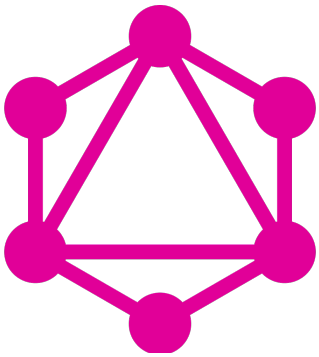
Introduction to GraphQL - Labs



Completed source code for all labs (for checking your work) can be found at:

`https://github.com/watzthisco/intro-to-graphql`

Version 1.0, October 2022
by Chris Minnick

www.watzthis.com

Disclaimers and Copyright Statement

Disclaimer

WatzThis? takes care to ensure the accuracy and quality of this courseware and related courseware files. We cannot guarantee the accuracy of these materials. The courseware and related files are provided without any warranty whatsoever, including but not limited to implied warranties of merchantability or fitness for a particular purpose. Use of screenshots, product names and icons in the courseware are for editorial purposes only. No such use should be construed to imply sponsorship or endorsement of the courseware, nor any affiliation of such entity with WatzThis?.

Third-Party Information

This courseware contains links to third-party web sites that are not under our control and we are not responsible for the content of any linked sites. If you access a third-party web site mentioned in this courseware, then you do so at your own risk. We provide these links only as a convenience, and the inclusion of the link does not imply that we endorse or accept responsibility for the content on those third-party web sites. Information in this courseware may change without notice and does not represent a commitment on the part of the authors and publishers.

Copyright

Help us improve our courseware

Please send your comments and suggestions via email to info@watzthis.com

Credits

About the Author

Chris Minnick (chris@minnick.com) is a prolific published author and the CEO of WatzThis, Inc.

He started his first company in 1996 and ran it successfully for over twenty years. During this time, he was responsible for the project management and development of hundreds of web and mobile projects for publishing and media clients.

Minnick has authored and co-authored over a dozen books including titles in the For Dummies series and several books for teaching kids to code. He has developed video courses for online training providers Pluralsight, O'Reilly Video, Ed2Go, and Skillshare. His current company, WatzThis, writes and maintains courseware that's used for training software developers at some of the largest companies in the world.

Minnick studied creative writing, literature, journalism, and film at the University of Michigan, and he worked in editorial, circulation, and online publishing capacities at newspapers and magazines in Ann Arbor, Detroit, and San Francisco. In addition to his technical writing and training, Chris is a novelist, painter, winemaker, swimmer, and musician.

Table of Contents

Setup Instructions

Course Requirements

To complete the labs in this course, you will need:

- A computer with MacOS, Windows, or Linux.
- Access to the Internet.
- A modern web browser.
- Ability to install software globally (or certain packages pre-installed as specified below).

Classroom Setup

These steps must be completed in advance if the students will not have administrative access to the computers in the classroom. Otherwise, these steps can be completed during the course as needed.

1. Install node.js on each student's computer. Go to nodejs.org and click the link to download the latest version from the LTS branch.
2. Install a code editor. We use Visual Studio Code in the course
3. Make sure Google Chrome is installed.
4. Install git on each student's computer. Git can be downloaded from *http://git-scm.com*. Select all the default options during installation.

Get the Repo

1. Open a command prompt.
    a. Use Terminal on MacOS (/Applications/Utilities/Terminal).
    b. Use gitbash on Windows (installed with git).
2. Enter `cd` to navigate to the user's home directory (or change to a directory where student files should be created).
3. Enter the following:

```
git clone https://github.com/chrisminnick/intro-to-graphql
```

Lab 1 - Getting Started with Apollo Server

Apollo is the leading open source GraphQL implementation. In this lab, you'll install some dependencies and get a first look at what Apollo Server can do.

Part 1: Set up a Node Package

- ☐ 1. Create a new project in VS Code.
- ☐ 2. Make two top-level directories, server and client
- ☐ 3. Open the integrated terminal and switch to the server directory

```
cd server
```

- ☐ 4. Initialize a Node package in /server

```
npm init -y
```

- ☐ 5. Make a src and a public folder in /server

  The src folder is where you'll write your server-side code, the public directory serves static assets

- ☐ 6. Make index.js in /server
- ☐ 7. Install dotenv and nodemon

```
npm install --save-dev dotenv nodemon
```

- ☐ 8. Add a start script to package.json to start up your server from src

```
"scripts": {
    "start": "nodemon src/index"
},
```

- ☐ 9. Install apollo-server and graphql

```
npm install apollo-server graphql
```

- ☐ 10. Start your server

```
npm run start
```

  At this point, index.js doesn't contain anything, so nodemon will just sit there and wait for it to change. When index.js changes, nodemon will restart the server.

- ☐ 11. Import ApolloServer into index.js

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

- ☐ 12. Create a server instance

```
const server = new ApolloServer({});
```

☐ 13. Start the server

```
server.listen().then(()=>{
  console.log('Server is running at
              http://localhost:4000');
}
```

Look at the console now and you'll see that the server has crashed. Read the error message, and you'll find out that ApolloServer requires a schema, modules, or typeDefs. We'll define a schema in the next part.

Part 2: Define a Schema

In this part, we'll define a schema for our app. A schema is a contract between the server and the client that defines what a GraphQL API can do and how clients can request or change data.

The app we'll be building is an exercise tracking app called "CoderFit." The idea of the app is that it allows programmers to track physical activities they do during the day.

Rather than starting to build the app by building the client or the database, we'll take a Schema-first approach. For now, we're just going to keep things simple and focus on displaying a Coder along with Activities performed by that coder.

☐ 1. Create a file named schema.js inside /server/src
☐ 2. Import the gql template literal from apollo-server

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

The gql tagged template literal wraps GraphQL strings and converts them to the format Apollo libraries expect, and it also enables syntax highlighting.

☐ 3. Declare a typeDefs constant and assign the gql template

```
const typeDefs = gql`
  # your type definitions here
`;
```

☐ 4. export typeDefs to create a module

```
module.exports = typeDefs;
```

The app will need to have several types of data:

- Coder: the person doing activities
- Activity: a single activity performed by a coder

To define a schema, we can use the `type` keyword followed by the name of the type.

- In curly braces, define individual fields using `fieldname:type` syntax.
- The type can be a single data type (Int, String, Float, Boolean, ID)
- ID (respresents a unique identifier not intended to be human-readable)
- The type can be another Type.
- Mark a field as required by using ! after the type.
- To define a field as a list, use square brackets `[]` around the type.

☐ 5. Create two types: Coder and Activity

```
type Coder {
  id: ID!
  name: String!
  description: String
  activities: [ID}
}

type Activity {
  id: ID!
  name: String!
  description: String
}
```

☐ 6. Comment your types.

- To create block comments, surround lines of text with tripple double quotes. """Like this"""
- To create single-line comments use single double quotes "like this"
- Put comments for a type immediately before the type
- Put comments for a field immediately before the field

The Query type defines the queries that are available to the client. We'll start with just a query to get all the coders:

☐ 7. Above your other types, define a Query type.

```
type Query {
  coders: [Coder]
}
```

☐ 8. Import your typeDefs into server/index.js. Under the line that imports ApolloServer, type the following:

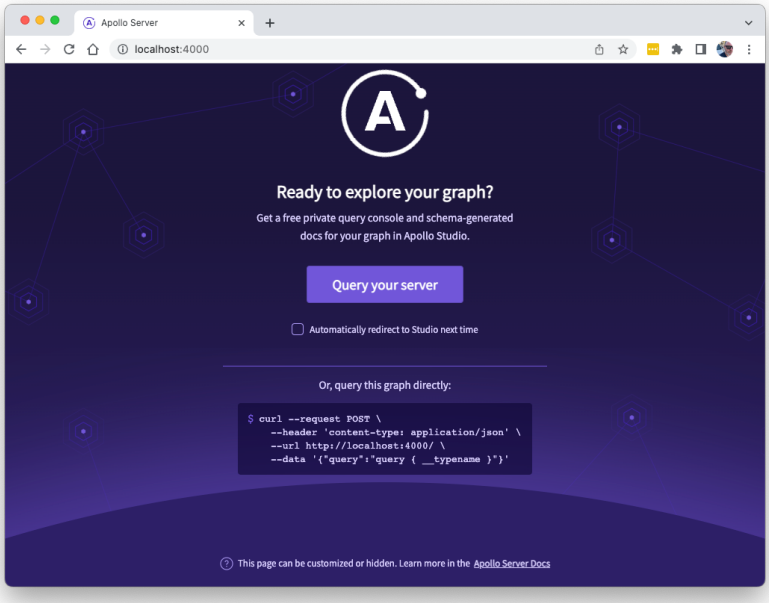
```
const typeDefs = require('./schema');
```

☐ 9. Pass the typeDefs to the server instance and set mocks to true. With mocks set to true, Apollo Server will generate basic mock data for fields.

```
const server = new ApolloServer({ typeDefs,
                                  mocks: true });
```

If you did everything correctly, the server should be running now.

- ☐ 10. Open http://localhost:4000/graphql in your browser. You'll see the Apollo Studio welcome page.



Lab 2: Using Apollo Studio

In this lab, you'll learn to use Apollo Studio to build and run queries.

- ☐ 1. With the server you built in Lab 1 running, go to http://localhost:4000
- ☐ 2. Click query your server.
- ☐ 3. You'll be taken to the studio interface.
- ☐ 4. Try running the example query and look at the response. Notice that it has the same fields as the query.
- ☐ 5. Add the name field to the query and run it again.

The mock data feature is great, but we can create some more realistic mock data by writing a mock object containing functions to return the certain values for each field.

- ☐ 6. In index.js, create a const named mocks containing an object with two properties: Query and Coder
- ☐ 7. The value of both properties should be a function that returns an object.

  So far you should have the following.

```
const mocks = {
  Query: () => ({}),
  Coder: () => ({}),
}
```

- ☐ 8. Add a single property to the Query prop, coders. The value should be a function that returns a an array containing 4 elements (or however many you want).

```
Query: () => ({
  coders: () => [...new Array(6)]
}),
```

- ☐ 9. Add the properties of coder (from the schema) to the object returned by coder.

  When you're done, your mock object should look something like this:

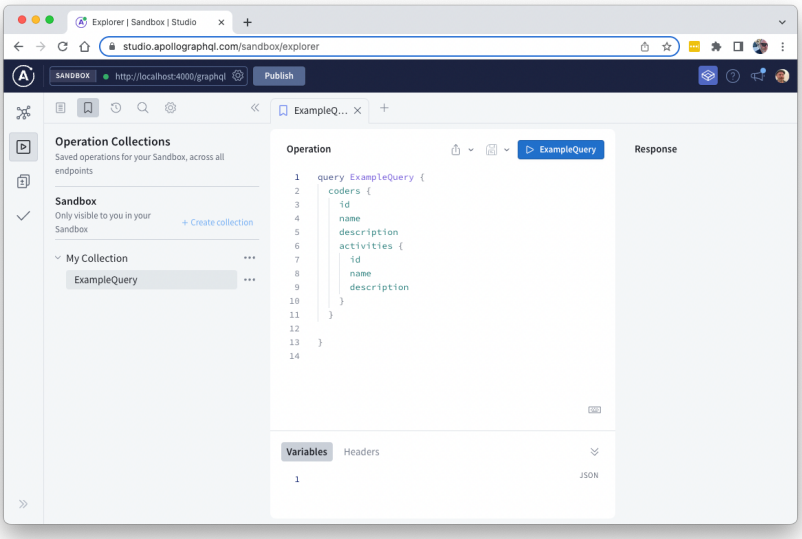
```
const mocks = {
  Query: () => ({
    coders: () => [...new Array(6)],
  }),
  Coder: () => ({
    id: () => '1',
    name: () => 'Peter Lundlehart',
    description: () => 'A coder who codes.',
    activities: () => [1],
  }),
  Activity: () => ({
    id: () => '1',
    name: () => 'Coding',
    description: () => 'Coding is fun'
  })
```

```
};
```

- ☐ 10. Pass mocks to the server instead of the mock = true argument:

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

- ☐ 11. Return to Apollo explorer and try some queries to see your mock data being returned.
- ☐ 12. Click the three dots menu the right of Fields in the Apollo Explorer and select 'Select all fields recursively.
- ☐ 13. Hover your mouse over coders and you'll see an arrow icon to the right of it. Click that to add all fields to the query.
- ☐ 14. Run the query.
- ☐ 15. Click the disk icon in the operation panel to save the query to a new collection.
- ☐ 16. Click the operation collections button at the top of the Apollo Explorer to see your collection and the saved query.



Lab 3: Creating a client

In this lab, you'll begin creating the app that will query the server.

- ☐ 1. Open the root of your project in the terminal
- ☐ 2. Scaffold a new React app using the method of your choice and name it coder-fit-client. For example:

```
npx create-react-app coder-fit-client
```

or

```
npm create vite@latest coder-fit-client -- --template
react
```

- ☐ 3. Start up your client app's dev server
- ☐ 4. Install graphql and the apollo client

```
npm install graphql @apollo/client
```

- ☐ 5. Import necessary components into index.js (or main.jsx)

```
import {ApolloClient, InMemoryCache, ApolloProvider}
from '@apollo/client';
```

- ☐ 6. Create a client instance:

```
const client = new ApolloClient({
  //options here
}
```

- ☐ 7. Set the uri and cache options for the client:

```
const client = new ApolloClient({
  uri: 'http://localhost:4000',
  cache: new InMemoryCache()
});
```

- ☐ 8. Wrap the <App> element with <ApolloProvider> and pass the client attribute.

```
ReactDOM.createRoot(document.getElementById('root'))
  .render(
  <React.StrictMode>
    <ApolloProvider client={client}>
      <App />
    </ApolloProvider>
  </React.StrictMode>
)
```

- ☐ 9. Import gql into App.js

```
import {gql} from '@apollo/client';
```

☐ 10. Declare a constant named CODERS to hold the query

```
const CODERS = gql `
  # Query here
`;
```

☐ 11. Copy your query from Apollo Explorer and paste it into the empty gql string.

☐ 12. Import the useQuery hook from @apollo/client

```
import {useQuery, gql} from '@apollo/client';
```

☐ 13. In the function, pass CODERS to useQuery and deconstruct the return values:

```
const {loading, error, data} = useQuery(CODERS);
```

☐ 14. Below that, return a loading message if loading is true.

```
if (loading) return 'Loading...';
```

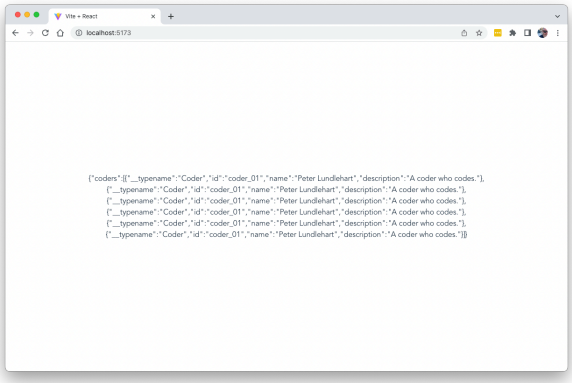
☐ 15. Return an error message if error is true:

```
if (error) return `Error: ${error.message}`;
```

☐ 16. Test your query by printing out the returned data, like this:

```
<div>{JSON.stringify(data)}</div>
```

The app will render something like this:



☐ 17. Make your component return a single element for each element returned by the query.

Lab 4: Connecting to a data source

In this lab, we'll connect the CoderFit app to a data source

- ☐ 1. Add the RESTDataSource class to the server

```
npm i apollo-datasource-rest
```

- ☐ 2. Make a folder named datasources in server/src
- ☐ 3. Create a file named coder-api.js in src/datasources
- ☐ 4. In coder-api.js, import RESTDataSource

```
const {RESTDataSource} = require('apollo-datasource-
rest');
```

- ☐ 5. Extend RESTDataSource to create CoderAPI and export it as a module

```
class CoderAPI extends RESTDataSource {

}
module.exports = CoderAPI;
```

- ☐ 6. Define a constructor method, call super(), and define the baseURL

```
constructor() {
  super();
  this.baseURL = 'http://localhost:3001';
}
```

- ☐ 7. Define a method named getAllCoders inside the CoderAPI class

```
getAllCoders() {
  return this.get('coders');
}
```

- ☐ 8. Define a getActivities method

```
getActivities(){
  return this.get(`activities`);
}
```

Lab 5: Creating Resolvers

Next we need to create Resolvers. A resolver function populates the data for a field in your schema.

☐ 1. Create a file named resolvers.js in server/src

☐ 2. Declare a resolvers constant and export it

```
const resolvers = {};

module.exports = resolvers;
```

☐ 3. Add a Query key to the resolvers object and add a resolver for coders

```
const resolvers = {
  Query: {
    coders: () => {},
}
```

☐ 4. Add 4 parameters to the coders resolver

```
coders: (parent, args, context, info) => {},
```

☐ 5. The only parameter we need to pass to coders is the context, so we can name the first two with underscores (one for the first param, two for the second). This is just a convention. We need to destructure the third to get to the object we need from it, dataSources:

```
coders: (_,__,{dataSources}) => {}
```

☐ 6. Return the results from the codersAPI method

```
coders: (_,__,{dataSources}) => {
  return dataSources.coderAPI.getAllCoders();
}
```

☐ 7. Create a new directory named api-server and initialize an npm repo in it

```
npm init
```

☐ 8. Install json-server in the new directory

```
npm i json-server
```

☐ 9. Create an npm script named start in json-server/package.json

```
"start":"json-server --watch db.json --port 3001"
```

☐ 10. Make a file named db.json in json-server and populate it with some coder data (or grab the sample file from this course's repo).

☐ 11. Run npm start to start the json-server.
☐ 12. Once it's installed, go to http://localhost:3001 to confirm that it's working and that your endpoints look good
☐ 13. In server/src/index.js, import the resolvers and replace the mocks object with the resolvers.

```
const resolvers = require('./resolvers');

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

☐ 14. Import the API

```
const CoderAPI = require('./datasources/coder-api');
```

☐ 15. Add dataSources to the server.

```
const server = new ApolloServer({
  typeDefs,
  resolvers,
  dataSources: () => {
    return (
      {coderAPI: new CoderAPI()}
      );
    };
}
```

☐ 16. Return to Apollo Studio and try querying your "remote" data to see if it works!

Lab 6: Making a mutation

Mutations are set up similarly to queries, but they use the Mutation type, the Mutation object for the resolver, and the useMutation hook in the client.

☐ 1. Add a mutation to add a coder to the schema. We'll make the mutation return an object, which we'll call AddCoderResponse.

```
type Mutation {
    addCoder(name: String!, description: String!):
AddCoderResponse!
  }

type AddCoderResponse {
  "Similar to HTTP status code, represents the status
of the mutation"
  code: Int!
  "Indicates whether the mutation was successful"
  success: Boolean!
  "Human-readable message for the UI"
  message: String!
  "New coder after a successful mutation"
  coder: Coder
}
```

☐ 2. Add a new object in resolvers.js for mutations and define addCoder.

```
Mutation: {
    // adds a new coder to the database
    addCoder: async (_, { name, description }, {
dataSources }) => {
      try {
        const coder = await
dataSources.coderAPI.addCoder(name, description);
        return {
          code: 200,
          success: true,
          message: `Successfully added coder ${name}`,
          coder,
        };
      } catch (err) {
        console.log(err);
        return {
          code: err.extensions.response.status,
          success: false,
          message: err.extensions.response.body,
          coder: null,
        };
      }
    },
  },
```

☐ 3. Add a function to coder-api for adding coders.

```
addCoder(name, description) {
  return this.post(`coders`, { name, description });
}
```

☐ 4. Open Apollo Explorer and test out addCoder

Lab 7: Making mutations from the client

In this lab, you'll create a form for adding coders and connect it the server

☐ 1. Make a new file named AddCoderForm.jsx and add a form to it with a stateful object that tracks the form fields.

```
import { useState } from 'react';

function AddCoderForm() {
  const [newCoder, setNewCoder] = useState({
    name: '',
    description: '',
  });

  return (
    <div className="add-coder">
      <h2>Add Coder</h2>
      <form>
        <input
          type="text"
          placeholder="Coder Name"
          value={newCoder.name}
          onChange={(e) => {
            setNewCoder({ ...newCoder, name:
e.target.value });
          }}
        />
        <input
          type="text"
          placeholder="Coder Description"
          value={newCoder.description}
          onChange={(e) => {
            setNewCoder({ ...newCoder, description:
e.target.value });
          }}
        />
        <button type="submit">Add Coder</button>
      </form>
    </div>
  );
}

export default AddCoderForm;
```

☐ 2. Define the ADD_CODER query

```
const ADD_CODER = gql`
  mutation AddCoderMutation($name: String!,
$description: String!) {
    addCoder(name: $name, description: $description) {
      code
```

```
      success
      message
      coder {
        name
        description
      }
    }
  }
`;
```

☐ 3. Import useMutation and gql

```
import { gql, useMutation } from '@apollo/client';
```

☐ 4. The useMutation hook returns an array. The first element of the returned array is a function. Call useMutation, passing in the Mutation and the variables.

```
  const [addCoder] = useMutation(ADD_CODER, {
    variables: { name: newCoder.name, description:
newCoder.description },
    // to observe what the mutation response returns
    onCompleted: (data) => {
      console.log(data);
    },
  });
```

☐ 5. Import the AddCodersForm into App and try it out.

You should see a success message in the console, and if you refresh the page, you'll see the new record. We don't want the user to have to refresh the page to see new data, though.

☐ 6. To cause useMutation to refetch the data after mutation, add a refetchQueries object to the options object you pass to it:

```
refetchQueries: [{ query: CODERS_QUERY }],
```

☐ 7. Import CODERS_QUERY from App:

```
import { CODERS_QUERY } from '../App';
```

☐ 8. Test it out!