



Universidad Austral de Chile

Facultad de Ciencias de la Ingeniería
Escuela de Ingeniería Civil en Informática

ARQUITECTURA DE MICROSERVICIOS PARA EL MODELAMIENTO DEL APRENDIZ

Proyecto para optar al título de
Ingeniero Civil en Informática

PROFESOR PATROCINANTE:

Julio D. Guerra H.

PROFESOR INFORMANTE:

Eliana Scheihing G.

PROFESOR INFORMANTE:

Cristian Olivares R.

PABLO JONAN SÁEZ PARRA

Valdivia - Chile
2022

ÍNDICE

ÍNDICE	I
ÍNDICE TABLAS.....	III
ÍNDICE FIGURAS	IV
1. RESUMEN.....	VIII
2. ABSTRACT	IX
3. INTRODUCCIÓN	1
3.1. Antecedentes Generales	1
3.2. Definición del problema.....	1
3.3. Propuesta	2
3.4. Objetivo General	2
3.5. Objetivos específicos	3
4. METODOLOGÍA DE DESARROLLO	4
4.1. Estudio del estado del arte de arquitecturas, herramientas y tecnologías relacionadas.....	4
4.2. Diseño de arquitectura.....	4
4.3. Desarrollo, validación y documentación.....	5
5. ESTUDIO DEL ESTADO DEL ARTE DE ARQUITECTURAS, HERRAMIENTAS Y TECNOLOGÍAS RELACIONADAS	8
5.1. Arquitectura de sistemas educativos.....	8
5.2. Arquitectura basada en microservicios	10
5.3. GraphQL	13
5.4. Entrevista usuario representativo	15
6. DISEÑO DE ARQUITECTURA.....	16
6.1. Arquitectura De Microservicios Para Learner Modeling.....	16
6.2. Autenticación	18
6.3. Gateway.....	19
7. MODELAMIENTO DE DATOS Y ENTIDADES	20
7.1. Usuarios.....	20

7.2.	Proyectos	24
7.3.	Dominio.....	25
7.4.	Acciones.....	30
7.5.	Contenido	32
7.6.	Estado de modelo	34
8.	DESARROLLO DE SERVICIOS	37
8.1.	Documentación	39
8.2.	Autenticación y Autorización	40
8.3.	Usuarios.....	43
8.4.	Gateway.....	55
8.5.	Pruebas de carga.....	61
9.	DESPLIEGUE.....	68
9.1.	Despliegue ideal con computación en la nube	68
9.2.	Despliegue en servidor único	69
10.	PLANTILLA DE DESARROLLO	74
10.1.	Librerías principales.....	74
10.2.	Código importante.....	74
11.	CONCLUSIONES Y TRABAJO FUTURO	84
11.1.	Retrospectiva.....	84
11.2.	Trabajo futuro.....	86
12.	REFERENCIAS.....	87
13.	ANEXOS.....	90
	Anexo A: Panel de administración, página de proyectos.....	90
	Anexo B: Cliente GraphQL Altair	90
	Anexo C: Documentación plataforma.....	92
	Anexo D: Repositorio principal GitHub	93

ÍNDICE TABLAS

Tabla	Página
Tabla 1. Tabla comparativa entre arquitectura de microservicios y monolítica.	12
Tabla 2. Definición, extensión y utilización de entidades por servicios.....	38

ÍNDICE FIGURAS

Figura	Página
Figura 1. GraphQL Voyager en parte del esquema.....	5
Figura 2. Borrador diagrama de relación entre parte del “Modelo de Aprendiz”	10
Figura 3. Arquitecturas monolíticas y basadas en microservicios	11
Figura 4. Relación entre servicios de arquitectura	18
Figura 5. Modelo de entidad “User”	22
Figura 6. Modelo de entidad “Group”.....	23
Figura 7. Diagrama de entidad relación entre “User” y “Group”	24
Figura 8. Modelo de entidad “Project”	25
Figura 9. Diagrama de entidad relación entre “Domain”, “KC”, “Project” y “Topic”....	26
Figura 10. Modelo de entidad “Domain”	27
Figura 11. Modelo de entidad “Topic”	29
Figura 12. Modelo de entidad “KC”	30
Figura 13. Modelo de entidad “Action”	32
Figura 14. Modelo de entidad “Content”	34
Figura 15. Modelo de entidad “ModelState”	36
Figura 16. Fragmento de código en proceso de autenticación	42
Figura 17. Fragmento de código en reglas de autorización	43
Figura 18. Esquema base entidad “User”	44
Figura 19. Esquema base entidad “Group”	46
Figura 20. Esquema “Query” y “Mutation” de servicio de usuarios	47
Figura 21. Esquema consultas administrativas de servicio de usuarios.....	48
Figura 22. Ejemplo de resolutor de campo “active” de entidad usuario	49
Figura 23. Resolutores con verificación de administración para módulo de usuarios....	49
Figura 24. Resolutor de “Query.currentUser”.....	50
Figura 25. Resolutor de “Query.users”	50
Figura 26. Resolutores de “AdminUserMutations” parte de módulo de usuarios	51
Figura 27. Resolutor de “AdminUserQueries.allUsers” parte de módulo de usuarios	52
Figura 28. Resolutores de “AdminUserMutations” parte de módulo de grupos.....	53
Figura 29. Resolutor de “Query.groups”.....	54
Figura 30. Esquema módulo de proyectos	54
Figura 31. Resolutores módulo de proyectos	55
Figura 32. Esquema “Proxy” GraphQL distribuido	56
Figura 33. Configuración de conexión con servicios.....	57
Figura 34. Conexión de “Gateway” con servicios	59

Figura 35. Configuración de “Stitching” para “Gateway”	60
Figura 36. Solicitud inicial de usuario en prueba de carga	62
Figura 37. Envío de acción por usuario en prueba de carga	63
Figura 38. Ejemplo de ejecución desde terminal de prueba de carga	65
Figura 39. Uso de CPU en ejecución de prueba de carga	66
Figura 40. Tiempos de respuesta para panel de administración.....	67
Figura 41. Despliegue ideal en nube de plataforma parcial	69
Figura 42. Configuración recomendada para Docker Compose	71
Figura 43. Configuraciones “Docker Compose” y variables de ambiente.....	72
Figura 44. Lista de servicios con puertos por defecto.....	72
Figura 45. Configuración monolítica recomendada para “Docker Compose”	73
Figura 46. Módulo de definición de cliente	75
Figura 47. Mensaje error de ejemplo en plantilla de desarrollo.....	76
Figura 48. Fragmento de código para sincronización de autenticación	77
Figura 49. Función utilitaria para reporte de acciones	79
Figura 50. Ejemplo de uso de función utilitaria para reporte de acciones	80
Figura 51. Diseño principal plantilla, modo claro.....	80
Figura 52. Diseño principal plantilla, modo oscuro	81
Figura 53. Definición barra de navegación plantilla.....	82
Figura 54. Barra de navegación con usuario sin autenticación.....	83
Figura 55. Proceso de cerrado de sesión	83

RESUMEN

El proyecto denominado “Arquitectura de microservicios para el modelamiento del aprendiz” nace de la creciente necesidad de nuevas plataformas de desarrollo para sistemas educativos inteligentes. Un ejemplo de estos son los sistemas de tutoría inteligente, en donde un concepto importante es el denominado “modelamiento del aprendiz”, el cual consiste en estimar los niveles de conocimiento en un área de estudio en base a evidencias de aprendizaje en la plataforma.

La propuesta es una plataforma que consiste en una serie de servicios y un *gateway*. Los servicios incluyen el manejo de acciones, definición de dominio y tópicos, y administración de proyectos y usuarios, mientras que el *gateway* se encarga de unir los servicios para la simplicación de uso. La plataforma busca satisfacer los requerimientos básicos para el desarrollo de sistemas educativos inteligentes con un enfoque en el uso de la popular arquitectura de microservicios, la cual ha sido impulsada por el gran crecimiento de plataformas de computación en la nube.

Un aspecto importante del proyecto es la utilización de tecnologías y herramientas de desarrollo innovadoras como GraphQL para la definición de los servicios, Node.js y TypeScript como el lenguaje de programación, y la combinación de tecnologías como PostgreSQL y Prisma para el manejo de base de datos.

La plataforma promueve la creación y experimentación de un ambiente de desarrollo innovador, lo que permite una gran flexibilidad y control sobre todos los procesos del sistema, dando una base completa para el avance de proyectos de nuevos sistemas educativos inteligentes de la universidad.

ABSTRACT

The project named “Microservices architecture for learner modeling” is born from the growing need of new development platforms for smart learning systems. An example of these are smart tutoring systems, where an important concept for these is the so-called “learner modeling”, which consists of estimating the knowledge levels in a study area based on learning evidence from the platform.

The proposal is a platform consisting of a series of services and a gateway. The services include the actions management, domain and topics definition, and administration of projects and users, meanwhile the gateway is responsible of joining the services to simplify the usage. The platform seeks to meet all the basic requirements for the development of smart learning systems with a focus on the usage of the popular microservices architecture, which has been boosted by the big growth of cloud computing platforms.

An important aspect of the project is the usage of innovative technologies and tools like GraphQL for the definition of the services, Node.js and TypeScript as the programming language, and the combination of technologies like PostgreSQL and Prisma for database management.

The platform promotes the creation and experimentation of an innovative development environment, this enables big flexibility and control over every process of the system, giving a complete foundation for the progress of new smart learning system projects of the university.

1. INTRODUCCIÓN

1.1. Antecedentes Generales

Los sistemas educativos inteligentes normalmente se consideran como plataformas educativas digitales en donde su enfoque consiste en mejorar la experiencia del aprendizaje con el uso de distintas tecnologías disponibles.

Dentro del espectro de sistemas educativos inteligentes se encuentra un tipo denominado “Sistema de tutoría inteligente” o “*Intelligent Tutoring System*”, en la cual el sistema computacional, de forma inmediata y customizada, da instrucciones y/o retroalimentación, usualmente sin la necesidad de intervención activa de profesores humanos.

En el área de tecnologías de aprendizaje existe un concepto denominado “*learner modeling*”, una forma de “*user modeling*”, que consiste en estimar los niveles de conocimiento del aprendiz en cada uno de los tópicos o conceptos del área de estudio en base a evidencias de aprendizaje en la misma plataforma.

El uso de sistemas educativos inteligentes en distintas instituciones educacionales, como universidades, ha crecido mucho con el pasar de los años, y en conjunto con ello, los requerimientos que han nacido para satisfacer distintas necesidades han aumentado a la par de éstas.

1.2. Definición del problema

En el ambiente de la investigación y desarrollo de herramientas educativas inteligentes actualmente existe una necesidad de plataformas que contengan y faciliten el uso de funcionalidades que son compartidas por una considerable porción del ecosistema.

Un ejemplo de sistemas educativos son los sistemas de tutoría inteligente. Estos requieren de una plataforma que permita modelar y administrar estudiantes, junto a sus accesos a distintos recursos asociados, en donde la simplificación y agilización del desarrollo y uso de este tipo de sistemas es crucial.

Uno de los paradigmas de investigación que involucra procesos de desarrollo de sistemas es el “*Design-based research*”. Este consiste en el desarrollo de soluciones a problemas en un ciclo iterativo de desarrollo y pruebas, buscando comprobar por qué, cuándo y cómo las innovaciones en la educación funcionan en la práctica. Este

paradigma se beneficia directamente de plataformas que sean flexibles y que den control total sobre todos los procesos de ésta.

1.3. Propuesta

El proyecto consiste en la creación de una plataforma que permita el modelamiento de los procesos de aprendizaje, la cual establece una arquitectura base para facilitar el desarrollo de sistemas educativos inteligentes, como tutores cognitivos. Este sistema establece mecanismos para el manejo de usuarios, grupos, dominios y contenidos, además de las relaciones entre ellos y el registro de acciones y seguimiento.

En conjunto con las funcionalidades que suelen ser comunes en los sistemas educativos inteligentes, también se espera agilizar el desarrollo y el uso de nuevas herramientas del nuevo y emergente ecosistema web.

La propuesta considera el desarrollo de un sistema de APIs GraphQL que facilitan un uso expresivo y seguro del lado del cliente, teniendo muchas herramientas disponibles, derivadas de la especificación GraphQL (GraphQL Specification, 2021). Todo a través de un único “endpoint”.

La propuesta también considera dentro de su implementación un entorno de gran escalabilidad y rendimiento. Estas incluyen tecnologías como PostgreSQL (PostgreSQL, s.f) como RDBMS y el uso principal de frameworks web en Node.js (Node.js, s.f) con TypeScript (TypeScript, s.f). La estrategia de desarrollo de software es “Monorepo”, la cual facilita el desarrollo paralelo de las distintas partes del sistema, y al mismo tiempo facilita la reutilización de código, simplifica el manejo de dependencias, la utilización de “commits atómicos”, y el gran potencial de colaboración entre distintos grupos de trabajo (Fernandez, 2021).

Un aspecto importante de la presente propuesta es la utilización de la popular y creciente arquitectura de microservicios. Este tipo de arquitectura ha ganado especial importancia en la actualidad dado el crecimiento exponencial que han tenido las soluciones de despliegue en “la nube”, ya que, una de las mayores ventajas que esta otorga es la gran capacidad de escalamiento horizontal.

1.4. Objetivo General

Crear una arquitectura de “*learner model*” altamente escalable y adaptable que dé soporte al desarrollo de contenido de aprendizaje inteligente y servicios asociados a los procesos de aprendizaje, usando una arquitectura de microservicios, con una

gran capacidad de procesamiento eficiente de grandes cantidades de datos.

1.5. Objetivos específicos

1. Estudiar el estado del arte de arquitecturas relacionadas y herramientas/tecnologías asociadas.
2. Diseñar una arquitectura de modelamiento del aprendiz que escale de forma eficiente a las necesidades.
3. Desarrollar y validar el modelamiento de datos.
4. Desarrollar y validar servicios.
5. Desarrollar y documentar plantillas de desarrollo.

2. METODOLOGÍA DE DESARROLLO

Durante el transcurso del proyecto, dependiendo del objetivo específico por cumplir, se definieron aspectos metodológicos acorde a la situación. Las metodologías fueron descriptas y categorizadas basadas aproximadamente acorde a la etapa y proceso de desarrollo.

2.1. Estudio del estado del arte de arquitecturas, herramientas y tecnologías relacionadas

La realización del estudio del arte de arquitecturas, herramientas y tecnologías se constituyó principalmente de reuniones preliminares e introductorias con el profesor patrocinante Ph.D Julio Guerra Hollstein, y del estudio del ecosistema de “aprendizaje inteligente” y desarrollo web requerido para la definición de requerimientos y marco teórico. Esto incluyó por ejemplo la lectura de documentación de arquitecturas existentes y papers de investigación del área, además de una entrevista a otro investigador con experiencia trabajando en plataformas similares.

Este proceso permitió entender el funcionamiento de plataformas existentes y la definición posterior de los servicios y las entidades del proyecto.

2.2. Diseño de arquitectura

Para el proceso de diseño de arquitectura, el estudio del ecosistema previamente realizado fue de especial relevancia. Esto constituyó en gran parte la definición de los requerimientos del proyecto.

Durante el transcurso del avance del diseño de distintas partes del proyecto, la realización de reuniones con el profesor patrocinante Ph.D Julio Guerra Hollstein, con frecuencia semanal, o en su defecto quincenal, pudieron definir, por ejemplo, qué funcionalidades la plataforma debe proveer y qué necesidades se deben satisfacer.

El proceso práctico de diseño consistió en mayor parte de prototipado incremental, ya que para esta etapa se construyó la base de desarrollo del proyecto, poniendo en práctica los requerimientos obtenidos y satisfaciendo las necesidades definidas, estos diseños fueron trabajados de forma iterativa incrementalmente.

Durante el proceso de diseño, la utilización de herramientas de visualización fueron de especial importancia, permitiendo poder plasmar la idea a implementar. En la Figura 3 se puede ver una herramienta llamada GraphQL Voyager (GraphQL Voyager, s.f), la cual es utilizada para la visualización interactiva en forma de grafos del esquema de la API GraphQL, en el transcurso de las reuniones semanales en las que se discutió el diseño de las entidades, el prototipado y su visualización gráfica fue de especial utilidad para plasmar las ideas propuestas.

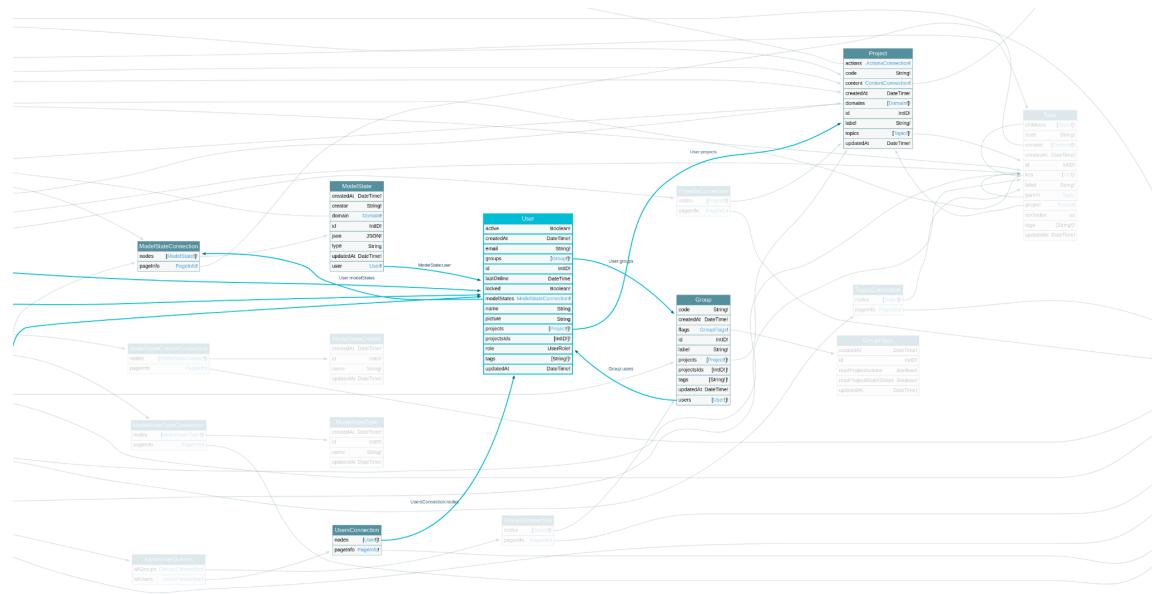


Figura 1. GraphQL Voyager en parte del esquema¹

2.3. Desarrollo, validación y documentación

El proceso de desarrollo y testing fue basado en la realización de prototipos incrementales iterativos. La programación del proyecto fue realizado utilizando el control de versiones Git (Git, s.f), siguiendo la estrategia de desarrollo “monorepo”, lo cual permite compartir dependencias y código entre los distintos servicios y testing automatizado.

2.3.1. Herramientas de desarrollo

El entorno de desarrollo del proyecto consistió en herramientas de desarrollo multiplataforma, ya que el desarrollo fue realizado de forma intermitente entre distintos sistemas operativos, Windows 10, macOS y Arch Linux. El editor de código utilizado fue Visual Studio Code (Visual Studio Code, s.f), el cual funciona

¹ La visualización completa del esquema utilizando GraphQL Voyager puede ser accedido desde <https://lm.inf.uach.cl/voyager>

excepcionalmente bien para el lenguaje de programación utilizado en el proyecto, TypeScript.

Este proyecto utiliza soluciones de integración continua. En específico, se hace uso de GitHub Actions (GitHub Actions, s.f) para la ejecución de tests automatizados y verificación de tipado. Con el objetivo de facilitar el despliegue del proyecto se hace uso de la integración continua provista por Docker Hub (Docker Hub, s.f), la cual construye la imagen utilizada para el proceso de despliegue.

2.3.2. Desarrollo de servicios y modelamiento de datos

El proceso de desarrollo de los servicios del proyecto fue realizado de forma normalmente paralelo al del modelamiento de datos, ya que dependiendo de la implementación que cada servicio requiera, el modelamiento de la entidad necesitada es diseñada e implementada.

El modelamiento de los datos fue realizado a través de las herramientas Prisma Schema (Prisma Schema, s.f) y Prisma Migrate (Prisma Migrate, s.f). La primera funciona a través de un esquema el cual define los modelos de la aplicación, representando las entidades del dominio de aplicación, mapean a las tablas de la base de datos relacional (PostgreSQL en este proyecto), forman la fundación consultas disponibles a través de la herramienta Prisma Client (Prisma Client, s.f) y proveen definiciones de tipado para los modelos, lo que permiten un acceso de tipado seguro a la base de datos. Luego de definir el esquema, Prisma Migrate se encarga de realizar los cambios requeridos en la instancia especificada de sistema de gestión de bases de datos.

Una vez que un servicio ya tiene acceso a los modelos de la base de datos, la implementación debe obedecer los esquemas definidos por la etapa de diseño, tomando ventaja de la forma de funcionar de GraphQL, y una vez que los resolutores se encuentran funcionando, se programan los tests automatizados que comprueban el funcionamiento correcto del servicio integrado.

2.3.3. Desarrollo de documentación

El proceso de desarrollo de documentación fue dividido en dos, la documentación del esquema compuesto por el conjunto de todos los servicios y documentación a nivel de desarrollo e implementación. Cabe considerar que este proceso fue realizado en su totalidad en el idioma Inglés, siguiendo la convención de programación en inglés y permitir la utilización y extensión del proyecto por personas no hispano-hablantes.

La documentación del esquema compuesto por el conjunto de los servicios fue realizada de tal manera que cada campo y relación de cualquier entidad del esquema tenga una descripción que describa comportamientos o explicación según corresponda.

La documentación de desarrollo e implementación fue realizada después de confirmar que la arquitectura funcione y el despliegue de prueba se encuentre funcionando. La documentación se construyó como una aplicación web, utilizando tecnologías y herramientas como MDX (MDX, s.f), Next.js (Next.js, s.f) y Nextra (Nextra, s.f). Este último se encarga de utilizar los dos anteriores y facilitar el diseño y convenciones que permiten trabajar directamente en escribir la documentación.

2.3.4. Desarrollo de plantillas de desarrollo y sistema de administración

Para el desarrollo de las plantillas de desarrollo se realizaron reuniones con compañeros en proyectos de título que harán uso del servicio. En el transcurso de las sesiones se conversó de lo que se incluye en la plantilla de desarrollo, cómo usar partes de ésta, y explicar parte del funcionamiento del servicio.

Parte del desarrollo del proyecto contempló el desarrollo de un panel de administración, el cual utiliza la plantilla de desarrollo y verifica de primera mano la funcionalidad de esta, y ayudó a definir partes esenciales de la plantilla.

3. ESTUDIO DEL ESTADO DEL ARTE DE ARQUITECTURAS, HERRAMIENTAS Y TECNOLOGÍAS RELACIONADAS

3.1. Arquitectura de sistemas educativos inteligentes

Los sistemas educativos inteligentes, al menos en el área de la tutoría inteligente (ITS / Intelligent Tutoring System), normalmente siguen una estructura que consiste en la división de los “concerns” en distintos modelos que pueden ser definidos por sí mismos, pero que deben contener interfaces de comunicación para que los modelos puedan entregarse información y comunicarse entre sí.

Considerando el alcance del presente proyecto, y que se usará un modelado de tipo “Feature-Based Modeling” (Brusilovsky, Peter & Peter, & Millán, Eva & Eva., 2007), que consiste en intentar modelar las características de usuarios individuales, como por ejemplo el conocimiento y metas de interés, y mientras el usuario trabaja con el sistema, las características pueden cambiar y el modelo hace seguimiento y representa un estado actualizado de las características modeladas.

3.1.1. Modelo de Dominio

El “**Modelo de Dominio**” busca listar y conceptualizar los distintos “tópicos” que el sistema debe ser capaz de entender y manejar. El concepto que se usa normalmente para los “tópicos” en el ambiente es “**Knowledge Components**” (abreviado como “**KCs**”), y es el nombre con el que se referirá al concepto en el transcurso del presente proyecto.

3.1.2. Modelo de Contenido

El “**Modelo de Contenido**” busca poder guardar, categorizar y suministrar todo el contenido que el sistema debe ser capaz de servir.

Este modelo normalmente tiene contacto estrecho con el “Modelo de Dominio” ya que en el proceso de categorización es necesario ser capaz de entregar el contenido apropiado dado distintos parámetros que vienen dados del proceso de modelado de dominio.

3.1.3. Modelo Pedagógico

El “**Modelo Pedagógico**” busca conceptualizar un nivel extra de categorización entre el “**Modelo de Dominio**” y el “**Modelo de Contenido**”, normalmente relacionado directamente a procesos de aprendizaje cognitivo, además de servir información extra a partir de posibles parámetros con respecto al estado actual del

usuario que pide el contenido, un ejemplo sencillo de esa “información extra” es información respecto a consejos o instrucciones relacionada al contenido respecto al dominio y posible estado actual del aprendiz.

Cabe recalcar que es posible que a nivel de implementación este modelo sea integrado dentro del mismo **“Modelo de Contenido”**, y esto dependerá netamente del nivel de “granularidad” y de especialidad funcional que se desea implementar, además de la facilidad que puede existir al ser implementado y conceptualizado de forma integrada.

3.1.4. **Modelo de Aprendiz**

Finalmente existe el **“Modelo de Aprendiz”**, que hace uso de la información y funcionalidades de los modelos anteriormente descritos, y que a partir de los datos obtenidos de un aprendiz (usuario) dentro del sistema, busca ser capaz de categorizar o dar información que se puede inferir al respecto del usuario utilizando información relevante del modelo de contenido y los registros de utilización y comportamiento del usuario con las aplicaciones y plataforma, con el objetivo principal de estimar niveles de conocimiento o aprendizaje en distintos elementos del modelo de dominio.

A nivel de implementación puede existir un modelo extra con comunicación directa con el **“Modelo de Aprendiz”**, al cual se denomina como **“Modelo de Estado”**, que se encarga de guardar y administrar los estados de todos los usuarios/aprendices obtenidos en los “logs” del sistema, y puede manejar un estado acumulativo que permite optimizaciones en términos de rendimiento y un historial de estados y registros que permite realizar análisis externo del sistema.

En la Figura 1 podemos ver la forma en la que los distintos modelos descritos se esperaría que se comuniquen, y por efectos de simplicidad se integró el **“Modelo Pedagógico”** dentro del **“Modelo de Contenido”**.

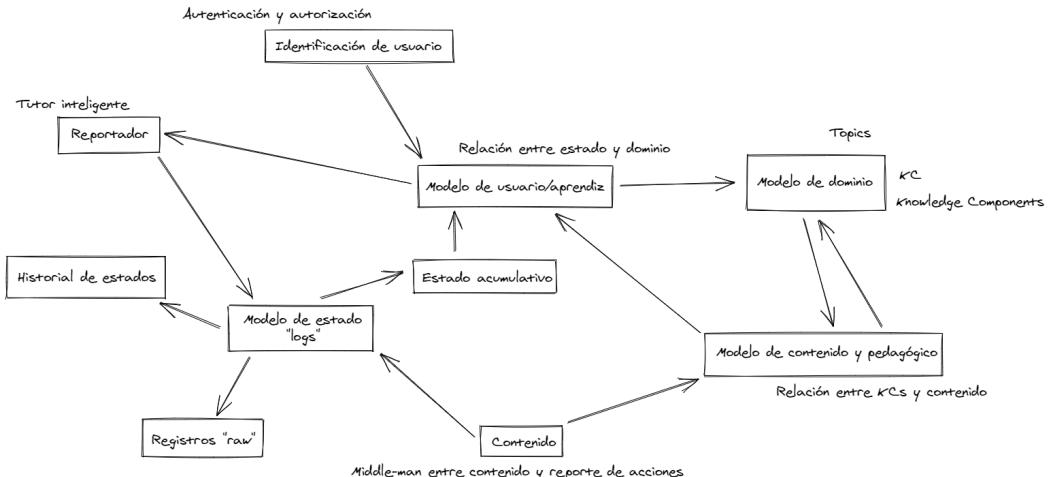


Figura 2. Borrador diagrama de relación entre parte del “Modelo de Aprendiz”²

3.2. Arquitectura basada en microservicios

Para la arquitectura del proyecto se va hacer uso de la arquitectura de microservicios, la cual consiste en la separación de los servicios que van a ser parte del sistema en distintos procesos en el que se pueden comunicar entre sí sobre una red, usando protocolos completamente independientes de la tecnologías de implementación del microservicio, como HTTP.

Normalmente los servicios son pequeños en tamaño, que funcionan a través de "mensajería", son limitados por su contexto, desplegables independientemente, descentralizados por naturaleza, y normalmente construidos y publicados a través de procesos automatizados.

En la Figura 2 se muestra a grandes rasgos la diferencia entre las arquitecturas tradicionales monolíticas y las arquitecturas basadas en microservicios.

² Diagrama realizado usando Excalidraw, accedido el 25 de enero, 2022, <https://excalidraw.com>

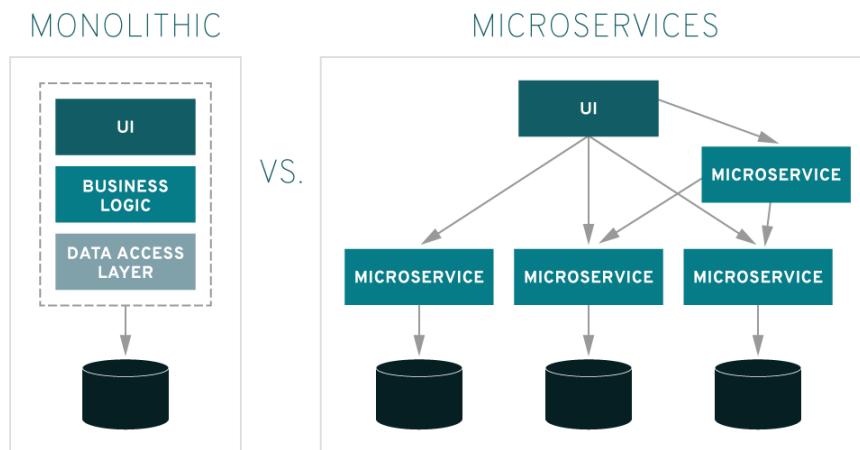


Figura 3. Arquitecturas monolíticas y basadas en microservicios³

En la Tabla 1 se compara la arquitectura de microservicios con la arquitectura tradicional monolítica considerando términos características generales, costos de desarrollo, formas de programación, métodos de despliegue y el uso de tecnologías.

³ Figura obtenida de *Why you should choose the microservices architecture?*. Accedido el 11 de mayo, 2021, desde <https://pretius.com/blog/benefits-of-microservices>.

Tabla 1. Tabla comparativa entre arquitectura de microservicios y monolítica.⁴

	Microservicios	Monolíticos
Pros	<ul style="list-style-type: none"> • Escalabilidad • Flexibilidad • Rendimiento rápido • Fiabilidad • Independencia 	<ul style="list-style-type: none"> • Desarrollo fácil • Costo-Eficiente • Tiempo-Eficiente • Intuitivo en las primeras etapas de desarrollo
Contras	<ul style="list-style-type: none"> • Requiere cambios en organizaciones y flujos de trabajo • No se adapta bien para proyectos pequeños • Puede conducir a retrasos de procesamiento de datos 	<ul style="list-style-type: none"> • Escalabilidad limitada • Un solo problema en alguna tecnología afecta todo el código • Despliegues de todo o nada complican nuevos lanzamientos • Características dependen fuertemente entre ellas • No es costo-eficiente a largo plazo
Costo	<ul style="list-style-type: none"> • Más alto al inicio de las etapas de desarrollo 	<ul style="list-style-type: none"> • Más alto una vez que el proyecto escala
Programación	<ul style="list-style-type: none"> • Distribuido en múltiples archivos de código • Cada servicio puede ocupar sus propios servicios de almacenamiento 	<ul style="list-style-type: none"> • Código y base de datos unificada para todo el producto
Despliegue	<ul style="list-style-type: none"> • Cada micro-servicio es desplegado individualmente 	<ul style="list-style-type: none"> • Todo el código necesita ser desplegado junto
Tecnologías	<ul style="list-style-type: none"> • Se da soporte al uso fácil de diferentes tecnologías (lenguajes de programación, marcos de trabajo, ambientes, entornos de ejecución, etc...) 	<ul style="list-style-type: none"> • El uso de una misma pila de tecnologías para todo el producto.

⁴ Basada en el blog de Vitaliy Ilyukha, *Monolith vs. Microservices: Why Companies Switch Back and Forth.* Accedido 25 de enero, 2022, desde <https://jelvix.com/blog/monolith-vs-microservices-architecture>

3.3. GraphQL

Una parte importante del presente proyecto es el uso de una especificación (y tecnologías relacionadas) denominada **GraphQL**, la cual funciona normalmente sobre protocolos conocidos en el ecosistema, como JSON y HTTP . Estos últimos protocolos (JSON y HTTP) no serán explicados ya que son considerados de dominio general, para mayor información puede visitar sitios como la documentación web MDN en HTTP (HTTP | MDN, s.f) y la documentación web MDN en JSON (JSON - JavaScript | MDX, s.f).

3.3.1. Qué es

GraphQL es un lenguaje de consulta para APIs y un ambiente de ejecución para cumplir las consultas con la data existente (Fundación GraphQL, s.f.). Esta no está ligada a una tecnología, lenguaje de programación o implementación específica, y la definición de lo que espera de las implementaciones se encuentran definidas en la especificación de GraphQL.

GraphQL provee una descripción completa y entendible de los datos en tu API, le da el poder a los clientes el poder de pedir la data exactamente como la necesitan y nada más, hace más fácil evolucionar APIs en el transcurso del tiempo, y habilita poderosas herramientas de desarrollo.

3.3.2. Cómo funciona

GraphQL consiste en una serie de distintas características que vienen de forma inherente con ésta:

- Sistema de tipos (**Type System**):

Sistema lógico que comprende un conjunto de reglas que asignan propiedades denominadas “tipos” a distintos constructos de un programa computacional. Estos “tipos” formalizan y hacen cumplir categorías usadas en la programación para “tipos de datos algebraicos”, “estructuras de datos” u otros componentes como “strings”, “arreglos”, “booleano” y muchos otros.

- Lenguaje de consulta (**Query Language**):

Normalmente denominado parte del “*GraphQL Schema Definition Language*” o “SDL”, funciona como language compartido entre servidor y cliente, y permite definir los requerimientos de datos del cliente, y para información al respecto del esquema de tipos parte de la api.

- Semánticas de ejecución (**Execution Engine**):

Define la ejecución esperada de la solicitud recibida respecto al esquema predefinido junto a sus tipos y resolutores (**resolvers**) de datos respectivos.

- Validaciones estáticas (***Parse & Validation***):

Tomando en consideración el esquema y la solicitud recibida, se realiza una interpretación y validación de la solicitud y datos obtenidos por parte del cliente.

- Introspección de tipos (***Introspection***):

El servidor puede proveer (puede ser omitido para efectos de políticas de seguridad) la posibilidad de otorgar toda la información al respecto del esquema de tipos y datos que el servidor ofrece, lo que permite el desarrollo de herramientas que pueden asegurar “*type safety*”, y proveer de instrumentos útiles para los procesos de desarrollo,

Y todo esto dando soporte para lectura (***query***), escritura (***mutation***) y suscripciones a cambios de datos (***subscription***).

3.3.3. Implementaciones de microservicios

Gracias a como funciona GraphQL, éste permite su implementación en ambientes con arquitecturas de microservicios de forma sencilla sin mayores inconvenientes, pero a lo largo de los años que lleva siendo implementado en muchos ambientes de producción se ha visto la necesidad de implementaciones especializadas para este tipo de arquitecturas, buscando dar satisfacción a necesidades mucho más específicas a la naturaleza del funcionamiento de los microservicios.

A continuación se habla al respecto de dos implementaciones comunes en el ambiente, “Federation” y “Stitching”, ambas con enfoques distintos, y para efectos del proyecto se considerarán ambas implementaciones dependiendo de la evolución y los requerimientos que salen en el transcurso de éste.

3.3.3.1. Federation

Esta implementación fue desarrollada por Apollo GraphQL, y por lo mismo es normalmente denominada como **Apollo Federation** (Apollo GraphQL Company. s.f.).

En enfoque de Apollo Federation busca adaptar servicios que deben ser “federados” para adoptar un estándar que extiende de la existente especificación de GraphQL. En este tipo de arquitectura el servicio “*Federation Gateway*” y los servicios “federados” mantienen una comunicación más explícita de los servicios entre sí y entre el “Gateway” con los servicios. Esto puede resultar conveniente cuando es requerido que los servicios sepan de la existencia de los servicios con los que se espera que funcionen a la par, por lo tanto, se permite una arquitectura de

microservicios en la cual los microservicios se pueden encontrar “acoplados”, pero al mismo tiempo, esto añade una capa extra de complejidad al sistema.

3.3.3.2. Stitching

Esta implementación consiste en la construcción de un “Gateway” que funciona como proxy que une los esquemas y delega las solicitudes a los servicios requeridos respectivos, las últimas versiones de ésta técnica se podrían considerar considerablemente comparables con “Federation” en términos de características (The Guild - GraphQL Tools - s.f.), pero aún así, esta técnica funciona mejor para casos en los cuales los servicios son débilmente acoplados (*loosely coupled*), y existe la posibilidad de su implementación sin la necesidad de adaptar los servicios para que puedan ser usados con este tipo de implementación.

3.4. Entrevista usuario representativo

Durante el proceso de formulación y prediseño del proyecto fue realizada una entrevista de 45 minutos con Jordan Barría, un investigador y estudiante de doctorado en “University of Pittsburgh”. Quien trabaja en el área de interés del proyecto, el cual se puede considerar un usuario representativo de interés con más de 5 años experiencia en el sector de sistemas de aprendizaje inteligente.

Producto de la entrevista, en donde se planteó la propuesta del presente proyecto, se mencionó la importancia de especificaciones que buscan la interoperabilidad de herramientas en el sector de aprendizaje “inteligente”. Las especificaciones mencionadas durante la entrevista fueron “IMS LTI” (IMS Learning Tools Interoperability, s.f) y “xAPI” (What is xAPI aka the Experience API, s.f).

Estas especificaciones (“IMS LTI” y “xAPI”) contribuyeron al proveer contexto y perspectiva para lo esperado de la plataforma, pero al momento de diseñar y desarrollar la plataforma nos encontramos con problemas de adaptación con las tecnologías utilizadas en el proyecto. Un ejemplo fue el uso de GraphQL, el cual impone reglas para la seguridad de tipado y estructuras de datos. Este tipo de reglas de desarrollo finalmente dificultaron la implementación de las especificaciones propuestas, por lo cual se optó por no cumplir estructuralmente con las especificaciones a favor del uso de nuevas tecnologías.

La entrevista además permitió corroborar una serie de requerimientos, como la facilidad de uso, facilidad de implementación, flexibilidad y que otorgue espacio de personalización para distintas posibles necesidades.

4. DISEÑO DE ARQUITECTURA

4.1. Arquitectura De Microservicios Para Learner Modeling

La arquitectura de microservicios para el “learner modeling” permite separar la ejecución, escalamiento y desarrollo de distintas capas y etapas de la lógica de la arquitectura.

Parte del proyecto es el diseño de los microservicios base con los que el sistema debe funcionar de forma fundamental.

4.1.1. Servicio “Users”

Este servicio se encarga de entregar al consumidor de la arquitectura tener la información del usuario autenticado, y la **información relacionada al usuario**, incluyendo por ejemplo, posibles agrupaciones de usuarios.

Este servicio además, en términos de administración del sistema, permite administrar los usuarios, incluyendo, por ejemplo, la creación y asignación de grupos.

4.1.2. Servicio “Projects”

Este servicio se encarga de manejar los **proyectos parte del sistema**, se esperaría por ejemplo que cada tutor sea un proyecto dentro del sistema.

En términos de administración del sistema, permite administrar los proyectos y la asignación de los proyectos a grupos y usuarios del sistema.

4.1.3. Servicio “Domain”

Este servicio se encarga del previamente denominado “modelo de dominio”, en la cual se encarga de categorizar los **dominios, tópicos** y “**KCs**” (*Knowledge Components*) que se manejan del sistema.

En términos de administración del sistema, este permite la creación y ajustes de los dominios, tópicos y “KCs”. Además considerando la posibilidad que los tópicos pueden manejar referencias con otros tópicos, lo que permite definir una estructura o jerarquía de tópicos, pero validando que no se permiten referencias circulares o ciclos.

4.1.4. Servicio “Content”

Este servicio se encarga del previamente denominado “**modelo de contenido**”, el cual a partir del “modelo de dominio”, provee del contenido con una granularidad arbitraria, en la cual cabe definir el contenido como datos arbitrarios que se adaptan al proyecto consumidor, siendo por ejemplo datos “JSON”, datos binarios en forma de base64 (que se almacenan como binarios con objetivos de optimización), o como “URIs” a servicios de terceros.

En términos de administración del sistema se necesita un control total del contenido que se va a manejar, ya que este será el que el aprendiz visualizará directa o indirectamente.

4.1.5. Servicio “Actions”

Este servicio se encarga de manejar los registros históricos de **acciones realizadas por los aprendices** de los distintos proyectos del sistema, en la cual los registros son parte clave de los procesos de análisis y del modelo constante de los usuarios dentro del sistema, y permite también realizar estudios de uso del sistema.

En términos de administración del sistema se resume en la capacidad de navegar los datos en una forma amigable y dar la posibilidad de obtener los datos de forma “bruta”.

4.1.6. Servicio “State”

El servicio “State” permite acceder a los **modelos del aprendiz almacenados y calculados en la plataforma**. Un “State” o más precisamente, un “Model State”, se refiere a la estimación de niveles de conocimiento de un usuario para todos los “KCs” de un dominio y momento específico, éste siendo calculado siguiendo un método específico (existen distintos algoritmos para estimar los modelos de conocimiento o el modelo del aprendiz).

El servicio “State” por lo tanto permite, por ejemplo, obtener el o los últimos modelos calculados para uno o varios usuarios, o todos los modelos de un usuario en orden cronológico, con lo que se permitiría por ejemplo, construir una traza de evolución de estos modelos.

4.1.7. Relación entre servicios

La relación entre los distintos servicios es débilmente acoplada en términos estrictos, ya que la conexión más estrecha entre ellas existe en términos de base de datos relacional, la cual funciona como orquestadora y la fuente de la verdad.

En la Figura 4 podemos ver como los servicios mencionados se encontrarían acoplados en términos conceptuales, por ejemplo, cómo “State” se encuentra conectado directa o indirectamente con otros servicios como “Users”, “Actions”, “Domain”.

Cabe ser mencionado que los procesos de verificación de autenticidad del consumidor del servicio se hace en cada servicio por separado, verificando de forma granular que los datos que se piden coincidan con los proyectos y rol del usuario autenticado.

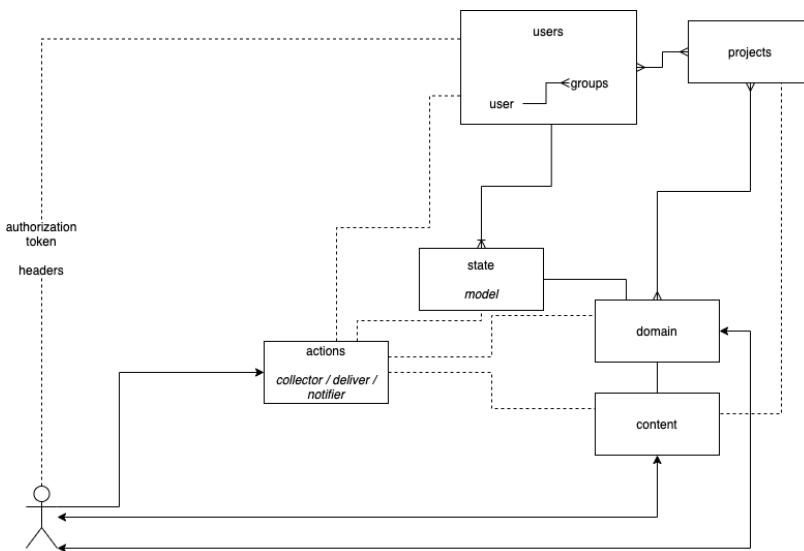


Figura 4. Relación entre servicios de arquitectura⁵

4.2. Autenticación

El proceso de autenticación en el sistema depende del servicio externo Auth0 (Auth0, s.f), el cual es una plataforma especializada que utiliza protocolos seguros como OAuth 2.0. **La autenticación se realiza de forma transparente para el usuario** y simplifica todo el proceso de desarrollo, además de permitir el uso de proveedores externos para inicio de sesión con cuentas externas como Google y Microsoft.

En términos de implementación, las aplicaciones web del lado del cliente envían al servicio tokens encriptados o “JWTs” (RFC 7519 - JSON Web Token, 2015) que

⁵ Diagrama construido utilizando “diagrams.net”, accedido el 25 de enero, 2022, desde <https://www.diagrams.net>

contienen información del usuario, los cuales desencriptan y verifican la identidad del usuario. Posteriormente el servicio automáticamente buscará (o creará si no existe) en la base de datos del sistema la entidad asociada al usuario autenticado, proveyendo información relevante del usuario para el funcionamiento del sistema, como su asociación a algún proyecto.

4.3. Gateway

Con el objetivo de simplificar y facilitar el uso del servicio, se implementa un “Gateway” (puerta de enlace o pasarela en español) que **junta todos los micro-servicios, coordinando las solicitudes del cliente con los servicios correspondientes** de forma eficiente. La presencia de este “Gateway” permite que los clientes apunten hacia un único “*endpoint*” y un único esquema con todo lo ofrecido por el conjunto de servicios.

El funcionamiento de éste es definido principalmente por la librería y técnica utilizada: [@graphql-tools/stitch](#) (Combining Schemas | Schema Stitching, s.f) y “Stitching” respectivamente, las cuales definen ciertas reglas básicas de diseño para hacer factible la combinación de los esquemas de cada servicio:

- Cualquier entidad que quiera ser extendida por más de un servicio necesita un campo “id” definido que debe ser único para la entidad.
- Cada servicio que extiende una entidad necesita una *query* estandarizada, la cual debe aceptar una lista de “ids”, y retorna una lista de las entidades solicitadas en el mismo orden del parámetro de “ids”. Y si cualquiera de las entidades solicitadas no es encontrada, la solicitud entera tiene que fallar.

Una funcionalidad que se espera por parte del “Gateway” es que éste debe estar continuamente “escuchando” a la posible aparición de nuevas instancias de los servicios posteriormente a la inicialización, con el objetivo de reiniciar y actualizar el esquema del conjunto de servicio acorde a posibles cambios en los contratos de tipo. Esto solo se espera que funcione para cambios menores de esquema, ya que cambios que incluyen, por ejemplo, la adición de nuevas entidades, pueden requerir una re-configuración del “Gateway”.

5. MODELAMIENTO DE DATOS Y ENTIDADES

El modelamiento de datos del proyecto sigue un **modelo relacional**. Este se encuentra definido usando Prisma Schema, abstrayendo y automatizando gran parte del modelamiento.

Gran parte del modelamiento fue diseñado a partir del uso y los requerimientos definidos a nivel de cada servicio, por lo cual la categorización de cada entidad está fuertemente relacionada al servicio en el cual se le hace mayor uso, pero no está restringido a solo ser usado por éste.

Cabe considerar que para efectos de “*Separation of Concerns*”, cuando una entidad es extendida por un servicio en el que éste no sea el “dueño” del tipo de entidad, este servicio sólo debe buscar y entregar la información mínima acerca de ella, por ejemplo, cuando el servicio de proyectos extiende la entidad de usuario, éste sólo entregará el campo identificador del usuario junto a las relaciones del usuario con sus proyectos, dejando la responsabilidad de entregar más información del usuario exclusivamente al servicio de usuarios.

5.1. Usuarios

5.1.1. “User”

El modelo de datos para el usuario contiene relaciones que permiten la unión del sistema de datos local con el sistema externo de autenticación, este se ve expresado en la Figura 5, a través de la entidad “UserUID”. Esta entidad contiene la referencia al identificador único definido por Auth0, y una referencia a la entidad de “User”. Esta relación se define como un “*one to many*”, lo que permite que un usuario sea creado de forma independiente de su método de autenticación, y el vínculo con su método de autenticación se haga sobre la marcha, permitiendo, por ejemplo, autenticación con el estándar par de credenciales “correo + contraseña” de forma paralela con autenticación con servicios externos que apuntan al mismo correo, como Google Gmail (Gmail: Free, Private and Secure Email, s.f) o Microsoft Outlook (Outlook - Free personal email and calendar from Microsoft, s.f).

La entidad “User” contiene datos personales de identificación como:

- “email”: Correo electrónico, utilizado como método principal de identificación entre usuarios
- “name”: Nombre o etiqueta opcional, normalmente nombre y/o apellido de la persona.

- “picture”: Fotografía que es automáticamente fijada a partir de los datos obtenidos por los métodos de autenticación externos.

Otros campos de la entidad:

- “tags”: Lista de etiquetas arbitrarias, permiten la clasificación libre de los usuarios.
- “locked”: Permite el bloqueo de un usuario al uso del sistema. Si un usuario bloqueado intenta usar el sistema, este responderá como si no hubiese usuario autenticado.
- “lastOnline”: Fecha del último ingreso del usuario al sistema.
- “role”: Rol del sistema, puede ser un usuario regular “USER” o super-usuario “ADMIN”, por defecto todos los usuarios creados son regulares.

Otros campos que se pueden ver en el esquema son de las relaciones con otras entidades, como “Group”, “Project”, “Action” y “ModelState”, además de los campos de marca de tiempo como “updatedAt” y “createdAt”, que están presentes en la mayor parte de las entidades del sistema.

```

enum UserRole {
    ADMIN
    USER
}

model User {
    id Int @id @default(autoincrement())
    uids UserUID[]

    email String @unique
    name String?
    picture String?

    tags String[]

    locked Boolean @default(false)

    lastOnline DateTime?

    role UserRole @default(USER)

    createdAt DateTime @default(now())
    updatedAt DateTime @default(now()) @updatedAt

    groups Group[]
    projects Project[]
    actions Action[]
    modelStates ModelState[]
}

model UserUID {
    uid String @id

    user User? @relation(fields: [userId], references: [id])
    userId Int?

    createdAt DateTime @default(now())
    updatedAt DateTime @default(now()) @updatedAt
}

```

Figura 5. Modelo de entidad “User”

5.1.2. “Group”

Fuertemente relacionada a la entidad de “User” existe la entidad “Group”, la cual permite agrupar usuarios, permitiendo asociar proyectos, fijar etiquetas y añadir permisos especiales, a todo el conjunto de usuarios.

En la Figura 6 se puede ver la definición del modelo de la entidad “Group”, en la cual podemos definir algunos de sus campos:

- “id”: Identificador numérico único
- “code”: Identificador único arbitrario de cadena de caracteres
- “label”: Identificador o etiqueta legible por humanos que describa brevemente el grupo.
- “tags”: Lista de etiquetas arbitrarias, permiten la clasificación libre de los grupos y de los usuarios parte del grupo.

- “createdAt”: Fecha de creación del grupo.
- “updatedAt”: Fecha de última actualización del grupo.

Además de la relación que permite asignar proyectos al grupo, la entidad de “Group” también permite la definición de permisos específicos a través de la entidad “GroupFlags”. Los “GroupFlags” interactúan directamente con el funcionamiento de los servicios. El permiso “readProjectActions” permite al usuario leer todas las acciones de los proyectos asociados al grupo, y el permiso “readProjectModelStates” permite al usuario leer todos estados de modelo de los proyectos asociados al grupo. Cabe considerar que los usuarios con rol “ADMIN” no necesitan de estos permisos extras para acceder a los datos mencionados.

```

model Group {
    id Int @id @default(autoincrement())
    code String @unique
    label String
    tags String[]
    users User[]
    projects Project[]
    flags GroupFlags?
    createdAt DateTime @default(now())
    updatedAt DateTime @default(now()) @updatedAt
}

model GroupFlags {
    id Int @id @default(autoincrement())
    readProjectActions Boolean @default(false)
    readProjectModelStates Boolean @default(false)
    createdAt DateTime @default(now())
    updatedAt DateTime @default(now()) @updatedAt
    group Group @relation(fields: [groupId], references: [id])
    groupId Int @unique
}

```

Figura 6. Modelo de entidad “Group”

En la Figura 7 se puede visualizar como los modelos “User” y “Group” interactúan en un diagrama de entidad relación, a la par con las entidades mencionadas en ésta sección.

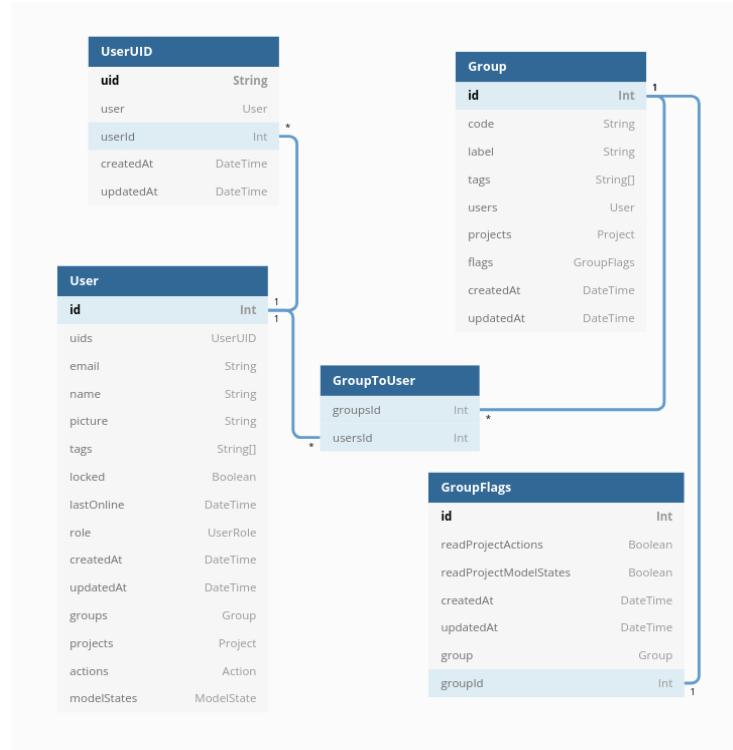


Figura 7. Diagrama de entidad relación entre “User” y “Group”⁶

5.2. Proyectos

El modelo de datos para el concepto de “Proyecto”, el cual define la forma principal de división del uso de la plataforma. Esta entidad es definida en relaciones con otras entidades, permitiendo definir el contexto global. En la Figura 8 se puede visualizar los vínculos directos que existen entre “Project” y las entidades “User”, “Group”, “Domain”, “Topic”, “Content” y “Action”.

Los campos no relacionales definidos en el modelo incluyen:

- “id”: Identificador numérico único, definido automáticamente por parte del RDBMS.
- “code”: Identificador único arbitrario de cadena de caracteres.
- “label”: Identificador o etiqueta legible por humanos que describa brevemente el proyecto.
- “createdAt”: Fecha de creación de la entidad.
- “updatedAt”: Fecha de última actualización de la entidad.

⁶ Diagrama construído utilizando dbdiagram.io - *Database relationship Diagrams Design Tool*, accedido el 25 de enero, 2022, desde <https://dbdiagram.io>

```

model Project {
    id Int @id @default(autoincrement())

    code String @unique
    label String

    users User[]
    groups Group[]

    domains Domain[]
    topics Topic[]

    content Content[]

    actions Action[]

    createdAt DateTime @default(now())
    updatedAt DateTime @default(now()) @updatedAt
}

```

Figura 8. Modelo de entidad “Project”

5.3. Dominio

El modelamiento de datos para las entidades relacionadas con el “Modelo de dominio” se dividió entre las entidades “Domain”, “Topic” y “KC”.

En la Figura 9 se puede visualizar como los modelos de esta sección en conjunto con la entidad “Project”, interactúan en un diagrama de entidad relación.



Figura 9. Diagrama de entidad relación entre “Domain”, “KC”, “Project” y “Topic”⁷

5.3.1. “Domain”

La entidad “Domain”, como se puede ver en la Figura 10, en este solo se definen como campos:

- “id”: Identificador numérico único, definido automáticamente por parte del RDBMS.
- “code”: Identificador único arbitrario de cadena de caracteres.
- “label”: Identificador o etiqueta legible por humanos que describa brevemente el dominio.
- “createdAt”: Fecha de creación del dominio.
- “updatedAt”: Fecha de última actualización del dominio.

⁷ Diagrama construido utilizando dbdiagram.io

```

model Domain {
    id Int @id @default(autoincrement())

    code String @unique
    label String

    projects Project[]

    kcs KC[]

    modelStates ModelState[]

    createdAt DateTime @default(now())
    updatedAt DateTime @default(now()) @updatedAt
}

```

Figura 10. Modelo de entidad “Domain”

La entidad “Domain” define una relación con la entidad “Project”, donde un dominio puede estar asociado con más de un proyecto, y un proyecto tener más de un dominio. Esta entidad también define una relación con la entidad “KC” (abreviación de Knowledge Component), en donde un “KC” tiene siempre un único “Domain” asociado, y un “Domain” puede tener más de un “KC”, y la relación entre “Domain” y “ModelState” es de la misma naturaleza, en la cual un “ModelState” tiene siempre un único “Domain” asociado, mientras que un “Domain” puede tener más de un “ModelState”.

5.3.2. “Topic”

Parte de esta sección de modelo de datos incluye el concepto definido con la entidad “Topic”, el cual es de gran importancia para el “Modelo de dominio”, ya que permite definir la estructura de tópicos o temas para la organización de contenido, esto se logra a través de la definición de relaciones “recursivas”, permitiendo definir estructuras en forma de árbol.

En la Figura 11 se puede visualizar como la entidad “Topic” es descrita, definiendo los campos:

- “id”: Identificador numérico único, definido automáticamente por parte del RDBMS.
- “code”: Identificador único arbitrario de cadena de caracteres.
- “label”: Identificador o etiqueta legible por humanos que describa brevemente el dominio.
- “sortIndex”: Valor numérico opcional que puede ser utilizado para determinar el orden de los tópicos que comparten “parent”.
- “tags”: Lista de etiquetas arbitrarias, permiten la clasificación libre de los tópicos.
- “createdAt”: Fecha de creación del tópico.
- “updatedAt”: Fecha de última actualización del tópico.

```

model Topic {
    id Int @id @default(autoincrement())

    code String @unique
    label String

    sortIndex Int?

    tags String[]

    project Project @relation(fields: [projectId], references: [id])
    projectId Int

    parent Topic? @relation("TopicToTopic", fields: [parentId], references: [id])
    parentId Int?
    childrens Topic[] @relation("TopicToTopic")

    content Content[]
    kcs KC[]
    actions Action[]

    createdAt DateTime @default(now())
    updatedAt DateTime @default(now()) @updatedAt

    @index([id, projectId])
}

```

Figura 11. Modelo de entidad “Topic”

La entidad “Topic” define relaciones con las entidades “Project” para definir el contexto de utilización del tópico, relación con la entidad “Content”, la cual utiliza los tópicos para ordenar y clasificar el contenido. Existe la relación del tópico con “KC”s, permitiendo agrupar los tópicos, y finalmente la relación con “Action”, la cual permite identificar el tópico de la acción registrada por las aplicaciones.

5.3.3. “KC”

La última entidad definida dentro del contexto de modelo de dominio es “KC”, la cual es una abreviación de “*Knowledge Component*”, el cual dentro de plataforma a nivel de manejo de datos es utilizado para la categorización, por ejemplo, de los tópicos y contenido. Mientras que a nivel conceptual, este define un concepto o elemento de aprendizaje, el cual para un tutor de aprendizaje, el modelo del aprendiz es construído estimando los niveles de mastería en todos los “KCs” de los dominios en base a la evidencia del contenido “superado” por el aprendiz (las acciones reportadas por las aplicaciones producto del uso del contenido y “KCs” involucrados).

La entidad “KC”, como se puede ver en la Figura 11, define los campos:

- “id”: Identificador numérico único, definido automáticamente por parte del RDBMS.
- “code”: Identificador único arbitrario de cadena de caracteres.
- “label”: Identificador o etiqueta legible por humanos que describa brevemente el “Knowledge Component”.
- “createdAt”: Fecha de creación del “KC”.
- “updatedAt”: Fecha de última actualización del “KC”.

```

model KC {
    id Int @id @default(autoincrement())
    code String @unique
    label String

    domain Domain @relation(fields: [domainId], references: [id])
    domainId Int

    topics Topic[]
    content Content[]
    actions Action[]

    createdAt DateTime @default(now())
    updatedAt DateTime @default(now()) @updatedAt
}

```

Figura 12. Modelo de entidad “KC”

5.4. Acciones

Parte del proceso de modelado incluye la necesidad de registrar las acciones de los usuarios al utilizar aplicaciones relacionadas a la plataforma, por lo cual el modelado para guardar estos datos tiene que ser flexible, y al mismo tiempo necesita tener una estructura que permita un nivel de predictibilidad que facilite la programación alrededor de los datos.

Un “Action” está básicamente compuesto por el usuario que realizó la acción, un descriptor de la acción, que es denominado “Verb”, y detalles de la acción realizada.

Cabe destacar que el componente principal “Verb” es descrito de forma flexible, en el cual, cuando una acción es emitida, si el verbo especificado no existe, el nuevo “verbo” es creado automáticamente antes de ser asociado a la nueva acción.

En la Figura 13 se puede visualizar como las entidades “Action” y “ActionVerb” son descritas, definiendo los campos:

- “id”: Identificador numérico único, definido automáticamente por parte del RDBMS.
- “verbName”: Identificador de “verbo” asociado a la acción.
- “timestamp”: Marca de tiempo enviada por aplicación emisora de la acción.
- “result”: Valor numérico flotante arbitrario y opcional, normalmente asociado a un resultado satisfactorio o fallido.
- “stepID”: Identificador arbitrario opcional de posible etapa actual de acción.
- “hintID”: Identificador arbitrario opcional de posible indicación visible por el usuario.
- “amount”: Valor numérico flotante arbitrario y opcional, normalmente asociado a una cantidad asociada a la acción realizada.
- “detail”: Valor de cadena de caracteres arbitrario que define detalles de la acción realizada.
- “extra”: Valor completamente flexible que debe seguir el formato JSON, que puede dar detalles complejos de la acción.
- “createdAt”: Fecha de creación del “Action”.
- “updatedAt”: Fecha de última actualización del “KC”.

La entidad también define relaciones que permiten definir el contexto, entidades que se definen para esto son:

- “Project”: El proyecto que emite la acción. Obligatorio y normalmente asociado a la aplicación responsable de la acción.
- “User”: El usuario que emite la acción. Obligatorio al momento del reporte de las acciones, pero es definido como opcional a nivel de base de datos para otorgar un extra de flexibilidad.
- “Content”: Contenido relacionado con acción reportada. Si la acción tiene relación directa con un contenido servido por la plataforma, se recomienda reportar el identificador para facilitar el seguimiento de las acciones.

```

model ActionVerb {
    id Int @id @default(autoincrement())

    name String @unique

    actions Action[]

    createdAt DateTime @default(now())
    updatedAt DateTime @default(now()) @updatedAt
}

model Action {
    id Int @id @default(autoincrement())

    verb     ActionVerb @relation(fields: [verbName], references: [name])
    verbName String

    timestamp DateTime

    result Float?

    project   Project @relation(fields: [projectId], references: [id])
    projectId Int

    user     User? @relation(fields: [userId], references: [id])
    userId Int?

    content   Content? @relation(fields: [contentId], references: [id])
    contentId Int?

    topic     Topic? @relation(fields: [topicId], references: [id])
    topicId Int?

    kcs      KC[]

    stepID String?
    hintID String?

    amount Float?

    detail String?

    extra Json?

    createdAt DateTime @default(now())
    updatedAt DateTime @default(now()) @updatedAt
}

```

Figura 13. Modelo de entidad “Action”

5.5. Contenido

Parte de la plataforma incluye la posibilidad de servir contenido que será utilizado en aplicaciones como tutores inteligentes, esta entidad tiene que ser flexible, permitir búsqueda precisa y categorización usando entidades relacionadas como “Topic” y “KC”.

En la Figura 14 se puede visualizar como la entidad “Content” es descrita, definiendo campos como:

- “id”: Identificador numérico único, definido automáticamente por parte del RDBMS.

- “code”: Identificador único arbitrario de cadena de caracteres.
- “label”: Identificador o etiqueta legible por humanos que describa brevemente el contenido.
- “sortIndex”: Valor numérico opcional que puede ser utilizado para determinar el orden del contenido que comparte la misma jerarquía.
- “tags”: Lista de etiquetas arbitrarias, permiten la clasificación libre del contenido
- “description”: Descripción detallada del contenido.
- “binary” y “binaryFilename”: Almacenamiento de contenido binario, en la cual ambos campos deben estar presentes de forma simultánea, y el nombre del archivo debe contener la extensión de la cual se puede inferir el tipo de archivo a consumir.
- “json”: Almacenamiento de contenido completamente libre que solo necesita ser de formato JSON.
- “url”: Referencia a contenido externo a través de un enlace web.
- “createdAt”: Fecha de creación del contenido.
- “updatedAt”: Fecha de última actualización del contenido.

La entidad “Content” define relaciones que permiten categorizar y facilitar la utilización de éste dentro de la plataforma, entre estos se incluyen:

- “Project”: El proyecto al cual el contenido pertenece. Obligatorio ya que es requerido para efectos de seguridad y categorización.
- “Topics”: Principal forma de categorización de los contenidos.
- “KCs”: Asociación del contenido con “KCs”. Especialmente útil para procesos de modelado y búsqueda de contenido.
- “Actions”: Asociación de acciones con el contenido. Esta relación no se construye desde esta entidad, sino más bien por parte de la entidad de “Action”.

```

model Content {
    id Int @id @default(autoincrement())

    code String @unique
    label String

    sortIndex Int?

    tags String[]

    project Project @relation(fields: [projectId], references: [id])
    projectId Int

    topics Topic[]

    kcs KC[]

    actions Action[]

    description String

    binary Bytes?
    binaryFilename String?

    json Json?
    url String?

    createdAt DateTime @default(now())
    updatedAt DateTime @default(now()) @updatedAt
}

```

Figura 14. Modelo de entidad “Content”

5.6. Estado de modelo

Uno de los grandes objetivos de la plataforma es permitir la vinculación de gran parte de los datos provistos y entregar modelamientos de recomendación o predicción, por lo cual fue necesario el diseño de modelamiento de almacenamiento y datos para ésto, la cual debe ser flexible, pero a la vez permita automatización.

Como se puede ver en la Figura 15, la definición de la entidad “ModelState” incluye la definición de dos entidades extras, “ModelStateType” y “ModelStateCreator”.

La forma más sencilla de categorización de distintos modelamientos almacenados de forma contigua es siendo divididas por “tipo”, para lo cual se creó una entidad paralela “ModelStateType”, que permita filtrar y categorizar de forma sencilla. Al momento de crear una nueva instancia de un modelo, ésta automáticamente será creada si no existía previamente.

Para el caso de “ModelStateCreator”, la separación de los modelos acorde de la aplicación que la creó es de gran relevancia, permitiendo un nivel extra de profundidad y flexibilidad al momento de buscar, crear y utilizar modelamientos creados por lógicas externas.

La entidad “ModelState” está constituida principalmente por:

- “id”: Identificador único numérico incremental, definido automáticamente por el RDBMS.
- “user”: El usuario al que el modelamiento tiene objetivo.
- “type”: Tipo de modelamiento.
- “creator”: Creador responsable del modelamiento.
- “domain”: Dominio que define contexto del modelamiento
- “json”: Contenido de modelamiento del aprendiz, el cual contiene estimaciones de nivel de aprendizaje y habilidad realizados por los métodos de modelamiento, normalmente relacionados con un subconjunto de “Knowledge components” del dominio asociado. Esta definición de campo es completamente flexible con el objetivo de poder contener distintos tipos de modelo en formato JSON.
- “createdAt”: Fecha de creación del modelamiento.
- “updatedAt”: Fecha de última actualización del modelamiento.

Debido a la naturaleza y la posibilidad de definir un modelamiento histórico, los campos de marca de tiempo, en conjunto con identificador numérico incremental tienen una importancia extra con respecto a las demás entidades, ya que permiten definir los “modelamientos actuales” versus estados del modelamiento previos.

```

model ModelStateType {
    id Int @id @default(autoincrement())
    name String @unique
    states ModelState[]
    createdAt DateTime @default(now())
    updatedAt DateTime @default(now()) @updatedAt
}

model ModelStateCreator {
    id Int @id @default(autoincrement())
    name String @unique
    states ModelState[]
    createdAt DateTime @default(now())
    updatedAt DateTime @default(now()) @updatedAt
}

model ModelState {
    id Int @id @default(autoincrement())
    user User @relation(fields: [userId], references: [id])
    userId Int
    stateType ModelStateType? @relation(fields: [type], references: [name])
    type String?
    stateCreator ModelStateCreator @relation(fields: [creator], references: [name])
    creator String
    domain Domain @relation(fields: [domainId], references: [id])
    domainId Int
    json Json
    createdAt DateTime @default(now())
    updatedAt DateTime @default(now()) @updatedAt
}

```

Figura 15. Modelo de entidad “ModelState”

6. DESARROLLO DE SERVICIOS

Parte esencial del proceso de desarrollo del proyecto constituyó la programación de los servicios que conforman la plataforma. Esto fue realizado utilizando el lenguaje de programación TypeScript, y consideró aspectos como el proceso de autenticación, autorización, y el desarrollo de servicios de usuarios, proyectos, dominio, contenido, acciones y de estado de modelamiento. Además se incorporó el servicio de “Gateway”, el cual orquesta los servicios para una utilización sencilla del conjunto de los servicios.

En la Tabla 2 se muestra como los servicios de la plataforma interactúan con las diferentes entidades definidas en la plataforma. Si un servicio define una entidad, este sirve todos los campos principales de la entidad de la base de datos, mientras que si un servicio extiende una entidad, el servicio le añade campos y/o relaciones con otras entidades. Finalmente, un servicio puede por ejemplo usar una entidad externa para extender una entidad definida dentro del servicio o para el caso de entidades como “User”, “Group” y “Project”, ser utilizadas para procesos de autenticación y autorización.

Tabla 2. Definición, extensión y utilización de entidades por servicios

	Servicio <i>Users</i>	Servicio <i>Domains</i>	Servicio <i>Projects</i>	Servicio <i>Content</i>	Servicio <i>Actions</i>	Servicio <i>State</i>
Entidad <i>User</i>	Define	Utiliza	Extiende	Utiliza	Utiliza	Extiende
Entidad <i>Group</i>	Define	Utiliza	Extiende	Utiliza	Utiliza	Utiliza
Entidad <i>Project</i>	Extiende	Extiende	Define	Extiende	Extiende	Utiliza
Entidad <i>Domain</i>	N/A	Define	Extiende	N/A	N/A	Extiende
Entidad <i>Topic</i>	N/A	Define	Extiende	Extiende	Utiliza	N/A
Entidad <i>KC</i>	N/A	Define	N/A	N/A	Utiliza	Utiliza ⁸
Entidad <i>Action</i>	N/A	N/A	N/A	N/A	Define	Utiliza ⁹
Entidad <i>Content</i>	N/A	Extiende	Extiende	Define	Utiliza	N/A
Entidad <i>ModelState</i>	N/A	N/A	N/A	N/A	N/A	Define

⁸ La utilización de la entidad “KC” es de forma indirecta

⁹ La utilización de la entidad “Action” es de forma indirecta

6.1. Documentación

A la par con desarrollo de los servicios se realizó un proceso de escritura de documentación, el cual fue dividido en dos partes, la primera es la documentación y descripción de todos los campos y tipos definidos en los esquemas, lo que permite facilitar el entendimiento y el uso de los servicios “backend” provistos por la plataforma. El segundo tipo de documentación desarrollado fue la escritura descriptiva de partes y procesos esenciales para el desarrollo y continuación de la programación en el proyecto, este se encuentra disponible en el sitio web <https://docs.lm.inf.uach.cl>. La documentación contiene:

- Instrucciones y explicaciones de cómo iniciar el ambiente de desarrollo del proyecto
- Como la arquitectura utilizada funciona y qué librerías son parte fundamental de la programación
- Ejemplos de implementación para el sistema de autenticación
- Información relevante acerca de la implementación para el sistema de base de datos
- Cómo el “Gateway” se configura
- Cómo se usa la API y herramientas de desarrollo útiles para este objetivo
- Cómo usar el panel de administración
- Descripción de plantilla de desarrollo para aplicaciones web
- Pasos recomendados para la creación de un nuevo servicio
- Cómo trabajar con el modelado de la base de datos dentro de la plataforma
- Como usar y crear herramientas para interactuar directamente con la base de datos
- Instrucciones de despliegue de la plataforma

Cabe considerar que todo el proceso de documentación y la programación en sí fue realizada en el idioma inglés, ya que esto facilita considerablemente el uso de términos nativos del lenguaje y la familiaridad de la programación en inglés.

Todos los servicios hacen uso de una lógica compartida utilizada para el proceso de autenticación y autorización, lo cual se profundizará en la siguiente sección. Luego se prosigue con el análisis y descripción en profundidad del servicio de usuarios a modo de ejemplo, considerando que la mayor parte de los servicios comparten una cantidad considerable de lógica y el entendimiento de éste implica la comprensión general del funcionamiento de todos los servicios actuales.

6.2. Autenticación y Autorización

6.2.1. Autenticación

El proceso de autenticación fue considerablemente simplificado gracias al uso del servicio externo Auth0. Esto permitió, por ejemplo, el uso de librerías open-source como “fastify-auth0-verify” (GitHub - nearform/fastify-auth0-verify, s.f) para la verificación de los tokens de autenticación en los servicios, en la cual el “Gateway” y los servicios esperan el token obtenido por Auth0 a través del “HTTP header” estándar “authorization” con el formato “Bearer {jwt}”.

Por el lado de las aplicaciones que harán uso de la plataforma, la utilización de librerías como “@auth0/auth0-react” (GitHub - auth0/auth0-react, s.f) facilitan el desarrollo de la autenticación. Esta librería se encuentra dentro de tanto la plantilla de desarrollo y del panel de administración, los cuales fueron parte importante del proyecto.

Los servicios de la plataforma hacen un uso compartido de código. Por ejemplo, en la Figura 16 se puede visualizar un fragmento del código ejecutado para la obtención del usuario asociado al token encriptado por parte del cliente, en la cual el “uid” en conjunto con el “email” son usados como los identificadores usados para buscar, crear y actualizar la entidad del usuario. Actualizando el campo “lastOnline”, utilizando los datos “name” y “picture” datos por Auth0 si estos no se encontraban previamente definidos, y bloquear la autenticación del usuario si este contiene un valor “true” en el campo “locked”.

6.2.2. Autorización

El proceso de autorización depende del resultado obtenido del proceso de autenticación, en la cual a partir de la información obtenida del usuario, cada resolutor relevante de cada servicio utiliza reglas, en su mayor parte predefinidas, que verifican la presencia del usuario, en conjunto con cualquier otra posible regla adicional. Estas reglas utilizan en su mayor parte aserciones, encapsuladas en forma de “Lazy Promises”, las cuales solo son ejecutadas una sola vez, y son ejecutadas sólo cuando estas son esperadas, normalmente usando la palabra clave “await”.

En la Figura 17 se encuentra un fragmento de la definición de algunas reglas, como lo es:

- “expectUser”: Verifica la existencia del usuario, en el cual si este no existe, se para la ejecución

- “expectAdmin”: Re-utiliza “expectUser”, y verifica que el usuario tenga el rol de “ADMIN”, y si no es así, se para la ejecución.
- “expectUserProjectsSet”: Re-utiliza “expectUser” y retorna el conjunto de “ids” de todos los proyectos del usuario, incluyendo los proyectos de los grupos en el cual el usuario es parte.

```

export function GetDBUser(auth0UserPromise: Promise<Auth0User | null>) {
  return LazyPromise(async () => {
    const user = await auth0UserPromise;

    if (!user) return null;

    const { sub: uid, email, name, picture } = user;

    if (!uid || !email) return null;

    const lastOnline = new Date();
    const userDb = await prisma.userUID
      .upsert({
        where: {
          uid,
        },
        create: {...},
        update: {...},
        select: {
          user: {
            include: {
              projects: {
                select: {
                  id: true,
                },
              },
              groups: {
                select: {
                  projects: {
                    select: {
                      id: true,
                    },
                  },
                  flags: {
                    select: {
                      readProjectActions: true,
                      readProjectModelStates: true,
                    },
                  },
                },
              },
            },
          },
        }
      })
      .then((data) => (data.user?.locked ? null : data.user));

    if (userDb && ((!userDb.name && name) || (!userDb.picture && picture))) {
      const updatedUser = await prisma.user.update({
        where: {...},
        data: {
          name: userDb.name ? undefined : (name || undefined),
          picture: userDb.picture ? undefined : (picture || undefined),
        },
        select: {
          name: true,
          picture: true,
        },
      });

      Object.assign(userDb, updatedUser);
    }

    return userDb;
  });
}

```

Figura 16. Fragmento de código en proceso de autenticación

```

export const Authorization = (userPromise: Promise<DBUser | null>) => {
  const expectUser = LazyPromise(async () => {
    const user = await userPromise;

    assert(user, "Forbidden!");

    return user;
  });
  const expectAdmin = LazyPromise(async () => {
    const user = await expectUser;

    assert(user.role === "ADMIN", "Forbidden");

    return user;
  });

  const expectUserProjectsSet = LazyPromise(async () => {
    const user = await expectUser;

    const projectIds = new Set<number>();

    for (const { id } of user.projects) {
      projectIds.add(id);
    }
    for (const { projects } of user.groups) {
      for (const { id } of projects) {
        projectIds.add(id);
      }
    }

    return projectIds;
  });

```

Figura 17. Fragmento de código en reglas de autorización

6.3. Usuarios

El servicio de usuarios consiste principalmente en la gestión de usuarios y grupos de la plataforma, incluyendo también la obtención del usuario en el proceso de autenticación.

Al igual que en todos los servicios y “Gateway”, estos son implementados utilizando librerías y “frameworks” como:

- Fastify¹⁰ - *Framework* web de alto rendimiento para Node.js
- GraphQL EZ¹¹ - Librería para construcción y despliegue de APIs GraphQL basado en plugins y aprovecha la utilización del sistema de plugins de ejecución llamado Envelop¹².

¹⁰ Fastify - <https://www.fastify.io>

¹¹ GraphQL EZ - <https://www.graphql-ez.com>

¹² Envelop - <https://www.envelop.dev>

6.3.1. Esquema

6.3.1.1. Entidades base

El servicio de usuarios define la entidad base, que es utilizada en distintas partes del servicio, como se puede ver en la Figura 19, este extiende el modelo “User” definido por el modelo de base de datos, como por ejemplo la asociación de usuarios con grupos, la asignación de usuarios con proyectos específicos. La entidad “User” también contiene campos como “active”, que permite saber de forma rápida si un usuario ha hecho uso del sistema, el campo “locked” que puede ser activado para bloquear el acceso del usuario a la plataforma, y campos de identificación personal como “email”, “name” y “picture”.

```
"User entity"
type User {
    """
    Active flag

    By default it starts as "false", and the first time the user accesses the system, it's set as "true"
    """

    active: Boolean!
    "Date of creation"
    createdat: Datetime!
    "Email Address"
    email: String!
    "Groups associated with the user"
    groups: [Group!]!
    "Unique numeric identifier"
    id: IntID!
    "Date of latest user access"
    lastOnline: DateTime
    """

    Locked user authentication

    If set as "true", user won't be able to use the system
    """

    locked: Boolean!
    "Name of person"
    name: String
    "Picture of user, set by external authentication service"
    picture: String
    "IDs of projects associated with the user"
    projectsIds: [IntID!]!
    "User role, by default is USER"
    role: UserRole!
    """

    Tags associated with the user

    Tags can be used to categorize or filter
    """

    tags: [String!]!
    "Date of last update"
    updatedAt: DateTime!
}

"Possible roles of an authenticated user"
enum UserRole {
    """
    Administrator of the system

    Most of the authorization logic is enabled
    """

    ADMIN
    "Default user role"
    USER
}
```

Figura 18. Esquema base entidad “User”

Otra entidad base definida en el servicio es “Group”. Esta entidad, igual que la mayor parte de las entidades de la plataforma, extiende en gran parte el modelo “Group” del modelo de base de datos. Como se puede ver en la Figura 19, esta define las asociaciones con la entidad “User”, la asociación con proyectos, los cuales son heredados por los usuarios, además de incluir campos estándar de identificación como “id”, “code”, “label” y “tags”.

Como ya fue mencionado previamente en el modelado de datos, los grupos también permiten dar privilegios funcionales dentro de la plataforma. Estos privilegios se encuentran definidos en la entidad “GroupFlags” con campos como “readProjectActions”, que permite la lectura de las acciones de los proyectos del grupo y “reactProjectModelStates”, que permite al usuario la lectura de los estados de modelo de los proyectos del grupo.

```

"""
Group Entity

- Used to group/cluster users
- Set permissions flags to the users
- Associate projects to users, allowing users to access the projects
"""

type Group {
    "Unique string identifier"
    code: String!
    "Date of creation"
    createdAt: DateTime!
    "Permissions flags"
    flags: GroupFlags!
    "Unique numeric identifier"
    id: IntID!
    "Human readable identifier"
    label: String!
    "IDs of projects associated with the group"
    projectsIds: [IntID!]!
    """
    Tags associated with the group

    Tags can be used to categorize or filter
    """
    tags: [String!]!
    "Date of last update"
    updatedAt: DateTime!
    "Users associated with the group"
    users: [User!]!
}

"Permissions flags of group"
type GroupFlags {
    "Date of creation"
    createdAt: DateTime!
    "Unique numeric identifier"
    id: IntID!
    "Allows the users part of the group to read all the actions of the projects of the group"
    readProjectActions: Boolean!
    "Allows the users part of the group to read all the model states of the projects of the group"
    readProjectModelStates: Boolean!
    "Date of last update"
    updatedAt: DateTime!
}

```

Figura 19. Esquema base entidad “Group”

6.3.1.2. Puntos de entrada

El esquema del servicio de usuarios, como gran parte de los servicios de plataforma, consiste primero en los puntos de entrada “Query” y “Mutation”.

Como se puede ver en la Figura 20, “Query” contiene consultas de administración, información de la autenticación, consultas de grupos y usuarios por “ids”, y una solicitud de prueba. Mientras que “Mutation” solo contiene mutaciones de administración y de prueba.

```

type Query {
    """
    Admin related user queries, only authenticated users with the role "ADMIN" can access
    """
    adminUsers: AdminUserQueries!
    "Authenticated user information"
    currentUser: User
    """
    Get all the groups associated with the specified identifiers

    The groups data is guaranteed to follow the specified identifiers order

    If any of the specified identifiers is not found or forbidden, query fails
    """
    groups(ids: [IntID!]!): [Group!]!
    "Returns 'Hello World!'"
    hello: String!
    """
    Get all the users associated with the specified identifiers

    The users data is guaranteed to follow the specified identifiers order

    If any of the specified identifiers is not found or forbidden, query fails
    """
    users(ids: [IntID!]!): [User!]!
}

type Mutation {
    """
    Admin related user mutations, only authenticated users with the role "ADMIN" can access
    """
    adminUsers: AdminUserMutations!
    "Returns 'Hello World!'"
    hello: String!
}

```

Figura 20. Esquema “Query” y “Mutation” de servicio de usuarios

6.3.1.3. Administración

Para la administración de la plataforma “Query” y “Mutation” utilizan los tipos “AdminUserQueries” y “AdminUserMutations” respectivamente, los cuales como se puede ver en la Figura 21, definen, por ejemplo, la navegación paginada de los usuarios y grupos, además de incluir métodos de manipulación de las entidades de usuarios y grupos.

```

"Admin User-Related Queries"
type AdminUserQueries {
    """
    Get all the groups currently in the system

    Pagination parameters are mandatory, but filters is optional, and therefore the search can be customized.
    """
    allGroups(
        filters: AdminGroupsFilter
        pagination: CursorConnectionArgs!
    ): GroupsConnection!
    """
    Get all the users currently in the system

    Pagination parameters are mandatory, but filters is optional, and therefore the search can be customized.
    """
    allUsers(
        filters: AdminUsersFilter
        pagination: CursorConnectionArgs!
    ): UsersConnection!
}

"Admin User-Related Mutations"
type AdminUserMutations {
    "Create a new group entity"
    createGroup(data: CreateGroupInput!): Group!
    "Set the projects of the specified users"
    setProjectsToUsers(projectIds: [IntID!]!, userIds: [IntID!]!): [User!]!
    "Set the users (by email) associated with the groups"
    setUsersGroups(groupIds: [IntID!], userEmails: [EmailAddress!]!): [Group!]!
    "Update an existent group entity"
    updateGroup(data: UpdateGroupInput!): Group!
    "Update an existent user entity"
    updateUser(data: UpdateUserInput!): User!
    "Upsert specified users with specified projects"
    upsertUsersWithProjects(
        emails: [EmailAddress!]!
        projectsIds: [IntID!]!
    ): [User!]!
}

```

Figura 21. Esquema consultas administrativas de servicio de usuarios

6.3.2. Resolutores

Los resolutores (“*resolvers*” en inglés), al igual que en todos los servicios, son divididos en módulos, en los cuales cada módulo define sus tipos a la par con sus “*resolvers*”, los cuales son unidos para formar un único esquema por servicio, y en el caso específico del servicio de usuarios, los módulos definidos son “usersModule”, “groupsModule”, y “projectsModule”.

6.3.2.1. Módulo de usuarios

Este módulo además de definir la entidad “User”, vista en la Figura 19, define como ciertos campos y “queries” relacionadas directamente con la entidad de usuario deben ser ejecutadas.

Un ejemplo de campo parte del usuario que extiende la entidad del modelo de base de datos es “active”, el cual como se puede ver en la Figura 22, simplemente funciona como un booleano que distingue si un usuario ha hecho uso de la

plataforma al menos una vez a partir del campo de base de datos “lastOnline”, el cual contiene la fecha del último acceso del usuario.

```
User: {
  active({ lastOnline }) {
    return lastOnline != null;
  },
},
```

Figura 22. Ejemplo de resolutor de campo “active” de entidad usuario

Parte importante de la lógica para la administración del sistema es poder asegurar que las consultas y mutaciones administrativas solo son accedidas por parte de usuarios con el rol de tipo “admin”, por lo cual, como se puede ver en la Figura 23, los tipos intermediarios “AdminUserQueries” y “AdminUserMutations”, a la par con sus campos respectivos en “Query” y “Mutation” llamados ambos “adminUsers”, hacen uso de las reglas de autorización como fueron mencionadas en el capítulo 6.2.2, al igual que en todos los módulos de la plataforma que extienden el sistema de administración.

```
Mutation: {
  async adminUsers(_root, _args, { authorization }) {
    await authorization.expectAdmin;

    return {};
  },
},
Query: {
  async adminUsers(_root, _args, { authorization }) {
    await authorization.expectAdmin;

    return {};
  },
},
```

Figura 23. Resolutores con verificación de administración para módulo de usuarios

El presente módulo hace uso de la lógica de autenticación y permite retornar información relevante del usuario como parte de la plataforma. Esto se encuentra

disponible a través de la solicitud “Query.currentUser”, y su resolutor, que puede verse en la Figura 24, re-utiliza la promesa “UserPromise” provista a través del contexto de la ejecución.

Como fue mencionado en el capítulo “Gateway” de diseño de arquitectura, todos los servicios que definan y extiendan una entidad que va a ser re-utilizada por otro servicio debe definir una operación estandarizada de la entidad, para el caso de la entidad de usuario la operación es denominada “users”. Como se puede ver en la Figura 25, “users” acepta una lista de identificadores y hace uso de una regla de autorización que verifica que el usuario autenticado solo puede leer de usuarios que sean parte de sus proyectos. Si el usuario tiene el rol de administrador, ignora cualquier restricción.

```
async currentUser(_root, _args, { UserPromise }) {
  const user = await UserPromise;

  if (user === null) return null;

  return user as Omit<typeof user, "groups">;
},
```

Figura 24. Resolutor de “Query.currentUser”

```
async users(_root, { ids }, { prisma, authorization }) {
  return getNodeIdsList(
    prisma.user.findMany({
      where: {
        id: {
          in: ids,
        },
        projects: await authorization.expectSomeProjectsInPrismaFilter,
      },
    }),
    ids
  );
},
```

Figura 25. Resolutor de “Query.users”

Como fue mencionado previamente, parte importante de la presente plataforma es la posibilidad de la administración a través de las operaciones definidas en los tipos con prefijo “Admin” de los esquemas.

En la Figura 26 se puede visualizar como las operaciones “upsertUsersWithProjects” y “updateUser” son definidas. Para el caso de la primera, ésta recibe una lista de emails e identificadores del proyecto a los cuales los usuarios serán asociados (la lista de identificadores de proyectos puede ser vacía). Esta operación utiliza una función auxiliar llamada “pMap”, la cual limita la ejecución

paralela de operaciones asíncronas a lotes de máximo 4 en este caso, creando cada usuario de forma individual si este no existía previamente. Una vez que el usuario ya se encuentra dentro del sistema, los proyectos especificados son asignados.

Para la necesidad de especificar ciertos campos y configuraciones asociadas al usuario existe la operación “updateUser”, que se recibe el identificador del usuario objetivo más los atributos que se desean cambiar, como los proyectos a ser asociados directamente al usuario, nombre, etiquetas o estado de bloqueo.

Finalmente el módulo de usuarios finaliza definiendo el resolutor “AdminUserQueries.allUsers”, el cual permite la navegación de la lista de usuarios presentes en la plataforma. En la Figura 27 se puede visualizar que éste funciona al utilizar los parámetros de paginación y permite la filtración de los usuarios a buscar utilizando una lista de etiquetas o un texto arbitrario que es usado para buscar su presencia parcial dentro de los campos “code” y “label”, además de su presencia en el campo de etiquetas (“tags”).

```
AdminUserMutations: {
  async upsertUsersWithProjects(
    _root,
    { emails, projectsIds },
    { prisma }
  ) {
    return pMap(
      emails,
      (email) => {
        return prisma.user.upsert({
          create: {
            email,
            projects: {
              connect: projectsIds.map((id) => ({ id })),
            },
          },
          where: {
            email,
          },
          update: {
            projects: {
              connect: projectsIds.map((id) => ({ id })),
            },
          },
        });
      },
      {
        concurrency: 4,
      }
    );
  },
  updateUser(
    _root,
    { data: { id, projectIds, tags, ...data } },
    { prisma }
  ) {
    return prisma.user.update({
      where: {
        id,
      },
      data: {
        ...data,
        projects: {
          set: projectIds.map((id) => ({ id })),
        },
        tags: {
          set: tags,
        },
      },
    });
  },
},
```

Figura 26. Resolutores de “AdminUserMutations” parte de módulo de usuarios

```

AdminUserQueries: {
  allUsers(_root, { pagination, filters }, { prisma }) {
    return ResolveCursorConnection(pagination, (connection) => {
      return prisma.user.findMany({
        ...connection,
        where: filters
      ? {
          tags: filters.tags
          ? {
              hasSome: filters.tags,
            }
          : undefined,
        OR: filters.textSearch
        ? [
          {
            email: {
              contains: filters.textSearch,
            },
            {
              name: {
                contains: filters.textSearch,
              },
            },
            {
              tags: {
                has: filters.textSearch,
              },
            },
          ],
          : undefined,
        }
        : undefined,
      });
    });
  },
},
}

```

Figura 27. Resolutor de “AdminUserQueries.allUsers” parte de módulo de usuarios

6.3.2.2. Módulo de grupos

El módulo de grupos empieza definiendo la entidad “Group”, la cual se puede visualizar en la Figura 19. Ésta sólo utiliza el modelo de base de datos “Group” y define sus relaciones con la entidad “User”.

Las operaciones de tipo mutación para la administración de los grupos, como se puede ver en la Figura 28, contienen las operaciones “setUserGroups”, la cual asigna usuarios a un grupo especificado; y las operación “createGroup” y “updateGroup”, las cuales permiten la creación y actualización de entidades.

La operación “setUserGroups” recibe como parámetros una lista de emails y una lista de identificadores de grupos. La lista de emails es validada para verificar la existencia de los usuarios previo a la actualización de los grupos, para luego utilizar la función auxiliar “pMap” que ejecuta las actualizaciones de los grupos en lotes de máximo 4 de forma simultánea. Mientras que las operaciones “createGroup” y “updateGroup” son definidas de forma estándar, recibiendo los campos esperados de la entidad.

Finalmente el módulo de grupos define la operación “groups” que es parte de la estandarización necesaria para la unión de los esquemas de los distintos servicios, la cual se puede ver en la Figura 29. Acepta una lista de identificadores y hace uso de una regla de autorización que verifica que el usuario autenticado solo puede leer

de grupos que sean parte de sus proyectos, o sí el usuario tiene el rol de administrador, ignora cualquier restricción.

```

AdminUserMutations: {
  async setUserGroups(_root, { usersEmails, groupIds }, { prisma }) {
    const usersEmailsSet = await prisma.user.findMany({
      where: {
        email: {
          in: usersEmails,
        },
      },
      select: {
        email: true,
      },
    });
    const foundsUsers = keyBy(usersEmailsSet, (v) => v.email);
    const notFoundEmails = usersEmails.filter((v) => {
      return !foundsUsers[v];
    });
    if (notFoundEmails.length) {
      throw Error(`Users Not Found: ${notFoundEmails.join()}`);
    }

    return pMap(
      groupIds,
      (id) => {
        return prisma.group.update({
          where: {
            id,
          },
          data: {
            users: {
              set: usersEmailsSet,
            },
          },
        });
      },
      {
        concurrency: 4,
      }
    );
  },
  async createGroup(
    _root,
    { data: { code, label, projectIds, tags, flags } },
    { prisma }
  ) {
    return prisma.group.create({
      data: {
        code,
        label,
        projects: {
          connect: projectIds.map((id) => ({ id })),
        },
        tags: {
          set: tags,
        },
        flags: {
          create: flags || [],
        },
      },
    });
  },
  async updateGroup(
    _root,
    { data: { id, code, label, projectIds, tags, flags } },
    { prisma }
  ) {
    return prisma.group.update({
      where: {
        id,
      },
      data: {
        code,
        label,
        projects: {
          set: projectIds.map((id) => ({ id })),
        },
        tags: {
          set: tags,
        },
        flags: flags
        ? {
            upsert: {
              create: flags,
              update: flags,
            },
          }
        : undefined,
      },
    });
  },
},

```

Figura 28. Resolutores de “AdminUserMutations” parte de módulo de grupos

```

Query: {
  async groups(_root, { ids }, { prisma, authorization }) {
    return getNodeIdList(
      prisma.group.findMany({
        where: {
          id: {
            in: ids,
          },
          projects: await authorization.expectSomeProjectsInPrismaFilter,
        },
      }),
      ids
    );
  },
},

```

Figura 29. Resolutor de “Query.groups”

6.3.2.3. Módulo de proyectos

El módulo de proyectos define parte de las relaciones entre las entidades de usuario y grupos con los proyectos de forma independiente del servicio de proyectos.

El esquema definido por este módulo se puede ver en la Figura 30. Los tipos “User” y “Group” son extendidos para contener una lista de los identificadores de los proyectos asociados, además de una mutación administrativa que permite fijar los proyectos para un grupo de usuarios.

```

extend type User {
  "IDs of projects associated with the user"
  projectsIds: [IntID!]!
}

extend type Group {
  "IDs of projects associated with the group"
  projectsIds: [IntID!]!
}

extend type AdminUserMutations {
  "Set the projects of the specified users"
  setProjectsToUsers(projectIds: [IntID!]!, userIds: [IntID!]!): [User!]!
}

```

Figura 30. Esquema módulo de proyectos

Los resolutores de este módulo, como se pueden ver en la Figura 31, hacen un uso extensivo de la herramienta “Prisma Client” para la lógica necesaria, como lo es con los resolutores de los campos “projectsIds” que seleccionan los “ids” de los proyectos asociados con la entidad y mapean estos identificadores a una lista plana. La lógica utilizada la mutación “setProjectsToUsers” simplemente hace una transacción actualizando los usuarios con los proyectos especificados.

```

AdminUserMutations: {
  setProjectsToUsers(_root, { projectIds, userIds }, { prisma }) {
    const projectsIdsDataSet: PrismaNS.UserUpdateInput = {
      projects: {
        set: projectIds.map((projectId) => {
          return {
            id: projectId,
          };
        }),
      },
    };
    return prisma.$transaction(
      userIds.map((id) => {
        return prisma.user.update({
          where: {
            id,
          },
          data: projectsIdsDataSet,
        });
      })
    );
  },
  User: {
    async projectsIds({ id }, _args, { prisma }) {
      return (
        (
          await prisma.user
            .findOne({
              where: {
                id,
              },
            })
            .projects({
              select: {
                id: true,
              },
            })
            )?.map((v) => v.id) || []
        );
      },
    },
    Group: {
      async projectsIds({ id }, _args, { prisma }) {
        return (
          (
            await prisma.group
              .findOne({
                where: {
                  id,
                },
              })
              .projects({
                select: {
                  id: true,
                },
              })
              )?.map((v) => v.id) || []
          );
        },
      },
    },
  },
}

```

Figura 31. Resolutores módulo de proyectos

6.4. Gateway

El “Gateway” o puerta de entrada de la plataforma tiene especial relevancia para el funcionamiento del sistema. Debido a la naturaleza de la arquitectura de microservicios, la utilización de muchos servicios de forma simultánea por parte de las aplicaciones es muy incómoda, ya que significa que el cliente es el responsable de coordinar y unir la información de los distintos servicios que la plataforma ofrece.

Este “Gateway” busca unir los esquemas de todos los servicios de la plataforma y orquestar operaciones que signifiquen el uso de 1 o más servicios de forma simultánea, lo que permite a los clientes preparar operaciones que contemplan una cantidad considerable de datos, involucrando múltiples servicios de forma simultánea.

Como ha sido previamente mencionado, el “Gateway” funciona al combinar el funcionamiento de distintas librerías, como “GraphQL Tools Stitching” y “Undici” (Node.js Undici, s.f) para la comunicación con los servicios, en conjunto con la misma combinación de librerías utilizada en los servicios: “Fastify” y “GraphQL EZ” para la API.

En la Figura 32 se puede visualizar de forma ejemplificada como el “Gateway” crea un esquema “Proxy” en memoria que se obtiene al combinar distintos sub-esquemas parte de cada uno de los servicios de la plataforma, lo que permite al cliente visualizar un único esquema.

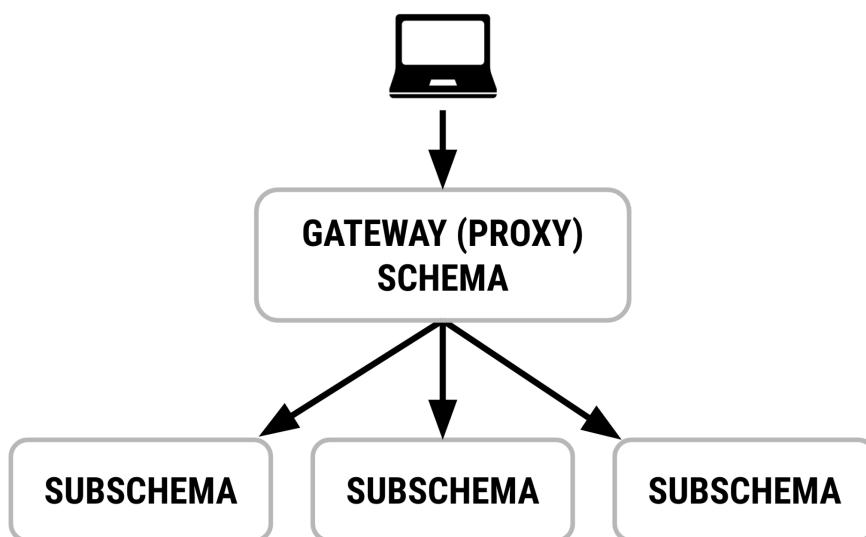


Figura 32. Esquema “Proxy” GraphQL distribuido¹³

La Figura 33 muestra un extracto de código presente en “/packages/gateway/src/services.ts” y “/packages/services/list.ts” dentro del repositorio que es utilizado para la lógica del “Gateway” con respecto a la conexión con los servicios. En el archivo “list.ts” se encuentran listados los servicios de la

¹³ Figura obtenida de *Combining schemas - GraphQL Tools*, accedido el 25 de enero, 2022, <https://www.graphql-tools.com/docs/schema-stitching/stitch-combining-schemas>

plataforma junto a los puertos por defecto de cada servicio. Si el “Gateway” no encuentra variables de ambiente conteniendo los enlaces apuntando a cada servicio, este procede a buscar los servicios a “localhost” con los puertos por defecto respectivos.

```
TS services.ts ×
packages > gateway > src > TS services.ts > ...
1 import { baseServicesList, ServiceName } from "api-base";
2 import type { ServiceSchemaConfig } from "./serviceSchema";
3 import { servicesSubschemaConfig } from "./stitchConfig";
4
5 export const getServicesConfigFromEnv = () => {
6   const services = Object.entries(baseServicesList) as Array<
7     [ServiceName, number]
8   >;
9
10 const servicesConfig: ServiceSchemaConfig[] = services.map(([name, port]) => {
11   const href = process.env[` ${name.toUpperCase()} _URL`];
12
13   return {
14     config: servicesSubschemaConfig,
15     name,
16     port,
17     href,
18   };
19 });
20
21 return servicesConfig;
22 };

TS list.ts ×
packages > services > TS list.ts > ...
1 export const baseServicesList = {
2   users: 3002,
3   actions: 3003,
4   content: 3004,
5   domain: 3005,
6   projects: 3006,
7   state: 3007,
8 } as const;
9
10 export const servicesNames = Object.keys(baseServicesList) as ServiceName[];
11
12 export type ServiceName = keyof typeof baseServicesList;
```

Figura 33. Configuración de conexión con servicios

La conexión del “Gateway” con cada servicio se realiza a través de la librería “undici”, la cual realiza conexiones con “pipelining”, lo que logra obtener grandes rendimientos.

Este proceso se puede visualizar en la Figura 34, la cual realiza una introspección del esquema del servicio objetivo y retorna el ejecutor del servicio junto con configuraciones relevantes para el proceso de unión de los esquemas de todos los servicios.

La configuración utilizada para la unión (“Stitching”) de los esquemas se puede visualizar en la Figura 35, la cual describe configuraciones estándar para las entidades que son definidas y/o extendidas por los servicios de la plataforma.

```

export async function getServiceSchema({
  name,
  href,
  port,
  config = servicesSubschemaConfig,
}: ServiceSchemaConfig) {
  const serviceUrl = new URL(href || `http://127.0.0.1:${port}/graphql`);

  const pathname = serviceUrl.pathname;

  const client = (ServicesClients[serviceUrl.origin] ||= new Client(
    serviceUrl.origin,
    {
      pipelining: 10,
    }
  ));

  const wsExecutor = getWsExecutor(serviceUrl);

  const remoteExecutor: AsyncExecutor<Partial<EZContext>> =
    async function remoteExecutor(args) {
      const { document, variables, context, operationName, operationType } =
        args;

      if (operationType === OperationTypeNode.SUBSCRIPTION)
        return wsExecutor(args);

      let query = DocumentPrintCache.get(document);

      if (query === null) {
        query = print(document);
        DocumentPrintCache.set(document, query);
      }

      const authorization = context?.request?.headers.authorization;

      const headers: IncomingHttpHeaders = {
        "Content-Type": "application/json",
        authorization,
      };

      if (IS_TEST) {...}

      const { body } = await client.request({
        path: pathname,
        body: JSON.stringify({ query, variables, operationName }),
        method: "POST",
        headers,
      });

      try {
        return await body.json();
      } catch (err) {...}
    };
}

const serviceSubschema: SubschemaConfig = {
  schema: await introspectSchema(remoteExecutor),
  executor: remoteExecutor,
  batch: true,
  ...config[name],
};

return serviceSubschema;
}

```

Figura 34. Conexión de “Gateway” con servicios

```

export const ProjectMerge: MergeConfig = {
  fieldName: "projects",
  selectionSet: "{ id }",
  key: ({ id }: Node) => id,
  argsFromKeys: (ids) => ({ ids }),
};

export const DomainMerge: MergeConfig = {
  fieldName: "domains",
  selectionSet: "{ id }",
  key: ({ id }: Node) => id,
  argsFromKeys: (ids) => ({ ids }),
};

export const TopicMerge: MergeConfig = {
  fieldName: "topics",
  selectionSet: "{ id }",
  key: ({ id }: Node) => id,
  argsFromKeys: (ids) => ({ ids }),
};

export const ContentMerge: MergeConfig = {
  fieldName: "content",
  selectionSet: "{ id }",
  key: ({ id }: Node) => id,
  argsFromKeys: (ids) => ({ ids }),
};

export const UsersMerge: MergeConfig = {
  fieldName: "users",
  selectionSet: "{ id }",
  key: ({ id }: Node) => id,
  argsFromKeys: (ids) => ({ ids }),
};

export const GroupsMerge: MergeConfig = {
  fieldName: "groups",
  selectionSet: "{ id }",
  key: ({ id }: Node) => id,
  argsFromKeys: (ids) => ({ ids }),
};

export const KCMerge: MergeConfig = {
  fieldName: "kcs",
  selectionSet: "{ id }",
  key: ({ id }: Node) => id,
  argsFromKeys: (ids) => ({ ids }),
};

const defaultMergeConfig = {
  Project: ProjectMerge,
  Domain: DomainMerge,
  Topic: TopicMerge,
  Content: ContentMerge,
  User: UsersMerge,
  Group: GroupsMerge,
  KC: KCMerge,
};

export const servicesSubschemaConfig = servicesNames.reduce(
  (acum, serviceName) => {
    acum[serviceName] = {
      batch: true,
      merge: {
        ...defaultMergeConfig,
      },
    };
    return acum;
  },
  {} as {
    [k in ServiceName]: Omit<SubschemaConfig, "schema">;
  }
);

```

Figura 35. Configuración de “Stitching” para “Gateway”

6.5. Pruebas de carga

6.5.1. Preparación

Con el objetivo de verificar el funcionamiento correcto de la plataforma a mediano y largo plazo, se realizaron pruebas de carga con datos generados de forma aleatoria, programada a medida del formato esperado de los datos.

La creación de datos empieza por creando una nueva base de datos temporal dentro de la instancia local de “PostgreSQL”. La inserción de datos busca crear cantidades altas pero que tengan sentido en términos de proporciones esperadas entre las entidades.

Este proceso crea 20 proyectos; 10.000 usuarios con entre 1 y 2 proyectos asociados directamente por usuario, 500 grupos con entre 0 y 100 usuarios cada uno, y entre 0 y 2 proyectos por grupo; 50 dominios con entre 1 y 7 proyectos asociados; 1500 tópicos con un proyecto asociado aleatorio entre todos los disponibles, en la cual la jerarquía entre los tópicos fue determinada de forma aleatoria dentro de tópicos del mismo proyecto; 500 “KCs” asociados con dominios aleatoriamente; y finalmente se crean 1500 entidades de contenido, los cuales por su complejidad y cantidad son creados de forma paralela usando todos los núcleos físicos del ordenador huésped.

Cada contenido obtiene un proyecto elegido aleatoriamente, en la cual se le asocia entre 1 y 4 “KCs”, 1 tópico, entre 0 y 3 etiquetas, y el dato central del contenido, que es determinado de forma probabilística, en la cual con un 30% de probabilidad solo se le asocia un enlace externo, 40% de probabilidad de contenido JSON aleatorio, y el restante 30% de probabilidad, se crea contenido binario, en la cual se genera un archivo de texto con entre 1.800 y 600.000 palabras que es convertido a un formato binario.

Una vez que el contenido se encuentra creado en la base de datos, se ejecutan en segundo plano instancias clusterizadas (basadas en la cantidad de núcleos del procesador huésped) de la plataforma, una instancia del panel de administración, y ejecuta de forma continua y en segundo plano hilos de ejecución basados en lotes de 100 usuarios, en lo cual cada usuario inicia el proceso solicitando una cantidad de información normal relacionada a sí mismo y del proyecto al que se le realizará la simulación de creación de acciones. Esta solicitud se puede visualizar en la Figura 36, en donde el usuario simulado solicita información al respecto de sí mismo e información relevante de su proyecto elegido aleatoriamente, incluyendo datos

como tópicos, contenidos y sus “KCs” disponibles. Estos datos luego son utilizados para el proceso del envío de acciones.

Una vez que el usuario simulado tiene en su posesión datos relevantes para el envío de acciones, se generan datos de forma probabilística, proceso que se puede visualizar en la Figura 37, en la cual se envía a la plataforma el proyecto en la cual se simula su uso: se elige un “verbo” de la acción de forma aleatoria, se elige entre 0 y 3 “KCs”, con un 20% de probabilidad se añade un contenido de forma aleatoria, con 20% de probabilidad se añade un tópico de forma aleatoria, y con distintas otras probabilidades se añaden distintos posibles campos de datos aceptados dentro de la acción.

```
const { currentUser, project } = await assertedQuery(
  gql(`/* GraphQL */
    query LoadTestCurrentUser($projectId: Int!) {
      currentUser {
        id
        email
      }
      project(id: $projectId) {
        id
        code
        label
        topics {
          id
          code
          label
        }
        content(pagination: { first: 50 }) {
          nodes {
            id
            code
            label
            tags
            kcs {
              id
              code
              label
            }
          }
        }
      }
    },
    {
      variables: {
        projectId: chosenProject.toString(),
      },
    }
);
```

Figura 36. Solicitud inicial de usuario en prueba de carga

```

const projectId = project.id.toString();

const topics = project.topics;
const content = project.content.nodes;
const kcs = uniqBy(
  project.content.nodes.flatMap((v) => v.kcs),
  (v) => v.id
);

async function MeasuredAction() {
  const start = performance.now();
  await assertedQuery(
    gql(`/* GraphQL */
      mutation LoadTestAction($data: ActionInput!) {
        action(data: $data)
      }
    `),
    {
      variables: {
        data: {
          projectId,
          timestamp: Date.now(),
          verbName: sample(verbNames)!,
          amount: probability(33) ? random(0, 100) : null,
          contentID: probability(20) ? sample(content)?.id : null,
          kcsIDs: sampleSize(kcs, random(0, 3)).map((v) => v.id),
          topicID: probability(20) ? sample(topics)?.id : null,
          detail: probability(50)
            ? generate({
                charset: "alphanumeric",
              })
            : null,
          extra: probability(40)
            ? JSON.parse(faker.datatype.json())
            : undefined,
          stepID: probability(20) ? random(0, 20).toString() : null,
          hintID: probability(20) ? random(0, 40).toString() : null,
          result: probability(15) ? random(0, 1, true) : null,
        },
      },
    }
  );
}

const end = performance.now();

return {
  duration: end - start,
};
}

```

Figura 37. Envío de acción por usuario en prueba de carga

6.5.2. Resultados

La ejecución de las pruebas fueron realizadas en un ordenador huésped con el procesador “AMD Ryzen 5900x” con 12 núcleos y 24 hilos, con clusterización de los servicios para el uso óptimo del procesador, por lo cual se puede considerar casi como un “best-case scenario”, y permite ver de lo que la plataforma es capaz en un despliegue monolítico con una cantidad baja de “overhead” en comunicación por red.

En la Figura 38 se puede visualizar los resultados en terminal obtenidos al ejecutar las pruebas de carga, en la cual este inicia añadiendo los datos necesarios dentro de la base de datos, y finaliza con la ejecución en segundo plano de inserción de acciones por usuarios simulados de la plataforma. En la Figura 39 se puede ver la utilización del procesador a lo largo de 1 minuto de uso de la plataforma a través de la simulación de acciones.

De la ejecución en segundo plano se pudieron obtener resultados que resultan ser buenos considerando la alta carga que se le da a éste, tanto en términos de cantidad de datos acumulativos en la base de datos, como de la tasa de datos en transferencia entre la API y “las aplicaciones”.

En términos de tiempos obtenidos, la inserción de 1000 acciones por parte de lotes de 100 usuarios toma entre 4 y 5 segundos, en donde se obtiene que el tiempo promedio de solicitud de la información de los usuarios resulta con una latencia promedio de 350 milisegundos y una mediana de 250 milisegundos por solicitud, y el proceso por lote resulta en una tasa de promedio de 200 acciones insertadas en la plataforma por segundo, con un promedio de latencia de 160 milisegundos por acción y una mediana de latencia por acción de 105 milisegundos.

Parte importante del uso de la plataforma incluye la administración de ésta, para lo cual se desarrolló un panel de administración que hace uso de las solicitudes administrativas definidas por los distintos servicios. Por la naturaleza y alcance del proyecto, no se pudo realizar una cuantificación con números promediados, pero sí se puede ver a través de la Figura 40 que la mayor parte de las solicitudes tienen tiempos de respuesta menores a los 500 milisegundos, y cabe considerar que las acciones en segundo plano se siguen en ejecución mientras se hizo uso del panel de administración para la presente prueba de carga.

```

{
  DATABASE_URL: 'postgresql://postgres:postgres@localhost:5789/test_wAIwhY4mL00aN1tKmnlwkZoV0qcROX',
  GATEWAY_URL: 'http://localhost:8080'
}
Creating projects
20 projects created
Creating users...
10% of users created
20% of users created
30% of users created
40% of users created
50% of users created
60% of users created
70% of users created
80% of users created
90% of users created
100% of users created
10000 users created
Creating groups...
500 groups created
Creating domains...
50 domains created
Creating topics...
1500 topics created
Associating topics...
1500 topics associated
Creating KCs...
500 KCs created
Creating content...
10% of content created
20% of content created
30% of content created
40% of content created
50% of content created
60% of content created
70% of content created
80% of content created
90% of content created
100% of content created
1500 content created
> mono@0.0.1 start /home/pablosz/learner-model-gql/packages/mono
> bob-tsm --keep-esm-loader src/index.ts

$ node --require=bob-tsm --loader=bob-tsm --enable-source-maps src/index.ts
Scope: all 16 workspace projects
..../client-admin dev:localhost$ concurrently -r "cross-env NEXT_PUBLIC_API_URL=http://localhost:8080/graphql next dev -p 4010" "pnpm -r graph:dev"
..../client-admin dev:localhost: ready - started server on 0.0.0.0:4010, url: http://localhost:4010
..../client-admin dev:localhost: info - Loaded env from /home/pablosz/learner-model-gql/packages/client-admin/.env.local
..../client-admin dev:localhost: warn - You have enabled experimental feature(s).
..../client-admin dev:localhost: warn - Some experimental features are not covered by semver, and may cause unexpected or broken application behavior. Use them at your own risk.
..../client-admin dev:localhost: warn - See https://github.com/nextjs/next/pull/10336 for more information.
..../client-admin dev:localhost: Scope: all 16 workspace projects
..../client-admin dev:localhost: ..../graph graph:dev$ graphql-codegen --config codegen.yml --watch
..../client-admin dev:localhost: [graph graph:dev]: (node:219177) ExperimentalWarning: stream/web is an experimental feature. This feature could change at any time
..../client-admin dev:localhost: (Use 'node --trace-warnings ...' to show where the warning was created)
..../client-admin dev:localhost: ..../graph graph:dev: [22:46:12] Parse configuration [started]
..../client-admin dev:localhost: ..../graph graph:dev: [22:46:12] Parse configuration [completed]
..../client-admin dev:localhost: ..../graph graph:dev: [22:46:12] Generate outputs [started]
..../client-admin dev:localhost: ..../graph graph:dev: [22:46:12] Load GraphQL schemas [started]
..../client-admin dev:localhost: ..../graph graph:dev: [22:46:12] Load GraphQL documents [completed]
..../client-admin dev:localhost: ..../graph graph:dev: [22:46:12] Load GraphQL documents [started]
..../client-admin dev:localhost: ..../graph graph:dev: [22:46:12] Load GraphQL documents [completed]
..../client-admin dev:localhost: ..../graph graph:dev: [22:46:12] Generate [started]
..../client-admin dev:localhost: ..../graph graph:dev: [22:46:13] Generate [completed]
..../client-admin dev:localhost: ..../graph graph:dev: [22:46:13] Generate to src/rq-gql/ (using EXPERIMENTAL preset "rq-gql/preset") [completed]
..../client-admin dev:localhost: ..../graph graph:dev: [22:46:13] Generate outputs [completed]
---Gateway ready---
---Action creation started---
..../client-admin dev:localhost: ..../graph graph:dev: i Watching for changes...
..../client-admin dev:localhost: event - compiled client and server successfully in 2.9s (772 modules)
{
  averageActionLatency: '126.15874442698248 ms',
  medianActionLatency: '117.84789500012994 ms',
  resultsTime: '5 seconds',
  resultsAmount: 1000,
  actionsPerSecond: 203.6381426902187,
  averageUserInfoLatency: '291.55184040997176 ms',
  medianUserInfoLatency: '281.275907005268 ms'
}
{
  averageActionLatency: '119.60963199800997 ms',
  medianActionLatency: '104.0918039996177 ms',
  resultsTime: '4 seconds',
  resultsAmount: 1000,
  actionsPerSecond: 263.75577438271864,
  averageUserInfoLatency: '244.95465592011809 ms',
  medianUserInfoLatency: '240.23758400045335 ms'
}

```

Figura 38. Ejemplo de ejecución desde terminal de prueba de carga



Figura 39. Uso de CPU en ejecución de prueba de carga¹⁴

¹⁴ Captura de programa “Gnome System Monitor” (System Monitor, GNOME Wiki, accedido el 25 de enero, 2022, desde <https://wiki.gnome.org/Apps/SystemMonitor>)

Query / Mutation	Status	Size	Time	URL
currentUser	200	211 B	12ms	http://localhost:8080/graphql
AllDomainsBase	200	3.28 kB	23ms	http://localhost:8080/graphql
AllIKCs	200	4.38 kB	348ms	http://localhost:8080/graphql
AllDomainsBase	200	3.28 kB	14ms	http://localhost:8080/graphql
AllProjects	200	13.3 kB	176ms	http://localhost:8080/graphql
AllContentBase	200	6.11 kB	14ms	http://localhost:8080/graphql
AllProjectsBase	200	1.4 kB	14ms	http://localhost:8080/graphql
AllVerbNames	200	1.99 kB	13ms	http://localhost:8080/graphql
AllIKCsBase	200	3.28 kB	151ms	http://localhost:8080/graphql
AllUsersBase	200	4.08 kB	150ms	http://localhost:8080/graphql
AllActions	200	25.6 kB	163ms	http://localhost:8080/graphql
AllContentBase	200	6.03 kB	217ms	http://localhost:8080/graphql
AllIKCsBase	200	3.29 kB	67ms	http://localhost:8080/graphql
AllTopicsBase	200	23.1 kB	301ms	http://localhost:8080/graphql
AllTopicsBase	200	24.9 kB	427ms	http://localhost:8080/graphql
AllActions	200	24.7 kB	239ms	http://localhost:8080/graphql
AllActions	200	26.1 kB	1.1s	http://localhost:8080/graphql
AllActions	200	24.6 kB	198ms	http://localhost:8080/graphql
AllActions	200	24.5 kB	257ms	http://localhost:8080/graphql
AllActions	200	25.8 kB	230ms	http://localhost:8080/graphql
AllProjectsBase	200	1.4 kB	354ms	http://localhost:8080/graphql
AdminUsers	200	8.21 kB	403ms	http://localhost:8080/graphql
AllContentBase	200	146 B	12ms	http://localhost:8080/graphql
AllTopicsBase	200	144 B	517ms	http://localhost:8080/graphql
AllTopicsBase	200	24.8 kB	282ms	http://localhost:8080/graphql
AllTopicsBase	200	13.6 kB	1.3s	http://localhost:8080/graphql
AllContentBase	200	6.27 kB	276ms	http://localhost:8080/graphql
AllContentBase	200	4.57 kB	619ms	http://localhost:8080/graphql
AllDomains	200	3.39 kB	10ms	http://localhost:8080/graphql
AllProjectsBase	200	1.4 kB	127ms	http://localhost:8080/graphql
AllGroups	200	195 kB	267ms	http://localhost:8080/graphql

Figura 40. Tiempos de respuesta para panel de administración¹⁵

¹⁵ Captura de programa “graphql-network-inspector”, accedido el 25 de enero, 2022, desde <https://github.com/warrenday/graphql-network-inspector>

7. DESPLIEGUE

La arquitectura de microservicios utilizada y planteada por el presente proyecto saca su mayor provecho del gran crecimiento de la industria “en la nube”, como lo son los servicios de Amazon AWS (Amazon AWS, s.f), Microsoft Azure (Microsoft Azure, s.f) y Google Cloud (Google Cloud, s.f). Pero ésta no es la única forma posible de despliegue, ya que existe la posibilidad de desplegar la plataforma de forma simplificada en una máquina única o también de forma mixta. En las siguientes secciones se plantean los diferentes escenarios de despliegue de la plataforma.

7.1. Despliegue ideal con computación en la nube

El despliegue a través de este tipo de servicios permite un escalamiento horizontal virtualmente “infinito” de la capacidad de procesamiento, en la cual cada servicio podría constituir un despliegue autónomo, más un balanceador de carga que es puesto en frente de todas las instancias del servicio. Un ejemplo de despliegue usando este método se puede visualizar en la Figura 41. En éste el “Gateway” en sí mismo es desplegado con escalamiento horizontal, y cada instancia de éste se comunica con los servicios pasando por los平衡adores de carga respectivos.

Este estilo de despliegue permite un escalamiento con límites muy altos, pero resulta prohibitivamente caro en recursos computacionales, aún en su versión mínima posible. Esto resulta muy inconveniente para el contexto del presente proyecto, por lo cual a corto y mediano plazo no es la infraestructura que será utilizada dentro de este trabajo.

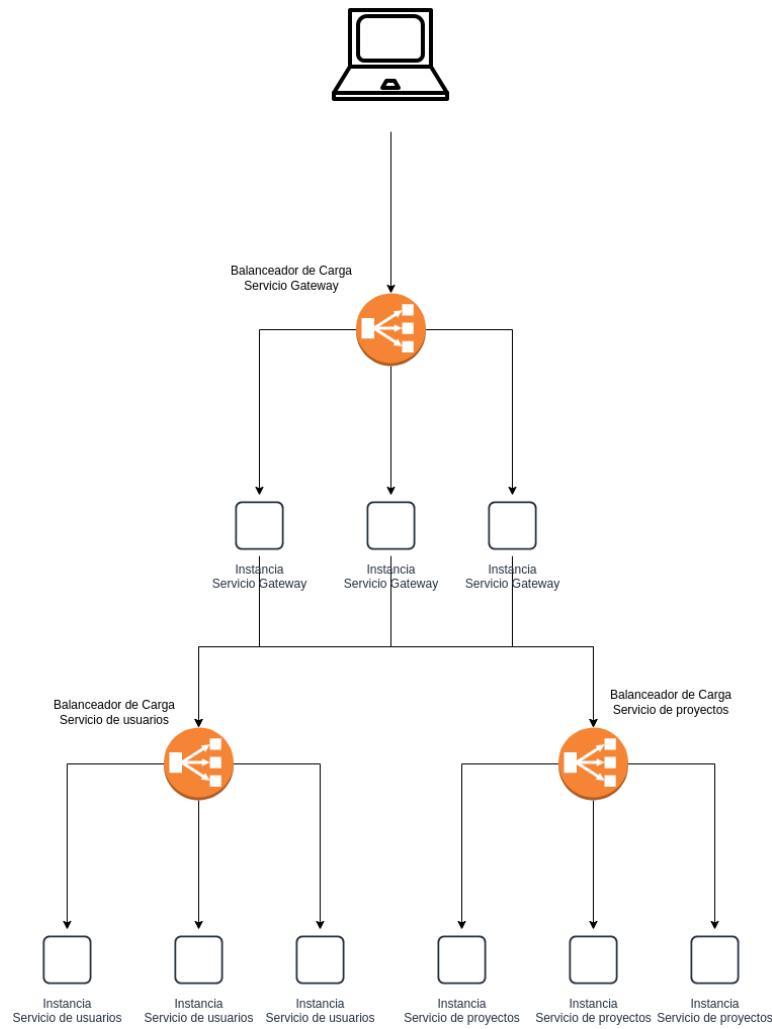


Figura 41. Despliegue ideal en nube de plataforma parcial¹⁶

7.2. Despliegue en servidor único

El despliegue en servidor único depende de la escalabilidad vertical del computador huésped. Una opción es que cada servicio reserve un puerto local determinado, y el “Gateway” se comunique a través de un “networking” local con cada servicio.

Otra opción de despliegue en un servidor único es la forma “monolítica”, en la cual la lógica del esquema de cada servicio es guardada en memoria y el “Gateway” utiliza cada esquema de forma parcial sin necesidad de “networking” para su utilización. Para estos tipos de despliegues las herramientas recomendadas son

¹⁶ Diagrama construido utilizando diagrams.net

Docker (Docker, s.f) y Docker Compose (Docker Compose, s.f), las cuales facilitan este proceso considerablemente.

7.2.1. Despliegue con intercomunicación local

En este tipo de despliegue el ambiente realiza las comunicaciones del “Gateway” con los microservicios a través de un “networking” local. Una configuración recomendada es la expuesta en la Figura 42, en donde cada servicio se ejecuta de forma independiente utilizando la misma imagen Docker, y las variables de ambiente son especificadas heredando de un archivo “.env” adyacente a la configuración principal, tal como se puede ver en la Figura 43. La sintaxis para el uso de variables de ambiente provenientes del archivo “.env” es “\${VARIABLE_DE_AMBIENTE}” en la configuración principal.

Una de las características principales de las imágenes Docker es la ejecución de un comando específico, el cual puede o no existir por defecto. En la imagen construída para el despliegue del presente proyecto, la cual es compartida por todos los servicios, deja el comando de ejecución sin definir, por lo cual, la lista de comandos recomendados para esta consiste en:

- Servicio “Gateway”: “pnpm -r start --filter=service-gateway”
- Servicio “state”: “pnpm -r start --filter=service-state”
- Servicio “projects”: “pnpm -r start --filter=service-projects”
- Servicio “domain”: “pnpm -r start --filter=service-domain”
- Servicio “content”: “pnpm -r start --filter=service-content”
- Servicio “actions”: “pnpm -r start --filter=service-actions”
- Servicio “users”: “pnpm -r start --filter=service-users”
- Aplicación de “migraciones” en base de datos: “pnpm -r migrate:deploy”

Todos los servicios esperan ciertas variables de ambiente:

- “DATABASE_URL”: Enlace obligatorio de conexión con base de datos
- “AUTH0_DOMAIN”, “AUTH0_CLIENT” y “AUTH0_SECRET”: Credenciales obligatorias con servicio externo “Auth0”
- “PORT”: Variable de ambiente opcional que define el puerto que el servicio objetivo utilizará, si este no es especificado recurre a puertos por defecto definidos en la Figura 44. Para el caso del “Gateway”, usa el puerto “8080” por defecto.
- “ADMIN_USER_EMAIL”: Variable de ambiente opcional que los servicios pueden utilizar para automáticamente crear un usuario administrador con la dirección especificada.

```

version: "3.7"
services:
  users:
    restart: always
    image: pabloszx/learner-model-gql
    network_mode: host
    environment:
      DATABASE_URL: ${DATABASE_URL}
      AUTH0_DOMAIN: ${AUTH0_DOMAIN}
      AUTH0_CLIENT: ${AUTH0_CLIENT}
      AUTH0_SECRET: ${AUTH0_SECRET}
    command: pnpm -r start --filter=service-users
  actions:
    restart: always
    image: pabloszx/learner-model-gql
    network_mode: host
    environment:
      DATABASE_URL: ${DATABASE_URL}
      AUTH0_DOMAIN: ${AUTH0_DOMAIN}
      AUTH0_CLIENT: ${AUTH0_CLIENT}
      AUTH0_SECRET: ${AUTH0_SECRET}
    command: pnpm -r start --filter=service-actions
  content:
    restart: always
    image: pabloszx/learner-model-gql
    network_mode: host
    environment:
      DATABASE_URL: ${DATABASE_URL}
      AUTH0_DOMAIN: ${AUTH0_DOMAIN}
      AUTH0_CLIENT: ${AUTH0_CLIENT}
      AUTH0_SECRET: ${AUTH0_SECRET}
    command: pnpm -r start --filter=service-content
  domain:
    restart: always
    image: pabloszx/learner-model-gql
    network_mode: host
    environment:
      DATABASE_URL: ${DATABASE_URL}
      AUTH0_DOMAIN: ${AUTH0_DOMAIN}
      AUTH0_CLIENT: ${AUTH0_CLIENT}
      AUTH0_SECRET: ${AUTH0_SECRET}
    command: pnpm -r start --filter=service-domain
  projects:
    restart: always
    image: pabloszx/learner-model-gql
    network_mode: host
    environment:
      DATABASE_URL: ${DATABASE_URL}
      AUTH0_DOMAIN: ${AUTH0_DOMAIN}
      AUTH0_CLIENT: ${AUTH0_CLIENT}
      AUTH0_SECRET: ${AUTH0_SECRET}
    command: pnpm -r start --filter=service-projects
  state:
    restart: always
    image: pabloszx/learner-model-gql
    network_mode: host
    environment:
      DATABASE_URL: ${DATABASE_URL}
      AUTH0_DOMAIN: ${AUTH0_DOMAIN}
      AUTH0_CLIENT: ${AUTH0_CLIENT}
      AUTH0_SECRET: ${AUTH0_SECRET}
    command: pnpm -r start --filter=service-state
  gateway:
    restart: always
    image: pabloszx/learner-model-gql
    network_mode: host
    environment:
      DATABASE_URL: ${DATABASE_URL}
      AUTH0_DOMAIN: ${AUTH0_DOMAIN}
      AUTH0_CLIENT: ${AUTH0_CLIENT}
      AUTH0_SECRET: ${AUTH0_SECRET}
      ADMIN_USER_EMAIL: ${ADMIN_USER_EMAIL}
    command: pnpm -r start --filter=service-gateway
  migration:
    restart: "no"
    image: pabloszx/learner-model-gql
    network_mode: host
    environment:
      DATABASE_URL: ${DATABASE_URL}
    command: pnpm -r migrate:deploy

```

Figura 42. Configuración recomendada para Docker Compose

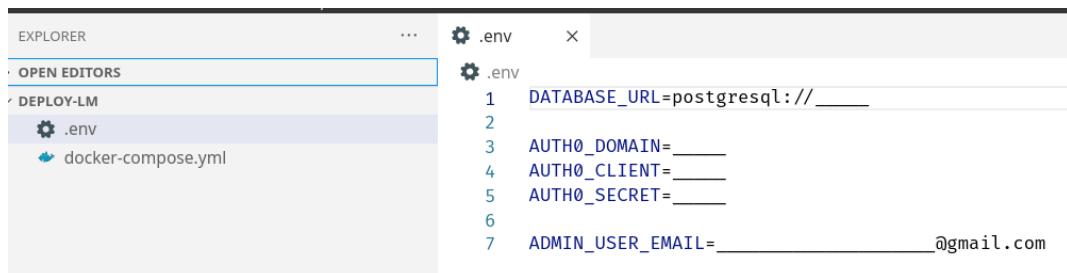


Figura 43. Configuraciones “Docker Compose” y variables de ambiente

```
export const baseServicesList = {
  users: 3002,
  actions: 3003,
  content: 3004,
  domain: 3005,
  projects: 3006,
  state: 3007,
}
```

Figura 44. Lista de servicios con puertos por defecto

Para el despliegue seguro de la plataforma con uso de SSL se recomienda el uso “Caddy Server” (Caddy Server, s.f), el cual realiza la generación automática de certificados y “Reverse proxy” hacia el “Gateway” de la plataforma.

Cabe considerar que el despliegue realizado con un único computador huésped parte de la Universidad Austral de Chile hizo uso de este tipo de despliegue, y se encuentra disponible en el enlace “<https://lm.inf.uach.cl>”, el cual su navegación web por defecto redirige a la representación visual del esquema de la plataforma utilizando la herramienta GraphQL Voyager.

7.2.2. Despliegue monolítico

Una de las mayores ventajas de utilizar el mismo ambiente de programación para todos los servicios y “Gateway” es la posibilidad de evitar el “overhead” que viene con las grandes cantidades de comunicaciones web entre el “Gateway” y los servicios. Esto es posible ya que con un código relativamente reducido se puede hacer una referencia directa a los servicios y realizar el proceso de “Stitching” entre los esquemas de los servicios en memoria.

Este tipo de despliegue es muy similar al despliegue con intercomunicación local, en el cual se recomienda el uso de “Docker Compose”. Para este caso se recomienda una configuración como se puede visualizar en la Figura 45. Esta crea una instancia temporal que verifica que las migraciones con la base de datos estén actualizadas, servicio llamado “migration” en la configuración, y la instancia principal llamada “mono” en la configuración, que ejecuta el conjunto de los servicios con las variables de ambiente esperadas.

```
version: "3.7"
services:
  mono:
    restart: always
    image: pabloszx/learner-model-gql
    network_mode: host
    environment:
      DATABASE_URL: ${DATABASE_URL}
      AUTH0_DOMAIN: ${AUTH0_DOMAIN}
      AUTH0_CLIENT: ${AUTH0_CLIENT}
      AUTH0_SECRET: ${AUTH0_SECRET}
      ADMIN_USER_EMAIL: ${ADMIN_USER_EMAIL}
    command: pnpm -r start --filter=mono
  migration:
    restart: "no"
    image: pabloszx/learner-model-gql
    network_mode: host
    environment:
      DATABASE_URL: ${DATABASE_URL}
    command: pnpm -r migrate:deploy
```

Figura 45. Configuración monolítica recomendada para “Docker Compose”

Este tipo de despliegue al igual que “Gateway” puede recibir la variable de ambiente “PORT” o por defecto recurre al puerto “8080”.

También cabe mencionar que el despliegue monolítico facilita la escalabilidad vertical utilizando funcionalidades como “Node.js Cluster” (Node.js Cluster, s.f). Éste permite que distintas instancias paralelas compartan un puerto y automáticamente se distribuye la carga entre las instancias.

8. PLANTILLA DE DESARROLLO

El objetivo principal de la plantilla de desarrollo es facilitar el uso de la plataforma, haciendo uso de librerías y un ambiente de desarrollo recomendado, creando convenciones alrededor de los servicios provistos por la plataforma.

Esta plantilla se encuentra disponible públicamente en el enlace “<https://github.com/PabloSzx/learner-model-gql-template>”, y su despliegue se encuentra disponible en la dirección “<https://template.learner-model-gql.pablosz.dev>” haciendo uso de la autenticación de la plataforma.

8.1. Librerías principales

- Next.js (Next.js, s.f) - Framework completo basado en React.js (React.js, s.f) para el desarrollo de aplicaciones web
- TypeScript - Lenguaje de programación que es un superconjunto de JavaScript con seguridad de tipado en mente. El uso de este es recomendado pero no obligatorio, ya que puede ser usado en conjunto con JavaScript.
- auth0-react - Librería oficial de integración de React con Auth0
- Chakra UI (Chakra UI, s.f) - Componentes de interfaz gráfica simples, modulares y accesibles
- GraphQL Code Generator (GraphQL Code Generator, s.f) - Generación de código con seguridad de tipado y un uso más fácil de servicios GraphQL.
- React Query (React Query, s.f) - Librería para el proceso de solicitud, “caching” y actualización de datos asíncronos en React.
- Valtio (GitHub - pmndrs/valtio, s.f) - Manejo y sincronización de “state” en React

8.2. Código importante

8.2.1. Definición cliente de API

Parte importante de una aplicación que necesita interactuar directamente con un servicio “backend” como en este caso es la plataforma del presente proyecto, es la definición de una porción de código que se encargue de definir el enlace web y ciertos comportamientos por defecto, como es el manejo de errores.

En la Figura 46 se puede visualizar como esta definición es realizada. El código se encuentra disponible en el repositorio en la ruta “/src/rqClient.ts”. Por defecto, la instancia de “React Query” cuando recibe un error por parte de la API, muestra un

mensaje de error tipo “Toast” de “Chakra UI” con el mensaje de error recibido. En la Figura 47 se puede ver como este mensaje es mostrado en la aplicación.

```
import { useToast } from "@chakra-ui/react";
import { memo, useEffect } from "react";
import { QueryClient } from "react-query";
import { RQGQLClient } from "rq-gql";
import { serializeError } from "serialize-error";
import { proxy, useSnapshot } from "valtio";
import { API_URL } from "./utils/constants";

export const queryClient = new QueryClient({
  defaultOptions: {
    mutations: {
      onError(err) {
        if (err instanceof Error) {
          errorState.message = err.message;
        } else {
          errorState.message = JSON.stringify(serializeError(err));
        }
      },
    },
    queries: {
      onError(err) {
        if (err instanceof Error) {
          errorState.message = err.message;
        } else {
          errorState.message = JSON.stringify(serializeError(err));
        }
      },
    },
  },
});

export const rqGQLClient = new RQGQLClient({
  endpoint: API_URL,
});

const errorState = proxy({
  message: null as string | null,
});

export const ErrorToast = memo(() => {
  const { message } = useSnapshot(errorState);

  const toast = useToast();

  useEffect(() => {
    if (!message) return;

    errorState.message = null;

    toast({
      title: message,
      status: "error",
    });
  }, [message]);

  return null;
});
```

Figura 46. Módulo de definición de cliente

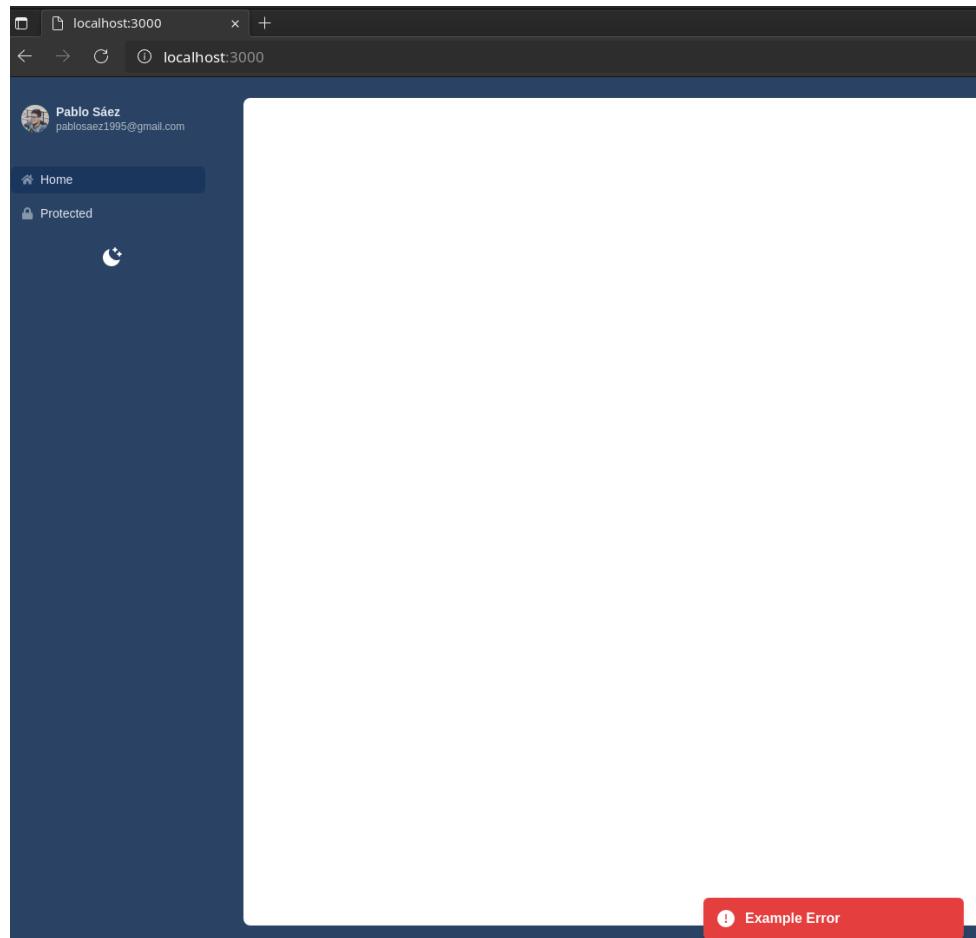


Figura 47. Mensaje error de ejemplo en plantilla de desarrollo

8.2.2. Implementación básica de autenticación

Parte esencial de la comunicación entre las aplicaciones y la plataforma “backend” del proyecto es la lógica necesaria para sincronizar la autenticación realizada con Auth0 con la autenticación del sistema de datos de la plataforma, por ello la plantilla contiene esta lógica “out-of-the-box”, y parte de esta se puede visualizar en la Figura 48, en donde se define el “proxy” de datos que almacena los datos de Auth0 y los datos de autenticación obtenidos a través de la comunicación con el “Gateway”.

La plantilla asume la utilización de la plataforma a través de la representación de un proyecto específico, el cual es definido estáticamente en la Figura 48 como “code: ‘example’”. Este debe cambiarse de acuerdo al proyecto objetivo de la aplicación.

```

export const AuthState = proxy<{
  auth0User: Auth0User | null;
  user: CurrentUserQuery["currentUser"];
  project: CurrentUserQuery["project"];
  isLoading: boolean;
  authorizationToken?: string;
}>({
  auth0User: null,
  user: null,
  project: null,
  isLoading: true,
});

export function SyncAuth() {
  const { user, getIdTokenClaims, isLoading } = useAuth0();
  const { authorization } = useSnapshot(rqGraphQLClient.headers);

  const latestGetIdToken = useLatestRef(getIdTokenClaims);

  const hasAuthorizationToken = !!authorization;

  const { isLoading: currentUserIsLoading } = useGQLQuery(
    gql(`query currentUser {
      currentUser {
        id
        email
        name
        role
        picture
        tags
        projects {
          id
          code
          label
        }
        groups {
          id
          code
          label
          tags
        }
      }
      project(code: "example") {
        id
        code
        label
      }
    `),
    undefined,
    {
      enabled: hasAuthorizationToken,
      onSuccess(data) {
        AuthState.user = data.currentUser;
        AuthState.project = data.project;
      },
      onSettled() {
        AuthState.isLoading = false;
      },
    }
  );
  useEffect(() => {
    AuthState.isLoading = currentUserIsLoading || isLoading;
  }, [isLoading, currentUserIsLoading]);

  useEffect(() => {
    AuthState.auth0User = user || null;
  }, [user]);

  useEffect(() => {
    if (user) {
      AuthState.isLoading = true;
      latestGetIdToken.current().then((data) => {
        AuthState.authorizationToken = rqGraphQLClient.headers.authorization = data
          ? `Bearer ${data.__raw}`
          : undefined;
        AuthState.isLoading = false;
      });
    }
  }, [user, latestGetIdToken]);
}

return <OnStart />;
}

export const useAuth = () => useSnapshot(AuthState);

```

Figura 48. Fragmento de código para sincronización de autenticación

8.2.3. Módulo utilitario para reporte de acciones

Una función importante en la plataforma es permitir el reporte de las acciones acciones de parte de las aplicaciones cliente. Para ello se define la función “useAction”, la cual se puede visualizar en la Figura 49. La función “useAction” utiliza los datos manejados en la Figura 48, al reutilizar el proyecto solicitado allí. Esta función falla inmediatamente si no se encuentra un identificador para el proyecto objetivo, y en caso contrario, se requiere como mínimo especificar el “verbo” de la acción, y esta función se encarga del resto.

```

import { useLatestRef, useToast } from "@chakra-ui/react";
import { useCallback } from "react";
import { useGQLMutation } from "rq-gql";
import { useAuth } from "../components/Auth";
import { ActionInput, gql } from "../graphql";

export type ActionArguments = Omit<ActionInput, "projectId" | "timestamp">;

export const useAction = (baseAction?: Partial<ActionArguments>) => {
  const toast = useToast();

  const latestBaseAction = useLatestRef(baseAction);

  const mutation = useGQLMutation(
    gql(`/* GraphQL */
      mutation Action($data: ActionInput!) {
        action(data: $data)
      }
    `),
    {
      onError(err) {
        console.error(err);
        if (process.env.NODE_ENV === "development") {
          toast({
            status: "error",
            title: "Error while sending Action to API (this message is only seen in Development Mode)",
            description: err.message,
          });
        }
      },
      retry: 3,
    }
  );

  const latestMutation = useLatestRef(mutation.mutate);

  const { project } = useAuth();

  const projectId = project?.id;

  return useCallback(
    (data?: Partial<ActionArguments>) => {
      if (!projectId) throw Error("Invalid projectId");

      const verbName = latestBaseAction.current?.verbName || data?.verbName;
      if (!verbName) throw Error("Invalid Action");

      latestMutation.current({
        data: {
          projectId,
          timestamp: Date.now(),
          ...latestBaseAction.current,
          ...data,
          verbName,
        },
      });
    },
    [projectId, latestMutation, latestBaseAction]
  );
};

```

Figura 49. Función utilitaria para reporte de acciones

En la Figura 50 se puede visualizar un ejemplo de componente usando la función “useAction”. El objetivo del componente “OnStart” (el cual es utilizado al final de la función “SyncAuth” en la Figura 48) es reportar que el usuario hizo uso de la plantilla una vez que se haya completado de forma satisfactoria el proceso de autenticación.

```

const OnStart = memo(function OnStart() {
  const { project } = useAuth();

  const startAction = useAction({
    verbName: "OpenTemplateApplication",
  });

  const projectId = project?.id;

  useEffect(() => {
    if (projectId) startAction();
  }, [projectId, startAction]);

  return null;
});

```

Figura 50. Ejemplo de uso de función utilitaria para reporte de acciones

8.2.4. Diseño principal

Para el diseño principal de la plantilla fueron utilizados algunos componentes del servicio “Chakra UI Pro” (Chakra UI Pro, s.f), el punto central de la definición de este diseño se encuentra disponible en la ruta “/src/components/MainLayout.tsx” del repositorio. Este diseño contiene un conmutador para cambiar el tema de un modo claro a un modo oscuro, en la Figura 51 se puede visualizar el modo claro, mientras que en la Figura 52 se puede visualizar el modo oscuro.

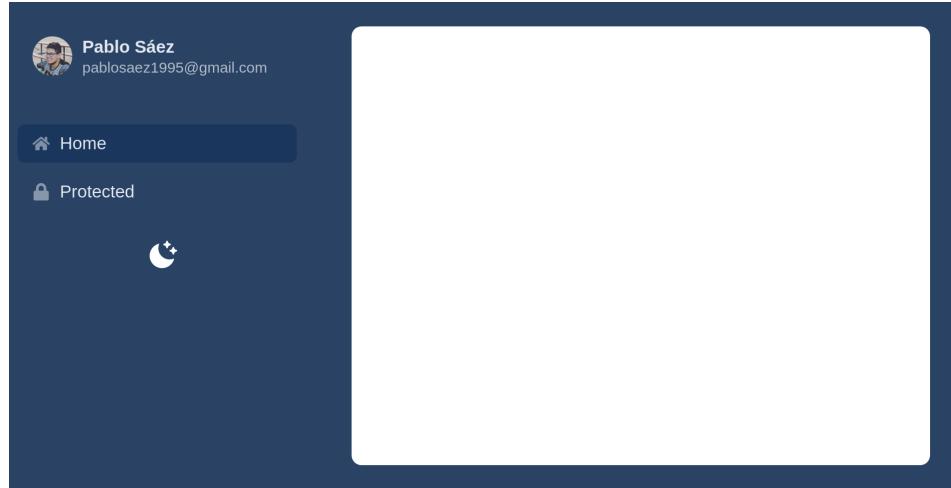


Figura 51. Diseño principal plantilla, modo claro

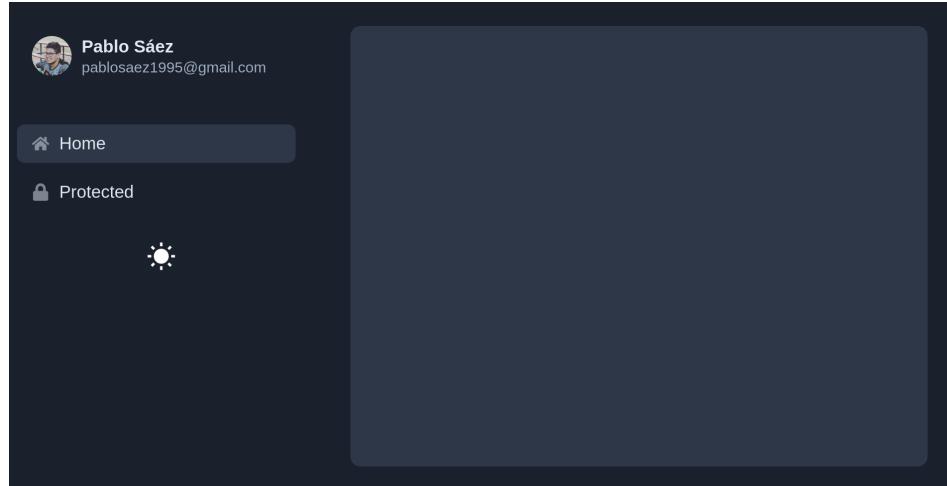


Figura 52. Diseño principal plantilla, modo oscuro

8.2.5. Barra de navegación de aplicación

Como se pudo visualizar en la Figura 51 y Figura 52, parte importante del diseño de la plantilla es la existencia de una barra de navegación, cuya definición se encuentra disponible en la ruta “/src/components/Navigation.tsx” del repositorio, y cuyo código se puede visualizar en la Figura 53. En este ejemplo de navegación se hace uso de información disponible del proceso de autenticación. En el ejemplo (Figura 53) la página siempre muestra un enlace hacia la raíz de la aplicación, y solo cuando el usuario se encuentra autenticado, en enlace hacia la página “/protected”, este se encuentra disponible.

```

import { Stack } from "@chakra-ui/react";
import { FaHome, FaLock } from "react-icons/fa";
import { useAuth } from "./Auth";
import { DarkModeToggle } from "./DarkModeToggle";
import { ScrollArea } from "./ScrollArea";
import { SidebarLink } from "./SidebarLink";

export function Navigation() {
  const { user } = useAuth();
  return (
    <ScrollArea pt="5" pb="6">
      <Stack pb="6">
        <SidebarLink icon={<FaHome />} href="/">
          Home
        </SidebarLink>

        {user ? (
          <SidebarLink icon={<FaLock />} href="/protected">
            Protected
          </SidebarLink>
        )}

      </Stack>
      <Stack alignItems="center">
        <DarkModeToggle />
      </Stack>
    </ScrollArea>
  );
}

```

Figura 53. Definición barra de navegación plantilla

En la Figura 54 se puede ver como la barra de navegación cambia cuando un usuario entra por primera vez y no se autentica con el sistema. El botón “Login” automáticamente redirige al sistema de autenticación de Auth0 y devuelve al usuario a la aplicación una vez que el usuario ingresa sus credenciales.

Por el contrario, en la Figura 55 se puede visualizar el proceso cuando un usuario se encuentra autenticado y desea cerrar sesión. Esto se puede realizar al presionar sobre la caja que muestra el usuario autenticado, este abre un “Popover” conteniendo un botón “Logout” que permite al usuario cerrar sesión.

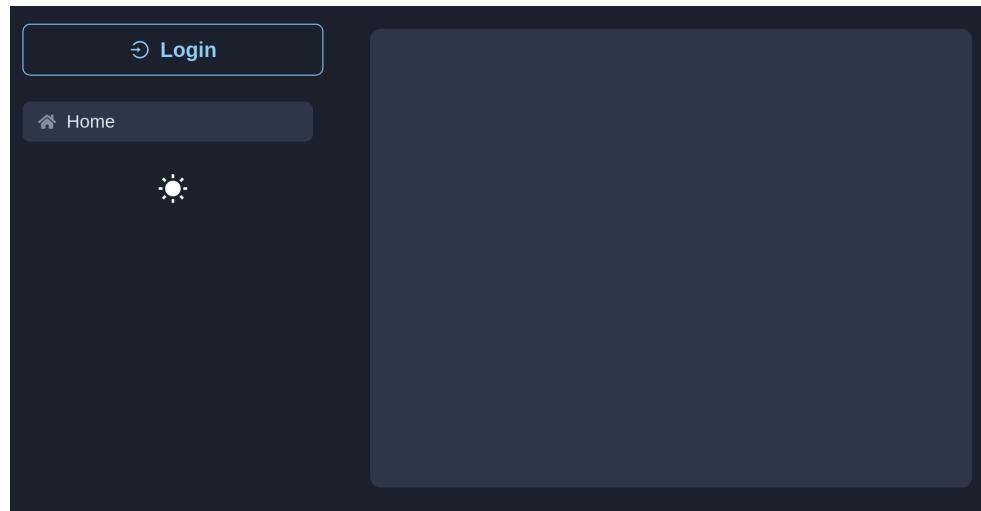


Figura 54. Barra de navegación con usuario sin autenticación

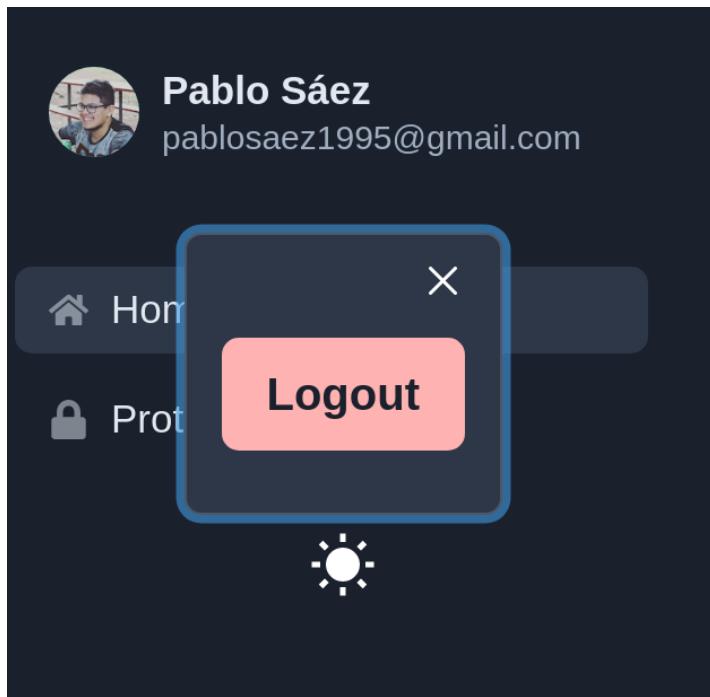


Figura 55. Proceso de cerrado de sesión

9. CONCLUSIONES Y TRABAJO FUTURO

Para el presente trabajo se diseñó e implementó una arquitectura web distribuida basada en tecnologías y herramientas de desarrollo web innovadoras, con el objetivo principal de su utilización en sistemas de educación inteligentes. Para ello se han obtenido requerimientos por parte de investigadores que definieron las funciones básicas que debe incluir dicha plataforma. Los requerimientos incluyeron la flexibilidad para el manejo de datos y la facilidad de integración en aplicaciones que necesitan hacer uso de la plataforma, como lo son los tutores de aprendizaje inteligente.

La plataforma construida consiste en una serie de servicios y un *gateway*. Los servicios incluyen el manejo de acciones, definición de dominio y tópicos, y administración de proyectos y usuarios, mientras que el *gateway* se encarga de unir los servicios para simplificar el uso de estos. Un ejemplo de integración es que la plataforma desde su desarrollo ha sido utilizada por dos trabajos de título que consisten en la implementación de tutores de aprendizaje inteligentes para la preparación pre-álgebra de la facultad de ingeniería de la Universidad Austral de Chile.

9.1. Retrospectiva

9.1.1. ¿Qué se hizo bien?

Durante el proceso de desarrollo una de las mayores prioridades fue la mejora continua de la experiencia de desarrollo, esto a través de la investigación y personalización de los procedimientos de desarrollo, creando herramientas especializadas y pruebas automatizadas. Un ejemplo de herramienta desarrollada fue la librería denominada “bob-tsm” (GitHub - PabloSzx/bob-esbuild, s.f), la cual hace la compilación del código TypeScript sobre la marcha con gran rendimiento. Todo esto logró crear una experiencia de desarrollo grata al reducir los tiempos de espera de forma considerable. Aunque el desarrollo de las herramientas de desarrollo fue realizada en el transcurso del proyecto de título sus beneficios van más allá de la plataforma actual, al permitirse su uso libre en cualquier proyecto que utilice el ecosistema Node.js y TypeScript.

Otro de los aspectos que tuvo especial efecto positivo fue el desarrollo de la plataforma utilizando la especificación GraphQL y todas las tecnologías asociadas que ésta conlleva. Esto permitió la definición preliminar de reglas y una documentación de la plataforma de forma implícita como lenguaje de comunicación. Esto tuvo efecto en el proceso de definición de contratos en forma de diseño de la plataforma, utilizando herramientas que facilitan la visualización y ejemplificación

de lo implementado, potencialmente disminuyendo tiempo en el desarrollo de lógica que no concuerda con los requerimientos finales.

9.1.2. ¿Qué seguiría haciendo?

Durante el progreso del proyecto uno de los objetivos continuos fue la de mejorar los estándares de desarrollo, esto involucró la investigación de formas de mejorar los procesos de desarrollo y la utilización de nuevas y mejores librerías que puedan suplir con las distintas necesidades encontradas en el desarrollo. Este es un proceso continuo que se debe seguir haciendo siempre, en especial en los ecosistemas con crecimiento continuo, como lo es el desarrollo web.

Una de las herramientas claves utilizadas en el desarrollo de la plataforma fue la implementación de pruebas automatizadas, esto permitió verificar la robustez, estabilidad y garantía de los servicios. Este proceso logró, de forma consistente, encontrar errores de codificación y lógica que serían considerablemente más difíciles de diagnosticar y arreglar sin la presencia de éstos, por lo que probaron ser de gran utilidad.

La especial recomendación de lo que se hizo en este proyecto y se debería realizar en todo proyecto que involucra desarrollo de software es el enfoque de tiempo al inicio para establecer un ambiente de desarrollo óptimo.

9.1.3. ¿Qué dejaría de hacer?

Una actividad negativa realizada en el transcurso del proyecto fue el mal manejo del tiempo dedicado para el trabajo en el proyecto, en la cual una parte considerable del trabajo fue de carácter nocturno. Parte de este problema pudo ser atribuido a la situación en la cual el proyecto fue realizado, la cual fue en medio de la pandemia con todas las actividades realizadas de manera *on-line* y desde el hogar, implicando la imposibilidad de dedicar tiempo fijo con trabajo realizado de forma presencial desde la universidad.

9.1.4. ¿Qué empezaría a hacer?

Uno de los potenciales que provee la arquitectura de microservicios es la posibilidad de implementar lógica en distintos ecosistemas y lenguajes de programación, esto en la práctica significa que hay potencial para la investigación y uso de ecosistemas que han ganado especial popularidad por obtener mejores rendimiento al hacer uso de compilación nativa, como lo es con los lenguajes de programación Go (The Go Programming Language, s.f) y Rust (Rust Programming Language, s.f). En estos lenguajes de programación el ecosistema alrededor de GraphQL se ha popularizado

y son una alternativa real al *stack* único de tecnologías utilizado para la versión presentada de la plataforma.

9.2. Trabajo futuro

Producto del proyecto se espera que la plataforma sea utilizada por nuevos trabajos de título, tanto de pregrado como de magister, con el objetivo de implementación de nuevos tutores educativos inteligentes y servicios relacionados como la adición de diversos algoritmos de modelamiento del aprendiz y de selección adaptativa de contenido.

10. REFERENCIAS

- Amazon AWS, accedido el 25 de enero, 2022, desde <https://aws.amazon.com>
- Apollo Federation, accedido el 25 de enero, 2022, desde <https://www.apollographql.com/apollo-federation>
- Auth0, accedido el 25 de enero, 2022, desde <https://auth0.com>
- GitHub - bob-esbuild, accedido el 10 de abril, 2022, desde <https://github.com/PabloSzx/bob-esbuild/tree/main/packages/bob-tsm>
- Brusilovsky & Eva Millán. (2007). User Models for Adaptive Hypermedia and Adaptive Educational Systems. 4321. 10.1007/978-3-540-72079-9_1.
- Caddy Server, accedido el 25 de enero, 2022, desde <https://caddyserver.com>
- Chakra UI, accedido el 25 de enero, 2022, desde <https://chakra-ui.com>
- Chakra UI Pro, accedido el 25 de enero, 2022, desde <https://pro.chakra-ui.com>
- Combining Schemas | Schema Stitching, accedido el 25 de enero, 2022, desde <https://www.graphql-tools.com/docs/schema-stitching/stitch-combining-schemas>
- dbdiagram.io - Database Relationship Diagrams Design Tool, accedido el 25 de enero, 2022, desde <https://dbdiagram.io>
- Docker, accedido el 25 de enero, 2022, desde <https://www.docker.com>
- Docker Compose, accedido el 25 de enero, 2022, desde <https://docs.docker.com/compose>
- Docker Hub, accedido el 25 enero, 2022, desde <https://hub.docker.com>
- Git, accedido el 25 de enero, 2022, desde <https://git-scm.com>
- GitHub - auth0/auth0-react, accedido el 25 de enero, 2022, desde <https://github.com/auth0/auth0-react>
- GitHub - nearform/fastify-auth0-verify, accedido el 25 de enero, 2022, desde <https://github.com/nearform/fastify-auth0-verify>
- GitHub - pmndrs/valtio, accedido el 25 de enero, 2022, desde <https://github.com/pmndrs/valtio>
- GitHub Actions, accedido el 25 de enero, 2022, desde <https://github.com/features/actions>
- Gmail: Free, Private and Secure Email, accedido el 25 de enero, 2022, <https://www.google.com/gmail/about/>
- Google Cloud - <https://cloud.google.com>
- GraphQL | A query language for your API, accedido el 25 de enero, 2022, desde <https://graphql.org>
- GraphQL Code Generator, accedido el 25 de enero, 2022, desde <https://www.graphql-code-generator.com>
- GraphQL Specification, accedido el 11 de mayo, 2021, desde <https://spec.graphql.org>

- GraphQL Voyager, accedido el 25 de enero, 2022, desde <https://github.com/APIs-guru/graphql-voyager>
- HTTP | MDN, accedido el 25 de enero, 2022, desde <https://developer.mozilla.org/es/docs/Web/HTTP>
- IMS Learning Tools Interoperability, accedido el 25 de enero, 2022, <https://www.imsglobal.org/activity/learning-tools-interoperability>
- JSON - JavaScript, accedido el 25 de enero, 2022, desde https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/JSON
- Microsoft Azure - <https://azure.microsoft.com>
- MDX, accedido el 25 de enero, 2022, desde <https://mdxjs.com>
- Next.js, accedido el 25 de enero, 2022, desde <https://nextjs.org>
- Nextra, accedido el 25 de enero, 2022, desde <https://github.com/shuding/nextra>
- Node.js, accedido el 25 de enero, 2022, desde <https://nodejs.org>
- Node.js Cluster, accedido el 25 de enero, 2022, desde <https://nodejs.org/docs/latest-v16.x/api/cluster.html>
- Node.js Undici, accedido el 25 de enero, 2022, desde <https://undici.nodejs.org>
- Outlook - Free personal email and calendar from Microsoft, accedido el 25 de enero, 2022, <https://outlook.live.com>
- PostgreSQL, accedido el 25 de enero, 2022, desde <https://postgresql.org>
- Prisma Schema, accedido el 25 de enero, 2022, desde <https://www.prisma.io/docs/concepts/components/prisma-schema>
- Prisma Migrate, accedido el 25 de enero, 2022, desde <https://www.prisma.io/docs/concepts/components/prisma-migrate>
- Prisma Client, accedido el 25 de enero, 2022, desde <https://www.prisma.io/docs/concepts/components/prisma-client>
- React.js, accedido el 25 de enero, 2022, desde <https://reactjs.org>
- React Query, accedido el 25 de enero, 2022, desde <https://react-query.tanstack.com>
- RFC 7519 - JSON Web Token (JWT), accedido el 25 de enero, 2022, desde <https://datatracker.ietf.org/doc/html/rfc7519>
- Rust Programming Language, accedido el 30 de enero, 2022, desde <https://www.rust-lang.org>
- System Monitor - GNOME Wiki, accedido el 25 de enero, 2022, desde <https://wiki.gnome.org/Apps/SystemMonitor>
- Tomas Fernandez (2021), What is monorepo? (and should you use it?), accedido el 11 de mayo, 2021, desde <https://semaphoreci.com/blog/what-is-monorepo>
- The Go Programming Language, accedido el 30 de enero, 2022, desde <https://go.dev>
- TypeScript, accedido el 25 de enero, 2022, desde <https://www.typescriptlang.org>

- Visual Studio Code, accedido el 25 de enero, 2022, desde
<https://code.visualstudio.com>
- What is xAPI aka Experience API, accedido el 25 de enero, 2022,
<https://xapi.com/overview>

11. ANEXOS

Anexo A: Panel de administración, página de proyectos

Disponible en <https://admin.lm.inf.uach.cl/projects>

The screenshot shows the 'Projects' section of the Altair GraphQL Admin interface. On the left, there is a sidebar with a user profile for 'Pablo Sáez' and a list of navigation items: Users, Groups, Domains, Topics, KC, Projects (which is highlighted), Content, Actions, Altair GraphQL Web Client, and Voyager GraphQL Schema Visualization. The main area displays a table of projects with the following data:

ID	CODE	LABEL	DOMAINS	CREATED AT	UPDATED AT	EDIT
1	project1	projeto 1	1 Matemáticas	10/11/2021, 23:18 GMT-3	25/11/2021, 12:14 GMT-3	
2	exampler	Example	No domains	15/11/2021, 15:00 GMT-3	15/11/2021, 15:00 GMT-3	
3	equation_tutor	equation_tutor	No domains	08/12/2021, 23:07 GMT-3	08/12/2021, 23:07 GMT-3	
4	factorize_tutor	factorize_tutor	1 Matemáticas	08/12/2021, 23:42 GMT-3	08/12/2021, 23:42 GMT-3	

Anexo B: Cliente GraphQL Altair

Disponible en <https://lm.inf.uach.cl/altair>

The screenshot shows the GraphQL playground interface. At the top, there's a header with a logo, the title "currentUser", a "+ Add new" button, and environment settings ("No environment" and "Docs"). Below the header, there's a toolbar with a "POST" button, the URL "https://lm.inf.uach.cl/graphql", and buttons for "Send Request", "Docs", and "Result". The main area is divided into sections: "Query", "Pre-request", "Post-request", "Result", "Subscription result", and "Response headers".

Query:

```
▶ (Run query currentUser)
1 ↴ query currentUser {
2   ↴ currentUser {
3     id
4     name
5     email
6   }
7 }
```

Result:

```
200 OK (93ms)
1 ↴ {
2   ↴ "data": {
3     ↴ "currentUser": {
4       "id": "1",
5       "name": "Pablo Sáez",
6       "email": "pablosaez1995@gmail.com"
7     }
8   }
9 }
```

VARIABLES:

Anexo C: Documentación plataforma

Disponible en <https://docs.lm.inf.uach.cl>

The screenshot shows a documentation page for 'Learner Model GQL'. The left sidebar has a 'Getting started' section with links to various topics like How it works?, Architecture, Auth, Database, Gateway, Instructions, API usage, Administration, Client side template, Create a service, Data modeling, Database tooling, and Deployment. The main content area has a title 'Learner Model GQL' and a sub-section 'Getting Started'. It includes sections for 'Installation' (with instructions for Node.js v16.x, Docker, and Git), 'Development' (with instructions for pnpm), and environment variables (DATABASE_URL, AUTH0_DOMAIN, AUTH0_CLIENT, AUTH0_SECRET, ADMIN_USER_EMAIL). There are also links for 'Architecture >' and 'Edit this page'.

Learner Model GQL

Getting started

How it works?

Architecture

Auth

Database

Gateway

Instructions

API usage

Administration

Client side template

Create a service

Data modeling

Database tooling

Deployment

Learner Model GQL

Learner Model GQL is a project based on a [microservices architecture](#) using [GraphQL](#).

Getting Started

Installation

For development with learner-model-gql it is required to have installed:

- Node.js v16.x, which can be installed in different ways based on the platform, or you can visit [nodejs.org](#)
- Docker for local database usage and testing, [Docker Desktop](#) can be used in macOS or Windows
- Git version control system

For Windows based systems, you might need to use tools like [Cmder](#) to be able to use bash-like commands.

Development

This project requires [pnpm](#), which is a dependency manager for Node.js, which you can install by following the instructions on [pnpm.io/installation](#).

Assuming you already have access to the confidential Auth0 Credentials

- Clone <https://github.com/PabloSzx/learner-model-gql>
- Inside the cloned repository, run `pnpm i` to install all the required dependencies
- Run `pnpm db:local`, this will start a local postgres database in the port `5789`, which can be used for development and its required for automated testing.
- In the root of the cloned repository, an empty `.env` should be present, you have to manually modify it to look something like:

```
DATABASE_URL=postgresql://postgres:postgres@localhost:5789/postgres
AUTH0_DOMAIN=learner-model-gql.us.auth0.com
AUTH0_CLIENT=__
AUTH0_SECRET=__
ADMIN_USER_EMAIL=put_your_email_here@gmail.com
```

The `AUTH0` environment variables have to be changed based on the confidential credentials, the `DATABASE_URL` can be adapted to point to an external database if required, and `ADMIN_USER_EMAIL` should be changed to set the starting admin user.

- After the local database is running and the `.env` is filled, run `pnpm migrate`, which will run the migrations on the local database.
- Run `pnpm dev`. The gateway's [Altair GraphQL Client](#) is going to be available at <http://localhost:8080/altair>, and administration client will be available at <http://localhost:4010>.

Architecture >

Anexo D: Repositorio principal GitHub

Disponible en <https://github.com/PabloSzx/learner-model-gql>

The screenshot shows the GitHub repository page for `PabloSzx/learner-model-gql`. The repository is public and has 908 commits. The code tab is selected, showing a list of commits. The repository has 0 forks and 1 star. It includes sections for About, Contributors, and Languages.

About

- `admin.learner-model-gql.pablosz.dev`
- Readme
- MIT License
- 1 star
- 2 watching
- 0 forks

Contributors 3

- PabloSzx Pablo Sáez
- renovate-bot WhiteSource Renovate
- renovate[bot]

Languages

- TypeScript 99.7%
- Other 0.3%

Code

main 3 branches 0 tags

File	Description	Time Ago
<code>PabloSzx remove unused type definition</code>	fixes	3 months ago
<code>.github</code>	tests documents codegen	5 months ago
<code>.husky</code>	force LF	3 months ago
<code>.vscode</code>	add valtio in libraries of template	2 days ago
<code>docs</code>	remove unused type definition	5 hours ago
<code>packages</code>	fixes	4 days ago
<code>test</code>	fixes	4 days ago
<code>.c8rc.json</code>	fixes	11 days ago
<code>.dockerignore</code>	optimize docker image	5 months ago
<code>.gitignore</code>	test db environment	5 months ago
<code>.graphqlrc.yml</code>	actions progress & testing	5 months ago
<code>.mochcharc.cjs</code>	fixes	3 months ago
<code>.npmrc</code>	update & fix node version	14 days ago
<code>.prettierignore</code>	minor fixes	5 months ago
<code>.prettierrc</code>	minor fixes	5 months ago
<code>Dockerfile</code>	move generated prisma client directory	11 days ago
<code>LICENSE</code>	Initial commit	8 months ago
<code>README.md</code>	reference new domain	26 days ago
<code>capitulos.md</code>	estructuramiento	2 months ago
<code>docker-compose.yml</code>	add commented "mono" in docker-compose	28 days ago
<code>package.json</code>	update rq-gql	2 days ago
<code>pnpm-lock.yaml</code>	update rq-gql	2 days ago
<code>pnpm-workspace.yaml</code>	move packages & update ts scripts	5 months ago
<code>pre-test.mjs</code>	use prisma migrate deploy	2 months ago
<code>renovate.json</code>	renovate ignore nextra	16 days ago
<code>schema.gql</code>	add kcs filter to admin content connection	10 hours ago
<code>tsconfig.json</code>	separate service types	14 days ago

README.md

learner-model-gql

Visit <https://docs.lm.inf.uach.cl> for documentation.



© 2022 GitHub, Inc.

Terms

Privacy

Security

Status

Docs

Contact GitHub

Pricing

API

Training

Blog

About