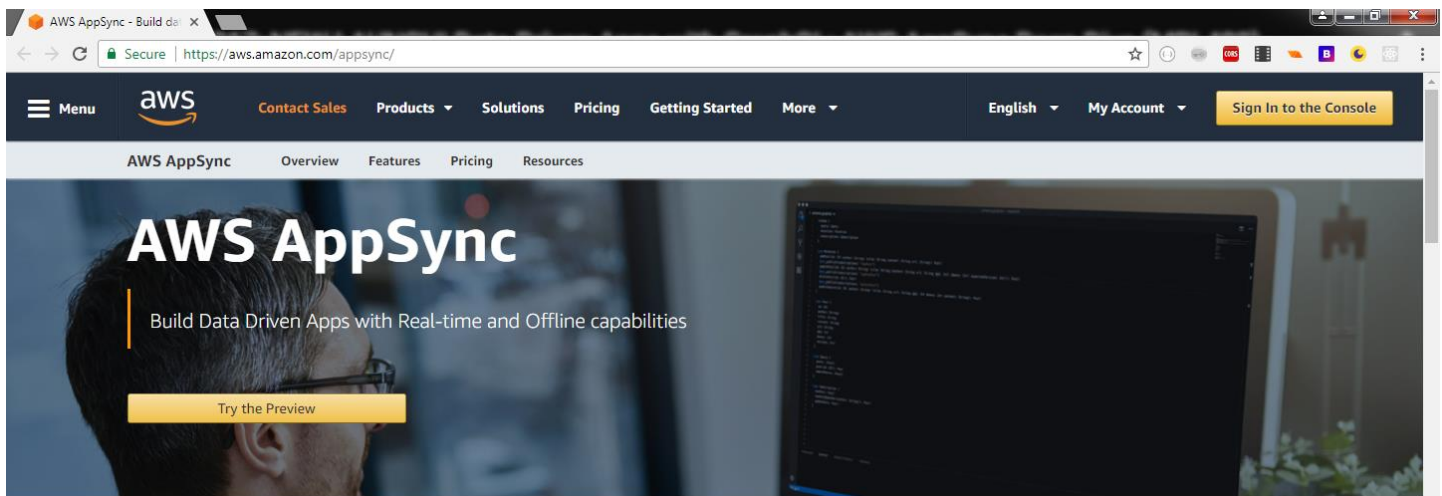




AWS AppSync is available in Preview and users can sign up to get access.



AWS AppSync automatically updates the data in web and mobile applications in real time, and updates data for offline users as soon as they reconnect. AppSync makes it easy to build collaborative mobile and web applications that deliver responsive, collaborative user experiences.

You can use AWS AppSync to build native mobile and web apps with iOS, Android, JavaScript and React Native. Get started by going to the AWS AppSync console, specify the data for your app with simple code statements, and AppSync will manage everything needed to store, process, and retrieve the data for your application.

State and data management are the foundations of high quality web and mobile applications in use today. Unfortunately it's often difficult to setup infrastructure for accessing data in a scalable, efficient and secure manner and easily integrate into your frontend or client framework of choice. In this session you'll see how using GraphQL with AWS AppSync allows your clients to quickly and securely access data in the cloud using Amazon DynamoDB, AWS Lambda or Amazon Elasticsearch. You'll see how flexible the system is to let web and mobile application developers define their data structures, mix and match the backing stores using GraphQL resolvers, and customize mapping templates as

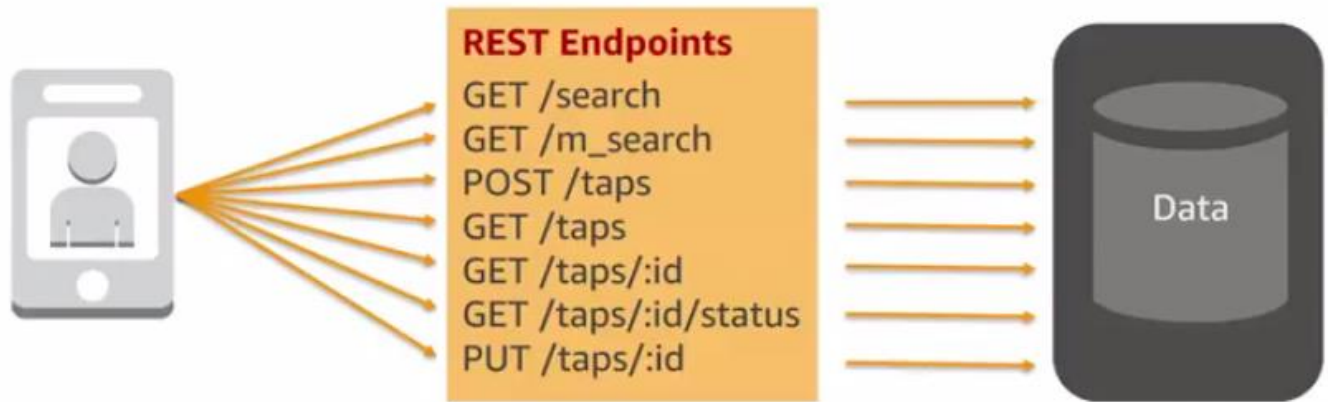
needed. We'll take you through effective use of GraphQL queries, mutations, and subscriptions for batch and realtime needs. Finally you'll see how easy it is for application developers to leverage the system using React Native, iOS and JavaScript web applications.



For mobile apps development today, you will need to do identity and access management for your users and you can use **IAM** and **Cognito** services for that. You also need to make your apps available for all form factors, you can also use the **Lambda**, **API Gateway** and the entire **Serverless** stack to build a scalable and reliable backend for your apps. Data is complex to manage with things like **synchronization**, **offline data availability** and **storing** them efficiently with managed state.

How do we do this today?

REST API



- ✓ Trivial to set up
- ✓ Standard HTTP Calls

- ✗ Relationships
- ✗ Lists with reduced information
- ✗ Query support
- ✗ Ordering and paging
- ✗ Notifications

AWS re:Invent

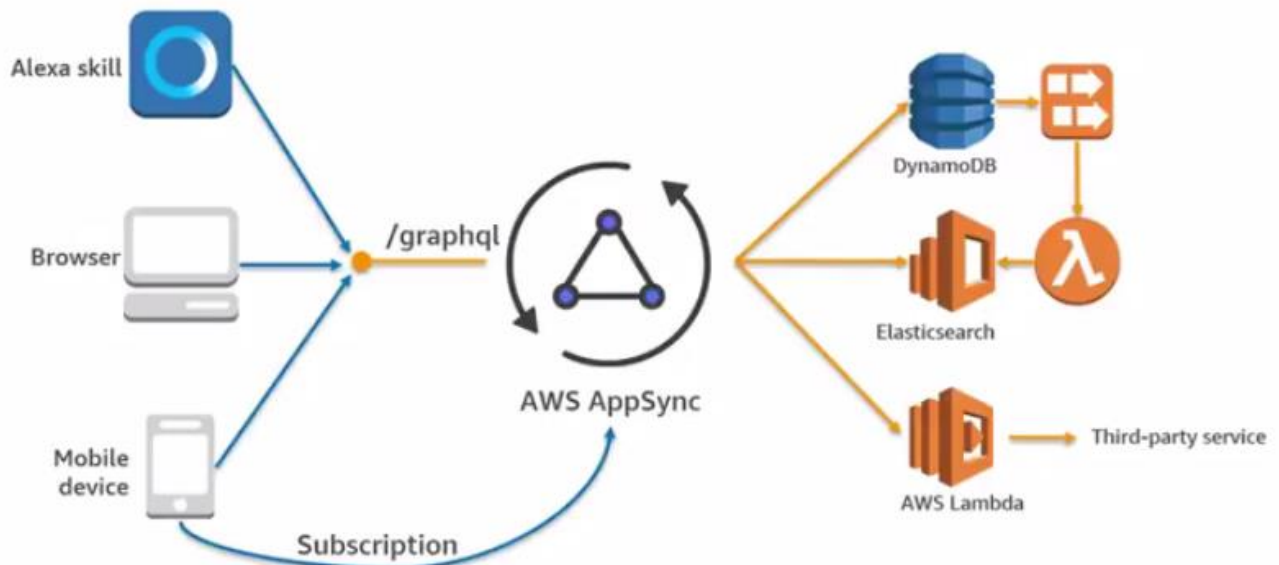
© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



We currently use **REST** or **SOAP APIs** where we set up a bunch of endpoints. But this is not the best use for mobile because of the reasons above. It is not easy to express relationships between the data in the REST API without doing joins in some query language.

Is there a better way?

MapTap with AWS AppSync



AWS re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



What is GraphQL?

- A query language for APIs
- A runtime for fulfilling queries with existing data
- A strongly typed contract between client and server applications

It's a great way for client applications to interact with data

GraphQL is not a database or a graph database, it is a protocol that clients can use to extract information from a server. GraphQL can be iterative, meaning that if you already have a huge infrastructure with a lot of data everywhere, you can add GraphQL on top of it and start building modern apps on top of legacy systems.

A query language for APIs

Queries read data

```
query {  
  post(id: 1) {  
    id  
    title  
  }  
}
```

Mutations write data

```
mutation {  
  createPost(title: "Intro to GraphQL") {  
    id  
    title  
  }  
}
```

Subscriptions are pushed data in real time

```
subscription {  
  onCreatePost {  
    id  
    title  
  }  
}
```

There are **3 base operation types in GraphQL**, **queries** that read data, **mutations** that write data, and **subscriptions** that are pushed data in real time.

How does GraphQL work?


```
type Query {  
  getTodos: [Todo]  
}  
  
type Todo {  
  id: ID!  
  name: String  
  description: String  
  priority: Int  
  dueDate: String  
}
```

Define your data

The first step in building a GraphQL API implementation is to **define your data model** using a **GraphQL Schema Definition Language** (SDL), this gives you the basic pieces of your schema. The **type Query {}** is going to be our root query type that is the entry point for all read operations that will occur against/within our applications data.

How does GraphQL work?

```
type Query {  
  getTodos: [Todo]  
}  
  
type Todo {  
  id: ID!  
  name: String  
  description: String  
  priority: Int  
  dueDate: String  
}
```



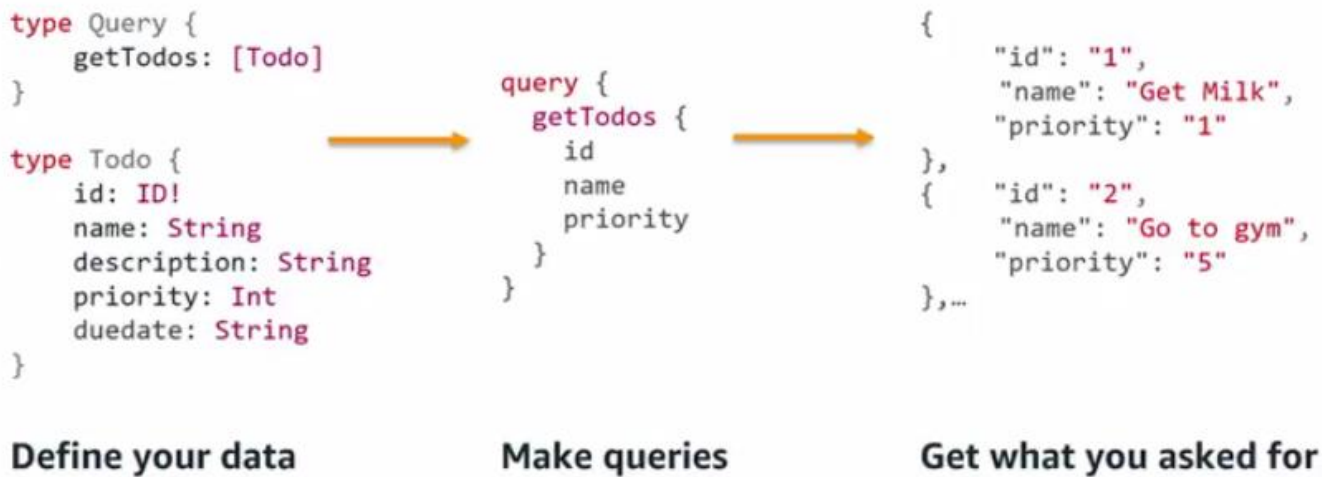
```
query {  
  getTodos {  
    id  
    name  
    priority  
  }  
}
```

Define your data

Make queries

After you define your schema, you can start writing queries. GraphQL is strongly typed meaning that all your queries are type checked before they can be executed.

How does GraphQL work?



You can get exactly what you need in your queries as above

Focus on apps, not infrastructure



Introducing AWS AppSync

AWS AppSync features

- Managed GraphQL service
- Connect to resources in your AWS account
- Built-in offline support
- Conflict resolution in the cloud
- Enterprise-level security features
- Simple and scalable real-time synchronization

How does AWS AppSync work?



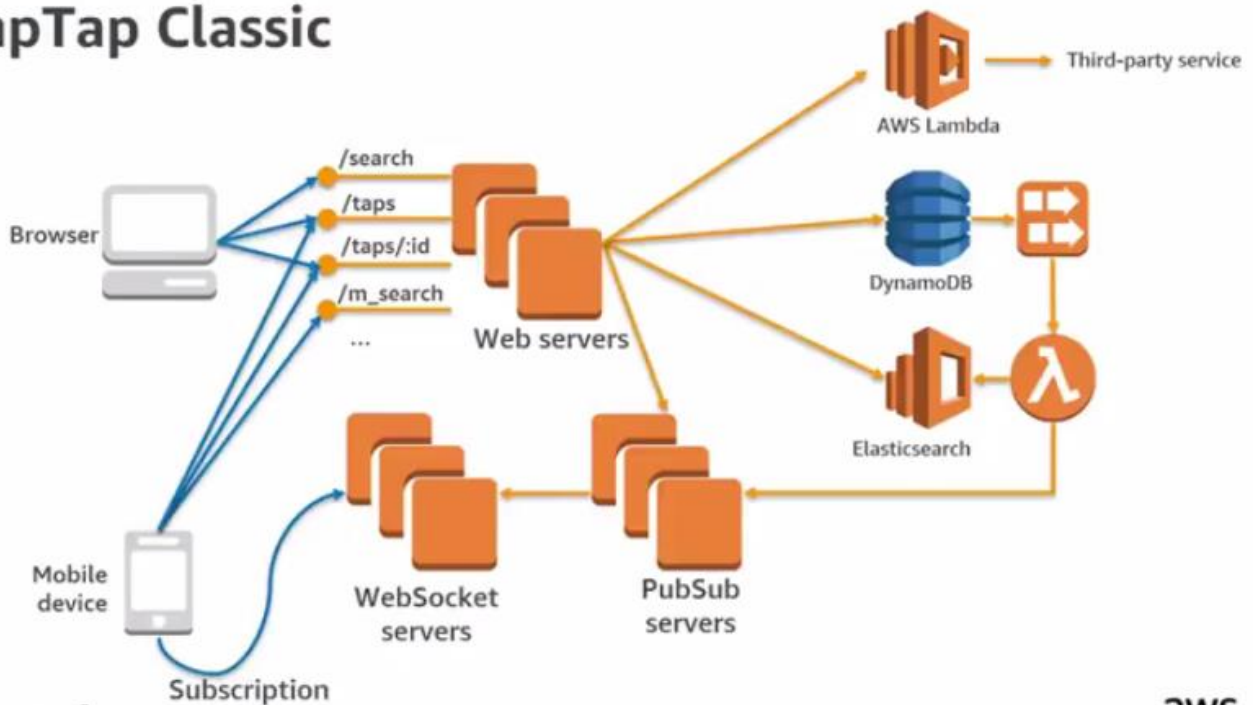
The first step is to create and define your schema, the schema defines your data model and the operations that you have access to, queries, mutations, and your subscriptions. Next you will have to connect to your data sources, this might be data sources already in your AWS account, we also support Lambda, DynamoDB, and Elasticsearch. Finally, you need to use the AWS client tooling or any OSS GraphQL tooling to connect to the GraphQL server and start using it in your applications.

Building data-driven apps with AWS AppSync

- Robust, scalable storage capable of handling millions of events a day
- Geospatial search
- Real-time updates for both mobile and web
- Mobile and web clients

For our Retail store dashboard app, we are going to use DynamoDB as our robust data storage engine. We are going to use Elasticsearch as our query index for geospatial data and analytics queries, Elasticsearch gets new data from DynamoDB streams using Lambdas. For real time updates we will be using GraphQL subscriptions, we will use React and React Native

MapTap Classic



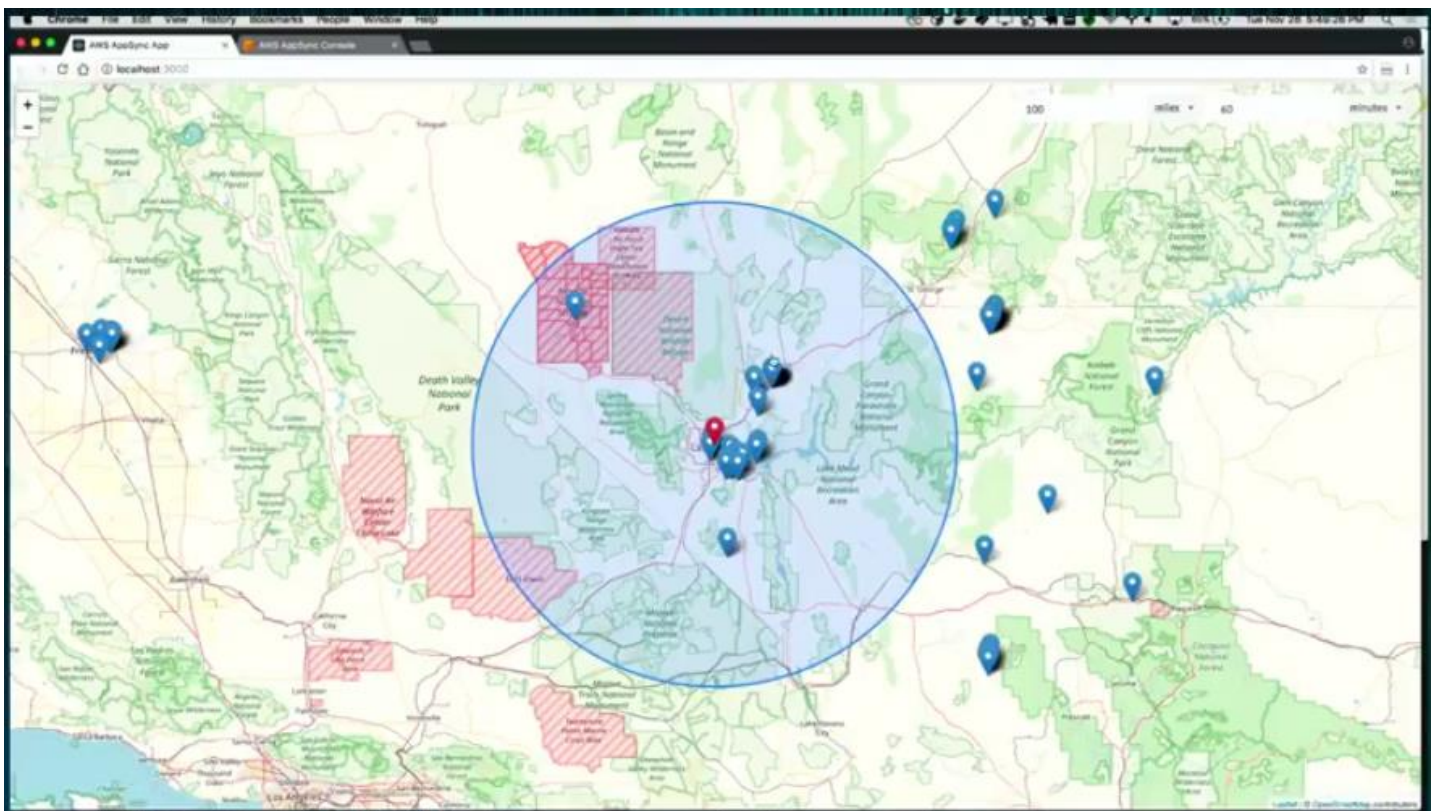
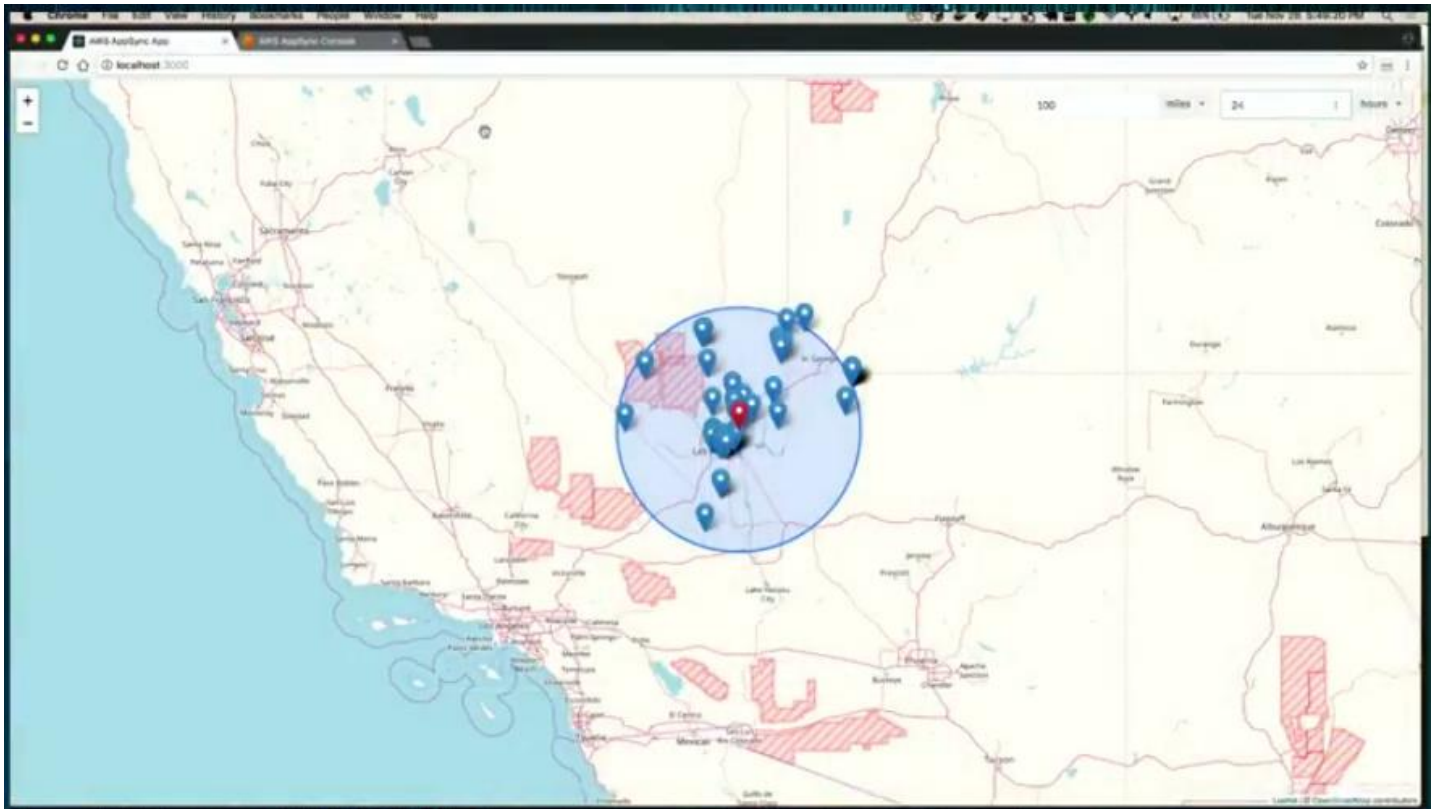
AWS re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

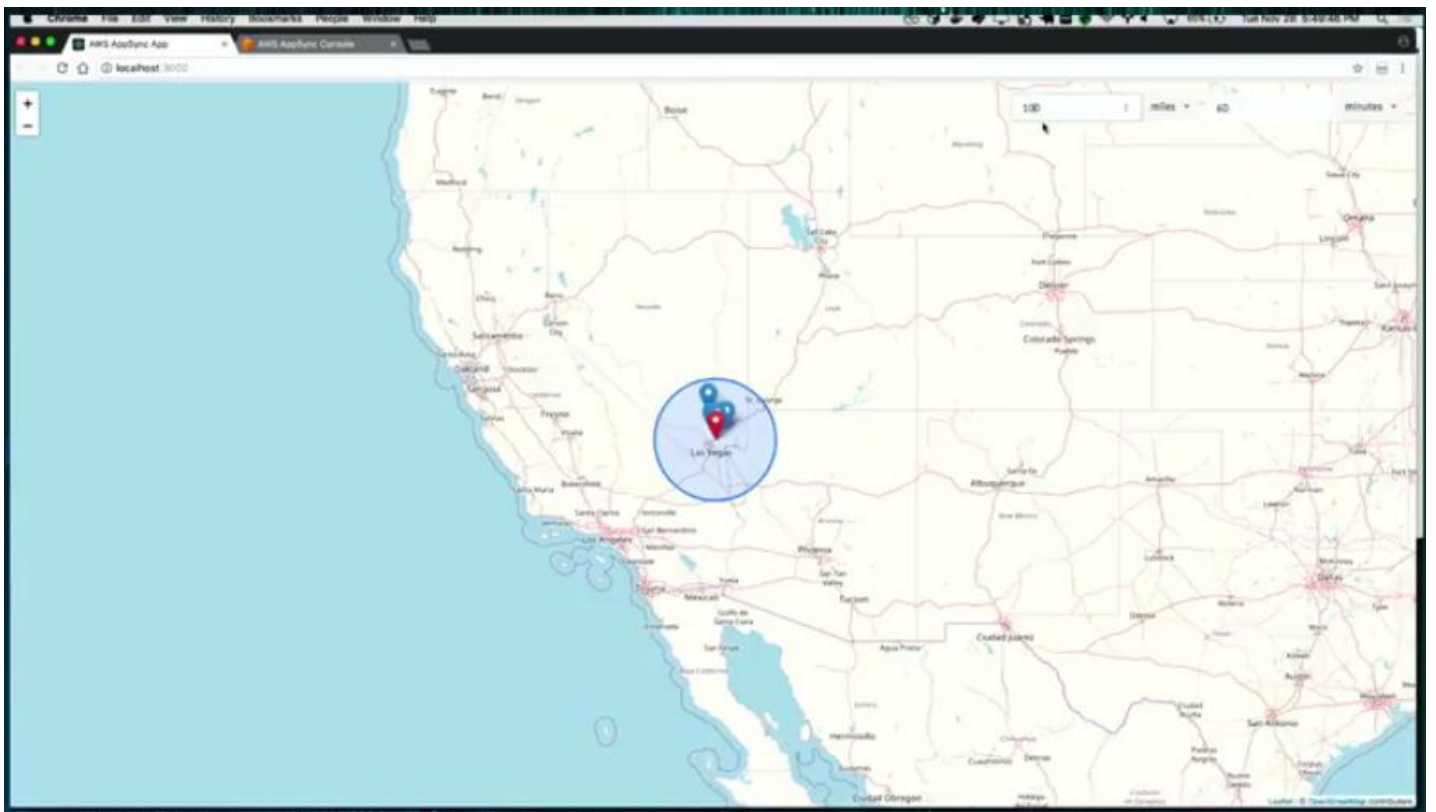
aws

This is what our infrastructure might look like without a service like AppSync. DynamoDB is our record store and source of truth, along with a fleet of web servers running on EC2, ECS, or API Gateway to build your HTTP REST endpoints. You will need **PubSub servers running Redis** to manage your subscriptions, with a suite of **WebSocket servers listening to the PubSub engines** and pushing data over channels down to connected clients.

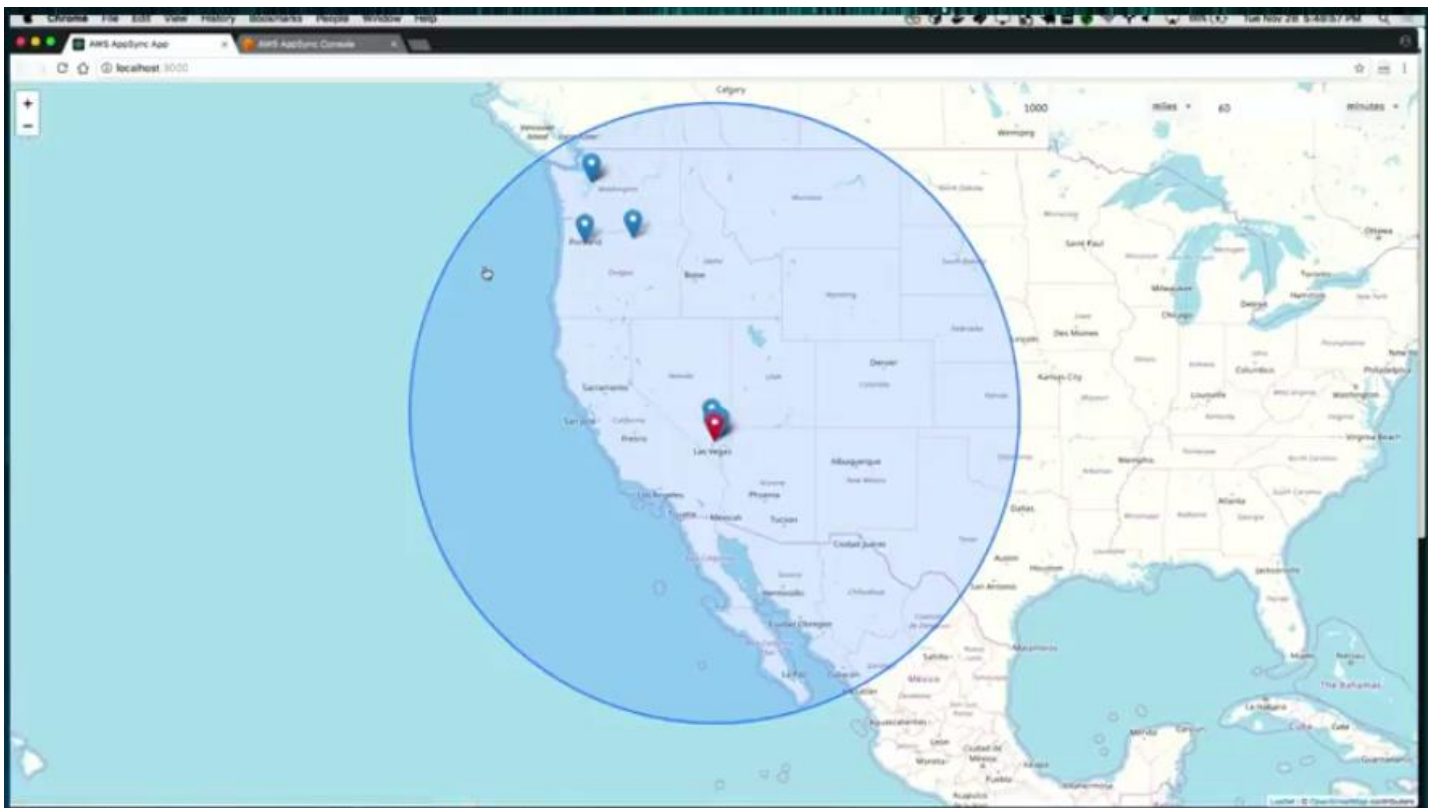
MapTap Demo



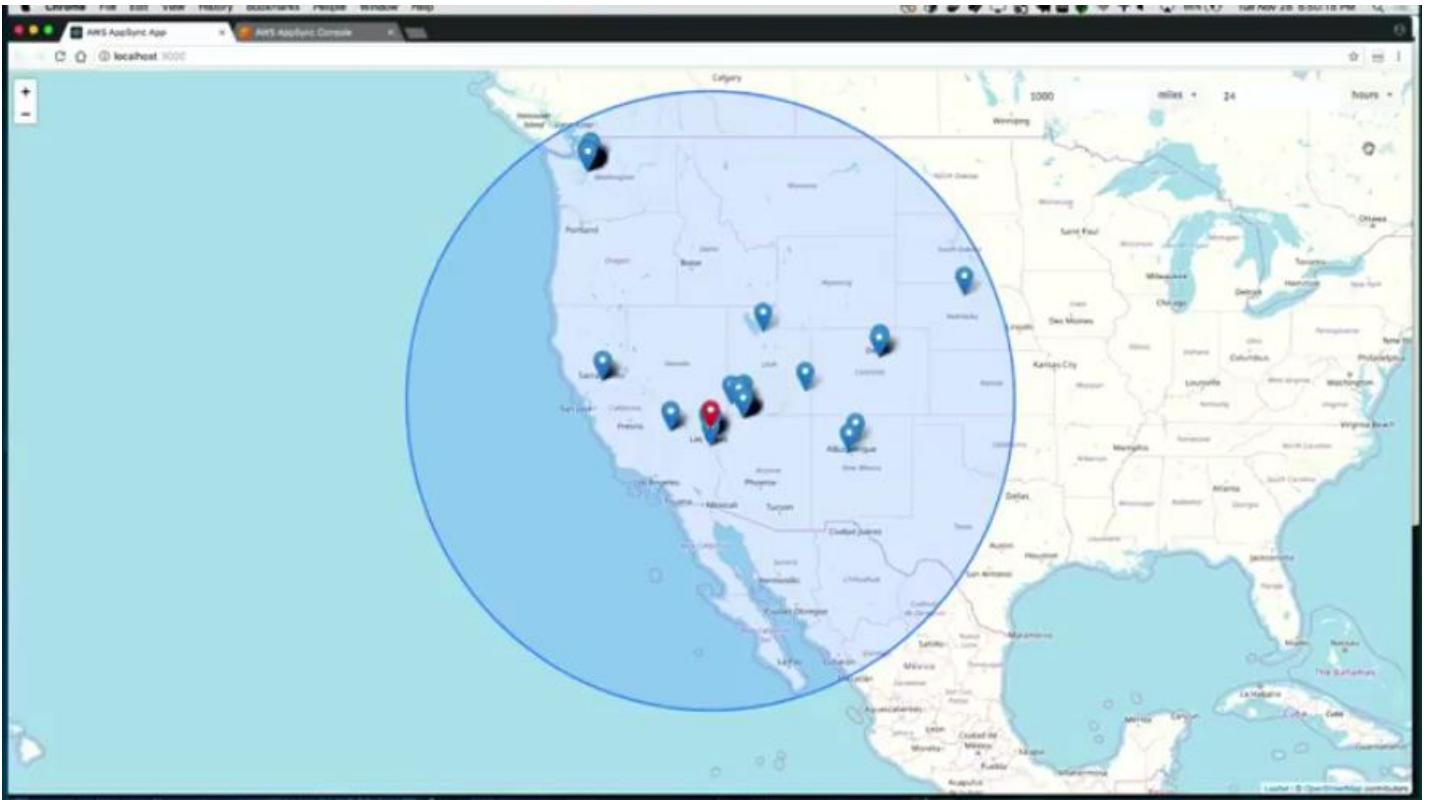
We see a map showing that we have just had a couple of purchases near Vegas in the last 60 minutes



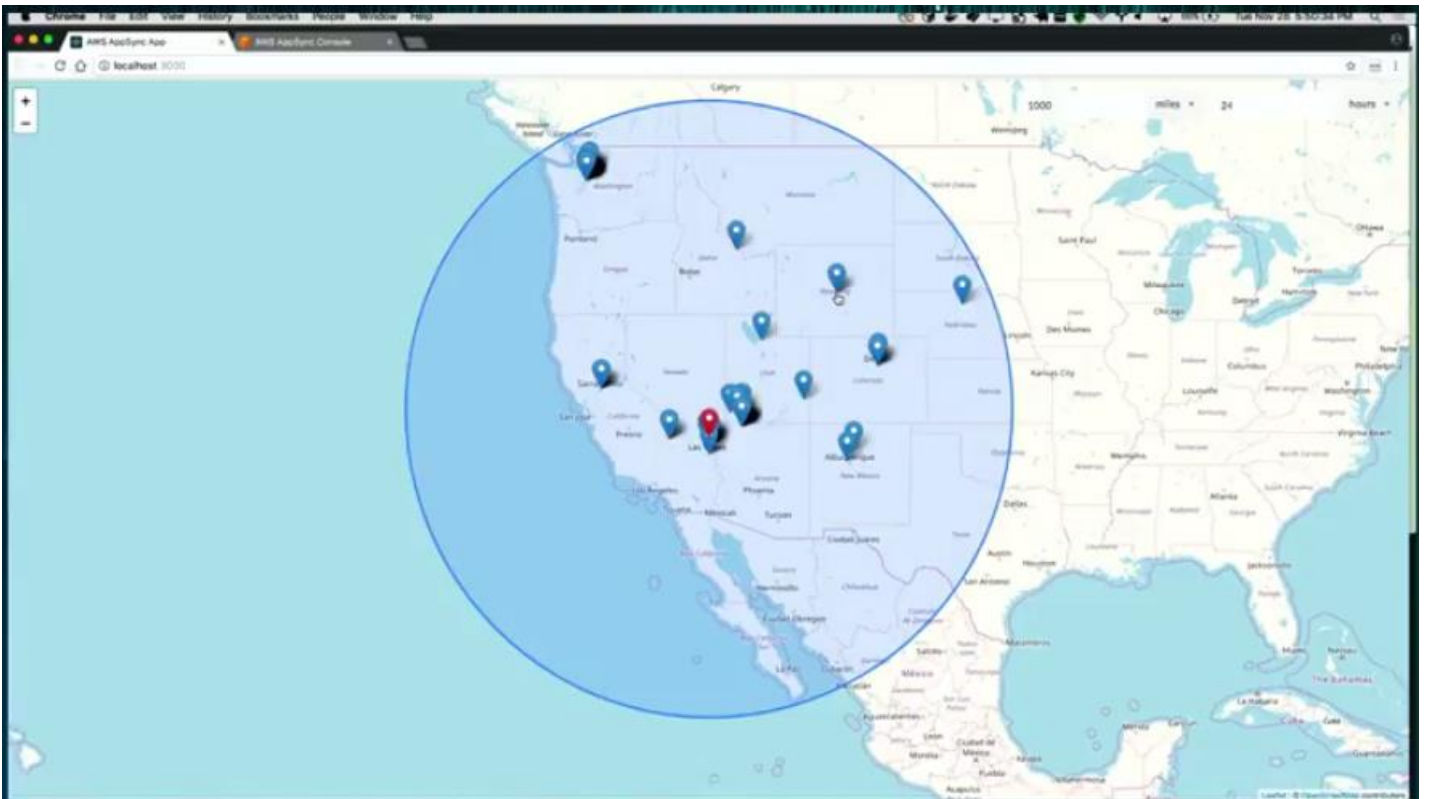
What if we want to do the query for purchases over 1000 miles instead of the current 100 miles?



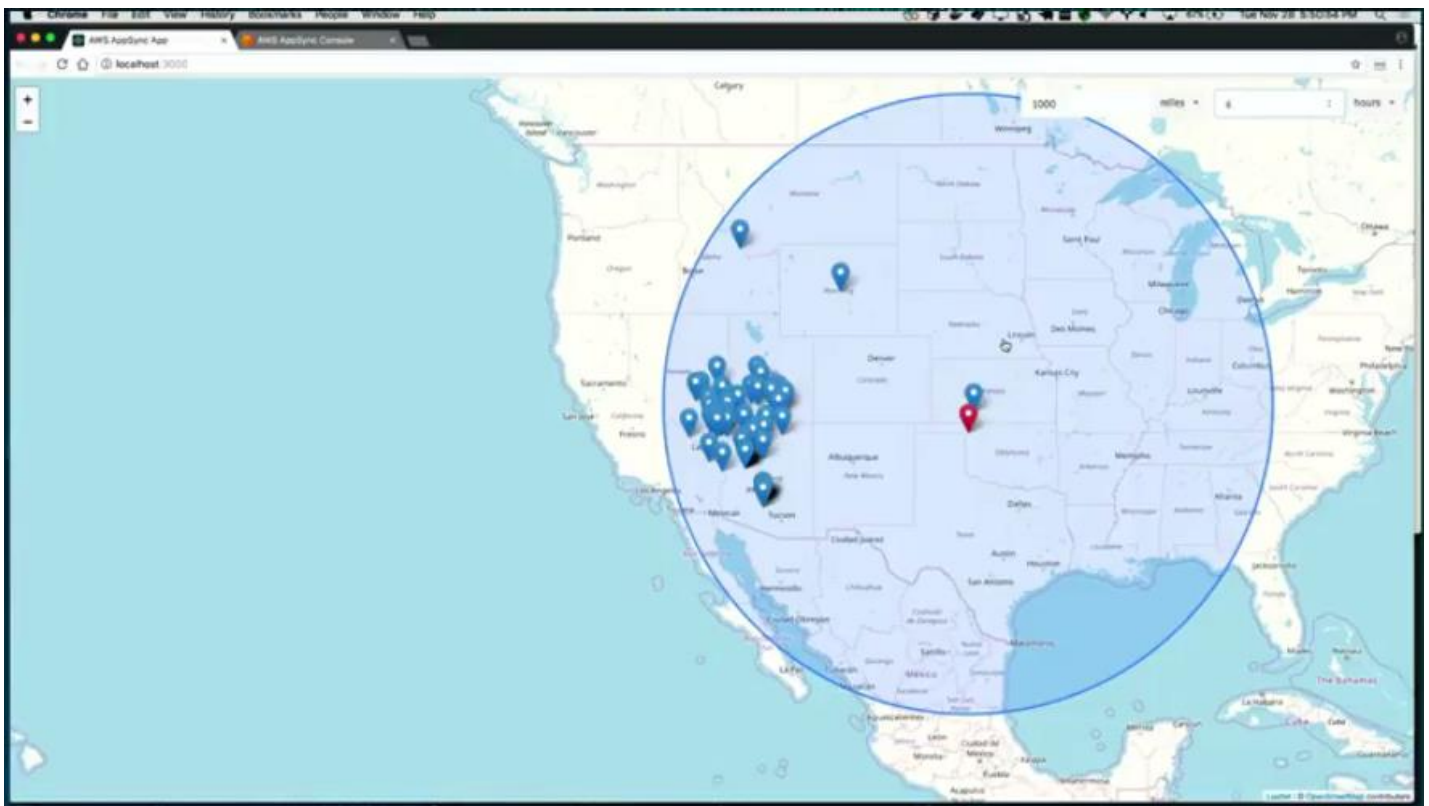
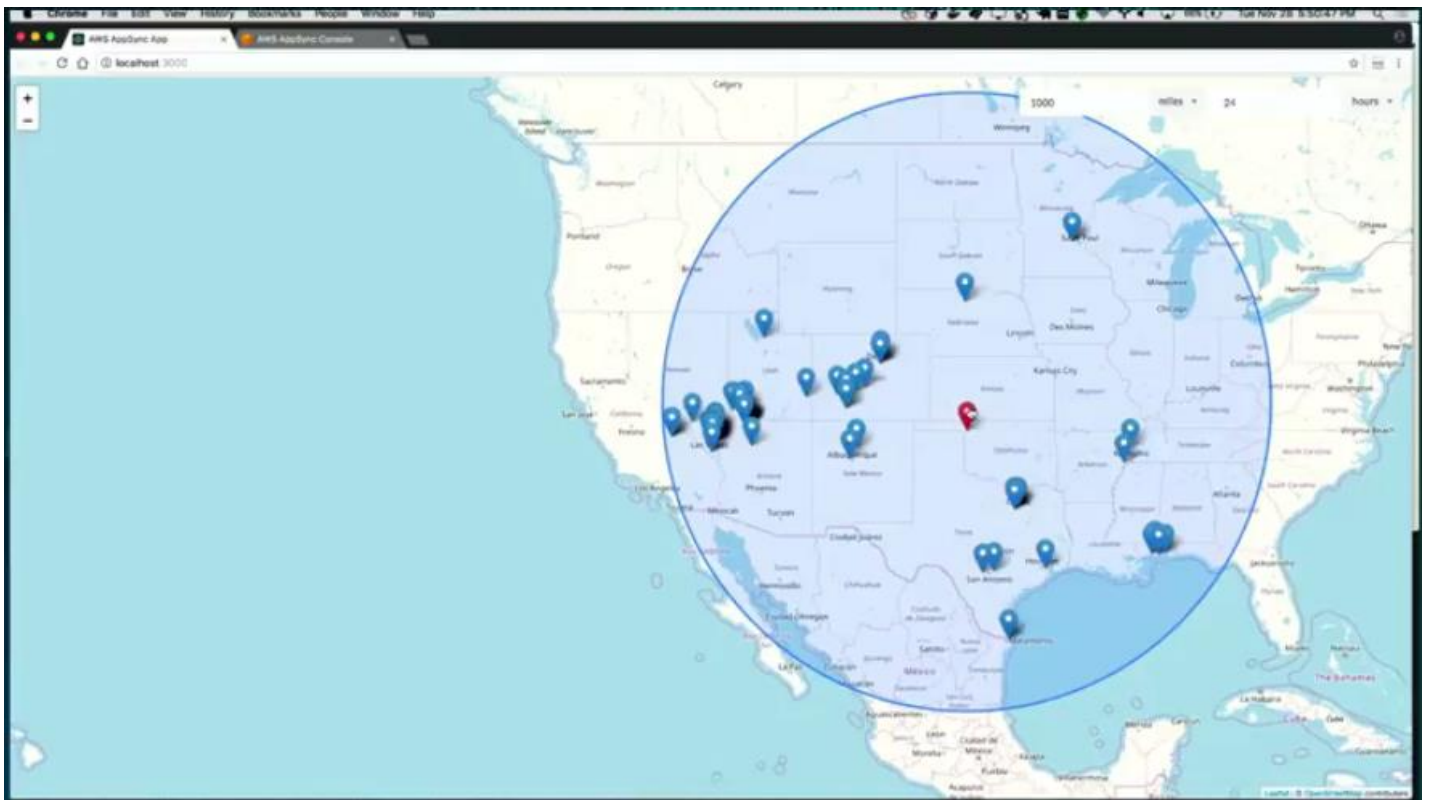
This is now a different query for the last 5 mins



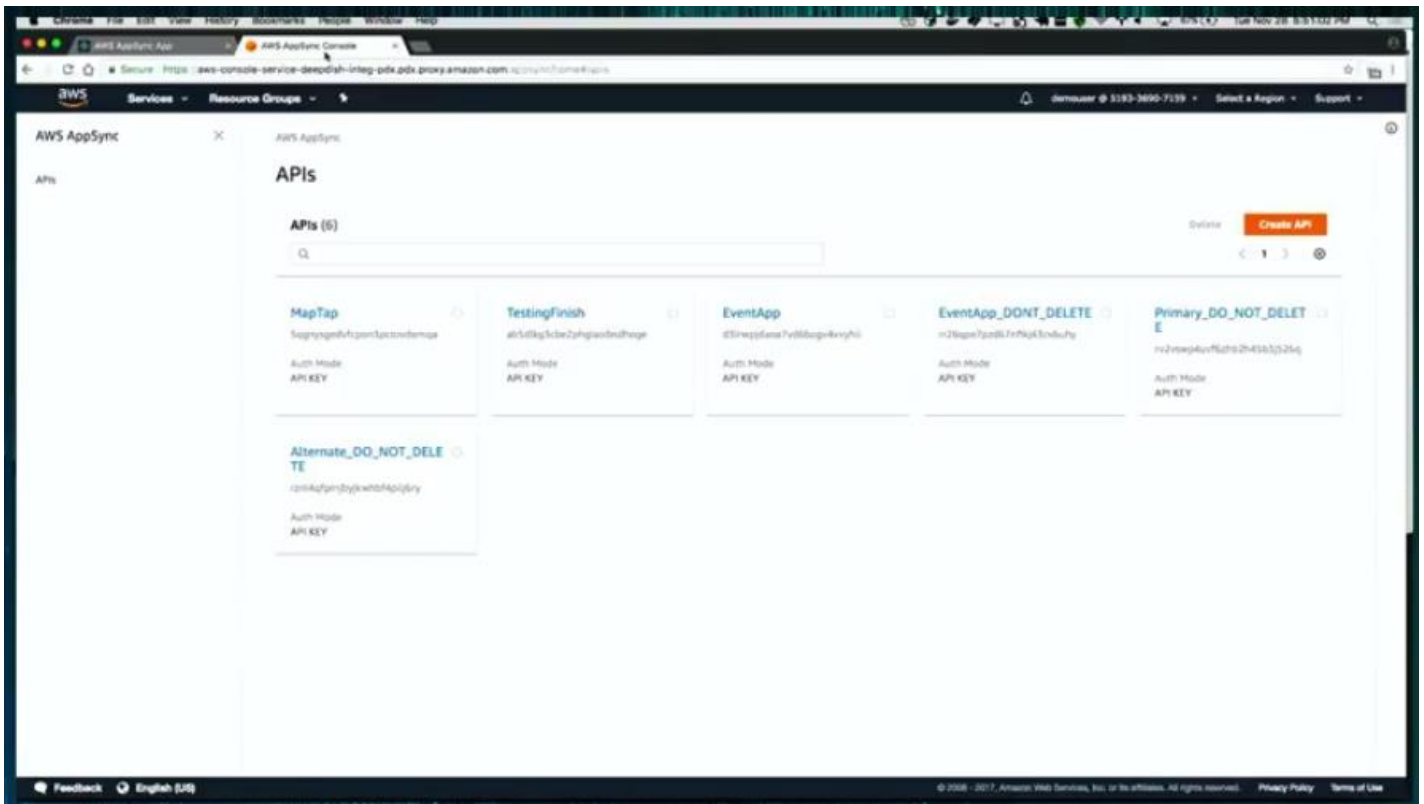
We can also do a query for purchases over the last 1 hour.



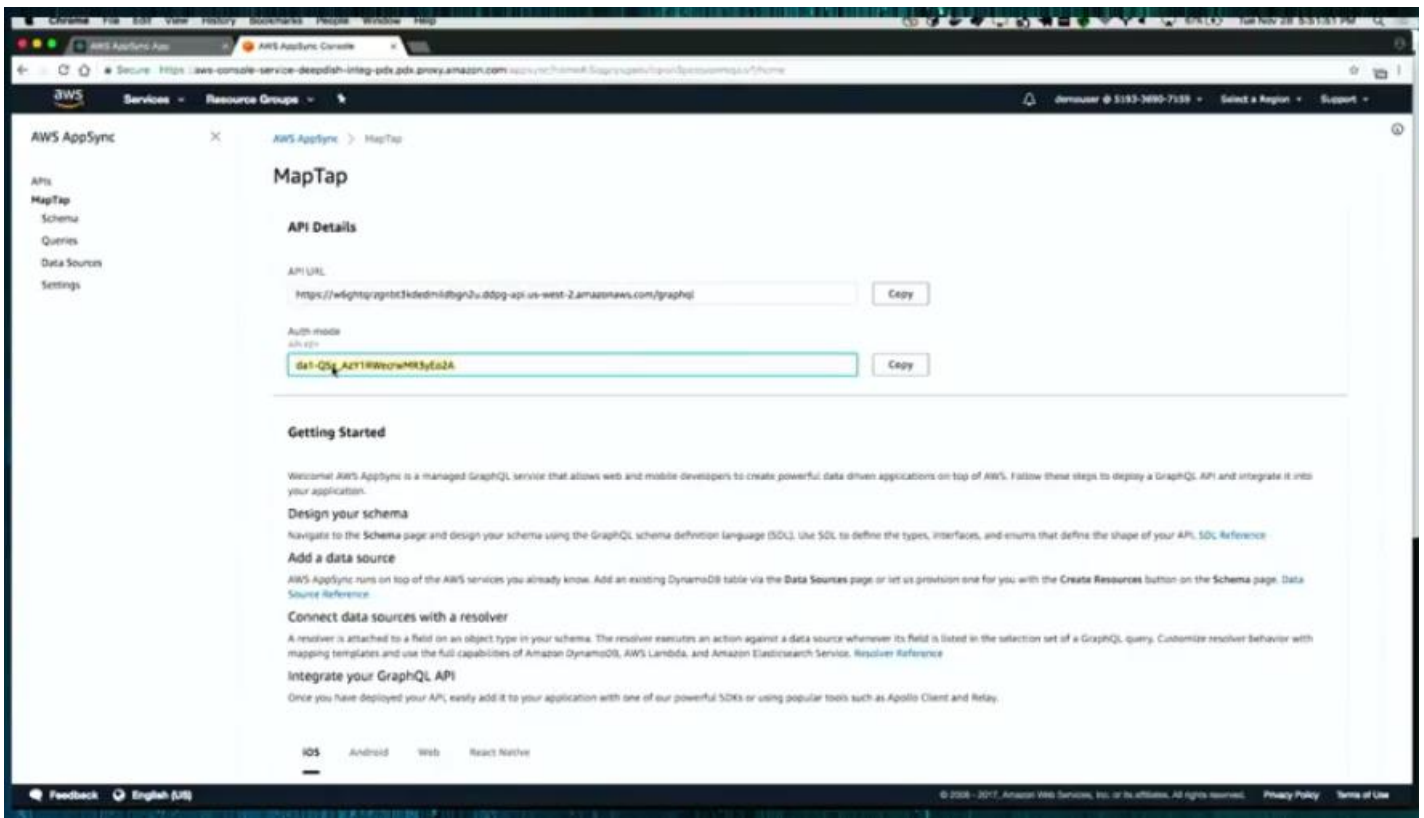
We can now *simulate a new purchase* by clicking on some spot with the search area. This will *fire a GraphQL mutation through AppSync, that goes through DynamoDB, streamed into Elasticsearch*, and we can immediately start querying it again in real time.



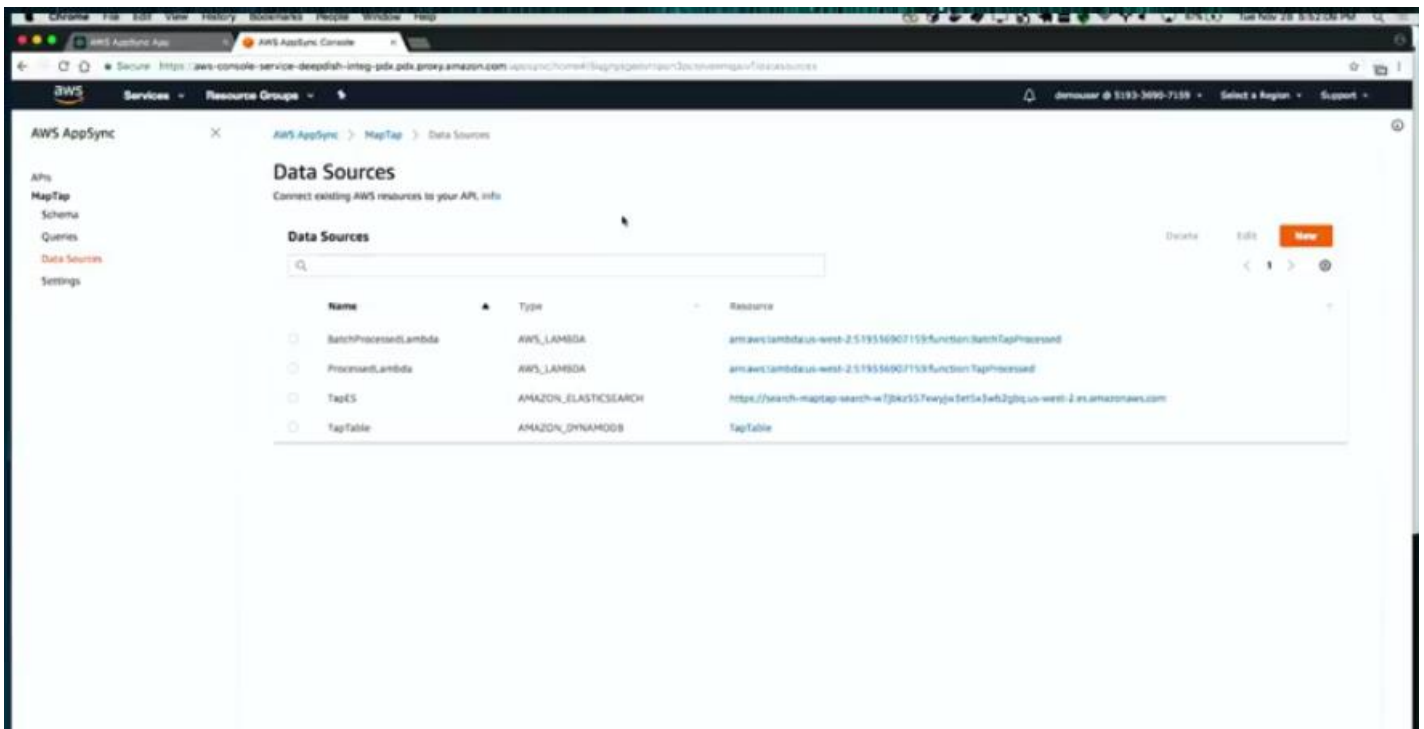
Let us now see how this is done below.



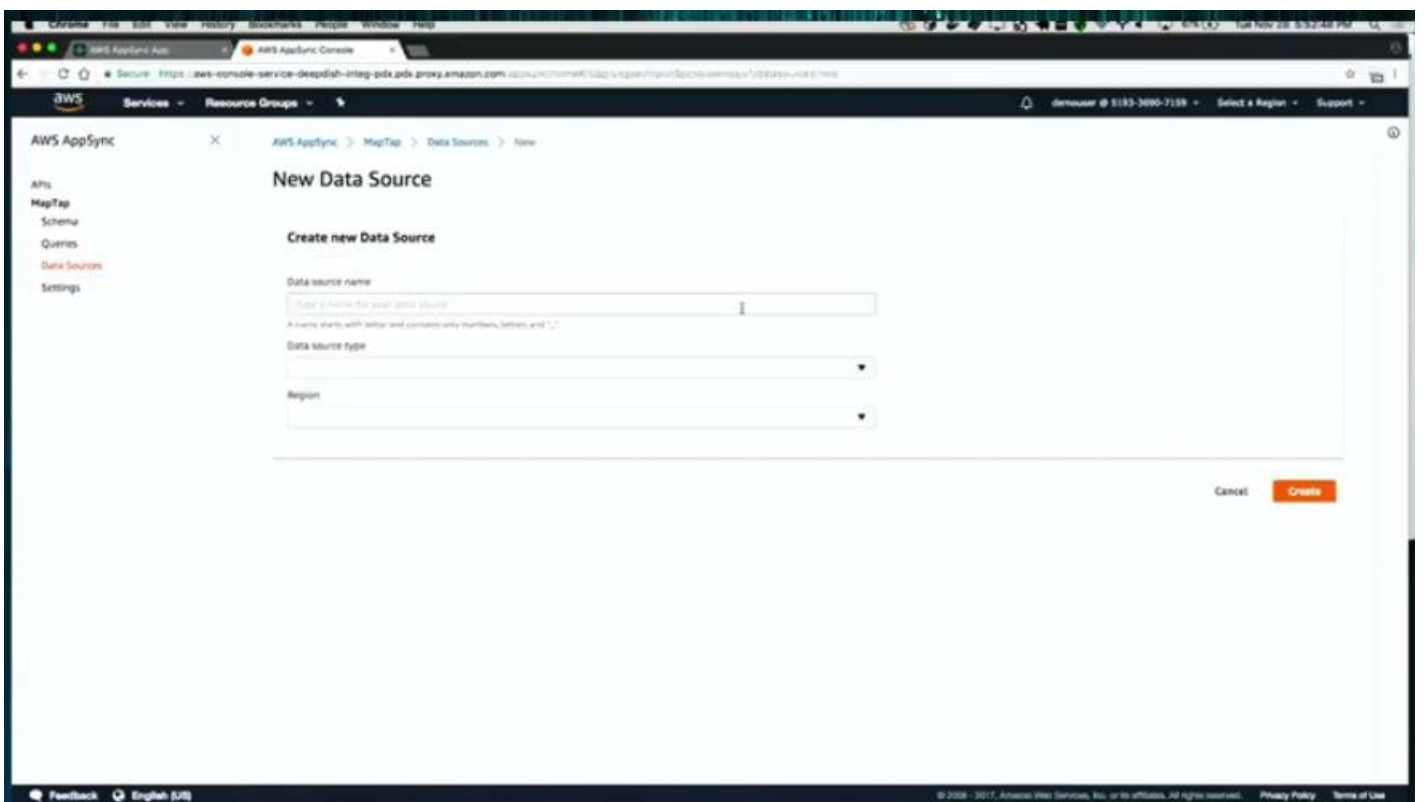
This is the AWS AppSync portal, it is the quickest way to get started with AppSync.

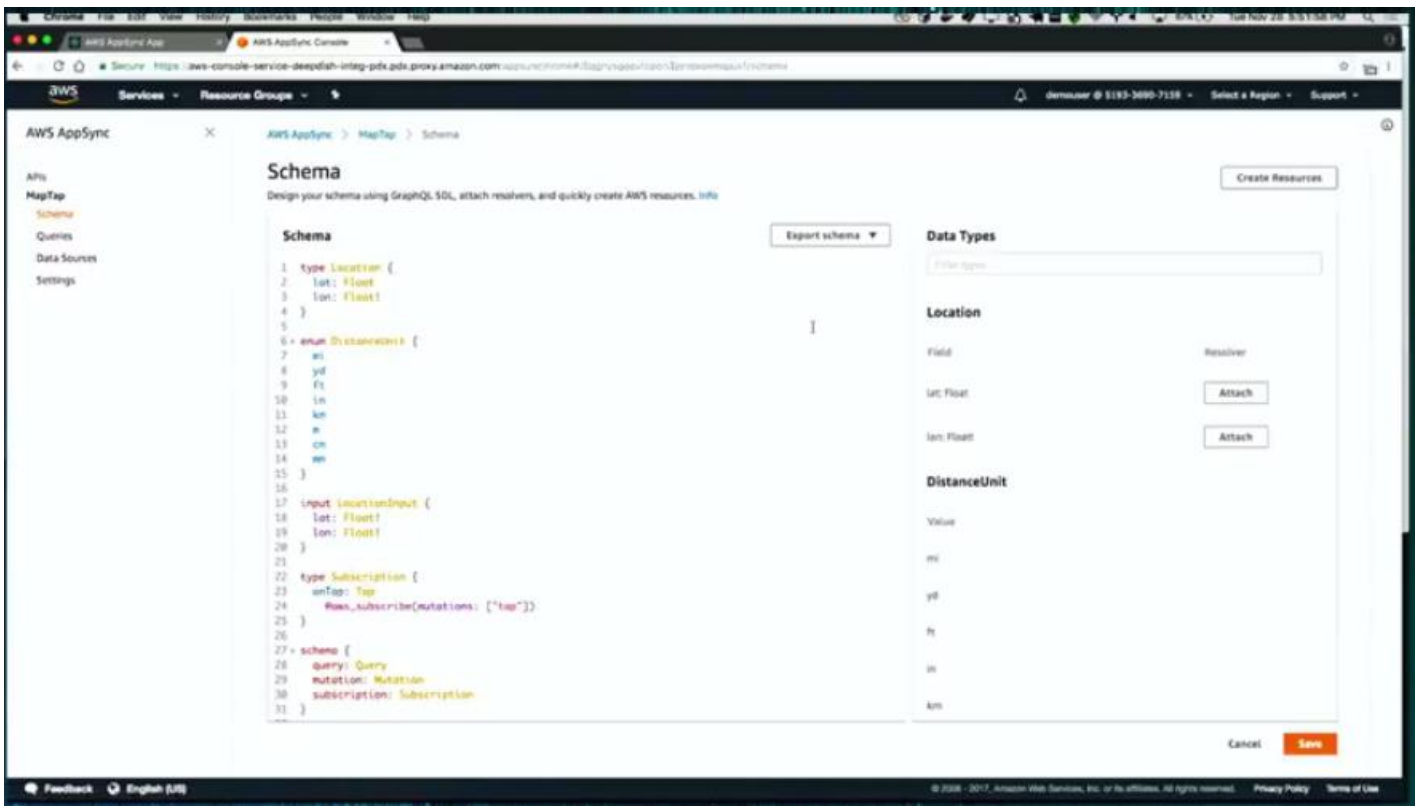


Here is our MapTap applications API homepage, the only endpoint you need to gain access to your entire schema is the **/graphql** endpoint for issuing queries to. Above, we currently have the API Key authentication enabled. you can put this token in the client request header for request authorization

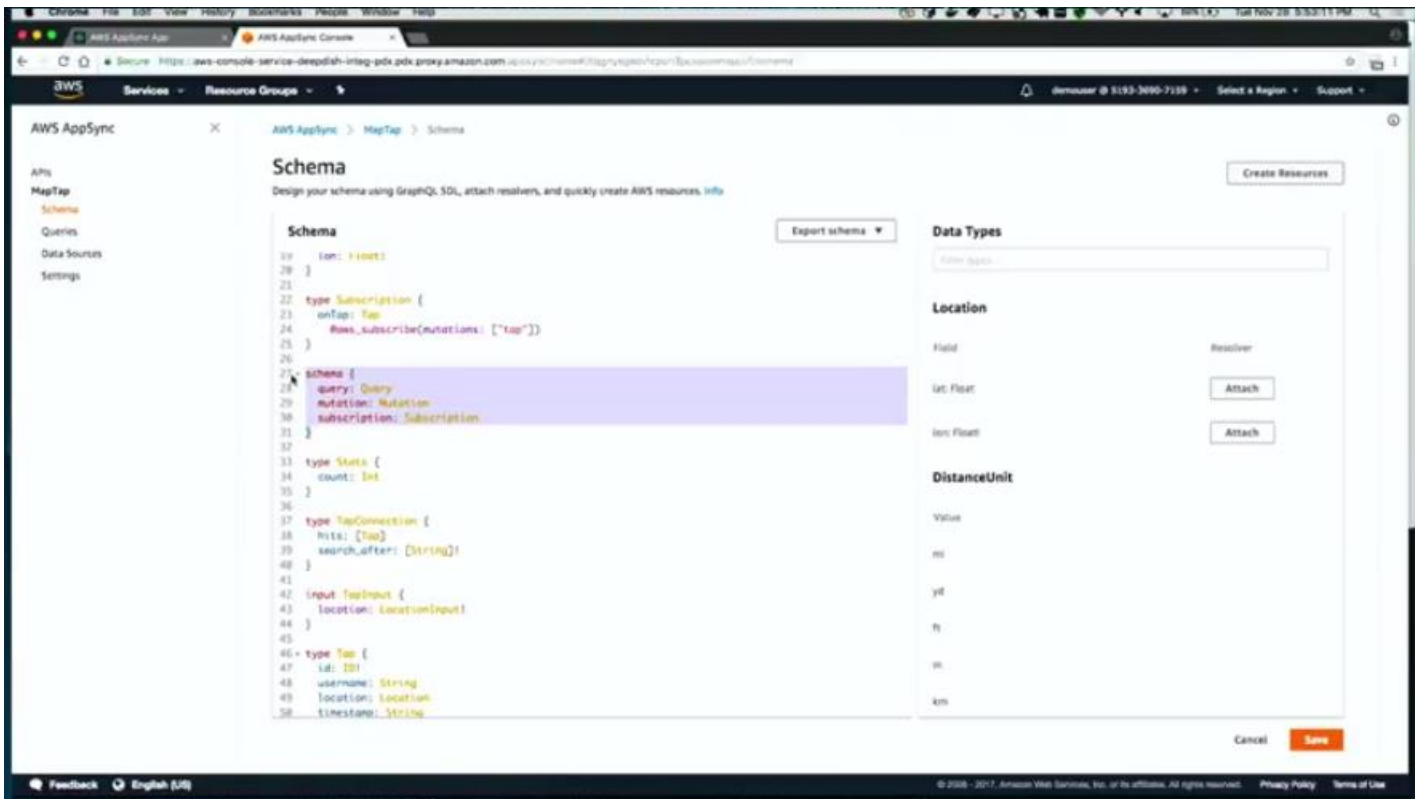


We also have 4 data sources hooked up with names that we can use for them within AppSync. We have an AWS Lambda source called the BatchProcessedLambda, another Lambda function data source called ProcessedLambda that isn't batched, our Amazon ElasticSearch cluster data source that is called TapES, and a DynamoDB table for our screen taps called TabTable. These are the 4 data sources we have at the moment, you can add more if you want.

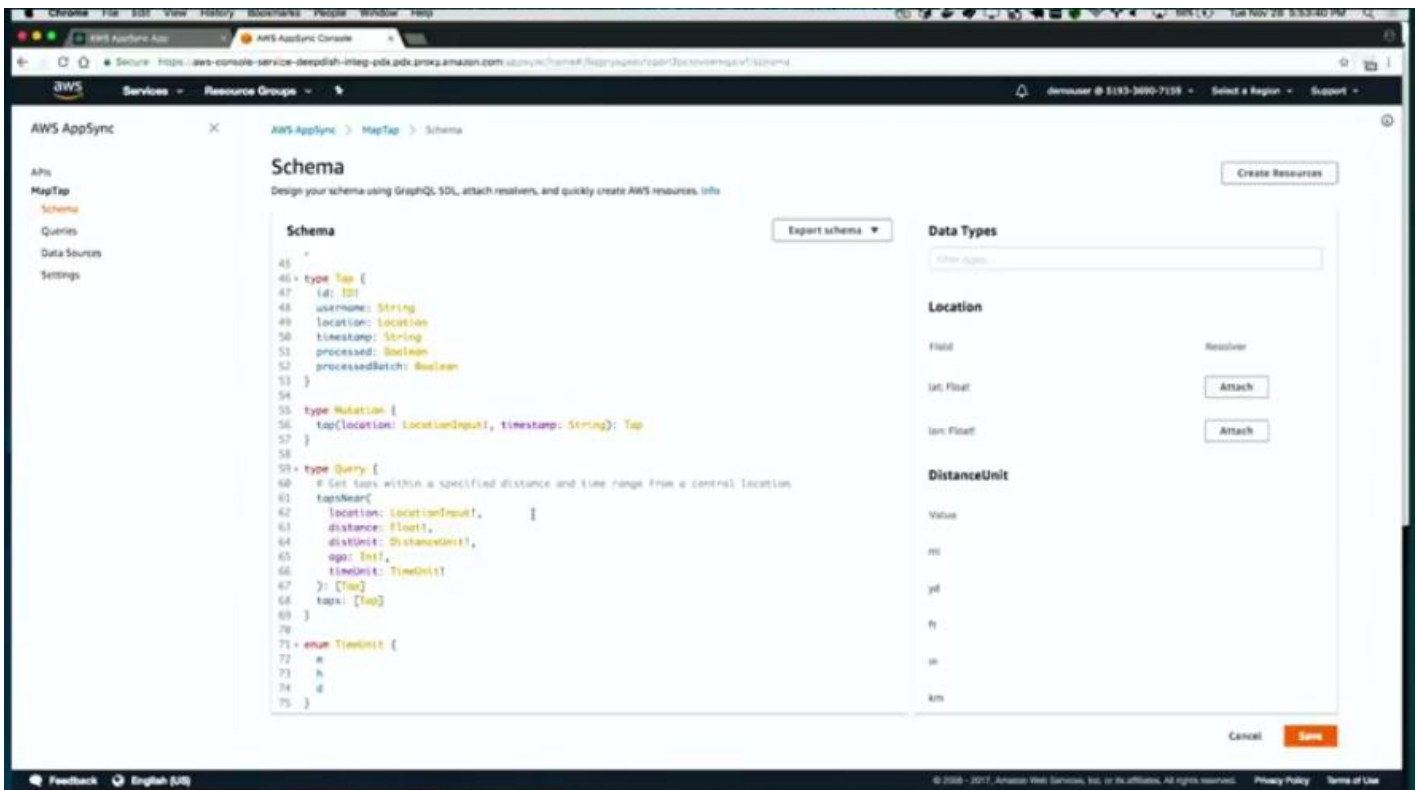




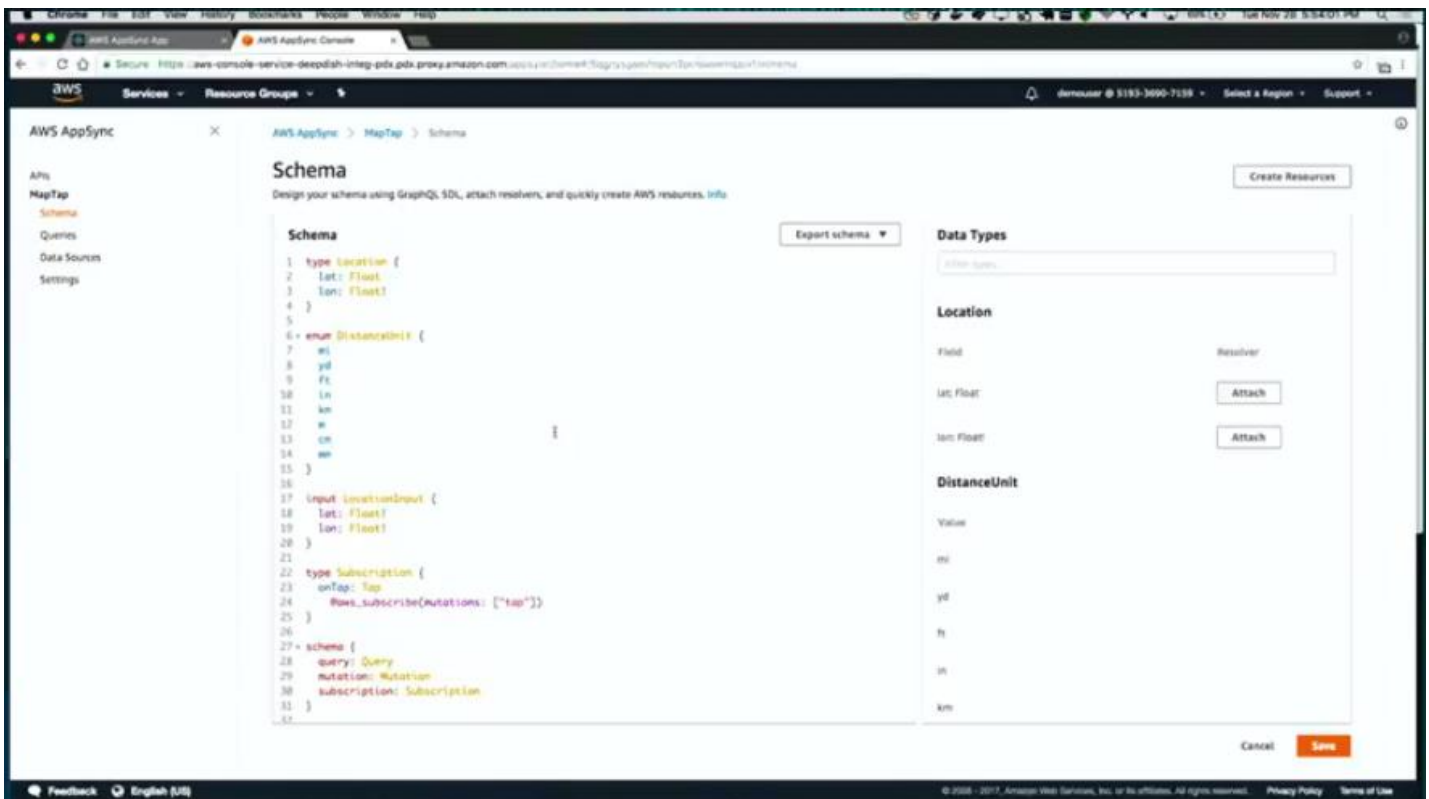
Here is your schema page where a lot of work will be done,



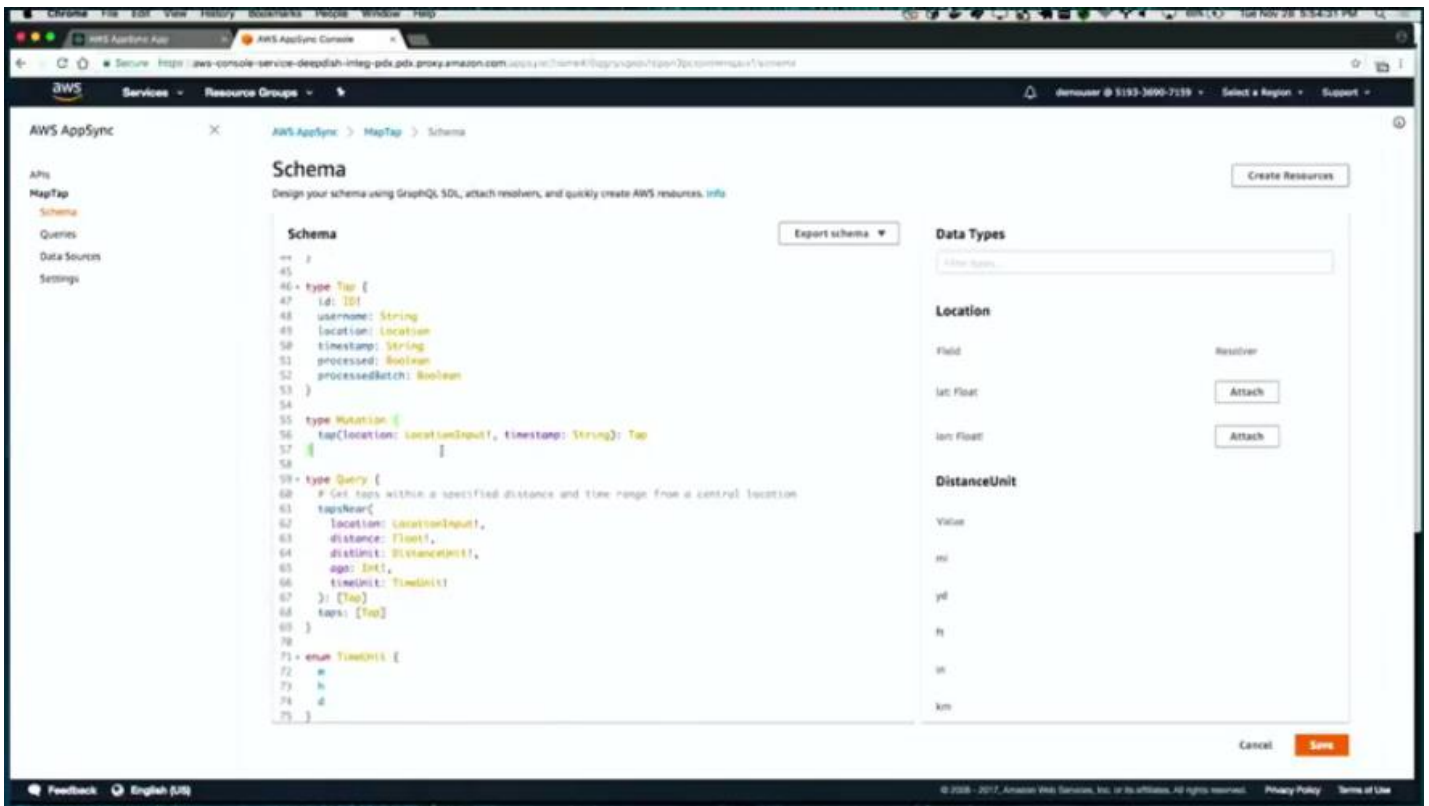
There is always going to be a **root schema definition** with the schema {} object as above, this is where we define the entry points into our GraphQL API with the 3 operations available defined inside.



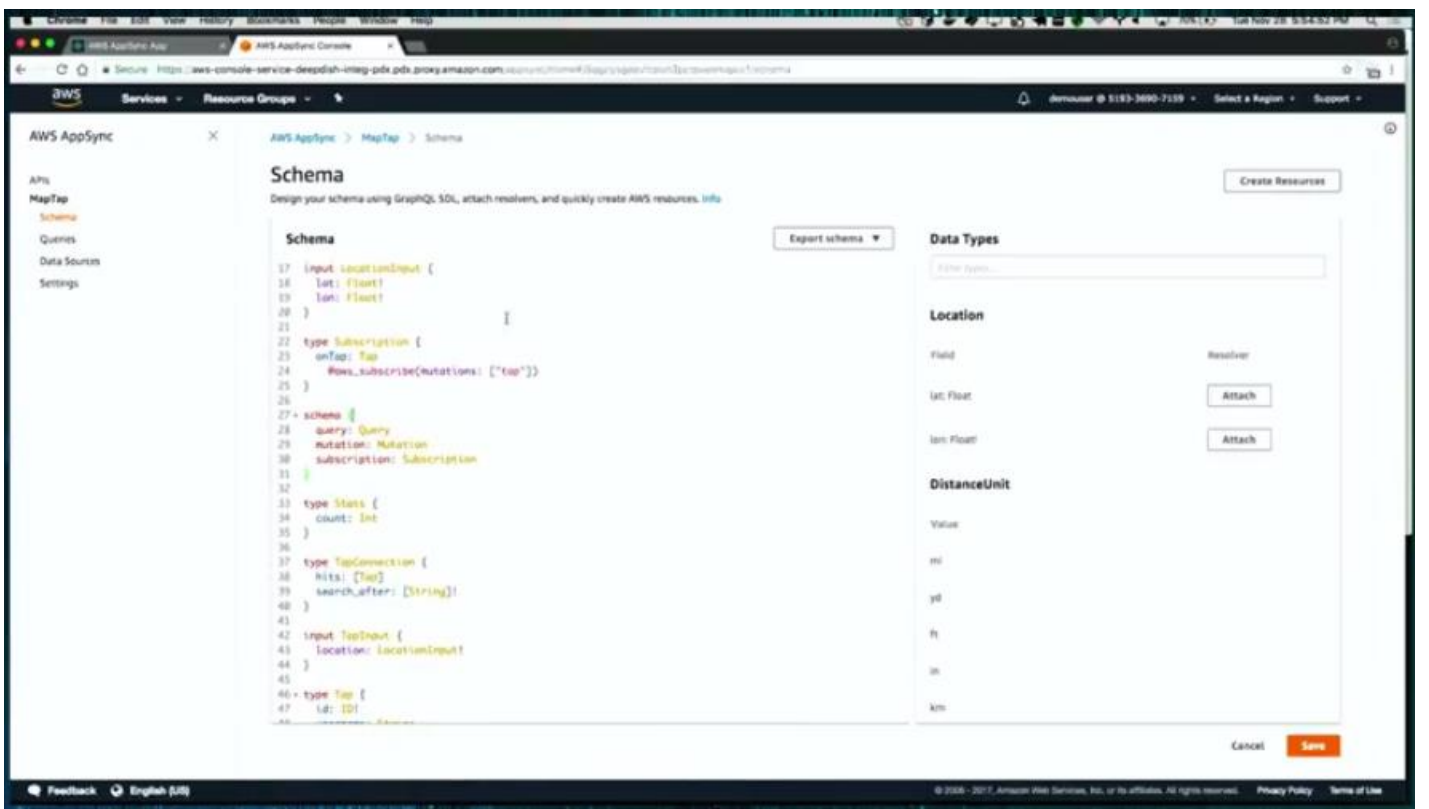
We have also defined 2 possible queries here called **TapsNear** and Taps



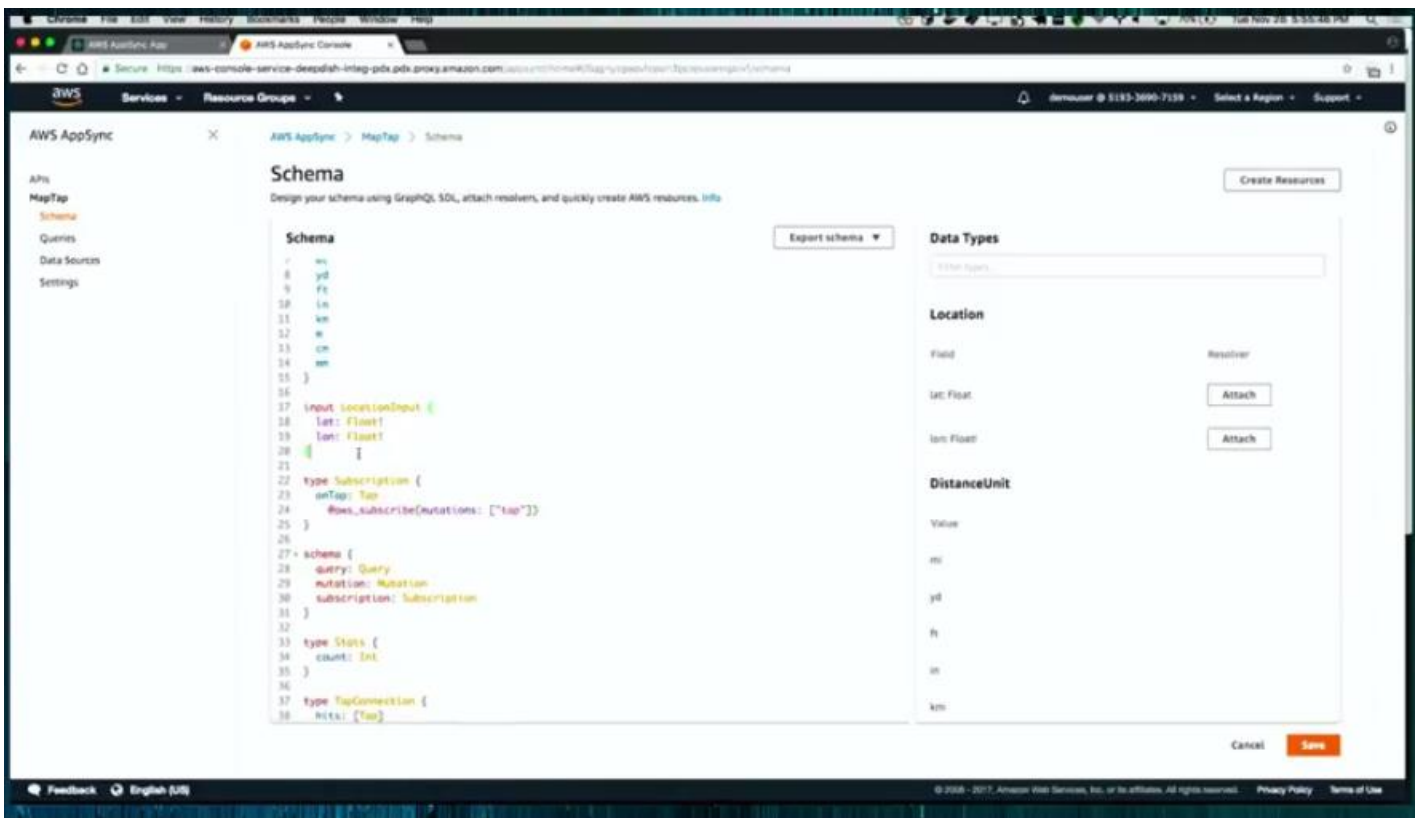
We defined an Enum to coordinate the ES meaning of the distance units.



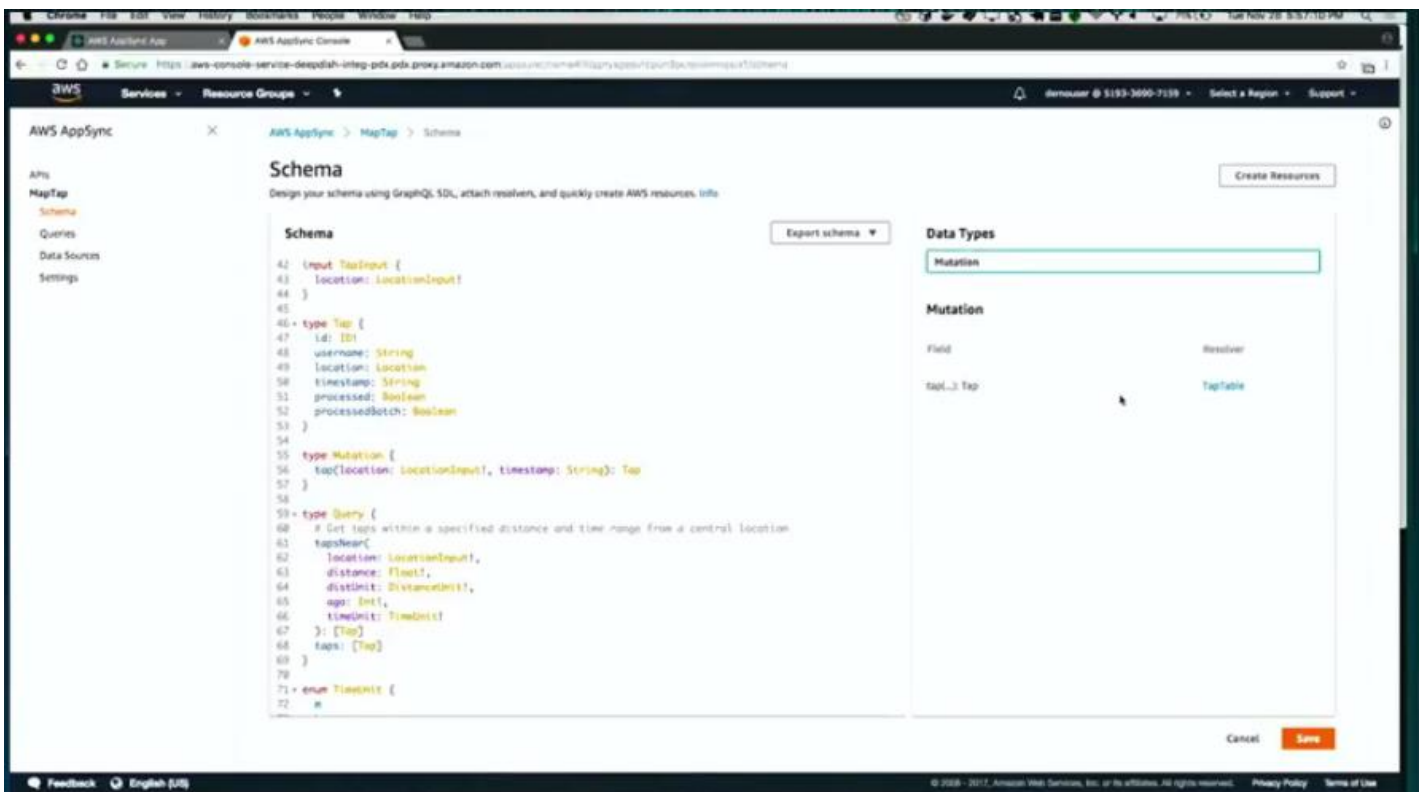
We also defined a single mutation called `tap()` that takes a location and a timestamp, this mutation returns back a `Tap` type that goes into DynamoDB and gets streamed into ES.

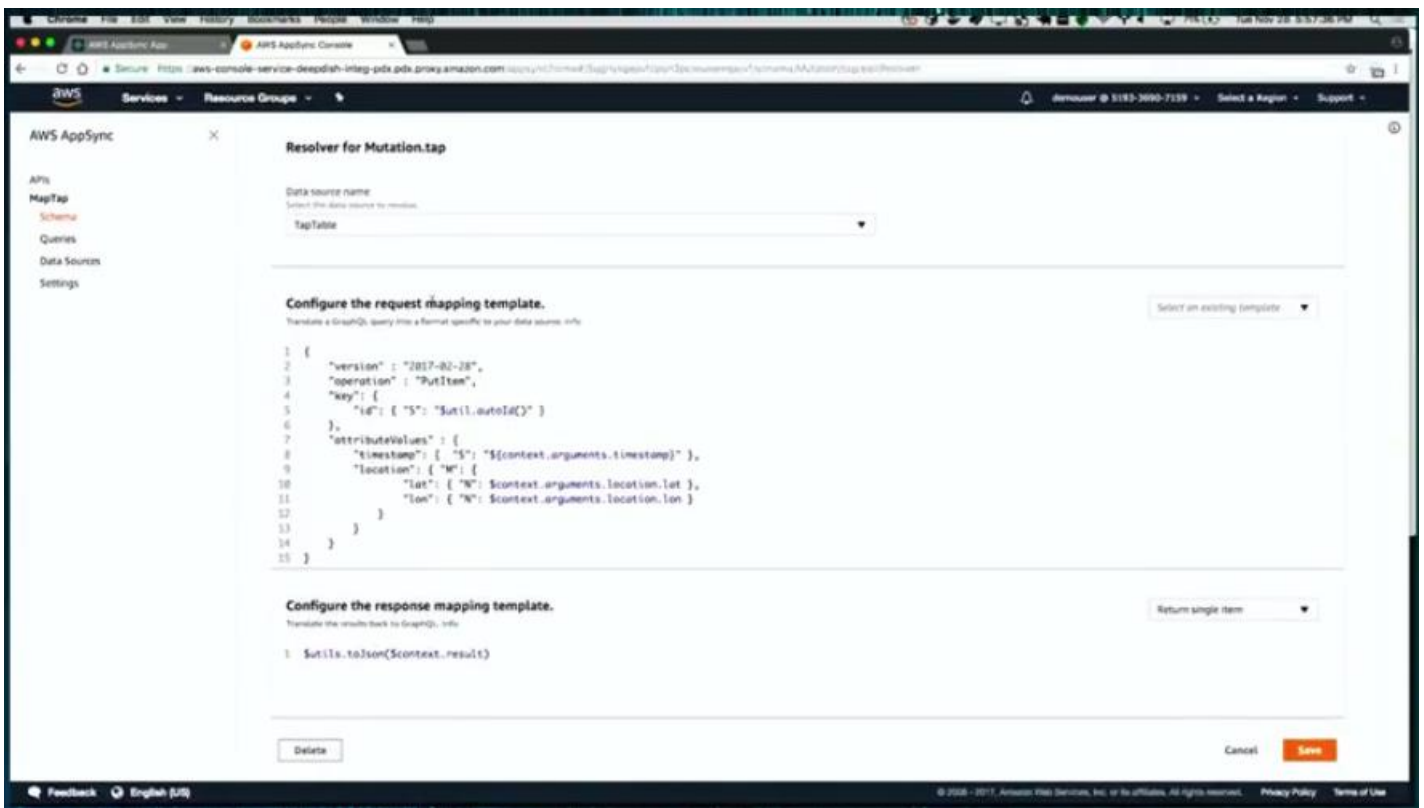


We also have a single subscription called `onTap` that returns a `Tap` type every time a click is made of the screen. The `rows_subscribe(mutations: ["tap"])` is called a directive provided by AppSync. A directive is any annotation that you want to provide to a schema or a query. **A subscription is simply a reaction to a mutation made elsewhere.**

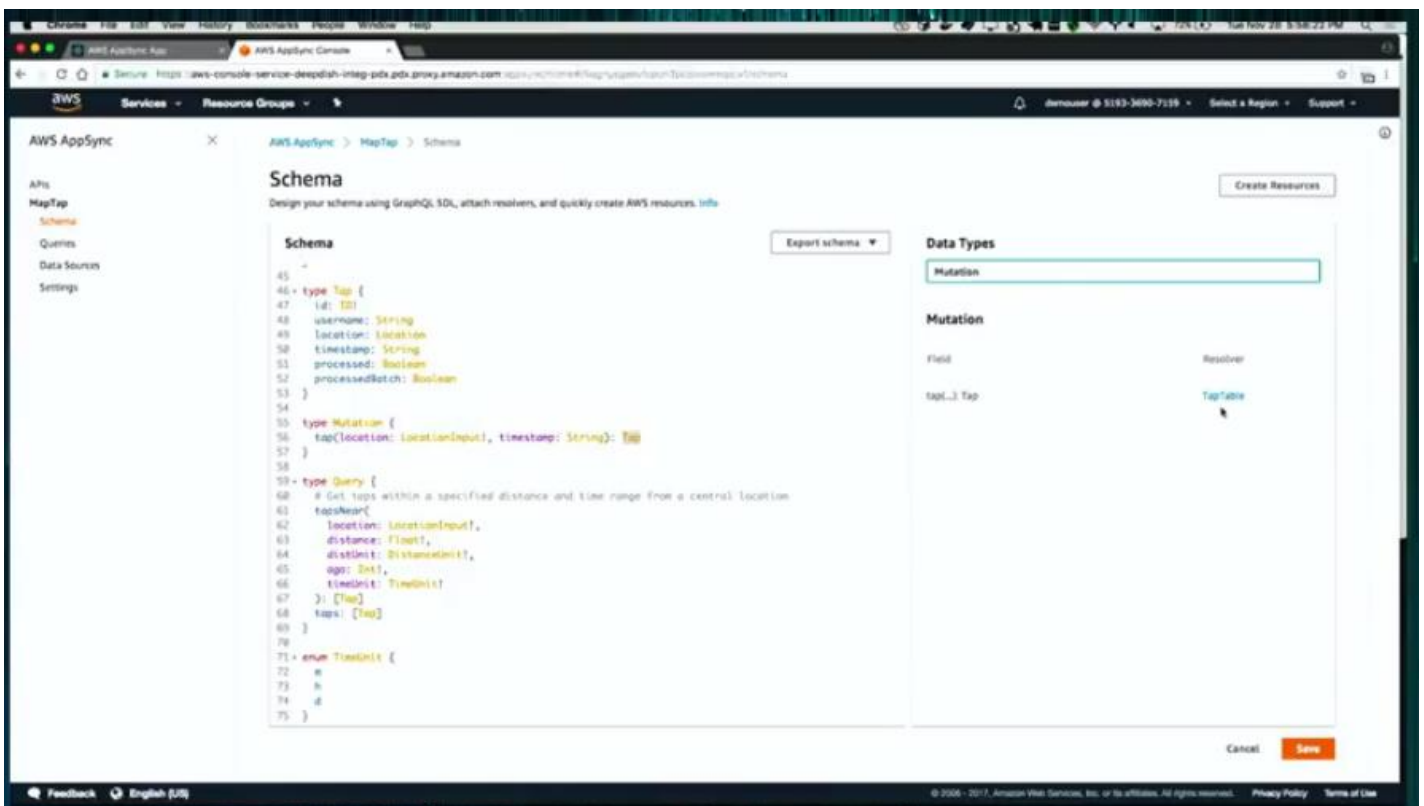


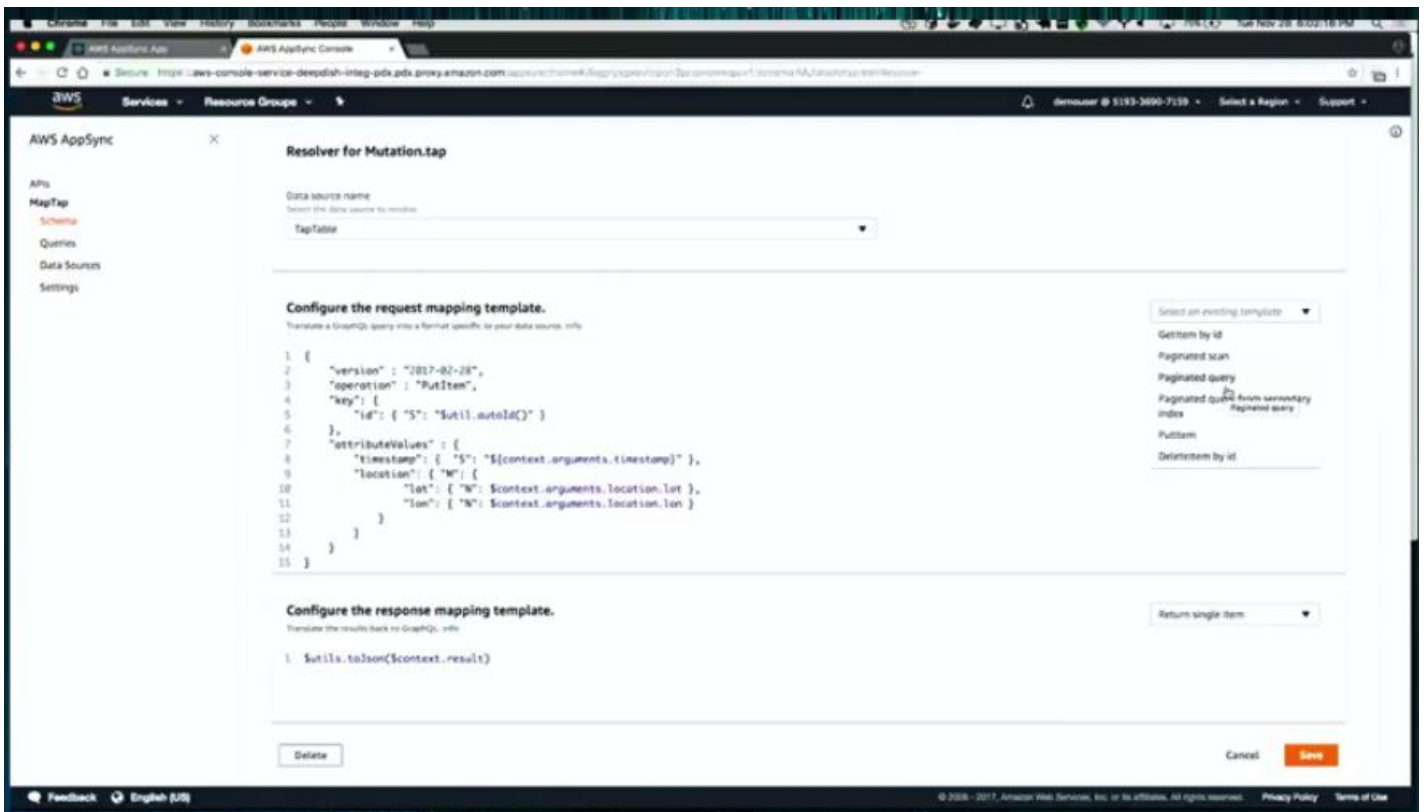
The input is a type defined in GraphQL for passing arguments into our types.



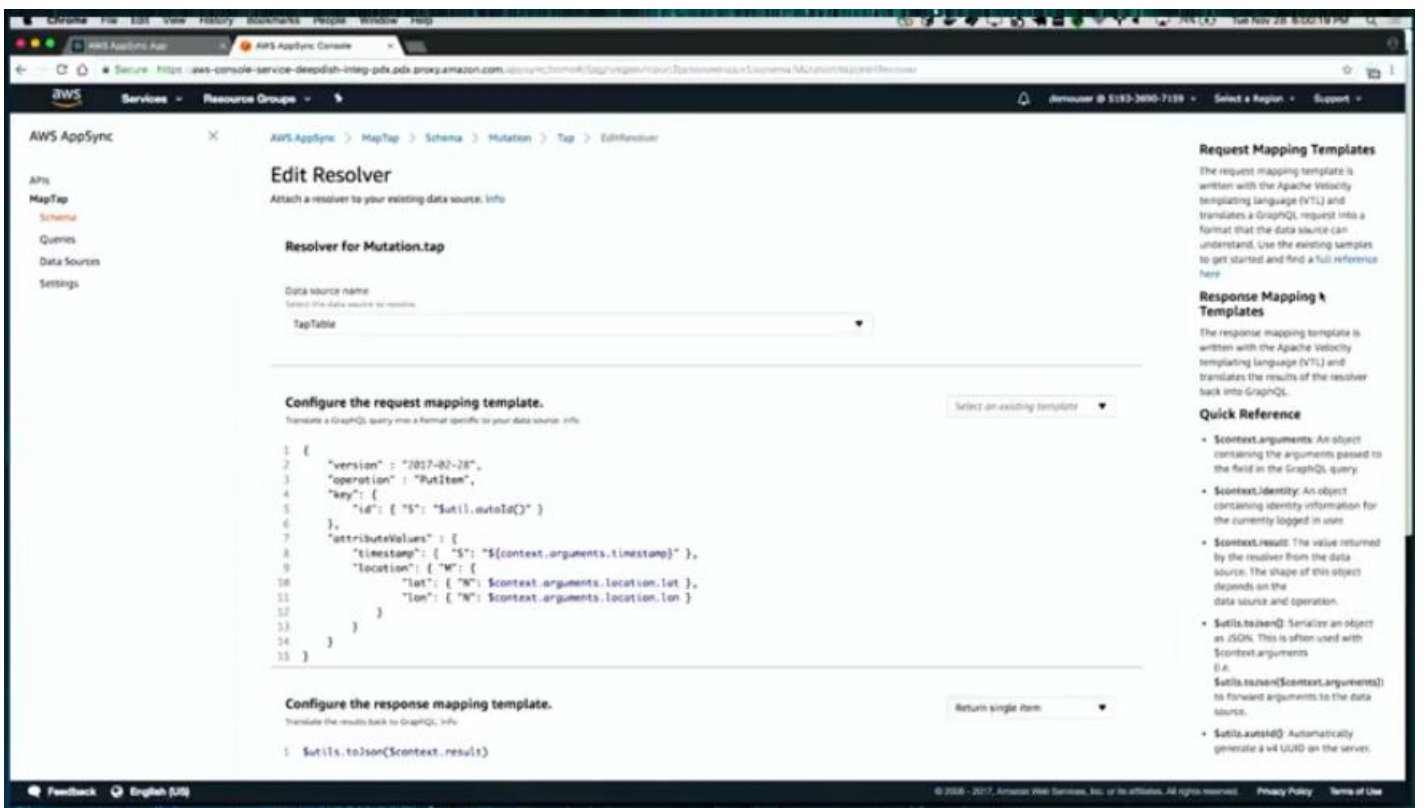


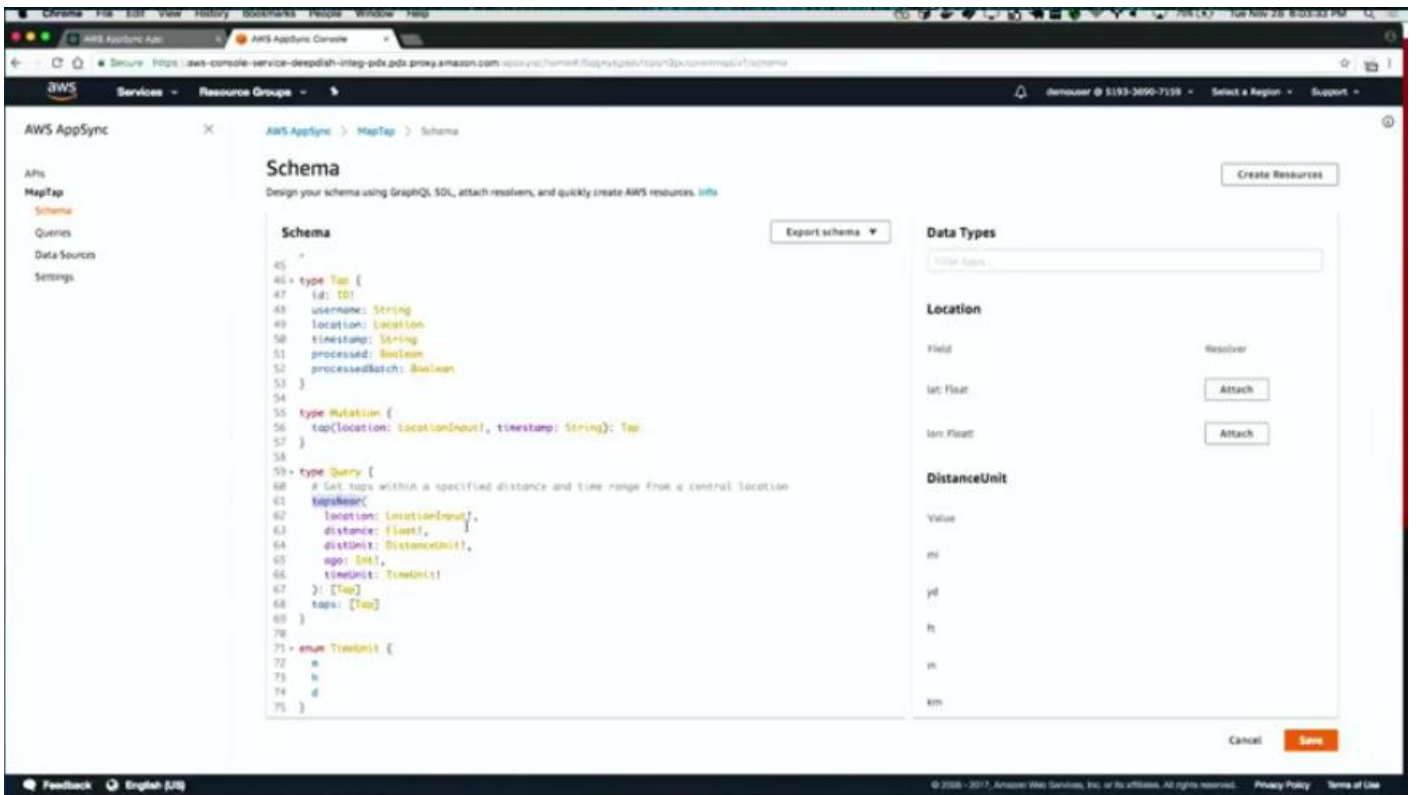
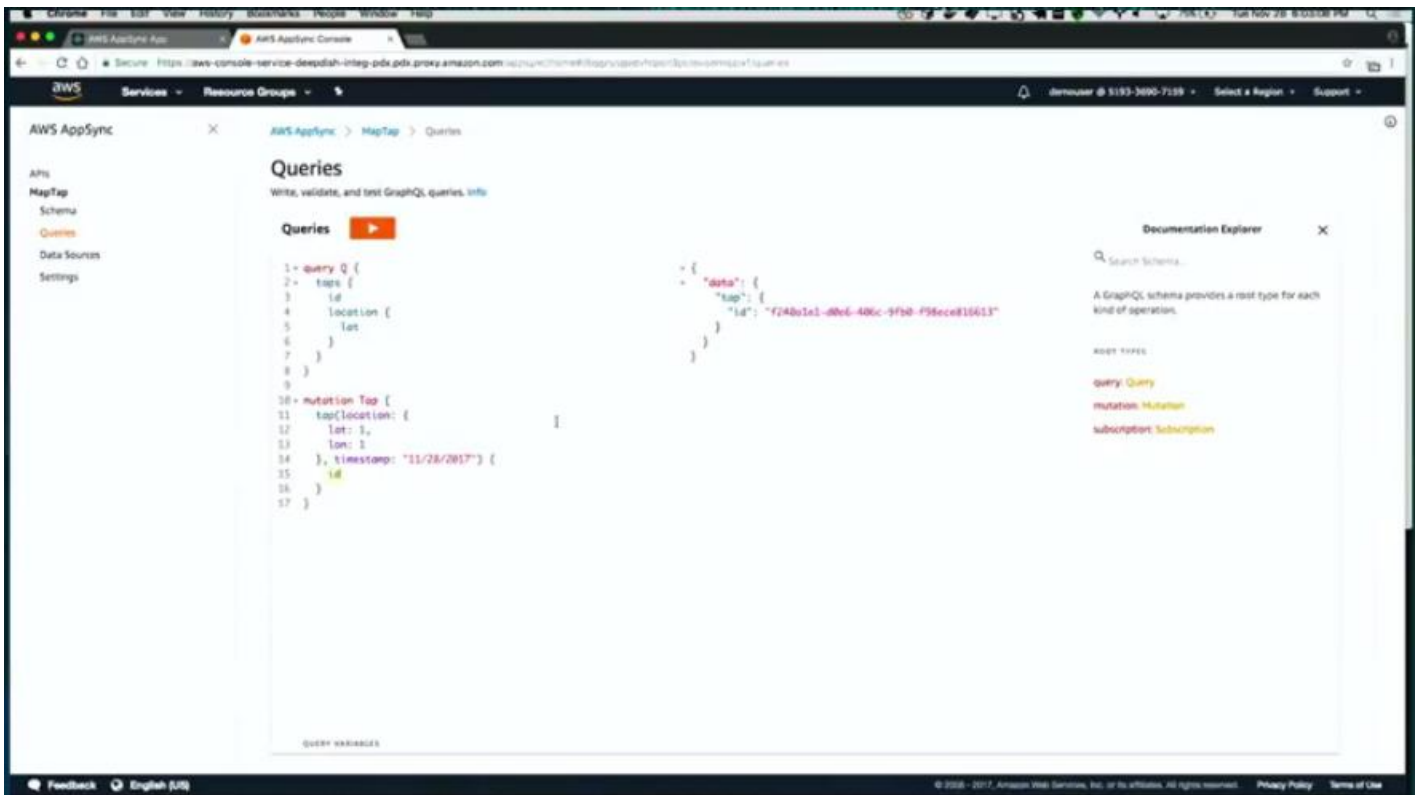
The Resolver connects our Schema to our Data Sources. The Resolver is like a function that is attached to a field and gets data to return for that field in the GraphQL execution.



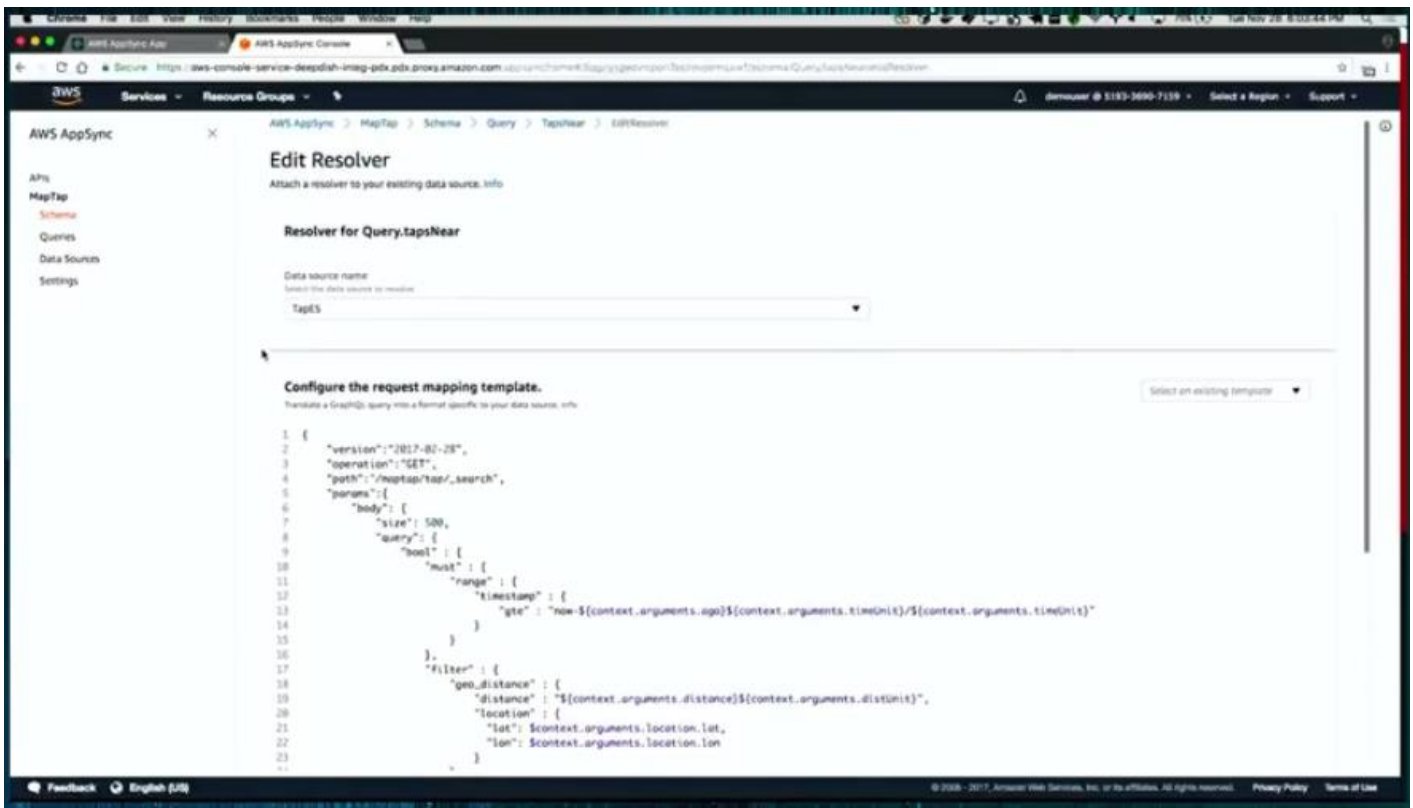


We use a templating language called Apache Velocity for the mapping template. It is a templating language that allows us to embed logic in it with some helper functions like serializing an object to JSON or autogenerating an UUID.

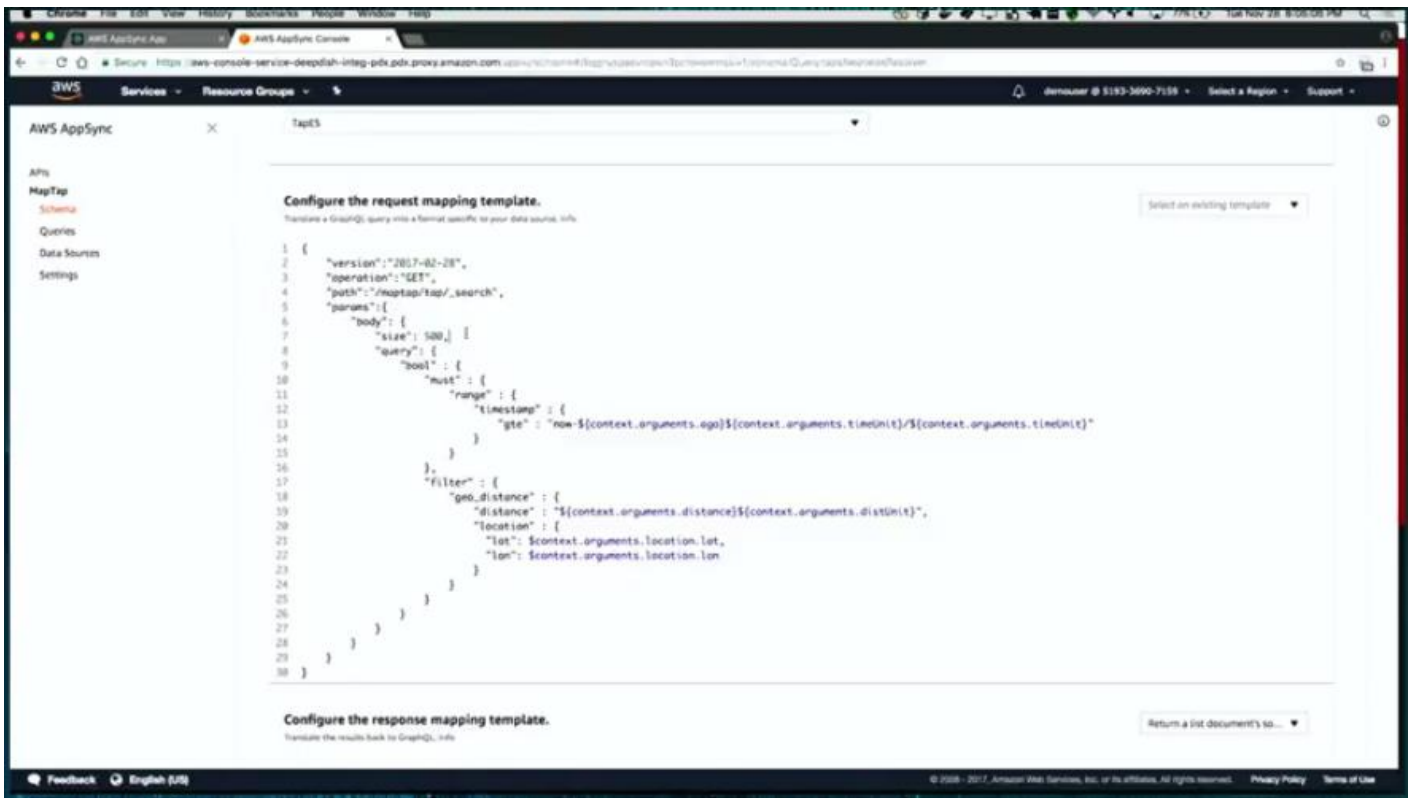




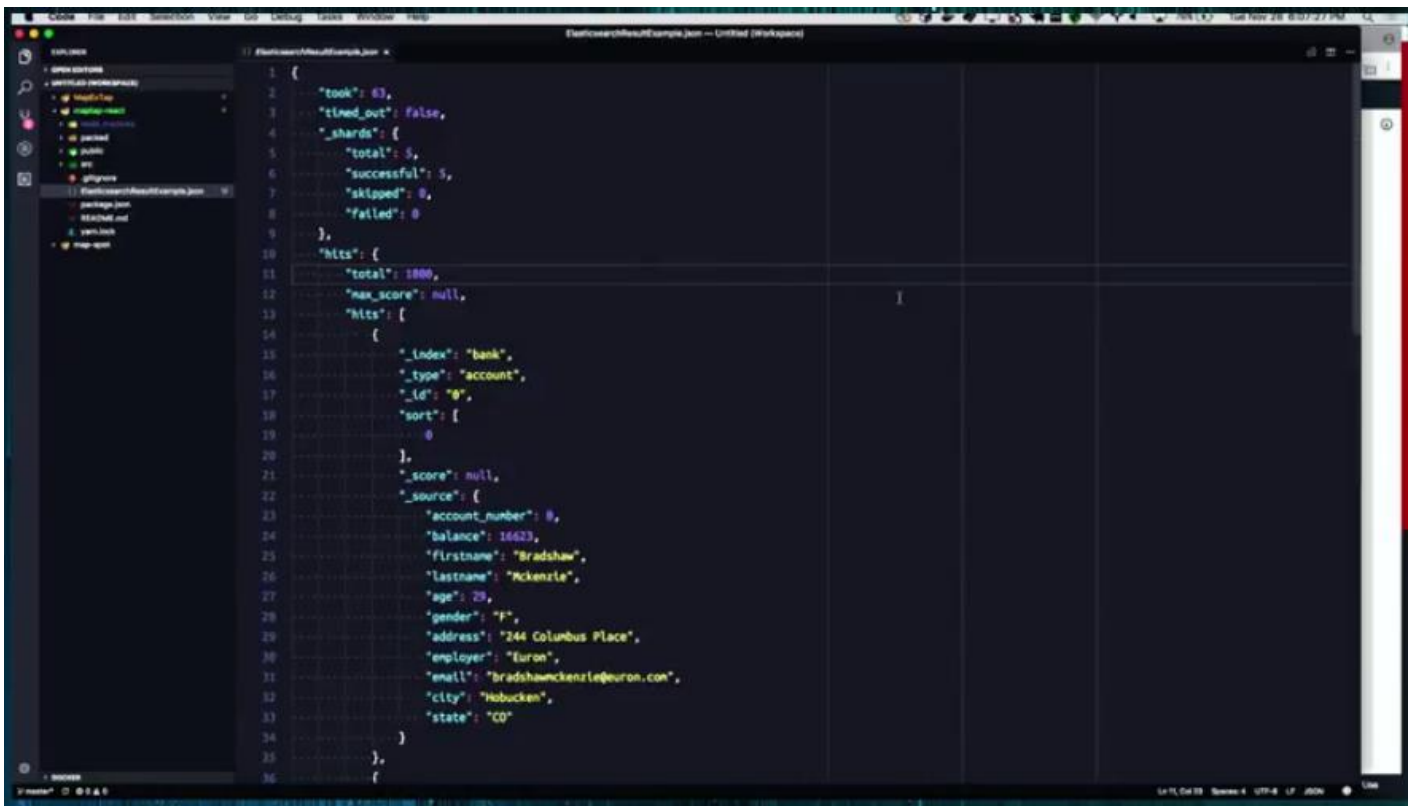
The tapsNear is our geospatial query that uses the full power of ElasticSearch to get use the taps within a define area



We have chosen the ES data source we defined TapES,

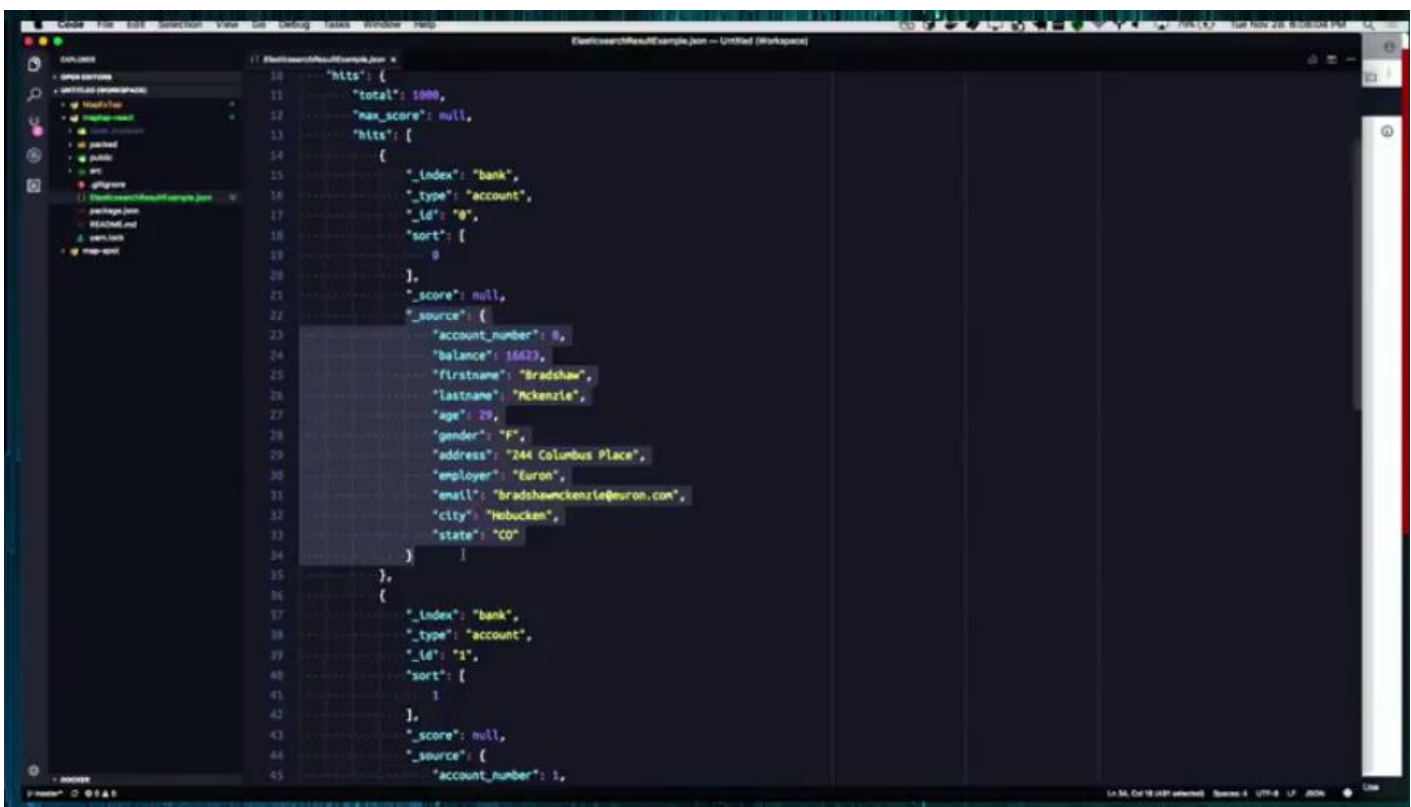


In the path field, **maptap** is the index we created in the ElasticSearch cluster, **tap** is the type in ES's index, **_search** is the URL for that particular type. Then you can define exactly what geospatial data you want with the full power of ES in the **params.body** object as above.



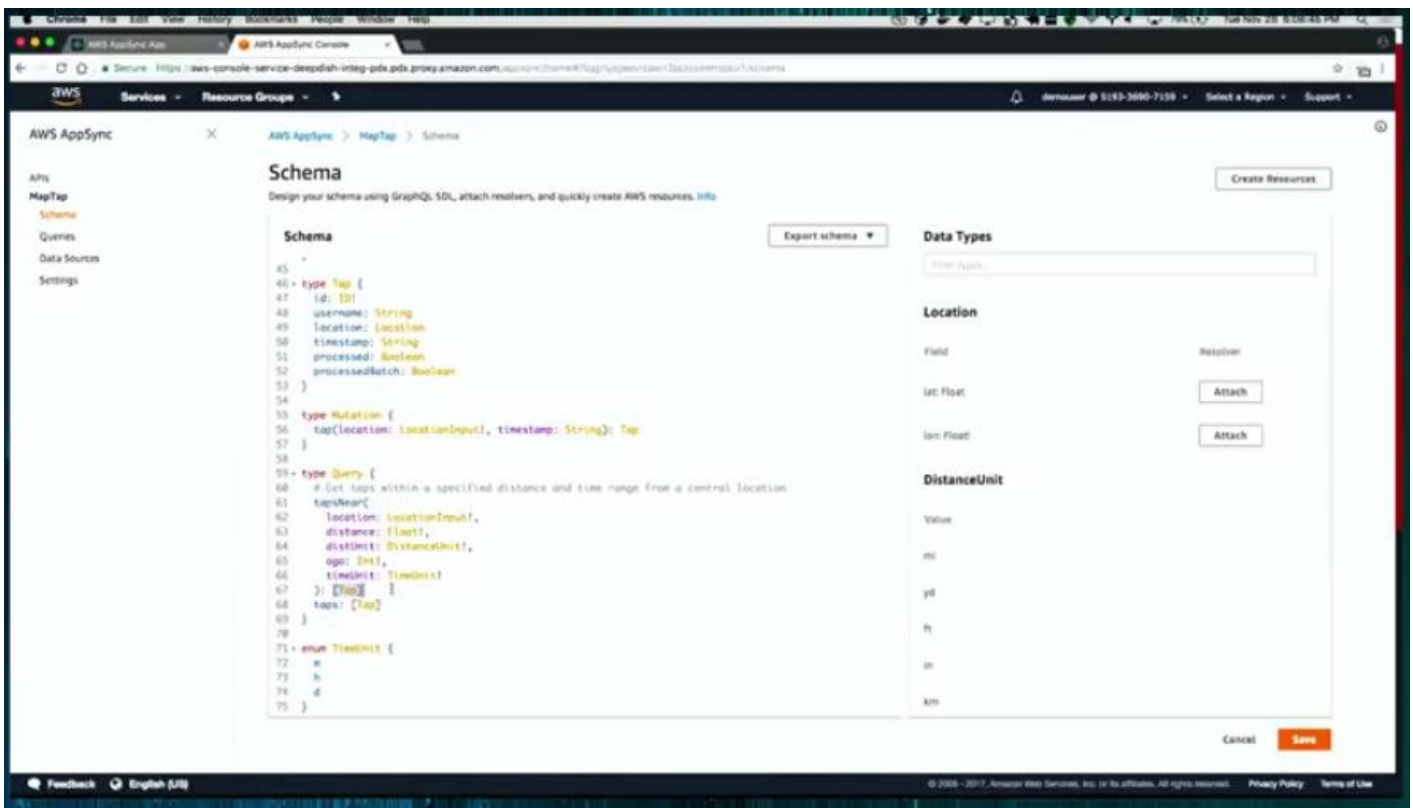
```
1 {
2   "took": 63,
3   "timed_out": false,
4   "_shards": {
5     "total": 5,
6     "successful": 5,
7     "skipped": 0,
8     "failed": 0
9   },
10  "hits": {
11    "total": 1000,
12    "max_score": null,
13    "hits": [
14      {
15        "_index": "bank",
16        "_type": "account",
17        "_id": "9",
18        "sort": [
19          0
20        ],
21        "_score": null,
22        "_source": {
23          "account_number": 0,
24          "balance": 16623,
25          "firstname": "Bradshaw",
26          "lastname": "Mckenzie",
27          "age": 29,
28          "gender": "F",
29          "address": "244 Columbus Place",
30          "employer": "Euron",
31          "email": "bradshawmckenzie@euron.com",
32          "city": "Hobucken",
33          "state": "CO"
34        }
35      }
36    ]
37  }
38 }
```

Above is an example of what Elasticsearch will return for that query. It is an object with a lot of entries that we need to filter out for what we want

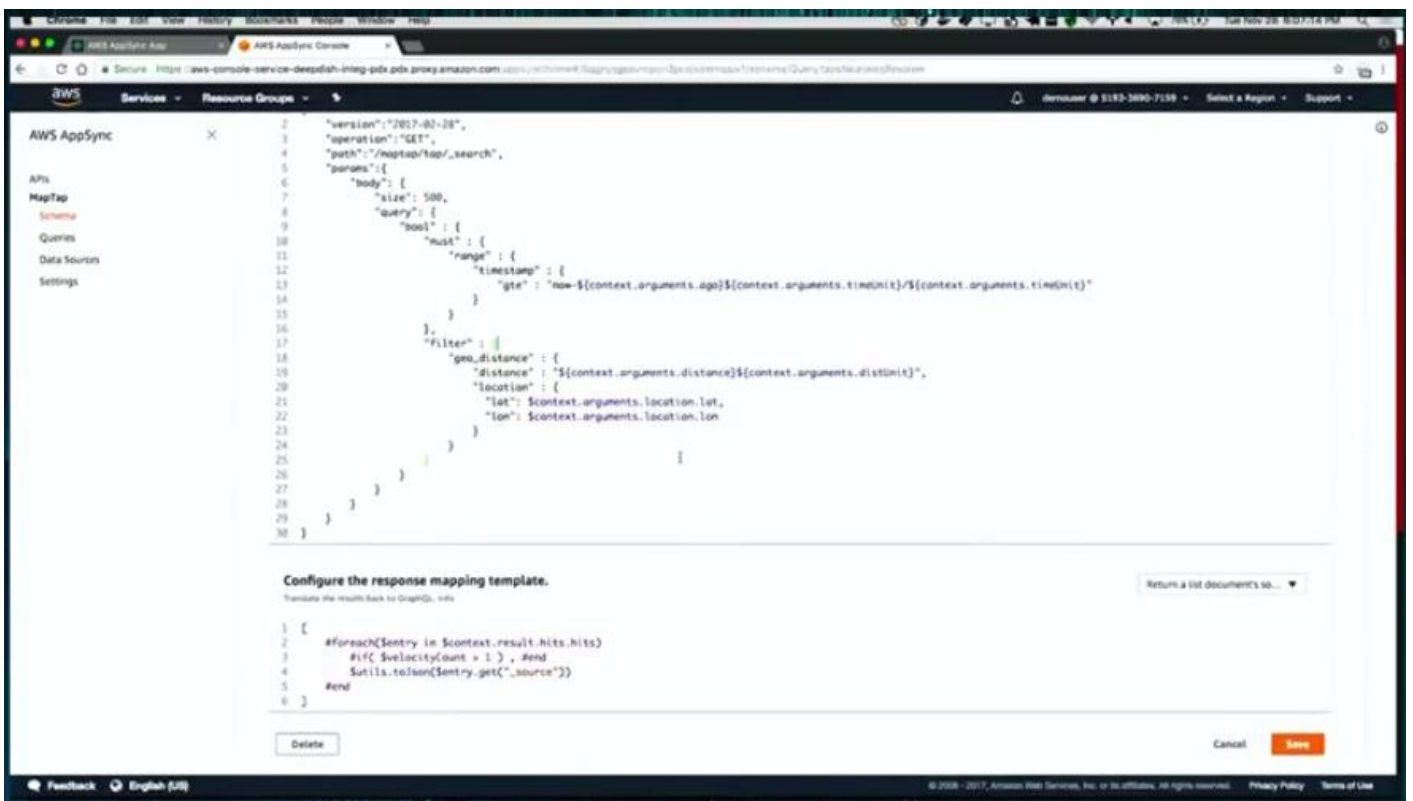


```
10  "hits": {
11    "total": 1000,
12    "max_score": null,
13    "hits": [
14      {
15        "_index": "bank",
16        "_type": "account",
17        "_id": "9",
18        "sort": [
19          0
20        ],
21        "_score": null,
22        "_source": {
23          "account_number": 0,
24          "balance": 16623,
25          "firstname": "Bradshaw",
26          "lastname": "Mckenzie",
27          "age": 29,
28          "gender": "F",
29          "address": "244 Columbus Place",
30          "employer": "Euron",
31          "email": "bradshawmckenzie@euron.com",
32          "city": "Hobucken",
33          "state": "CO"
34        }
35      },
36      {
37        "_index": "bank",
38        "_type": "account",
39        "_id": "1",
40        "sort": [
41          1
42        ],
43        "_score": null,
44        "_source": {
45          "account_number": 1,
```

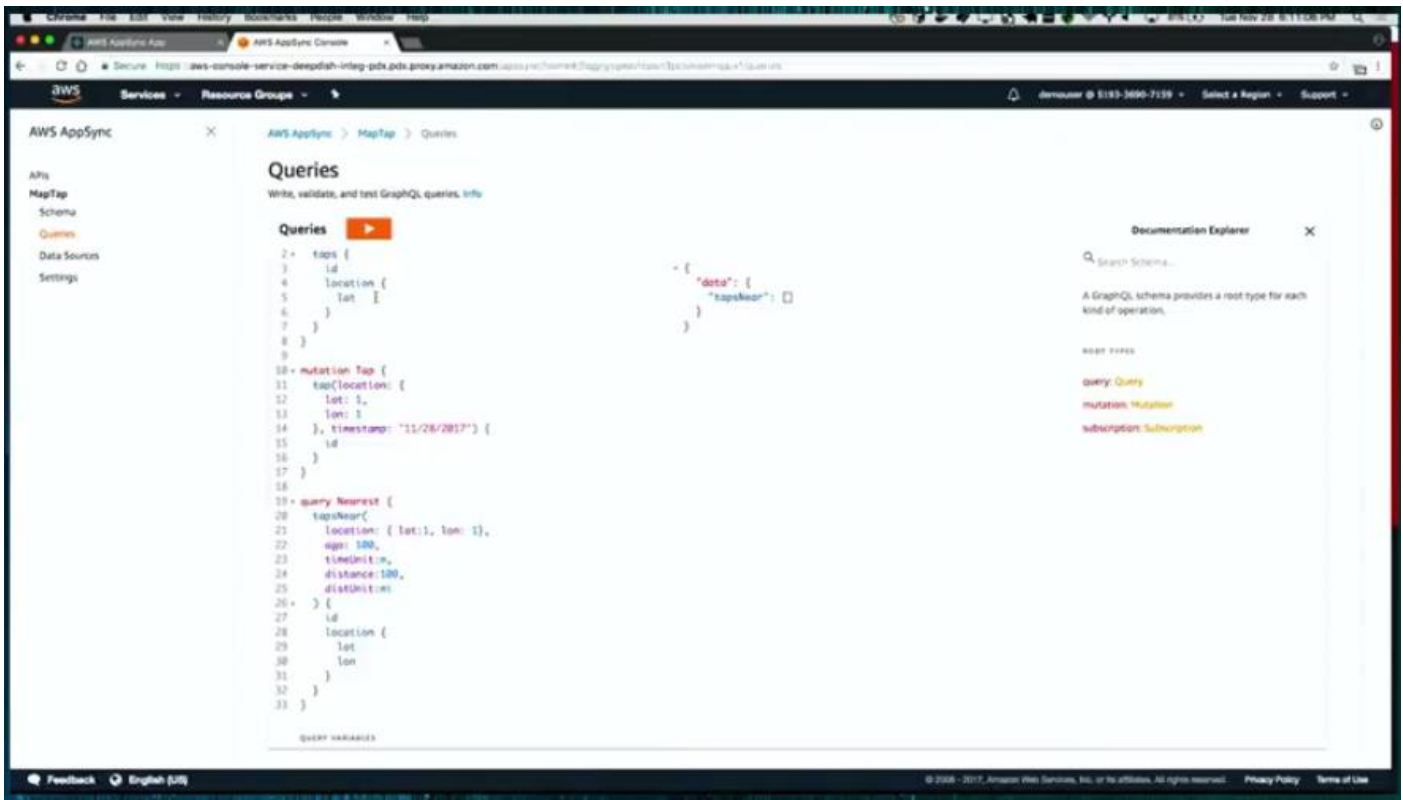
This is what we exactly want without having to call a Lambda to take it out for us.



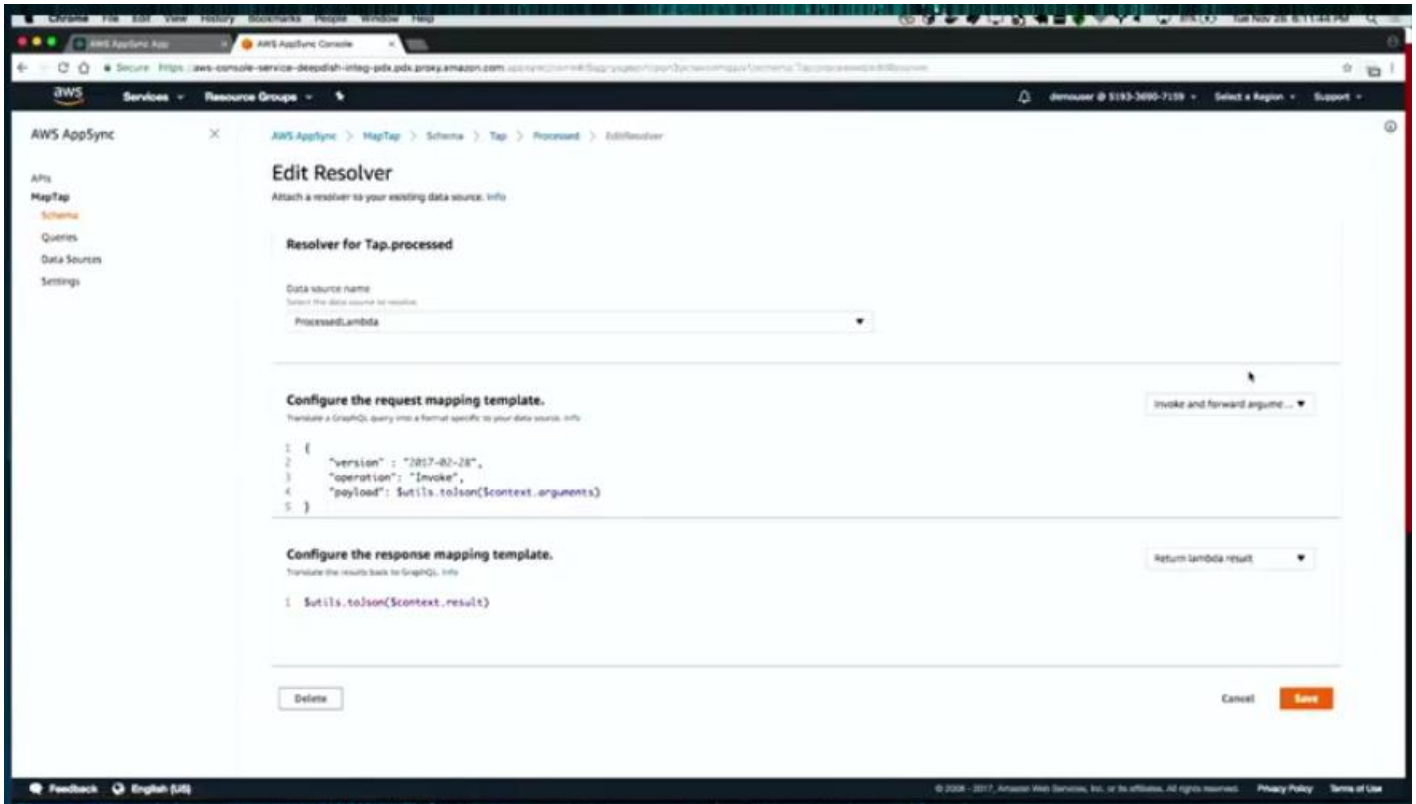
Recall from the schema definition that the `tapsNear` query expects to get back an array of taps like `[Tap]`. We need the result from ElasticSearch to look like a JSON serialized into an array of Taps `[Tap]` back from querying GraphQL.



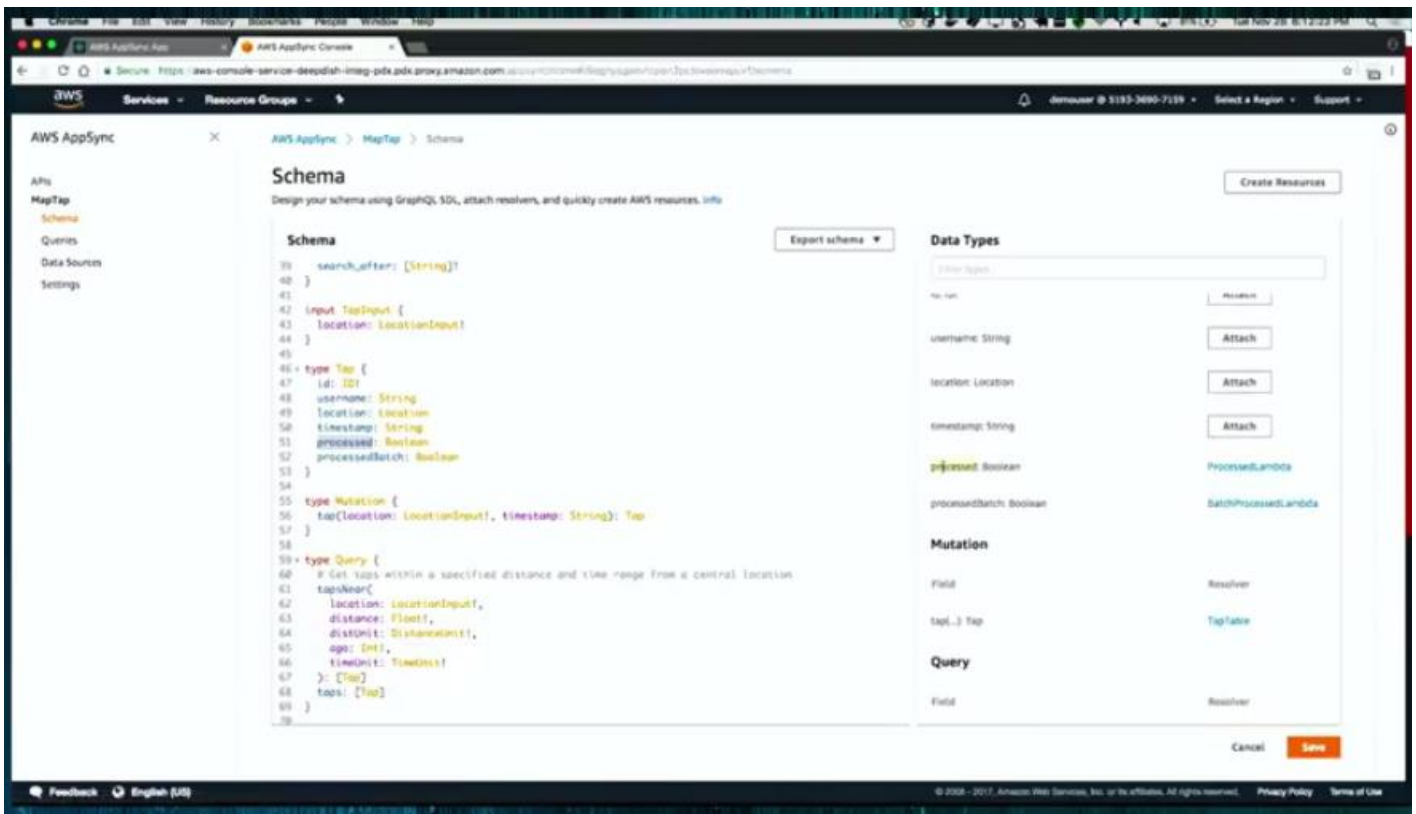
In the response, we lay out the outer bracket of the array using `[...]`. Because Velocity allows us to put logic within our templates, we then iterate through each entry in the `$context.result.hits.hits`. we then use the velocity helper called `if($velocityCount > 1)`, then just return the source of the response back.



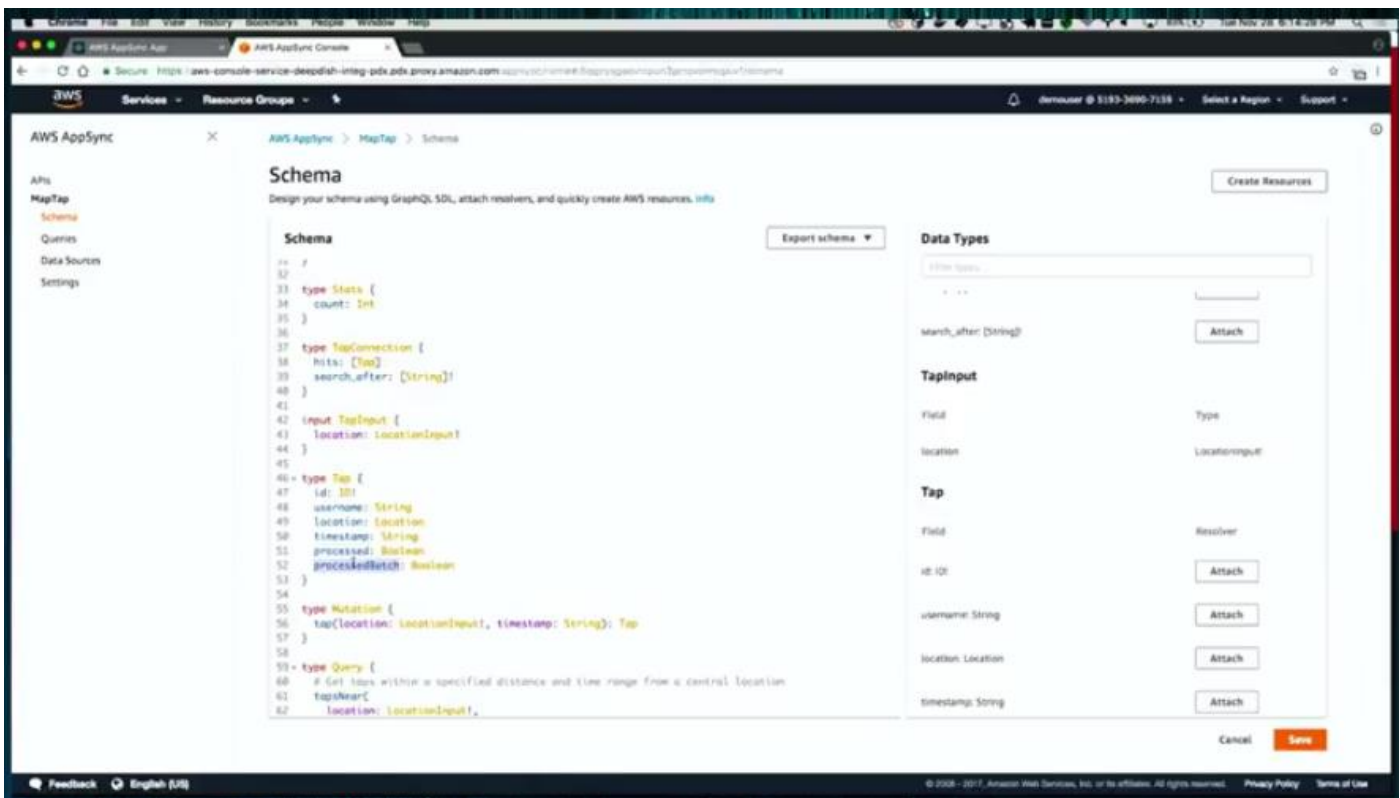
We can now create a new query called Nearest that runs tapsNear query as above to get back results



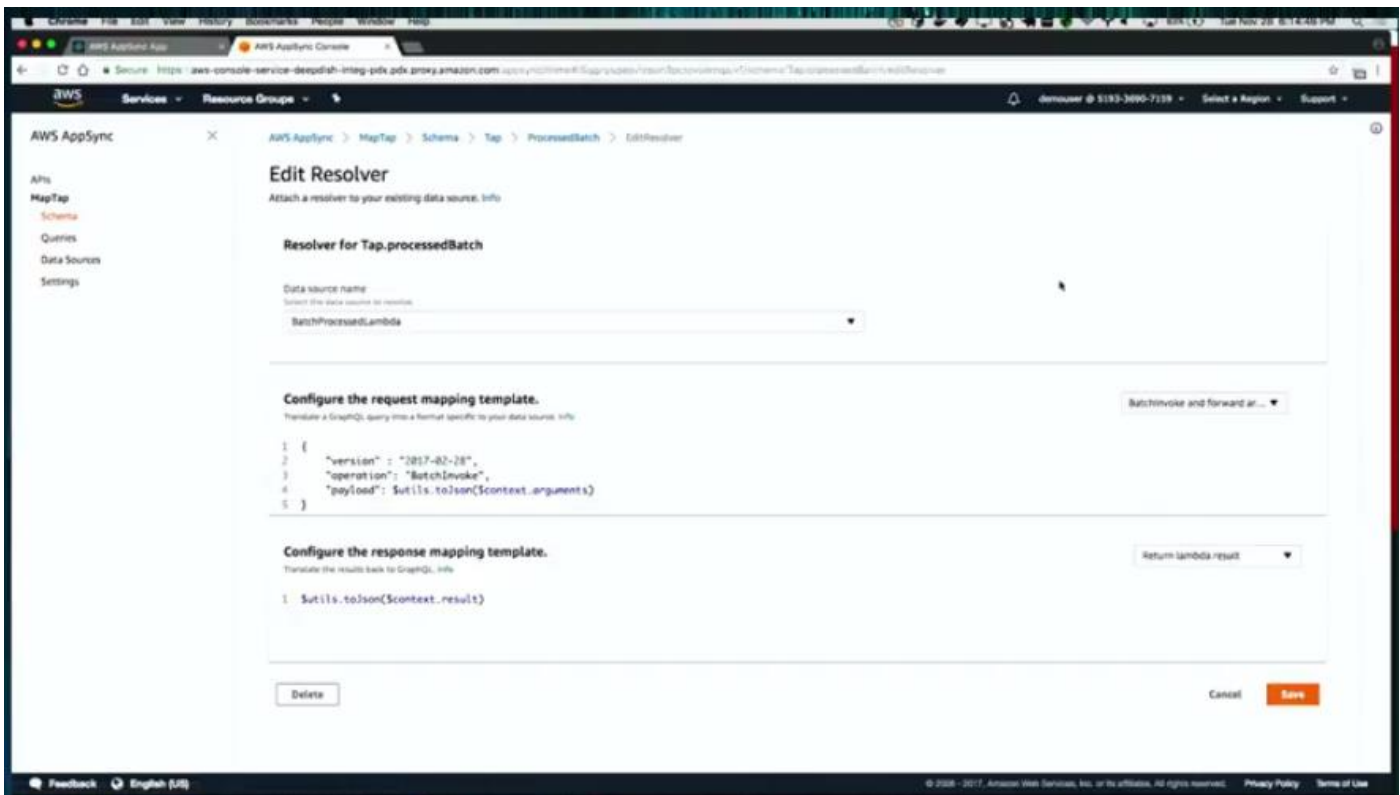
This ProcessedLambda simulates a call out to a third party like a credit agency.



This processed field in our schema is a Boolean that returns the result of the Lambda calling the 3rd party endpoint. But this becomes an issue when we have a lot of taps to call the ProcessedLambda calls for in a N+1 problem scenario, this is why we need a **BatchedProcessedLambda** for per request problem called the **data loader pattern**.



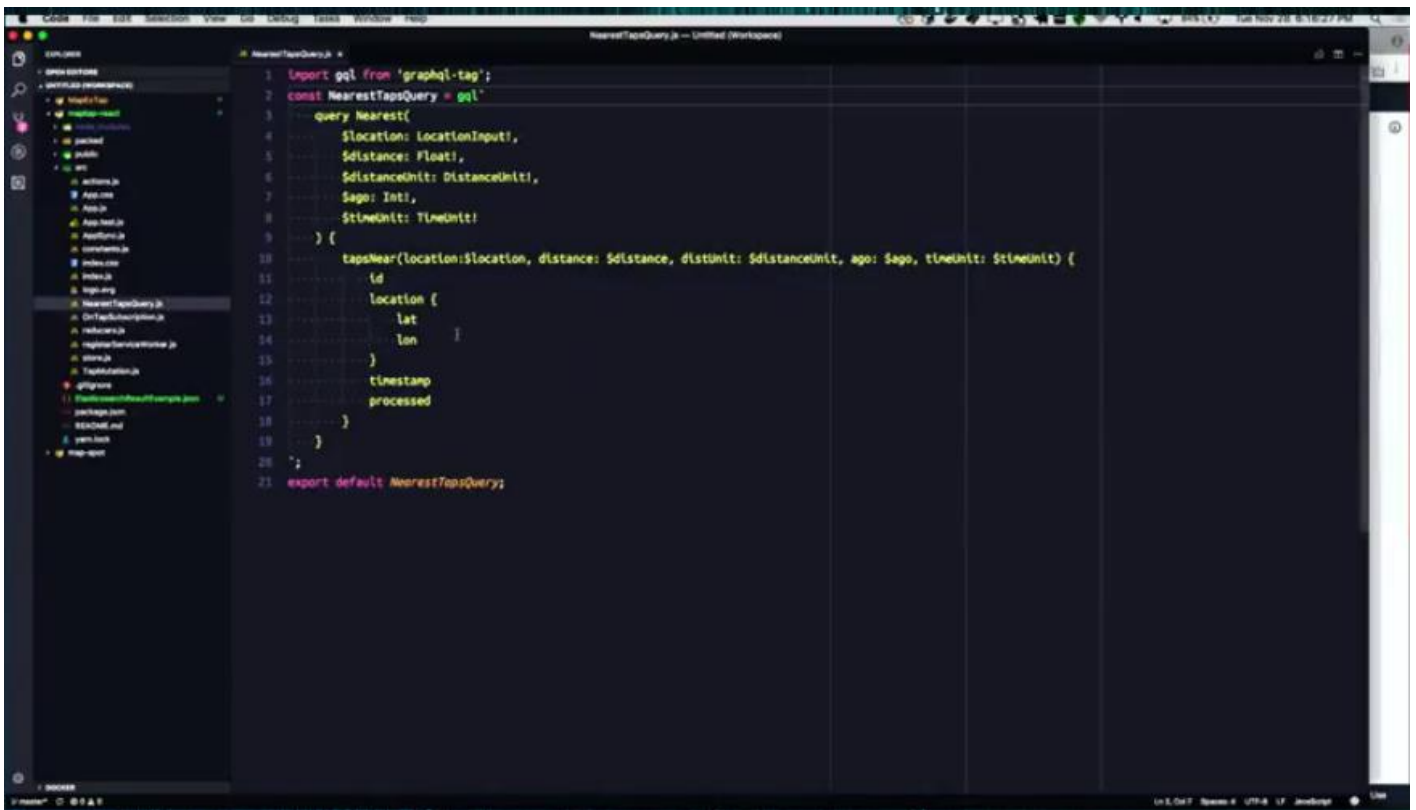
The **processedBatch** Boolean can be set to true to have this N+1 lambda calls reduced to 1 single lambda call for us.



This is the batch resolver that uses the **BatchInvoke() operation** instead of the earlier **invoke() operation** in the non-batch case. The resulting payload will be in the exact same other that the calls will be made in.

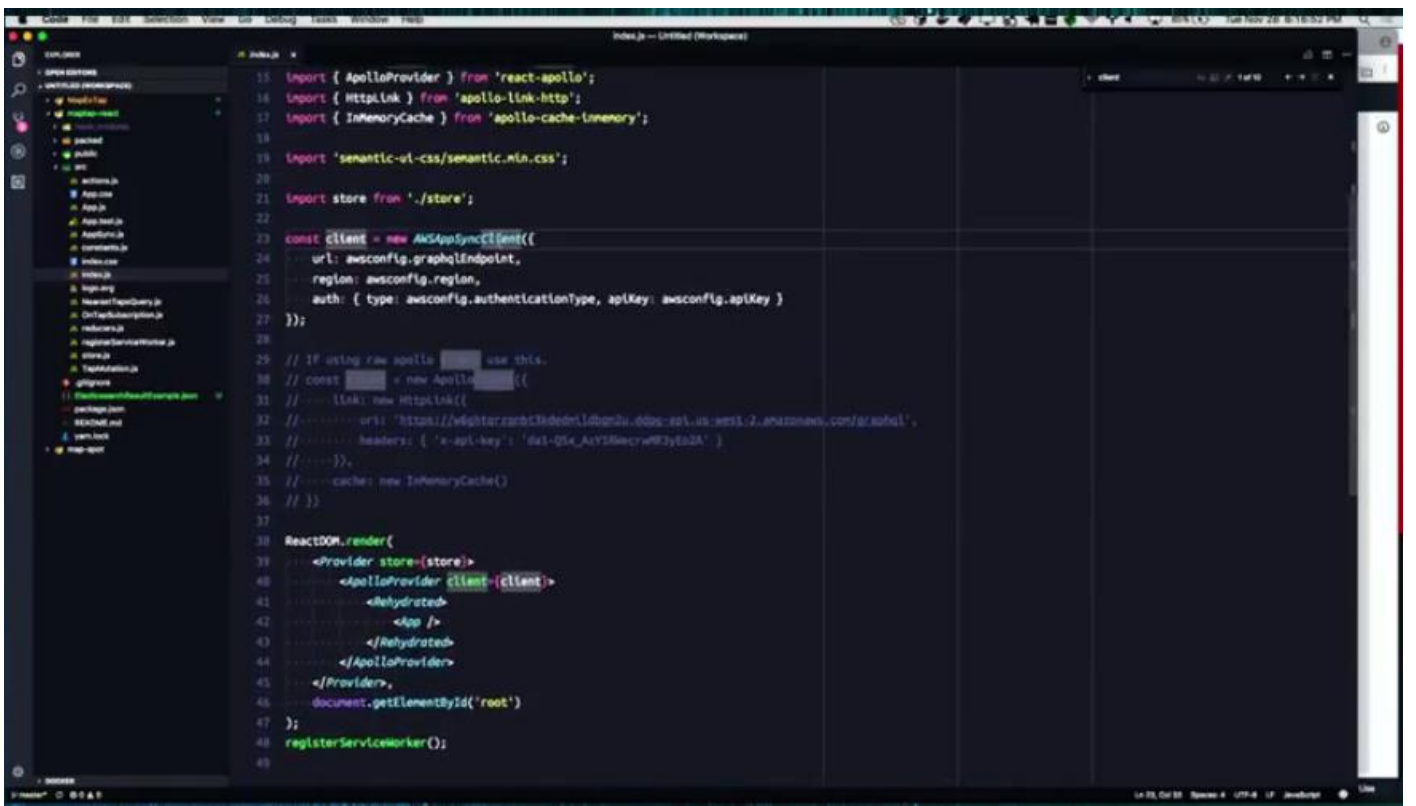


This is a standard React app within one file,



```
1 import gql from 'graphql-tag';
2 const NearestTapsQuery = gql`
3   query Nearest(
4     $location: LocationInput!,
5     $distance: Float!,
6     $distanceUnit: DistanceUnit!,
7     $sapo: Int!,
8     $timeUnit: TimeUnit!
9   ) {
10     tapsNear(location: $location, distance: $distance, distUnit: $distanceUnit, ago: $sapo, timeUnit: $timeUnit) {
11       id
12       location {
13         lat
14         lon
15       }
16       timestamp
17       processed
18     }
19   }
20 `;
21 export default NearestTapsQuery;
```

we have our GraphQL defined in this JS file which are imported and used in the app



```
15 import { ApolloProvider } from 'react-apollo';
16 import { HttpLink } from 'apollo-link-http';
17 import { InMemoryCache } from 'apollo-cache-inmemory';
18
19 import 'semantic-ui-css/semantic.min.css';
20
21 import store from './store';
22
23 const client = new ApolloClient({
24   url: awsconfig.graphqlEndpoint,
25   region: awsconfig.region,
26   auth: { type: awsconfig.authenticationType, apiKey: awsconfig.apiKey }
27 });
28
29 // If using raw apollo use this.
30 const client = new ApolloClient({
31   // link: new HttpLink({
32     // url: 'https://w8gtrznbt3dedvldbnlu.dop.east-us-west-2.amazonaws.com/graphql',
33     // headers: { 'x-api-key': 'dal-Q5x_AuY56acwWQ2y5a2A' }
34   // }),
35   // cache: new InMemoryCache()
36 });
37
38 ReactDOM.render(
39   <Provider store={store}>
40     <ApolloProvider client={client}>
41       <Rehydrated>
42         <App />
43       </Rehydrated>
44     </ApolloProvider>
45   </Provider>,
46   document.getElementById('root')
47 );
48 registerServiceWorker();
```

You set up your AppSyncClient that is built on top of the OSS Apollo client in this file

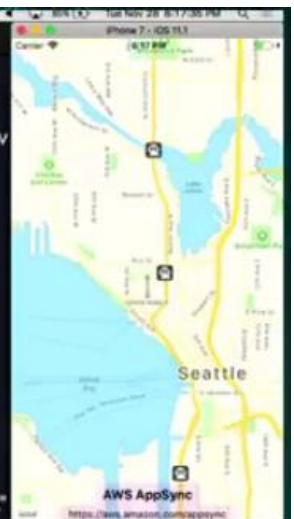

```
Compiled with warnings.

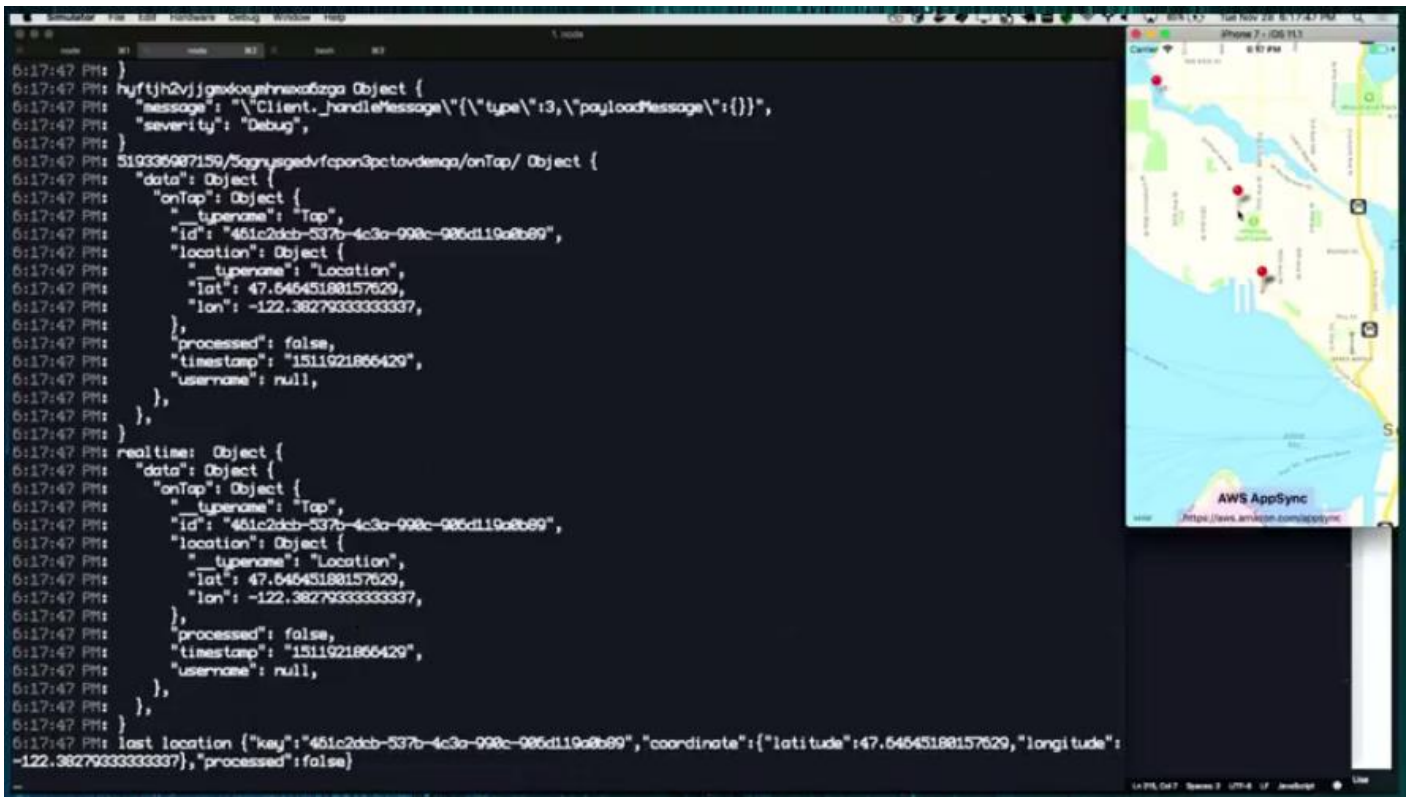
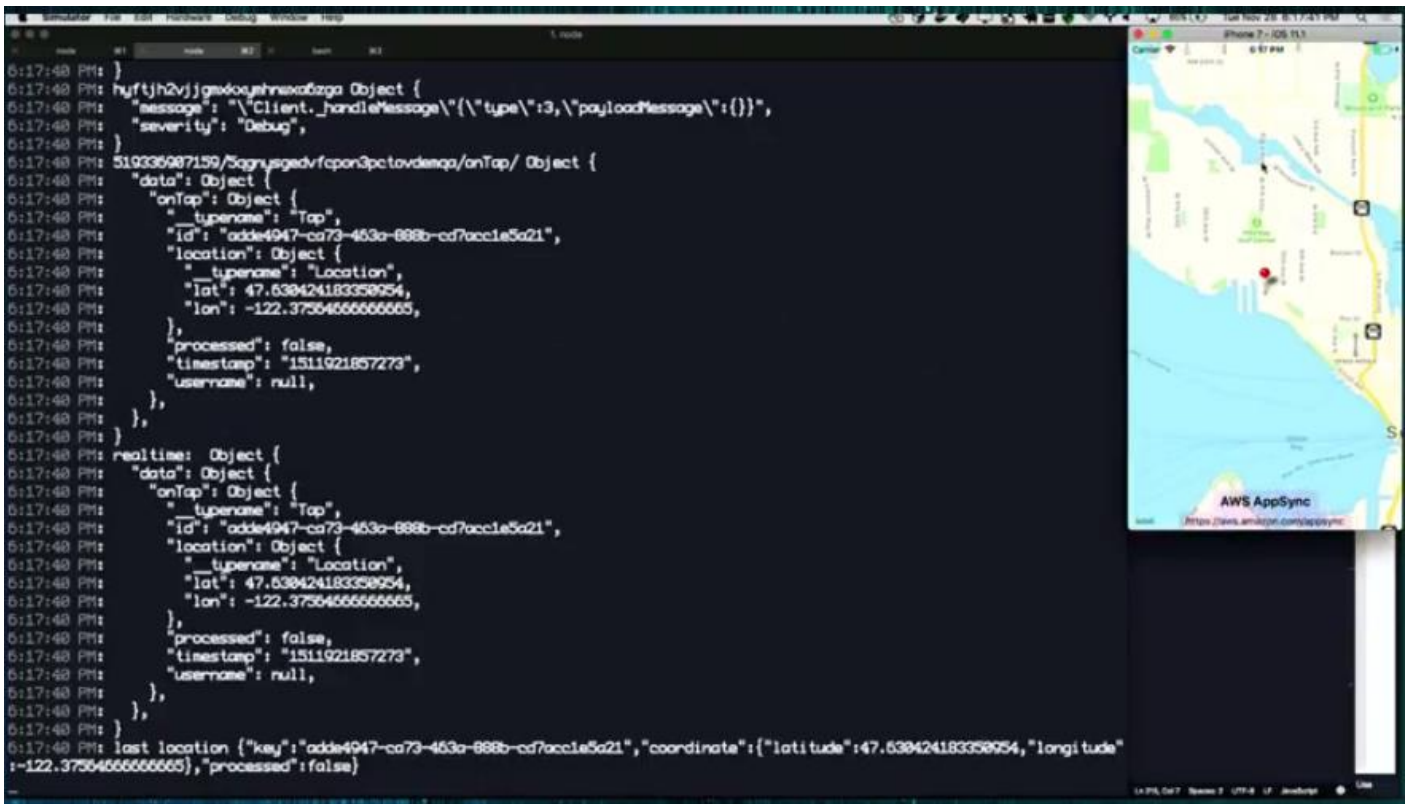
./src/index.js
Line 14: 'ApolloClient' is defined but never used  no-unused-vars
Line 16: 'HttpLink' is defined but never used           no-unused-vars
Line 17: 'InMemoryCache' is defined but never used      no-unused-vars

./src/App.js
Line 4: 'gql' is defined but never used             no-unused-vars
Line 8: 'Select' is defined but never used             no-unused-vars
Line 14: 'logo' is defined but never used              no-unused-vars

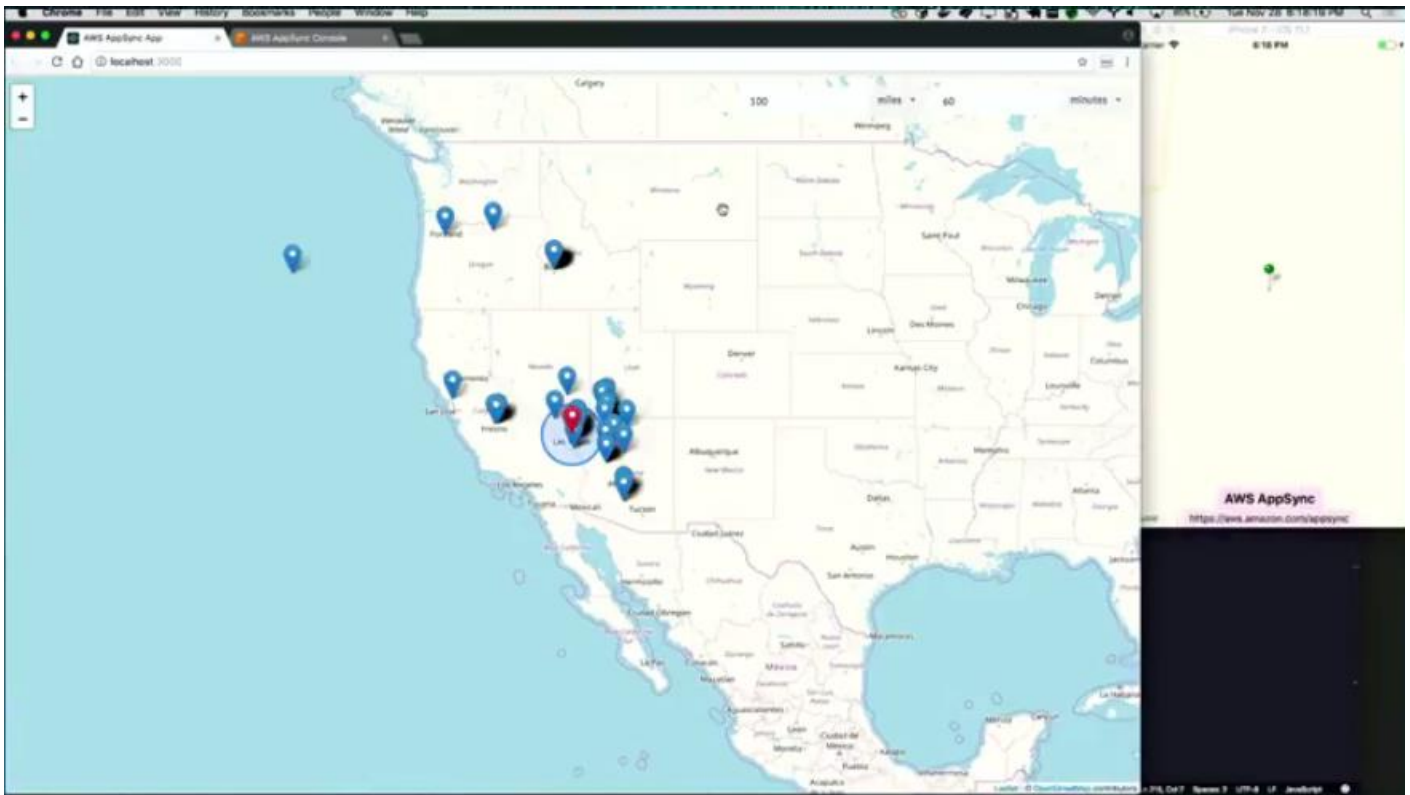
Search for the keywords to learn more about each warning.
To ignore, add // eslint-disable-next-line to the line before.
```

```
:true,\"cleanSession\":true>false\",
6:17:33 PM:   "severity": "Debug",
6:17:33 PM: }
6:17:34 PM: hyftjh2vjvgwdoayhnxwazga Object {
6:17:34 PM:   "message": "\\Client._socket_send\\(\"\\type\\":1,\"useSSL\\":true,\"mqttVersion\\":3,\"keepAliveInterval\\":60,\"mqttv
6:17:34 PM:   "severity": "Debug",
6:17:34 PM: }
6:17:34 PM: hyftjh2vjvgwdoayhnxwazga Object {
6:17:34 PM:   "message": "\\Client._on_socket_message\\(\"{}\",
6:17:34 PM:   "severity": "Debug",
6:17:34 PM: }
6:17:34 PM: hyftjh2vjvgwdoayhnxwazga Object {
6:17:34 PM:   "message": "\\Client._handleMessage\\(\"\\type\\":2,\"returnCode\\":0)\",
6:17:34 PM:   "severity": "Debug",
6:17:34 PM: }
6:17:34 PM: Doing setup for 1 topics Array [
6:17:34 PM:   "519336987159/5agnysgedvfcpn3pctovdesqa/onTap/",
6:17:34 PM: ]
6:17:34 PM: hyftjh2vjvgwdoayhnxwazga Object {
6:17:34 PM:   "message": "\\Client.subscribe\\(\"519336987159/5agnysgedvfcpn3pctovdesqa/onTap/\"{}\",
6:17:34 PM:   "severity": "Debug",
6:17:34 PM: }
6:17:34 PM: hyftjh2vjvgwdoayhnxwazga Object {
6:17:34 PM:   "message": "\\Client._socket_send\\(\"\\type\\":8,\"topics\\":[\"519336987159/5agnysgedvfcpn3pctovdesqa/onTap/\"],\"
6:17:34 PM:   "severity": "Debug",
6:17:34 PM: }
6:17:34 PM: hyftjh2vjvgwdoayhnxwazga Object {
6:17:34 PM:   "message": "\\Client._on_socket_message\\(\"{}\",
6:17:34 PM:   "severity": "Debug",
6:17:34 PM: }
6:17:34 PM: hyftjh2vjvgwdoayhnxwazga Object {
6:17:34 PM:   "message": "\\Client._handleMessage\\(\"\\type\\":9,\"messageIdentifier\\":1,\"returnCode\\":{\\\"0\\":0}\\\"}\",
6:17:34 PM:   "severity": "Debug",
6:17:34 PM: }
6:17:34 PM: All topics subscribed Array [
6:17:34 PM:   "519336987159/5agnysgedvfcpn3pctovdesqa/onTap/",
6:17:34 PM: ]
```

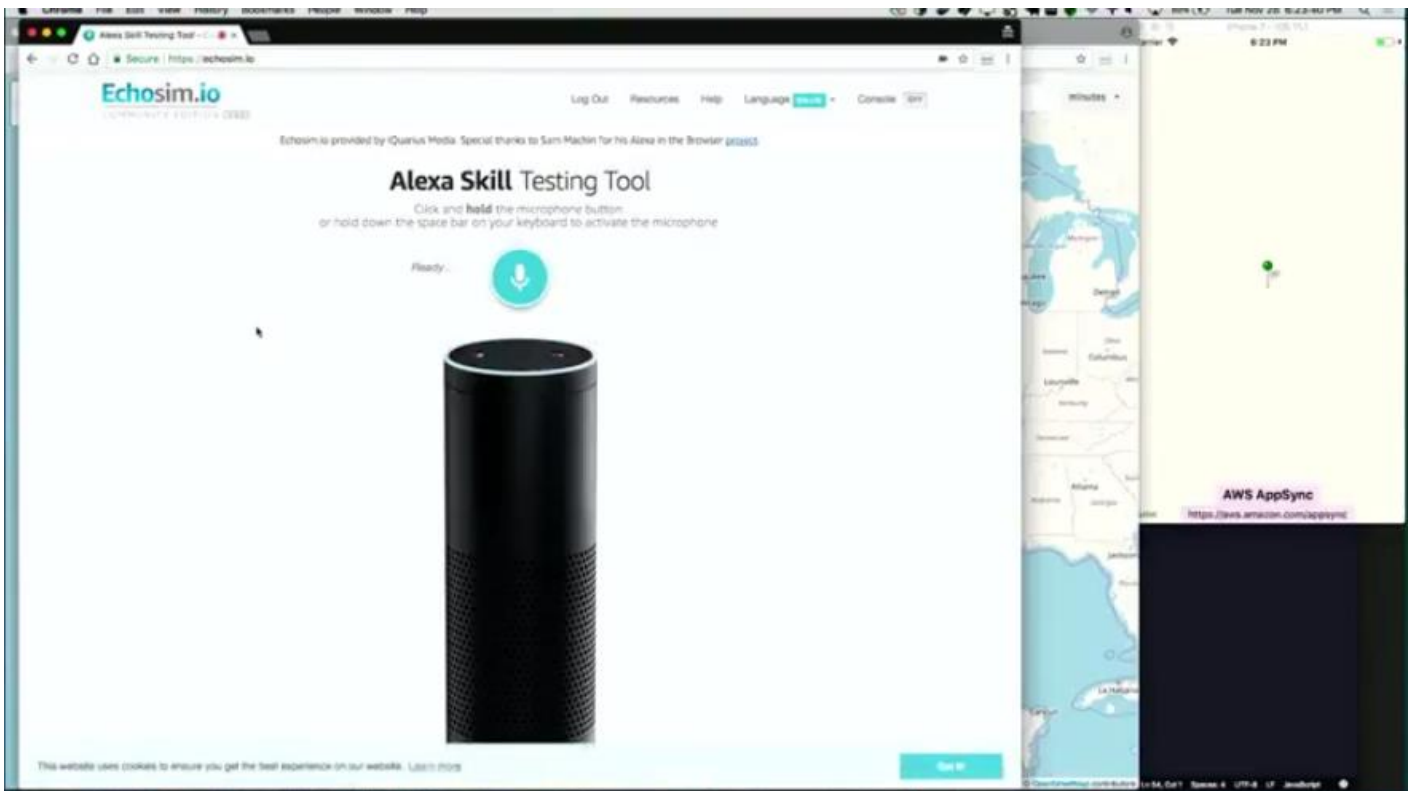




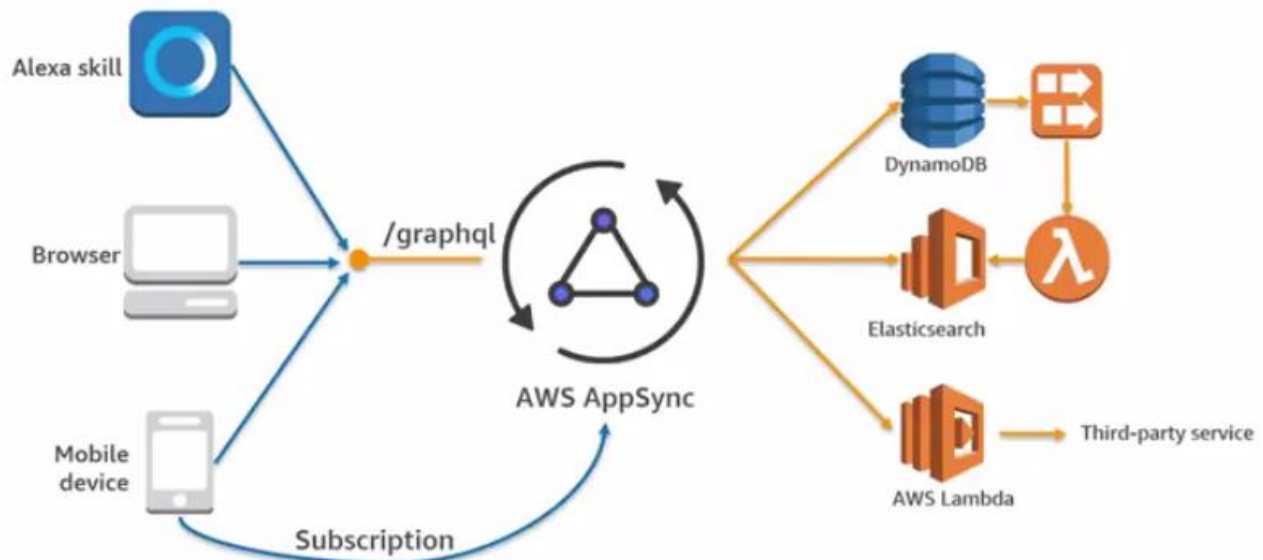
We then see the tap on screen and also see the responses on the server



We also see that the subscriptions are working when we click on any subscribed device



MapTap with AWS AppSync



AWS re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



AWS AppSync benefits

- Clients receive the data they ask for. Nothing more, nothing less.
- Get many resources from many data sources with a single request.
- Self-documenting APIs with Introspection.
- A strong type system.
- More powerful developer tools.
- Simpler API evolution.

AWS re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



MBL 402 – Data-driven Apps with GraphQL and AWS AppSync

**AWS
re:Invent**

THANK YOU!

Sign up at

<https://aws.amazon.com/appsync>

**AWS
re:Invent**

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

