Diseño de Aplicaciones en la Nube

Arquitectura Desacoplada vs Acoplada en AWS

Computación en la Nube

Dunia Suárez Rodríguez

25/26

ULPGC

Este repositorio contiene el código fuente, la configuración de infraestructura y la documentación correspondiente a la práctica.

Contexto y Objetivo

El objetivo principal de esta práctica es diseñar y desplegar una aplicación funcional y robusta utilizando servicios fundamentales de AWS. La aplicación implementa una **API RESTful** que permite realizar operaciones **CRUD**:

- **Create** = `POST`
- **Read** = `GET`
- **Update** = `PUT`
- **Delete** = `DELETE`

sobre una colección de "items".

La **API REST** es la API estándar para crear servicios web hoy en día. Se basa en el **protocolo HTTP**. Cada petición contiene toda la información necesaria para ser procesada(stateless). Las url `/items`, `/items/{id}` identifican el recurso sobre el que se quiere operar y el método HTTP define la acción.

Se hace uso de API Gateway para "ocultar servicios" para que el cliente(Postman, usuario) solo conozca la URL y la APIkey de API Gateway. De esta forma, el cliente no sabe (ni puede acceder) a los detalles del **servicio de cómputo** (sea **ECS Fargate** en la acoplada o **Lambda** en la desacoplada) que ejecuta la lógica. Esto permite:

- Ocultar la infrastructura interna
- Proteger los servicios de backend de ataques directos
- Centralizar la seguridad

API Gateway es un servicio serverless que además escala automáticamente para manejar millones de peticiones.

- **Serverless** → hay servidores pero AWS se encarga al 100% de gestionarlos
- **Escala automáticamente** → se expande para absorber cualquier pico de tráfico y se contrae cuando este desaparece

Archivos comunes en ambas arquitecturas

La carpetas /app/db y /app/models en ambas arquitecturas contienen exactamente lo mismo.

Archivo /db/db.py

Define una **interfaz** con los siguientes métodos:

- `initialize()`: Para configurar la conexión o crear tablas.
- `create_note()`: Para añadir una nueva nota.
- `get_note()`: Para obtener una nota por su ID.
- `get_all_notes()`: Para obtener todas las notas.
- `update_note()`: Para modificar una nota existente.
- `delete_note()`: Para borrar una nota.

Archivo /db/dynamodb_db.py

Contiene la implementación concreta de la interfaz de db.py

Los siguientes archivos también son iguales en ambas arquitecturas:

Archivo db_dynamodb.yml

- **Parameters**: define el nombre de la tabla; notes-board
- **Resources**
  - `AWS::DynamoDB::Table` → define el recurso de aws
  - `AttributeDefinitions` y `KeySchema` → definen que la clave principal de la tabla será un note_id
  - `BillingMode: PAY_PER_REQUEST` → solo pagas por las lecturas/escrituras realizadas
  - **Outputs:** `TableName` y `TableArn` → CloudFormation devuelve estos valores. Son útiles para que otros servicios los usen

Archivo ecr.yml

Su trabajo es crear un **repositorio en Amazon ECR (Elastic Container Registry)**. Es el servicio donde guardas y gestionas las **imágenes de Docker** de tu aplicación.

- **Parameters**: define el nombre del repositorio → notes-app
- **Resources**
  - `Resources: ECRRepository` → define el recurso principal: un repositorio en ECR.
  - `LifecyclePolicy`:
    - `"countNumber": 2` y `"action": "expire"` → hace que solo se conserven las 2 imágenes más recientes(5 en el caso de la arquitectura desacoplada) y borra automáticamente el resto
  - **Outputs:** `RepositoryUri`, `RepositoryName` y `RepositoryArn` → CloudFormation devuelve la URL, nombre y el ARN(Amazon Resource Name), del repositorio, respectivamente.

Versión de Arquitectura Acoplada (Monolítica)

Arquitectura y Servicios Utilizados

Para la implementación de un diseño acoplado, los servicios de AWS utilizados son:

Amazon ECS

El servicio de cómputo donde se ejecuta la aplicación (el backend). Esta instancia (o contenedor) contiene toda la lógica de negocio para procesar las operaciones CRUD.

ECSSecurityGroup

Crea un grupo de seguridad(un firewall virtual). Solo permite conexiones entrantes(`Ingress`) que cumplan:

- Protocolo tcp
- Puerto 8080(en el que escucha la aplicación dentro del contenedor
- `CidrIp(172.31.0.0/16)`: así no se permite el tráfico de 0.0.0.0/0(Internet), sino solo de un rango de IP interno de la VPC. En nuestro caso, solo el balanceador de carga de la VPC puede hablar con el contenedor

Network Load Balancer(NBL)

Recibe las peticiones y las distribuye en el tráfico, sin analizarlo, a los contenedores sanos. El balanceador no tiene IP pública por lo que no es accesible desde Internet. El API Gateway hablará con el a través del VPC Link.

TargetGroup

Define un grupo de destinatarios a los que el NBL enviará tráfico usando el protocolo TCP y el puerto 8080. Los destinatarios serán direcciones IP(las IPs privadas que Fargate asigna a cada contenedor, en nuestro caso solo uno). Se usa health check para verificar si los contenedores a los que se les envía tráfico funcionan correctamente, si no lo hacen se les dejará de enviar peticiones.

Listener

Escucha el tráfico que llega al NBL y lo reenvia por defecto a los TargetGroup

ECSCluster

Actúa como un **contenedor lógico** o un espacio de nombres para todos los recursos de ECS.

TaskDefinition

Es la especificación técnica de la tarea que se va a ejecutar. Define todos los parámetros de ejecución, incluyendo:

- La imagen de Docker exacta que se debe usar.
- Los recursos de cómputo asignados: `Cpu: 256` (0.25 vCPU) y `Memory: 512` (0.5 GB).
- Los roles de IAM que la tarea necesita para permisos.
- Las variables de entorno (`DB_DYNAMONAME`) y la configuración de logs.

ECSService

Se asegura que el número deseado de copias esté siempre en ejecución para que la tarea también esté en ejecución.

Es en este servicio donde se define la propiedad `LaunchType: FARGATE`.

AWS Fargate es un **motor de cómputo serverless** para contenedores. Elimina la necesidad de que gestionemos los servidores subyacentes. Nosotros solo proporcionamos la `TaskDefinition`, y Fargate se encarga de aprovisionar, ejecutar y gestionar la infraestructura necesaria para correr ese contenedor.

Aunque Fargate es serverless (sin gestión de servidores), nuestro uso de `DesiredCount: 1` lo convierte en un servicio **"siempre encendido" (always-on)**, lo que genera el costo fijo por hora 24/7 que veremos en el análisis de precios.

Para garantizar seguridad, debemos asegurarnos que nuestro contenedor no tenga IP pública, para ello debemos poner `AssignPublicIp: DISABLED`.

> Al cambiar la configuración de la IP hacemos que el contenedor no reciba tráfico de Internet, pero también impide enviar tráfico a Internet. Para solucionarlo existen dos opciones:
>
> - **Red NAT**
> - **VPC Endpoints(la escogida)** → crean una entrada privada a los servicios requeridos de AWS dentro de la propia VPC, así el tráfico solo viaja por la red interna de AWS, sin tocar el Internet público.

VPC Endpoints

Como explicamos en la sección superior implementaremos Endpoints para que nuestro tráfico viaje por una red interna. Es importante implementar un grupo de seguridad a los endpoints para que solo los contenedores ECS puedan hablar con ellos por HTTPS. Los endpoints establecidos son:

- `ECRApiEndpoint` → necesario para que Fargate pueda llamar a ECR
- `ECRDkrEndpoint` → necesario para descargar Docker imagen de contenedor
- `S3GatewayEndpoint` → necesario pues ECR almacena las capas de las imágenes de Docker en S3
- `CloudWatchLogsEndpoint` → permite al contenedor enviar los logs a CloudWatch
- `DynamoDBEndpoint` → permite la conexión a Dynamodb

API Gateway

Actúa como la puerta de entrada (front-door) para todas las peticiones HTTP. Gestiona los endpoints y dirige el tráfico hacia nuestro servicio de cómputo.

VPC Link

Es un puente privado que permite al servicio de API Gateway hablar con el NBL(que es interno de mi VPC privada)

- `RestAPI`: crea la API `notes-api` en sí
- `ItemsResource`: crea la ruta /items
- `ItemResource`: crea la rura /{id}

Métodos CRUD:

Estos recursos conectan un verbo HTTP a una ruta y le dice qué hacer:

- `PostItemsMethod`
- `GetItemsMethod`
- `GetItemMethod`
- `PutItemMethod`
- `DeleteItemMethod`

Con el HTTP_PROXY se pasa la petición HTTP casi intacta al backend, nuestro ECS Fargate.

Gestión de CORS(Cross-Origin Resource Sharing)

> Necesario para cumplir con la "Política del Mismo Origen" (Same-Origin Policy) de los navegadores. Por defecto, un navegador prohíbe que el JavaScript de una página web (ej. `mi-frontend.com`) haga peticiones a una API en un dominio diferente (ej. `api-de-aws.com`).

> CORS es el mecanismo por el cual nuestra API (el servidor) le indica al navegador que **confía** en ese otro dominio y le da permiso para enviar peticiones. Esto se gestiona a través de una petición de "pre-comprobación" (`OPTIONS`) que el navegador envía automáticamente antes de la petición real (como `POST` o `PUT`).

`OptionsItemsMethod` y `OptionsItemMethod` :

- `HttpMethod: OPTIONS` define el método para pedir permiso
- `Type: MOCK` → la API se encarga de este proceso y genera la respuesta sin llamar al backend. Es más barato que despertar al contenedor de Fargate o invocar una Lambda solo para devolver una respuesta estática.
- `IntegrationResponses` → donde permitimos que cualquier dominio llame a nuestra aplicación, se permitan los métodos especificados y las cabeceras especificadas

APIDeployment

Crea un **snapshot** (una instantánea inmutable) de toda la configuración de la API (rutas, métodos, integraciones) en un momento específico

APIStage

Publica ese *snapshot* en una URL pública (ej. `/prod`) para que los usuarios puedan acceder a él. Cualquier cambio en la API requiere un nuevo `APIDeployment` para que el `APIStage` lo publique.

Seguridad:

- `APIKey` → crea la contraseña
- `UsagePlan` → usado para agrupar las claves
- `UsagePlanKey` → vincula la APIKey con el UsagePlan

Outputs

Lo que CloudFormation te muestra en la consola cuando termina de crear todo el stack y lo necesario para empezar a usar la API: APIEndpoint y APIKey

---

Requirements.txt

Las dependencias de esta versión están definidas en el archivo requirements.txt y son:

- `flask==3.1.2` → es el **framework web**. Se usa para construir la API y manejar las peticiones web.
- `boto3==1.21.32` → es el **SDK oficial de Amazon Web Services (AWS)**. Se usa para interactuar con servicios de AWS. Para la gestión de archivos (subir o descargar imágenes a **Amazon S3**) o usar otros servicios como bases de datos (DynamoDB).
- `python-dotenv==1.0.0` → es una **herramienta de configuración**. Carga variables de entorno (como contraseñas de la base de datos, claves de API, etc.) desde un archivo `.env` al entorno de la aplicación. Esto es una buena práctica de seguridad para no "quemar" (hardcodear) secretos en el código.
- `pydantic==2.11.7` → es una biblioteca de **validación de datos**. Se usa para definir cómo deben ser los datos que tu API recibe o envía. Se asegura de que los datos tengan el tipo correcto (ej. que un email *parezca* un email, que un ID sea un número) antes de procesarlos.

Dockerfile

Este archivo contiene las instrucciones que Docker lee para construir la imagen de tu aplicación.

Main.py

Este archivo es el **servidor principal de Flask** que se ejecuta dentro del contenedor de Fargate, sirviendo como el controlador central para toda la API.

Función y Estructura

- **Framework:** Utiliza **Flask** para definir la aplicación web y el **enrutamiento** de las peticiones HTTP.
- **Diseño:** Implementa el patrón de Repositorio, **instanciando la base de datos una sola vez** (`db = DynamoDBDatabase()`) al inicio, ya que el servidor está siempre encendido (`always-on`).
- **CORS:** La función `add_cors_headers` garantiza que el tráfico *cross-origin* sea permitido, cumpliendo con las políticas de seguridad del navegador.
- **CRUD**: define todos los endpoints CRUD

Pricing

A continuación, se estima el costo mensual de la arquitectura acoplada para una simulación de **100,000 peticiones al mes** en la región us-east-1 (N. Virginia).
Esta arquitectura tiene un costo base fijo de aprox. **$69.94/mes** por los servicios que deben estar encendidos 24/7. El tráfico de 100,000 peticiones añade un costo variable de solo **$0.40**.

Costos Fijos

- **VPC Endpoints (AWS PrivateLink):**
  Este es el costo más alto. Se crean **3 Endpoints** de tipo "Interface" (ECR.api, ECR.dkr, CloudWatch.logs) para que la tarea de Fargate acceda a AWS de forma privada. La plantilla usa **2 Zonas de Disponibilidad (AZs)** para alta disponibilidad. AWS cobra por endpoint y por AZ, por lo tanto:

  > **Cálculo:** 3 Endpoints × 2 AZs × 730 horas/mes × $0.01/hora = **$43.80/mes**

- **Network Load Balancer (NLB)(contiene un costo variable):**
  El NLB es necesario para que API Gateway se conecte a la VPC mediante el VPC Link.
  - **Costo Fijo (por hora):** 1 NLB × 730 horas/mes × $0.0225/hora = **$16.43**
  - **Costo Variable (LCU):** El tráfico de 100k peticiones es tan bajo (0.04 conexiones/seg) que el costo variable es de solo **$0.04**.
    - peticiones 100k al mes ⇒ 3.333 al día ⇒ 139 por hora ⇒ 0.038 por segundo ~= 0.04 por segundo

  > **Cálculo:** $16.43 costo fijo + $0.04 costo variable = **$16.47/mes**

- **AWS Fargate (ECS):**
  Este es el costo de tener `DesiredCount: 1` (una copia) de tu contenedor funcionando 24/7.
  - **Configuración:** `0.25 vCPU` (de `Cpu: 256`) y `0.5 GB RAM` (de `Memory: 512`).
  - **Costo de vCPU:** 1 Tarea × 24 horas/día × 30.42 tareas/mes × 0.25 vCPU × $0.04048/hora = **$7.39**
  - **Costo de RAM:** 1 Tarea × 24 horas/día × 30.42 tareas/mes × 0.5 GB × $0.004445/hora = **$1.62**

  > **Cálculo:** $7.39 costo cpu + $1.62 costo ram = **$9.01 / mes**

- **Amazon ECR: $0.01 / mes**
  Costo de almacenar imagen de Docker (69 MB).

  > **Cálculo:** 0.069 GB × $0.10/GB = **$0.0069** (redondeado a $0.01)

Costos Variables (Por 100k Peticiones)

- **API Gateway (REST API): $0.35 / mes**
  Se paga $3.50 por cada millón de peticiones (después de la capa gratuita).

  > **Cálculo:** (100,000 / 1,000,000) × $3.50 = **$0.35**

- **Amazon DynamoDB(contiene un costo fijo):**
  - **Uso:** 100,000 peticiones (50k R, 50k W), 1 GB de almacenamiento y 1KB tamaño medio item.

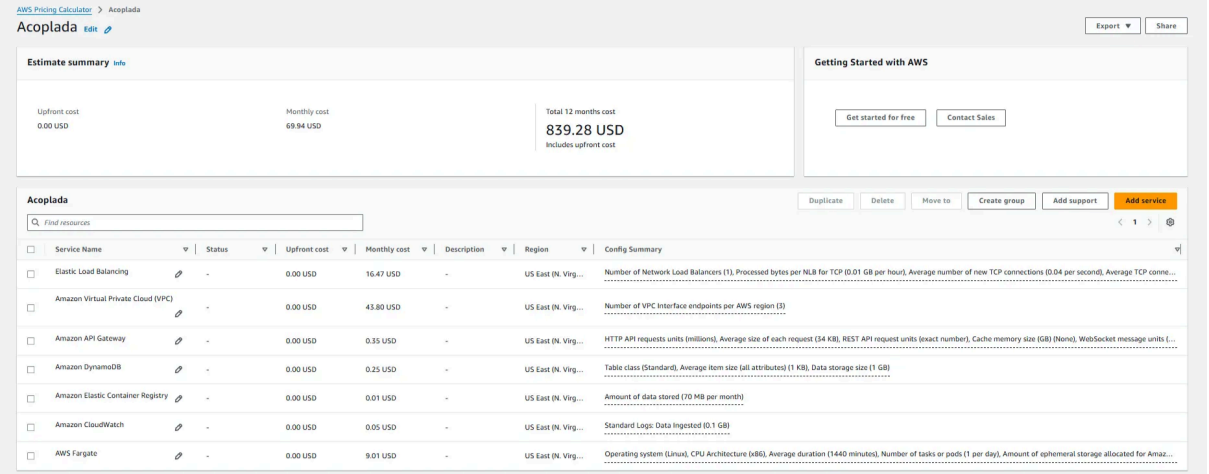
  > **Cálculo sin Free Tier**: $0.00 (Escritura) + $0.00 (Lectura) + $0.25 (Almacenamiento: COSTO FIJO) = **$0.25 USD / mes**.

  > **Cálculo con Free Tier**: **$0.00** (gratis 25 millones de peticiones y 25 GB de almacenamiento)

- **Amazon CloudWatch Logs:**
  - **Uso:** Estimamos 0.1 GB de ingesta de logs (1KB por cada petición).
  - La Capa Gratuita Permanente de CloudWatch (5 GB de ingesta/mes) cubre la totalidad de nuestro uso. Pero sin dicha capa:

  > **Cálculo sin Free Tier**: 0.10 GB x $0.50(Standard Logs Data Ingested cost) = **$0.05**

  > **Cálculo con Free Tier**: **$0.00** (5 GB de ingesta/mes)



Ejecución arquitectura acoplada

1. Lanzar la pila con la plantilla dynamodb_db.yml
2. Lanzar la pila con la plantilla ecr.yml
3. Construir la imagen de docker:

```
docker build --provenance=false -f Dockerfile -t notes-app:latest .
docker tag notes-app:latest 339712904114.dkr.ecr.us-east-1.amazonaws.com/notes-app:latest
docker push 339712904114.dkr.ecr.us-east-1.amazonaws.com/notes-app:latest
```

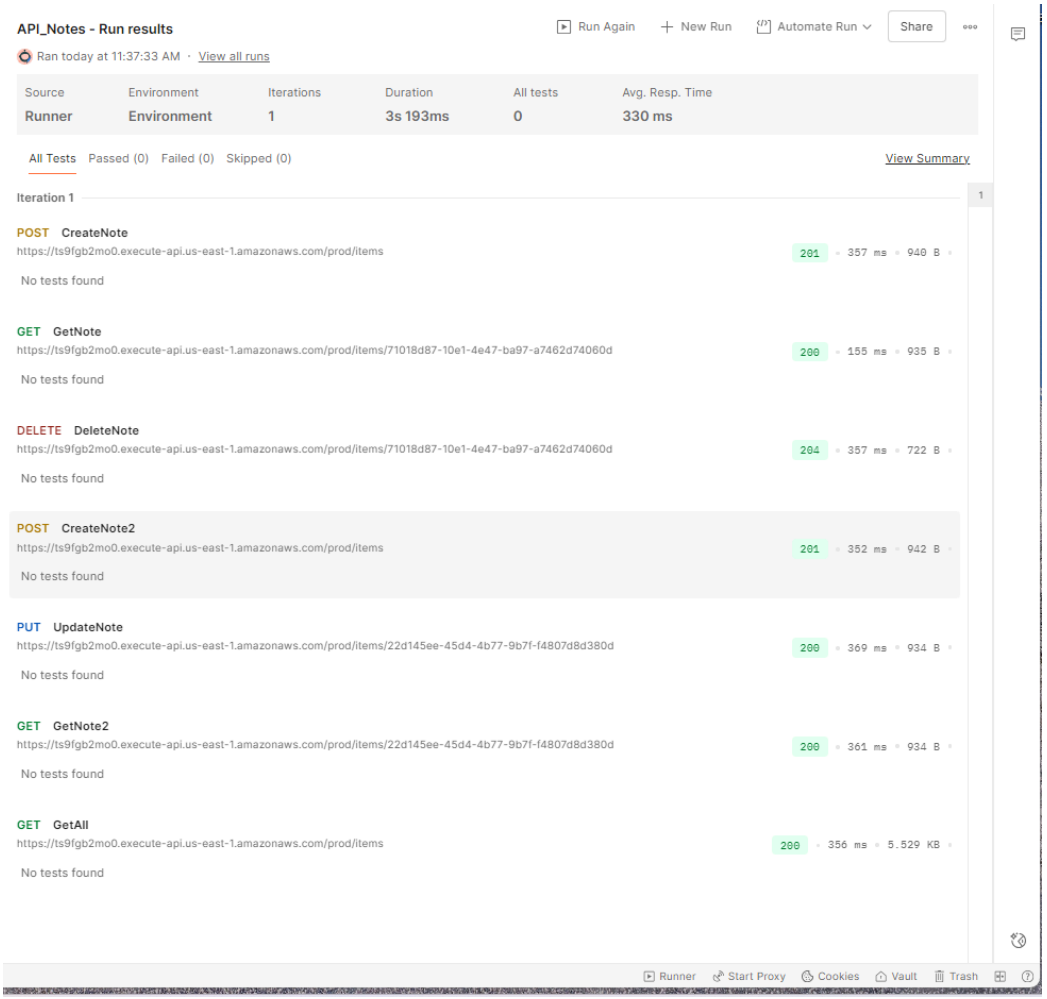
4. Lanzar la pila main.yml especificando 2 subnets y el vpc

Postman

Para la comprobación de la funcionalidad de los métodos CRUD, se adjuntan en el repositorio dos archivos obtenidos de la aplicación Postman:

- API_Notes.postman_collection.json
- API_Notes.postman_test_run.json → conjunto de tests pasados por la aplicación

Adjunto captura del run de los tests:



Versión de Arquitectura Descoplada

En esta versión desplegaremos la API REST usando una arquitectura serverless, nativa de la nube de AWS. Es una **alternativa moderna** a la arquitectura acoplada pues, aunque esta es robusta, introduce una sobrecarga operativa significativa:

- **Complejidad de Red:** Requiere una configuración avanzada de VPC, Subnets, VPC Endpoints y Balanceadores de Carga.
- **Costos Fijos:** El Balanceador de Carga (NLB) y los servicios de ECS (incluso en Fargate) incurren en costos "siempre activos" (always-on), independientemente del tráfico real.
- **Gestión de Cómputo:** Se debe gestionar un clúster, definiciones de tareas y servicios, incluso para una API simple.

Los cambios más significativos entre la acoplada y la desacoplada son:

- **Descomposición y Cómputo:** Reemplazo del clúster ECS/Fargate (aplicación monolítica) por **5 funciones Lambda independientes** (una para cada operación CRUD). AWS Lambda es un servicio de cómputo serverless que elimina la necesidad de aprovisionar o gestionar servidores subyacentes.

- **Integración:** Reemplazo de la combinación **API Gateway + VPC Link + NLB** por la integración directa **API Gateway + Lambdas** mediante `AWS_PROXY` (en lugar de `HTTP_PROXY` ,como usabamos en la acoplada).
    - **AWS_PROXY** → es un tipo de integración clave en Amazon API Gateway que simplifica radicalmente la conexión entre el gateway y los servicios de backend de AWS, especialmente **AWS Lambda**.

      ¿Cómo funciona? →

      1. `AWS_PROXY` toma la petición y la empaqueta en un formato JSON específico que Lambda entiende
      2. Cuando tu código Lambda termina de ejecutarse, devuelve un objeto JSON a API Gateway, `AWS_PROXY` lee ese objeto JSON y lo convierte de nuevo a una respuesta HTTP estándar.
- **Seguridad y Permisos:** Se elimina toda la infraestructura de red (**VPC, Security Groups y VPC Endpoints**), sustituyendo el firewall de red por un **control de acceso basado en identidad (IAM)**. Específicamente, se requiere un `AWS::Lambda::Permission` para dar permiso explícito a API Gateway para **invocar** cada función, asegurando un mínimo privilegio.
- **Contenedorización:** En el ECR se almacenan **5 imágenes** en vez de solo 1, lo que refleja el principio de microservicios de "una unidad de despliegue por función".

Arquitectura y Servicios Utilizados

La implementación se basa en el principio de **Single Responsibility** (Responsabilidad Única), descomponiendo la aplicación monolítica en funciones Lambda independientes.

Para esta implementación el rol que se le asigna a las lambdas debe tener los siguientes permisos para que la API funcione:

- Permisos de lectura/escritura sobre la tabla de Dynamodb
- Permisos para que las funciones Lambda puedan escribir sus logs

Para la implementación de un diseño desacoplado, los servicios de AWS utilizados son:

Funciones Lambda

Servicio que reemplaza a la infraestructura de ECS/Fargate, implementando un modelo de **pago por ejecución**.

- `CreateNoteLambda` (para **POST /items**)
- `GetNoteLambda` (para **GET /items/{id}**)
- `GetAllNotesLambda` (para **GET /items**)

- `UpdateNoteLambda` (para **PUT /items/{id}**)
- `DeleteNoteLambda` (para **DELETE /items/{id}**)

Características:

- PackageType: Image → las lambdas se despliegan usando imágenes de Docker almacenadas en ECR
- Cada función maneja una sola ruta, lo que permite escalarlas y actualizarlas de forma independiente.

API Gateway

Definición de Rutas (Resources):

- `ItemsResource`: Define la ruta principal `/items`.
- `ItemResource`: Define la ruta parametrizada `/{id}` (que resulta en `/items/{id}`).

Integración con Lambda (`Type: AWS_PROXY`):

Elimina toda la capa de red intermedia, simplificando la arquitectura y reduciendo la latencia. Cada método de la API (como `POST`, `GET`, etc.) está conectado directamente al ARN de una función Lambda específica:

- `PostItemsMethod` → **CreateNoteLambda**
- `GetItemsMethod` → **GetAllNotesLambda**
- `GetItemMethod` → **GetNoteLambda**
- `PutItemMethod` → **UpdateNoteLambda**
- `DeleteItemMethod` → **DeleteNoteLambda**

Soporte para CORS (`OPTIONS`):

`OptionsItemsMethod` y `OptionsItemMethod`

La plantilla define explícitamente los métodos `OPTIONS` para las rutas `/items` y `/items/{id}`.

Utiliza una **Integración** `MOCK` para que el propio API Gateway responda a estas peticiones (sin invocar una Lambda), devolviendo las cabeceras `Access-Control-Allow-*` necesarias. Esto es más rápido y económico.

Seguridad y despliegue
La gestión de la seguridad y el despliegue de la API es idéntica a la versión acoplada. Se utiliza una `APIKey` y un `UsagePlan` para proteger los endpoints y un `APIStage` (`prod`) para publicar la API en una URL pública.

Permisos de Lambda

Por defecto, los servicios de AWS operan con "denegación por defecto". Esto significa que, aunque hemos configurado el API Gateway para que intente llamar a las Lambdas,

las funciones Lambda rechazarán esas llamadas a menos que les demos permiso explícito.

Los permisos creados son:

- `CreateNoteLambdaPermission`
- `GetNoteLambdaPermission`
- `GetAllNoteLambdaPermission`
- `UpdateNoteLambdaPermission`
- `DeleteNoteLambdaPermission`

Esta sección de la plantilla crea esos permisos.

- `Type: AWS::Lambda::Permission`: A diferencia de un Rol de IAM (que da permisos desde la Lambda hacia afuera), esto es una **política basada en recursos**. Se adjunta directamente a la función Lambda y le dice qué servicios externos pueden invocarla.
- `Principal: apigateway.amazonaws.com`: Esto identifica quién puede invocar la función. En este caso, es el servicio de API Gateway.
- `Action: lambda:InvokeFunction`: Este es el permiso específico que se concede: la capacidad de ejecutar la función.
- `SourceArn: ...`: Esta es una medida de seguridad crucial. Limita la invocación no solo al servicio de API Gateway, sino a **nuestra API Gateway específica** (`!Ref RestAPI`) y al **recurso y método HTTP exactos** (ej. `/*/POST/items`). Esto evita que cualquier otra API Gateway en nuestra cuenta de AWS pueda invocar esta función.

Outputs

Lo que CloudFormation te muestra en la consola cuando termina de crear todo el stack y lo necesario para empezar a usar la API: APIEndpoint y APIKey.

Carpetas lambdas

Se ha creado una carpeta para cada función porque, en esta arquitectura, cada una de ellas es una micro-aplicación independiente que se empaqueta y despliega en su propia imagen de Docker. El contenido de cada una se basa en 3 archivos:

Requirements.txt

Las dependencias de esta versión están definidas en el archivo requirements.txt y son:

- `boto3==1.21.32` → es el **SDK oficial de Amazon Web Services (AWS)**. Se usa para interactuar con servicios de AWS. Para la gestión de archivos (subir o descargar imágenes a **Amazon S3**) o usar otros servicios como bases de datos (DynamoDB).
- `pydantic==2.11.7` → es una biblioteca de **validación de datos**. Se usa para definir cómo deben ser los datos que tu API recibe o envía. Se asegura de que los datos tengan el

tipo correcto (ej. que un email *parezca* un email, que un ID sea un número) antes de procesarlos.

Dockerfile

Este archivo contiene las instrucciones que Docker lee para construir la imagen de tu aplicación.

Main.py

Define la función handler que AWS Lambda buscará. Actua como controlador que une la petición de API Gateway con la lógica de base de datos

Cada main.py sigue el mismo patrón:

1. Reciben toda la información de la petición HTTP
2. Extrae la información:
   - `create_note/main.py` y `update_note/main.py` extraen el `event['body']`.
   - `get_note/main.py` y `delete_note/main.py` extraen el ID de `event['pathParameters']['id']`.
   - `get_all_notes/main.py` no necesita extraer nada.
3. Importa DynamoDBDatabase y el modelo Note.py y llama al **único método** que le corresponde (ej. `db.create_note(...)`, `db.get_note(...)`, etc.).
4. Devuelve una respuesta: Construye y devuelve un diccionario JSON con el formato que API Gateway espera, conteniendo el `statusCode` (200, 201, 404, etc.) y el `body` (generalmente, los datos de la nota en formato JSON).

Pricing

A continuación, se estima el costo mensual y anual de la arquitectura desacoplada para una simulación de **100,000 peticiones al mes** en la región us-east-1 (N. Virginia).
Esta arquitectura es un modelo 100% **"pago por uso"**. Si no recibe tráfico, el costo es de centavos (solo por el almacenamiento).

Costo Fijo

- **Amazon ECR: $0.03 / mes**
  - En lugar de 1 imagen monolítica (69 MB), ahora almacenamos **5 imágenes** de aproximadamente 300MB (una para cada función).
  - **Estimación:** 5 imágenes × 300 MB/imagen = 1500 MB.

  > **Cálculo:** 1,5 GB × $0.10/GB = **$0.15**

Costo variable

- **AWS Lambda (El Cómputo): $0.85 / mes**
  Este servicio reemplaza a Fargate y su costo tiene dos componentes, ambos basados en el uso:
  - **Costo por Peticiones:** AWS cobra ~$0.20 por cada millón de peticiones.
    - **Cálculo**: (100.000 / 1.000.000) × $0.20 = **$0.02**
  - **Costo por Duración (GB-Segundo):** Se paga por la memoria asignada (`512 MB`) multiplicada por el tiempo que tarda la función en ejecutarse.
    - Memoria: `512 MB` = `0.5 GB`
    - Duración: Basado en las pruebas de Postman (~900ms), estimamos **1 segundo** por petición.
    - Total GB-Segundos: 100.000 peticiones × 1 segundo × 0.5 GB = **50.000 GB-Segundos**
    - **Cálculo**: 50.000 GB-Segundos × $0.0000166667 = **$0.83**

  > **Cálculo sin Free Tier**: **$0.02** + **$0.83 = $0.85**

  > **Cálculo con Free Tier**: **$0.00** (gratis 1 millón de peticiones)

- **API Gateway (REST API): $0.35 / mes**
  Se paga $3.50 por cada millón de peticiones (después de la capa gratuita).

  > **Cálculo:** (100,000 / 1,000,000) × $3.50 = **$0.35**

- **Amazon DynamoDB(contiene un gasto fijo):**
  - **Uso:** 100,000 peticiones (50k R, 50k W), 1 GB de almacenamiento y 1KB tamaño medio item.

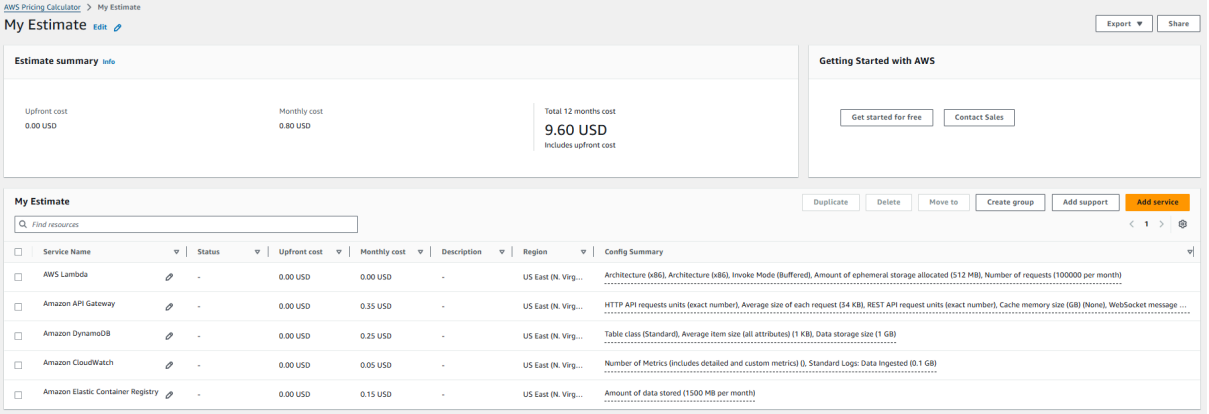
  > **Cálculo sin Free Tier**: $0.00 (Escritura) + $0.00 (Lectura) + $0.25 (Almacenamiento: COSTO FIJO) = **$0.25 USD / mes**.

  > **Cálculo con Free Tier**: **$0.00** (gratis 25 millones de peticiones y 25 GB de almacenamiento)

- **Amazon CloudWatch Logs:**
  - **Uso:** Estimamos 0.1 GB de ingesta de logs (1KB por cada petición).
  - La Capa Gratuita Permanente de CloudWatch (5 GB de ingesta/mes) cubre la totalidad de nuestro uso. Pero sin dicha capa:

  > **Cálculo sin Free Tier**: 0.10 GB x $0.50(Standard Logs Data Ingested cost) = **$0.05**

  > **Cálculo con Free Tier**: **$0.00** (5 GB de ingesta/mes)



Ejecución arquitectura desacoplada

1. Lanzar la pila con la plantilla dynamodb_db.yml
2. Lanzar la pila con la plantilla ecr.yml
3. Construir las imagenes de docker(una por cada función lambda) que se almacenan en el mismo repositorio:

```
----------create-note----------

docker build --provenance=false -f lambda_src/create_note/Dockerfile -t notes-app:create-note .
docker tag notes-app:create-note 339712904114.dkr.ecr.us-east-1.amazonaws.com/notes-app:create-note
docker push 339712904114.dkr.ecr.us-east-1.amazonaws.com/notes-app:create-note

----------get-note----------

docker build --provenance=false -f lambda_src/get_note/Dockerfile -t notes-app:get-note .
docker tag notes-app:get-note 339712904114.dkr.ecr.us-east-1.amazonaws.com/notes-app:get-note
docker push 339712904114.dkr.ecr.us-east-1.amazonaws.com/notes-app:get-note

----------get-all-notes----------

docker build --provenance=false -f lambda_src/get_all_notes/Dockerfile -t notes-app:get-all-notes .
docker tag notes-app:get-all-notes 339712904114.dkr.ecr.us-east-1.amazonaws.com/notes-app:get-all-notes
docker push 339712904114.dkr.ecr.us-east-1.amazonaws.com/notes-app:get-all
```

```
-notes

------------update-note-----------------

docker build --provenance=false -f lambda_src/update_note/Dockerfile -t notes
-app:update-note .
docker tag notes-app:update-note 339712904114.dkr.ecr.us-east-1.amazonaws.
com/notes-app:update-note
docker push 339712904114.dkr.ecr.us-east-1.amazonaws.com/notes-app:updat
e-note

------------delete-note-----------------

docker build --provenance=false -f lambda_src/delete_note/Dockerfile -t notes-
app:delete-note .
docker tag notes-app:delete-note 339712904114.dkr.ecr.us-east-1.amazonaws.c
om/notes-app:delete-note
docker push 339712904114.dkr.ecr.us-east-1.amazonaws.com/notes-app:delete
-note
```

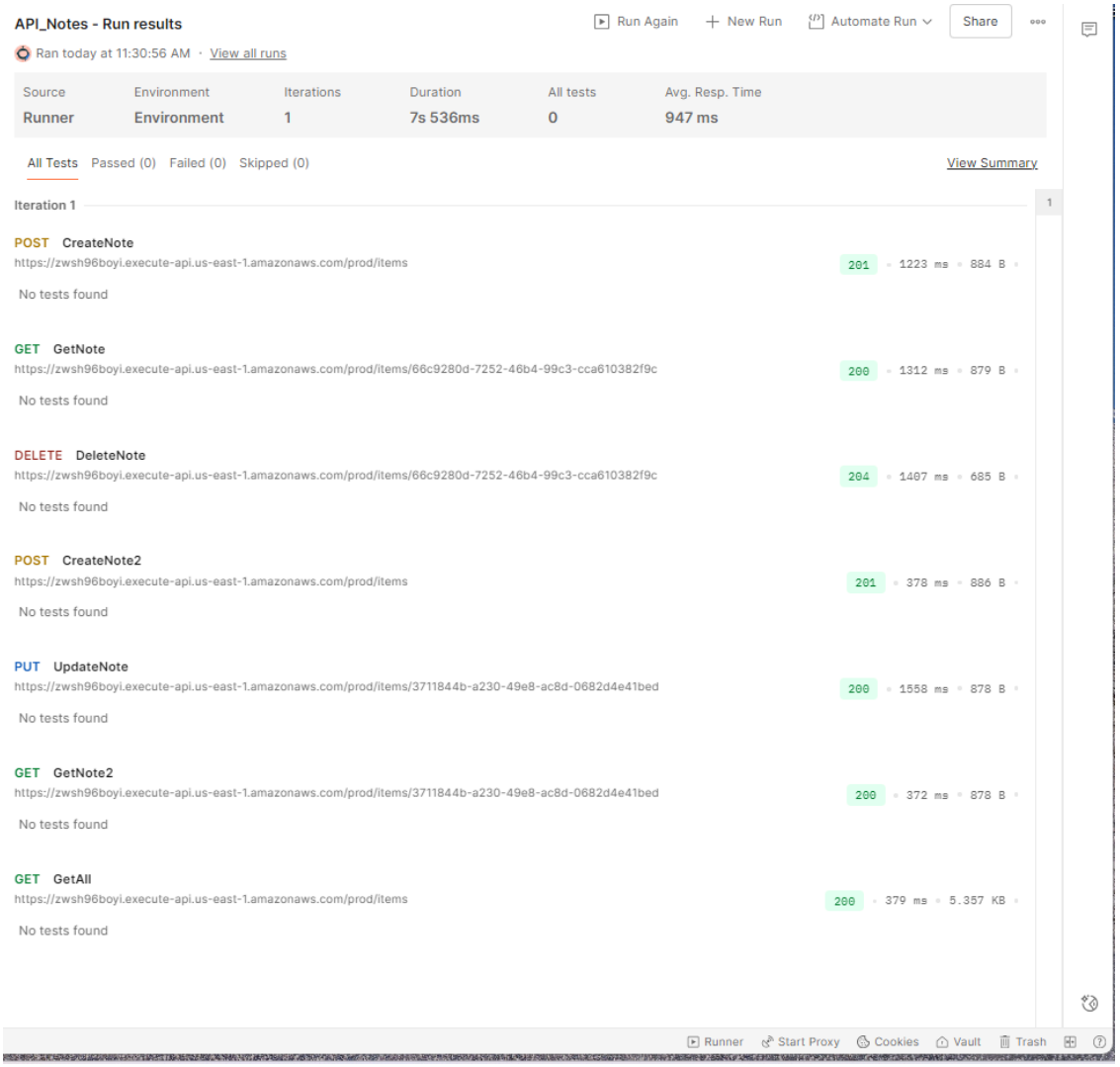
4. Lanzar la pila main.yml especificando 2 subnets y el vpc

Postman

Para la comprobación de la funcionalidad de los métodos CRUD, se adjuntan en el repositorio dos archivos obtenidos de la aplicación Postman:

- API_Notes.postman_collection.json
- API_Notes.postman_test_run.json → conjunto de tests pasados por la aplicación

Adjunto captura del run de los tests:



Se ha hecho uso de la IA gemini para entender la estructura del proyecto y muchos de los conceptos tratados y sobretodo para el arreglo de errores al lanzar las pilas.