Serverless GraphQL
Backend Architecture

GraphQL is a new API technology that provides a more efficient, powerful and flexible alternative to REST. It was open-sourced by Facebook in 2015, but has been in internal use already since 2012 to power their native mobile apps. GraphQL has seen major adoption since it was released and is already used in production by bigger companies like Twitter, GitHub and Shopify. Combined with serverless functions, another popular server-side technology, GraphQL has the potential to completely change the way how backend development works. At the core of this new architecture, there is event-driven business logic as well as the global type system that's defined in the GraphQL schema. The schema serves as the contract for the communication between server and client. At Graphcool, we've been using these technologies in production for more than a year and want to share our experience and ideas with the community.

Agenda

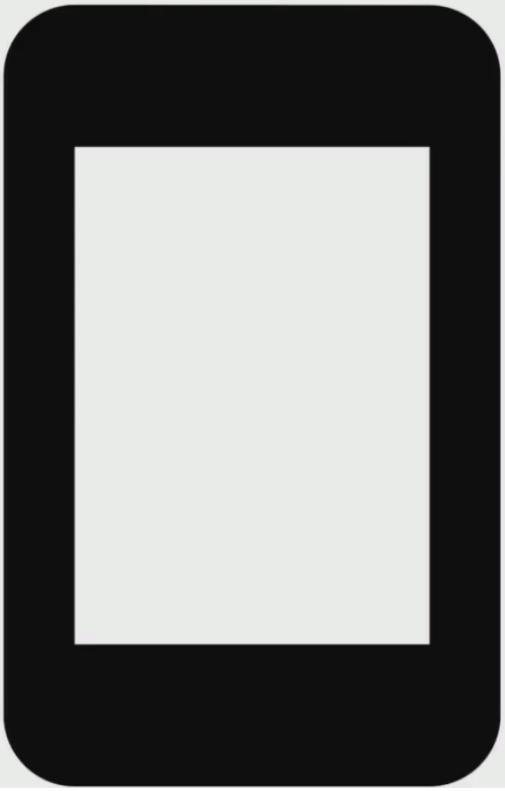
1. GraphQL Introduction
2. Serverless Functions
3. Serverless GraphQL Backends
4. Demo

GraphQL Introduction

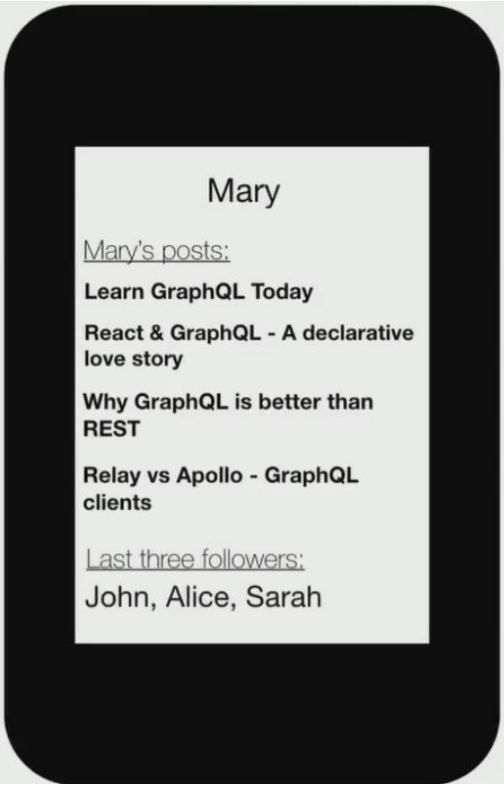
What's GraphQL?

- new **API standard** by Facebook
- **query language** for **APIs**
- **declarative** way of fetching & updating data

Example: **Blogging App**

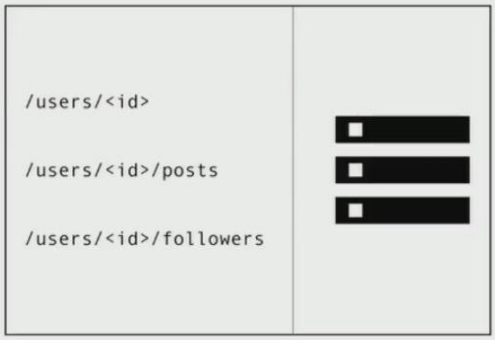


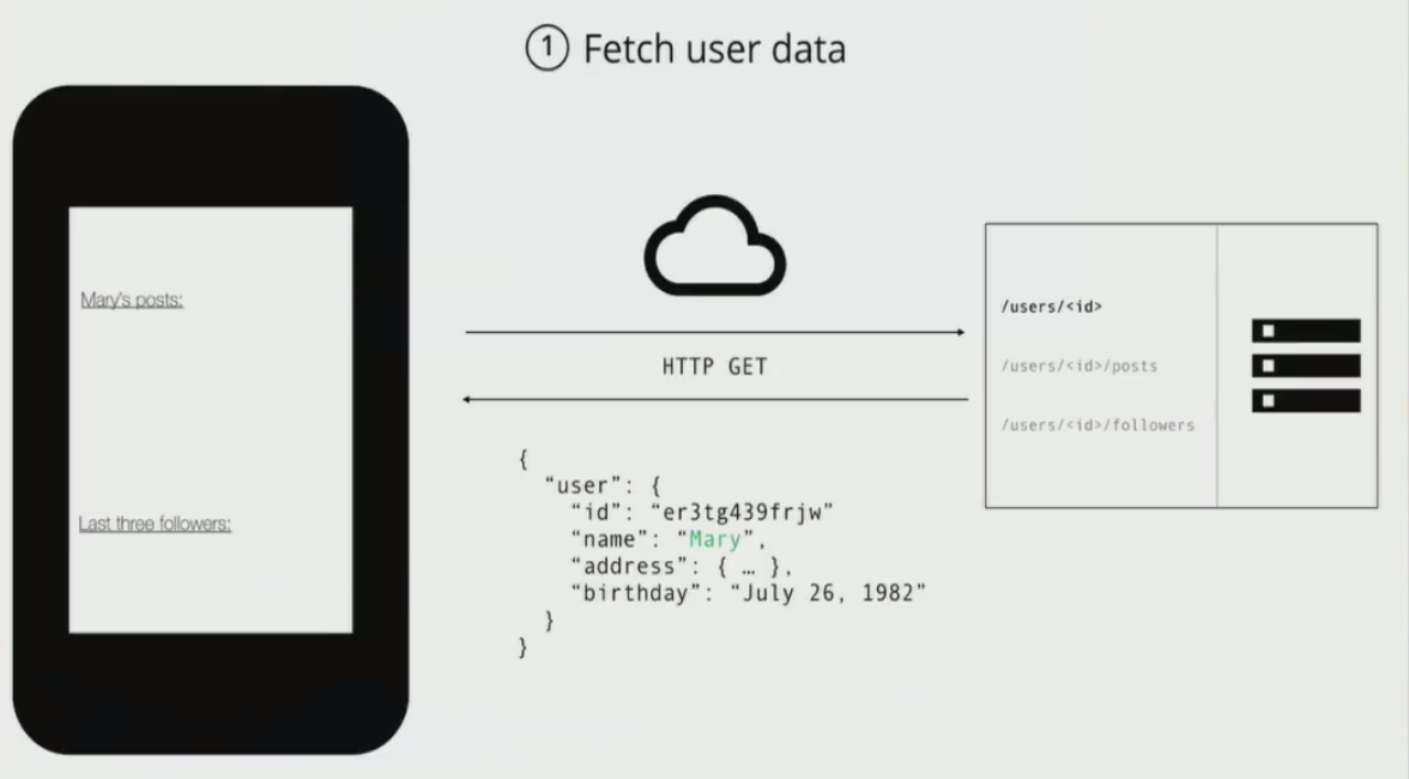
Example: **Blogging App**



Example: Blogging App with REST

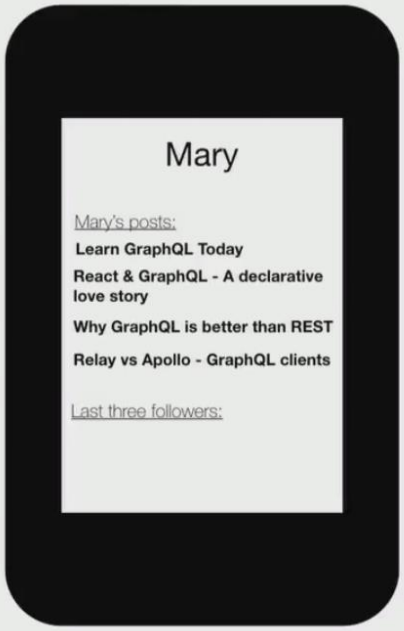
3 API endpoints

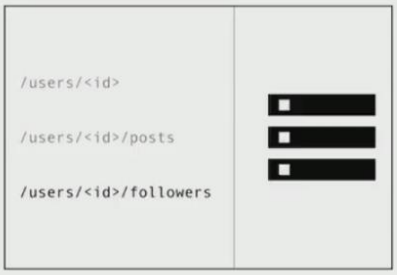




We have to make 3 requests to get the information we need for this page

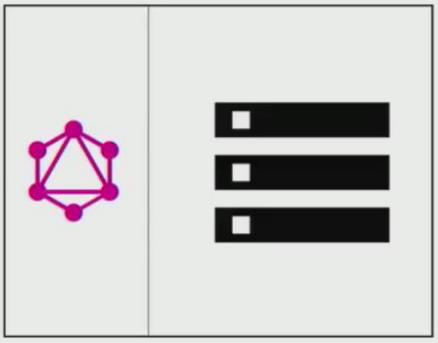
③ Fetch followers



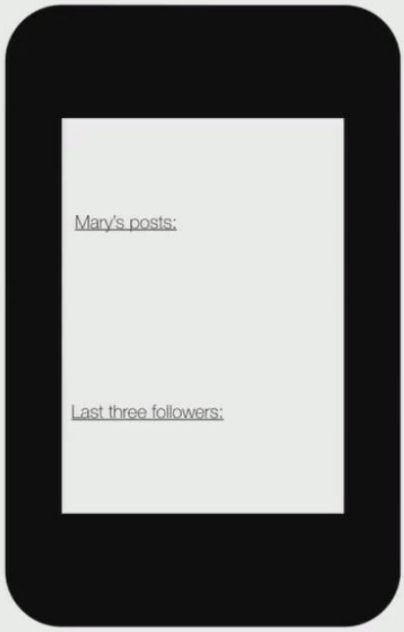


Example: Blogging App with GraphQL

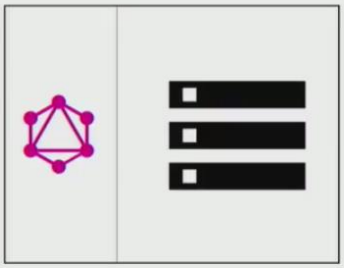
1 API endpoint



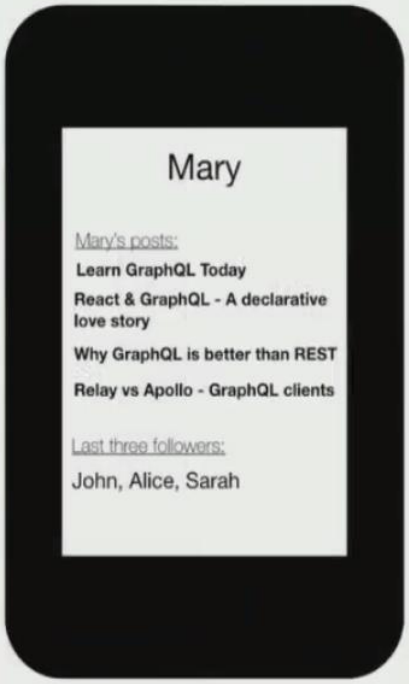
① Fetch everything with a single request



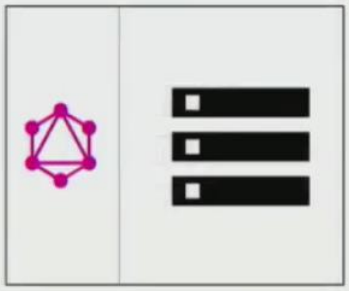

```
query {
  User(id: "er3tg439frjw") {
    name
    posts {
      title
    }
    followers(last: 3) {
      name
    }
  }
}
```



① Fetch everything with a single request




```
{
  "data": {
    "User": {
      "name": "Mary",
      "posts": [
        { title: "Learn GraphQL today" },
        { title: "React & GraphQL - A declarative love story" }
        { title: "Why GraphQL is better than REST" }
        { title: "Relay vs Apollo - GraphQL Clients" }
      ],
      "followers": [
        { name: "John" },
        { name: "Alice" },
        { name: "Sarah" },
      ]
    }
  }
}
```



The GraphQL Schema

- **strongly typed** & written in **Schema Definition Language (SDL)**
- **defines API** (= contract for client-server communication)
- **root types**: `Query, Mutation, Subscription`

Another Example: Chat

```
type Person {
  name: String!
  messages: [Message!]!
}
```

```
type Message {
  text: String!
  sentBy: Person
}
```

Root Types: **Query**

```
{
  Message(id: “1”) {
    text
    sentBy {
      name
    }
  }
}
```

```
type Query {
  Message(id: ID!): Message
}
```

The root types have a special role in the GraphQL schema, we send a message for a particular message with id of 1

Root Types: **Query**

```
{
  allMessages {
    text
    sentBy {
      name
    }
  }
}
```

```
type Query {
  Message(id: ID!): Message
  allMessages: [Message!]!
}
```

Root Types: Mutation

```
mutation {
  createMessage(text:"Hi") {
    id
  }
}
```

```
type Mutation {
  createMessage(text: String!): Message
}
```

Root Types: Mutation

```
mutation {
  updateMessage(
    id: "1",
    text: "Hi"
  ) {
    id
  }
}
```

```
type Mutation {
  createMessage(text: String!): Message
  updateMessage(id: ID!, text: String!): Message
}
```

Root Types: Mutation

```
mutation {
  deleteMessage(id: "1") {
    id
  }
}
```

```
type Mutation {
  createMessage(text: String!): Message
  updateMessage(id: ID!, text: String!): Message
  deleteMessage(id: ID!): Message
}
```

Full* Schema

```
type Query {
  Message(id: ID!): Message
  allMessages: [Message!]!
}

type Mutation {
  createMessage(text: String!): Message
  updateMessage(id: ID!, text: String!): Message
  deleteMessage(id: ID!): Message
}

type Message {
  text: String!
  sentBy: Person
}

type Person {
  name: String!
  messages: [Message!]!
}
```

@nikolasburk

This is the full schema that defines the GraphQL API that allows us to do CRUD functionalities for the Message type

Serverless Functions

Breaking the Monolith 💥

Monolithic Architectures

- simple team structure
- less communication overhead
- global type safety*

- hard to test
- deployment workflows
- bad scalability

Microservices

- solve many problems of the monolith
- new **challenges**:
  - organize and orchestrate the **interplay of services**
  - separating **stateful and stateless** components
  - **dependencies** between microservices

Serverless Functions (FaaS)

- **deploy individual functions**, not servers
- can be invoked via **HTTP webhook**
- **FaaS providers**
  - AWS Lambda
  - Google Cloud Functions...
  - ...

We write just a single function in a JS file as our serverless function to be deployed into a service like Lambda

Using the Graphcool Framework to build **Serverless GraphQL Backends**

The Graphcool Framework

A new level of abstraction for backend development

- automatically generated **CRUD GraphQL API** based on data model
- **event-driven** core to implement business logic
- **global type system** determined by GraphQL schema

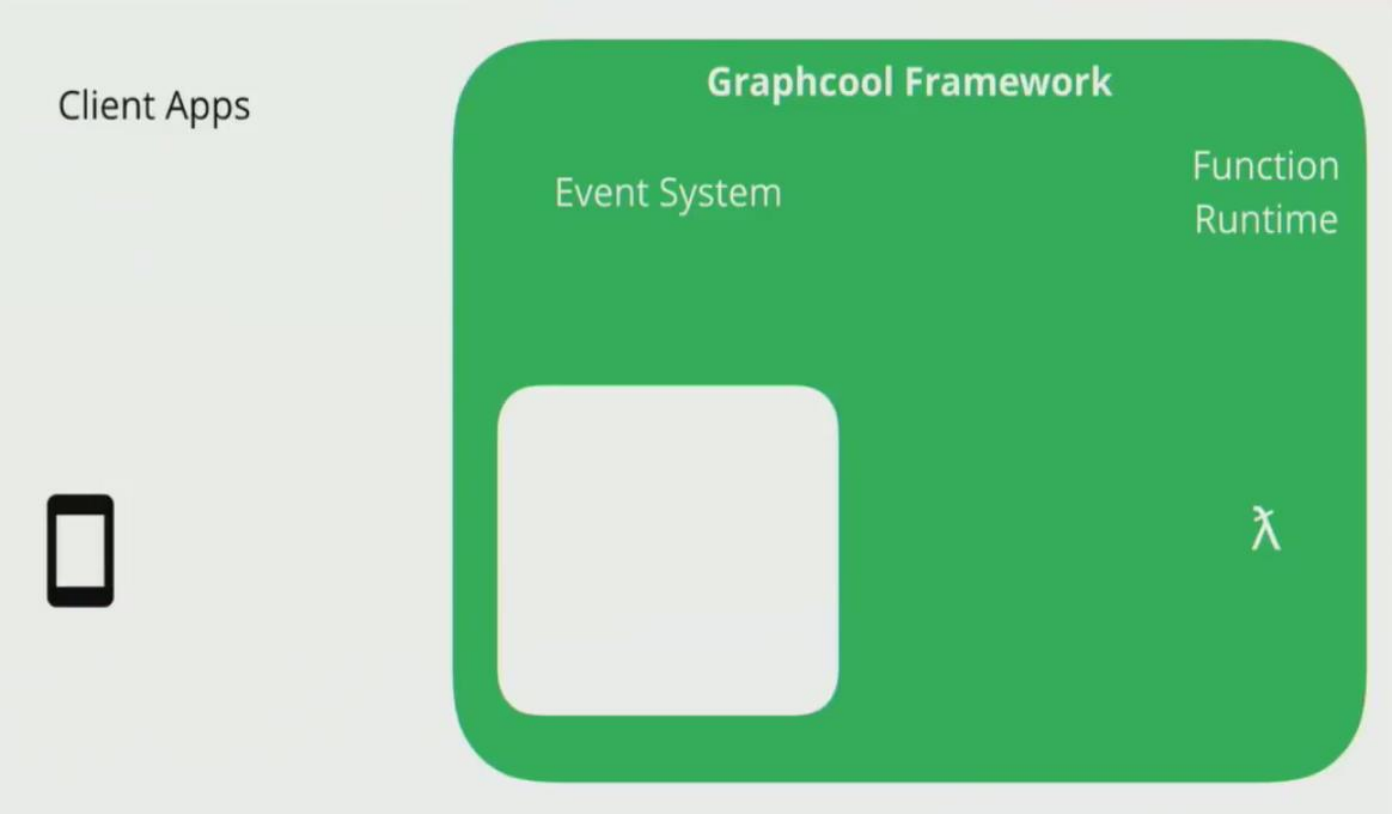
Serverless Functions w/ Graphcool

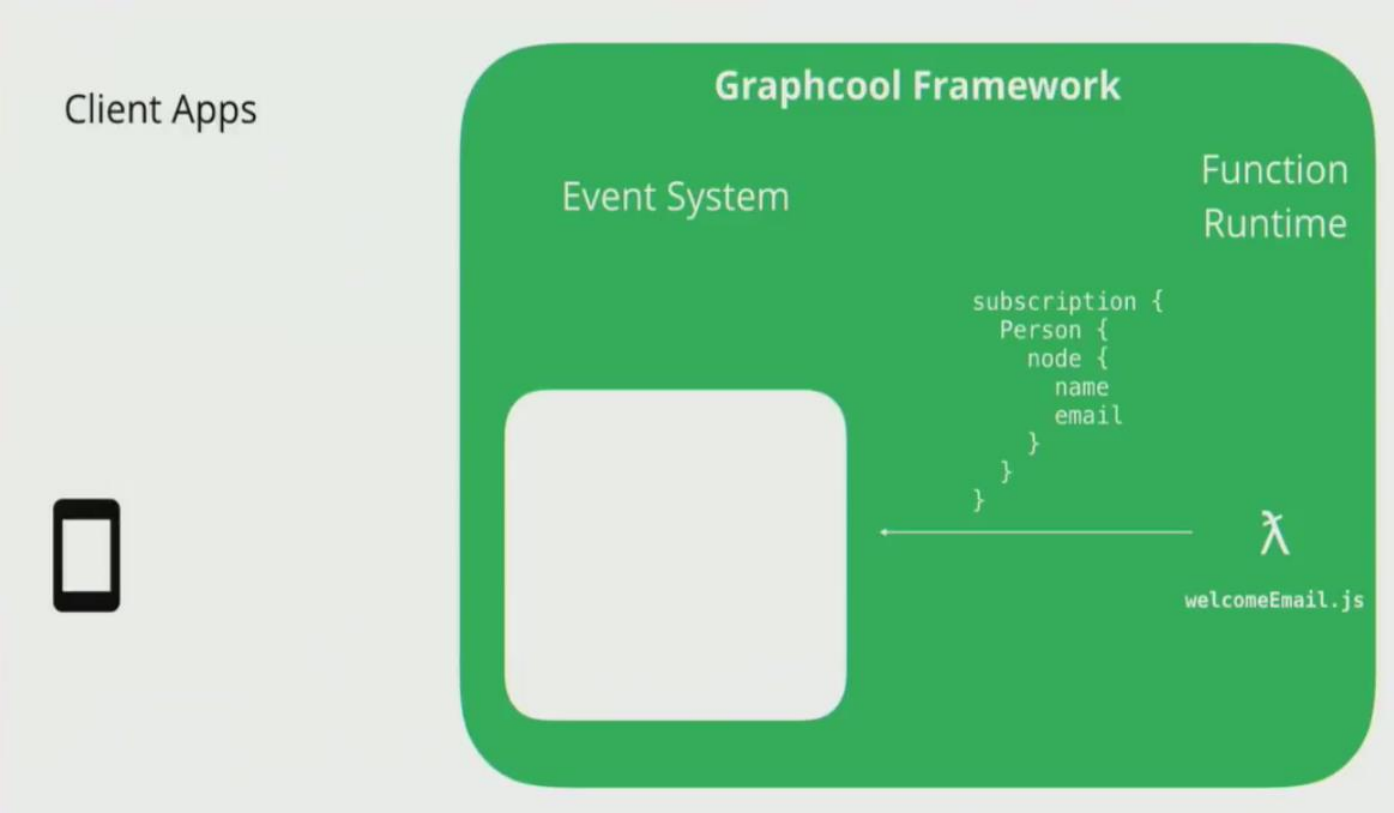
- **Hooks** - synchronous data validation & transformation
- **Subscriptions** - triggering asynchronous events
- **Resolvers** - custom GraphQL queries & mutations

We have 3 types of functions.

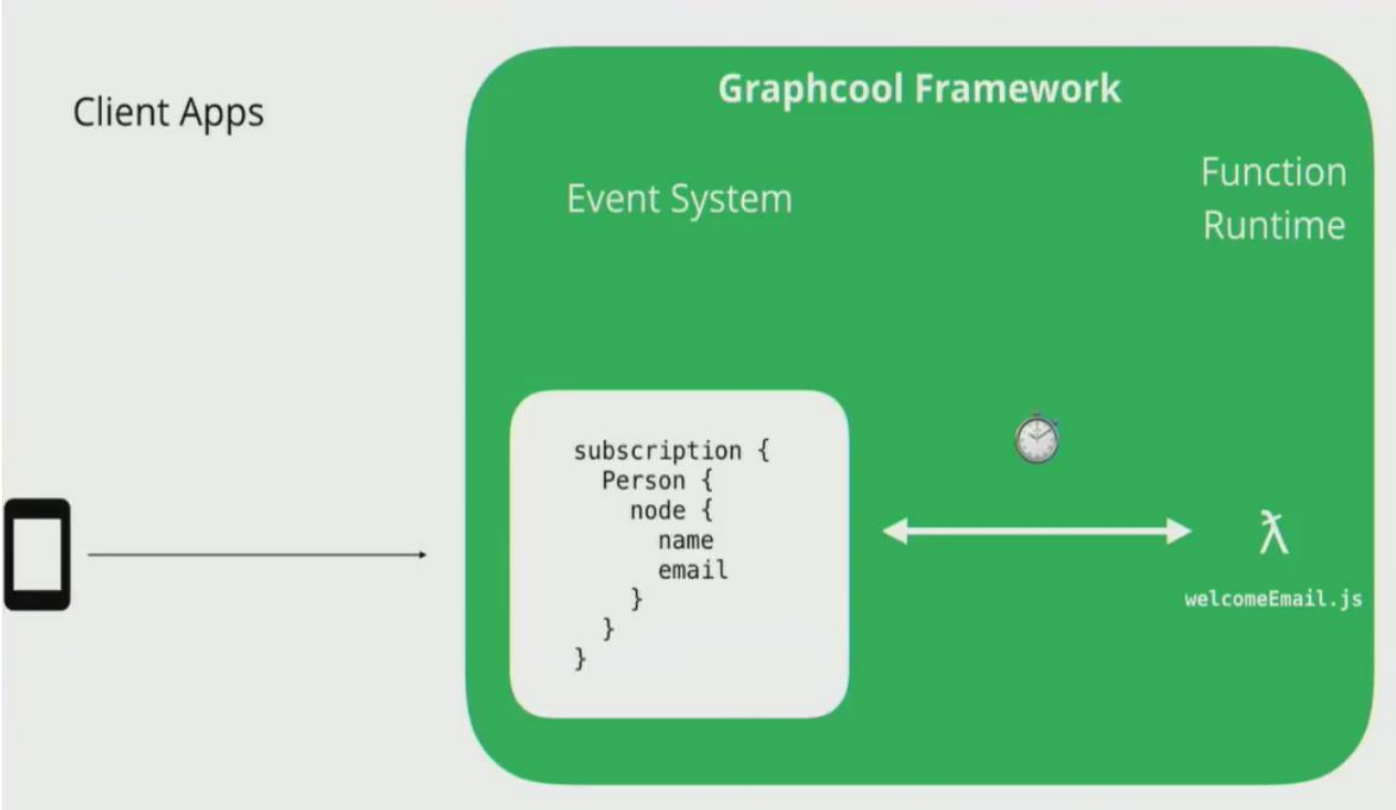
Typesafe Events Example (*Subscription*)

Send Welcome Email to new Users

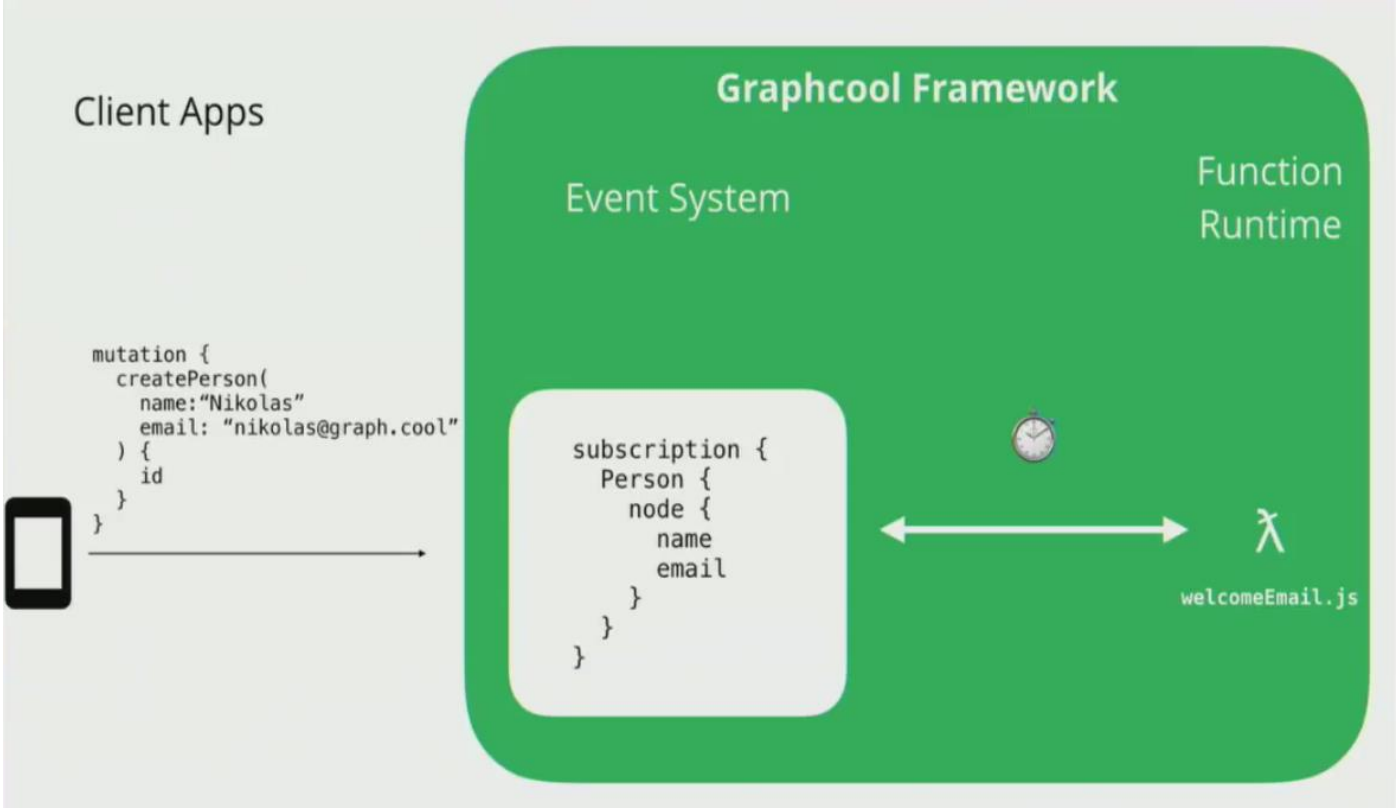




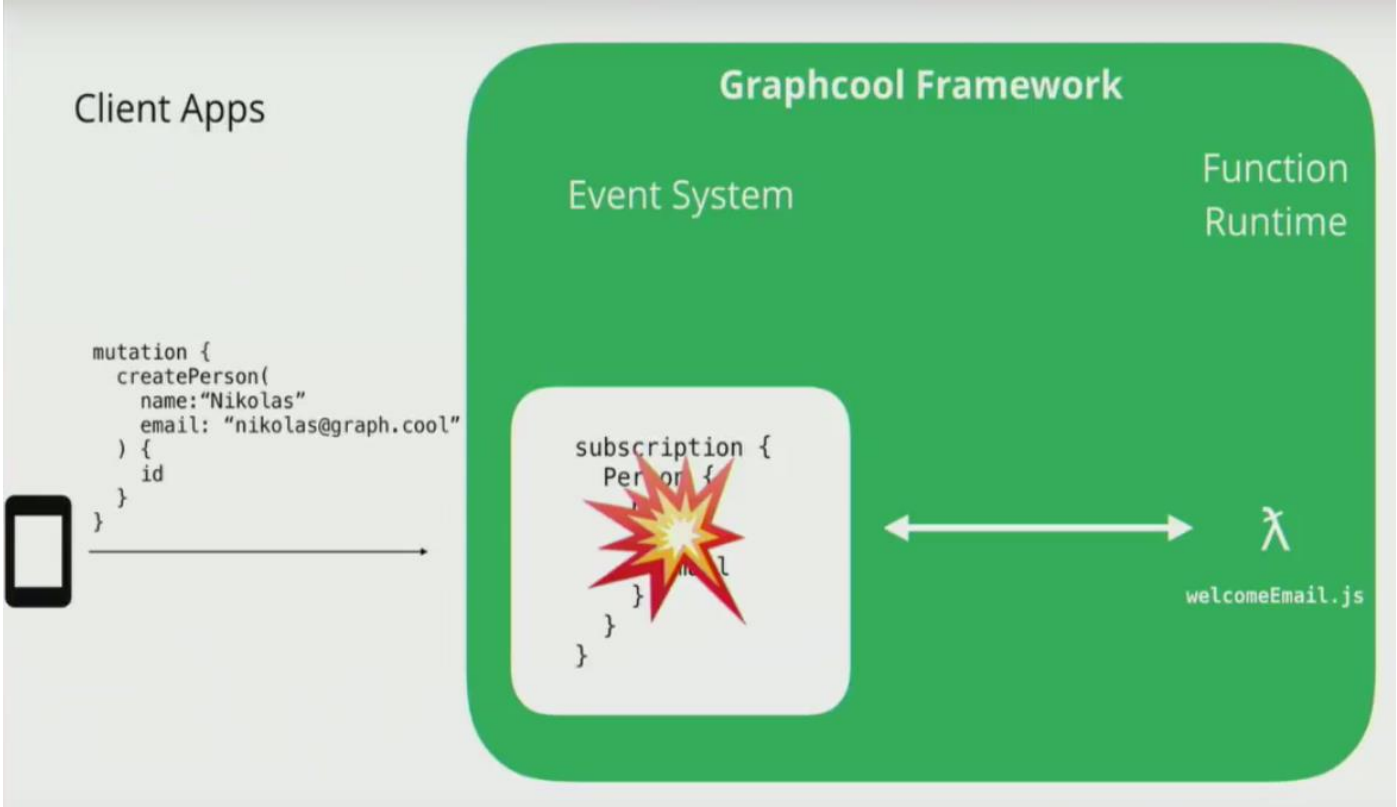
To setup the subscription and be informed of Person* events when new users are created, we have to send over the above GraphQL subscription query to the Event System



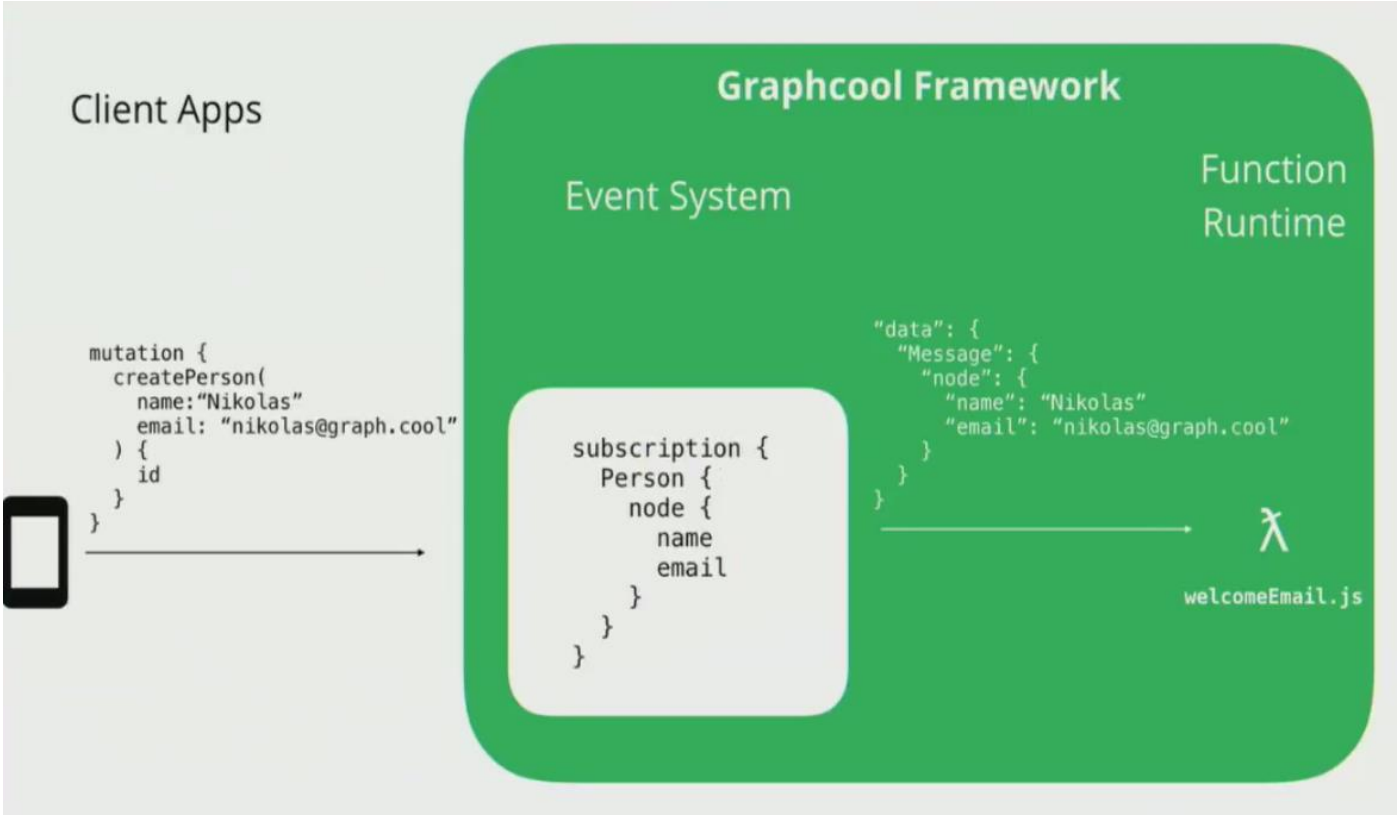
This is now stored in the Event System and will start waiting for a Person* event to happen that relates to the Person type. This could be about a Person update, Person Created, or Person Deleted event.



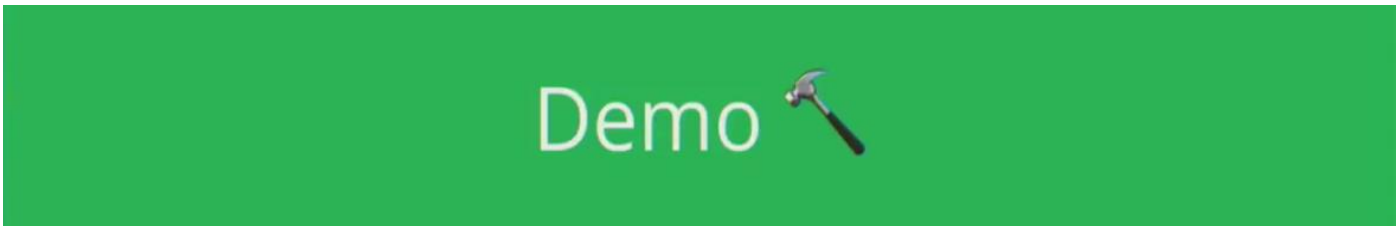
When a person registers, our app sends the createPerson() mutation call to the GraphQL service with the payload above



Now the subscription gets fired and it sends over the payload to the Function Runtime (the lambda function) as below



The welcomeEmail.js lambda function is now triggered with the correct payload and does its work



```
1 types: ./types.graphql
2
3 permissions:
4 - operation: "*"
5
6 functions:
7   welcomeEmail:
8     handler:
9       code:
10        src: ./src/welcomeEmail.js
11    type: subscription
12    query: ./src/welcomeEmail.graphql
13  # randomDogImage:
14  #   handler:
15  #     code:
16  #       src: ./src/randomDogImage.js
17  #   type: resolver
18  #   schema: ./src/randomDogImage.graphql
19  # checkName:
20  #   handler:
```

Let us see how to get started with a GraphQL project. Above is the graphql.yml file where we define the schema. The ***./types.graphql*** types on line 1 define the data model for your application.

```
1 type Person {
2   id: ID! @isUnique
3   createdAt: DateTime!
4   updatedAt: DateTime!
5   name: String!
6   email: String
7 }
8
9
```

The fields with the ! mark are required.

```
nburk@macbook-pro ~/P/t/2/n/demo> ls]
fish: Unknown command 'ls]'
nburk@macbook-pro ~/P/t/2/n/demo> ls
graphcool.yml src          types.graphql
nburk@macbook-pro ~/P/t/2/n/demo> graphcool deploy

Creating project Shardmask Ridge... ✔
Deploying...
```

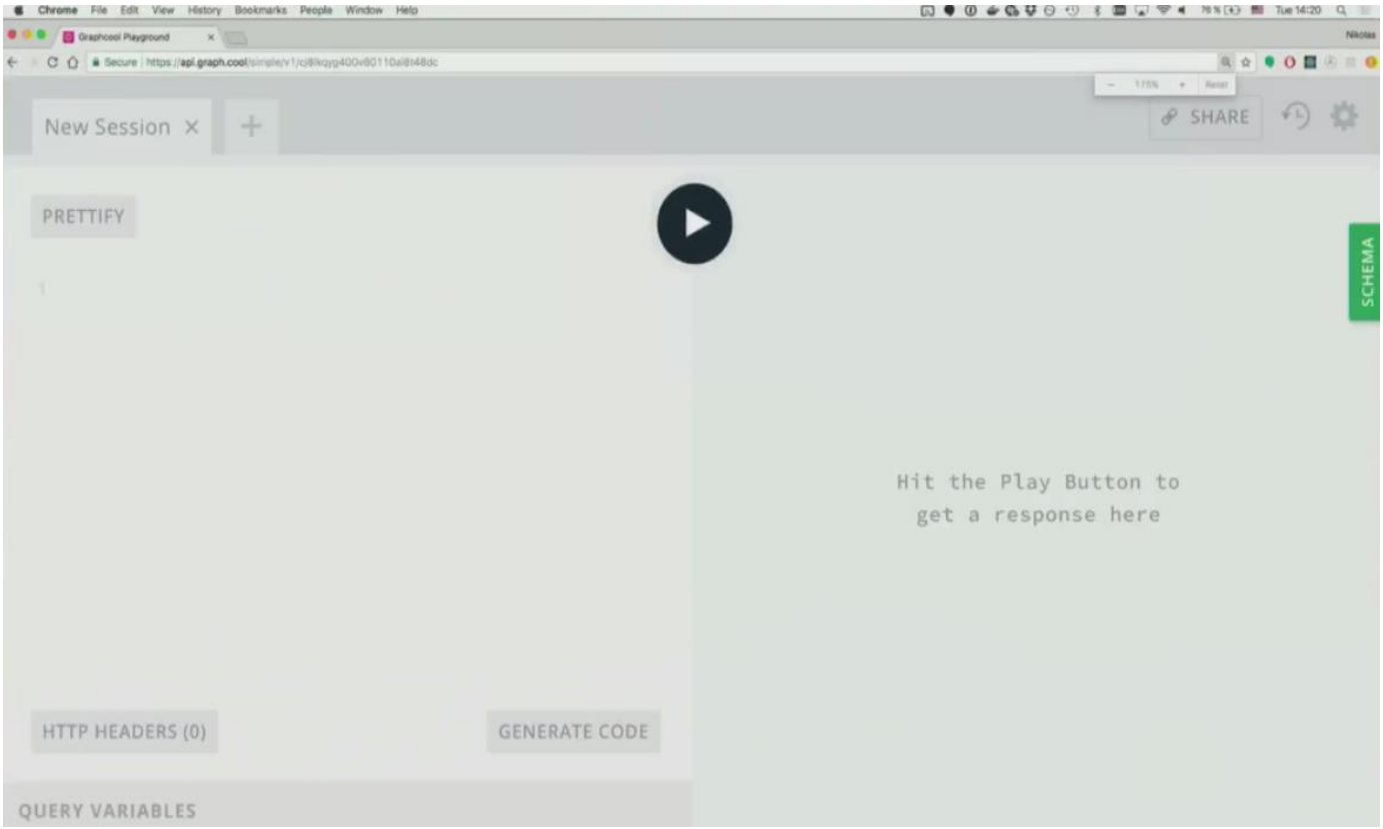
We can now see how to generate the CRUD API for you using the ***graphcool deploy*** command as above

```
Types

  Person
   + A new type with the name `Person` is created.
   ├ +  A new field with the name `createdAt` and type `DateTime!` is created.
   ├ +  A new field with the name `updatedAt` and type `DateTime!` is created.
   ├ +  A new field with the name `name` and type `String!` is created.
   └ +  A new field with the name `email` and type `String` is created.

Subscription Functions

  welcomeEmail
   + A new subscription with the name `welcomeEmail` is created.

Permissions

  Wildcard Permission
   ? The wildcard permission for all types is added.

nburk@macbook-pro ~/P/t/2/n/demo>
```

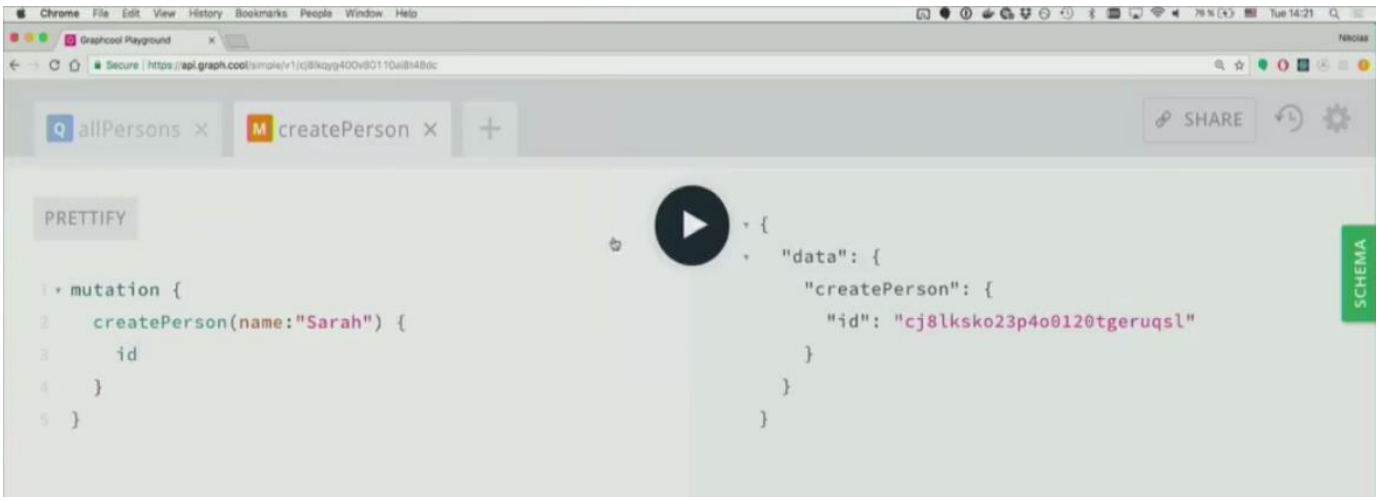
We can test this API

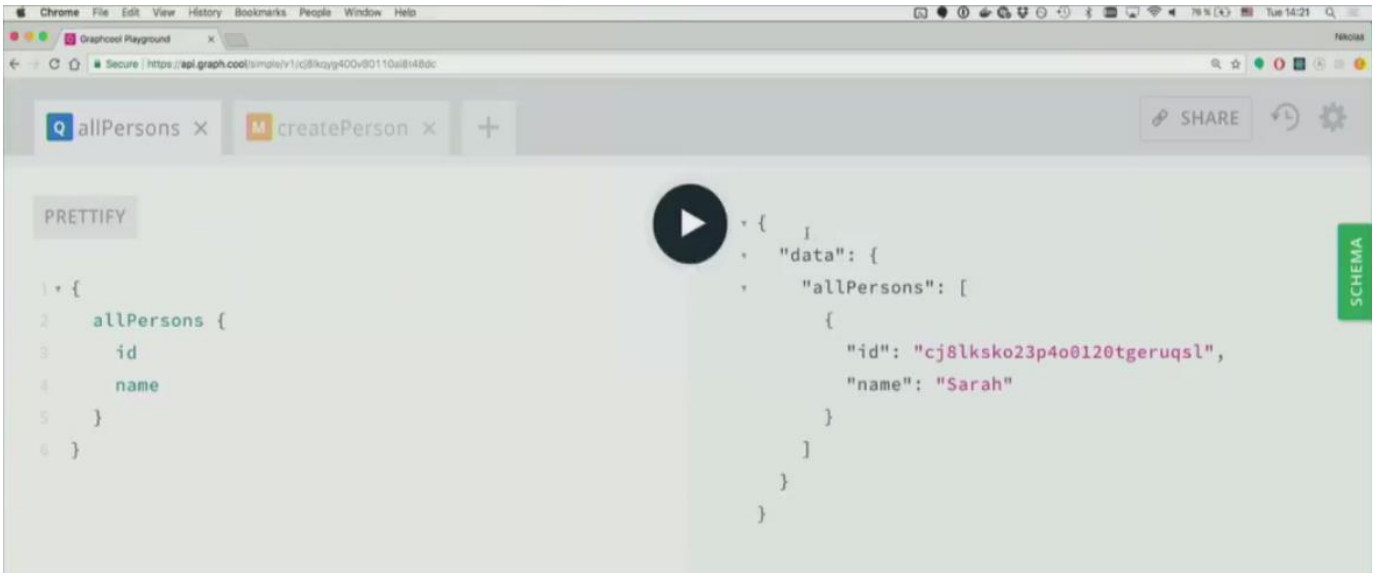
```
  welcomeEmail
   + A new subscription with the name `welcomeEmail` is created.

Permissions

  Wildcard Permission
   ? The wildcard permission for all types is added.

nburk@macbook-pro ~/P/t/2/n/demo> graphcool info

Environment: Project:
------------ ------------------------------------------
dev          Shardmask Ridge (cj8lkqyg400v80110ai8t48dc)

API:           Endpoint:
-------------- ------------------------------------------------------------
Simple         https://api.graph.cool/simple/v1/cj8lkqyg400v80110ai8t48dc
Relay          https://api.graph.cool/relay/v1/cj8lkqyg400v80110ai8t48dc
Subscriptions  wss://subscriptions.graph.cool/v1/cj8lkqyg400v80110ai8t48dc
File           https://api.graph.cool/file/v1/cj8lkqyg400v80110ai8t48dc

nburk@macbook-pro ~/P/t/2/n/demo>
```

We can use the ***graphcool info*** command to see information about the generated API, then we can copy the actual GraphQL API playground endpoint generated for us

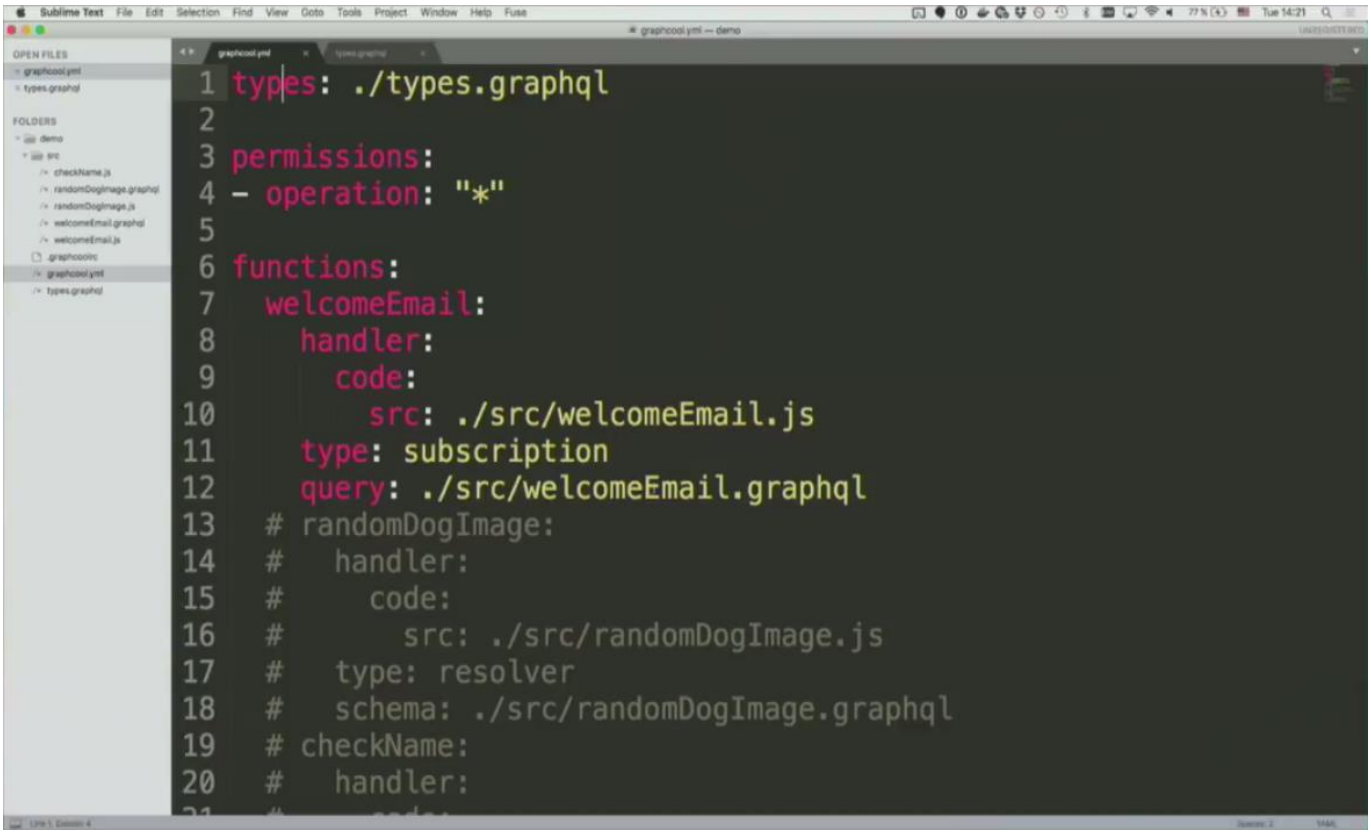


This is the GrapQL Playground where we can explore this GraphQL API





This shows the GraphQL CRUD API that we get just by defining the type in the ***graphql.yml*** configuration file earlier. Next, let us see how we can now integrate serverless functions with this GraphQL API.



Let us see how to send a welcome email to someone who just registered with our app.

```
subscription {
  Person(filter: {
    mutation_in: [CREATED]
  }) {
    node {
      name
      email
    }
  }
}
```

This is the subscription query we are going to use. We are using the filter object to express the fact that we are only interested in CREATED mutations calls. This will prevent this function from getting fired for UPDATED and DELETED mutations too

```
'use-latest'

const fetch = require('isomorphic-fetch')
const Base64 = require('Base64')
const FormData = require('form-data')

const apiKey = 'api:key-d804f7506526ffef735e2623154596e2'
const url = 'https://api.mailgun.net/v3/sandbox3e7f1f0e803e4e5cl

module.exports = event => {

  const customerName = event.data.Person.node.name
  const customerEmail = event.data.Person.node.email

  const form = new FormData()
  form.append('from', 'Graphcool <hello@graph.cool>')
  form.append('to', `${customerName} <${customerEmail}>`)
  form.append('subject', 'Hello from Graphcool')
  form.append('text', `Welcome ${customerName}, great to have yo
```

This is the code for the welcomeEmail.js file where we are taking in the Name and Email of the new user from the form

```
10 module.exports = event => {
11
12   const customerName = event.data.Person.node.name
13   const customerEmail = event.data.Person.node.email
14
15   const form = new FormData()
16   form.append('from', 'Graphcool <hello@graph.cool>')
17   form.append('to', `${customerName} <${customerEmail}>`)
18   form.append('subject', 'Hello from Graphcool')
19   form.append('text', `Welcome ${customerName}, great to have yo
20
21   fetch(url, {
22     headers: {
23       'Authorization': `Basic ${Base64.btoa(apiKey)}`
24     },
25     method: 'POST',
26     body: form
27   })
28
29 }
```

Transactional Email API

Secure | https://www.mailgun.com

mailgun

PRICING FEATURES RESOURCES MANAGED RAPID FIRE CONTACT LOG IN SIGN UP

The Email Service For Developers

Powerful APIs that enable you to send, receive and track email effortlessly.

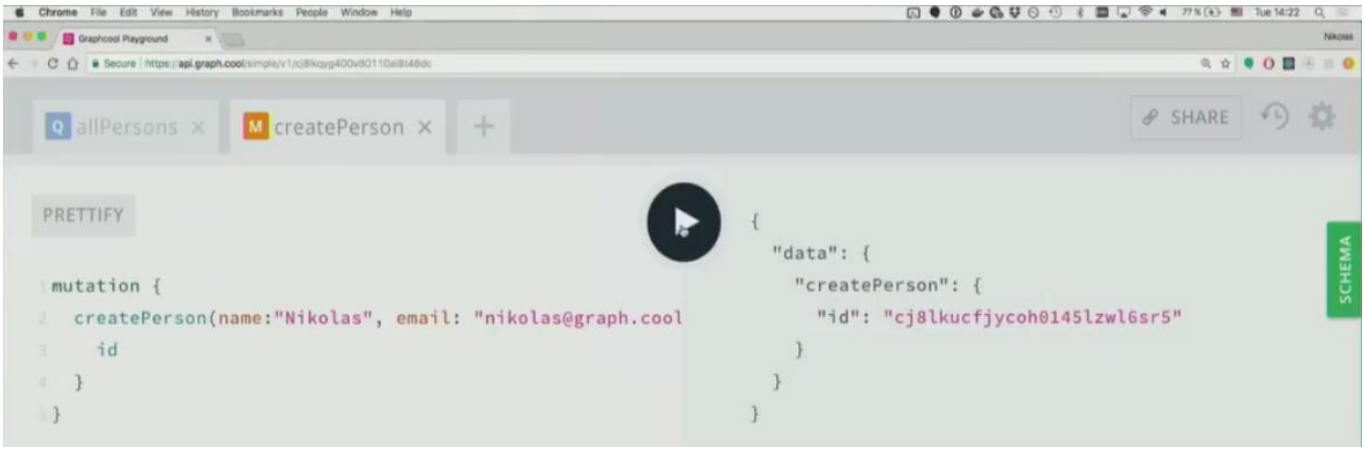
10,000 emails free every month.

Try Mailgun Now

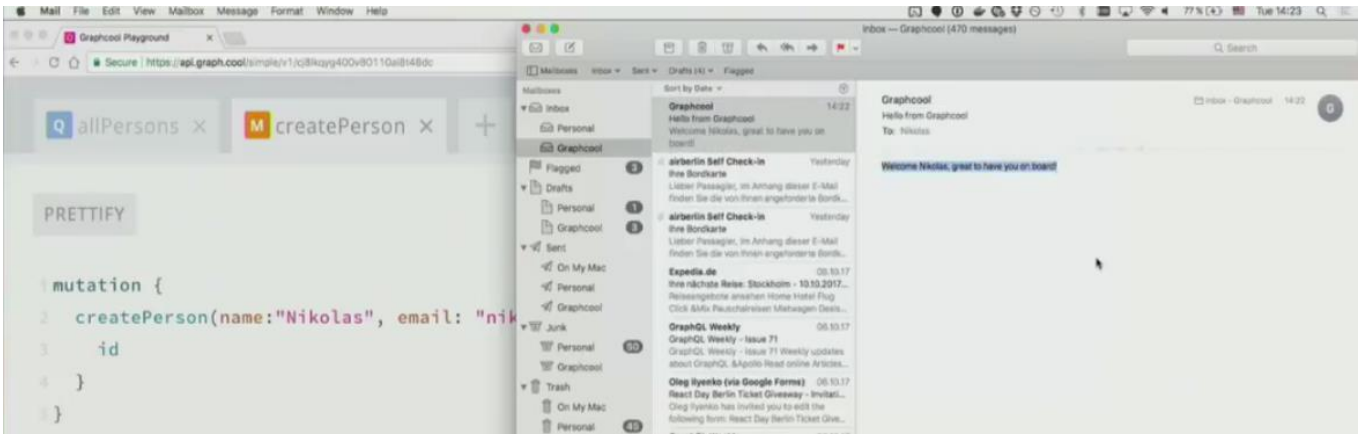
Managed Service For Big Senders

Curl Ruby Python Php Java C#

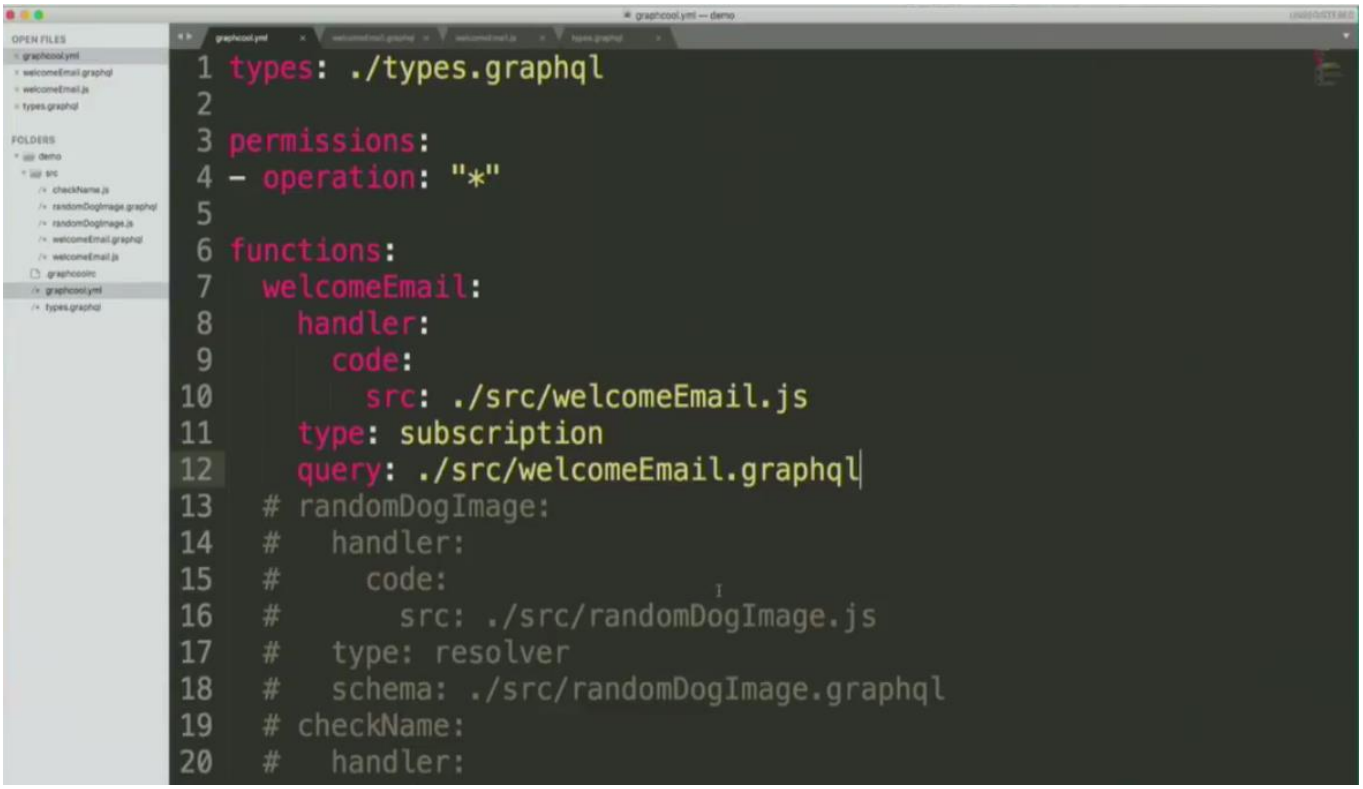
```
# Try our API. Copy & run this in your terminal.
curl -s --user 'api:key-3ax6xnjp29jd6fds4gc373sgvjxteol0' \
    https://api.mailgun.net/v3/samples.mailgun.org/messages \
    -F from='Excited User <excited@samples.mailgun.org>' \
    -F to='devs@mailgun.net' \
    -F subject='Hello' \
    -F text='Testing some Mailgun awesomeness!'
```



We then can use the playground to trigger a create event for a new user



We can confirm that the trigger worked and an email was sent with the specified greeting. This is how we can integrate subscriptions for real time events



Let us now see a use case where we want to put a GraphQL layer on top of existing REST API layer

```yaml
- operation: "*"

functions:
  welcomeEmail:
    handler:
      code:
        src: ./src/welcomeEmail.js
    type: subscription
    query: ./src/welcomeEmail.graphql
  randomDogImage:
    handler:
      code:
        src: ./src/randomDogImage.js
    type: resolver
    schema: ./src/randomDogImage.graphql
  # checkName:
  #   handler:
  #     code:
  #       src: ./src/checkName.js
```

We have defined a resolver function that uses the following query

```
type RandomDogImagePayload {
  url: String!
}

extend type Query {
  randomDogImage: RandomDogImagePayload!
}
```

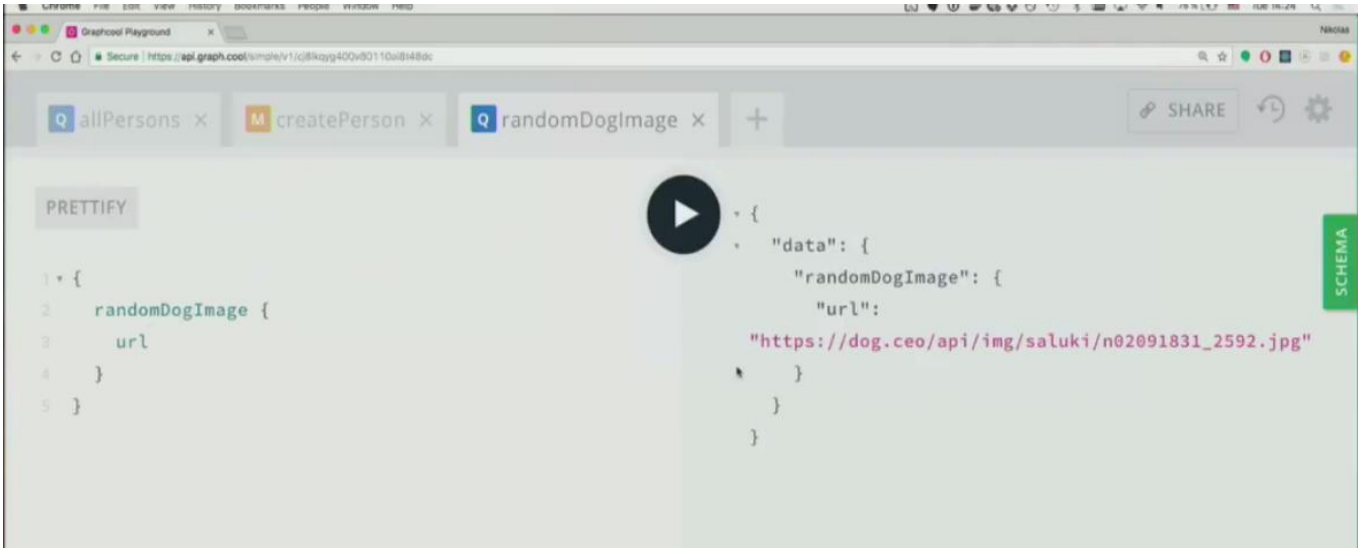
We want to retrieve a random dog Image from a public dog image API. ***randomDogImage*** is the root field that we have to call for that API.

```javascript
require('isomorphic-fetch')

const url = 'https://dog.ceo/api/breeds/image/random'

module.exports = () => {
  return fetch(url)
    .then(response => response.json())
    .then(responseData => {
      const randomDogImageData = responseData.message
      const randomDogImage = { url: randomDogImageData }
      return { data: randomDogImage }
    })
}
```

This is the implementation with the HTTP endpoint that we are effectively wrapping with our GraphQL API right now

```
Simple          https://api.graph.cool/simple/v1/cj8lkqyg400v80110ai8t48dc
Relay           https://api.graph.cool/relay/v1/cj8lkqyg400v80110ai8t48dc
Subscriptions   wss://subscriptions.graph.cool/v1/cj8lkqyg400v80110ai8t48dc
File            https://api.graph.cool/file/v1/cj8lkqyg400v80110ai8t48dc

nburk@macbook-pro ~/P/t/2/n/demo> graphcool deploy
Deploying to cj8lkqyg400v80110ai8t48dc with env dev...
```

We then run the ***graphcool deploy*** command again to activate this function



We can now go to the playground and call the GraphQL endpoint used to wrap the REST endpoint to get the randomImage URL we want

The result works

```
4 - operation: "*"
5
6 functions:
7   welcomeEmail:
8     handler:
9       code:
10        src: ./src/welcomeEmail.js
11    type: subscription
12    query: ./src/welcomeEmail.graphql
13  randomDogImage:
14    handler:
15      code:
16        src: ./src/randomDogImage.js
17    type: resolver
18    schema: ./src/randomDogImage.graphql
19  # checkName:
20  #   handler:
21  #     code:
22  #       src: ./src/checkName.js
23  #   type: operationBefore
```

We can also do data validation in GraphQL

Big news coming up!

Stay tuned... 😏

Thank you! 🙇‍♀️