A PDF Reference for

The Modern GraphQL Bootcamp

Version 1.0

Taught by Andrew Mead

Table of Contents

Section 1: Course Overview

In this first section, we're going to explore what you'll learn in this course. You'll learn why GraphQL is a great language to know, and you'll explore what's covered in the class.

There are no individual lecture notes for this first section. These lectures are important, but they don't cover any JavaScript features.

Section 2: GraphQL Basics: Schemas and Queries

Lesson 1: Section Intro

In this section, you're going to start exploring GraphQL. You'll learn how to query data via the query operation, and you'll learn how to change data using the mutation operation.

Lesson 2: What is a Graph?

In this lesson, you're going to learn how to think in graphs. This will set us up to work with a GraphQL API in the next lesson.

This is a presentation video. Please refer to the video for the visualization.

Lesson 3: GraphQL Queries

In this video, you're going to make your very first GraphQL query. Queries are used to fetch data from a GraphQL API.

GraphQL Query

There are three operations that can be performed in GraphQL. The query, the mutation, and the subscription. The query allows you to fetch data. The mutation allows you to change data. The subscription allows you to watch data for changes. In this section, we'll be focusing on the query operation.

The query syntax allows the client to describe exactly what data it would like back. To start the process, we provide the operation name followed by a selection set. Below is an example of an empty query operation.

```
query {

}
```

The query above isn't requesting any data, which is invalid. There's no need to perform a query if you don't want any data back, but we can change that by specifying what data we want.

The following valid query requests a single field `course`.

```
query {
  course
}
```

The result of this query would be the following JSON.

```
{
  "data": {
    "course": "GraphQL"
  }
}
```

All of the response data can be found on the "data" property. Remember, in GraphQL the client specifies the exact data it needs. That means the client knows the response structure ahead of time. We've asked for a single field "course" and got a single "course" property back on our "data" object.

GraphQL Schema

Another great feature of GraphQL is that it's self-documenting. GraphQL APIs expose a schema that describes exactly what operations could be performed on the API and what data could be requested. This prevents the need for manual documentation, and it enables tools like GraphQL Playground to validate the structure of your operations before ever sending the operation to the server.

Documentation Links

- Queries
- The Anatomy of a GraphQL Query

Lesson 4: Nested GraphQL Queries

In this lesson, you're going to explore nested GraphQL queries. This will allow you to work with complex data structures like objects and arrays of objects.

Queries and Custom Types

GraphQL allows you to define custom types specific to your application. A custom type is nothing more than a template for an object. Imagine you were building an online shopping platform. You'd likely have a "Product" custom type with a few fields that store information about a product. This could include "name" for the product name, "price" for the product price, "inStock" to track if the product is in stock, and anything else you might need. You'd likely have other custom types as well, such as "Customer" to store customer specific information like their name, address, shipping preferences, etc.

GraphQL is a strongly typed language. All the data you have access to is of a specific type which can be found in the schema. A field like "course" might resolve to a string value, while a field like "me" might resolve to a custom type such as a "User" type.

When querying for a custom type, the resolved JSON value is an object. Your query needs to explicitly list out all the fields it wants from the object. Imagine I had a "User" type with the fields "id", "name", and "email". Let's say that the resolved value for the "me" query is of the "User" type. The following query would be invalid because it's not explicit about what fields it wants from "me".

```
query {
  course
  me
}
```

To fix that query, a selection set needs to be provided for the "me" query. This is done after the query name in a set of curly braces. The following valid query defines a selection set for "me" by requesting the "id", "name", and "email" of the user.

```
query {
  course
  me {
    id
    name
    email
  }
}
```

Once again, the response structure will match up with the query structure. Below is the JSON response.

```
{
  "data": {
    "course": "GraphQL",
    "me": {
      "id": "affb67e1-0d3b-40ae-b58d-7935a5506012",
      "name": "Andrew Mead",
      "email": "andrew@example.com"
    }
  }
}
```

Queries and Arrays of Custom Types

GraphQL supports arrays of custom types, which is nothing more than an array of objects. This allows you to model data like:

1. A list of all recent posts
2. A list of all comments for a specific post
3. A list of all users who've signed up for a given application

The "users" query below is an example of a query that resolves to an array of objects. Each object in the array is of the `User` type. It's still necessary to provide a selection set where you list out what fields you want from each user. This example query is asking for the "id" and "name" of each user.

```
query {
  users {
    id
    name
  }
}
```

Once again, the structure of the query dictates the fields in the response JSON. There's a single "users" property on "data". It's an array of objects where each object represents a different user. Each object contains the "id" and "name" of that specific user. The "email" field wasn't listed in the query, so it won't be found in the response.

```
{
  "data": {
    "users": [
      {
        "id": "affb67e1-0d3b-40ae-b58d-7935a5506012",
        "name": "Andrew"
      },
      {
        "id": "c2d96c35-e28a-4bfd-b075-3427bc09426a",
        "name": "Sarah"
      },
      {
        "id": "8b457570-06fa-4195-a712-d24bbc1b5294",
        "name": "Michael"
      }
    ]
  }
}
```

Documentation Links

- Queries
- The Anatomy of a GraphQL Query

Lesson 5: Setting up Babel

In this lesson, you're going to set up Babel. This will allow you to take advantage of all the latest JavaScript features such as the ES6 import/export syntax.

Configuring Babel for Node.js

Configuring Node.js to use Babel requires a few steps. First up, you'll need to install the necessary npm modules. Both `babel-cli` and `babel-preset-env` are required. You can install both with the following command.

```
npm install babel-cli@6.26.0 babel-preset-env@1.7.0
```

From there, you'll need to configure Babel by creating a `.babelrc` file in the root of your project. In here, you need to tell Babel that it should use the `babel-preset-env` preset that was installed. The following JSON gets that done.

```
{
  "presets": [
    "env"
  ]
}
```

Next, you can set up a start script that allows you to run the Node app after passing it though Babel.

```
{
  "scripts": {
    "start": "babel-node src/index.js"
  }
}
```

Run the script with `npm run start` and now you can take advantage of the import/export syntax in Node.js!

Documentation Links

- Babel
- Try Babel out

Lesson 6: ES6 Import/Export

In this video, you're going to learn about JavaScript modules. JavaScript modules make it easy to share code between files, allowing us to create more advanced applications without messy code.

Named Exports and Imports

There are two sides to the module system. There's a file that sets up some exports, and then there are one or more files that import the exported functionality. You might have a file called add.js that exports a function for adding numbers. Any file in the program that needs to add numbers can then import the exported function from add.js.

It's important to note that each file, which could also be referred to as a module, has its own scope. That means exporting and importing is the only way to share code.

The file below demonstrates this by setting up two functions and exporting them using the `export` statement. It exports a function for adding two numbers and a function for subtracting two numbers. You can export as many things as you need; in this case, two exports is all that's necessary.

```
// utilities.js
const add = (a, b) => a + b
const subtract = (a, b) => a - b

export { add, subtract }
```

The `import` statement is used to grab exports from another file or module. The above file already sets up some exports, so these can be used by another file as shown in the code below. In this case, I'm assuming that both files are in the same directory.

```
// index.js
import { add, subtract } from './utilities'

console.log(add(32, 1)) // Will print: 33
console.log(subtractadd(32, 1)) // Will print: 31
```

The Default Export and Import

A module can export as many named exports as needed. A module can also choose to set up a single default export. This is completely optional, but it provides a nice way to set up imports and exports if a file exports just one thing.

The code below now exports three functions. `add` and `subtract` are set up as named exports. `square` is set up as the default export.

```
// utilities.js
const add = (a, b) => a + b
const subtract = (a, b) => a - b
const square = (x) => x * x

export { add, subtract, square as default }
```

Grabbing a default export requires a small change to the import statement. Instead of naming “square” in the curly braces, it gets set up just before the curly braces. You can also name it anything you like. It’s not the name that links it to the correct export; it’s the fact that it’s the default export.

The code below grabs the `square` export and stores it as `otherSquare`. It also grabs the named export `add`.

```
// index.js
import otherSquare, { add } from './utilities'

console.log(add(32, 1)) // Will print: 33
console.log(otherSquare(10)) // Will print: 100
```

Documentation Links

- Import statement
- Export statement

Lesson 7: Creating Your Own GraphQL API

In this lesson, you’re going to set up your very own GraphQL API!

Picking a Server: graphql-yoga

GraphQL is nothing more than a specification for how GraphQL works. It’s not an implementation. Before you can use GraphQL, you need to pick an implementation that works with the language/stack you’re using. With Node.js, graphql-yoga is a fantastic option. It supports the entire GraphQL spec, and it couldn’t be easier to use.

You can install `graphql-yoga` with the following command.

```
npm install graphql-yoga@1.14.10
```

Now you can use `graphql-yoga` to create your GraphQL API server. Here's a barebone implementation.

```
import { GraphQLServer } from 'graphql-yoga'

// Type definitions (schema)
const typeDefs = `
    type Query {
        hello: String!
    }
`

// Resolvers
const resolvers = {
    Query: {
        hello() {
            return 'This is my first query!'
        }
    }
}

const server = new GraphQLServer({
    typeDefs,
    resolvers
})

server.start(() => {
    console.log('The server is up!')
})
```

That example contains two things you haven't seen set. The first is the schema. The second are the resolvers.

Schema

In the implementation above, the `typeDefs` variable is where you define the types that make up your application schema. This is where you'll define all the operations you want the server to support. It's also where you'll define any custom types your application needs. For now, the only type defined is the built-in `Query` type.

All your queries need to be defined in the `Query` type. The query definition is made up of two parts, a query name and query type.

The query name can be anything you like, but it should be descriptive. “posts” would be a great name for a query that returns a list of posts. “me” would be a great name for a query that returns your profile information. In the case above, the query “hello” is used for this initial example.

The second piece to the query definition is the type. What type of data is coming back? For the “hello” query, the definition says that a string will be the resolved value. You’ll learn more about the built-in types and type definitions in a few lessons.

Resolvers

While `typeDefs` allows you to define the operations, it’s the resolvers that contain the code that runs when an operation is executed. Notice that the `resolvers` object mirrors the structure of `typeDefs`. The `resolvers` object contains a single property named `Query`. The `Query` object contains a single property named `hello`. The value for `hello` is a function, and this function runs when the “hello” query is executed.

The return value for your resolver methods needs to match up with the types in your type definitions. The type definition for “hello” is a string, so the resolver method for “hello” must return a string. In the example above, “This is my first query!” is the string that returns.

GraphQl Playground

GraphQL Playground is built right into `graphql-yoga`. After starting up the server, you can access it at its default location, `localhost:4000`. From there, you can use the following query to fetch your data.

```
query {
  hello
}
```

Below is the expected JSON response.

```
{
  "data": {
    "hello": "This is my first query!"
  }
}
```

Documentation Links

- Graphql-yoga library
- GraphQL specification

Lesson 8: GraphQL Scalar Types

In this lesson, you're going to learn about the five scalar types supported by GraphQL.

Scalar Types

A scalar value is a single discrete value. A scalar type is a type that stores a single value. There are five built-in scalar types in GraphQL.

1. ID - Used to store unique identifier
2. String - Used to store string data as UTF-8 characters
3. Boolean - Used to store true or false
4. Int - Used to store 32-bit integer numbers
5. Float - Used to store double-precision floating-point numbers

The opposite of a scalar value is a non-scalar value. This would include arrays and objects, which are collection of values as opposed to a single discrete value.

Here's an example of a type definition that uses all five scalar types.

```
type Query {
    title: String!
    price: Float!
    releaseYear: Int
    rating: Float
    inStock: Boolean!
}
```

There are five queries defined, which means that a few new resolvers are necessary.

```
const resolvers = {
    Query: {
        title() {
            return 'The War of Art'
        },
        price() {
            return 12.99
        },
        releaseYear() {
            return null
        },
        rating() {
            return 5
        },
        inStock() {
            return true
        }
    }
}
```

Nullable & Non-Nullable Types

In the above type definition, `rating` has a type of `Float`, while `price` has a type of `Float!`. The first type without `!` is nullable while the second type with `!` is non-nullable. This is the difference between a nullable and a non-nullable value.

A nullable value like `rating` could return either a floating point number or `null`. A non-nullable value like `price` will always return a floating point number. `price` cannot return null.

Documentation Links

- Scalar types

Lesson 9: Live Reload for GraphQL-Yoga

In this lesson, you're going to set up nodemon with Node and Babel. Nodemon will automatically restart your server after you make any changes, allowing you to speed up development.

Setting up Nodemon

Nodemon can be installed as an npm module using the following command.

```
npm install nodemon@1.17.5 --save-dev
```

From there, a small change to the start script will allow us to use nodemon with our existing setup.

```
{
  "scripts": {
    "start": "nodemon src/index.js --exec babel-node"
  }
}
```

Documentation Links

- Nodemon documentation

Lesson 10: Creating Custom Types

In this lesson, you'll be creating your first custom object types. This will allow you to model your application data and represent it in the application schema.

Creating a Custom Object Type

In the previous lesson, you set up five queries related to a product. This included title, price, releaseYear, rating, and inStock. This is a good start, but you don't want an entirely new query for every field of data needed. To address this, you'll want to have queries that resolve to a custom type instead of a scalar type.

Creating an object type requires a new type definition in `typeDefs`. An empty type definition for a product might look like this.

```
type Product {

}
```

All the fields for a "Product" get defined in the curly braces. The syntax for this looks very similar to how queries are defined in the "Query" type. For each field, you pick a field name and set up the type definition for each. Below, the same five fields from the previous lesson have been added onto the object type with the addition of an `id` field to store the unique identifier for each product.

```
type Product {
  id: ID!
  title: String!
  price: Float!
  releaseYear: Int
  rating: Float
  inStock: Boolean!
}
```

Using an Object Type

This can now be used as the type for a query. Here's an example `product` query along with its resolver method.

```
type Query {
  product: Product!
}
```

The `product` query has `Product!` as its type. Just like with scalar types, `!` signifies that this is non-nullable. The query will always send back an object that matches up with the `User` type. Below, the `product` resolver does just that by returning an object with all the necessary properties.

```
const resolvers = {
    Query: {
        product() {
            return {
              id: '123',
              title: 'Watch',
              price: 39.99,
              rating: 4.8,
              inStock: false
            }
        }
    }
}
```

Querying an Object Type

A query could now grab whatever it needed from the product. The query below grabs three fields from the product.

```
query {
  product {
    id
    title
    rating
  }
}
```

Below is the response to the query with the three properties requested.

```
{
  "data": {
    "product": {
      "id": "123",
      "title": "Watch",
      "rating": 4.8
    }
  }
}
```

Documentation Links

- Constructing types

Lesson 11: Operation Arguments

In this lesson, you're going to explore operation arguments. This allows you to pass data from the client to the GraphQL server.

Defining Operation Arguments

Operation arguments are similar to function arguments. It's a way to pass data along with your GraphQL query. Imagine you want to get all posts that contain "GraphQL" in the title. You need a way to pass that data along with your request so the server can query the database and filter just the posts that match. This would be done with operation arguments.

In GraphQL, an operation needs to explicitly define the arguments it accepts. Arguments can be of any type including all the scalar types as well as any object types you create. Arguments can also be required or optional.

In the example below, the `greeting` query has been configured to accept a single argument. Each argument has a name as well as a type. In this case, the argument name is "name" and it expects a string to be provided as the argument value.

```
type Query {
        greeting(name: String): String!
    }
```

With the change to `typeDefs` complete, the resolver function for the `greeting` query can now access the argument. All provided arguments are accessible via the second argument passed to the resolve method which is typically named `"args"`. This argument is an object with a property for every argument provided. The resolver below checks if the name was provided, and returns either a custom or generic greeting.

```
const resolvers = {
    Query: {
        greeting(parent, args, ctx, info) {
            if (args.name) {
                return `Hello, ${args.name}!`
            } else {
                return 'Hello!'
            }
        }
    }
}
```

Note that the `name` argument is currently optional. This is because the type is `String` which is nullable. The argument could be set up as required by changing the type to `String!` as shown below.

```
type Query {
        greeting(name: String!): String!
    }
```

Passing Operation Arguments

Now GraphQL queries can pass in `name` when querying the GraphQL API. The example below uses the string `"Andrew"` as the value for the `name` argument.

```
query {
  greeting(name: "Andrew")
}
```

The response containing the customized greeting can be seen in the JSON below.

```
{
  "data": {
    "product": "Hello, Andrew!"
  }
}
```

Documentation Links

- Operation arguments

Lesson 12: Working with Arrays: Part I

In this lesson, you'll learn how to work with arrays in GraphQL. This lesson covers arrays where the array elements are scalar values such as string, numbers, or booleans.

Array Type Definition

Remember that GraphQL is a strongly typed language. That means if a fields value should be an array, the type definition needs to reflect that.

In the example below, a `grades` query is set up to return an array of integers. The import part of that code being the type definition `[Int!]!`.

```
type Query {
  grades: [Int!]!
}
```

There are two parts to the type definition, the first being `[]!`. This signifies the value is going to be a non-nullable array. The inner type definition defines the type for the elements in the array. In the case above, that would be `Int!` which says that any array element will be a non-nullable integer.

From there, the resolver function for `grades` can return an array of integers as shown in the example below.

```
const resolvers = {
  Query: {
    grades() {
      return [
        99,
        74,
        100,
        3
      ]
    }
  }
}
```

Querying for an Array

Querying for an array of scalar values is just like querying for a scalar value. The below query would get the array of numbers back.

```
query {
  grades
}
```

The JSON response is shown below.

```
{
  "data": {
    "grades": [99, 74, 100, 3]
  }
}
```

Lesson 13: Working with Arrays: Part II

In this lesson, you're going to continue learning about arrays. Specifically, you'll learn how to work with arrays of objects, such as an array of posts or an array of users.

Array of Objects Type Definition

The type definition for an array of objects is almost identical to the type definition for an array of scalar values. The schema below sets up a `users` query which returns an array of `User` objects.

```
type Query {
  users: [User!]!
}

type User {
  id: ID!
  name: String!
  age: Int
}
```

The resolver function is now responsible for returning an array of objects, where each object matches the `User` type. That means each object needs an `id` and `name` property, with an optional `age` property being possible as well.

```
const resolvers = {
  Query: {
    users() {
      return [{
        id: '1',
        name: 'Jamie'
      }, {
        id: '2',
        name: 'Andrew',
        age: 27
      }, {
        id: '3',
        name: 'Katie'
      }]
    }
  }
}
```

Querying for an Array of Objects

When querying an object type, you've seen that it's necessary to provide a selection set where you describe all the fields you want. The same is true when working with an array of object types. It's necessary to explicitly list out what fields you need from each object in the array.

The following operation queries for the `id` field for each user.

```
query {
  users {
    id
  }
}
```

This second operation queries for all of the available fields for each user.

```
query {
  users {
    id
    name
    age
  }
}
```

The JSON response is shown below.

```
{
  "data": {
    "users": [
      {
        "id": "1",
        "name": "Jamie",
        "age": null
      },
      {
        "id": "2",
        "name": "Andrew",
        "age": 27
      },
      {
        "id": "3",
        "name": "Katie",
        "age": null
      }
    ]
  }
}
```


Lesson 14: Relational Data: Basics

In this lesson, you'll start to explore relational data in GraphQL. Setting up relationships between your object types allows you to query based on those relationships, which is one of the best features of GraphQL.

Setting up Associations

Associations get set up in the object type definition. The type definition below sets up both `User` and `Post`, where every post has an `author` field that links to a user.

```
type User {
  id: ID!
  name: String!
  email: String!
  age: Int
}

type Post {
  id: ID!
  title: String!
  body: String!
  published: Boolean!
  author: User!
}
```

This relationship requires a new resolver method. This new method is responsible for returning the user for a given post. Notice that the `author` method below lives on a new `Post` property. For associations, the root property name needs to match up with the object type name. The method name needs to match up with the new field name.

The post data is provided via the first argument, which is typically named `parent`. This means `parent.author` is where the author id can be accessed and used to determine which user is the author for the post.

```
const resolvers = {
  // Query object hidden for brevity
  Post: {
    author(parent, args, ctx, info) {
      return users.find((user) => {
        return user.id === parent.author
      })
    }
  }
}
```

Querying Relational Data

In your queries, you can now request the author for a post. In the example below, the `id` and `title` fields are requested for every post. Along with `id` and `title`, the author's name has also been requested using the new `author` field on `Post`.

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

Lesson 15: Relational Data: Arrays

In this lesson, you'll continue exploring relational data in GraphQL by setting up an array-based association.

Setting up an Array-Based Associations

A link already exists for getting the author of a given post. The goal now is to create a link that lets you get all the posts that belong to a given user.

The example below does this by adding a `posts` property on to `User` with the type of `[Post!]!`.

```
type User {
  id: ID!
  name: String!
  email: String!
  age: Int
  posts: [Post!]!
}

type Post {
  id: ID!
  title: String!
  body: String!
  published: Boolean!
  author: User!
}
```

Once again, a new resolver method is required. The job of this method is to return an array of posts that belong to the given user. The user's data is accessible via `parent`, so the function can use the user's id to determine if a given post belongs to them.

```
const resolvers = {
  // Query and Post objects hidden for brevity
  User: {
      posts(parent, args, ctx, info) {
        return posts.filter((post) => {
          return post.author === parent.id
        })
      }
    }
}
```

Querying Array-Based Relational Data

Queries for that GraphQL API can now request the `posts` field on any user. The example below gets a couple of scalar values for each user. It also fetches all the posts for that user, getting the `id` and `title` for each and every post.

```
query {
  users {
    id
    name
    posts {
      id
      title
    }
  }
}
```

Lesson 16: Comment Challenge: Part I

In this lesson, you're going to start a three-part challenge series designed to test and reinforce what you've learned so far in the course.

There are no notes as no new language features were explored.

Lesson 17: Comment Challenge: Part II

In this lesson, you're going to continue implementing comments by setting up associations between the comment and the post.

There are no notes as no new language features were explored.

Lesson 18: Comment Challenge: Part III

In this lesson, you're going to wrap up this challenge series by setting up associations between the comment and the user.

There are no notes as no new language features were explored.

Section 3: GraphQL Basics: Mutations

Lesson 1: Section Intro

In this section, you'll be exploring GraphQL muations. Mutations let you create, update, and delete data using GraphQL. By the end of this section, you'll have everything you need to perform all the CRUD operations (create, read, update, delete).

Lesson 2: Creating Data with Mutations: Part I

In this lesson, you'll create and execute your very first mutation. This will allow clients to perform operations like signing up a new user or creating a new post.

Defining a Mutation

Mutation, like queries, must be defined in your application schema. Defining a mutation is similar to defining a query. The major difference is that queries are defined on the `Query` type while mutations are defined on the `Mutation` type.

The example below defines a mutation for creating a user called `createUser`, which accepts a set of arguments. The `name` and `email` arguments are set up as required by being non-nullable. The `age` argument is set up to be optional. The value returned will be the `User` that was created.

```
type Mutation {
    createUser(name: String!, email: String!, age: Int): User!
}
```

It's necessary to define a resolver for mutations. Resolver methods for mutations live on the `Mutation` property of the `resolvers` object. The example below sets up a `createUser` method which matches up with `createUser` from the mutation definition above. Resolver

methods for mutations get called with the same set of arguments as resolver methods for queries.

```
import uuidv4 from 'uuid/v4'

const resolvers = {
    // Other properties hidden for brevity
    Mutation: {
        createUser(parent, args, ctx, info) {
            const emailTaken = users.some((user) => user.email ===
args.email)

            if (emailTaken) {
                throw new Error('Email taken')
            }

            const user = {
                id: uuidv4(),
                name: args.name,
                email: args.email,
                age: args.age
            }

            users.push(user)

            return user
        }
    }
}
```

Performing a Mutation

When performing a mutation, the `mutation` operation must be used. The example below runs the `createUser` mutation passing in values for both `name` and `email`. It also provides a selection set asking for the four scalar fields on `User`.

```
mutation {
  createUser(name: "Andrew", email: "testing@example.com"){
    id
    name
    email
    age
  }
}
```

The JSON response is shown below.

```
{
  "data": {
    "createUser": {
      "id": "8257f14f-80b0-4313-ad35-9047e3b5f851",
      "name": "Andrew",
      "email": "testing2@example.com",
      "age": null
    }
  }
}
```

If the same operation was executed two times in a row, the second operation would fail because the email would already be taken. The JSON response below shows what would come back from a `createUser` mutation where the provided email is already registered.

```
{
  "data": null,
  "errors": [
    {
      "message": "Email taken",
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ],
      "path": [
        "createUser"
      ]
    }
  ]
}
```

Documentation Links

- Mutations

Lesson 3: Creating Data with Mutations: Part II

In this lesson, you'll continue to explore mutations. You'll set up mutations for types that have required associations with other types.

Mutations and Data Associations

When creating a new post, `title`, `body`, and `published` are all required arguments. Additionally, `author` is also a required argument. The value for `author` would be the id of the author that created the post.

```
type Mutation {
        createPost(title: String!, body: String!, published: Boolean!,
author: ID!): Post!
    }
```

Below, the resolver method for `createPost` verifies that there is a user whose id matches up with the value for the author argument. If no user is found with that id, an error is thrown and the post is not created. If a user is found, the post is created and pushed onto the `posts` array.

```
const resolvers = {
    // Other properties hidden for brevity
    Mutation: {
        createPost(parent, args, ctx, info) {
            const userExists = users.some((user) => user.id === args.author)

            if (!userExists) {
                throw new Error('User not found')
            }

            const post = {
                id: uuidv4(),
                title: args.title,
                body: args.body,
                published: args.published,
                author: args.author
            }

            posts.push(post)

            return post
        }
    }
}
```

With the type definition and resolver method in place, `createPost` can be executed. The operation below creates a new post and associates it with the user whose id is `"2"`.

```
mutation {
  createPost(
    title:"My new post",
    body:"",
    published:false,
    author:"2"
  ){
    id
    title
    body
    published
    author {
      name
    }
    comments {
      id
    }
  }
}
```

The JSON response for the above operation is shown below.

```
{
  "data": {
    "createPost": {
      "id": "18735d13-6b96-4eba-9b85-81da86853231",
      "title": "My new post",
      "body": "",
      "published": false,
      "author": {
        "name": "Sarah"
      },
      "comments": []
    }
  }
}
```

Lesson 4: The Object Spread Operator with Node.js

In this lesson, you'll be configuring babel to support the object spread opeartor. You'll also learn why this opeartor is useful and how it can be used to clean up our resolver methods.

The Object Spread Operator

To support this operation, a new Babel plugin must be installed and configured. The npm command below installs the module necessary.

```
npm install babel-plugin-transform-object-rest-spread@6.26.0
```

In `.babelrc`, the plugin needs to be added to the `plugins` array as shown below. This will ensure that babel converts all usages of the spread operator into more universally supported JavaScript.

```
{
    "presets": [
        "env"
    ],
    "plugins": [
        "transform-object-rest-spread"
    ]
}
```

The object spread operator makes it easy to copy properties from one object over to another.

The example below defines two objects, `one` and `two`. `one` has a `name` and `age` property while `two` has an `age` and `location` property. `two` also gets all the properties from `one` by "spreading it out" using the following syntax `...one`. The final value for `two` is an object that contains `age`, `location`, and `name`.

```
const one = {
    name: 'Andrew',
    age: 27
}

const two = {
    age: 25,
    location: 'Philadelphia',
    ...one
}

console.log(two)

// Will print:
// {
//   age: 27,
//   location: "Philadelphia",
//   name: "Andrew"
// }
```

Documentation Links

- Object spread operator

Lesson 5: The Input Type

In this lesson, you're going to learn about the input type. This allows you to set up arguments as objects, giving you more control over how your operations function.

The Input Type

Until now, you've only used scalar values for operation arguments, though arguments don't have to be simple scalar values.

The example below shows how `createUser` can be modified to accept just a single argument. The mutation still needs the `name`, `email`, and optional `age` in order to function, but now those get added as properties on `data`.

```
type Mutation {
    createUser(data: CreateUserInput!): User!
}

input CreateUserInput {
    name: String!
    email: String!
    age: Int
}
```

The resolver method for `createUser` can now access all three fields on `args.data`.

```
createUser(parent, args, ctx, info) {
    // Can access name, email, and age on args.data
},
```

The code below runs the new `createUser` operation. An object is provided as the value for the `data` argument, and that object has properties that match up with the `CreateUserInput` type definition above.

```
mutation {
  createUser(
    data:{
      name:"Jess",
      email:"jess@example.com"
    }
  ){
    id
  }
}
```

Documentation Links

- Input types

Lesson 6: Deleting Data with Mutations: Part I

In this lesson, you'll learn how to use mutations to remove data from a GraphQL back-end.

Deleting Data with Mutations

Defining a mutation for deleting data is no different than defining a mutation for creating data. The code below defines a `deleteUser` mutation for deleting a user. Mutations for deleting data typically require a single argument, the id of the data that should be removed. In the case of `deleteUser` it accepts a single argument, `id`, whose value should be set to the id of the user to delete.

```
type Mutation {
    deleteUser(id: ID!): User!
}
```

When deleting data, it's important to clean up any associated data as well. Every user in this application has a set of posts and comments they've written. That means it's necessary to delete:

1. The user
2. Any posts written by the user
3. All comments on the deleted posts (regardless of which users created the comments)
4. All comments left by the user on any other posts

The resolver below takes care of all four of these considerations.

```
const resolvers = {
    // Other properties hidden for brevity
    Mutation: {
        deleteUser(parent, args, ctx, info) {
            const userIndex = users.findIndex((user) => user.id === args.id)

            if (userIndex === -1) {
                throw new Error('User not found')
            }

            const deletedUsers = users.splice(userIndex, 1)

            posts = posts.filter((post) => {
                const match = post.author === args.id

                if (match) {
                    comments = comments.filter((comment) => comment.post !==
post.id)
                }

                return !match
            })
            comments = comments.filter((comment) => comment.author !==
args.id)

            return deletedUsers[0]
        }
    }
}
```

Lesson 7: Deleting Data with Mutations: : Part II

In this lesson, it's going to be up to you to create mutations for deleting posts and comments.

There are no notes for this challenge video as no new language features were explored.

Lesson 8: A Pro GraphQL Project Structure: Part I

In this lesson, you're going to start refactoring your application's architecture. The goal is to break up `index.js` into smaller files that are easier to manage and easier to scale.

Context

When working with large applications, it's not realistic to keep everything in a single file. Code like the application resolvers will be spread across several different files. That can make sharing things like database connections a bit tricky.

Context is designed to address this. When creating the GraphQL server, the context property can be set. Values set on context object are passed into all resolver methods via the third `ctx` argument.

In the code below, the context object is given a single property db. `db` contains the arrays of users, posts, and comments.

```
const server = new GraphQLServer({
    typeDefs: './src/schema.graphql',
    resolvers,
    context: {
        db
    }
})
```

That context value can now be accessed from any resolver method. The `users` resolver method below destructures the context grabbing `db`. It then uses `db.users` to access the saved users.

```
const resolvers = {
    // Other properties hidden for brevity
    Query: {
        users(parent, args, { db }, info) {
            if (!args.query) {
                return db.users
            }

            return db.users.filter((user) => {
                return
user.name.toLowerCase().includes(args.query.toLowerCase())
            })
        },
    }
}
```

Documentation Links

- Context

Lesson 9: A Pro GraphQL Project Structure: Part II

In this lesson, you'll wrap up refactoring the application by splitting the application resolvers up across several smaller files.

There are no notes for this video as no new language features were explored.

Lesson 10: update 1

In this lesson, you'll learn how to create mutations that update data on a GraphQL back-end. This is the last missing piece from being able to perform the basic CRUD operations (create, read, update, delete).

Updating Data with Mutations

The `updateUser` mutation below requires two arguments. The first, `id`, stores the id of the user that should be updated. The second, `data`, contains all of the data that should be updated. Notice that all of the fields on `UpdateUserInput` are nullable. It's not required that you update any particular set of fields. It's up to the client to pass in the fields they actually want to change.

```
type Mutation {
    updateUser(id: ID!, data: UpdateUserInput!): User!
}

input UpdateUserInput {
    name: String
    email: String
    age: Int
}
```

Like with `deleteUser`, the resolver for `updateUser` needs to verify that the user that's being updated actually exists. If they do exist, the properties provided on `data` get used to update the saved user.

```
updateUser(parent, args, { db }, info) {
        const { id, data } = args
        const user = db.users.find((user) => user.id === id)

        if (!user) {
            throw new Error('User not found')
        }

        if (typeof data.email === 'string') {
            const emailTaken = db.users.some((user) => user.email === 
data.email)

            if (emailTaken) {
                throw new Error('Email taken')
            }

            user.email = data.email
        }

        if (typeof data.name === 'string') {
            user.name = data.name
        }

        if (typeof data.age !== 'undefined') {
            user.age = data.age
        }

        return user
    },
```

Lesson 11: Updating Data with Mutations: Part II

In this lesson, it's up to you to create mutations for updating posts and comments.

There are no notes for this challenge video as no new language features were explored.

Section 4: GraphQL Basics: Subscriptions

Lesson 1: Section Intro

In this section, you'll be exploring GraphQL subscriptions. Subscriptions give clients a way to subscribe to data changes and get notified by the server when data changes.

Subscriptions make it possible to create real-time applications where the client renders data changes in real-time.

Lesson 2: GraphQL Subscription Basics

In this lesson, you'll create your first subscription. The goal here is to explore the basic syntax used for subscriptions and actually get some real-time data sent from the server to the client.

Setting up the PubSub Utility

Setting up a subscription on the server requires the use of the `PubSub` utility from `graphql-yoga`. This is used to both create the subscription and to publish data to it. It's best to add the `PubSub` onto the servers `context` object as shown below.

```
import { GraphQLServer, PubSub } from 'graphql-yoga'
import db from './db'
import Query from './resolvers/Query'
import Mutation from './resolvers/Mutation'
import Subscription from './resolvers/Subscription'
import User from './resolvers/User'
import Post from './resolvers/Post'
import Comment from './resolvers/Comment'

const pubsub = new PubSub()

const server = new GraphQLServer({
    typeDefs: './src/schema.graphql',
    resolvers: {
        Query,
        Mutation,
        Subscription,
        User,
        Post,
        Comment
    },
    context: {
        db,
        pubsub
    }
})

server.start(() => {
    console.log('The server is up!')
})
```

Creating a Subscription

The type definition for a subscription is similar to the type definitions for queries and mutations. The example below shows the `Subscription` type definition with one subscription setup. The subscription name is `count` and the value that'll be published will be an integer.

Publishing is the act of sending new data from the server to the subscribing client. In the case of this example, it's a number that gets sent back to the client every second.

```
type Subscription {
    count: Int!
}
```

The subscription also needs to be configured on the resolver object. As with queries and mutations, it's best to create a separate file to manage that code.

Below is an example `Subscription` resolver file. The `count` property is set up for the `count` subscription from above. The `subscribe` method on `count` is where the subscription is set up.

The two important things are the call to `pubsub.asyncIterator` and the call to `pubsub.publish`. All subscribe methods need to return a call to `pubsub.asyncIterator` with a channel name. Data can then be sent to the channel by calling `pubsub.publish` with the same channel name as the first argument. The second argument should contain the data to be sent to the subscribing client.

```
const Subscription = {
    count: {
        subscribe(parent, args, { pubsub }, info) {
            let count = 0

            setInterval(() => {
                count++
                pubsub.publish('count', {
                    count
                })
            }, 1000)

            return pubsub.asyncIterator('count')
        }
    }
}

export { Subscription as default }
```

Subscribing on the Client

Below is the example operation that would subscribe to the `count` subscription. Subscriptions use web sockets, so the connection will actually stay open until it's closed by either the client or the server. The client won't get any data back until data is published by the server.

```
subscription {
  count
}
```

Below are the three separate payloads that would get sent back. In the case of the counting example above, there's a one-second delay between each of the payloads.

```
{
  "data": {
    "count": 1
  }
}

{
  "data": {
    "count": 2
  }
}

{
  "data": {
    "count": 3
  }
}
```

Lesson 3: Setting up a Comments Subscription

In this lesson, you'll be creating a GraphQL subscription for the blogging application. The subscription will allow clients to subscribe to new comments added on a given post. That would allow the front-end, whether it's a mobile app or website, to render new comments to the visitors in real-time.

Setting up the Comments Subscription

Below is the type definition for the `comment` subscription. It requires a single argument, `postId`, which is the id of the post that should be watched for new comments.

```
type Subscription {
    comment(postId: ID!): Comment!
}
```

The subscribe method for `comment` has a couple of jobs. First up, it verifies that there is indeed a post whose id matches up with the value provided for the `postId` argument. The other job is to return an asyncIterator with a channel name specific to the post. Given the code below, if the value for `postId` was `1`, the channel name would be `comment 1`.

```
const Subscription = {
    comment: {
        subscribe(parent, { postId }, { db, pubsub }, info){
            const post = db.posts.find((post) => post.id === postId &&
post.published)

            if (!post) {
                throw new Error('Post not found')
            }

            return pubsub.asyncIterator(`comment ${postId}`)
        }
    }
}

export { Subscription as default }
```

The subscription is complete, but subscribes won't actually receive data if there are no `pubsub.publish` calls that are publishing data to that channel. To fix this, all that's needed is a call to `pubsub.publish` in the `createComment` mutation resolver where comments actually get created.

```
const Mutation = {
    createComment(parent, args, { db, pubsub }, info) {
        const userExists = db.users.some((user) => user.id ===
args.data.author)
        const postExists = db.posts.some((post) => post.id === args.data.post
&& post.published)

        if (!userExists || !postExists) {
            throw new Error('Unable to find user and post')
        }

        const comment = {
            id: uuidv4(),
            ...args.data
        }

        db.comments.push(comment)
        pubsub.publish(`comment ${args.data.post}`, { comment })

        return comment
    }
}
```

Subscribing to New Comments

The operation below allows the client to subscribe to all new comments for the post with an id of `11`. Notice that the client can request fields on the comment as well as any related data such as information about the comment author.

```
subscription {
  comment(postId:"11"){
    id
    text
    author{
      id
      name
    }
  }
}
```

Below is an example JSON payload that the client would receive when a new comment is added for a post they've subscribed to.

```
{
  "data": {
    "comment": {
      "id": "f6925dbb-8899-4be5-9ab6-365e698931c2",
      "text": "My new comment",
      "author": {
        "id": "1",
        "name": "Andrew"
      }
    }
  }
}
```

Lesson 4: Setting up a Posts Subscription

In this lesson, it's your job to create a subscription that allows clients to be notified when new posts are created.

There are no notes for this challenge video as no new language features were explored.

Lesson 5: Expanding the Posts Subscription for Edits and Deletions

In this lesson, you'll be expanding the post subscription. The server currently notifies subscribers when posts are created. This is a good start, but the server should also notify clients when posts are updated or deleted. This will allow the client to keep the UI completely up to date.

Updating the Subscription Payload

It's not necessary to add new subscriptions for updated and deleted posts. Instead, the subscription payload will be updated to send back both the post data via the `data` field and the mutation type via the `mutation` field. The value for `mutation` could be either `"CREATED"`, `"UPDATED"`, or `"DELETED"`.

```
type Subscription {
    post: PostSubscriptionPayload!
}

type PostSubscriptionPayload {
    mutation: String!
    data: Post!
}
```

With the payload changed, all `pubsub.publish` calls for that subscription need to be updated to provide both the `data` and `mutation` fields. The example below shows the `publish` method call for a deleted post.

```
const Mutation = {
    deletePost(parent, args, { db, pubsub }, info) {
        const postIndex = db.posts.findIndex((post) => post.id === args.id)

        if (postIndex === -1) {
            throw new Error('Post not found')
        }

        const [post] = db.posts.splice(postIndex, 1)

        db.comments = db.comments.filter((comment) => comment.post !==
args.id)

        if (post.published) {
            pubsub.publish('post', {
                post: {
                    mutation: 'DELETED',
                    data: post
                }
            })
        }

        return post
    }
}
```

The client can now request both the mutation type and data.

```
subscription {
  post{
    mutation
    data {
      id
      title
      body
      author{
        id
        name
      }
    }
  }
}
```

Below is the JSON payload for the above subscription. Because of the `mutation` field, the client now knows why it's being notified. In this case, it's being notified because the post has been deleted. The client can do whatever it likes with this information, such as removing the comment from the screen.

```
{
  "data": {
    "post": {
      "mutation": "UPDATED",
      "data": {
        "id": "10",
        "title": "GraphQL 101",
        "body": "Something new...",
        "author": {
          "id": "1",
          "name": "Andrew"
        }
      }
    }
  }
}
```

Lesson 6: Expanding the Comments Subscription for Edits and Deletions

In this lesson, it's going to be up to you to expand the comments subscription. The goal is to notify subscribers when comments are created, updated, or deleted.

There are no notes for this challenge video as no new language features were explored.

Lesson 7: Enums

In this lesson, you're going to learn about enums (enumerations) in GraphQL.

Working with Enums

Enums are a special type in GraphQL that defines a set of constants. So you could have a `PowerState` enum for a laptop where the set of constants includes `on`, `off`, and `asleep`. Below is an example of that enum definition.

```
enum PowerState {
    on
    off
    asleep
}
```

A fields type can then be `PowerState`. The only valid values for that field would be one of the defined constants.

Below is the enum definition for the `mutation` field. This limits the values for `mutation` to either `CREATED`, `UPDATED`, or `DELETED`. Always use enums when a field can only be one of a fixed set of values.

```
enum MutationType {
    CREATED
    UPDATED
    DELETED
}

type PostSubscriptionPayload {
    mutation: MutationType!
    data: Post!
}

type CommentSubscriptionPayload {
    mutation: MutationType!
    data: Comment!
}
```

Documentation Links

- Enums

Section 5: Database Storage with Prisma

Lesson 1: Section Intro

In this section, you'll be integrating Prisma into your GraphQL project. Prisma is a GraphQL specific ORM (Object Relational Mapping) that makes it easy to integrate data storage into your GraphQL applications.

Lesson 2: What is Prisma?

In this lesson, you'll learn exactly what Prisma is and where it fits into your application's architecture.

There are no notes for this video, though this video does contain an important visualization.

Lesson 3: Prisma Mac Setup

In this lesson, you'll be setting up your Mac to work with Prisma. There are a few tools and dependencies you'll need before it can be used.

There are three things you'll need to do:

1. Set up a Postgres database on Heroku.
2. Install PGAdmin 4 and connect to the Heroku database.
3. Install Docker Community Edition and run it.

You can find links for all three below.

Documentation Links

- Heroku
- PGAdmin 4
- Docker Community Edition

Lesson 4: Prisma Windows Setup

In this lesson, you'll be setting up your Windows machine to work with Prisma. There are a few tools and dependencies you'll need before it can be used.

There are three things you'll need to do:

1. Set up a Postgres database on Heroku.
2. Install PGAdmin 4 and connect to the Heroku database.
3. Install Docker Community Edition and run it.

You can find links for all three below.

Documentation Links

- Heroku
- PGAdmin 4
- Docker Community Edition

Lesson 5: Prisma Linux Setup (Ubuntu)

In this lesson, you'll be setting up your Linux machine to work with Prisma. There are a few tools and dependencies you'll need before it can be used.

There are three things you'll need to do:

1. Set up a Postgres database on Heroku.
2. Install PGAdmin 4 and connect to the Heroku database.
3. Install Docker Community Edition and run it.

You can find links for all three below.

Documentation Links

- Heroku
- PGAdmin 4
- Docker Community Edition

Lesson 6: Prisma 101

In this lesson, you'll be creating and running your first Prisma service using the tools set up in the previous videos.

Creating a Prisma Application

First up, the Prisma command line tool needs to be installed.

```
npm i -g prisma@1.12.0
```

With Prisma installed, `prisma init` can be used to generate a new project. `prisma init` requires you provide a name with the project.

```
prisma init prisma
```

The initialization command will auto-generate a couple of files to get your Prisma project up and running. To generate these files correctly, `prisma init` will ask a series of questions about the database you want to use. This is where you'll provide the database connection details.

It's important to note that `prisma init` isn't doing anything magical behind the scenes. It's just generating three files. Those three files could have just as well been created from scratch.

The project can be deployed with the following commands.

```
cd prisma

docker-compose up -d

prisma deploy
```

Don't forget, it was necessary to set `ssl` to `true` in the database connection details in `docker-compose.yml`.

Lesson 7: Exploring the Prisma GraphQL API

In this video, you're going to explore the GraphQL API that's provided by Prisma.

Exploring the GraphQL API

Prisma automatically generates a GraphQL API based on the type definitions in `datamodel.graphql`. That means you won't be manually defining queries, mutations, or subscriptions for your types. All that's necessary is to define the types themselves.

```
type User {
  id: ID! @unique
  name: String!
}
```

By defining a `User`, prisma will automatically generate queries, mutations, and subscriptions to manage users such as `createUser`, `users`, and more. Refer to the schema in GraphQL Playground to determine what operations you can perform. The operation below creates a new user which gets stored in the Postgres database.

```
mutation {
  createUser(
    data:{
      name:"Vikram"
    }
  ){
    id
    name
  }
}
```

The JSON response is shown below.

```
{
  "data": {
    "createUser": {
      "id": "cjjsq2rqf001d0822g2if54em",
      "name": "Vikram"
    }
  }
}
```

Lesson 8: Add Post type to Prisma

In this lesson, you'll learn how to customize your Prisma datamodel and how to push those changes to the Prisma service.

Prisma supports all the same features you get with GraphQL. Types can be configured with a combination of scalar fields and fields that link to other types. Below is a `datamodel.graphql` file that shows this in action.

```
type User {
  id: ID! @unique
  name: String!
  email: String! @unique
  posts: [Post!]!
}

type Post {
  id: ID! @unique
  title: String!
  body: String!
  published: Boolean!
  author: User!
}
```

Changing `datamodel.graphql` is not enough to get the Prisma GraphQL API to change. Your changes must be deployed with `prisma deploy`. After deployment, you'll see a new set of queries, mutations, and subscriptions in the API schema.

The operation below creates a new post for an existing user.

```
mutation {
  createPost(
    data:{
      title:"Prisma post",
      body:"",
      published:false,
      author:{
        connect:{
          id:"cjjucbpfg004i0822onkb9z2l"
        }
      }
    }
  ){
    id
    title
    body
    published
    author{
      id
      name
    }
  }
}
```

The JSON response is shown below. Just like with the non-prisma API, you're able to request related data such as information about the author of the post.

```
{
  "data": {
    "createPost": {
      "author": {
        "id": "cjjucbpfg004i0822onkb9z2l",
        "name": "Vikram"
      },
      "body": "",
      "published": false,
      "id": "cjjud0s7200580822gywi3e7y",
      "title": "Prisma post"
    }
  }
}
```

Lesson 9: Adding Comment Type to Prisma

In this lesson, it's going to be up to you to integrate comments into the Prisma project.

There are no notes for this challenge video as no new language features were explored.

Lesson 10: Integrating Prisma into a Node.js Project

In this lesson, you'll learn how to communicate with the Prisma GraphQL API from Node.js. That will allow you to read and write to the database right from Node.js itself.

Installing Prisma Bindings

While Prisma isn't a Node.js specific tool, the `prisma-binding` library makes it easy to query the Prisma API from Node.js. Setting this up will require both `prisma-binding` and `graphql-cli`.

```
npm install prisma-binding@2.1.1 graphql-cli@2.16.4
```

Configuring Prisma Bindings

When configuring `prisma-binding`, it's necessary to provide both the API endpoint and a copy of the APIs schema. The `graphql get-schema` command provided by `graphql-cli` can help with this.

First, a `.graphqlconfig` is required to determine where the schema should be saved.

```
{
    "projects": {
        "prisma": {
            "schemaPath": "src/generated/prisma.graphql",
            "extensions": {
                "endpoints": {
                    "default": "http://localhost:4466"
                }
            }
        }
    }
}
```

From there, a script can be added to the `scripts` object in `package.json` that'll fetch and save the schema.

```
{
  "scripts": {
    "get-schema": "graphql get-schema -p prisma"
  }
}
```

After running `npm run get-schema`, `prisma-binding` can now be configured from Node.js. The example below includes both the URL of the API and the path to the generated schema file.

```
// prisma.js
import { Prisma } from 'prisma-binding'

const prisma = new Prisma({
    typeDefs: 'src/generated/prisma.graphql',
    endpoint: 'localhost:4466'
})
```

Documentation Links

- GitHub: prisma-binding
- GitHub: graphql-cli

Lesson 11: Using Prisma Bindings

In this lesson, you'll be exploring how to perform queries using the bindings you just configured. This will allow you to fetch data out of the database.

Exploring the API

You already know that the GraphQL API provided by Prisma is automatically generated based on the contents of `datamodel.graphql`. If a new type is added, then new queries, mutations, and subscriptions will be added the next time the project is deployed.

We see this same behavior in Node.js. All the Prisma queries are accessible via methods on `prisma.query`. The example below shows how the `users` query and `comments` query can both be called from Node.js by accessing the `users` and `comments` methods on `prisma.query`.

All query methods accept two arguments. The first is where operation arguments can be provided. The second is where the selection set is provided. These methods return a promise that'll be fulfilled with the selected data.

```
prisma.query.users(null, '{ id name posts { id title } }').then((data) => {
    console.log(JSON.stringify(data, undefined, 2))
})

prisma.query.comments(null, '{ id text author { id name } }').then((data) =>
{
    console.log(JSON.stringify(data, undefined, 2))
})
```

Lesson 12: Mutations with Prisma Bindings

In this lesson, you'll learn how to preform mutations using Prisma bindings. This will allow you to create, update, and delete data right from Node.js.

Mutations with Prisma Bindings

You can access all available mutations on `prisma.mutation`. Just like with `prisma.query`, `prisma.mutation` has an auto-generated set of methods, one for each mutation available. As an example, if you wanted to use the `deletePost` mutation, you'd call `prisma.mutation.deletePost`.

The example below creates a new post using `createPost`. Refer to the schema tab in GraphQL playground to determine what mutations you have access to, and what arguments those mutations take.

```
prisma.mutation.createPost({
    data: {
        title: "GraphQL 101",
        body: "",
        published: false,
        author: {
            connect: {
                id: "cjjybkwx5006h0822n32vw7dj"
            }
        }
    }
}, '{ id title body published }').then((data) => {
    console.log(data)
})
```

Lesson 13: Using Async/Await with Prisma Bindings

In this lesson, you'll learn how to use async/await with the bindings provided by Prisma. Integrating async/await will simplify your asynchronous code, making it easier to read, write, and maintain.

In the example below, `createPostForUser` is created as an async function. That function has two jobs:

1. Use `prisma.mutation.createPost` to create a new post.
2. Use `prisma.query.use` to fetch the user for the post.

Without async/await, it would be necessary to chain together some promises creating more complex nested code. With async/await, you can await the promise provided by both methods, allowing you to query for the user only after the post has been created.

```
const createPostForUser = async (authorId, data) => {
    const post = await prisma.mutation.createPost({
        data: {
            ...data,
            author: {
                connect: {
                    id: authorId
                }
            }
        }
    }, '{ id }')
    const user = await prisma.query.user({
        where: {
            id: authorId
        }
    }, '{ id name email posts { id title published } }')
    return user
}
```

Below in an example call to `createPostForUser`.

```
createPostForUser('cjjucl3yu004x0822dq5tipuz', {
    title: 'Great books to read',
    body: 'The War of Art',
    published: true
}).then((user) => {
    console.log(JSON.stringify(user, undefined, 2))
})
```

Lesson 14: Checking If Data Exists Using Prisma Bindings

In this lesson, you'll learn how to check if a given record exists using the bindings provided by Prisma. This will enable checks such as checking if a given user exists with a specific email address.

Using the "exists" Bindings

The third property on `prisma` is `prisma.exists`. This object contains a single method for each of the types you've defined. If you set up a `User` and `Post` type in `datamodel.graphql`, then you'll have a `User` and `Post` method on `prisma.exists`.

The example below shows off a modified version of `updatePostForUser`. The function first uses `prisma.exists.Post` to check if the post that's supposed to be updated actually

exists in the database. If the post doesn't exist, a custom error is thrown. If the post does exist, the post will then be updated.

```
const updatePostForUser = async (postId, data) => {
    const postExists = await prisma.exists.Post({ id: postId })

    if (!postExists) {
        throw new Error('Post not found')
    }

    const post = await prisma.mutation.updatePost({
        where: {
            id: postId
        },
        data
    }, '{ author { id name email posts { id title published } } }')

    return post.author
}
```

Below is an example call to `updatePostForUser`. This example uses `catch` to catch any errors that are thrown.

```
updatePostForUser("power", { published: true }).then((user) => {
    console.log(JSON.stringify(user, undefined, 2))
}).catch((error) => {
    console.log(error.message)
})
```

Lesson 15: Customizing Type Relationships

In this lesson, you'll learn how to customize the relationships between types. This gives you fine grain control over what happens to related data when a given record gets removed.

Working with the "relation" Directive

Prisma comes with built-in directives that you can use in `datamodel.graphl`. You've already seen the `unique` directive in action. Now it's time to explore `relation`.

The `relation` directive can be used to customize the relationship between types. This is great because it allows you to customize what happens to related data when a given record gets removed. For example, if a user gets deleted, what should happen to the

user's posts? Should the posts stay around and be made anonymous, or should the posts get removed too?

The example below shows the relationship between a job and an applicant. By setting `onDelete` to `CASCASE`, you're saying that related data should be removed. In the example below, removing a job would cause all applications for that job to also be removed. By setting `onDelete` to `SET_NULL`, you're saying that related data should not be removed. In the case of removing an individual application, the job itself should not be removed.

```
type Job {
  id: ID! @unique
  title: String!
  qualifications: String!
  salary: Int!
  applicants: [Applicant!]! @relation(name: "ApplicantToJob",
onDelete: CASCADE)
}

type Applicant {
  id: ID! @unique
  name: String!
  email: String! @unique
  resume: String!
  job: Job! @relation(name: "ApplicantToJob", onDelete: SET_NULL)
}
```

Lesson 16: Modeling a Review System with Prisma: Set Up

In this challenge video, it'll be up to you to model a review website using what you've learned about GraphQL and Prisma.

There are no notes for this setup video as no new language features were explored.

Lesson 17: Modeling a Review System with Prisma: Solution

In this challenge solution, you'll go over the steps necessary to create a Prisma application for a review website.

There are no notes for this challenge video as no new language features were explored.

Section 6: Authentication with GraphQL

Lesson 1: Section Intro

In this section, you'll learn how to create an authentication system using GraphQL. This will allow users to sign up and login to the application with an email and password. You'll also learn how to limit access to your GraphQL API based on authentication.

Lesson 2: Adding Prisma into GraphQL Queries

In this lesson, you'll start to integrate Prisma into your Node.js GraphQL API. The goal here is to set up the Node.js application as the middleman between the client and Prisma, allowing you to lock down your data with authentication.

Adding Prisma to the Context

In order to use Prisma from our GraphQL resolvers, it's necessary to create an instance of `prisma-binding` and add it to the application context.

```
import { Prisma } from 'prisma-binding'

const prisma = new Prisma({
    typeDefs: 'src/generated/prisma.graphql',
    endpoint: 'http://localhost:4466'
})

export { prisma as default }
```

With the context configured, resolver methods can now access `prisma.query`, `prisma.mutation`, `prisma.subscription`, and `prisma.exists` to interact with the database. There's no need to worry about authentication just yet.

The example below shows how the `users` query can access `prisma.query.users` to fetch all users from the database.

```
const Query = {
    users(parent, args, { prisma }, info) {
        return prisma.query.users(null, info)
    }
}
```

The `prisma-binding` methods for queries, mutations, and subscriptions take two arguments. The first is for operation arguments. The second is for the selection set. There are actually three different types you can provide for the second argument:

1. A string - A string can be provided allowing you to define your own selection set like you can do in GraphQL playground.
2. Nothing - `undefined` can be provided in which case all scalar fields for the type will be selected.
3. An info object - As seen in the example above, the `info` object can also be provided. This allows the selection set from the client to be passed through to Prisma.

Documentation Links

- Prisma-binding Library

Lesson 3: Integrating Operation Arguments

In this lesson, you'll learn how to take operation arguments provided to the Node.js API and pass them through to the Prisma API. This will allow you to set up filtering, sorting, and pagination.

Integrating Operation Arguments

The `users` query below takes in a single string argument named `query`. Before Prisma, this was used to allow clients to search for users by their name or email. The resolver below allows for the same searching by using the operation arguments supported by the Prisma API.

As always, the GraphQL Playground for Prisma can be used to see all the arguments that each query, mutation, or subscription supports.

```
const Query = {
    users(parent, args, { prisma }, info) {
        const opArgs = {}

        if (args.query) {
            opArgs.where = {
                OR: [{
                    name_contains: args.query
                }, {
                    email_contains: args.query
                }]
            }
        }

        return prisma.query.users(opArgs, info)
    }
}
```

With support for the argument added, the operation below is able to search for users that have `"Andrew"` as part of the user's name or email.

```
query {
  users (
    query:"Andrew"
  ) {
    id
    name
    email
  }
}
```

Lesson 4: Refactoring Custom Type Resolvers

In this lesson, you'll explore how Prisma makes it unnecessary to set up custom type resolvers for associated data.

Refactoring Custom Type Resolvers

Earlier in the class, you learned how to set up relationships between your types, such as a relationship between users and posts. This allowed you to use a query like the one below to fetch a user's posts.

```
query {
  users {
    id
    name
    email
    posts {
      id
      title
    }
  }
}
```

To support that query, you had to set up resolvers for `User.posts` and `User.comments`. The same was true for the other types which included `Post.author`, `Post.comments`, `Comment.author`, and `Comment.post` as shown below.

```
const User = {
    posts(parent, args, { db }, info) {
        return db.posts.filter((post) => {
            return post.author === parent.id
        })
    },
    comments(parent, args, { db }, info) {
        return db.comments.filter((comment) => {
            return comment.author === parent.id
        })
    }
}

export { User as default }
```

With Prisma, this is no longer necessary. Prisma has built-in support for relational data. All you need to do is pass the `info` argument to `prisma-binding`. The `info` arguments contain the entire selection set provided by the client, giving Prisma everything it needs to pull the necessary data out of the database.

That means `User.js`, `Comment.js` and `Post.js` can be refactored to remove all resolvers as shown below.

```
const User = {

}

export { User as default }
```

Lesson 5: Adding Prisma into GraphQL Mutations

In this lesson, you'll start integrating Prisma into the mutations for your Node.js GraphQL API. This is similar to what you've already done for your application queries.

Adding Prisma into GraphQL Mutations

In lesson 2, your instance of `prisma-binding` was added to the application context. This allowed you to use `prisma.query` to fetch data out of the database. In the example below, `prisma.exists` and `prisma.mutation` are both used to refactor the `createUser` mutation to save data into the database.

```
const Mutation = {
    async createUser(parent, args, { prisma }, info) {
        const emailTaken = await prisma.exists.User({ email: args.data.email
})

        if (emailTaken) {
            throw new Error('Email taken')
        }

        return prisma.mutation.createUser({ data: args.data }, info)
    }
}
```

Note that the call to `prisma.mutation.createUser` takes `info` as the second argument. This allows Prisma to send back all the data the client requested in its selection set.

Lesson 6: Adding Prisma into GraphQL Update Mutations: Part I

In this lesson, it's up to you to continue integrating Prisma into the Node.js application.

There are no notes for this challenge video as no new language features were explored.

Lesson 7: Adding Prisma into GraphQL Update Mutations: Part II

In this lesson, it's up to you to continue integrating Prisma into the Node.js application.

There are no notes for this challenge video as no new language features were explored.

Lesson 8: Adding Prisma into GraphQL Subscriptions

In this lesson, you'll learn how to set up your subscriptions to work with Prisma. The good news is that you'll be able to vastly simplify your application code.

Adding Prisma into GraphQL Subscriptions

Before Prisma, setting up subscriptions was a bit of a pain. Creating the subscription itself wasn't bad, but it was necessary to call `pubsub.publish` from our mutations to notify the subscription of some event. This made our mutation resolvers messy and error prone.

With Prisma, it's no longer necessary to manually call `pubsub.publish`. Prisma is managing all our data, so it already knows when data is created, updated, or deleted. That means you can remove all `pubsub.publish` calls from our application.

As shown below, all you need to do is use the correct subscription methods from your resolver. The example below uses `prisma.subscription.comment` to subscribe to comments, and it uses the `where` operation argument to subscribe to comments for a single post.

```
const Subscription = {
    comment: {
        subscribe(parent, { postId }, { prisma }, info){
            return prisma.subscription.comment({
                where: {
                    node: {
                        post: {
                            id: postId
                        }
                    }
                }
            }, info)
        }
    }
}
```

Below is the altered subscription payload that matches up with the payload used by Prisma itself.

```
type CommentSubscriptionPayload {
    mutation: MutationType!
    node: Comment
}
```

Lesson 9: Closing Prisma to the Outside World

In this lesson, you'll learn how to lock down access to the Prisma API. Right now, the Prisma API is open to the public, allowing anyone to read and write from the database without needing to authenticate. That's a problem.

Configuring a Prisma Secret

To restrict access to the database, Prisma allows a secret to be configured. The secret, similar to a password, is required in order to read or write from the database using the Prisma API.

That can be done by adding a single line to `prisma.yml`.

```
endpoint: http://localhost:4466
datamodel: datamodel.graphql
secret: thisismysupersecrettext
```

Then redeploy your Prisma app.

```
prisma deploy
```

You don't want to lock everyone out of Prisma. The secret should be shared with the Node.js application, giving it exclusive rights to interact with the Prisma API. That can be done by adding `secret` onto the options object for the `prisma-binding` constructor functions.

```
import { Prisma } from 'prisma-binding'

const prisma = new Prisma({
    typeDefs: 'src/generated/prisma.graphql',
    endpoint: 'http://localhost:4466',
    secret: 'thisismysupersecrettext'
})

export { prisma as default }
```

Generating Access Tokens

You can still access the Prisma API from GraphQL playground for development purposes. First, generate a token.

```
prisma token
```

From there, you'll want to set up an HTTP authorization header that uses the generated token.

```
{
  "Authorization":"Bearer YourTokenHere"
}
```

Lesson 10: Allowing for Generated Schemas

In this lesson, you'll make a small update to ensure that you can still fetch the generated schema from Prisma, even though Prisma is not protected by the secret.

Allowing for Generated Schemas

If you run `npm run get-schema`, it's going to fail. This is because `get-schema` is trying to get the generated schema via `http://localhost:4466`, which is not protected by the secret.

To get `npm run get-schema` working again, all you need to do is add `prisma` to `extensions` as shown below. This will give `get-schema` direct access to the application, allowing it to properly fetch the generated schema.

```
{
    "projects": {
        "prisma": {
            "schemaPath": "src/generated/prisma.graphql",
            "extensions": {
                "prisma": "prisma/prisma.yml",
                "endpoints": {
                    "default": "http://localhost:4466"
                }
            }
        }
    }
}
```

From there, all you need to do is update the `get-schema` script in `package.json` with the correct project name.

```
graphql get-schema -p prisma
```

Lesson 11: Storing Passwords

In this lesson, you'll start adding password support into the application. The goal here is to require a password when someone signs up for the application.

Property Storing Passwords

To add password support, new users must be required to provide a password when signing up. That means a new argument needs to be added for `createUser` in `schema.graphql`.

From there, the password needs to be validated and securely stored. Storing passwords in plain text is highly insecure and irresponsible. All passwords should be securely hashed before being stored in the database. There are plenty of exiting algorithms that can be used to hash the passwords, my personal favorite being bcrypt.

The `createUser` mutation below shows how to hash and store the password.

```
import bcrypt from 'bcryptjs'

const Mutation = {
    async createUser(parent, args, { prisma }, info) {
        if (args.data.password.length < 8) {
            throw new Error('Password must be 8 characters or longer.')
        }

        const password = await bcrypt.hash(args.data.password, 10)

        return prisma.mutation.createUser({
            data: {
                ...args.data,
                password
            }
        }, info)
    }
}
```

Documentation Links

- Bcrypt
- Salting
- Rainbow Looup Table

Lesson 12: Creating Auth Tokens with JSON Web Tokens

In this lesson, you're going to learn about JSON Web Tokens. JSON Web Tokens, also known as JWTs, provide you with a secure way to create authentication tokens for your application.

Working with JWTs

A JWT is nothing more than a string. When a user signs up or logs in to the application, a JWT will be generated and provided to them. The client can then store this token and send it along with future requests to authenticate itself.

Below is a modified version of the `createUser` mutation. The mutation was updated to send back both the user and the authentication token.

```
import jwt from 'jsonwebtoken'

const Mutation = {
    async createUser(parent, args, { prisma }, info) {
        if (args.data.password.length < 8) {
            throw new Error('Password must be 8 characters or longer.')
        }

        const password = await bcrypt.hash(args.data.password, 10)
        const user = await prisma.mutation.createUser({
            data: {
                ...args.data,
                password
            }
        })

        return {
            user,
            token: jwt.sign({ userId: user.id }, 'thisisasecret')
        }
    }
}
```

Documentation Links

- JSON Web Token Introduction
- JSON Web Token Wikipedia
- Creating Tokens

Lesson 13: Logging in Existing Users

In this lesson, it'll be up to you to create a new mutation that allows users to log in to the application.

There are no notes for this challenge video as no new language features were explored.

Lesson 14: Validating Auth Tokens

In this lesson, you'll learn how to validate an authentication token.

Providing to Token with the Request

The Node.js GraphQL API is going to consist of a mix of public and private operations. As any example, an unauthenticated user should be able to fetch all published posts. On the other side of the coin, an unauthenticated user should not be able to fetch any unpublished draft posts.

To authenticate, the client will send along an authorization header that includes the auth token provided when either signing up or logging in.

```
{
  "Authorization": "Bearer
eyJhbGciaiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOiJjamtqczMBcGYwMDQ3MDg3MzJt
NHJoaWFjIiwiaWF0IjoxNTMzNjU2NzE2fQ.MHSey8h1xeIfUmMvV75Vhmzvbx7R7jR65ZWr--
r2oVY"
}
```

Validating the Token

In the Node.js application, the goal is to get the header into the resolver methods by adding it onto the application context. The **`context`** property can be set equal to a function. This function is called with a single argument which contains information about the request. That's where the authorization header lives.

```
const server = new GraphQLServer({
    typeDefs: './src/schema.graphql',
    resolvers: {
        Query,
        Mutation,
        Subscription,
        User,
        Post,
        Comment
    },
    context(request) {
        return {
            db,
            pubsub,
            prisma,
            request
        }
    }
})
```

With **`request`** being added onto the context, it can now be used inside the resolver methods. The example below shows how **`createPost`** can be put behind authentication.

First up, the **`request`** object is passed to **`getUserId`** which is defined below. **`getUserId`** is in charge of extracting the token from the authorization headers, verifying the token, and

returning the authenticated users id. `getUserId` will throw an error if authentication fails for any reason.

```
const Mutation = {
    createPost(parent, args, { prisma, request }, info) {
        const userId = getUserId(request)

        return prisma.mutation.createPost({
            data: {
                title: args.data.title,
                body: args.data.body,
                published: args.data.published,
                author: {
                    connect: {
                        id: userId
                    }
                }
            }
        }, info)
    }
}
```

Below is the implementation of `getUserId` which extracts and validates the token.

```
import jwt from 'jsonwebtoken'

const getUserId = (request) => {
    const header = request.request.headers.authorization

    if (!header) {
        throw new Error('Authentication required')
    }

    const token = header.replace('Bearer ', '')
    const decoded = jwt.verify(token, 'thisisasecret')

    return decoded.userId
}

export { getUserId as default }
```

Documentation Links

- Context documentation

- JWT verification

Lesson 15: Locking Down Mutations (Users)

In this video, you'll start locking down your mutations. This will ensure the client is authenticated before any data in the database is created, updated, or deleted.

Locking Down a Mutation

Below is a modified version of the `deletePost` mutation that is sitting behind authentication. Once again, `getUserId` is used to ensure the request was made by an authenticated user.

Even after authentication, it's important to restrict what the user can do. As shown below, the authenticated user will be able to delete the post by id, but only if they are the original author of the post. This will prevent user A from deleting a post created by user B.

```
const Mutation = {
    async deletePost(parent, args, { prisma, request }, info) {
        const userId = getUserId(request)
        const postExists = await prisma.exists.Post({
            id: args.id,
            author: {
                id: userId
            }
        })

        if (!postExists) {
            throw new Error('Unable to delete post')
        }

        return prisma.mutation.deletePost({
            where: {
                id: args.id
            }
        }, info)
    }
}
```

Lesson 16: Locking Down Mutations (Posts and Comments)

In this lesson, it'll be up to you to lock down the other application mutations.

There are no notes for this challenge video as no new language features were explored.

Lesson 17: Locking Down Queries: Part I

With your mutations behind authentication, in this video, you'll learn how to lock down your application queries.

What Should Require Authentication?

Locking down mutations is pretty easy. The entire mutation is either public or private. For example, the login mutation is public while the mutation for deleting a comment is private. It becomes less black and white when working with queries.

A great example is the `post` query which allows an individual post to be fetched by id. A user should not need to be authenticated if they want to read a published post. What if the post isn't published though? In that case, the user should be authenticated and they should also be the author of the post.

To account for this, `getUserId` should take an argument that determines if authentication is required or optional. If authentication is marked as optional and the user isn't authenticated, `null` will be returned instead of the user id.

```
const getUserId = (request, requireAuth = true) => {
    const header = request.request.headers.authorization

    if (header) {
        const token = header.replace('Bearer ', '')
        const decoded = jwt.verify(token, 'thisisasecret')
        return decoded.userId
    }

    if (requireAuth) {
        throw new Error('Authentication required')
    }

    return null
}
```

Below is a modified version of `post` that's sitting behind authentication. First, it checks if the user is authenticated by calling `getUserId`. Notice that `false` is provided as the value for the `requireAuth` argument. If the user is authenticated, `userId` will get set to their string id. If the user is not authenticated, `userId` will get set to `null`.

From there, `prisma.query.posts` is called to get the post by id. It also makes sure that the post is either published or owned by the authenticated user.

```
const Mutation = {
    async post(parent, args, { prisma, request }, info) {
        const userId = getUserId(request, false)

        const posts = await prisma.query.posts({
            where: {
                id: args.id,
                OR: [{
                    published: true
                }, {
                    author: {
                        id: userId
                    }
                }]
            }
        }, info)

        if (posts.length === 0) {
            throw new Error('Post not found')
        }

        return posts[0]
    }
}
```

Lesson 18: Locking Down Queries: Part II

In this video, you'll continue to lock down your application queries.

There are no notes for this lesson as no new features were explored.

Lesson 19: Locking Down Individual Type Fields

In this lesson, you'll learn how to hide an individual field behind authentication. This will give you a way to hide user email address from the public.

Hiding User Email Addresses

If a field will be hidden from unauthenticated users, it should be marked as nullable in `schema.graphql`. If the users email will be hidden from unauthenticated users, its type should be changed from `String!` to `String`.

The email address isn't something that should always be hidden. A user should be able to see their own email address. To set that up, a resolver is required for `User.email` as shown below.

Notice that `getUserId` is called with `requireAuth` set to `false`. If the user is authenticated and they're fetching their own profile, the email address will be returned. If the user is unauthenticated or they're fetching someone else's profile, `null` will be returned instead of the user's email.

```
import getUserId from '../utils/getUserId'

const User = {
    email(parent, args, { request }, info) {
        const userId = getUserId(request, false)

        if (userId && userId === parent.id) {
            return parent.email
        } else {
            return null
        }
    }
}

export { User as default }
```

Lesson 20: Fragments

In this lesson, you'll learn about GraphQL fragments. Fragments give you a reusable way to define a selection set. No longer will you have to list out all the fields for a user, post, or comment every time you define your selection sets.

Fragments in GraphQL Playground

Fragments are a GraphQL feature and are not specific to Prisma. They can be used on the client to define reusable selection sets. The example below defines a `userFields` fragment for the `User` type. The fragment lists out all of the fields from `User` that it wants. In this case, `userFields` is asking for the user's id, name, and email.

Fragments can be used anywhere you'd normally define a selection set. This could be for a query, mutation, or subscription. In the example below, the `users` query is using the fragment to ask for all the scalar fields for the user without having to individually list them out.

```
query {
  users {
    ...userFields
    posts{
      id
      title
      published
    }
  }
}

fragment userFields on User {
  id
  name
  email
}
```

The response, shown below, contains all the requested fields.

```
{
  "data": {
    "users": [
      {
        "id": "cjkjs28fg003x0873dqzy26y8",
        "name": "Jess",
        "email": null,
        "posts": []
      },
      {
        "id": "cjkl8qxky005z0873eix4o03n",
        "name": "Andrew",
        "email": null,
        "posts": [
          {
            "id": "cjkl8rfiz00630873lz1zv2ho",
            "title": "Updated post by Andrew",
            "published": true
          }
        ]
      }
    ]
  }
}
```

Integrating Fragments into Prisma

Fragments can also be used when interacting with Prisma. To configure this, you need to use `extractFragmentReplacements` from `prisma-binding`. You call that function with the resolver object and it returns the extracted fragments. From there, you set a `fragmentReplacements` property on the option object for both `Prisma` and `GraphQLServer`. The value for `fragmentReplacements` should be the extracted fragments from `extractFragmentReplacements`.

The code below shows how fragments can be used when interacting with Prisma. It allows our resolver method to have access to the user id even if the original selection set from the client never asked for it.

```
import getUserId from '../utils/getUserId'

const User = {
    email: {
        fragment: 'fragment userId on User { id }',
        resolve(parent, args, { request }, info) {
            const userId = getUserId(request, false)

            if (userId && userId === parent.id) {
                return parent.email
            } else {
                return null
            }
        }
    }
}

export { User as default }
```

Lesson 21: Cleaning up Some Edge Cases

In this lesson, it'll be up to you to clean up a few edge cases in the application.

There are no notes for this challenge video as no new language features were explored.

Lesson 22: Locking Down Subscriptions

In this lesson, you'll learn how to put your subscriptions behind authentication.

Locking Down Subscriptions

Before `getUserId` can be used from inside a subscription resolver, a small change needs to be made to `getUserId`. This is because the authorization header for subscriptions lives on a different property than it does for queries and mutations. The snippet below shows the one line that needs to change.

```
const getUserId = (request, requireAuth = true) => {
    const header = request.request ? request.request.headers.authorization :
request.connection.context.Authorization

    // The rest of the function is the same and has been removed for brevity
}
```

With `getUserId` configured, it can now be used inside resolver methods for subscriptions. The example below shows a `myPost` subscription which requires authentication. Once authenticated, it will notify you about changes to any post that you've authored.

```
const Subscription = {
    myPost: {
        subscribe(parent, args, { prisma, request }, info) {
            const userId = getUserId(request)

            return prisma.subscription.post({
                where: {
                    node: {
                        author: {
                            id: userId
                        }
                    }
                }
            }, info)
        }
    }
}
```

Lesson 23: Token Expiration

In this lesson, you'll learn how to expire JSON Web Tokens after a specific period of time.

Creating Tokens that Expire

To create a token that expires, you need to set up a single option when signing the token. `jwt.sign` accepts a third argument, an objects object. This object can be configured with an `expiresIn` value to expire the token after a specific period of time. The example below generates a token that expires in 7 days.

```
jwt.sign({ userId }, 'thisisasecret', { expiresIn: '7 days' })
```

That's it. There's no need to change how tokens are verified. `jwt.verify` will automatically check if a given token is expired. If the token is expired, an error will be thrown.

Documentation Links

- expiresIn option (scroll down to the options list)

Lesson 24: Password Updates

In this video, you'll allow users to update their passwords.

There are no notes for this lesson as no new features were explored.

Section 7: Pagination and Sorting with GraphQL

Lesson 1: Section Intro

In this section, you'll learn how to paginate and sort your GraphQL data. This will give more options to clients so they can request just the data they need, in the order they want it in.

Lesson 2: Pagination

In this lesson, you'll learn how to integrate pagination into your GraphQL queries. This will allow clients to fetch a limited number of posts, such as the first 20, with the ability to fetch the next set of 20 when they need them.

Pagination

There are two new arguments that'll be added to queries that need to support pagination, `first` and `skip`. `first` can be used to determine how many records should be fetched. Setting `first` to `10` would give you the first 10 records. `skip` can be used to skip a number of records.

Image `first` is `10` and `skip` is `0`. This would skip 0 records and then get the first 10. That would be the first 10. If you want to fetch the second set of 10, you'd keep `first` as `10` and you'd set `skip` to `10`. That would skip 10 records and then first the first 10. That would be records 11 through 20.

Prisma has built-in support for `first` and `skip`. Pagination can be set up by first adding the arguments to the query.

```
type Query {
    users(query: String, first: Int, skip: Int): [User!]!
}
```

Next, the resolver can be updated to pass the `first` and `skip` arguments through to Node.

```
const Query = {
    users(parent, args, { prisma }, info) {
        const opArgs = {
            first: args.first,
            skip: args.skip
        }

        if (args.query) {
            opArgs.where = {
                OR: [{
                    name_contains: args.query
                }]
            }
        }

        return prisma.query.users(opArgs, info)
    }
}
```

A query like the following can then be used to fetch a specific set of records. The below query will fetch the 3rd page of users where each page contains 3 users. That would be users 7 through 9.

```
query {
  users(first:3, skip:6){
    id
    name
  }
}
```

Lesson 3: Pagination Using Cursors

In this lesson, you'll learn how to set up pagination using cursors. This is an alternative way to specify which records should be fetched.

Pagination Using Cursors

Pagination using cursors requires a new argument named `after`. The value for `after` will be the id of the record you want to start fetching from.

Imagine `first` is `50` and `after` is undefined. That would fetch the first 50 records. The next set of 50 record can be fetched by modifying `after`. `after` would have it's value set to the last record from the initial set of 50. So `first` would still be `50` and `after` would be something like `'somerandomid'`.

Prisma has built-in support for this. After the query is set up to support `after`, the resolver can be updated to pass those arguments down to Prisma.

```
type Query {
    users(query: String, first: Int, skip: Int, after: String):
[User!]!
}


const Query = {
    users(parent, args, { prisma }, info) {
        const opArgs = {
            first: args.first,
            skip: args.skip,
            after: args.after
        }

        if (args.query) {
            opArgs.where = {
                OR: [{
                    name_contains: args.query
                }]
            }
        }

        return prisma.query.users(opArgs, info)
    }
}
```

Lesson 4: Working with createdAt and updatedAt

In this lesson, you'll learn how to work with createdAt and updatedAt. These timestamps keep track of when a given record was created and when it was last updated.

Working with createdAt and updatedAt

There are three properties Prisma manages, `id`, `updatedAt`, and `createdAt`. `updatedAt` and `createdAt` can be exposed by updating the prisma datamodel to include those fields. The type for both should be `DateTime!`, which is a Prisma provided type for these fields.

```
type Comment {
  id: ID! @unique
  text: String!
  author: User! @relation(name: "CommentToUser", onDelete: SET_NULL)
  post: Post!@relation(name: "CommentToPost", onDelete: SET_NULL)
  updatedAt: DateTime!
  createdAt: DateTime!
}
```

Next, your model in `schema.graphql` can be updated to support both properties. In this case, both fields should have the type `String!`.

```
type Comment {
    id: ID!
    text: String!
    author: User!
    post: Post!
    updatedAt: String!
    createdAt: String!
}
```

Now, clients can add `createdAt` and `updatedAt` to their selection sets. Below is an example query that fetches both timestamps for comments.

```
query {
  comments{
    id
    text
    updatedAt
    createdAt
  }
}
```

The JSON response below shows those timestamps along with the `id` and `text` fields for each comment.

```
{
  "data": {
    "comments": [
      {
        "id": "cjklaufyx009h0873rhadqtpz",
        "text": "Jess Comment 3",
        "updatedAt": "2018-08-08T15:39:30.881Z",
        "createdAt": "2018-08-08T15:39:30.881Z"
      },
      {
        "id": "cjklaugv5009n087301y9ybs2",
        "text": "This is updated with auth",
        "updatedAt": "2018-08-08T15:42:57.661Z",
        "createdAt": "2018-08-08T15:39:31.953Z"
      }
    ]
  }
}
```


Lesson 5: Sorting Data

In this lesson, you'll learn how to sort data with GraphQL and Prisma. This will allow clients to fetch data sorted in a way that's useful to them. For example, blog posts on the home page should probably be sorted with the newest posts first.

Sorting Data

Prisma exposes an `orderBy` argument for all operations that respond with an array of items. Records of a given type can be sorted by any of their field values in either ascending or descending fashion. Like with our pagination arguments, `orderBy` should be accepted by our operations and passed through to Prisma.

Below is a modified definition of the `users` query that supports `orderBy`. The type `UserOrderByInput` comes directly from Prisma. The import syntax makes it easy to import definitions from other GraphQL files.

```
# import UserOrderByInput from './generated/prisma.graphql'

type Query {
    users(query: String, first: Int, skip: Int, after: String,
orderBy: UserOrderByInput): [User!]!
}
```

From there, the Node resolver should be updated to pass that argument through to Prisma.

const Query = { users(parent, args, { prisma }, info) { const opArgs = { first: args.first, skip: args.skip, after: args.after, orderBy: args.orderBy }

```
    if (args.query) {
        opArgs.where = {
            OR: [{
                name_contains: args.query
            }]
        }
    }

    return prisma.query.users(opArgs, info)
}
```

}

The example query below sets `orderBy` to `createdAt_DESC`. This would return the users with the newest users first and the oldest users last.

```
query {
  users(orderBy:createdAt_DESC){
    id
    name
    email
    updatedAt
    createdAt
  }
}
```

Documentation Links

- Graphql-import

Section 8: Authentication with GraphQL

Lesson 1: Section Into: Production Deployment

In this section, you'll be deploying your app to production. By the end, you'll have a live GraphQL API that can be accessed by anyone with an internet connection.

Lesson 2: Creating a Prisma Service

In this lesson, you'll set up your production database and production Prisma server using Prisma Cloud. Prisma Cloud makes it simple to deploy and manage your Prisma application.

First up, head over to app.prisma.io and create an account.

From there, you'll want to navigate to the servers page and create a new server. The setup wizard will ask you what services you want to use for the database and server. Currently, the only supported option is Heroku, so select that and link your existing Heroku account.

By the end, you'll have a production database and a production Prisma server.

Documentation Links

- Prisma Cloud

Lesson 3: Prisma Configuration and Deployment

In this lesson, you'll deploy your Prisma project to the production server you created in the last lesson.

Creating Configuration Files

Your Prisma project is going to run in multiple environments. This includes a development environment and a production environment. The development environment is on your local machine, and the production environment in on Heroku.

Supporting multiple environments will require a few small changes to the project. As an example, `prisma.yml` has a hardcoded endpoint value. The endpoint value is used to determine where to deploy the application, and right now it's always getting deployed to `http://localhost:4466`.

To address this, `prisma.yml` will rely on environment variables instead of hardcoded values. First, create a `config` directory with two files, `dev.env` and `prod.env`. Next, update `dev.env` to create a new environment variable.

```
PRISMA_ENDPOINT=http://localhost:4466
```

Then update `prisma.yml` to rely on that variable's value as opposed to relying on the hardcoded endpoint.

```
endpoint: ${env:PRISMA_ENDPOINT}
datamodel: datamodel.graphql
secret: thisismysupersecrettext
```

Now you can deploy to development by running `prisma deploy` and specifying the config file you want to use.

```
prisma deploy -e ../config/dev.env
```

Deploying to Production

Deploying to production will require you to log in to Prima Cloud via the Prisma command line interface.

```
prisma login
```

That command will open a browser window asking you if you want to grant access. Make sure to grant account access before moving on.

From here, you're going to deploy to production by using `prod.env` instead of `dev.env`.

```
prisma deploy -e ../config/prod.env
```

Since `prod.env` is empty, Prisma won't know where to deploy your project. It's going to list out all the servers you've created, including the server you created in the last lesson. Select the server you created, and then pick a name and stage for your service.

That command will write a URL to `prisma.yml`. That's the production URL. Copy that production URL over to `prod.env`.

```
PRISMA_ENDPOINT=PUT-YOUR-PRODUCTION-URL-HERE
```

Then you can remove the line that was auto-generated and uncomment your line, bringing the file back to its previous state.

```
endpoint: ${env:PRISMA_ENDPOINT}
datamodel: datamodel.graphql
secret: thisismysupersecrettext
```

Now you have a system for deploying Prisma to both development and production.

Lesson 4: Exploring the Production Prisma Instance

In this lesson, you'll explore the production Prisma service you deployed in the last one.

There are no notes for this lesson, as nothing new was covered.

Lesson 5: Node.js Production App Deployment: Part I

In this lesson, you'll start getting the Node.js application ready for deployment. There are a few changes that need to be made before Heroku will be able to run the app.

Installing the Tools

To start, you'll need to install Git and the Heroku CLI. You'll also need to login to the Heroku CLI so you can create and manage your production Node.js application.

```
heroku login
```

Configuring the Port

Heroku requires your application to start the server on a specific port. The value for that port is set as an environment variable on Heroku. The code starts the server on the correct port if a `PORT` environment variable is set, otherwise it uses port `4000`.

```
server.start({ port: process.env.PORT || 4000 }, () => {
    console.log('The server is up!')
})
```

Connecting to the Correct endpoint

In `prisma.js`, the endpoint our app connects to also needs to change. Instead of the hardcoded string `http://localhost:4466`, the value will come from the `PRISMA_ENDPOINT` environment variable.

```
const prisma = new Prisma({
    typeDefs: 'src/generated/prisma.graphql',
    endpoint: process.env.PRISMA_ENDPOINT,
    secret: 'thisismysupersecrettext',
    fragmentReplacements
})
```

To ensure that there's a value for `PRISMA_ENDPOINT` when the app runs locally, the npm library `env-cmd` will be used.

```
npm install env-cmd@8.0.2
```

The `dev` script in package.json can then be changed to the following, which will first load in the environment variables from `dev.env`.

```
env-cmd ./config/dev.env nodemon src/index.js --ext js,graphql --exec babel-node
```

Documentation Links

- Git
- Heroku CLI

Lesson 6: Node.js Production App Deployment: Part II

In this lesson, you'll be wrapping up the deployment of the Node.js application. By the end, your GraphQL API will be deployed live to the web.

Heroku Scripts

Heroku uses the scripts in `package.json` to start up your application. This application needs two. The first is `heroku-postbuild` which will let us run Babel. The second is `start` which will start up the web server.

Below is a partial snippet of `package.json`. Other properties and scripts are hidden for brevity.

```
{
    "scripts": {
        "start": "node dist/index.js",
        "heroku-postbuild": "babel src --out-dir dist --copy-files",
    }
}
```

Switching from `babel-node` to `babel` will require you to add one additional package.

```
npm install @babel/polyfill@7.0.0
```

This will ensure that features like async/await continue to work in production.

All you need to do is add a single import statement to the top of `src/index.js` to set this up.

```
import '@babel/polyfill/noConflict'
```

Adding Version Control

With the scripts set up, the Node.js app is ready to run on Heroku. Git, a version control system, will be used to track changes to the code and to send those changes to Heroku so a new version of the app can be deployed.

There are two folders we don't want to add to version control. The first is `node_modules`. Heroku will install the dependencies listed in package.json, so there's no need to include this folder in Git. The second is `config`. This folder contains sensitive information and should be left out of version control. This can be done by creating a `.gitignore` file in the root of the project with the following contents.

```
node_modules/
config/
```

It's now time to initialize a Git repository.

```
git init
```

From there, you can use `git add` to add your project files to the staging area so they'll be included with your next commit.

```
git add .
```

Next up, the initial commit can be created.

```
git commit -am "Init commit"
```

It's now time to create the Heroku application. This will create a new application (on the free tier) using the account you logged in with in the last video.

```
heroku create
```

Finally, your application code can be pushed to Heroku.

```
git push heroku master
```

The deployment process can take about 30 seconds to a minute. After it's done, you should be able to view your Node.js app using the URL provided by Heroku.

Documentation Links

- Babel Polyfill

Lesson 7: Node.js Production Environment Variables

In this lesson, you'll be pulling passwords and secrets out of the code base and into environment variables.

There are no notes for this challenge video as no new language features were explored.

Section 9: Apollo Client and Testing GraphQL

Lesson 1: Section Intro

In this section, you'll be setting up an automated test suite for GraphQL. You'll also learn how to use Apollo Client to send off GraphQL operations from Node.js and client-side JavaScript.

Lesson 2: Setting up a Test Environment

In this lesson, you'll be setting up a Prisma test environment. This will create a separate space in the database for test data. This will ensure that our test cases can't mess up our development or production data.

You can create a new environment by defining a new environment file in the `config` directory. The `test.env` file below has an updated endpoint value where `default` is the service name and `test` is the stage name.

```
PRISMA_ENDPOINT=http://localhost:4466/default/test
PRISMA_SECRET=pi389xjam2b3pjsd0
JWT_SECRET=23oijdsm23809sdf
```

The test environment can be deployed by running the following command from the `prisma` directory.

```
prisma deploy -e ../config/test.env
```

Lesson 3: Installing and Exploring Jest

In this lesson, you'll be setting up a testing environment using Jest. Jest is a test framework that gives you everything you need to build out your test cases. Unlike other test frameworks, Jest also ships with an assertion library, making it simple to get started.

Installing and Setting Up Jest

You can install the `jest` npm module using the following command.

```
npm install jest@23.5.0
```

From there, the test script can be used to run Jest. When executed, Jest will try to locate and run your test files.

```
{
  "scripts": {
    "test": "jest --watch"
  }
}
```

By default, Jest is going to look for files that end with `.test.js` inside the `tests` directory.

Below is an example `user.test.js` file which would live in the `tests` folder. The `test` function is used to set a single test case, though it doesn't test anything at the moment.

```
test('Should create a new user', () => {

})
```

Documentation Links

- Jest

Lesson 4: Testing and Assertions

In this lesson, you'll be writing your first test cases for real JavaScript functions.

The `test` function takes two arguments. The first is a string name that describes what the test is testing. The second argument is the actual test case. This is a function that Jest runs when it executes the test suite. If that function throws an error, Jest will mark the test as a failure. If that function doesn't throw an error, Jest will will mark test test as having passed.

As an example, take the file below which defines an `isValidPassword` function.

```
const isValidPassword = (password) => {
    return password.length >= 8 &&
!password.toLowerCase().includes('password')
}

export { isValidPassword }
```

That function can be tested by importing it from a test suite file. The test case itself can then call the function and make an assertion about the value it got back. In the example test case below, the provided password is too short. If `isValidPassword` is working as expected, it should return `false` since the password is less than 8 characters. The assertion below checks the value.

```
import { isValidPassword } from '../src/utils/user.js'

test('Should reject password shorter than 8 characters', () => {
    const isValid = isValidPassword('abc123')

    expect(isValid).toBe(false)
})
```

Documentation Links

- Expect

Lesson 5: Apollo Client in the Browser: Part I

In this lesson, you'll learn how to use Parcel to get a web app up and running quickly. Parcel comes with a module system and Babel support baked right in, so it couldn't be easier to get started.

You can install Parcel with the following command.

```
npm install parcel-bundler@1.9.7
```

From there, Parcel can be executed using a script in `package.json`. The only required argument is a path to the HTML file for your application. That HTML file can then link in JavaScript files and those files will have access to all the ES6/ES6 features you'd expect.

```
{
  "scripts": {
    "start": "parcel src/index.html"
  },
  "devDependencies": {
    "parcel-bundler": "^1.9.7"
  }
}
```

Then start it up with `npm run start` or `npm start`.

Documentation Links

- Parcel

Lesson 6: Apollo Client in the Browser: Part II

In this lesson, you'll learn how to use Apollo Client to fire off GraphQL operations from the browser. That'll give you everything you need to build out a web-based client for your application.

Apollo Boost

Apollo isn't a single library. It's a collection of libraries for both the client and the server. One of those libraries is Apollo Boost. Apollo Boost is a zero configuration client that gives you everything you need to send GraphQL queries and mutations to a GraphQL server.

First up, install it.

```
npm install apollo-boost@0.1.16
```

From there, you can import and configure it, providing the URL for the GraphQL server you want to communicate with.

```
import ApolloBoost, { gql } from 'apollo-boost'

const client = new ApolloBoost({
    uri: 'http://localhost:4000'
})
```

With the client created, it's time to run an operation. The example below sets up a query operation that fetches the id and name of all users. The name of each user is then rendered to the screen using a bit of DOM manipulation.

```
const getUsers = gql`
    query {
        users {
            id
            name
        }
    }
`

client.query({
    query: getUsers
}).then((response) => {
    let html = ''

    response.data.users.forEach((user) => {
        html += `
            <div>
                <h3>${user.name}</h3>
            </div>
        `
    })

    document.getElementById('users').innerHTML = html
})
```

Documentation Links

- Apollo Boost

Lesson 7: Configuring Jest to Start the GraphQL Server

In this lesson, you'll be setting up Jest to start and stop the GraphQL server. This will ensure you can continue to use a single command to run your test suite.

Configuring Jest

Jest supports over 50 different configuration options. These can be used to fine-tune Jest to fit your needs. For our purposes, we want jest to start and stop our GraphQL server, so we don't have to have it running in a seprate terminal window.

First up, `env-cmd` will be installed allowing Jest to access our test environement variables.

```
npm install env-cmd@8.0.2
```

From there, the test script in `package.json` can be updated to load in the environment variables before starting up the test suite. The Jest options `globalSetup` and `globalTeardown` have also been configured. These options provide a path to a file that Jest will run when it's setting up or tearing down the testing environment.

```
{
  "scripts": {
    "test": "env-cmd ./config/test.env jest --watch"
  },
  "jest": {
    "globalSetup": "./tests/jest/globalSetup.js",
    "globalTeardown": "./tests/jest/globalTeardown.js"
  }
}
```

Both the startup and teardown files should export a single function as shown in the examples below.

```
require('babel-register')
require('@babel/polyfill')
const server = require('../../src/server').default

module.exports = async () => {
    // Do something to start the app up
    global.httpServer = await server.start({ port: 4000 })
}


module.exports = async () => {
    // Do something to tear the app down
    await global.httpServer.close()
}
```

Documentation Links

- Jest configuration options
- env-cmd

Lesson 8: Testing Mutations

In this lesson, you'll write your first test case that tests the GraphQL application. This will allow your test suite to fire off GraphQL operations and then assert something about the response from the server.

The setup below is identical to the setup used when working with Apollo Boost in the browser. First up, the mutation operation is defined. From there, the operation is passed to the client which will send the operation off to the server and get the response. Lastly, an assertion is made.

```
test('Should create a new user', async () => {
    const createUser = gql`
        mutation {
            createUser(
                data: {
                    name: "Andrew",
                    email: "andrew@example.com",
                    password: "MyPass123"
                }
            ){
                token,
                user {
                    id
                }
            }
        }
    `

    const response = await client.mutate({
        mutation: createUser
    })

    const exists = await prisma.exists.User({ id:
response.data.createUser.user.id })
    expect(exists).toBe(true)
})
```

Lesson 9: Seeding the Database with Test Data

In this lesson, you'll learn how to set up the test database with dummy data. This is useful for testing functionality like the ability to login, which will require an existing user to be stored in the database.

Seeding the Database

The goal here is to populate the database with some data that your test cases can use. Using a Jest lifecycle method, you can configure a function to run before or after your test cases run. The test data should be reset before each test case so each time a test runs it's running under the same conditions.

There are 4 main lifecycle methods, `beforeAll`, `beforeEach`, `afterAll`, and `afterEach`. By setting up `beforeEach` as shown below, you'll be able to reset the database between test cases.

```
beforeEach(async () => {
    // Wipe all test data and data that test cases may have added
    await prisma.mutation.deleteManyUsers()

    // Add test data base in
    const user = await prisma.mutation.createUser({
        data: {
            name: 'Jen',
            email: 'jen@example.com',
            password: bcrypt.hashSync('Red098!@#$')
        }
    })
})
```

Documentation Links

- Jest lifecycle methods

Lesson 10: Testing Queries

In this lesson, you'll learn how to write test cases for GraphQL queries. This is now possible because you have test data in the database to actually query.

Testing a query is the same as testing a mutation. You'll need to define a mutation, send that off the server, and assert something about the response. The example below fetches all posts and asserts that only published posts are sent back.

```
test('Should expose published posts', async () => {
    const getPosts = gql`
        query {
            posts {
                id
                title
                body
                published
            }
        }
    `
    const response = await client.query({ query: getPosts })

    expect(response.data.posts.length).toBe(1)
    expect(response.data.posts[0].published).toBe(true)
})
```

Lesson 11: Expecting GraphQL Operations to Fail

In this lesson, you'll learn how to use Jest to expect an operation to fail. This is useful for a lot of situations, including testing that new users can't sign up for an account with an email that's already in use.

Expecting Something to Fail

You typically don't want things to fail, but when testing you often want to make sure that something fails when it's supposed to. Jest provides a `toThrow` assertion you can use to ensure that a given function throws an error.

The test case below uses `toThrow` to check that the function passed to `expect` throws an error. If that function throws an error, the test will pass. If that function doesn't throw an error, the test will fail.

```
test('Should throw an error', async () => {
    expect(() => {
        throw new Error()
    }).toThrow()
})
```

`toThrow` can also be used when testing GraphQL opeartions. The key here is to pass a promise to `expect`. As shown below, you can use `toThrow` to assert that the promise ends up being rejected.

```
test('Should not login with bad credentials', async () => {
    const login = gql`
        mutation {
            login(
                data: {
                    email: "jen@example.com",
                    password: "red098!@#$"
                }
            ){
                token
            }
        }
    `

    await expect(
        client.mutate({ mutation: login })
    ).rejects.toThrow()
})
```

Documentation Links

- toThrow
- rejects

Lesson 12: Supporting Multiple Test Suites and Authentication

In this lesson, you'll be doing a bit of refactoring to break the test cases out into more than one file.

There are no notes for this lesson as no new language features were explored.

Lesson 13: Testing with Authentication: Part I

In this lesson, you'll learn how to test GraphQL operations that require authentication. This will let you write tests that ensure logged-in users are able to access their data and manipulate it.

Setting up Apollo Boost with Authentication

Apollo Boost can be configured to pass along an authentication token to the GraphQL app. This is done by setting up the `request` callback function which allows you to manipulate requests before Apollo Boost sends them off to the server.

The `getClient` function below lets you generate a new client while also providing a JWT to use for authentication. The function itself takes the token and sets up the authorization header for all requests.

```
import ApolloBoost from 'apollo-boost'

const getClient = (jwt) => {
    return new ApolloBoost({
        uri: 'http://localhost:4000',
        request(operation) {
            if (jwt) {
                operation.setContext({
                    headers: {
                        Authorization: `Bearer ${jwt}`
                    }
                })
            }
        }
    })
}

export { getClient as default }
```

Test cases can then generate a new client and interact with the GraphQL app as an authenticated user.

```
test('My test case', async () => {
    const client = getClient(userOne.jwt)
    // Send off an operation that requires authentication
})
```

Lesson 14: Testing with Authentication: Part II

In this lesson, you'll continue to test GraphQL operations that require authentication.

There are no notes for this challenge video as no new language features were explored.

Lesson 15: GraphQL Variables: Part I

In this lesson, you're going to learn about variables in GraphQL. Variables provide a better way to set up dynamic values in your operations.

GraphQL Variables

GraphQL variables provide a mechanism for pulling argument values out of the operation string. This lets you create reusable operations. Each time the operation is executed, it can be executed with a different set of variables values.

The two code sample below show how thing can be done for the `createUser` mutation.

First up, the operation definition needs to define all the variables it should accept as well as the types for those variables. In GraphQL, all your variables will be prefixed with `$` as shown with `$data` below. `$data` is then used as the value for the `data` argument that `createUser` expects.

```
const createUser = gql`
    mutation($data:CreateUserInput!) {
        createUser(
            data: $data
        ){
            token,
            user {
                id
                name
                email
            }
        }
    }
`
```

Variables can now be used with the operation by providing a variables object along with the query or mutation. The snippet below provides the values required for creating a new user.

```
const variables = {
    data: {
        name: 'Andrew',
        email: 'andrew@example.com',
        password: 'MyPass123'
    }
}

const response = await client.mutate({
    mutation: createUser,
    variables
})
```

Lesson 16: GraphQL Variables: Part II

In this lesson, it'll be up to you to use GraphQL variables in your application.

There are no notes for this challenge video as no new language features were explored.

Lesson 17: Testing Comments

In this lesson, it'll be up to you to use what you learned to create some comment test data and write a couple of tests for comments.

There are no notes for this challenge video as no new language features were explored.

Lesson 18: Testing Subscriptions

In this lesson, you'll learn how to test GraphQL subscriptions. It's the last operation to explore testing for.

At this time, Apollo Boost doesn't support subscriptions. This is a problem because subscriptions are essential for real-time applications. As a temporary work around I've put together an alternative version of `getClient.js` linked below.

The goal is to set up a subscription and then manipulate Prisma in a way that should trigger the subscription. From there, you can assert that the data the subscription received looks correct.

Remember that all of this is possible because of the `done` argument setup on the test case function. When the `done` argument is added, Jest will wait for it to be called before considering the test complete. That allows the subscription callback to run the assertions before the test is done.

```
test('Should subscribe to comments for a post', async (done) => {
    const variables = {
        postId: postOne.post.id
    }
    client.subscribe({ query: subscribeToComments, variables }).subscribe({
        next(response) {
            expect(response.data.comment.mutation).toBe('DELETED')
            done()
        }
    })

    await prisma.mutation.deleteComment({ where: { id: commentOne.comment.id
}})
})
```

Documentation Links

- Alternative version of getClient.js

Lesson 19: Test Case Ideas

In this lesson, it'll be up to you to use what you learned to create some comment test data and write a couple of tests for comments.

There are no notes for this challenge video as no new language features were explored.

Documentation Links

- Test case ideas

Section 10: Creating a Boilerplate Project

Lesson 1: Section Intro

In this short section, you'll be ripping apart the blogging application to create a boilerplate GraphQL application. This boilerplate will serve as a great starting point for your next app and it'll let you avoid ever building up an app from an empty directory again.

Lesson 2: Setting up a Test Environment

In this lesson, you'll be duplicating the blogging app and pulling out everything specific to blogging. That'll leave you with a boilerplate GraphQL app with GraphQL, Prisma, and authentication all built-in.

There are no notes for this lecture, but the final boilerplate can be downloaded as part of the lecture resources for this lesson.

Lesson 3: Using the Boilerplate Project

In this lesson, you'll learn how to get started with the boilerplate project. You'll learn how to set it up for the development, test, and production environments.

Using the boilerplate project

1. Duplicate the boilerplate folder.
2. Update the `PRISMA_ENDPOINT` environment variable for all three environments. Use your app name for the service and the correct stage for the stage name.
3. Ensure you have your local (Docker) and production (Prisma Cloud) instances of Prisma running.

4. Deploy Prisma to all three environments. From the `prisma` directoy, use `prisma deploy` with the `-e` flag to provide the path to the correct environment file.
5. Run `npm install` from the project root.
6. Run `npm run test` to run the test suite.
7. Run `npm run dev` to run the development server.
8. Use `heroku create` to create a new Heroku project.
9. Use `heroku config:set KEY=VALUE` to set up all three environment variables on Heroku that exist in `prod.env`.
10. Run `git push heorku master` to deploy your app to production
11. Start building out your app!