COME GESTIRE UN APP GRAPHQL SU AWS USANDO APPSYNC AMPLIFY E TROPOSPHERE

AWS Amplify | AWS CloudFormation | AWS CodePipeline | DevOps | GraphQL | Serverless

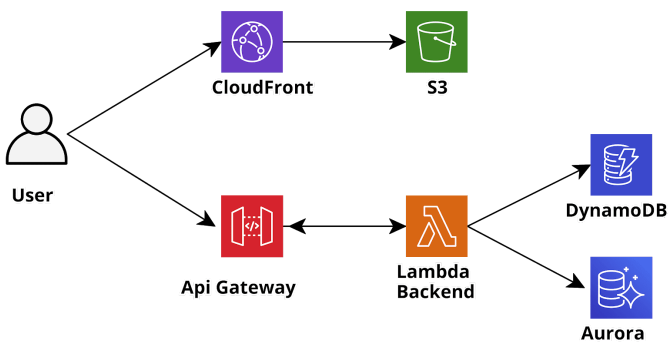
beSharp | 17 Aprile 2020

Al momento il **paradigma Serverless** è uno delle modalità più diffuse di implementazione di nuove **applicazioni sia Web based che Mobile**. Diversamente dall'architettura classica in cui il back-end è esposto da uno o più server Web costantemente connessi al database, **il back-end di un'applicazione Serverless** viene in genere distribuito utilizzando servizi **FaaS** (**Function as a Service**) e il database è di solito uno dei nuovi database NoSQL gestiti e scalabili, che possono essere interrogati direttamente utilizzando chiamate API HTTPS.

Su Amazon Web Services un'architettura come questa è in genere implementata utilizzando AWS Lambda Functions (FaaS) e DynamoDB (database NoSQL). Se è necessario un database SQL (ad es. per strutture dati complesse e/o fortemente gerarchiche) DynamoDB può essere sostituito da Aurora Serverless, un database Postgres (e MySQL) compatibile e scalabile automaticamente, completamente gestito da AWS che può anche essere interrogato tramite chiamate API HTTPS utilizzando il nuovo servizio AWS RDS DataApis.

In una tipica **applicazione API REST**, le funzioni Lambda sono in genere invocate tramite **AWS API Gateway** che riceve le richieste https dai client degli utenti, recupera i parametri e li inoltra alle funzioni Lambda che eseguono la logica di business e infine restituiscono una risposta all'API Gateway che può ulteriormente modificarla e decorarla, prima di rispondere al client. In questa configurazione, **l'autenticazione** degli utenti viene spesso gestita tramite AWS Cognito che è un servizio di registrazione, accesso e controllo degli accessi gestito da AWS. Se si utilizza Cognito, API Gateway può essere configurato per inoltrare le richieste alle **funzioni Lambda** solo se sono fornite di un token JWT di Cognito valido nell'header Authorization: la funzione Lambda dovrà poi occuparsi di gestire l'autorizzazione dell'utente (questo utente ha il permesso di eseguire questa azione su quella risorsa?) ricavando l'identità dell'utente e/o i suoi permessi dal token JWT di Cognito.

A volte potrebbe essere utile chiamare l'API Gateway utilizzando l'autenticazione AWS IAM (tramite Cognito Identity Pools) anziché l'autenticazione Cognito di base: in questo modo, le APIs della nostra applicazione saranno protette dallo stesso algoritmo di firma IAM v4 estremamente sicuro utilizzato da AWS per proteggere le proprie API: tutte le chiamate HTTPS ai servizi di AWS (ad es. S3, SQS, ecc.) sono firmate usando questo algoritmo. Inoltre, utilizzando l'associazione tra i gruppi di Cognito e i ruoli IAM, possiamo eseguire diversi tipi di autorizzazione di base direttamente a livello di API Gateway e non verrà nemmeno addebitato alcun costo per richieste non autorizzate!



Sketch of a typical AWS serverless application

Questi tipi di configurazione sono ormai estremamente comuni e sono stati ampiamente discussi in questo blog, tuttavia non ci siamo mai occupati dell'altro tipo di **integrazioni con il backend delle applicazioni moderne: GraphQL.**

GraphQL è un paradigma relativamente nuovo introdotto da Facebook nel 2015 per **gestire l'interazione tra frontend e backend.** La sua filosofia è significativamente diversa dal paradigma REST: mentre in una tipica applicazione RESTful utilizziamo i verbi HTTP (GET, POST, PUT, PATCH, DELETE) per definire il tipo di azione che stiamo eseguendo con una specifica chiamata API e usiamo path parameters, query parameters e il corpo della richiesta (in formato JSON) per specificare l'argomento dell'azione in corso, in GraphQL **tutte le chiamate HTTP sono in realtà chiamate POST** al backend e il tipo di azione da eseguire è completamente definito nel corpo della richiesta API . GraphQL definisce il suo linguaggio specifico per il body delle richieste, il quale definisce solo tre tipi di azioni: **query, mutations e subscriptions.** Le query vengono utilizzate per recuperare informazioni su un'entità, le mutations vengono utilizzate per modificare o creare un'entità esistente mentre le sottoscrizioni consentono al cliente di ricevere notifiche tramite Websocket per eventi specifici (ad esempio, le mutazioni di un'entità eseguite da altri utenti). Contrariamente al paradigma REST, che di solito è molto rigido, il linguaggio GraphQL è molto più dinamico e consente al frontend di richiedere direttamente le entità (e i singoli campi delle entità) di cui ha bisogno senza la necessità che il backend gestisca direttamente il filtro: il backend deve solo essere conforme allo standard GraphQL.

Solo per dare un rapido esempio di come funziona questo linguaggio supponiamo che il front-end abbia bisogno di un elenco di utenti esistenti, la richiesta sarà simile alla seguente:

```
query listUsers{
    listUsers(
        limit: 2
        )
        {
            items {
                user_id,
                email,
            }
        }
}
```

Questo tornerà una lista di utenti (è gestita anche la paginazione tramite un nextToken).

```
{
    "data": {
        "listUsers": {
            "items": [
                {
                    "user_id": "B2EF3212E32F423E",
                    "email": "test.graph@besharp.it"
                },
                {
                    "user_id": "A2EF4512E32F45RK",
                    "email": "test.full@besharp.it"
                }
            ],
            "nextToken": "tyJ3ZXJzaW9uIjoyLCJ0b2tl"
        }
    }
}
```

Ora vogliamo creare un nuovo utente:
Request:

```
mutation createUser
{
    createUser(
    input:
    {
        email: "test@besharp.it"
    }
        )
        {
            user_id,
            email
        }

}
```

Questa mutazione creerà il nuovo utente o restituirà **un'eccezione se già esiste.**
Uno dei vantaggi di GraphQL è che **il frontend può ottenere esattamente le informazioni di cui ha bisogno minimizzando il numero di chiamate API** e il peso delle risposte: ad esempio se vogliamo

solo il nome di un utente e non ci importa dell'email:
Request:

```
query getUser{
    getUser(user_id: "A2EF4512E32F45RK")
    {
        name
    }
}
```

Response:

```
{
    "data": {
        "getUser": {
            "name": "Test Full"
        }
    }
}
```

Se invece dovessimo avere anche bisogno del suo user id:

```
query getUser{
    getUser(user_id: "A2EF4512E32F45RK")
    {
        user_id
  email
        name
    }
}
```

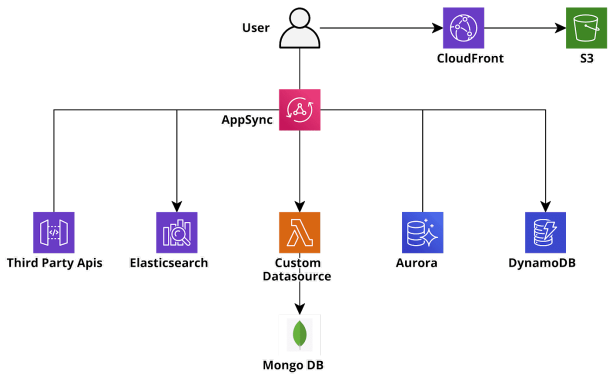
Response:

```
{
    "data": {
        "getUser": {
            "user_id": "A2EF4512E32F45RK"
            "email": "test.full@besharp.it",
            "name": "Test Full"
        }
    }
}
```

Inoltre, GraphQL è stato sviluppato per consentire al frontend di recuperare dati da fonti eterogenee. Ad esempio, pensiamo ad un'entità 'utente' che ha anche altre proprietà oltre a quelle finora considerate: l'utente potrebbe essere memorizzato in un database relazionale (ad esempio Aurora Serverless Postgres) ma potrebbe avere un campo GraphQL che descrive la sua località geografica (con relative proprietà) che è memorizzata in DynamoDB, inoltre a livello di GraphQL potremmo aggiungere un campo "Meteo locale" alla località che recupera i dati delle previsioni

meteorologiche locali da un'API di terze parti e che è popolato a runtime quando viene eseguita una query GetUser o ListUser.

Mentre le funzionalità del linguaggio GraphQL sono eccezionali, è sempre stato molto difficile configurare un back-end GraphQL utilizzando il paradigma serverless prima del rilascio di AWS AppSync. Questo nuovo servizio AWS ha infatti lo stesso ruolo di API Gateway per un'applicazione RESTful: riceve le chiamate API, controlla l'Autenticazione – Autorizzazione e le inoltra alla Lambda corretta o ad altri servizi di back-end. AppSync inoltre fa ancora di più: è possibile scrivere un backend personalizzato minimale utilizzando i modelli VTL (Apache Velocity) per collegare query e mutazioni direttamente alle origini dati (DynamoDB, Aurora Serverless, AWS ElasticSearch, API di terze parti). Per query e mutazioni più complicate è ovviamente necessario un backend 'reale' (ad es. per la crittografia custom per un campo del database). In questo caso, AppSync, proprio come Api Gateway, può richiamare una funzione Lambda per eseguire le operazioni necessarie e quindi utilizzare la risposta (o le risposte) per ricostruire il JSON finale che deve essere restituito al client.



Sketch of a typical AWS GraphQL (AppSync) serverless application

La bellezza di questo approccio è che possiamo **configurare AppSync per chiamare una funzione lambda** ogni volta che deve popolare un campo particolare di una data entità oppure popolare una intera entità. Quindi se eseguiamo una query di list, per ogni risultato restituito, una Lambda dedicata può essere eseguita per valutare un campo specifico, ad esempio il campo delle previsioni del tempo personalizzate nell'esempio sopra, mentre gli altri campi proverranno direttamente dai database. Questo è il motivo per cui **la risposta sarà sempre molto rapida** anche per le query che richiedono un numero elevato di operazioni personalizzate, dato che tali operazioni verranno sempre parallelizzate. Va comunque tenuto conto del fatto che, poiché le funzioni lambda sono fatturate per numero di esecuzione e per tempo di esecuzione totale, le query mal progettate possono comportare costi di AWS elevati ed imprevisti, perciò, in generale, è sempre utile impiegare il tempo necessario per pianificare accuratamente la corretta query e l'utilizzo previsto delle Lambda.

Sfortunatamente il prezzo da pagare per l'elevata configurabilità e flessibilità di AppSync è **la complessità e la "verbosità" della sua configurazione:** anche per un progetto molto semplice con una manciata di entità e una o due tabelle Dynamo è infatti necessario un file di schema lungo e dettagliato che descrive tutte le entità, le loro eventuali relazioni e come devono essere eseguite le mutazioni e le query. Inoltre, per ogni query e mutazione devono essere creati 2 file vtl (Velocity),

uno per l'analisi della richiesta e uno per l'analisi dell'output. Se si desidera utilizzare una funzione lambda o una catena di funzioni lambda (risolutori di pipeline), per popolare o recuperare un campo o un'entità, la complessità della configurazione aumenta in modo significativo così come lo sforzo necessario per mantenere e testare il progetto. Inoltre, creare e aggiornare l'infrastructure as Code (IaC) di un insieme di configurazioni così ampio, fortemente integrato e spesso in rapida evoluzione in un modo rapido, ripetibile e affidabile può essere un compito estremamente complesso.

Per semplificare la gestione del progetto è quindi necessario l'utilizzo di **un framework per automatizzare la creazione delle parti banali e ripetitive** delle configurazioni di AppSync. Il **framework Serverless** (molto diffuso) ha un plug-in AppSync dedicato, tuttavia il suo utilizzo non è molto più semplice o più veloce della creazione manuale del CloudFormation nativo per AppSync.

Ciò che abbiamo scoperto dopo svariati test è che una miscela di CloudFormation/Troposphere nativi con il framework AWS Amplify può risolvere il problema. Amplify è una piattaforma di sviluppo per **creare rapidamente applicazioni Web e Mobile utilizzando i servizi AWS Serverless** (Lambda, API Gateway, Appsync, Dynamo, SQS, ecc.) in modo trasparente. Sebbene sia spesso troppo limitato per un'applicazione complessa del mondo reale e, in generale, più che altro pensato per sviluppatori con conoscenze di AWS molto limitate che vogliono configurare rapidamente un'applicazione funzionante sfruttando i più moderni servizi serverless, la sua **integrazione con AppSync** è molto ben fatta e consente agli sviluppatori di configurare rapidamente un'integrazione GraphQL basata su DynamoDB (o Aurora) senza tutte le complessità e la verbosità descritte precedentemente. Inoltre, rende davvero semplice e facile sfruttare AWS Cognito per l'autenticazione e l'autorizzazione. E se Cognito da solo non dovesse soddisfare le vostre esigenze, è anche possibile usare la IAM authentication, tramite i Cognito Identity Pools.

Ecco come abbiamo creato **un back-end AppSync con Amplify e CloudFormation** in modo rapido, affidabile e facilmente gestibile: prima di tutto è necessario installare Amplify Cli come descritto nella documentazione di Amplify. Dopo aver fatto ciò, è possibile seguire la guida e continuare a la creazione di un progetto Amplify in una cartella dedicata tramite i comandi:

```
amplify configure
amplify init
amplify add api -> choose GraphQL
```

Tutti i comandi sono interattivi: scegli le opzioni più adatte alle tue esigenze! Dopo aver creato il progetto GraphQL api sarà necessario **modificare il file schema.graphql** nella cartella api/backend/: questo è l'unico file che è necessario modificare se l'applicazione non ha bisogno di integrazioni complicate! La sintassi di questo file è quella del linguaggio GraphQL con alcune annotazioni personalizzate di Amplify. Utilizzando queste annotazioni è possibile definire quali entità sone tabella Dynamo, creare Global Secondary Indexes, connessioni ad altre tabelle, campi personalizzati che usano le lambda come data source, impostare il livello di autorizzazione per ciascuna entità usando il token JWT di Cognito e molto altro ancora!
Quando si è soddisfatti della propria configurazione di GraphQL è possibile eseguire:

```
amplify push
```

E Amplify creerà ed eseguirà i template CloudFormation per creare o aggiornare AppSync!

Nonostante le varie opzioni disponibili in Amplify GraphQL nel caso di progetti complessi, probabilmente sarà necessario usare alcune funzionalità di AppSync non presenti di default tuttavia questo non è un problema: quando si esegue amplify push viene generata **una cartella di build** (.gitignored) e contiene la versione corrente di cloudFormation, tuttavia esiste anche una cartella api/backend // stacks che contiene un cloudFormation json (CustomResources.json) senza risorse: è possibile aggiungere qui ulteriori risorse custom. È anche possibile usare **Troposphere** per creare questo file, sfruttando così le funzionalità avanzate di Python nella creazione del modello CloudFormation. Nel caso fossero anche necessari anche file VTL personalizzati è possibile aggiungerli alla cartella api/backend//resolvers.

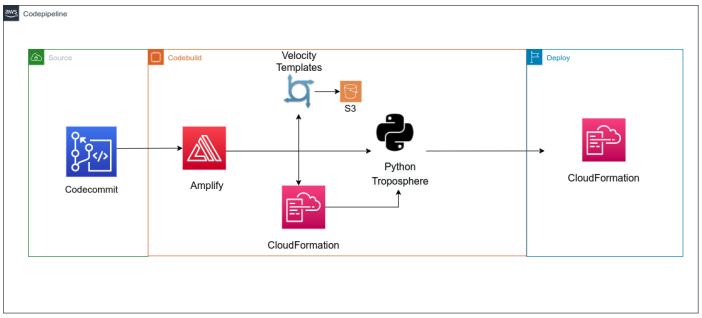
Dopo l'entusiasmo iniziale dei primi rilasci di solito iniziano tuttavia ad emergere i problemi legati all'uso di Amplify. Infatti è necessario mantenere vari componenti diversi: il file schema.graphql; la configurazione CloudFormation/Troposphere per le risorse personalizzate e i file VTL custom. Se si è scelto di usare Troposphere sarà anche necessario ricordare di eseguire lo script per creare i file yml di CloudFormation prima di eseguire amplify push.

Sono perciò necessari **ulteriori passaggi di automazione:** la soluzione più elegante è certamente la **creazione di una pipeline** (AWS CodePipeline) per eseguire il deploy della nostra applicazione GraphQL. La Source della pipeline può essere il repository AWS CodeCommit del progetto Amplify, oppure qualsiasi altro repository git (GitHub e Bitbucket sono supportati di default mentre altre configurazioni git lo sono tramite integrazioni custom).
Per far ciò iniziamo ripulendo tutto ciò che Amplify ha creato nel nostro account. Ricreeremo tutto da zero senza usare amplify push. Le risorse da eliminare sono gli stack Amplify di CloudFormation, i bucket S3 relativi all'amplificazione e l'applicazione Amplify nella console di AWS Amplify.

Dopo aver pulito l'account AWS si può rimuovere tutto dal progetto, esclusa la cartella backend mentre il file team-provider-info.json dovrebbe essere riempito con un json vuoto (basta scrivere {}). Ora è possibile creare un file buildspec.yml nella directory principale che verrà utilizzato da AWS CodeBuild:

```
version: '0.2'
phases:
    install:
        runtime-versions:
            python: 3.7
            nodejs: 12
        commands:
            - npm install -g @aws-amplify/cli
            - pip install -r requirements.txt
    build:
        commands:
            - echo "{ \"projectPath\": \"$(pwd)\", \"envName\": \"${ENVIRONMENT}\"}" >
amplify/.config/local-env-info.json
            - amplify api gql-compile
```

Questi comandi creeranno **i file di CloudFormation e i VTL** nella cartella build senza eseguirli. A questo punto sarà solo necessario aggiungere al codebuild config la variabile ENVIRONMENT che identifica l'ambiente per cui stiamo eseguendo la build e utilizzare il file principale di CloudFormation nella cartella build come output di codebuild (build/cloudformation-template.json). Nel caso si stia usando Troposphere è anche necessario aggiungere uno step aggiuntivo per compilare Python in CloudFormation. Bisogna infine caricare gli altri modelli di CloudFormation e i file VTL in un bucket S3 di propria scelta.
Il flusso è quello mostrato nell'immagine seguente:



L'ultimo passaggio della CodePipeline consiste nell'**eseguire il modello di CloudFormation** (output di codebuild) in uno step di deploy dedicato. CloudFormation si occuperà di scaricare gli altri template da s3 e di eseguirli come stack nested.

Ora hai il tuo progetto GraphQL personalizzato, configurabile e facilmente gestibile supportato da Appsync e Dynamo e deployato con Amplify e CloudFormation!

Se ti è piaciuta questa soluzione e desideri avere ulteriori dettagli, non esitare a scrivere nei commenti o a contattarci!

beSharp

Dal 2011 beSharp guida le aziende italiane sul Cloud. Dalla piccola impresa alla grande multinazionale, dal manifatturiero al terziario avanzato, aiutiamo le realtà più all'avanguardia a realizzare progetti innovativi in campo IT.

Get in touch

beSharp.it
proud2becloud@besharp.it