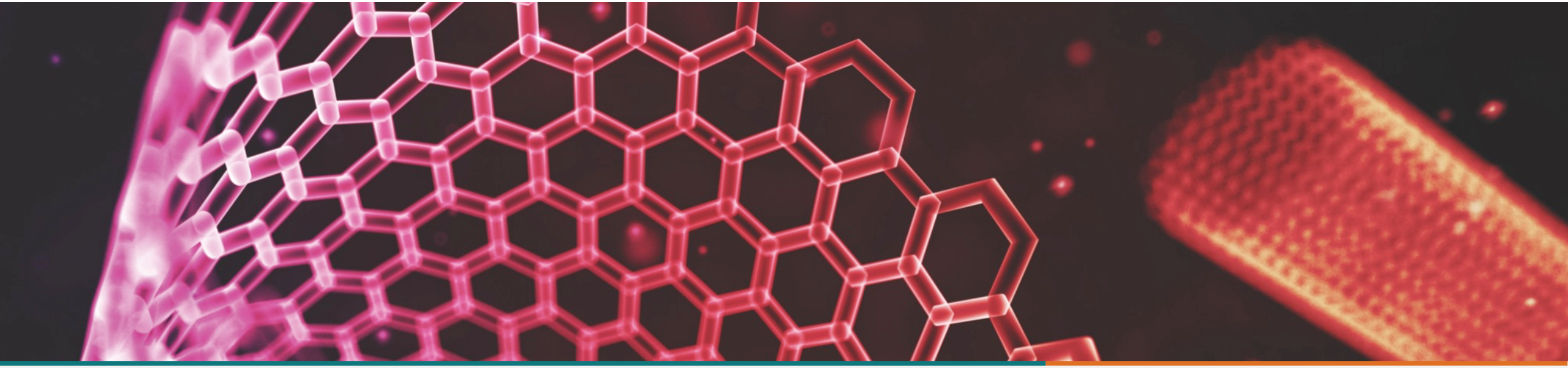
STEVENS
INSTITUTE of TECHNOLOGY
Schaefer School of Engineering & Science

CS 554 – Web Programming II

GraphQL



STEVENS
INSTITUTE of TECHNOLOGY

Schaefer School of
Engineering & Science

stevens.edu

Patrick Hill
Adjunct Professor
Computer Science Department
Patrick.Hill@stevens.edu

What is GraphQL?

GraphQL is an open source query language created by Facebook.

Before GraphQL went open source in 2015, Facebook used it internally for their mobile applications since 2012

We run GraphQL as a server which takes requests for data.

A query language like GraphQL on the server-side and client-side lets the client decide which data it needs by making a single request to the server. We only have one endpoint as opposed to REST where you have many endpoints.

The ecosystem around GraphQL is growing horizontally by offering multiple programming languages, but also vertically, with libraries on top of GraphQL like Apollo and Relay.

We will be using Apollo Server and Apollo Client for our lecture examples.

What is GraphQL?

A GraphQL operation is either a:

- **query** (read)
- **mutation** (write)
- **subscription** (continuous read).

Each of those operations is only a string that needs to be constructed according to the GraphQL query language specification. Fortunately, GraphQL is evolving all the time, so there may be other operations in the future.

Once this GraphQL operation reaches the backend application, it can be interpreted against the entire GraphQL schema there and resolved with data for the frontend application. GraphQL is not opinionated about the network layer, which is often HTTP, nor about the payload format, which is usually JSON. It isn't opinionated about the application architecture at all. It is only a query language.

REST Example

Say we have an end point:

http://www.patrickhill.nyc/books/:id

This returns: id, title, genre, reviews, summary, authorId

And an endpoint:

http://www.patrickhill.nyc/authors/:id

This returns: id, name, age, bio, bookIds

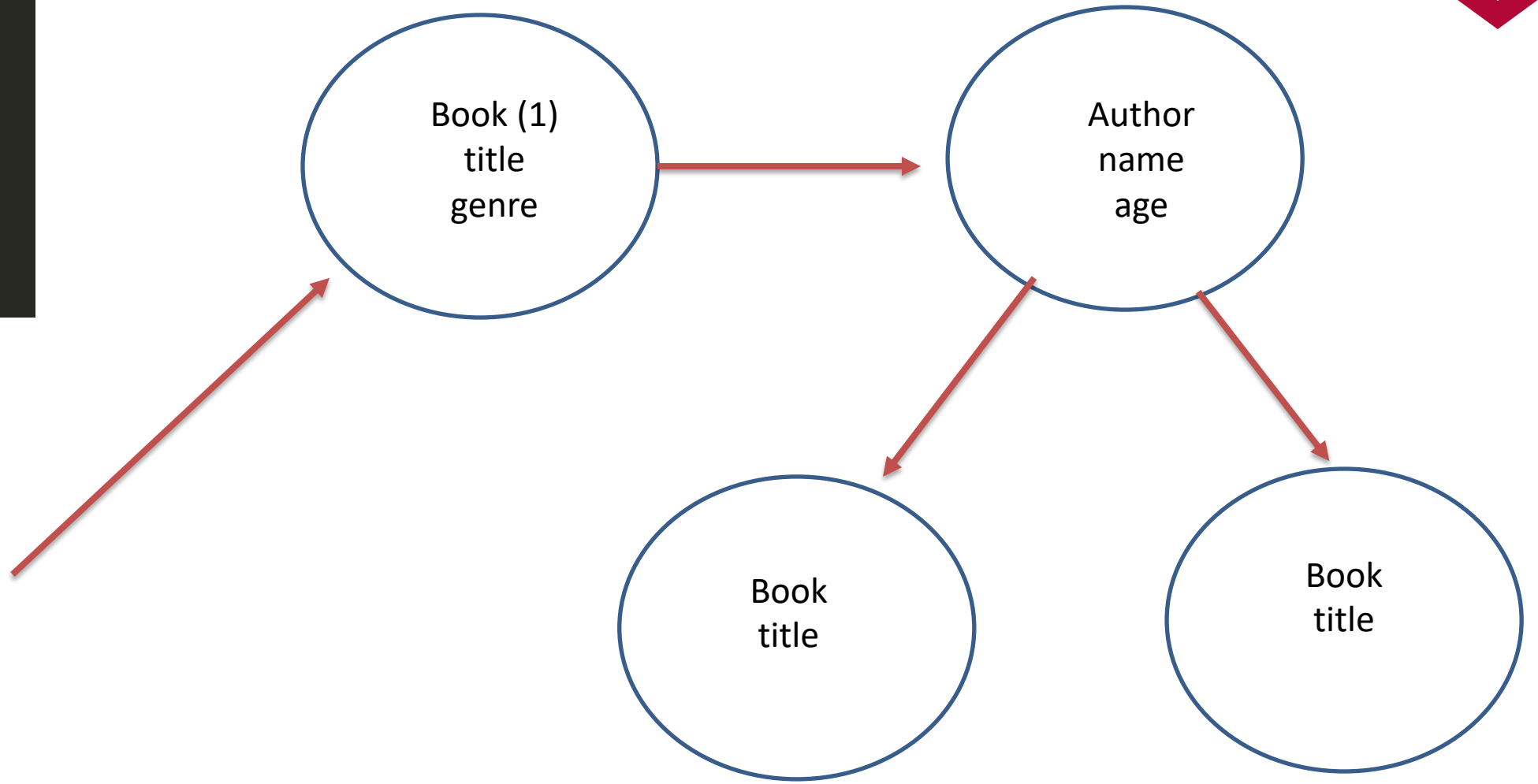
GraphQL Query

```
book(id: “729ff4a0-ed20-4aa3-b64b-6b5a2c120916”) {
    title
    genre
    author {
        name
        books(limit 2){
            name
        }
    }
}
```

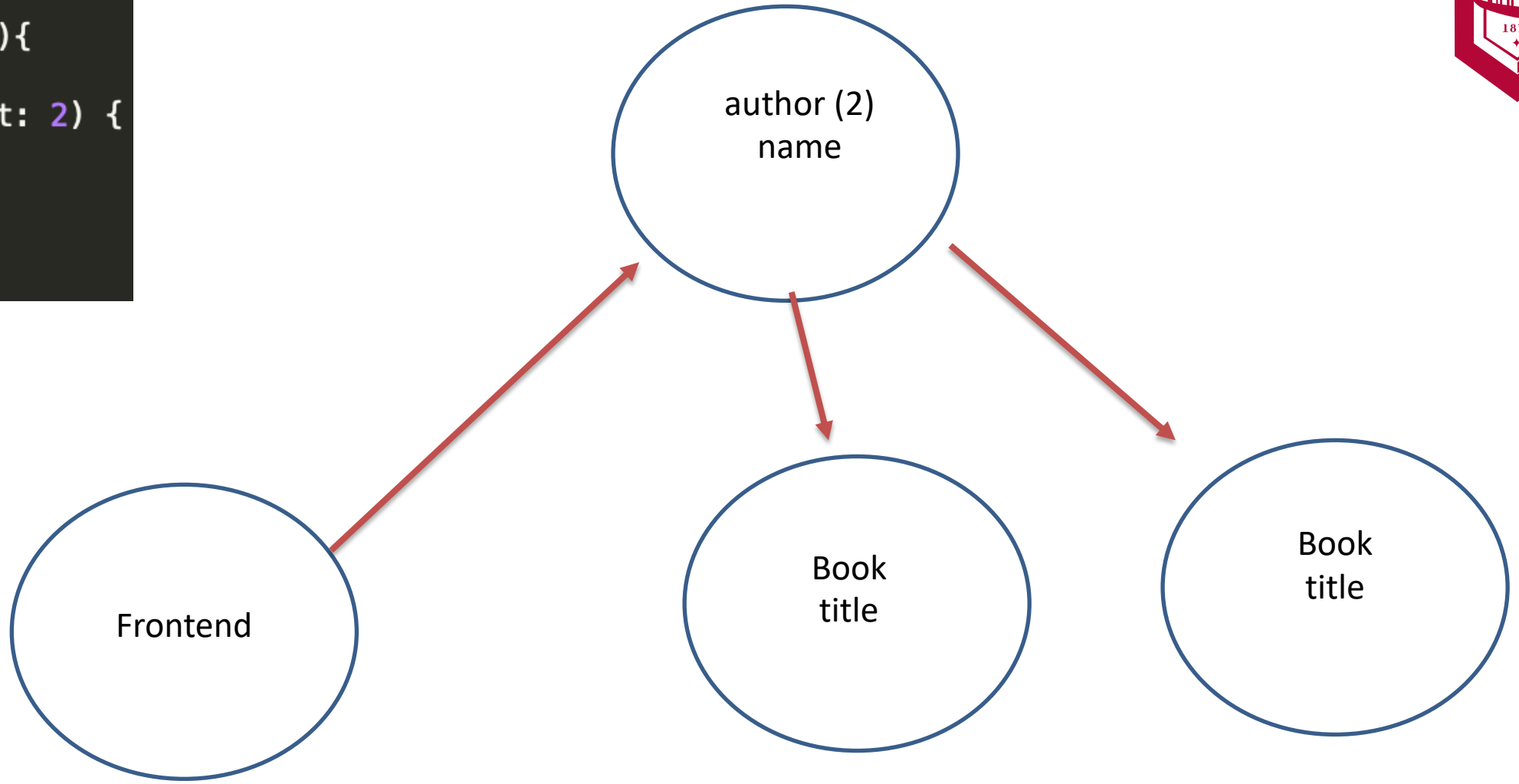
GraphQL Results


```
{
   "data": {
      ”book": {
         ”title": ”The Shining",
         ”genre": ”Fiction/Horror",
         ”author":{
        ”name": “Stephen King”
            ”books": [
          {
                  ”name": “Pet Sematary”
              },
          {
                  ”name": “Christine”
              }
            [
        }
      }
   }
}
```

```
{
  book (id: 1){
    title
    genre
    author: {
      name
      age
      books (limit: 2){
        title
      }
    }
  }
}
```



```
{
  author (id: 2){
    name
    books (limit: 2) {
      title
    }
  }
}
```



GraphQL Advantages

Declarative Data Fetching

As you've seen, GraphQL embraces declarative data fetching with its queries. The client selects data along with its entities with fields across relationships in one query request. GraphQL decides which fields are needed for its UI, and it almost acts as UI-driven data fetching

GraphQL Advantages

No Overfetching with GraphQL

There is no overfetching in GraphQL. A mobile client usually overfetches data when there is an identical API as the web client with a RESTful API. With GraphQL, the mobile client can choose a different set of fields, so it can fetch only the information needed for what's onscreen.

GraphQL Advantages

GraphQL for React, Angular, Node and Co.

GraphQL is decoupled from any frontend or backend solution. The reference implementation of GraphQL is written in JavaScript, so the usage of GraphQL in Angular, Vue, Express, Hapi, Koa and other JavaScript libraries on the client-side and server-side is possible, and that's just the JavaScript ecosystem. GraphQL does mimic REST's programming language-agnostic interface between two entities, such as client or server.

GraphQL Advantages

Single Source of Truth

The GraphQL schema is the single source of truth in GraphQL applications. It provides a central location, where all available data is described. The GraphQL schema is usually defined on server- side, but clients can read (query) and write (mutation) data based on the schema.

Essentially, the server-side application offers all information about what is available on its side, and the client-side application asks for part of it by performing GraphQL queries, or alters part of it using GraphQL mutations.

GraphQL Advantages

GraphQL embraces modern Trends

GraphQL embraces modern trends on how applications are built. You may only have one backend application, but multiple clients on the web, phones, and smartwatches depending on its data.

GraphQL Advantages

GraphQL Schema Stitching

Schema stitching makes it possible to create one schema out of multiple schemas. Think about a microservices architecture for your backend where microservice handles the business logic and data for a specific domain. In this case, each microservice can define its own GraphQL schema, after which you'd use schema stitching to weave them into one that is accessed by the client. Each microservice can have its own GraphQL endpoint, where one GraphQL API gateway consolidates all schemas into one global schema.

GraphQL Advantages

Strongly Typed GraphQL

GraphQL is a strongly typed query language because it is written in the expressive GraphQL Schema Definition Language (SDL). Being strongly-typed makes GraphQL less error prone, can be validated during compile-time and can be used for supportive IDE/editor integrations such as auto-completion and validation.

GraphQL Advantages

GraphQL Versioning

In GraphQL there are no API versions as there used to be in REST. In REST it is normal to offer multiple versions of an API (e.g. api.domain.com/v1/, api.domain.com/v2/), because the resources or the structure of the resources may change over time. In GraphQL it is possible to deprecate the API on a field level. Thus a client receives a deprecation warning when querying a deprecated field. After a while, the deprecated field may be removed from the schema when not many clients are using it anymore. This makes it possible to evolve a GraphQL API over time without the need for a versioning.

GraphQL Disadvantages

GraphQL Query Complexity

People often mistake GraphQL as a replacement for server-side databases, but it’s just a query language. Once a query needs to be resolved with data on the server, a GraphQL agnostic implementation usually performs database access. GraphQL isn’t opinionated about that.

GraphQL Disadvantages

GraphQL Rate Limiting

Another problem is rate limiting. Whereas in REST it is simpler to say, “we allow only so many resource requests in one day”, it becomes difficult to make such a statement for individual GraphQL operations, because it can be everything between a cheap or expensive operation. That’s where companies with public GraphQL APIs come up with their specific rate limiting calculations which often boil down to the previously mentioned maximum query depths and query complexity weighting.

GraphQL Disadvantages

GraphQL Caching

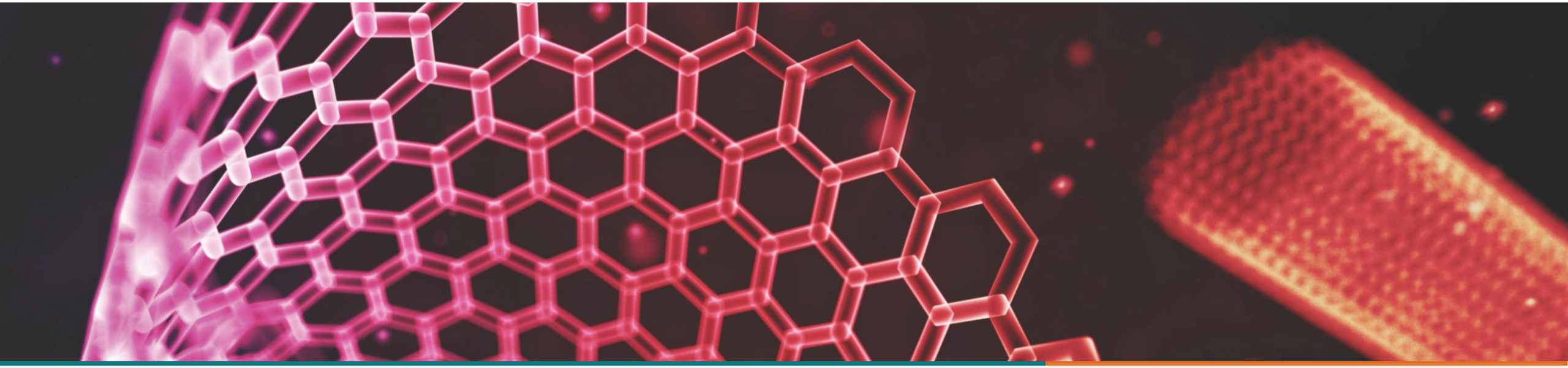
Implementing a simplified cache with GraphQL is more complex than implementing it in REST. In REST, resources are accessed with URLs, so you can cache on a resource level because you have the resource URL as identifier. In GraphQL, this becomes complex because each query can be different, even though it operates on the same entity. You may only request just the name of an author in one query, but want to know the email address in the next. That's where you need a more fine-grained cache at field level, which can be difficult to implement. However, most of the libraries built on top of GraphQL offer caching mechanisms out of the box.

Why Not REST?

GraphQL is an alternative to the commonly used RESTful architecture that connects client and server applications. Since REST comes with a URL for each resource, it often leads to inefficient waterfall requests. For instance, imagine you want to fetch an author entity identified by an id, and then you fetch all the articles by this author using the author's id. In GraphQL, this is a single request, which is more efficient. If you only want to fetch the author's articles without the whole author entity, GraphQL lets you to select only the parts you need. In REST, you would overfetch the entire author entity.

There are still cases where REST is a valuable approach for connecting client and server applications, though. Applications are often resource-driven and don't need a flexible query language like GraphQL. However, I recommend you to give GraphQL a shot when developing your next client server architecture to see if it fits your needs.

Installing GraphQL



Installing GraphQL

We need to install the GraphQL and GraphQL CLI(optional) global packages

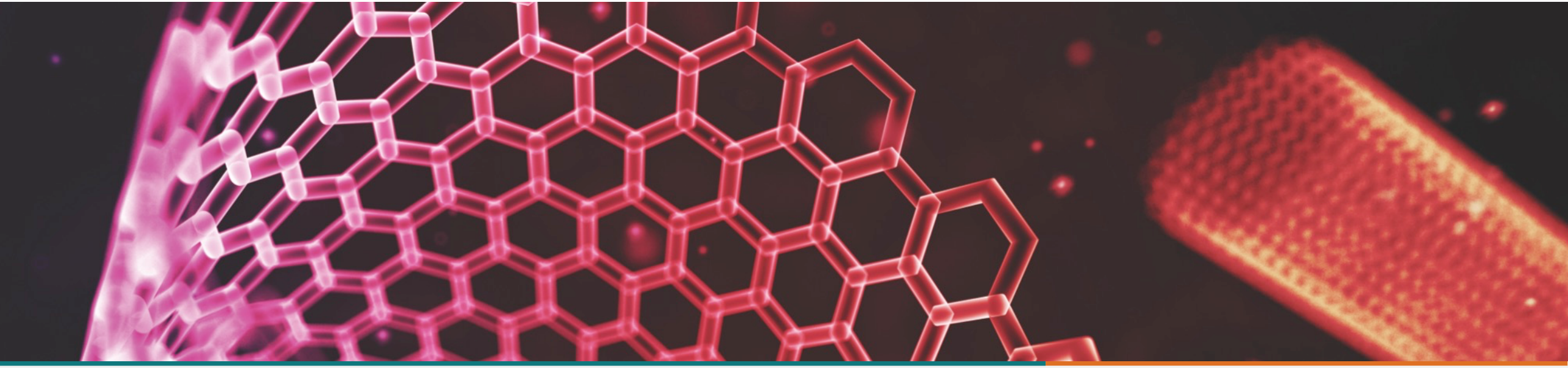
- ***npm install –g graphql-cli***
- ***npm install –g graphql***

We need to also install the Apollo global package

- ***npm install –g apollo***

We are going to have node two projects. One for the server and one for the client.

GraphQL Server With Apollo



Installing Apollo Server

We need to install the **apollo-server** npm package locally into our project

- ***npm install apollo-server***

We also need to install the graphql npm package locally into our project

- ***npm install graphql***

We only need one JavaScript file for this example: **index.js**

Importing Apollo Server

In index.js we import Apollo Server and gql

```
const { ApolloServer, gql } = require('apollo-server');
```

We then need to set up a few things for our GraphQL server

- **Type Definitions**
- **Resolvers**

then we just run our server and listen for requests!

Type Definitions

We create type definitions for our queries, mutations and our data

We wrap our typeDefs into a gql template string like so:

```
const typeDefs = gql`
# typedefs go here
`
```

```
const typeDefs = gql`
type Query {
    employers: [Employer]
    employees: [Employee]
    employer(id: Int): Employer
    employee(id: String): Employee
  }

  type Employer {
    id: Int
    name: String
    employees: [Employee]
    numOfEmployees: Int
  }
`
```

Resolvers

A resolver is a function that's responsible for populating the data for a single field in your schema. It can populate that data in any way you define, such as by fetching data from a back-end database or a third-party API. Our example this week is just using some static data but normally in the resolver, you would call your DB functions to get your data from the Database. Just as our routes in Express called our DB functions. The only difference is we only need one "API" route instead of defining many routes!

Resolvers

```
const resolvers = {
  Query: {
    employer: (_, args) => employers.filter((e) => e.id === args.id)[0],
    employee: (_, args) => employees.filter((e) => e.id === args.id)[0],
    employers: () => employers,
    employees: () => employees
  },
  Employer: {
    numOfEmployees: (parentValue) => {
      console.log(`parentValue in Employer`, parentValue);
      return employees.filter((e) => e.employerId === parentValue.id).length;
    },
    employees: (parentValue) => {
      return employees.filter((e) => e.employerId === parentValue.id);
    }
  },
  Employee: {
    employer: (parentValue) => {
      return employers.filter((e) => e.id === parentValue.employerId)[0];
    }
  },
```

parentValue - References the type def that called it

so for example when we execute numOfEmployees we can reference the parent's properties with the parentValue Paramater

args - Used for passing any arguments in from the client for example, when we call **addEmployee(firstName: String!, lastName: String!, employerId: Int!): Employee**

Resolvers

```
},
Mutation: {
  addEmployee: (_, args) => {
    const newEmployee = {
      id: uuid.v4(),
      firstName: args.firstName,
      lastName: args.lastName,
      employerId: args.employerId
    };
    employees.push(newEmployee);
    return newEmployee;
  },
  removeEmployee: (_, args) => {
    return lodash.remove(employees, (e) => e.id == args.id);
  },
```

Running the Server

After we set up our type definitions and our resolvers then we need to run our Apollo server and wait for requests. We pass in the type definitions and resolvers as arguments:

```
const server = new ApolloServer({ typeDefs, resolvers });

server.listen().then(({ url }) => {
  console.log(`🚀  Server ready at ${url} 🚀`);
});
```

Then we run the sever by running index.js

```
🚀  Server ready at http://localhost:4000/ 🚀
```

Our server is now ready and waiting for requests. Now we can start on our client!

BUT WAIT!!!.....

There's more!!!!!

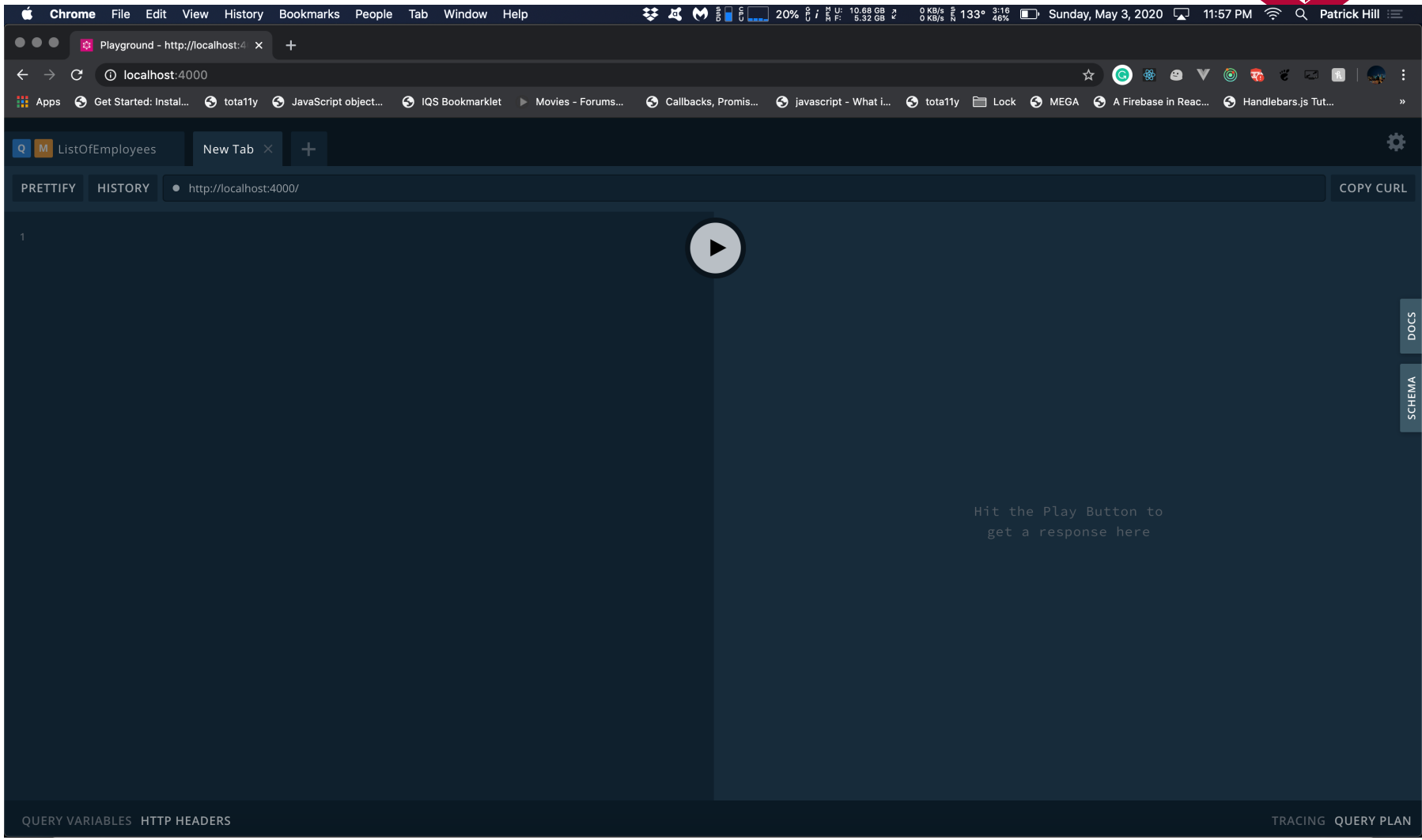
GraphQL Server Playground

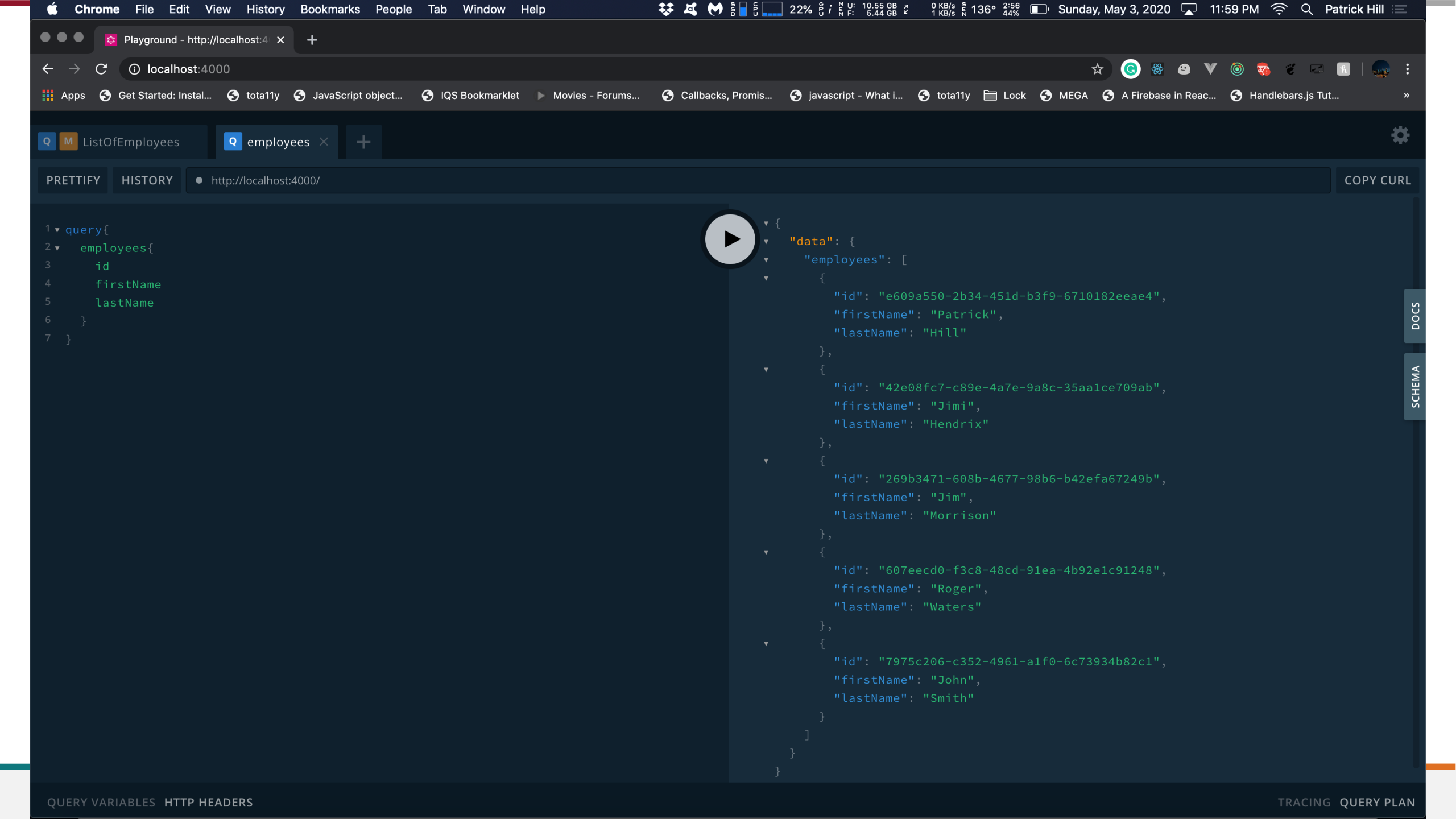
Before we start discuss the client, let's test out our server. But how?, you ask?

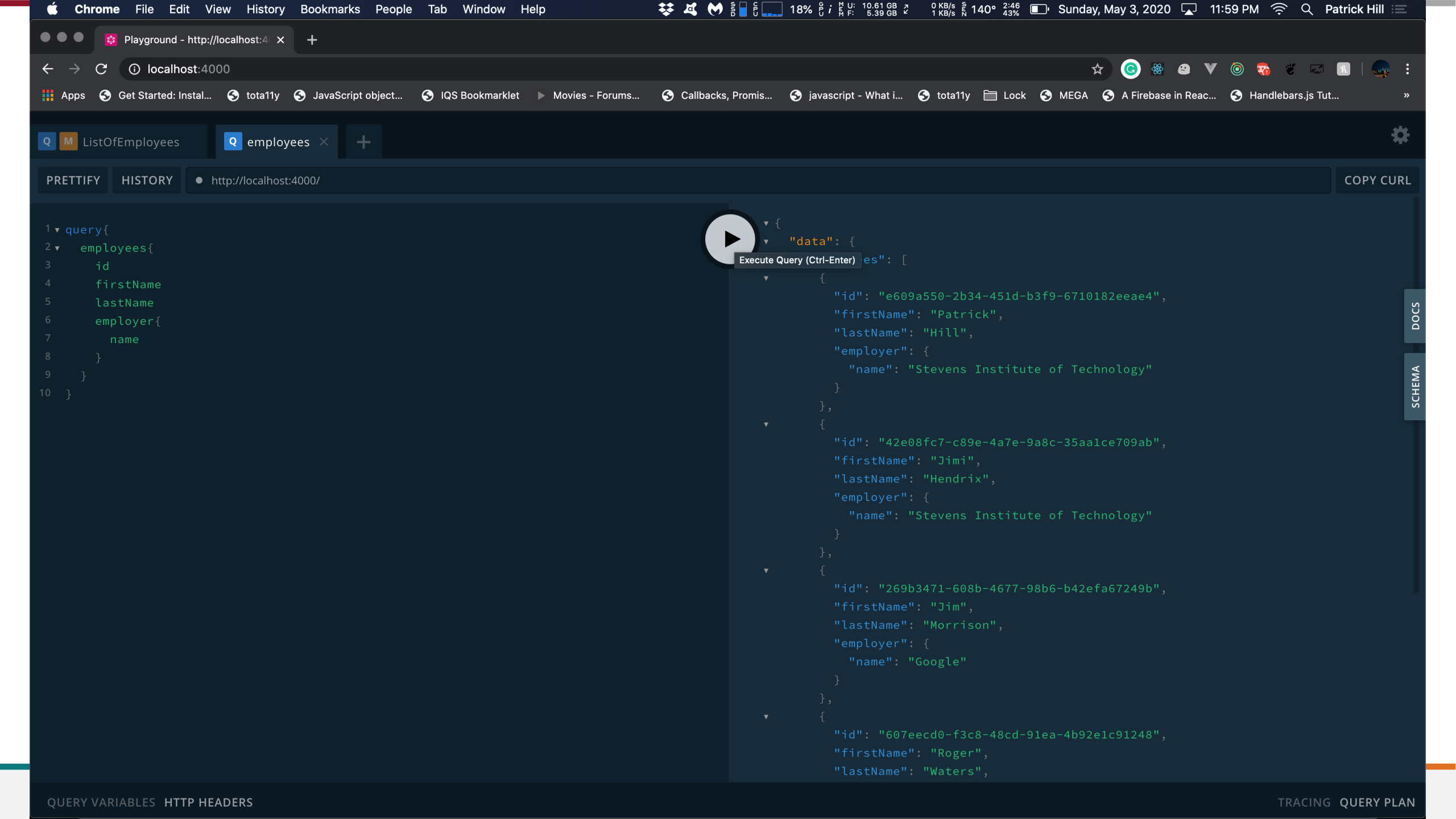
We just open up our browser and point to http://localhost:4000

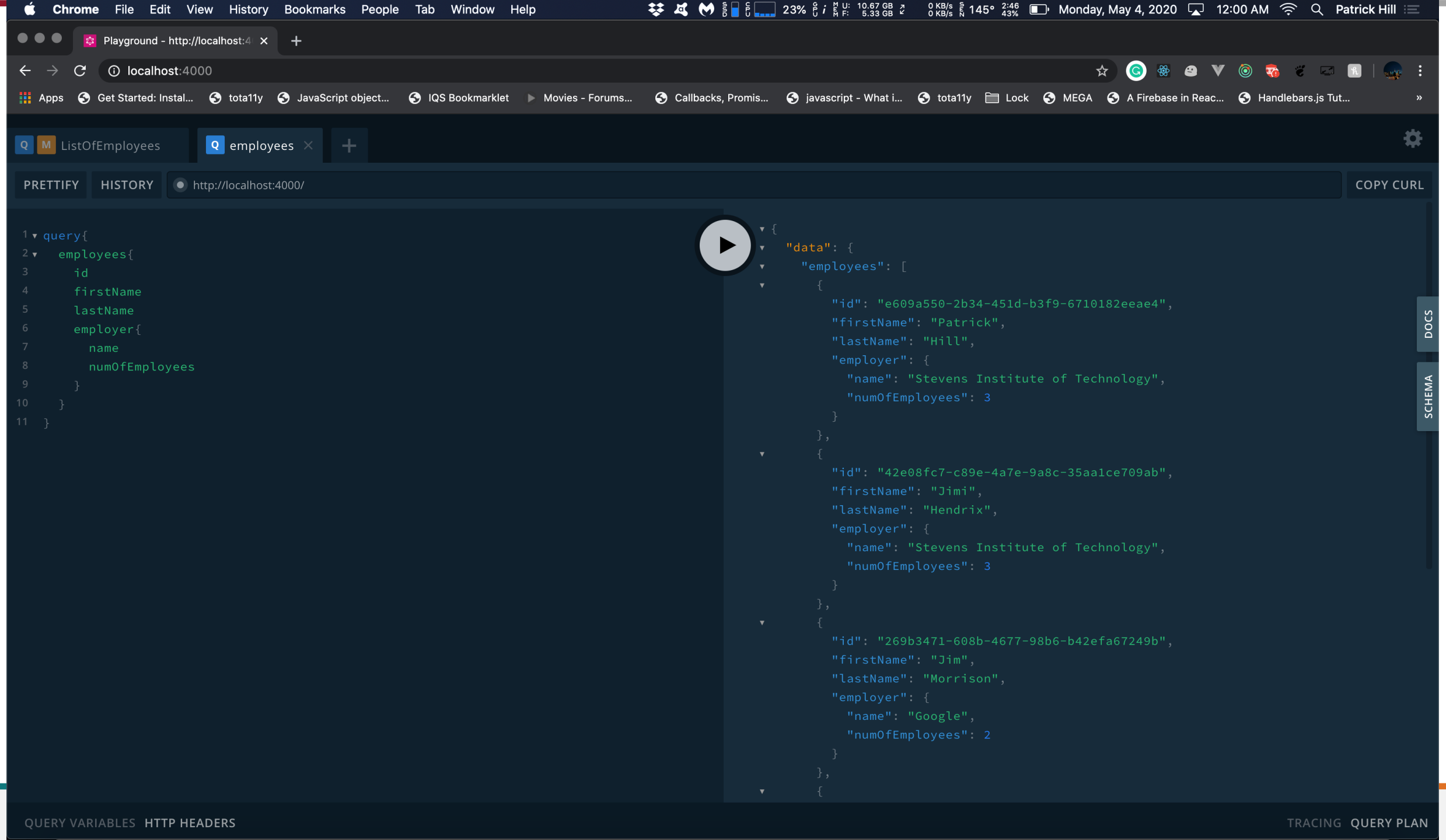
We can now run our queries and test them out in the playground!

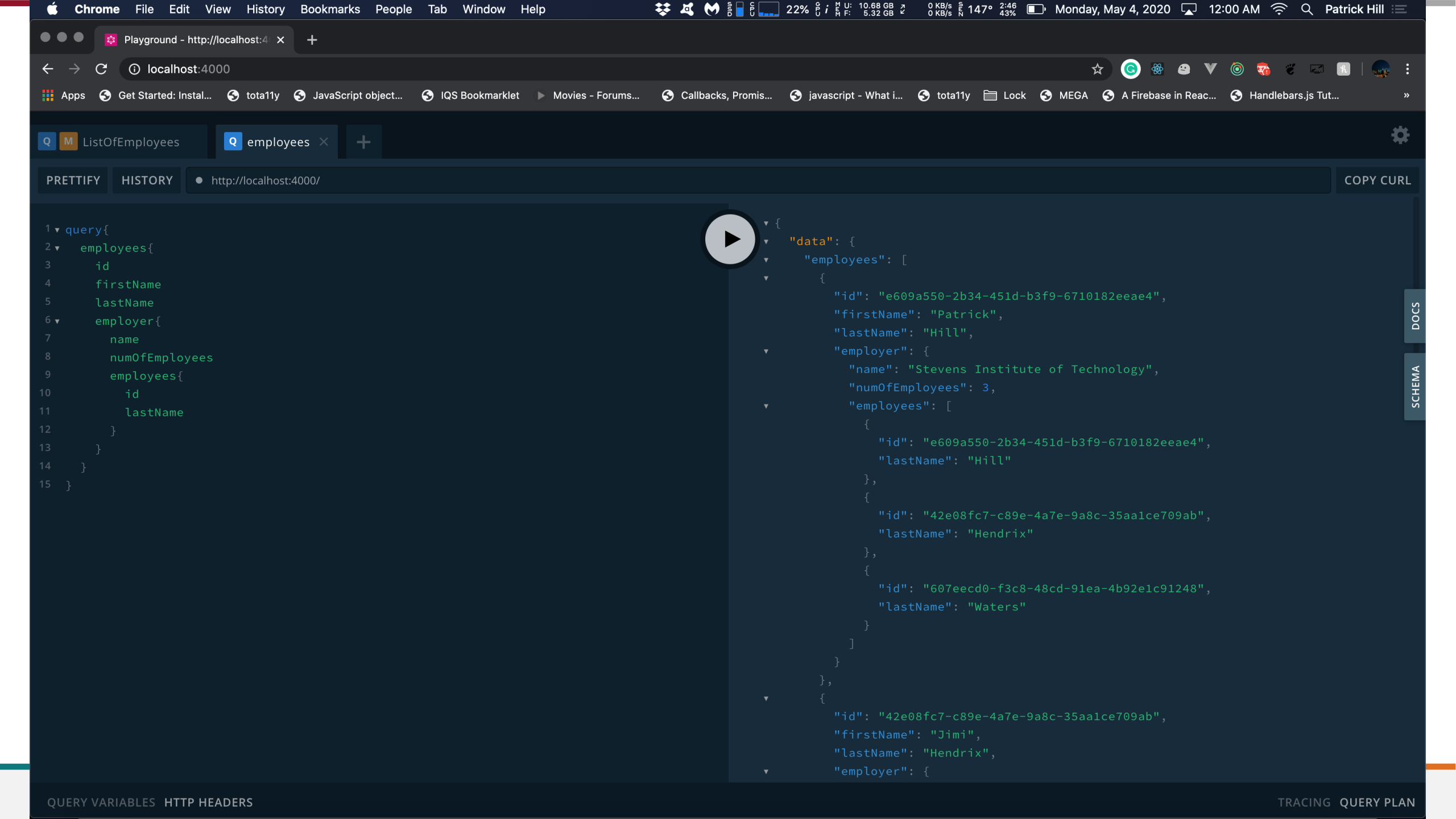
These queries will be called by the client which is making a request to **ONE** URL for all the data it needs



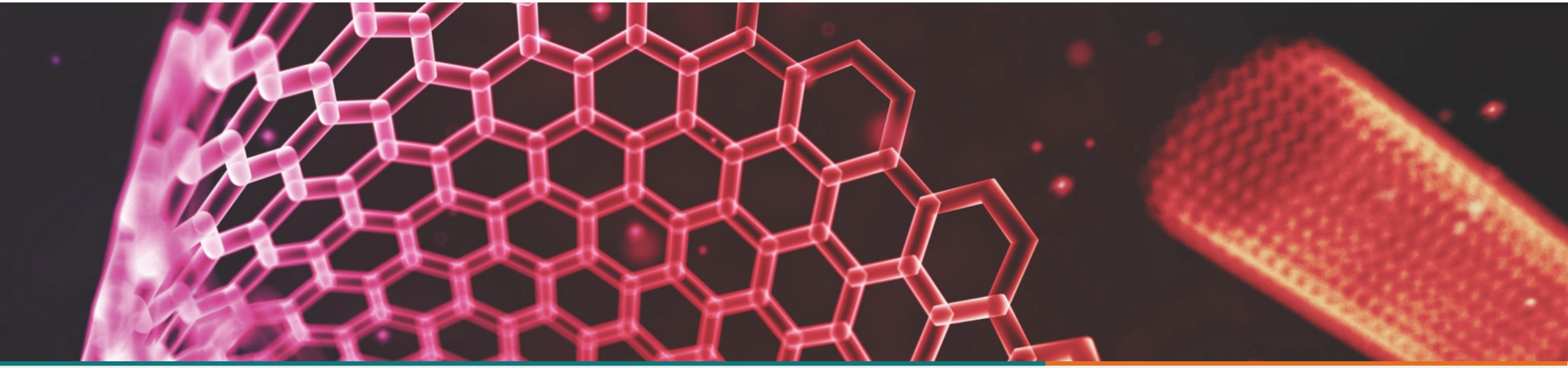

Chrome File Edit View History Bookmarks People Tab Window Help
Sunday, May 3, 2020 11:59 PM Patrick Hill
Playground - http://localhost:4
localhost:4000
Apps Get Started: Instal... tota11y JavaScript object... IQS Bookmarklet Movies - Forums... Callbacks, Promis... javascript - What i... tota11y Lock MEGA A Firebase in Reac... Handlebars.js Tut...
ListOfEmployees
employees
PRETTIFY
HISTORY
http://localhost:4000/
COPY CURL
query{
employees{
id
firstName
lastName
}
}
{
"data": {
"employees": [
{
"id": "e609a550-2b34-451d-b3f9-6710182eeae4",
"firstName": "Patrick",
"lastName": "Hill"
},
{
"id": "42e08fc7-c89e-4a7e-9a8c-35aa1ce709ab",
"firstName": "Jimi",
"lastName": "Hendrix"
},
{
"id": "269b3471-608b-4677-98b6-b42efa67249b",
"firstName": "Jim",
"lastName": "Morrison"
},
{
"id": "607eecd0-f3c8-48cd-91ea-4b92e1c91248",
"firstName": "Roger",
"lastName": "Waters"
},
{
"id": "7975c206-c352-4961-a1f0-6c73934b82c1",
"firstName": "John",
"lastName": "Smith"
}
]
}
}
DOCS
SCHEMA
QUERY VARIABLES HTTP HEADERS
TRACING QUERY PLAN


Chrome File Edit View History Bookmarks People Tab Window Help
Sunday, May 3, 2020 11:59 PM Patrick Hill
Playground - http://localhost:4
localhost:4000
Apps Get Started: Instal... tota11y JavaScript object... IQS Bookmarklet Movies - Forums... Callbacks, Promis... javascript - What i... tota11y Lock MEGA A Firebase in Reac... Handlebars.js Tut...
ListOfEmployees
employees
PRETTIFY
HISTORY
http://localhost:4000/
COPY CURL
query{
employees{
id
firstName
lastName
employer{
name
}
}
}
Execute Query (Ctrl-Enter)
{
"data": {
"id": "e609a550-2b34-451d-b3f9-6710182eeae4",
"firstName": "Patrick",
"lastName": "Hill",
"employer": {
"name": "Stevens Institute of Technology"
}
},
{
"id": "42e08fc7-c89e-4a7e-9a8c-35aa1ce709ab",
"firstName": "Jimi",
"lastName": "Hendrix",
"employer": {
"name": "Stevens Institute of Technology"
}
},
{
"id": "269b3471-608b-4677-98b6-b42efa67249b",
"firstName": "Jim",
"lastName": "Morrison",
"employer": {
"name": "Google"
}
},
{
"id": "607eecd0-f3c8-48cd-91ea-4b92e1c91248",
"firstName": "Roger",
"lastName": "Waters",
DOCS
SCHEMA
QUERY VARIABLES HTTP HEADERS
TRACING QUERY PLAN


Chrome File Edit View History Bookmarks People Tab Window Help
Monday, May 4, 2020 12:00 AM Patrick Hill
Playground - http://localhost:4
localhost:4000
Apps Get Started: Instal... tota11y JavaScript object... IQS Bookmarklet Movies - Forums... Callbacks, Promis... javascript - What i... tota11y Lock MEGA A Firebase in Reac... Handlebars.js Tut...
ListOfEmployees
employees
PRETTIFY
HISTORY
http://localhost:4000/
COPY CURL
query{
employees{
id
firstName
lastName
employer{
name
numOfEmployees
}
}
}
{
"data": {
"employees": [
{
"id": "e609a550-2b34-451d-b3f9-6710182eeae4",
"firstName": "Patrick",
"lastName": "Hill",
"employer": {
"name": "Stevens Institute of Technology",
"numOfEmployees": 3
}
},
{
"id": "42e08fc7-c89e-4a7e-9a8c-35aa1ce709ab",
"firstName": "Jimi",
"lastName": "Hendrix",
"employer": {
"name": "Stevens Institute of Technology",
"numOfEmployees": 3
}
},
{
"id": "269b3471-608b-4677-98b6-b42efa67249b",
"firstName": "Jim",
"lastName": "Morrison",
"employer": {
"name": "Google",
"numOfEmployees": 2
}
},
{
DOCS
SCHEMA
QUERY VARIABLES HTTP HEADERS
TRACING QUERY PLAN


Chrome File Edit View History Bookmarks People Tab Window Help
Monday, May 4, 2020 12:00 AM Patrick Hill
Playground - http://localhost:4
localhost:4000
Apps Get Started: Instal... tota11y JavaScript object... IQS Bookmarklet Movies - Forums... Callbacks, Promis... javascript - What i... tota11y Lock MEGA A Firebase in Reac... Handlebars.js Tut...
ListOfEmployees
employees
PRETTIFY
HISTORY
http://localhost:4000/
COPY CURL
query{
employees{
id
firstName
lastName
employer{
name
numOfEmployees
employees{
id
lastName
}
}
}
}
{
"data": {
"employees": [
{
"id": "e609a550-2b34-451d-b3f9-6710182eeae4",
"firstName": "Patrick",
"lastName": "Hill",
"employer": {
"name": "Stevens Institute of Technology",
"numOfEmployees": 3,
"employees": [
{
"id": "e609a550-2b34-451d-b3f9-6710182eeae4",
"lastName": "Hill"
},
{
"id": "42e08fc7-c89e-4a7e-9a8c-35aa1ce709ab",
"lastName": "Hendrix"
},
{
"id": "607eecd0-f3c8-48cd-91ea-4b92e1c91248",
"lastName": "Waters"
}
]
}
},
{
"id": "42e08fc7-c89e-4a7e-9a8c-35aa1ce709ab",
"firstName": "Jimi",
"lastName": "Hendrix",
"employer": {
DOCS
SCHEMA
QUERY VARIABLES HTTP HEADERS
TRACING QUERY PLAN

Client Using React and Apollo Client



Stuff We Will Use

We are going to create our client using React and Apollo Client.

In the lecture video we will use create-react-app to create our react application. We also need to install the apollo-client and graphql npm packages into our project

- ***npm install @apollo-client***
- ***npm install graphql***

We then need to import a few things from apollo-client and configure it:

```
import { ApolloClient, HttpLink, InMemoryCache, ApolloProvider } from '@apollo/client';
const client = new ApolloClient({
  cache: new InMemoryCache(),
  link: new HttpLink({
    uri: 'http://localhost:4000'
  })
});
```

Apollo Provider

We then wrap our App.js component in the provider passing in the client that was set up.

That's it! We can now write our react components that will call queries and mutations.

```
function App() {
  return (
    <ApolloProvider client={client}>
      <Router>
        <div>
          <header className="App-header">
            <h1 className="App-title">
              GraphQL With Apollo Client/Server Demo
            </h1>
            <nav>
              <NavLink className="navlink" to="/">
                Home
              </NavLink>
              <NavLink className="navlink" to="/employees">
                Employees
              </NavLink>

              <NavLink className="navlink" to="/employers">
                Employers
              </NavLink>
            </nav>
          </header>
          <Route exact path="/" component={Home} />
          <Route path="/employees/" component={Employees} />
          <Route path="/employers/" component={Employers} />
        </div>
      </Router>
    </ApolloProvider>
  );
}

export default App;
```

useQuery and useMutation React Hooks

We are going to use two React hooks that are built into apollo-client to interact with our GraphQL server.

- **useQuery**
- **userMutation**

We will see in our lecture code how to use them along with our React components.

We are going to have a component that lists the employees, one that lists the employers, how many employees they have and the employee names. We will also be using mutations to create, edit and delete employees utilizing the typedefs, queries and mutations we defined in our server.

Questions?