web content, such as for an in-app browser. [MKWebView, un objeto que muestra contenido web interactivo, como si fuera un navegador web in-app] `https://developer.apple.com/documentation/webkit/wkwebview`

ventana donde se muestra el contenido HTML obtenido de una URL externa.

5.10.1.3. Interceptando eventos

Las pruebas con Webview mostrando el contenido de un *widget* resultaron positivas, la sección de la aplicación *mobile* responsable de mostrar un *widget* en particular estaba conformada por un título y luego un elemento Webview que ocupaba el 100 % del espacio restante de la pantalla. Hubo algunos contratiempos, y si bien podíamos ejecutar cualquier *widget* y este se comportaría y vería como si estuviésemos viéndolo en la ventana de previsualización del *IDE Web*, encontramos que los hipervínculos a secciones internas del mismo documento (los cuales tienen un formato estándar de *[protocolo]://[url]#[nombreDelAncla]*) provocaban que se abriera el navegador por defecto del celular, comportamiento que rompía el efecto de estar viendo contenido especialmente diseñado para plataformas móviles.

Siguiendo con los lineamientos iniciales, el desarrollo de los *widgets* tenía que ser lo más agnóstico de la plataforma como fuera posible. Se buscó una solución que no involucrase tener que programar el *widget* usando alguna sintáxis especial para este caso de los enlace autorreferenciales o anclas. Encontramos la solución en la documentación sobre Webview, la Application Programming Interface (API) permite especificar varios parámetros útiles y entre uno de ellos la posibilidad de escuchar eventos de navegación provenientes del documento que se este mostrando, interceptarlos e incluso cancelarlos. Entonces fue cuestión de capturar el evento correspondiente a un *intento de navegación* (seguir un vínculo) y luego asegurarnos de solo dejar continuar a aquellos eventos que no incluya el dominio utilizado por el visualizador de *widgets* junto con el formato anteriormente descrito. Ante este caso se cancelaba el evento evitando su propagación y se hacía un cambio en la propiedad que dictaminaba qué URL debía mostrarse para que se visualice en la misma ventana la sección correspondiente.

Capítulo 6

Conclusiones y trabajos futuros

A continuación se describirán las conclusiones del presente trabajo y algunos de los posibles trabajos futuros que pueden continuar desarrollándose como resultado de la experiencia de desarrollo del *IDE Web*.

6.1. Conclusiones

De lo expuesto en la presente tesina se concluye que implementar una plataforma de desarrollo enriquecida completamente orientada para facilitar el acceso a información confidencial es posible. Se ha dejado constancia sobre las limitaciones técnicas encontradas durante el proyecto, y cómo el equipo ha encontrado soluciones para poder sortear estos obstáculos siempre bajo un marco regido por las reglas impuestas por la normativa de HIPAA, y teniendo en cuenta las decisiones de negocio que la empresa ha tomado en pos de lograr un producto comercialmente atractivo.

Se analizaron las tecnologías que forman la base del proyecto como **MeteorJs**, SQLite y tecnologías específicas de *front-end* como ReactJs y React Native. También se estudió sobre el software complementario que ha sido requerido como *plugins* de MeteorJs y librerías externas de ReactJs. Esta combinación de tecnologías ha sido el resultado de aprovechar la experiencia de los desarrolladores involucrados y también fue acorde a los estándares del desarrollo web moderno.

El enfoque principal ha estado puesto en facilitarle el trabajo al editor y brindarle las herramientas necesarias sin perder la capacidad de manejo y auditoría que se realiza por parte de la empresa por sobre las aplicaciones y

widgets publicados.

Mediante todo lo expuesto se ha logrado:

- Un sistema de desarrollo completamente basado en la web con una interfaz clara, ágil y en tiempo real, incluyendo control de usuarios y el sistema de revisión y publicación de aplicaciones.
- El editor integrado de código con *feedback* inmediato de lo que se está escribiendo.
- La posibilidad de escribir el código de estilos usando dos dialectos distintos a gusto del editor.
- Acceso a las bases de datos de los usuarios de manera veloz y segura conforme la normativa vigente.

6.2. Trabajos futuros

Existen dos puntos de vista para listar los *trabajos futuros*. Desde la perspectiva del producto que se documentó en este trabajo existen varias mejoras que se pudieron identificar durante el desarrollo, y que debido a la naturaleza del proyecto y a la necesidad inmediata de obtener resultados iniciales, fueron pospuestos para una versión posterior. En la misma lista es prudente también agregar las mejoras de tecnologías que han sucedido durante el tiempo de desarrollo del proyecto y que deberían tenerse en cuenta como *Deuda técnica*.

- Actualizar MeteorJs para obtener soporte para TypeScript.
- Actualizar ReactJs a su última versión.
- Implementar Sandpack para renderizar el editor y su previsualización.
- Implementar un sistema de replicación de archivos que permita el escalamiento horizontal de las diferentes bases de datos SQLite de los usuarios y así lograr una optimización en tiempos de carga y evitar cuellos de botella.
- Implementar un *Diseño Responsive* para el *IDE Web* que permita la administración de apps y widgets junto con las opciones de publicación.

A continuación se presentan algunos trabajos futuros que pueden desarrollarse como resultado de esta investigación o que, por exceder el alcance de esta tesina, no han podido ser tratados con la suficiente profundidad.

- Documentar cómo fue desarrollado el cliente móvil para iOS. Detallando particularidades de la plataforma, el *AppStore* y todo el ecosistema de herramientas y servicios que rodean a la publicación de una nueva aplicación en dicho mercado.
- Documentar el desarrollo de una app en particular con un caso de uso concreto, desde el requerimiento inicial hasta la investigación pertinente de dónde se sacarían los datos para luego hablar sobre cómo se presentarán al usuario de manera correcta. Todo esto se puede encarar desde el punto de vista de un editor que usa el *IDE Web* como plataforma de desarrollo pero que no necesariamente está familiarizado con los pormenores de la implementación descrita en este documento.
- Analizar y documentar el desarrollo de un módulo de entrada de información médica dentro del *IDE Web*. Durante este proyecto, en muchas ocasiones se planteó cómo lograr la persistencia en la base de datos de la empresa de la información que el usuario ingresa en un *widget*, para uso futuro. Dicha información es la contenida en un simple formulario o la encontrada en un archivo subido, hasta casos más complejos donde, haciendo uso de la API de HealthKit[20], se pudieran compartir datos médicos almacenados en el celular del usuario.

Glosario

ACID Atomicity, Consistency, Isolation, and Durability. 33

AJAX JavaScript asíncrono y XML, de sus siglas en inglés *Asynchronous JavaScript And XML* . 23

android Es un sistema operativo de código abierto utilizado principalmente en dispositivos móviles. Escrito principalmente en Java y basado en el sistema operativo Linux, fue desarrollado inicialmente por Android Inc. y finalmente fue comprado por Google en 2005. 21, 22, 73

API Application Programming Interface. 6, 24, 40, 44, 47, 75, 78

Arboles B Los árboles B son estructuras de datos de árbol que se encuentran comúnmente en las implementaciones de bases de datos y sistemas de archivos. Al igual que los árboles binarios de búsqueda, son árboles balanceados de búsqueda, pero cada nodo puede poseer más de dos hijos. Los árboles B mantienen los datos ordenados y las inserciones y eliminaciones se realizan en tiempo logarítmico amortizado. 33

Arboles B+ Un árbol B+ es una variación de un árbol B con la diferencia de que el árbol B+ permite la repetición de claves en nodos internos hasta llegar a los nodos hoja mientras que el árbol B no permite dicha repetición. 33

BA Business Associates. 9

biotecnólogo Los biotecnólogos estudian la biología, la ciencia de los seres vivos, asociada a la tecnología. Investigan y desarrollan el uso de la biología para resolver problemas en áreas tales como la salud, la industrias farmacéutica y química, la agricultura, la producción de alimentos y la protección del medio ambiente. 8, 40, 41

bottom-up En el diseño bottom-up las partes individuales se diseñan con detalle y luego se enlazan para formar componentes más grandes, que a su vez se enlazan hasta que se forma el sistema completo. 18

brainstorming El brainstorming es una popular técnica utilizada para encontrar ideas basada en la creatividad espontánea y sin filtros. Las 4 leyes del brainstorming: Cantidad antes que calidad; No a las críticas, discusiones o comentarios durante la sesión; Registrar todas las ideas; Pensar con originalidad e inspirarse mutuamente. 37

BSON Binary JSON. 32

C++ C++ es un lenguaje de programación diseñado en 1979 por Bjarne Stroustrup. La intención de su creación fue extender al lenguaje de programación C y añadir mecanismos que permiten la manipulación de objetos. 24

CDD Component Driven Development. 18

CE Covered Entities. 9

CMS Content Managment System. 44

CSS Cascading Style Sheets. 23, 60, 61

currificar Currificar es la técnica inventada por Moses Schönfinkel y Gottlob Frege que consiste en transformar una función que utiliza múltiples argumentos (o más específicamente una n-tupla como argumento) en una secuencia de funciones que utilizan un único argumento. `https://es.wikipedia.org/wiki/Currificaci%C3%B3n`. 19

data Scientist Un Data Scientist o científico de datos es el profesional que se dedica a recolectar, analizar e interpretar grandes volúmenes de datos para extraer la información relevante de ellos. Son personas que aplican sus conocimientos en matemáticas, estadística y programación para analizar e interpretar los datos de que disponen las empresas y sacar información valiosa de ellos. 40

DB Database. 27, 62

Deuda técnica Con deuda técnica, se definen los errores, carencias y deficiencias deliberadas o involuntarias en el código que se generan por falta de comunicación, dirección de equipo, cualificación o publicación apresurada de productos y que aumentan constantemente debido a la falta de refactorización.. 77

Diseño Responsive El diseño web responsive o adaptativo es una técnica de diseño web que busca la correcta visualización de una misma página en distintos dispositivos. Desde computadoras de escritorio a tablets y móviles.. 77

ECMAScript ECMAScript específicamente es el estándar que a partir del año 2015 a la actualidad se encarga de regir como debe ser interpretado y funcionar el lenguaje JavaScript, siendo este interpretado y procesado por multitud de plataformas, entre las que se encuentran los navegadores web, NodeJs u otros ambientes como el desarrollo de aplicaciones para los distintos sistemas operativos que actualmente existen en el mercado. Los responsables de dichos navegadores y JavaScript deben encargarse de interpretar el lenguaje tal como lo fija ECMAScript. 17, 23, 24

EJSON Extended JSON. 27

enlace autorreferenciales o anclas Los vínculos autorreferenciales o enlaces ancla son enlaces web (HTML) que indican no solo el documento sino también un punto especifico dentro del mismo mediante el uso de un marcador # en la url con el *id* del elemento o sección a vincular.. 75

ePHI Electronic Protected Health Information. 11–13

EUA Estados Unidos de América. 9

hackaton Un encuentro de programadores cuyo objetivo es el desarrollo colaborativo de software. 7

Hash md5 En criptografía, MD5 (abreviatura de Message-Digest Algorithm 5, Algoritmo de Resumen del Mensaje 5) es un algoritmo de reducción criptográfico de 128 bits ampliamente usado. `https://es.wikipedia.org/wiki/MD5`. 66

HIPAA Health Insurance Portability and Accountability Act. 9–13, 70, 76

Hosting Es el servicio que provee a los usuarios de Internet un espacio de almacenamiento en línea, que permite publicar todo el contenido relacionado con un sitio o servicio web. 9

HTML HyperText Markup Language. 18, 23, 74, 75

I/O Se refiere a operaciones de entrada-salida, como por ejemplo escribir un dato en memoria o leer un archivo del sistema operativo. 24

IDE Integrated Development Environment. 15

iOS iOS es un sistema operativo creado por la marca Apple para uso exclusivo de sus dispositivos móviles: iPhone, iPad, y iPod, de código cerrado.. 21, 22, 73, 74

Kotlin Kotlin es un lenguaje de programación de tipado estático que corre sobre la máquina virtual de Java y que también puede ser compilado a código fuente de JavaScript. Es desarrollado principalmente por la empresa JetBrains.. 21

MVP Minimum Viable Product. 50

NFC Near Field Communication. 6

NPM Node Package Manager. 15, 25, 44

open source El código abierto es una filosofía de trabajo y colaboración seguida por los miembros de la comunidad de open source. Esta filosofía se basa en la libertad intelectual y los principios fundamentales: transparencia, colaboración, entrega, inclusión y comunidad. 15, 18, 21, 83

percentil El percentil es una medida de posición usada en estadística que indica, una vez ordenados los datos de menor a mayor, el valor de la variable por debajo del cual se encuentra un porcentaje dado de observaciones en un grupo. Por ejemplo, el percentil 20.º es el valor bajo el cual se encuentran el 20 por ciento de las observaciones, y el 80 % restante son mayores. 70

PHI Protected Health Information. 9–13

POC Proof of concept. 62

pruebas de usabilidad Las pruebas de usabilidad refieren a un método para probar la funcionalidad de un sitio web, una aplicación u otro producto, y consisten en evaluar un producto probándolo con usuarios representativos reales mientras intentan completar tareas en él. Los usuarios suelen ser observados y su interacción grabada para posterior análisis. Es común el uso de herramientas que permiten el seguimiento ocular para poder determinar que es lo que el usuario está mirando en la pantalla. 8

QA Quality Assurance - Contol de calidad. 7

Redis Redis es una excelente opción para implementar una caché en memoria de alta disponibilidad a fin de reducir la latencia de acceso a los datos, incrementar la capacidad de procesamiento y aliviar la carga de la aplicación y de la base de datos relacional o NoSQL. Redis puede suministrar elementos solicitados frecuentemente con tiempos de respuesta inferiores a un milisegundo, y permite ajustar la escala con facilidad en caso de cargas mayores sin tener que ampliar el costoso backend. 72

Requerimientos funcionales Los requerimientos funcionales de un sistema, son aquellos que describen cualquier actividad que este deba realizar, en otras palabras, el comportamiento o función particular de un sistema o software cuando se cumplen ciertas condiciones. 36

REST Representational State Transfer. 40

Schemaless Las base de datos tradicionales tienen estructuras para organizar la infracción que son bien definidas y se conocen de antemano, las base de datos schemaless no tienen esa restricción permitiendo mayor flexibilidad a la hora de almacenar documentos con estructuras variables o incompletas.. 32

Single Page Applications Una SPA (*por sus siglas en ingles Single Page Application*) es una aplicación web implementada de forma que se ejecuta sobre un solo documento HTML y va cargando su contenido de manera asincronica . 18

SQL Structured Query Language. 31, 32, 38, 40–42, 62

Thread-safe La seguridad en hilos es un concepto de programación aplicable en el contexto de los programas multihilos. Una pieza de código es segura en cuanto a los hilos si funciona correctamente durante la ejecución simultánea de múltiples hilos. 33

TypeScript Lenguaje open source desarrollado por Microsoft que extiende la sintaxis de para añadir tipado estático y objetos basados en clases. Este código se *transpila* a código y es totalmente compatible con este. 77

UI User Interface. 18, 21, 26, 39, 40, 60, 61, 73

UTF-16 Unicode (or Universal Coded Character Set) Transformation Format – 16-bit. 34

UTF-8 Unicode (or Universal Coded Character Set) Transformation Format – 8-bit. 34

UX User experience. 60

Wearables Un *wearable* es un dispositivo electrónico que se usa en el cuerpo humano y que interactúa con otros aparatos para transmitir o recoger algún tipo de datos. El ejemplo más claro y más conocido de *wearable* lo constituyen los relojes inteligentes y las pulseras de actividad, pero hay muchos más. 74

Índice de figuras

4.1. Componente SandPack renderizado con opciones por defecto . 16

5.1. Diseño general de la UI propuesta 40
5.2. Interconexiones del Ide Web con otras partes del ecosistema . 43
5.3. Pantalla de login 47
5.4. Diagrama de estados posibles de un widget 51
5.5. Pantalla de inicio 52
5.6. Tarjeta de un *widget* en la pantalla de inicio junto con el menú contextual desplegado 53
5.7. Modal de creación de widget 54
5.8. Pantalla de edición de un *widget* 59
5.9. Ejemplo de segmentación y aislamiento de la información de un usuario en particular 62

Listado de bloques de código

1. Implementación mínima de un editor de código Sandpack . . . 16
2. Ejemplo de archivo package.json con la información del proyecto npm . . . 25
3. Ejemplo de servidor web diciendo 'hola mundo' escrito en Node.js usando Express . . . 26
4. Ejemplo de creación de colección en MeteorJs . . . 27
5. Ejemplo básico de publicación y suscripción . . . 28
6. Ejemplo básico de un método de meteor. . . . 29
7. Ejemplo de como Babel transforma código escrito en ES2015 en código compatible con navegadores antiguos. . . . 30
8. Declaración del método de MeteorJs que se encarga de convertir el código escrito en *código transpilado* . . . 39
9. Métodos del cliente del Login Service disponibles . . . 44
10. Clase y método de Meteor para acceder al contenido localizado del CMS . . . 46
11. Función de verificación de token y sincronización del usuario entre bases de datos . . . 48
12. Método agregado a la jerarquía de clases de componentes ReactJs para facilitar acceso a la base de datos . . . 56
13. Ejemplo de uso del método query heredado de WidgetComponent 57
14. Método del cliente para la precarga de tablas . . . 64
15. Método de meteor del lado del servidor que precarga las tablas 65
16. Método de creación de la base de datos SQLite por cada usuario 66
17. Método de creación de la tabla metadata . . . 67
18. Lista de tablas y versiones . . . 68
19. Función principal de precarga . . . 69
20. Funciones de creación, consulta y borrado para una tabla de ejemplo . . . 71

Bibliografía

[1] What is citizen science, and how can you get involved? https://mashable.com/archive/citizen-science.

[2] Emmanuel Agu, Peder Pedersen, Diane Strong, Bengisu Tulu, Qian He, Lei Wang, and Yejin Li. The smartphone as a medical device assessing enablers, benefits and challenges. *2013 Workshop on Design Challenges in Mobile Medical Device Systems*, 2013.

[3] Global smartphone market share: By quarter. https://www.counterpointresearch.com/wp-content/uploads/2022/02/Counterpoint-Research-Q4-2021-Global-Smartphone-Market.pdf, May 2022.

[4] Apple a15 bionic vs snapdragon 888+: chipset comparison. https://techarena24.com/apple-a15-bionic-vs-snapdragon-888-chipset-comparison/#:~:text=Moreover%2C%20the%20Apple%20A15%20Bionic,(Tera%20Operations%20Per%20Second), May 2022.

[5] Protected health information (phi): Everything you need to know about hipaa and phi. https://www.totalhipaa.com/protected-health-information-phi-101/, May 2022.

[6] Hipaa for developers: 2022 hipaa compliant developer guide. https://www.atlantic.net/hipaa-compliant-hosting/hipaa-compliant-developer-guide/, Jun 2022.

[7] Sandpack introduction. https://sandpack.codesandbox.io/docs/.

[8] Alex Banks and Eve Porcello. *Learning React: Functional Web Development with React and Redux*. OŔeilly Media, May 2017.

[9] Katie Lawson. What are single page applications and why do people like them so much? https://www.bloomreach.com/en/blog/2018/what-is-a-single-page-application.

[10] Jesse James Garrett. Ajax: A new approach to web applications. https://immagic.com/eLibrary/ARCHIVES/GENERAL/ADTVPATH/A050218G.pdf, 2005.

[11] Node.js Docs. Overview of blocking vs non-blocking. https://nodejs.org/es/docs/guides/blocking-vs-non-blocking/.

[12] npm Docs. Creating a default package.json file. https://docs.npmjs.com/creating-a-package-json-file.

[13] Meteor Cloud Docs. Documentation of meteor's publication and subscription api. https://docs.meteor.com/api/pubsub.html.

[14] Matthew O'Riordan. Everything you need to know about publish/subscribe. https://ably.com/topic/pub-sub#advantages-of-using-publish-subscribe-pattern, July 2020.

[15] Astronomy documentation. https://jagi.github.io/meteor-astronomy, Jan 2019.

[16] Karl Seguin. *El pequeño libro de Mongo*. OpenSource, 2019.

[17] Grant Allen and Mike Owens. *The Definitive Guide to SQLite (Expert's Voice in Open Source)*. Apress, May 2011.

[18] Sibsankar Haldar. *SQLite Database System Design and Implementation*. Motorola Mobility, Inc, 2011.

[19] Aaron Parecki. *OAuth 2.0 Simplified: A Guide to Building OAuth 2.0 Servers*. Lulu.com, May 2018.

[20] Healthkit - access and share health and fitness data while maintaining the user's privacy and control. https://developer.apple.com/documentation/healthkit.