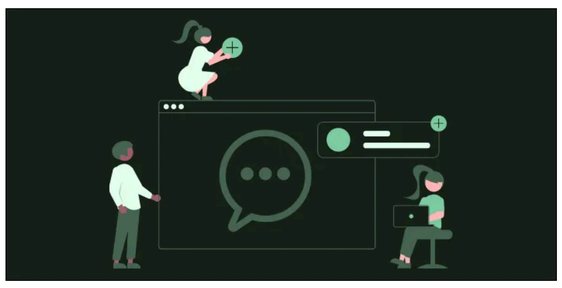
# Building a Massively Scalable Serverless Chat Application with AWS AppSync

Sarah Hamilton · Follow

Published in Serverless Transformation · 9 min read · May 11, 2021





## Introduction

When demand strikes, developers are tasked with building features ***quickly***.

My team knew this all too well when we were set to build a highly scalable chat app in ***just 4 weeks,*** when demand for our client's video conferencing service went through the roof at the beginning of the first Covid-19 lockdown.


Google Trends: Searches for 'Video Conferencing' in the UK. There was a significant increase in demand in March 2020.

When building the chat application discussed above we had to take into account the following requirements:

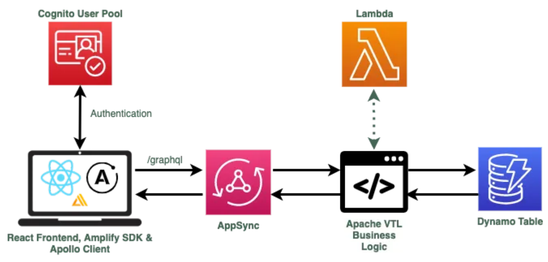
- Release to production quickly — their systems were failing to cope with the increased traffic levels
- Scale to 250,000 chat participants per video event
- Build the capability in their existing development team to maintain and further iterate the chat system

The chat application needed to have functionality where users could talk to each other in real-time. We used websockets rather than long-polling due to their cost efficiency and performance at scale (read more about the pros and cons of long-polling vs websockets).

The answer to all of the above was to use AWS AppSync, a fully-managed Serverless GraphQL interface to a number of different data sources — it offers speed of delivery, scale and ease of use. API Gateway websockets were also considered, but due to our requirements for mass broadcast of messages, it was not a viable option (there is no way to broadcast messages to all connected clients with one API call — you need an API call per connection). In addition, the GraphQL interface of AppSync allows for rapid frontend development.

> *My next article will be a tutorial on how I created chat emoji reactions in just 3 days — so watch out!*

## Architecture




Chat application architecture

This simple architecture enabled us to have a high speed of delivery for our chat application while also have a clear set of technologies that would need to be taught to the existing team.

The architecture is based around AppSync, which integrates with Cognito user pools for our authentication. DynamoDB is also leveraged as our data source (read about DynamoDB in the next section).

All of this hooks up to our React frontend which uses the Apollo Client to fire our requests to the GraphQL server from the frontend. Note, the Serverless Framework was used to manage the IaC (infrastructure as code), while amplify simply provided the SDK for frontend interaction with backend deployed services.

Let's look at the basics of GraphQL and DynamoDB before looking at how AppSync can provide an interface to our frontend. If you're familiar with these concepts you can skip ahead to the code examples.

## DynamoDB

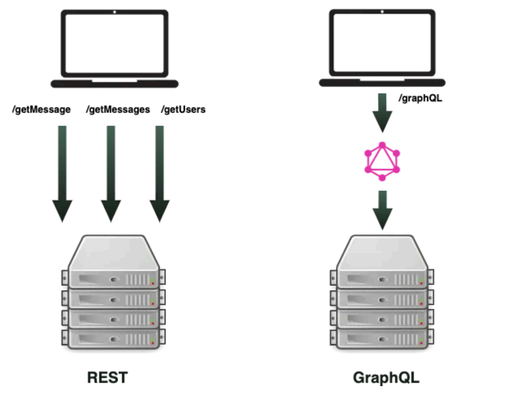
Being a NoSQL Serverless Database, DynamoDB provides a fully scalable solution to our needs without the need to manage servers. It was built for enormous, high velocity use cases and big companies such as AirBnb and Samsung use the service, so we're in good company! In addition DynamoDB streams came in use later on for further analytics integrations.

The fact that AppSync integrates with DynamoDB is ideal as it means we need to do minimal set-up and therefore increase our speed of delivery. See our Dynamo schema for the chat application below, which takes full advantage of single-table design.

| Entity | PK | SK | Attributes |
|---|---|---|---|
| **Message** | ROOM#<roomId> | MESSAGE#<messageId> | message<br>messageId<br>repliedAt<br>user: {firstName, lastName, email} |
| **Thread** | ROOM#<roomId> | THREAD#<messageId> | message<br>messageId<br>user: {firstName, lastName, email} |

DynamoDB Schema for basic Serverless chat application

## GraphQL Explained




With REST we need to call separate endpoints to get the information you require. With GraphQL we need only call the same GraphQL endpoint.

In a nutshell, GraphQL is an alternative way to connect your client applications to your backend. It was developed at Facebook, to improve bandwidth efficiency since mobile devices don't always have a good internet connection. Most of us are more familiar with REST, so let's compare it to GraphQL to get a better understanding.

- When using GraphQL we simply call ***one endpoint*** rather than having separate defined endpoints as in REST.
- *No more overfetching* with GraphQL! The client decides what data they want back from a request to the GraphQL endpoint. With REST we get ***all*** the information back for that specific request. Thus, GraphQL is a great solution to reducing valuable bandwidth.
- *No more underfetching* with GraphQL! Consider a simple Chat service. With REST to get the delivery information of a customer we may need to call the 'getUserDetails' endpoint to return the 'userId' to then call the 'getUserMessages' endpoint to return the user's messages. With GraphQL we can return this information all in one go!

The paradigm shift from REST to GraphQL may seem intimidating, but it's quick to learn and very enjoyable!

We define a schema in the backend which outlines exactly which actions are available for the client to perform against our data. The actions available in GraphQL are queries, mutations and subscriptions. Let's dig into these a little.

#### Queries

Queries in GraphQL are analogous to GET requests in REST. Queries are a way of fetching data (in our case we are fetching from DynamoDB as our database so we'll use this as our example going forward).

See how we define a query above to get messages. We pass in a roomId as an argument which we can use in our query to only get messages for that specific chat room. We then specify that the return type is an array of messages, where 'Message' is a type.

#### Mutations

Mutations in GraphQL are similar to POST requests in REST. Mutations do what they say, they are way of mutating the data. This could be adding an item to a DynamoDB table or changing an attribute. See above how we set up a sendMessage mutation — we pass in the roomId and message as arguments so that we can add it with that information as an item to our DynamoDB table. The return type is a Message.

#### Subscriptions

AppSync subscriptions, which allow for real-time updates, are where the fun begins. For our chat application, we want to 'subscribe' to the mutation event 'sendMessage' so that the frontend application updates when a message is sent with the new message to all users. It does this by setting up a websocket which is 'listening' for the sendMessage mutation. When the event is received in the frontend we can update the frontend with the information that we have received (see more detail about this in the 'apollo client' section).

You can see here that we pass in the roomId which means that only users in that particular chat room will be listening for messages sent.

### AppSync Explained

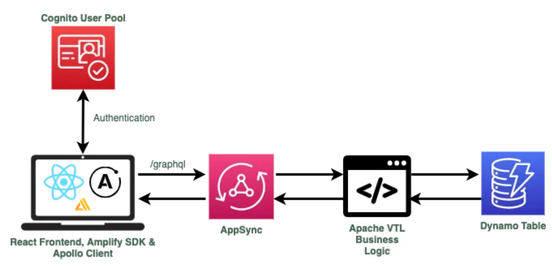
AppSync is a fully managed AWS Serverless implementation of GraphQL that scales to millions of users and offers multi-AZ — multi-AZ (availability zone) increases the availability of the service because each AZ is separate from each other and isolated from disasters. AppSync is taking care of our scalability goals out of the box. The fact that AppSync takes care of the 'heavy-lifting' makes this easier to train up developers quickly on the project.

In addition, AppSync has direct integration with DynamoDB, Lambda, RDS, ElasticSearch and HTTP. Since our data is stored in DynamoDB, another Serverless AWS offering for NoSQL data storage, this suits the project well.

AppSync also integrates with Cognito user pools which is ideal for our application as we're using Cognito as our main authentication provider. You simply need to add which group is authenticated to perform the operation in the GraphQL schema. In this case we allow the 'User' Cognito group to be authorised to send messages. It really is as simple as that!

*For Serverless Framework users:* We use the Serverless framework as our IaC to manage backend resources. It can take a few minutes to deploy changes in the backend to the cloud which can lead to a slow development cycle. However, AppSync has a wonderful console to develop in. You need to deploy the Serverless stack once, make changes in the AWS AppSync console until you have the desired results, then copy the code back into your development code.

### Resolvers

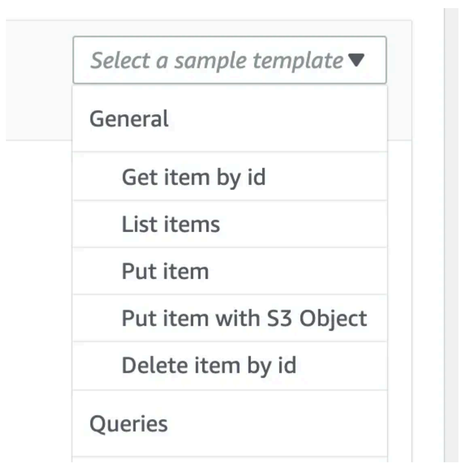



End to end flow of the chat application architecture. We use Apache VTL for our business logic.

In order to communicate with our data source we need to use resolvers — this is the connection between GraphQL and our data source.

The resolvers are written in Apache Velocity Template Language (VTL) which takes the request as an input and outputs a JSON document. VTL can be a big learning curve when getting used to AppSync, since it isn't the easiest of languages to pick up. However, this should not stop you! AWS provides a variety of templates which cover a lot of use cases such as getting and putting items into your data source.

> *We hear that resolvers will soon have JavaScript support which we are very excited about!*




When we need to perform more complicated business logic, we prefer to create a Lambda Function to handle this — we are much more familiar with JavaScript. Speed of delivery, extensibility and developer experience comes at the cost of an extra request and latency.

See below our VTL logic for querying messages for a particular chat room that we describes in the 'Queries' section.

VTL logic for querying a DynamoDB table for messages for a particular event chat room.

### Pipeline Resolvers

Sometimes we may need to perform multiple operations to resolve the GraphQL field — in this case we use a great feature of AppSync called pipeline resolvers.

Pipeline resolvers consist of a 'before' mapping template, followed by a number of functions, finishing with an 'after' mapping template.

In our case we use 2 functions in a pipeline resolver to mutate the DynamoDB table twice before resolving our request.

As an example we use a pipeline resolver for starting a thread on a question. To make use of single table design we have 'MESSAGE' rows and 'THREAD' rows. When a person replies in a thread, we need to add a THREAD row and mutate the MESSAGE row to add a 'repliedAt' timestamp. This requires 2 separate requests. See below how we do this with a pipeline resolver.

Once we add a THREAD row to the table we need to mutate the MESSAGE row.

### Frontend — Amplify & Apollo Client

To authenticate our users we use the AWS Amplify SDK with Cognito. Amplify provides pre-built UI components to cater for sign-in and sign-out. We configure the authentication by using 'Auth' from aws-amplify and using our user pool information (from the Cognito user pool deployed by the Serverless Framework). We take advantage of this as it greatly reduces the development time for authentication.

To make the requests from the frontend we use the Apollo Client- it is our favourite tool for interacting with our backend as it is compatible with TypeScript, providing a great developer experience, and it manages state in the frontend using a caching mechanism.

### Cost

AppSync's stated costs are slightly more expensive than Query and Data Modification Operations in API Gateway. However, AppSync generally works out to be less expensive, as you'll be making fewer requests when using GraphQL (no more underfetching)!

- $4.00 per million Query and Data Modification Operations ($3.50 per million in API Gateway)
- $2.00 per million Real-time Updates

### Tips and Tricks

- Turn to the AWS AppSync console for local development. If you are using Serverless it removes the needs to deploy to see your changes! The console that AWS provides is one of the reasons I have enjoyed working with AppSync so much.
- Use Lambda Functions for complex logic if you aren't familiar with Apache VTL
- Unit test VTL mapping templates. We use functions from amplify-appsync-simulator to help us with this
- Use the Apollo dev tools plugin. It is a huge help when debugging in the frontend

### Conclusion

AppSync allows us to focus on the code and not the underlying infrastructure which means we are able to build a product **quickly.** With AppSync being one of AWS's Serverless offerings, the total cost of ownership

(TCO) is reduced as there is no need to invest in the maintenance of the underlying infrastructure.

Did we achieve our aim to build a scalable chat app quickly and train up existing developers on the project using AWS AppSync? **Absolutely!**

Aws Appsync | Serverless | Cognito | Chat App Development | Dynamodb

## Written by Sarah Hamilton

Follow

181 Followers · Writer for Serverless Transformation

Application Engineer at The LEGO Group and Serverless Enthusiast

### More from Sarah Hamilton and Serverless Transformation

Sarah Hamilton in Engineers @ The LEGO Group

**Resolving Bottlenecks of Lambda Triggered By Kinesis—Part 1: Dat...**

When it comes to scaling Lambda Functions triggered by Kinesis Data Streams it can be...

10 min read · May 16, 2022

81 2

Xavier Lefèvre in Serverless Transformation

**Asynchronous client interaction in AWS Serverless: Polling,...**

Event-driven Serverless often requires async update to the client. There are several ways ...

6 min read · Apr 18, 2020

730 7

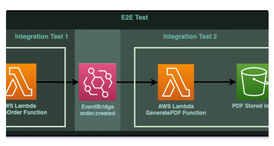
Xavier Lefèvre in Serverless Transformation

**What a typical 100% Serverless Architecture looks like in AWS!**

If you are new to serverless and looking for a high level web architecture guide, you've...

11 min read · May 19, 2020

2.8K 19




Sarah Hamilton in Serverless Transformation

**Bridge Integrity—Integration Testing Strategy for EventBridge...**

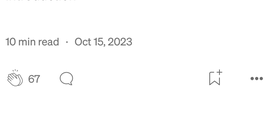
Testing Event-Driven architectures is a challenge. We share our integration testing...

7 min read · Feb 23, 2021

489 1

See all from Sarah Hamilton | See all from Serverless Transformation

## Recommended from Medium



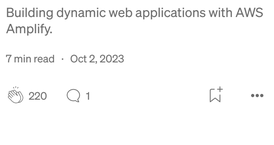
Sonu Kumar (Amit)

**Designing Real-time Chat Applications with Firebase...**

In today's interconnected world, real-time communication plays a crucial role in...

3 min read · Oct 11, 2023

4



Pravin Mahale in Globant

**Real-time NodeJs chat application using AWS WebSocket and...**

Build a chat app using cloud functions

8 min read · Jan 3, 2024

107

### Lists



General Coding Knowledge
20 stories · 993 saves



Leadership
48 stories · 262 saves

Natural Language Processing
1265 stories · 752 saves



Sidath Munasinghe

**Real-Time Magic: Building WebSocket APIs with AWS API...**

Introduction

10 min read · Oct 15, 2023

67




Trishant Kumar in Simform Engineering

**Vue.js and AWS Amplify: Unleashing the Full Potential**

Building dynamic web applications with AWS Amplify.

7 min read · Oct 2, 2023

220 1




Muzaffer Yılmaz in Trendyol Tech

**How We Used Server-Sent Events (SSE) to Deliver Real-Time...**

We will show how we used Server-Sent Events (SSE) and Redis for our notification...

6 min read · May 17, 2023

712 7



Durgaprasad Budhwani

**How to Set Up AWS S3 to EventBridge Rules for Efficient...**

Imagine AWS S3 as a giant, secure warehouse where you can store all sorts of digital object...

2 min read · Feb 26, 2024

148

See more recommendations