AWS APPSYNC

SERVERLESS GRAPHQL APIS AT SCALE

INTRODUCTION 👋

Since Facebook released its previously internally used query language **GraphQL** in 2016, it has seen an outstanding increase in adoptions for all kinds of applications.

So it didn't come as a surprise when AWS built a managed solution called AppSync and released it in 2018. AppSync is becoming more and more popular and is already being used by large companies, including BMW and Sky.

CACHING ⚡

AppSync allows you to cache data on the server-side and by that enjoying several benefits:

- In-memory caching
- Improved performance
- Decreased latency

In the background, ElastiCache for Redis is used.

Caching can be used at different levels:

- **Full-request**: *$context.arguments* and *$context.identity* is used as key for caches.
- **Per-resolver**: Each resolver will be cached. Individual keys based on the context.

Amplify ⚠️

Amplify allows you to generate VTL resolvers from your data sources **automatically**.
With their CLI you can simply define an annotated GraphQL schema and all of your resolvers are created automatically.

Simply start with **amplify add api** and run through the wizard with selecting GraphQL as your target service. You'll be asked if you want to edit your schema in your favorite text editor.

Via **amplify push** the API will be deployed to AWS so you can immediately start using it.

GRAPHQL

REST APIs can become a burden for clients with an ever growing number of endpoints. Also, there's maybe never a perfectly fitting one to only fetch the data actually needed, so clients often end up receiving more data than necessary.

GraphQL was built to solve those problems by enabling the client to specify how to structure the data when it is returned by the server.

```
query {
  customer {
    id
    orders
    name
  }
}
```

GraphQL's top-level operations are:

- **Query** - read-only fetch
- **Mutation** - write, followed by a fetch
- **Subscription** - long-lived connection for receiving data

DATA SOURCES & RESOLVER 🔎

A resolver defines how AppSync processes your queries and mutations. It's the **glue** between your schema and the actual data sources.

One of the main advantages of AppSync is that you can **natively integrate with different AWS services** without writing any code.

A resolver is attached to a field, for example to the orders of a customer.

There are three different **kinds of resolvers**:

- **VTL Resolver** - connect your data directly to AWS resources like DynamoDB, RDS, or OpenSearch with the Apache **V**elocity **T**emplate **L**anguage. This does not require you to write any code, but only mappings between inputs and outputs.
- **Direct Lambda Resolvers** - if you want to use Lambda as your resolver you don't need to use VTL and attach your Lambda function directly. This is the most flexible approach that has basically no limits.
- **Pipeline Resolvers** - execute multiple resolvers before and after the execution.

COGNITO INTEGRATION

AppSync natively integrates with Cognito, which allows you to do **fine-grained access control at a per-field level.**

REAL-TIME DATA VIA SOCKETS ⚡

Subscriptions give you the ability to send real-time data to your clients without the need of any additional services.
It is natively integrated into AppSync.
For every mutation your client gets updated automatically.
For example in React simply use *useSubscription*.

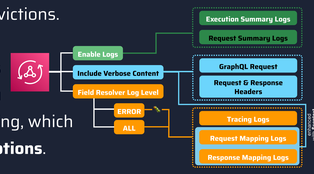
You can also use enhanced filtering via JSON in the response mapping template of the subscriptions resolvers.
Even multiple rules can be used together either in the same filter (**AND**) or while having multiple filters inside a group (**OR**).

```
type Subscription {
    addedPost: Post
    @aws_subscribe(mutations: ["addPost"])
    updatedPost: Post
    @aws_subscribe(mutations: ["updatePost"])
    deletedPost: Post
    @aws_subscribe(mutations: ["deletePost"])
}
```

MONITORING 👮

CloudWatch already collects a lot of metrics for you, including the number of HTTP 4xx and 5xx errors, request latency, the number of API requests and caching statistics like hits, misses or evictions.

For detailed insights about resolvers, you need to enable logging, which comes with **many options**.



⚠️ Verbose logging for all requests will generate very deep insights, but also increase the number of logs by an **order of magnitude**. Be careful, as CloudWatch ingestion is pricy.

For easy observability out-of-the-box, make use of external services like Dashbird.io which doesn't require you to create custom alarms or notifications.

DATA TYPES 🔶

Every GraphQL object in AppSync has a name and fields. Fields in turn can have sub-fields.

Each field's type must resolve to a scalar type, either from the default set of GraphQL or from AppSync's extensions prefixed with AWS, or to one of your own custom object types.

GraphQL Scalars	AWS Scalars
ID: Unique identifier, serialized as a String	**AWSDate**: Extended ISO-8601 date (YYYY-MM-DD)
String: A UTF-8 character sequence	**AWSTime**: Extended ISO-8601 time (hh:mm:ss.sss)
Int: An integer value	**AWSDateTime**: Combination of Date & Time (**T** separator)
Float: An floating point value	**AWSTimestamp**: seconds before/after 1970
Boolean: A Boolean value, either true or false	**AWSEmail**: email address as defined by RFC 822
	AWSJSON: valid JSON construct - automatically parsed
	AWSPhone: phone which can contain dashes/hyphens
	AWSURL: URL as defined by RFC 1738
	AWSIPAddress: a valid IPv4 or IPv6 address.

SCHEMA 📒

Schema files contain your GraphQL definitions incuding **query**, **mutation** & **subscription**.

You can create your own **object types** which include fields with scalar types or even other object types. Meaning, you can **deeply nest objects**.

By adding **mutations**, you define how items will be inserted or updated and what are the return types.

Subscriptions allow you to listen to certain events in your API, e.g. a creation of a new customer.

```
type Mutation {
  createCustomer: Customer!
}

type Subscription {
  customerCreated: Post
  @aws_subscribe(mutations: [
    "createCustomer"
  ])
}

type Query {
  getCustomer: [Customer!]
}

type Customer {
  id: ID!
  name: String!
  orders: [Order]
}

type Order {
  id: ID!
  productIds: [String]
}
```

AUTHORIZERS 🔐

You can protect your API with an Authorizer.

Your authorizer can be one of the type:

- **API_KEY** - authorized by API key.
- **AWS_LAMBDA** - authorized by a Lambda function.
- **AWS_IAM** - authorized via AWS IAM.
- **OPENID_CONNECT**- authorized via OpenID Connect.
- **AMAZON_COGNITO_USER_POOLS** - authorized by a Cognito identity pool.

This secures your **whole API.**
You can also set authorizers on field level. For example if you want to secure your **Customer** with IAM you can add the annotation @**aws_iam.** Similar to all other methods.

```
type Customer @aws_iam {
  id: ID!
  name: String
  orders: [Order!]
}
```

CODEGEN 👩‍🎨

Codegen is a CLI tool that allows you to create types, queries, and mutations **directly** from your GraphQL schema.

You simply need to define where your schema is located (**schema.graphql**) and define where to save your generated files & it's done. 🎉

```
overwrite: true
schema:
  - schema.graphql

generates:
  appsync.d.ts:
    plugins:
      - typescript
```

For all custom types like **Customer** in our example you can simply define **custom scalars** and map them to specific types.

Codegen has **tons of plugins** out there for creating all sorts of types and code **from your GraphQL schema.**

aws