Valeiras, Lautaro Alberto

Sistema de gestión y monitoreo de servicios de Windows

2020

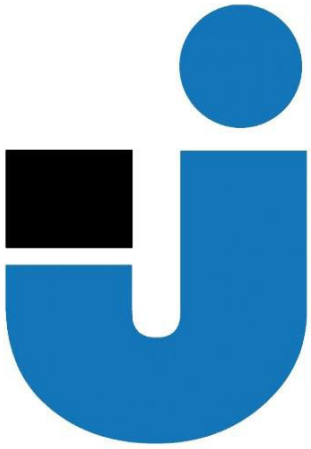
Instituto: Ingeniería y Agronomía

Carrera: Ingeniería en Informática

Universidad Nacional Arturo Jauretche

Instituto de Ingeniería y Agronomía

Ingeniería en Informática



TRABAJO FINAL DE LA PRÁCTICA PROFESIONAL SUPERVISADA

Sistema de gestión y monitoreo de servicios de Windows

Estudiante:

Lautaro Alberto Valeiras

Tutores:

Prof. Lía Lavigna

Dr. Ing. Martin Morales

Ing. Rafael Villalba

Ing. Marco Gobbi

Buenos Aires, 2020

PRÁCTICA PROFESIONAL SUPERVISADA (PPS)
Sistema de gestión y monitoreo: DMSCC-UI.
Informe de Avance

DATOS DEL ESTUDIANTE

Apellido y Nombres: Valeiras Lautaro Alberto

DNI: 39.484.756

Nº de Legajo: 14580

Correo electrónico: lautaro.valeiras@hotmail.com

Cantidad de materias aprobadas al comienzo de la PPS: 45/46

PPS enmarcada en artículo (4 ó 7) de la Resolución (CS) 103/16. (en caso de ser artículo 7 aclarar en cuál de las dos alternativas posibles se encuadra)

DOCENTE SUPERVISOR

Apellido y Nombres: Dr. Ing. Martin Morales

Correo electrónico: martin.morales@unaj.edu.ar

DOCENTE TUTOR DEL TALLER DE APOYO A LA PRODUCCIÓN DE TEXTOS ACADÉMICOS DE LA UNAJ

Apellido y Nombres: Prof. Lía Lavigna

Correo electrónico: lialavigna@gmail.com

DATOS DE LA ORGANIZACIÓN DONDE SE REALIZA LA PPS

Nombre o Razón Social: Tecnom S.R.L

Dirección: Diagonal 74 Nro. 1463, B1900BZI La Plata, Buenos Aires

Teléfono: 011 5254-9236

Sector: Producto

TUTOR DE LA ORGANIZACIONAL

Apellido y Nombres: Ing. Rafael Villalba / Ing. Marco Gobbi

Correo electrónico: rvillalba@tecnom.com.ar / mgobbi@tecnom.com.ar

FIRMA DEL COORDINADOR DE LA CARRERA

Tabla de contenido

Imágenes y gráficos

1. INTRODUCCIÓN

1. Presentación

El presente trabajo está enfocado principalmente en el desarrollo de una aplicación Web, llamada "**DMSCC-UI**" (dealership management system cloud connector user interface), utilizando todos los recursos y conocimientos adquiridos durante la carrera Ingeniería en Informática.

Este desarrollo será considerado una herramienta de gran utilidad para la empresa Tecnom S.R.L. La compañía se encuentra ubicada en la ciudad de La Plata y se dedica al desarrollo de aplicaciones CRM (Customer Relationship Management). Un CRM es una solución de gestión de las relaciones con clientes y orientada normalmente a administrar tres áreas básicas: la gestión comercial, el marketing y el servicio postventa o de atención al cliente. Tecnom, por su parte, permite a sus CRM integrarse con aplicaciones de otras compañías para darle un valor agregado, y para manejar estas integraciones cuentan con un software llamado DMSCC (dealership management system cloud connector). La aplicación propuesta en este trabajo (DMSCC-UI), trabajará sobre esta última permitiendo, entre otras cosas, facilitar el trabajo del área de infraestructura de la organización, recortando el tiempo de respuesta ante eventuales fallos del DMSCC.

Los productos CRM, desarrollados por Tecnom, están enfocados al negocio de concesionarios, los cuales cuentan con la opción de integrarse a diversas fuentes externas, una de ellas son los **Sistemas DMS (dealership management system),** que llevan el control del taller de los concesionarios (registros de turnos, servicios de post-ventas), stock de vehículos, registro de ventas históricas, etc. Los sistemas DMS, **no son productos desarrollados por Tecnom**, éstos son propios de cada concesionario, es decir, de clientes de Tecnom.

2. Problemática

En la actualidad la compañía posee una herramienta desarrollada por su propio personal llamada **DMS Cloud-Connector (o DMSCC),** la cual, ofrece un servicio de Windows que lee información directamente desde la base de datos del, ya mencionado, sistema **"DMS"** y lo comunica a la aplicación CRM (Producto de Tecnom). Es decir, permite la integración entre ambos sistemas, por un lado, el producto (CRM) y por el otro, los sistemas DMS (propios de los clientes de Tecnom).

Este servicio se instala, en muchos de los casos, localmente en cada servidor de los concesionarios que deseen integrar el CRM con sus respectivos **sistemas DMS**, esto último, conlleva los siguientes problemas:

-**Falta de visibilidad de donde se localiza cada conector:** En qué servidor se encuentra instalado, de qué manera se puede acceder a dicho servidor, quién es el responsable en el concesionario en caso de que se necesite acceder. Además, es necesario destacar que algunos servers son administrados por Tecnom y otros administrados por sus clientes, es decir, por cada concesionario.

-**Falta de capacidad de acción sobre los conectores:** En caso de que ocurra un problema con el DMSCC (conector entre el DMS y el CRM), el tiempo de respuestas para la recuperación es muy largo, ya que no existe hoy un método que permita localizar el error rápidamente por la falta de visibilidad nombrada anteriormente. Conjuntamente, al momento de dar de alta un nuevo conector (DMSCC) o actualizar una versión del mismo, es necesario acceder manualmente a cada servidor que se encuentra localizado en los respectivos concesionarios.

3. Propuesta

El propósito de este proyecto será, como primer alcance, solucionar la falta de visibilidad en donde se localiza cada conector, por lo que se espera obtener

transparencia y orden, con el desarrollo de una aplicación que centralice la información de forma dinámica y fácil de observar. De tal modo, que la información detallada, permita gestionar al sector de infraestructura las acciones requeridas ante fallos de los conectores y/o DMS, permitiendo una rápida recuperación, elevando así el valor agregado del producto.

Estas tareas serán desarrolladas en su totalidad con herramientas de Amazon, cuya infraestructura es completamente Serverless, es decir, no necesita de la instalación de servidores físicos.

Las herramientas utilizadas son las siguientes:

AWS Amplify Framework: Es un framework (marco de trabajo, conjunto de herramientas), emergente desarrollado por Amazon con la cual se puede construir una aplicación Web de forma rápida y sencilla.

Lambdas: AWS Lambda es un servicio informático que permite ejecutar el "código de programación" sin aprovisionar ni administrar servidores. AWS Lambda ejecuta el código sólo cuando es necesario, y se escala de manera automática, pasando de pocas solicitudes al día a miles por segundo.

DynamoDB: Es una base de datos '*no relacional*' que ofrece rendimiento en milisegundos de un solo dígito a cualquier escala. Se trata de una base de datos duradera de varias regiones, completamente administrada, que cuenta con copia de seguridad, restauración y seguridad integradas, y almacenamiento de caché en memoria para aplicaciones a escala de Internet.

AppSync: Es un servicio administrado que utiliza **GraphQL (Análogo a API Rest)** para facilitar a las aplicaciones la obtención de los datos específicos que necesitan.

Con AppSync se pueden crear aplicaciones de escala ajustable, incluidas aquellas que necesitan actualizaciones en tiempo real en una gama de orígenes de datos, como almacenes de datos NoSQL, bases de datos relacionales, APIs y sus orígenes de datos personalizados con AWS Lambda.

4. Objetivos

1) Desarrollar una aplicación Web, donde se centralicen los datos de infraestructura de las aplicaciones, DMSCC (herramienta de integración) y los sistemas DMS (sistemas propios de los clientes de Tecnom).

2) Ganar visibilidad y orden, para así, poder brindar un producto de alta disponibilidad a los clientes.

3) Ganar capacidad de acción ante fallos e inconvenientes provenientes de los sistemas DMS.

4) Informar, automáticamente, el alta de nuevos conectores a las aplicaciones del sector de infraestructura.

5) Testear y explorar las tecnologías usadas en el desarrollo de esta solución, para integrarlas a futuros nuevos productos de Tecnom.

5. Tareas a ejecutar

1 - Investigar Tecnologías

Como tarea inicial, se procederá a investigar las tecnologías de Amazon que se implementarán, es decir, descubrir su sintaxis y los problemas que éstas resuelven.

AWS Cognito → Seguridad y Autenticación.
AWS AppSync → Servicio que permite sincronización en tiempo real con la base de datos y administrar las API GraphQL.
API GraphQL → Nuevo paradigma de acceso a datos mediante protocolo HTTP.

AWS Lambda →Servicio de Amazon que permita ejecutar "código de programación" sin necesidad de administrar ni aprovisionar infraestructura, es decir, servidores.

2 -Definir pantallas para la aplicación

La herramienta utilizada para la creación de las vistas, es Angular 8, por lo que en esta etapa se desarrollan todas las pantallas/vistas de la aplicación.

3 - Asegurar, autorizar y autenticar

En esta fase se delimitan las métricas de acceso a la aplicación, método de "Login", y perfiles a implementar en el caso de que aplique.

4 - Definir las entidades para el backend a desarrollar con Amplify

En esta etapa se definen las entidades y modelado de datos que serán almacenados en la base de datos de la aplicación, así como también los métodos de acceso a los mismos.

5- Realizar lecturas de métricas de los CRM y DMSCC

En esta etapa se definen los métodos para la obtención de datos de los productos (CRM) y los sistemas DMSCC en cuestión, para centrarlos en la aplicación.

6 - Observar lista de conectores

Listar todos los conectores, con la información más importante disponible a simple vista y mostrar su detalle al seleccionarlos.

7- Iniciar las cargas de datos

Realizar la primera carga de datos de forma manual, es decir, que por cada aplicación CRM, adjuntar su respectivo DMSCC asociado.

8 - Poner en marcha la producción

Luego de la carga inicial, asignar un nombre de dominio, por ejemplo ("*www.dmscc-ui-tecnom.com.ar*") y ejecutar la aplicación productiva.

Cronograma de trabajo

Tareas	Responsables	Horas (hs)	Estado
Pre-requisitos			
Investigacion sobre Amplify	Valeiras Lautaro		
Definir objetivos	Rafael Villalba/Marco Gobbi		
Inicio			
Aplicacion de prueba("hola mundo")	Valeiras Lautaro	20	Completado
Definicion de pantallas/vistas	Valeiras Lautaro	35	Completado
Seguridad y autenticacion	Valeiras Lautaro	10	Completado
Desarrollo			Completado
Defincion de entidades y modelado de datos	Valeiras Lautaro	30	Completado
Desarrollo API GraphQL	Valeiras Lautaro	40	Completado
Ver lista de conectores y poder buscar/filtrar	Valeiras Lautaro	20	Completado
Ver detalle de un conector	Valeiras Lautaro	10	Completado
Completar desarrollo	Valeiras Lautaro		Completado
Operaciones			Completado
Carga Inicial de datos	Valeiras Lautaro	15	Completado
Prueba del sistema	Valeiras Lautaro	10	Completado
Puesta en producción	Valeiras Lautaro	10	Completado
Lanzamiento			
		200	

2. DESARROLLO

1. Investigación de tecnologías y definición de objetivos

Como se explicó en la introducción, se procederá a investigar las tecnologías de Amazon que se implementarán, es decir, se descubrirá su sintaxis y los problemas que éstas resuelven, principalmente *Amplify*, que es la que contiene todas las herramientas necesarias para el desarrollo de este proyecto.

Amplify es una plataforma de desarrollo serverless diseñado para crear aplicaciones móviles y aplicaciones web.

Consta de tres características clave:

-Bibliotecas: vienen con un conjunto de bibliotecas integrales para JavaScript, Android e iOS asociadas con servicios en la nube, como análisis, autenticación, almacenamiento, notificaciones, REST y API GraphQL, AR / VR.

-Amplify CLI: herramienta de línea de comandos para configurar y administrar infraestructuras sin servidor en AWS.

-Componentes de la interfaz de usuario: conjunto de componentes de interfaz de usuario para React, ReactNative, Angular, Ionic y Vue para autenticación, selector de fotos y para facilitar sus desarrollos.

Concepto serverless:

Serverless significa literalmente “sin servidor”, es decir, es un tipo de arquitectura donde los servidores dejan de existir para el desarrollador y, en cambio, el código se ejecuta en “ambientes de ejecución” que administran proveedores como Amazon, Google, IBM. El proveedor define qué lenguajes y versiones soporta y el desarrollador se encarga de escribir el código en dichos lenguajes. Cuando su función es invocada, ya sea por un request HTTP u otro evento, el ambiente de ejecución es iniciado, el código se ejecuta e inmediatamente el ambiente desaparece. Ejemplo de esto es si la función es invocada mil veces, el proveedor se encarga de escalar y generar el número de ambientes necesarios para responder a las mil invocaciones.

De modo que, los proveedores cobran por el tiempo de ejecución del código y mientras más rápido termine su función, menor será el costo que se pague. Por

lo tanto, se busca que la función sea pequeña y con un único propósito, y por este motivo, serverless es relacionada frecuentemente con micro-servicios.

Las funciones serverless son sencillas de usar cuando no se requiere guardar estado en memoria. Debido a que no se tiene control acerca de cuándo los ambientes de ejecución son creados o destruidos, es decir, no se puede asumir que al guardar un dato en la memoria de la función éste se mantenga allí cuando la función sea nuevamente invocada.

Las funciones Serverless que brinda Amazon son las *Lambdas* y éstas soportan los siguientes entornos de ejecución:

- Node.js
- Java 8
- Python
- .Net Core:
- Go 1.x

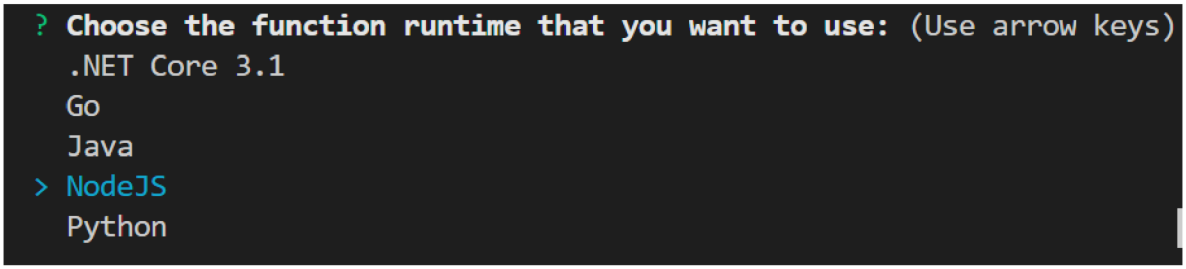


Figura #1 Selección de lenguaje de programación
Fuente: Elaboración propia

Un ejemplo de composición de una *Lambda* desarrollada en Node.js es lo que se observa en la Figura #2.

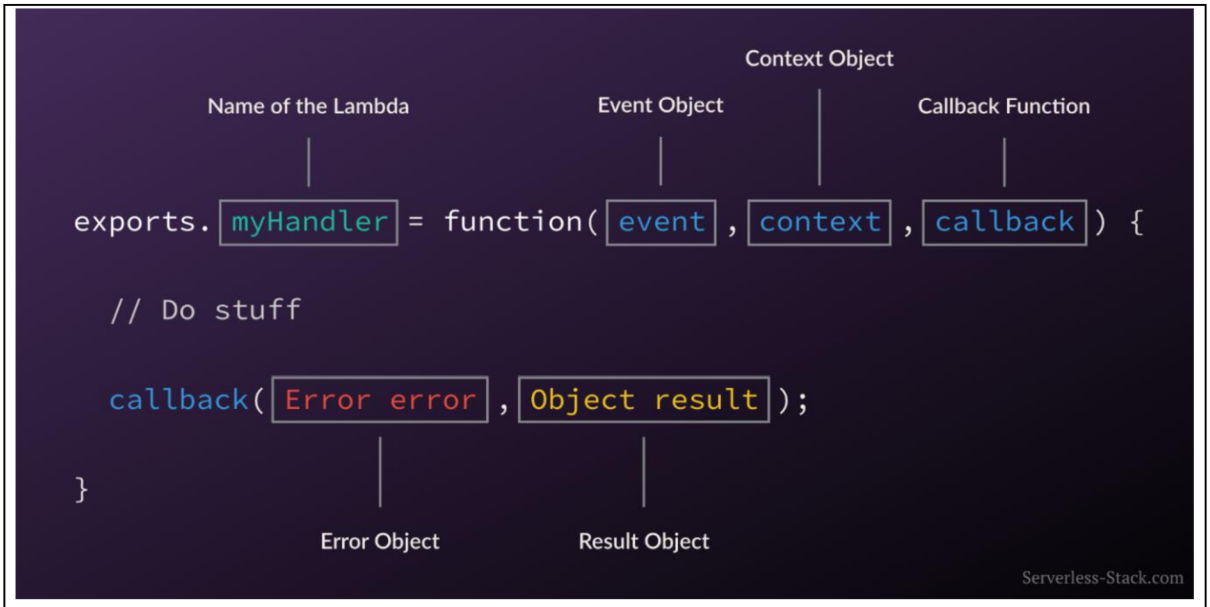


Figura #2 Estructura de una lambda en Node.js

Fuente: https://docs.aws.amazon.com/es_es/lambda/latest/dg/applications-usecases.html

Una *Lambda* puede ejecutarse durante un máximo de 900 segundos o 15 minutos.

Esto significa que la *Lambda* no está destinada a procesos de larga ejecución y la tarifa que cobra Amazon es en base al tiempo de ejecución de la función, y se calcula desde que comienza a ejecutarse hasta que termina.

Para recapitular se puede expresar que *Amplify*, permite el uso de *Lambdas*, autenticación de usuarios, almacenamiento de datos de manera segura, cubriendo todo este flujo de trabajo mediante su herramienta de línea de comando (CLI).

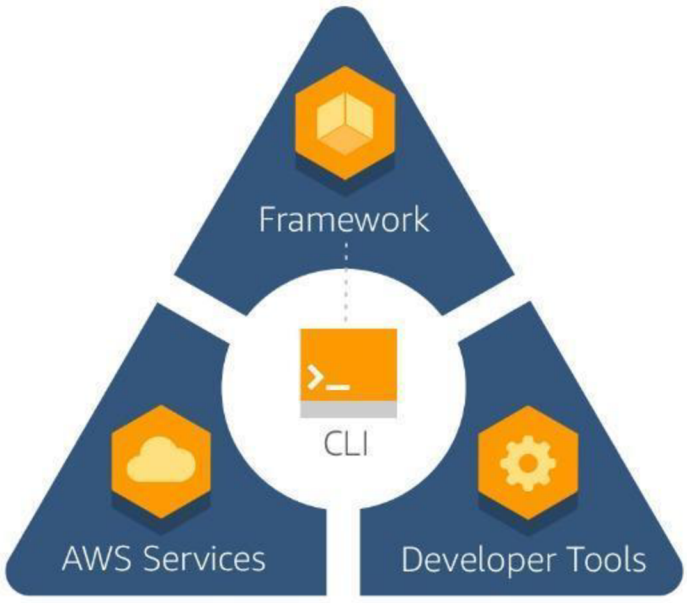


Figura #3 Estructura de la CLI
Fuente: https://aws.amazon.com/es/amplify/

Esta CLI de AWS *Amplify* permite añadir o modificar de forma local los recursos que sean necesarios para la aplicación creada, esto se ejecuta mediante AWS cloudformation la cual está asociada a una cuenta de Amazon, de aquí el concepto "infraestructura como código".

Otras de las herramientas claves de *Amplify* que se utilizarán en este proyecto son AWS AppSync y GraphQL.

AppSync es un nuevo servicio que permite administrar y actualizar datos de aplicaciones móviles en tiempo real, entre los dispositivos y la nube, para que sea posible esta comunicación se utiliza un lenguaje de datos denominado GraphQL.

GraphQL es un lenguaje de consultas (al igual que es SQL), desarrollado por facebook en el 2012, el cual está diseñado para que el cliente (frontend, app móvil) se encargue de diseñar sus propias consultas independientemente a la fuente de datos, ya que admite tanto base de datos relacionales como no

relacionales, o también su fuente de datos puede ser una API REST u otra API GraphQL.

Al ser un lenguaje de consultas, éste puede describir los datos exactos que necesita, permitiendo así enviar una única solicitud HTTP y obtener múltiples recursos. Esto significa menos sobrecarga de red y más rendimiento, y a diferencia de las API REST tradicionales, las API GraphQL poseen un único endpoint donde se realizan las peticiones únicamente mediante el método POST, y los datos se obtendrán según la Query que se diseñe.

En la Figura #4 se observa el ejemplo de una petición a un endpoint de GraphQL.

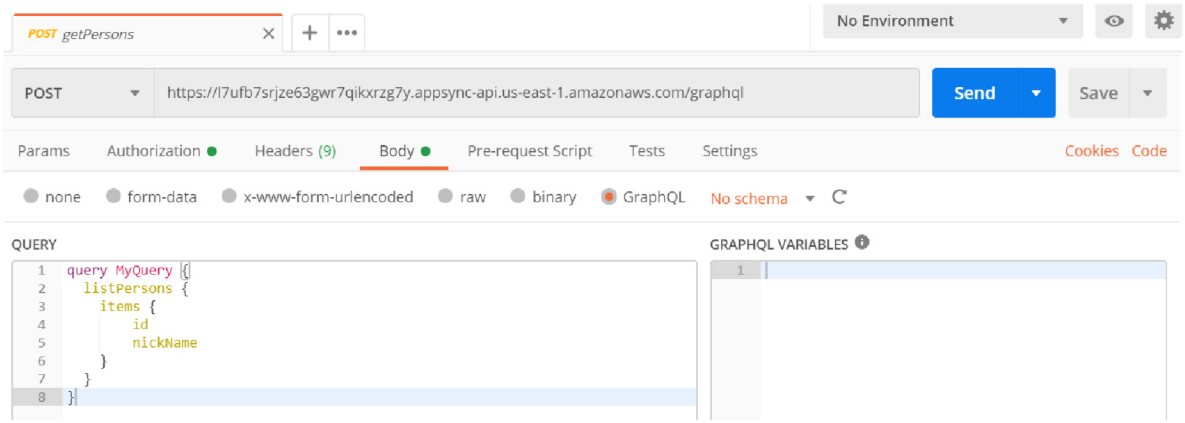


Figura #4: Ejemplo de una petición a un endpoint GraphQL.
Fuente: Elaboración propia

1. Construcción de una API GraphQL:

A. Schema

Es el primer paso para construir una API *GraphQL*, aquí se define toda la estructura de información que devolverá la consulta. En el esquema se definen los tipos de datos, las relaciones entre ellos, y directivas personalizadas (customquery, mutations).

Sistemas de tipos: *GraphQL* trabaja con un sistema propio de tipos, que permite describir a una API con tipos de datos. La estructura de datos definida de este modo son las que crean el marco para las consultas y cada tipo consta de

uno o varios campos, que a su vez contienen sus propios tipos de datos. Este sistema individual sirve a *GraphQL* como punto de orientación para validar consultas y poder rechazar las erróneas.

Sintaxis:

GraphQL proporciona su propio lenguaje de definición de esquema (SDL), que proporciona una forma independiente de definir una estructura de datos sin importar qué lenguaje de programación se use.

Types:

Los elementos básicos de un esquema *GraphQL* son los tipos de objetos, éstos se crean a partir de los tipos escalares incorporados según se muestran en la Figura #5.

```
ID,
String,
Int,
Float,
Boolean
```

Figura #5 Datos primitivos soportados en GraphQL.
Fuente: Elaboración propia

A continuación, en la Figura #6 se ejemplifica el esquema de *GraphQL.*

```
# Sistema CRM (Tecnom)
type App @model {
  id:ID!
  key_name:String!
  build:String!
  url:String!
  stage:String!
  status:String
  alerts:String
  connectors:[DMSConnector] @connection(name:"DMS-APP")
}
```

Figura #6: Ejemplo de un esquema de GraphQL
Fuente: Elaboración propia

B. Queries

Las Queries sirven para obtener los datos (como en REST {GET}) del servidor, en el caso de este proyecto, *Amplify* se encargará de diseñar todas las Queries en base al schema que se haya definido previamente, sólo es necesario diseñarlas para casos puntuales de uso.

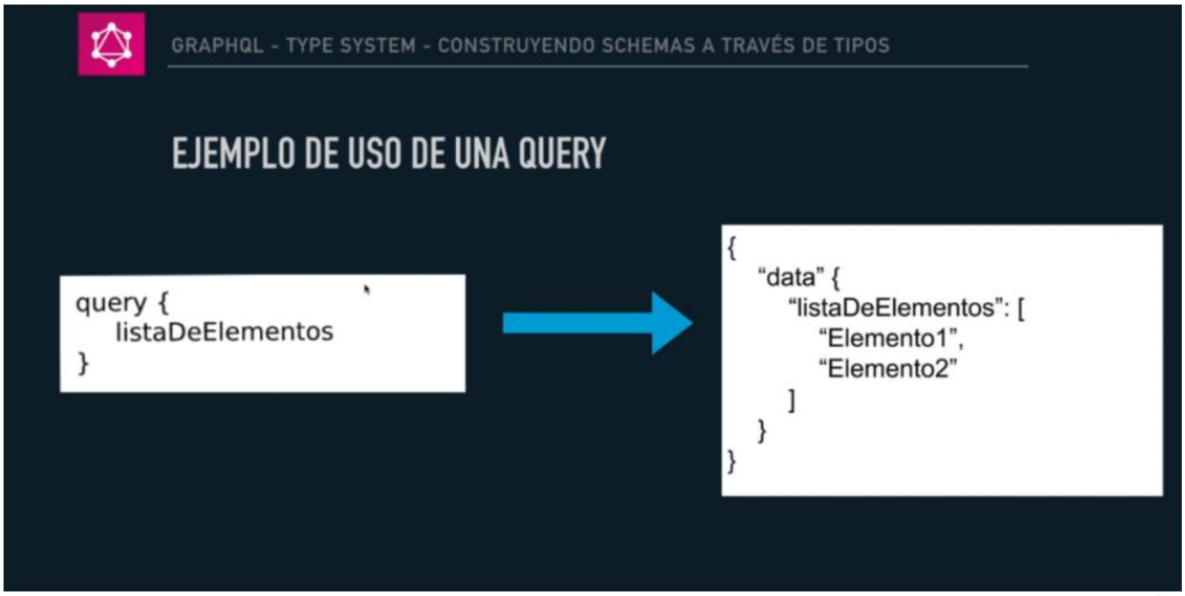


Figura #7: Ejemplo de una consulta
Fuente: Curso Udemy

En el sector izquierdo de la Figura #7 se observa el formato de una Query en una petición al servidor, y en el sector derecho se visualiza la respuesta con los datos.

C. Mutations

Las Mutations son operaciones enviadas al servidor para crear, actualizar o eliminar datos y serán equivalentes a los métodos POST, PUT, PATCH, DELETE en APIs basadas en REST. En este caso la Mutation será similar a una función, es decir, recibirá ciertos parámetros, realizará un cambio y retornará una respuesta.

En *Amplify* las Mutations, al igual que con las Queries, son generadas automáticamente en base al schema definido previamente, según se ejemplifica en la Figura #6.

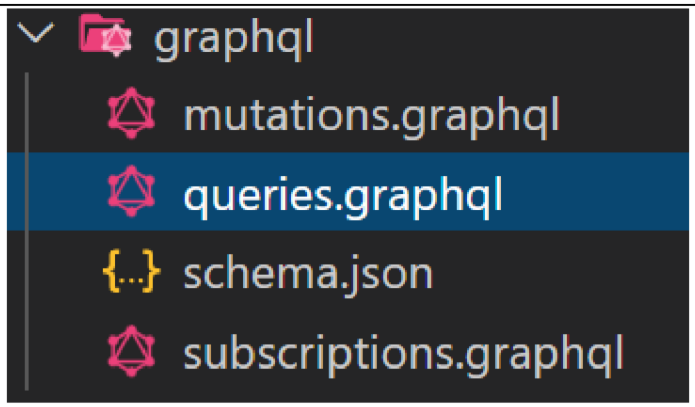


Figura #8: Queries, mutations y subscriptions generados por Amplify
Fuente: Elaboración propia

D. Subscriptions

Los Suscripciones son puntos de entrada que sirven para obtener la información en tiempo real, las cuales se realizan agregando un manejador que notifica las suscripciones al servidor mediante WebSocket, lo que permite una interacción en tiempo real ante cualquier cambio en la información.

De esta manera, se reitera que *Amplify* es quien se encarga de generarlas, lo que permite al desarrollador ahorrar tiempo de trabajo.

2. Definición de vistas y casos de uso

1. C01[1] Lista de conectores, búsqueda y filtración

En la pantalla principal de la aplicación se observará la lista de todos los dms-cloud-connectors, pertenecientes a los CRM de los clientes que se encuentran activos en producción con su información más importante:

- CRM (aplicación asociada): nombre del CRM por ejemplo 'vwmotors'
- Sistema DMS: nombre del DMS asociado.
- Instancia del conector: Tipo de conector (dmscc, *Lambda*), versión, estado
- Servidor: nombre del servidor en el que se encuentra alojado el conector y el dueño del mismo (cliente o Tecnom)

[1] C01 = Caso de uso 1

Adicionalmente, la vista debe tener un buscador donde se filtren las siguientes propiedades:

- Ver solo los sistemas dms 'Autodealer'.
- Ver solo los conectores de testing.
- Ver solo conectores instalados en determinado server.

2. C02 Detalles de un conector

Ubicados en la misma vista del caso C01, se selecciona un conector (al hacer click) e inmediatamente se puede visualizar el detalle del mismo, con un nivel de zoom mayor a lo visto en C01.

- CRM (aplicación asociada):

- Key_name → nombre del crm
- Build → versión del crm instalado
- Url → dirección de la página web del crm
- Stage → ambiente del crm (producción/testing)
- Status → estado del crm (activo/inactivo)

- Sistema DMS:

- name: nombre del sistema dms, ejemplo: 'autodealer'
- version: versión del sistema, ejemplo: 'v19.11.003'
- versionDb: versión de la base de datos.

- Servidor db:

- key_name: nombre del servidor del dms.
- so: sistema operativo del servidor.
- status: estado del servidor (activo/inactivo).
- owner: propietario del servidor (tecnom/cliente).
- descripcion: información extra sobre el servidor.

- Instancia del servidor:

- version: versión instalada del conector, y su caducidad.

- status: estado del conector (encendido/apagado).
- config: configuración extra de los conectores
- jobs: tareas programadas del conector

- Servidor dmscc:

- key_name: nombre del servidor del dmscc.
- so: sistema operativo del servidor.
- status: estado del servidor (activo/inactivo).
- owner: propietario del servidor (tecnom/cliente).
- description: información extra sobre el servidor.

3. C03 Alta de un conector

En esta puede ser posible dar de alta un nuevo conector, con todos los atributos marcados en el caso C02, junto a sus correspondientes servidores donde se encuentran alojados.

4. C04 Edición de un conector

Partiendo de la lista de información del caso C02, se editan los datos del conector: activar o desactivar Jobs(tareas programadas) y modificar datos de los servidores en los que se encuentran alojados.

3. Arquitectura y desarrollo de la Api

Previo a realizar el modelado de datos, se diseñó un esquema de cómo será la arquitectura de la aplicación, esto señala las aplicaciones con las que se comunica y cómo éstas obtienen los datos que luego se utilizarán para mostrar y procesar.

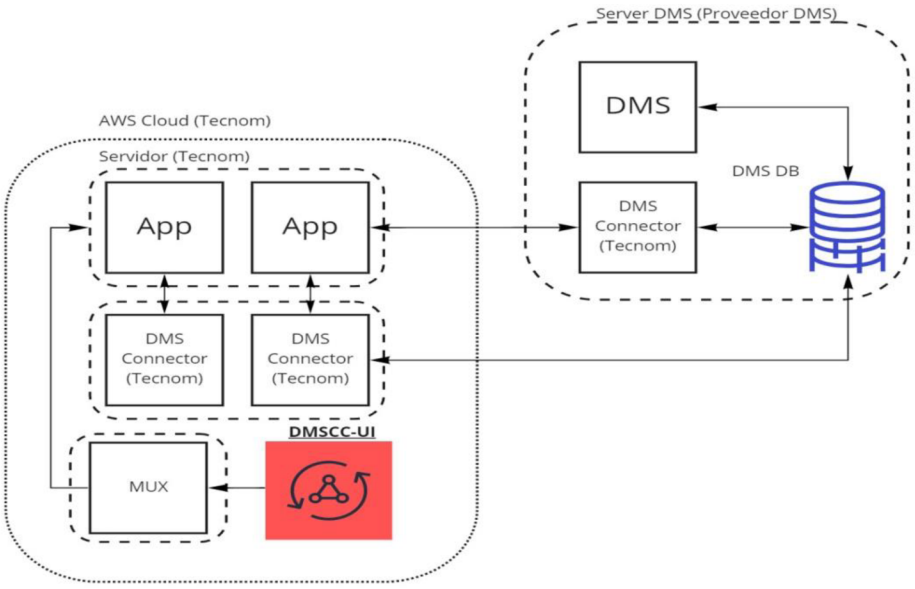


Figura #9: Arquitectura de la aplicación
Fuente: Elaboración propia con la herramienta MIRO

Como se observa en la Figura #9, el sector que se encuentra a la izquierda, pertenece a los servidores que son administrados por Tecnom, mientras que a la derecha son los servidores administrados por el personal de los clientes.

La aplicación desarrollada en este proyecto (DMSCC-UI), recibirá información en tiempo real de los CRM de Tecnom junto a sus *DMSCCONNECTORS* mediante peticiones REST a MUX.

MUX es una aplicación de administración desarrollada por Tecnom, que se encarga de dar altas y bajas de CRM, es decir, cada vez que aparece un nuevo cliente mediante MUX es que se le asigna un CRM. Adicionalmente, MUX brinda información de los dmsc-connectors, ya que a cada CRM eventualmente se le asigna un dmscconector para obtener los datos de los dms que se encuentran en la sección derecha de la Figura #9, cuyos datos son administrados por el cliente.

En cuanto a los *DMSCCONNECTORS*, pueden estar en los servers administrados por Tecnom, o en los servers del cliente, esto dependerá de las políticas aplicadas en cada caso.

1. Desarrollo API GraphQL:

Requisitos previos, poseer una cuenta en Amazon web services (https://aws.amazon.com/es/console/).

Tener instalado:

- Visual Studio Code
- Node.js
- Angular 8

Previo a desarrollar la API, se debe posicionar dentro del directorio de un proyecto de Angular (framework de frontend elegido para este proyecto) inicializado. Pero esta sección será explicada más adelante, ya que en este apartado se detallará la creación de la API desde línea de comandos como lo establece *Amplify*.

1. Instalación de dependencias mediante línea de comandos:

Dentro de una terminal CMD o desde la terminal que permite iniciar Visual Studio Code.

- Amplify CLI: **npm install -g @aws-amplify/cli**
- Librerías para angular: **npminstallaws-amplifyaws-amplify-angular**

2. Inicio de proyecto:

Como primer paso, se procede a configurar la cuenta personal de AWS ejecutando el comando amplify configure y automáticamente se abrirá una serie de pasos sencillos para configurar la región en la que se ubicarán los recursos producidos, además de la creación de un usuario para acceder dichos recursos, como se muestra en la Figura #10.

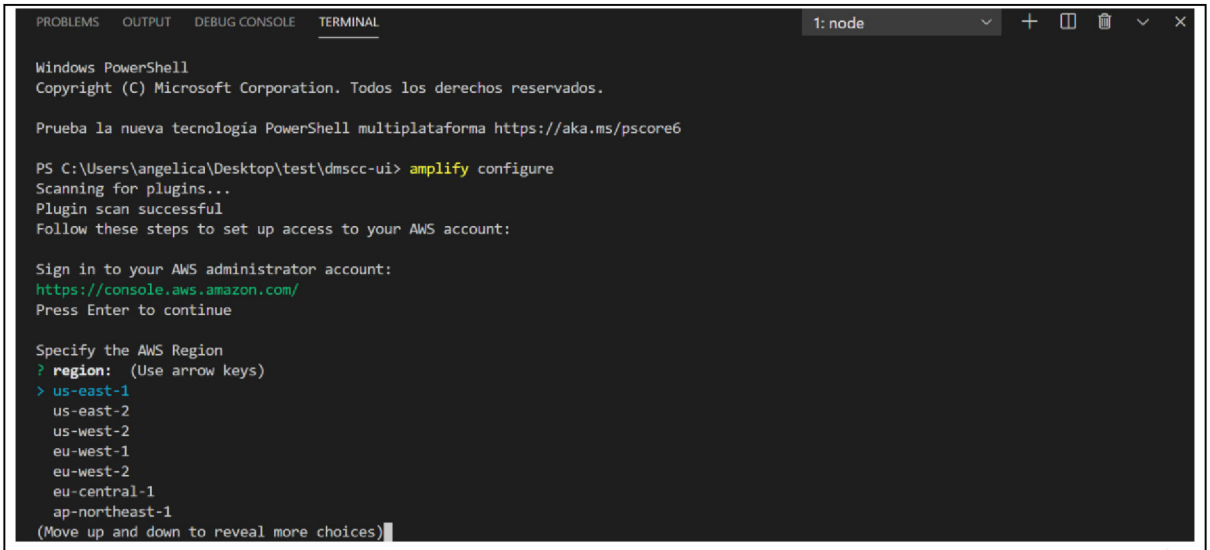


Figura #10: Ejemplo de configuración de cuenta de AWS.
Fuente: Elaboración propia

Dentro de la consola de administración, se debe crear el usuario y una clave de acceso para acceder sin necesidad de autenticarse cada vez que se crea o se accede a un recurso de la nube. A continuación, siguen las Figuras #11, #12, #13 y #14 donde se ejemplifica el procedimiento de creación.

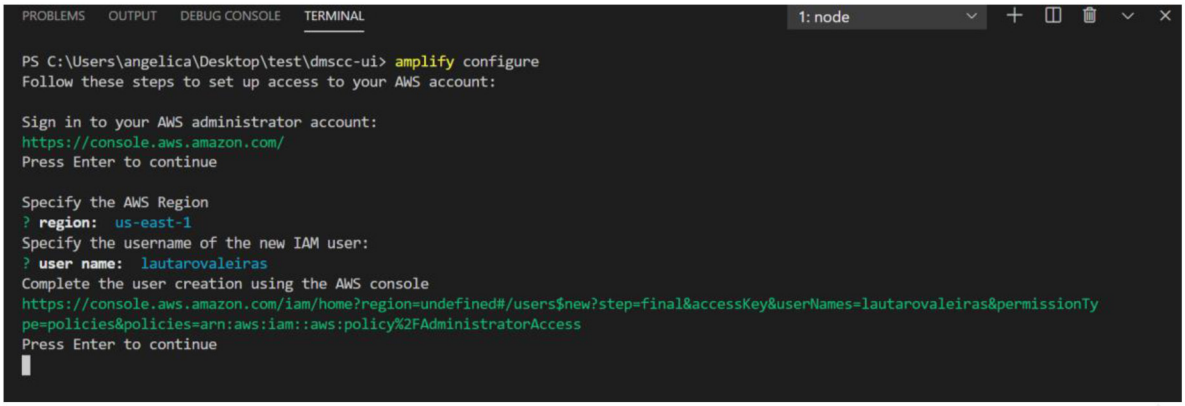


Figura #11: Configuración de cuenta AWS
Fuente: Elaboración propia

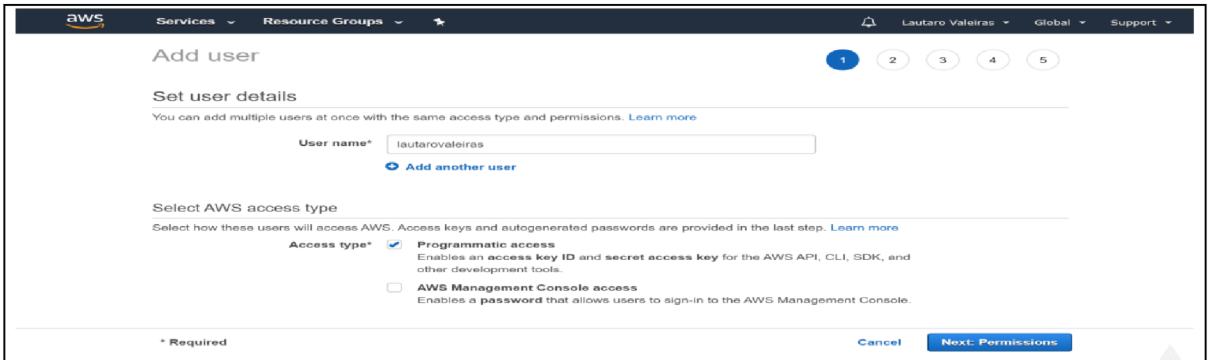


Figura #12: Configuración de cuenta AWS,continuación de figura #11

Fuente: Elaboración propia

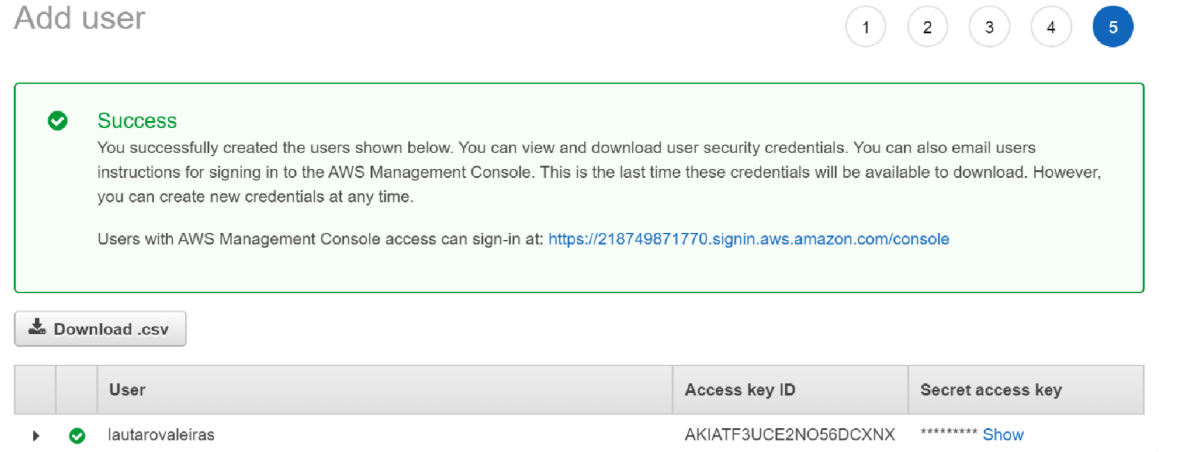


Figura #13: Ejemplo de la generación de credenciales

Fuente: Elaboración propia

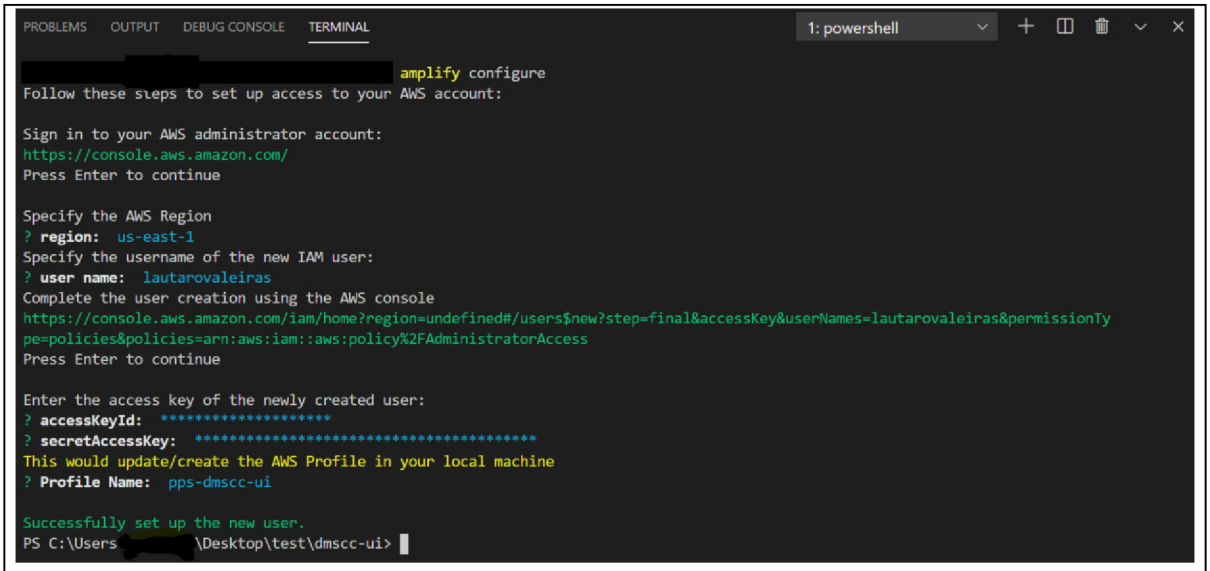


Figura #14: Insertando credenciales de usuario IAM AWS

Fuente: Elaboración propia

3. *Integración de* Amplify:

En esta instancia se procede a crear el proyecto con *Amplify CLI*, dando inicio a un ambiente de desarrollo en la nube donde posteriormente se añadirán los distintos recursos y el código de la aplicación.

Como primer paso se debe ejecutar el comando: "amplify init", para iniciar el proyecto, asignar un nombre, el environment de desarrollo, y el framework en el que se desarrollará el frontend, tal y como se ejemplifica en la Figura #15.

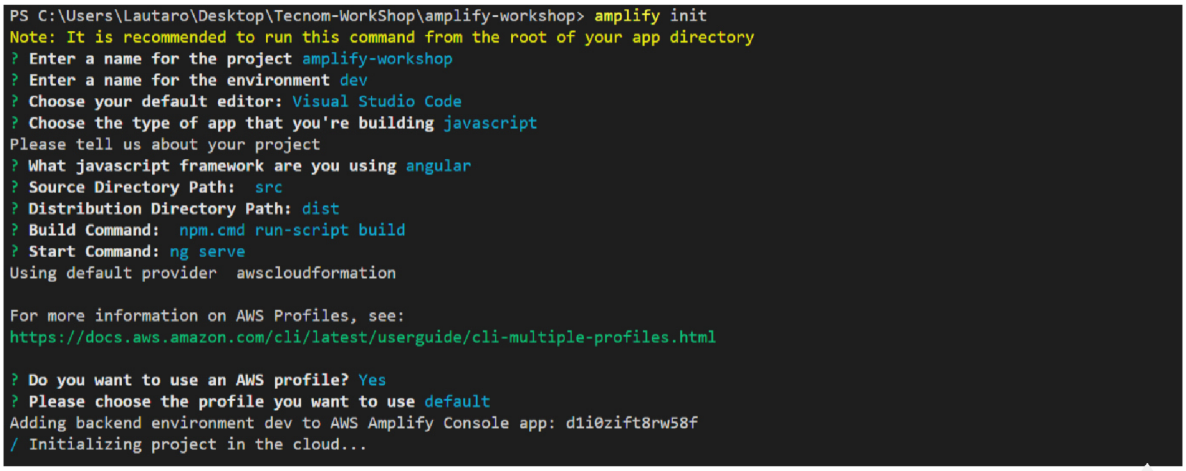


Figura #15: Ejemplo de creación de proyecto con "Amplify init"

Fuente: Elaboración propia

4. Categorías:

Amplify posee varias categorías (herramientas) que permiten que desde la línea de comandos se construya una aplicación completa y robusta.
En este proyecto se utilizarán las siguientes categorías:
-Api: Modelado de datos y mapeo de los mismos mediante Querys y Mutations a una base de datos no relacional (DynamoDB) por defecto.
- Auth: Métodos de autenticación para asegurar la API.
-Functions: Categoría que permite desarrollar *Lambdas* (explicado anteriormente en ***"2.1 - Investigación de tecnologías y definición de objetivos")*** desde el ambiente de desarrollo de *Amplify*, e invocarlas ante determinados eventos que así lo requieran.
-Scheduler: Esta categoría no es propia de *Amplify*, sino que se decidió crearla de forma manual, por determinados requerimientos que surgieron, los que serán abordados con posterioridad.

En esta sección se desarrollará la configuración y diseño la Api.

Pasos:

Ejecutar el comando amplify add api, donde se elige el tipo de API(*GraphQL*), un nombre con el cual se identificará en el grupo de recursos en la nube, el tipo de autenticación (ver en la sección "2.4 - Métodos de autenticación") y, por último, la definición del esquema de datos, como se ejemplifican en las Figuras #16 y #17 respectivamente. Cabe destacar, tal y como se muestra en la Figura #17, la configuración de *"Conflict detection"* la cual brinda una serie estrategias que se utilizarán ante posibles cambios de datos ya que *Appsync* permite realizar operaciones, aun cuando se pierde la conexión de internet.

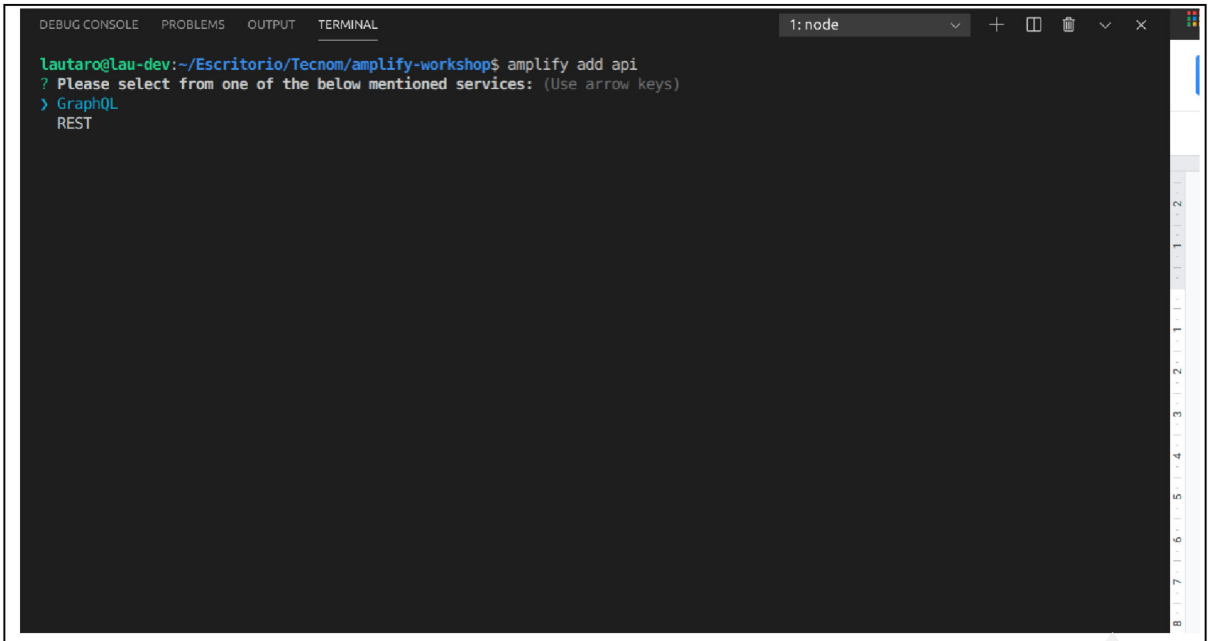


Figura #16: Elección del tipo de api

Fuente: Elaboración propia

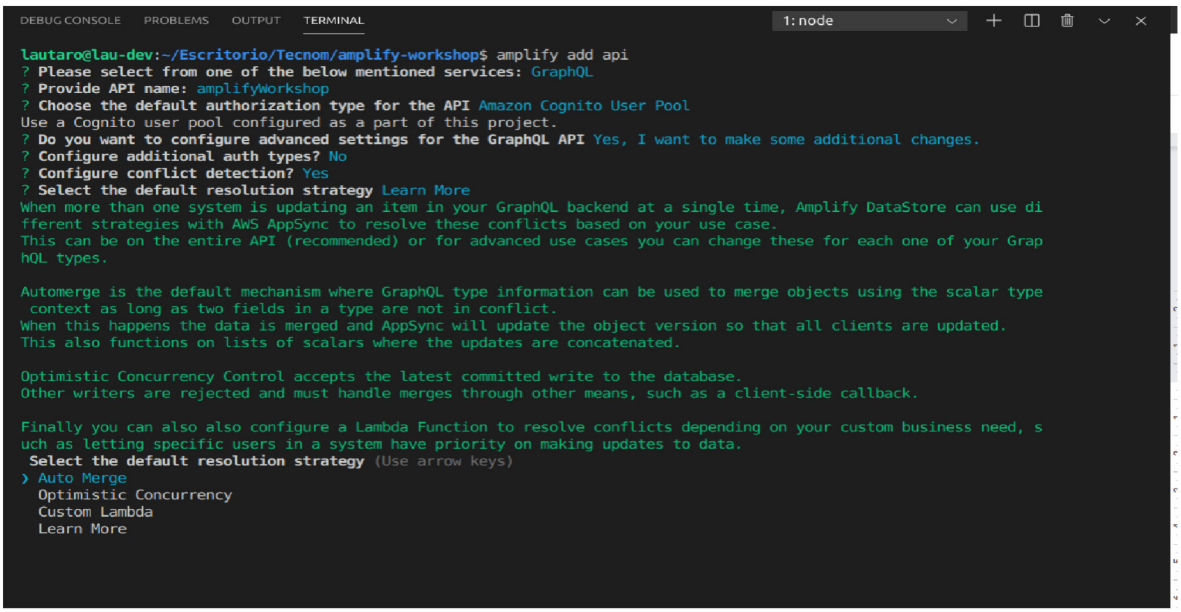


Figura #17: Conflict detection

Fuente: Elaboración propia.

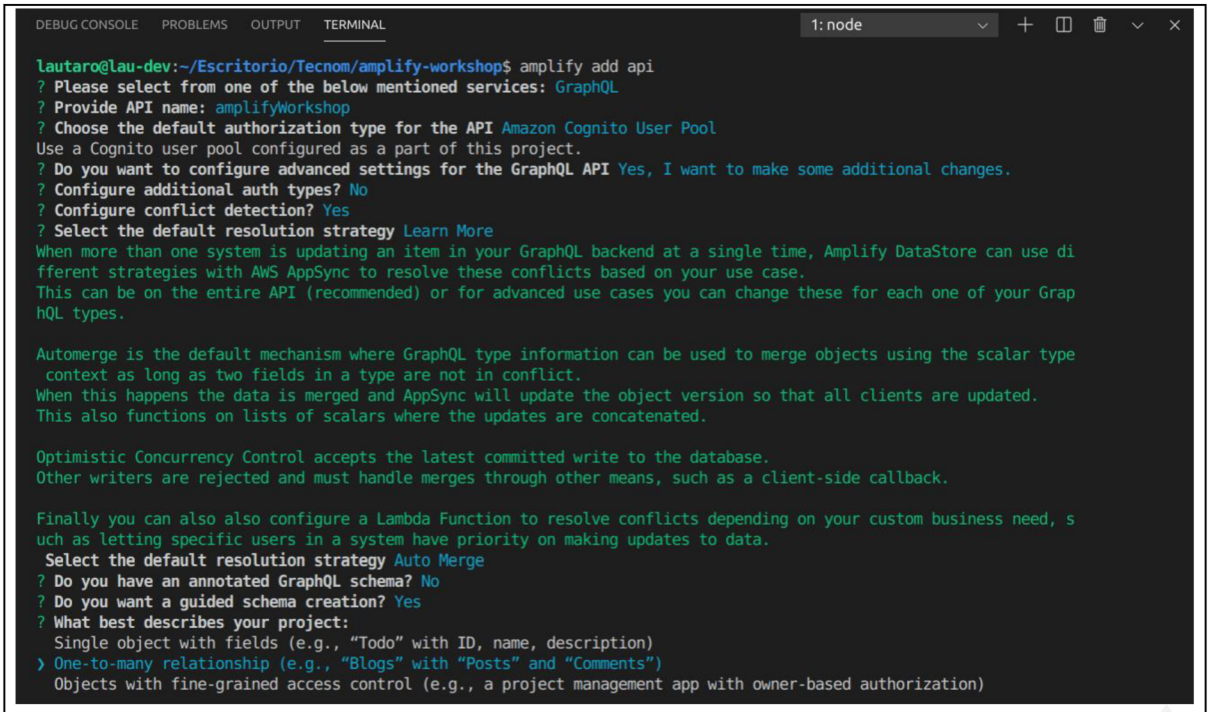


Figura #18: Selección de un esquema ejemplo

Fuente: Elaboración propia

En la siguiente captura, Figura #19, se observa una serie de ejemplos de esquemas de datos como base (ejemplo de blogs), es donde se selecciona uno al azar y, posteriormente, se editará el modelo personalizado.

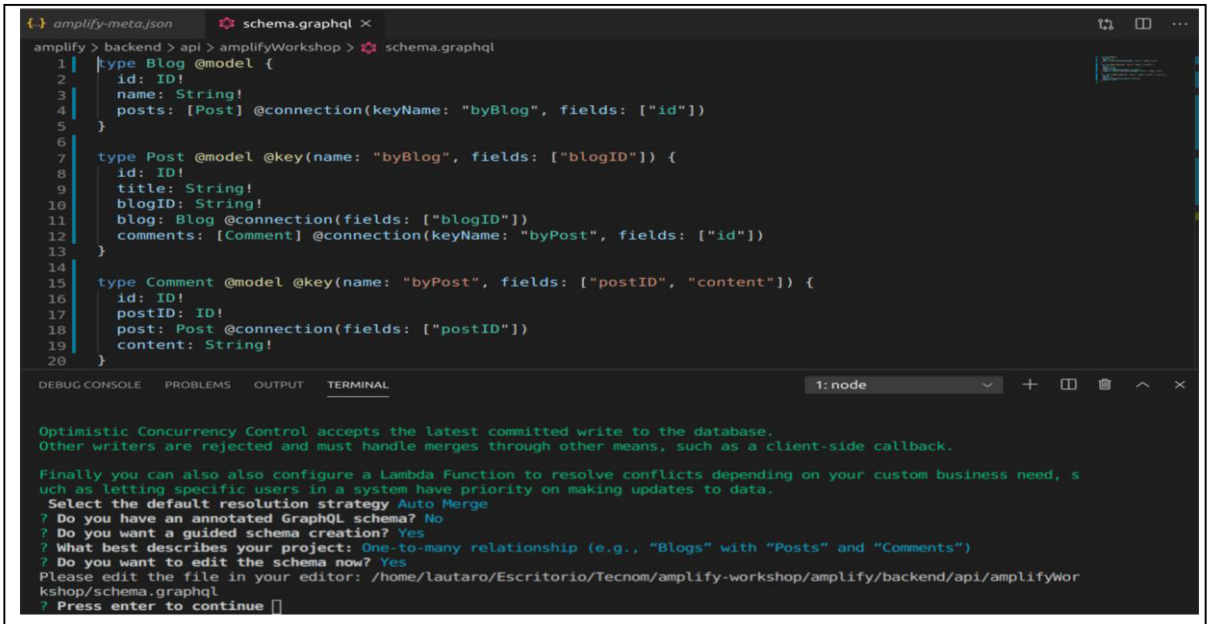


Figura #19: Esquema ejemplo de Post y Blogs

Fuente: Elaboración propia

Modelo DMSCC-UI:

Para la definición del presente modelo, se partió de la base de la definición de vistas, quedando como primer boceto el siguiente:

La entidad **App**, la cual guarda información de los CRM, tiene una relación de 1 → N con la entidad *DMSCCONNECTOR*, ya que un CRM puede tener conectores de test y los conectores productivos.

La entidad **DMSDB** hace referencia los sistemas dms de los clientes de Tecnom, que contienen los datos de los talleres y stock explicados en la *Introducción* del presente proyecto. Éstos a su vez poseenuna conexión de N → 1 con los *DMSCCONNECTOR*, ya que un sistema DMS puede llegar a tener muchos *DMSCCONNECTOR*que lean su información para notificar al CRM(App) y una conexión 1 → N con la entidad Server, ya que un server puede tener muchos sistemas DMSDB.

La entidad *DMSCCONNECTOR*, la más importante de este modelo, posee toda la información de vital importancia para el área de infraestructura. Esta entidad establece relaciones 1 → N con las entidades App, DMSDB y Server.

Por último, la entidad Server posee información de los conectores o sistemas dms alojados, ya que éste puede tener un *DMSCCONNECTOR* o un dmsdb en el mismo servidor, como bien se explicó en la Arquitectura del sistema. Los resultados parciales observados hasta el momento se muestran en las Figuras #20, #21 y #22.

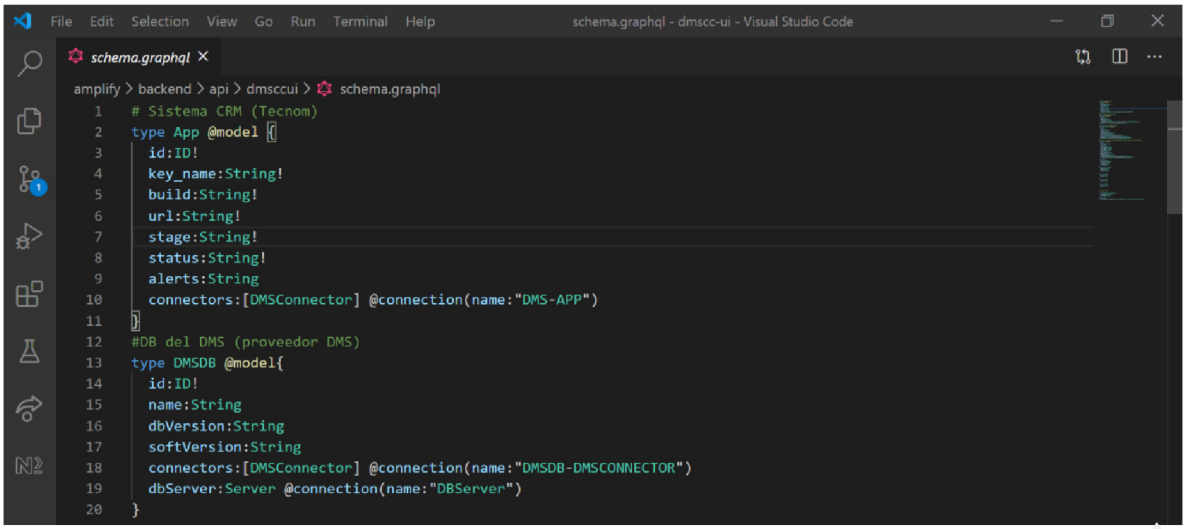


Figura #20: Esquema dmscc-ui
Fuente: Elaboración propia

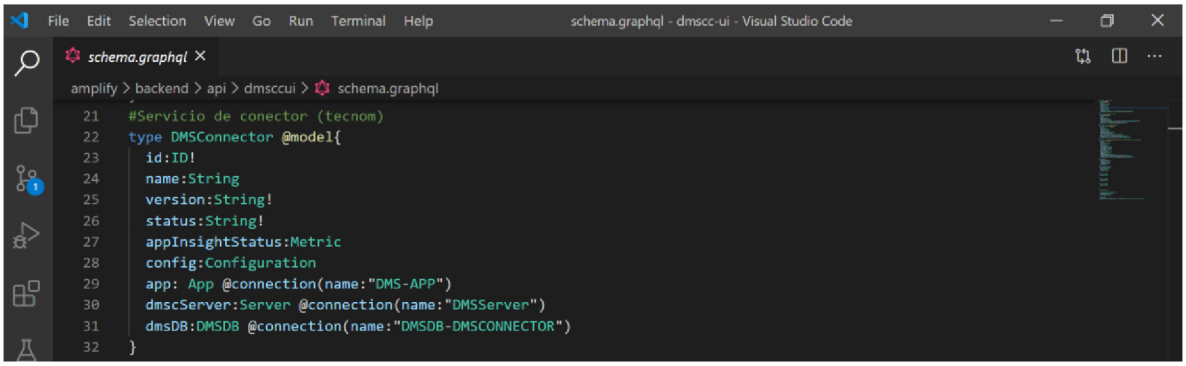


Figura #21: Esquema dmscc-ui, continuación de figura #20
Fuente: Elaboración propia

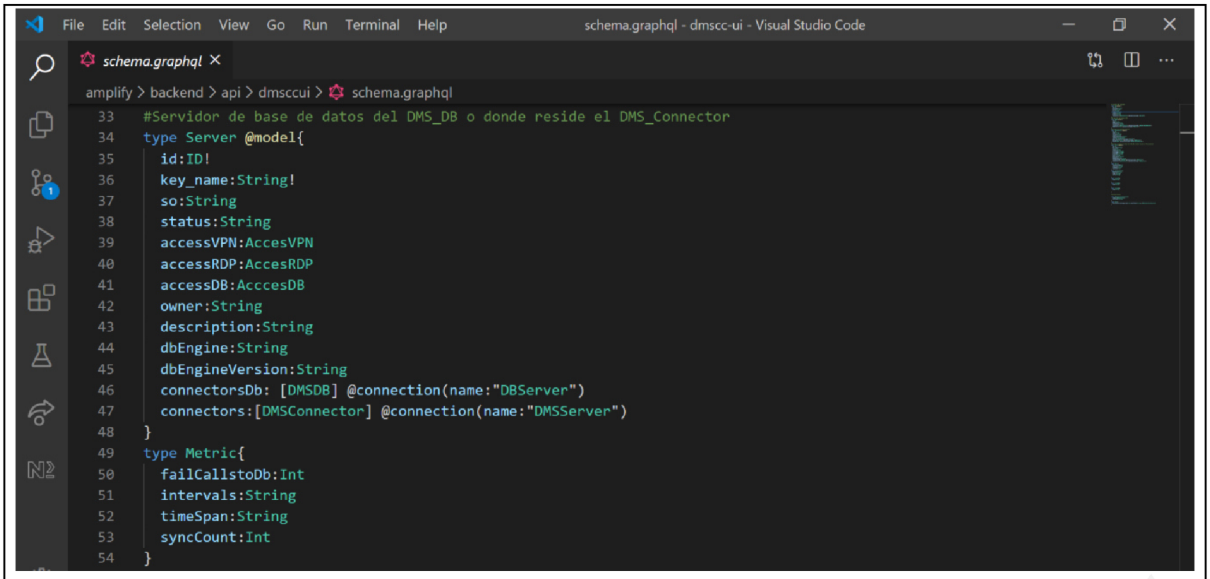


Figura #22: Esquema dmscc-ui, continuación de figura #21

Fuente: Elaboración propia

Conceptos clave:

Cada entidad que se quiere guardar en la base de datos (en este caso DynamoDB), debe llevar la notación **@model**, esto, al igual que un ORM tradicional, mapea la entidad con una tabla que es creada en la base de datos junto a los atributos definidos. Por otro lado, se entiende que, una de las características de una base de datos no relacional es que posee una única tabla con todos los datos y conexiones, con alta redundancia. Sin embargo, *Amplify* al momento de crear la base de datos omite esta característica y crea una tabla por entidad, es decir, al estilo de las bases de datos relacionales, donde las conexiones entre tablas las maneja mediante índices(al estilo fk en relacionales).

Relaciones:

Para las relaciones entre tablas se utiliza la notación **@connection** y esto guarda el id con la referencia a la tabla. Luego al realizar las consultas, GraphQL se encarga de obtener el objeto referenciado.

Adicionalmente, cuando se quiere guardar un objeto dentro de un atributo, en lugar de un tipo de dato primitivo (String, Int, Boolean), se declara dicho objeto, al igual que una entidad, con el término “type”, pero éste no debe llevar la notación **@model** (ya que de otro modo se crearía una tabla en la base datos)y luego se

le indicaría al tipo de atributo con el nombre que se declaró , como se ejemplifica en las Figuras #22 y #23 .

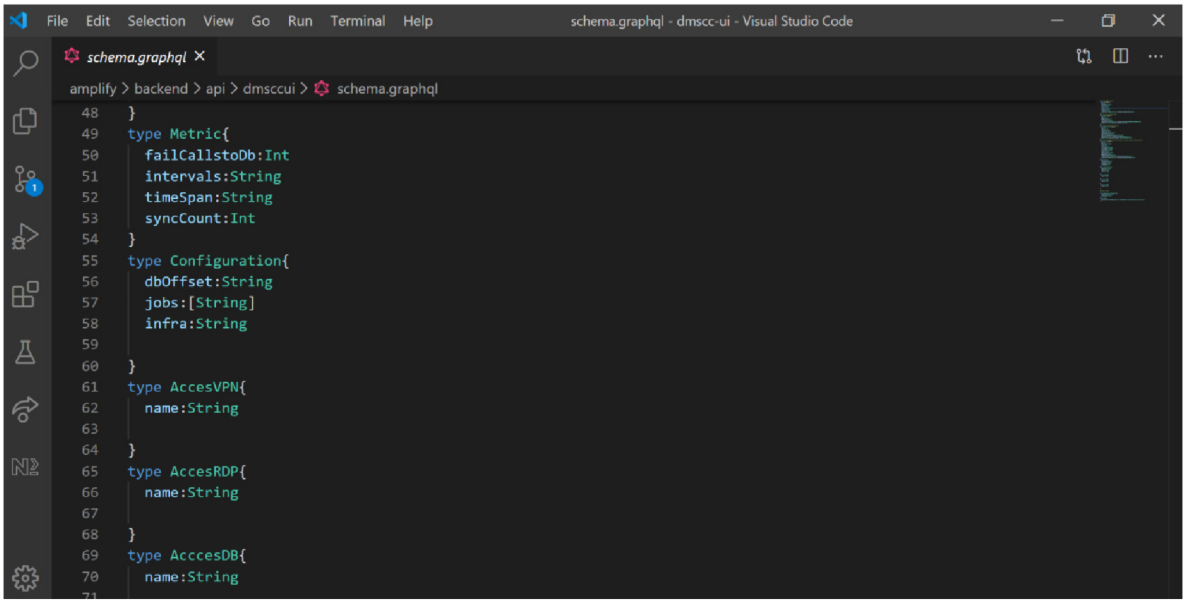


Figura #23: Ejemplo de definición de objetos

Fuente: Elaboración propia

Una vez definido el esquema de la aplicación, se procede a ejecutar el comando amplify push, el cual a partir de éste creará las tablas en la base de datos, junto a las Querys, los Mutations y las Subscriptions para el acceso a los mismos mediante un endpoint y las configuraciones de autenticación en el archivo aws-exports.js.

A continuación, se muestran los ejemplos de la *ejecución de "Amplify push"*, en la Figura #24 se observa el endpoint GraphQL, en la Figura #25 el resultado de las tablas creadas en la base de datos, en la #26 las aplicaciones de *Amplify* administradas actualmente ("dmscc-admin" es la correspondiente al presente proyecto) y, por último, en la Figura #27 se detallan las relaciones entre las tablas mediante los índices.

```
GraphQL endpoint: https://qt4az2hg7few5lk7htrwqhhqme.appsync-api.us-east-1.amazonaws.com/graphql
Hosted UI Endpoint: https://dmsccadmin-test.auth.us-east-1.amazoncognito.com/
Test Your Hosted UI Endpoint: https://dmsccadmin-test.auth.us-east-1.amazoncognito.com/login?response_type=code&client_id=71hl7hj585hqova4disjkcimhb&redirect_uri=http://localhost:4200/
```

Figura #24: Resultado de la ejecución de "Amplify push"

Fuente: Elaboración propia

Resultado en la base de datos:

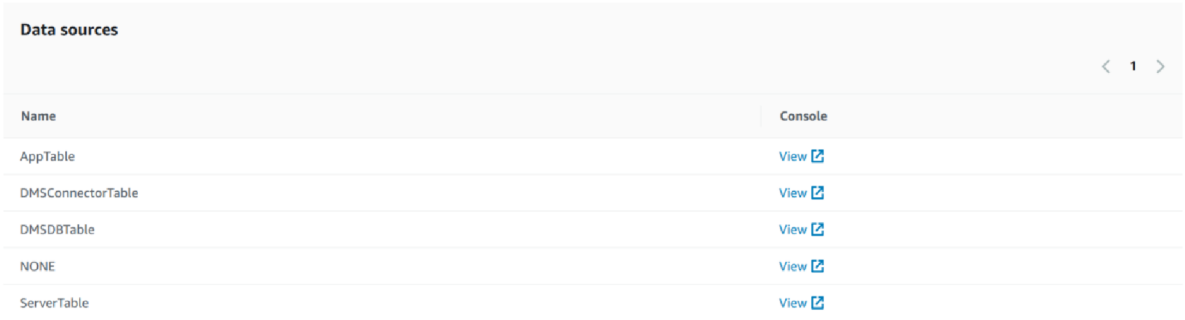


Figura #25: Tablas generadas por Amplify en DynamoDB
Fuente: Elaboración propia

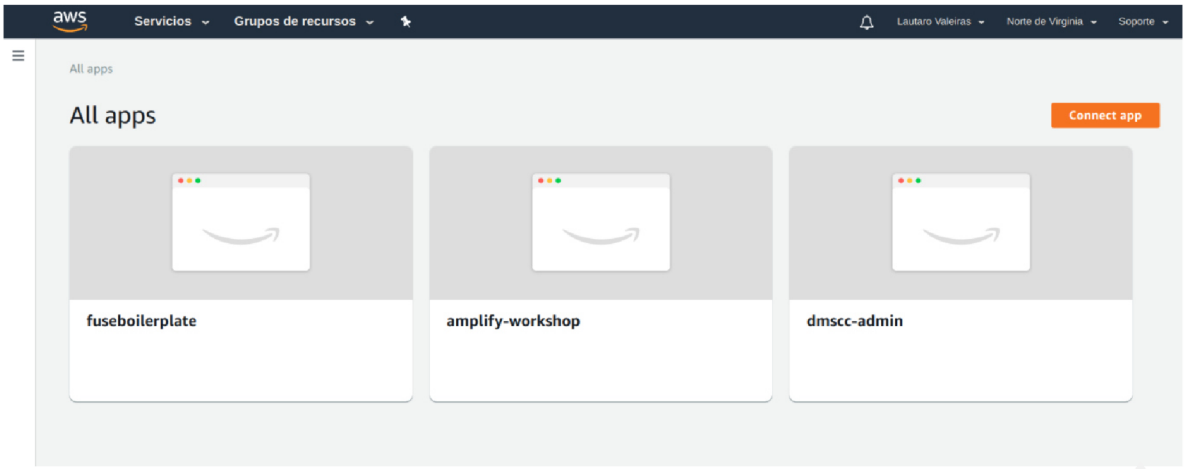


Figura #26: Ejemplo de administración de apps en aws
Fuente: Elaboración propia

Índices:

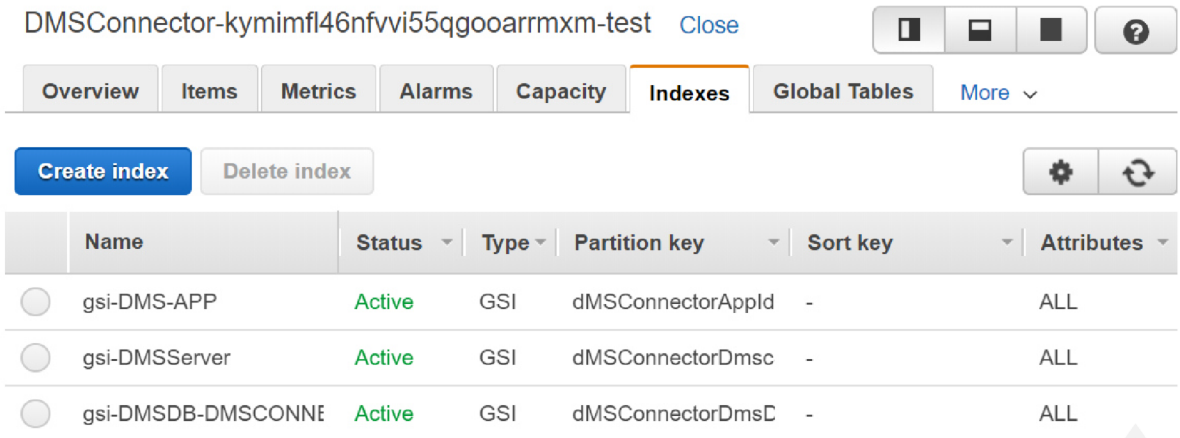


Figura #27: Índices generados a partir del esquema realizado previamente
Fuente: Elaboración propia

Resultado de Queries, Mutations y Subscriptions generados por *Amplify* de forma automática, según se muestra en la Figura #28.

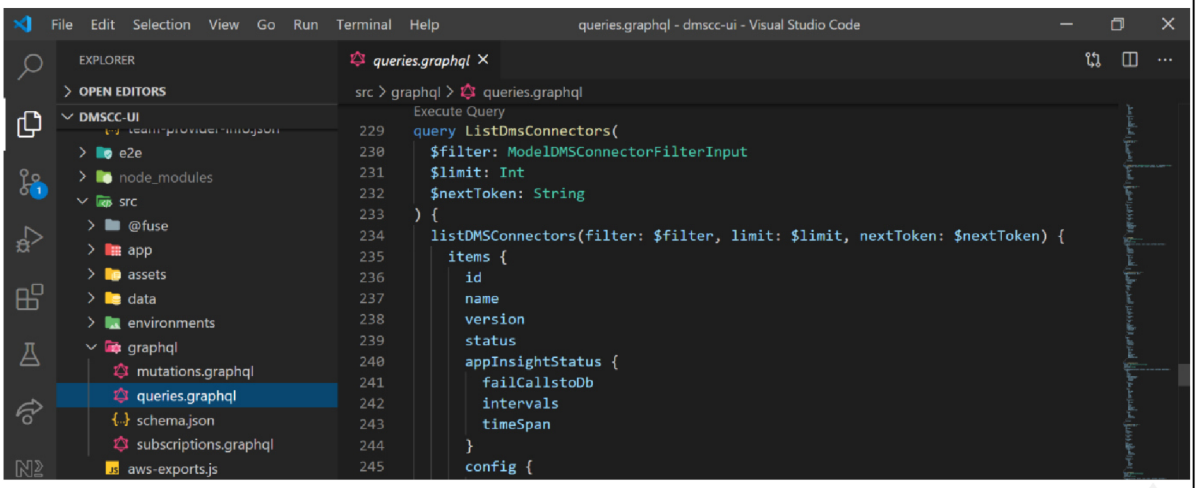


Figura #28: Ejemplo de Query generada por Amplify
Fuente: Elaboración propia

Adicionalmente, *Amplify* genera un archivo llamado **aws-exports.js**, el cual contiene la información necesaria para acceder a la API (endpoint, credenciales para la autenticación). Ejemplo en la Figura #29.

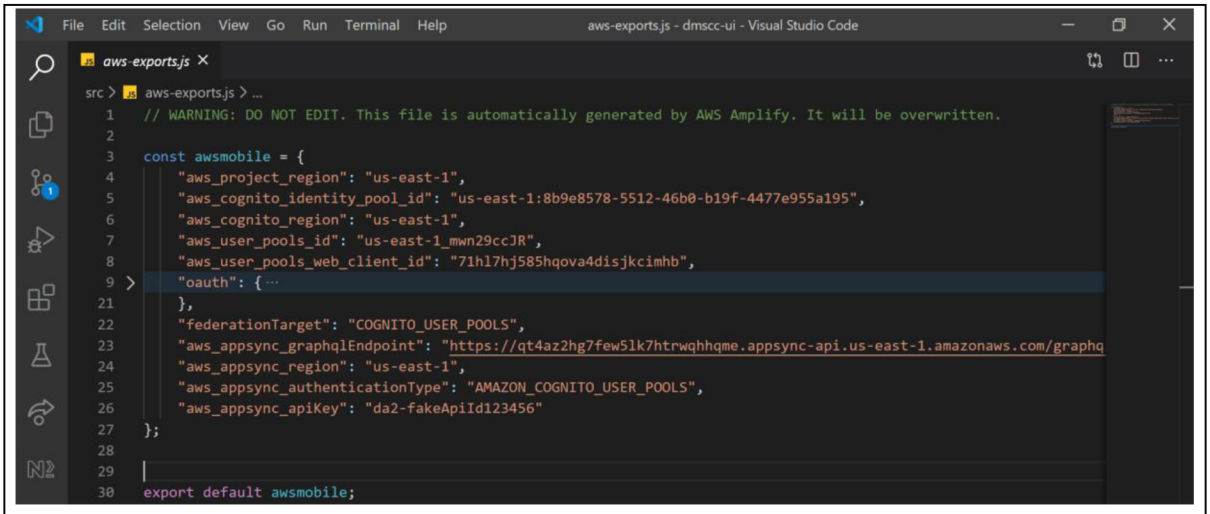


Figura #29: Archivo aws-exports.js generado por Amplify

Fuente: Elaboración propia

A continuación, en la Figura #30 se pueden observar los directorios creados por *Amplify* en el proyecto, donde se detallan categorías agregadas.

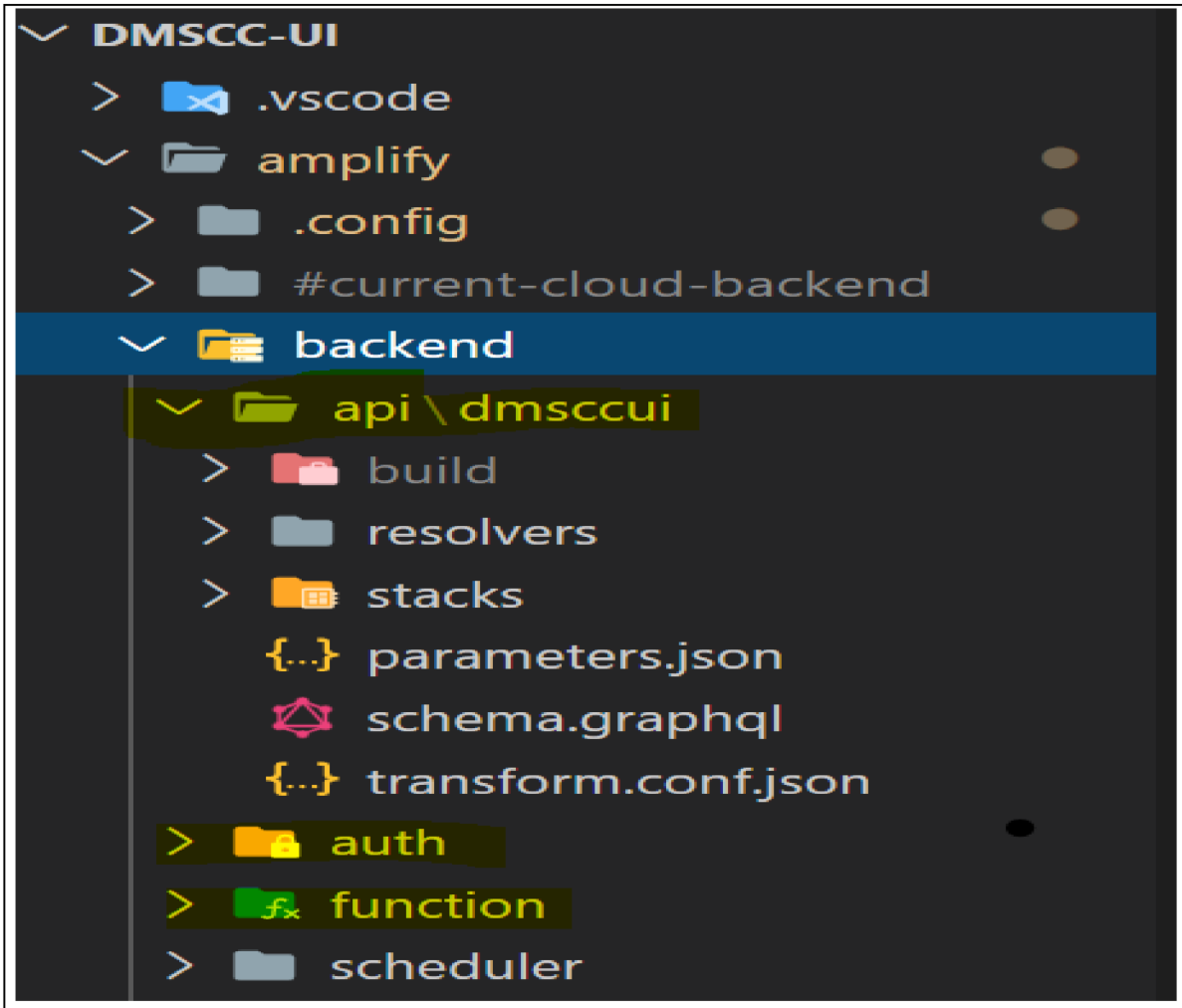


Figura #30: Carpeta de Amplify con todas sus configuraciones
Fuente: Elaboración propia

4. Métodos de autenticación

En esta sección, se procede a configurar la securización de la API, mediante la categoría de *Amplify* **Auth.**

Amplify brinda la posibilidad de asegurar una API de diferentes maneras:

-API-key: genera una clave pública con la cual se permite acceder a los recursos que se definen como públicos, mediante el endpoint generado.

-CognitoUser Pools: crea un grupo de usuarios, los que tendrán acceso a los recursos de la aplicación, mediante verificación de correo o teléfono.

-CognitoUser Pools Federated Identities: al igual que CognitoUser Pools, crea grupo de usuarios, pero permitiendo darse de alta con servicios terceros, como puede ser Google o Facebook.

Para DMSCC-UI se decidió por el método de autenticación "Cognito user pools federated identities", ya que mediante éste sus usuarios podrán darse de alta de manera fácil y rápida mediante sus correos de Google de la empresa (@tecnom.com.ar).

Adicionalmente, el flujo de trabajo al crear la autenticación, permite crear funciones de filtrado previo al darse de alta el usuario.

En el presente proyecto se utilizará un filtrado Whitelist, donde permita únicamente dar alta usuarios que posean el dominio de "@tecnom.com.ar", mediante cuentas de Google.

Pasos:

Ejecutar amplify add auth tal y como se observa en la figura #31, lo que lleva automáticamente a una serie de sencillas configuraciones tales como: tipo de autenticación (Figura #32, se escoge "email") y el ya mencionado filtrado Whitelist, observados en los ejemplos de las Figuras #33 y #34.

```
PS C:\Users\Lautaro\Desktop\Tecnom-WorkShop\amplify-workshop> amplify add auth
Using service: Cognito, provided by: awscloudformation

 The current configured provider is Amazon Cognito.

 Do you want to use the default authentication and security configuration?
  Default configuration
> Default configuration with Social Provider (Federation)
  Manual configuration
  I want to learn more.
```

Figura #31: Ejecución del comando Amplify add auth,
Fuente: Elaboración propia

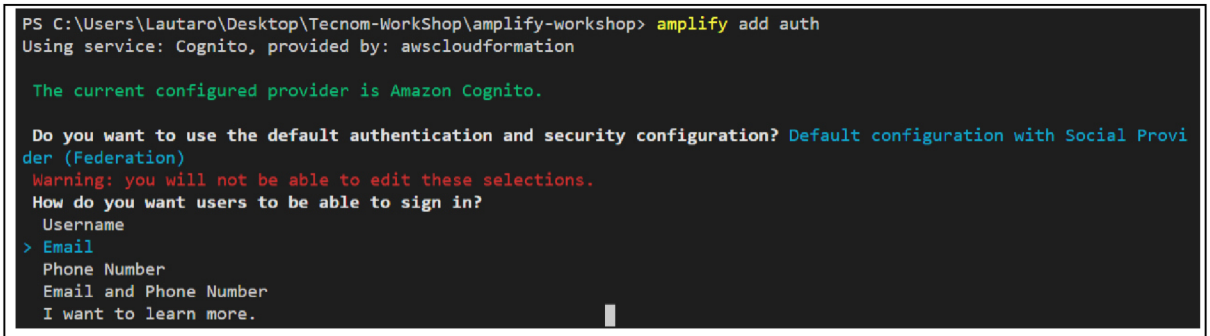


Figura #32: Configuración de auth, continuación de figura # 31

Fuente: Elaboración propia

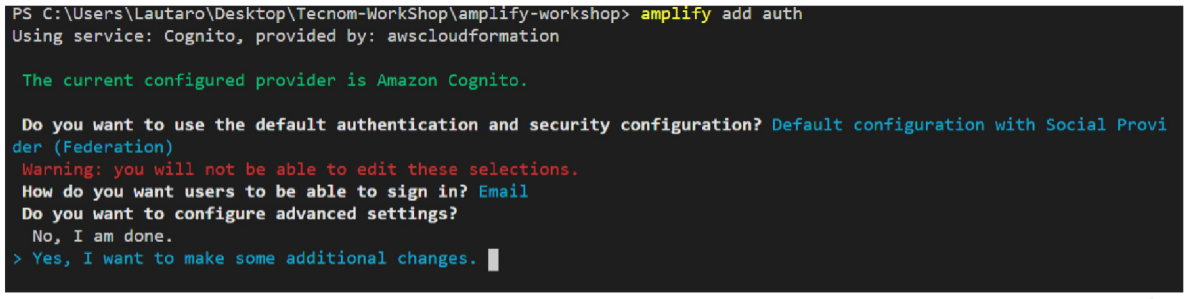


Figura #33: Configuración de auth, continuación de figura #32

Fuente: Elaboración propia

Como se ha expresado previamente, en las Figuras #34,#35,#36 y #37 se ejemplifican las operaciones realizadas con *Amplify* CLI para configurar el filtrado de alta de usuarios:

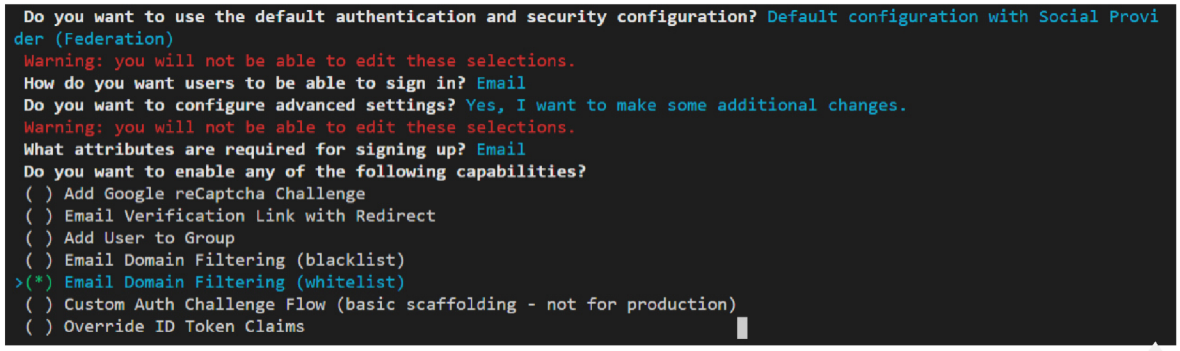


Figura #34: Configuración de auth, filtrado de usuarios

Fuente: Elaboración propia

```
Do you want to use the default authentication and security configuration? Default configuration with Social Provi
der (Federation)
Warning: you will not be able to edit these selections.
How do you want users to be able to sign in? Email
Do you want to configure advanced settings? Yes, I want to make some additional changes.
Warning: you will not be able to edit these selections.
What attributes are required for signing up? Email
Do you want to enable any of the following capabilities? Email Domain Filtering (whitelist)
What domain name prefix you want us to create for you? tecnom-validator-email
```

Figura #35: Configuración de auth, filtrado de usuarios, continuación de figura #34

Fuente: Elaboración propia

```
Do you want to use the default authentication and security configuration? Default configuration with Social Provi
der (Federation)
Warning: you will not be able to edit these selections.
How do you want users to be able to sign in? Email
Do you want to configure advanced settings? Yes, I want to make some additional changes.
Warning: you will not be able to edit these selections.
What attributes are required for signing up? Email
Do you want to enable any of the following capabilities? Email Domain Filtering (whitelist)
What domain name prefix you want us to create for you? tecnom-validator-email
Enter your redirect signin URI: http://localhost:4200/
```

Figura #36: Configuración de auth, redirección al terminar el registro

Fuente: Elaboración propia

```
Warning: you will not be able to edit these selections.
How do you want users to be able to sign in? Email
Do you want to configure advanced settings? Yes, I want to make some additional changes.
Warning: you will not be able to edit these selections.
What attributes are required for signing up? Email
Do you want to enable any of the following capabilities? Email Domain Filtering (whitelist)
What domain name prefix you want us to create for you? tecnom-validator-email
Enter your redirect signin URI: http://localhost:4200/
? Do you want to add another redirect signin URI No
Enter your redirect signout URI: http://localhost:4200/
? Do you want to add another redirect signout URI No
Select the social providers you want to configure for your user pool:
( ) Facebook
>(*) Google
( ) Login With Amazon
```

Figura #37: Configuración de auth, elección del tipo de cuenta a registrar

Fuente: Elaboración propia

Posteriormente al paso que se menciona en la Figura #37 es necesario configurar, en la consola para desarrolladores que brinda Google (https://console.developers.google.com),el alta de una nueva aplicación. En el presente caso, DMSCC-UI, y así genera las credenciales que permite a la aplicación dar de alta usuarios con cuentas de Google válidas, como se detallan en las Figuras #38 y #39.

Luego, las claves generadas con la consola de Google, son insertadas en la aplicación de este proyecto tal y como se ejemplifica en las Figuras #40 y #41.

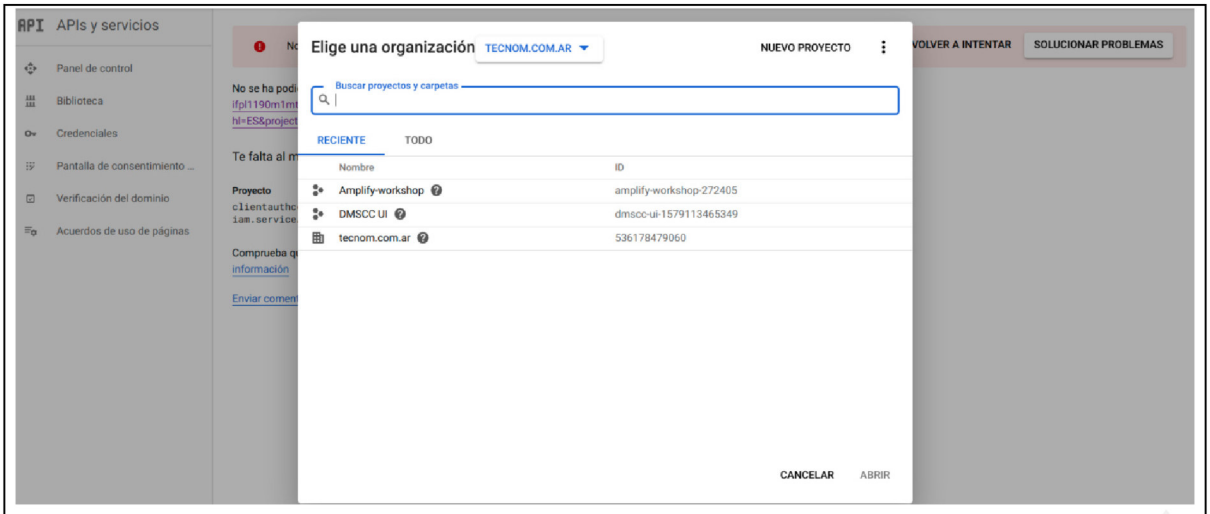


Figura #38: Creación de proyecto en Google DMSCC-UI

Fuente: Elaboración propia

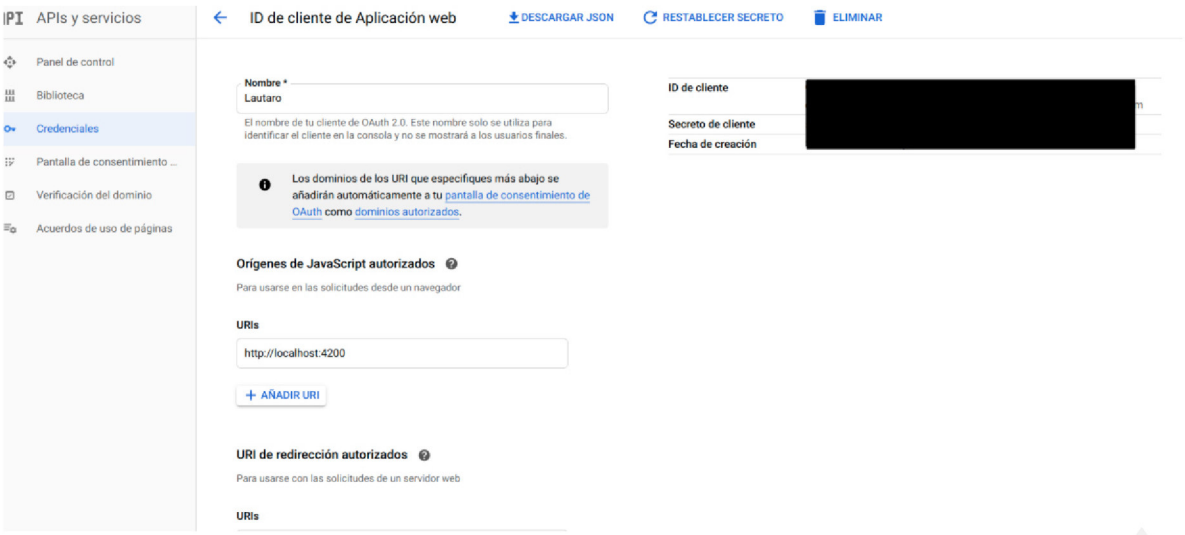


Figura #39: Credenciales generadas en Google para DMSCC-UI.

Fuente: Elaboración propia

```
How do you want users to be able to sign in? Email
Do you want to configure advanced settings? Yes, I want to make some additional changes.
Warning: you will not be able to edit these selections.
What attributes are required for signing up? Email
Do you want to enable any of the following capabilities? Email Domain Filtering (whitelist)
What domain name prefix you want us to create for you? tecnom-validator-email
Enter your redirect signin URI: http://localhost:4200/
? Do you want to add another redirect signin URI No
Enter your redirect signout URI: http://localhost:4200/
? Do you want to add another redirect signout URI No
Select the social providers you want to configure for your user pool: Google

You've opted to allow users to authenticate via Google.  If you haven't already, you'll need to go to https://dev
elopers.google.com/identity and create an App ID.

Enter your Google Web Client ID for your OAuth flow:
```

Figura #40: Inserción de las credenciales generadas como muestra la figura 38
Fuente: Elaboración propia

```
Do you want to enable any of the following capabilities? Email Domain Filtering (whitelist)
What domain name prefix you want us to create for you? tecnom-validator-email
Enter your redirect signin URI: http://localhost:4200/
? Do you want to add another redirect signin URI No
Enter your redirect signout URI: http://localhost:4200/
? Do you want to add another redirect signout URI No
Select the social providers you want to configure for your user pool: Google

You've opted to allow users to authenticate via Google.  If you haven't already, you'll need to go to https://dev
elopers.google.com/identity and create an App ID.

Enter your Google Web Client ID for your OAuth flow:  937007592057-bak2q6rknpblvrie6bvqohakcc4kgvis.apps.googleus
ercontent.com
Enter your Google Web Client Secret for your OAuth flow:  ur9PMyAYhNssTYgcMpcPpDhx
```

Figura #41: Inserción de las credenciales generadas como muestra la figura #40
Fuente: Elaboración propia

```
Do you want to enable any of the following capabilities? Email Domain Filtering (whitelist)
What domain name prefix you want us to create for you? tecnom-validator-email
Enter your redirect signin URI: http://localhost:4200/
? Do you want to add another redirect signin URI No
Enter your redirect signout URI: http://localhost:4200/
? Do you want to add another redirect signout URI No
Select the social providers you want to configure for your user pool: Google

You've opted to allow users to authenticate via Google.  If you haven't already, you'll need to go to https://developer
s.google.com/identity and create an App ID.

Enter your Google Web Client ID for your OAuth flow:  937007592057-bak2q6rknpblvrie6bvqohakcc4kgvis.apps.googleusercont
ent.com
? Enter a comma-delimited list of allowed email domains (example: 'mydomain.com, myotherdomain.com'). tecnom.com.ar
Succesfully added the Lambda function locally
```

Figura #42: Definición de dominio permitido para ingresar a dmscc-ui (tecnom.com.ar)
Fuente: Elaboración propia

Una vez finalizado el flujo de trabajo, se procede a ejecutar el comando "A*mplify*push" nuevamente, para subir y actualizar las configuraciones establecidas.

5. Listado, ABM y filtrado frontend

En esta sección se procede a configurar el framework de front-end elegido (Angular 8) y el diseño de las vistas a mostrar.

Como boceto inicial, se partió desde una maqueta predefinida con los diseños, logos y tipados que utiliza la organización para sus productos.

Adicionalmente, se utilizaron las librerías de "Angular materials" (https://material.angular.io/) para realizar las tablas donde se mostrarán los datos, los filtros para realizar la búsqueda de los mismos y los Forms para los ABM (alta, baja, modificación).

Previo al diseño de las vistas, se procedió a modificar ciertos archivos del proyecto de Angular, para permitir a A*mplify* instalar y configurar sus dependencias en el proyecto, tales como las consultas (Querys, Mutations, Subscriptions) y posibilitar la autenticación a la Api *GraphQL* mediante el archivo aws-exports.js (generado automáticamente por A*mplify*).

A continuación, se indican los archivos a modificar, tal y como se ejemplifican en las Figuras #42, #43 , #44 y #45:

- main.ts
- login.component.ts
- auth.service.ts

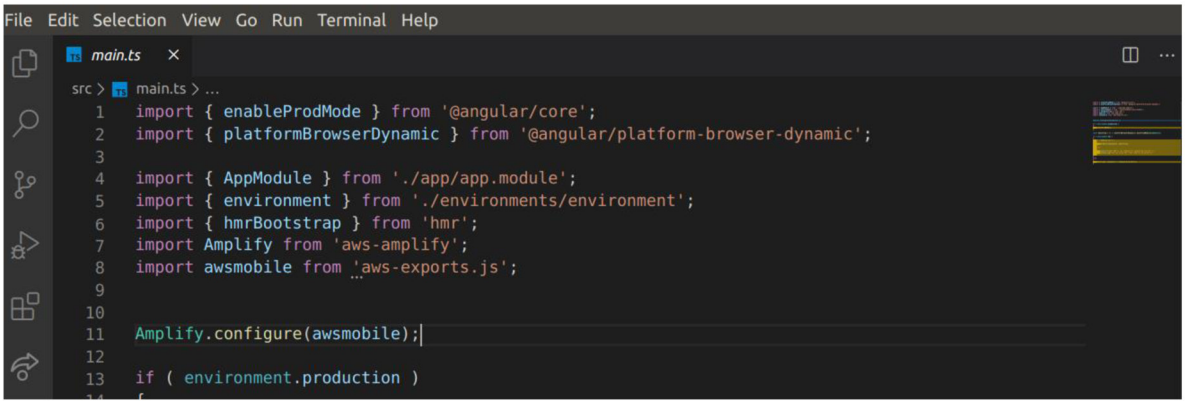


Figura #43: Archivo main.ts
Fuente: Elaboración propia

```
File  Edit  Selection  View  Go  Run  Terminal  Help
login.component.ts
src > app > auth > login > login.component.ts > ...
56
57    /**
58     * On init
59     */
60    ngOnInit(): void {
61      this.loginForm = this._formBuilder.group({
62        email: ['', [Validators.required, Validators.email]],
63        password: ['', Validators.required]
64      });
65    }
66
67    get emailInput(): any { return this.loginForm.get('email'); }
68    get passwordInput(): any { return this.loginForm.get('password'); }
69
70 >  signIn(): void {…
94    }
95
96    async signInGoogle(){
97      const socialResult = await this.auth.socialSignIn(AuthService.GOOGLE);
98
99
```

Figura #44: Archivo login.component.ts

Fuente: Elaboración propia

```
auth.service.ts
src > app > auth > auth.service.ts > AuthService > socialSignIn > provider
15  @Injectable({
16    providedIn: 'root'
17  })
18  export class AuthService {
19    public loggedIn: boolean;
20    private _authState: Subject<CognitoUser | any> = new Subject<CognitoUser | any>();
21    authState: Observable<CognitoUser | any> = this._authState.asObservable();
22
23    public static SIGN_IN = 'signIn';
24    public static SIGN_OUT = 'signOut';
25    public static FACEBOOK = CognitoHostedUIIdentityProvider.Facebook;
26    public static GOOGLE = CognitoHostedUIIdentityProvider.Google;
27
28    constructor() {
29      Hub.listen('auth', data => {
30        const { channel, payload } = data;
31        if (channel === 'auth') {
32          this._authState.next(payload.event);
33        }
34      });
35    }
36    socialSignIn(provider: CognitoHostedUIIdentityProvider): Promise<ICredentials> {
37      return Auth.federatedSignIn({
38        provider: provider
39      });
```

Figura #45: Archivo auth.service.ts

Fuente: Elaboración propia

1. Vista de LOGIN

Una vez finalizada la configuración inicial, como segundo paso es establecer el login, el cual ya viene predefinido en la plantilla brindada por la organización y que contiene las siguientes vistas, como se representa en las Figuras #46 y #47.

-Vista de Login

-Vista de Home

-Vista de Barra de Nav simple

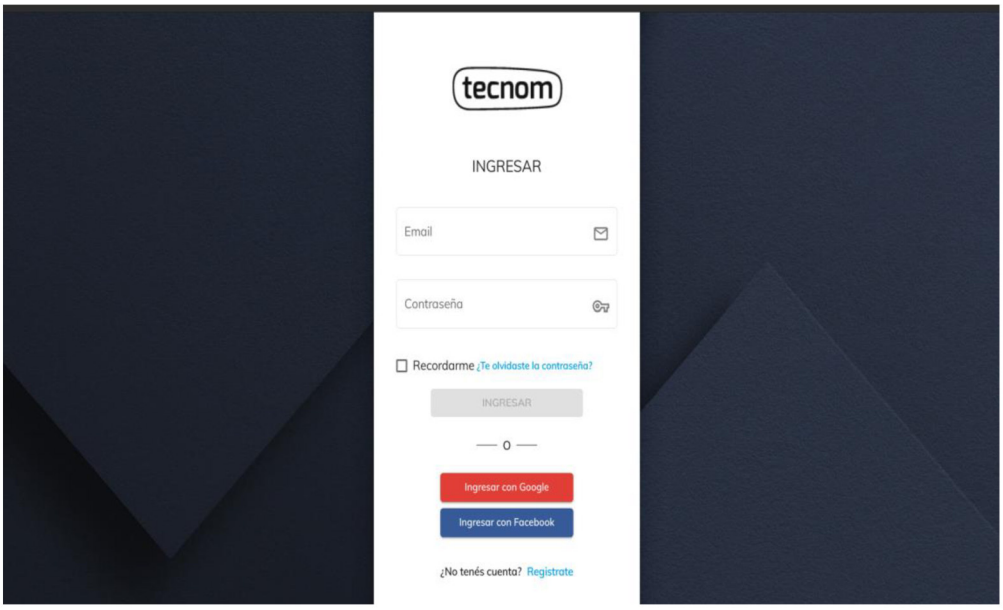


Figura #46: Vista de loginTecnom

Fuente: Elaboración propia

Figura #47: Vista de home Tecnom junto a la barra de navegación.

Fuente: Elaboración propia

2. Vista de CRM

Partiendo de la plantilla mostrada en las Figuras #46 y #47, se procede a diseñar la tabla que contendrá el listado de los *DMSCCONNECTORS* y los CRM. Para esto, el sector de infraestructura de la compañía brindó un listado de clientes en formato JSON, con datos reales, para realizar las pruebas pertinentes en el frontend.

clientes.js

src > data > clientes.js > <unknown> > "Web Server"

```
module.exports = [
    {
        "Client Key Name": "aback",
        "Sistema Nombre": "Oeste Autos Aback",
        "Website": "www.oesteautos.com.ar",
        "Business Name": "ABACK S.A. (Oeste Autos)",
        "Sistema Version": 1746,
        "Sistema Marcas": [
            "ford"
        ],
        "Sistema Mercado": "Autos",
        "Cliente Fecha Alta": "2016-02-10T14:53:00Z",
        "Fecha Ingreso Produccion": "2016-06-14T13:29:14.02Z",
        "Db Name": "CRMAbackDBAux",
        "Db Server": "nubuxdb2",
        "Log File Size Gb": 0.5,
        "Data File Size Gb": 1.2,
        "Db Fecha Alta": "2016-06-14T13:29:14.02Z",
        "Sitio Prendido": "FALSE",
        "Url": "https://aback.tecnomcrm.com/",
        "Web Server": "topolino",
        "Build Id": 1746.27,
        "Version Id": 1746
    },
```

Figura #48: Datos de prueba
Fuente: Elaboración propia

Con los datos de prueba, se procede a realizar la tabla con la librería Angular Materials (Figura #49) la que, desde su página web, brinda una serie de ejemplos con paginado y filtrado de datos inyectados de forma dinámica en tablas, y con los estilos ya configurados. Para esto fue necesario seguir los pasos y de manera muy simple y rápida se pudo replicar la lógica para el esquema creado.

A continuación, en la Figura #50se mostrará la vista de CRM, la que contendrá un listado con todas las aplicaciones de Tecnom que se encuentran en producción, con los datos de prueba que fueron brindados. Luego, esta misma lógica es la que se aplicará para realizar las vista de los *DMSCONNECTORS*, los que tendrán la información propia, junto con las aplicaciones asociadas, mostrada en la vista que ejemplificada en la Figura #50.

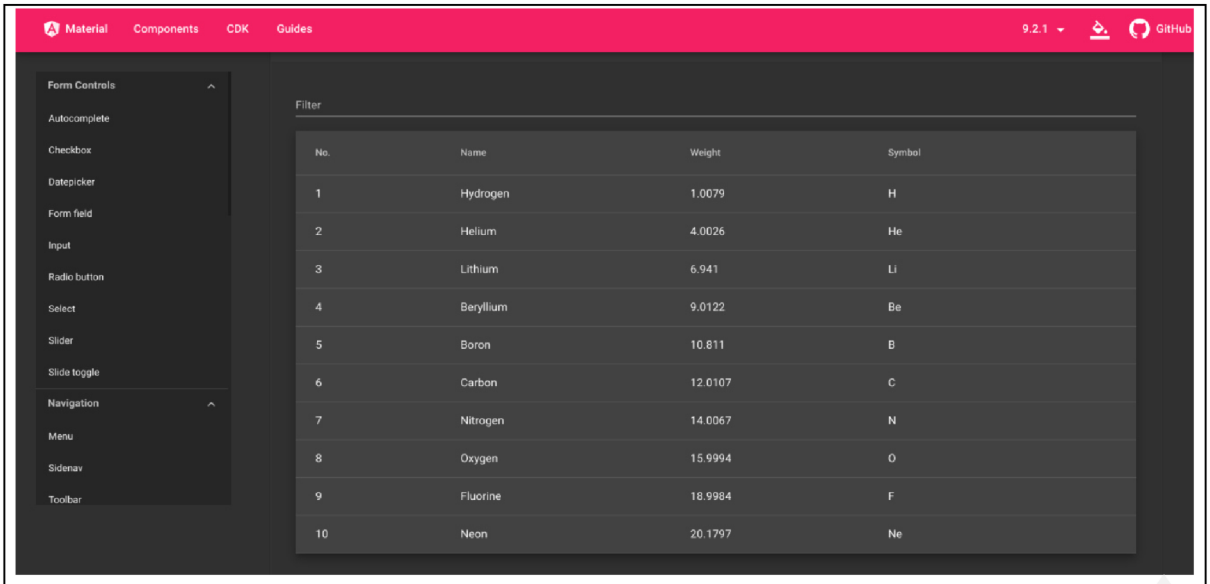


Figura #49: Tablas de angular materials

Fuente: https://material.angular.io/components/table/examples

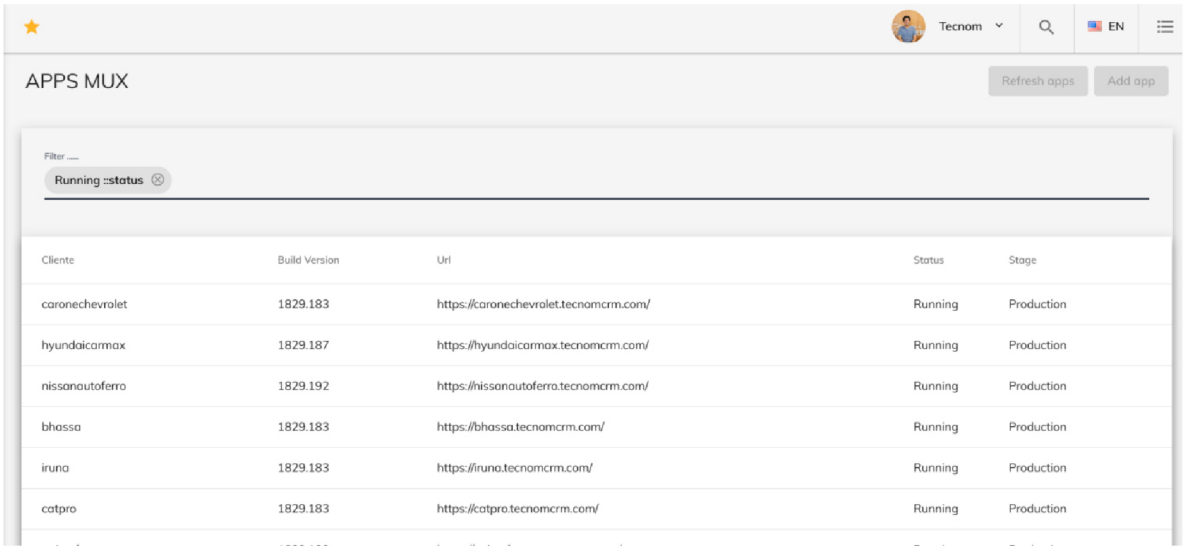


Figura #50: Resultado vista de CRM

Fuente: Elaboración propia

3. Vista DMSCCONECTORS

Para la vista de *DMSCCONNECTORS* se decidió implementar un listado con filas expandibles, donde se permitan visualizar los detalles descritos en el

diseño de dicho *DMSCCONNECTOR*, con el objetivo de mostrar la información detallada de forma rápida y sencilla para el usuario.

Para la vista del detalle al igual que el diseño de las tablas (Listado), se implementaron nuevamente librerías de Angular Materials, en esta ocasión, "Cards" (Figura #52), las que permiten mostrar información en pequeños cuadros. Para esto se creará una *Card* que indicará el contenido de los CRM, una *Card* para detallar información de los *DMSCCONNECTORS*, y una *Card* para enseñar información los sistemas "DMS".

Para que el desarrollo sea más ordenado se implementaron dos componentes por separado, uno, el componente padre ("list-dmscc"), que contendrá la lógica de negocio para el listado de *DMSCCONNECTORS*, y otro, componente hijo ("details") que contendrá las "Cards" con el detalle de los *DMSCCONNECTORS*, tal y como se ejemplifica en la Figura #51.

El resultado de las vistas obtenidas se puede observar en las Figuras #53 y #54, lasque contienen un *DMSCCONNECTOR* filtrado y el detalle del mismo respectivamente.

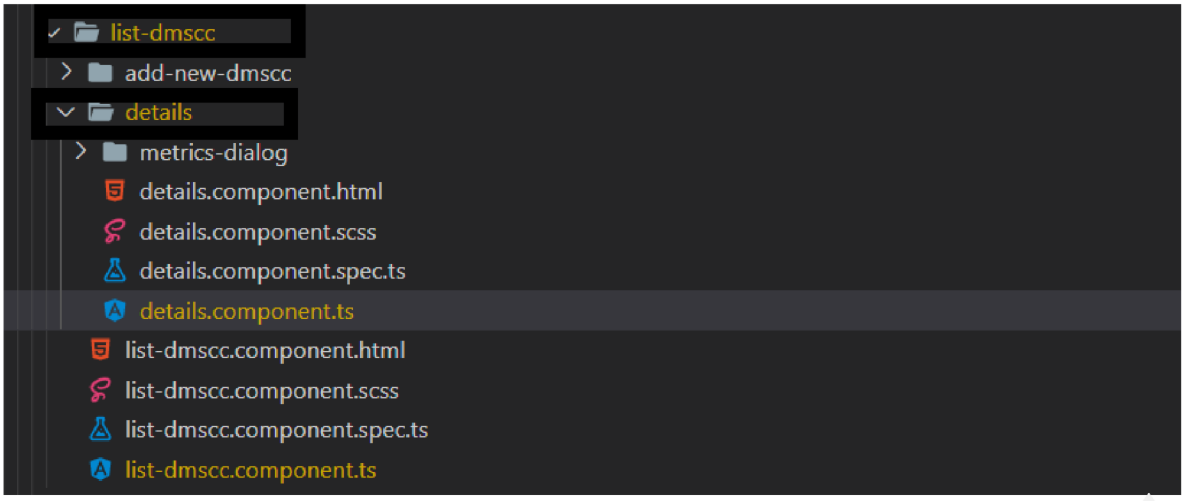


Figura #51: Componente padre-hijo DMSCCONNECTORS

Fuente: Elaboración propia

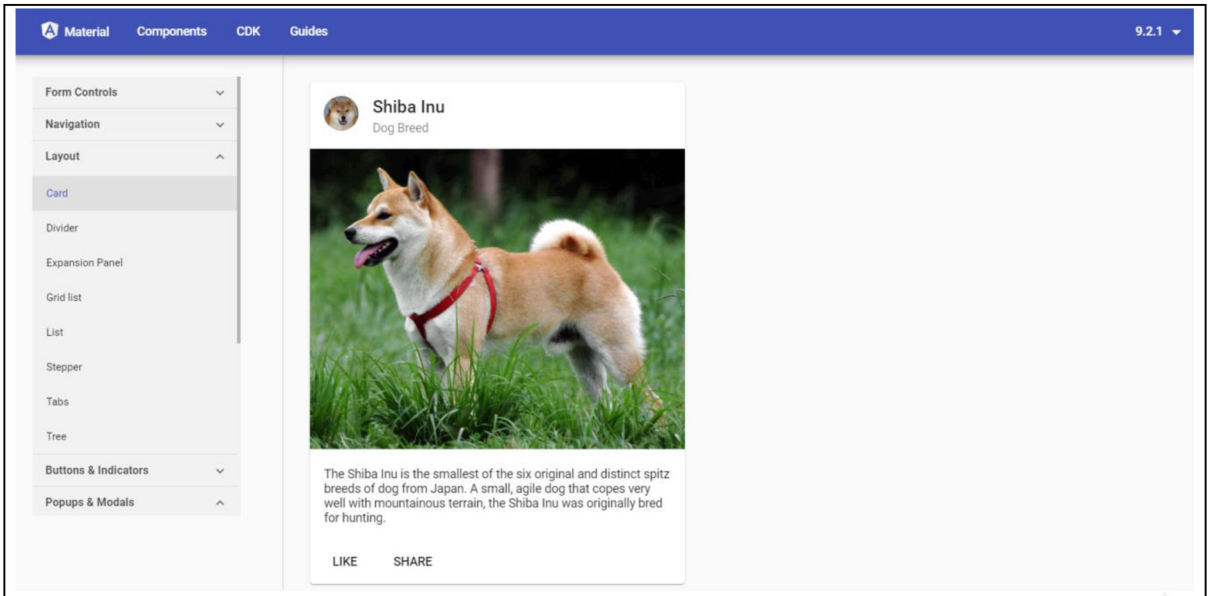


Figura #52: Ejemplo de una Card con Angular Materials

Fuente: https://material.angular.io/components/card/examples

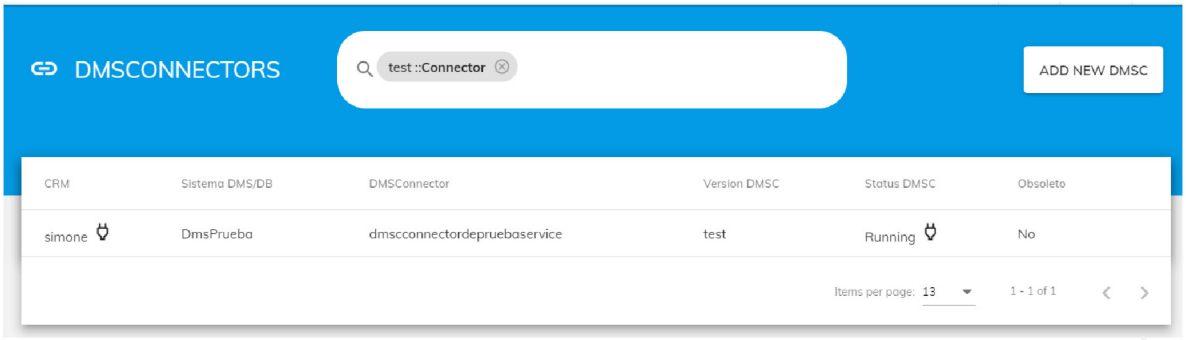


Figura #53: Ejemplo de filtrado de un DMSCCONNECTOR

Fuente: Elaboración propia

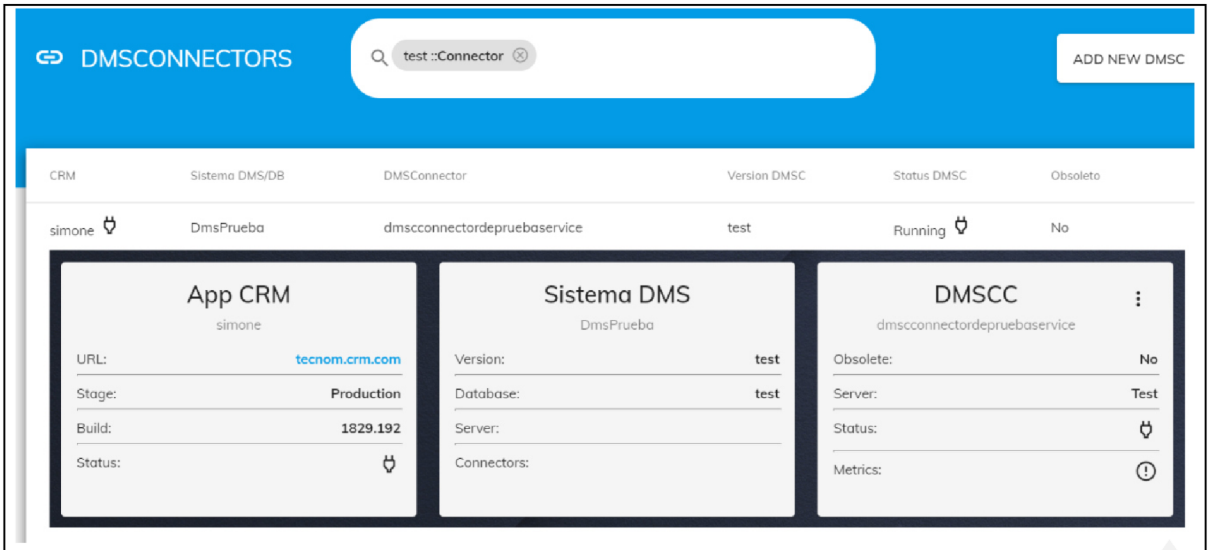


Figura #54: Ejemplo del detalle de un DMSCCONNECTOR con cards
Fuente: Elaboración propia

4. Vista SERVER

Una vez desarrolladas las vistas ejemplificadas en las figuras #53 y #54, posteriormente será necesario diseñar la pantalla que mostrará la información detallada de los servidores donde se alojen los DMSCCONNECTORS, tal y como se explicó en la sección "2.2 - *Definición de vistas*". Esta Vista contendrá tres apartados, en el primero se visualizarán los detalles del servidor(Nombre, Propietario, Sistema Operativo, Estado)tal y como se observa en la Figura #55, el segundo contendrá los DMSCCONNECTORS alojados en dicho servidor(Figura #56) y finalmente el tercer apartado listará los sistemas DMS que se encuentren en el servidor seleccionado(Ver Figura #57).

El acceso a dicha vista será a través de la pantalla mostrada en la Figura #54, al "Clickear" el nombre del servidor (en el ejemplo es "Test").

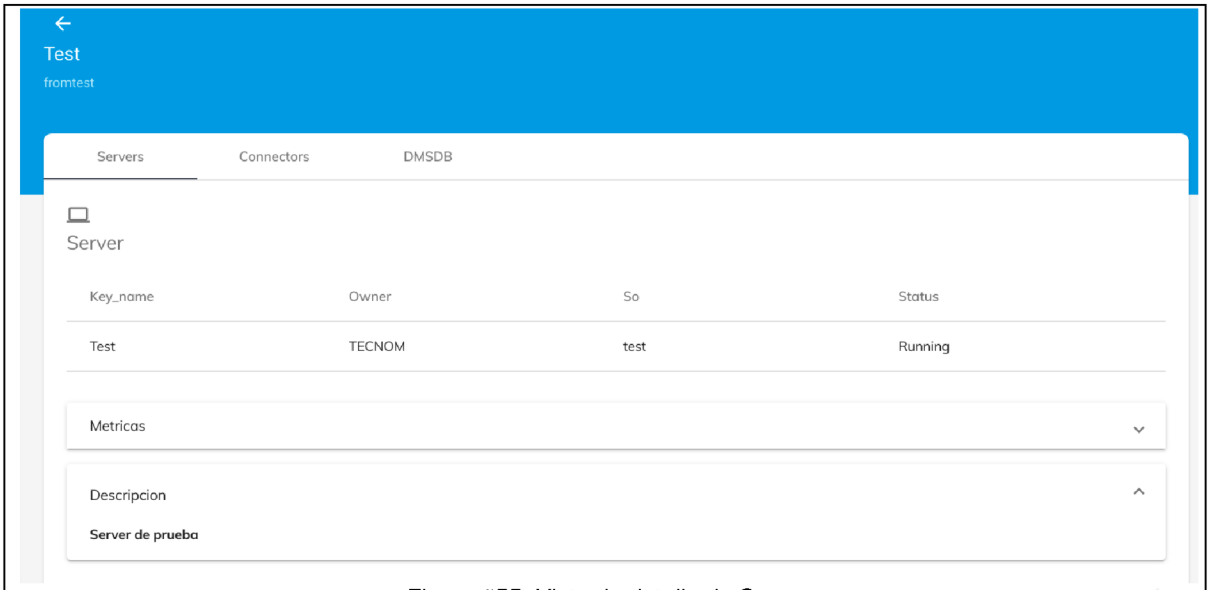


Figura #55: Vista de detalle de Server
Fuente: Elaboración propia

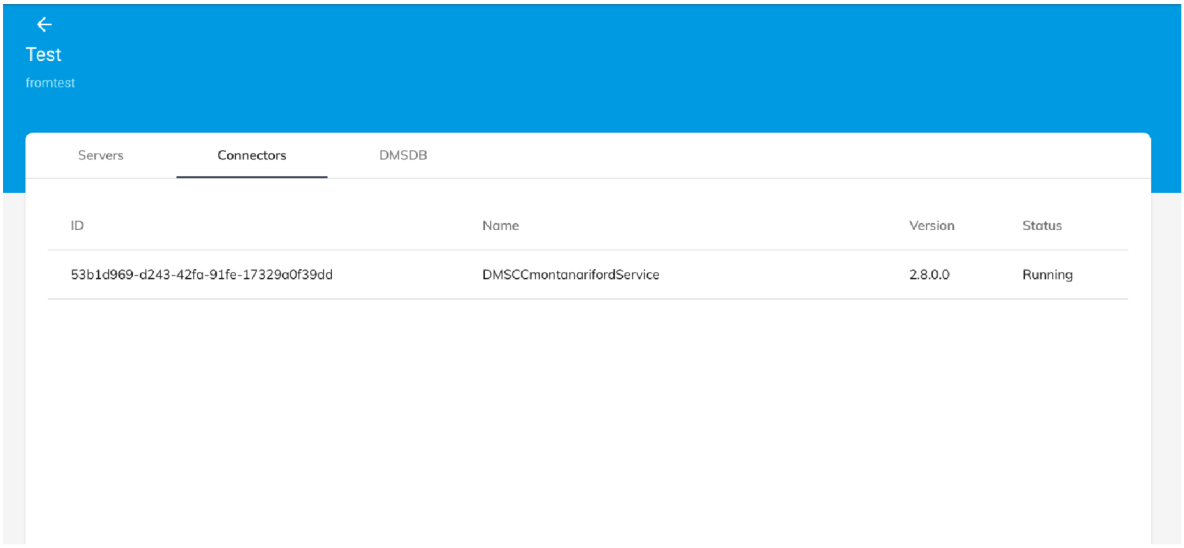


Figura #56: Vista de Server, apartado de Connectors.
Fuente: Elaboración propia

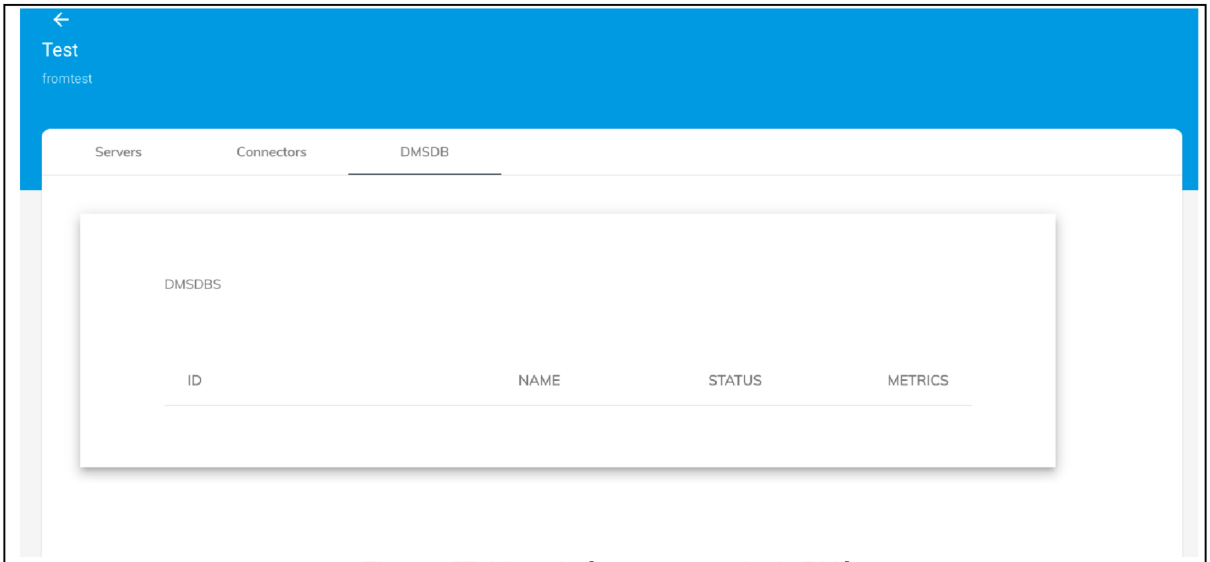


Figura #57: Vista de Server, apartado de DMS
Fuente: Elaboración propia

5. Vista de ABM

Por último, con lo que engloba al frontend, se desarrollarán las vistas para el ABM de *DMSCCONNECTORS,* las que contendrán una serie de Forms e inputs que engloben los datos necesarios para dar de alta un nuevo *DMSCCONNECTOR* en la base de datos. Para esto es necesario tener las siguientes consideraciones:

1) Al dar de alta un DMSCC, es requisito contar un CRM cargado para asociárselo.

2) Debe existir al menos un DMS, puede ser de test, para asignarlo al DMSCC en cuestión.

3) El alta de servidores puedes ser opcional, ya que al principio no se tendrá información de éstos, y más adelante, se lo agregará al editar dicho DMSCC.

Otra aclaración importante es que la aplicación solo contendrá el ABM de DMSCCONNECTOS, por lo que en este mismo es donde se podrán añadir los servidores y Sistemas DMS. Lo que respecta a la carga de CRM, se desarrollará una *Lambda* que mediante una API obtendrá los datos de los CRM del sistema MUX el cual los administra, esto es explicado en la sección “2.7 - Carga de datos”.

A continuación, en las Figuras #58, #59 y #60 se muestran los resultados de las vistas del ABM, en los que se utilizaron las librerías de Angular Materials para realizar un Stepper.

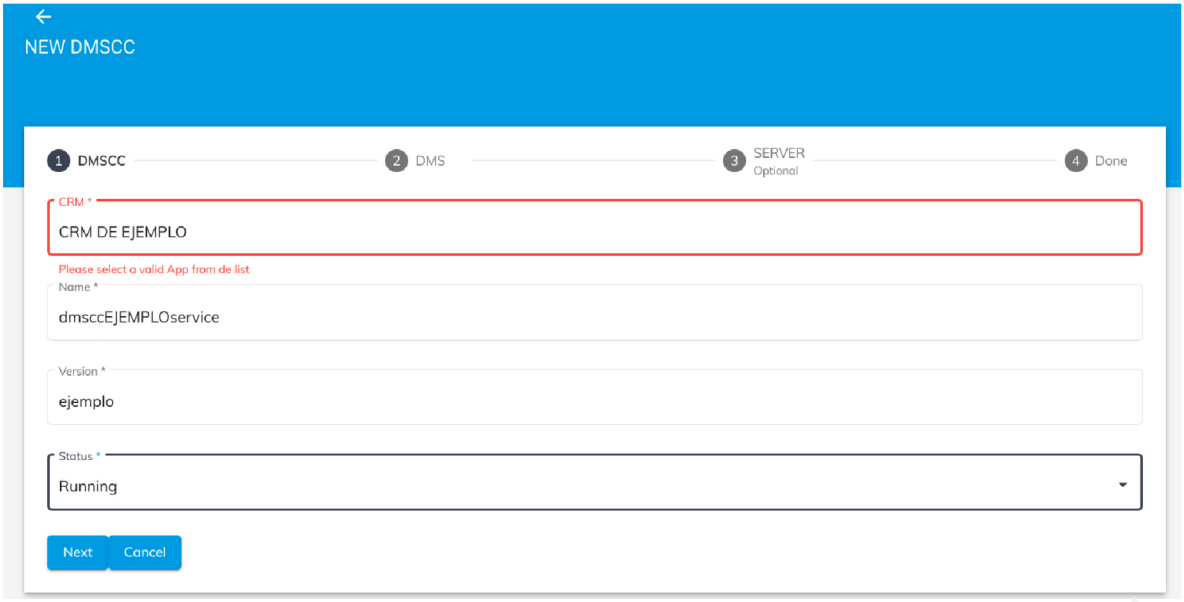


Figura #58: Carga de datos del DMSCC, incluyendo el CRM Asociado
Fuente: Elaboración propia

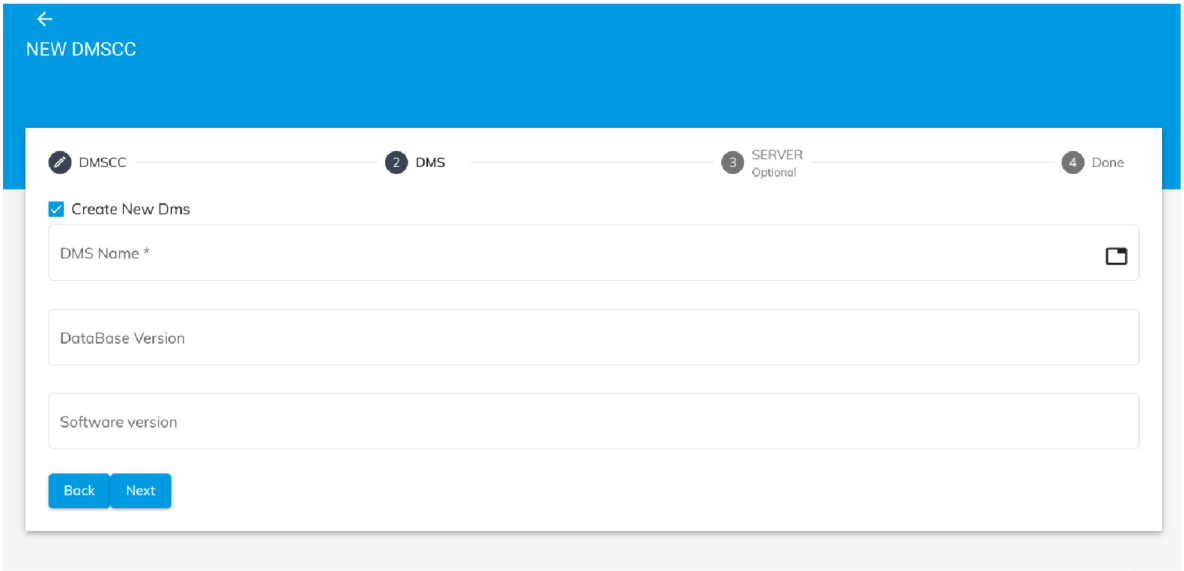


Figura #59: Carga de datos del DMS asociado al DMSCC
Fuente: Elaboración propia

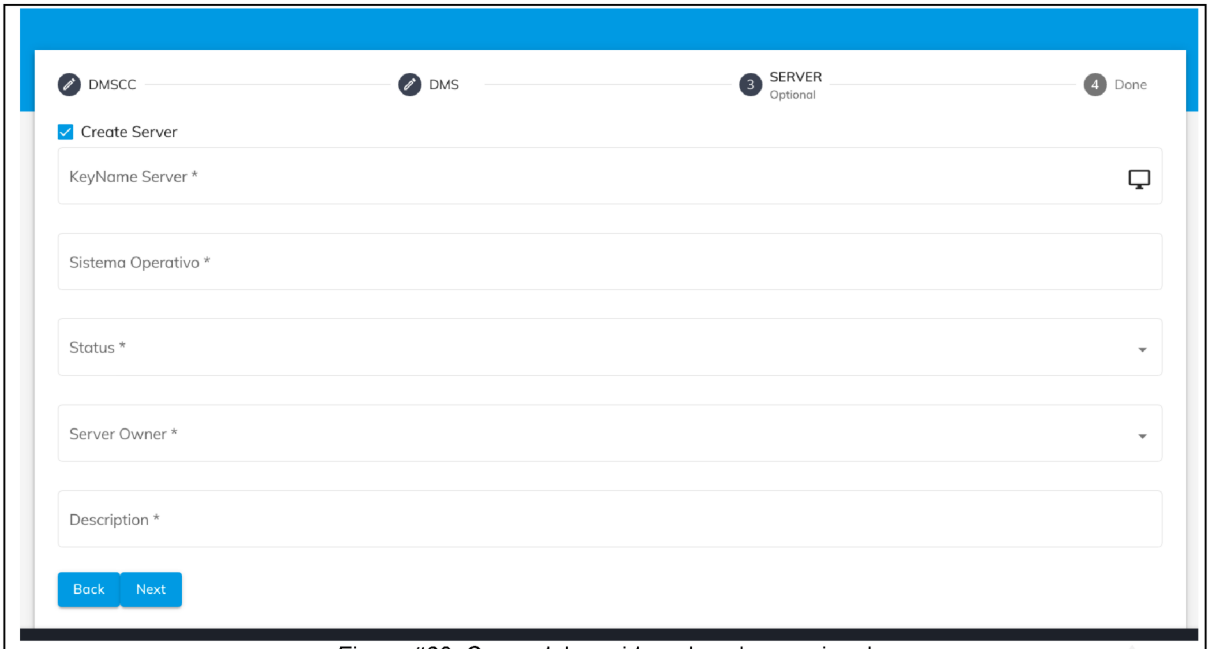


Figura #60: Carga del servidor, el cual es opcional.
Fuente: Elaboración propia

6. Lectura de métricas, desarrollo de lambdas

1. Escenario

Para llevar a cabo una monitorización más detallada de los DMSCC, fue necesario implementar una *Lambda* que realice una lectura periódica sobre las métricas y estado de los mismos mediante "Azure Application Insights".
Azure Application Insights un servicio que ofrece la nube de Microsoft (Azure), que proporciona herramientas para monitorizar, analizar y detectar errores de rendimiento en las aplicaciones alojadas en dicha nube. Además, ofrece la posibilidad de insertar trazas personalizadas y registrar errores en estas aplicaciones.

La utilización del servicio permite tener de una forma rápida y precisa, los datos de análisis relacionados con el rendimiento y funcionamiento de la aplicación. También ofrece la posibilidad de configurar alertas personalizadas en las que, por ejemplo, se puede recibir un email cada vez que se produzca un determinado error. Actualmente el sistema MUX (aplicación de gestión de los CRM

de la empresa), utiliza AppInsight para leer las métricas correspondientes a dichos CRM, tal y como se ejemplifica en la Figura #61, en la que se pueden ver diversas métricas como el porcentaje del CPU utilizado por la web, tareas programadas fallidas, errores en actualizaciones. La obtención de estas métricas se realiza mediante peticiones a la API expuesta por AppInsights.

Figura #61: Ejemplo de lectura de métricas en MUX.

Fuente: Captura brindada por Tecnom

Para este proyecto se utilizarán las siguientes métricas:

dependencies/failed: sum

Cantidad de llamadas fallidas al DMS por parte del DMSCC [2](no debería haber, pero son usuales, en el caso de cambio usuario, revocación permisos, fallas del server, timeouts, etc). Es una métrica importante para reducir a ceroen

[2]DMSCC : DMSCCONNECTORS

cuanto comiencen a generarse los problemas. Generalmente es responsabilidad del área de infraestructura solucionar estos inconvenientes.

A continuación, en la Figura #62 se detalla la estructura de la API de AppInsights a consultar, en la que se observa el id del cliente y la cantidad de fallos (SUM). Este esquema es similar a todas las métricas, donde sólo cambia el parámetroMetricID.

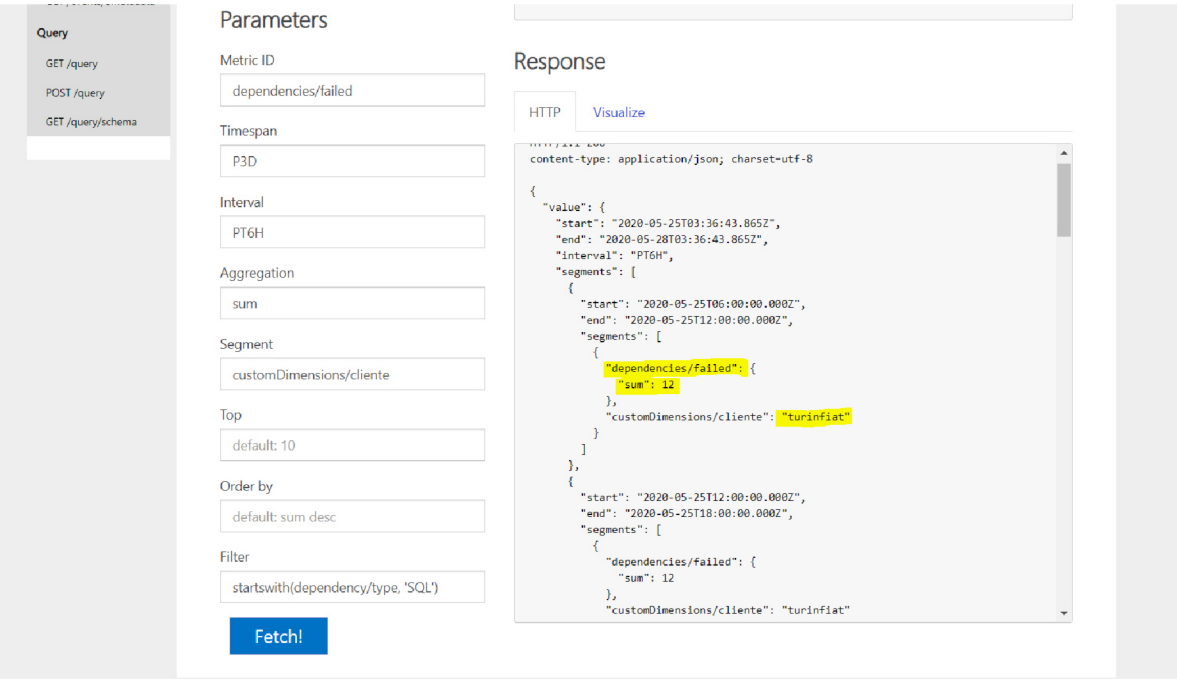


Figura #62: Consola de AppInsights

Fuente: https://dev.applicationinsights.io/reference

customMetrics/sync_count: sum

Esta métrica detalla los recursos sincronizados entre los DMS y el CRM, por lo que, si el valor es 0 o cercano a éste, podría indicar fallos en el DMSCC, que no estén permitiendo sincronizar.

customMetrics/pending_count: sum

Cantidad de recursos que no están siendo sincronizados (que están en DMS y no se están enviando al CRM). Análogamente a la métrica anterior, ésta debería tener un valor cercano a cero.

2. Implementación:

Como primer paso se procedió a agregar la función (*Lambda*) a desarrollar, ejecutando por consola en la raíz del proyecto el siguiente comando: amplify add function, mediante el cual se podrá configurar el nombre de la *Lambda (nombre elegido "**aiSyncFunction**"),* el lenguaje de programación, en este caso se optó por NodeJS y una plantilla de ejemplo "Hello World". Una vez finalizado Amplify creará la lista de recursos en el directorio Amplify/backend/functions/nombre de la *Lambda,* donde se encuentra el código que se fuera a desarrollar.

A modo de ejemplificación, la Figura #63 indica el paso de configuración de la *Lambda* y la Figura #64, el directorio con los recursos creados por *Amplify*, donde se aloja el código a programar (src/index.js).

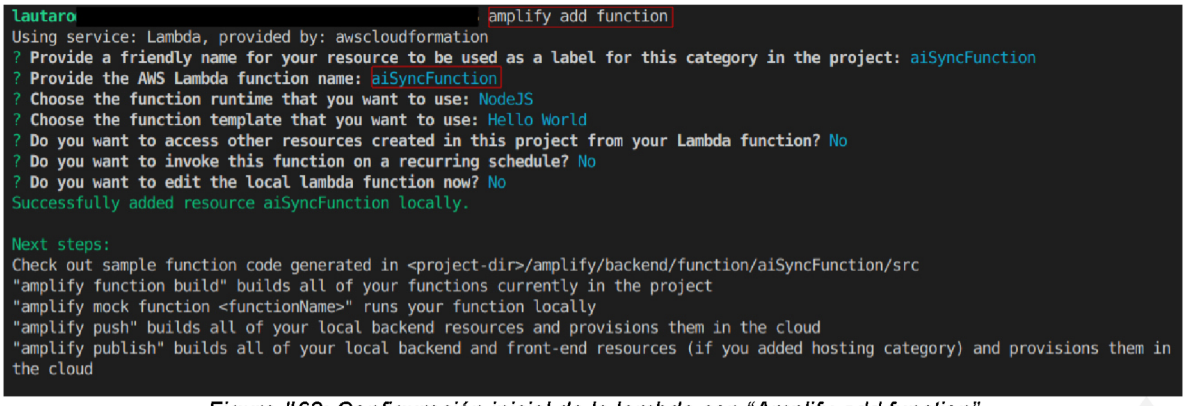


Figura #63: Configuración inicial de la lambda con "Amplify add function"

Fuente: Elaboración propia

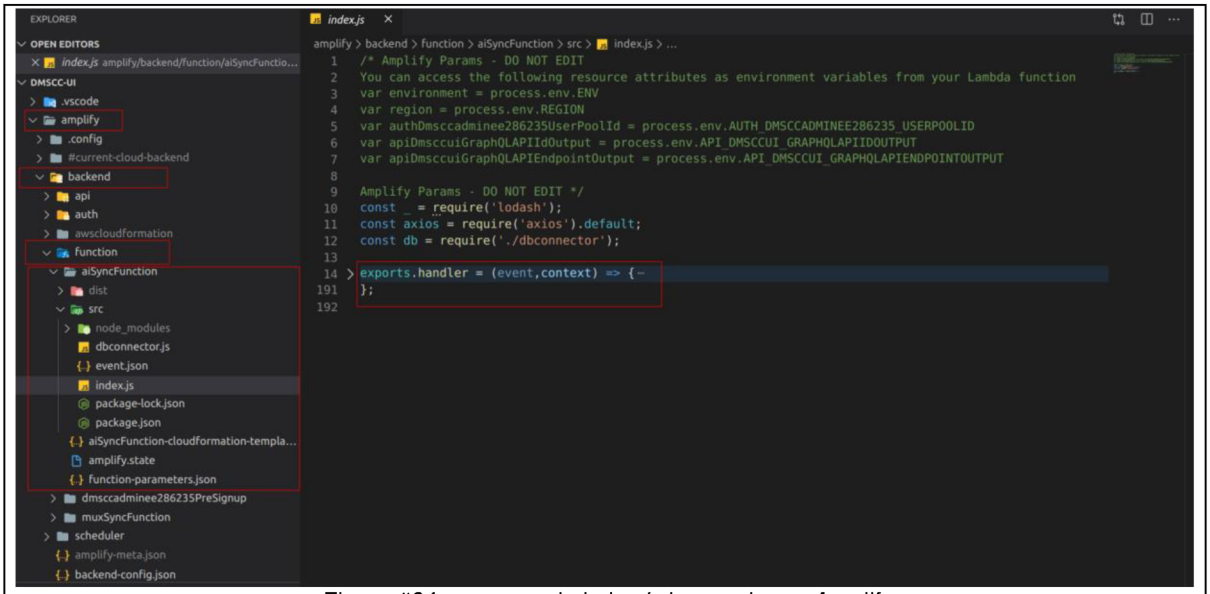


Figura #64: recursos de la lambda creado por Amplify

Fuente: Elaboración propia

A continuación, se mostrará en detalle el código de desarrollo de esta funcionalidad.

Se utilizaron las siguientes librerías para NodeJs:

- Lodash: Librería de JavaScript para trabajar con colecciones de objetos.

- Axios: Axios es una librería JavaScript que puede ejecutarse en el navegador y que permite hacer sencillas las operaciones como cliente HTTP, por lo que permite configurar y realizar solicitudes a un servidor y recibir respuestas fáciles de procesar.

- Aws-sdk: SDK es una librería que permite eliminar la complejidad de la codificación, ya que proporciona objetos JavaScript para los servicios de AWS, como Amazon S3, Amazon EC2, DynamoDB.

Con lo que respecta a la lógica, consta básicamente de varias peticiones GET (una por métrica) a la API expuesta por AppInsights explicada al inicio de la presente sección, mediante la librería Axios, como se ejemplifica en la Figura #65.

```
// Metrica Dependencies/Failed:
axios.get("https://api.applicationinsights.io/v1/apps/
}).then(async (res)=>{
  console.log(res);

  // Ordeno la data
  var arr=[];
  res.data.value.segments.forEach(element => {
    element.segments.forEach(element => {
      if(!_.find(arr,{client:element["customDimensions/cliente"]})){…
      }
      else{…
      }

    });
  });

  console.log(arr)

  // Armo la data para el update
  db.buildResultArray(arr).then((updatedApps)=>{
    console.log(updatedApps);

    db.updateTableData(updatedApps).then(()=>{
      context.succeed({message:"OK"});
    },(err)=>{
      console.log(err);
      context.fail({message:err.message});
    });
  },(err)=>{
    console.log(err);
    context.fail({message:err.message});
  })
```

Figura #65: Petición a App Insight con Axios.

Fuente: Elaboración propia

Luego, con los datos recibidos en la respuesta, se obtiene una sumarización total de la cantidad de fallos (SUM) indexado por el código de cliente o KeyName, el cual pertenece a los CRM. Como resultado se arma un objeto, el cual contiene el CRM y el valor de la métrica consultada. En el caso de la Figura #66 el valor de la métrica corresponde a **dependencies/failed: sum.**(el proceso se repite para las dos métricas restantes).

```
// Ordeno la data
var arr=[];
res.data.value.segments.forEach(element => {
  element.segments.forEach(element => {
    if(!_.find(arr,{client:element["customDimensions/cliente"]})){
      arr.push({
        client:element["customDimensions/cliente"],
        fails:element["dependencies/failed"].sum,
        intervals:res.data.value.interval,
        time:res.data.value.start
```

Figura #66: Objeto armado a partir de los datos obtenidos

Fuente: Elaboración propia

Una vez armado el objeto con las métricas, es necesario insertarlo en la base de datos, DynamoDB, de este proyecto (DMSCC-UI).

Para esto se creó un archivo nuevo llamado “dbconnector.js”, el cual contendrá la lógica encargada de realizar las inserciones y actualizaciones de los datos mencionados.

Para llevar a cabo este procedimiento, se utilizó la librería AWS-SDK que permite modificar de manera directa la base de datos seteando las credenciales del usuario IAM de AWS con permisos totales a la misma, tal y como se puede observar en la Figura #67. Posteriormente, se realizó una consulta a la base de datos para obtener todas los CRM cargados en DMSCC-UI y así comparar con los CRM obtenidos de AppInsights, para luego insertar los coincidentes en los DMSCCONNECTORS que posean dicho CRM. En las Figuras #68 se ejemplifica cómo es la consulta para la obtención de todas las Apps alojadas en DMSCC-UI y en la Figura #69 se muestra cómo es la inserción de los datos en la tabla de los *DMSCCONNECTORS*.

dbconnector.js

amplify > backend > function > aiSyncFunction > src > dbconnector.js > AWS

```
1  var AWS = require('aws-sdk');
2  // Set the region
3  AWS.config.region = 'us-east-1';
4  AWS.config.accessKeyId = [redacted]
5  AWS.config.secretAccessKey = [redacted];
6
```

Figura #67: Configuración de credenciales

Fuente: Elaboración propia

```
33  getAllApps() {
34    var db = new AWS.DynamoDB.DocumentClient();
35
36    var params = {
37      TableName: 'App' + process.env.DynamoDBTablePosfix,
38      //TableName:'AppTable',
39      Region: 'us-east-1'
40    }
41    return db.scan(params).promise().then((res) => {
42      console.log(res.Items)
43      return res.Items;
44    }, (err) => {
45      console.log(err);
46    });
47  },
```

Figura #68: Consulta para obtener todas los CRM de DMSCC-UI

Fuente: Elaboración propia

```
73  updateOneApp(app,dmscc){
74    if(!dmscc)
75      return Promise.resolve();
76
77    var db = new AWS.DynamoDB.DocumentClient();
78    var params = {
79      TableName: 'DMSConnector' + process.env.DynamoDBTablePosfix,
80      //TableName:'AppTable',
81      //Region: 'us-east-1',
82      Key: {
83        "id": dmscc.id
84      },
85      UpdateExpression: "set appInsightStatus.failCallstoDb = :f, appInsightStatus.timeSpan=:t, appInsightStatus.
86      ExpressionAttributeValues: {
87        ":f": app.value,
88        ":t": app.start,
89        ":i": app.inter
90      },
91      ReturnValues: "UPDATED_NEW"
92    }
93    return db.update(params).promise();
94  }
```

Figura #69: Actualización de las métricas obtenidos en la tabla DMSCC

Fuente: Elaboración propia

3. Scheduler

Una vez finalizado el desarrollo de *Lambda,* el siguiente desafío consiste en buscar la forma de generar un planificador (Scheduler) que ejecute la *Lambda* automáticamente cada cierto tiempo, para mantener sincronizado los datos con AppInsights. Las versiones de *Amplify* CLI posteriores a las 4.16 permiten agregar un planificador desde la línea de comandos, pero en el momento que se desarrolló este proyecto aún no estaba implementado.

Como solución se decidió implementar un recurso personalizado al que se llamó *Scheduler,* el cual permitió la ejecución automática de la *Lambda* cada 1hs. Para llevar a cabo ésto fue necesario implementar un pequeño desarrollo utilizando los conceptos de AWS SAM.

AWS SAM (Serverless Application Model): es un framework de código abierto que permite crear aplicaciones Serverless en AWS. Consta de dos componentes: AWS SAM template y AWS SAM CLI, para este proyecto sólo se utilizarán parte del primero para crear el scheduler.

AWS SAM templates specification: a grandes rasgos, es una plantilla que permite describir los recursos de AWS a utilizar. Si bien en este proyecto no se utilizará SAM de forma directa, sus conceptos servirán para crear el recurso AWS::Events::Rule, el cual permitirá invocar la *Lambda* que se creó en el paso anterior automáticamente según el rango de tiempo que se le pase por parámetro, en este caso, cada 1 hora.

La plantilla debe contener:

- Nombre y permisos de la *Lambda* a invocar (aiSyncFuntion).

- Recurso o evento que invocara a *aiSyncFuntion*.

Como primer paso y como se muestra a continuación en la Figura #70, se creó una carpeta llamada "scheduler" en el directorio amplify/backend, que contendrá la plantilla ya mencionada.

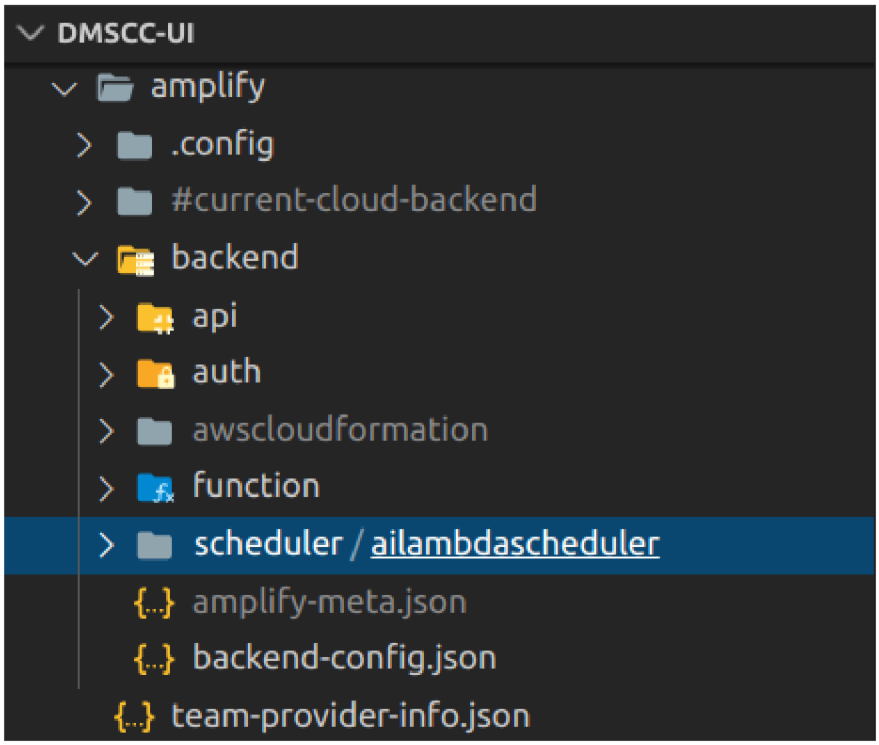


Figura #70: Carpeta Scheduler

Fuente: Elaboración propia

El siguiente paso es crear la plantilla.JSON que contendrá el evento que invocará la *Lambda* junto con los permisos correspondientes (Ver Figura #71).

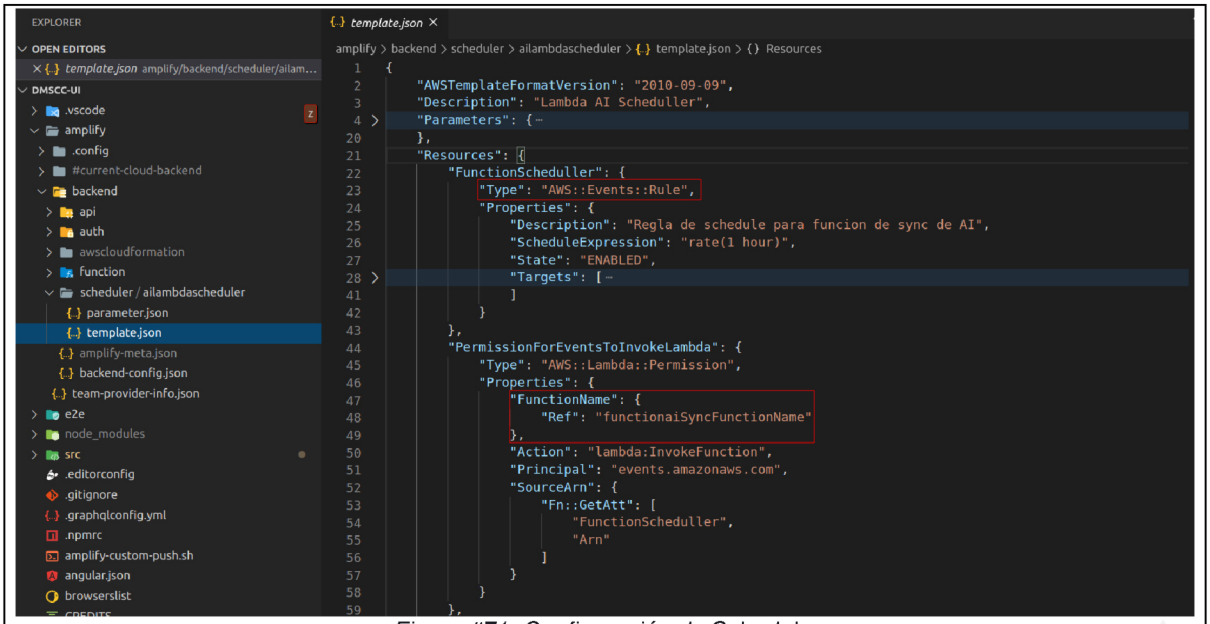


Figura #71: Configuración de Scheduler

Fuente: Elaboración propia

Como último paso antes de subirlo a la nube, es necesario editar el archivo backend-config.json, ubicado en la carpeta amplify/backend, el cual contiene todas las categorías agregadas previamente (api, auth , functions) para que *Amplify* detecte al ”Schedule” como un nuevo tipo de categoría (Ver Figura #72). Cabe destacar que este archivo es generado y modificado automáticamente por *Amplify* al agregar o quitar las categorías, pero en este caso fue necesario añadirla manualmente ya que “scheduler” es una categoría personalizada.

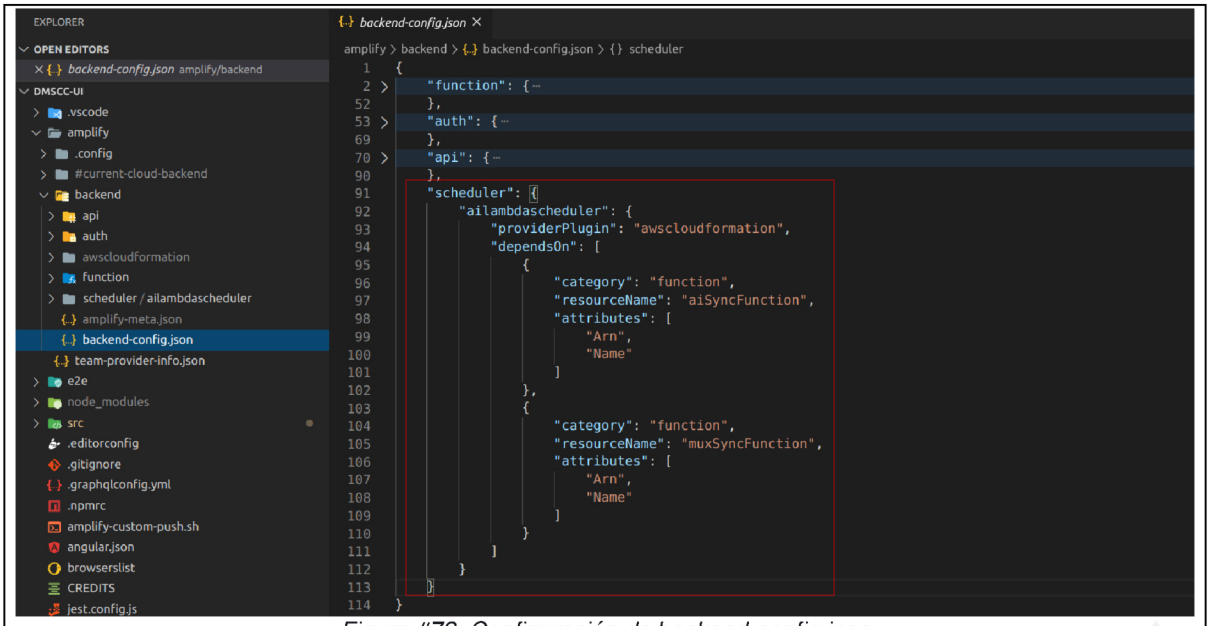


Figura #72: Configuración de backend-config.json

Fuente: Elaboración propia

A continuación, en la Figura #73 se muestra la categoría scheduler añadida exitosamente luego de ejecutar "*Amplify* push".

Figura #73: Scheduler Añadido

Fuente: Elaboración propia

4. Vista MÉTRICAS

Para terminar esta sección, se mostrarán los resultados del diseño del frontend con los datos de las métricas obtenidos de AppInsights con la ejecución periódica de la *Lambda* aiSyncFunction. Para el desarrollo de la vista se utilizaron los cuadros de diálogo que ofrece la librería de angular material.

Para acceder a ver dichas métricas, se incorporó un ícono resaltado dentro de la vista de detalle de DMSCCONNECTORS (ver sección “2.5 - Listado, ABM y filtrado frontend”), el cual al seleccionarlo abre el diálogo que contiene las métricas de AppInsights correspondiente a dicho DMSCC. Los ejemplos de esto se observan en las Figuras #73 y #74.

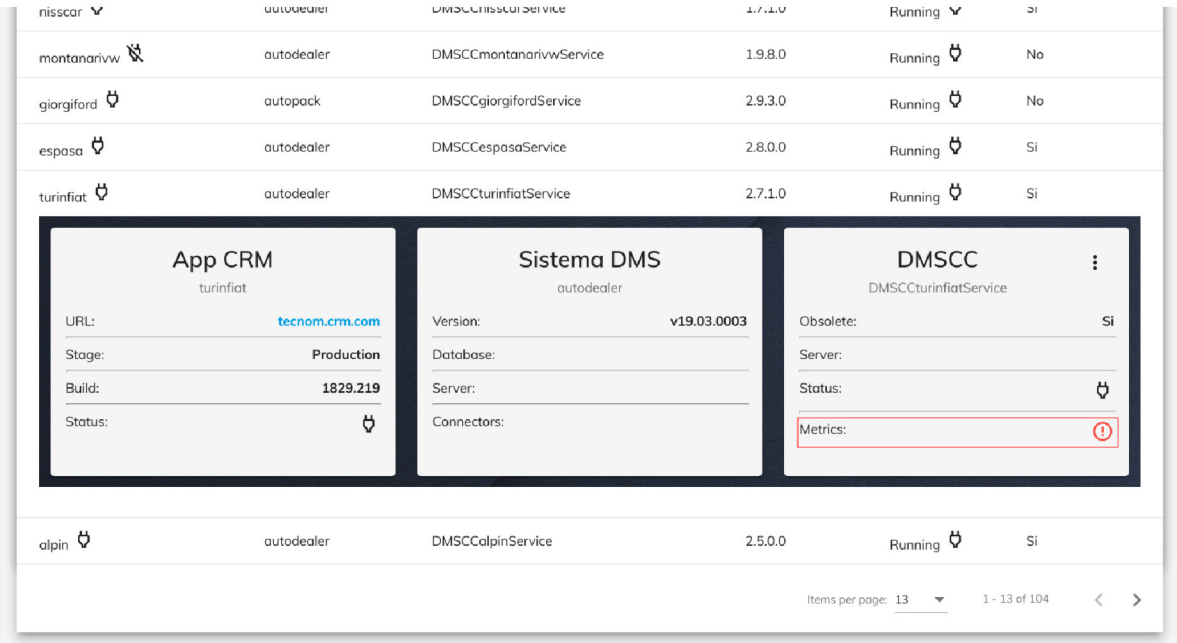


Figura #74: Icono de acceso a métrica.

Fuente: Elaboración propia

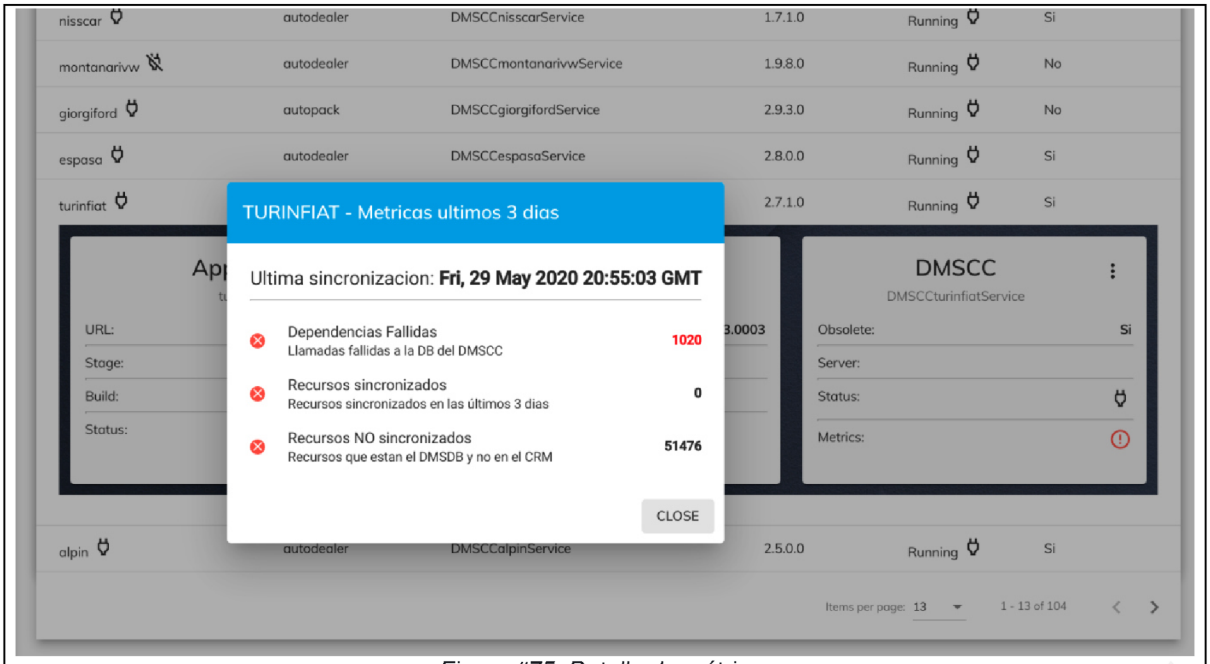


Figura #75: Detalle de métricas

Fuente: Elaboración propia

7. Carga de datos

En primera instancia, antes de cargar los *DMSCCONNECTORS* es necesario obtener todos los CRM (clientes) que se encuentran en producción.

Para realizar esto se desarrolló una *Lambda* que realice peticiones periódicas a la API expuesta por la aplicación MUX (sistemas de gestión de CRM), la que brindará a este proyecto los datos de todos los CRM de producción en tiempo real y una vez cargados, se procederá a cargar de forma manual los *DMSCCONNECTORS* mediante el ABM desarrollado en el frontend.

1. Implementación

Como primer paso, al igual que en la sección 2.6 “Lectura de métricas, desarrollo de lambdas.”, se crea la *Lambda* mediante el comando *Amplify* add function donde se establecen las configuraciones básicas como el nombre (“muxSyncFunction”, este nombre es debido a que mantendrá comunicación con

el sistema externo MUX), recursos a los que tendrá acceso y el lenguaje de programación.

Esta *Lambda* será desarrollada en Nodejs y, además, se le añadirán los permisos para acceder a la API de GraphQL (desarrollada en la sección 2.3"Arquitectura y desarrollo de la Api") tal y como se detalla en la Figura #76.

Figura #76: Recursos de la Lambda "muxSyncFunction"

Fuente: Elaboración propia

2. Desarrollo de peticiones a la API de MUX.

Una vez creada la *Lambda* se procede a desarrollar lo que será el proceso de obtener los CRM a partir del endpoint brindado por MUX. Para esto, mediante postman, se realizaron las pruebas pertinentes para analizar la estructura de los datos recibidos previo a la programación de la *Lambda*.

Para poder tener los permisos necesarios al endpoint de MUX, es necesario realizar una consulta mediante un POST al endpoint de Cognito User Pools, brindado por el personal de Tecnom, el cual que devolverá un TOKEN que servirá luego para consultar el endpoint correspondiente a MUX que contiene los CRM.

En las Figura #77 observa la consulta al endpoint, donde se obtendrá el "barer TOKEN".

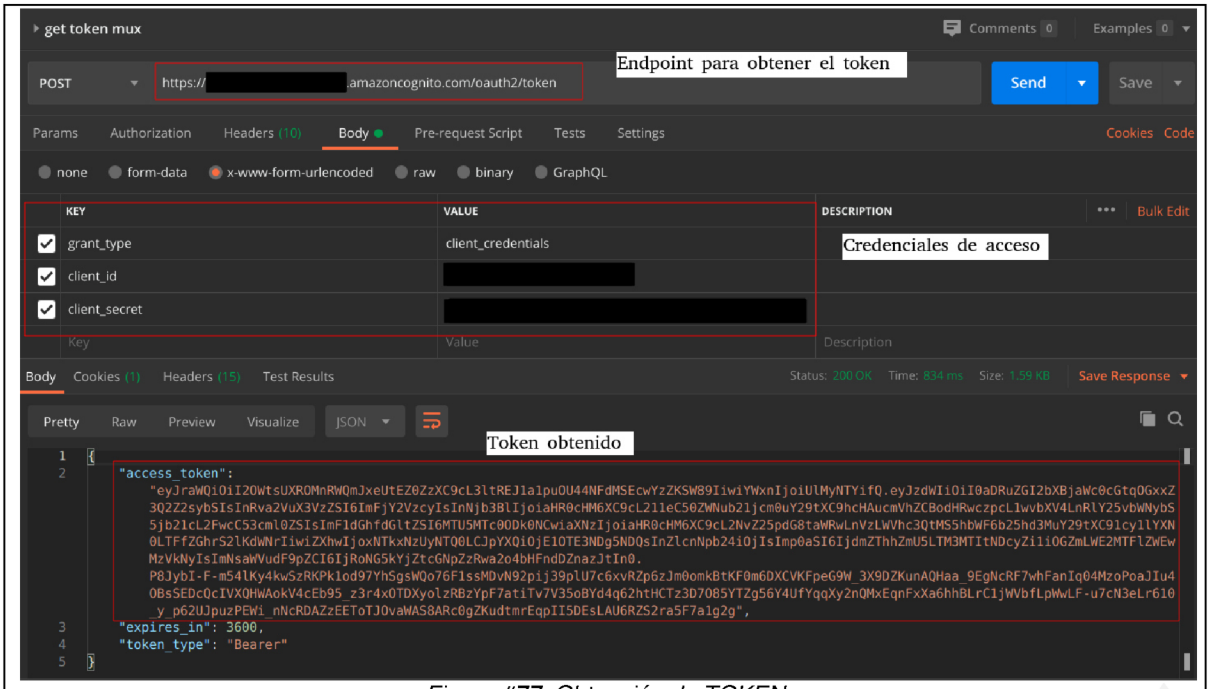


Figura #77: Obtención de TOKEN

Fuente: Elaboración propia

Una vez obtenido el TOKEN, a continuación, como se observa en la Figura #78, se realiza una petición GET al endpoint de MUX pasando el TOKEN como parámetro en la sección “Authorization” y obteniendo como resultado en la respuesta todos los CRM de la compañía.

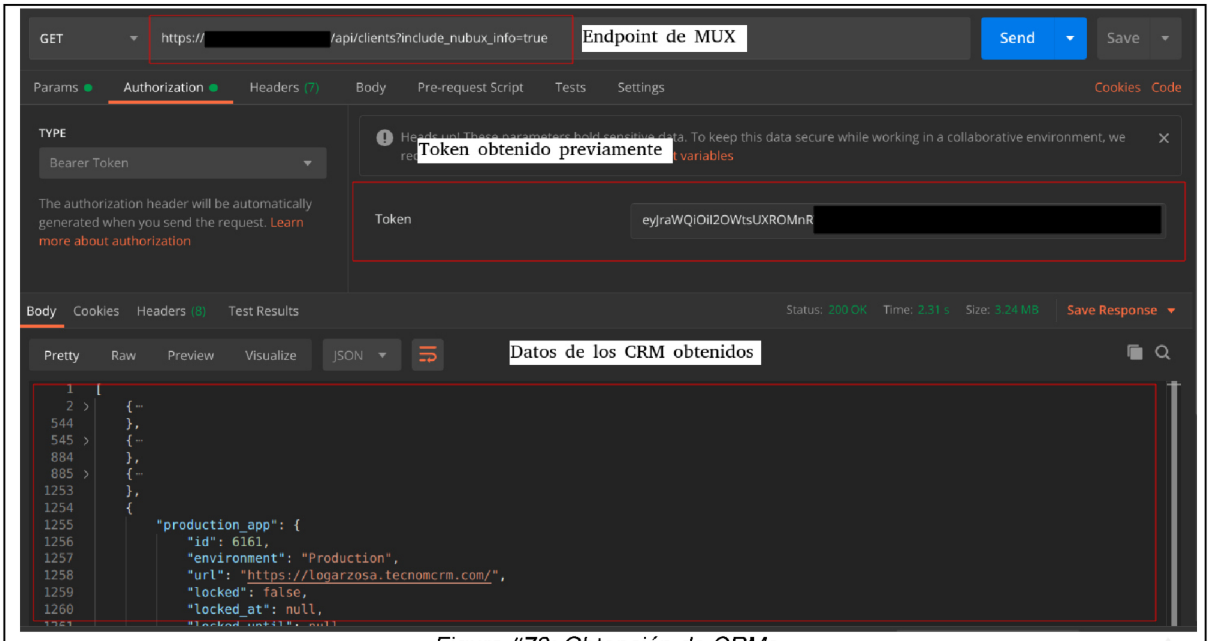


Figura #78: Obtención de CRMs

Fuente: Elaboración propia

3. Análisis de datos obtenidos:

En la Figura # 79 se mostrará la estructura de los datos recibidos, en la que se puede observar los atributos que son necesarios, tal y como se especificó en el diseño (Ver sección 2.2 – “Definición de vistas y casos de uso”) entre ellos se destacanel KeyName, Url, Build, Versión.Además, los CRM que se tendrán en cuenta son los que contengan activo el atributo “production_app”, lo que significa que el CRM se encuentra en producción.

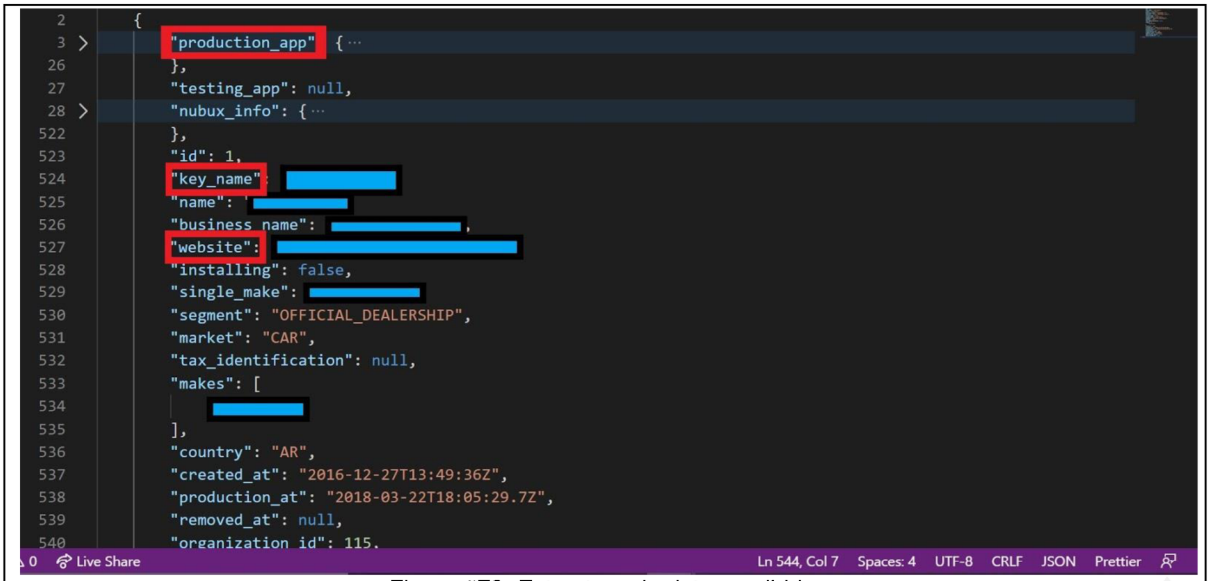


Figura #79: Estructura de datos recibidos

Fuente: Elaboración propia

4. Programación de petición a MUX

En lo que respecta a nivel de código de programación, se utilizaron las mismas librerías ya mencionadas en la sección 2.6 “Lectura de métricas, desarrollo de lambdas”. (Axios, AWS-SDK, Lodash). Para realizar las peticiones a MUX, mediante la librería de Axios, se desarrollaron dos funciones: en una se realiza la petición a la url que devuelve el TOKEN de autenticación, y en la segunda función, con dicho TOKEN almacenado en una variable, se realiza la petición a MUX para obtener los CRM.

A continuación, en las Figuras #80 y #81 se ejemplifican estos pasos.

```
function getAccessToken() {
    return new Promise((resolve, reject) => {
        let requestCofig = {
            headers: { 'Content-Type': 'application/x-www-form-urlencoded' },
        };

        axios
            .post(
                poolConfig.urlToken,
                querystring.stringify({
                    grant_type: 'client_credentials',
                    client_id: poolConfig.clientId,
                    client_secret: poolConfig.client_secret,
                }),
                requestCofig
            )
            .then(function (res) {
                if (res.status == 200) resolve(res.data.access_token);
                else throw new Error('Error getting tokens');
            })
            .catch(function (res) {
                console.log(res);
                reject(res);
            });
```

Figura #80: Estructura de datos recibidos ,continuación de figura #79

Fuente: Elaboración propia

```
function getMuxClients(token) {
  return new Promise((resolve, reject) => {
    let requestCofig = {
      headers: { Authorization: 'Bearer ' + token },
    };

    axios
      .get( [redacted] 'clients?include_nubux_info=true', requestCofig)
      .then(function (res) {
        if (res.status == 200) resolve(res.data);
        else throw new Error('Error get Mux Clients');
      })
      .catch(function (res) {
        console.log(res);
        reject(res);
      });
  });
}
```

Figura #81: Función para obtener los CRM

Fuente: Elaboración propia

Una vez establecidas las funciones, éstas son invocadas y almacenadas en una variable (ver Figura #82) que será utilizada luego para guardar dichos CRM en la base de datos de este proyecto mediante la API de GraphQL.

```
// get Clients MUX
const muxClientsForApps = getAccessToken()
  .then(function (token) {
    const result = getMuxClients(token)
      .then(function (data) {
        console.log('clients');

        //console.log(data)

        return data;
      })
      .catch((err) => {
        console.log(err);
        context.fail('Error get clients');
      });
    return result;
  })
  .catch(function (err) {···
  });
```

Figura #82: Innovaciones de funciones

Fuente: Elaboración propia

5. Desarrollo de peticiones a la API GraphQL de DMSCC-UI:

Como primer paso, antes de desarrollar las funciones en esta *Lambda* que realizarán las peticiones al endpoint de GraphQL, será necesario declarar en el Schema de GraphQL los tipos de autenticación que serán utilizados para conceder acceso a los recursos que exponen.

En este caso la *Lambda* "muxSyncFunction" se autenticará mediante IAM de AWS, utilizando las credenciales de la aplicación mediante las librerías de AWS-SDK.

Una vez decidido el tipo de autenticación, se procede a modificar el archivo "schema.graphql", añadiendo en la cabecera de las entidades la notación "@auth", donde se debe especificar tanto la autenticación por IAM, como la autenticación por Cognito Federated Providers (Ej : Google) ya que ésta última es la que utiliza el frontend para acceder al servicio que expone las consultas a la base de datos(ver Figura #83).

La autorización del tipo "Private" engloba todas las autorizaciones pertenecientes a Cognito User Pools, mientras que para especificar una autenticación por IAM, es necesario declararlo de forma explícita.

```
# Sistema CRM (Tecnom)
type App
  @model
  @auth(
    rules: [
      { allow: private, operations: [create, read, update, delete] }
      {
        allow: private
        provider: iam
        operations: [create, read, update, delete]
      }
    ]
  ) {
  id: ID!
  key_name: String!
  build: String!
  url: String!
  stage: String!
  status: String!
  alerts: String
  connectors: [DMSConnector] @connection(name: "DMS-APP")
}
```

Figura #83: Configuración de autenticación

Fuente: Elaboración propia

Una vez definidas estas configuraciones, se procede a desarrollar a nivel de código la petición al endpoint de GraphQL, para poder insertar los CRM obtenidos desde MUX.

Se comprende que las APIsGraphQL exponen un único endpoint donde mediante consultas se puede obtener cualquier tipo de dato que se necesite, dicho esto en la Figura #85, se ejemplifica una función que realiza una Query al endpoint de GraphQL utilizando la librería de "https" para realizar una petición POST.

Para obtener dicho endpoint se accede a las variables de entorno establecidas al momento que se creó la *Lambda*, entre ellas, además, se encuentran los atributos necesarios para poder realizar la consulta, tales como: la región donde se aloja la aplicación y el ambiente de desarrollo correspondiente.

Para finalizar, en lo que respecta al handler de la *Lambda* (analogía de un main), se insertan las funciones realizadas previamente, realizando la carga de datos definitiva a la aplicación de este proyecto, cuyo flujo de ejecución es el graficado en la Figura #84.

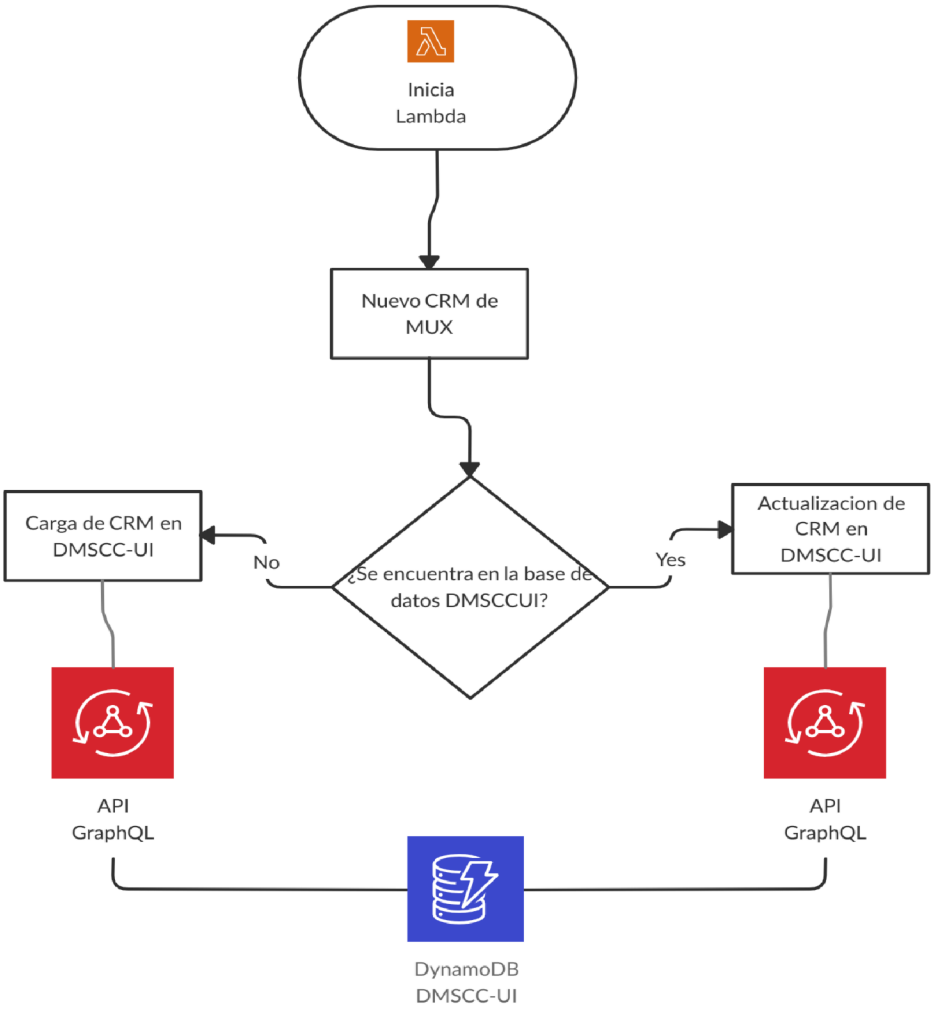


Figura #84: Diagrama de Flujo muxSyncFunction.
Fuente: Elaboración propia


```
const AWS = require('aws-sdk');
const urlParse = require('url').URL;
const https = require('https');
const appsyncUrl = process.env.API_DMSCCUI_GRAPHQLAPIENDPOINTOUTPUT;
const endpoint = new urlParse(appsyncUrl).hostname.toString();
const req = new AWS.HttpRequest(appsyncUrl, region);

module.exports = {
};
function getApiListApps(token) {
}

function createApp(app) {
  const createAppMutation = `mutation CreateApp($input: CreateAppInput!, $condition: ModelAppConditionInput) {
      createApp(input: $input, condition: $condition) {
        __typename
        id
        key_name
      }
    }`;
  req.method = 'POST';
  req.headers.host = endpoint;
  req.headers['Content-type'] = 'application/json';
  req.body = JSON.stringify({
    query: createAppMutation,
    operationName: 'CreateApp',
    variables: {
      input: {
        key_name: app.key_name,
        build: app.build,
        url: app.url,
        stage: app.stage,
        status: app.status,
        alerts: app.alerts
      }
    }
  });
  const signer = new AWS.Signers.V4(req, 'appsync', true);
  signer.addAuthorization(AWS.config.credentials, AWS.util.date.getDate());

  return new Promise((resolve, reject) => {
    const httpRequest = https.request({ ...req, host: endpoint }, result => {
      var buffers = [];
      result.on('data', data => {
        buffers.push(data);
      });
      result.on('end', function() {
        resolve(JSON.parse(Buffer.concat(buffers).toString()));
      });
      result.on('error', error => {
        console.log(error);
        reject(eror);
      });
    });
    httpRequest.write(req.body);
    httpRequest.end();
  });
}
```

Figura #85: QueryGraphQL
Fuente: Elaboración propia

Finalizado el desarrollo a nivel de código de programación, al igual que con la *Lambda* creada en la sección 2.6 “Lectura de métricas, desarrollo de lambdas”, se le asigna el Scheduler, para que la misma pueda ser invocada automáticamente cada 1 hora y así mantener los datos sincronizados permitiendo realizar las lecturas de métricas correspondientes.

A continuación, en las Figura #86 se muestra el Scheduler desde el panel de control de AWS.

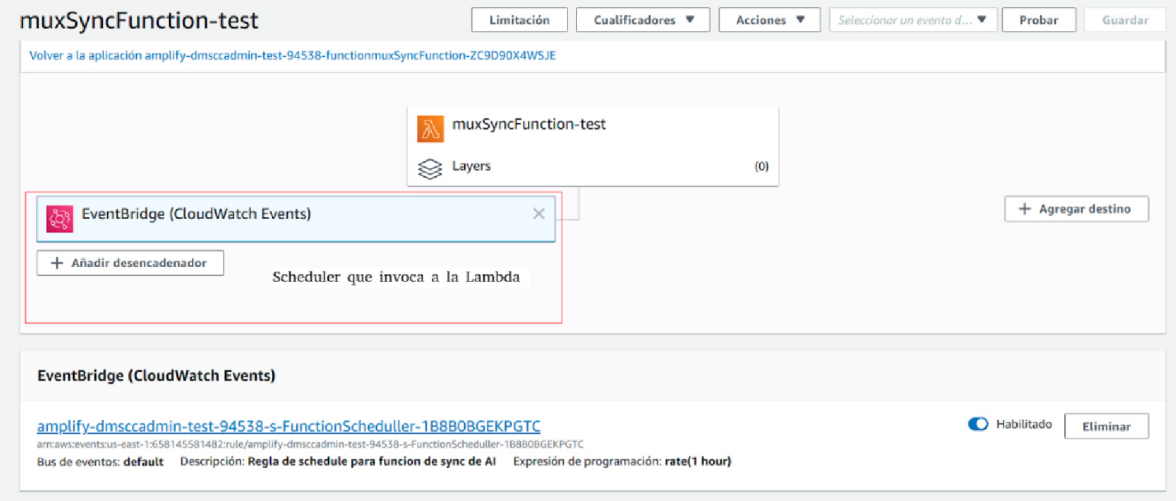


Figura #86: Scheduler desde AWS console

Fuente: Elaboración propia

8. Puesta en producción

Una vez terminado el desarrollo de las *Lambdas* que se encargan de insertar los datos, la aplicación ya está lista para el uso.

En primera instancia, se decidió crear dos ambientes, uno de producción y otro de testing, en este último se utilizará para probarlas funcionalidades y en el caso de encontrar fallos o bugs, reportarlos para luego solucionarlos.

Para la implementación, *Amplify* incorpora "Continuous Deployment (CD)", éste es un concepto que permite un proceso de lanzamiento de software utilice pruebas automatizadas. Adicionalmente valida si los cambios en una base de código son correctos y estables para implementación automática en un ambiente de producción.

Amplify provee un flujo de trabajo basado en Git para implementar y alojar aplicaciones "Serverless FullStack". Una aplicación Serverless FullStack está conformada por un backend creado con recursos de la nube, como las API GraphQL o REST, almacenamiento de archivos y datos, y un frontend creado con frameworks de aplicaciones de una sola página, como React, **Angular** o Vue.

La funcionalidad de las aplicaciones Serverless FullStack suelen estar distribuidas entre el código de frontend, que se ejecuta en el navegador, y la lógica de negocio del backend, que se ejecuta en la nube. Esto hace que la implementación de aplicaciones sea compleja y que insuma tiempo, ya que se deben coordinar con atención los ciclos de lanzamiento para garantizar que el frontend y el backend sean compatibles, y que las características nuevas no afecten negativamente los entornos de producción de los clientes.

Amplify agiliza el ciclo de lanzamiento de aplicaciones mediante el suministro de un flujo de trabajo simple para implementar aplicaciones "Serverless Fullstack" Solo debe conectar el repositorio de código de su aplicación (GitHub) a la consola de *Amplify* y las modificaciones que se realicen en el frontend y el backend se implementarán en un único flujo de trabajo en cada actualización de código.

Esto significa que el desarrollador puede implementar nuevas funcionalidades de forma rápida y sencilla con solo realizar un "push" en su repositorio (GitHub en el caso de este proyecto), el cual como se muestra en las figuras se encuentra vinculado con su correspondiente ambiente de backend.

En las Figura #87 se observan los ambientes master y testing establecidos.

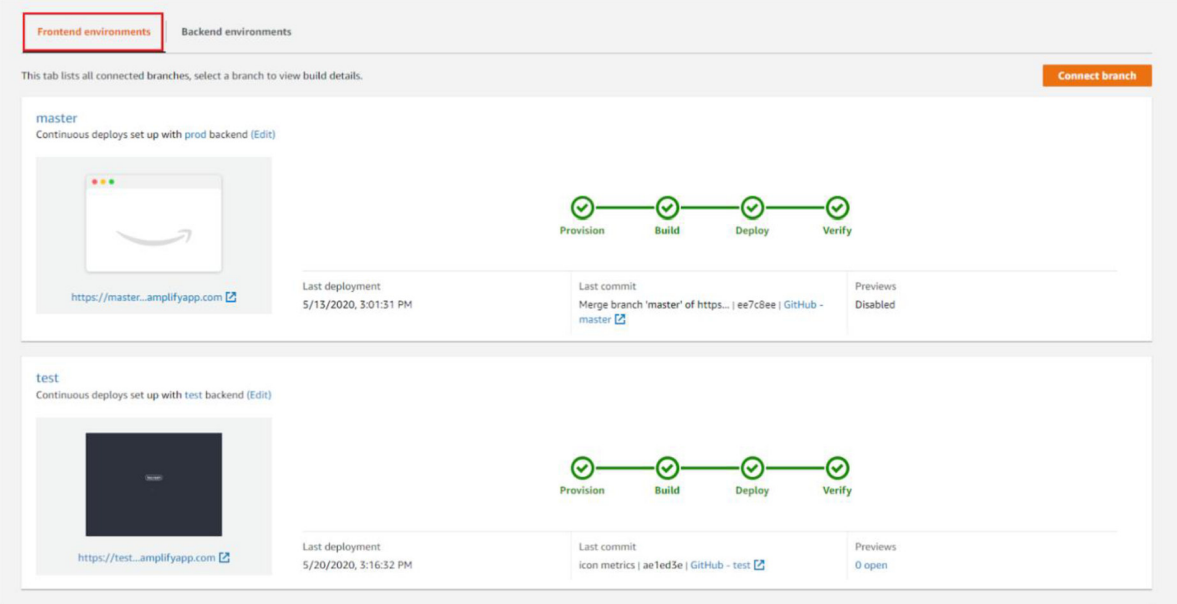


Figura #87: Ambientes de CD

Fuente: Elaboración propia

9. Errores

Durante el desarrollo e implementación de este proyecto surgieron varios errores puntuales y/o defectos de la tecnología propiamente dicha ya que en el momento que fue implementada aún no contaba con sus correspondientes parches.

1. Imposibilidad de agregar Scheduler.

Según lo manifestado en la sección 2.6 “Lectura de métricas, desarrollo de lambdas”, para implementar un evento que se encargue de invocar una *Lambda* “aiSynuncFunction” cada cierto periodo de tiempo, fue necesario recurrir a los tutores de la organización para interiorizar en la tecnología SAM de AWS, y así tener una introducción al tema y lo que abarca éste. Esto se debió al momento en que la versión de *Amplify* utilizada no admitía la configuración automática de dicha funcionalidad. Sin embargo, a partir de la versión 4.18 esta funcionalidad fue añadida al flujo de trabajo de *Amplify* CLI, por lo que en la actualidad desarrollar lo explicado en la sección 2.6 “Lectura de métricas, desarrollo de lambdas” ya no es necesario.

2. Funcionalidad de Continuos Deployment.

Otro inconveniente fue implementar el “Continuos Deployment” debido a que el método de autenticación de usuarios a la aplicación de este proyecto es por Google. Esto conlleva a realizar un redireccionamiento previo al ingresar a la aplicación hacia el HostedUI, lugar donde se obtiene un TOKEN válido provisto por Google para luego conceder el acceso a DMSCC-UI.

Las URL de redireccionamiento son definidas al instante de crear la categoría Auth desde la consola y al momento de crearlas se utilizaron las direcciones de http://localhost, luego en la puesta en producción, fue necesario cambiar estas URL de redireccionamiento de Login y Logout. En este último paso fue donde se encontró un fallo de *Amplify* CLI el cual no permitía modificar estas

URL, por lo que fue necesario modificar manualmente el archivo donde se encontraban estas direcciones, lo que no se considera como una práctica acorde.

Sin embargo, en las versiones posteriores de *Amplify* CLI, este error fue solucionado ya que este problema fue reportado por los distintos desarrolladores en los foros de *Amplify* y AWS.

3. Nombre de aplicación

Por último, otro error de configuración importante fue al momento de asignar los nombres de los entornos de desarrollo de backend de DMSCC-UI. En un principio, al crear este proyecto con *Amplify*, en el flujo de configuraciones tales como el nombre que llevaría la aplicación para identificarlo en la nube, éste no advertía los inconvenientes que trajo consigo incluir nombres que contengan el carácter "-".

En un principio a la aplicación se la identificó como "DMSCC-ADMIN", con el carácter "-" separando ambas palabras y esto llevó a errores al instante en que se crearon los ambientes con nombres como "test", "dev", "prod". Esto fue debió a que los nombres son comunes en los entornos que utilizan las demás aplicaciones de la empresa almacenados en la nube de AWS y este caracter ("-") llevó a Cloudformation a crear recursos duplicados al momento de subir a la nube dichos ambientes

La solución fue modificar desde la línea de comandos de *Amplify* el nombre del proyecto, de DMSCC-ADMIN a "dmscAdmin" (quitando el "-"). Adicionalmente, en las últimas versiones de *Amplify* CLI este error fue solucionado ya que no se permiten incluir este tipo de caracteres.

3. CONCLUSIONES

El objetivo principal de esta práctica profesional supervisada, fue el de desarrollar una aplicación que permitiera centralizar los datos y métricas de los Sistemas DMSCC y DMS. El segundo objetivo que se planteó en la introducción, era facilitarla monitorización de estos sistemas, ya que es de gran importancia para el área de infraestructura de la organización, poder brindar una capacidad de

respuesta ante fallos mucho mayor a la que se tenía previamente a la implementación de DMSCC-UI. Esto conlleva indirectamente a generar un producto de alta disponibilidad o de rápida recuperación ante fallos y cumpliría con el tercer objetivo propuesto.

Finalmente, terminado el desarrollo de DMSCC-UI, estos objetivos fueron alcanzados casi en su totalidad como bien se anticiparon en el resultado de las vistas indicadas en la sección "**2.5. Listado, ABM y filtrado frontend.**", en las que se puede observar el detalle completo de los DMSCC y los estados de los mismos. Sin embargo, una de las funcionalidades propuestas al inicio este proyecto, y que correspondería con el cuarto objetivo, la cual incluía dar de alta nuevos DMSCC desde la aplicación DMSCC-UI, no fue desarrollada ya que requería una participación y demanda importante del área de producto lo cual no era posible de brindar en dicho momento. Por ende, ésta es una de las funcionalidades a seguir trabajando en el futuro, conjuntamente con los desarrolladores de Tecnom, que permita mejorar y fortalecer esta herramienta.

Adicionalmente, otro de los objetivos planteados, fuera del esquema de DMSCC-UI, fue el de introducir el framework de AWS "*Amplify*" como una nueva herramienta del Stack tecnológico de Tecnom. El desarrollo de DMSCC-UI sirvió como prueba y exploración de la mayoría de las funcionalidades de AWS "*Amplify*" y dejó en evidencia las facilidades a la hora de programar, que brinda esta tecnología, por lo que fue rápidamente implementada por otro de los equipos de Tecnom, al ver los resultados de "*Amplify*" sobre DMSCC-UI.

1. REFLEXIÓN SOBRE LA PRÁCTICA PROFESIONAL SUPERVISADA COMO ESPACIO DE FORMACIÓN

Durante el transcurso de esta práctica profesional, tuve la oportunidad de implementar métodos ágiles adquiridos a lo largo de la carrera de Ingeniería en Informática orientados a proyectos de software, los que permiten trabajar de manera colaborativa y recibir una retroalimentación constante en el proceso de desarrollo. Al mismo tiempo, en mi caso particular, me brindó la posibilidad de tener mi primera experiencia laboral orientada al desarrollo de software, donde

además de adquirir nuevos conocimientos técnicos sobre las tecnologías que se trataron, aprendí a trabajar de manera organizada. Esto se reflejó al establecer objetivos de corto y largo plazo según los requerimientos que fueran surgiendo a lo largo del proyecto, implementando y comprendiendo el significado de aplicar las buenas prácticas en el proceso de desarrollo de software.

Finalmente, en mi opinión, las prácticas profesionales supervisadas son imprescindibles para la formación final de un Ingeniero en informática, ya que me permiten llevar a cabo en el día a día de trabajo los conocimientos teóricos adquiridos durante la carrera, y obtener la experiencia necesaria. En mi caso particular, dicha práctica, posibilitó insertarme en el mercado laborar, sin dejar de lado el desarrollo personal, lo cual implica el fortalecimiento de valores sociales como la comunicación, la responsabilidad, la honestidad a la hora de tratar con los distintos recursos humanos de la organización.

– Bibliografía

• Online
•Amazon CognitoDeveloperGuide Recuperado de https://docs.aws.amazon.com/es_es/cognito/latest/developerguide/google.html.

• Online
• Amazon Guía para desarrolladores Versión de API 2012-08-10 Recuperado de https://docs.aws.amazon.com/es_es/amazondynamodb/latest/developerguide/dynamodb-dg.pdf#workbench.

• Online
•Amplify Docs Angular Framework. Recuperado de https://docs.amplify.aws/start/q/integration/angular.

• Online
• AWS Amplify Guía del Usuario de la consola. Recuperado de https://docs.aws.amazon.com/es_es/amplify/latest/userguide/getting-started.html

• Online
• AWS Identity and Access MangementGuía de usuario. Recuperado de https://docs.aws.amazon.com/es_es/IAM/latest/UserGuide/introduction.html

• Online
• Introduction to GraphQL. Recuperado de https://graphql.org/learn/