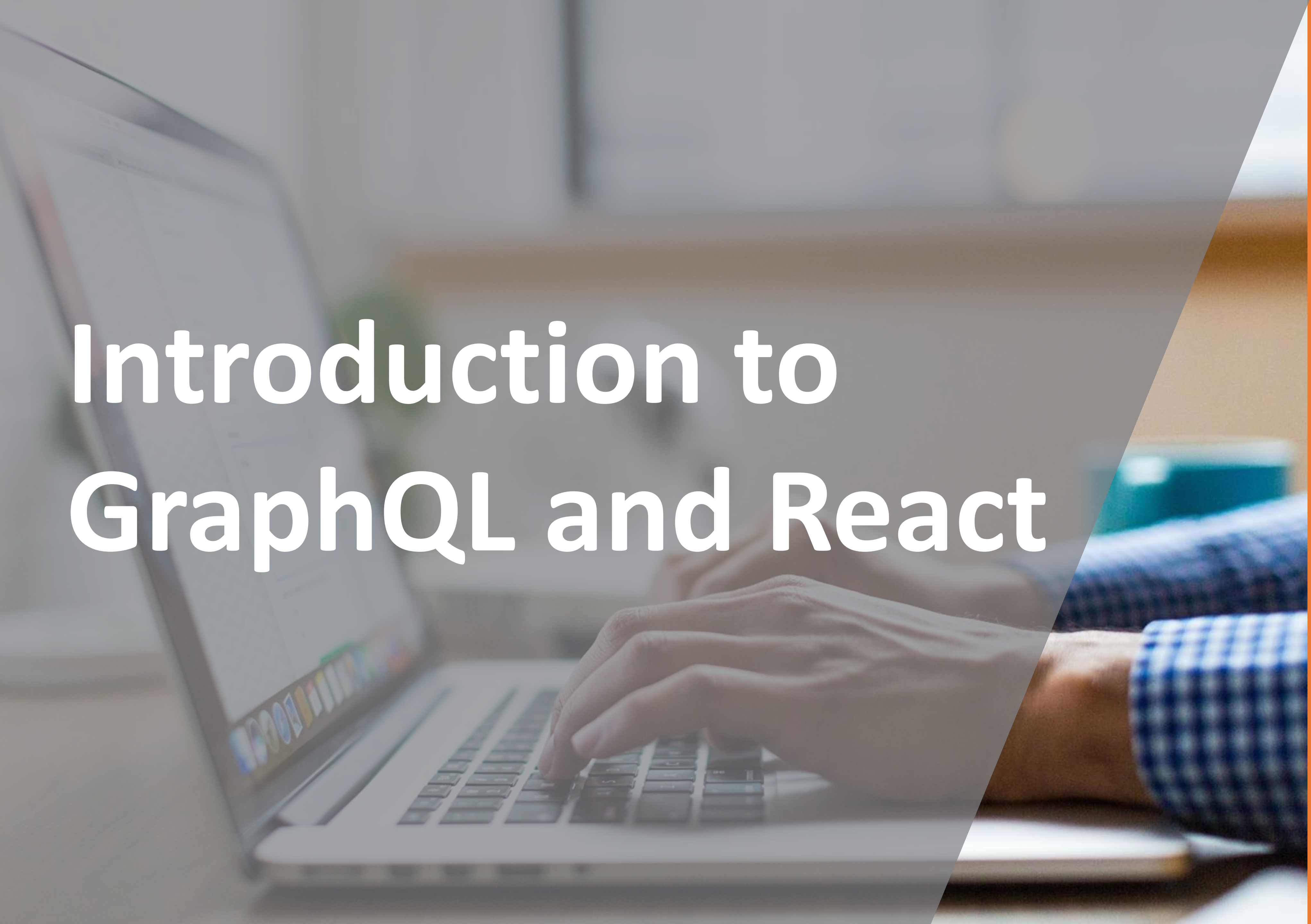
Introduction to GraphQL and React



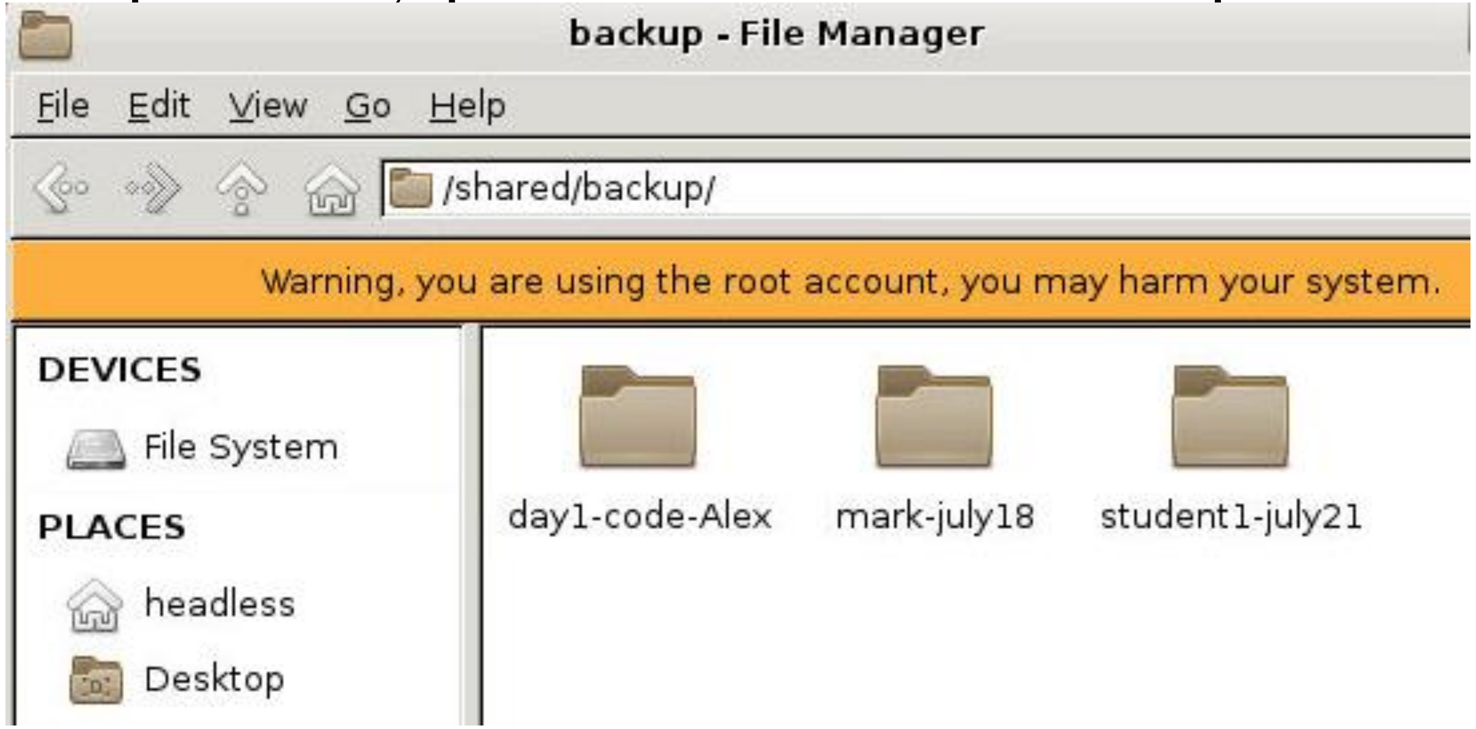
TriveraTech
TECHNOLOGY TRAINING

Table of Contents

1. Introduction to GraphQL: 4
2. Exploring GraphQL APIs: 42
3. Preparing Your Development Environment: 80
4. Setting up GraphQL with Express.js: 137
5. Connecting to The Database: 190
6. Integrating React into the Back end with Apollo: 283
7. Reusable React Components: 370

Lab Environment Overview

- Shared backup storage is mounted at "/shared/backup".
- Create new folder and backup your code/work at the end of each day.
- **Note:** Do not place any personal information or passwords in this folder.



Warning, you are using the root account, you may harm your system.

1. Introduction to GraphQL

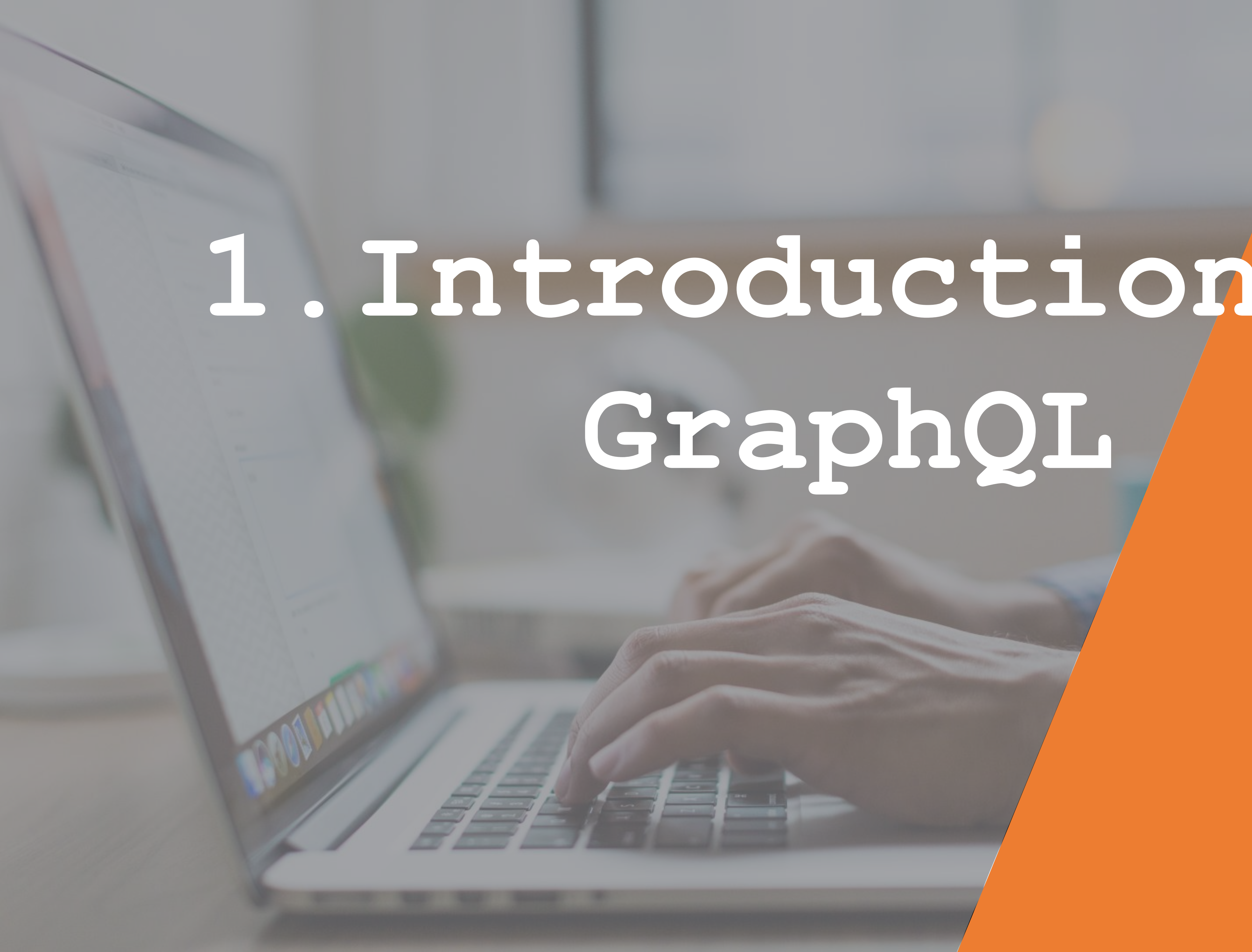


Table of Contents

- **What is GraphQL?**
- Why GraphQL?
- GraphQL problems

Introduction to Grap

This lesson covers

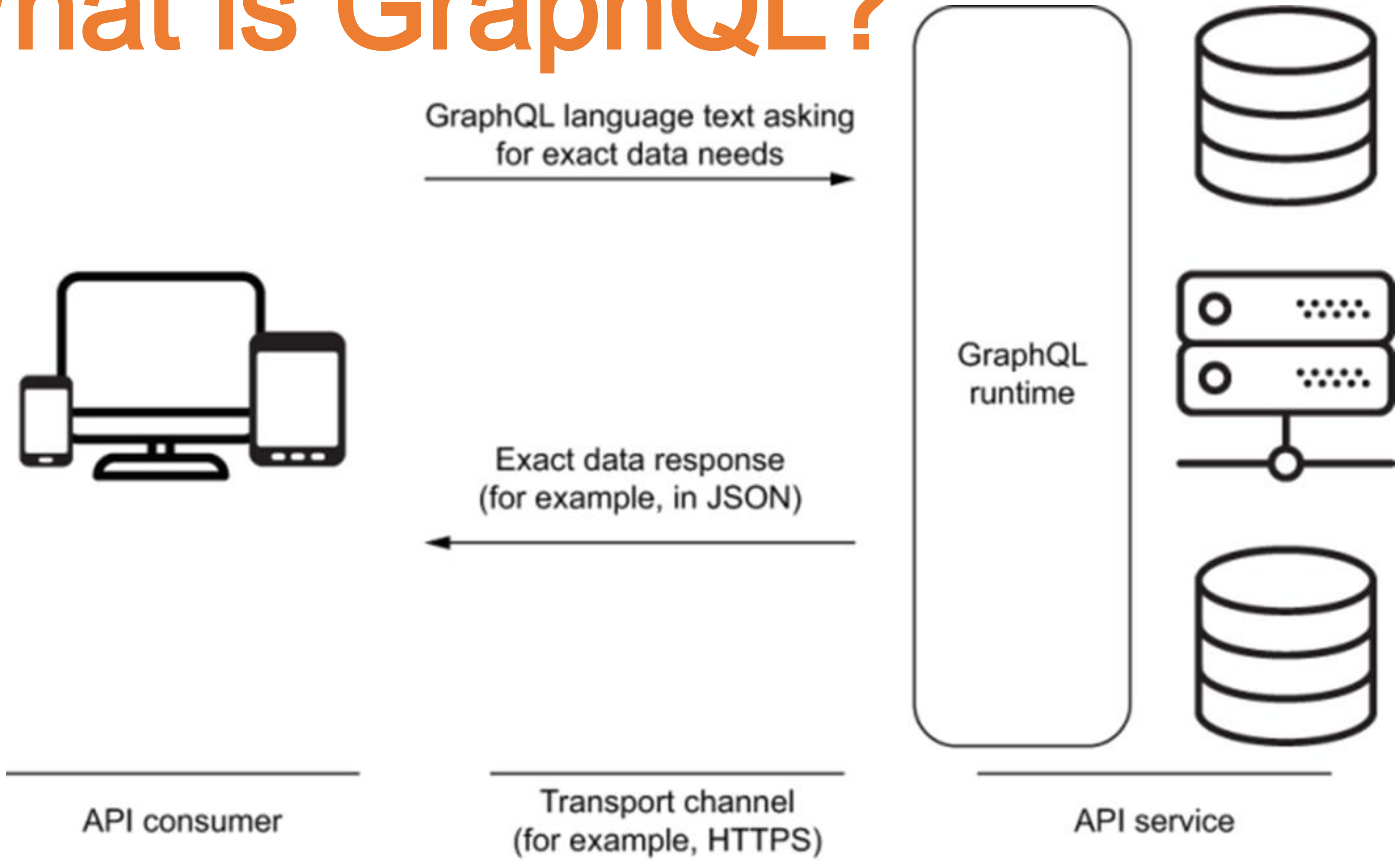
- Understanding GraphQL and the design concepts behind it
- How GraphQL differs from alternatives like REST APIs
- Understanding the language used by GraphQL clients and services
- Understanding the advantages and disadvantages of GraphQL

What is GraphQL?

- The word graph in GraphQL comes from the fact that the best way to represent data in the real world is with a graph-like data structure.

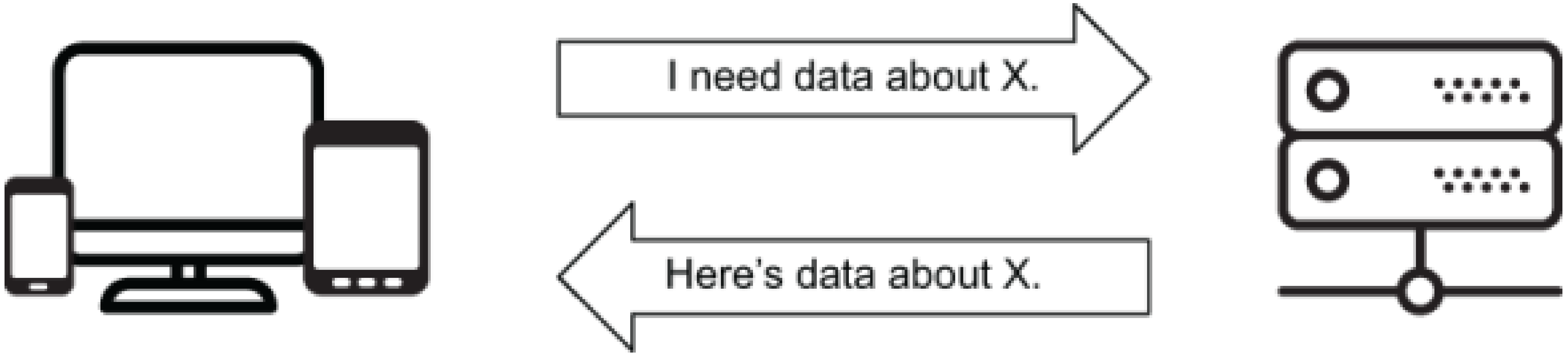
- If you analyze any data model, big or small, you’ll always find it to be a graph of objects with many relations between them.

What is GraphQL?


GraphQL language text asking for exact data needs
Exact data response (for example, in JSON)
GraphQL runtime
API consumer
Transport channel (for example, HTTPS)
API service

The big picture

- For example, an API can enable the communication that needs to happen between a web client and a database server.



The big picture

- For example, if we have a table of data about a company’s employees, the following is an example SQL statement to read data about the employees in one department.

```
SELECT id, first_name, last_name, email,
birth_date, hire_date
FROM employees
WHERE department = 'ENGINEERING'
```

The big picture

- Here is another example SQL statement that inserts data for a new employee.

```
INSERT INTO employees (first_name, last_name,
email, birth_date, hire_date)
VALUES ('Jane', 'Doe', 'jane@doe.name',
'01/01/1990', '01/01/2020')
```

The big picture

```
{
  "data": {
    "employee":{
      "id": 42,
      "name": "Jane Doe",
      "email": "jane@doe.name",
      "birthDate": "01/01/1990",
      "hireDate": "01/01/2020"
    }
  }
}
```

The big picture

```
{
  "select": {
    "fields": ["name", "email", "birthDate",
"hireDate"],
    "from": "employees",
    "where": {
      "id": {
        "equals": 42
      }
    }
  }
}
```

The big picture

```
{
  employee(id: 42) {
    name
    email
    birthDate
    hireDate
  }
}
```

GraphQL is a language

- A query language like GraphQL (or SQL) is different from programming languages like JavaScript and Python.
- You cannot use the GraphQL language to create user interfaces or perform complex computations.
- Query languages have more specific use cases, and they often require the use of programming languages to make them work.

GraphQL is a language

- For example, here’s a hypothetical single GraphQL query that represents both of John’s questions to Jane.

```
{
  timeLightNeedsToTravel(toPlanet: "Earth") {
    fromTheSun: from(star: "Sun")
    fromTheMoon: from(moon: "Moon")
  }
}
```

GraphQL is a service

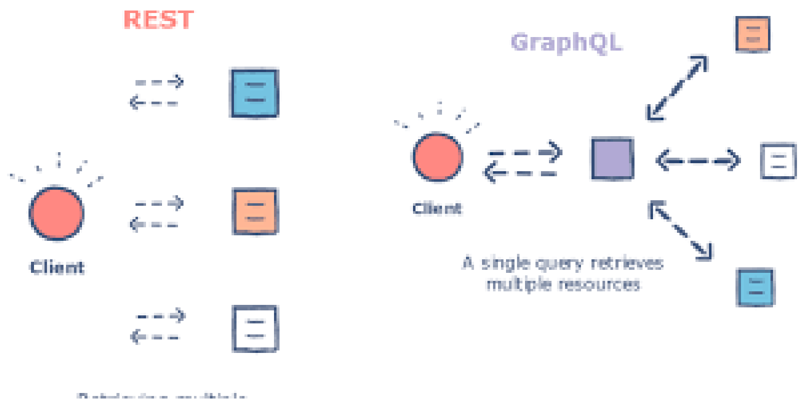
A GraphQL service can be written in any programming language, and it can be conceptually split into two major parts, structure and behavior:

- The structure is defined with a strongly typed schema.
- The behavior is naturally implemented with functions that in the GraphQL world are called resolver functions.

GraphQL is a service

- To understand how resolvers work, let's take the query in listing 1.5 (simplified) and assume a client sent it to a GraphQL service.

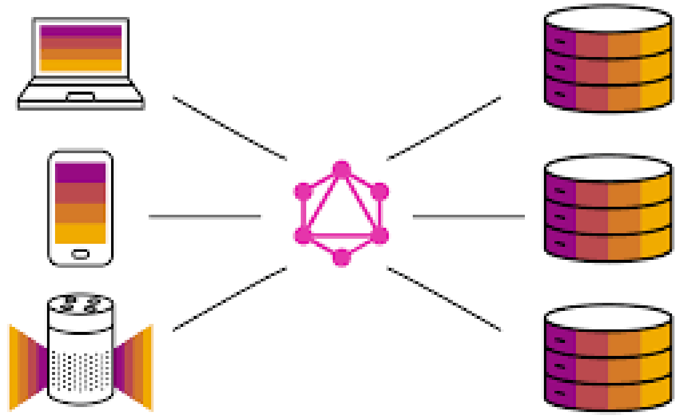
```
query {
  employee(id: 42) {
    name
    email
  }
}
```



GraphQL is a service

- Here's an example to represent the Employee type using GraphQL's schema language.

```
type Employee(id: Int!) {
  name: String!
  email: String!
}
```



GraphQL is a service

- The employee field's resolver function for employee #42 might return an object like the following.

```
{
  "id": 42,
  "first_name": "Jane",
  "last_name": "Doe",
  "email": "jane@doe.name",
  "birth_date": "01/01/1990",
  "hire_date": "01/01/2020"
}
```

GraphQL is a service

- Let’s say we have the following (JavaScript) functions representing the server resolver functions for the name and email fields:

```
// Resolver functions
const name => (source) => `${source.first_name}
${source.last_name}`;
const email => (source) => source.email;
```

GraphQL is a service

- The GraphQL service uses all the responses of these three resolver functions to put together the following single response for the query in listing 1.7.

```
{
  data: {
    employee: {
      name: 'Jane Doe',
      email: 'jane@doe.name'
    }
  }
}
```

Table of Contents

- What is GraphQL?
- **Why GraphQL?**
- GraphQL problems

TriveraTech
TECHNOLOGY TRAINING

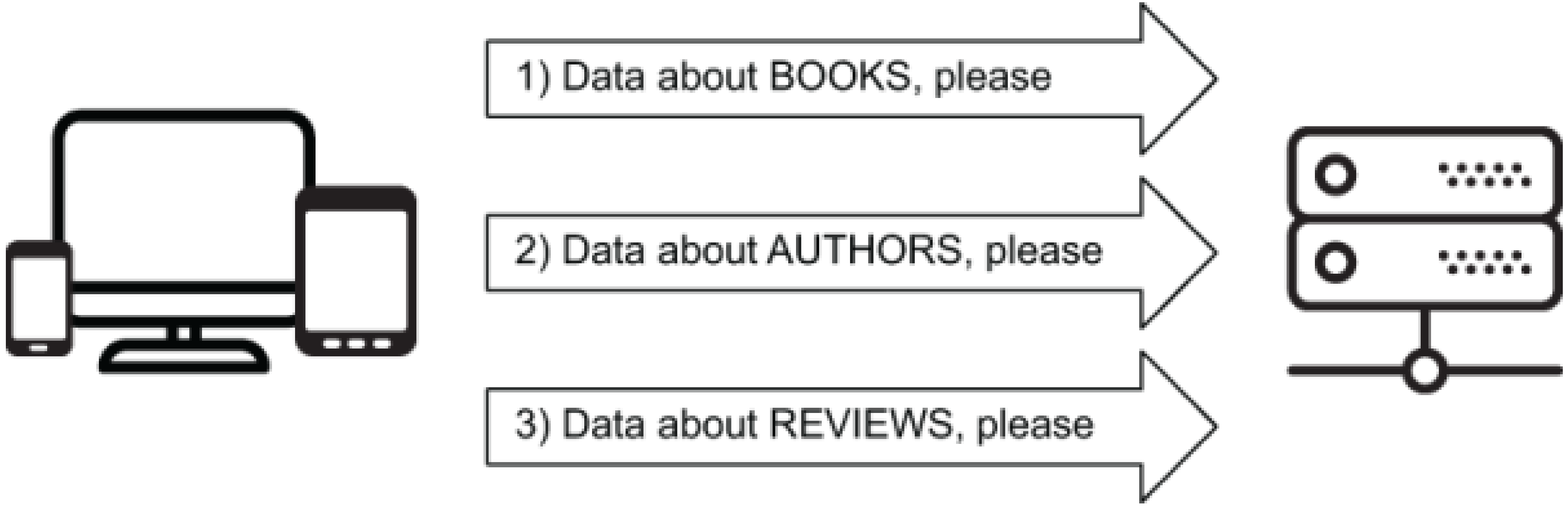
Why GraphQL?

- GraphQL is not the only-or even the first-technology to encourage creating efficient data APIs.

- You can use a JSON-based API with a custom query language or implement the Open Data Protocol (OData) on top of a REST API.

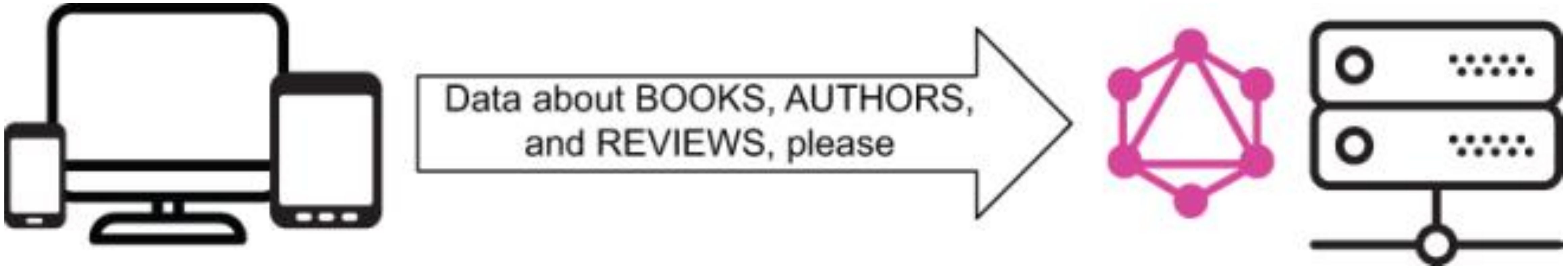
- GraphQL provides comprehensive standards and structures to implement API features in maintainable and scalable ways.

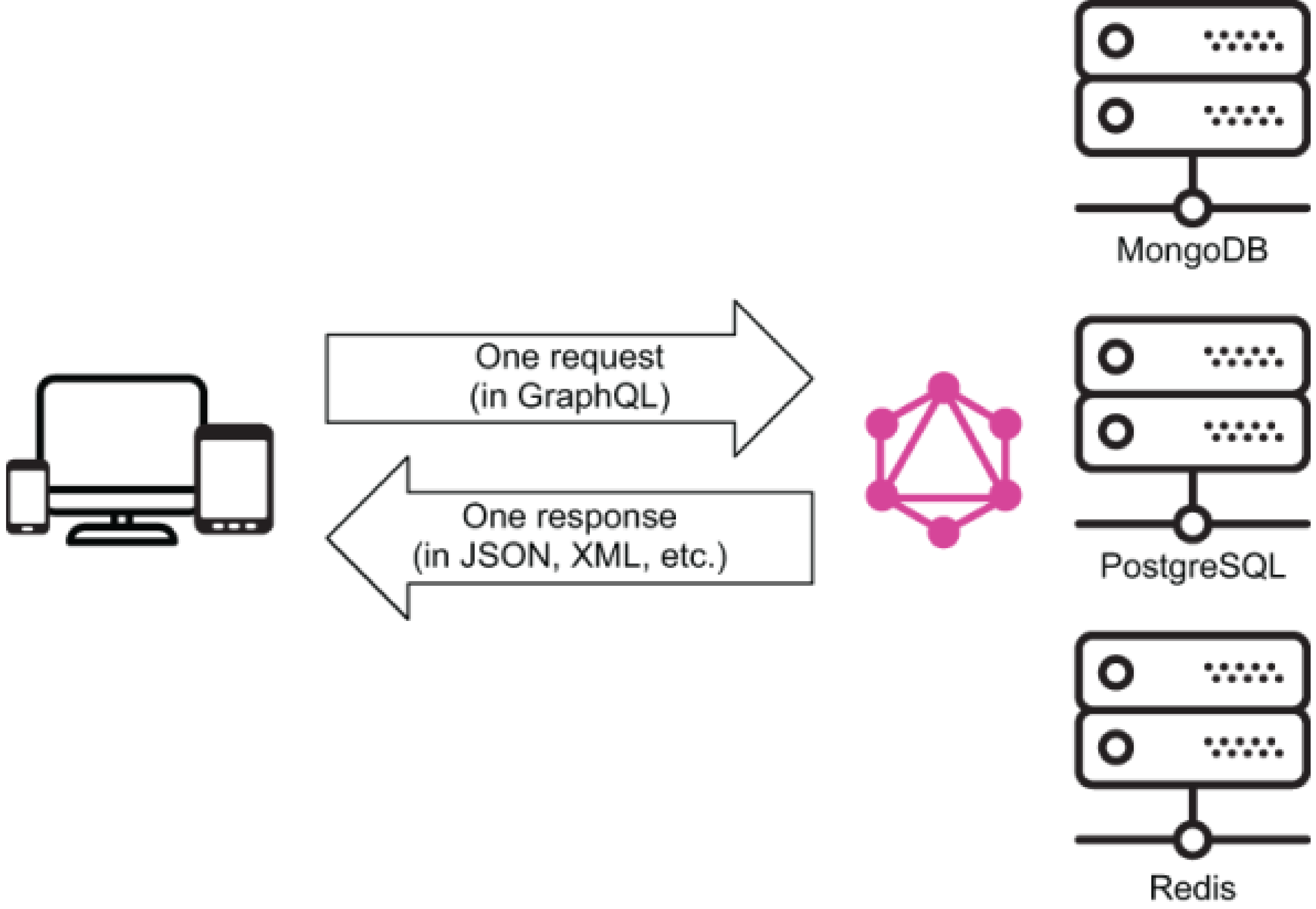
Why GraphQL?


1) Data about BOOKS, please
2) Data about AUTHORS, please
3) Data about REVIEWS, please

Why GraphQL?

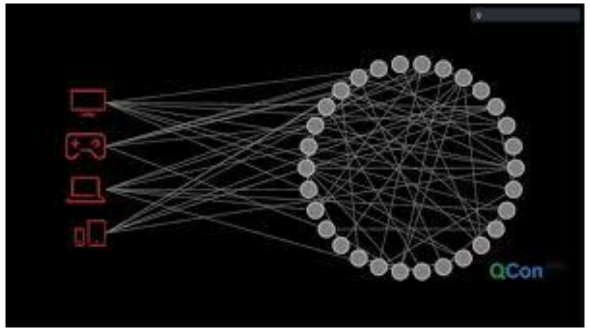
- You can customize a REST-based API to provide one exact endpoint per view, but that's not the norm.
- You will have to implement it without a standard guide.




One request
(in GraphQL)
One response
(in JSON, XML, etc.)
MongoDB
PostgreSQL
Redis

What about REST APIs?

- GraphQL APIs are often compared to REST APIs because the latter have been the most popular choice for data APIs demanded by web and mobile applications.
- GraphQL provides a technological alternative to REST APIS



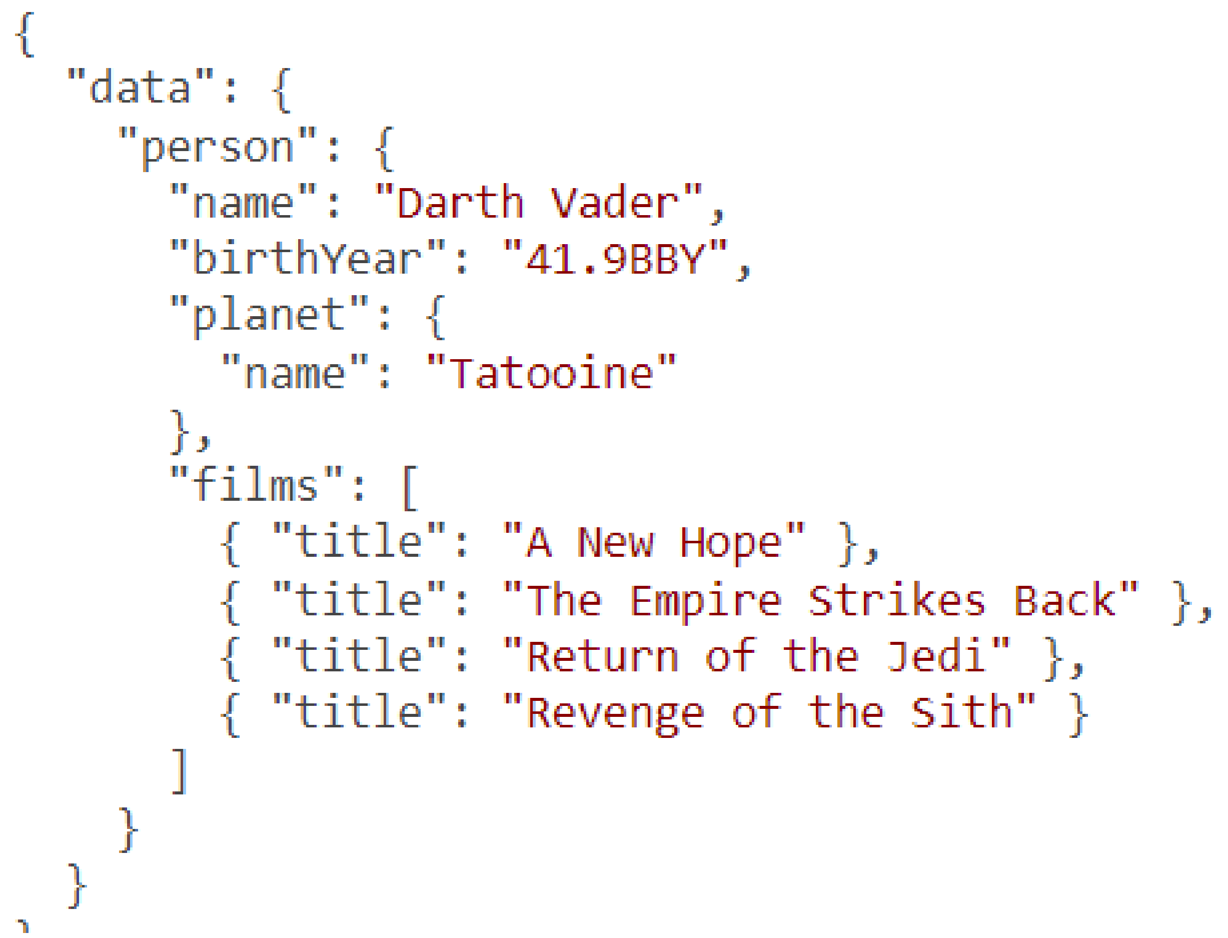
The GraphQL way

To see the GraphQL way of solving the REST API problems we have talked about, you need to understand the concepts and design decisions behind GraphQL. Let’s review the major ones.

- The typed Graph schema
- The declarative language
- The single endpoint and client language
- The simple versioning

REST APIs and GraphQL APIs in action

- The JSON data for this view could be something like the following.



```
{
  "data": {
    "person": {
      "name": "Darth Vader",
      "birthYear": "41.9BBY",
      "planet": {
        "name": "Tatooine"
      },
      "films": [
        { "title": "A New Hope" },
        { "title": "The Empire Strikes Back" },
        { "title": "Return of the Jedi" },
        { "title": "Revenge of the Sith" }
      ]
    }
  }
```

REST APIs and GraphQL APIs in action

```
// The Container Component:
<PersonProfile
person={data.person}></PersonProfile>
// The PersonProfile Component:
Name: {data.person.name}
Birth Year: {data.person.birthYear}
Planet: {data.person.planet.name}
Films: {data.person.films.map(film =>
film.title)}
```

REST APIs and GraphQL APIs in action

- You need a Star Wars character’s information.

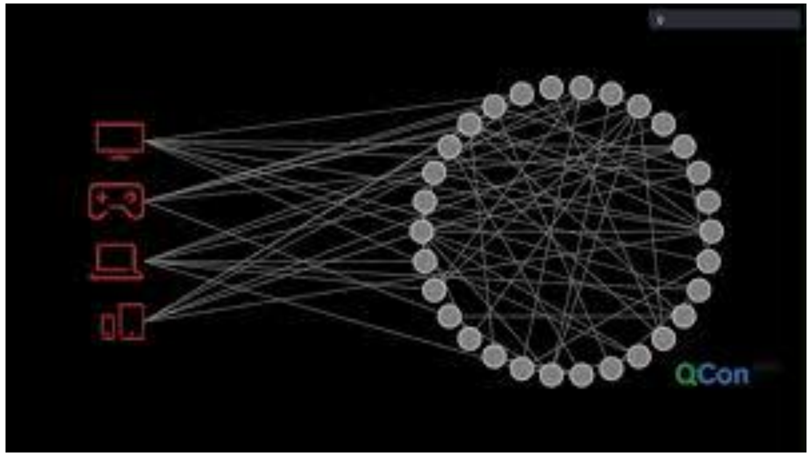
- Assuming that you know that character’s ID, a REST API is expected to expose that information with an endpoint like this:

```
GET - /people/{id}
```

REST APIs and GraphQL APIs in action

- The JSON response for this request could be something like the following:

```
{
  "name": "Darth Vader",
  "birthYear": "41.9BBY",
  "planetId": 1
  "filmIds": [1, 2, 3, 6],
   .—.—.
}
```



In English	In GraphQL

The view needs:
 	
a person’s name,
 	
birth year,
 	
planet’s name,
 	
and the titles of all their films.
	`{`
	`  person(ID: ·-·-·) {`
	`    name`
	`    birthYear`
	`    planet {`
	`      name`
	`    }`
	`    films {`
	`      title`
	`    }`
	`  }`
	`}`

GraphQL query (question)	Needed JSON (answer)

GraphQL query (question):

```
{
  person(ID: ·-·-·) {
    name
    birthYear
    planet {
      name
    }
    films {
      title
    }
  }
}
```

Needed JSON (answer):

```
{
  "data": {
    "person": {
      "name": "Darth Vader",
      "birthYear": "41.9BBY",
      "planet": {
        "name": "Tatooine"
       },
      "films": [
        { "title": "A New Hope" },
        { "title": "The Empire Strikes Back" },
        { "title": "Return of the Jedi" },
        { "title": "Revenge of the Sith" }
      ]
    }
  }
}
```

```
{
  person(personID: 4) {
    name
    birthYear
    homeworld {
      name
    }
    filmConnection {
      films {
        title
      }
    }
  }
}
```

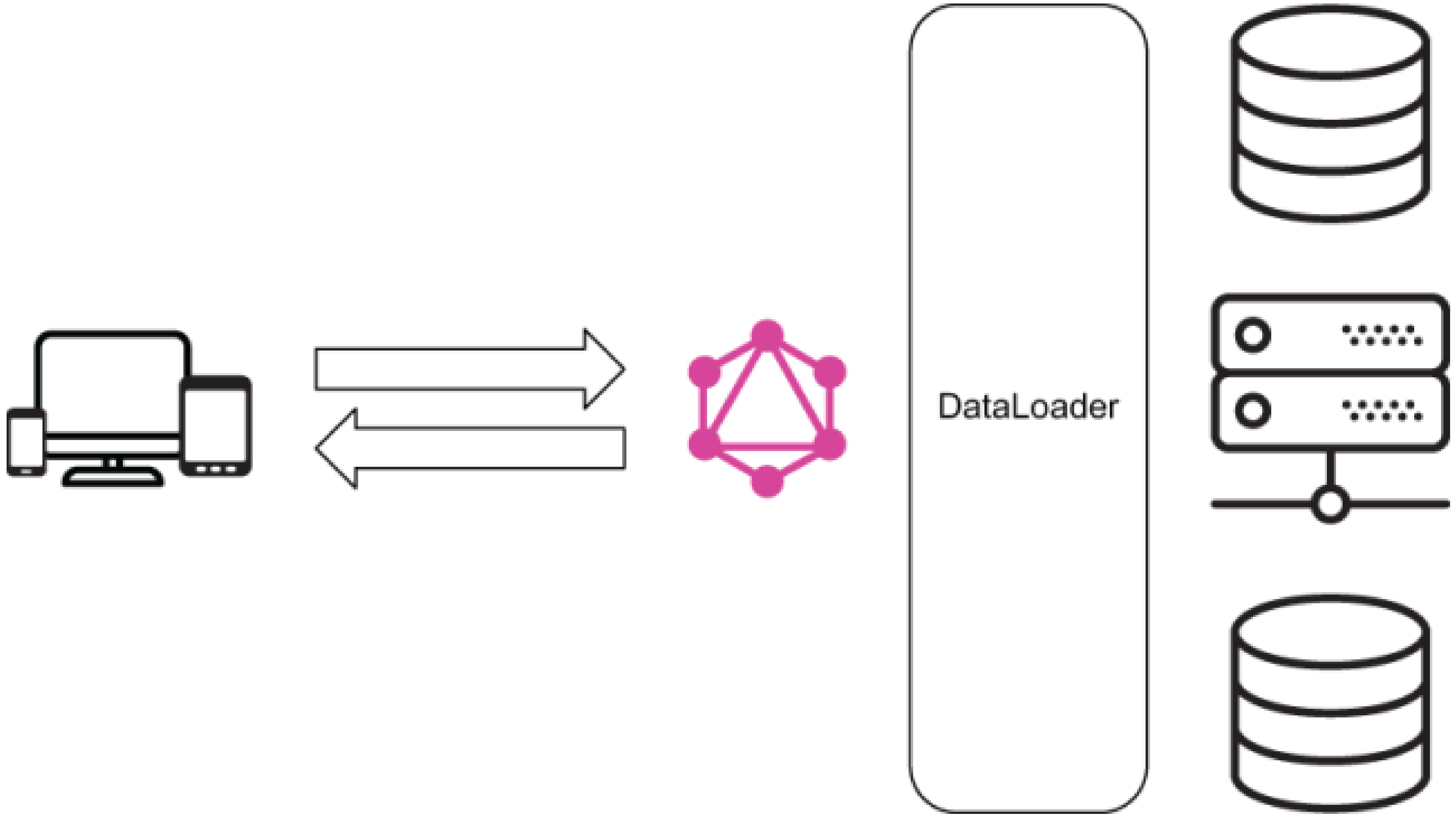
Table of Contents

- What is GraphQL?
- Why GraphQL?
- **GraphQL problems**

GraphQL problems

Perfect solutions are fairy tales. The flexibility that GraphQL introduces opens a door to some clear issues and concerns.

- Security
- Caching and optimizing
- Learning curve


DataLoader

Summary

- The best way to represent data in the real world is with a graph data structure. A data model is a graph of related objects. GraphQL embraces this fact.
- A GraphQL system has two primary components: the query language, which can be used by consumers of data APIs to request their exact data needs; and the runtime layer on the backend, which publishes a public schema describing the capabilities and requirements of data models

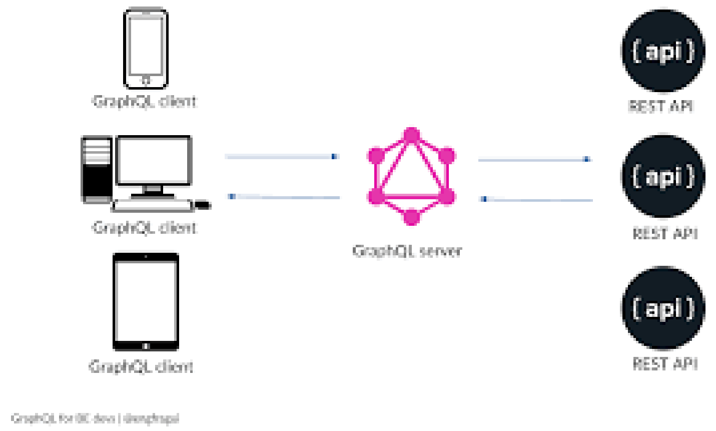
"Complete Lab"

2. Exploring GraphQL APIs

Table of Contents

- **The GraphiQL editor**
- The basics of the GraphQL language
- Examples from the GitHub API

Exploring GraphQL APIs



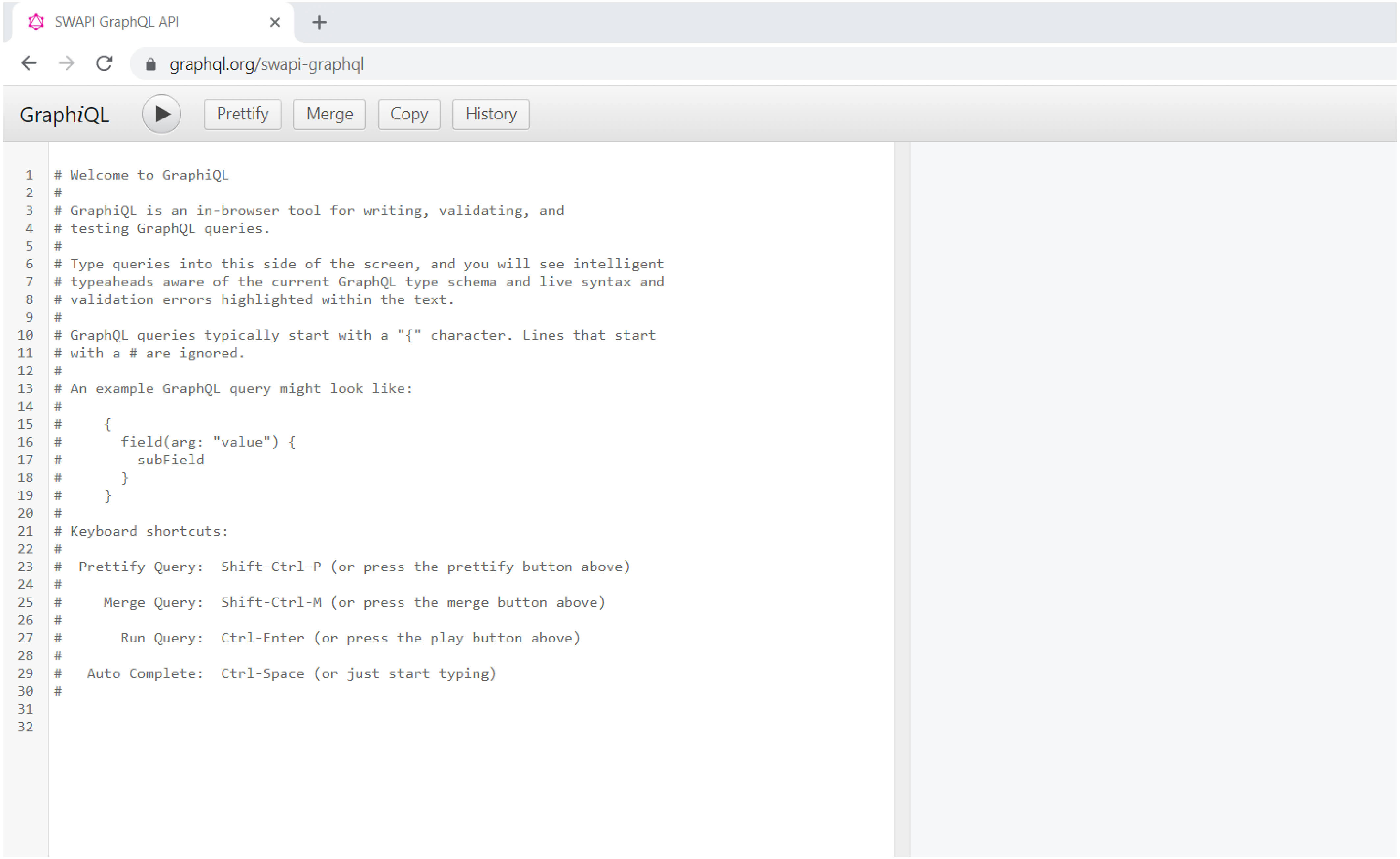
This lesson covers

- Using GraphQL’s in-browser IDE to test GraphQL requests
- Exploring the fundamentals of sending GraphQL data requests
- Exploring read and write example operations from the GitHub GraphQL API
- Exploring GraphQL’s introspective features

The GraphiQL editor

- When thinking about the requests your client applications need to make to servers, you can benefit from a graphical tool to first help you come up with these requests and then test them before committing to them in application code.

- Such a tool can also help you improve these requests, validate your improvements, and debug any requests that are running into problems.

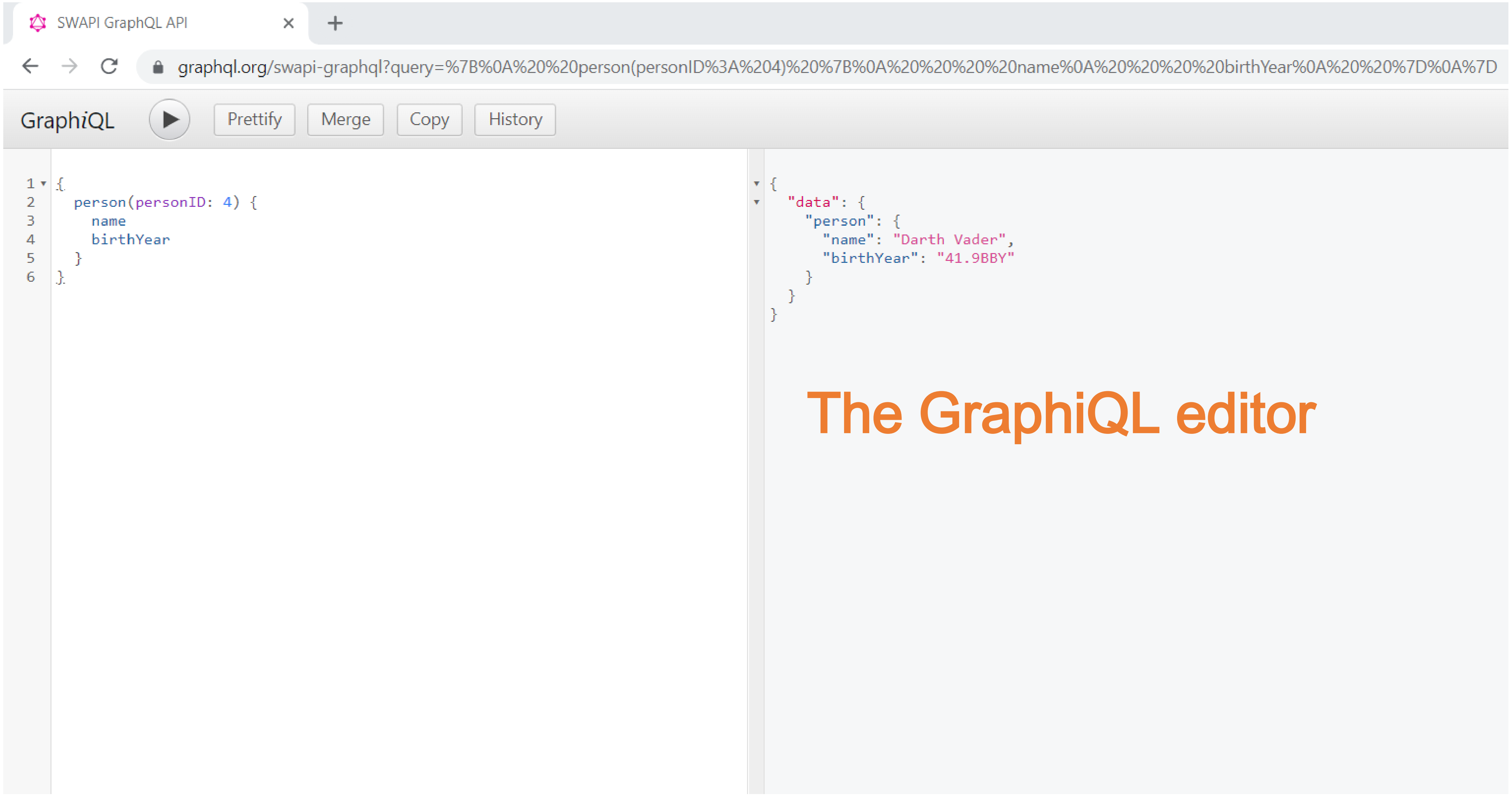


```
# Welcome to GraphiQL
#
# GraphiQL is an in-browser tool for writing, validating, and
# testing GraphQL queries.
#
# Type queries into this side of the screen, and you will see intelligent
# typeaheads aware of the current GraphQL type schema and live syntax and
# validation errors highlighted within the text.
#
# GraphQL queries typically start with a "{" character. Lines that start
# with a # are ignored.
#
# An example GraphQL query might look like:
#
#     {
#       field(arg: "value") {
#         subField
#       }
#     }
#
# Keyboard shortcuts:
#
#  Prettify Query:  Shift-Ctrl-P (or press the prettify button above)
#
#     Merge Query:  Shift-Ctrl-M (or press the merge button above)
#
#       Run Query:  Ctrl-Enter (or press the play button above)
#
#   Auto Complete:  Ctrl-Space (or just start typing)
#


```

The GraphiQL editor

- Go ahead and type the following simple GraphQL query in the editor.

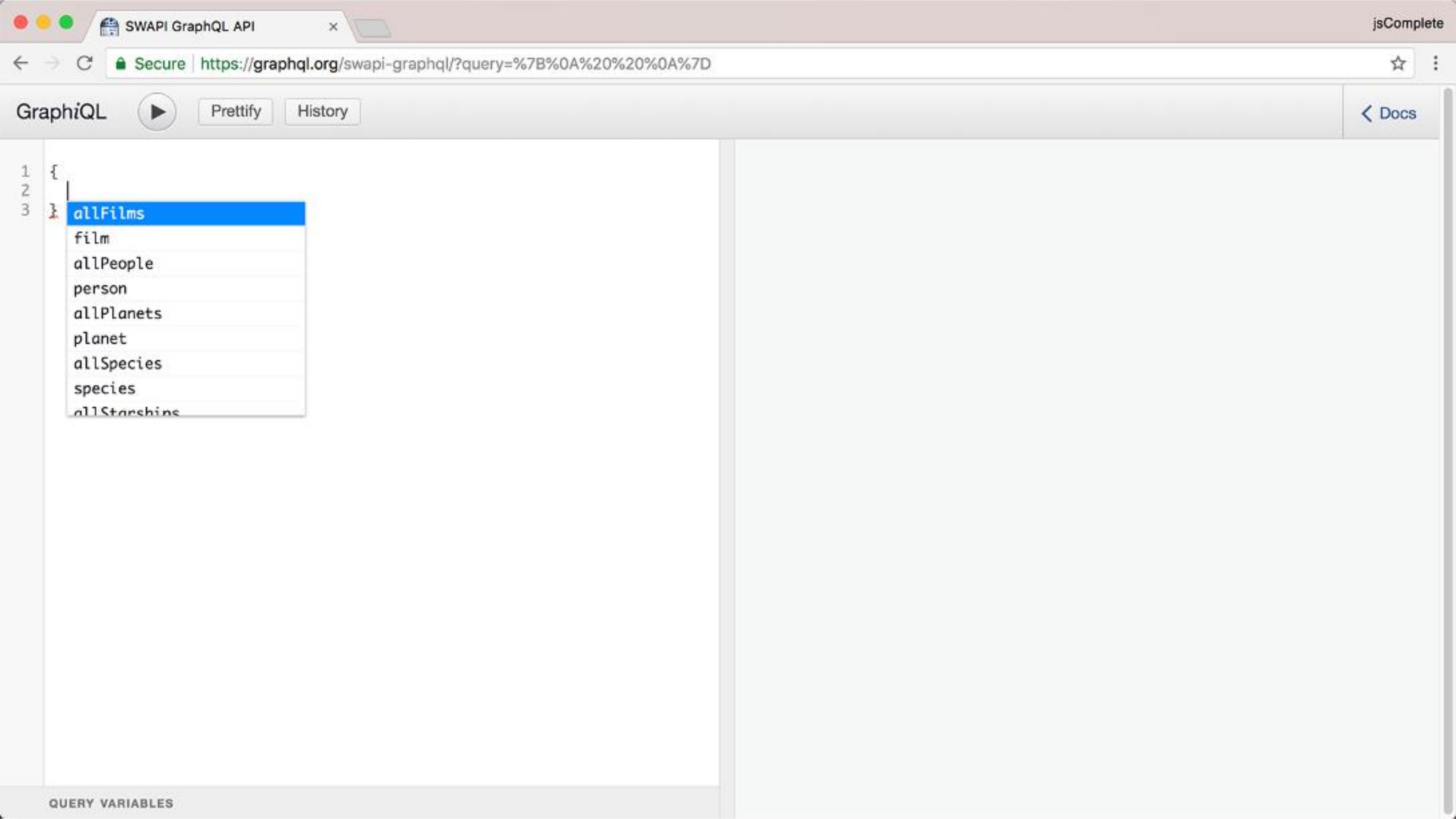
```
{
  person(personID: 4) {
    name
    birthYear
  }
}
```


SWAPI GraphQL API
graphql.org/swapi-graphql?query=%7B%0A%20%20person(personID%3A%204)%20%7B%0A%20%20%20%20name%0A%20%20%20%20birthYear%0A%20%20%7D%0A%7D
GraphiQL
Prettify
Merge
Copy
History

```
{
  person(personID: 4) {
    name
    birthYear
  }
}
```

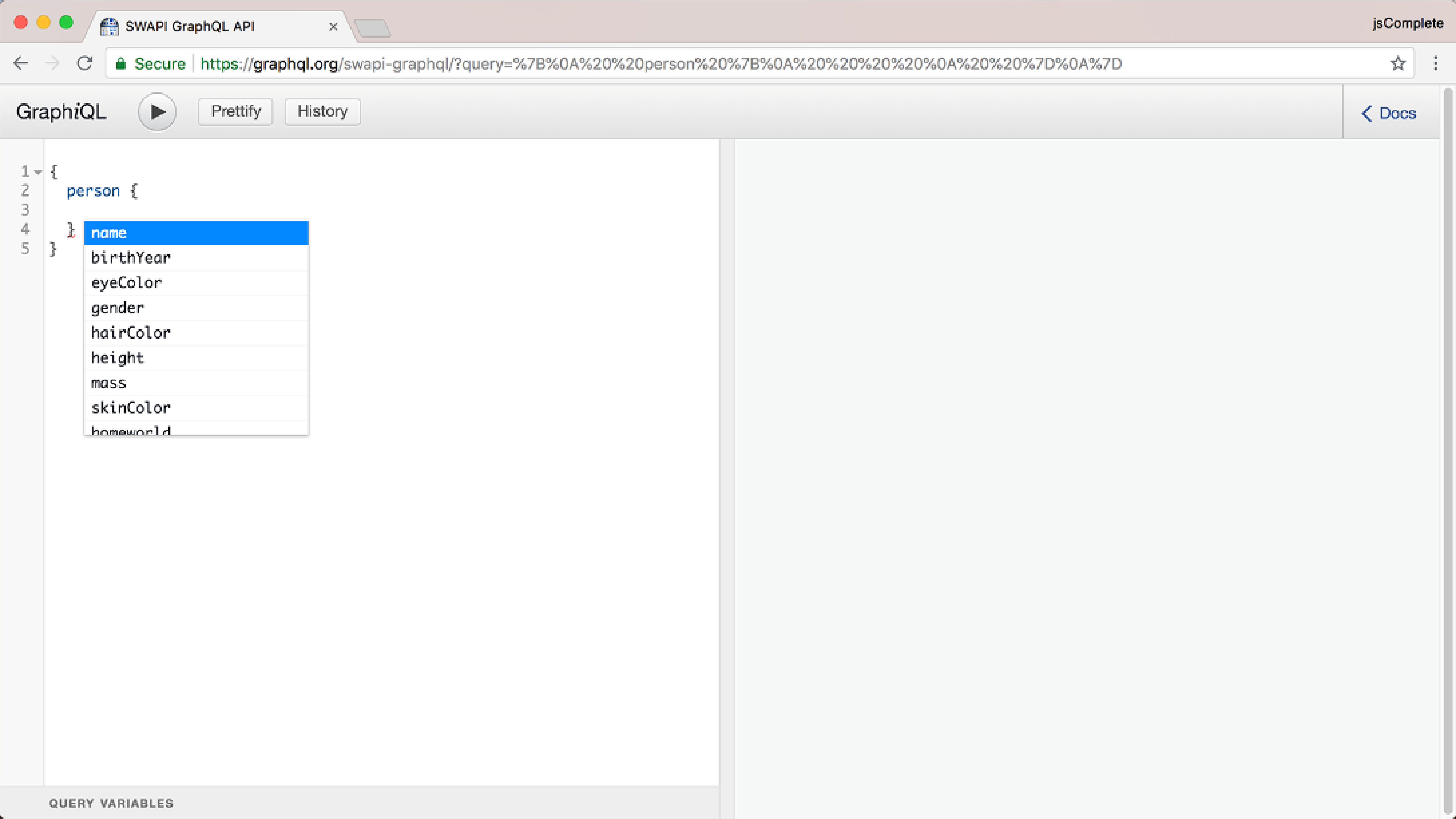
```
{
  "data": {
    "person": {
      "name": "Darth Vader",
      "birthYear": "41.9BBY"
    }
  }
}
```

The GraphiQL editor



```
{
  
}
```

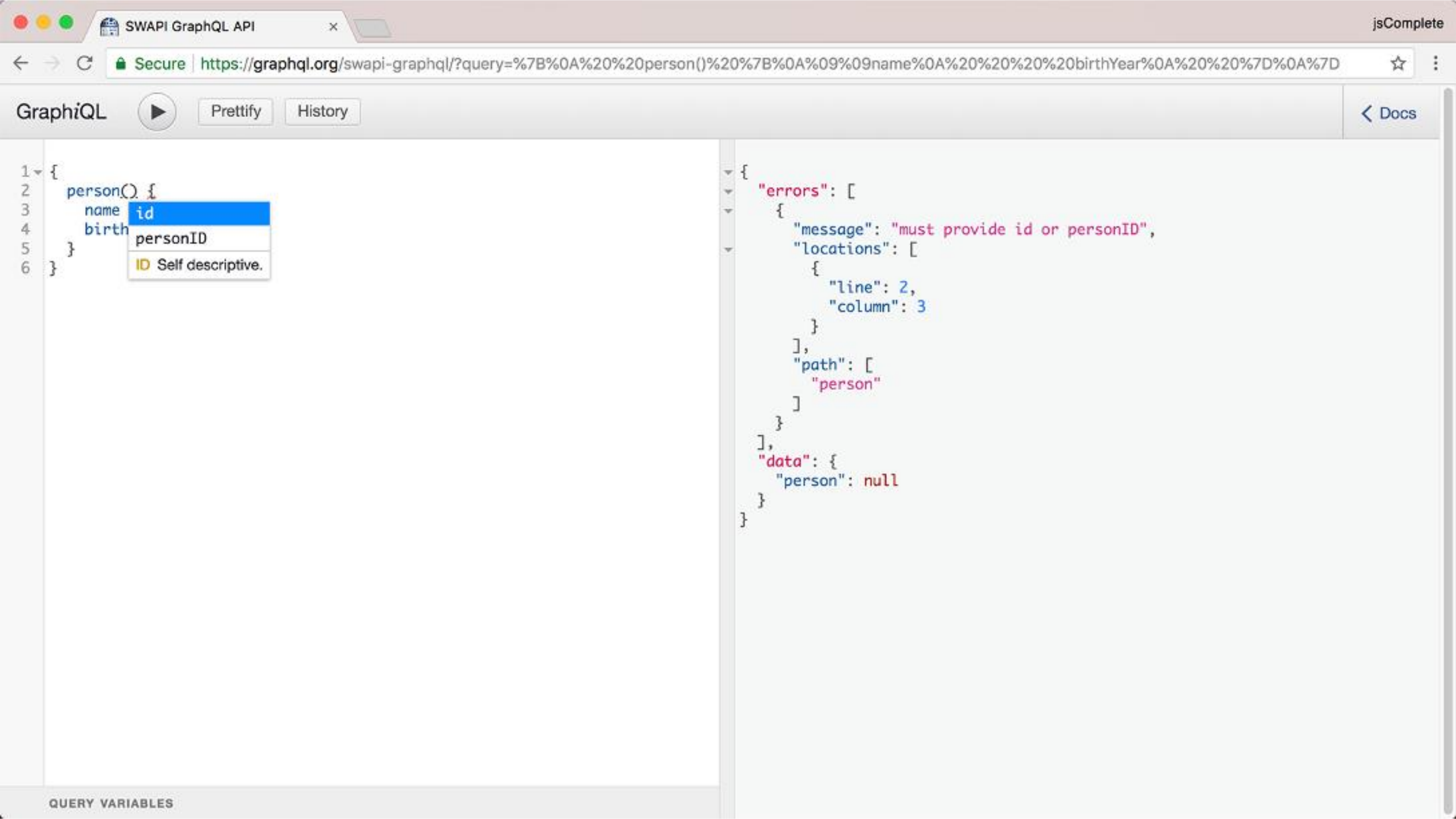
```
allFilms
film
allPeople
person
allPlanets
planet
allSpecies
species
allStarships
```


SWAPI GraphQL API
jsComplete
Secure
https://graphql.org/swapi-graphql/?query=%7B%0A%20%20person%20%7B%0A%20%20%20%20%0A%20%20%7D%0A%7D
GraphiQL
Prettify
History
Docs
QUERY VARIABLES

```
{
  person {

  }
}
```

```
name
birthYear
eyeColor
gender
hairColor
height
mass
skinColor
homeworld
```

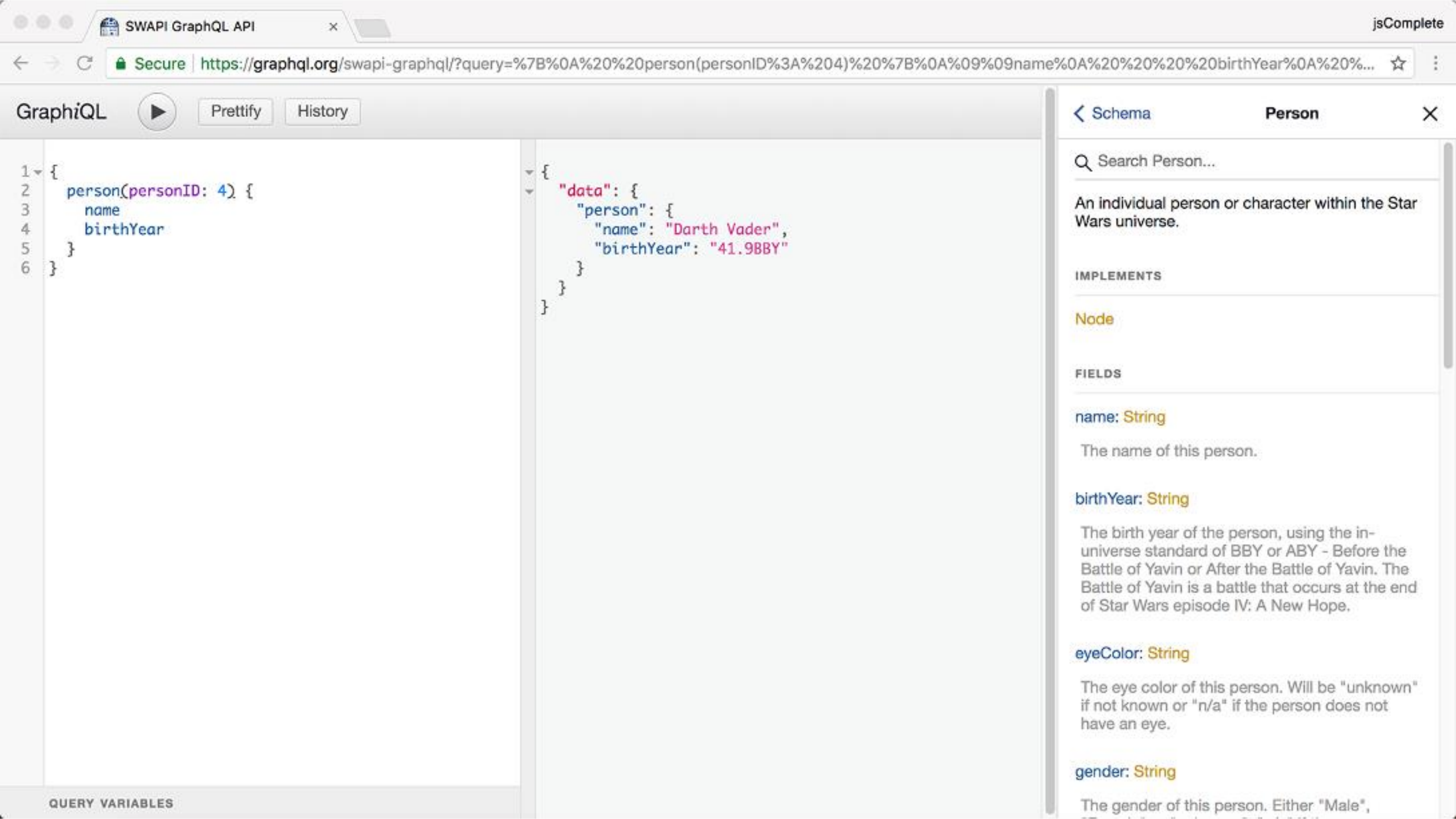
```
{
  "errors": [
    {
      "message": "must provide id or personID",
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ],
      "path": [
        "person"
      ]
    }
  ],
  "data": {
    "person": null
  }
}
```


SWAPI GraphQL API
jsComplete
Secure https://graphql.org/swapi-graphql/?query=%7B%0A%20%20person()%20%7B%0A%09%09name%0A%20%20%20%20birthYear%0A%20%20%7D%0A%7D
GraphiQL
Prettify
History
Docs
QUERY VARIABLES

```
{
  person() {
    name
    birth
  }
}
```

```
id
personID
ID Self descriptive.
```

```
{
  "errors": [
    {
      "message": "must provide id or personID",
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ],
      "path": [
        "person"
      ]
    }
  ],
  "data": {
    "person": null
  }
}
```


SWAPI GraphQL API
jsComplete
Secure
https://graphql.org/swapi-graphql/?query=%7B%0A%20%20person(personID%3A%204)%20%7B%0A%09%09name%0A%20%20%20%20birthYear%0A%20%...
GraphiQL
Prettify
History
QUERY VARIABLES
Schema
Person
Search Person...

```
{
  person(personID: 4) {
    name
    birthYear
  }
}
```

```
{
  "data": {
    "person": {
      "name": "Darth Vader",
      "birthYear": "41.9BBY"
    }
  }
}
```

An individual person or character within the Star Wars universe.

IMPLEMENTS

Node

FIELDS

name: String

The name of this person.

birthYear: String

The birth year of the person, using the in-universe standard of BBY or ABY - Before the Battle of Yavin or After the Battle of Yavin. The Battle of Yavin is a battle that occurs at the end of Star Wars episode IV: A New Hope.

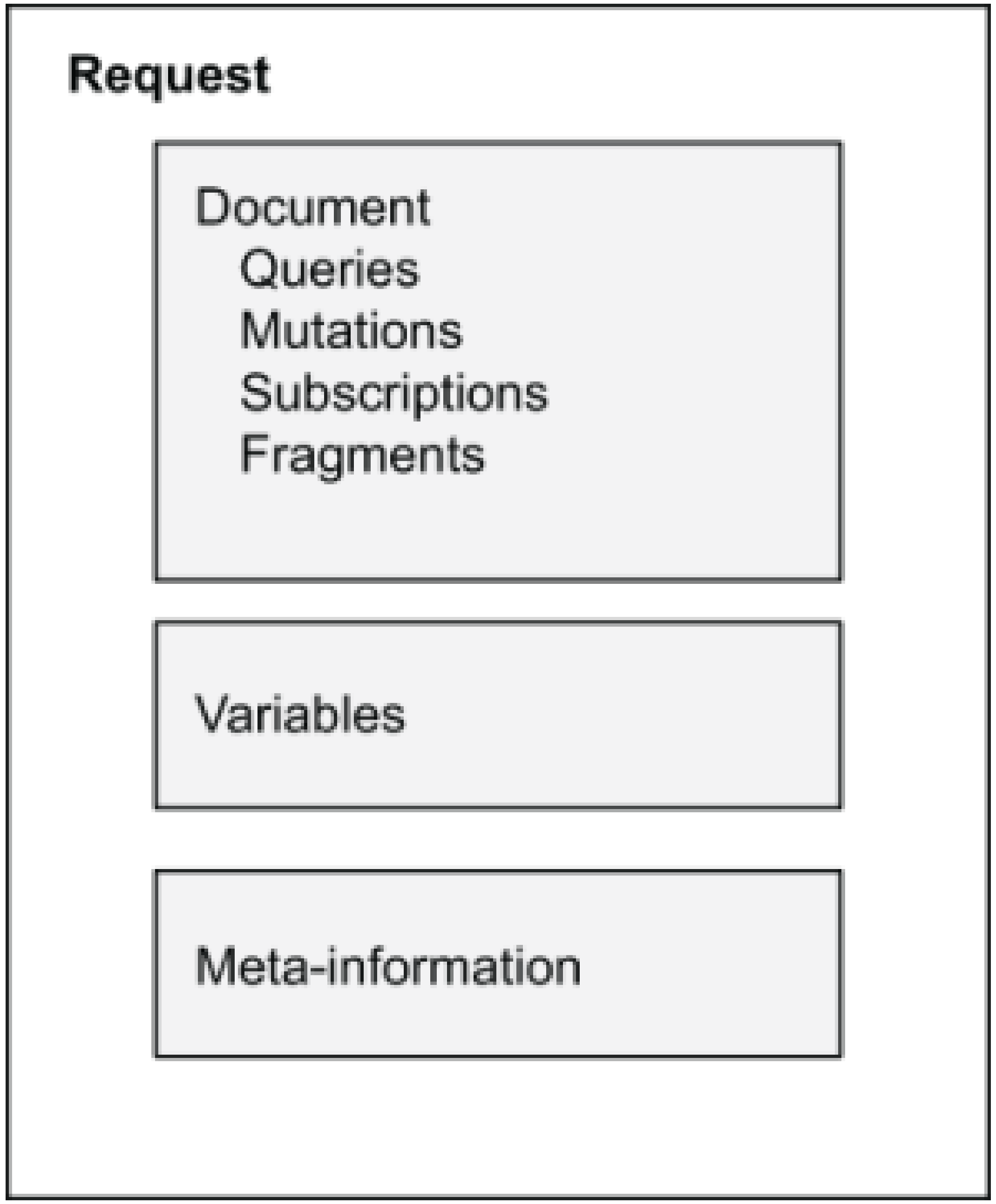
eyeColor: String

The eye color of this person. Will be "unknown" if not known or "n/a" if the person does not have an eye.

gender: String

The gender of this person. Either "Male",

Requests


Request
Document
Queries
Mutations
Subscriptions
Fragments
Variables
Meta-information

```
query GetEmployees($active: Boolean!) {
  allEmployees(active: $active) {
    ...employeeInfo
  }
}

query FindEmployee {
  employee(id: $employeeId) {
    ...employeeInfo
  }
}

fragment employeeInfo on Employee {
  name
  email
  startDate
}
```

Table of Contents

- The GraphiQL editor
- **The basics of the GraphQL language**
- Examples from the GitHub API

The basics of the GraphQL language

- Since this document uses generic variables (the ones starting with the $ sign), we need a JSON object to represent values specific to a request.

```
{
  "active": true,
  "employeeId": 42
}
```

The basics of the GraphQL language

- Also, since the document contains more than one operation (GetEmployees and FindEmployee), the request needs to provide the desired operation to be executed.

```
operationName="GetEmployees"
```

The basics of the GraphQL language

- The example in listing 2.3 represented a query operation. Here is a hypothetical example of a mutation operation.

```
mutation RateStory {
  addRating(storyId: 123, rating: 5) {
    story {
      averageRating
    }
  }
}
```

The basics of the GraphQL language

- Here is a hypothetical example of a subscription operation.

```
subscription StoriesRating {
  allStories {
    id
    averageRating
  }
}
```

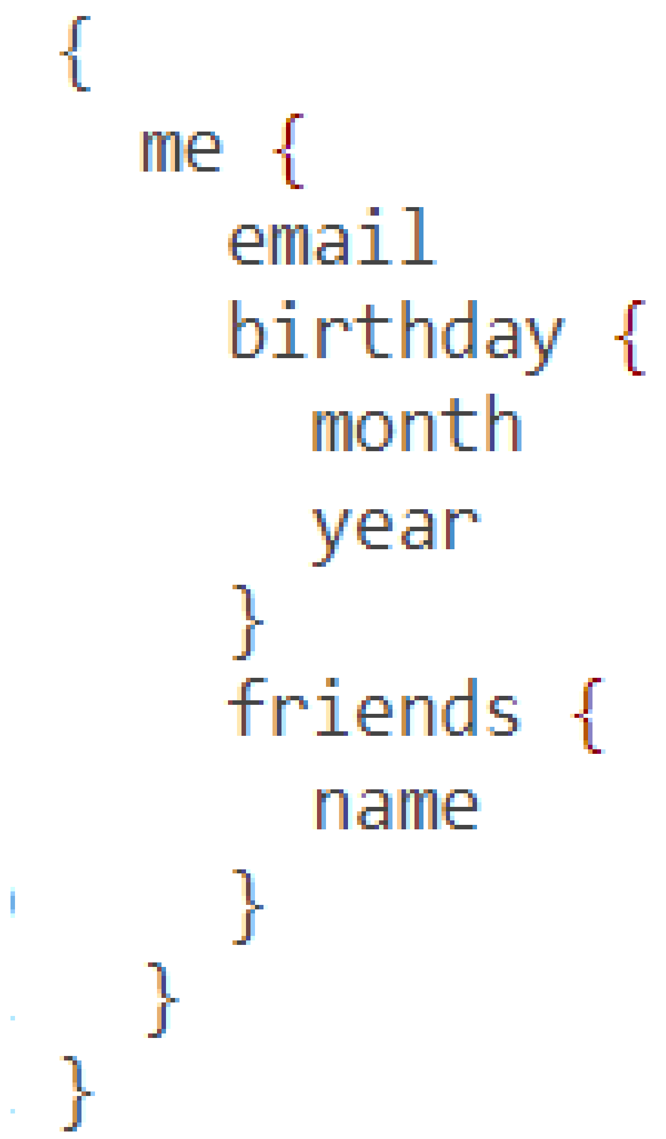
Fields

- One of the core elements in the text of a GraphQL operation is the field. The simplest way to think about a GraphQL operation is as a way to select fields on objects.

- A field always appears within a selection set (inside a pair of curly brackets), and it describes one discrete piece of information that you can retrieve about an object.

Fields

- Here is an example GraphQL query with different types of fields.



```
{
  me {
    email
    birthday {
      month
      year
    }
    friends {
      name
    }
  }
}
```

Fields

- Some typical examples of root fields include references to a currently logged-in user.
- These fields are often named viewer or me. For example:

```
{
  me {
    username
    fullName
  }
}
```

Fields

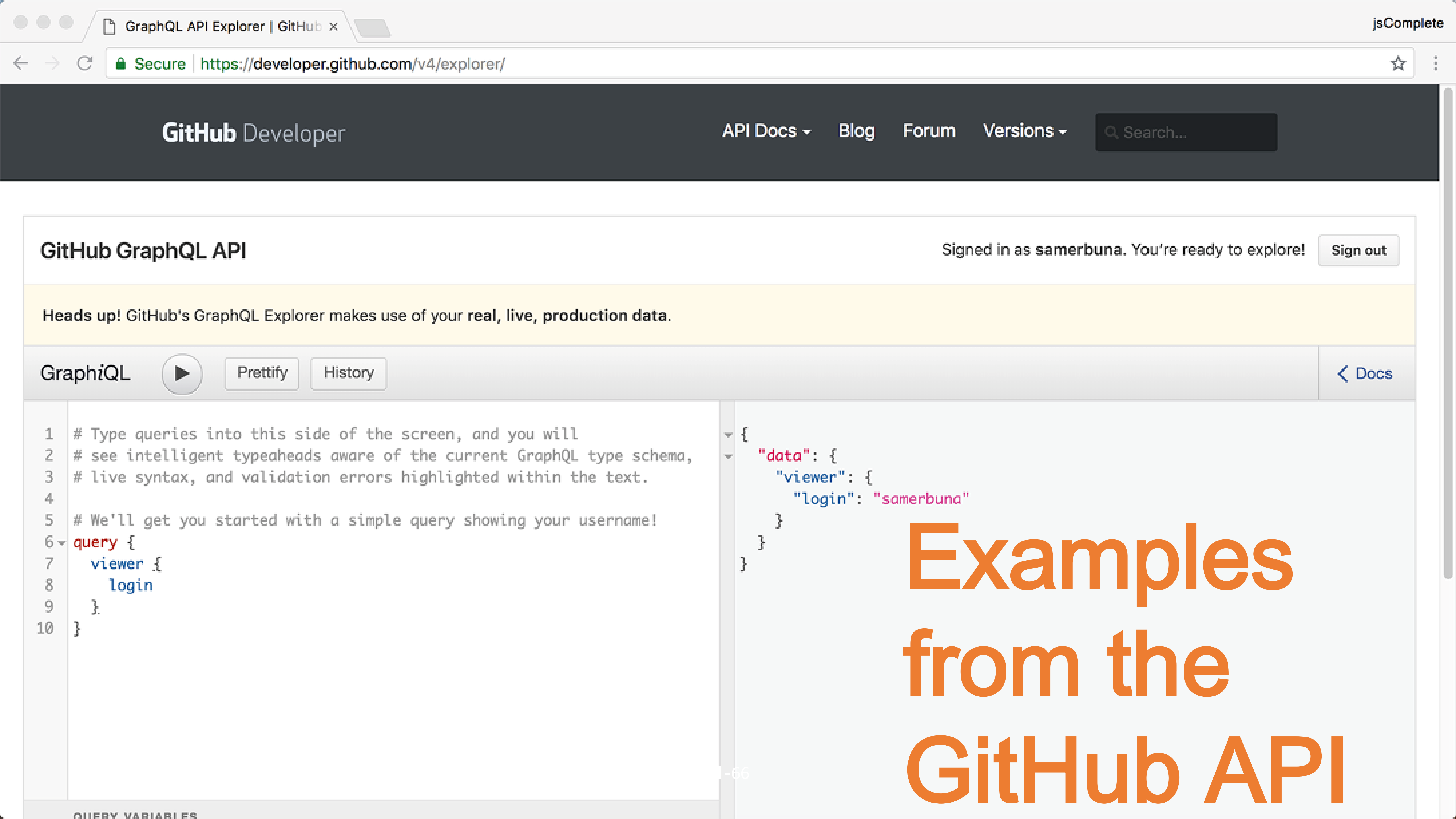
- Root fields are also generally used to access certain types of data referenced by a unique identifier. For example:

```
# Ask for the user whose ID equal to 42
{
  user(id: 42) {
    fullName
  }
}
```

TriveraTech TECHNOLOGY TRAINING

Table of Contents

- The GraphiQL editor
- The basics of the GraphQL language
- **Examples from the GitHub API**


GraphQL API Explorer | GitHub
jsComplete
Secure https://developer.github.com/v4/explorer/
GitHub Developer
API Docs
Blog
Forum
Versions
Search...
GitHub GraphQL API
Signed in as samerbuna. You're ready to explore!
Sign out
Heads up! GitHub's GraphQL Explorer makes use of your real, live, production data.
GraphiQL
Prettify
History
Docs
QUERY VARIABLES

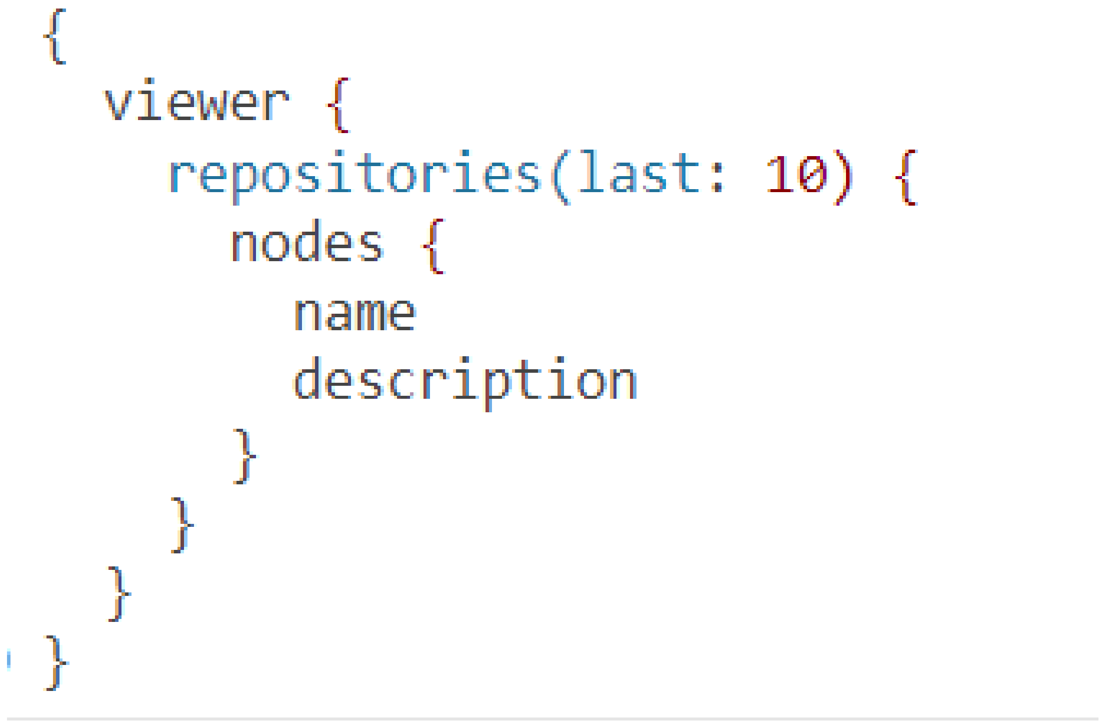
```
# Type queries into this side of the screen, and you will
# see intelligent typeaheads aware of the current GraphQL type schema,
# live syntax, and validation errors highlighted within the text.

# We'll get you started with a simple query showing your username!
query {
  viewer {
    login
  }
}
```

```
{
  "data": {
    "viewer": {
      "login": "samerbuna"
    }
  }
}
```

Examples from the GitHub API

Reading data from GitHub

- For example, here is a query to see information about the most recent 10 repositories that you own or contribute to.



```
{
  viewer {
    repositories(last: 10) {
      nodes {
        name
        description
      }
    }
  }
}
```

Examples from the GitHub API

- Here is another query to see all the supported licenses in GitHub along with their URLs.

```
{
  licenses {
    name
    url
  }
}
```

TriveraTech TECHNOLOGY TRAINING

```
{
  repository(owner: "facebook", name: "graphql")
{
    issues(first: 10) {
      nodes {
        title
        createdAt
        author {
          login
        }
      }
    }
  }
}
```

Updating data at GitHub

```
mutation {
  addStar(input: { starrableId:
"MDEwOlJlcG9zaXRvcnkxMjU2ODEwMDY=" }) {
    starrable {
      stargazers {
        totalCount
      }
    }
  }
}
```

TriveraTech TECHNOLOGY TRAINING

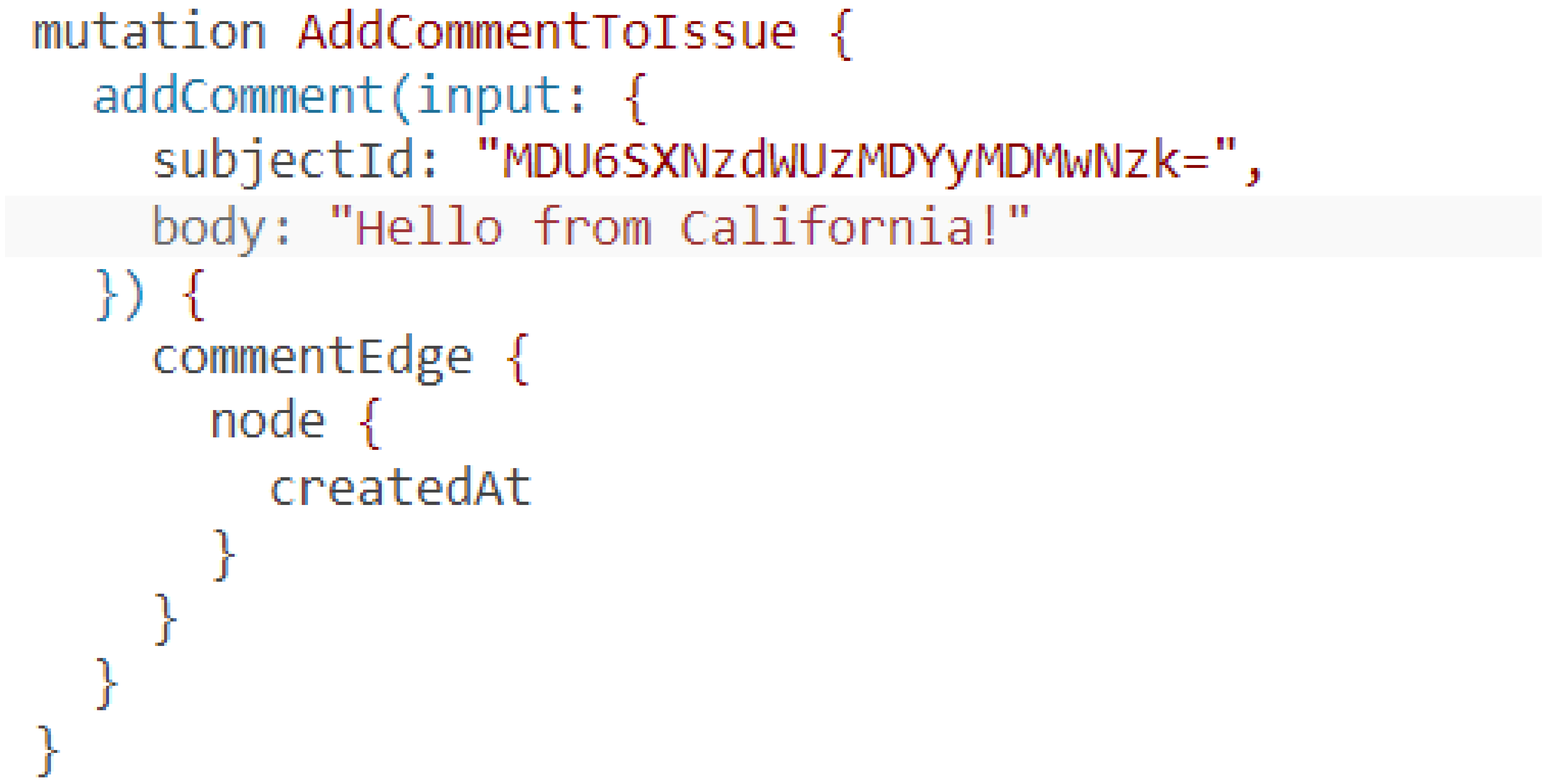
Updating data at GitHub

```
{
  repository(name: "graphql", owner: "fenago") {
    id
  }
}
```

Updating data at GitHub

```
query GetIssueInfo {
  repository(owner: "fenago", name: "graphql") {
    issue(number: 1) {
      id
      title
    }
  }
}
```

Updating data at GitHub



```
mutation AddCommentToIssue {
  addComment(input: {
    subjectId: "MDU6SXNzdWUzMDYyMDMwNzk=",
    body: "Hello from California!"
  }) {
    commentEdge {
      node {
        createdAt
      }
    }
  }
}
```

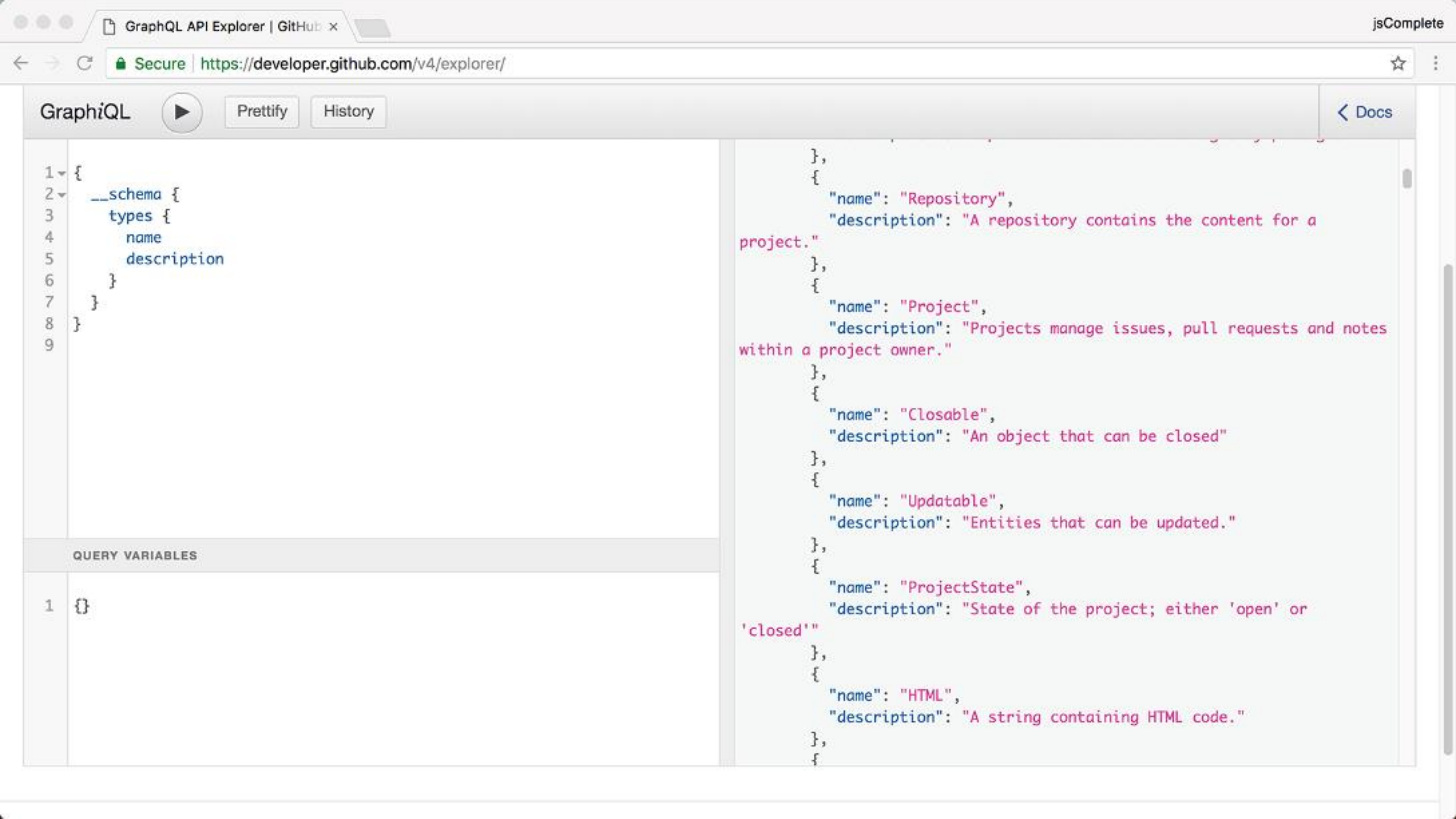
Introspective queries

- GraphQL APIs support introspective queries that can be used to answer questions about the API schema.

- This introspection support gives GraphQL tools powerful functionality, and it drives the features we have been using in the GraphiQL editor.

Introspective queries

```
{
  __schema {
    types {
      name
      description
    }
  }
}
```


GraphQL API Explorer | GitHub
jsComplete
Secure https://developer.github.com/v4/explorer/
GraphiQL
Prettify
History
Docs
QUERY VARIABLES

```
{
  __schema {
    types {
      name
      description
    }
  }
}
```

```
{}
```

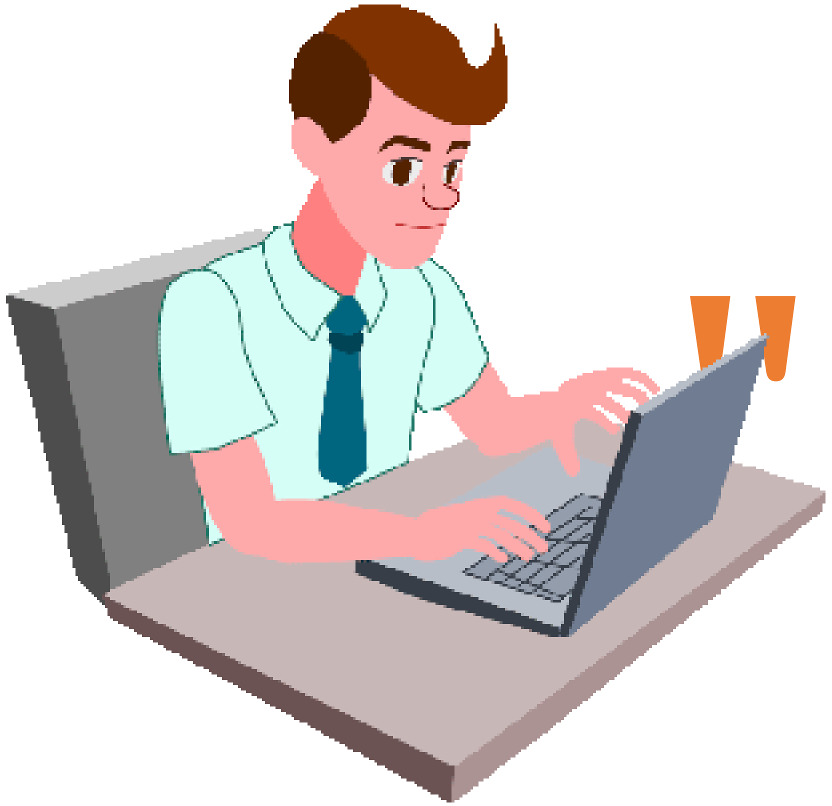
```
        },
        {
          "name": "Repository",
          "description": "A repository contains the content for a
project."
        },
        {
          "name": "Project",
          "description": "Projects manage issues, pull requests and notes
within a project owner."
        },
        {
          "name": "Closable",
          "description": "An object that can be closed"
        },
        {
          "name": "Updatable",
          "description": "Entities that can be updated."
        },
        {
          "name": "ProjectState",
          "description": "State of the project; either 'open' or
'closed'"
        },
        {
          "name": "HTML",
          "description": "A string containing HTML code."
        },
        {
```

Updating data at GitHub

```
{
  __type(name: "Commit") {
    fields {
      name
      args {
        name
      }
    }
  }
}
```

Summary

- GraphiQL is an in-browser IDE for writing and testing GraphQL requests.
- It offers many great features to write, validate, and inspect GraphQL queries and mutations.
- These features are made possible thanks to GraphQL's introspective nature, which comes with its mandatory schemas.



"Complete Lab"

3: Preparing Your Development Environment

Preparing Your Development Environment

This lesson covers the following topics:

- Architecture and technology
- Thinking critically about how to architect a stack
- Building the React and GraphQL stack
- Installing and configuring Node.js
- Setting up a React development environment with webpack, Babel, and other requirements
- Debugging React applications using Chrome DevTools and React Developer Tools
- Using webpack-bundle-analyzer to check the bundle size

Application architecture

- Since its initial release in 2015, GraphQL has become the new alternative to the standard SOAP and REST APIs.
- GraphQL is a specification, like SOAP and REST, that you can follow to structure your application and data flow.
- It is so innovative because it allows you to query specific fields of entities, such as users and posts.

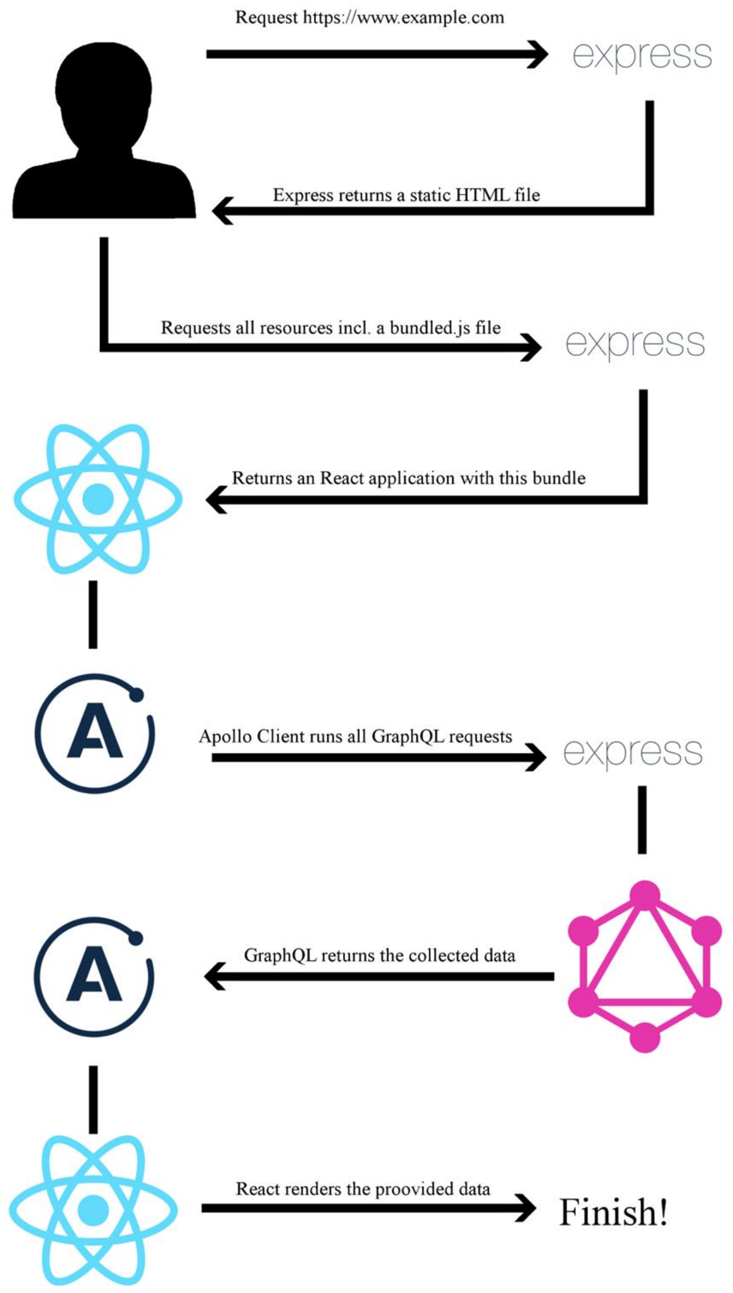
Application architecture

- For example, a query for a post may look like this:

```
post {
  id
  text
  user {
    user_id
    name
  }
}
```

The basic setup

- The basic setup to make an application work is the logical request flow, which looks as follows:


Request https://www.example.com
express
Express returns a static HTML file
Requests all resources incl. a bundled.js file
express
Returns an React application with this bundle
Apollo Client runs all GraphQL requests
express
GraphQL returns the collected data
React renders the proovided data
Finish!

Installing and configuring Node.js

The first step for preparing for our project is to install Node.js. There are two ways to do this:

- One option is to install the Node Version Manager (NVM).
- The benefit of using NVM is that you are easily able to run multiple versions of Node.js side by side and this handles the installation process for you on nearly all UNIX-based systems, such as Linux and macOS.
- Within this course, we do not need the option to switch between different versions of Node.js.

Installing and configuring Node.js

- First, let's add the correct repository for our package manager by running:

```
curl -sL https://deb.nodesource.com/setup_10.x | sudo -E bash –
```

- Next, install Node.js and the build tools for native modules, using the following command:

```
sudo apt-get install -y nodejs build-essential
```

- Finally, let's open a terminal now and verify that the installation was successful:

```
node --version
```

TriveraTech TECHNOLOGY TRAINING

Setting up React

- The development environment for our project is ready.
- In this section, we are going to install and configure React, which is one primary aspect of this course.
- Let's start by creating a new directory for our project:

```
mkdir ~/graphbook
cd ~/graphbook
```

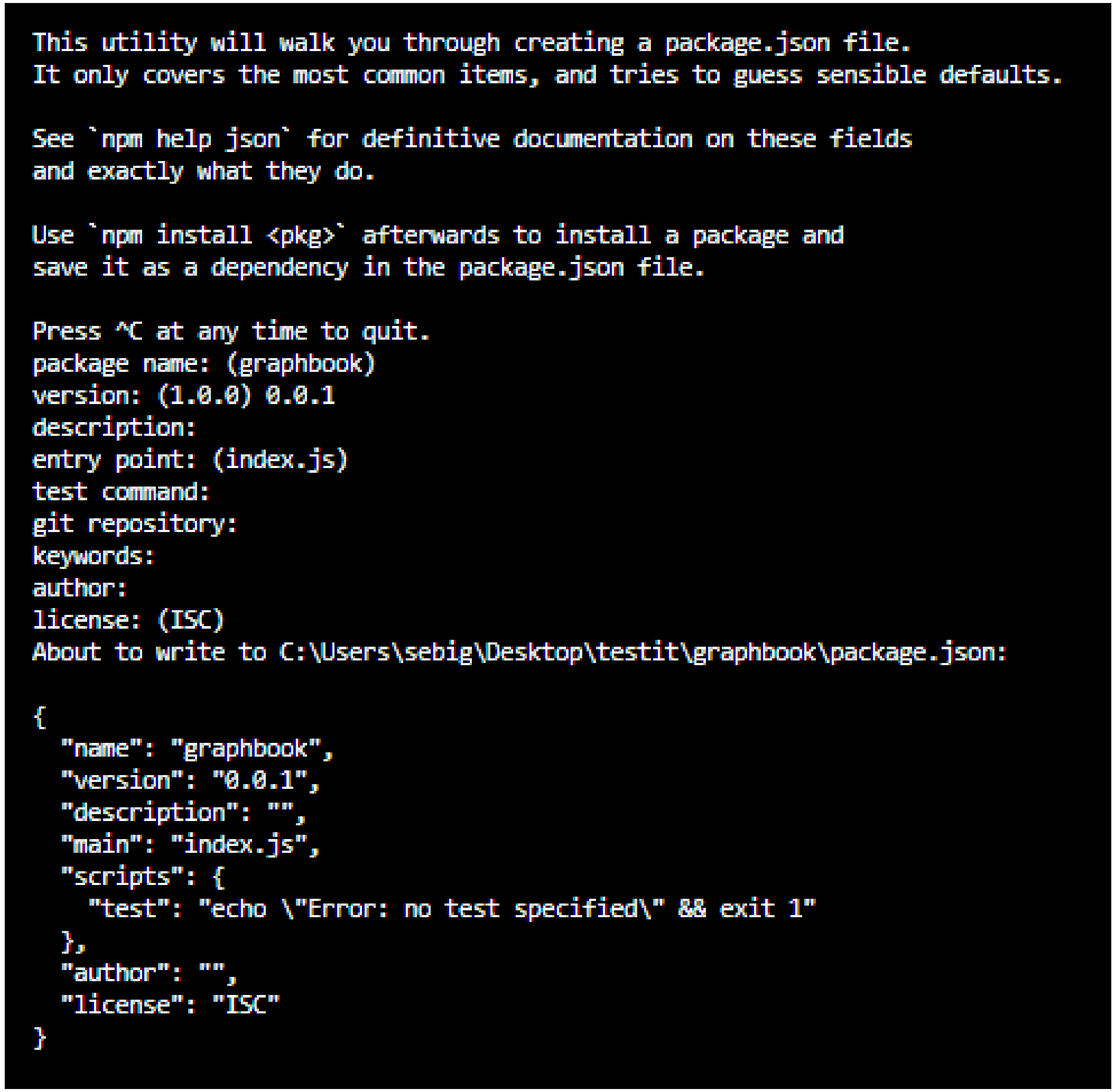
Setting up React

- Just run npm init to create an empty package.json file:

```
npm init
```

Setting up React

You can see an example of the command line in the following screenshot:



```
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (graphbook)
version: (1.0.0) 0.0.1
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to C:\Users\sebig\Desktop\testit\graphbook\package.json:

{
  "name": "graphbook",
  "version": "0.0.1",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Setting up React

- The first and most crucial dependency for this course is React.

- Use npm to add React to our project:

```
npm install --save react react-dom
```

Preparing and configuring webpack

- Create a separate directory for our index.html file:

```
mkdir public
touch index.html
```

- Save this inside index.html:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
      scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Graphbook</title>
  </head>
  <body>
    <div id="root"></div>
  </body>
</html>
```

Preparing and configuring webpack

- To bundle our JavaScript code, we need to install webpack and all of its dependencies as follows:

```
npm install --save-dev @babel/core babel-eslint babel-loader
@babel/preset-env @babel/preset-react clean-webpack-plugin css-loader
eslint file-loader html-webpack-plugin style-loader url-loader webpack
webpack-cli webpack-dev-server @babel/plugin-proposal-decorators
@babel/plugin-proposal-function-sent @babel/plugin-proposal-export-
namespace-from @babel/plugin-proposal-numeric-separator
@babel/plugin-proposal-throw-expressions @babel/plugin-proposal-class-
properties
```

Preparing and configuring webpack

- The following handy shortcut installs the eslint configuration created by the people at Airbnb, including all peer dependencies.
- Execute it straight away:

```
npx install-peerdeps --dev eslint-config-airbnb
```

Preparing and configuring webpack

- Create a .eslintrc file in the root of your project folder to use the airbnb configuration:

```
{
  "extends": ["airbnb"],
  "env": {
    "browser": true,
    "node": true
  },
  "rules": {
    "react/jsx-filename-extension": "off"
  }
}
```

Preparing and configuring webpack

- Enter the following:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/1_1.txt

Preparing and configuring webpack

- With this in mind, let's move on.
- We are missing the src/client/index.js file from our webpack configuration, so let's create it as follows:

```
mkdir src/client
cd src/client
touch index.js
```

Preparing and configuring webpack

- Add this line to the scripts object inside package.json:

```
"client": "webpack-dev-server --devtool inline-source-map --hot --config webpack.client.config.js"
```

Render your first React component

- There are many best practices for React. The central philosophy behind it is to split up our code into separate components where possible.

- We are going to cover this approach in more detail later in lesson 5, Reusable React Components.

Render your first React component

- The index.js file should include this code:

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

ReactDOM.render(<App/>,
document.getElementById('root'));
```

Render your first React component

- Create the App.js file next to your index.js file, with the following content:

```
import React, { Component } from 'react';
export default class App extends Component {
  render() {
    return (
      <div>Hello World!</div>
    )
  }
}
```

Render your first React component

- Let's create a .babelrc file in the root folder with this content:

```
{
  "plugins": [
    ["@babel/plugin-proposal-decorators", { "legacy": true }],
    "@babel/plugin-proposal-function-sent",
    "@babel/plugin-proposal-export-namespace-from",
    "@babel/plugin-proposal-numeric-separator",
    "@babel/plugin-proposal-throw-expressions",
    ["@babel/plugin-proposal-class-properties", { "loose": false }]
  ],
  "presets": ["@babel/env","@babel/react"]
}
```

Rendering arrays from React state

- Hello World! is a must for every good programming course, but this is not what we are aiming for when we use React.

- A social network such as Facebook or Graphbook, which we are writing at the moment, needs a news feed and an input to post news.

Let's implement this.

- Define a new variable above your App class like this:

```
const posts = [{
  id: 2,
  text: 'Lorem ipsum',
  user: {
    avatar: '/uploads/avatar1.png',
    username: 'Test User'
  }
},
{
  id: 1,
  text: 'Lorem ipsum',
  user: {
    avatar: '/uploads/avatar2.png',
    username: 'Test User 2'
  }
}];
```

- Replace the current content of your render method with the following code:

```
const { posts } = this.state;
return (
  <div className="container">
    <div className="feed">
      {posts.map((post, i) =>
        <div key={post.id} className="post">
          <div className="header">
            <img src={post.user.avatar} />
            <h2>{post.user.username}</h2>
          </div>
          <p className="content">
            {post.text}
          </p>
        </div>
      )}
    </div>
  </div>
)
```

Rendering arrays from React state

- To get our posts into the state, we can define them inside our class with property initializers.

- Add this to the top of the App class:

```
state = {
  posts: posts
}
```

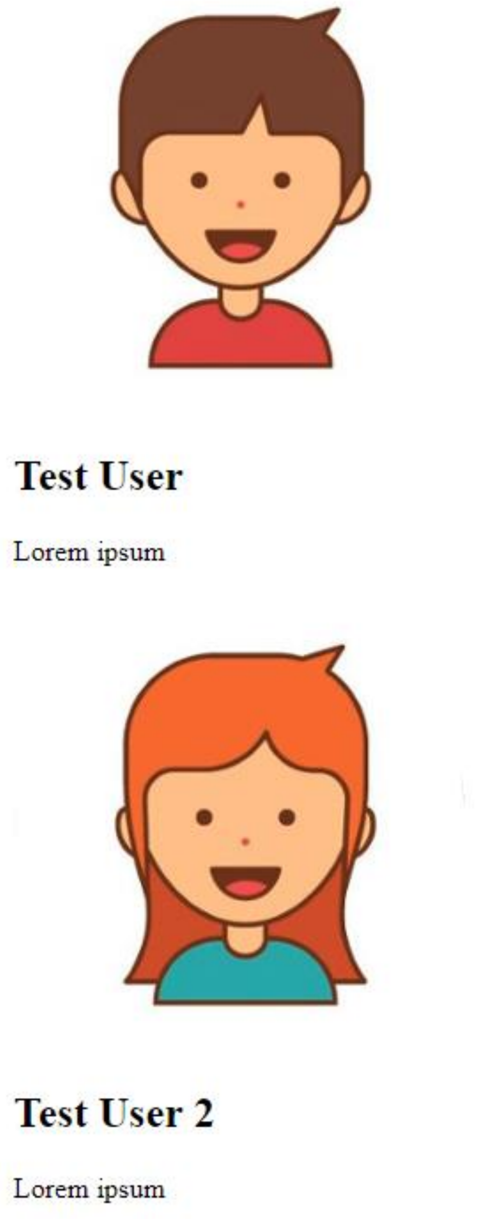
Rendering arrays from React state

- The older way of implementing this—without using the ES6 feature—was to create a constructor:

```
constructor(props) {
  super(props);

  this.state = {
    posts: posts
  };
}
```

Rendering arrays from React state

- The preceding method is much cleaner, and I recommend this for readability purposes.
- When saving, you should be able to see rendered posts. They should look like this:



Test User

Lorem ipsum

Test User 2

Lorem ipsum

CSS with webpack

- The posts from the preceding picture have not been designed yet.
- I have already added CSS classes to the HTML our component returns.
- Instead of using CSS to make our posts look better, another method is to use CSS-in-JS using packages such as styled-components, which is a React package.

CSS with webpack

- What we've already done in our webpack.client.config.js file is to specify a CSS rule, as you can see in the following code snippet:

```
{
  test: /\.css$/,
  use: ['style-loader', 'css-loader'],
},
```

TriveraTech
TECHNOLOGY TRAINING

CSS with webpack

- Create a style.css file in/assets/cssand fill in the following:

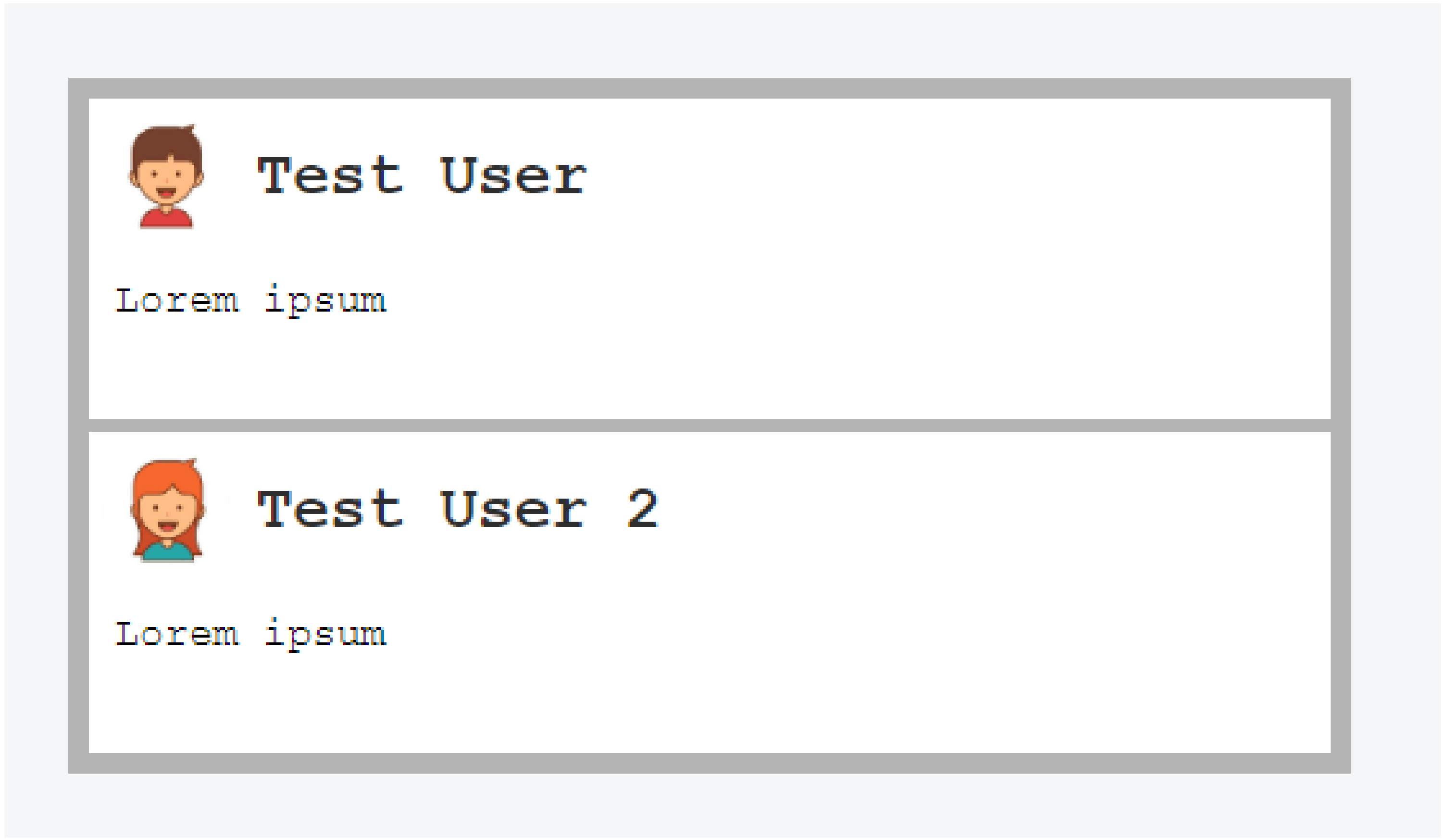
Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/1_2.txt

CSS with webpack

- In your App.js file, add the following behind the React import statement:

```
import '../../assets/css/style.css';
```

CSS with webpack



Test User

Lorem ipsum

Test User 2

Lorem ipsum

Event handling and state updates with React

- Add this above the div with the feed class:

```
<div className="postForm">
  <form onSubmit={this.handleSubmit}>
    <textarea value={postContent}
onChange={this.handlePostContentChange}
      placeholder="Write your custom post!"/>
    <input type="submit" value="Submit" />
  </form>
</div>
```

Event handling and state updates with React

- Create an empty string variable at the state property initializer, as follows:

```
state = {
  posts: posts,
  postContent: ''
}
```

- Then, extract this from the class state inside the render method:

```
const { posts, postContent } = this.state;
```

Event handling and state updates with React

- The logical step is to implement this function:

```
handlePostContentChange = (event) => {
  this.setState({postContent: event.target.value})
}
```

- Maybe you are used to writing this a little differently, like this:

```
handlePostContentChange(event) {
  this.setState({postContent: event.target.value})
}
```

Event handling and state updates with React

- In this case, you would need to write a constructor for your class and manually bind the scope to your function as follows:

```
this.handlePostContentChange =
this.handlePostContentChange.bind(this);
```

- Look at your browser again.
- The form is there, but it is not pretty, so add this CSS:

```
form {
  padding-bottom: 20px;
}
form textarea {
  width: calc(100% - 20px);
  padding: 10px;
  border-color: #bbb;
}
form [type=submit] {
  border: none;
  background-color: #6ca6fd;
  color: #fff;
  padding: 10px;
  border-radius: 5px;
  font-size: 14px;
  float: right;
}
```

- The last step is to implement the handleSubmit function for our form:

```
handleSubmit = (event) => {
  event.preventDefault();
  const newPost = {
    id: this.state.posts.length + 1,
    text: this.state.postContent,
    user: {
      avatar: '/uploads/avatar1.png',
      username: 'Fake User'
    }
  };
  this.setState((prevState) => ({
    posts: [newPost, ...prevState.posts],
    postContent: ''
  }));
}
```

Controlling document heads with React Helmet

- When developing a web application, it is crucial that you can control your document heads.

- You might want to change the title or description, based on the content you are presenting.

- React Helmet is a great package that offers this on the fly, including overriding multiple headers and server-side rendering.

Controlling document heads with React Helmet

- Install it with the following command:

```
npm install --save react-helmet
```

Controlling document heads with React Helmet

- Import react-helmet at the top of the file:

```
import { Helmet } from 'react-helmet';
```

- Add Helmet itself directly above postFormdiv:

```
<Helmet>
  <title>Graphbook - Feed</title>
  <meta name="description" content="Newsfeed of all
your friends on
    Graphbook" />
</Helmet>
```

Production build with webpack

- A production bundle does merge all JavaScript files, but also CSS files into two separate files.
- Those can be used directly in the browser.
- To bundle CSS files, we will rely on another webpack plugin, called MiniCss:

```
npm install --save-dev mini-css-extract-plugin
```

Production build with webpack

- We do not want to change the current webpack.client.config.js file, because it is made for development work.

- Add this command to the scripts object of your package.json:

```
"client:build": "webpack --config
webpack.client.build.config.js"
```

Production build with webpack

- The mode needs to be production, not development.

- Require the MiniCss plugin:

```
const MiniCssExtractPlugin = require('mini-css-extract-plugin');
```

Production build with webpack

- Replace the current CSS rule:

```
{
  test: /\.css$/,
  use: [{ loader: MiniCssExtractPlugin.loader,
    options: {
      publicPath: '../'
    }
 }, 'css-loader'],
},
```

Production build with webpack

- Lastly, add the plugin to the plugins at the bottom of the configuration file:

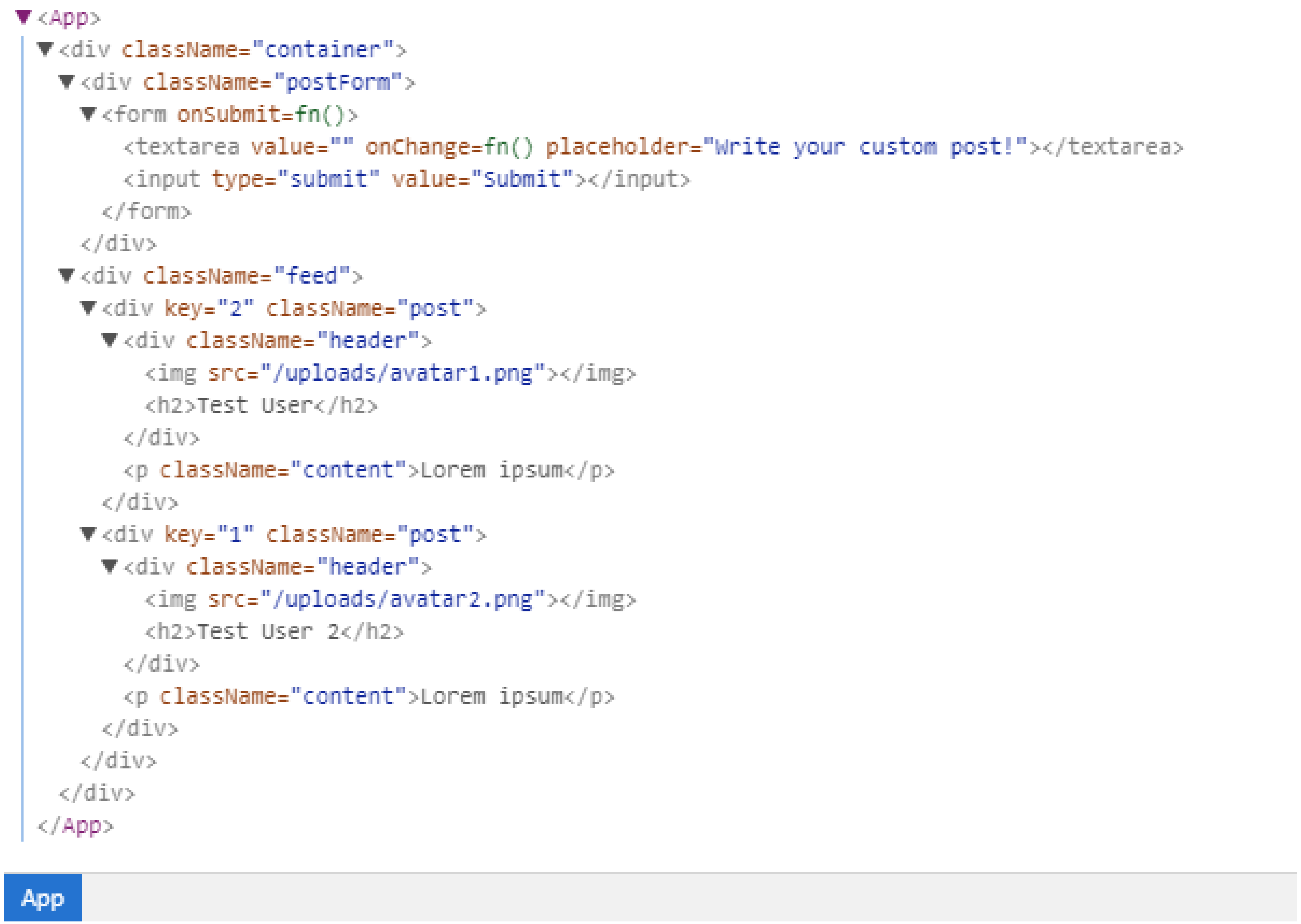
```
new MiniCssExtractPlugin({
  filename: 'bundle.css',
})
```

Useful development tools

- When working with React, you will want to know why your application rendered in the way that it did.

- You need to know which properties your components received and how their current state looks.

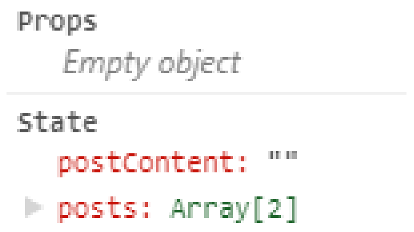
- Since this is not displayed in the DOM or anywhere else in Chrome DevTools, you need a separate plugin.



```
▼<App>
  ▼<div className="container">
    ▼<div className="postForm">
      ▼<form onSubmit=fn()>
          <textarea value="" onChange=fn() placeholder="Write your custom post!"></textarea>
          <input type="submit" value="Submit"></input>
        </form>
      </div>
    ▼<div className="feed">
      ▼<div key="2" className="post">
        ▼<div className="header">
            <img src="/uploads/avatar1.png"></img>
            <h2>Test User</h2>
          </div>
          <p className="content">Lorem ipsum</p>
        </div>
      ▼<div key="1" className="post">
        ▼<div className="header">
            <img src="/uploads/avatar2.png"></img>
            <h2>Test User 2</h2>
          </div>
          <p className="content">Lorem ipsum</p>
        </div>
      </div>
    </div>
  </App>
```

Useful development tools

- By clicking a component, your right-hand panel will show its properties, state, and context.
- You can try this with the App component, which is the only real React component:



```
Props
   Empty object
State
   postContent: ""
 ▶ posts: Array[2]
```

Analyzing bundle size

- People that are trying to use as little bandwidth as possible will want to keep their bundle size low.

- I recommend that you always keep an eye on this, especially when requiring more modules via npm.

- In this case, you can quickly end up with a huge bundle size, since npm packages tend to require other npm packages themselves.

Analyzing bundle size

- Install this with the following:

```
npm install --save-dev webpack-bundle-analyzer
```

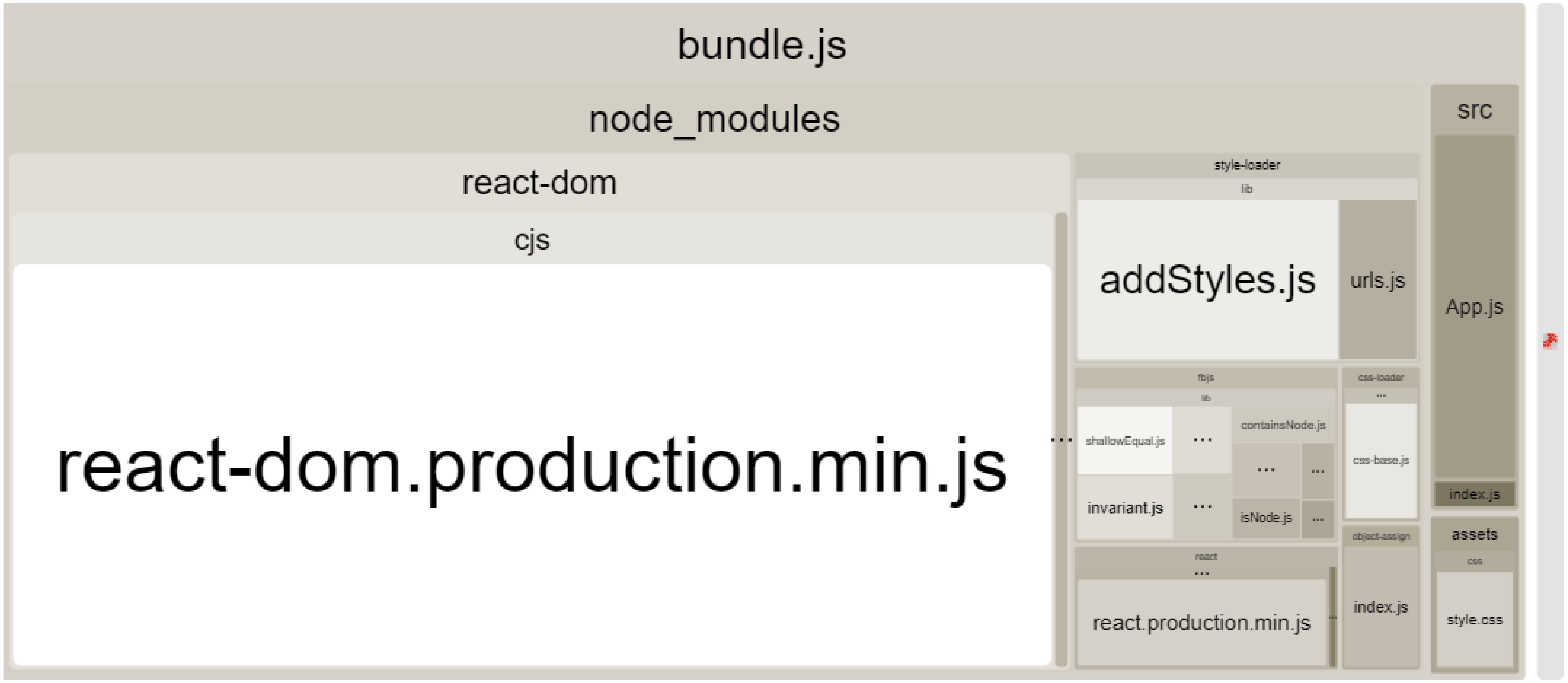
Analyzing bundle size

- The analyze command spins up the webpack-bundle-analyzer, showing us how our bundle is built together and how big each package that we use is.

- Do this as follows:

```
npm run stats
npm run analyze
```

Analyzing bundle size



Summary

- In this lesson, we completed a working React setup.
- This is a good starting point for our front end. We can write and build static web pages with this setup.
- The next lesson primarily focuses on our setup for the back end.

"Complete Lab"

4: Setting up GraphQL with Express.js

Setting up GraphQL with Express.js

This lesson covers the following points:

- Express.js installation and explanation
- Routing in Express.js
- Middleware in Express.js
- Binding Apollo Server to a GraphQL endpoint
- Serving static assets with Express.js
- Back end debugging and logging

Node.js and Express.js

- One primary goal of this course is to set up a GraphQL API, which is then consumed by our React front end.

- To accept network requests (especially GraphQL requests), we are going to set up a Node.js web server.

- The most significant competitors in the Node.js web server area are Express.js, Koa, and Hapi. In this course, we are going to use Express.js.

Node.js and Express.js

- Installing Express.js is pretty easy.

- We can use npm in the same way as in the first lesson:

```
npm install --save express
```

Node.js and Express.js

- In the first lesson, we created all JavaScript files directly in the src/client folder.
- Now, let's create a separate folder for our server-side code.
- This separation gives us a tidy directory structure.
- We will create the folder with the following command:

```
mkdir src/server
```

TriveraTech
TECHNOLOGY TRAINING

Setting up Express.js

- First, we import express from node_modules, which we just installed. We can use import here since our back end gets transpiled by Babel.

- We are also going to set up webpack for the server-side code in a later in lesson 9, Implementing Server-Side Rendering.

```
import express from 'express';
```

Setting up Express.js

- We initialize the server with the express command.
- The result is stored in the app variable.
- Everything our back end does is executed through this object.

```
const app = express();
```

Setting up Express.js

- Then, we specify the routes that accept requests.
- For this straightforward introduction, we accept all HTTP GET requests matching any path, by using the app.get method.
- Other HTTP Methods are catchable with app.post, app.put, and so on.

```
app.get('*', (req, res) => res.send('Hello World!'));
app.listen(8000, () => console.log('Listening on port 8000!'));
```

TriveraTech
TECHNOLOGY TRAINING

Running Express.js in development

- We will add the following line to the scripts property of the package.json file:

```
"server": "nodemon --exec babel-node --watch
src/server src/server/index.js“
```

- As you can see, we are using a command called nodemon. We need to install it first:

```
npm install --save nodemon
```

Running Express.js in development

- Furthermore, we must install the @babel/node package, because we are transpiling the back end code with Babel, using the --exec babel-node option.

- It allows the use of the import statement:

```
npm install --save-dev @babel/node
```

Running Express.js in development

- Start the server now:

```
npm run server
```

- When you now go to your browser and enter http://localhost:8000, you will see the text Hello World! from our Express.js callback function.

Routing in Express.js

- Understanding routing is essential to extend our back end code.
- We are going to play through some simple routing examples.
- In general, routing stands for how an application responds to specific endpoints and methods.
- In Express.js, one path can respond to different HTTP methods and can have multiple handler functions.

TriveraTech TECHNOLOGY TRAINING

Routing in Express.js

- Here is a simple example. Replace this with the current app.get line:

```
app.get('/', function (req, res, next) {
  console.log('first function');
  next();
}, function (req, res) {
  console.log('second function');
  res.send('Hello World!');
});
```

TriveraTech TECHNOLOGY TRAINING

Serving our production build

- Again, replace the previous routing example with the following:

```
import path from 'path';

const root = path.join(__dirname, '../../');

app.use('/', express.static(path.join(root, 'dist/client')));
app.use('/uploads', express.static(path.join(root, 'uploads')));
app.get('/', (req, res) => {
  res.sendFile(path.join(root, '/dist/client/index.html'));
});
```

Using Express.js middleware

- Express.js provides great ways to write efficient back ends without duplicating code.

- Every middleware function receives a request, a response, and next. It needs to run next to pass control further to the next handler function. Otherwise, you will receive a timeout.

Using Express.js middleware

- The root path '/' is used to catch any request.

```
app.get('/', function (req, res, next) {
```

- We randomly generate a number with Math.random between 1 and 10.

```
var random = Math.random() * (10 -1) + 1;
```

Using Express.js middleware

- If the number is higher than 5, we run the next('route') function to skip to the next app.get with the same path.

```
if (random > 5) next('route')
```

Using Express.js middleware

- If the number is lower than 0.5, we execute the next function without any parameters and go to the next handler function. This handler will log us 'first'.

```
else next()
}, function (req, res, next) {
  res.send('first');
})
app.get('/', function (req, res, next) {
  res.send('second');
})
```

Installing important middleware

- For our application, we have already used one built-in Express.js middleware: express.static.

- Throughout this course, we continue to install further middleware:

```
npm install --save compression cors helmet
```

Installing important middleware

- Now, execute the import statement on the new packages inside the server index.js file so that all dependencies are available within the file:

```
import helmet from 'helmet';
import cors from 'cors';
import compress from 'compression';
```

Express Helmet

- We can enable the Express.js Helmet middleware as follows in the server index.js file:

```
app.use(helmet());
app.use(helmet.contentSecurityPolicy({
  directives: {
    defaultSrc: ["'self'"],
    scriptSrc: ["'self'", "'unsafe-inline'"],
    styleSrc: ["'self'", "'unsafe-inline'"],
    imgSrc: ["'self'", "data:", "*.amazonaws.com"]
  }
}));
app.use(helmet.referrerPolicy({ policy: 'same-origin' }));
```

TriveraTech TECHNOLOGY TRAINING

Compression with Express.js

- Enabling compression for Express.js saves you and your user bandwidth, and this is pretty easy to do.
- The following code must also be added to the server index.js file:

```
app.use(compress());
```

CORS in Express.js

- We want our GraphQL API to be accessible from any website, app, or system.
- A good idea might be to build an app or offer the API to other companies or developers so that they can use it.
- When using APIs via Ajax, the main problem is that the API needs to send the correct Access-Control-Allow-Origin header.

CORS in Express.js

- Allow CORS (Cross-origin resource sharing) requests with the following command to the index.js file:

```
app.use(cors());
```

Combining Express.js with Apollo

- First things first; we need to install the Apollo and GraphQL dependencies:

```
npm install --save apollo-server-express graphql graphql-tools
```

TriveraTech TECHNOLOGY TRAINING

Combining Express.js with Apollo

- Create a separate folder for services. A service can be GraphQL or other routes:

```
mkdir src/server/services/
mkdir src/server/services/graphql
```

Combining Express.js with Apollo

- We require the apollo-server-express and graphql-tools packages.

```
import { ApolloServer } from 'apollo-server-express';
import { makeExecutableSchema } from 'graphql-tools';
```

Combining Express.js with Apollo

- The GraphQL schema is the representation of the API, that is, the data and functions a client can request or run.
- Resolver functions are the implementation of the schema. Both need to match 100 percent.
- You cannot return a field or run a mutation that is not inside the schema.

```
import Resolvers from './resolvers';
import Schema from './schema';
```

Combining Express.js with Apollo

- The makeExecutableSchema function throws an error when you define a query or mutation that is not in the schema.
- The resulting schema is executable by our GraphQL server resolving the data or running the mutations we request.

```
const executableSchema = makeExecutableSchema({
typeDefs: Schema,
resolvers: Resolvers
});
```

TriveraTech TECHNOLOGY TRAINING

Combining Express.js with Apollo

- In our resolver functions, we can access the request if we need to.

```
const server = new ApolloServer({
schema: executableSchema,
context: ({ req }) => req
});
```

- This index.js file exports the initialized server object, which handles all GraphQL requests.

```
export default server;
```

Combining Express.js with Apollo

- Create an index.js file in the services folder and enter the following code:

```
import graphql from './graphql';

export default {
graphql,
};
```

Combining Express.js with Apollo

- To make our GraphQL server publicly accessible to our clients, we are going to bind the Apollo Server to the /graphql path.

Import the services index.js file in the server/index.js file as follows:

```
import services from './services';
```

Combining Express.js with Apollo

- The services object only holds the graphql index.
- Now we must bind the GraphQL server to the Express.js web server with the following code:

```
const serviceNames = Object.keys(services);

for (let i = 0; i < serviceNames.length; i += 1) {
const name = serviceNames[i];
if (name === 'graphql') {
services[name].applyMiddleware({ app });
} else {
app.use(`/${name}`, services[name]);
}
}
```

TriveraTech TECHNOLOGY TRAINING

Writing your first GraphQL schema

- Let's start by creating a schema.js inside the graphql folder.
- You can also stitch multiple smaller schemas to one bigger schema.
- This would be cleaner and would make sense when your application, types, and fields grow.
- For this course, one file is okay and we insert the following code into the schema.js file:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/2_1.txt

Implementing GraphQL resolvers

- Create a resolvers.js file in the graphql folder as follows:

```
const resolvers = {
RootQuery: {
posts(root, args, context) {
return [];
},
},
};

export default resolvers;
```

Sending GraphQL queries

- You can test our new function when you send the following JSON as a POST request to http://localhost:8000/graphql:

```
{
"operationName": null,
"query": "{
posts {
id
text
}
}",
"variables": {}
}
```

Sending GraphQL queries

- You can insert the content of the query property and hit the play button.
- Because we set up Helmet to secure our application, we need to deactivate it in development.
- Otherwise, the GraphQLi instance is not going to work.
- Just wrap the Helmet initialization inside this if statement:

```
if(process.env.NODE_ENV === 'development')
```

Sending GraphQL queries

- The resulting answer of POST should look like the following code snippet:

```
{
"data": {
"posts": []
}
}
```

Sending GraphQL queries

- Replace the content of the posts function in the GraphQL resolvers with this:

```
return posts;
```

Using multiples types in GraphQL schemas

- Let's create a User type and use it with our posts. First, add it somewhere to the schema:

```
type User {
avatar: String
username: String
}
```

- Now that we have a User type, we need to use it inside the Post type. Add it to the Post type as follows:

```
user: User
```

TriveraTech TECHNOLOGY TRAINING

- The user field allows us to have a sub-object inside our posts with the post's author information.
- Our extended query to test this looks like the following:

```
"query":"{
posts {
id
text
user {
avatar
username
}
}
}"
```

Writing your first GraphQL mutation

- One thing our client already offered was to add new posts to the fake data temporarily.
- We can realize this in the back end by using GraphQL mutations.
- Starting with the schema, we need to add the mutation as well as the input types as follows:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/2_2.txt

Writing your first GraphQL mutation

- The last step is to enable the mutations in our schema for the Apollo Server:

```
schema {
query: RootQuery
mutation: RootMutation
}
```

- The addPost resolver function needs to be implemented now in the resolvers.js file.
- Add the following RootMutation object to the RootQuery in resolvers.js:

```
RootMutation: {
addPost(root, { post, user }, context) {
const postObject = {
...post,
user,
id: posts.length + 1,
};
posts.push(postObject);
return postObject;
},
},
```

Writing your first GraphQL mutation

- You can run this mutation via your preferred HTTP client like this:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/2_3.txt

Back end debugging and logging

- There are two things that are very important here: the first is that we need to implement logging for our back end in case we receive errors from our users

- The second is that we need to look into Postman to debug our GraphQL API efficiently.

Logging in Node.js

The most popular logging package for Node.js is called winston. Configure winston by following the steps below:

- Install winston with npm:

```
npm install --save winston
```

- We create a new folder for all of the helper functions of the back end:

```
mkdir src/server/helpers
```

Logging in Node.js

- Then, insert a logger.js file in the new folder with the following content:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/2_4.txt

Logging in Node.js

- To test this, we can try the winston logger in the only mutation we have.
- In resolvers.js, add this to the top of the file:

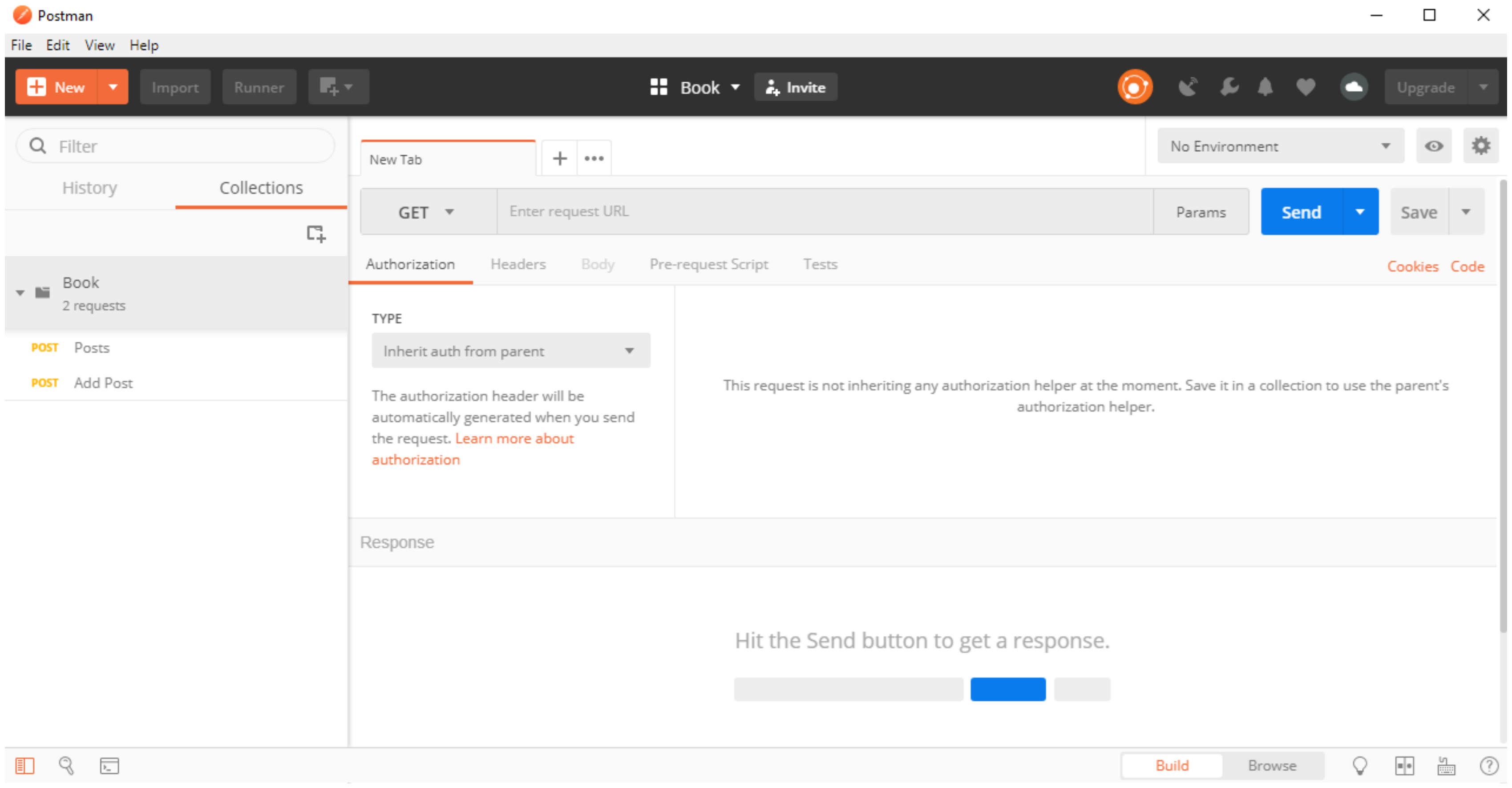
```
import logger from '../../helpers/logger’;
```

- Now, we can extend the addPost function by logging the following:

```
logger.log({ level: 'info', message: 'Post was created' });
```

Debugging with Postman

- When you have finished the installation, it should look something like this:

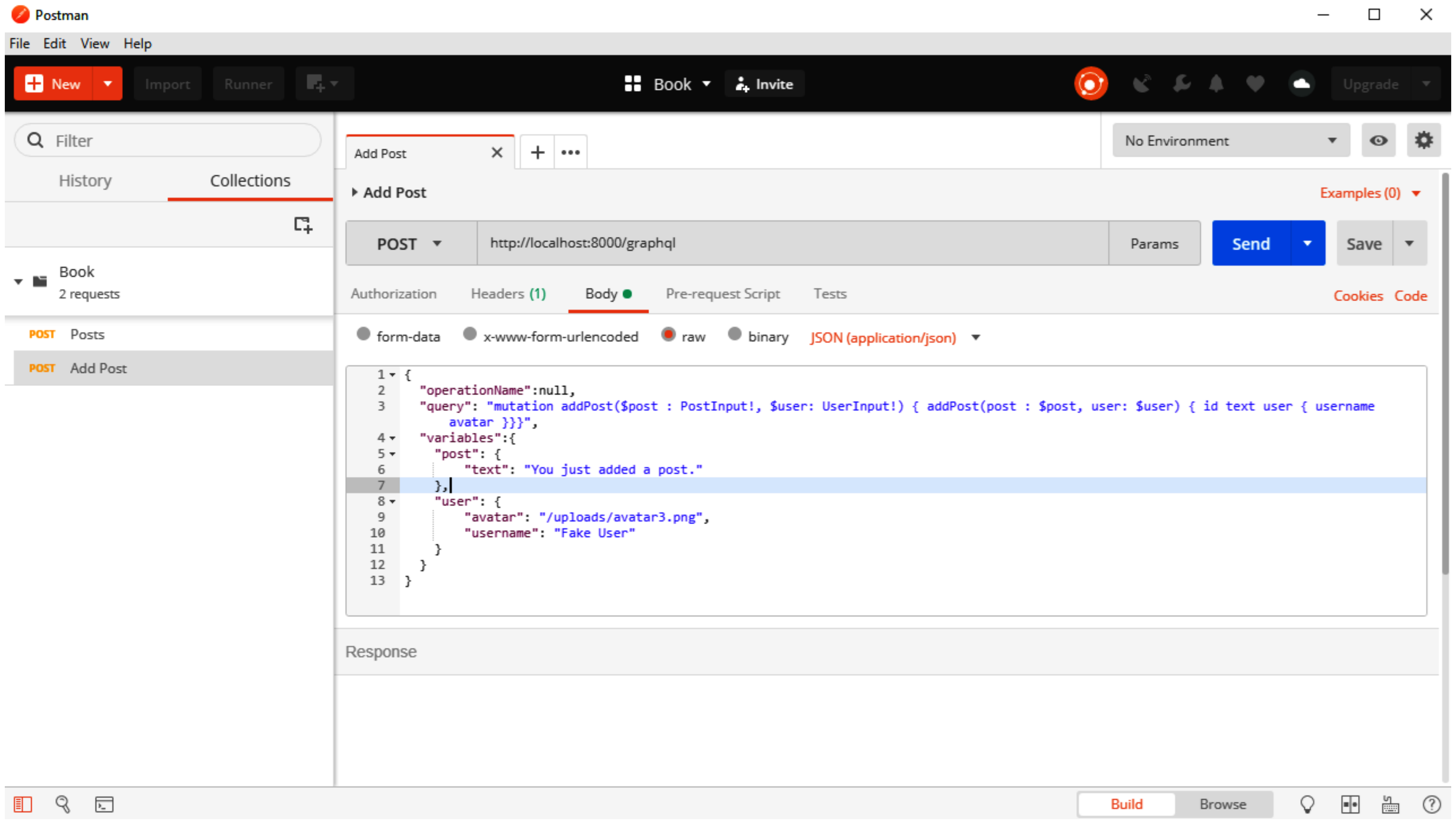


The authorization header will be automatically generated when you send the request. Learn more about authorization

This request is not inheriting any authorization helper at the moment. Save it in a collection to use the parent's authorization helper.

Hit the Send button to get a response.

- As an example, the following screenshot shows you how the Add Post mutation looks in Postman:



POST http://localhost:8000/graphql

```
{
  "operationName":null,
  "query": "mutation addPost($post : PostInput!, $user: UserInput!) { addPost(post : $post, user: $user) { id text user { username
      avatar }}}",
  "variables":{
    "post": {
        "text": "You just added a post."
    },
    "user": {
        "avatar": "/uploads/avatar3.png",
        "username": "Fake User"
    }
  }
}
```

Summary

- At this point, we have set up our Node.js server with Express.js and bound Apollo Server to respond to requests on a GraphQL endpoint.
- We are able to handle queries, return fake data, and mutate that data with GraphQL mutations.
- Furthermore, we can log every process in our Node.js server.

"Complete Lab"

5: Connecting to The Database

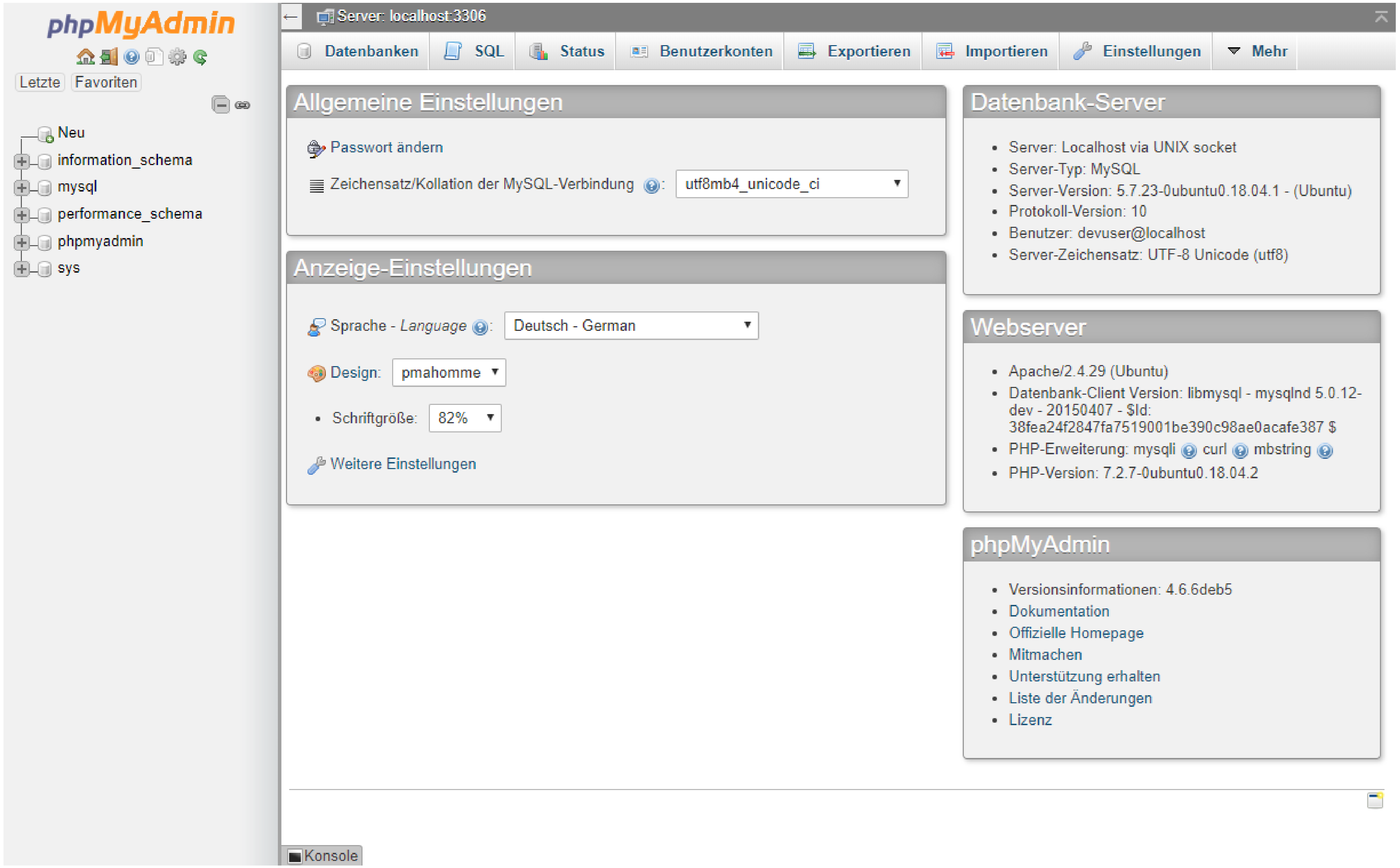
Connecting to The Database

This lesson will cover the following points:

- Using databases with GraphQL
- Using Sequelize in Node.js
- Writing database models
- Performing database migrations with Sequelize
- Seeding data with Sequelize
- Using Apollo together with Sequelize

Using databases in GraphQL

- GraphQL is a protocol for sending and receiving data. Apollo is one of the many libraries that you can use to implement that protocol.
- Neither GraphQL (in its specifications) nor Apollo work directly on the data layer.
- Where the data that you put into your response comes from, and where the data that you send with your request is saved, are up to the user to decide.



Allgemeine Einstellungen

Passwort ändern

Zeichensatz/Kollation der MySQL-Verbindung: utf8mb4_unicode_ci

Anzeige-Einstellungen

Sprache - *Language*: Deutsch - German

Design: pmahomme

- Schriftgröße: 82%

Weitere Einstellungen

Datenbank-Server

- Server: Localhost via UNIX socket
- Server-Typ: MySQL
- Server-Version: 5.7.23-0ubuntu0.18.04.1 - (Ubuntu)
- Protokoll-Version: 10
- Benutzer: devuser@localhost
- Server-Zeichensatz: UTF-8 Unicode (utf8)

Webserver

- Apache/2.4.29 (Ubuntu)
- Datenbank-Client Version: libmysql - mysqlnd 5.0.12-dev - 20150407 - $Id: 38fea24f2847fa7519001be390c98ae0acafe387 $
- PHP-Erweiterung: mysqli curl mbstring
- PHP-Version: 7.2.7-0ubuntu0.18.04.2

phpMyAdmin

- Versionsinformationen: 4.6.6deb5
- Dokumentation
- Offizielle Homepage
- Mitmachen
- Unterstützung erhalten
- Liste der Änderungen
- Lizenz

Creating a database in MySQL

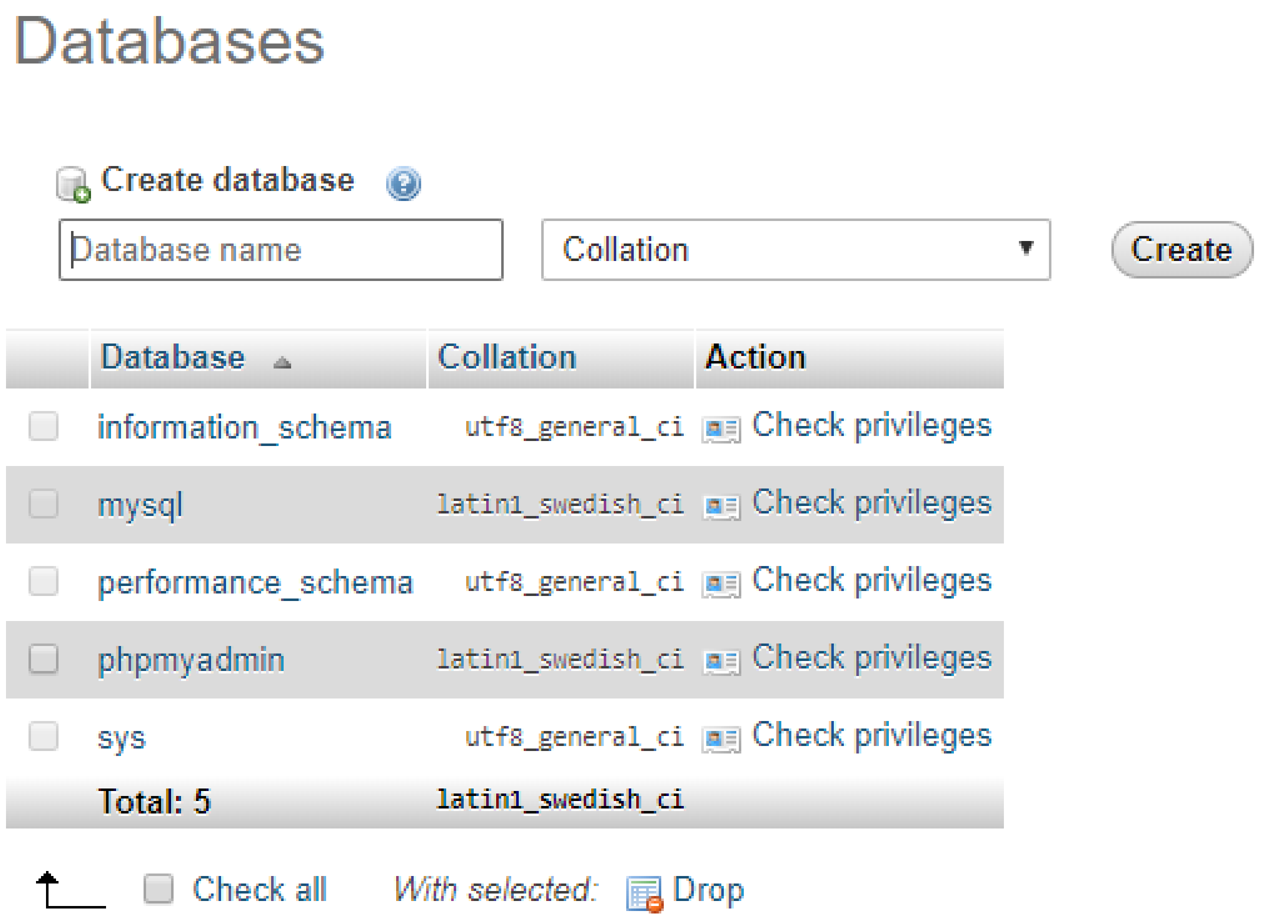
- You can run raw SQL commands in the SQL tab of phpMyAdmin.

- The corresponding command to create a new database looks as follows:

```
CREATE DATABASE graphbook_dev CHARACTER
SET utf8 COLLATE utf8_general_ci;
```

Creating a database in MySQL

- You will be presented with a screen like the following.
- It shows all databases including their collation of your MySQL server:



Databases

	Database	Collation	Action
☐	information_schema	utf8_general_ci	Check privileges
☐	mysql	latin1_swedish_ci	Check privileges
☐	performance_schema	utf8_general_ci	Check privileges
☐	phpmyadmin	latin1_swedish_ci	Check privileges
☐	sys	utf8_general_ci	Check privileges
	Total: 5	latin1_swedish_ci	

Integrating Sequelize into our stack

Install Sequelize in your project via npm.

- We will also install a second package, called mysql2:

```
npm install --save sequelize mysql2
```

- The mysql2 package allows Sequelize to speak with our MySQL server.
- Sequelize is just a wrapper around the various libraries for the different database systems.
- It offers great features for intuitive model usage, as well as functions for creating and updating database structures and inserting development data.

Connecting to a database with Sequelize

- The first step is to initialize the connection from Sequelize to our MySQL server.

- To do this, we will create a new folder and file, as follows:

```
mkdir src/server/database
touch src/server/database/index.js
```

Connecting to a database with Sequelize

- Inside of the index.js database, we will establish a connection to our database with Sequelize.

- Internally, Sequelize relies on the mysql2 package, but we do not use it on our own, which is very convenient:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/3_1.txt

Writing database models

- After creating a connection to our MySQL server via Sequelize, we want to use it.
- However, our database is missing a table or structure that we can query or manipulate.
- Creating those is the next thing that we need to do.
- Currently, we have two GraphQL entities: User and Post.
- Sequelize lets us create a database schema for each of our GraphQL entities.

Writing database models

- Let's create the first model for our posts.
- Create two new folders (one called models, and the other, migrations) next to the database folder:

```
mkdir src/server/models
mkdir src/server/migrations
```

Your first database model

- We will use the Sequelize CLI to generate our first database model. Install it globally with the following command:

```
npm install -g sequelize-cli
```

- This gives you the ability to run the sequelize command inside of your Terminal.
- The Sequelize CLI allows us to generate the model automatically. This can be done by running the following command:

```
sequelize model:generate --models-path src/server/models --migrations-path src/server/migrations --name Post --attributes text:text
```

- The following model file was created for us:

```
'use strict';

module.exports = (sequelize, DataTypes) => {
var Post = sequelize.define('Post', {
text: DataTypes.TEXT
}, {});

Post.associate = function(models) {
// associations can be defined here
};

return Post;
};
```

TriveraTech TECHNOLOGY TRAINING

Your first database migration

A migration file has multiple advantages, such as the following:

1. Migrations allow us to track database changes through our regular version control system, such as Git or SVN. Every change to our database structure should be covered in a migration file.
2. It also enables us to write updates that automatically apply database changes for new versions of our application.

Your first database migration

- Our first migration file creates a Posts table and adds all required columns, as follows:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/3_4.txt

Your first database migration

- To execute this migration, we use the Sequelize CLI again, as follows:

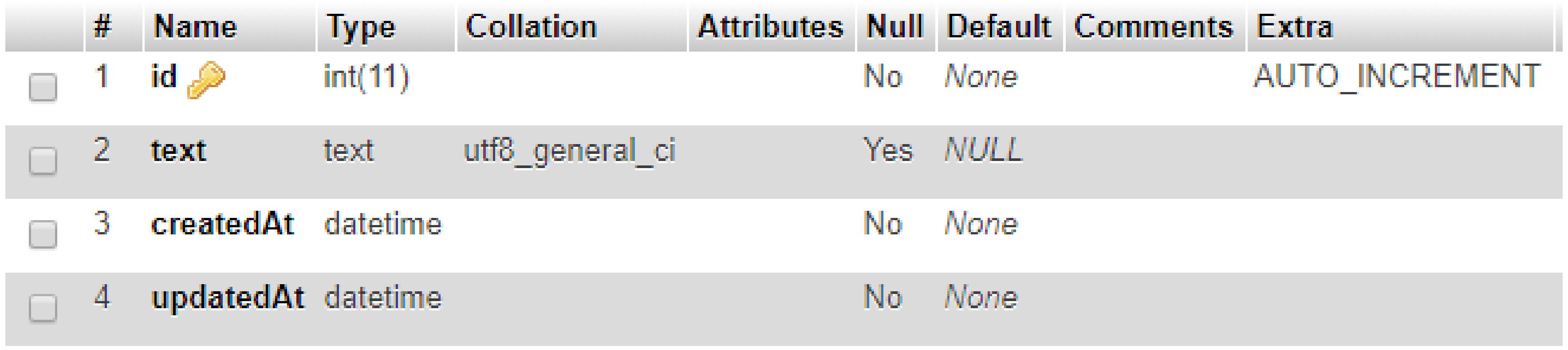
```
sequelize db:migrate --migrations-path
src/server/migrations --config src/server/config/index.js
```

TriveraTech TECHNOLOGY TRAINING

Your first database migration

- Look inside of phpMyAdmin, Here, you will find the new table, called Posts.

- The structure of the table should look as follows:

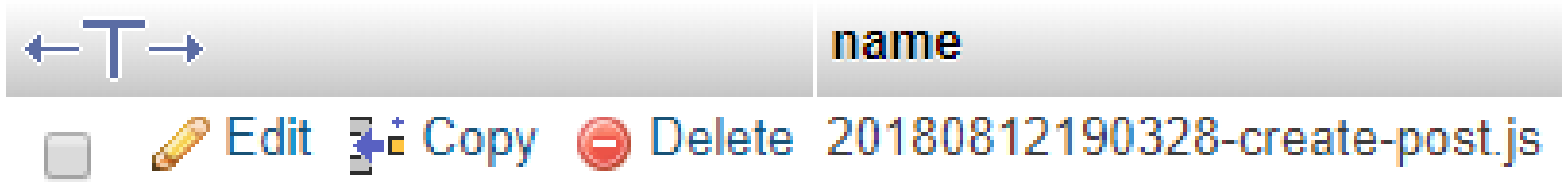


	#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra
☐	1	**id** 🔑	int(11)			No	*None*		AUTO_INCREMENT
☐	2	**text**	text	utf8_general_ci		Yes	*NULL*		
☐	3	**createdAt**	datetime			No	*None*		
☐	4	**updatedAt**	datetime			No	*None*		

Your first database migration

- Every time that you use Sequelize and its migration feature, you will have an additional table, called SequelizeMeta.

- The contents of the table should look as follows:



←T→	name
☐ Edit Copy Delete	20180812190328-create-post.js

Importing models with Sequelize

- We want to import all of our database models at once, in a central file.
- Our database connection instantiator will then use this file on the other side.
- Create an index.js file in the models folder, and fill in the following code:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/3_5.txt

Importing models with Sequelize

- In development, we must execute the babel-plugin-require-context-hook/register hook to load the require.context function at the top.
- This package must be installed with npm, with the following command:

```
npm install --save-dev babel-plugin-require-context-hook
```

Importing models with Sequelize

- We need to load the plugin with the start of our development server, so, open the package.json file and edit the server script, as follows:

```
nodemon --exec babel-node --plugins require-context-
hook --watch src/server src/server/index.js
```

Importing models with Sequelize

- Now, we want to use our models.

- Go back to the index.js database file and import all models through the aggregation index.js file that we just created:

```
import models from '../models';
```

Importing models with Sequelize

- Before exporting the db object at the end of the file, we need to run the models wrapper to read all model .js files.
- We pass our Sequelize instance as a parameter, as follows:

```
const db = {
models: models(sequelize),
sequelize,
};
```

TriveraTech TECHNOLOGY TRAINING

Importing models with Sequelize

- We create the global database instance in the index.js file of the root server folder.

- Add the following code:

```
import db from './database';
```

Seeding data with Sequelize

- Create a new folder, called seeders:

```
mkdir src/server/seeders
```

- Now, we can run our next Sequelize CLI command, in order to generate a boilerplate file:

```
sequelize seed:generate --name fake-posts --seeders-path src/server/seeders
```

Seeding data with Sequelize

- Seeders are great for importing test data into a database for development.

- Our seed file has the timestamp and the words fake-posts in the name, and should look as follows:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/3_6.txt

Seeding data with Sequelize

- It is just an empty boilerplate file.
- We need to edit this file to create the fake posts that we already had in our backend.
- This file looks like our migration from the previous section.
- Replace the contents of the file with the following code:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/3_7.txt

Seeding data with Sequelize

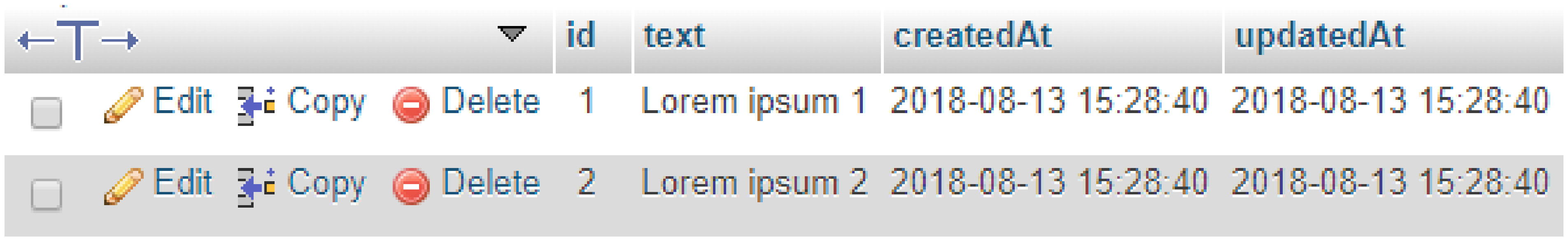
- The down migration bulk deletes all rows in the table, since this is the apparent reverse action of the up migration.

- Execute all of the seeds from the seeders folder with the following command:

```
sequelize db:seed:all --seeders-path src/server/seeders
--config src/server/config/index.js
```

Seeding data with Sequelize

- The following screenshot shows a filled Posts table:



	id	text	createdAt	updatedAt
Edit Copy Delete	1	Lorem ipsum 1	2018-08-13 15:28:40	2018-08-13 15:28:40
Edit Copy Delete	2	Lorem ipsum 2	2018-08-13 15:28:40	2018-08-13 15:28:40

Using Sequelize with Apollo

- The database object is initialized upon starting the server within the root index.js file.
- We pass it from this global location down to the spots where we rely on the database, This way, we do not import the database file repeatedly, but have a single instance that handles all database queries for us.
- The services that we want to publicize through the GraphQL API need access to our MySQL database.

Global database instance

- To pass the database down to our GraphQL resolvers, we create a new object in the server index.js file:

```
import db from './database';

const utils = {
db,
};
```

TriveraTech
TECHNOLOGY TRAINING

Global database instance

- Replace the line where we import the services folder, as follows:

```
import servicesLoader from './services';
const services = servicesLoader(utils);
```

Global database instance

- To do this, go to the services index.js file and change the contents of the file, as follows:

```
import graphql from './graphql';

export default utils => ({
graphql: graphql(utils),
});
```

TriveraTech
TECHNOLOGY TRAINING

- Open the index.js file from the graphql folder and replace everything but the require statements at the top with the following code:

```
export default (utils) => {
const executableSchema = makeExecutableSchema({
typeDefs: Schema,
resolvers: Resolvers.call(utils),
});

const server = new ApolloServer({
schema: executableSchema,
context: ({ req }) => req,
});

return server;
};
```

Global database instance

- Surround the resolvers object in this file with a function, and return the resolvers object from inside of the function:

```
export default function resolver() {
...
return resolvers;
}
```

Running the first database query

- Now, we want to finally use the database.
- Add the following code to the top of the export default function resolver statement:

```
const { db } = this;
const { Post } = db.models;
```

Running the first database query

- We can query all posts through the Sequelize model, instead of returning the fake posts from before.
- Replace the posts property within the RootQuery with the following code:

```
posts(root, args, context) {
return Post.findAll({order: [['createdAt', 'DESC']]});
},
```

TriveraTech TECHNOLOGY TRAINING

Global database instance

- You can start the server with npm run server and execute the GraphQL posts query from lesson 2, Setting Up GraphQL with Express.js, again.

- The output will look as follows:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/3_8.txt

One-to-one relationships in Sequelize

- We need to associate each post with a user, to fill the gap that we have created in our GraphQL response.
- A post has to have an author. It would not make sense to have a post without an associated user.
- First, we will generate a User model and migration. We will use the Sequelize CLI again, as follows:

```
sequelize model:generate --models-path src/server/models --
migrations-path src/server/migrations --name User --
attributes avatar:string,username:string
```

Updating the table structure with migrations

- We have to write a third migration, adding the userId column to our Post table, but also including it in our database Post model.

- Generating a boilerplate migration file is very easy with the Sequelize CLI:

```
sequelize migration:create --migrations-path
src/server/migrations --name add-userId-to-post
```

TriveraTech TECHNOLOGY TRAINING

Updating the table structure with migrations

- You can directly replace the content, as follows:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/3_9.txt

Updating the table structure with migrations

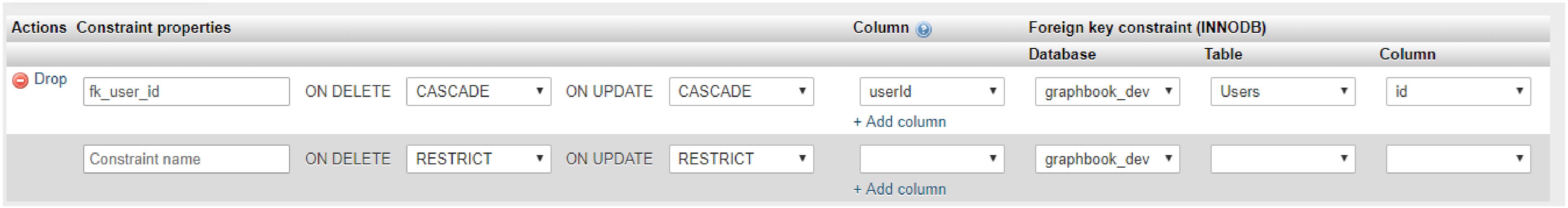
- Rerun the migration, in order to see what changes occurred:

```
sequelize db:migrate --migrations-path
src/server/migrations --config src/server/config/index.js
```

Updating the table structure with migrations

- Take a look at the relation view of the Posts table in phpMyAdmin.

- You can find it under the Structure view, by clicking on Relation view:



Actions	Constraint properties					Column	Foreign key constraint (INNODB)		
							Database	Table	Column
Drop	fk_user_id	ON DELETE	CASCADE	ON UPDATE	CASCADE	userId + Add column	graphbook_dev	Users	id
	Constraint name	ON DELETE	RESTRICT	ON UPDATE	RESTRICT	 + Add column	graphbook_dev		

Updating the table structure with migrations

- If you receive an error when running migrations, you can easily undo them, as follows:

```
sequelize db:migrate:undo --migrations-path
src/server/migrations --config src/server/config/index.js
```

Updating the table structure with migrations

- You can also revert all migrations at once, or only revert to one specific migration, so that you can go back to a specific timestamp:

```
sequelize db:migrate:undo:all --to XXXXXXXXXXXXXX-create-posts.js --migrations-path src/server/migrations --config src/server/config/index.js
```

TriveraTech TECHNOLOGY TRAINING

Model associations in Sequelize

- Now that we have the relationship configured with the foreign key, it also needs to be configured inside of our Sequelize model.
- Go back to the Post model file and replace the associate function with the following:

```
Post.associate = function(models) {
Post.belongsTo(models.User);
};
```

Model associations in Sequelize

- Do not forget to add the userId as a queryable field to the Post model itself, as follows:

```
userId:  DataTypes.INTEGER,
```

- The User model needs to implement the reverse association, too. Add the following code to the User model file:

```
User.associate = function(models) {
User.hasMany(models.Post);
};
```

Model associations in Sequelize

- We must extend our current resolvers.js file.
- Add the Post property to the resolvers object, as follows:

```
Post: {
user(post, args, context) {
return post.getUser();
},
},
```

Seeding foreign key data

- We use the Sequelize CLI to generate an empty seeders file, as follows:

```
sequelize seed:generate --name fake-users --seeders-path src/server/seeders
```

Seeding foreign key data

- Fill in the following code to insert the fake users:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/3_10.txt

- We must maintain the relationships as configured in our database. Adjust the posts seed file to reflect this, and add a userId to both posts in the up migration:

```
return queryInterface.bulkInsert('Posts', [{
text: 'Lorem ipsum 1',
userId: usersRows[0].id,
createdAt: new Date(),
updatedAt: new Date(),
},
{
text: 'Lorem ipsum 2',
userId: usersRows[1].id,
createdAt: new Date(),
updatedAt: new Date(),
}],
{});
```

TriveraTech TECHNOLOGY TRAINING

Seeding foreign key data

- There are two options here. You can either manually truncate the tables through phpMyAdmin and SQL statements, or you can use the Sequelize CLI.
- It is easier to use the CLI, but the result will be the same either way. The following command will undo all seeds:

```
sequelize db:seed:undo:all --seeders-path
src/server/seeders --config src/server/config/index.js
```

Seeding foreign key data

- You can fix this by selecting all users with a raw query in the post seed file.
- We can pass the retrieved user IDs statically.
- Replace the up property with the following:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/3_11.txt

Seeding foreign key data

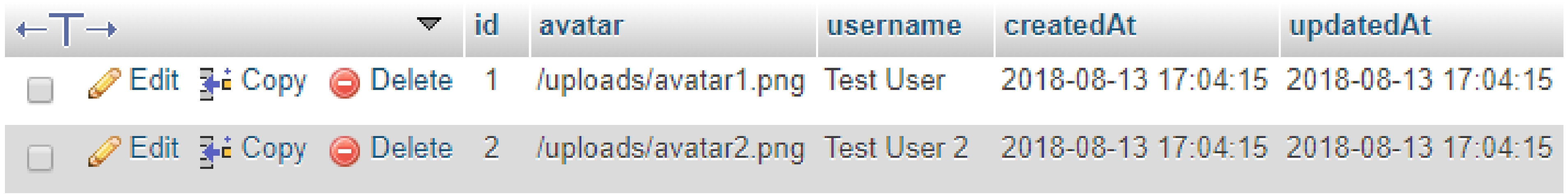
- We have not gotten any further now, since the posts are still inserted before the users.
- From my point of view, the easiest way to fix this is to rename the seeder files.
- Simply adjust the timestamp of the fake user seed file to be before the post seed file's timestamp, or the other way around. Again, execute all seeds, as follows:

```
sequelize db:seed:all --seeders-path src/server/seeders --config src/server/config/index.js
```

TriveraTech
TECHNOLOGY TRAINING

Seeding foreign key data

- If you take a look inside your database, you should see a filled Posts table, including the userId.
- The Users table should look like the following screenshot:



	id	avatar	username	createdAt	updatedAt
Edit Copy Delete	1	/uploads/avatar1.png	Test User	2018-08-13 17:04:15	2018-08-13 17:04:15
Edit Copy Delete	2	/uploads/avatar2.png	Test User 2	2018-08-13 17:04:15	2018-08-13 17:04:15

Mutating data with Sequelize

- Requesting data from our database via the GraphQL API works.
- Now comes the tough part: adding a new post to the Posts table.
- Before we start, we must extract the new database model from the db object at the top of the exported function in our resolvers.js file:

```
const { Post, User } = db.models;
```

TriveraTech TECHNOLOGY TRAINING

Mutating data with Sequelize

- We have to edit the GraphQL resolvers to add the new post.

- Replace the old addPost function with the new one, as shown in the following code snippet:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/3_12.txt

Mutating data with Sequelize

- Everything is set now, To test our API, we are going to use Postman again, We need to change the addPost request.
- The userInput that we added before is not needed anymore, because the backend statically chooses the first user out of our database.
- You can send the following request body:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/3_13.txt

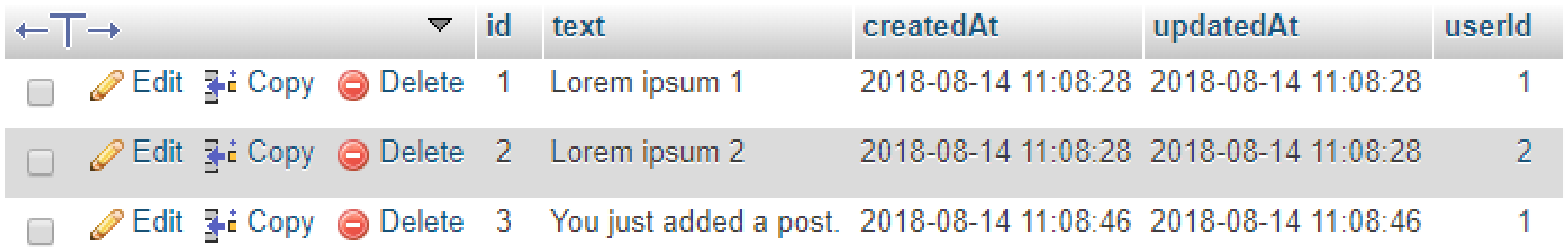
Mutating data with Sequelize

- Your GraphQL schema must reflect this change, so remove the userInput from there, too:

```
addPost (
post: PostInput!
): Post
```

Mutating data with Sequelize

- Running the addPost GraphQL mutation now adds a post to the Posts table, as you can see in the following screenshot:



←T→				id	text	createdAt	updatedAt	userId
☐	Edit	Copy	Delete	1	Lorem ipsum 1	2018-08-14 11:08:28	2018-08-14 11:08:28	1
☐	Edit	Copy	Delete	2	Lorem ipsum 2	2018-08-14 11:08:28	2018-08-14 11:08:28	2
☐	Edit	Copy	Delete	3	You just added a post.	2018-08-14 11:08:46	2018-08-14 11:08:46	1

TriveraTech TECHNOLOGY TRAINING

Many-to-many relationships

- Facebook provides users with various ways to interact.
- Currently, we only have the opportunity to request and insert posts.
- As in the case of Facebook, we want to have chats with our friends and colleagues.
- We will introduce two new entities to cover this.

Model and migrations

- When transferring this into real code, we first generate the Chat model.

- The problem here is that we have a many-to-many relationship between users and chats.

- In MySQL, this kind of relationship requires a table, to store the relations between all entities separately.

Chat model

- Let's start by creating the Chat model and migration.
- A chat itself does not store any data; we use it for grouping specific users' messages:

```
sequelize model:generate --models-path
src/server/models --migrations-path
src/server/migrations --name Chat --attributes
firstName:string,lastName:string,email:string
```

Chat model

- Generate the migration for our association table, as follows:

```
sequelize migration:create --migrations-path
src/server/migrations --name create-user-chats
```

Chat model

- References inside of a migration automatically create foreign key constraints for us.

- The migration file should look like the following code snippet:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/3_14.txt

Chat model

- Associate the user to the Chat model via the new relation table in the User model, as follows:

```
User.belongsToMany(models.Chat, { through:
'users_chats' });
```

- Do the same for the Chat model, as follows:

```
Chat.belongsToMany(models.User, { through:
'users_chats' });
```

- Rerun the migrations to let the changes take effect:

```
sequelize db:migrate --migrations-path src/server/migrations --config src/server/config/index.js
```

- The following screenshot shows how your database should look now:

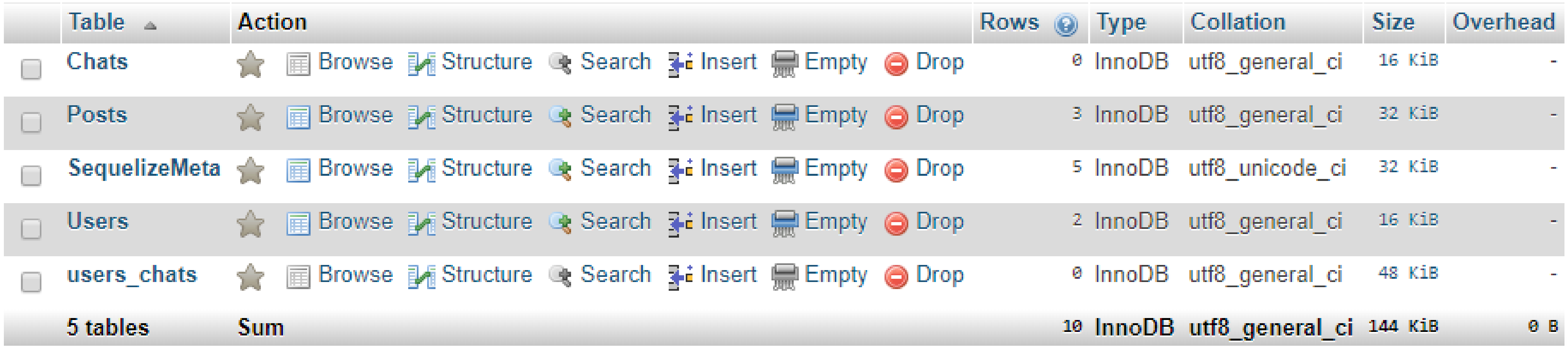
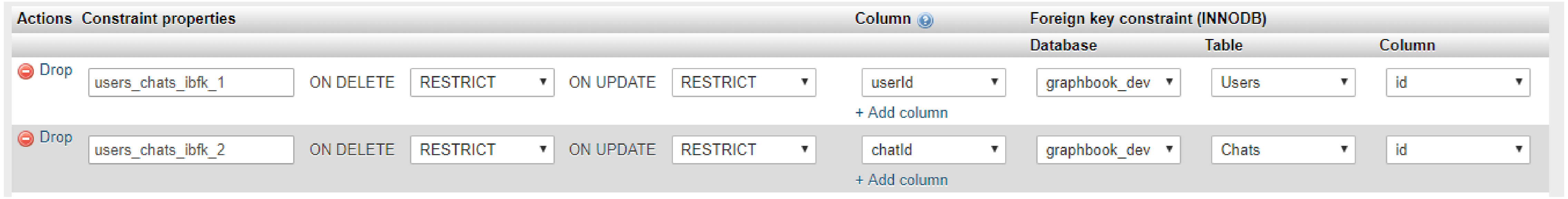


	Table	Action							Rows	Type	Collation	Size	Overhead
☐	Chats	☆	Browse	Structure	Search	Insert	Empty	Drop	0	InnoDB	utf8_general_ci	16 KiB	-
☐	Posts	☆	Browse	Structure	Search	Insert	Empty	Drop	3	InnoDB	utf8_general_ci	32 KiB	-
☐	SequelizeMeta	☆	Browse	Structure	Search	Insert	Empty	Drop	5	InnoDB	utf8_unicode_ci	32 KiB	-
☐	Users	☆	Browse	Structure	Search	Insert	Empty	Drop	2	InnoDB	utf8_general_ci	16 KiB	-
☐	users_chats	☆	Browse	Structure	Search	Insert	Empty	Drop	0	InnoDB	utf8_general_ci	48 KiB	-
	5 tables	Sum							10	InnoDB	utf8_general_ci	144 KiB	0 B

Chat model

- You should see two foreign key constraints in the relation view of the users_chats table.

- The naming is done automatically:



Actions	Constraint properties					Column	Foreign key constraint (INNODB)		
							Database	Table	Column
Drop	users_chats_ibfk_1	ON DELETE	RESTRICT	ON UPDATE	RESTRICT	userId + Add column	graphbook_dev	Users	id
Drop	users_chats_ibfk_2	ON DELETE	RESTRICT	ON UPDATE	RESTRICT	chatId + Add column	graphbook_dev	Chats	id

Message model

- A message is much like a post, except that it is only readable inside of a chat, and is not public to everyone.
- Generate the model and migration file with the CLI, as follows:

```
sequelize model:generate --models-path
src/server/models --migrations-path src/server/migrations
--name Message --attributes
text:string,userId:integer,chatId:integer
```

- Add the missing references in the created migration file, as follows:

```
userId: {
  type: Sequelize.INTEGER,
  references: {
    model: 'Users',
    key: 'id'
  },
  onDelete: 'SET NULL',
  onUpdate: 'cascade',
},
chatId: {
  type: Sequelize.INTEGER,
  references: {
    model: 'Chats',
    key: 'id'
  },
  onDelete: 'cascade',
  onUpdate: 'cascade',
},
```

- Now, we can run the migrations again, in order to create the Messages table using the sequelize db:migrate Terminal command.

```
{
  "operationName":null,
  "query": "mutation addPost($post : PostInput!) { addPost(post :
$post) {
    id text user { username avatar }}}",
  "variables":{
    "post": {
      "text": "You just added a post."
    }
  }
}
```

Message model

- Replace the associate function of the Message model with the following code:

```
Message.associate = function(models) {
  Message.belongsTo(models.User);
  Message.belongsTo(models.Chat);
};
```

TriveraTech TECHNOLOGY TRAINING

Message model

- In the preceding code, we define that every message is related to exactly one user and chat.
- On the other side, we must also associate the Chat model with the messages.
- Add the following code to the associate function of the Chat model:

```
Chat.hasMany(models.Message);
```

Chats and messages in GraphQL

- We have introduced some new entities with messages and chats.
- Let's include those in our Apollo schema.
- In the following code, you can see an excerpt of the changed entities, fields, and parameters of our GraphQL schema:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/3_15.txt

Chats and messages in GraphQL

- These factors should be implemented in our resolvers, too.

- Our resolvers should look as follows:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/3_16.txt

Chats and messages in GraphQL

- It is important that we are using the new models here.
- We should not forget to extract them from the db.models object inside of the resolver function.
- It must look as follows:

```
const { Post, User, Chat, Message } = db.models;
```

Chats and messages in GraphQL

- You can send this GraphQL request to test the changes:

```
{
  "operationName":null,
  "query": "{ chats { id users { id } messages { id text
user { id username
  } } } }",
  "variables":{}
}
```

Chats and messages in GraphQL

- The response should give us an empty chats array, as follows:

```
{
  "data": {
    "chats": []
  }
}
```

Seeding many-to-many data

- Testing our implementation requires data in our database.
- We have three new tables, so we will create three new seeders, in order to get some test data to work with.
- Let's start with the chats, as follows:

```
sequelize seed:generate --name fake-chats --seeders-path src/server/seeders
```

Seeding many-to-many data

- Now, replace the new seeder file with the following code.
- Running the following code creates a chat in our database.
- We do not need more than two timestamps, because the chat ID is generated automatically:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/3_17.txt

Seeding many-to-many data

- Next, we insert the relation between two users and the new chat.
- We do this by creating two entries in the users_chats table where we reference them.
- Now, generate the boilerplate seed file, as follows:

```
sequelize seed:generate --name fake-chats-users-relations --seeders-path src/server/seeders
```

- Our seed should look much like the previous ones, as follows:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/3_18.txt

Seeding many-to-many data

- The last table without any data is the Messages table. Generate the seed file, as follows:

```
sequelize seed:generate --name fake-messages --seeders-path src/server/seeders
```

- Again, replace the generated boilerplate code, as follows:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/3_19.txt

Seeding many-to-many data

- Try to run the GraphQL chats query again, as follows:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/3_20.txt

Seeding many-to-many data

- Add a RootQuery chat that takes a chatId as a parameter:

```
chat(root, { chatId }, context) {
  return Chat.findById(chatId, {
    include: [{
      model: User,
      required: true,
    },
    {
      model: Message,
    }],
  });
},
```

- Add the new query to the GraphQL schema, under RootQuery:

```
chat(chatId: Int): Chat
```

- Send the GraphQL request to test the implementation, as follows:

```
{
  "operationName":null,
  "query": "query($chatId: Int!){ chat(chatId: $chatId) {
    id users { id } messages { id text user { id username } } } }",
  "variables":{ "chatId": 1 }
}
```

Creating a new chat

- Add the addChat function to the RootMutation in the resolvers.js file, as follows:

```
addChat(root, { chat }, context) {
  logger.log({
    level: 'info',
    message: 'Message was created',
  });
  return Chat.create().then((newChat) => {
    return Promise.all([
      newChat.setUsers(chat.users),
    ]).then(() => {
      return newChat;
    });
  });
},
```

Creating a new chat

- We have to add the new input type and mutation, as follows:

```
input ChatInput {
  users: [Int]
}

type RootMutation {
  addPost (
    post: PostInput!
  ): Post
  addChat (
    chat: ChatInput!
  ): Chat
}
```

Creating a new chat

- Test the new GraphQL addChat mutation as your request body:

```
{
  "operationName":null,
  "query": "mutation addChat($chat: ChatInput!) { addChat(chat:
    $chat) { id users { id } }}",
  "variables":{
    "chat": {
      "users": [1, 2]
    }
  }
}
```

Creating a new message

- Add the addMessage function to the RootMutation in the resolvers.js file, as follows:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/3_21.txt

Then, add the new mutation to your GraphQL schema.

- We also have a new input type for our messages:

```
input MessageInput {
  text: String!
  chatId: Int!
}

type RootMutation {
  addPost (
    post: PostInput!
  ): Post
  addChat (
    chat: ChatInput!
  ): Chat
  addMessage (
    message: MessageInput!
  ): Message
}
```

Creating a new message

- You can send the request in the same way as the addPost request:

```
{
   "operationName":null,
   "query": "mutation addMessage($message : MessageInput!) {
     addMessage(message : $message) { id text }}",
   "variables":{
     "message": {
       "text": "You just added a message.",
       "chatId": 1
     }
   }
}
```

Summary

- Our goal in this lesson was to create a working backend with a database as storage, which we have achieved pretty well.
- We can add further entities and migrate and seed them with Sequelize.
- Migrating our database changes won't be a problem for us when it comes to going into production.

"Complete Lab"

6: Integrating React into the Back end with Apollo

Integrating React into the Back end with Apollo

This lesson will cover the following points:

- Installing and configuring Apollo Client
- Sending requests with GQL and Apollo's Query component
- Mutating data with Apollo
- Debugging with Apollo Client Developer Tools

Setting up Apollo Client

- To start, we must install the React Apollo Client library.
- Apollo Client is a GraphQL client that offers excellent integration with React, and the ability to easily fetch data from our GraphQL API.
- Furthermore, it handles actions such as caching and subscriptions, to implement real-time communication with your GraphQL back end.

Installing Apollo Client

- We use npm to install our client dependencies, as follows:

```
npm install --save apollo-client apollo-cache-inmemory
apollo-link-http apollo-link-error apollo-link react-apollo
```

Installing Apollo Client

- To get started with the manual setup of the Apollo Client, create a new folder and file for the client, as follows:

```
mkdir src/client/apollo
touch src/client/apollo/index.js
```

TriveraTech TECHNOLOGY TRAINING

Installing Apollo Client

- Just insert the following code:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/4_1.txt

Testing the Apollo Client

- First, install this package with npm, as follows:

```
npm install --save graphql-tag
```

- Import the package at the top of the Apollo Client setup, as follows:

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

- Then, add the following code before the client is exported:

```
client.query({
query: gql`
{
posts {
id
text
user {
avatar
username
}
}
}`
}).then(result => console.log(result));
```

Testing the Apollo Client

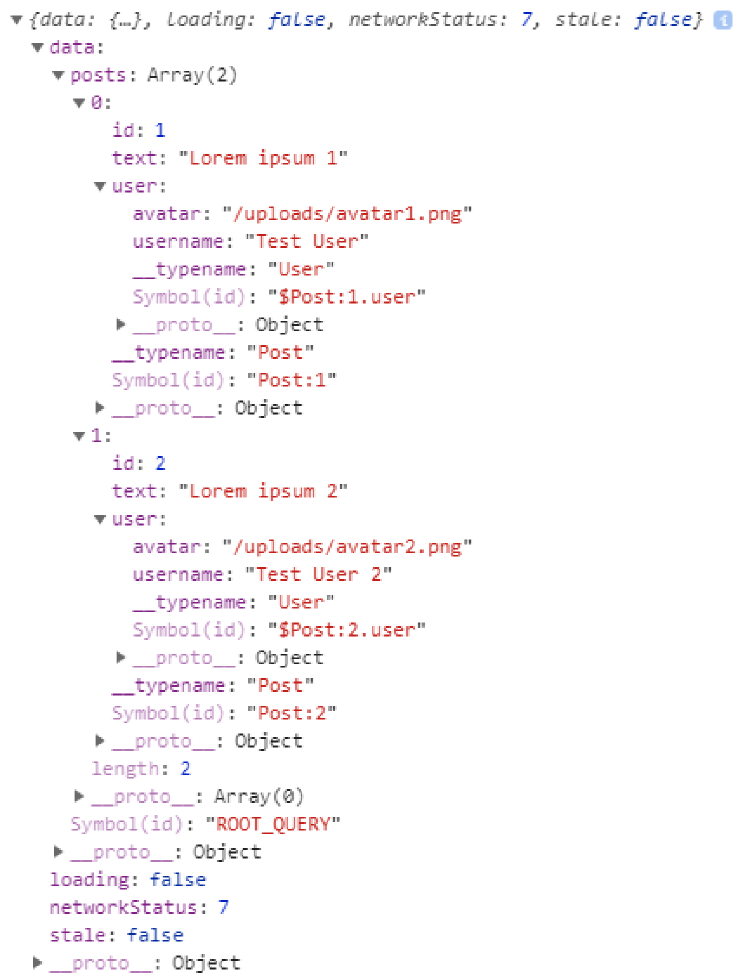
- However, we have forgotten something: the client is set up in our new file, but it is not yet used anywhere.

- Import it in the index.js root file of our client React app, below the import of the App class:

```
import client from './apollo';
```

Testing the Apollo Client

- The output should look like the following screenshot:



```
▼{data: {…}, loading: false, networkStatus: 7, stale: false}
  ▼data:
    ▼posts: Array(2)
      ▼0:
          id: 1
          text: "Lorem ipsum 1"
        ▼user:
            avatar: "/uploads/avatar1.png"
            username: "Test User"
            __typename: "User"
            Symbol(id): "$Post:1.user"
          ▶__proto__: Object
          __typename: "Post"
          Symbol(id): "Post:1"
        ▶__proto__: Object
      ▼1:
          id: 2
          text: "Lorem ipsum 2"
        ▼user:
            avatar: "/uploads/avatar2.png"
            username: "Test User 2"
            __typename: "User"
            Symbol(id): "$Post:2.user"
          ▶__proto__: Object
          __typename: "Post"
          Symbol(id): "Post:2"
        ▶__proto__: Object
        length: 2
      ▶__proto__: Array(0)
      Symbol(id): "ROOT_QUERY"
    ▶__proto__: Object
    loading: false
    networkStatus: 7
    stale: false
  ▶__proto__: Object
```

Binding the Apollo Client to React

- We have tested Apollo Client, and have confirmed that it works.
- However, React does not yet have access to it.
- Since Apollo Client is going to be used everywhere in our application, we can set it up in our root index.js file, as follows:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/4_2.txt

Using the Apollo Client in React

Follow the instructions below to connect your first React component with the Apollo Client:

- Clone the App.js file to another file, called Feed.js.
- Remove all parts where React Helmet is used, and rename the class Feed, instead of App.
- From the App.js file, remove all of the parts that we have left in the Feed class.

Using the Apollo Client in React

- Furthermore, we must render the Feed class inside of the App class.

- It should like the following code:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/4_3.txt

Querying in React with the Apollo Client

- There are two main approaches offered by Apollo that can be used to request data.
- The first one is a higher-order component (HoC), provided by the react-apollo package.
- The second one is the Query component of Apollo, which is a special React component.
- Both approaches have their advantages and disadvantages.

Apollo HoC query

- A higher-order component is a function that takes a component as input and returns a new component.
- This method is used in many cases wherein we have multiple components all relying on the same functionalities, such as querying for data.

Using the Apollo Client in React

To see a real example of this, we use the posts feed. Follow these instructions to get a working Apollo Query HoC:

- Remove the demo posts from the top of the Feed.js file.
- Remove the posts field from the state initializer.
- Import graphl-tag and parse our query with it, as follows:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/4_4.txt

TriveraTech TECHNOLOGY TRAINING

Using the Apollo Client in React

- Replace everything in the render function, before the final return statement, with the following code:

```
const { posts, loading, error } = this.props;
const { postContent } = this.state;

if(loading) {
return "Loading...";
}
if(error) {
return error.message;
}
```

TriveraTech TECHNOLOGY TRAINING

Using the Apollo Client in React

- Remove the export statement from the Feed class.
- We will export the new component returned from the HoC at the end of the file.
- The export must look as follows:

```
export default graphql(GET_POSTS, {
props: ({ data: { loading, error, posts } }) => ({
loading,
posts,
error
})
})(Feed)
```

TriveraTech TECHNOLOGY TRAINING

The Apollo Query component

- We will now take a look at the second approach, which is also the approach of the official Apollo documentation.
- Before getting started, undo the HoC implementation to send requests from the previous section.
- The new way of fetching data through the Apollo Client is via render props, or render functions

The Apollo Query component

- Remove the demo posts from the top of the Feed.js file.
- Remove the posts from the state and stop extracting them from the component state in the render method, too.
- Import the Query component from the react-apollo package and graphl-tag, as follows:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/4_5.txt

The Apollo Query component

- The Query component can now be rendered.
- The only parameter, for now, is the parsed query that we want to send.
- Replace the complete render method with the following code:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/4_6.txt

Mutations with the Apollo Client

- We have replaced the way that we get the data in our client, The next step is to switch the way that we create new posts, too.
- Before Apollo Client, we had to add the new fake posts to the array of demo posts manually, within the memory of the browser.
- Now, everything in our text area is sent with the addPost mutation to our GraphQL API, through Apollo Client.

TriveraTech TECHNOLOGY TRAINING

The Apollo Mutation HoC

Follow these instructions to set up the mutation HoC:

- Import the compose method from the react-apollo package, as follows:

```
import { graphql, compose } from 'react-apollo';
```

- Add the addPost mutation and parse it with graphql-tag:

```
const ADD_POST = gql`
mutation addPost($post : PostInput!) {
addPost(post : $post) {
id
text
user {
username
avatar
}
}
}
`;
```

The Apollo Mutation HoC

- Now, we will use the compose function of react-apollo, which takes a set of GraphQL queries, or mutations.
- These are run or passed as functions to the component.
- Add the following code to the bottom, and remove the old export statement:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/4_7.txt

The Apollo Mutation HoC

- The addPost function is available under the properties of the Feed component, as we specified it in the preceding code.
- When giving a variables object as a parameter, we can fill in our input fields in the mutation, as is expected by our GraphQL schema:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/4_8.txt

The Apollo Mutation component

- Import the Mutation component from the react-apollo package, as follows:

```
import { Query, Mutation } from 'react-apollo';
```

- Export the Feed component again, as we did in the previous Query component example.
- Remove the ADD_POST_MUTATION and GET_POSTS_QUERY variables when doing so:

```
export default class Feed extends Component
```

The Apollo Mutation component

- Next, add the Mutation component inside of the render function.

- We will surround the form with this component, as this is the only part of our content where we need to send a mutation:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/4_9.txt

Updating the UI with the Apollo Client

We use these solutions in different scenarios.

- Let's take a look at some examples, Refetching makes sense if further logic is implemented on the server, which is hidden from the client when requesting a list of items, and which is not applied when inserting only one item.
- In these cases, the client cannot simulate the state of the typical response of a server.
- Updating the cache, however, makes sense when adding or updating items in a list, like our post feed.
- The client can insert the new post at the top of the feed.

Refetching queries

- As mentioned previously, this is the easiest method to update your user interface.
- The only step is to set an array of queries to be refetched.
- The Mutation component should look as follows:

```
<Mutation
refetchQueries={[{query: GET_POSTS}]}
```

Updating the Apollo cache

- We want to explicitly add only the new post to the cache of the Apollo Client.
- Using the cache helps us to save data, by not refetching the complete feed or rerendering the complete list.
- To update the cache, you should remove the refetchQueries property.
- You can then introduce a new property, called update, as shown in the following code:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/4_10.txt

The Apollo Mutation component

- The advantage is that the user can see the new result, instead of waiting for the response of the server.
- This solution makes the application feel faster and more responsive.
- This section expects the update function of the Mutation component to already be implemented.
- Otherwise, this UI feature will not work. We need to add the optimisticResponse property to our mutation, as follows:

Refer to the file 4_11.txt

- To set a particular class on the list item, we conditionally set the correct className in our map loop.
- Insert the following code into the render method:

```
{posts.map((post, i) =>
  <div key={post.id} className={'post ' + (post.id < 0 ? 'optimistic':
    '')}>
    <div className="header">
      <img src={post.user.avatar} />
      <h2>{post.user.username}</h2>
    </div>
    <p className="content">
      {post.text}
    </p>
  </div>
)}
```

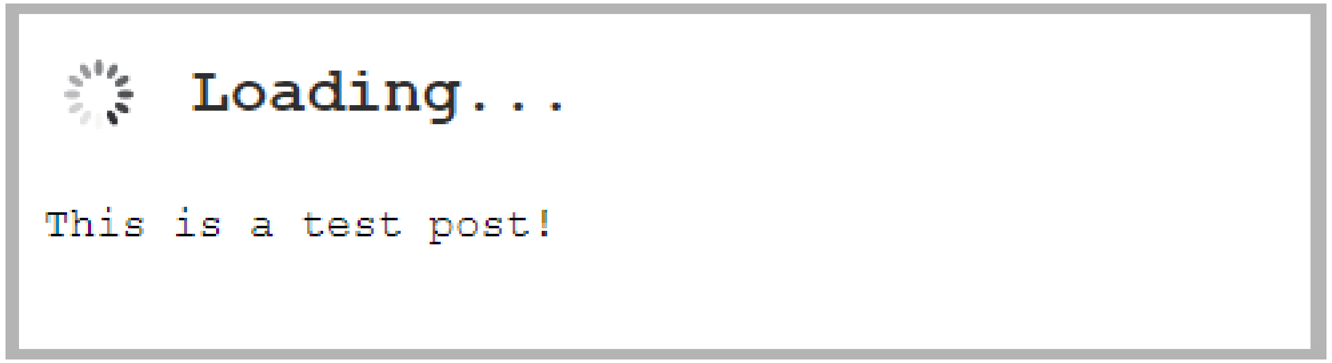
The Apollo Mutation component

- An example CSS style for this might look as follows:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/4_12.txt

The Apollo Mutation component

- CSS animations make your applications more modern and flexible.
- If you experience issues when viewing these in your browser, you may need to check whether your browser supports them.
- You can see the result in the following screenshot:



```
Loading...

This is a test post!
```

Polling with the Query component

Polling is nothing more than rerunning a request after a specified interval. This procedure is the simplest way to implement real-time updates for our news feed. However, multiple issues are associated with polling, as follows:

- It is inefficient to send requests without knowing whether there is any new data. The browser might send dozens of requests without ever receiving a new post.

Polling with the Query component

- There are some use cases in which polling makes sense.
- One example is a real-time graph, in which every axis tick is displayed to the user, whether there is data or not.
- You do not need to use an interrupt-based solution, since you want to show everything.
- Despite the issues that come with polling, let's quickly run through how it works.
- All you need to do is fill in the pollInterval property, as follows:

```
<Query query={GET_POSTS} pollInterval={5000}>
```

Implementing chats and messages

- In the previous lesson, we programmed a pretty dynamic way of creating chats and messages with your friends and colleagues, either one-on-one or in a group.

- There are some things that we have not discussed yet, such as authentication, real-time subscriptions, and friend relationships.

Fetching and displaying chats

- Our news feed is working as we expected. Now, we also want to cover chats.
- As with our feed, we need to query for every chat that the current user (or, in our case, the first user) is associated with.
- The initial step is to get the rendering working with some demo chats.
- Instead of writing the data on our own, as we did in the first lesson, we can now execute the chats query.

Fetching and displaying chats

- Send the GraphQL query. The best options involve Apollo Client Developer Tools, if you already know how they work.
- Otherwise, you can rely on Postman, as you did previously:

```
query {
  chats {
    id
    users {
      avatar
      username
    }
  }
}
```

- Copy the complete response over to an array inside of the Chats.js file, as follows. Add it to the top of the file:

```
const chats = [{
  "id": 1,
  "users": [{
      "id": 1,
      "avatar": "/uploads/avatar1.png",
      "username": "Test User"
    },
    {
      "id": 2,
      "avatar": "/uploads/avatar2.png",
      "username": "Test User 2"
    }]
  }
```

Fetching and displaying chats

- Import React ahead of the chats variable.

- Otherwise, we will not be able to render any React components:

```
import React, { Component } from 'react';
```

Fetching and displaying chats

Set up the React Component. I have provided the basic markup here.

- Just copy it beneath the chats variable.
- I am going to explain the logic of the new component shortly:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/4_13.txt

Fetching and displaying chats

- To save the file size in our CSS file, replace the two post header styles to also cover the style of the chats, as follows:

```
.post .header > *, .chats .chat .header > * {
  display: inline-block;
  vertical-align: middle;
}

.post .header img, .chats .chat .header img {
  width: 50px;
  margin: 5px;
}
```

Fetching and displaying chats

- We must append the following CSS to the bottom of the style.css file:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/4_14.txt

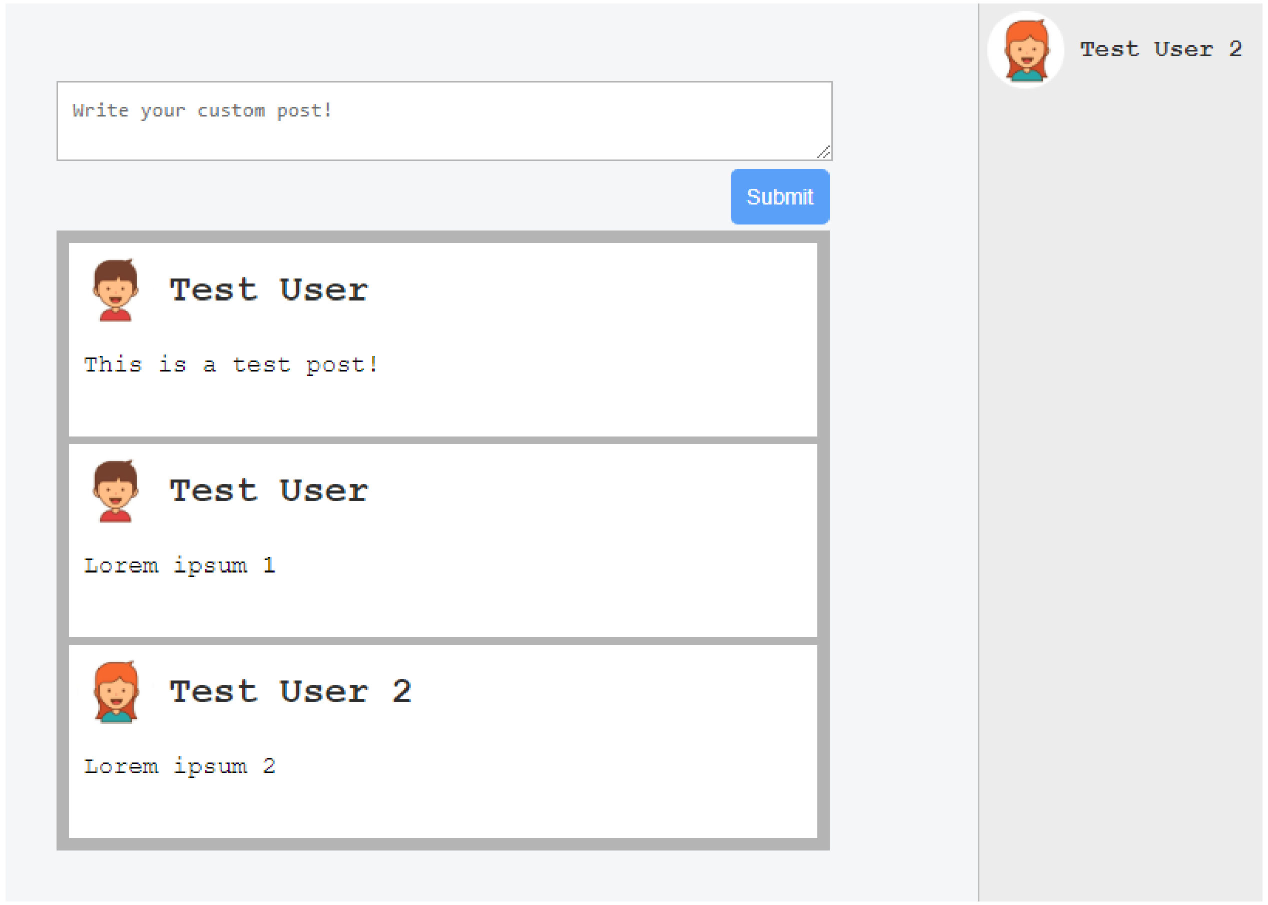
Fetching and displaying chats

- To get the code working, we must also import the Chats class in our App.js file:

```
import Chats from './Chats’;
```

- Render the Chats class inside of the render method, beneath the Feed class inside of the App.js file.

- The current code generates the following screenshot:


Write your custom post!

Test User

This is a test post!

Test User

Lorem ipsum 1

Test User 2

Lorem ipsum 2

Fetching and displaying chats

- Instead, we will add a new property to the chat entity, called lastMessage.
- That way, we will only get the newest message.
- We will add the new field to the GraphQL schema of our chat type, in the back end code, as follows:

```
lastMessage: Message
```

Fetching and displaying chats

- If you return the promise directly, you will receive null in the response from the server, because an array is not a valid response for a single message entity:

```
lastMessage(chat, args, context) {
  return chat.getMessages({limit: 1, order: [['id',
'DESC']]}).then((message) => {
    return message[0];
  });
},
```

Fetching and displaying chats

- You can add the new property to our static data, inside of Chats.js.

- Rerunning the query (as we did in step 1) would also be possible:

```
"lastMessage": {
  "text": "This is a third test message."
}
```

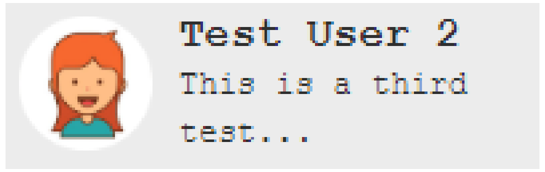
TriveraTech
TECHNOLOGY TRAINING

Fetching and displaying chats

- We can render the new message with a simple span tag beneath the h2 of the username.
- Copy it directly into the render method, inside of our Chats class:

```
<span>{this.shorten(chat.lastMessage.text)}</span>
```

- The result of the preceding changes renders every chat row with the last message inside of the chat.
- This looks like the following screenshot:



Test User 2

This is a third test...

Fetching and displaying chats

- Since everything is displayed correctly from our test data, we can introduce the Query component, in order to fetch all of the data from our GraphQL API, We can remove the chats array.

- Then, we will import all of the dependencies and parse the GraphQL query, as in the following code:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/4_15.txt

Fetching and displaying chats

- Our new render method does not change much.
- We just include the Apollo Query component, as follows:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/4_16.txt

Fetching and displaying messages

- We will start with the Query component from the beginning.
- First, however, we have to store the chats that were opened by a click from the user.
- Every chat is displayed in a separate, small chat window, like in Facebook.
- Add a new state variable to save the ids of all of the opened chats to the Chats class:

```
state = {
  openChats: []
}
```

- To let our component insert something into the array of open chats, we will add the new openChat method to our Chats class:

```
openChat = (id) => {
  var openChats = this.state.openChats.slice();

  if(openChats.indexOf(id) === -1) {
    if(openChats.length > 2) {
      openChats = openChats.slice(1);
    }
    openChats.push(id);
  }

  this.setState({ openChats });
}
```

Fetching and displaying messages

- The last step is to bind the onClick event to our component.

- In the map function, we can replace the wrapping div tag with the following line:

```
<div key={"chat" + chat.id} className="chat"
onClick={() => self.openChat(chat.id)}>
```

TriveraTech TECHNOLOGY TRAINING

Fetching and displaying messages

- Add a surrounding wrapper div tag to the whole render method:

```
<div className="wrapper">
```

- We insert the new markup for the open chats next to the chats panel.
- You cannot insert it inside the panel directly, due to the CSS that we are going to use:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/4_17.txt

Fetching and displaying messages

- As you can see in the preceding code, we are not only passing the chat id as a parameter to the variables property of the Query component, but we also use another query stored in the GET_CHAT variable.
- We must parse this query first, with graphql-tag. Add the following code to the top of the file:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/4_18.txt

Fetching and displaying messages

- Because we rely on the openChats state variable, we must extract it in our render method.

- Add the following code before the return state, in the render method:

```
const self = this;
const { openChats } = this.state;
```

Fetching and displaying messages

- The close button function relies on the closeChat method, which we will implement in our Chats class:

```
closeChat = (id) => {
  var openChats = this.state.openChats.slice();

  const index = openChats.indexOf(id);
  openChats.splice(index,1),

  this.setState({ openChats });
}
```

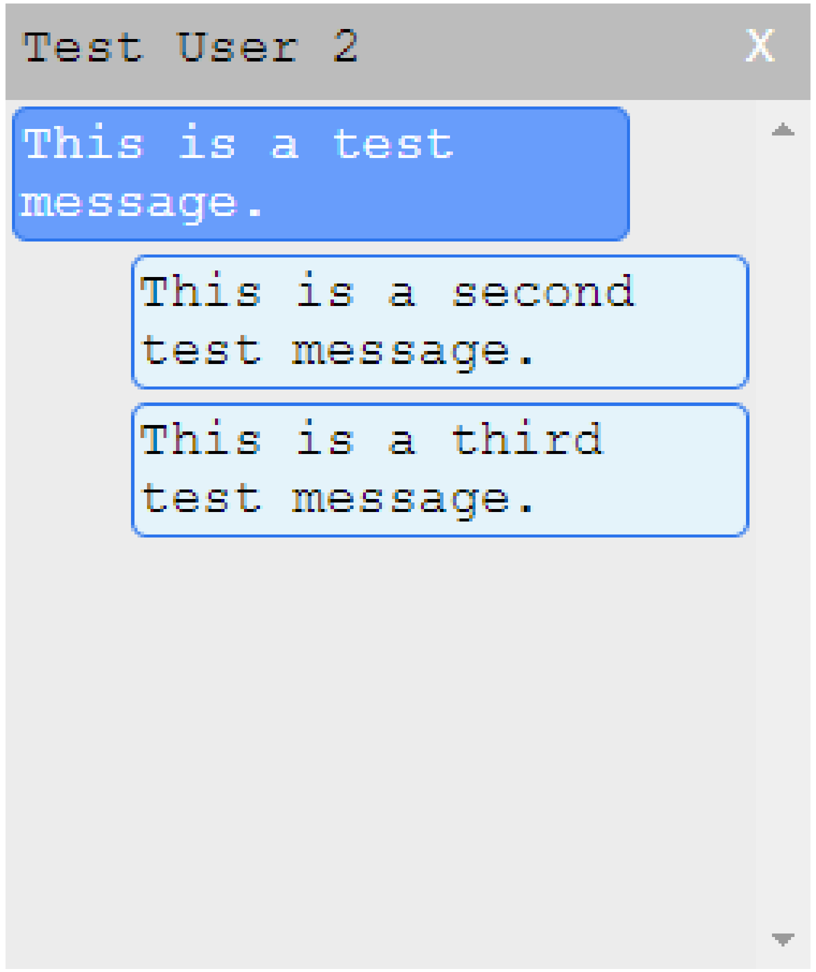
Fetching and displaying messages

- The last thing missing is some styling. The CSS is pretty big.
- Every message from the other users should be displayed on the left, and our own messages on the right, in order to differentiate them.
- Insert the following CSS into the style.css file:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/4_19.txt

Fetching and displaying messages

- Take a look at the following screenshot:


Test User 2

This is a test message.

This is a second test message.

This is a third test message.

Sending messages through Mutations

- First, parse the mutation at the top, next to the other requests:

```
const ADD_MESSAGE = gql`
  mutation addMessage($message : MessageInput!) {
    addMessage(message : $message) {
      id
      text
      user {
        id
      }
    }
  }
`;
```

Sending messages through Mutations

- For each open chat, we will have one input where the user can type his message.
- There are multiple solutions to save all of the inputs' text inside the React component's state.
- For now, we will keep it simple, but we will take a look at a better way to do this in the lesson 5, Reusable React Components.
- Open a new object inside of the state initializer in our Chats class:

```
textInputs: {}
```

Sending messages through Mutations

- Import the Mutation component from the react-apollo package, as follows:

```
import { Query, Mutation } from 'react-apollo';
```

- Replace the existing openChat and closeChat methods with the following code:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/4_20.txt

Sending messages through Mutations

- Now, we must also handle the change event of the input by implementing a special function, as follows:

```
onChangeChatInput = (event, id) => {
  event.preventDefault();
  var textInputs = Object.assign({}, this.state.textInputs);
  textInputs[id] = event.target.value;
  this.setState({ textInputs });
}
```

TriveraTech TECHNOLOGY TRAINING

Sending messages through Mutations

- The Mutation component is rendered before the input, so that we can pass the mutation function to the input.

- The input inside receives the onChange property, in order to execute the onChangeChatInput function while typing:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/4_21.txt

- The implementation of the handleKeyPress method is pretty straightforward. Just copy it into our component, as follows:

```
handleKeyPress = (event, id, addMessage) => {
  const self = this;
  var textInputs = Object.assign({}, this.state.textInputs);

  if (event.key === 'Enter' && textInputs[id].length) {
    addMessage({ variables: { message: { text: textInputs[id], chatId: id
}
     } }).then(() => {
      textInputs[id] = '';
      self.setState({ textInputs });
    });
  }
}
```

Sending messages through Mutations

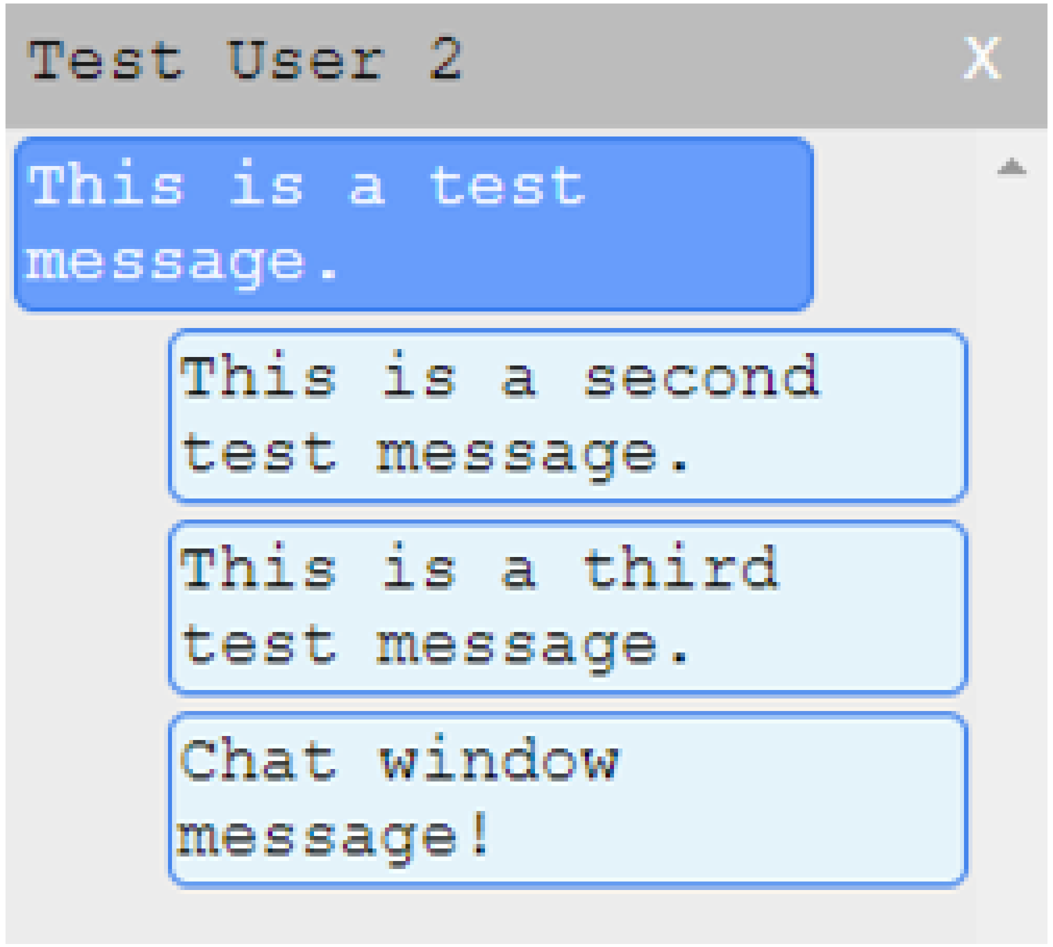
- Let's quickly add some CSS to our style.css file, to make the input field look good:

```
.chatWindow .input self. SetState {
  width: calc(100% - 4px);
  border: none;
  padding: 2px;
}
.chatWindow .input input:focus {
  outline: none;
}
```

TriveraTech
TECHNOLOGY TRAINING

Sending messages through Mutations

- The following screenshot shows the chat window, with a new message inserted through the chat window input:



Test User 2

X

This is a test message.

This is a second test message.

This is a third test message.

Chat window message!

Pagination in React and GraphQL

- By pagination, most of the time, we mean the batch querying of data.
- Currently, we query for all posts, chats, and messages in our database.
- If you think about how much data Facebook stores inside one chat with your friends, you will realize that it is unrealistic to fetch all of the messages and data ever shared at once.

Pagination in React and GraphQL

- Add a new RootQuery to our GraphQl schema, as follows:

```
postsFeed(page: Int, limit: Int): PostFeed
```

- The PostFeed type only holds the posts field. Later on, in the development of the application, you can return more information, such as the overall count of items, the page count, and so on:

```
type PostFeed {
  posts: [Post]
}
```

Pagination in React and GraphQL

- Next, we must implement the PostFeed entity in our resolvers.js file.
- Copy the new resolver function over to the resolvers file, as follows:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/4_22.txt

Pagination in React and GraphQL

- Our front end needs some adjustments to support pagination.
- Install a new React package with npm, which provides us with an infinite scroll implementation:

```
npm install react-infinite-scroller –save
```

TriveraTech TECHNOLOGY TRAINING

Pagination in React and GraphQL

- You are free to program this on your own, but we are not going to cover that here.

- Go back to the Feed.js file, replace the GET_POSTS query, and import the react-infinite-scroller package, with the following code:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/4_23.txt

Pagination in React and GraphQL

- Since the postsFeed query expects parameters other than the standard query from before, we need to edit our Query component in the render method.
- The changed lines are as follows:

```
<Query query={GET_POSTS} variables={{page: 0, limit: 10}}>
  {({ loading, error, data, fetchMore }) => {
  if (loading) return <p>Loading...</p>;
  if (error) return error.message;

  const { postsFeed } = data;
  const { posts } = postsFeed;
```

TriveraTech TECHNOLOGY TRAINING

Pagination in React and GraphQL

- According to the new data structure defined in our GraphQL schema, we extract the posts array from the postsFeed object.

- Replace the markup of the div tag of our current feed to make use of our new infinite scroll package:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/4_24.txt

Pagination in React and GraphQL

- It is important that we initialize the hasMore and page index state variable in our class first.
- Insert the following code:

```
state = {
  postContent: '',
  hasMore: true,
  page: 0,
}
```

Pagination in React and GraphQL

- Of course, we must also extract the hasMore variable in the render method of our class:

```
const { postContent, hasMore } = this.state;
```

Pagination in React and GraphQL

- We need to implement the loadMore function before running the infinite scroller.
- It relies on the page variable that we just configured.
- The loadMore function should look like the following code:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/4_25.txt

Pagination in React and GraphQL

- We have a second layer, postsFeed, as the parent of the posts array.
- Change the update function to get it working again, as follows:

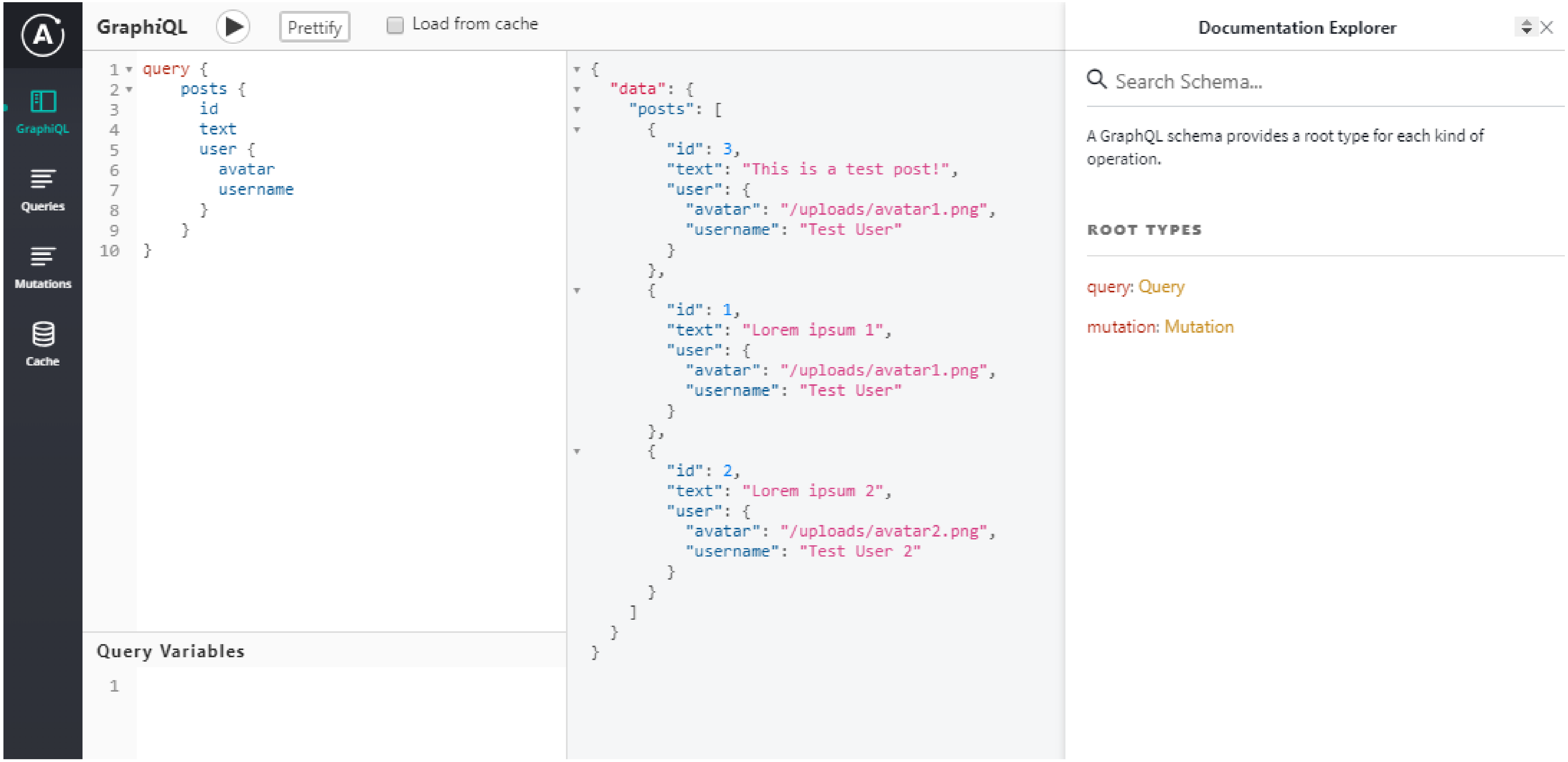
```
update = {(store, { data: { addPost } }) => {
  const variables = { page: 0, limit: 10 };
  const data = store.readQuery({ query: GET_POSTS,
variables });
  data.postsFeed.posts.unshift(addPost);
  store.writeQuery({ query: GET_POSTS, variables, data });
}}
```

Debugging with the Apollo Client Developer Tools

- Apollo Client Developer Tools is another Chrome extension, allowing you to send Apollo requests.

- While Postman is great in many ways, it does not integrate with our application, and does not implement any GraphQL-specific features.

- Apollo Client Developer Tools rely on the Apollo Client that we set up very early on in this lesson.



```
query {
    posts {
      id
      text
      user {
        avatar
        username
      }
    }
}
```

```
{
  "data": {
    "posts": [
      {
        "id": 3,
        "text": "This is a test post!",
        "user": {
          "avatar": "/uploads/avatar1.png",
          "username": "Test User"
        }
      },
      {
        "id": 1,
        "text": "Lorem ipsum 1",
        "user": {
          "avatar": "/uploads/avatar1.png",
          "username": "Test User"
        }
      },
      {
        "id": 2,
        "text": "Lorem ipsum 2",
        "user": {
          "avatar": "/uploads/avatar2.png",
          "username": "Test User 2"
        }
      }
    ]
  }
}
```

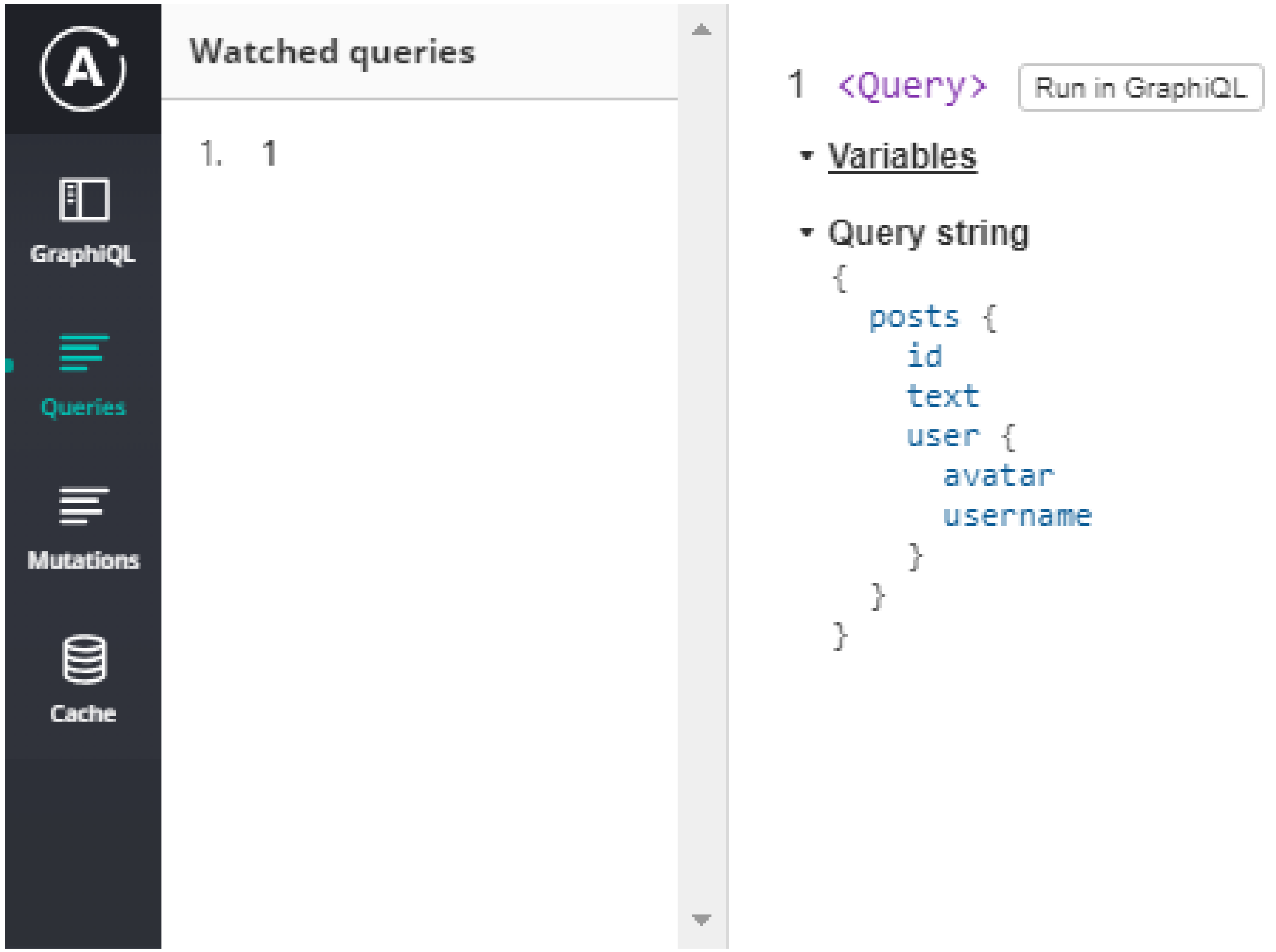
A GraphQL schema provides a root type for each kind of operation.

ROOT TYPES

query: Query

mutation: Mutation

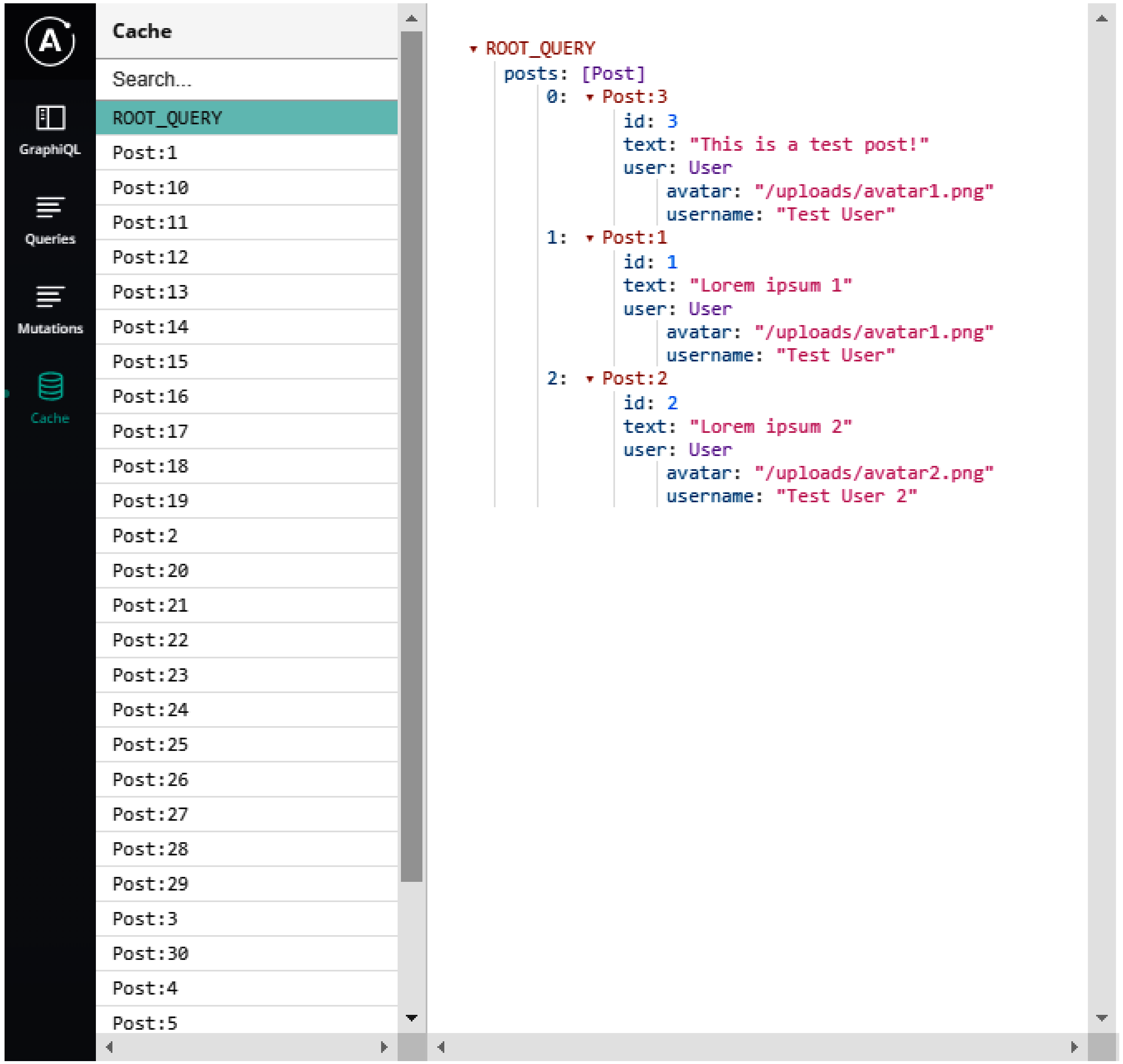
Debugging with the Apollo Client Developer Tools



```
{
  posts {
    id
    text
    user {
      avatar
      username
    }
  }
}
```

- The last window is Cache.

- Here, you are able to see all of the data stored inside the Apollo cache:



Cache

Search...

```
ROOT_QUERY
Post:1
Post:10
Post:11
Post:12
Post:13
Post:14
Post:15
Post:16
Post:17
Post:18
Post:19
Post:2
Post:20
Post:21
Post:22
Post:23
Post:24
Post:25
Post:26
Post:27
Post:28
Post:29
Post:3
Post:30
Post:4
Post:5
```

```
▾ ROOT_QUERY
    posts: [Post]
      0: ▾ Post:3
           id: 3
           text: "This is a test post!"
           user: User
             avatar: "/uploads/avatar1.png"
             username: "Test User"
      1: ▾ Post:1
           id: 1
           text: "Lorem ipsum 1"
           user: User
             avatar: "/uploads/avatar1.png"
             username: "Test User"
      2: ▾ Post:2
           id: 2
           text: "Lorem ipsum 2"
           user: User
             avatar: "/uploads/avatar2.png"
             username: "Test User 2"
```

Summary

- In this lesson, you learned how to connect our GraphQL API to React.
- To do this, we used Apollo Client to manage the cache and the state of our components, and to update the React and the actual DOM of the browser.
- We looked at how to send queries and mutations against our server in two different ways.

"Complete Lab"

7: Reusable React Components

Reusable React Components

This lesson will cover everything you need to know in order to write efficient and reusable React components. It will cover the following topics:

- React patterns
- Structured React components
- Rendering nested components
- The React Context API
- The Apollo Consumer component

Introducing React patterns

We will go over the most commonly used patterns that React offers, as follows:

- Controlled components
- Stateless functions
- Conditional rendering
- Rendering children

Controlled components

- By definition, a component is uncontrolled whenever the value is not set by a property through React, but only saved and taken from the real browser DOM.

- The value of an input is then retrieved from a reference to the DOM Node, and is not managed and taken from React's component state.

Controlled components

- The following code shows the post form where the user will be able to submit new posts.

- I have excluded the rendering logic for the complete feed, as it is not a part of the pattern that I want to show you:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/5_1.txt

Stateless functions

- One fundamental and efficient solution for writing well-structured and reusable React components is the use of stateless functions.

- As you might expect, stateless functions are functions, not React components.

- They are not able to store any states; only properties can be used to pass and render data.

Stateless functions

- Beginning with the file structure, we will create a new folder for our new components (or stateless functions), as follows:

```
mkdir src/client/components
```

- Many parts of our application need to be reworked. Create a new file for our first stateless function, as follows:

```
touch src/client/components/loading.js
```

Stateless functions

- Currently, we display a dull and boring Loading... message when our GraphQL requests are running.

- Let's change this by inserting the following code into the loading.js file:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/5_2.txt

Stateless functions

- Lastly, we are returning a simple div tag with the CSS style and the bouncer class.
- What's missing here is the CSS styling.
- The code should look as follows; just add it to our style.css file:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/5_3.txt

Stateless functions

- First, import the new loading spinner to the top of your files, as follows:

```
import Loading from './components/loading’;
```

- You can then render the stateless function like any normal component, as follows:

```
if (loading) return <Loading />;
```

- Generally, you can accomplish conditional rendering by using the curly brace syntax. An example of an if statement is as follows:

```
render() {
  const { shouldRender } = this.state;

  return (
    <div className="conditional">
      {(shouldRender === true) && (
        <p>Successful conditional rendering!</p>
      )}
    </div>
  )
}
```

Rendering child components

- In all of the code that we have written so far, we have directly written the markup like it is rendered to real HTML.

- A great feature that React offers is the ability to pass children to other components.

- The parent component decides what is done with its children.

- Create an error.js file next to the loading.js file in the components folder, as follows:

```
import React, { Component } from 'react';

export default class Error extends Component {
  render() {
    const { children } = this.props;

    return (
      <div className="error message">
        {children}
      </div>
    );
  }
}
```

Rendering child components

- To start using the new Error component, you can simply import it.

- The markup for the new component is as follows:

```
if (error) return <Error><p>{error.message}</p></Error>;
```

Rendering child components

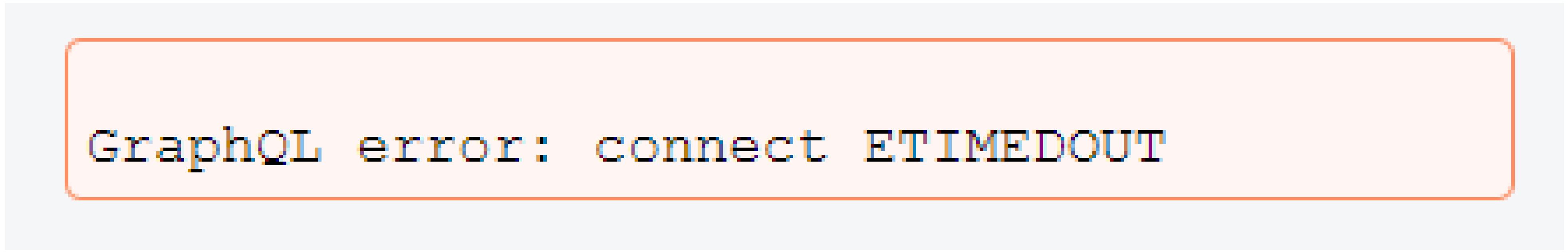
- Add some CSS, and everything should be finished, as shown in the following code snippet:

```
.message {
  margin: 20px auto;
  padding: 5px;
  max-width: 400px;
}
.error.message {
  border-radius: 5px;
  background-color: #FFF7F5;
  border: 1px solid #FF9566;
  width: 100%;
}
```

TriveraTech TECHNOLOGY TRAINING

Rendering child components

- A working result might look as follows:



```
GraphQL error: connect ETIMEDOUT
```

Structuring our React application

- We have already improved some things by using React patterns.

- You should do some homework and introduce those patterns wherever possible.

- When writing applications, one key objective is to keep them modular and readable, but also as understandable as possible.

The React file structure

- We have already saved our Loading and Error components in the components folder.
- Still, there are many parts of our components that we did not save in separate files, to improve the readability of this course.
- I will explain the most important solution for unreadable React code in one example.

The React file structure

- Instead of creating a post.js file in our components folder, we should first create another post folder, as follows:

```
mkdir src/client/components/post
```

The React file structure

- Create a new header.js file in the components/post folder, as follows:

```
import React from 'react';

export default ({post}) =>
  <div className="header">
    <img src={post.user.avatar} />
    <div>
      <h2>{post.user.username}</h2>
    </div>
  </div>
```

TriveraTech TECHNOLOGY TRAINING

The React file structure

- Up next is the post content, which represents the body of a post item.
- Add the following code inside of a new file, called content.js:

```
import React from 'react';

export default ({post}) =>
  <p className="content">
    {post.text}
  </p>
```

- The main file is a new index.js file in the new post folder. It should look as follows:

```
import React, { Component } from 'react';
import PostHeader from './header';
import PostContent from './content';

export default class Post extends Component {
  render() {
    const { post } = this.props;

    return (
      <div className={"post " + (post.id < 0 ? "optimistic": "")}>
        <PostHeader post={post}/>
        <PostContent post={post}/>
      </div>
    )
  }
}
```

The React file structure

- You can now use the new Post component in the feed list with ease.
- Just replace the old code inside the loop, as follows:

```
<Post key={post.id} post={post} />
```

TriveraTech TECHNOLOGY TRAINING

Efficient Apollo React components

- We have successfully replaced the post items in our feed with a React component, instead of raw markup.
- A major part, which I dislike very much, is the Apollo Query component and Mutation component, and how we are using these at the moment directly inside the render method of our components.
- I will show you a quick workaround to make these components more readable.

The Apollo Query component

- Create a new queries folder inside of the components folder, as follows:

```
mkdir src/client/components/queries
```

The Apollo Query component

- The query that we want to remove from our view layer is the postsFeed query.
- You can define the naming conventions for this, but I would recommend using the RootQuery name as the filename, as long as it works.
- So, we should create a postsFeed.js file in the queries folder, and insert the following code:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/5_4.txt

The Apollo Query component

- Preparing our next component, we split the infinite scroll area into a second file.

- Place a feedlist.js into the components/posts folder, as follows:

Refer to the file 5_5.txt

The Apollo Query component

- Import the new components in the Feed.js, as follows:

```
import FeedList from './components/post/feedlist';
import PostsFeedQuery from
'./components/queries/postsFeed';
```

TriveraTech
TECHNOLOGY TRAINING

The Apollo Query component

- Replace the div tag with the feed class name and our two new components, as follows:

```
<PostsFeedQuery>
  <FeedList />
</PostsFeedQuery>
```

TriveraTech
TECHNOLOGY TRAINING

The Apollo Mutation component

- Create a new folder for all your mutations, as follows:

mkdir src/client/components/mutations

- Next, we want to outsource the mutation into a special file.
- To do so, create the addPost.js file, named after the GraphQL mutation itself, Insert the following code:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/5_6.txt

The Apollo Mutation component

- Going on, we should build a post form component that only handles the creation of new posts.

- Just call it form.js, and place it inside of the post's components folder.

- The code must look like the following snippet:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/5_7.txt

TriveraTech TECHNOLOGY TRAINING

The Apollo Mutation component

- Lastly, we finalize the Feed.js main file. It should look as follows:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/5_8.txt

Extending Graphbook

- Adding a drop-down menu to the posts, in order to allow for deleting or updating the content.
- Creating a global user object with the React Context API.
- Using Apollo Consumer as an alternative to the React Context API.
- Implementing a top bar as the first component rendered above all of the views. We can search for users in our database from a search bar, and we can show the logged-in user from the global user object.

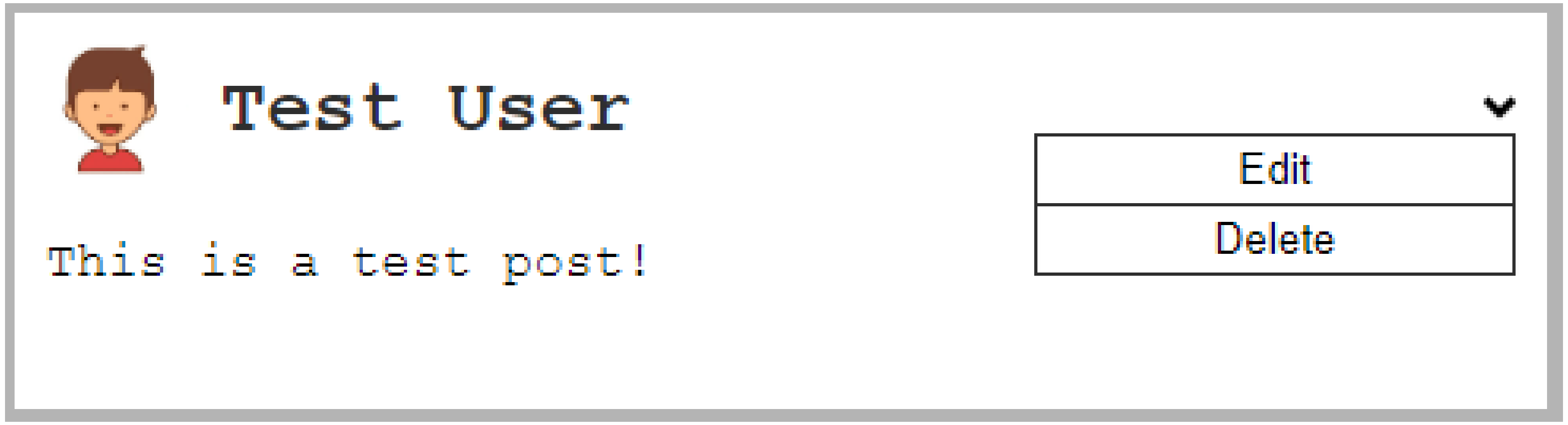
The React context menu

We will layout the plan that we want to follow:

- Rendering a simple icon with FontAwesome
- Building React helper components
- Handling the onClick event and setting the correct component state
- Using the conditional rendering pattern to show the drop-down menu, if the component state is set correctly
- Adding buttons to the menu and binding mutations to them

The React context menu

- The following is a preview screenshot, showing how the final implemented feature should look:


Test User

This is a test post!

FontAwesome in React

- As you may have noticed, we have not installed FontAwesome yet.
- Let's fix this with npm:

```
npm i --save @fortawesome/fontawesome-svg-core
@fortawesome/free-solid-svg-icons
@fortawesome/free-brands-svg-icons
@fortawesome/react-fontawesome
```

FontAwesome in React

- Creating a separate file for FontAwesome will help us to have a clean import.
- Save the following code under the fontawesome.js file, inside of the components folder:

```
import { library } from '@fortawesome/fontawesome-svg-core';
import { faAngleDown } from '@fortawesome/free-solid-svg-icons';
library.add(faAngleDown);
```

FontAwesome in React

- The only place where we need this file is within our root App.js file.

- It ensures that all of our custom React components can display the imported icons.

- Add the following import statement to the top:

```
import './components/fontawesome';
```

TriveraTech
TECHNOLOGY TRAINING

React helper components

- Production-ready applications need to be polished as much as possible. Implementing reusable React components is one of the most important things to do.

- You should notice that drop-down menus are a common topic when building client-side applications.

- They are global parts of the front end and appear everywhere throughout our components.

React helper components

- Logically, the first step is to create a new folder to store all of the helper components, as follows:

mkdir src/client/components/helpers

- Create a new file, called dropdown.js, as the helper component:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/5_9.txt

- Replace the handleClick method and add the componentWillUnmount React method, as follows:

```
componentWillUnmount() {
  document.removeEventListener('click', this.handleClick, true);
}
handleClick = () => {
  const { show } = this.state;

  this.setState({show: !show}, () => {
    if(!show) {
      document.addEventListener('click', this.handleClick);
    } else {
      document.removeEventListener('click', this.handleClick);
    }
  });
}
```

The GraphQL updatePost mutation

- There is a new mutation that we need to insert into our schema, as follows:

```
updatePost (
  post: PostInput!
  postId: Int!
): Post
```

The GraphQL updatePost mutation

- Once it is inside of our schema, the implementation to execute the mutation will follow.

- Copy the following code over to the resolvers.js file, in the RootMutation field:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/5_10.txt

The GraphQL updatePost mutation

- Add the new updatePost mutation to the new file, as follows:

```
const UPDATE_POST = gql`
  mutation updatePost($post : PostInput!, $postId : Int!) {
    updatePost(post : $post, postId : $postId) {
      id
      text
    }
  }
`;
```

The GraphQL updatePost mutation

- Create an UpdatePostMutation class, as follows:

```
export default class UpdatePostMutation extends Component
{
  state = {
    postContent: this.props.post.text
  }
  changePostContent = (value) => {
    this.setState({postContent: value})
  }
}
```

TriveraTech TECHNOLOGY TRAINING

The GraphQL updatePost mutation

- A React component always needs a render method.
- This one is going to be a bit bigger:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/5_11.txt

The GraphQL updatePost mutation

- We will handle this within the Post component itself, because we want to edit the post in place, and do not want to open a modal or a specific Edit page.
- Go to your post's index.js file and exchange it with the new one, as follows:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/5_12.txt

The GraphQL updatePost mutation

- We already have a post form that we can reuse with some adjustments, as you can see in the following code snippet.
- To use our standard post submission form as an update form, we must make some small adjustments.
- Open and edit the form.js file, as follows:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/5_13.txt

- Go to your post header file. The header is a great place to insert the drop-down component, as follows:

```
import React from 'react';
import Dropdown from '../helpers/dropdown';
import { FontAwesomeIcon } from '@fortawesome/react-fontawesome';
export default ({post, changeState}) =>
  <div className="header">
    <img src={post.user.avatar} />
    <div>
      <h2>{post.user.username}</h2>
    </div>
    <Dropdown trigger={<FontAwesomeIcon icon="angle-down" />}>
      <button onClick={changeState}>Edit</button>
    </Dropdown>
  </div>
```

The Apollo deletePost mutation

- Edit the GraphQL schema. The deletePost mutation needs to go inside of the RootMutation object.
- The new Response type serves as a return value, as deleted posts cannot be returned because they do not exist.
- Note that we only need the postId parameter, and do not send the complete post:

```
type Response {
  success: Boolean
}
deletePost (
  postId: Int!
): Response
```

The Apollo deletePost mutation

- Add the missing GraphQL resolver function, The code is pretty much the same as from the update resolver, except that only a number is returned by the destroy method of Sequelize, not an array.
- It represents the number of deleted rows, We return an object with the success field.
- This field indicates whether our front end should throw an error:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/5_14.txt

TriveraTech TECHNOLOGY TRAINING

The Apollo deletePost mutation

- Add the new deletePost mutation, as follows:

```
const DELETE_POST = gql`
  mutation deletePost($postId : Int!) {
    deletePost(postId : $postId) {
      success
    }
  }
`;
```

The Apollo deletePost mutation

- Lastly, insert the new component's code:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/5_15.txt

The Apollo deletePost mutation

- Open the header.js file and import the following mutation:

```
import DeletePostMutation from '../mutations/deletePost';
```

- Instead of directly adding the new button to our header, we will create another stateless function, as follows:

```
const DeleteButton = ({deletePost, postId}) =>
  <button onClick={() => {
    deletePost({ variables: { postId } })
  }}>
    Delete
  </button>
```

The Apollo deletePost mutation

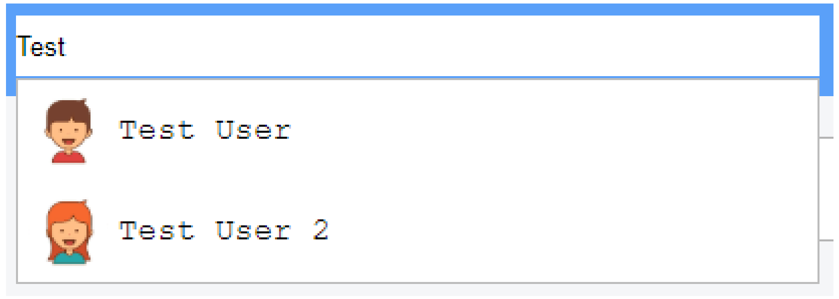
- Insert both the mutation and the delete button into the header function, below the 'Edit' button, as follows:

```
<DeletePostMutation post={post}>
  <DeleteButton />
</DeletePostMutation>
```

TriveraTech
TECHNOLOGY TRAINING

The React application bar

- The first thing that we will implement is the simple search for users and the information about the logged-in user.
- We will begin with the search component, because it is really complex.
- The following screenshot shows a preview of what we are going to build:



The React application bar

- Edit the GraphQL schema and fill in the new RootQuery and type, as follows:

```
type UsersSearch {
  users: [User]
}

usersSearch(page: Int, limit: Int, text: String!): UsersSearch
```

The React application bar

- Furthermore, the resolver function looks pretty much the same as the postsFeed resolver function.

- You can add the following code straight into the resolvers.js file, as follows:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/5_16.txt

The React application bar

- To enable this operator, we must import the sequelize package and extract the Op object from it, as follows:

```
import Sequelize from 'sequelize';
const Op = Sequelize.Op;
```

The React application bar

- Import all of the dependencies, and parse the new GraphQL query with the graphql-tag package, Note that we have three parameters.

- The text field is a required property for the variables that we send with our GraphQL request:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/5_17.txt

The React application bar

- Paste in the UsersSearchQuery class, as shown in the following code.

- In comparison to the PostsFeedQuery class, I have added the text property to the variables and handed over the refetch method to all subsequent children:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/5_18.txt

The React application bar

- Continuing with our plan, we will create the application bar in a separate file.

- Create a new folder, called bar, below the components folder and the index.js file.

- Fill it in with the following code:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/5_19.txt

The React application bar

- The SearchBar class lives inside of a separate file. Just create a search.js file in the bar folder, as follows:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/5_20.txt

The React application bar

- Next, we will implement the SearchList. This behaves like the posts feed, but only renders something if a response is given with at least one user.
- The list is displayed as a drop-down menu and is hidden whenever the browser window is clicked on.
- Create a file called searchList.js inside of the bar folder, with the following code:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/5_21.txt

The React Context API versus Apollo Consumer

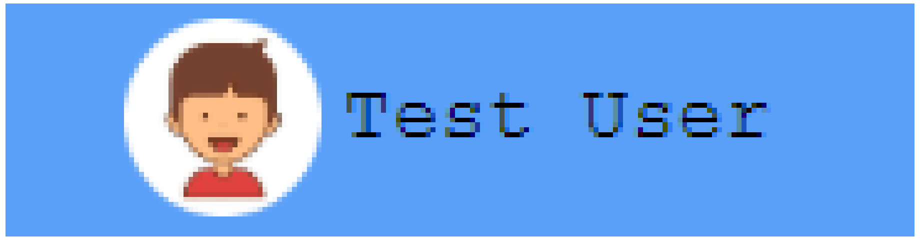
- There are two ways to handle global variables in the stack that we are using at the moment.

- These are the new React Context API and the Apollo Consumer functionality.

- From version 16.3 of React, there is a new Context API that allows you to define global providers offering data through deeply nested components.

The React Context API versus Apollo Consumer

- Both of the approaches will result in the following output:



```
Test User
```

- The best option is to show you the two alternatives right away, so that you can identify your preferred method.

The React Context API

The following is a short explanation of this method:

- Context: This is a React approach for sharing data between components, without having to pass it through the complete tree.
- Provider: This is a global component, mostly used at just one point in your code. It enables you to access the specific context data.
- Consumer: This is a component that can be used at many different points in your application, reading the data behind the context that you are referring to.

The React Context API

- As always, we need to import all of the dependencies.
- Furthermore, we will set up a new empty context.
- The createContext function will return one provider and consumer to use throughout the application, as follows:

```
import React, { Component, createContext } from 'react';
const { Provider, Consumer } = createContext();
```

TriveraTech
TECHNOLOGY TRAINING

The React Context API

- Now, we want to use the provider, The best option here is to create a special UserProvider component.
- Later, when we have authentication, we can adjust it to do the GraphQL query, and then share the resultant data in our front end.
- For now, we will stick with fake data. Insert the following code:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/5_22.txt

The React Context API

- We will set up a special UserConsumer component that takes care of passing the data to the underlying components by cloning them with React's cloneElement function:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/5_23.txt

The React Context API

- We need to introduce the provider at an early point in our code base.
- The best approach is to import the UserProvider in the App.js file, as follows:

```
import { UserProvider } from './components/context/user';
```

- Use the provider as follows, and wrap it around all essential components:

```
<UserProvider>
  <Bar />
  <Feed />
  <Chats />
</UserProvider>
```

The React Context API

- We could also have written the UserBar class as a stateless function, but we might need to extend this component in a later lesson.

- Insert the following code:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/5_24.txt

The React Context API

- Open the index.js file for the top bar and add the following code to the render method, next to the SearchBar component:

```
<UserConsumer>
  <UserBar />
</UserConsumer>
```

The React Context API

- Obviously, you need to import both of the components at the top of the file, as follows:

```
import UserBar from './user';
import { UserConsumer } from '../context/user';
```

Apollo Consumer

- Nearly all of the code that we have written can stay as it was in the previous section.
- We just need to remove the UserProvider from the App class, because it is not needed anymore for the Apollo Consumer.
- Open up the user.js in the context folder and replace the contents with the following code:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/5_25.txt

TriveraTech TECHNOLOGY TRAINING

Documenting React applications

- We have put a lot of work and code into our React application.

- To be honest, we can improve upon our code base by documenting it.

- We did not comment on our code, we did not add React component property type definitions, and we have no automated documentation tool.

Setting up React Styleguidist

React Styleguidist and our application rely on webpack. Just follow these instructions to get a working copy of it:

- Install React Styleguidist using npm, as follows:

```
npm install --save-dev react-styleguidist
```

TriveraTech TECHNOLOGY TRAINING

- Usually, the folder structure is expected to be src/components, but we have a client folder between the src and components folder, So, we must configure React Styleguidist to let it understand our folder structure.
- Create a styleguide.config.js in the root folder of the project to configure it, as follows:

```
const path = require('path')
module.exports = {
  components: 'src/client/components/**/*.js',
  require: [
    path.join(__dirname, 'assets/css/style.css')
  ]
  webpackConfig: require('./webpack.client.config')
}
```

Setting up React Styleguidist

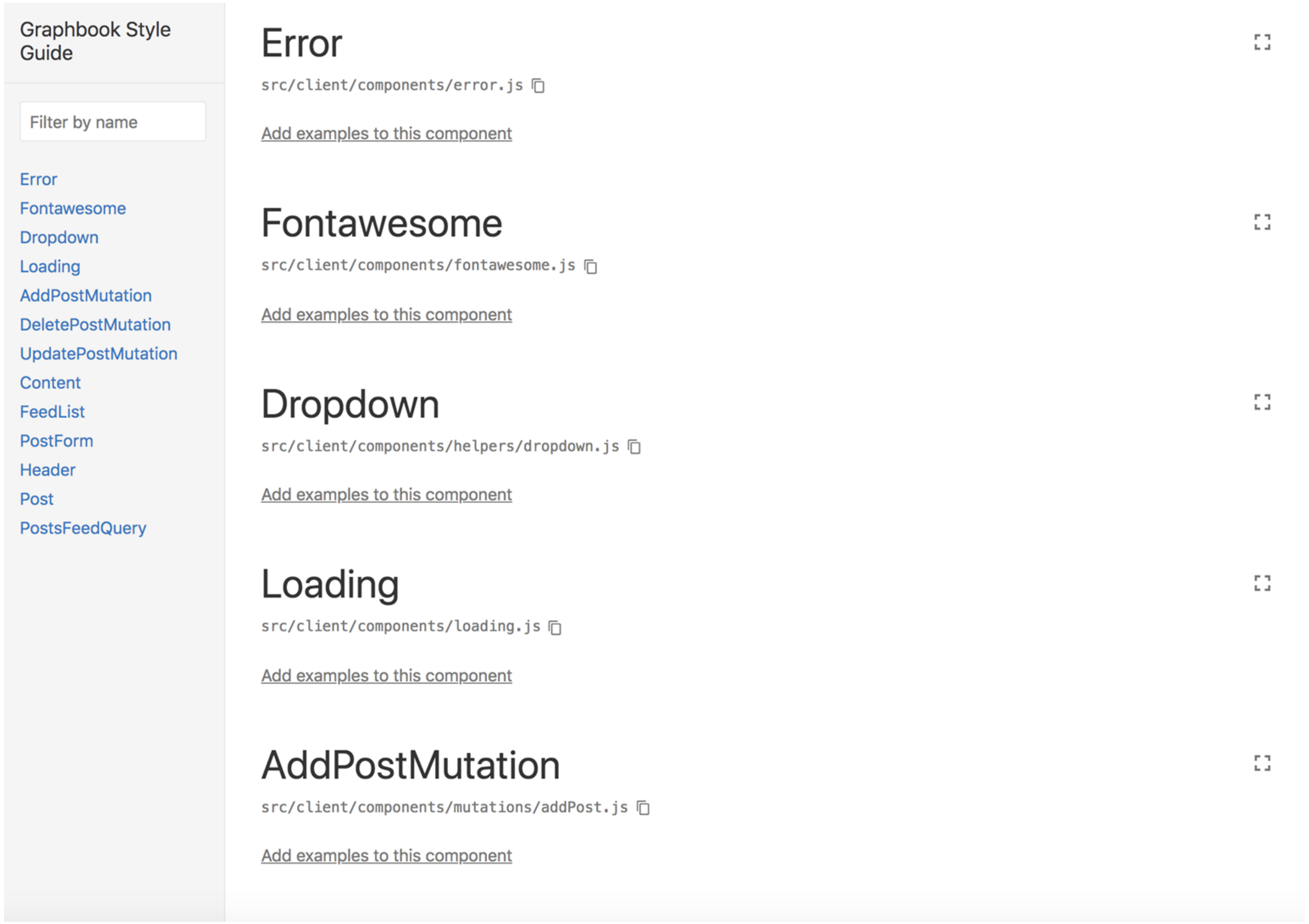
- Styleguidist provides two ways to view the documentation.
- One is to build the documentation statically, in production mode, with this command:

```
npx styleguidist build
```

Setting up React Styleguidist

- The second method, for development cases, lets Styleguidist run and create the documentation on the fly, using webpack:

```
npx styleguidist server
```



Graphbook Style Guide

Filter by name

- Error
- Fontawesome
- Dropdown
- Loading
- AddPostMutation
- DeletePostMutation
- UpdatePostMutation
- Content
- FeedList
- PostForm
- Header
- Post
- PostsFeedQuery

Error

`src/client/components/error.js`

Add examples to this component

Fontawesome

`src/client/components/fontawesome.js`

Add examples to this component

Dropdown

`src/client/components/helpers/dropdown.js`

Add examples to this component

Loading

`src/client/components/loading.js`

Add examples to this component

AddPostMutation

`src/client/components/mutations/addPost.js`

Add examples to this component

React PropTypes

There are two React features that we did have covered yet, as follows:

- If your components have optional parameters, it can make sense to have default properties in the first place. To do this, you can specify defaultProps as a static property, in the same way as with the state initializers.
- The important part is the propTypes field, which you can fill for all of your components with the custom properties that they accept.

React PropTypes

- A new package is required to define the property types, as follows:

```
npm install --save prop-types
```

React PropTypes

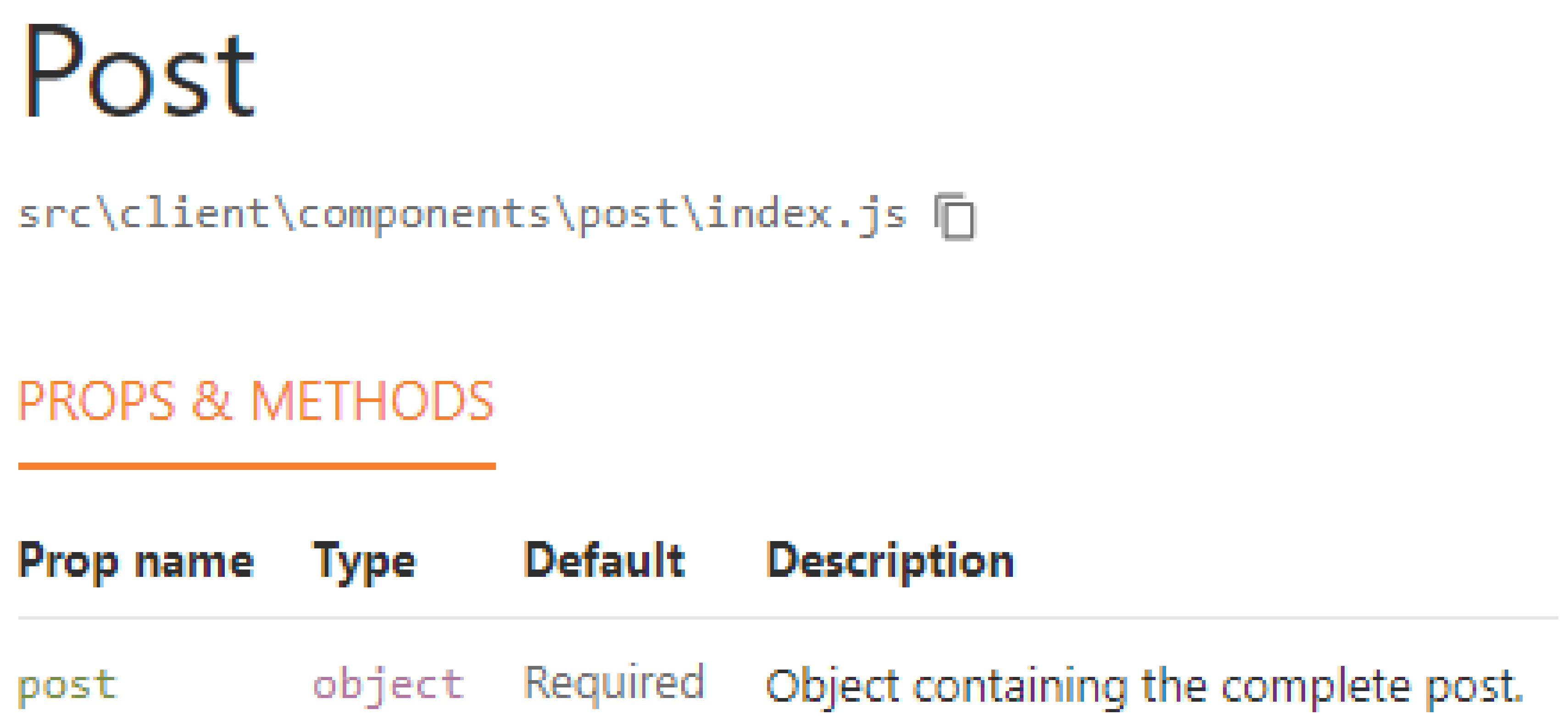
- Now, open your Post component's index.js file.
- We need to import the new package at the top of the Post component's index.js file:

```
import PropTypes from 'prop-types';
```

- Next, we will add the new field to our component, above the state initializers:

```
static propTypes = {
  /** Object containing the complete post. */
  post: PropTypes.object.isRequired,
}
```

React PropTypes



Post

`src\client\components\post\index.js`

PROPS & METHODS

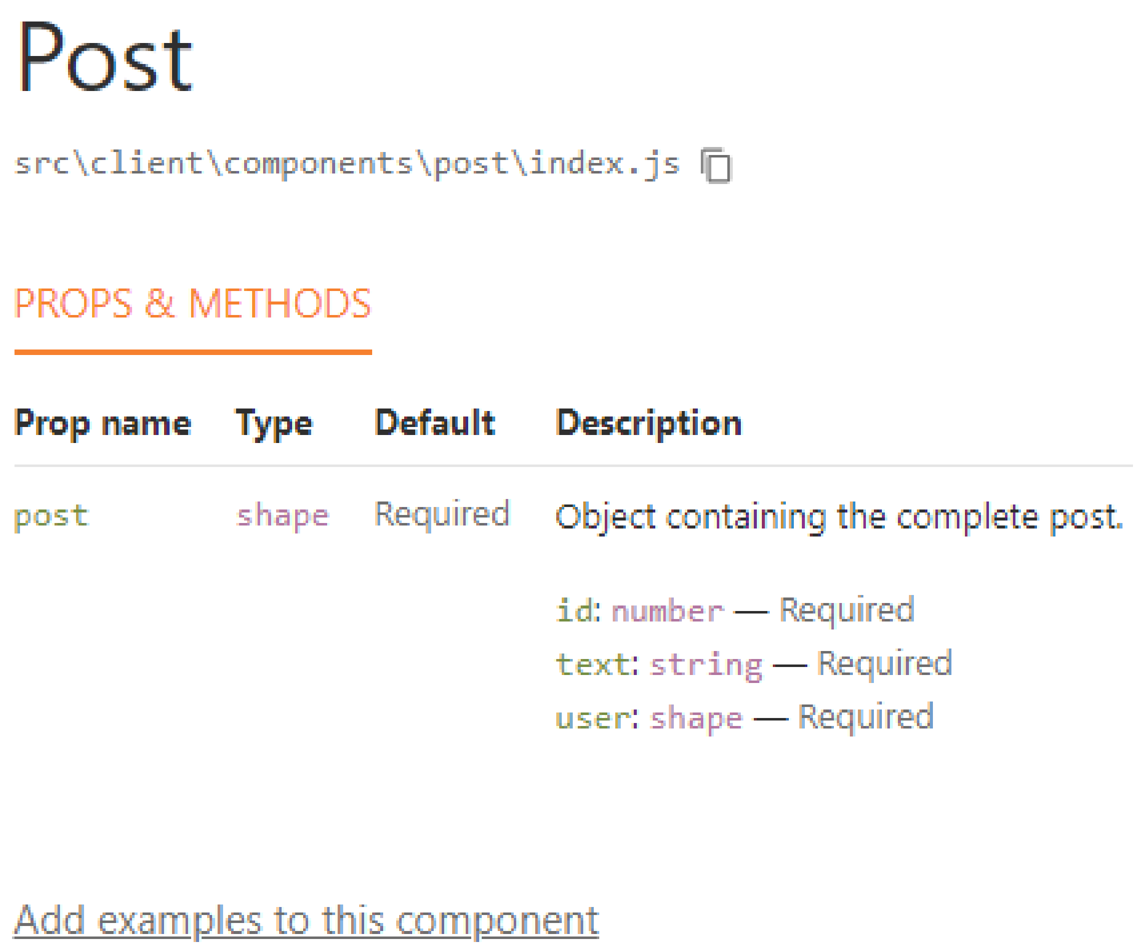
Prop name	Type	Default	Description
post	object	Required	Object containing the complete post.

- The best way to document a post object is to define which properties a post should include, at least for this specific component.
- Replace the property definition, as follows:

```
static propTypes = {
  /** Object containing the complete post. */
  post: PropTypes.shape({
    id: PropTypes.number.isRequired,
    text: PropTypes.string.isRequired,
    user: PropTypes.shape({
      avatar: PropTypes.string.isRequired,
      username: PropTypes.string.isRequired,
    }).isRequired
  }).isRequired,
}
```

React PropTypes

- The output from React Styleguidist now looks like the following screenshot:



Post

src\client\components\post\index.js

PROPS & METHODS

Prop name	Type	Default	Description
post	shape	Required	Object containing the complete post. id: number — Required text: string — Required user: shape — Required

Add examples to this component

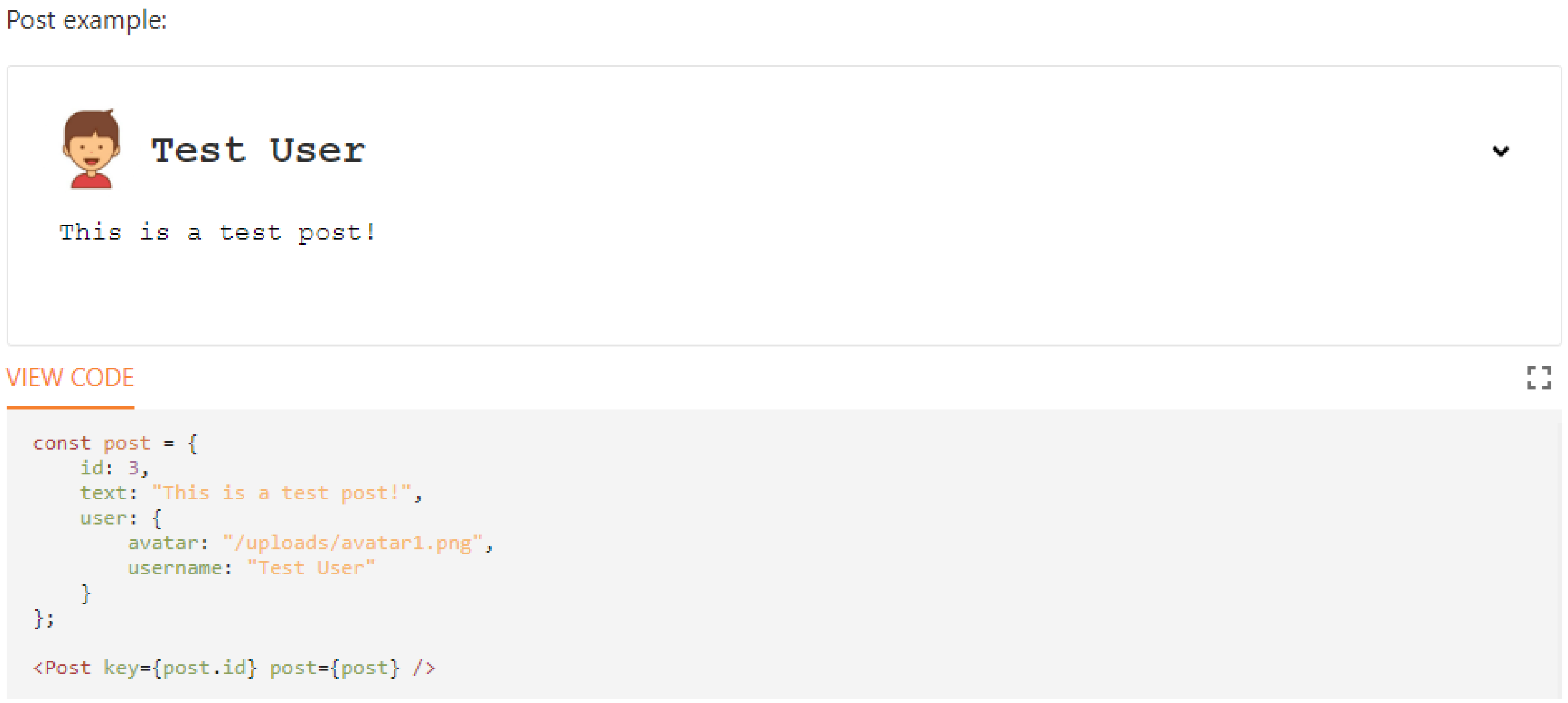
React PropTypes

- For our Post component, we need to create an index.md file, next to the index.js file in the post folder.
- React Styleguidist proposes creating either a Readme.md or Post.md file, but those did not work for me.
- The index.md file should look as follows:

Refer to https://github.com/fenago/react-graphql-intro/blob/master/snippets/5_26.txt

React PropTypes

- React Styleguidist automatically rerenders the documentation and generates the following output:



Post example:

Test User

This is a test post!

VIEW CODE

```
const post = {
    id: 3,
    text: "This is a test post!",
    user: {
        avatar: "/uploads/avatar1.png",
        username: "Test User"
    }
};

<Post key={post.id} post={post} />
```

Summary

Through this lesson, you have gained a lot of experience in writing a React application.

- You have applied multiple React patterns to different use cases, such as children passing through a pattern and conditional rendering.

- Furthermore, you now know how to document your code correctly.

"Complete Lab"