GraphQL Deep Dive - Designing Schemas and Automating Deployment

Application Development with AWS AppSync and AWS Amplify

Michael Paris
SDE
AWS

Agenda

Who is this talk for?

What is AWS AppSync?

What is AWS Amplify?

How does it fit together and help you build applications?

Who is this talk for?

I want to build a new app or website

and I want it to work on every platform

I want easily leverage AWS from my existing web or mobile app

and I don’t want to rewrite everything

I want to learn about cool new development tools

like React, GraphQL, CLIs, and serverless technologies

I want to focus less on ops and configuration and more on my product

What is GraphQL?

“GraphQL is a query language for APIs and a runtime for fulfilling those queries with your existing data. GraphQL provides a complete and understandable description of the data in your API, gives clients the power to ask for exactly what they need and nothing more, makes it easier to evolve APIs over time, and enables powerful developer tools.”

graphql.org

A query language for APIs…

Queries read data

```
query {
  getPost(id: ”1”) {
    id
    title
  }
}
```

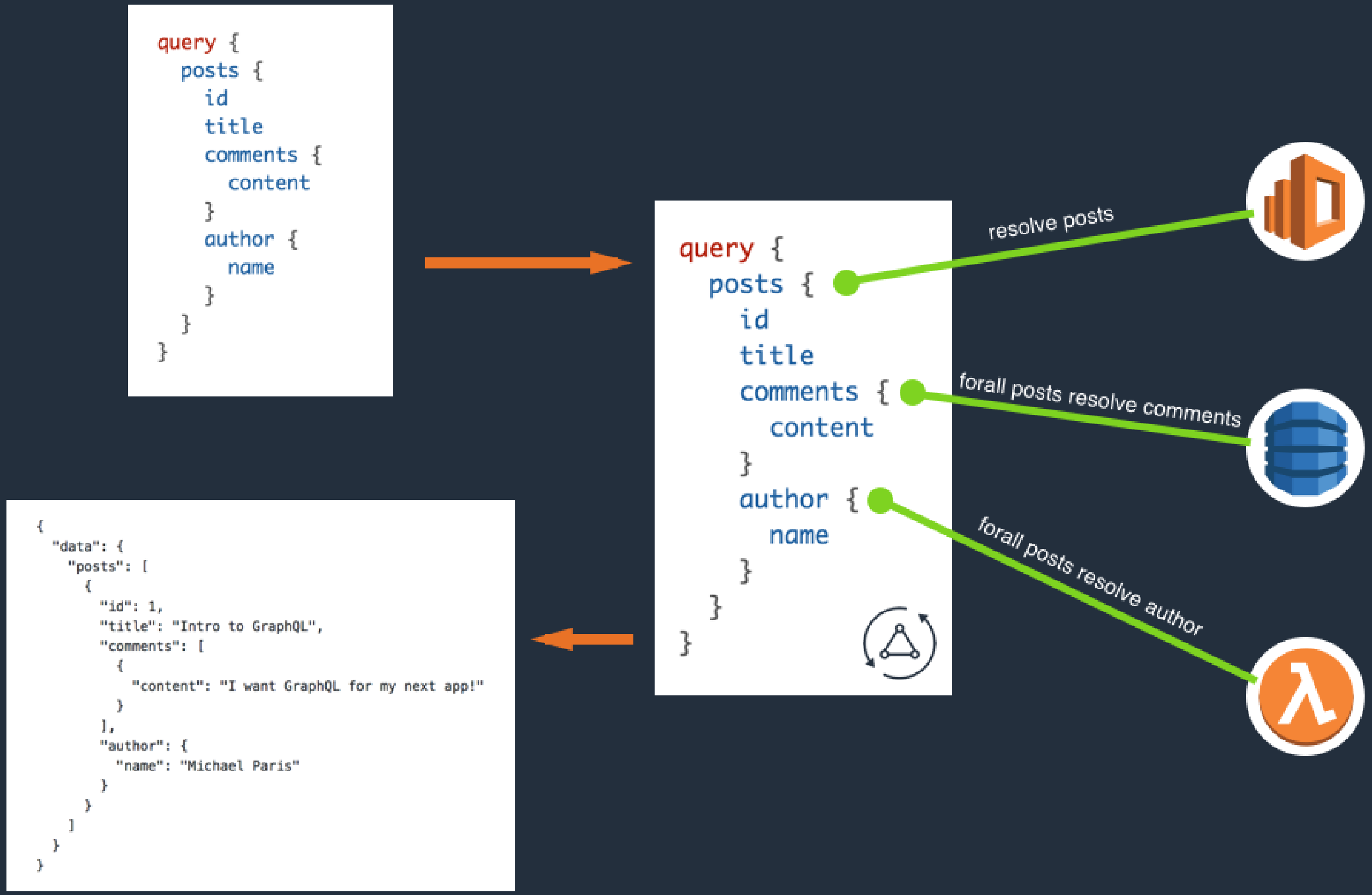
Mutations write data

```
mutation {
  createPost(title: “GraphQL @re:invent”) {
    id
    title
  }
}
```

Subscriptions are pushed data in real-time

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

A runtime for fulfilling those queries with your existing data



```
query {
  posts {
    id
    title
    comments {
      content
    }
    author {
      name
    }
  }
}
```

```
query {
  posts {
    id
    title
    comments {
      content
    }
    author {
      name
    }
  }
}
```

```
{
  "data": {
    "posts": [
      {
        "id": 1,
        "title": "Intro to GraphQL",
        "comments": [
          {
            "content": "I want GraphQL for my next app!"
          }
        ],
        "author": {
          "name": "Michael Paris"
        }
      }
    ]
  }
}
```

A complete and understandable description of the data in your API…

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

type Query {
  # Get a post by id.
  getPost(id: ID!): Post

  # Paginate through posts
  listPosts(limit: Int, nextToken: String): PostConnection
}
```

What is AWS AppSync?

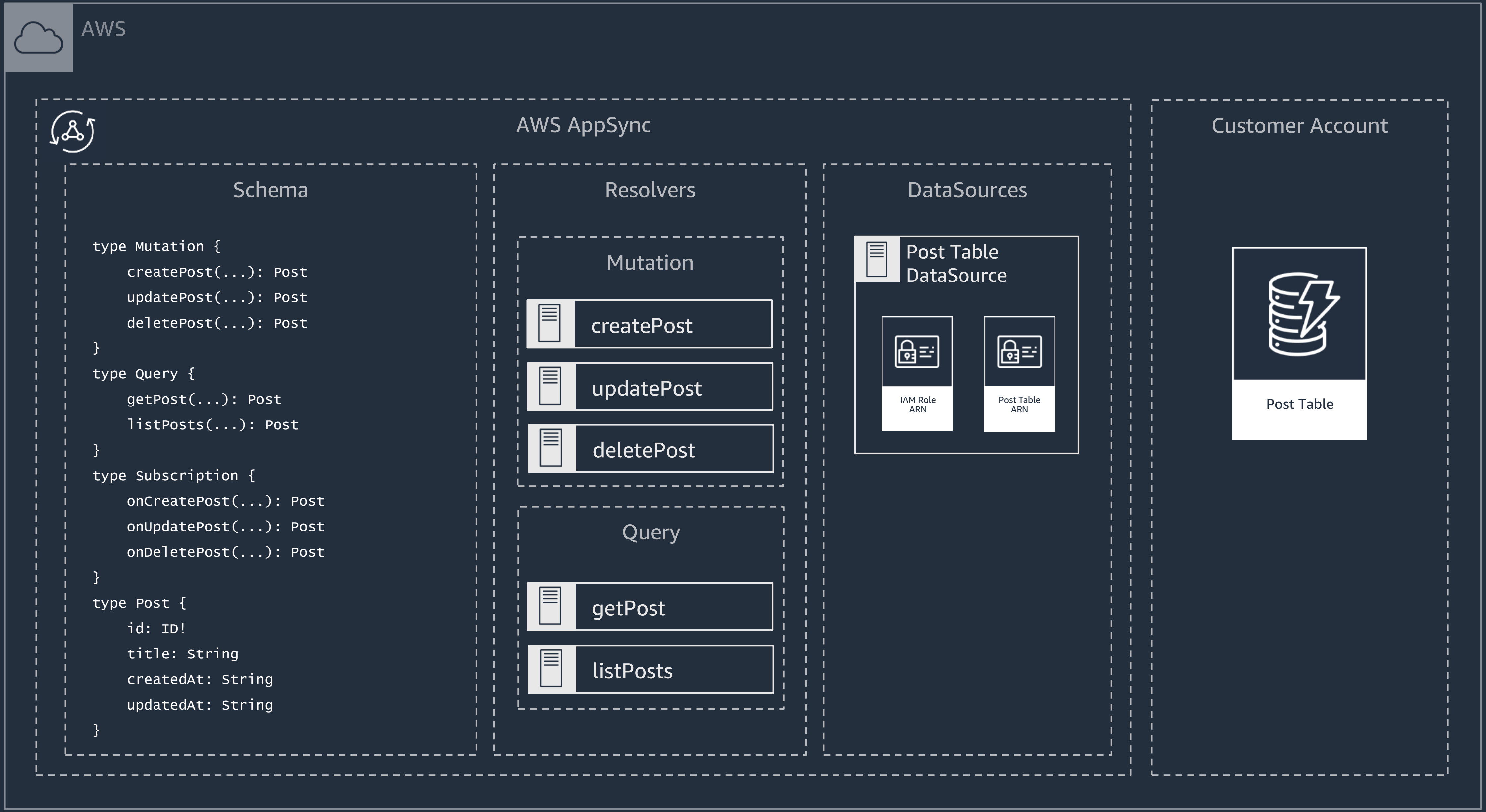
A managed GraphQL gateway.

- Define the shape of your API using GraphQL schema definition language (SDL)
- Attach data sources that reference other AWS resources
- Write resolvers that fetch data from data sources and attach them to fields

A real-time data broker.

- Subscribe to any mutation out of the box
- Reliable message delivery via MQTT over WebSockets

A Basic AppSync API



```
type Mutation {
    createPost(...): Post
    updatePost(...): Post
    deletePost(...): Post
}
type Query {
    getPost(...): Post
    listPosts(...): Post
}
type Subscription {
    onCreatePost(...): Post
    onUpdatePost(...): Post
    onDeletePost(...): Post
}
type Post {
    id: ID!
    title: String
    createdAt: String
    updatedAt: String
}
```

What is AWS Amplify?

A set of libraries that simplify using AWS from web & mobile apps

- Includes a number of categories Api, Auth, Storage, Analytics, etc.
- Platform specific wrappers for React, Angular, Ionic, React Native, iOS, Android
- Focused on solving common use cases based on customer feedback

A CLI that simplifies deploying to AWS resources for web & mobile apps

- Configure and deploy full application backends in a few key strokes
- Use the GraphQL transform to simplify the development of backend APIs

AWS Amplify

`$ amplify add auth`	Add an Amazon Cognito User Pool
`$ amplify add storage`	Create and secure an Amazon S3 bucket
`$ amplify add api`	Add an AWS AppSync API
`$ amplify push`	Deploy via AWS CloudFormation

Amplify + AppSync

Rapidly build scalable, data-driven applications

Leverages a new open-source project called the GraphQL Transform

- Declaratively define your application’s data model using GraphQL SDL
- Check your schema.graphql into git for easy version control
- Transforms your data model into a CloudFormation document that implements it

Powered by GraphQL directives

- Comes with a set of pre-built transformers
  - @model, @auth, @connection, @versioned, and @searchable

Let’s look at a few examples

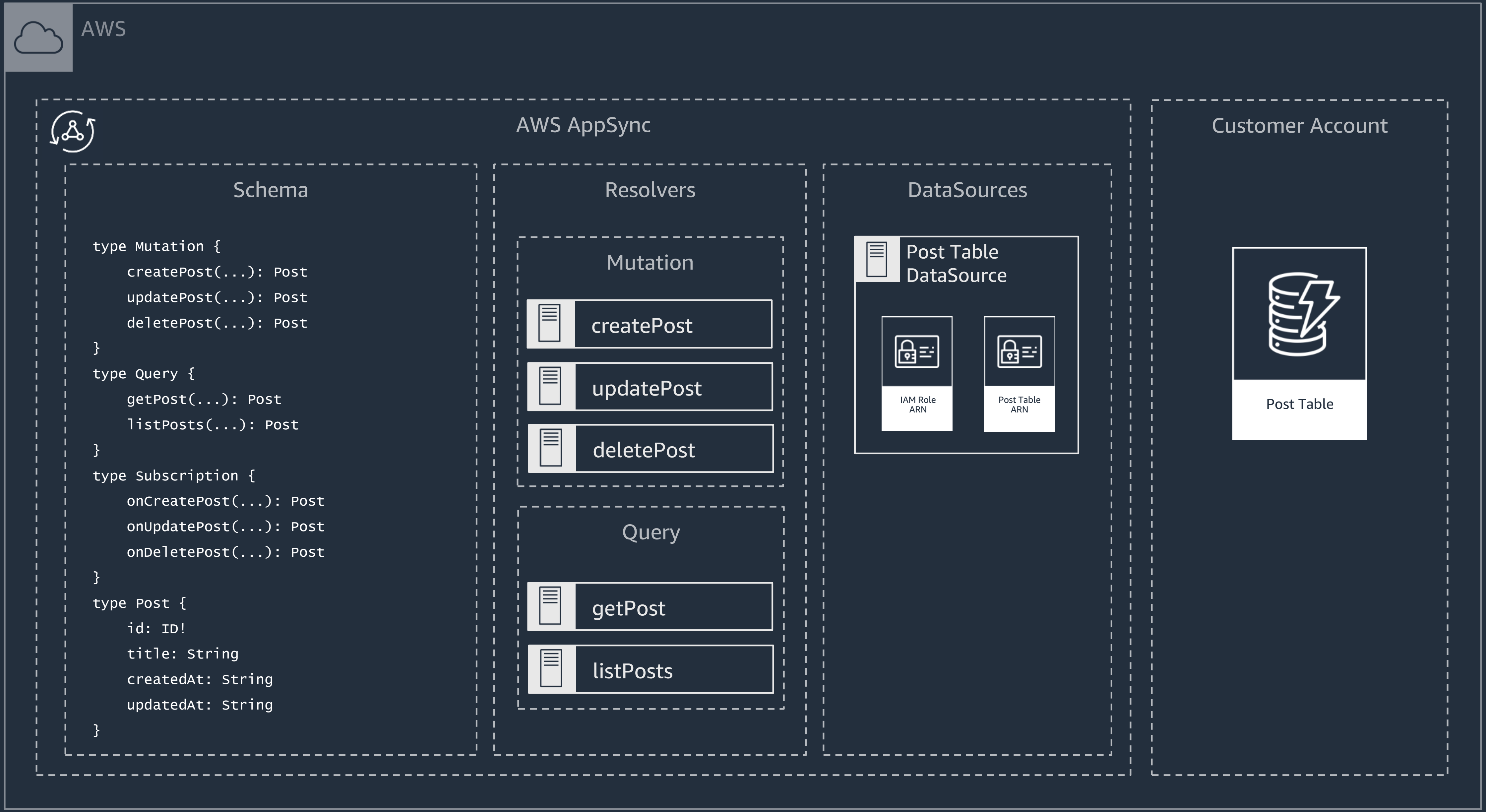
The @model transformer

```
$ amplify add api
```

```
# schema.graphql
type Post @model {
  id: ID!
  title: String!
}
```

```
$ amplify push
```

And voila!



```
type Mutation {
    createPost(...): Post
    updatePost(...): Post
    deletePost(...): Post
}
type Query {
    getPost(...): Post
    listPosts(...): Post
}
type Subscription {
    onCreatePost(...): Post
    onUpdatePost(...): Post
    onDeletePost(...): Post
}
type Post {
    id: ID!
    title: String
    createdAt: String
    updatedAt: String
}
```

Your API is ready to go.

Let’s query it.

Create an object

```
mutation CreatePost {
  createPost(input: {
    title: “A new post!”
  }) {
    id
    title
  }
}
```

```
{
    “data”: {
        “createPost”: {
            ”id”: “678736c1-6a64-44ca-a4be-8a76913ed4c8”,
            “title”: “A new post!”
        }
    }
}
```

Create a post with a DynamoDB PutItem operation.

Get an object

```
query GetPost {
  getPost(
    id: ”678736c1-6a64-44ca-a4be-8a76913ed4c8”
  ) {
    id
    title
  }
}
```

```
{
    “data”: {
        “getPost”: {
            ”id”: “678736c1-6a64-44ca-a4be-8a76913ed4c8”,
            “title”: “A new post!”
        }
    }
}
```

Resolve the post with a DynamoDB GetItem operation.

List and paginate objects

```
query ListPosts {
  listPosts(limit: 10, nextToken: “...”) {
    items {
      id
      title
    }
    nextToken
  }
}
```

```
{
    “data”: {
        “listPosts”: {
            “items”: [{
                ”id”: “678736c1-6a64-44ca-a4be-8a76913ed4c8”,
                “title”: “A new post!”
            }],
            “nextToken”: “...”
        }
    }
}
```

Resolve the post with a DynamoDB Scan operation.

Update an object

```
mutation UpdatePost {
  updatePost(input: {
    id: “678736c1-6a64-44ca-a4be-8a76913ed4c8”,
    title: “A different post!”
  }) {
    id
    title
  }
}
```

```
{
    “data”: {
        “updatePost”: {
            ”id”: “678736c1-6a64-44ca-a4be-8a76913ed4c8”,
            “title”: “A different post!”
        }
    }
}
```

Update a post with a DynamoDB UpdateItem operation.

Delete an object

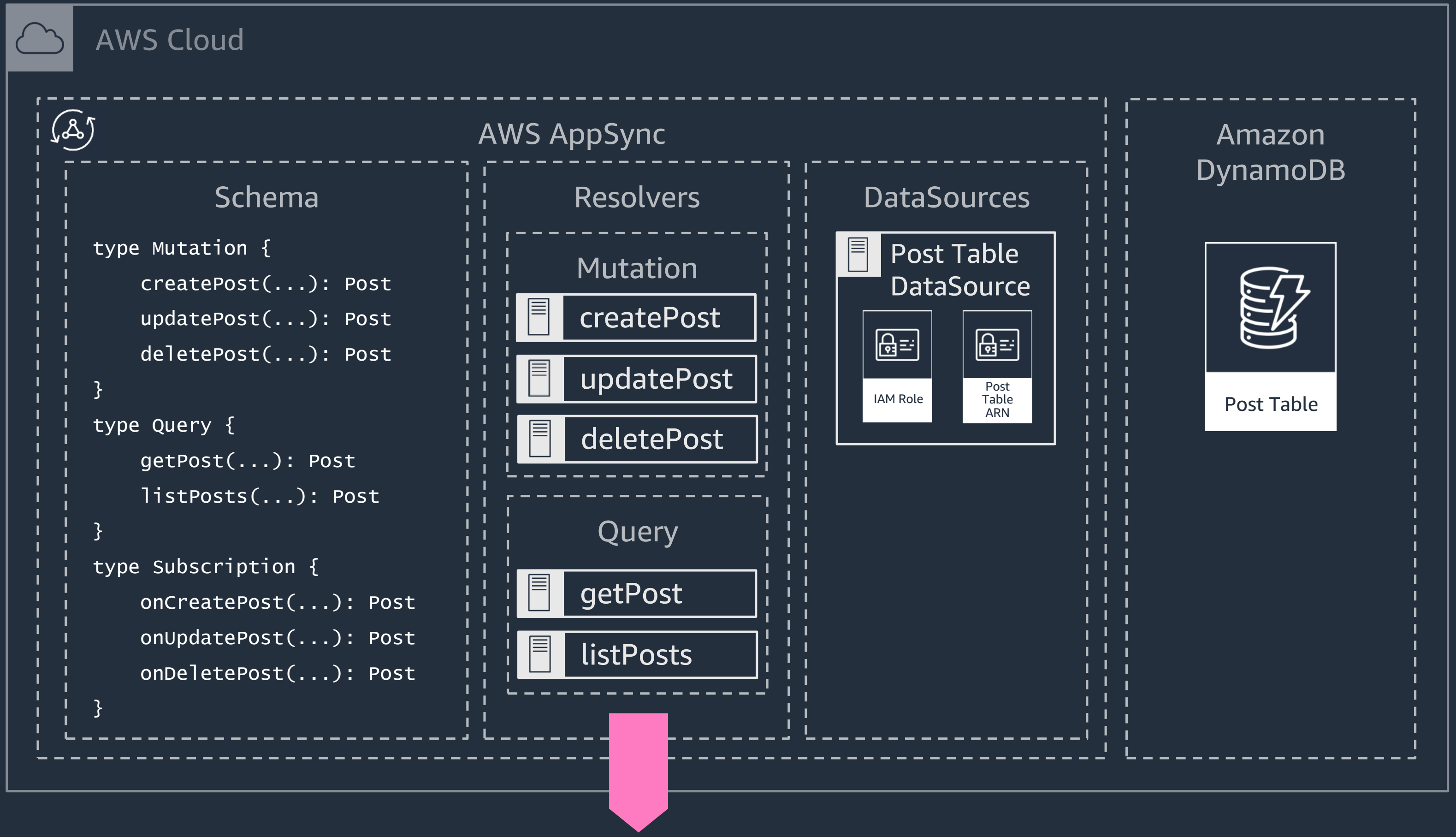
```
mutation DeletePost {
  deletePost(input: {
    id: “678736c1-6a64-44ca-a4be-8a76913ed4c8”
  }) {
    id
    title
  }
}
```

```
{
    “data”: {
        “deletePost”: {
            ”id”: “678736c1-6a64-44ca-a4be-8a76913ed4c8”,
            “title”: “A different post!”
        }
    }
}
```

Delete a post with a DynamoDB DeleteItem operation.

@auth (static group)

```
type Post @model @auth(rules: [
  { allow: groups, groups: ["Admin"] }
]) {
  id: ID!
  title: String!
}
```

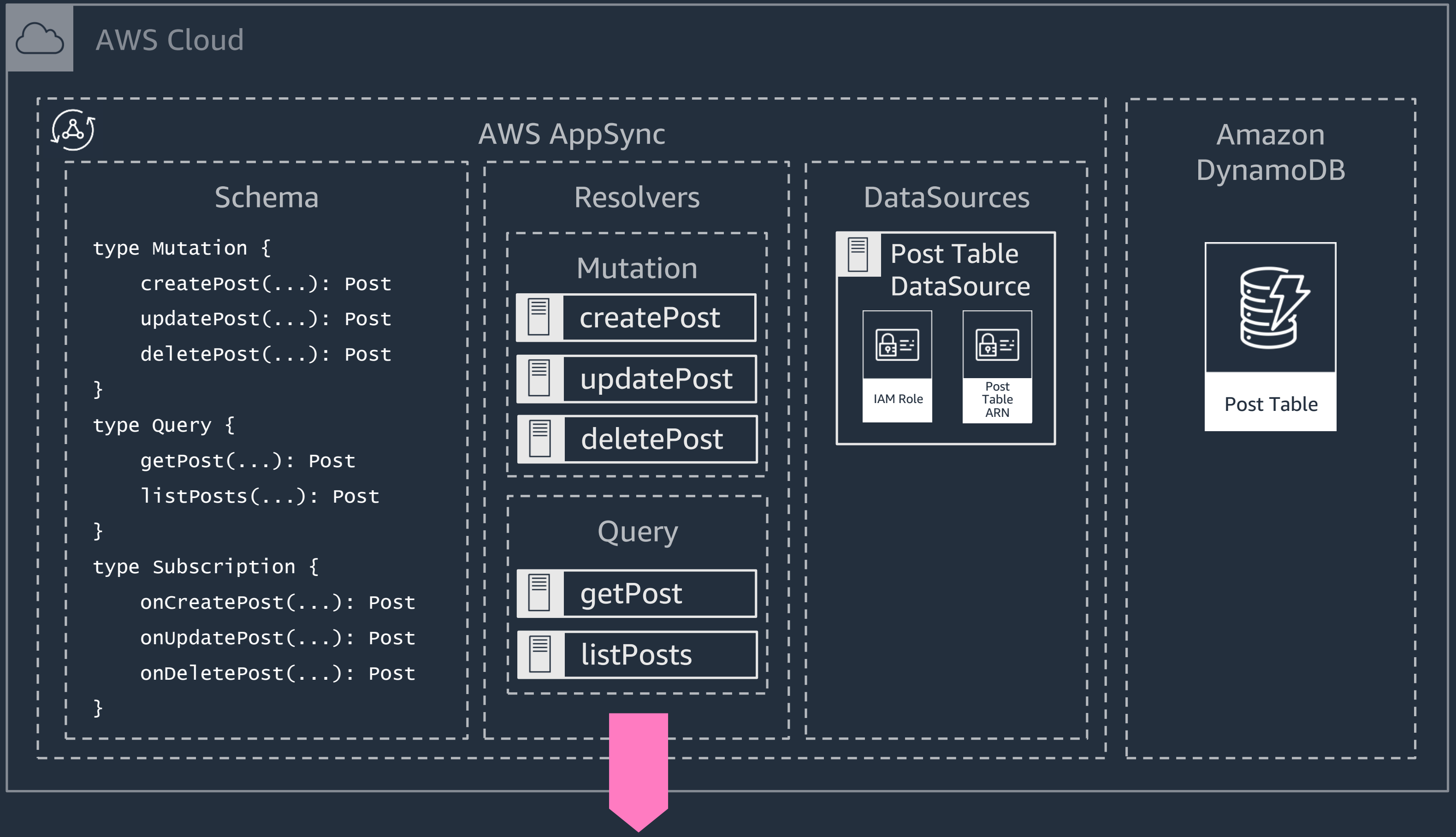


```
type Mutation {
    createPost(...): Post
    updatePost(...): Post
    deletePost(...): Post
}
type Query {
    getPost(...): Post
    listPosts(...): Post
}
type Subscription {
    onCreatePost(...): Post
    onUpdatePost(...): Post
    onDeletePost(...): Post
}
```

If the logged in user is not in the "Admin" group, mutations will fail and queries will return null.

@auth (dynamic group)

```
type Post @model @auth(rules: [{
    allow: groups,
    groupsField: “allowedGroups”
  }]) {
  id: ID!
  title: String!
  allowedGroups: [String!]!
}
```

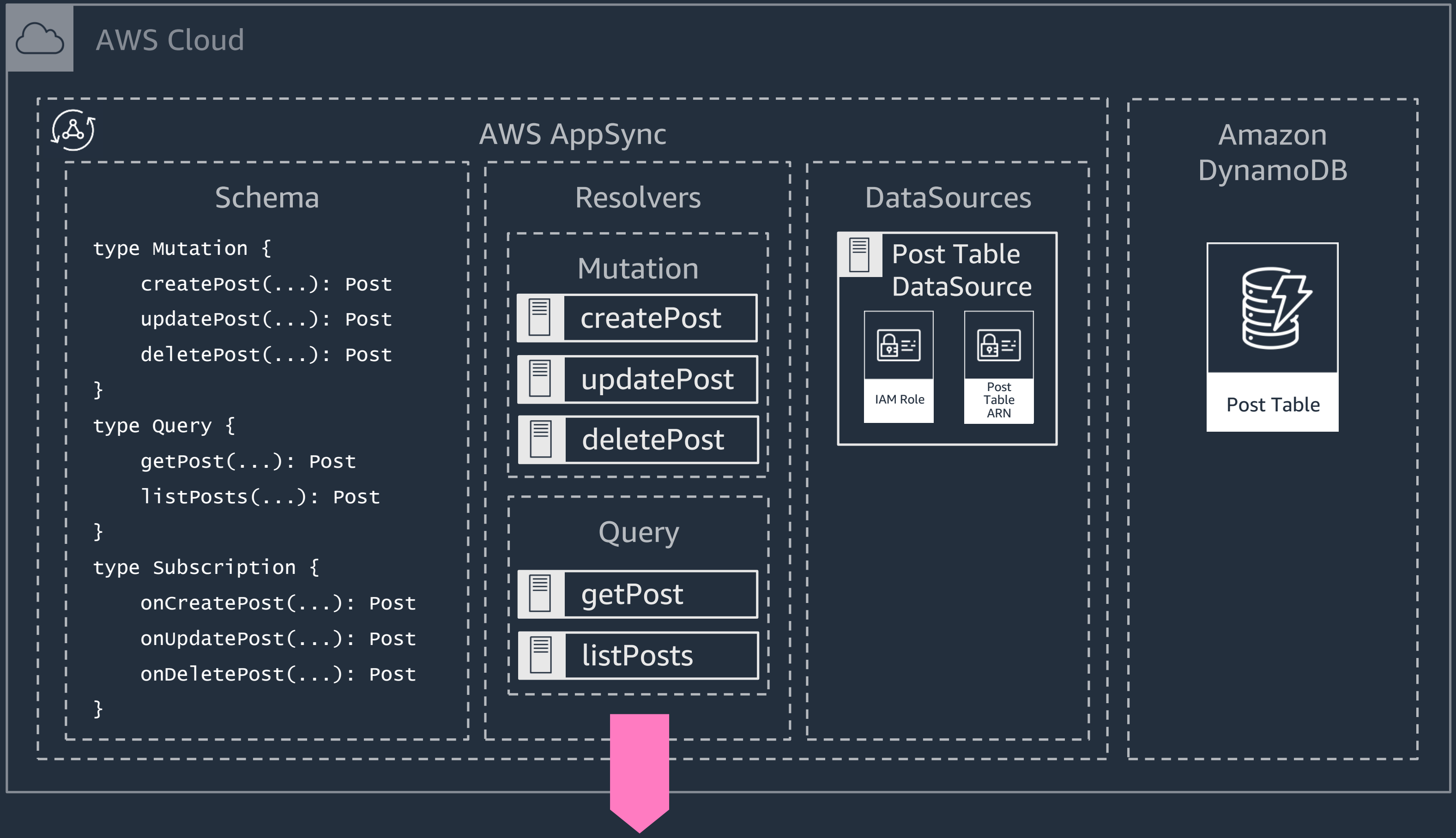


```
type Mutation {
    createPost(...): Post
    updatePost(...): Post
    deletePost(...): Post
}
type Query {
    getPost(...): Post
    listPosts(...): Post
}
type Subscription {
    onCreatePost(...): Post
    onUpdatePost(...): Post
    onDeletePost(...): Post
}
```

If the logged in user is not a member of any of the groups in the object’s allowedGroups field, mutations will fail and queries will return null.

@auth (ownership)

```
type Post @model @auth(rules: [
  { allow: owner, ownerField: "editors" }
]) {
  id: ID!
  title: String!
  editors: [String!]!
}
```

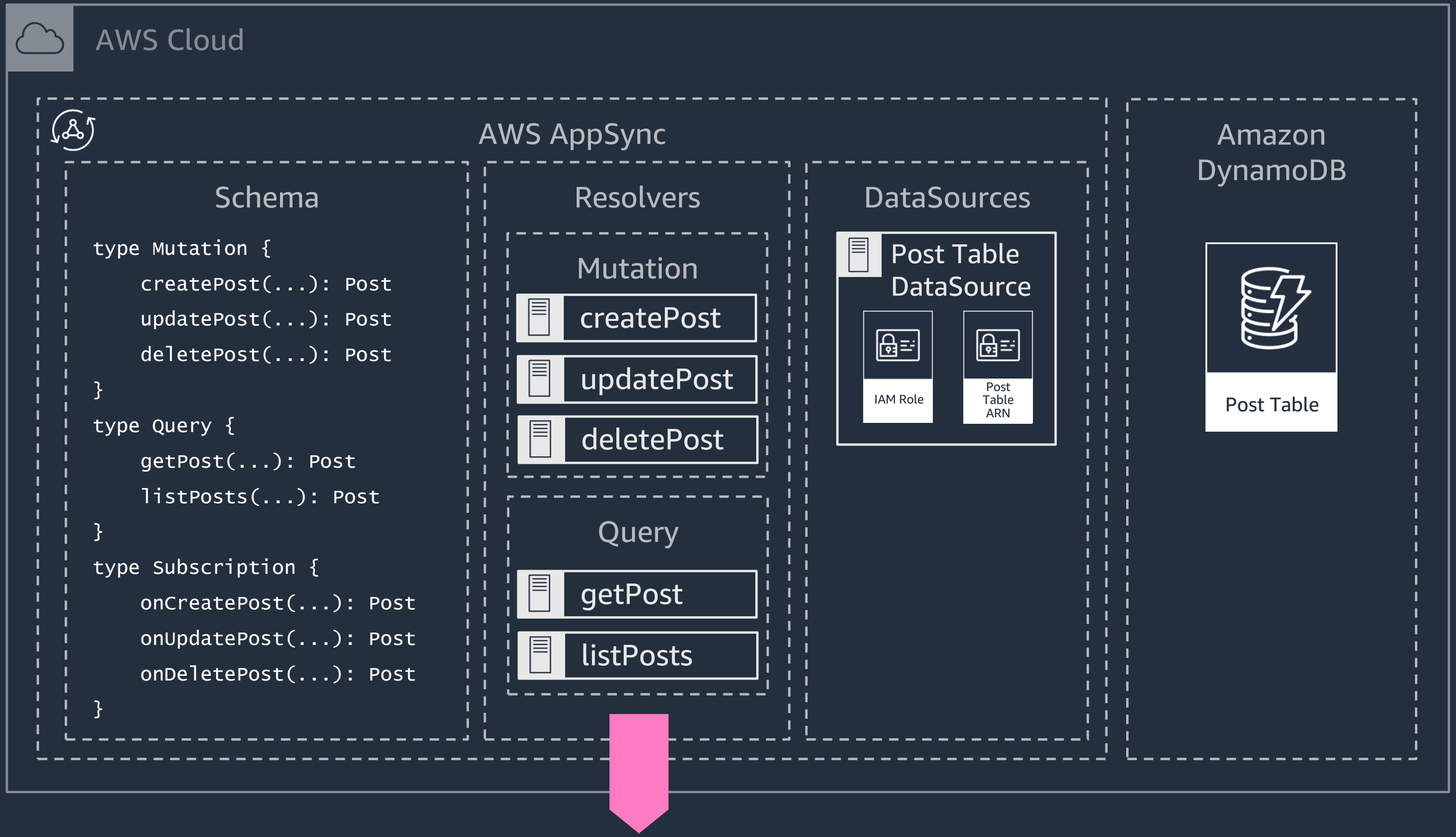


```
type Mutation {
    createPost(...): Post
    updatePost(...): Post
    deletePost(...): Post
}
type Query {
    getPost(...): Post
    listPosts(...): Post
}
type Subscription {
    onCreatePost(...): Post
    onUpdatePost(...): Post
    onDeletePost(...): Post
}
```

If the logged in user is not specified in the object's editors field, mutations will fail and queries will return null.

@auth (combo)

```
type Post @model @auth(rules: [
  { allow: groups, groups: ["Admin"] },
  { allow: owner, ownerField: "editors" }
]) {
  id: ID!
  title: String!
  editors: [String!]!
}
```

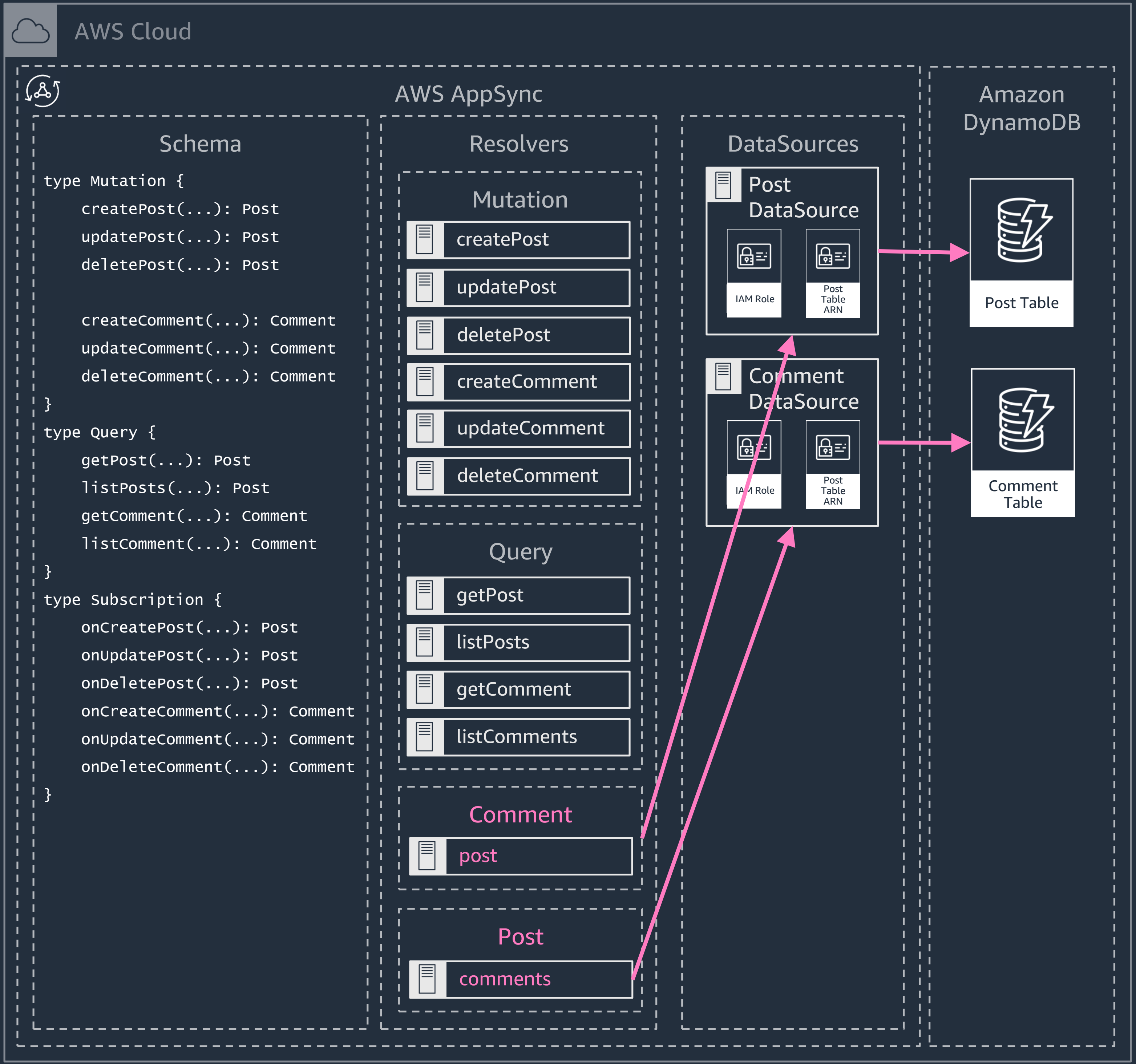


```
type Mutation {
    createPost(...): Post
    updatePost(...): Post
    deletePost(...): Post
}
type Query {
    getPost(...): Post
    listPosts(...): Post
}
type Subscription {
    onCreatePost(...): Post
    onUpdatePost(...): Post
    onDeletePost(...): Post
}
```

Combine rules for more complex authorization strategies.

@connection

```
type Post @model {
  id: ID!
  title: String!
  comments: [Comment!]!
    @connection(name: “PostComments”)
}
type Comment @model {
  id: ID!
  content: String!
  post: Post
    @connection(name: “PostComments”)
}
```



```
type Mutation {
    createPost(...): Post
    updatePost(...): Post
    deletePost(...): Post

    createComment(...): Comment
    updateComment(...): Comment
    deleteComment(...): Comment
}
type Query {
    getPost(...): Post
    listPosts(...): Post
    getComment(...): Comment
    listComment(...): Comment
}
type Subscription {
    onCreatePost(...): Post
    onUpdatePost(...): Post
    onDeletePost(...): Post
    onCreateComment(...): Comment
    onUpdateComment(...): Comment
    onDeleteComment(...): Comment
}
```

Also creates a GSI on the Comment table named gsi-PostComments

Querying with @connection

```
query GetPostAndComments {
  getPost(id: ”678736c1-6a64-44ca-a4be-8a76913ed4c8”) {
    id
    title
    comments(limit: 10, nextToken: “...”) {
      items {
        id
        content
      }
      nextToken
    }
  }
}
```

Resolve the post with a DynamoDB GetItem request.

Resolve the comments for a specific post using a DynamoDB Query operation for comment’s where the postId is equal to the id of the parent post object.

The transform transparently configures a GSI named gsi-PostComments and wires up the resolvers to use it.

@versioned

```
type Post @model @versioned {
  id: ID!
  title: String!
}
```

→

```
type Post {
  id: ID!
  title: String!
  version: Int!
}
input UpdatePostInput {
  id: ID!
  title: String
  expectedVersion: Int!
}
# Also added to DeletePostInput
```

Querying with @versioned

```
mutation CreatePost {
  createPost(input: {
    title: “A new post!”
  }) {
    id
    title
    version
  }
}
```

```
{
  “data”: {
    “createPost”: {
      “id”: “678736c1-6a64-44ca-a4be-8a76913ed4c8”,
      “title”: “A new post!”,
      “version”: 1
    }
  }
}
```

Querying with @versioned

```
mutation UpdatePost {
  updatePost(input: {
    id: “678736c1-6a64-44ca-a4be-8a76913ed4c8”,
    title: “An updated post!”,
    expectedVersion: 1
  }) {
    id
    title
    version
  }
}
```

```
{
  “data”: {
    “updatePost”: {
      “id”: “678736c1-6a64-44ca-a4be-8a76913ed4c8”,
      “title”: “An updated post!”,
      “version”: 2
    }
  }
}
```

Querying with @versioned

Meanwhile, a second user logged on and fetched the object after it was created but before it was updated. That user then went offline and while offline tried to update the object.

After coming back online, the application attempts…

Querying with @versioned

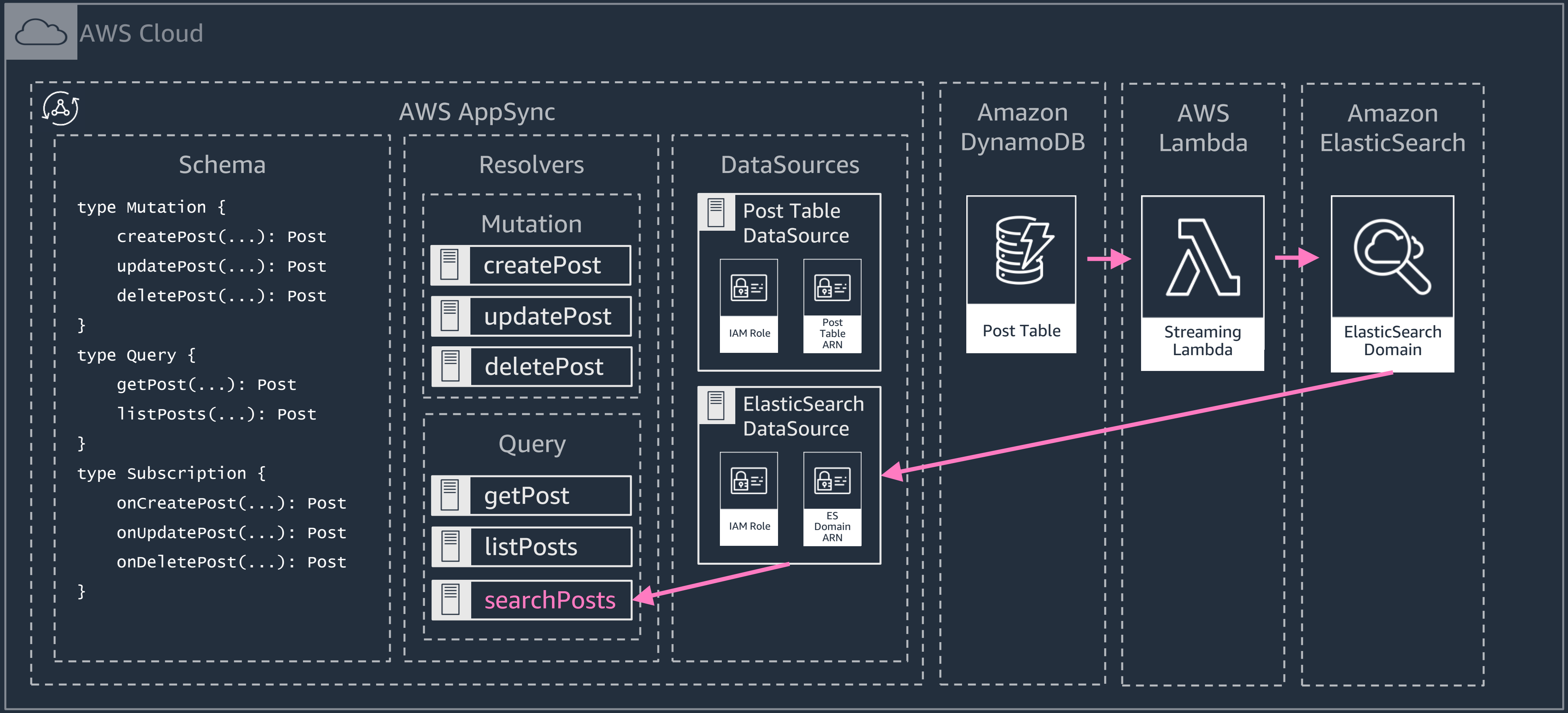
```
mutation UpdatePost {
  updatePost(input: {
    id: “678736c1-6a64-44ca-a4be-8a76913ed4c8”,
    title: “I prefer this title!”,
    expectedVersion: 1
  }) {
    id
    title
    version
  }
}
```

throws ConditionalUpdateFailedException

This is conflict detection. The version on the server is 2 so the update is rejected. You may now handle the conflict with conflict resolution logic in your app or via an AWS Lambda function.

@searchable

```
type Post
  @model
  @searchable
{
  id: ID!
  title: String!
}
```



```
type Mutation {
    createPost(...): Post
    updatePost(...): Post
    deletePost(...): Post
}
type Query {
    getPost(...): Post
    listPosts(...): Post
}
type Subscription {
    onCreatePost(...): Post
    onUpdatePost(...): Post
    onDeletePost(...): Post
}
```

Search objects

```
query SearchPosts {
  searchPosts(filter: {
    title: {
      wildcard: “*new*”
    }
  }) {
    items {
      id
      title
    }
  }
}
```

```
{
    “data”: {
        “searchPosts”: {
            “items”: [{
                ”id”: “678736c1-6a64-44ca-a4be-8a76913ed4c8”,
                “title”: “A new post!”
            }]
        }
    }
}
```

Resolve the posts with an ElasticSearch wildcard query.

Amplify API Overview

Simple, declarative configuration.

Design your data model & let the tool do the work.

Works seamlessly with the AWS Amplify Framework.

Instant offline & real-time support in client apps.

All data is stored in your AWS Account.

Use all the other great tools & services AWS has to offer.

The AWS Amplify Library

The Amplify Library

Easily connect AWS services to web and mobile apps.

- Categories include API, Analytics, Auth, Function, Storage, etc

Simple to use abstractions for common use cases.

Support for many frameworks & platforms.

- React, React-Native, iOS, Android, Ionic, Angular, etc.

Configure Amplify

```
import Amplify from 'aws-amplify';

import awsconfig from './aws-exports';


Amplify.configure(awsconfig);
```

Run a query

```
import { API } from 'aws-amplify';

const createPostMutation = `

    mutation CreatePost($input: CreatePostInput!) {

        createPost(input: $input) {

            id title

        }

    }

}`

const newPost = { title: `A new post` }

const createPostResponse = await API.graphql(graphqlOperation(createPostMutation, { input: newPost }));
```

Or connect a component (React & React Native)

```
const PostsList = props => (
    <ul>
        {props.data.listPosts.items.map(
            post => <li key={post.id}>{post.title}</li>
        )}
    </ul>
)
```

```
const listPostsQuery = gql(`
  query ListPosts {
    listPosts {
      items {
        id
        title
      }
    }
  }
`)

export default graphql(listPostsQuery)(PostsList)
```

AWS Amplify Codegen

```
$ amplify add codegen

$ amplify codegen

# Generate queries & native types from your GraphQL API

# for iOS, Android, TypeScript, and Flow
```

And a lot more!

https://aws-amplify.github.io

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

@mikeparisstuff

Thanks

And now for a demo!