HOW TO TAME GRAPHQL: AWS APPSYNC, AMPLIFY AND CLOUDFORMATION TO THE RESCUE!

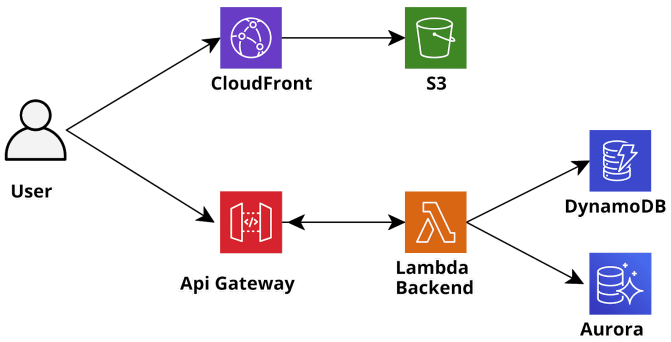
AWS Amplify | AWS CloudFormation | AWS CodePipeline | DevOps | GraphQL | Serverless

beSharp | 17 April 2020

The **Serverless paradigm** nowadays is a common way to implement novel **web-based and mobile applications**: differently from the classical architecture where the backend is exposed by one or more web servers persistently connected to the database, the **backend of a serverless application** is typically deployed using **FaaS** (**Function as a Service**) services and the database is one of the novel managed, scalable, NoSQL databases which can be queried directly using stateless https API calls. On AWS (Amazon Cloud Services) an architecture like this is typically implemented using AWS Lambda Functions (FaaS) and DynamoDB (NoSQL database). If a SQL database is needed (e.g. for complicated and/or strongly hierarchical data structures) DynamoDB can be replaced by Aurora Serverless, a Postgres (and MySQL) compatible and automatically scalable database completely managed by AWS that can also be queried through HTTPS APIs calls using the novel AWS RDS DataApis service.

In a typical **REST APIs application**, the Lambda functions are invoked by **AWS API Gateway** who receives the https requests from the users' clients, retrieves the parameters and forwards them to the Lambda functions who performs the business logic and finally return a response to the API Gateway who can further modify and decorate it, before returning it to the client. In this setup, the **Authentication** of the users is often managed through AWS Cognito which is a User Sign-Up, Sign-In, and Access Control service managed by AWS. If Cognito is used the API Gateway can be configured to forward the requests to the Lambda functions only if they comes with a valid Cognito JWT token in the Authorization header: the **Lambda function** should then manage the **User Authorization** (can this user perform this action on that resource?) fetching user identity and/or permissions from the Cognito JWT token. Sometimes it could be useful to call the API gateway using the AWS IAM authentication (through Cognito Identity Pools) instead of the basic Cognito authentication: in this way, the API of our application will be protected by the same extremely safe IAM v4 signature algorithm used by AWS to protect its own APIs: if fact all HTTPS calls to the AWS

services (e.g S3, SQS, etc) are signed using this algorithm. Furthermore using the association between Cognito Groups and IAM Roles we can perform several types of basic authorization directly at the API gateway level and you won't even be billed for unauthorized requests!



Sketch o a typical AWS serverless application

These types of setups are now extremely common and have been widely discussed in this blog, however we never talked about the elephant in the room: **what should I do if I need GraphQL?**

GraphQL is a relatively new paradigm introduced by Facebook 2015 to **manage the interaction between frontend and backend.** Its philosophy is significantly different from the REST paradigm: while in a typical RESTful application we use HTTP verbs (GET, POST, PUT, PATCH, DELETE) to define the type of action we are carrying out with a specific API call and we use path parameters, query parameters and the request body (in JSON format) to specify the argument of the ongoing action, in GraphQL **all the HTTP calls are actually POST calls** to the backend and the type of action carried out is completely defined in the API request body. GraphQL defines its specific language for the body which defines just three types of actions: **Query, Mutations and Subscriptions.** Queries are used to retrieve information about an entity, Mutations are used to modify or Create an existing database entity while subscriptions allow the Client to receive Websocket notifications for specific events (e.g. database mutations carried out by other users for a specific entity). In contrast with the REST paradigm which is usually very rigid the GraphQL language is much more dynamic and allows the frontend to directly ask for the entities (and entities fields) it needs without the need for the backend to manage the filtering directly: the backend just needs to be compliant with the GraphQL standard.

Just to give a quick example of how this language works let's assume the front end needs a list of existing users, the request will look like this:

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

```
        }
}
```

This will return a list of users (pagination is also managed through a nextToken).

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

Now we want to create a new user:

Request:

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

}
```

This mutation will create the **new user or return an exception** if it already exists.

One of the advantages of GraphQL is that **the frontend can get exactly the information it needs minimizing the number of API calls** and the weight of the responses: for example if we just want the name of a user and do not care about the email:

Request:

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

```
    {
        name
    }
}
```

Response:

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

Instead if we also want mail and user id:

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

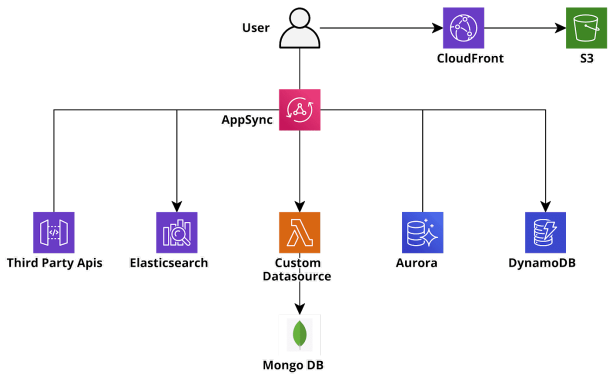
Response:

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

Furthermore, GraphQL has been developed to allow the frontend to fetch data from heterogeneous sources. For example, let's think about a user entity which also has other properties: the user entity itself could be stored in a relational database (e.g. Aurora Serverless Postgres) but it could have a GraphQL field describing the location which is stored in DynamoDB, furthermore at the GraphQL level we could add a 'Local Weather' field to the location which retrieves the local weather forecast data from a third party API and which is populated runtime when a GetUser or ListUser query is carried out.

While the features of the GraphQL language are great it was very difficult to setup a backend using the serverless paradigm before the release of AWS AppSync. This new AWS service in fact has the same role of API Gateway for a RESTful application: it receives the API calls, checks the Authentication – Authorization and forwards them to the correct Lambda or other backend

services. AppSync does even more since you can write a minimal custom backend using VTL (Apache Velocity) templates to connect queries and mutations directly to the data sources (DynamoDB, Aurora Serverless, AWS ElasticSearch, third party APIs). For more complicated queries e and mutations a real backend layer is obviously needed (e.g. a custom encryption for a database field). In this case, AppSync just like Api Gateway can invoke a Lambda function to carry out the heavy lifting and then use the response (or the response) to reconstruct the final JSON that should be returned to the client.



Sketch o a typical AWS GraphQL (AppSync) serverless application

The beauty of this approach is that we can configure **AppSync to call a lambda function** every time it has to populate a particular field of a given entity or to populate a whole entity, so if we execute a list query for each returned result a dedicate Lambda function will run to evaluate a specific field, such as the custom weather forecast field in the example above, while the other fields will come directly from the databases. This is why the **response will be always very fast** even for queries that require a large number of custom operations since these operations will always be parallelized. It should anyway be noted that, since lambda functions are billed per number of execution and per total execution time, ill designed queries can result in unexpected high AWS bills and, in general, it is always useful to take the time to thoroughly plan the correct queries and the lambda forecasted usage.

Unfortunately the price to pay for the AppSync high configurability and flexibility is the significant **complexity and verbosity of its configuration:** even for a very simple project with a handful of entities and one or two Dynamo tables, a long and verbose schema.graphql is needed, which describes all the entities, their eventual relations and how the mutations and queries should be executed. Furthermore 2 vtl (Velocity) files, one for the request parsing and one for the output parsing should be created for each query and mutation. If you want to use a lambda function, or a chain of lambda functions (pipeline resolvers), to populate or retrieve some field or the whole of a particular entity, the configuration complexity significantly increases as does the effort needed to maintain and test the project. Furthermore creating and updating the infrastructure as code of such a large, tightly coupled and often quickly changing body of configurations in a fast, repeatable and reliable way can be a daunting task.

In order to simplify the management of the project a framework to automate the deployment of the trivial and repetitive parts of the AppSync configurations is thus needed. While the very

widespread Serverless framework has a dedicated AppSync plugin, its usage is not much simpler and or faster than the native AppSync cloudformation even if the configurability is very high. What we found after some test is that a blend of native cloudformation/troposphere and AWS Amplify framework is what did the job for us. Amplify is a development platform to **quickly build web and mobile applications using the AWS Serverless services** (Lambda, API gateway appsync dynamo sqs, etc) in a transparent manner. While it is often too limited for a complex real world application and more oriented towards startup with very limited AWS knowledge who want to quickly set up a working application leveraging the most modern AWS services, its **App sync integration is very well made** and allows the developers to quickly set up a DynamoDb (or Aurora) based GraphQL integration without all the complexities and verbosity outlined above. Furthermore it makes it really simple and easy to leverage AWS Cognito for Authentication and Cognito groups or Custom fields for Authorization. And if Cognito alone does not fit your needs you can also use it with IAM authentication using the Cognito identity Pools!

Here is how we created an **AppSync backend with Amplify** and Appsync in a quick, reliable and easily maintainable way: first of all one needs to install the amplify cli as described in the Amplify documentation. After doing this one can follow the guide and go on to create an Amplify project in a dedicated folder with the commands:

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

All the commands are interactive: choose the options that better fits your needs! After creating the **GraphQL API Project** you'll need to modify the schema.graphql in the API/backend/<API_NAME> folder: this is the only file you'll need if your application does not need complicated integrations! The syntax is the one of the GraphQL language with some custom Amplify annotations. Using these annotations it is possible to define which entity is a Dynamo table, and in case of Dynamo backed entities creating global secondary index, connections to other tables, configure custom fields lambdas, set the authorization level for each entity using the cognito JWT token and much more! After you are satisfied with the GraphQL configuration you can run:

```
amplify push
```

And amplify will create and run the needed cloudformation for you!

Despite the several option available in Amplify GraphQL if your project is something more than a basic MVP you'll probably end up needing some functionality which is not there, but don't worry: when you run amplify push **a build folder** (.gitignored) is generated which contains the current version of cloudformation, however, there is also an API/backend/<API_NAME>/stacks folder which contains a Cloudformation json (CustomResources.json) with no resources: you can add the additional resources to it and it will run after all the other cloudformation files. If, like us, you don't like to write plain cloudformation you can take **a step more** and recreate it using Troposphere, leveraging the advanced functionalities of Python in the creation of the Cloudformation template. If

you also need custom vtl files, which is likely, you can add them to the api/backend/<API_NAME>/resolvers folder.

However after the initial excitement of the first deployments you'll probably start to find a few problems in this procedure. In fact, you'll need to maintain several different component: the schema.graphql; the cloudformation for the custom resources and the custom vtl files and if you used Troposphere you must remember to run the python generator script before executing amplify push.

Some further automation step is clearly needed: the most elegant thing to do is probably to **create a custom AWS CodePipeline to deploy our GraphQL application.** The source of the pipeline can be the AWS CodeCommit repository of your amplify project or any other git repository (GitHub and Bitbucket are supported by default, other git configurations through custom integrations).

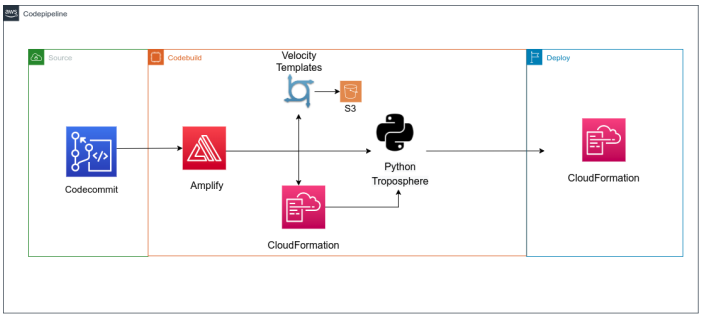
In order to do so let's start by cleaning up all the mess Amplify created in our account: since amplify is an instrument created for AWS rookies it does quite a bad job with cloudformation and s3 management with terrible naming conventions which makes it nearly impossible to understand what is going on under the hood. We'll recreate everything from scratch without using amplify push! The resources you'll need to delete are the amplify cloudformation stacks, the amplify related S3 buckets and the amplify application in the AWS Amplify console.

After cleaning up you can remove everything from your project except the backend folder and the team-provider-info.json should be filled with an empty json (just write {} in it). Now go on to create a buildspec.yml file in the root directory to be used by AWS CodeBuild, it should contain something like this:

```
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
```

These commands will create the **cloudformation and vtl files** in the build folder without running them. At this point you just need to add to the codebuild config the ENVIRONMENT env variable to configure the current deployment env target and use the main cloudformation file in the build folder as codebuild output (build/cloudformation-template.json). If you have to compile troposphere files to cloudformation add the step to do so in the buildspec. Finally, you need to update the other cloudformation templates and vtl file in a S3 bucket of your choice.

The flow is the one shown in the image below:



The last step of the codepipeline is to **run the cloudformation template** (codebuild output) in a dedicated deployment step. Cloudformation will take care of downloading the other templates from s3 and running them as nested stacks.

Now you have your **custom, configurable and easily maintainable GraphQL project backed by Appsync and dynamo and deployed by Amplify and cloudformation**, enjoy!

If you liked this solution and would like to have further details do not hesitate to write in the comment or to contact us!

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