

Rapport projet S51/52

KHENISSI Tejeddinne
WELTY Alexandre

Année 2024/2025

Sommaire

I. Introduction	3
II. Application théorique	4
A. Schéma théorique	4
B. Une application serverless	4
C. Liste des services utilisés	4
1. Notre application	4
D. Description des services utilisés (pas corriger)	5
1. Route 53	5
2. Amplify	5
3.	5
4. Cognito	5
5. AppSync	6
6. ElasticSearch	6
7. DynamoDB	6
8. Lambda	6
9. API Gateway	6
10. PrivateLink	6
11. Network Load Balancer (NLB)	6
12. Application Load Balancer (ALB)	6
13. Elastic Container Registry (ECR)	7
14. Amazon Elastic Compute (ECS)	7
15. Fargate	7
16. DAX (DynamoDB Accelerator)	7
17. Simple Queue Service (SQS)	7
18. PinPoint	7
19. Amazon Simple Notification Service (SNS)	7
20. Kinesis Data Firehose	7
21. S3	7
22. Glue	8
23. Athena	8
24. QuickSight	8
25. CodeCatalyst	8
E. Fonctionnement technique	8
1. frontend, AppSync, Premier Backend	8
2. passage du frontend au backend de l'application	10
3. backend de l'application	10
4. Analyse des données	11
5. construire et déployer l'application	11
F. Frontend alternative	12
G. Backend alternative	12
III. Application pratique	13
A. Schéma pratique première version 1	13
B. Schéma pratique deuxième version 2	14

C. Liste des services utilisés	14
D. Fonctionnement de l'application	14
1. Application client	14
2. Application restaurant	19
E. Description des services utilisés	20
1. DynamoDB	20
a) Explication technique	20
2. Cognito	20
a) Explication technique	20
3. lambda	20
a) getRestaurants	20
b) InsertReservation	21
c) getMyReservation	21
4. API Gateway	21
a) Explication technique	21
5. CloudFront	22
a) Explication technique	22
6. S3 Bucket	22
a) Explication technique	22
7. SNS	23
a) Explication technique	23
8. IAM	23
a) Explication technique	23
9. Explication du code	23
a) Cognito	23
F. identifiant de connexion	27

Un dossier imgs contient les différentes images utilisées.

Notre architecture théorique : imgs/architecture/serverlessAppDemoV2.drawio.png

Notre architecture pratique : imgs/architecture/serverlessAppFull.drawio.png

I. Introduction

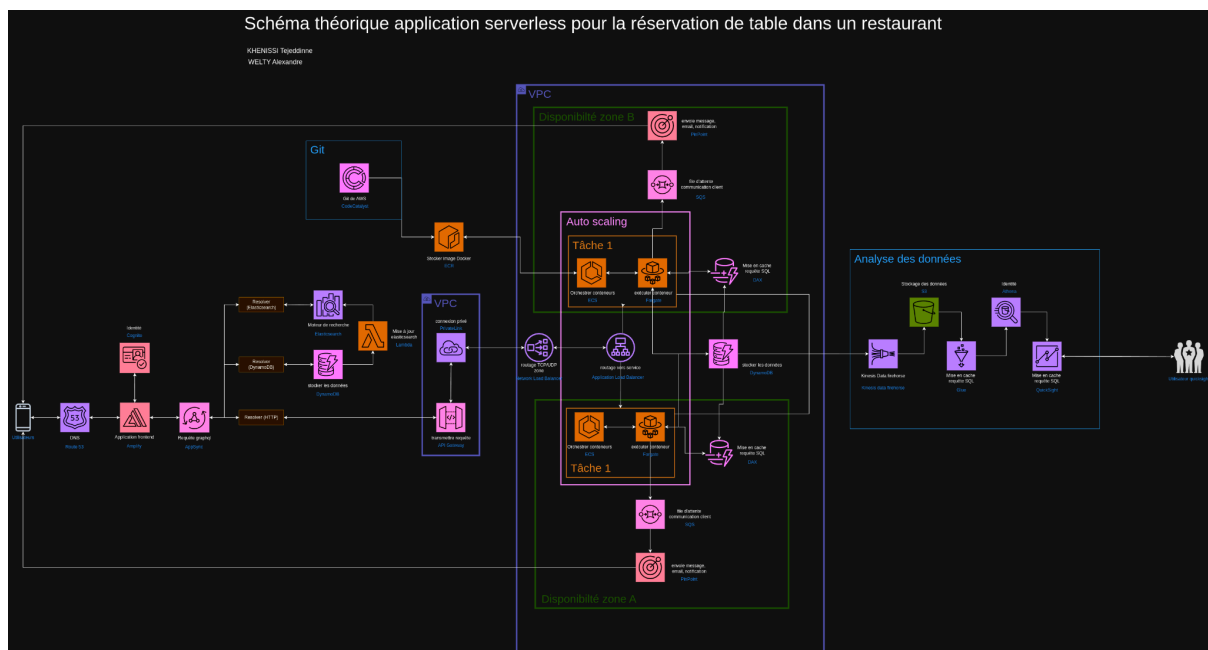
La gastronomie française, réputée à travers le monde pour son raffinement et son excellence, est le fruit d'un savoir-faire unique et d'une tradition culinaire séculaire. Pendant des décennies, le Guide Michelin, également connu sous le nom de "Guide Rouge", a été la référence incontestée en matière de critique gastronomique, guidant amateurs et passionnés vers les meilleures tables du pays. Pourtant, avec l'évolution des modes de vie et l'avènement de la révolution numérique, les habitudes des consommateurs ont considérablement changé. La nouvelle génération, davantage tournée vers des solutions instantanées et accessibles en ligne, connaît peu ou pas ce prestigieux guide autrefois emblématique. Face à ce bouleversement des usages, les restaurateurs ont dû s'adapter, en intégrant les services de réservation en ligne, afin de répondre aux attentes d'une clientèle plus connectée et exigeante. Cette transformation témoigne de l'évolution de la gastronomie française, qui s'efforce de conjuguer tradition et modernité pour continuer à séduire un public en perpétuelle mutation.

Face à l'évolution des comportements des consommateurs et à l'importance croissante des outils numériques, nous avons décidé de mettre en place un service de réservation en ligne pour les restaurants. Cette application web a pour objectif de faciliter la recherche, la réservation et la gestion des tables, aussi bien pour les clients que pour les restaurateurs. D'un côté, le client peut se connecter sur le site pour rechercher des restaurants correspondant à ses critères, s'inscrire, et une fois son inscription validée, il a la possibilité de réserver une table en ligne. Il peut également consulter l'historique de ses réservations et suivre ses réservations en cours, tout en ayant accès aux informations détaillées des restaurants. De l'autre côté, les restaurateurs disposent d'un espace dédié qui leur permet de s'inscrire sur l'application et de gérer leur établissement. Ils ont la possibilité de consulter la liste des clients ayant réservé dans leur établissement et de mettre à jour les informations de leur restaurant. De plus, un service de notifications est disponible afin de les avertir en temps réel lorsqu'une réservation est effectuée. Cela leur permet de mieux organiser l'accueil des clients et d'optimiser la gestion de leur établissement.

Ce projet vise ainsi à moderniser la relation entre les clients et les restaurateurs, en offrant une solution rapide, pratique et connectée, tout en garantissant une expérience utilisateur fluide et intuitive pour tous les acteurs du secteur de la restauration.

II. Application théorique

A. Schéma théorique



Le schéma présenté ci-dessous illustre l'architecture technique que nous avons conçue pour notre application de réservation de tables dans les restaurants. Il reflète notre choix d'une approche serverless, intégrant de multiples services AWS pour répondre efficacement aux besoins de performance, de scalabilité, et de gestion des données. Chaque composant de ce schéma joue un rôle clé dans le traitement des requêtes utilisateurs, le stockage des informations, et l'analyse des données, tout en assurant une disponibilité et une sécurité optimales. Cette représentation visuelle permet de comprendre comment les différents services interagissent entre eux pour offrir une expérience utilisateur fluide et réactive.

B. Une application serverless

Nous avons opté pour une architecture serverless via AWS pour simplifier le développement de notre application de réservation en ligne. AWS gère automatiquement l'infrastructure, garantissant scalabilité et haute disponibilité, tout en nous facturant uniquement les ressources utilisées. Les principaux avantages incluent une scalabilité automatique, une gestion simplifiée et une réduction des coûts. Toutefois, cela entraîne une dépendance vis-à-vis d'AWS, des coûts parfois difficiles à estimer et un débogage plus complexe. (listes non exhaustive)

C. Liste des services utilisés

1. Notre application

Afin de réaliser notre application nous avons utilisé les services suivants :

- Route 53

- Amplify
- Cognito
- AppSync
- Elasticsearch
- DynamoDB
- Lambda
- API Gateway
- PrivateLink
- Network load balancer (NLB)
- Active load balancer (ALB)
- Elastic Container Registry (ECR)
- Amazon Elastic Compute (ECS)
- fargate
- Dax
- Simple queue service SQS
- PinPoint
- Amazon Simple Notification (SNS)
- Kinesis data firehose
- S3
- Glue
- Athena
- QuickSight
- Code Catalyst

D. Description des services utilisés (pas corriger)

1. Route 53

Route 53 est un service de DNS (Domain Name System) scalable et hautement disponible, qui permet de gérer la résolution de noms de domaine et la routage du trafic utilisateur vers les applications AWS. Il propose également des fonctionnalités de routage basées sur la latence, la géolocalisation et la santé des endpoints.

2. Amplify

3.

Amplify est un ensemble d'outils et de services permettant de développer et de déployer facilement des applications web et mobiles. Il fournit des services de backend (API, authentification, stockage) et des bibliothèques front-end pour créer des applications full-stack rapidement.

4. Cognito

Cognito permet de gérer l'authentification, l'autorisation et les utilisateurs pour les applications web et mobiles. Il prend en charge les connexions via des identités fédérées (Google, Facebook) ainsi que les utilisateurs créés dans un pool d'identité propre à Cognito.

5. AppSync

AppSync est un service de gestion GraphQL qui simplifie le développement d'API et la synchronisation des données entre plusieurs sources, tout en gérant les abonnements en temps réel, la mise en cache et les résolutions d'autorisations.

6. Elasticsearch

OpenSearch Service est un service entièrement géré qui permet d'effectuer des recherches, des analyses et de la visualisation de données en temps réel. Il est couramment utilisé pour la surveillance, l'analyse de logs, et les recherches full-text.

7. DynamoDB

DynamoDB est une base de données NoSQL entièrement gérée offrant une latence à une seule milliseconde. Elle est idéale pour les applications nécessitant un accès rapide et flexible à de grandes quantités de données, sans gestion de la capacité ou de la mise à l'échelle.

8. Lambda

Lambda est un service de calcul sans serveur qui exécute du code en réponse à des événements, tels que des modifications dans une base de données ou des requêtes HTTP. Il permet de créer des applications en se concentrant uniquement sur le code, sans gérer les serveurs sous-jacents.

9. API Gateway

API Gateway permet de créer, de publier, de gérer et de sécuriser des API à n'importe quelle échelle. Il prend en charge les API RESTful et WebSocket, ainsi que l'intégration avec d'autres services AWS comme Lambda et DynamoDB.

10. PrivateLink

PrivateLink permet de connecter de manière privée des VPC (Virtual Private Cloud) à d'autres services AWS ou à des services tiers sans exposer le trafic sur Internet. Il sécurise la communication entre les VPC et réduit la surface d'attaque.

11. Network Load Balancer (NLB)

Network Load Balancer distribue le trafic réseau en fonction d'adresses IP et est conçu pour gérer des millions de requêtes à faible latence. Il est particulièrement adapté aux charges de travail nécessitant une haute performance réseau.

12. Application Load Balancer (ALB)

Application Load Balancer distribue le trafic HTTP/HTTPS en fonction de l'URL, des en-têtes et des cookies. Il est conçu pour les applications web basées sur les microservices ou les architectures conteneurisées.

13. Elastic Container Registry (ECR)

Elastic Container Registry (ECR) est un registre Docker privé qui stocke, partage et déploie des conteneurs Docker pour faciliter la gestion des conteneurs dans AWS.

14. Elastic Compute (ECS)

ECS (Elastic Container Service) est un service de gestion de conteneurs qui permet d'exécuter, de gérer et de mettre à l'échelle des conteneurs Docker sur des clusters EC2 ou Fargate, facilitant la gestion des applications conteneurisées.

15. Fargate

Fargate est un moteur de calcul pour ECS et EKS (Kubernetes) qui permet d'exécuter des conteneurs sans gérer les serveurs. Il s'occupe de la gestion de l'infrastructure, offrant un déploiement serverless de conteneurs.

16. DAX (DynamoDB Accelerator)

DynamoDB Accelerator (DAX) est un cache en mémoire pour DynamoDB qui améliore considérablement la vitesse d'accès aux données, en réduisant les temps de latence en lecture à des microsecondes.

17. Simple Queue Service (SQS)

SQS (Simple Queue Service) est un service de file d'attente entièrement géré qui permet de découpler les composants des applications et d'assurer une communication asynchrone entre différents services ou microservices.

18. PinPoint

Pinpoint est un service de marketing et d'engagement client qui permet d'envoyer des notifications, des e-mails et des messages SMS, et de suivre les interactions utilisateurs. Il est utilisé pour les campagnes marketing et la communication personnalisée.

19. Simple Notification Service (SNS)

SNS (Simple Notification Service) est un service de messagerie entièrement géré qui facilite la livraison de messages entre applications ou vers des abonnés, par e-mail, SMS ou via d'autres protocoles.

20. Kinesis Data Firehose

Kinesis Data Firehose est un service permettant de capturer, de transformer et de charger des flux de données en temps réel vers des destinations comme S3, Redshift, ou Elasticsearch, pour l'analyse et la visualisation des données.

21. S3

S3 (Simple Storage Service) est un service de stockage d'objets permettant de stocker et de récupérer n'importe quelle quantité de données à l'échelle mondiale. Il est utilisé pour le

stockage de fichiers, la sauvegarde, l'archivage et comme source de données pour les applications.

22. Glue

Glue est un service d'intégration de données entièrement géré qui facilite l'extraction, la transformation et le chargement (ETL) des données. Il permet de préparer les données pour l'analyse et supporte des sources de données multiples.

23. Athena

Athena est un service d'interrogation interactif qui permet de lancer des requêtes SQL sur des données stockées dans S3. Il est utilisé pour analyser des données non structurées sans besoin de les charger dans une base de données.

24. QuickSight

QuickSight est un service de visualisation de données permettant de créer des rapports interactifs et des tableaux de bord. Il offre des fonctionnalités d'analyse prédictive et d'intégration avec d'autres services AWS.

25. CodeCatalyst

CodeCatalyst est un ensemble de services de développement cloud-native qui aide les développeurs à concevoir, à tester, à surveiller et à déployer des applications. Il inclut des pipelines CI/CD, la gestion de code source, et des outils de collaboration pour faciliter le développement agile.

E. Fonctionnement technique

1. frontend, AppSync, Premier Backend

Lorsqu'une personne souhaite accéder à notre application, elle y parvient via **Route 53**, qui sert de gestionnaire de noms de domaine (DNS). Route 53 dirige le trafic utilisateur vers le contenu de l'application en fonction des configurations DNS et des règles de routage définies. Cela inclut la répartition du trafic entre plusieurs ressources, dont CloudFront, pour offrir la meilleure performance et disponibilité possible.

Ensuite, cette personne peut interagir avec le site web (contenu statique) via **AWS Amplify**. Amplify est un service complet qui gère la distribution de contenu statique de manière automatisée. Il intègre par défaut **CloudFront** comme réseau de distribution de contenu (CDN) pour mettre en cache les fichiers statiques sur des points de présence à travers le monde, réduisant ainsi les temps de chargement et améliorant les performances globales de l'application. Amplify automatise également d'autres configurations comme l'hébergement et la gestion des certificats SSL/TLS pour garantir une connexion sécurisée.

Lorsque l'utilisateur souhaite s'authentifier pour accéder aux fonctionnalités protégées de l'application, il utilise **Cognito**. Cognito gère l'inscription, la connexion, et la gestion des sessions utilisateurs via des pools d'utilisateurs. Lors de l'authentification, un token JWT (JSON Web Token) est généré par Cognito. Ce token contient des informations

sur l'utilisateur (comme son identifiant et ses permissions) et est utilisé pour prouver son identité lors de l'accès à d'autres services. Ce token JWT est ensuite stocké sur le client et transmis dans les en-têtes de chaque requête.

Les utilisateurs authentifiés peuvent alors interagir avec les services back-end en effectuant des appels API via **AppSync**, un service GraphQL qui facilite la gestion de la communication entre le front-end et les sources de données back-end. Grâce à AppSync, les utilisateurs peuvent effectuer des opérations comme des requêtes, des mutations, et des abonnements. AppSync utilise les informations du token JWT pour s'assurer que seuls les utilisateurs authentifiés peuvent effectuer ces opérations, et les requêtes sont traitées de manière sécurisée. Dans notre application, nous avons trois types de résolveurs principaux :

- **Résolveur DynamoDB**
 - Ce résolveur est connecté à une base de données **DynamoDB**, utilisée pour stocker des informations comme la liste des clients et la liste des restaurants. Chaque fois qu'une opération de type mutation est effectuée via AppSync (par exemple, l'ajout d'un nouveau client ou d'un restaurant), le résolveur DynamoDB se charge de mapper cette opération vers une requête DynamoDB. Cela signifie que les données sont ajoutées, modifiées ou supprimées dans la table DynamoDB en fonction de l'opération GraphQL effectuée.
 - Lorsque des modifications de données (telles que l'ajout d'un nouveau restaurant ou d'un client) sont effectuées dans DynamoDB, un **Lambda** est automatiquement déclenché pour synchroniser ces modifications avec **Elasticsearch Service (ES)**.
- **Résolveur Elasticsearch**
 - AppSync utilise un **résolveur spécifique à Elasticsearch** pour interagir directement avec le service ES. Cela permet de traduire les requêtes GraphQL de type "query" vers des requêtes de recherche Elasticsearch. Les résultats renvoyés par Elasticsearch sont ensuite mappés et formatés pour être conformes au schéma GraphQL défini dans AppSync.
 - Ce résolveur est utilisé chaque fois qu'une recherche est effectuée dans l'application. Par exemple, un client peut rechercher un restaurant par nom, emplacement ou catégorie, et le résolveur Elasticsearch se chargera de faire correspondre les données de l'index ES aux requêtes de l'utilisateur.
- **Résolveur HTTP**
 - Un troisième type de résolveur utilisé par AppSync est le **résolveur HTTP**, qui permet de transmettre des requêtes à d'autres services via **API Gateway**. Cela permet de communiquer avec des services externes ou d'exposer des fonctionnalités spécifiques via une API REST.
 - Par exemple, si une action utilisateur nécessite de récupérer des informations à partir d'un autre service non géré par DynamoDB ou Elasticsearch, le résolveur HTTP peut envoyer la requête correspondante via API Gateway, récupérer la réponse, et la transmettre à l'utilisateur final. Ce mécanisme est idéal pour intégrer des services tiers ou pour étendre les fonctionnalités de l'application.

2. passage du frontend au backend de l'application

Le résolveur HTTP dans AppSync envoie la requête HTTP vers **API Gateway**, qui sert de point d'entrée centralisé pour toutes les requêtes API de l'application. Ce point d'entrée gère l'authentification, l'autorisation, et applique les contrôles de sécurité nécessaires avant de transmettre la requête au service backend approprié. L'API à laquelle la requête est transmise est hébergée au sein d'un **VPC**, ce qui signifie qu'elle n'est pas directement accessible depuis internet. Cela ajoute un niveau de sécurité supplémentaire, car les ressources du backend ne sont exposées qu'au réseau interne d'AWS. Pour que **API Gateway** puisse communiquer avec cette API située dans un autre VPC, **PrivateLink** est utilisé. PrivateLink encapsule les connexions entre API Gateway et le service backend, ici **ECS sur Fargate**, pour assurer une transmission sécurisée du trafic entre les VPC.

3. backend de l'application

Le NLB est configuré pour distribuer le trafic entrant vers les services backend. Il utilise des **intégrations privées** et est associé à des points de terminaison spécifiques. Chaque point de terminaison est configuré avec un port dédié attribué à un service précis. Le NLB est réparti sur plusieurs **zones de disponibilité** afin de garantir une haute disponibilité et une tolérance aux pannes. Le trafic est acheminé vers des **Application Load Balancers (ALB)** situés dans des sous-réseaux privés.

Le ALB reçoit les requêtes du NLB et les redirige vers le **cluster ECS** (Elastic Container Service) sur **Fargate** en fonction des règles de routage définies (par exemple, en fonction de l'URL, du chemin d'accès, ou de l'en-tête de la requête). Chaque ALB est configuré pour rediriger le trafic vers le service ou la tâche approprié exécuté sur ECS, répartissant ainsi la charge de manière équilibrée sur les conteneurs du cluster.

ECS gère le déploiement des conteneurs et l'exécution des tâches ou services. ECS utilise **Fargate** pour exécuter des conteneurs serverless, ce qui signifie qu'il n'est pas nécessaire de gérer les instances sous-jacentes. À chaque fois qu'un conteneur est déployé ou mis à jour, ECS tire la dernière version de l'image Docker du service à partir de **Elastic Container Registry (ECR)**, garantissant que l'application utilise toujours la version la plus récente de son code.

ECS et Fargate sont configurés avec des stratégies d'**auto-scaling** pour s'adapter automatiquement à la charge du trafic. Cela permet de déployer plus de conteneurs lorsque le trafic augmente et de réduire les ressources lorsque la demande diminue, optimisant ainsi les coûts et les performances.

Les conteneurs ECS sur Fargate accèdent à **DynamoDB Accelerator (DAX)**, une couche de cache en mémoire conçue pour DynamoDB. DAX permet de récupérer les informations fréquemment consultées plus rapidement, réduisant ainsi la latence des requêtes de lecture. L'accès à DAX améliore considérablement les performances de l'application, surtout lors des pics de trafic, en réduisant la latence et en renvoyant des réponses aux utilisateurs de manière plus efficace.

Pinpoint est utilisé pour gérer tous les scénarios de communication marketing. Il segmente le public de la campagne, cible les bons clients en fonction de critères définis (par exemple, l'emplacement géographique ou les interactions précédentes), et personnalise les messages avec un contenu pertinent. Pinpoint gère également l'envoi d'e-mails, de SMS ou

de notifications push pour tenir les utilisateurs informés des promotions, des mises à jour, ou d'autres communications marketing.

Certaines tâches nécessitent des notifications ou la communication entre différents services de l'application. Dans ce cas, **Simple Notification Service (SNS)** est utilisé pour envoyer des messages entre les services. SNS peut publier des notifications à d'autres services, déclencher des fonctions Lambda, ou envoyer des notifications aux utilisateurs finaux par e-mail ou SMS.

Autoscaling de groupe ECS/Fargate: Adapte le nombre de conteneurs.

Autoscaling ECS: Adapte le nombre d'instances EC2 (géré automatiquement par AWS) dans un cluster ECS

Autoscaling ALB: Adapte le nombre de cibles recevant le trafic via l'ALB.

4. Analyse des données

L'application collecte des données non structurées (logs, événements, etc.) qui sont transmises en quasi temps réel à Kinesis Data Firehose. Ce service serverless se charge ensuite de les envoyer vers le data lake stocké dans S3. Kinesis Data Firehose simplifie l'ingestion de données, ne nécessitant aucune administration et offrant une tarification à l'usage.

Une fois les données dans S3, Glue entre en jeu. Ce service ETL serverless extrait, transforme et charge les données pour les préparer à l'analyse. Glue utilise le catalogue de données Glue pour stocker les métadonnées (noms de tables, colonnes, etc.). Un crawler Glue explore automatiquement les données dans S3 pour déduire le schéma, mais il est également possible de le définir manuellement.

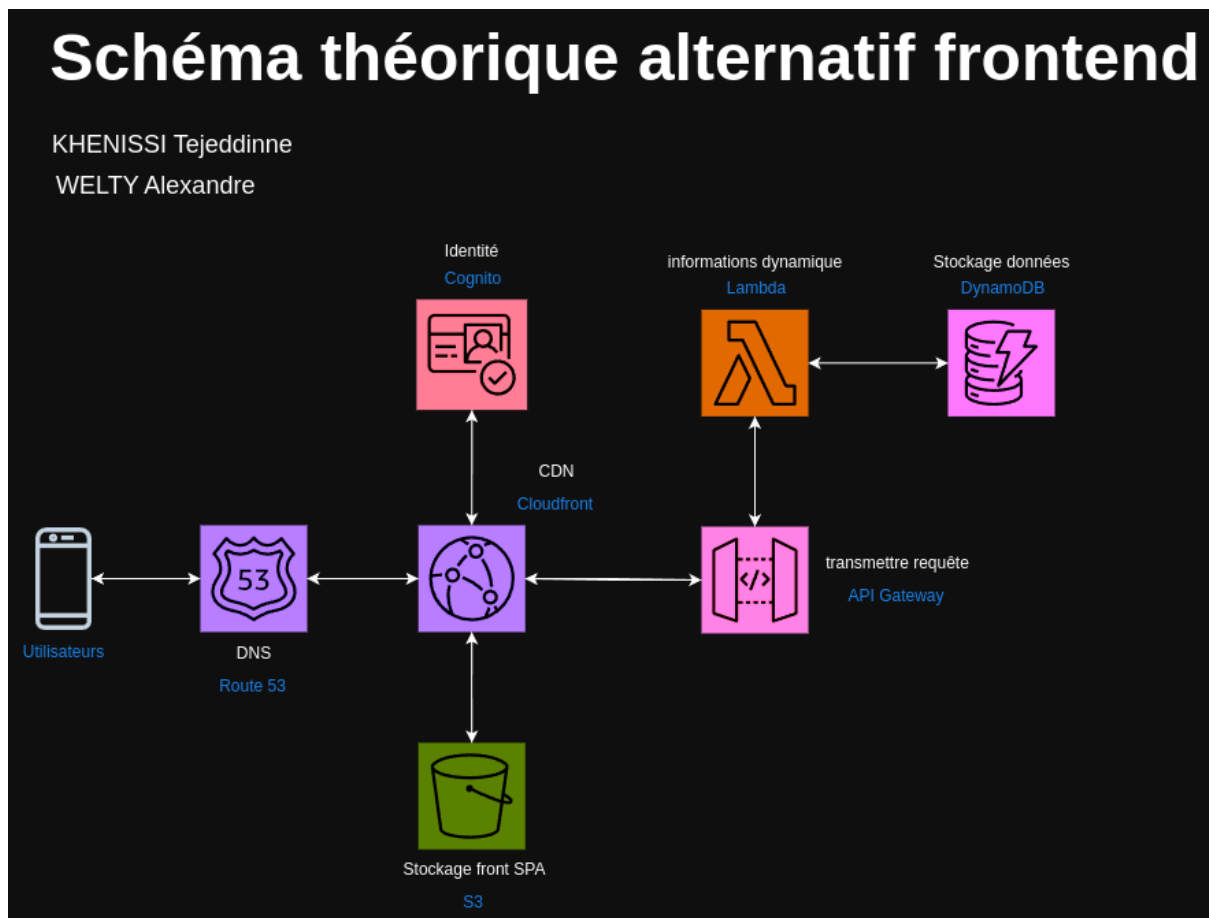
Athena permet ensuite d'interroger facilement les données structurées du data lake. Ce service de requêtes interactif utilise Presto pour exécuter des requêtes SQL sur les données cataloguées par Glue.

Enfin, QuickSight permet de visualiser et d'analyser les données. En se connectant à Athena, QuickSight offre la possibilité de créer des tableaux de bord interactifs et des visualisations pour explorer les données et en tirer des insights.

5. construire et déployer l'application

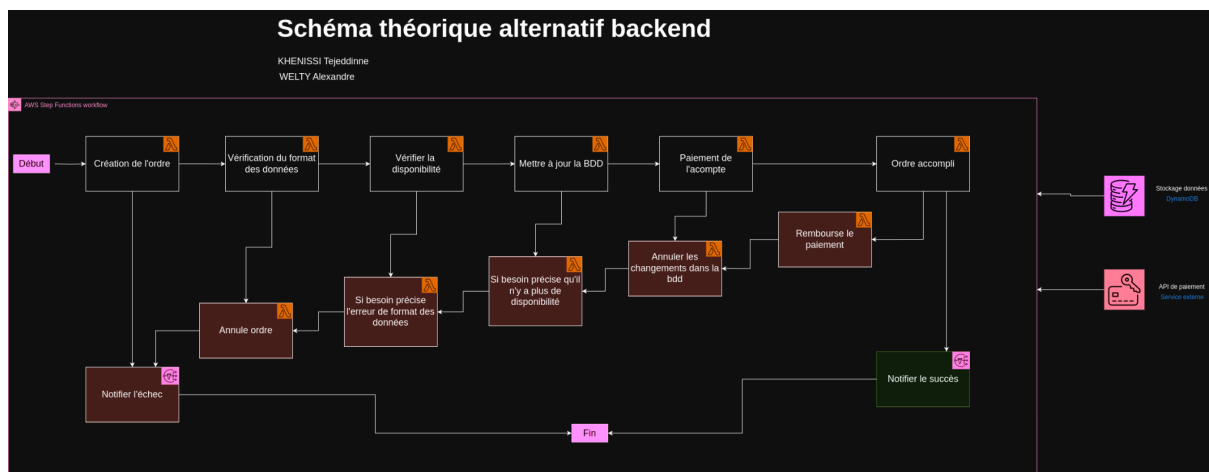
Pour le développement de notre application, nous tirons parti de la puissance de Code Catalyst. Cette plateforme unifiée nous permet de gérer efficacement l'ensemble du cycle de vie du code, en simplifiant le versionning, l'automatisation des builds et la mise en place de pipelines CI/CD.

F. Frontend alternative



On remplace le service Amplify par S3 et Cloudfront.

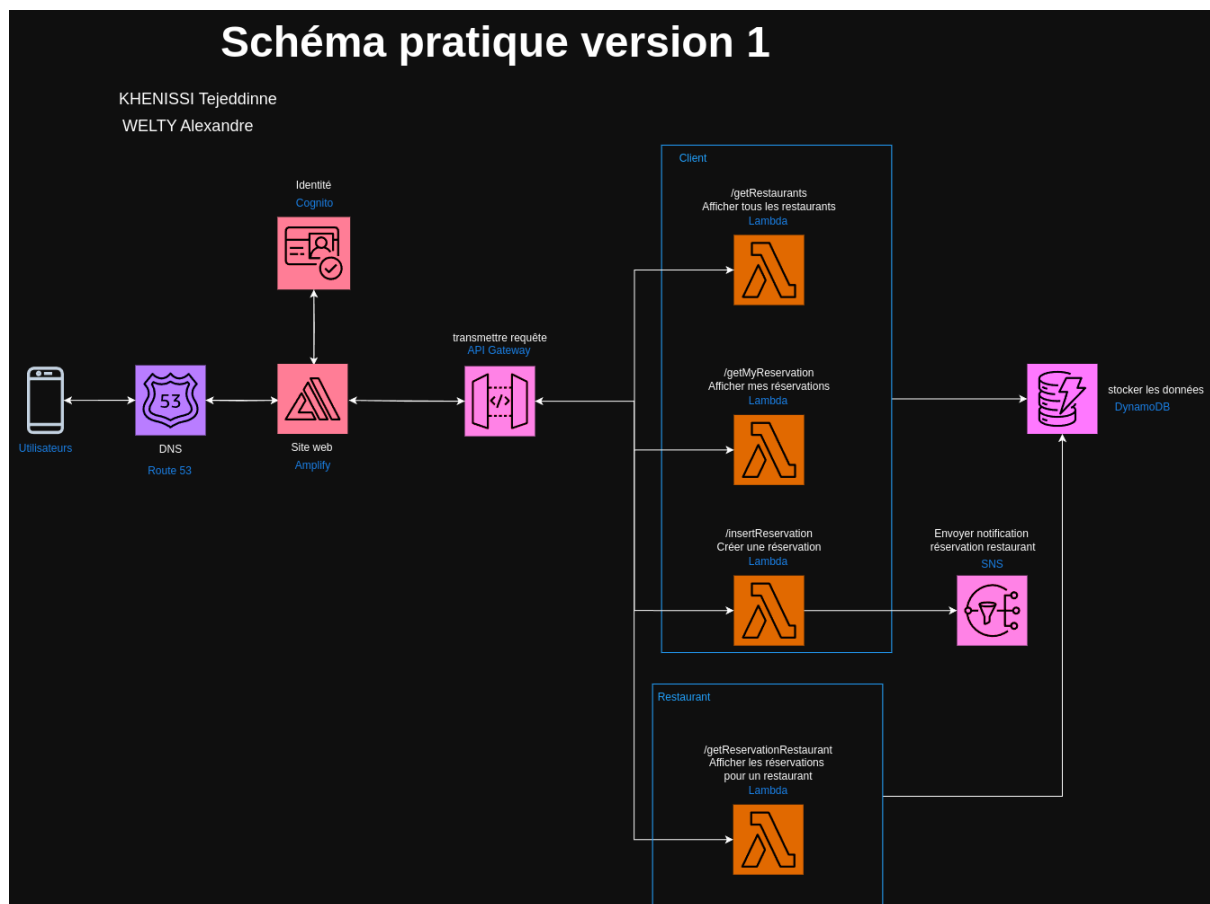
G.Backend alternative



On remplace ECR, ECS, fargate par une suite de fonction Lambda ici le tout est encapsulé dans un service aws step functions.

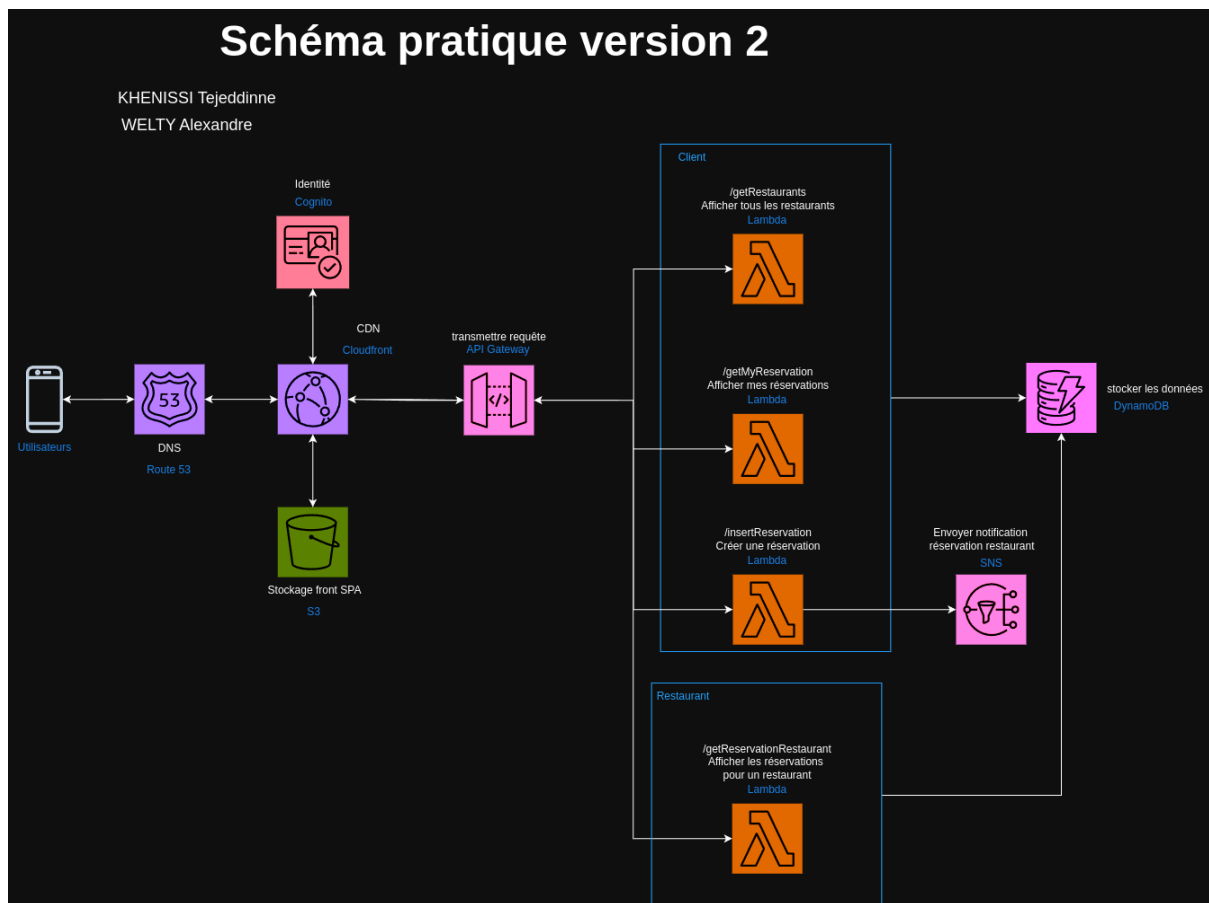
III. Application pratique

A. Schéma pratique première version 1



Nous avons décidé de ne pas implémenter cette architecture car amplify gen2 génère automatiquement l'authentification avec cognito, la création/configuration de l'api, les fonctions lambda, la base de donnée etc.

B. Schéma pratique deuxième version 2



C'est cette version que nous avons décidé d'implémenter.

C. Liste des services utilisés

- Route 53
- CloudFront
- S3
- Cognito
- API Gateway
- Lambda
- DynamoDB
- SNS

D. Fonctionnement de l'application

1. Application client

L'utilisateur se connecte à l'application via un nom de domaine géré par **Route 53**. Route 53 sert de gestionnaire de noms de domaine (DNS) et redirige le trafic vers le service de distribution de contenu approprié, qui est **CloudFront**. **CloudFront** est utilisé comme

réseau de distribution de contenu (CDN). Il met en cache et distribue les fichiers statiques de l'application à partir d'un **S3**. Cela améliore les performances de l'application en fournissant le contenu à partir de points de présence géographiquement proches des utilisateurs. Le bucket S3 stocke le code statique du site web qui est distribué de manière sécurisée par CloudFront. CloudFront prend en charge la gestion des certificats SSL/TLS pour garantir que toutes les connexions entre le client et le CDN sont sécurisées. **Cognito** gère l'authentification des utilisateurs. Lorsque l'utilisateur se connecte ou s'inscrit sur l'application, Cognito génère un token JWT qui contient les informations d'identification et les permissions de l'utilisateur. Ce token est ensuite utilisé pour accéder aux fonctionnalités protégées de l'application. Cognito est configuré avec un pool d'utilisateurs pour gérer les sessions et les identités des utilisateurs. Le frontend interagit avec les services backend via **API Gateway**. API Gateway reçoit les requêtes de l'application (par exemple, afficher la liste des restaurants, consulter les réservations, créer une réservation), vérifie l'authenticité du token JWT généré par Cognito, et transmet ensuite ces requêtes aux fonctions Lambda appropriées. Chaque route dans API Gateway est configurée pour rediriger vers une **fonction Lambda** spécifique en fonction de l'opération demandée (lecture, écriture, etc.). Il y a 3 fonctions lambda qui sont :

- **Afficher tous les restaurants** : Cette Lambda interroge une base de données (par exemple, DynamoDB) ou un service de recherche (Elasticsearch) pour récupérer la liste des restaurants disponibles. Elle renvoie ensuite les résultats à API Gateway, qui les transmet au frontend.
- **Afficher les réservations de l'utilisateur** : Cette Lambda interroge la base de données pour récupérer la liste des réservations de l'utilisateur actuel. Le token JWT est utilisé pour s'assurer que l'utilisateur accède uniquement à ses propres informations.
- **Créer une réservation** : Cette Lambda reçoit les détails de la réservation à créer (restaurant, date, nombre de personnes) et les enregistre dans la base de données. Une fois la réservation effectuée, la fonction Lambda déclenche une action supplémentaire pour envoyer un e-mail au restaurateur concerné.

Inscription :

S'inscrire

Nom:

Email:

téléphone:

Mot de passe:

Mot de passe:

Connexion :

Se connecter

Pas de compte ? clique [ici](#) pour t'inscrire

Application :

Mes informations cognito

Email : welty.alex67@gmail.com

Nom : alexandre 1

Téléphone : +33664568074

Username : 28114360-d0a1-70f3-4474-1011b21af6e4

Se déconnecter

Vue Utilisateur

Ajouter une réservation

Nom restaurant

Email restaurant

Téléphone restaurant

Date réservation (JJ-MM-AAAA / 'midi' ou 'soir')

Enregistrer réservation

Afficher tous les restaurants

Restaurant ID	Nom	Mail	Téléphone
e8018300-b051-702b-977f-85b31506795d	buerehiesel	angryjoker2525@gmail.com	+33666666666
28e17300-e0f1-70c9-f79f-b3af828535a9	be one	memepepe67310@gmail.com	+33668088567
38018330-b001-707d-39e0-34b06b19a1e6	restaurant au saumon	restaurantausaumon@gmail.com	+33388870183

Voir les restaurants

Voir toutes mes réservations

Date	Nom	Mail	Téléphone
11-09-2024 / midi	be one	memepepe67310@gmail.com	+33668088567
12-09-2024 / soir	buerehiesel	angryjoker2525@gmail.com	+33666666666
08-09-2024 / midi	be one	memepepe67310@gmail.com	+33668088567
09-09-2024 / midi	be one	memepepe67310@gmail.com	+33668088567
07-09-2024 / midi	restaurant au saumon	restaurantausaumon@gmail.com	+33388870183
12-09-2024 / midi	buerehiesel	angryjoker2525@gmail.com	+33666666666
06-09-2024 / midi	restaurant au saumon	restaurantausaumon@gmail.com	+33388870183
10-09-2024 / midi	restaurant au saumon	restaurantausaumon@gmail.com	+33388870183

Voir mes réservations

2. Application restaurant

La seule différence est que le restaurant n'a accès qu'à deux fonctions lambda :

- Une fonction lambda pour s'inscrire au service SNS et ainsi recevoir les notifications lors d'une réservation
- Une fonction lambda pour afficher la liste des clients qui ont réservé.

Mes informations cognito

Email : memepepe67310@gmail.com

Nom : be one

Téléphone : +33668088567

Username : 28e17300-e0f1-70c9-f79f-b3af828535a9

Se déconnecter

S'inscrire a sns

S'inscrire

Vue Restaurateur

Voir toutes mes réservations

Date	Nom	Mail	Téléphone
11-09-2024 / midi	alexandre 1	welty.alex67@gmail.com	+33664568074
08-09-2024 / midi	alexandre 1	welty.alex67@gmail.com	+33664568074
09-09-2024 / midi	alexandre 1	welty.alex67@gmail.com	+33664568074
06-09-2024 / soir	alexandre 2	welty.alex.aw.67@gmail.com	+33664568074
07-09-2024 / soir	alexandre 2	welty.alex.aw.67@gmail.com	+33664568074

Voir mes réservations



E. Description des services utilisés

Ici on détaille le processus d'implémentation de l'application client.

1. DynamoDB

a) Explication technique

pour cette application nous avons créé une seule table afin de stocker des réservations. Pour créer une table DynamoDB, dans la console aws il suffit de chercher dynamoDB puis de cliquer sur créer une table, on entre un nom, puis une clé de de partition et en option une clé pour le triage. La table contient les champs suivant :

Items returned (15)								
<div><div></div><div>Actions</div><div>Create item</div></div>								
<div>< 1 >  </div>								
<input type="checkbox"/>	reservationId (String)	clientEmail	clientName	clientPhone	reservationDate	restaurantEmail	restaurantName	restaurantPhone
<input type="checkbox"/>	067ce9c9-876f-4f47-b459-...	welty.alex67...	alexandre 1	+33664568074	11-09-2024 / midi	memepepe67310...	be one	+33668088567
<input type="checkbox"/>	c0be95cd-9383-4d2d-b2bf-...	welty.alex67...	alexandre 1	+33664568074	12-09-2024 / soir	angryjoker2525@...	buerehiesel	+33666666666

2. Cognito

a) Explication technique

Afin de créer une pool sur cognito on cherche cognito dans la console sur aws, on peut cliquer sur create pool, on sélectionne l'attribut qui va servir à s'identifier (ici l'email). dans l'option MFA on sélectionne no MFA. ensuite on peut ajouter des attributs supplémentaires comme le nom, le numéro de téléphone, etc... Puis on entre un nom pour la pool et on clique sur créer.

Pour cette application nous avons créé 2 pools, une pool pour les clients, une pool pour les restaurateurs. comme ça si un client est aussi un restaurateur et qu'il ne possède un seul mail, il peut s'inscrire sur les deux applications.

<input type="radio"/>	applicationFinalTest	us-west-2_pREFWuNuW	2 days ago	2 days ago
<input type="radio"/>	applicationFinalTestRestaurant	us-west-2_s6rhkaq7V	Yesterday	Yesterday

Utilisateur connecté pool client :

	User name	Email address	Email verified	Confirmation status	Status
<input type="radio"/>	28114360-d0a1-70f3-4474-1011b21af6e4	welty.alex67@gmail.com	Yes	Confirmed	Enabled
<input type="radio"/>	68a1c3c0-2001-70f1-236a-e1979372dffc	welty.alex.aw.67@gmail.com	Yes	Confirmed	Enabled

3. lambda

a) getRestaurants

Pour créer un fonction lambda on cherche lambda dans al console sur aws, on entre un nom pour la fonction on sélectionne le langage (ici python 3.12). Il faut aussi attribuer un rôle/permission à la fonction afin de lui donner le droit d'accéder à des ressources. puis on peut créer la fonction. Ensuite on cherche la fonction que nous avons créée puis nous ajoutons le code. voir le fichier lambda/client/getRestaurants.py

On teste la fonction et ajoute des paramètres d'event :

```

1 {
2   "queryStringParameters": {
3     "clientEmail": "alex@example.com"
4   }
5 }

```

b) InsertReservation

Idem que getRestaurants. voir le fichier lambda/client/insertReservation.py

c) getMyReservation

Idem que getRestaurants. Cette fonction lambda utilise un query paramètre. dans la réponse de la fonction il faut ajouter ceci afin d'éviter le blocage de la requête par cors :

```

'headers': {
  'Access-Control-Allow-Origin': '*',
  'Access-Control-Allow-Methods': 'GET, POST, OPTIONS',
  'Access-Control-Allow-Headers': 'Content-Type, Authorization, X-Requested-With'
},

```

voir le fichier lambda/client/getMyReservations.py

4. API Gateway

a) Explication technique

Pour créer une API on peut cliquer sur API Gateway, puis create api puis rest api ensuite on entre un nom. enfin dans api endpoint type on peut sélectionner edge-optimized afin de ne pas dépendre de la région utilisé lors de la création de l'api (ici on la crée sous us-west-2).

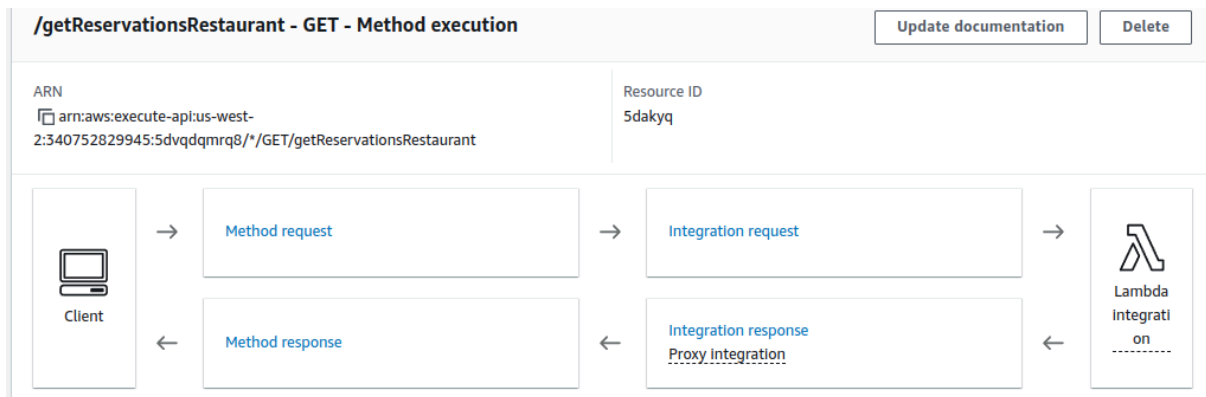
Une fois l'API crée on peut ajouter des ressources, sur chaque ressource on peut créer des méthodes GET, POST, etc... :

The screenshot shows the AWS API Gateway console. On the left is a sidebar with a tree view of resources. The main panel is titled 'Resource details' and shows the path '/getClientReservations' with a resource ID 'Synkqf'. Below this, there is a table of methods. The table has columns for Method type, Integration type, Authorization, and API key. Two methods are listed: GET (Lambda integration, None authorization, Not required API key) and OPTIONS (Mock integration, None authorization, Not required API key).

Method type	Integration type	Authorization	API key
GET	Lambda	None	Not required
OPTIONS	Mock	None	Not required

Certaines fonction lambda nécessite un paramètre, dans la resource correspondante dans integration request il faut que lambda proxy integration soit vraie. Dans une intégration proxy Lambda, API Gateway transmet directement la requête entrante du client à la fonction Lambda sous forme d'un objet événement.

Cet objet événement contient les informations suivantes : paramètre de la requête, corps de la requête.



Enfin pour chaque ressource il faut activer CORS. puis on peut déployer l'API.

5. CloudFront

a) Explication technique

Pour intégrer CloudFront, dans le service CloudFront on crée une nouvelle distribution, puis il faut entrer les valeurs suivantes :

- ajouter une origine qui correspond à notre bucket ou est stocké le site web.
- entrer un nom.
- cocher la case ne pas activer de sécurité supplémentaire (waf).
- et entrer le fichier root de l'application par exemple index.html dans default root object.
- puis on peut cliquer sur créer la distribution, un message va être affiché en nous demandant de copier les règles de permission à entrer dans le bucket S3.

<input type="checkbox"/>	ID	Description	Type	Domain name
<input type="checkbox"/>	E2VMZKP14VNHWW	-	Production	d9wfex879bofr.clou...
<input type="checkbox"/>	E10ZK3SUR3M5AP	-	Production	duzje0nd4e645.clou...

6. S3 Bucket

a) Explication technique

Nous avons détaillé la procédure de création d'un Bucket S3 en classe. ici on crée un bucket pour l'application client, on upload les fichiers dans properties ou active l'option static website hosting. Puis dans permission on block l'accès public au bucket et on colle les permissions qu'on a copiés lors de la création de cloudfront.

7. SNS

a) Explication technique

Afin d'utiliser SNS, dans la console aws on cherche le service sns, on clique sur créer un topic, on lui donne un nom etc. lorsqu'un restaurateur est connecté sur son profil on lui laisse la possibilité de s'inscrire au service sns. dans la fonction lambda on utilise l'arn :

ARN

`arn:aws:sns:us-west-2:340752829945:applicationRestaurant`

afin de créer l'inscription. lorsqu'il vérifie son mail on obtient :

Subscriptions (4)					Edit	Delete	Request confirmation	Conf
<input type="text" value="Search"/>								
	ID	Endpoint	Status	Protocol				
<input type="radio"/>	1e2bc3dc-0a5b-4b49-ad29-ec89c499533e	memepepe67310@gmail.com	Confirmed	EMAIL				
<input type="radio"/>	c43da85b-fafa-410d-816f-bc1952aed6fc	angryjoker2525@gmail.com	Confirmed	EMAIL				
<input type="radio"/>	d45ca794-c71d-4a43-a30a-3c0713666608	restaurentausaumon@gmail.com	Confirmed	EMAIL				

8. IAM

a) Explication technique

On a créé 2 rôles pour les fonctions lambda, un rôle ReadWriteDynamoDB, et un rôle fullAccessAdministrator.

9. Explication du code

On utilise **amazon-cognito-identity.min.js** et **aws-cognito-sdk.min.js** afin d'utiliser le service cognito.

Dans le fichier : **/applicationClient/js/cognito-auth.js** : on s'occupe de l'authentification d'un utilisateur.

par exemple :

- on récupère les infos d'une pool :

```
var poolData = {
  UserPoolId: _config.cognito.userPoolId,
  ClientId: _config.cognito.userPoolClientId,
};
```

- on crée une instance :

```
userPool = new AmazonCognitoIdentity.CognitoUserPool(poolData);
```


- qu'on peut ensuite utiliser pour se connecter, se déconnecter etc.

```
signOut = function signOut() {
    var currentUser = userPool.getCurrentUser();
    if (currentUser != null) {
        currentUser.signOut();
        window.location.href = "index.html";
    } else {
        console.log("No user is currently signed in.");
    }
};
```

- on peut aussi inscrire un utilisateur :

ici on passe les datas

```
function register(email, password, name, phoneNumber, onSuccess, onFailure) {
    console.log("Type of onSuccess:", typeof onSuccess);
    console.log("Type of onFailure:", typeof onFailure);

    var dataEmail = {
        Name: "email",
        Value: email,
    };
};
```

[...]

ici on initialise les objets qui vont être envoyé à cognito

```
var attributeEmail = new AmazonCognitoIdentity.CognitoUserAttribute(
    dataEmail
);
var attributeName = new AmazonCognitoIdentity.CognitoUserAttribute(
    dataName
);
var attributePhoneNumber = new AmazonCognitoIdentity.CognitoUserAttribute(
    dataPhoneNumber
);
```

[...]

Ici on commence l'envoi à cognito.

```
userPool.signUp(  
  toUsername(email),  
  password,  
  [attributeEmail, attributeName, attributePhoneNumber], ...  
  if (typeof onSuccess === "function") {  
    onSuccess(result);  
  } else {  
    console.error("onSuccess is not a function during signUpCallback");  
  }  
}  
);
```

le fichier : **/applicationClient/js/config.js** : qui contient les informations de la pool que l'on souhaite utiliser

```
applicationClient / js / config.js / cognito  
window._config = {  
  cognito: {  
    userPoolId: "us-west-2_pREFWuI  
    userPoolClientId: "3rg0u5dpj6a  
    region: "us-west-2",  
  },  
};
```

le fichier : **/applicationClient/script.js** : permet de créer les requêtes à envoyer.

ici on récupère les données du formulaire puis avec ajax on crée la requête et on effectue une action en fonction du retour

```
3  var API_ENDPOINT_POST_RESERVATION =
4      "https://5dvqdqmrq8.execute-api.us-west-2.amazonaws.com/prod/postReservation";
5
6  document.getElementById("saveReservation").onclick = function () {
7      var inputData = {
8          restaurantName: $("#restaurantName").val(),
9          restaurantEmail: $("#restaurantEmail").val(),
10         restaurantPhone: $("#restaurantPhone").val(),
11         reservationDate: $("#reservationDate").val(),
12         clientName: $("#userNameDisplay").text(),
13         clientEmail: $("#userEmailDisplay").text(),
14         clientPhone: $("#userPhoneNumberDisplay").text(),
15     };
16
17     $.ajax({
18         url: API_ENDPOINT_POST_RESERVATION,
19         type: "POST",
20         data: JSON.stringify(inputData),
21         contentType: "application/json; charset=utf-8",
22         success: function (response) {
23             alert("Réservation client enregistrée avec succès !");
24         },
25         error: function () {
26             alert("Erreur lors de l'enregistrement de la réservation.");
27         },
28     });
29 }
```

Lorsque l'on utilise des query parameters il faut utiliser la méthode fetch qui utilise des en-têtes standard. et ajouter :

```
fetch(API_ENDPOINT_SUBSCRIBE, {
  method: "GET",
  mode: "cors",
  headers: {
    "Content-Type": "application/json",
  },
})
```

quelques exemples des fonctions lambda :

- **lambda/client/insertReservation.py :**

Ici on configure sns et dynamoDB pour utiliser une table spécifique et un service sns spécifique.

```
dynamodb = boto3.resource('dynamodb')
sns_client = boto3.client('sns', region_name='us-west-2')
table = dynamodb.Table('ReservationData')
SNS_TOPIC_ARN = 'arn:aws:sns:us-west-2:340752829945:applicationRestaurant'
```

ici on envoie un email avec sns :

```
sns_message = (
    f"Bonjour {restaurant_name},\n\n"
    f"Vous avez une nouvelle réservation ! Voici les détails :\n"
    f"Date de la réservation : {reservation_date}\n"
    f"Nom du client : {client_name}\n"
    f"E-mail du client : {client_email}\n"
    f"Téléphone du client : {client_phone}\n\n"
    f"Veuillez vérifier votre système de réservation pour plus d'informations.\n\n"
    f"Merci,\nL'équipe de gestion des réservations"
)

print("Message SNS créé")

response = sns_client.publish(
    TopicArn=SNS_TOPIC_ARN,
    Message=sns_message,
    Subject=f"Nouvelle réservation pour {restaurant_name}"
)

print(f"Notification envoyée via SNS : {response}")
```

Puis on renvoie la réponse :

```
return {
    'statusCode': 200,
    'body': json.dumps({'message': 'Reservation saved and notification sent successfully!', 'reservationId': reservation_id})
}

except Exception as e:
    print(f"Erreur lors de l'abonnement à SNS : {str(e)}")
    return {
        'statusCode': 500,
        'body': json.dumps({'error': f"Error saving reservation data or sending notification: {str(e)}"})
    }
```

- **lambda/client/insertReservation.py :**

ici on essaye de récupérer le paramètre clientEmail passer en argument de la requête. on renvoie une erreur si clientEmail n'existe pas en configurant manuellement le headers

```
try:
    params = event.get('queryStringParameters', {})
    client_email = params.get("clientEmail")

    if not client_email:
        return {
            'statusCode': 400,
            'headers': {
                'Access-Control-Allow-Origin': '*',
                'Access-Control-Allow-Methods': 'GET, POST, OPTIONS',
                'Access-Control-Allow-Headers': 'Content-Type, Authorization, X-Requested-With'
            },
            'body': json.dumps({'error': 'clientEmail parameter is required'})
        }
```

F. identifiant de connexion

Il y a 2 applications, voici des comptes de connexions pour chaque application, des données sont déjà présentes sur ces comptes. Vous pouvez vous inscrire. vous trouverez dans les explications un lien avec route 53 et un lien avec cloudfront. au 6 octobre le lien avec cloudfront fonctionne, le lien avec route 53 ne fonctionne pas encore (problème de transfert nom de domaine)

Application client :

- lien (avec route 53): <https://welty.ovh/client>
- lien (avec cloudfront) : <https://duzje0nd4e645.cloudfront.net/>
- Tout fonctionne en lançant le fichier index.html dans le dossier application client (sauf la redirection page d'inscription -> page d'accueil.
- Compte 1 :
 - mail : welty.alex67@gmail.com
 - mdp : Azerty67*
- Compte 2 :
 - mail : welty.alex.aw.67@gmail.com
 - mdp : Azerty67*

Si un client souhaite ajouter une réservation il doit d'abord cliquer sur le bouton voir tous les restaurants dans la section afficher tous les restaurants. ensuite il faut copier-coller les informations du restaurant manuellement et cliquer sur enregistrer réservation. lorsqu'on copie les valeurs nom et mail d'un restaurant dans le formulaire il arrive qu'un caractère de tabulation soit ajouté à la fin de la chaîne de caractères, il faut le supprimer (jamais essayé avec)

Mes informations cognito

Email : welty.alex67@gmail.com
Nom : alexandre 1
Téléphone : +33664568074
Username : 28114360-d0a1-70f3-4474-1011b21af6e4

Se déconnecter

Vue Utilisateur

Ajouter une réservation

Nom restaurant

restaurant au saumon

Email restaurant

restaurentausaumon@gmail.com

Téléphone restaurant

+33388870183

Date réservation (JJ-MM-AAAA / 'midi' ou 'soir')

11-09-2024 / midi

Enregistrer réservation

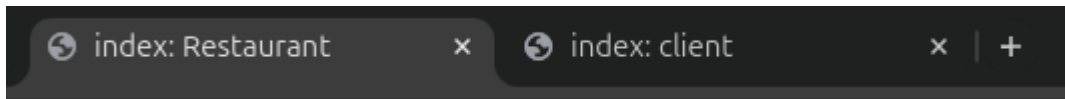
Application restaurant :

- lien (avec route 53): <https://welty.ovh/restaurant>
- lien (avec cloudfront) : <https://d9wfex879bofr.cloudfront.net>
- Tout fonctionne en lançant le fichier index.html dans le dossier application client (sauf la redirection page d'inscription -> page d'accueil.
- Compte 1 :
 - mail : restaurentausaumon@gmail.com
 - mdp : Azerty67*
- Compte 2 :
 - mail : memepepe67310@gmail.com
 - mdp : Azerty67*

Par défaut un restaurateur ne s'inscrit pas au service sns et ne reçoit donc pas d'email de notification lors d'une réservation, pour recevoir une notification email il faut se connecter puis s'inscrire au service SNS en cliquant sur le bouton s'inscrire :

S'inscrire a sns

S'inscrire



lien public github du projet : <https://github.com/weltya/s51>