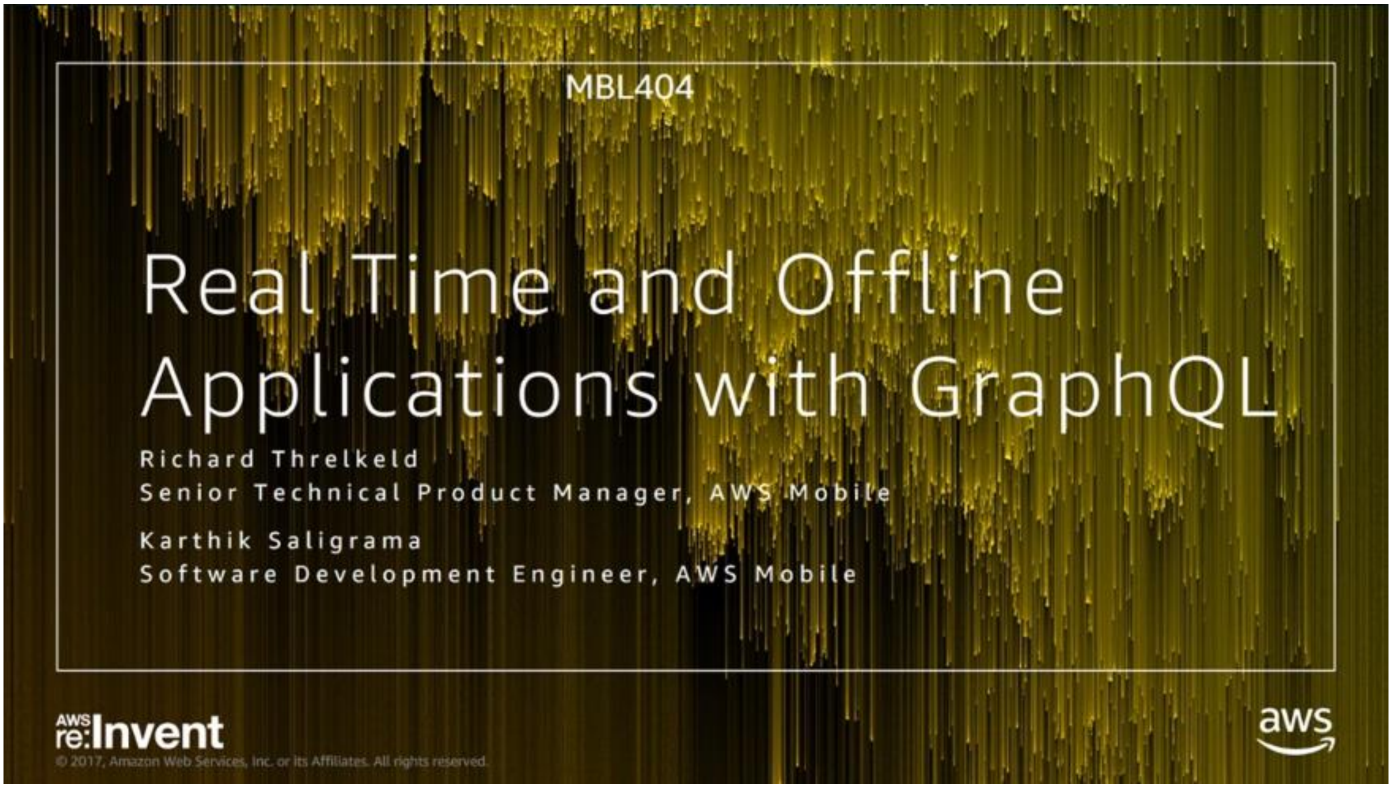


MBL404

Real Time and Offline Applications with GraphQL

Richard Threlkeld
Senior Technical Product Manager, AWS Mobile

Karthik Saligrama
Software Development Engineer, AWS Mobile

AWS re:Invent

aws

All application developers today need to be concerned with offline access, realtime communications and efficient data fetching. These techniques are no longer optional for great user experiences yet are difficult to engineer and scale from scratch. In this session you'll get a deep dive on using AWS AppSync to enable your applications for offline access, including optimistic updates on lossy connections, with just a few lines of code. You'll learn how application data synchronization takes place with the cloud, how you can control the process, programming interfaces for native applications such as iOS and JavaScript based applications across the web, React Native, and Ionic. Additionally you'll see how using GraphQL enables your application to efficiently leverage the network for queries and mutations while still having a scalable and fast connection for realtime updates when using subscriptions to data changes.

Offline/real time use cases

Users expect data immediately
- Banking alerts
- News stories
- Multi-player games
- Chat applications
- Shared whiteboards
- AR/VR experiences
- Document collaboration

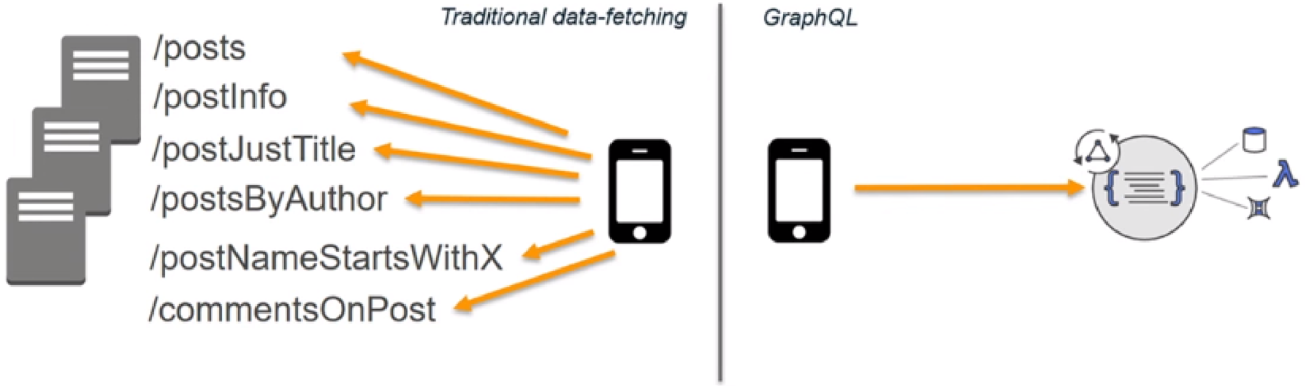
Users expect data availability offline
- Financial transactions
- News articles
- Games
- Messaging (pending chat)
- Document collaboration

What is GraphQL?

Open, declarative data-fetching specification

!= Graph database

Use NoSQL, Relational, HTTP, etc.



How does GraphQL work?

```
type Query {
    getTodos: [Todo]
}

type Todo {
    id: ID!
    name: String
    description: String
    priority: Int
    duedate: String
}
```

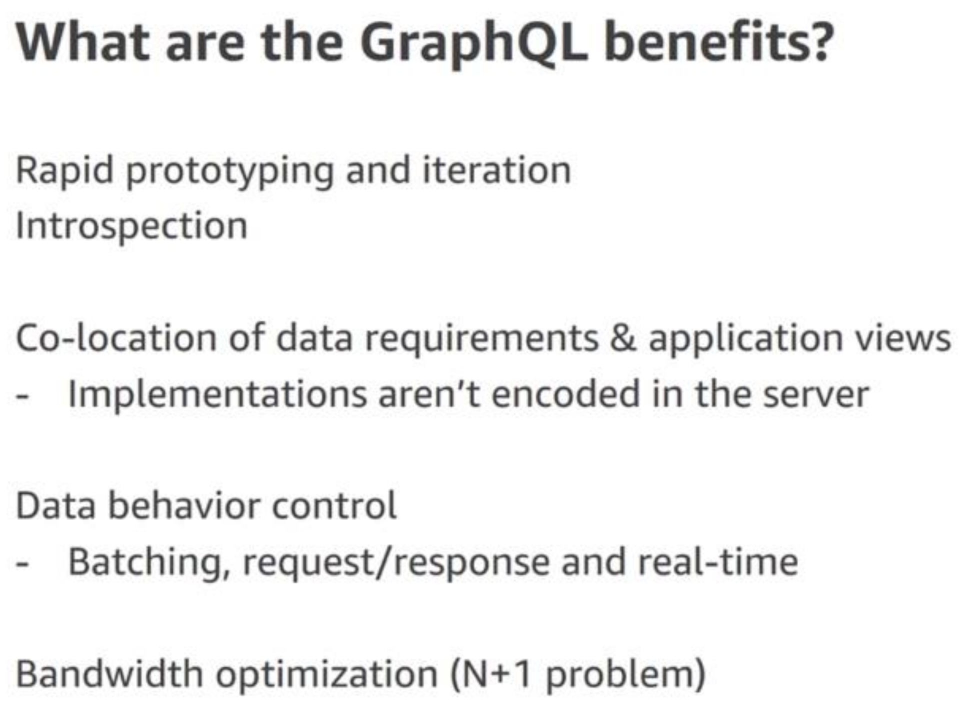
```
query {
  getTodos {
    id
    name
    priority
  }
}
```

```
{
    "id": "1",
   "name": "Get Milk",
    “priority": "1"
},
{
    "id": “2",
   "name": “Go to gym",
    “priority": “5"
},…
```

Model data with application schema

Client requests what it needs

Only that data is returned



What are the GraphQL benefits?

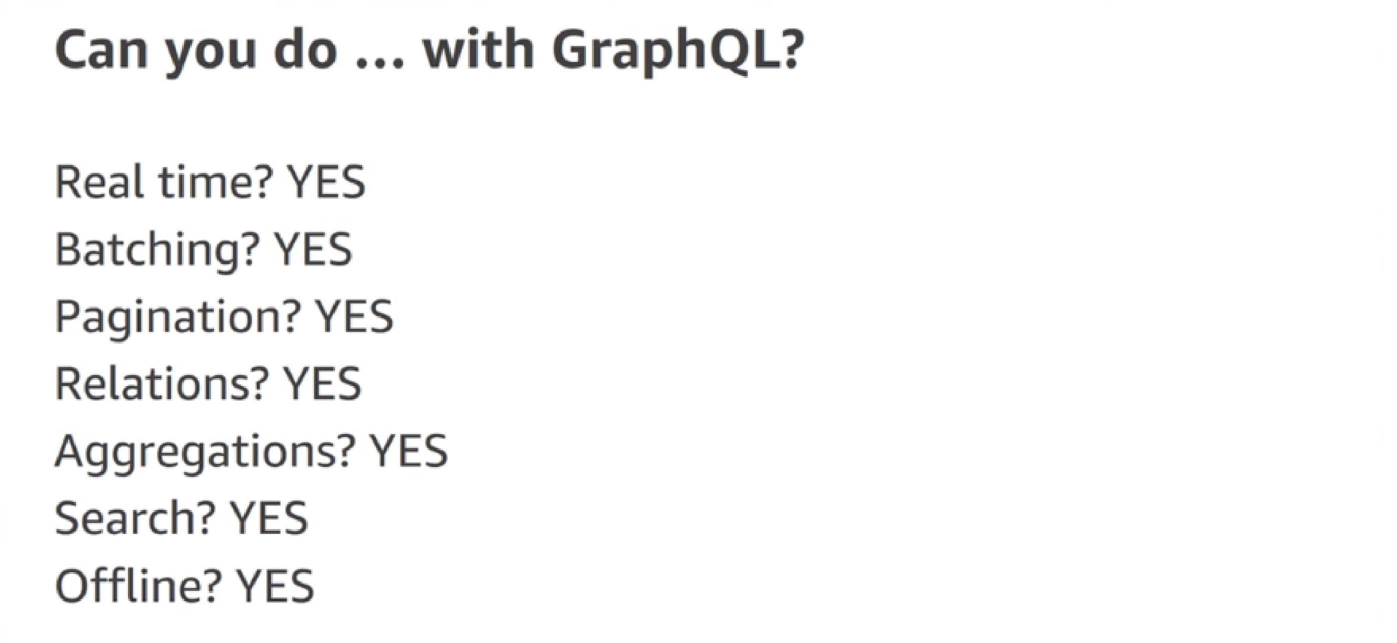
Rapid prototyping and iteration
Introspection

Co-location of data requirements & application views
- Implementations aren’t encoded in the server

Data behavior control
- Batching, request/response and real-time

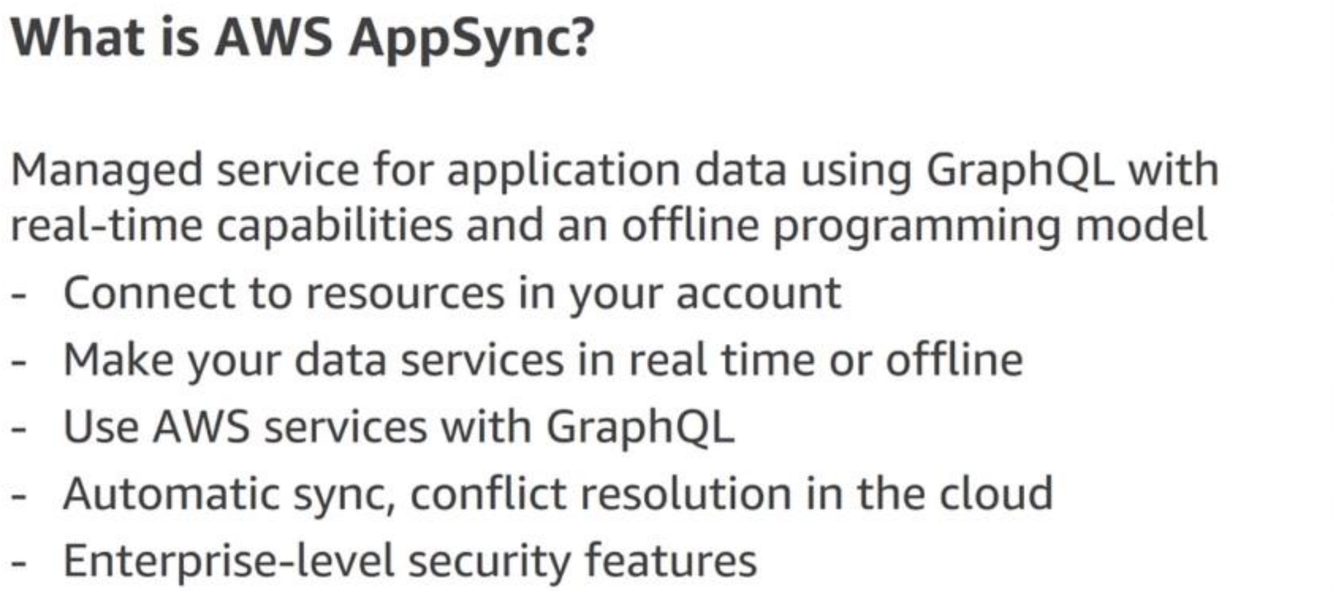
Bandwidth optimization (N+1 problem)

You can use the normal request/response call for normal data and use GraphQL subscriptions for the data that needs to be real time from your DynamoDB and Redshift databases or other data sources



Can you do … with GraphQL?

Real time? YES
Batching? YES
Pagination? YES
Relations? YES
Aggregations? YES
Search? YES
Offline? YES



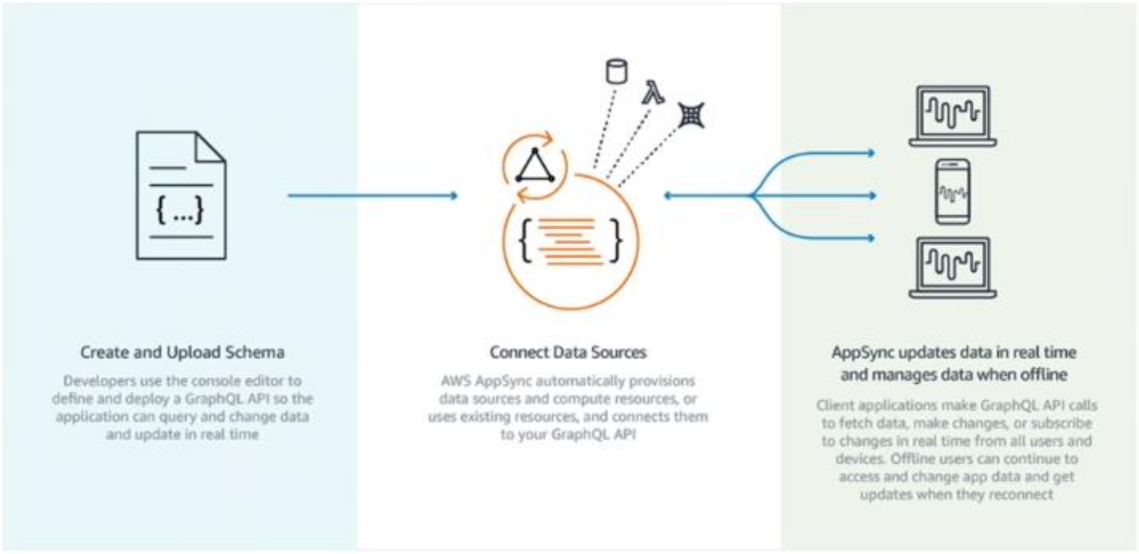
What is AWS AppSync?

Managed service for application data using GraphQL with real-time capabilities and an offline programming model

- Connect to resources in your account
- Make your data services in real time or offline
- Use AWS services with GraphQL
- Automatic sync, conflict resolution in the cloud
- Enterprise-level security features

You can make any of your data source (DynamoDB, Elasticsearch, Lambda) real time by hooking them up to GQL subscriptions or respond to mutations. You are provided some API Keys when you create a GQL endpoint, you can also use AWS IAM, Cognito user pools. You can use regular IAM policies with Roles and assign this to GQL types, who can actually invoke them, you can use Cognito user pools so that anybody using the user pools will be sending across a JWT token when invoking GQL operations, you can also use groups for your user pools like your HR group or Developer group and then assign allow/deny permissions based on the group's user. You can also do fine-grained access control with runtime checks within the GraphQL resolvers themselves to deny or allow access to specific pieces of data.

How does AWS AppSync work?



Create and Upload Schema

Developers use the console editor to define and deploy a GraphQL API so the application can query and change data and update in real time

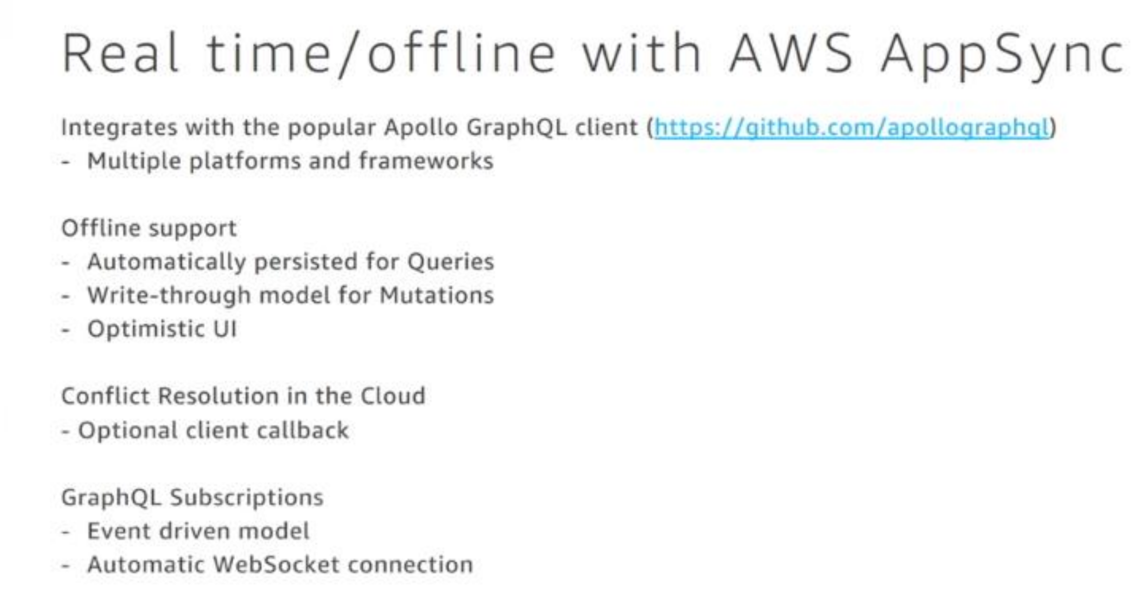
Connect Data Sources

AWS AppSync automatically provisions data sources and compute resources, or uses existing resources, and connects them to your GraphQL API

AppSync updates data in real time and manages data when offline

Client applications make GraphQL API calls to fetch data, make changes, or subscribe to changes in real time from all users and devices. Offline users can continue to access and change app data and get updates when they reconnect

We can provision DynamoDB resources based off a GQL schema, this allows you to own your own data and provide queries and wire resolvers for you automatically



Real time/offline with AWS AppSync

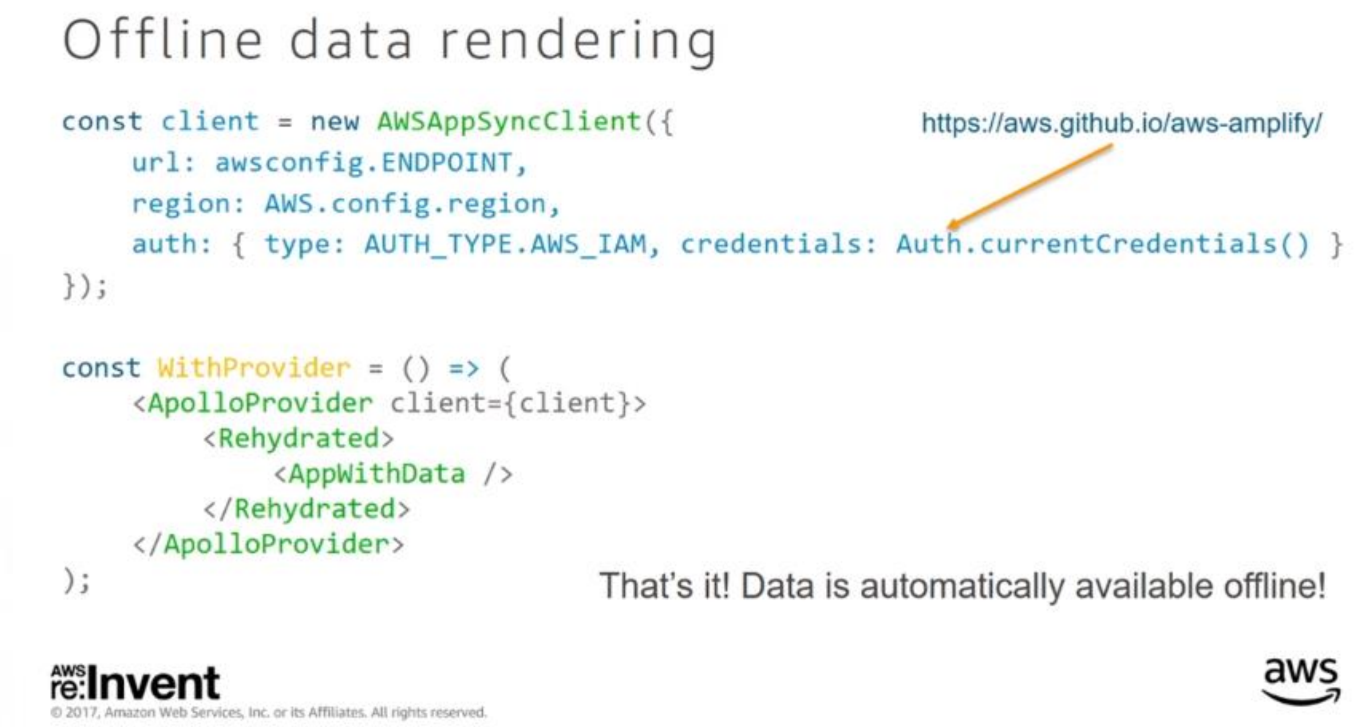
Integrates with the popular Apollo GraphQL client (https://github.com/apollographql)
- Multiple platforms and frameworks

Offline support
- Automatically persisted for Queries
- Write-through model for Mutations
- Optimistic UI

Conflict Resolution in the Cloud
- Optional client callback

GraphQL Subscriptions
- Event driven model
- Automatic WebSocket connection

Using our SDK allows you to have authorizations wired in for you, you just include the SDK, give it your specific authorization scheme like IAM or API Keys, and you can automatically persist your queries offline in a managed cache. The SDK uses a MQTT-over-websocket connection underneath.



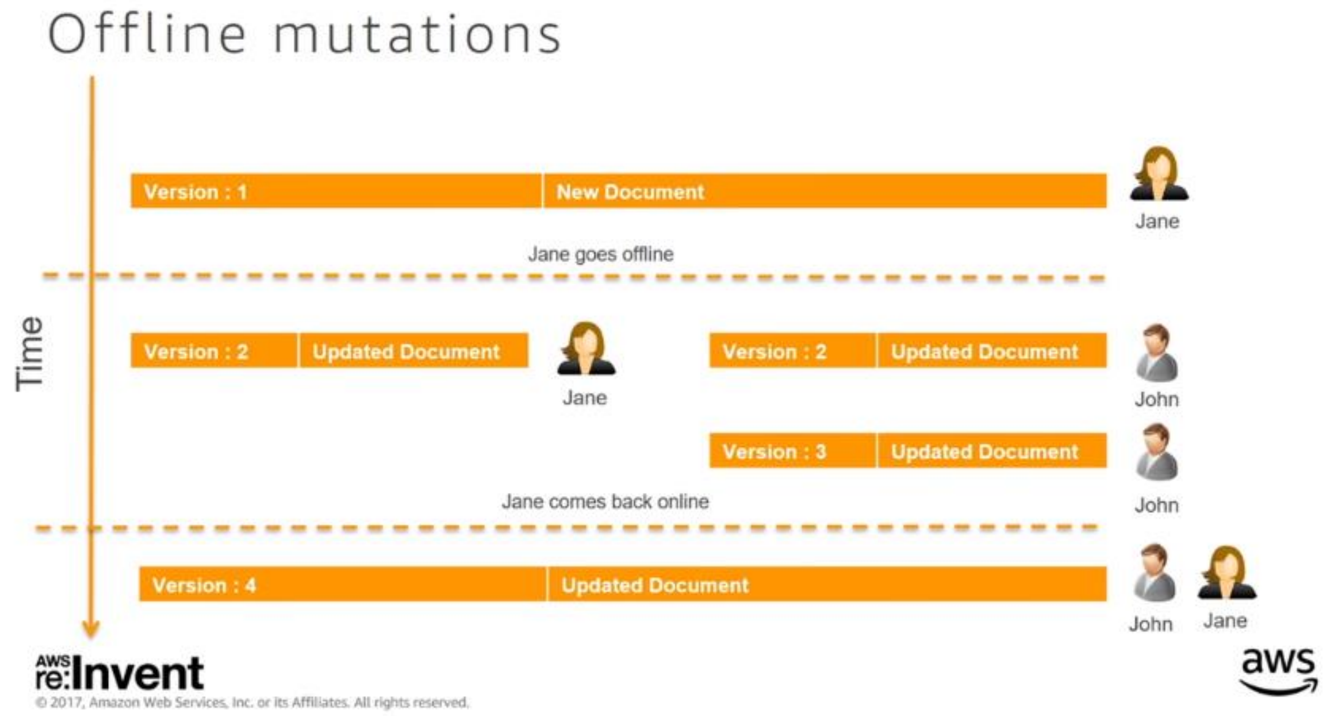
Offline data rendering

```
const client = new AWSAppSyncClient({
    url: awsconfig.ENDPOINT,
    region: AWS.config.region,
    auth: { type: AUTH_TYPE.AWS_IAM, credentials: Auth.currentCredentials() }
});

const WithProvider = () => (
    <ApolloProvider client={client}>
        <Rehydrated>
            <AppWithData />
        </Rehydrated>
    </ApolloProvider>
);
```

https://aws.github.io/aws-amplify/

That's it! Data is automatically available offline!

This is how you can do a query when using the AppSync SDK, you create a client and give the details of your GQL endpoint. The ***auth*** type could be IAM, API Keys, or Cognito user pools with auth.getUserSession.getAccessToken(<JWT Token>). You can then wire up the client using the Apollo client as above



Offline mutations

Time

Version : 1 | New Document — Jane

Jane goes offline

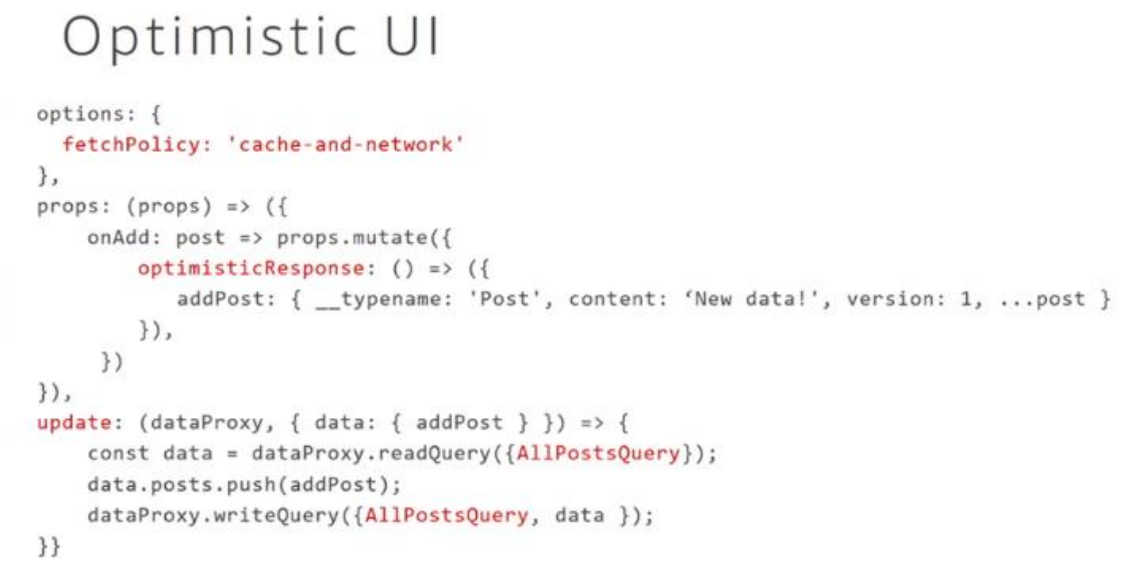
Version : 2 | Updated Document — Jane

Version : 2 | Updated Document — John

Version : 3 | Updated Document — John

Jane comes back online

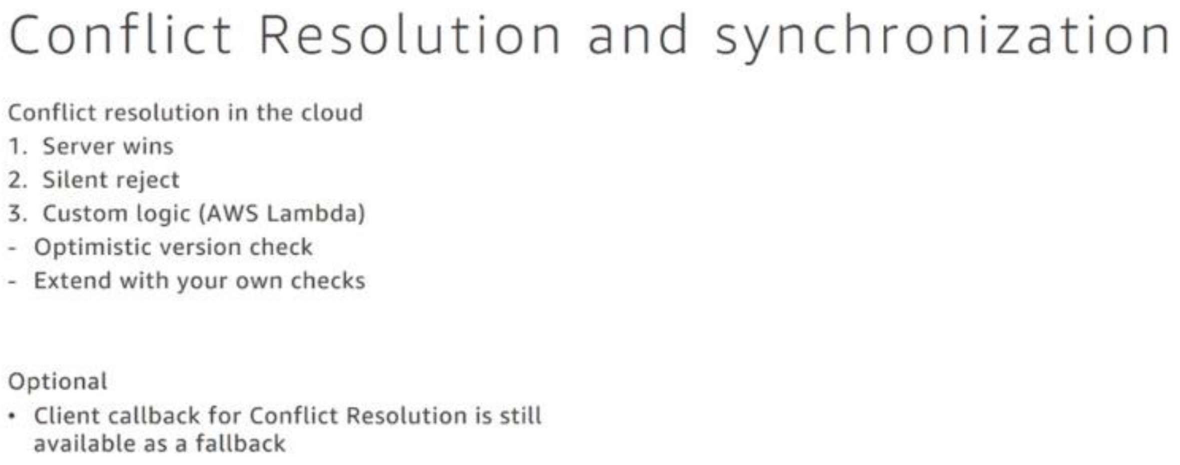
Version : 4 | Updated Document — John, Jane



Optimistic UI

```
options: {
  fetchPolicy: 'cache-and-network'
},
props: (props) => ({
    onAdd: post => props.mutate({
        optimisticResponse: () => ({
           addPost: { __typename: 'Post', content: 'New data!', version: 1, ...post }
        }),
     })
}),
update: (dataProxy, { data: { addPost } }) => {
    const data = dataProxy.readQuery({AllPostsQuery});
    data.posts.push(addPost);
    dataProxy.writeQuery({AllPostsQuery, data });
}}
```

We can do this with AWS AppSync client with the same React application using ApolloReact. AppSync supports all the 4 fetch policies, the ***cache-and-network*** policy above allows the developer to always check the cache before making a network call. We also support the 3 other policies, Cache-only, network-only, and cache-first. Next, you need to specify an ***OptimisticResponse*** which tells us what the needed view data is going to look like if the call succeeds. The ***addPost*** property then becomes an update call that we make to update the cache for the view. This means that all the views using the ***AllPostsQuery*** query will get their data updated automatically.



Conflict Resolution and synchronization

Conflict resolution in the cloud

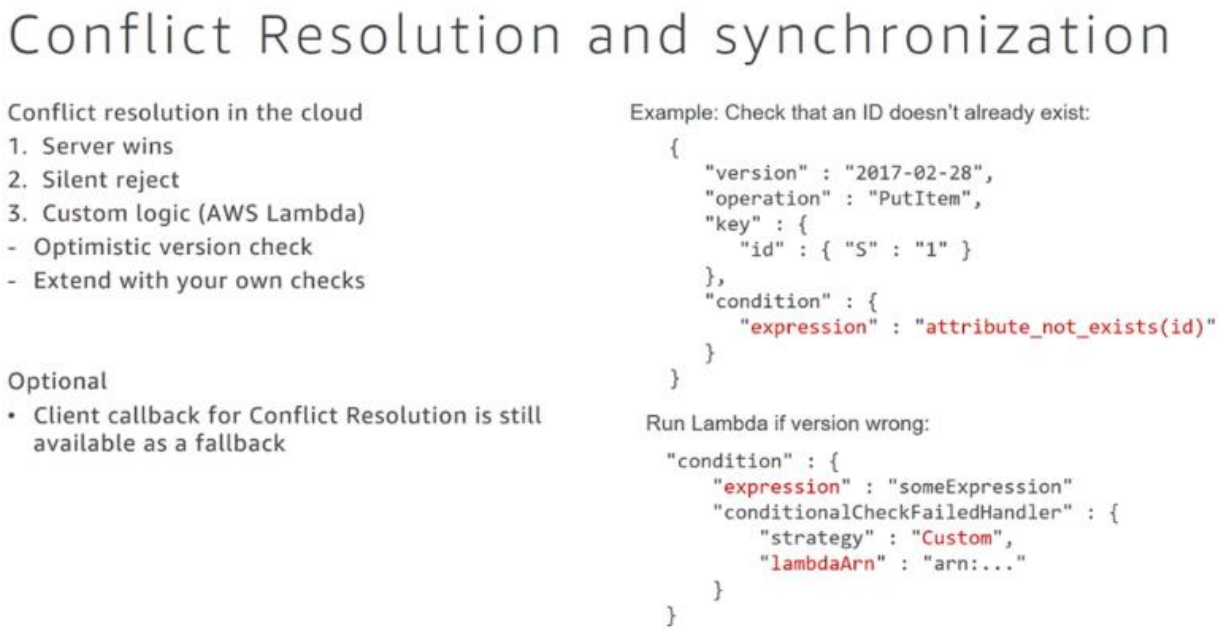
1. Server wins
2. Silent reject
3. Custom logic (AWS Lambda)

- Optimistic version check
- Extend with your own checks

Optional

- Client callback for Conflict Resolution is still available as a fallback

In case of conflict when a user comes back online while the data has changed, AppSync gives you 3 strategies of resolving conflicts listed above.



Conflict Resolution and synchronization

Conflict resolution in the cloud

1. Server wins
2. Silent reject
3. Custom logic (AWS Lambda)

- Optimistic version check
- Extend with your own checks

Optional

- Client callback for Conflict Resolution is still available as a fallback

Example: Check that an ID doesn't already exist:

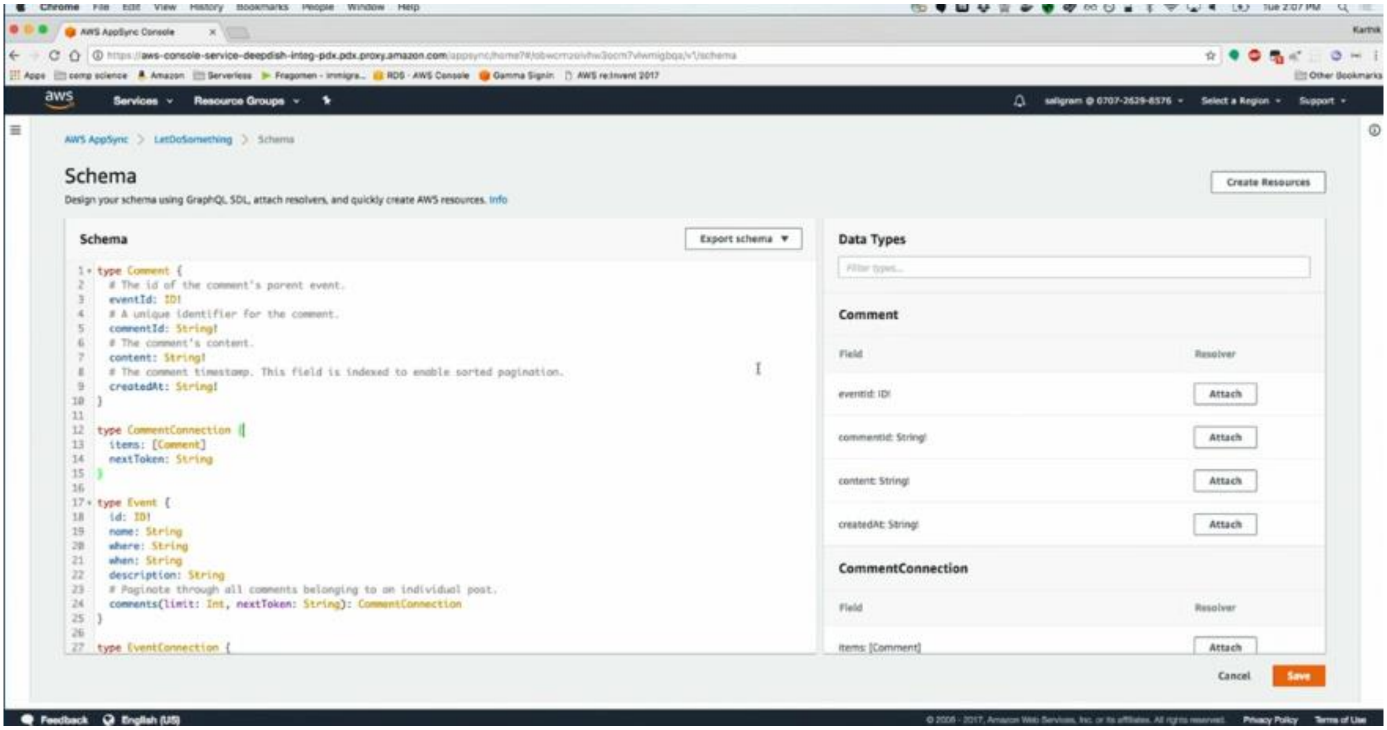
```
{
   "version" : "2017-02-28",
   "operation" : "PutItem",
   "key" : {
      "id" : { "S" : "1" }
   },
   "condition" : {
      "expression" : "attribute_not_exists(id)"
   }
}
```

Run Lambda if version wrong:

```
"condition" : {
    "expression" : "someExpression"
    "conditionalCheckFailedHandler" : {
        "strategy" : "Custom",
        "lambdaArn" : "arn:..."
    }
}
```

To do conflict resolution with AppSync, we start off with a mapping template which is a Velocity Template that helps to convert your GQL specification to your underlying data source specification, like converting a GQL call to a DynamoDB PutItem call. We can also specify a Boolean conditional expression to choose which strategy to use.

Demo: Offline data with AppSync



Schema

Design your schema using GraphQL SDL, attach resolvers, and quickly create AWS resources. Info

Schema

```
type Comment {
  # The id of the comment's parent event.
  eventId: ID!
  # A unique identifier for the comment.
  commentId: String!
  # The comment's content.
  content: String!
  # The comment timestamp. This field is indexed to enable sorted pagination.
  createdAt: String!
}

type CommentConnection {
  items: [Comment]
  nextToken: String
}

type Event {
  id: ID!
  name: String
  where: String
  when: String
  description: String
  # Paginate through all comments belonging to an individual post.
  comments(limit: Int, nextToken: String): CommentConnection
}

type EventConnection {
```

Data Types

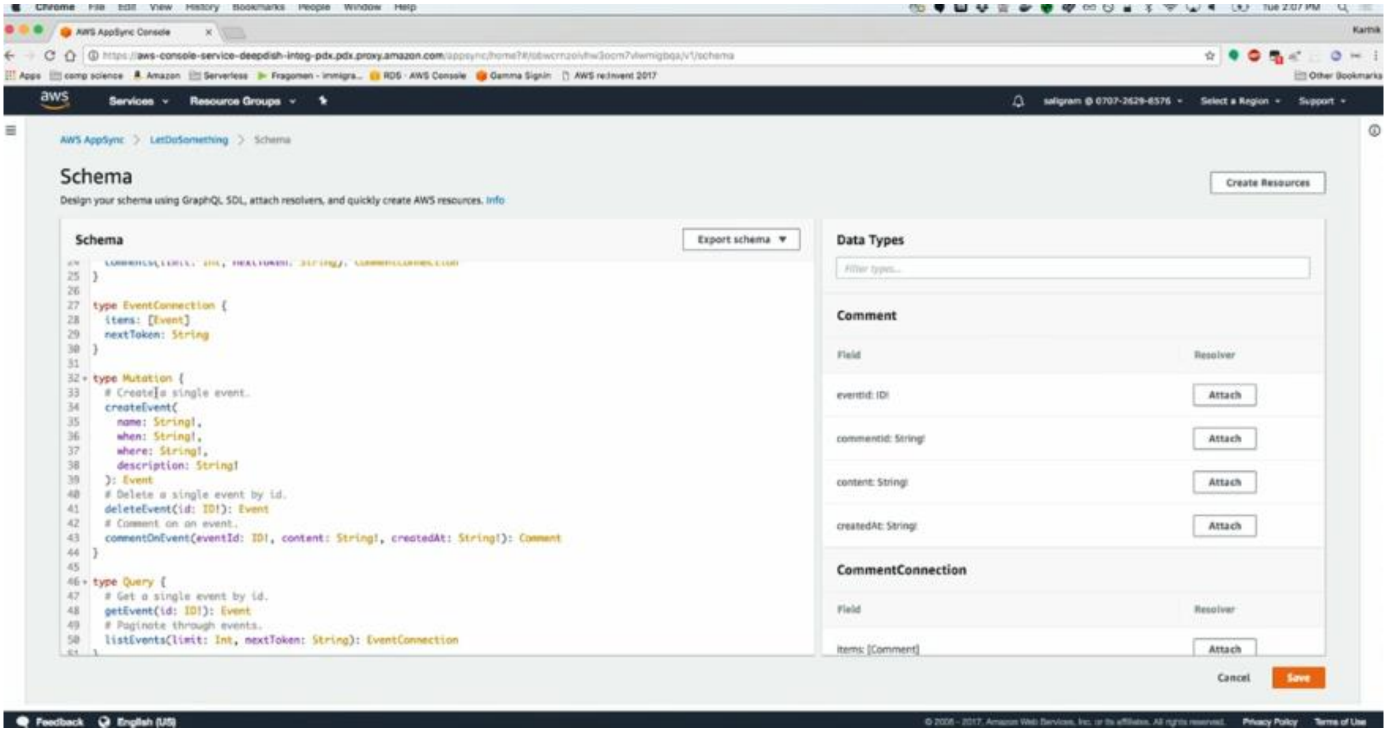
Comment

Field	Resolver
eventId: ID!	Attach
commentId: String!	Attach
content: String!	Attach
createdAt: String!	Attach

CommentConnection

Field	Resolver
items: [Comment]	Attach

We have created an app called EventsApp with the schema shown above



Schema

Design your schema using GraphQL SDL, attach resolvers, and quickly create AWS resources. Info

Schema

```
  comments(limit: Int, nextToken: String): CommentConnection
}

type EventConnection {
  items: [Event]
  nextToken: String
}

type Mutation {
  # Create a single event.
  createEvent(
    name: String!,
    when: String!,
    where: String!,
    description: String!
  ): Event
  # Delete a single event by id.
  deleteEvent(id: ID!): Event
  # Comment on an event.
  commentOnEvent(eventId: ID!, content: String!, createdAt: String!): Comment
}

type Query {
  # Get a single event by id.
  getEvent(id: ID!): Event
  # Paginate through events.
  listEvents(limit: Int, nextToken: String): EventConnection
}
```

Data Types

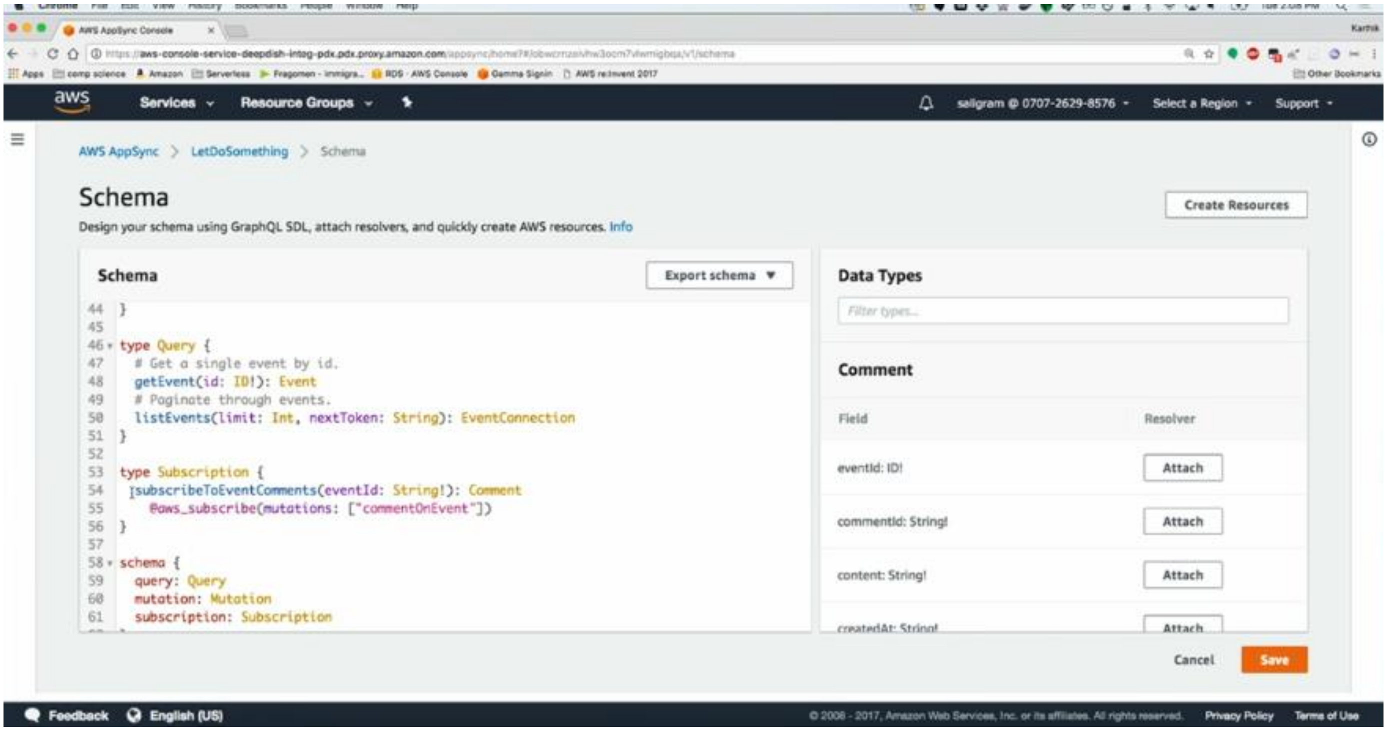
Comment

Field	Resolver
eventId: ID!	Attach
commentId: String!	Attach
content: String!	Attach
createdAt: String!	Attach

CommentConnection

Field	Resolver
items: [Comment]	Attach

We have 3 mutations called createEvent, deleteEvent, and commentEvent.



AWS AppSync > LetDoSomething > Schema

Schema

Design your schema using GraphQL SDL, attach resolvers, and quickly create AWS resources. Info

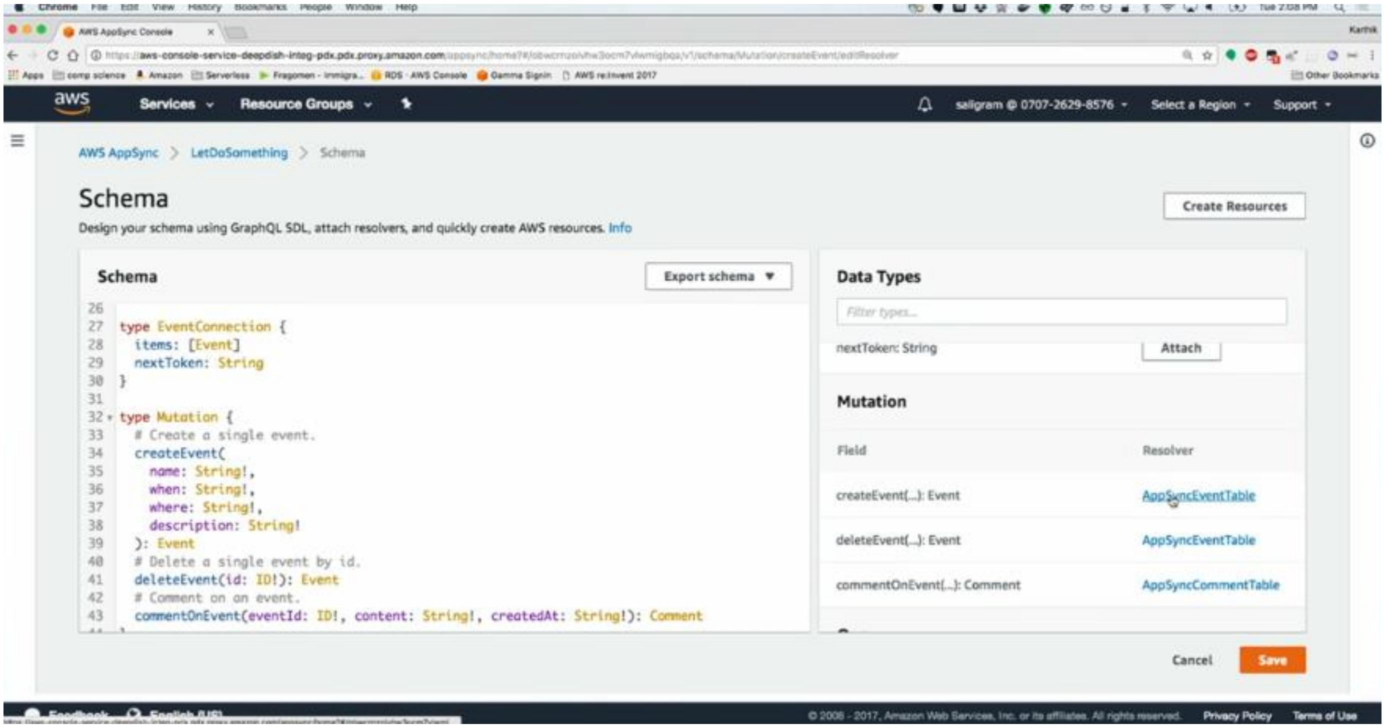
```
}

type Query {
  # Get a single event by id.
  getEvent(id: ID!): Event
  # Paginate through events.
  listEvents(limit: Int, nextToken: String): EventConnection
}

type Subscription {
 subscribeToEventComments(eventId: String!): Comment
   @aws_subscribe(mutations: ["commentOnEvent"])
}

schema {
  query: Query
  mutation: Mutation
  subscription: Subscription
```

Data Types

Comment

Field	Resolver
eventId: ID!	Attach
commentId: String!	Attach
content: String!	Attach
createdAt: String!	Attach

We also have 2 Queries called ***getEvent*** and ***listEvents***. We also have Subscription called ***subscribeToEventComments***.



AWS AppSync > LetDoSomething > Schema

Schema

Design your schema using GraphQL SDL, attach resolvers, and quickly create AWS resources. Info

```

type EventConnection {
  items: [Event]
  nextToken: String
}

type Mutation {
  # Create a single event.
  createEvent(
    name: String!,
    when: String!,
    where: String!,
    description: String!
  ): Event
  # Delete a single event by id.
  deleteEvent(id: ID!): Event
  # Comment on an event.
  commentOnEvent(eventId: ID!, content: String!, createdAt: String!): Comment
```

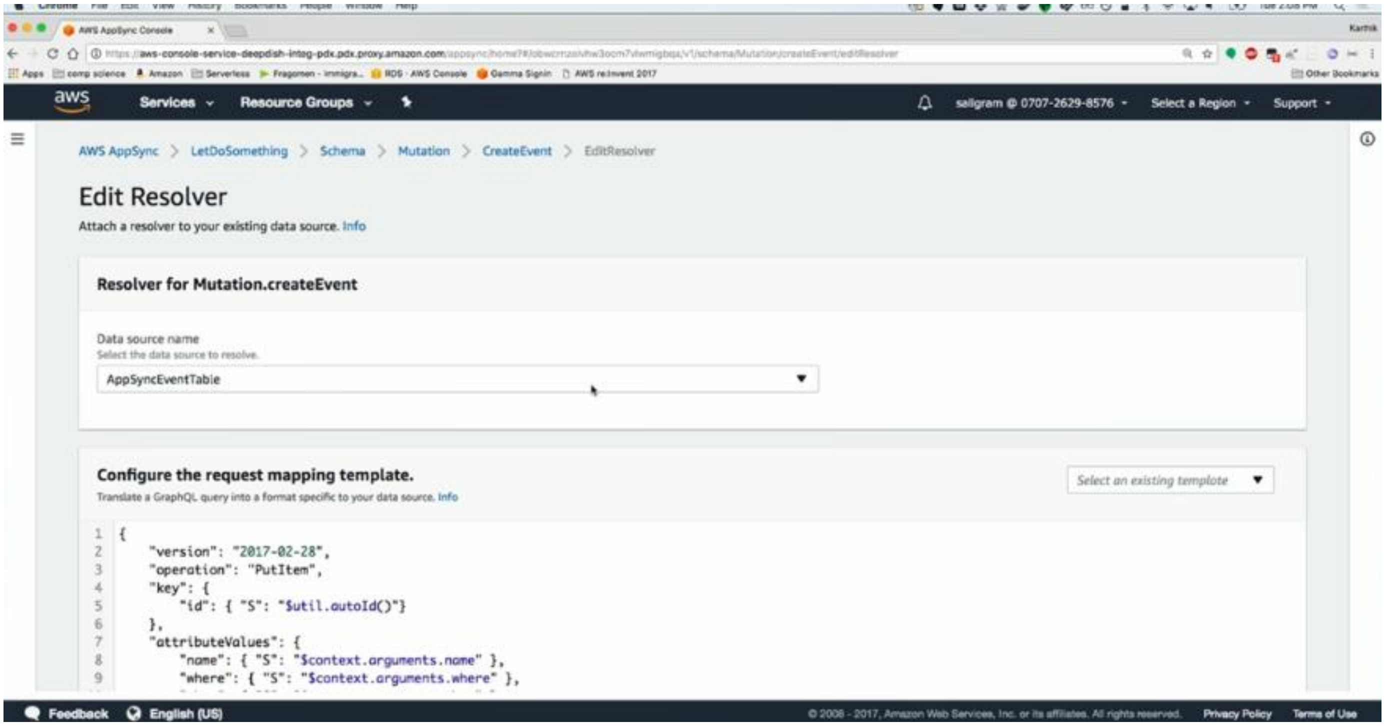
Data Types

Field	Resolver
nextToken: String	Attach

Mutation

Field	Resolver
createEvent(...): Event	AppSyncEventTable
deleteEvent(...): Event	AppSyncEventTable
commentOnEvent(...): Comment	AppSyncCommentTable

Note that each of the data types are Types in GQL and we can attach a resolver to them as we can see on the createEvent above


Services
Resource Groups
saligram @ 0707-2629-8576
Select a Region
Support
AWS AppSync > LetDoSomething > Schema > Mutation > CreateEvent > EditResolver
Select an existing template
Feedback
English (US)
© 2008 - 2017, Amazon Web Services, Inc. or its affiliates. All rights reserved.
Privacy Policy
Terms of Use

Edit Resolver

Attach a resolver to your existing data source. Info

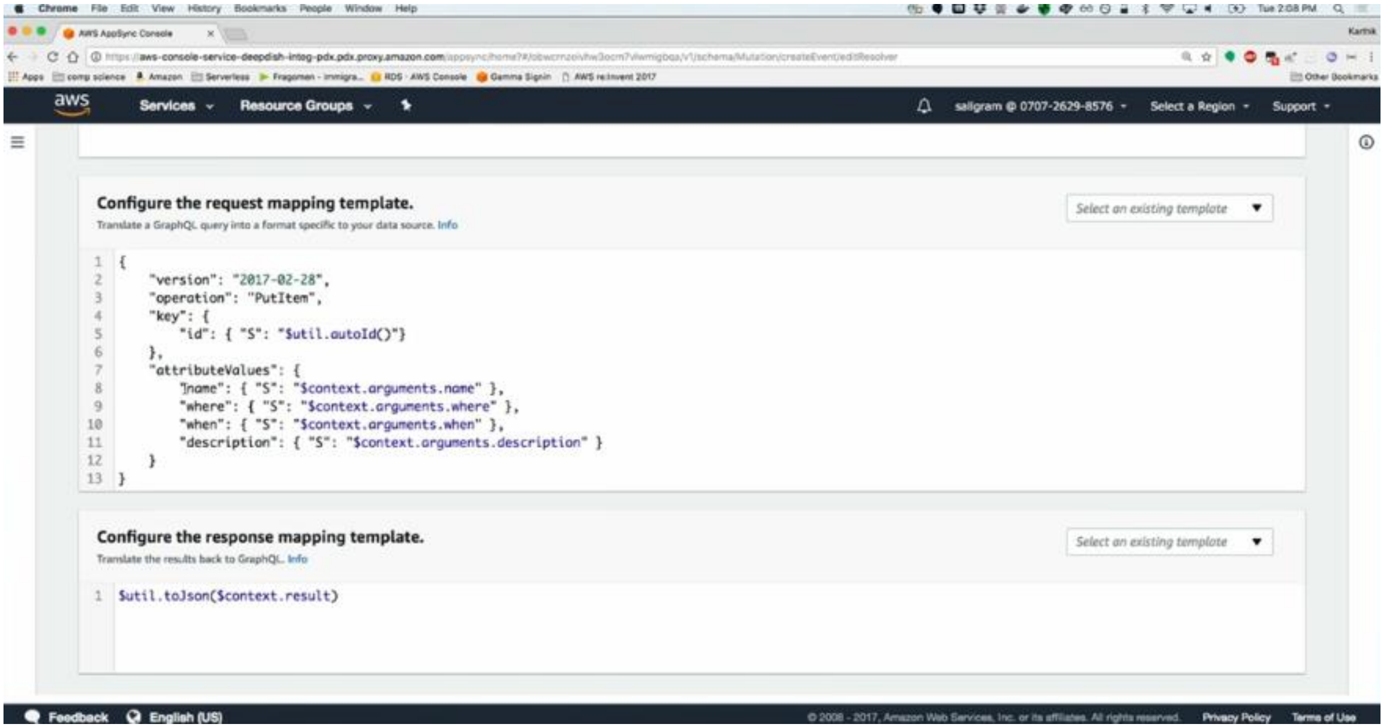
Resolver for Mutation.createEvent

Data source name
Select the data source to resolve.
AppSyncEventTable

Configure the request mapping template.
Translate a GraphQL query into a format specific to your data source. Info

```
{
    "version": "2017-02-28",
    "operation": "PutItem",
    "key": {
        "id": { "S": "$util.autoId()"}
    },
    "attributeValues": {
        "name": { "S": "$context.arguments.name" },
        "where": { "S": "$context.arguments.where" },
```

Above is what our mapping templates look like for Mutations. We have a DynamoDB table created for this that acts as the data source


Services
Resource Groups
saligram @ 0707-2629-8576
Select a Region
Support
Select an existing template
Select an existing template
Feedback
English (US)
© 2008 - 2017, Amazon Web Services, Inc. or its affiliates. All rights reserved.
Privacy Policy
Terms of Use

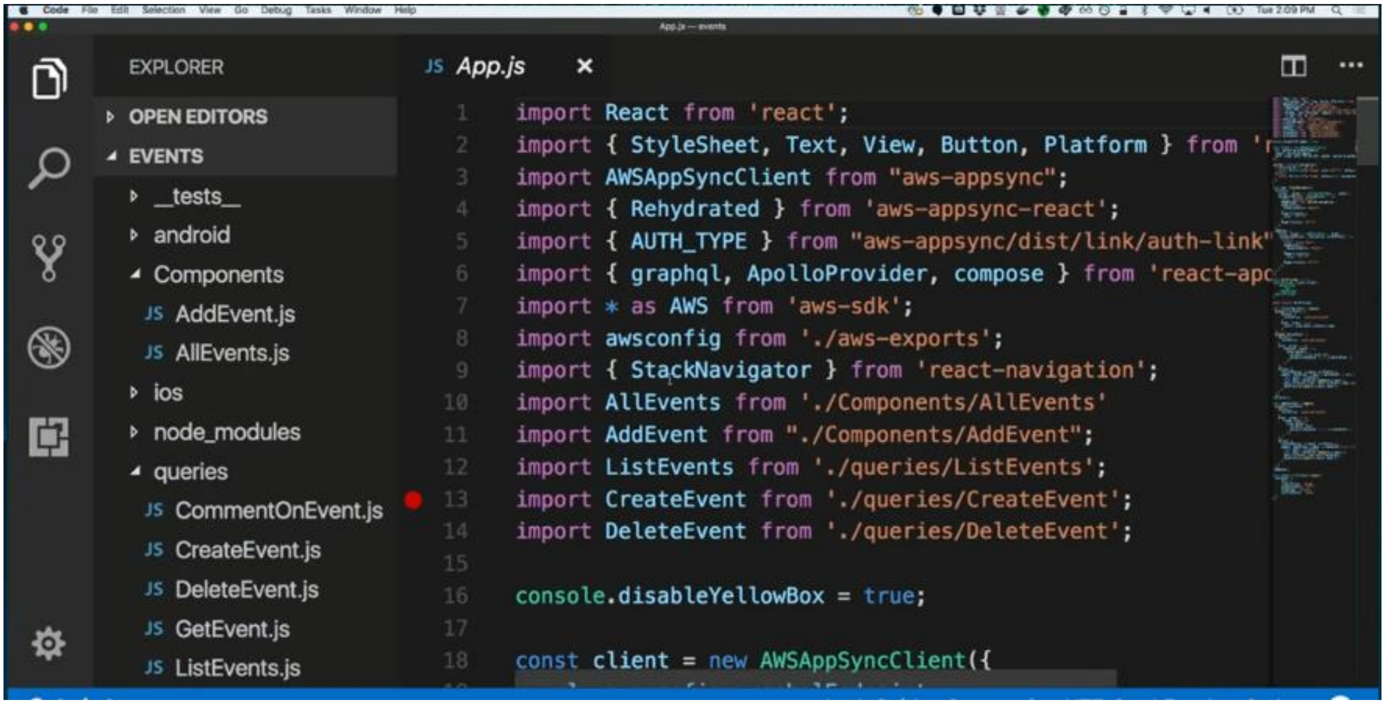
Configure the request mapping template.
Translate a GraphQL query into a format specific to your data source. Info

```
{
    "version": "2017-02-28",
    "operation": "PutItem",
    "key": {
        "id": { "S": "$util.autoId()"}
    },
    "attributeValues": {
        "name": { "S": "$context.arguments.name" },
        "where": { "S": "$context.arguments.where" },
        "when": { "S": "$context.arguments.when" },
        "description": { "S": "$context.arguments.description" }
    }
}
```

Configure the response mapping template.
Translate the results back to GraphQL. Info

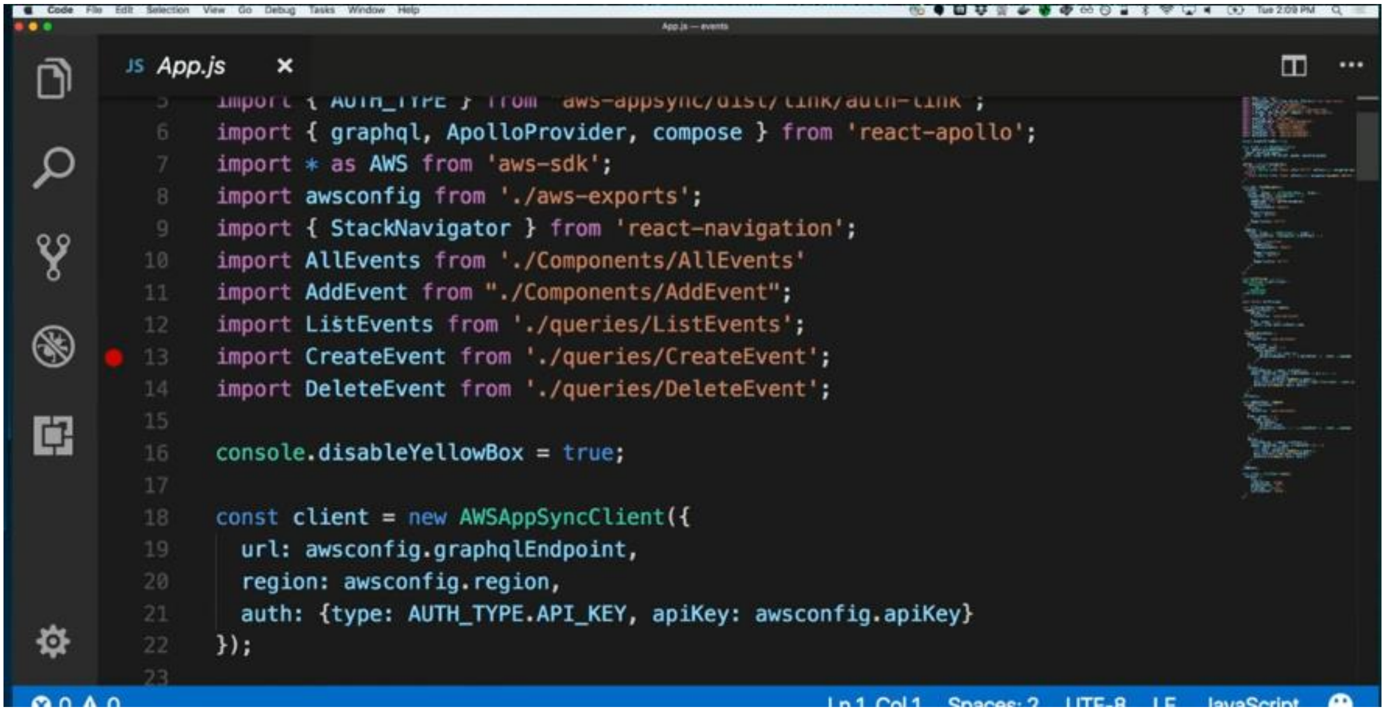
```
$util.toJson($context.result)
```

We also have ***2 mapping templates***, a ***request mapping template*** and a ***response mapping template***. The request mapping template is basically a ***PutItem*** call to the DynamoDB table with a ***hashKey*** called ***id***. We are using a utility function that allows us to ***generate a UUID*** as the value of the id. The extra attributes are mapped to ***arguments*** that come as part of our ***createEvent mutation***. Then in our response mapping template, we are taking the response we got back from ***DynamoDB*** and converting it to ***JSON*** before returning it back to the client.

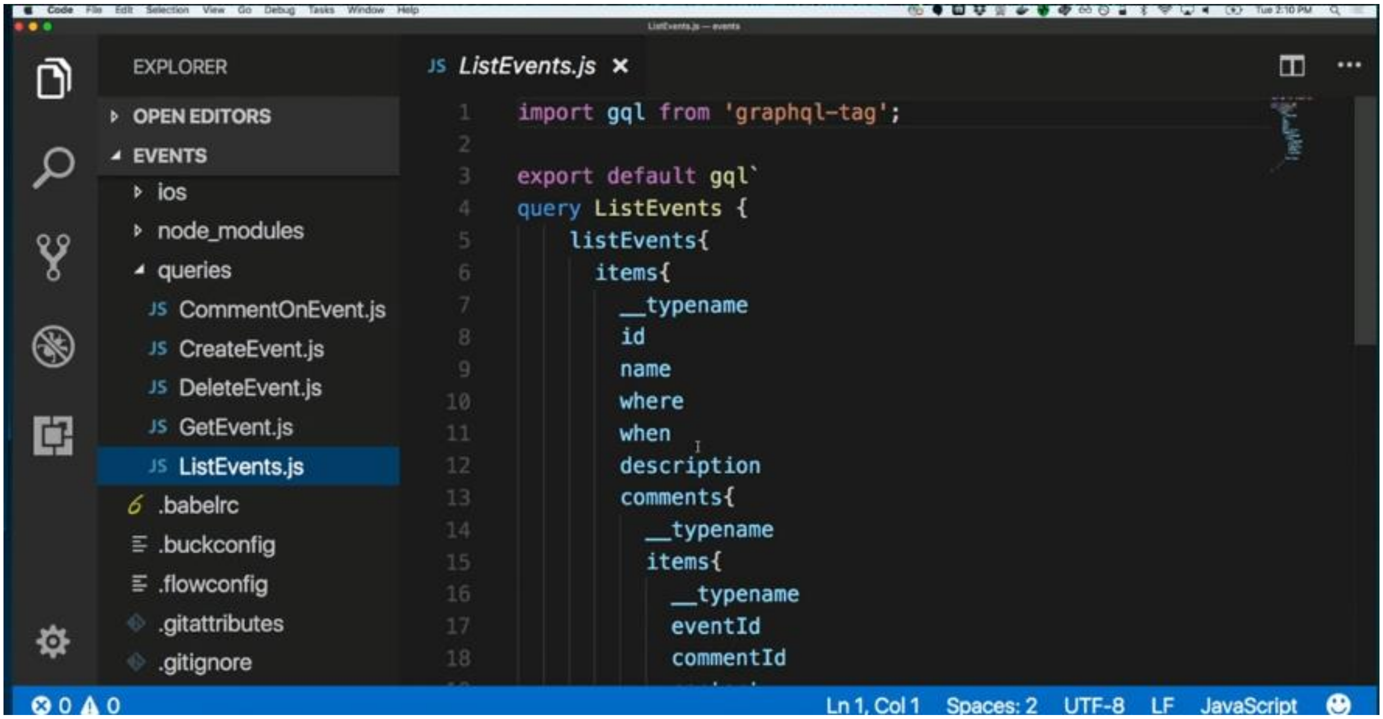


```
1  import React from 'react';
2  import { StyleSheet, Text, View, Button, Platform } from 'r
3  import AWSAppSyncClient from "aws-appsync";
4  import { Rehydrated } from 'aws-appsync-react';
5  import { AUTH_TYPE } from "aws-appsync/dist/link/auth-link"
6  import { graphql, ApolloProvider, compose } from 'react-apo
7  import * as AWS from 'aws-sdk';
8  import awsconfig from './aws-exports';
9  import { StackNavigator } from 'react-navigation';
10 import AllEvents from './Components/AllEvents'
11 import AddEvent from "./Components/AddEvent";
12 import ListEvents from './queries/ListEvents';
13 import CreateEvent from './queries/CreateEvent';
14 import DeleteEvent from './queries/DeleteEvent';
15
16 console.disableYellowBox = true;
17
18 const client = new AWSAppSyncClient({
```

On the client side, we can see what this process looks like using our React Native app with the AWS AppSync client. We also import the libraries needed like react-apollo on line 6. We have 2 screens in our app called AllEvents and AddEvent as seen from lines 10 and 11. We also have 3 queries that we are using in the application on lines 12, 13, and 14.

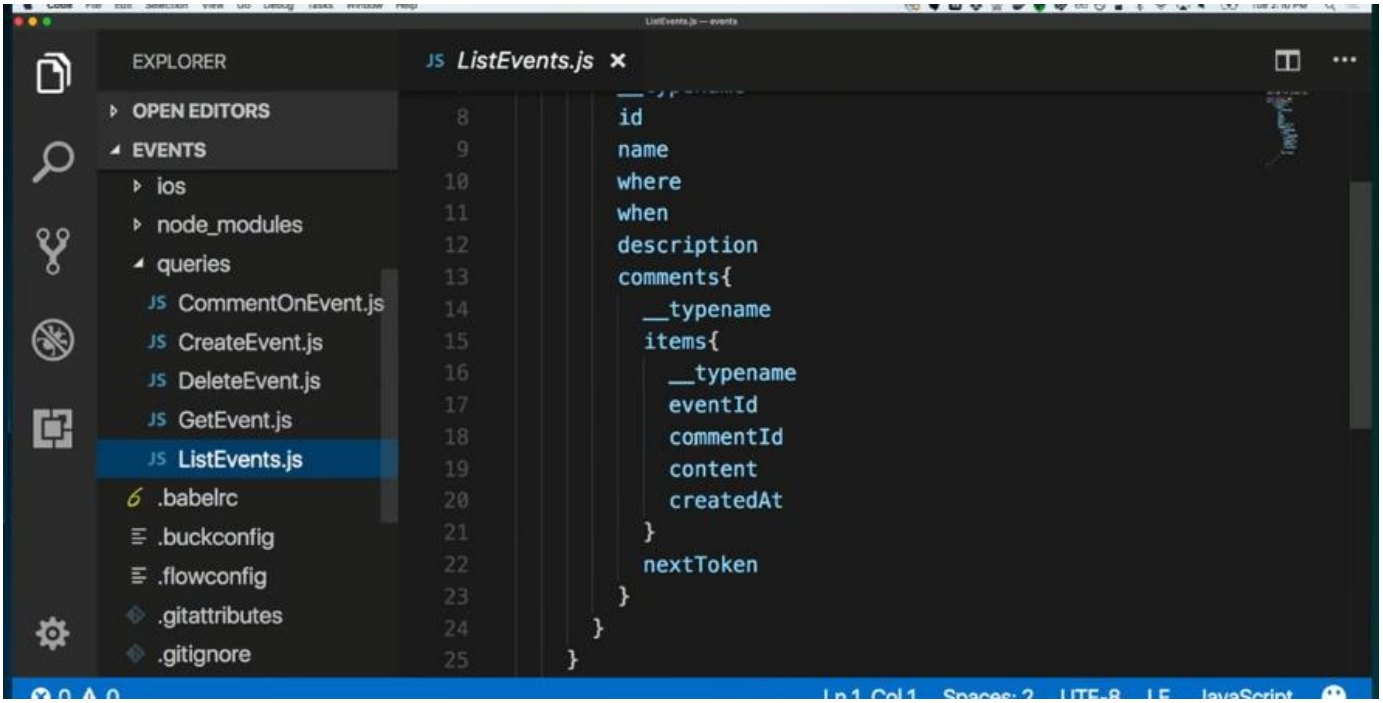


```
5  import { AUTH_TYPE } from "aws-appsync/dist/link/auth-link";
6  import { graphql, ApolloProvider, compose } from 'react-apollo';
7  import * as AWS from 'aws-sdk';
8  import awsconfig from './aws-exports';
9  import { StackNavigator } from 'react-navigation';
10 import AllEvents from './Components/AllEvents'
11 import AddEvent from "./Components/AddEvent";
12 import ListEvents from './queries/ListEvents';
13 import CreateEvent from './queries/CreateEvent';
14 import DeleteEvent from './queries/DeleteEvent';
15
16 console.disableYellowBox = true;
17
18 const client = new AWSAppSyncClient({
19   url: awsconfig.graphqlEndpoint,
20   region: awsconfig.region,
21   auth: {type: AUTH_TYPE.API_KEY, apiKey: awsconfig.apiKey}
22 });
23
```

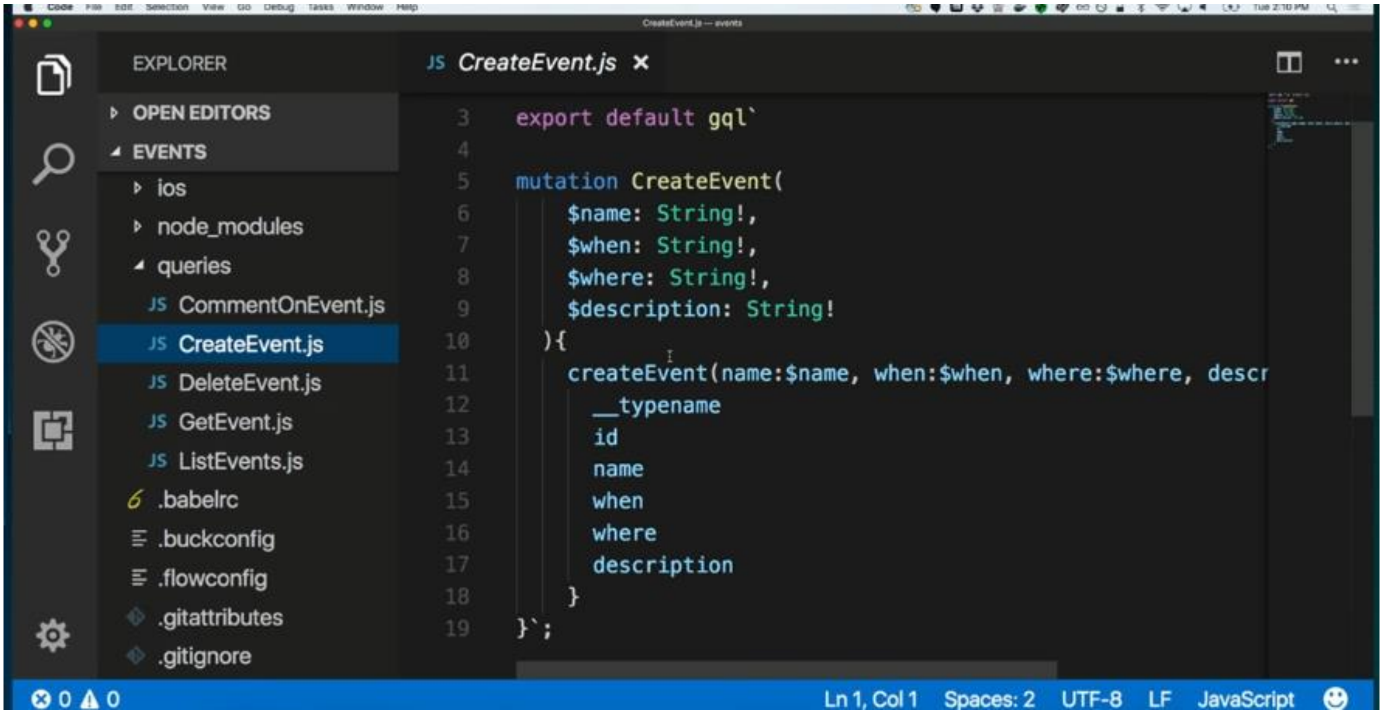


```
import gql from 'graphql-tag';

export default gql`
query ListEvents {
    listEvents{
      items{
        __typename
        id
        name
        where
        when
        description
        comments{
          __typename
          items{
            __typename
            eventId
            commentId
```

The ListEvents.js file contains a GQL query for listing all the available events, it asks for Items, comments, and items for the comments

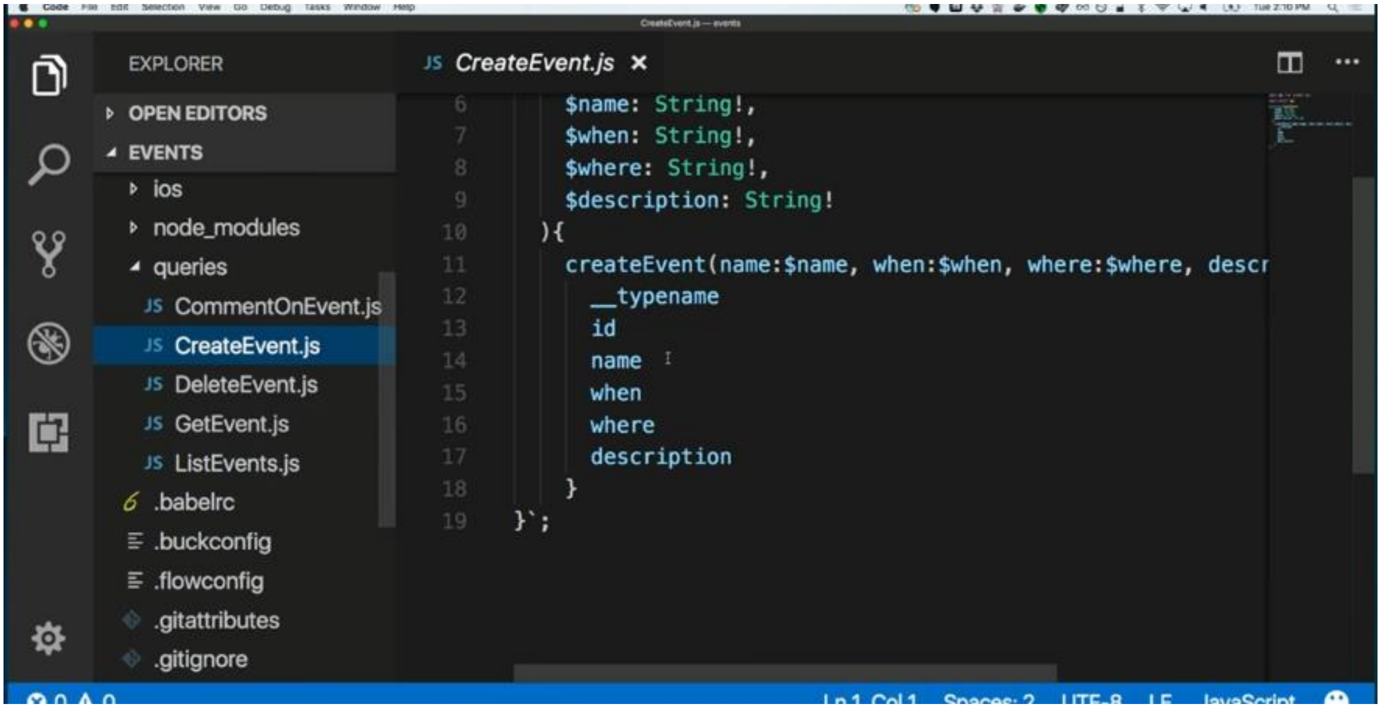


```
        id
        name
        where
        when
        description
        comments{
          __typename
          items{
            __typename
            eventId
            commentId
            content
            createdAt
          }
          nextToken
        }
      }
    }
```

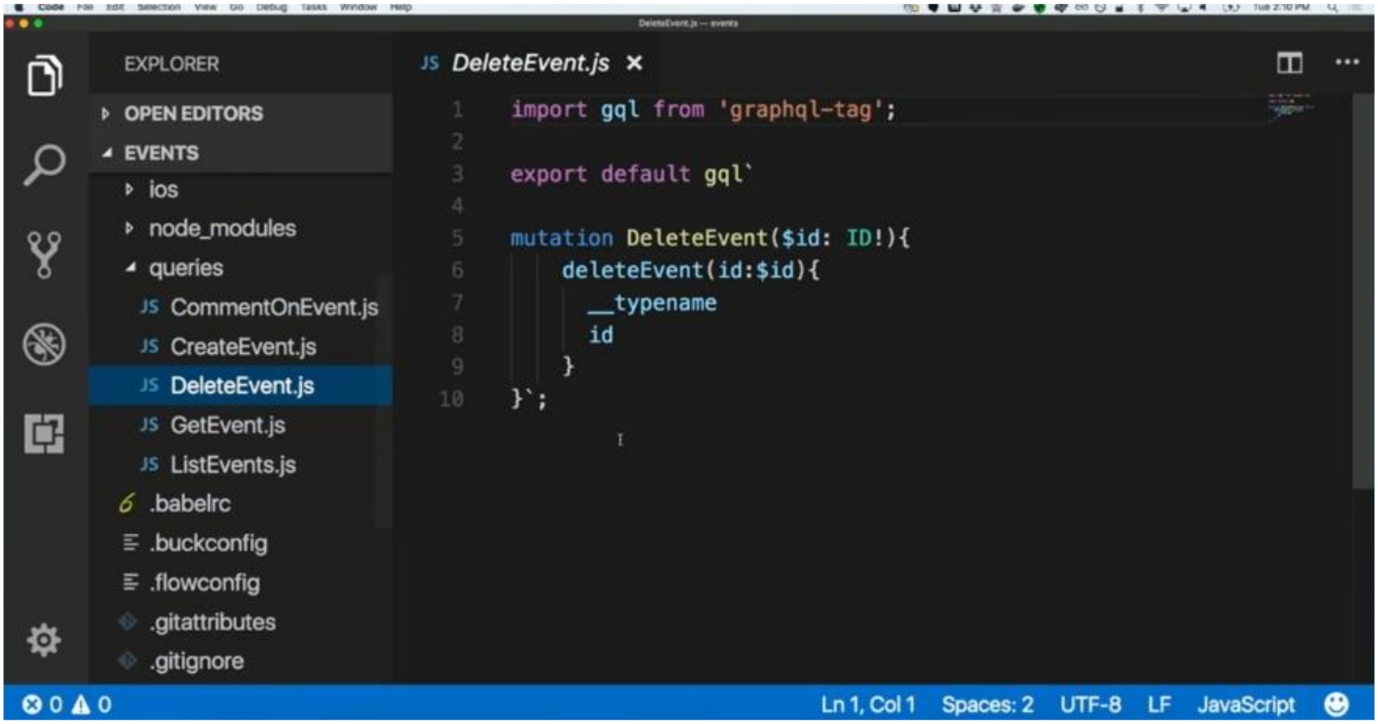


```
export default gql`

mutation CreateEvent(
    $name: String!,
    $when: String!,
    $where: String!,
    $description: String!
  ){
    createEvent(name:$name, when:$when, where:$where, descr
      __typename
      id
      name
      when
      where
      description
    }
}`;
```

The CreateEvent.js file contains the CreateEvent mutation that maps to the mutation shown earlier

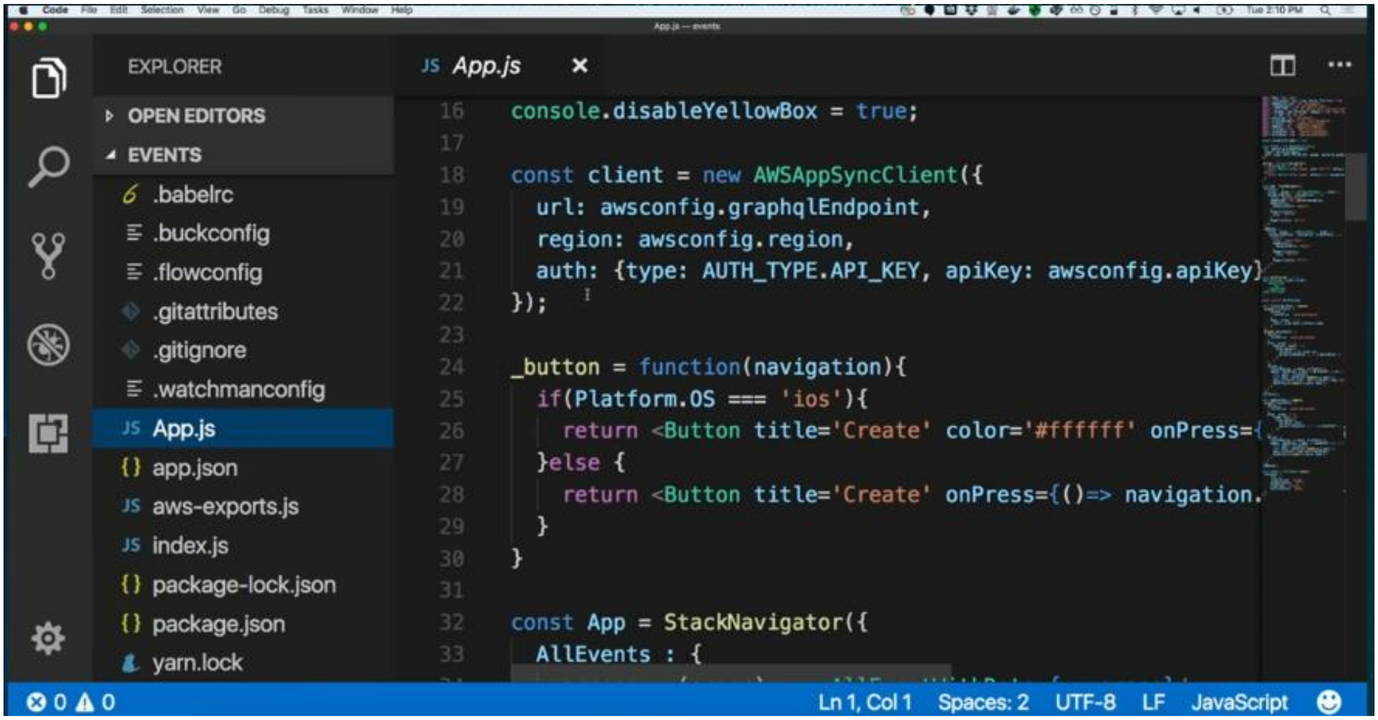


```
    $name: String!,
    $when: String!,
    $where: String!,
    $description: String!
  ){
    createEvent(name:$name, when:$when, where:$where, descr
      __typename
      id
      name
      when
      where
      description
    }
}`;
```



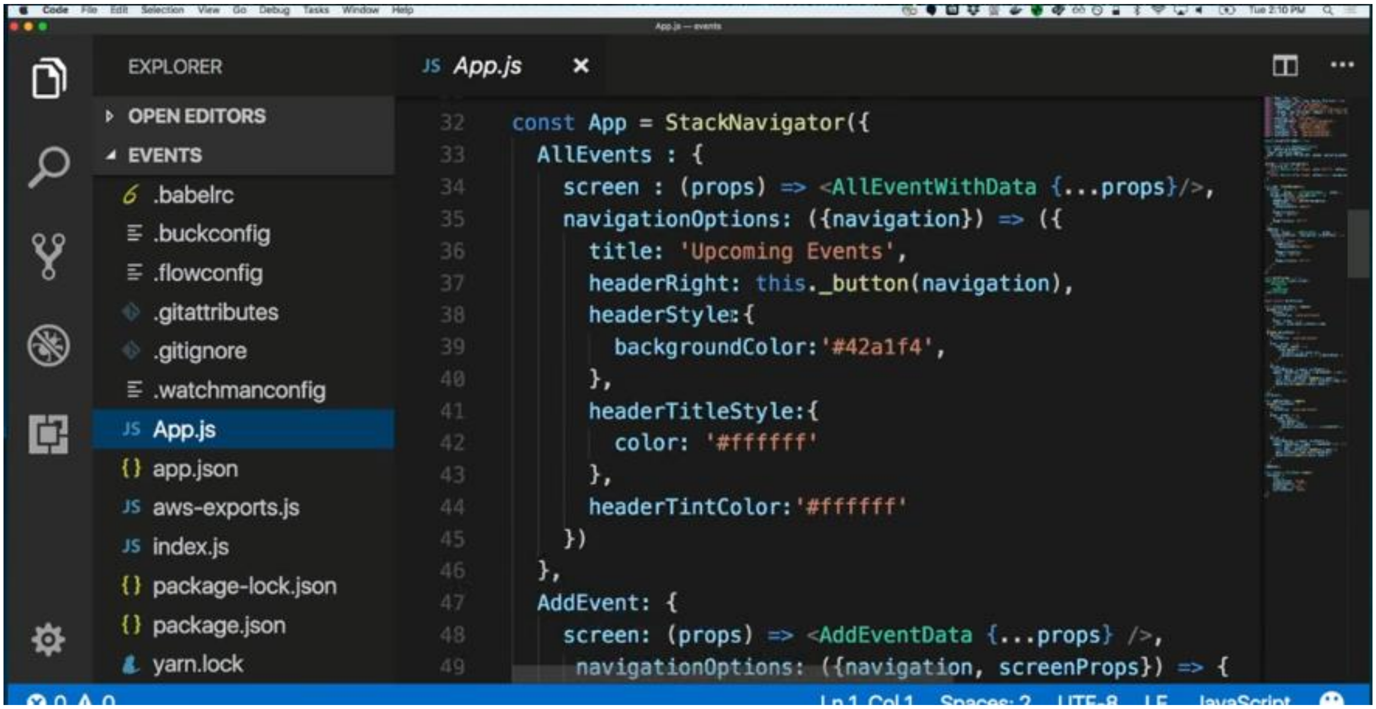
```javascript
import gql from 'graphql-tag';

export default gql`

mutation DeleteEvent($id: ID!){
    deleteEvent(id:$id){
      __typename
      id
    }
}`;
```

The DeleteEvent.js file contains a mutation

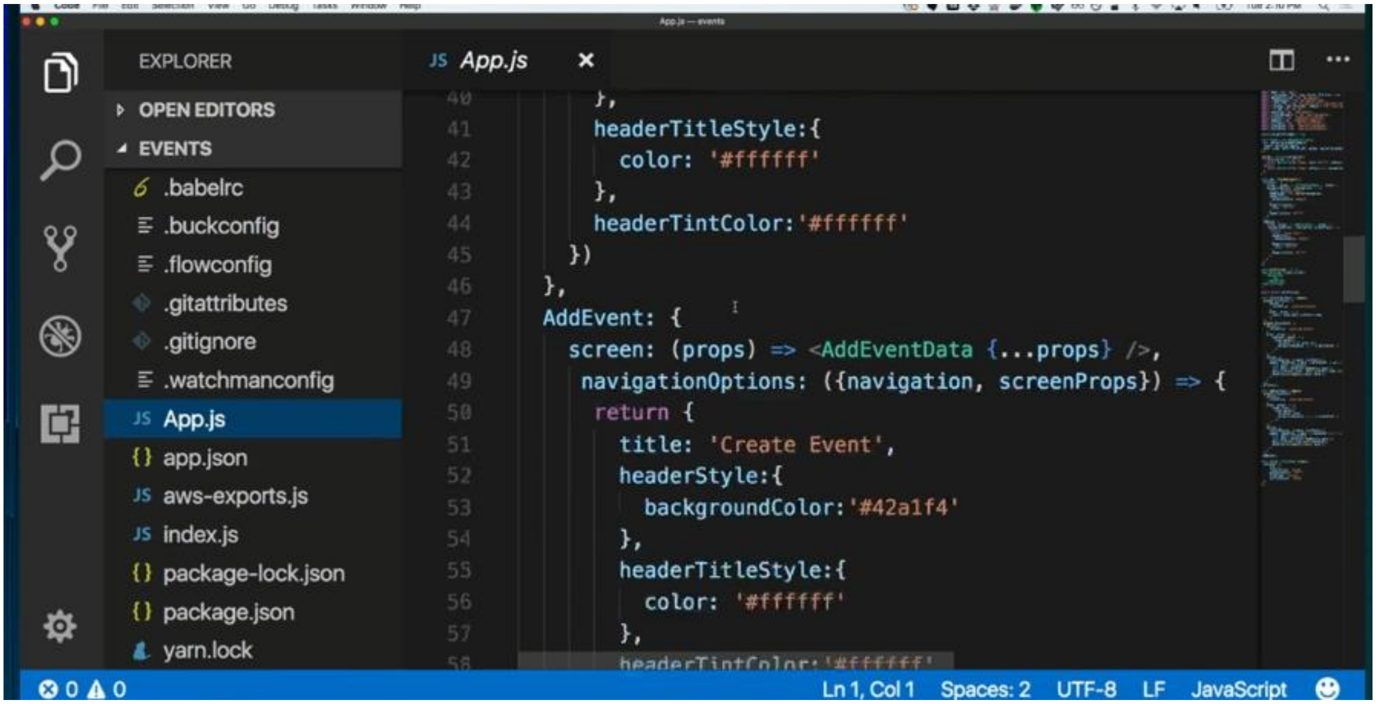


```javascript
16  console.disableYellowBox = true;
17
18  const client = new AWSAppSyncClient({
19    url: awsconfig.graphqlEndpoint,
20    region: awsconfig.region,
21    auth: {type: AUTH_TYPE.API_KEY, apiKey: awsconfig.apiKey}
22  });
23
24  _button = function(navigation){
25    if(Platform.OS === 'ios'){
26      return <Button title='Create' color='#ffffff' onPress={
27    }else {
28      return <Button title='Create' onPress={()=> navigation.
29    }
30  }
31
32  const App = StackNavigator({
33    AllEvents : {
```

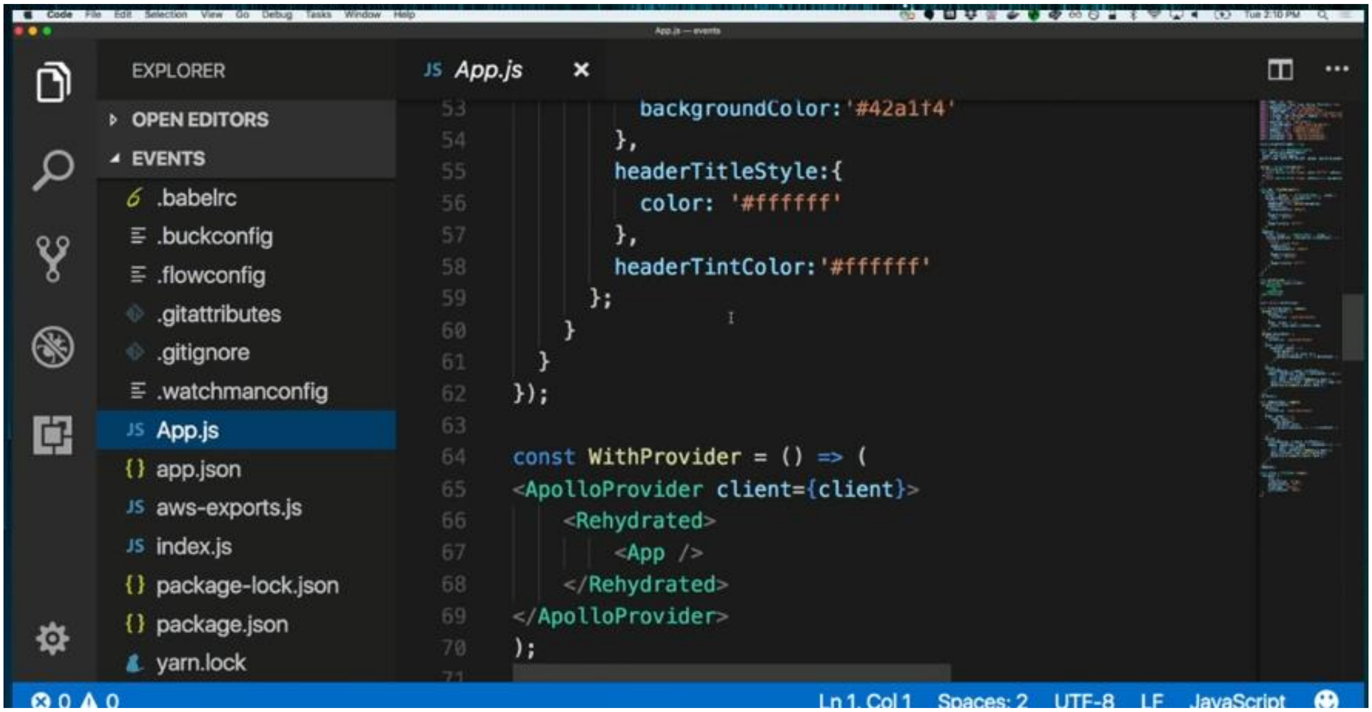
In our App.js file, we are using the API Key authorization mechanism for our client as in line 21. We created a new client and provided the needed details for it to connect to AppSync like the GQL endpoint and the AWS region.



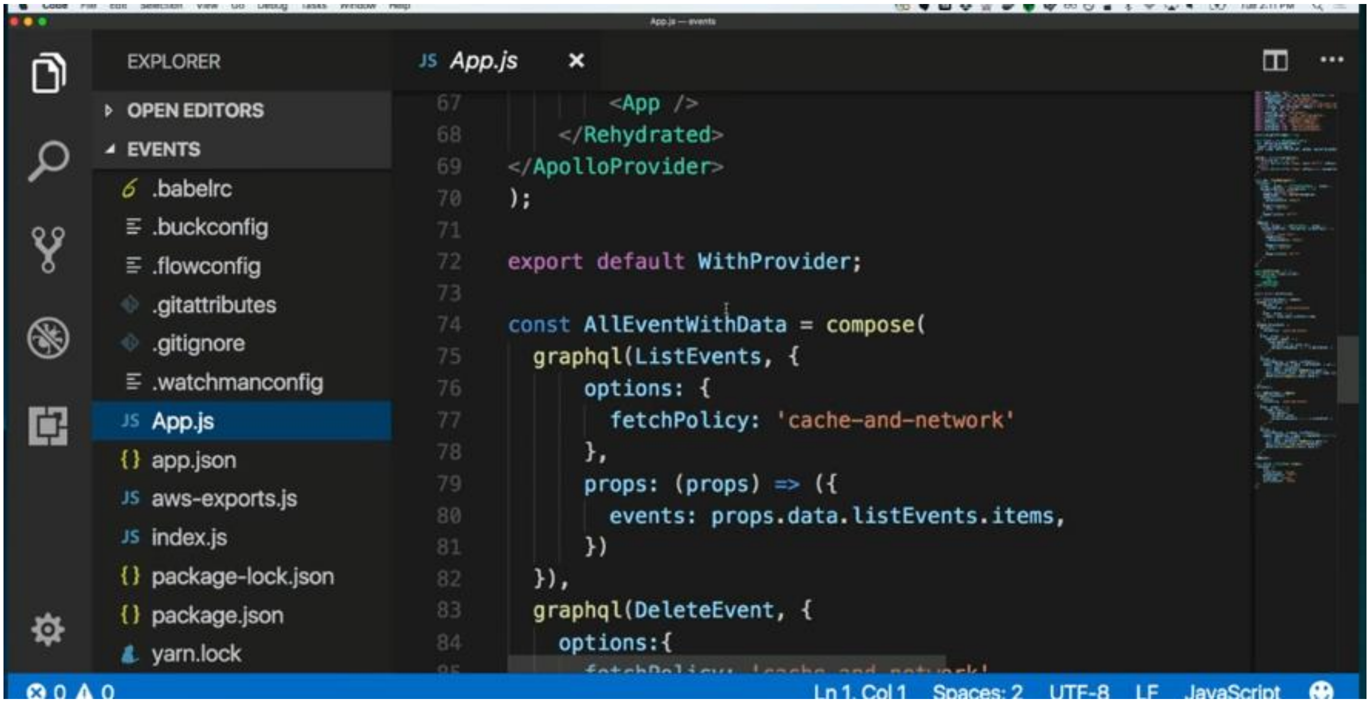
```
const App = StackNavigator({
  AllEvents : {
    screen : (props) => <AllEventWithData {...props}/>,
    navigationOptions: ({navigation}) => ({
      title: 'Upcoming Events',
      headerRight: this._button(navigation),
      headerStyle:{
        backgroundColor:'#42a1f4',
      },
      headerTitleStyle:{
        color: '#ffffff'
      },
      headerTintColor:'#ffffff'
    })
  },
  AddEvent: {
    screen: (props) => <AddEventData {...props} />,
     navigationOptions: ({navigation, screenProps}) => {
```



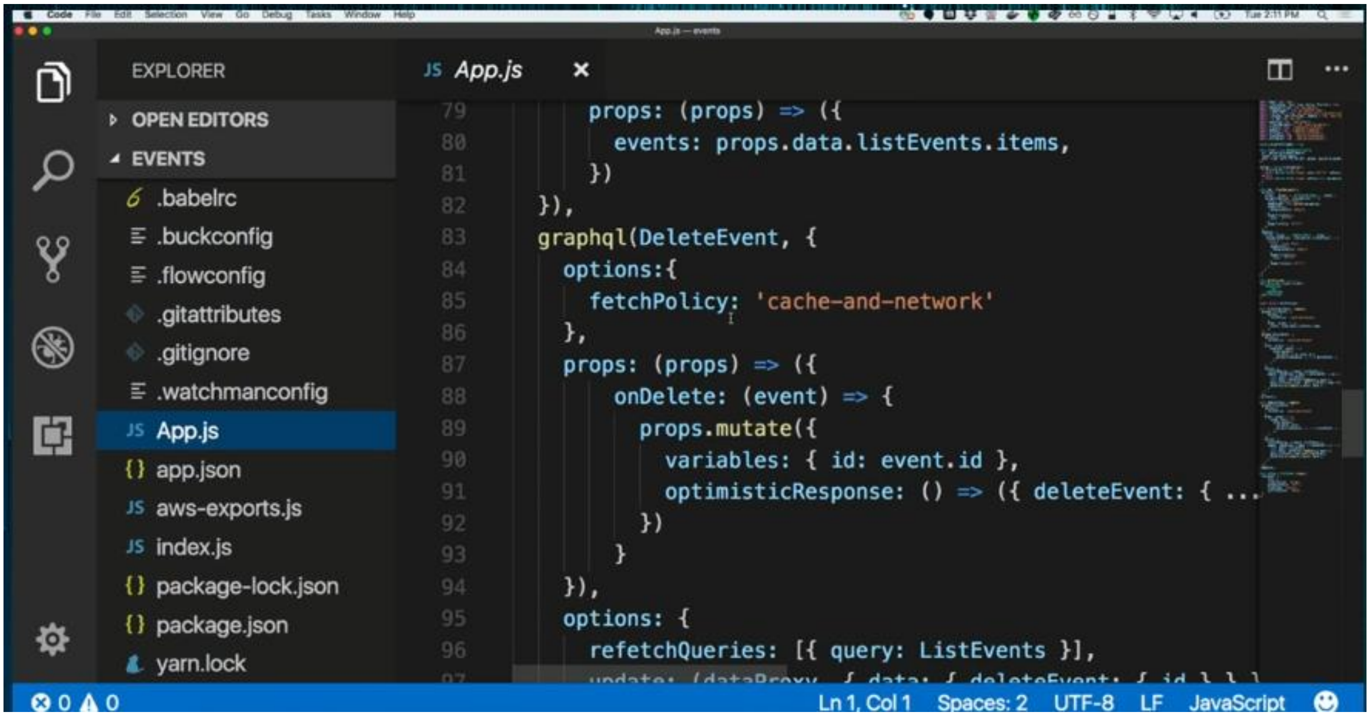
```
      },
      headerTitleStyle:{
        color: '#ffffff'
      },
      headerTintColor:'#ffffff'
    })
  },
  AddEvent: {
    screen: (props) => <AddEventData {...props} />,
     navigationOptions: ({navigation, screenProps}) => {
      return {
        title: 'Create Event',
        headerStyle:{
          backgroundColor:'#42a1f4'
        },
        headerTitleStyle:{
          color: '#ffffff'
        },
        headerTintColor:'#ffffff'
```



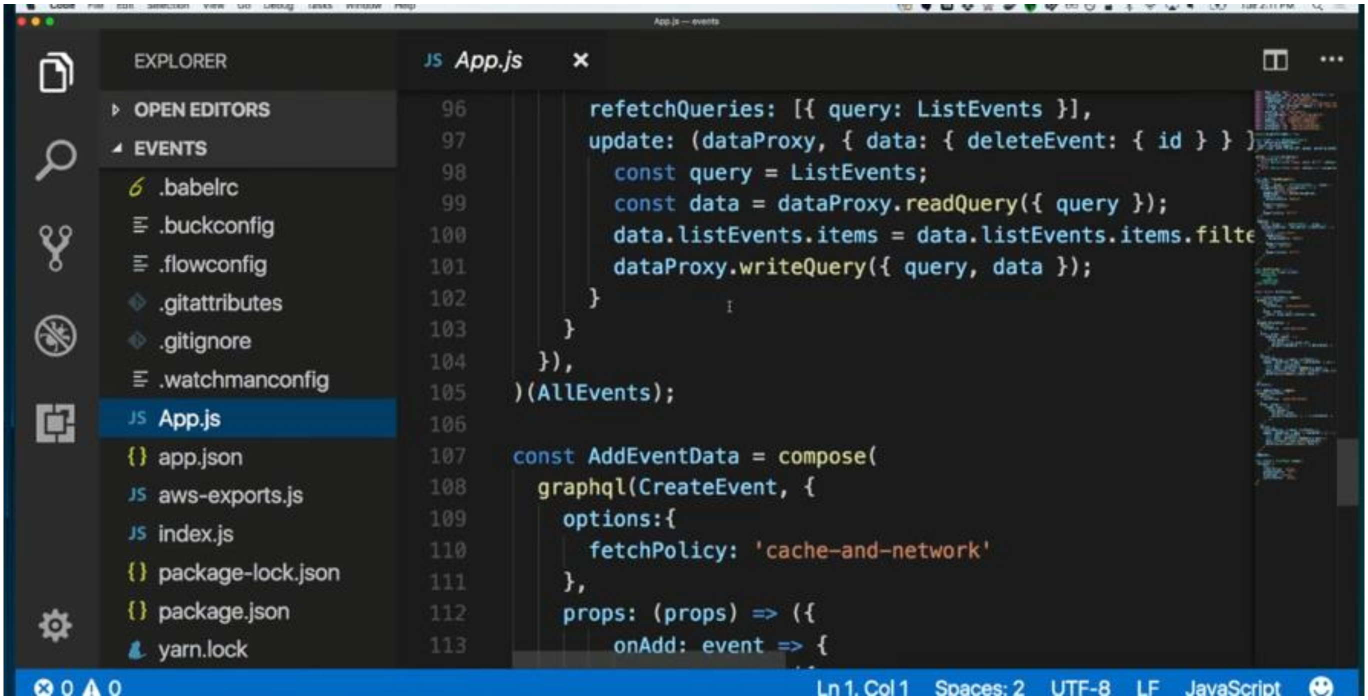
```
     backgroundColor:'#42a1f4'
    },
    headerTitleStyle:{
     color: '#ffffff'
    },
    headerTintColor:'#ffffff'
   };
  }
 }
});

const WithProvider = () => (
<ApolloProvider client={client}>
  <Rehydrated>
    <App />
  </Rehydrated>
</ApolloProvider>
);
```



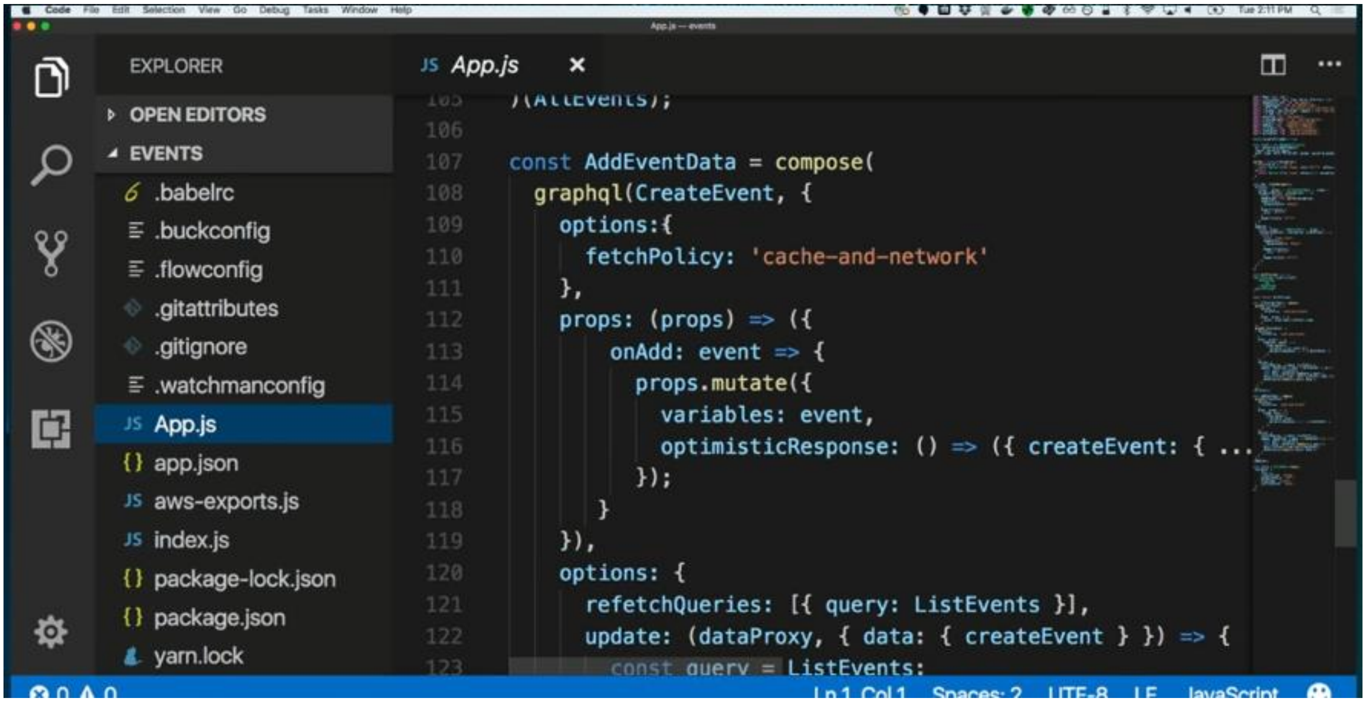
```
    <App />
  </Rehydrated>
</ApolloProvider>
);

export default WithProvider;

const AllEventWithData = compose(
 graphql(ListEvents, {
   options: {
    fetchPolicy: 'cache-and-network'
   },
   props: (props) => ({
    events: props.data.listEvents.items,
   })
 }),
 graphql(DeleteEvent, {
  options:{
   fetchPolicy: 'cache-and-network'
```



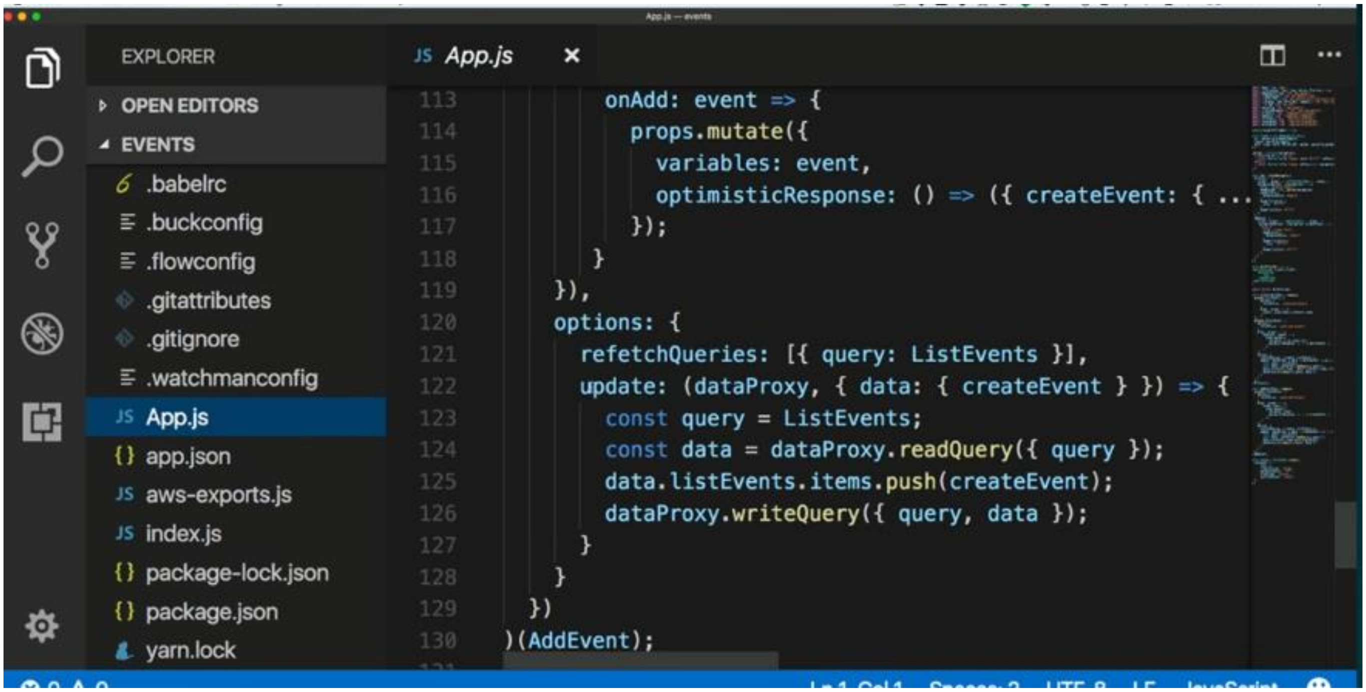
```
      props: (props) => ({
        events: props.data.listEvents.items,
      })
  }),
  graphql(DeleteEvent, {
    options:{
      fetchPolicy: 'cache-and-network'
    },
    props: (props) => ({
      onDelete: (event) => {
        props.mutate({
          variables: { id: event.id },
          optimisticResponse: () => ({ deleteEvent: { ...
        })
      }
    }),
    options: {
      refetchQueries: [{ query: ListEvents }],
      update: (dataProxy, { data: { deleteEvent: { id } } }
```



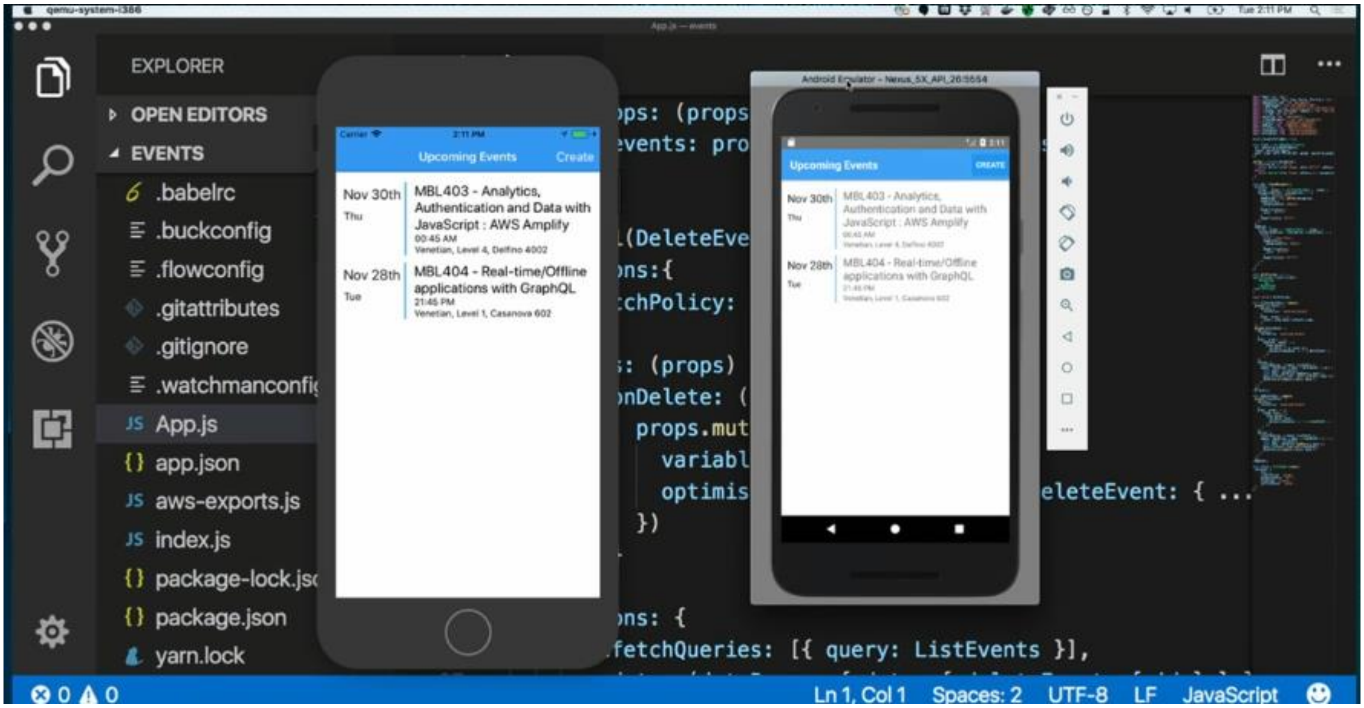
```
      refetchQueries: [{ query: ListEvents }],
      update: (dataProxy, { data: { deleteEvent: { id } } }
        const query = ListEvents;
        const data = dataProxy.readQuery({ query });
        data.listEvents.items = data.listEvents.items.filte
        dataProxy.writeQuery({ query, data });
      }
    }
  }),
)(AllEvents);

const AddEventData = compose(
  graphql(CreateEvent, {
    options:{
      fetchPolicy: 'cache-and-network'
    },
    props: (props) => ({
      onAdd: event => {
```



```
)(AllEvents);

const AddEventData = compose(
  graphql(CreateEvent, {
    options:{
      fetchPolicy: 'cache-and-network'
    },
    props: (props) => ({
        onAdd: event => {
          props.mutate({
            variables: event,
            optimisticResponse: () => ({ createEvent: { ...
          });
        }
    }),
    options: {
      refetchQueries: [{ query: ListEvents }],
      update: (dataProxy, { data: { createEvent } }) => {
        const query = ListEvents;
```

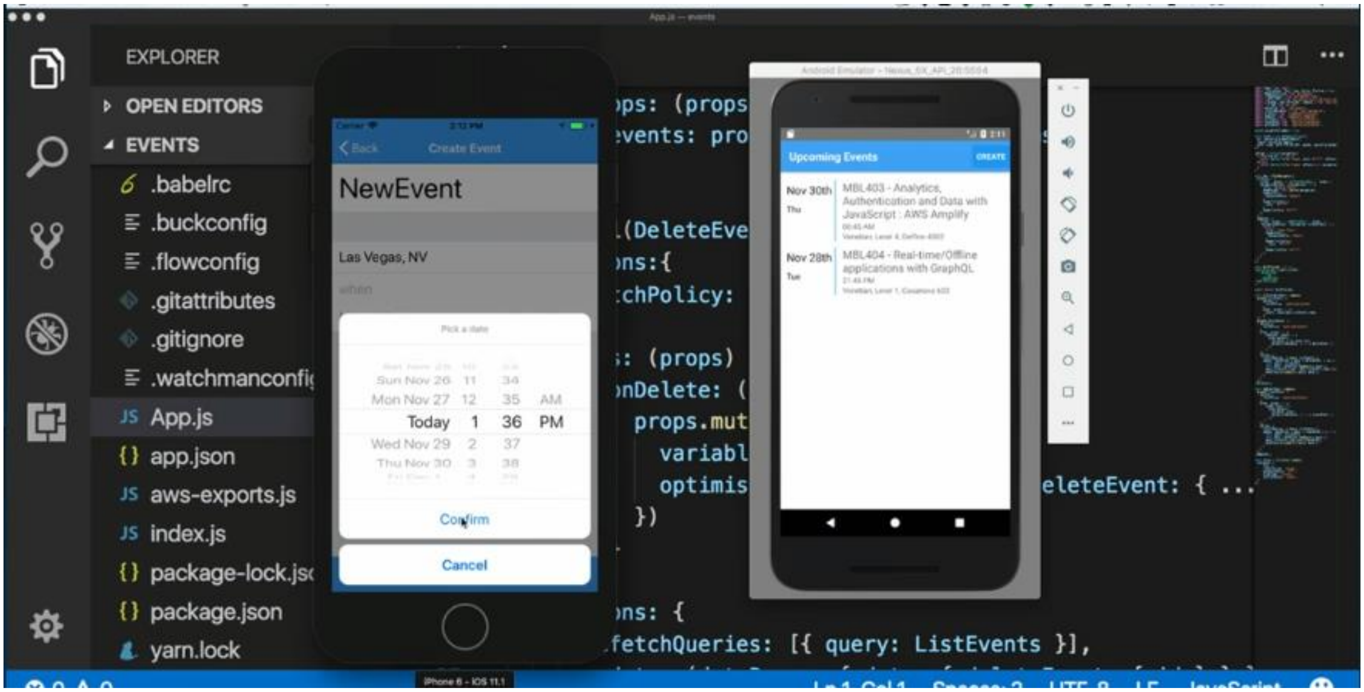


```
        onAdd: event => {
          props.mutate({
            variables: event,
            optimisticResponse: () => ({ createEvent: { ...
          });
        }
    }),
    options: {
      refetchQueries: [{ query: ListEvents }],
      update: (dataProxy, { data: { createEvent } }) => {
        const query = ListEvents;
        const data = dataProxy.readQuery({ query });
        data.listEvents.items.push(createEvent);
        dataProxy.writeQuery({ query, data });
      }
    }
  })
)(AddEvent);
```



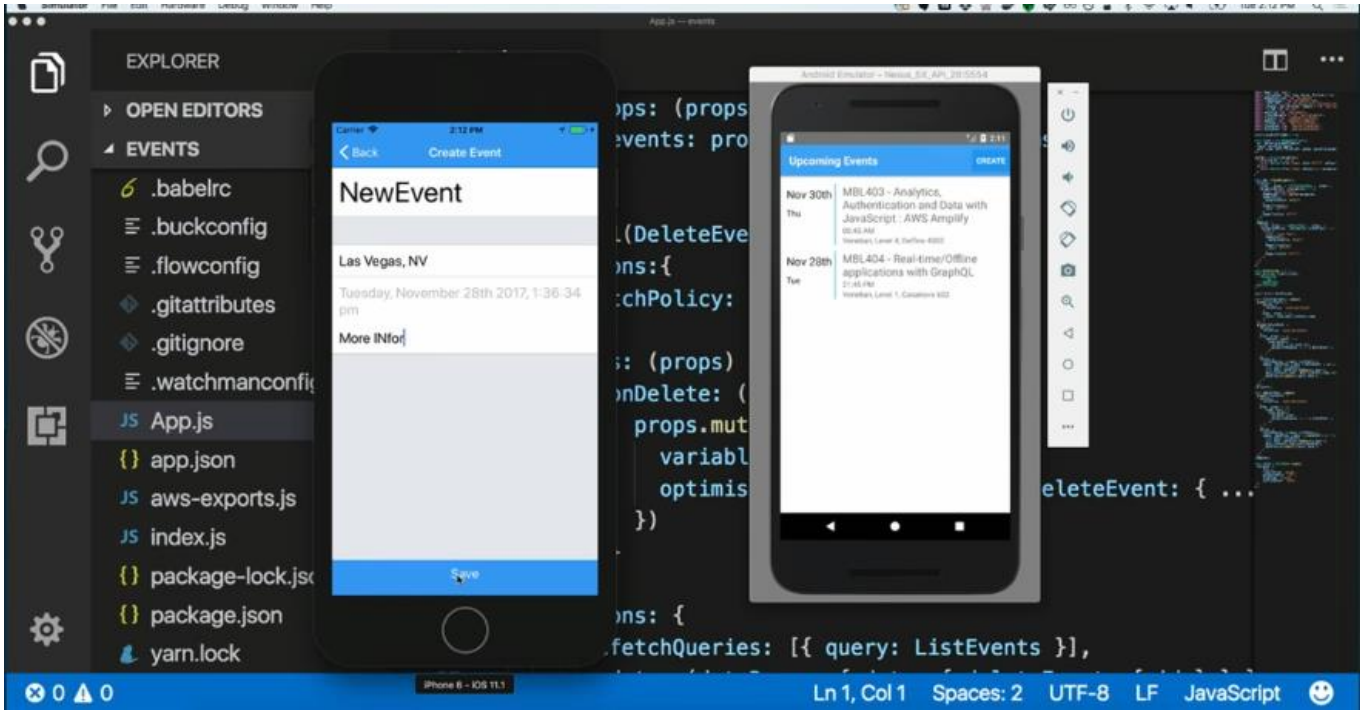
```
ps: (props
events: pro
(DeleteEve
ns:{
chPolicy:
: (props)
nDelete: (
props.mut
  variabl
  optimis
})
eleteEvent: { ...
ns: {
fetchQueries: [{ query: ListEvents }],
```

We have the app running in an emulator for both Android and IOS as above.



```
ps: (props
events: pro
(DeleteEve
ns:{
chPolicy:
: (props)
nDelete: (
props.mut
  variabl
  optimis
})
eleteEvent: { ...
ns: {
fetchQueries: [{ query: ListEvents }],
```

We click the Create Event button and start entering the event details



Back
Create Event

NewEvent

Las Vegas, NV

Tuesday, November 28th 2017, 1:36:34 pm

More INfor

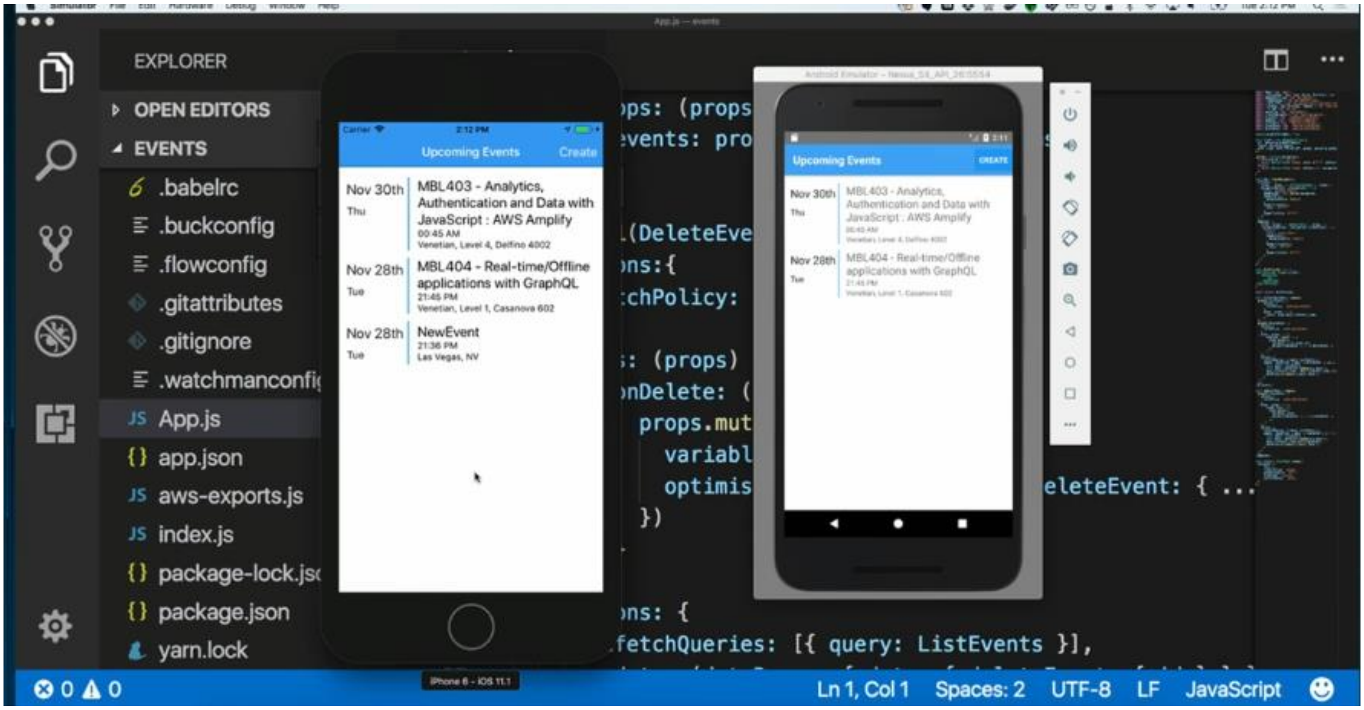
Save

Upcoming Events CREATE

Nov 30th Thu | MBL403 - Analytics, Authentication and Data with JavaScript : AWS Amplify 00:45 AM Venetian, Level 4, Delfino 4002

Nov 28th Tue | MBL404 - Real-time/Offline applications with GraphQL 21:45 PM Venetian, Level 1, Casanova 602

```
ps: (props
events: pro

(DeleteEve
ns:{
chPolicy:

: (props)
nDelete: (
  props.mut
    variabl
    optimis                                  eleteEvent: { ...
  })

ns: {
fetchQueries: [{ query: ListEvents }],
```



Upcoming Events Create

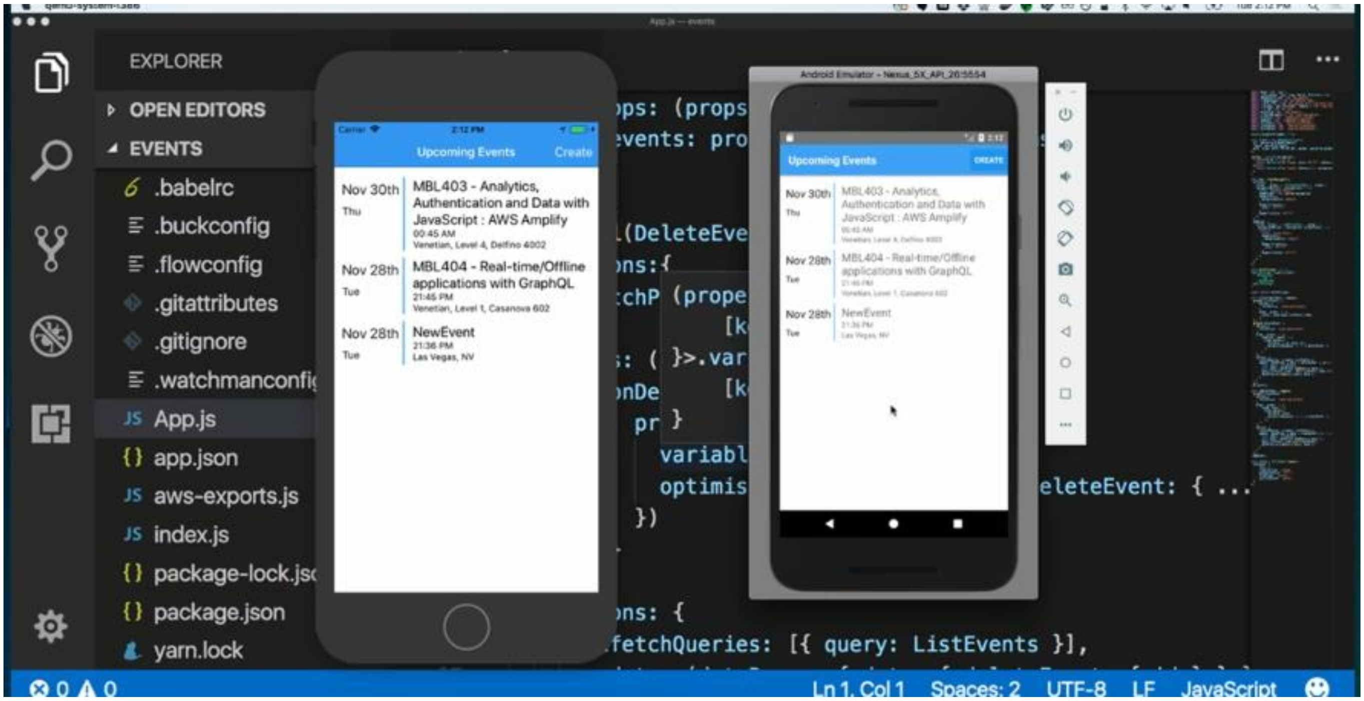
Nov 30th Thu | MBL403 - Analytics, Authentication and Data with JavaScript : AWS Amplify 00:45 AM Venetian, Level 4, Delfino 4002

Nov 28th Tue | MBL404 - Real-time/Offline applications with GraphQL 21:45 PM Venetian, Level 1, Casanova 602

Nov 28th Tue | NewEvent 21:36 PM Las Vegas, NV

```
ps: (props
events: pro

(DeleteEve
ns:{
chPolicy:

: (props)
nDelete: (
  props.mut
    variabl
    optimis                                  eleteEvent: { ...
  })

ns: {
fetchQueries: [{ query: ListEvents }],
```

As soon as we add the event, we see the view update as above on the left



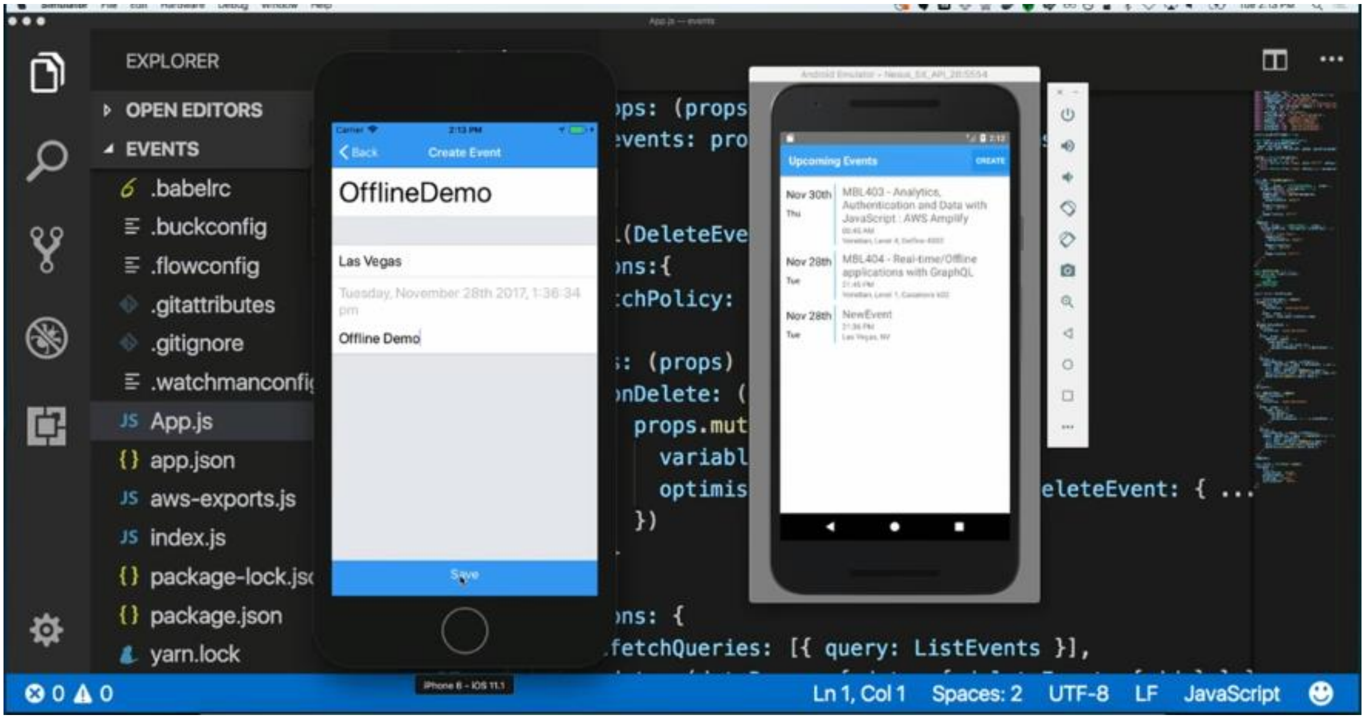
Nov 30th Thu — MBL403 - Analytics, Authentication and Data with JavaScript : AWS Amplify, 00:45 AM, Venetian, Level 4, Delfino 4002

Nov 28th Tue — MBL404 - Real-time/Offline applications with GraphQL, 21:45 PM, Venetian, Level 1, Casanova 602

Nov 28th Tue — NewEvent, 21:36 PM, Las Vegas, NV

```
ps: (props
vents: pro

(DeleteEve
ns:{
chP (prope
          [k
: ( }>.var
nDe       [k
 pr }
   variabl
   optimis                               eleteEvent: { ...
 })

ns: {
fetchQueries: [{ query: ListEvents }],
```

We can manually refresh the screen on the right to see the update as above. We can easily swap this out with a pull-to-refresh capability or wire up subscriptions for getting change events notification



OfflineDemo

Las Vegas

Tuesday, November 28th 2017, 1:36:34 pm

Offline Demo

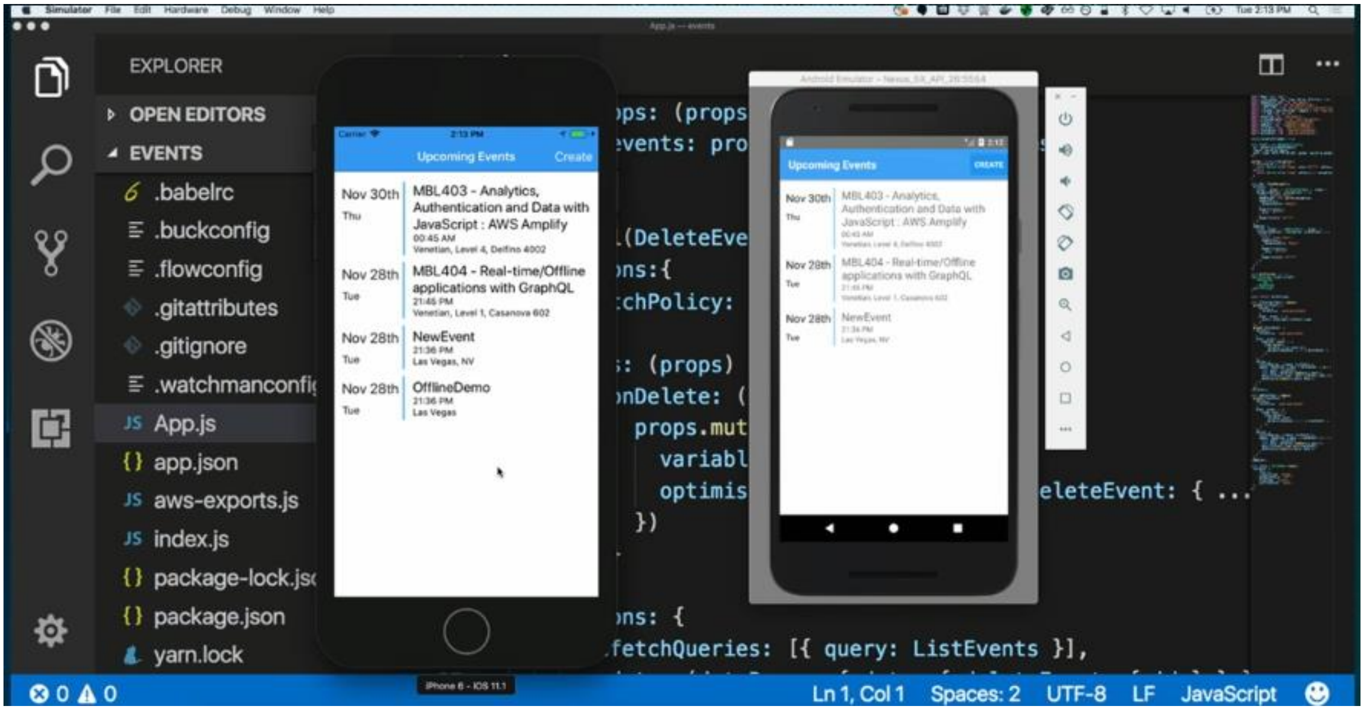
Nov 30th Thu — MBL403 - Analytics, Authentication and Data with JavaScript : AWS Amplify

Nov 28th Tue — MBL404 - Real-time/Offline applications with GraphQL

Nov 28th Tue — NewEvent, Las Vegas, NV

```
ps: (props
vents: pro

(DeleteEve
ns:{
chPolicy:

: (props)
nDelete: (
 props.mut
   variabl
   optimis                               eleteEvent: { ...
 })

ns: {
fetchQueries: [{ query: ListEvents }],
```

We go offline and create a new event

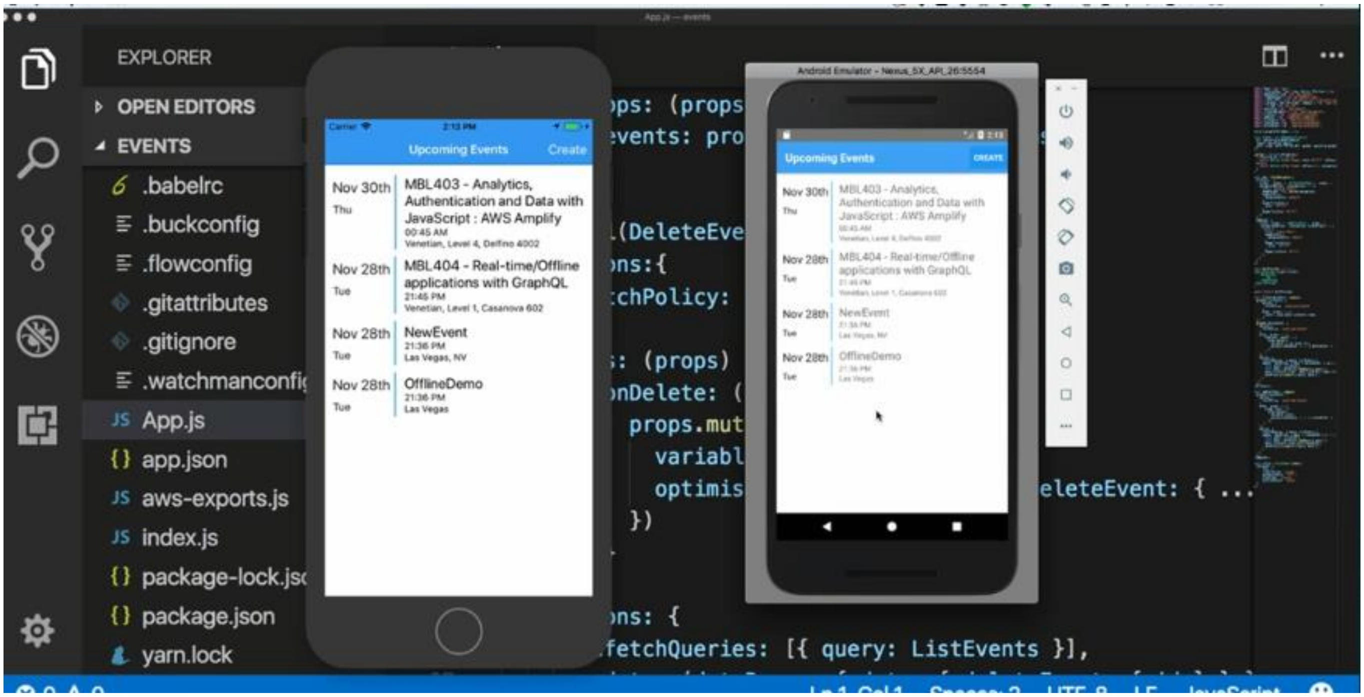


Upcoming Events Create

Nov 30th Thu	MBL403 - Analytics, Authentication and Data with JavaScript : AWS Amplify 00:45 AM Venetian, Level 4, Delfino 4002
Nov 28th Tue	MBL404 - Real-time/Offline applications with GraphQL 21:45 PM Venetian, Level 1, Casanova 602
Nov 28th Tue	NewEvent 21:36 PM Las Vegas, NV
Nov 28th Tue	OfflineDemo 21:36 PM Las Vegas

```
)ps: (props
events: pro

.(DeleteEve
)ns:{
:chPolicy:

:: (props)
)nDelete: (
  props.mut
    variabl
    optimis
  })
                                    eleteEvent: { ...

)ns: {
fetchQueries: [{ query: ListEvents }],
```

The view is now updated with the latest information on the left



Upcoming Events Create

Nov 30th Thu	MBL403 - Analytics, Authentication and Data with JavaScript : AWS Amplify 00:45 AM Venetian, Level 4, Delfino 4002
Nov 28th Tue	MBL404 - Real-time/Offline applications with GraphQL 21:45 PM Venetian, Level 1, Casanova 602
Nov 28th Tue	NewEvent 21:36 PM Las Vegas, NV
Nov 28th Tue	OfflineDemo 21:36 PM Las Vegas

```
)ps: (props
events: pro

.(DeleteEve
)ns:{
:chPolicy:

:: (props)
)nDelete: (
  props.mut
    variabl
    optimis
  })
                                    eleteEvent: { ...

)ns: {
fetchQueries: [{ query: ListEvents }],
```

We then go back online and refresh the right screen to see the entry that we made while offline show up too

Demo: Offline data with AppSync

Client experience and configuration

Offline is a write-through "Store"

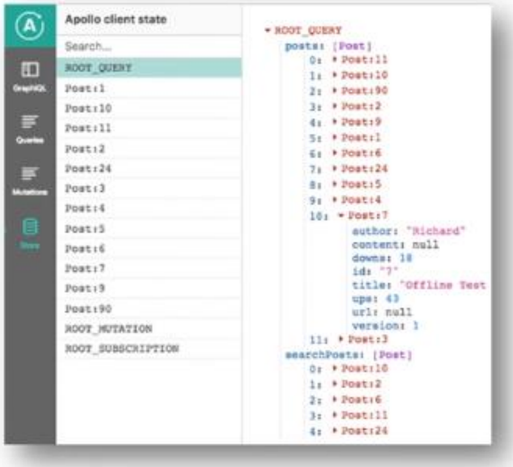
- Persistent storage mediums back the Apollo normalized cache
- Local Storage for web
- SQLite on hybrid/native platforms

SQLite database can be preloaded

- Hydrate after installing from AppStore

Offline client can be configured

- Wifi only vs. wifi & carrier



Apollo client state

Search...

```
ROOT_QUERY
Post:1
Post:10
Post:11
Post:2
Post:24
Post:3
Post:4
Post:5
Post:6
Post:7
Post:9
Post:90
ROOT_MUTATION
ROOT_SUBSCRIPTION
```

```
▾ ROOT_QUERY
   posts: [Post]
      0: ▸ Post:11
      1: ▸ Post:10
      2: ▸ Post:90
      3: ▸ Post:2
      4: ▸ Post:9
      5: ▸ Post:1
      6: ▸ Post:6
      7: ▸ Post:24
      8: ▸ Post:5
      9: ▸ Post:4
      10: ▾ Post:7
            author: "Richard"
            content: null
            downs: 18
            id: "7"
            title: "Offline Test
            ups: 43
            url: null
            version: 1
      11: ▸ Post:3
   searchPosts: [Post]
      0: ▸ Post:10
      1: ▸ Post:2
      2: ▸ Post:6
      3: ▸ Post:11
      4: ▸ Post:24
```

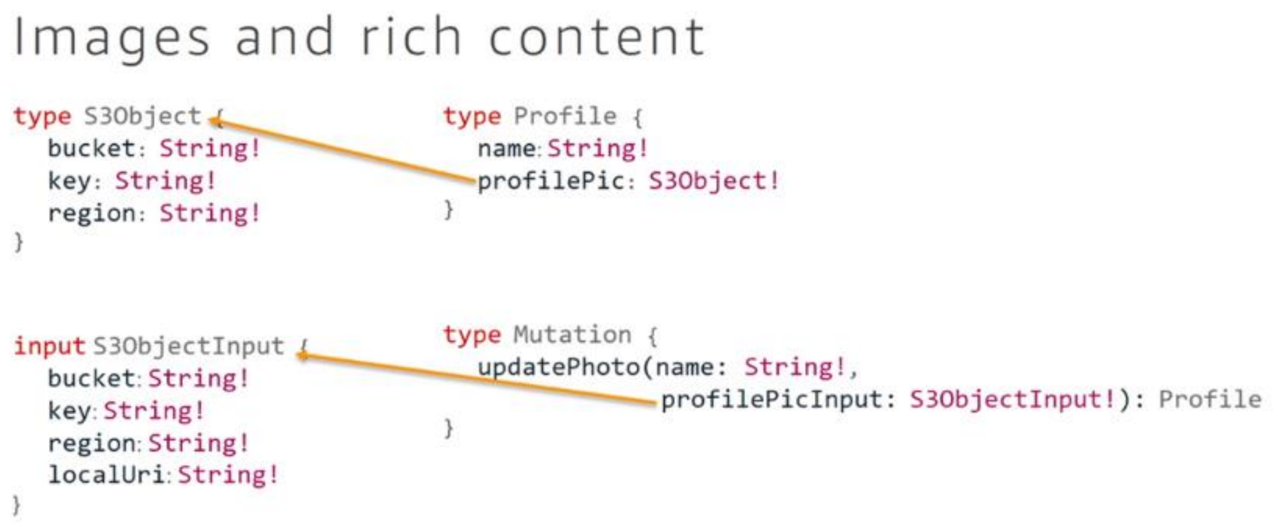
Images and rich content

```
type S3Object {
   bucket: String!
   key: String!
   region: String!
}
```

```
type Profile {
   name: String!
   profilePic: S3Object!
}
```

```
input S3ObjectInput {
   bucket: String!
   key: String!
   region: String!
   localUri: String!
}
```

```
type Mutation {
   updatePhoto(name: String!,
               profilePicInput: S3ObjectInput!): Profile
}
```



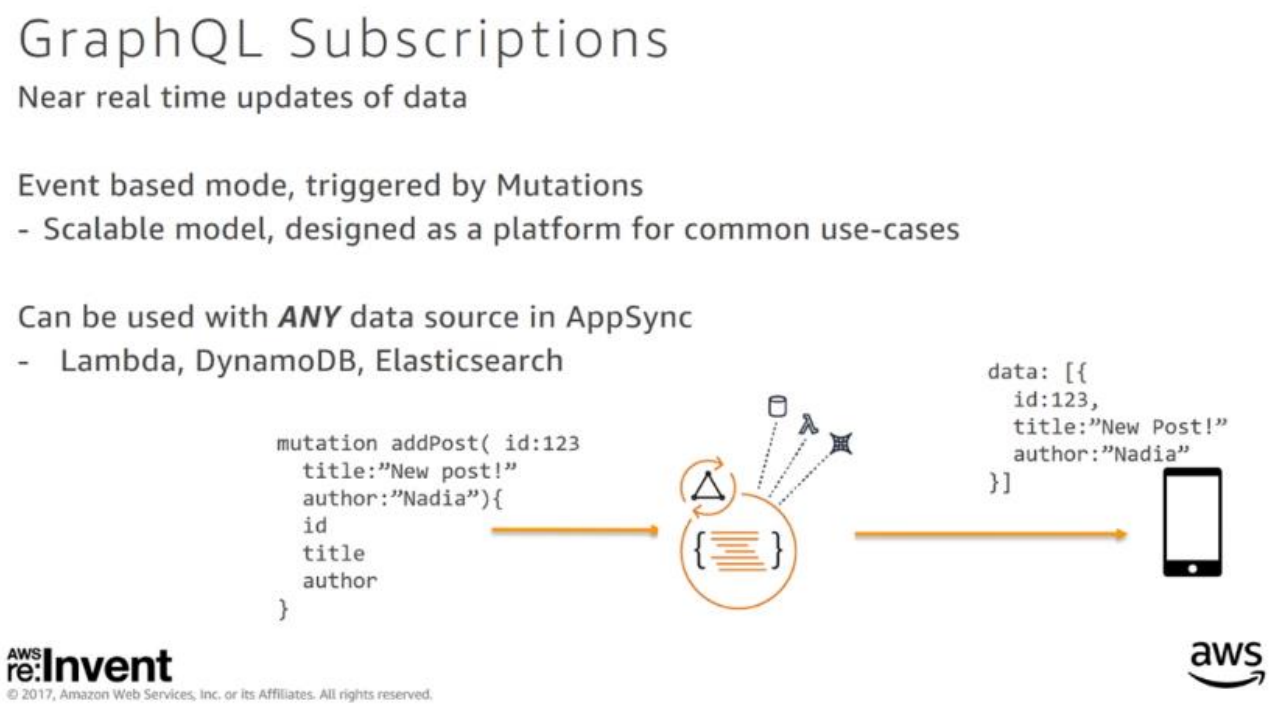
Images and rich content

```
type S3Object {
  bucket: String!
  key: String!
  region: String!
}
```

```
type Profile {
  name: String!
  profilePic: S3Object!
}
```

```
input S3ObjectInput {
  bucket: String!
  key: String!
  region: String!
  localUri: String!
}
```

```
type Mutation {
  updatePhoto(name: String!,
              profilePicInput: S3ObjectInput!): Profile
}
```



GraphQL Subscriptions

Near real time updates of data

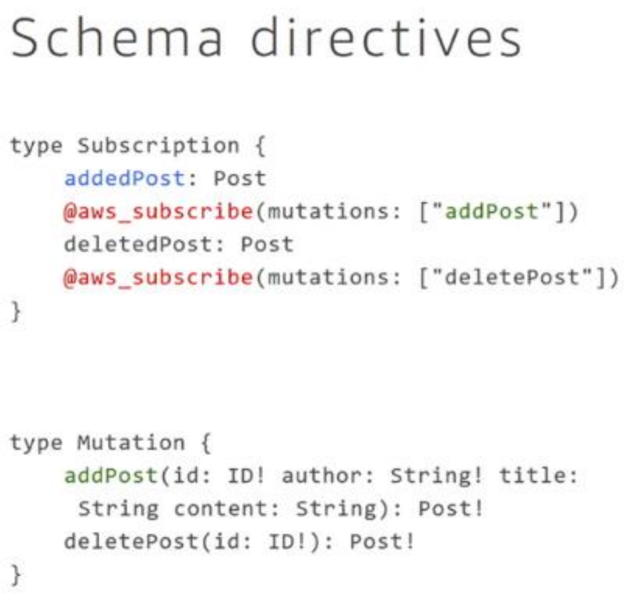
Event based mode, triggered by Mutations
- Scalable model, designed as a platform for common use-cases

Can be used with ***ANY*** data source in AppSync
- Lambda, DynamoDB, Elasticsearch

```
mutation addPost( id:123
  title:”New post!”
  author:”Nadia”){
  id
  title
  author
}
```

```
data: [{
  id:123,
  title:”New Post!”
  author:”Nadia”
}]
```

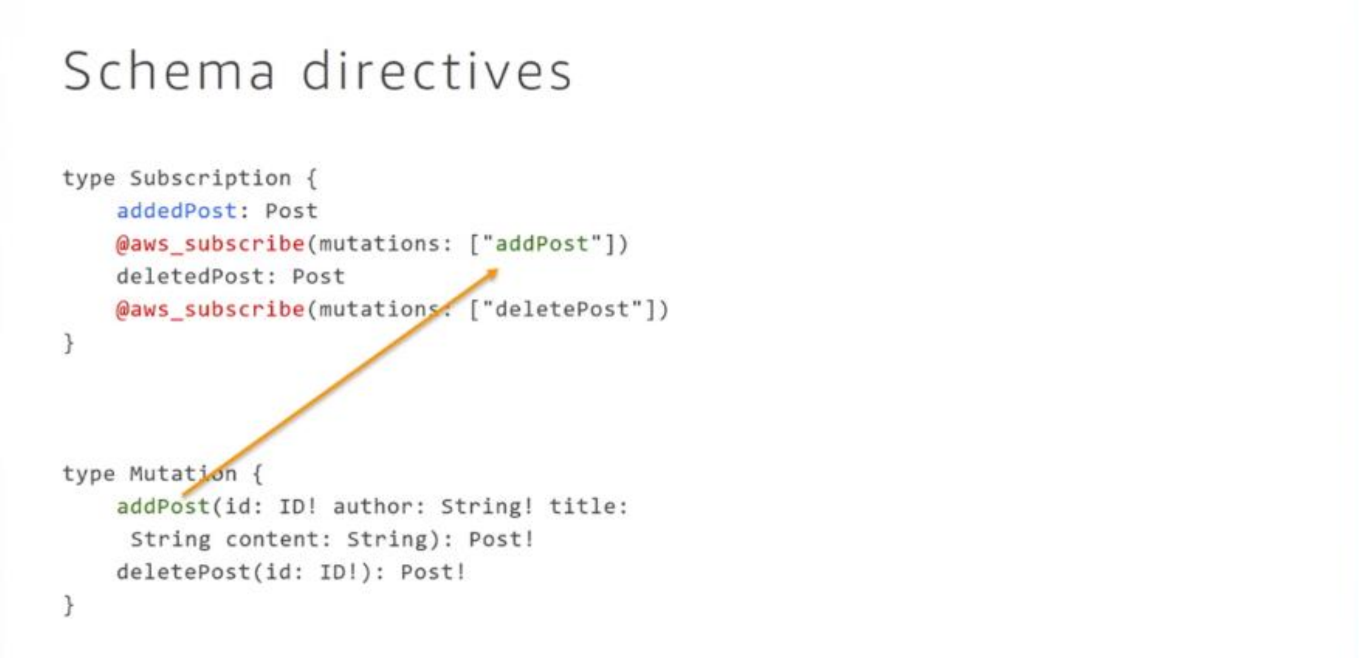
In our events-based model implementation, subscriptions are triggered in response to mutations on the system. This is for both scalability and use cases scenarios.



Schema directives

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


type Mutation {
    addPost(id: ID! author: String! title:
     String content: String): Post!
    deletePost(id: ID!): Post!
}
```

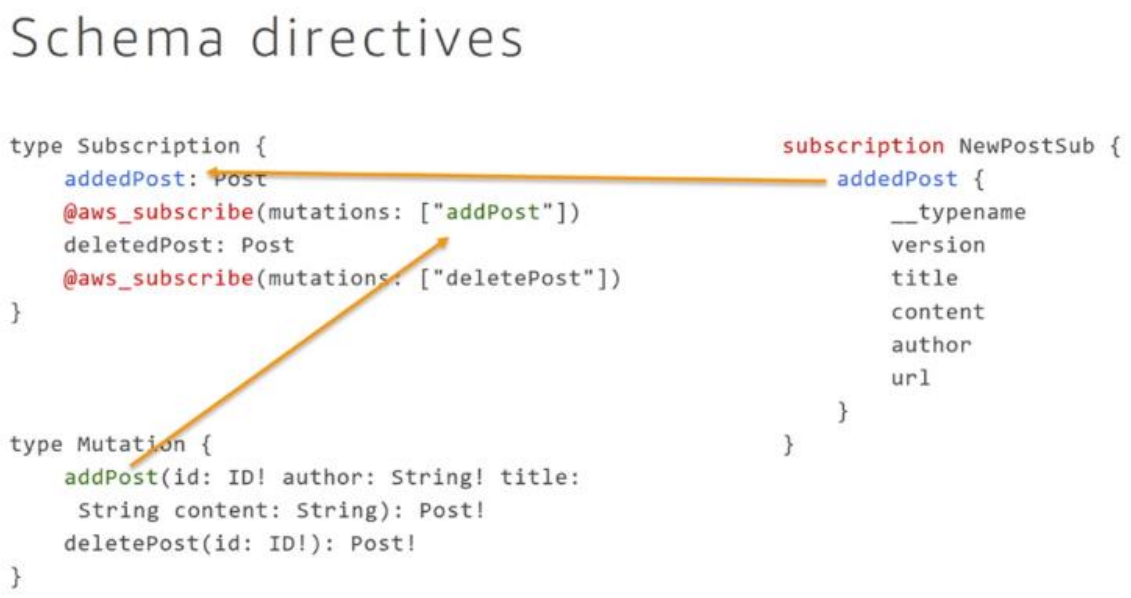
Let us see how the schema is configured for real time app updates in a demo. The data that comes down to your subscription channel is basically triggered off a mutation. Above we have a subscription type called ***addedPost*** which has a return type of Post, we also have a type called Mutation that has a return type of a nullable Post (Post!).



Schema directives

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


type Mutation {
    addPost(id: ID! author: String! title:
     String content: String): Post!
    deletePost(id: ID!): Post!
}
```

We can then use the ***@aws_subscribe directive*** to add a list of mutations like addPost and deletePost above. We can also have N mutations that trigger a single subscription, or have a single mutation that triggers multiple subscriptions as long as the underlying return type is the same.

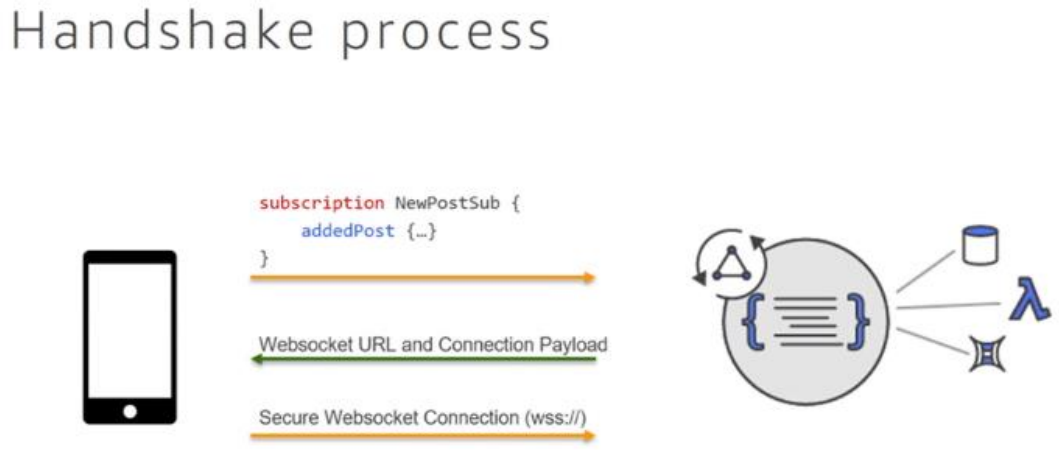


Schema directives

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


type Mutation {
    addPost(id: ID! author: String! title:
     String content: String): Post!
    deletePost(id: ID!): Post!
}
```

```
subscription NewPostSub {
    addedPost {
        __typename
        version
        title
        content
        author
        url
    }
}
```

When we make the subscription call like NewPostSub above, it is going to trigger the addedPost function and start opening up an MQTT channel and start getting new information using that topic/channel



Handshake process

```
subscription NewPostSub {
    addedPost {…}
}
```

Websocket URL and Connection Payload

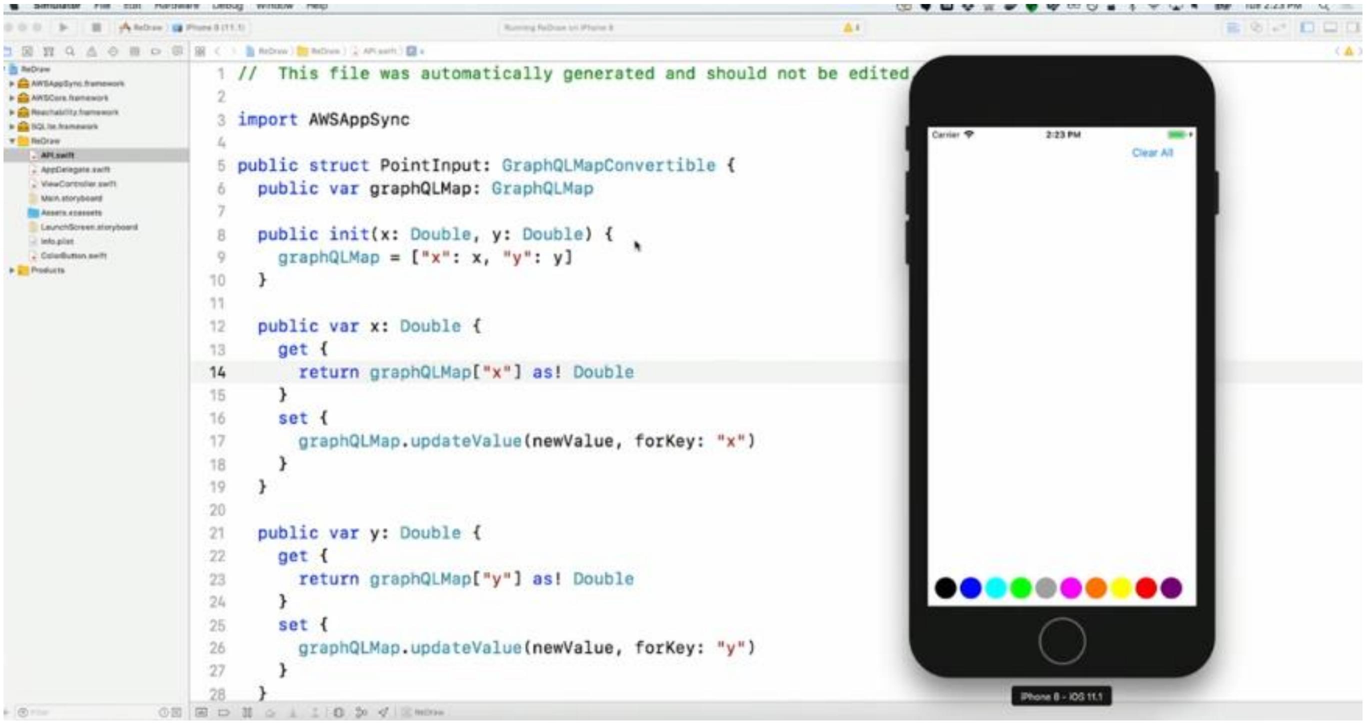
Secure Websocket Connection (wss://)

We have a handshake process behind the scenes that is a 3-step process when using a 3rd party client. Whenever a subscription is requested from the client, we return back a websocket URL connection and a list of topics that the client can subscribe to. The client then makes an MQTT connection call with the websocket URL and if this succeeds, the client can then subscribe to the list of topics that was returned back earlier as part of the subscribe call. When using the AppSync client SDK, this handshake process is automatically done in the background.

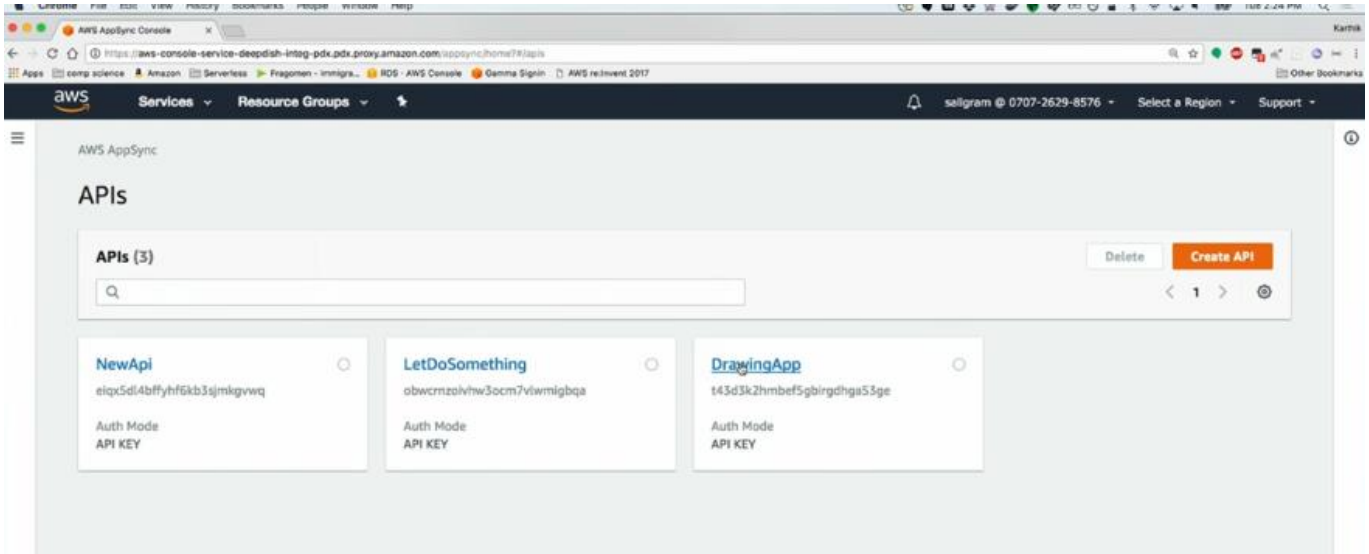
Real time UI updates

```
const AllPostsWithData = compose(
  graphql(AllPostsQuery, { options: { fetchPolicy: 'cache-and-network‘ },
    props: (props) => ({
      posts: props.data.posts,
      subscribeToNewPosts: params => {
          props.data.subscribeToMore({
              document: NewPostsSubscription,
              updateQuery: (prev, { subscriptionData: { newPost } }) => ({
                 ...prev,
                 posts: [newPost, ...prev.posts.filter(post => post.id !== newPost.id)]
              })
            });
        });
    …//more code
```

Demo: Real time data with AppSync



```
//  This file was automatically generated and should not be edited

import AWSAppSync

public struct PointInput: GraphQLMapConvertible {
  public var graphQLMap: GraphQLMap

  public init(x: Double, y: Double) {
    graphQLMap = ["x": x, "y": y]
  }

  public var x: Double {
    get {
      return graphQLMap["x"] as! Double
    }
    set {
      graphQLMap.updateValue(newValue, forKey: "x")
    }
  }

  public var y: Double {
    get {
      return graphQLMap["y"] as! Double
    }
    set {
      graphQLMap.updateValue(newValue, forKey: "y")
    }
  }
```



AWS AppSync

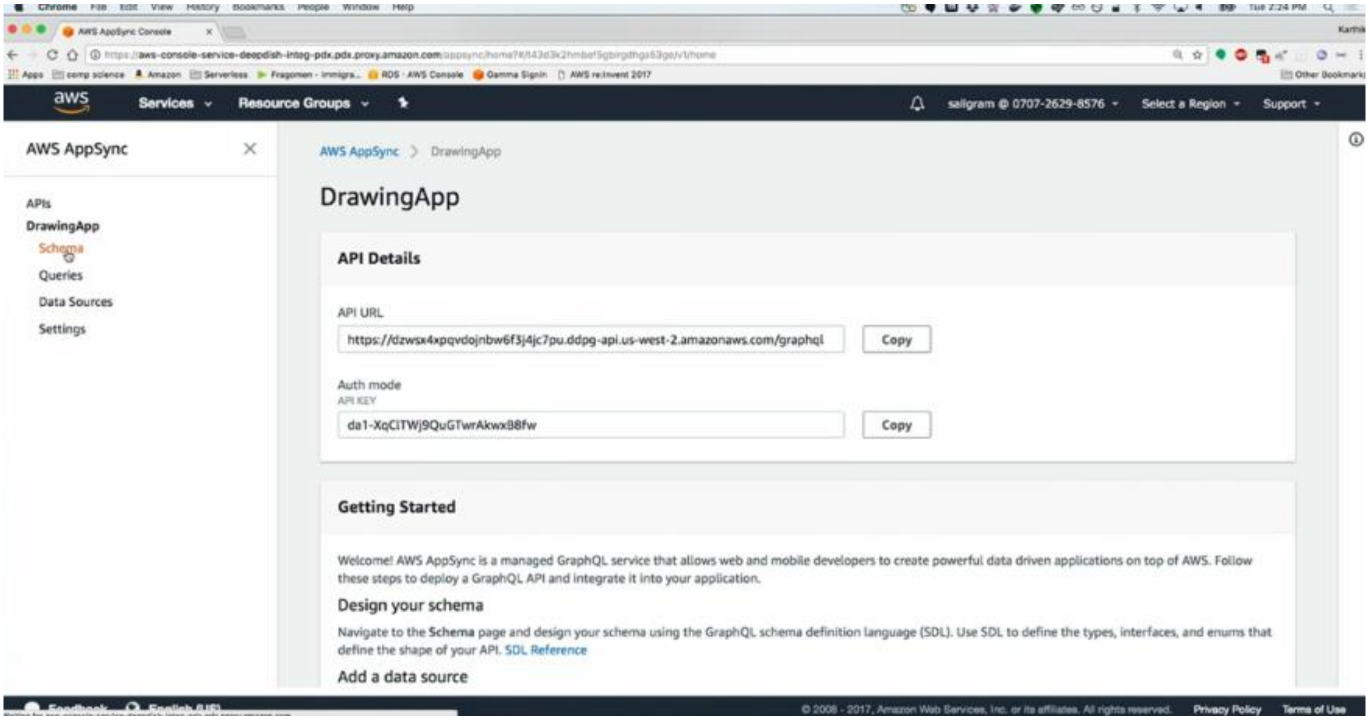
APIs

APIs (3)

NewApi
eiqx5dl4bffyhf6kb3sjmkgvwq
Auth Mode
API KEY

LetDoSomething
obwcrnzoivhw3ocm7viwmigbqa
Auth Mode
API KEY

DrawingApp
t43d3k2hmbef5gbirgdhga53ge
Auth Mode
API KEY



AWS AppSync > DrawingApp

DrawingApp

API Details

API URL
https://dzwsx4xpqvdojnbw6f3j4jc7pu.ddpg-api.us-west-2.amazonaws.com/graphql

Auth mode
API KEY
da1-XqCiTWj9QuGTwrAkwx88fw

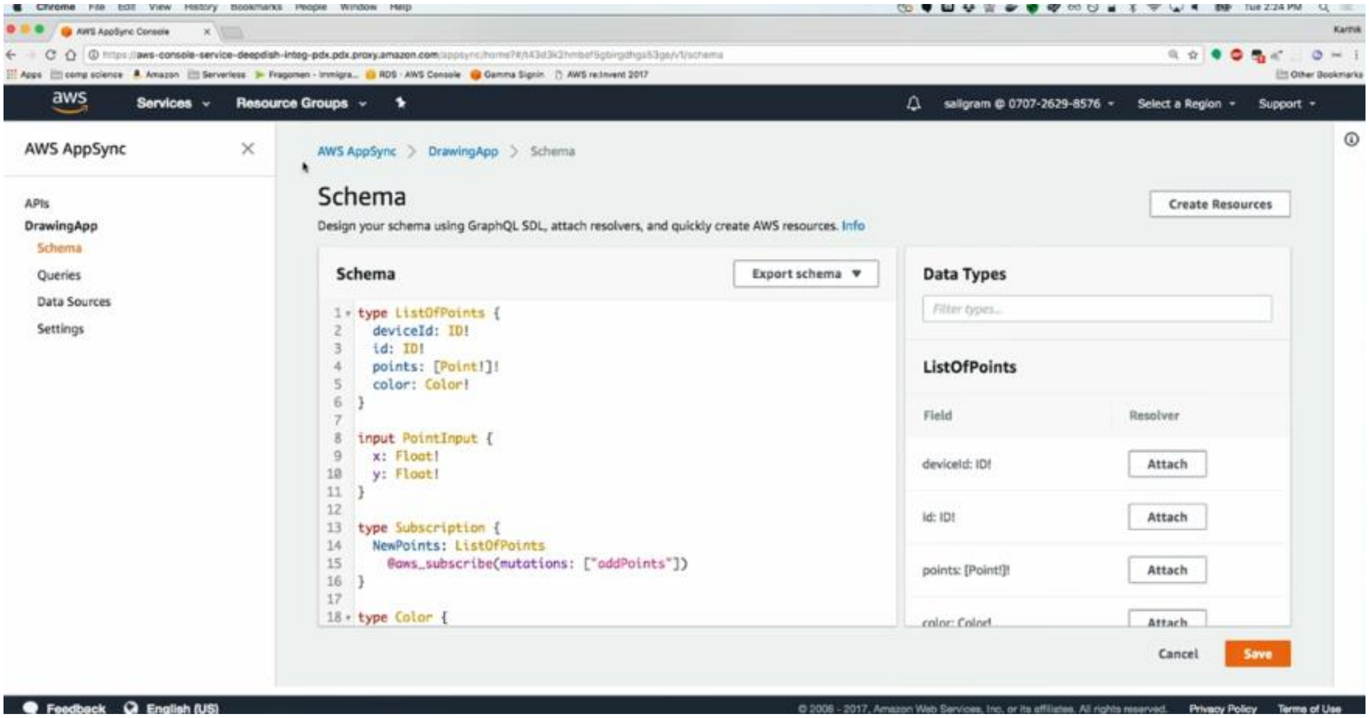
Getting Started

Welcome! AWS AppSync is a managed GraphQL service that allows web and mobile developers to create powerful data driven applications on top of AWS. Follow these steps to deploy a GraphQL API and integrate it into your application.

Design your schema

Navigate to the **Schema** page and design your schema using the GraphQL schema definition language (SDL). Use SDL to define the types, interfaces, and enums that define the shape of your API. SDL Reference

Add a data source



AWS AppSync > DrawingApp > Schema

Schema

Design your schema using GraphQL SDL, attach resolvers, and quickly create AWS resources. Info

Schema

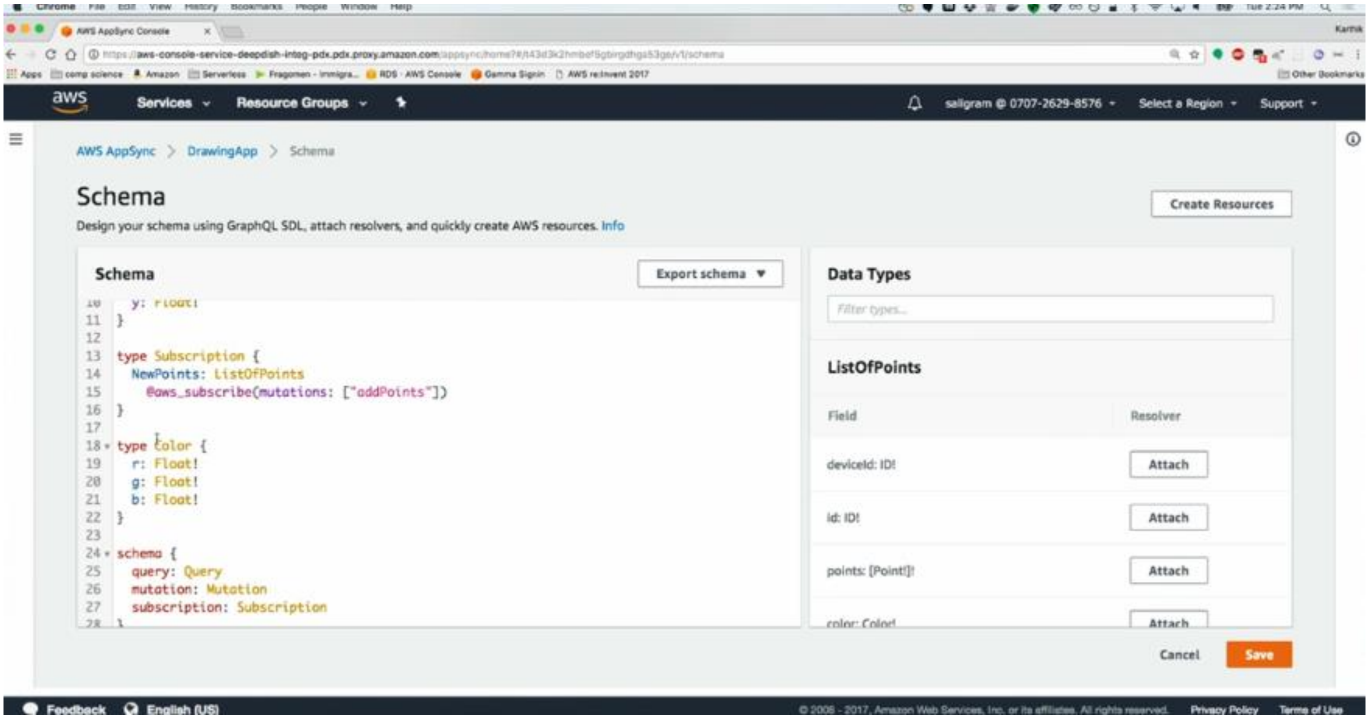
```
type ListOfPoints {
  deviceId: ID!
  id: ID!
  points: [Point!]!
  color: Color!
}

input PointInput {
  x: Float!
  y: Float!
}

type Subscription {
  NewPoints: ListOfPoints
    @aws_subscribe(mutations: ["addPoints"])
}

type Color {
```

Data Types

ListOfPoints

Field	Resolver
deviceId: ID!	Attach
id: ID!	Attach
points: [Point!]!	Attach
color: Color!	Attach

This is what our ***schema*** currently looks like, we have a ***type*** called ***Point*** with the x, y, z coordinates as attributes.



AWS AppSync > DrawingApp > Schema

Schema

Design your schema using GraphQL SDL, attach resolvers, and quickly create AWS resources. Info

Schema

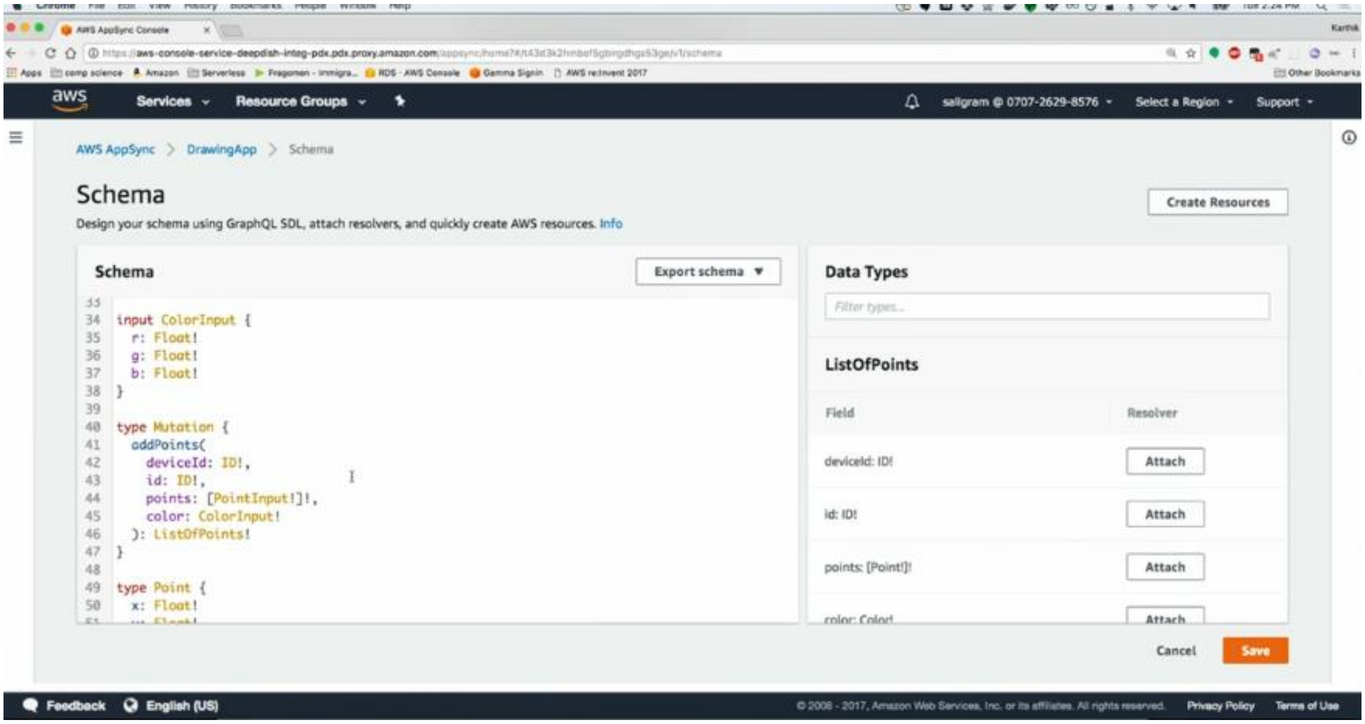
```
  y: Float!
}

type Subscription {
  NewPoints: ListOfPoints
    @aws_subscribe(mutations: ["addPoints"])
}

type Color {
  r: Float!
  g: Float!
  b: Float!
}

schema {
  query: Query
  mutation: Mutation
  subscription: Subscription
}
```

Data Types

ListOfPoints

Field	Resolver
deviceId: ID!	Attach
id: ID!	Attach
points: [Point!]!	Attach
color: Color!	Attach

We also have a ***type*** called ***Color*** with RGB values as above.


aws
Services
Resource Groups
saligram @ 0707-2629-8576
Select a Region
Support
AWS AppSync > DrawingApp > Schema
Create Resources
Export schema
Cancel
Save
Feedback
English (US)
© 2008 - 2017, Amazon Web Services, Inc. or its affiliates. All rights reserved.
Privacy Policy
Terms of Use

Schema

Design your schema using GraphQL SDL, attach resolvers, and quickly create AWS resources. Info

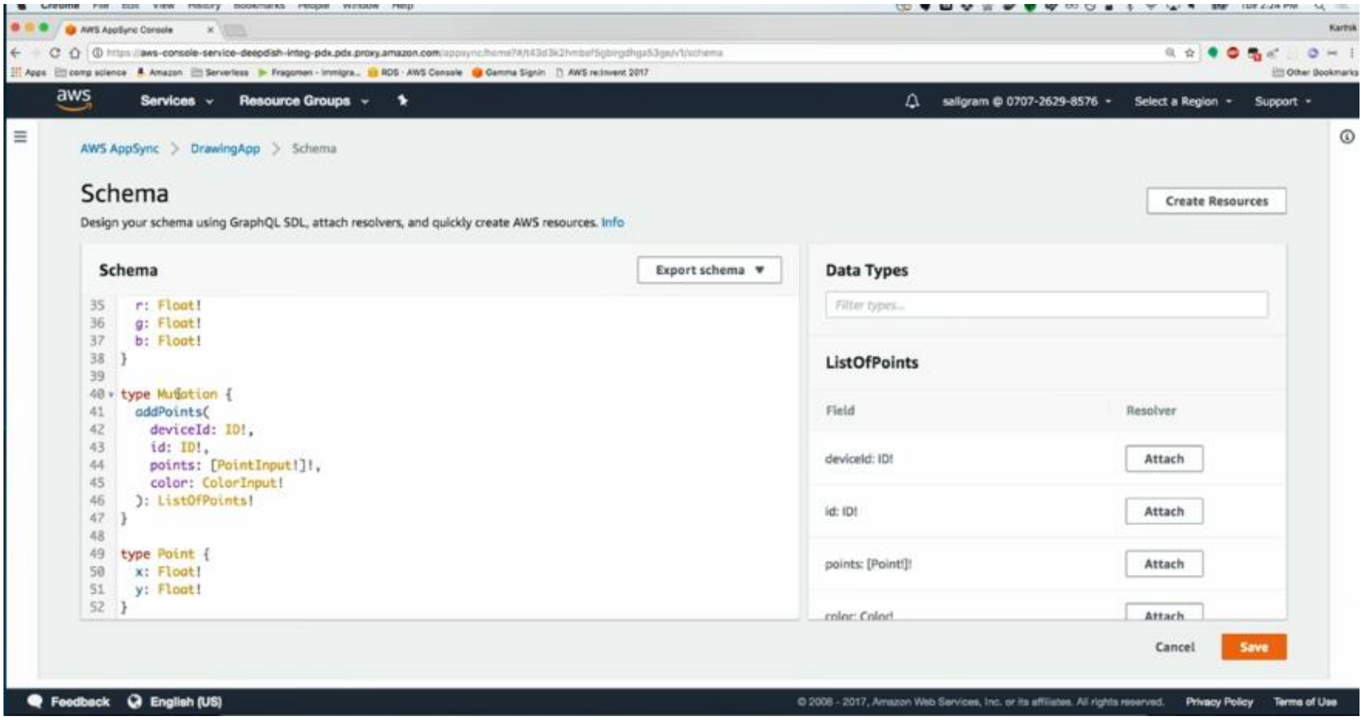
Schema

```
input ColorInput {
  r: Float!
  g: Float!
  b: Float!
}

type Mutation {
  addPoints(
    deviceId: ID!,
    id: ID!,
    points: [PointInput!]!,
    color: ColorInput!
  ): ListOfPoints!
}

type Point {
  x: Float!
```

Data Types

ListOfPoints

Field	Resolver
deviceId: ID!	Attach
id: ID!	Attach
points: [Point!]!	Attach
color: Color!	Attach


aws
Services
Resource Groups
saligram @ 0707-2629-8576
Select a Region
Support
AWS AppSync > DrawingApp > Schema
Create Resources
Export schema
Cancel
Save
Feedback
English (US)
© 2008 - 2017, Amazon Web Services, Inc. or its affiliates. All rights reserved.
Privacy Policy
Terms of Use

Schema

Design your schema using GraphQL SDL, attach resolvers, and quickly create AWS resources. Info

Schema

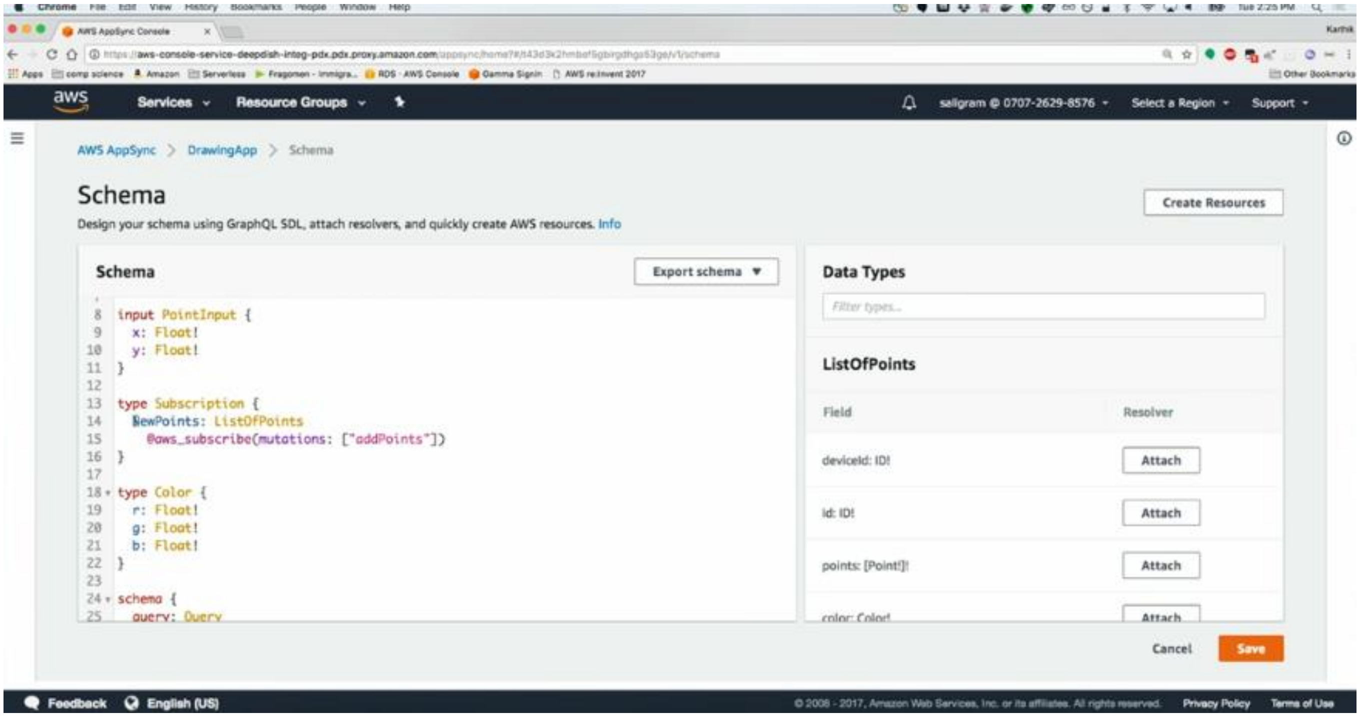
```
  r: Float!
  g: Float!
  b: Float!
}

type Mutation {
  addPoints(
    deviceId: ID!,
    id: ID!,
    points: [PointInput!]!,
    color: ColorInput!
  ): ListOfPoints!
}

type Point {
  x: Float!
  y: Float!
}
```

Data Types

ListOfPoints

Field	Resolver
deviceId: ID!	Attach
id: ID!	Attach
points: [Point!]!	Attach
color: Color!	Attach

We then have a ***Mutation*** called ***addPoints*** that takes in a ***deviceId***, ***id***, ***points***, and ***color***, it returns back a ***nullable ListOfPoints*** type.

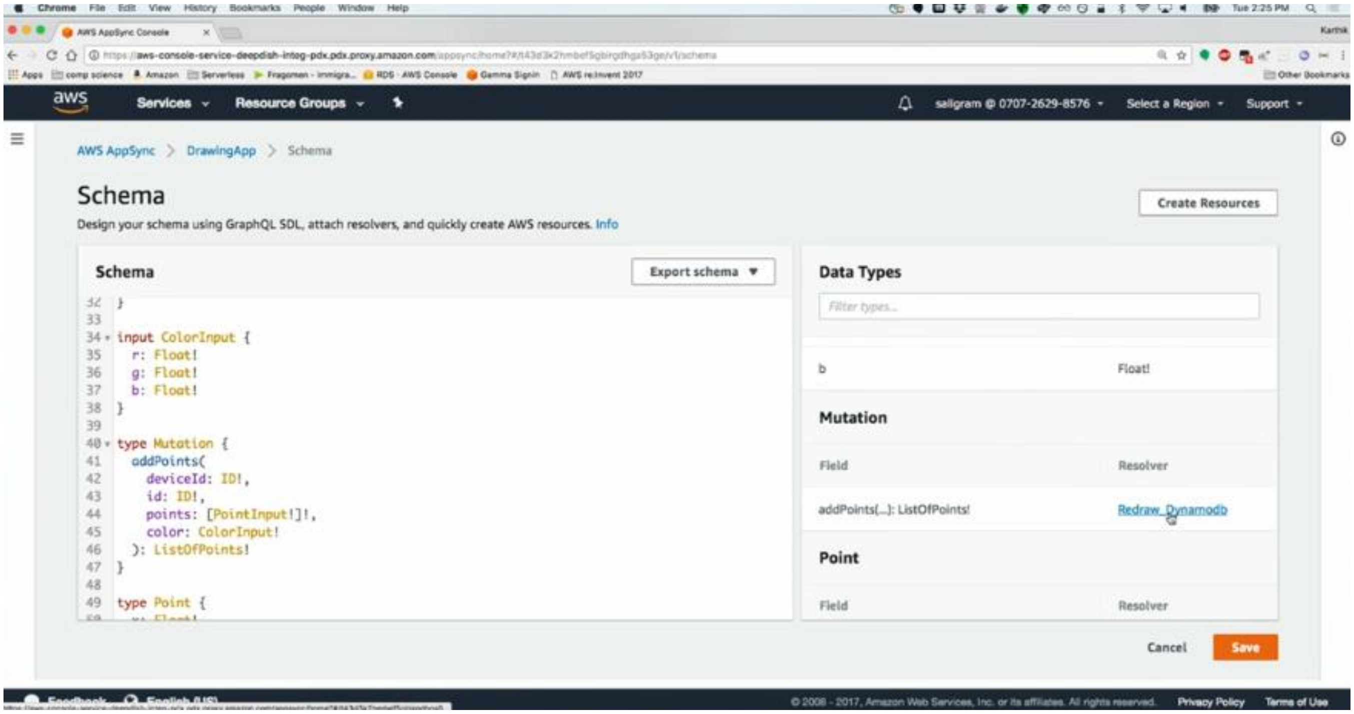

AWS AppSync > DrawingApp > Schema
Schema
Design your schema using GraphQL SDL, attach resolvers, and quickly create AWS resources. Info
Create Resources
Export schema
Data Types
Filter types...
Cancel
Save

```
8  input PointInput {
9    x: Float!
10   y: Float!
11 }
12
13 type Subscription {
14   NewPoints: ListOfPoints
15     @aws_subscribe(mutations: ["addPoints"])
16 }
17
18 type Color {
19   r: Float!
20   g: Float!
21   b: Float!
22 }
23
24 schema {
25   query: Query
```

ListOfPoints

Field	Resolver
deviceId: ID!	Attach
id: ID!	Attach
points: [Point!]!	Attach
color: Color!	Attach

We also have a ***Subscription*** called ***NewPoints*** that returns a ListOfPoints type, where we have an @aws_subscribe directive with mutation pointing to the ***addPoints***, which is the name of a filed in our ***Mutation type*** above on line 44


AWS AppSync > DrawingApp > Schema
Schema
Design your schema using GraphQL SDL, attach resolvers, and quickly create AWS resources. Info
Create Resources
Export schema
Data Types
Filter types...
Cancel
Save

```
32 }
33
34 input ColorInput {
35   r: Float!
36   g: Float!
37   b: Float!
38 }
39
40 type Mutation {
41   addPoints(
42     deviceId: ID!,
43     id: ID!,
44     points: [PointInput!]!,
45     color: ColorInput!
46   ): ListOfPoints!
47 }
48
49 type Point {
```

b	Float!

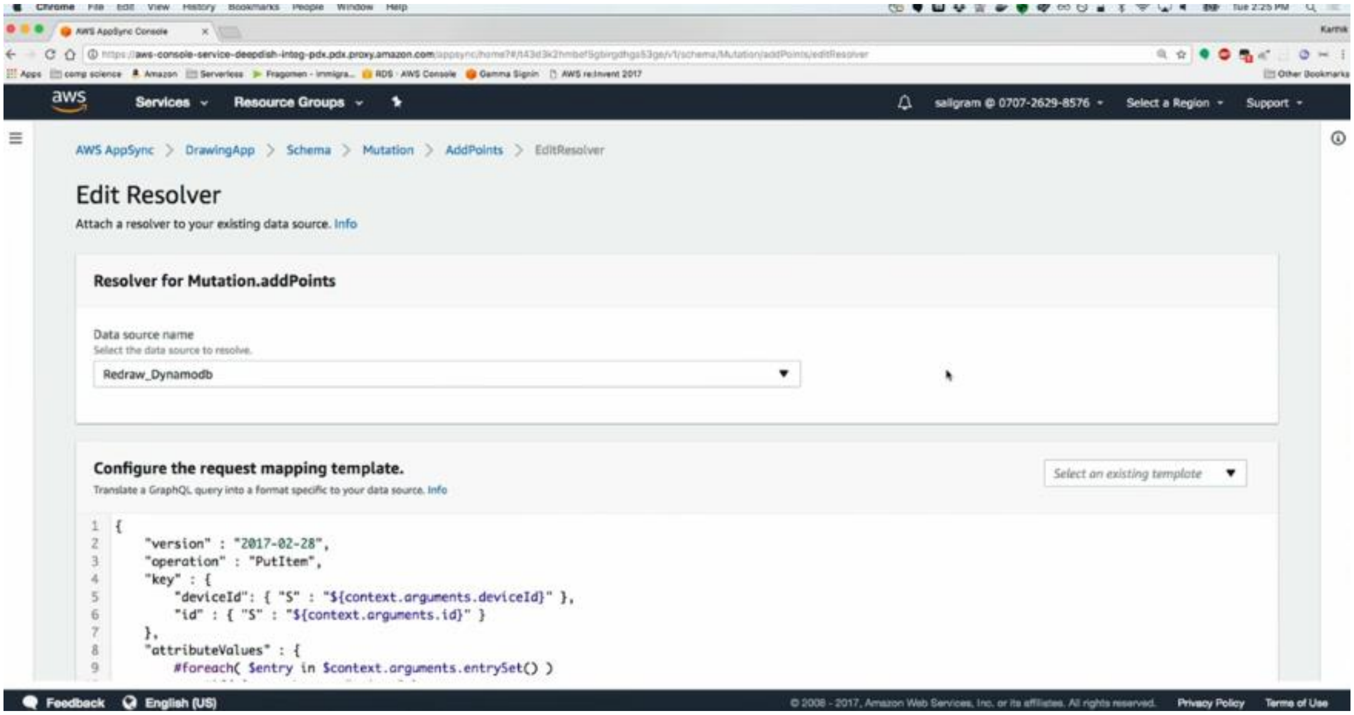
Mutation

Field	Resolver
addPoints(...): ListOfPoints!	Redraw_Dynamodb

Point

Field	Resolver

We also have this wired up to our DynamoDB table


Chrome File Edit View History Bookmarks People Window Help
AWS AppSync Console
aws
Services
Resource Groups
saligram @ 0707-2629-8576
Select a Region
Support
Feedback
English (US)
© 2008 - 2017, Amazon Web Services, Inc. or its affiliates. All rights reserved.
Privacy Policy
Terms of Use

AWS AppSync > DrawingApp > Schema > Mutation > AddPoints > EditResolver

Edit Resolver

Attach a resolver to your existing data source. Info

Resolver for Mutation.addPoints

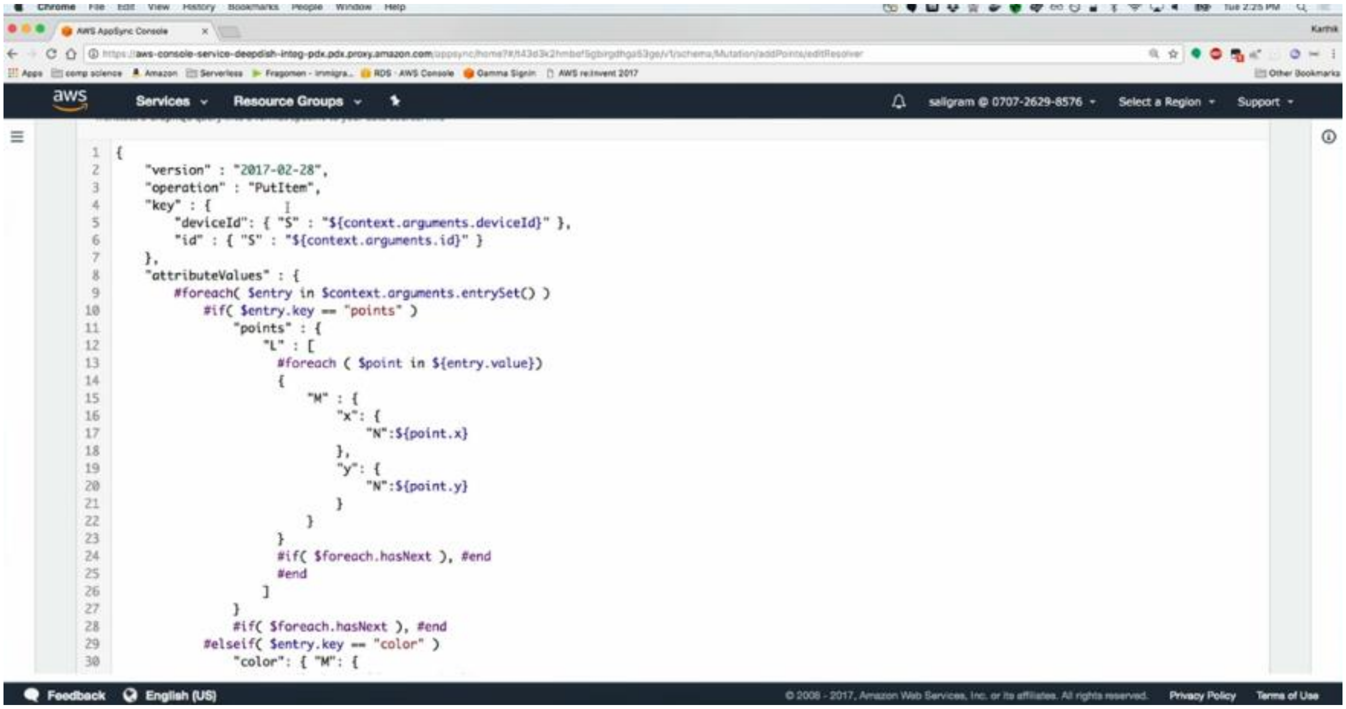
Data source name
Select the data source to resolve.
Redraw_Dynamodb

Configure the request mapping template.

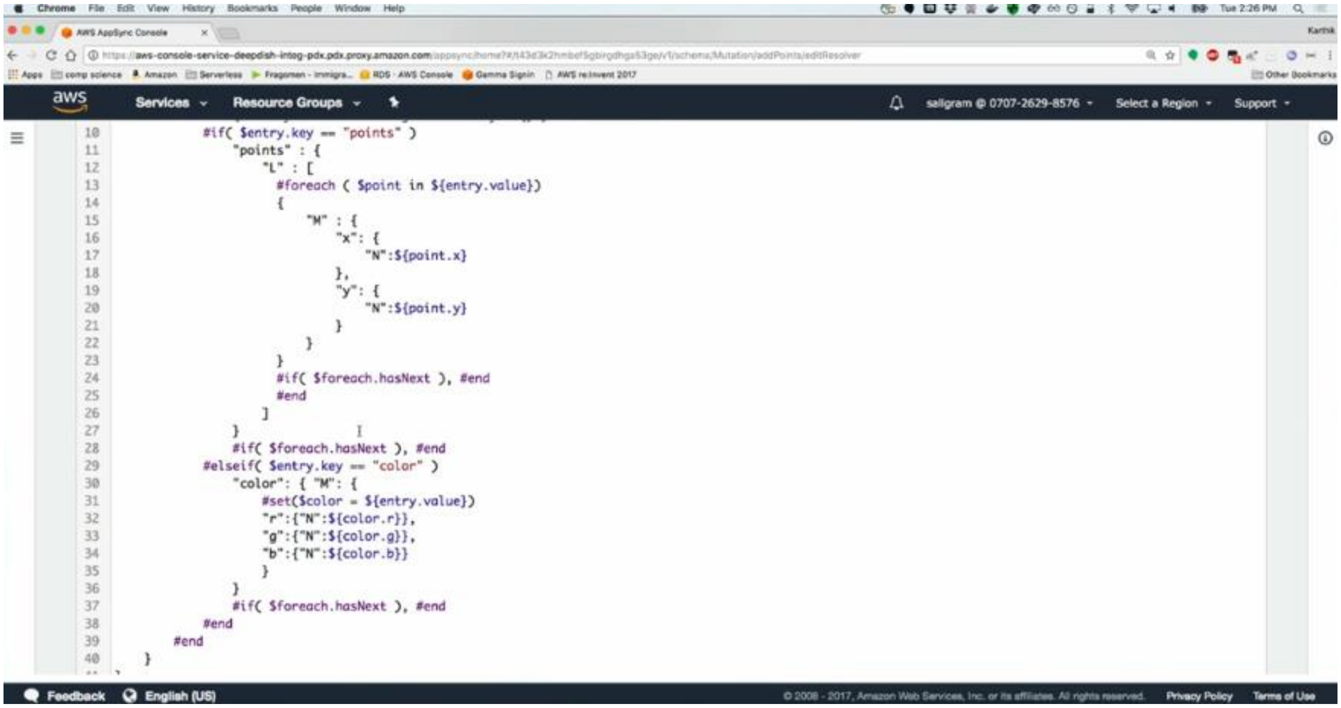
Translate a GraphQL query into a format specific to your data source. Info

Select an existing template

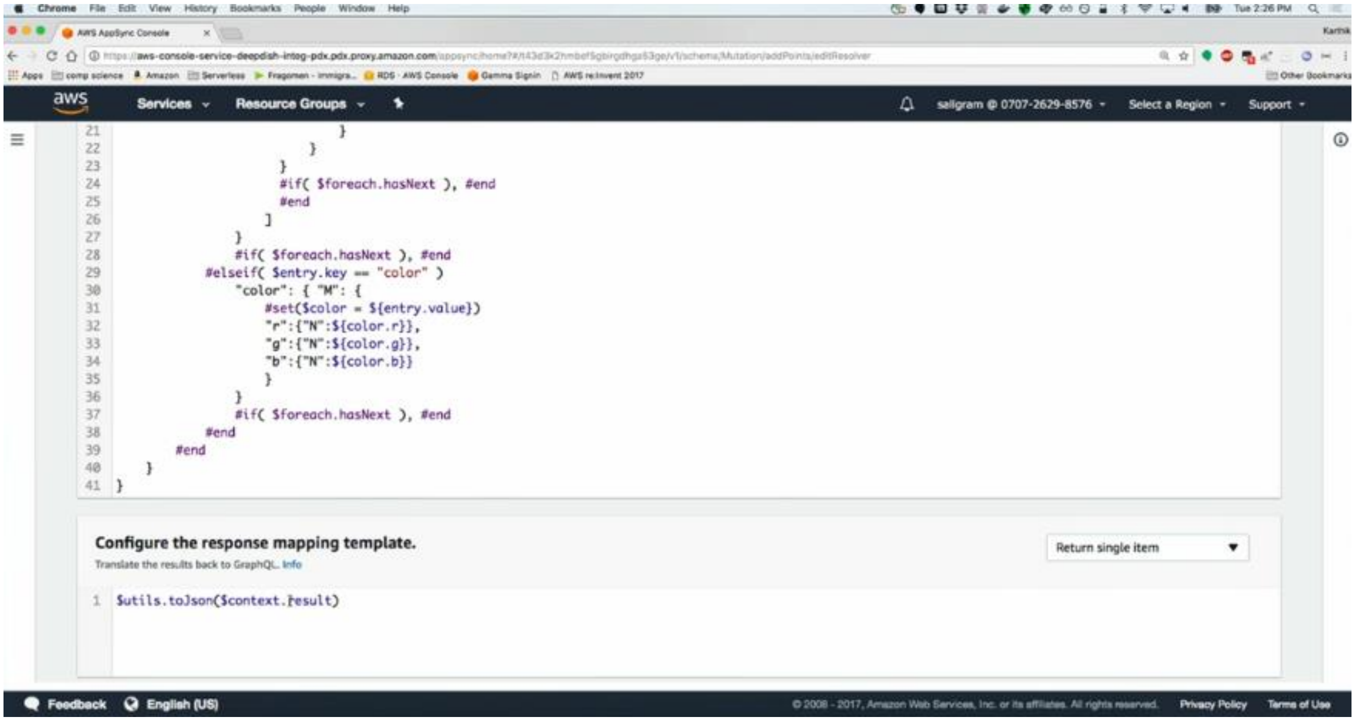
```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key" : {
        "deviceId": { "S" : "${context.arguments.deviceId}" },
        "id" : { "S" : "${context.arguments.id}" }
    },
    "attributeValues" : {
        #foreach( $entry in $context.arguments.entrySet() )
```


Chrome File Edit View History Bookmarks People Window Help
AWS AppSync Console
aws
Services
Resource Groups
saligram @ 0707-2629-8576
Select a Region
Support
Feedback
English (US)
© 2008 - 2017, Amazon Web Services, Inc. or its affiliates. All rights reserved.
Privacy Policy
Terms of Use

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key" : {
        "deviceId": { "S" : "${context.arguments.deviceId}" },
        "id" : { "S" : "${context.arguments.id}" }
    },
    "attributeValues" : {
        #foreach( $entry in $context.arguments.entrySet() )
            #if( $entry.key == "points" )
                "points" : {
                    "L" : [
                      #foreach ( $point in ${entry.value})
                      {
                          "M" : {
                              "x": {
                                  "N":${point.x}
                              },
                              "y": {
                                  "N":${point.y}
                              }
                          }
                      }
                      #if( $foreach.hasNext ), #end
                      #end
                    ]
                }
                #if( $foreach.hasNext ), #end
            #elseif( $entry.key == "color" )
                "color": { "M": {
```



```
            #if( $entry.key == "points" )
                "points" : {
                    "L" : [
                      #foreach ( $point in ${entry.value})
                      {
                          "M" : {
                              "x": {
                                  "N":${point.x}
                              },
                              "y": {
                                  "N":${point.y}
                              }
                          }
                      }
                      #if( $foreach.hasNext ), #end
                      #end
                    ]
                }
                #if( $foreach.hasNext ), #end
            #elseif( $entry.key == "color" )
                "color": { "M": {
                    #set($color = ${entry.value})
                    "r":{"N":${color.r}},
                    "g":{"N":${color.g}},
                    "b":{"N":${color.b}}
                    }
                }
                #if( $foreach.hasNext ), #end
            #end
        #end
    }
```



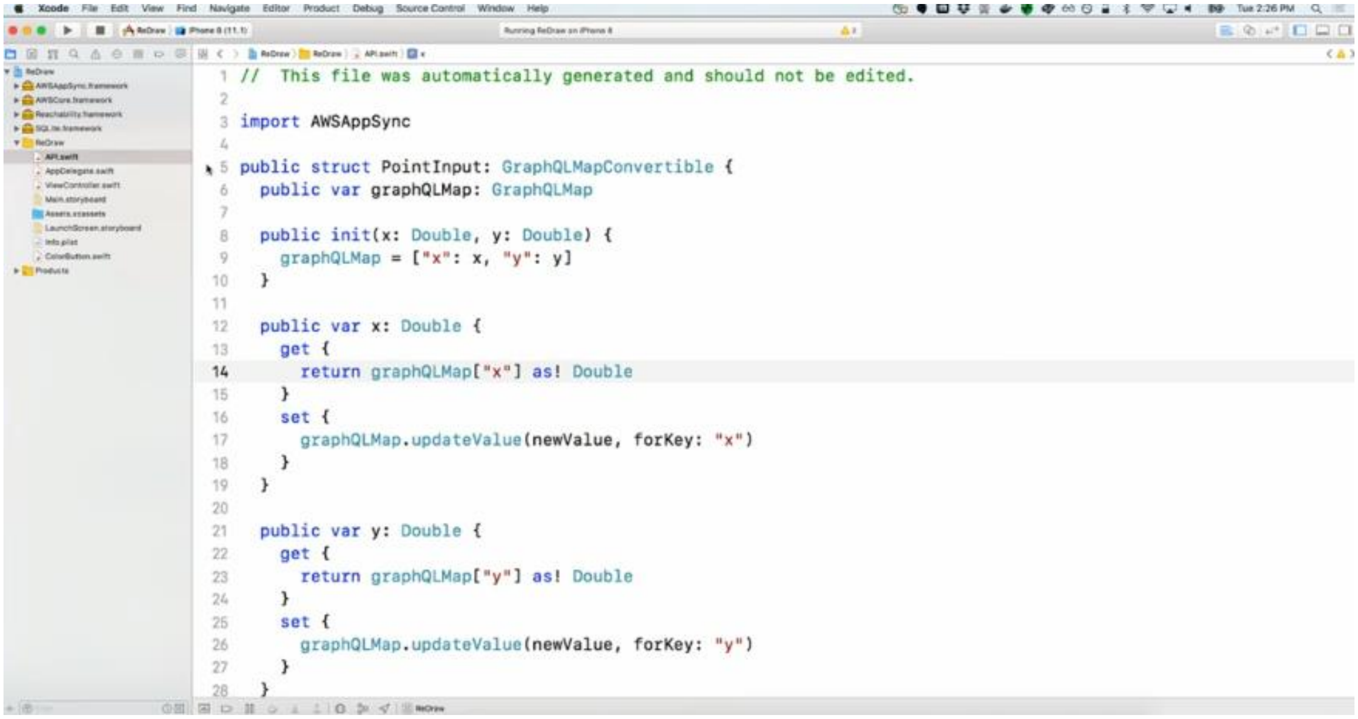
```
                              }
                          }
                      }
                      #if( $foreach.hasNext ), #end
                      #end
                    ]
                }
                #if( $foreach.hasNext ), #end
            #elseif( $entry.key == "color" )
                "color": { "M": {
                    #set($color = ${entry.value})
                    "r":{"N":${color.r}},
                    "g":{"N":${color.g}},
                    "b":{"N":${color.b}}
                    }
                }
                #if( $foreach.hasNext ), #end
            #end
        #end
    }
}
```

Configure the response mapping template.

Translate the results back to GraphQL. Info

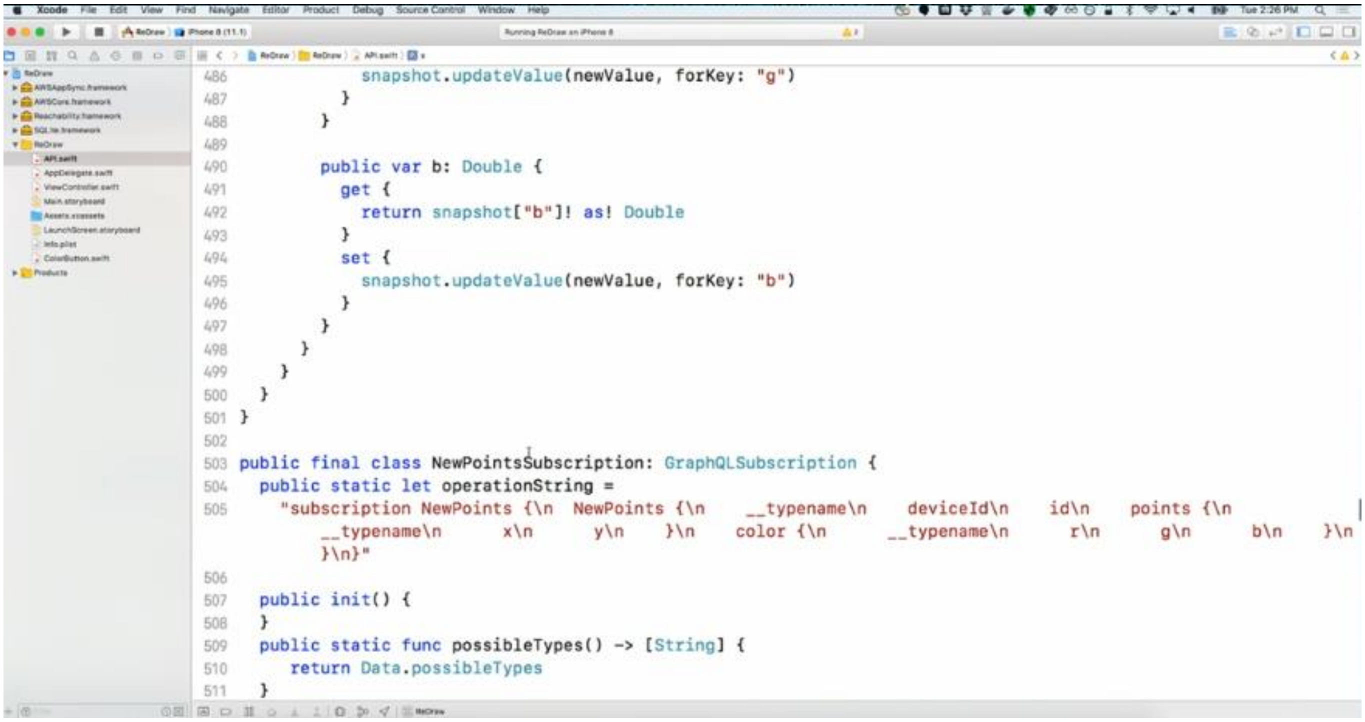
Return single item ▼

```
$utils.toJson($context.result)
```



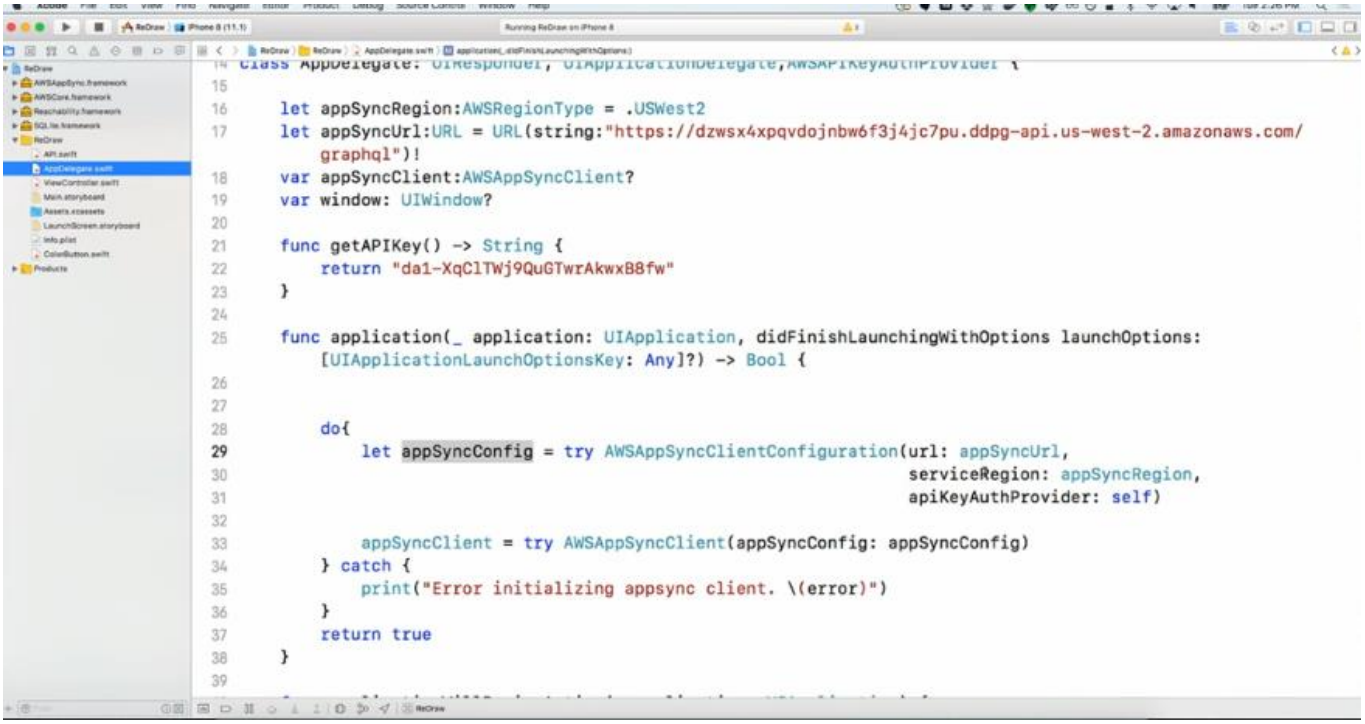
```swift
//  This file was automatically generated and should not be edited.

import AWSAppSync

public struct PointInput: GraphQLMapConvertible {
  public var graphQLMap: GraphQLMap

  public init(x: Double, y: Double) {
    graphQLMap = ["x": x, "y": y]
  }

  public var x: Double {
    get {
      return graphQLMap["x"] as! Double
    }
    set {
      graphQLMap.updateValue(newValue, forKey: "x")
    }
  }

  public var y: Double {
    get {
      return graphQLMap["y"] as! Double
    }
    set {
      graphQLMap.updateValue(newValue, forKey: "y")
    }
  }
```

We have an IOS app and used the Apollo client’s codegen capabilities to generate strongly typed objects already



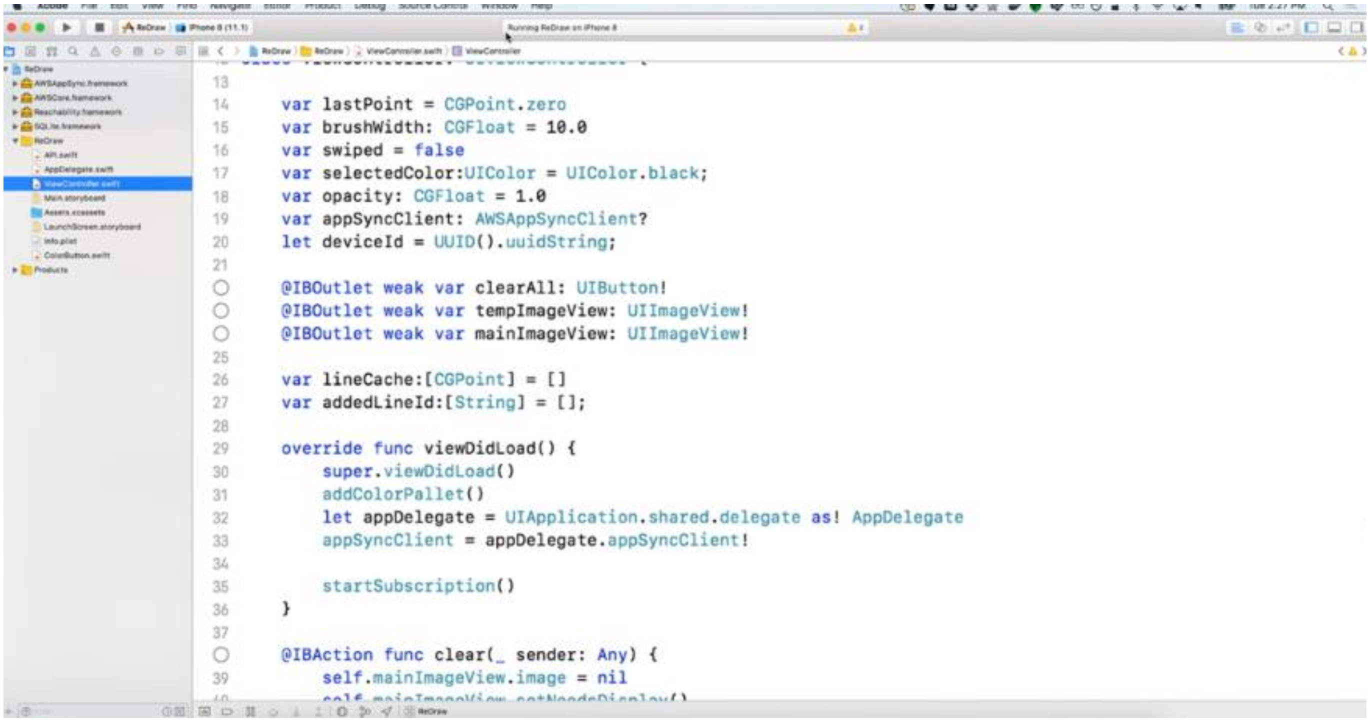
```swift
              snapshot.updateValue(newValue, forKey: "g")
            }
          }

          public var b: Double {
            get {
              return snapshot["b"]! as! Double
            }
            set {
              snapshot.updateValue(newValue, forKey: "b")
            }
          }
        }
      }
    }
  }
}

public final class NewPointsSubscription: GraphQLSubscription {
  public static let operationString =
    "subscription NewPoints {\n  NewPoints {\n    __typename\n    deviceId\n    id\n    points {\n
        __typename\n      x\n      y\n    }\n    color {\n      __typename\n      r\n      g\n      b\n    }\n
        }\n}"

  public init() {
  }
  public static func possibleTypes() -> [String] {
    return Data.possibleTypes
  }
```

It created a NewPointSubscription that we can use to listen for changes

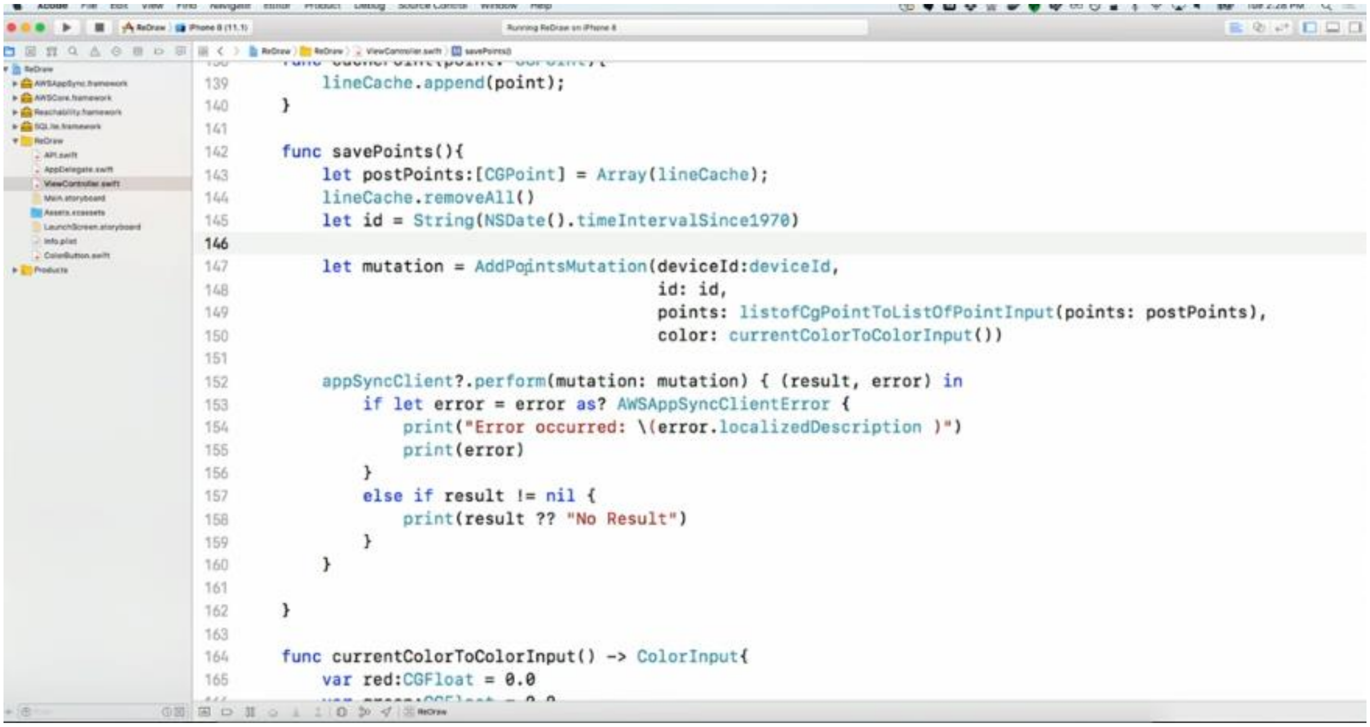


```swift
class AppDelegate: UIResponder, UIApplicationDelegate,AWSAPIKeyAuthProvider {

    let appSyncRegion:AWSRegionType = .USWest2
    let appSyncUrl:URL = URL(string:"https://dzwsx4xpqvdojnbw6f3j4jc7pu.ddpg-api.us-west-2.amazonaws.com/
        graphql")!
    var appSyncClient:AWSAppSyncClient?
    var window: UIWindow?

    func getAPIKey() -> String {
        return "da1-XqClTWj9QuGTwrAkwxB8fw"
    }

    func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions:
        [UIApplicationLaunchOptionsKey: Any]?) -> Bool {


        do{
            let appSyncConfig = try AWSAppSyncClientConfiguration(url: appSyncUrl,
                                                                  serviceRegion: appSyncRegion,
                                                                  apiKeyAuthProvider: self)

            appSyncClient = try AWSAppSyncClient(appSyncConfig: appSyncConfig)
        } catch {
            print("Error initializing appsync client. \(error)")
        }
        return true
    }

```

We have also created an appSyncConfig to use with our AppSync client as above. We are going to use this AppSync client in our View controllers to make calls to GQL endpoint.

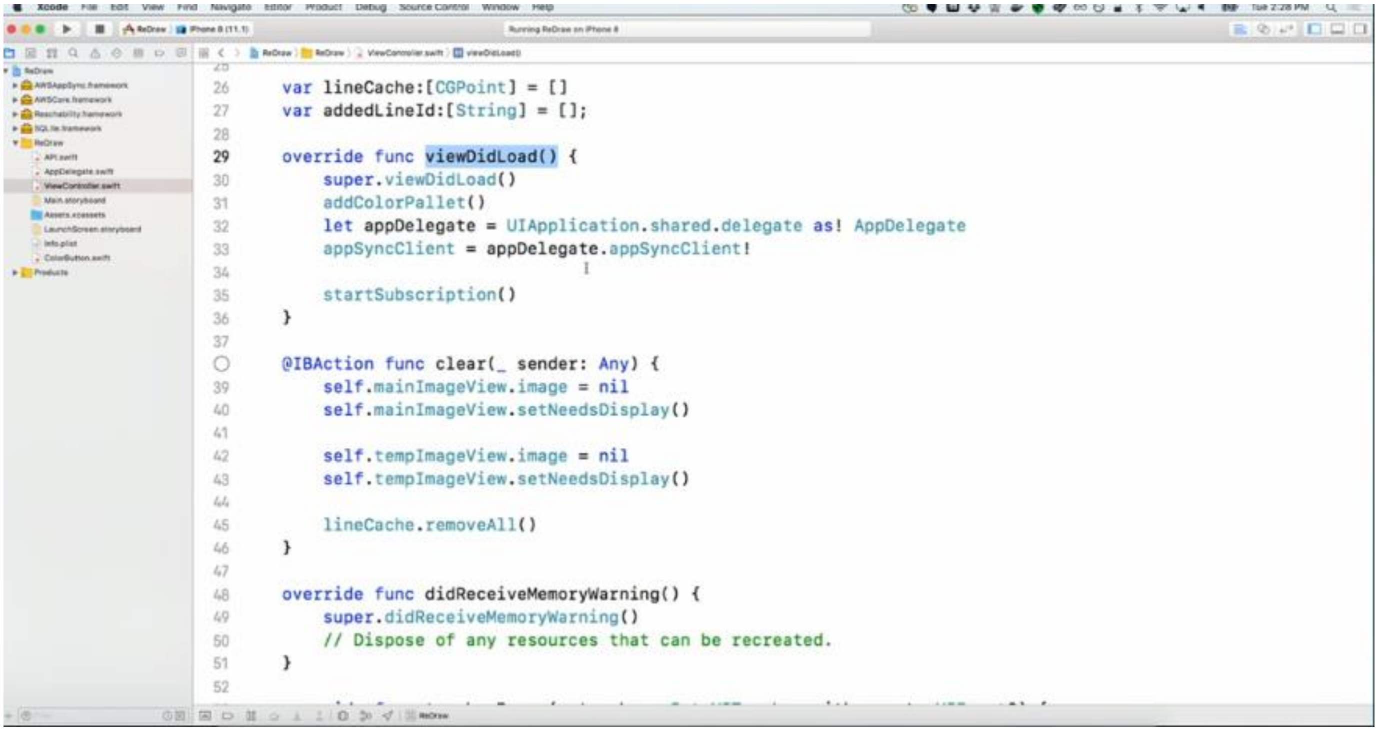


```swift
class ViewController: UIViewController {

    var lastPoint = CGPoint.zero
    var brushWidth: CGFloat = 10.0
    var swiped = false
    var selectedColor:UIColor = UIColor.black;
    var opacity: CGFloat = 1.0
    var appSyncClient: AWSAppSyncClient?
    let deviceId = UUID().uuidString;

    @IBOutlet weak var clearAll: UIButton!
    @IBOutlet weak var tempImageView: UIImageView!
    @IBOutlet weak var mainImageView: UIImageView!

    var lineCache:[CGPoint] = []
    var addedLineId:[String] = [];

    override func viewDidLoad() {
        super.viewDidLoad()
        addColorPallet()
        let appDelegate = UIApplication.shared.delegate as! AppDelegate
        appSyncClient = appDelegate.appSyncClient!

        startSubscription()
    }

    @IBAction func clear(_ sender: Any) {
        self.mainImageView.image = nil
        self.mainImageView.setNeedsDisplay()
```



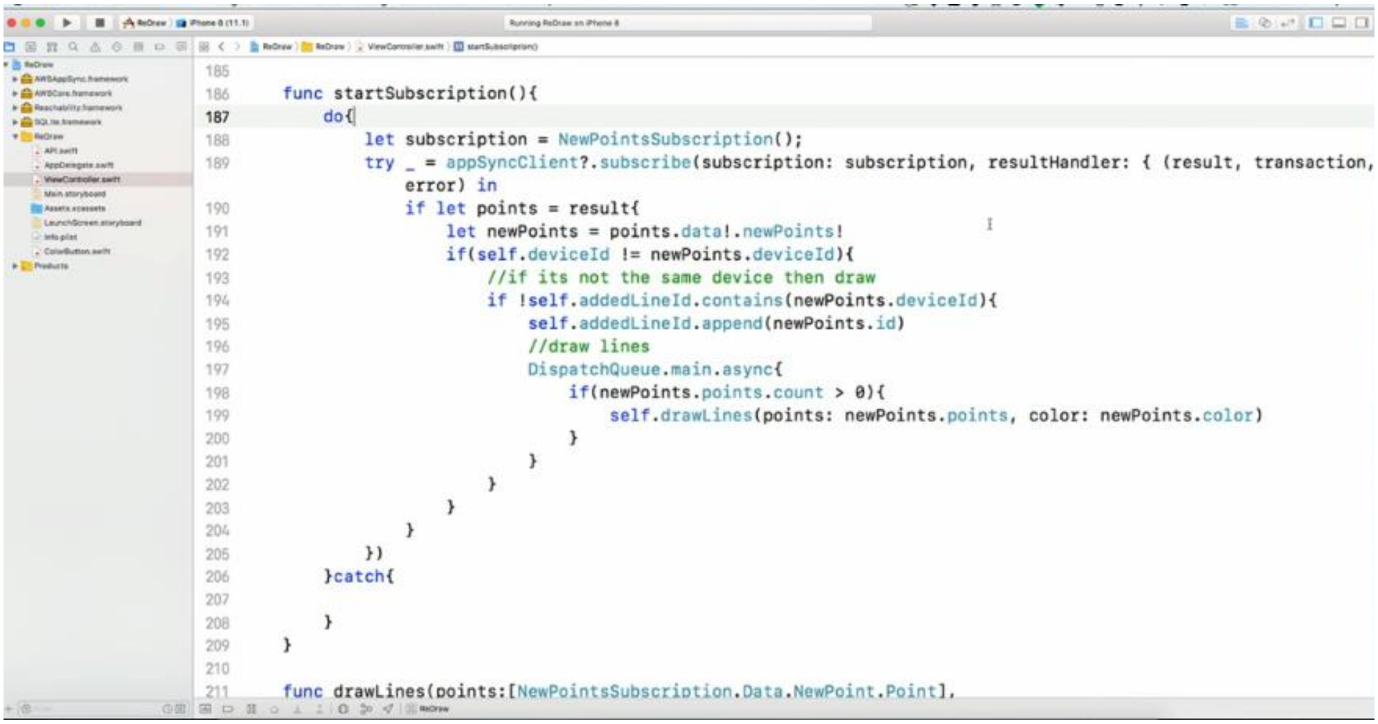
```
139         lineCache.append(point);
140     }
141 
142     func savePoints(){
143         let postPoints:[CGPoint] = Array(lineCache);
144         lineCache.removeAll()
145         let id = String(NSDate().timeIntervalSince1970)
146 
147         let mutation = AddPointsMutation(deviceId:deviceId,
148                                          id: id,
149                                          points: listofCgPointToListOfPointInput(points: postPoints),
150                                          color: currentColorToColorInput())
151 
152         appSyncClient?.perform(mutation: mutation) { (result, error) in
153             if let error = error as? AWSAppSyncClientError {
154                 print("Error occurred: \(error.localizedDescription )")
155                 print(error)
156             }
157             else if result != nil {
158                 print(result ?? "No Result")
159             }
160         }
161 
162     }
163 
164     func currentColorToColorInput() -> ColorInput{
165         var red:CGFloat = 0.0
```

We create a mutations object on line 147 and using the client on line 152



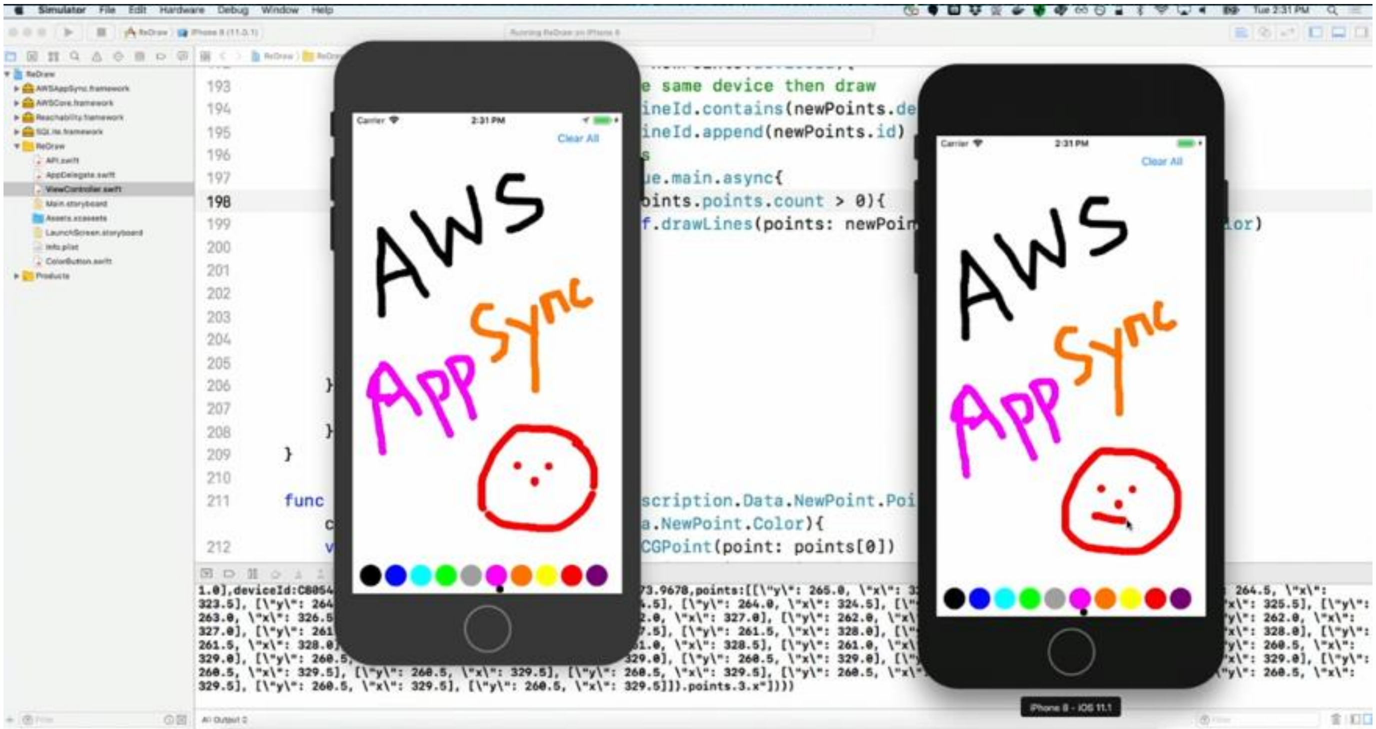
```
    var lineCache:[CGPoint] = []
    var addedLineId:[String] = [];

    override func viewDidLoad() {
        super.viewDidLoad()
        addColorPallet()
        let appDelegate = UIApplication.shared.delegate as! AppDelegate
        appSyncClient = appDelegate.appSyncClient!

        startSubscription()
    }

    @IBAction func clear(_ sender: Any) {
        self.mainImageView.image = nil
        self.mainImageView.setNeedsDisplay()

        self.tempImageView.image = nil
        self.tempImageView.setNeedsDisplay()

        lineCache.removeAll()
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }
```

As soon as the view loads, we will start a subscription channel that keeps listening for changes



```
185
186     func startSubscription(){
187         do{
188             let subscription = NewPointsSubscription();
189             try _ = appSyncClient?.subscribe(subscription: subscription, resultHandler: { (result, transaction,
                    error) in
190                 if let points = result{
191                     let newPoints = points.data!.newPoints!
192                     if(self.deviceId != newPoints.deviceId){
193                         //if its not the same device then draw
194                         if !self.addedLineId.contains(newPoints.deviceId){
195                             self.addedLineId.append(newPoints.id)
196                             //draw lines
197                             DispatchQueue.main.async{
198                                 if(newPoints.points.count > 0){
199                                     self.drawLines(points: newPoints.points, color: newPoints.color)
200                                 }
201                             }
202                         }
203                     }
204                 }
205             })
206         }catch{
207
208         }
209     }
210
211     func drawLines(points:[NewPointsSubscription.Data.NewPoint.Point],
```

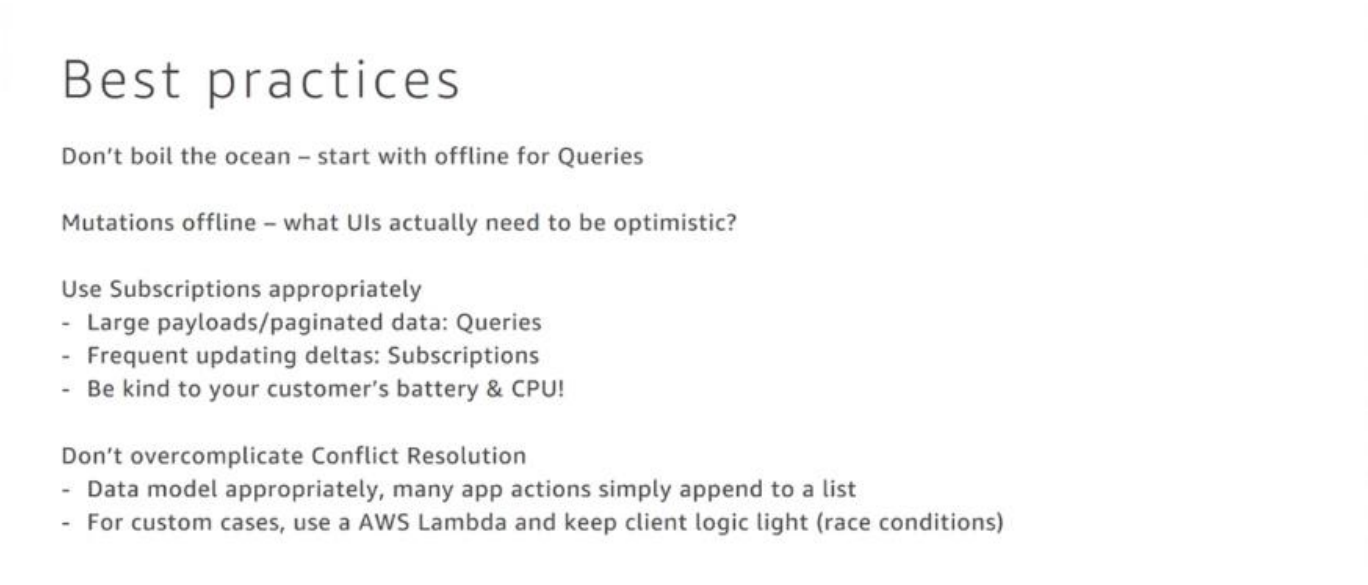
The resultHandler on line 189 is a callback for whenever a new event comes into the client



```
193                 e same device then draw
194                 ineId.contains(newPoints.de
195                 ineId.append(newPoints.id)
196                 s
197                 ue.main.async{
198                 oints.points.count > 0){
199                 f.drawLines(points: newPoi                    lor)
200
201
202
203
204
205
206         }
207
208         }
209     }
210
211     func                scription.Data.NewPoint.Poi
            c               a.NewPoint.Color){
212         v               CGPoint(point: points[0])
```

```
1.0],deviceId:C8054          73.9678,points:[[\"y\": 265.0, \"x\": 3          264.5, \"x\":
323.5], [\"y\": 264          .5], [\"y\": 264.0, \"x\": 324.5], [\"          x\": 325.5], [\"y\":
263.0, \"x\": 326.5          2.0, \"x\": 327.0], [\"y\": 262.0, \"x\          y\": 262.0, \"x\":
327.0], [\"y\": 261          7.5], [\"y\": 261.5, \"x\": 328.0], [\"          x\": 328.0], [\"y\":
261.5, \"x\": 328.0          1.0, \"x\": 328.5], [\"y\": 261.0, \"x\          y\": 260.5, \"x\":
329.0], [\"y\": 260.5,       329.0], [\"y\": 260.5, \"x\": 329.0], [\"          x\": 329.0], [\"y\":
260.5, \"x\": 329.5], [\"y\": 260.5, \"x\": 329.5], [\"y\": 260.5, \"x\": 329.5], [\"y\": 260.5, \"x\"          y\": 260.5, \"x\":
329.5], [\"y\": 260.5, \"x\": 329.5], [\"y\": 260.5, \"x\": 329.5]]).points.3.x"])))
```

Both the devices are now connected via the subscription topic and the information is available in near real time for updates using Subscriptions.



Best practices

Don't boil the ocean – start with offline for Queries

Mutations offline – what UIs actually need to be optimistic?

Use Subscriptions appropriately
- Large payloads/paginated data: Queries
- Frequent updating deltas: Subscriptions
- Be kind to your customer's battery & CPU!

Don't overcomplicate Conflict Resolution
- Data model appropriately, many app actions simply append to a list
- For custom cases, use a AWS Lambda and keep client logic light (race conditions)

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

AWS re:Invent

Thank You!!!