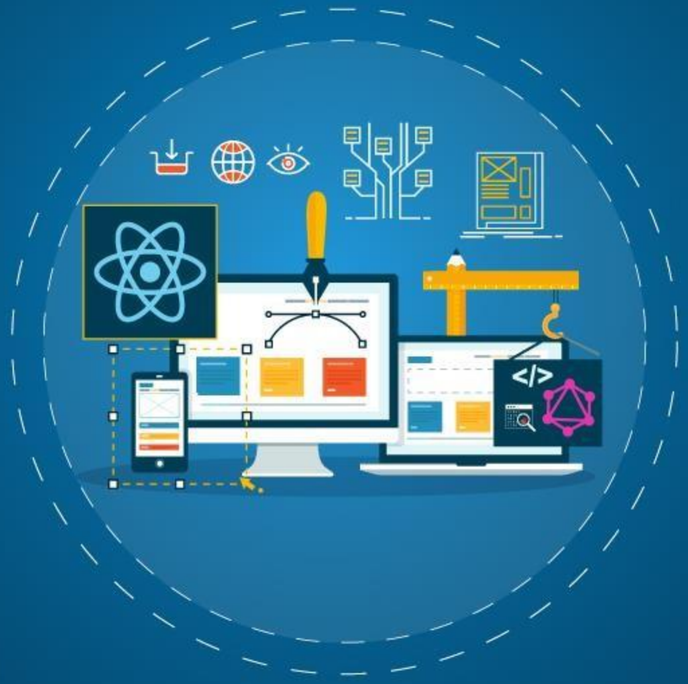
React With Redux Certification Training

COURSE OUTLINE
MODULE 07

1. Introduction to Web Development and React
2. Components and Styling the Application Layout
3. Handling Navigation with Routes
4. React State Management using Redux
5. Asynchronous Programming with Saga Middleware



6. React Hooks
7. **Fetching Data using GraphQL**
8. React Application Testing and Deployment
9. Introduction to React Native
10. Building React Native Applications with APIs

Topics

Following are the topics covered in this module:

- What is GraphQL?
- Cons of Rest API
- Pros of GraphQL
- Frontend backend communication using GraphQL
- Type system
- GraphQL datatypes
- Modifiers
- Schemas
- GraphiQL tool
- Express framework
- NPM libraries to build server side of GraphQL
- Build a GraphQL API
- Apollo client
- NPM libraries to build client side of GraphQL
- How to setup Apollo client
- Fetch space launch data using Apollo-GraphQL

Objectives

After completion of this module you should be able to:

- Understand the role of GraphQL
- Write queries using GraphQL
- Make use of GraphiQL tool to execute queries
- Recognize NPM packages to implement GraphQL queries
- Build a GraphQL API
- Setup Apollo Client
- Establish frontend and backend communication using Apollo-GarapQL

GraphQL

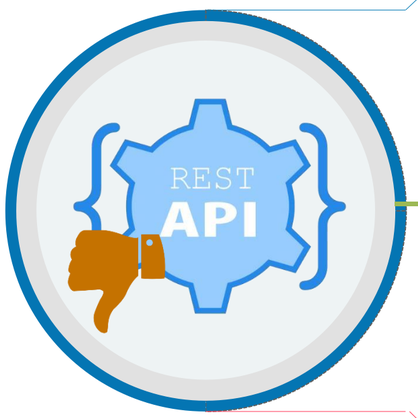
What Is GraphQL?

GraphQL is a query language for APIs and a server-side runtime for executing the queries.

- It is also known as a ***syntax*** that describes how to ask for remote data
- It was originally designed to ***support clients*** requesting data from a server
- It uses a ***type system*** to define data
- It is an ***alternative*** of ***REST architecture***
- Here you write queries using an ***object structure***

Cons Of Rest API

Cons Of REST API



REST API requires ***multiple round trips*** to fetch related resources

It often leads to ***under-fetching*** (not getting everything in one go) or ***over-fetching*** (getting more than what is needed in one go)

As the application grows, the number of ***endpoints*** that are required can also ***increase*** and make the ***code maintenance*** much harder

Requesting Data Using REST API

Client needs to create multiple requests to multiple endpoints to get the data.

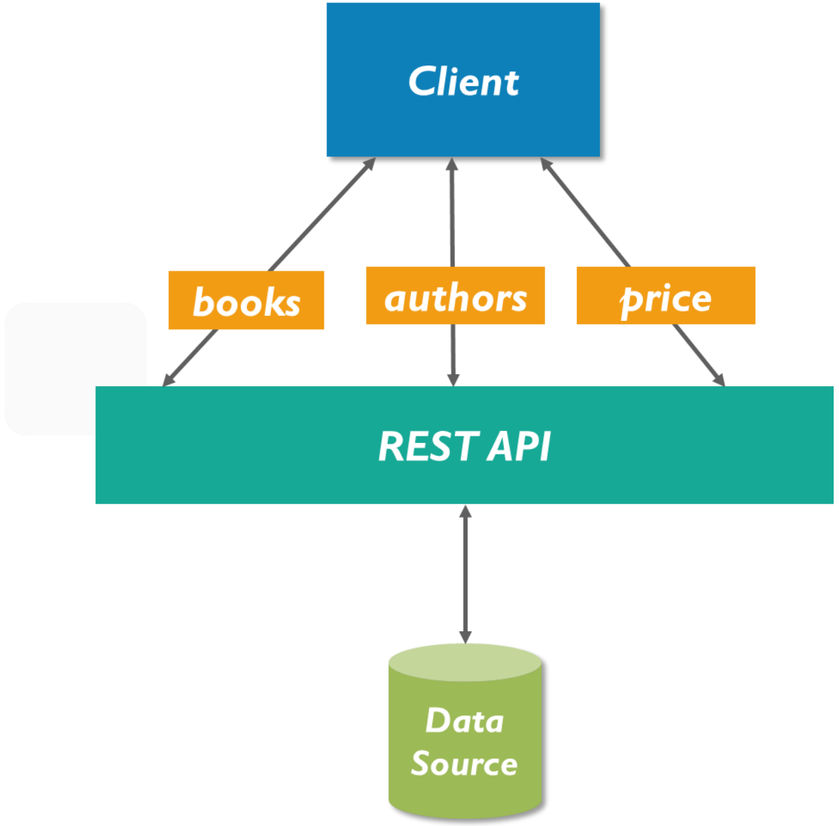


Fig: Fetching data using REST API

Query Structure

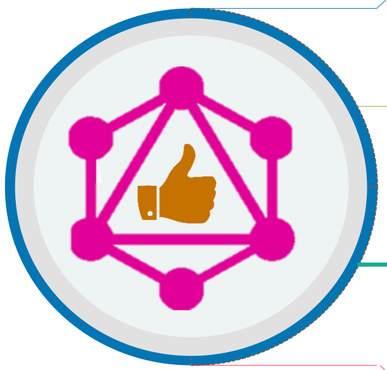
Get all books:
domain.com/books

Get all authors:
domain.com/authors

Get all prices:
domain.com/prices

Pros Of GraphQL

Pros Of GraphQL



Query responses are decided by the ***client*** rather than the server. A GraphQL query returns exactly what a client asks for and nothing more

A GraphQL query itself is ***a hierarchical set*** of fields. The query is shaped just like the data it returns. This helps product engineers to describe data requirements easily

A GraphQL query can be ensured to be valid within a ***GraphQL type system*** at development time allowing the server to make guarantees about the response

GraphQL has ***single endpoint***

Data Fetching By GraphQL

To fetch data we send query in single request. This query specifies the exact fields that client needs.

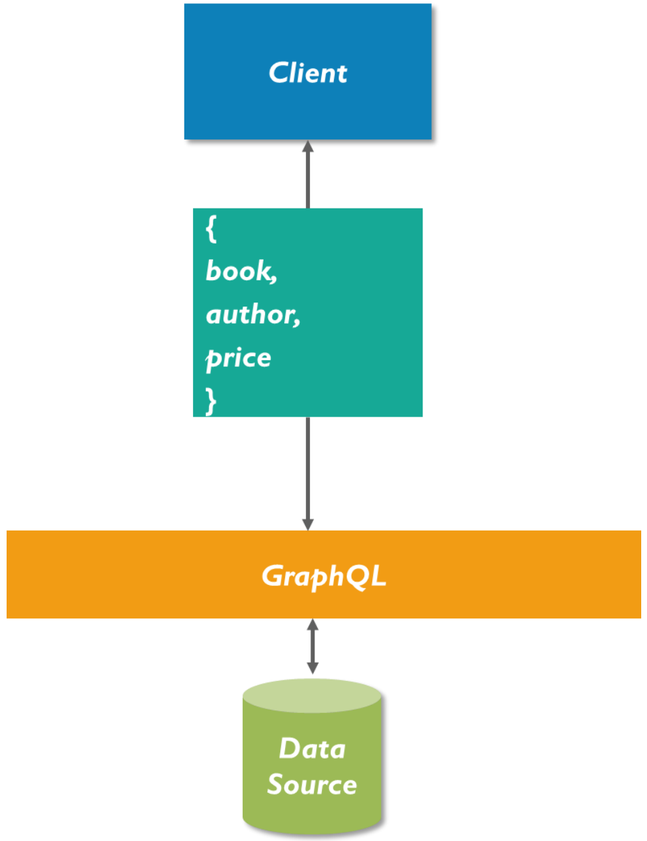


Fig: Fetching data using GraphQL

Query Structure

```
{
  book(id: 1){
        title
        genre
        author{
              name
              age
                 price{
                        }
                 }
                 }
                }
```

Frontend Backend Communication Using GraphQL

Frontend Backend Communication Using GraphQL

- The web server is built on ***Node.Js and Express*** framework
- A request is made to the ***GraphQL Server*** by ***React application*** (built using ***Apollo Client library***) or ***GraphiQL*** browser application
- The query will be ***parsed*** and ***validated*** against a defined ***schema***
- If the request schema passes the validation, then the associated ***resolver functions*** will be executed
- The resolver will contain code to fetch data from an ***API or a database***

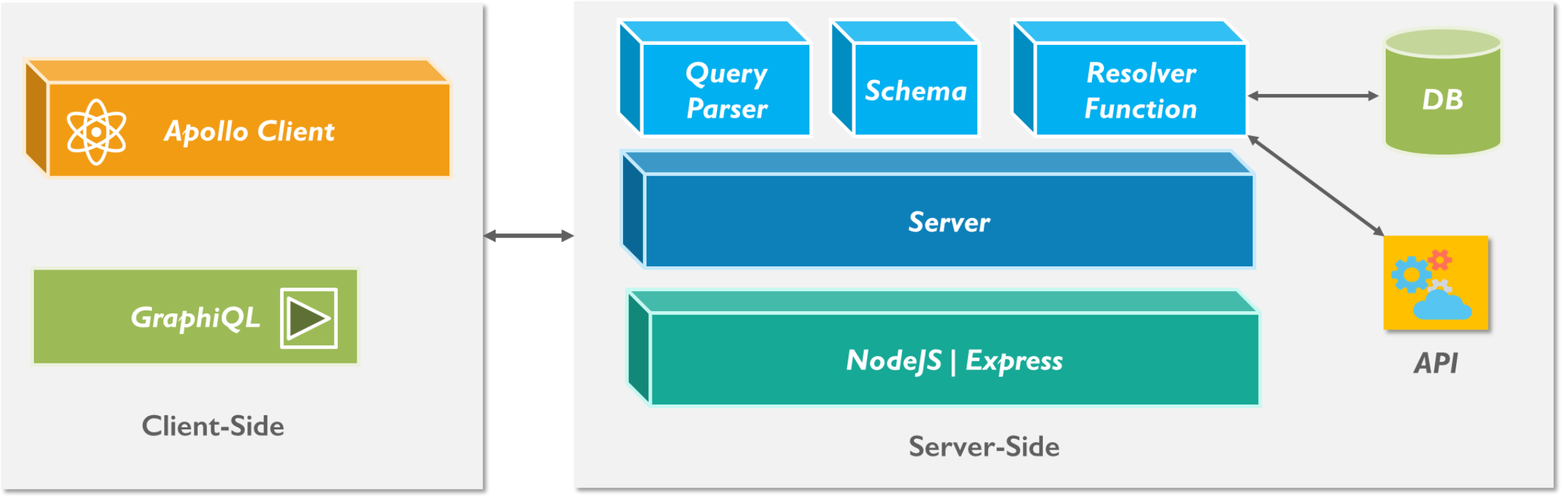


Fig: Fetching Data using Apollo client and GraphQL

Type System

Type System

Type System defines various data types that can be used in a GraphQL application.

- A type system adds ***type safety checks*** to a program
- This validation happens at ***compile time*** or during the ***execution*** of the code
- As per ***dynamic nature*** of JavaScript, you can assign a string to a variable then use this same variable like a number
- A type system will ***flag*** the ***programming mishaps*** and let you know of a possible ***error***
- It has ***fields*** that represent the ***data*** associated with ***each object***

GraphQL Data Types

GraphQL Data Types: Scalar

Scalar

Object

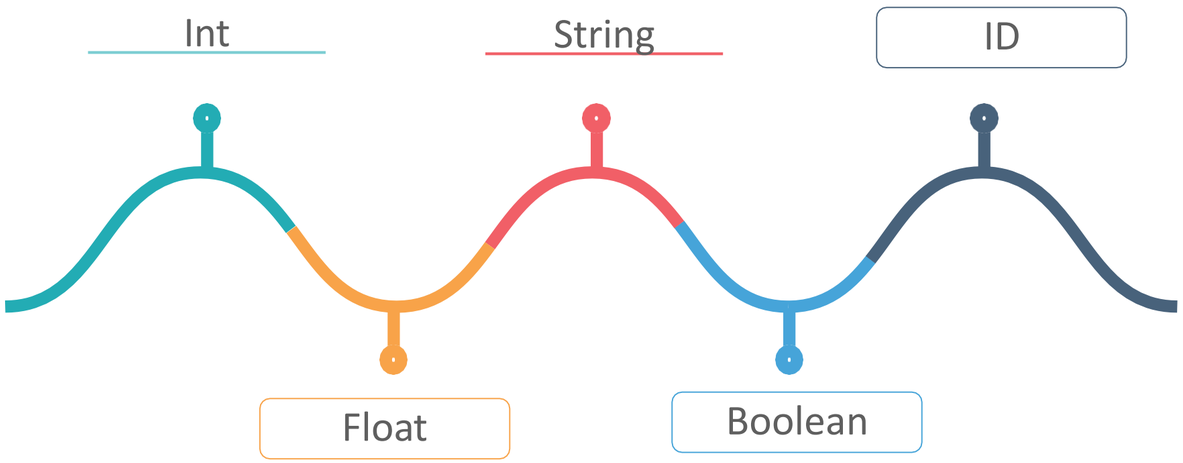
Query

Mutation

Enum

Scalar types are primitive data types that can store only a single value.

The default scalar types of GraphQL are:



Syntax- field: data_type

Example- Author: String

ID is used as a unique identifier to fetch an object or as a key to cache data.

GraphQL Data Types: Object

Scalar

Object

Query

Mutation

Enum

The ***object type*** represents a ***group of fields***.
It is composed of multiple scalar types.

Syntax

```
const{

  field1
  field2
  ....
}=graphql;
```

Example

```
const {
   GraphQLObjectType,
   GraphQLString,
   GraphQLInt,
   GraphQLSchema,
   GraphQLList,
   GraphQLNonNull} = graphql;
```

GraphQL Data Types: Query

Scalar

Object

Query

Mutation

Enum

Query type defines, what piece of information we can get from the data.

Syntax

```
type Query {
  field1: data_type
  field2:data_type
}
```

Example

```
type Query{
person(personID: 5){
  firstname: String
  age: Int
  score:Float
}
}
```

GraphQL Data Types: Mutation

Scalar

Object

Query

Mutation

Enum

Mutations are operations sent to the server to **add, update** or **delete** data. They are comparable to the POST, UPDATE, PATCH and DELETE requests of a REST API,

Syntax

```
type Mutation {
  field1: data_type
  field2(param1:data_type,
param2:data_type,
...
paramN:data_type)
}
```

Example

```
type Mutation{
  addUser( firstName: “Raj”, age:20){
  id
  firstName
  age
}}
```

GraphQL Data Types: Enum

Scalar

Object

Query

Mutation

Enum

An ***Enum*** is similar to a scalar type. They are useful in a situation where the value for a field must be from a prescribed list of options.

Syntax

```
type enum_name{
  value1
  value2
}
```

Example

```
type Days_of_Week{
  SUNDAY
  MONDAY
  TUESDAY
  WEDNESDAY
  THURSDAY
  FRIDAY
  SATURDAY
}
```

Modifier in *GraphQL* is used to specify the server particular return value.

It has two categories:

Type modifiers

List modifiers

Modifiers

Type modifiers

List modifiers

Here we add an ***exclamation*** at the end of each scalar type. This enforce the server to always return ***non null values.***

We can write them as:

```
field: String!

field: Int!

field:Float!

field:Boolean!

field: ID!
```

Modifiers

Type modifiers

List modifiers

Here we add ***square brackets*** around the scalar types.
They are used to enforce the server to always return a ***list of values.***

We can write them as:

```
field: [String]

field: [Int]

field: [Float]

field: [Boolean]

field: [ID]
```

Schemas

Schemas In GraphQL

Schema describes the functionality that is available to the clients which connects to GraphQL server.

- Every schema must have a ***query type***
- Schema has the ability to create ***relationships*** between ***types*** (Example: Companies and Users)
- It defines which ***data-fetching (querying)*** and ***data-manipulation (mutating)*** operations can be executed by the client
- ***GraphQL*** contains a set of types, which completely describe the set of possible data you can query on that service. Then, when queries come in, they are validated with the defined schema and later executed

Example Of Schema

```
const {
    GraphQLObjectType,
    GraphQLString,
    GraphQLInt,
    GraphQLSchema} = graphql;


// Create Object for user
const UserType = new GraphQLObjectType({
    name:'User',
    fields:{
        id: {type: GraphQLString},
        firstName: {type: GraphQLString},
        age:{type: GraphQLInt}
    }
})

//Define Root Query
const RootQuery = new GraphQLObjectType({
    name:'RootQueryType',
    fields:{
        user:{
            type:UserType,
            args: {id:{type:GraphQLString}},
            resolve(parentValue, args){
                return _.find(users,{id:args.id})
}}}})
```

GraphiQL Tool

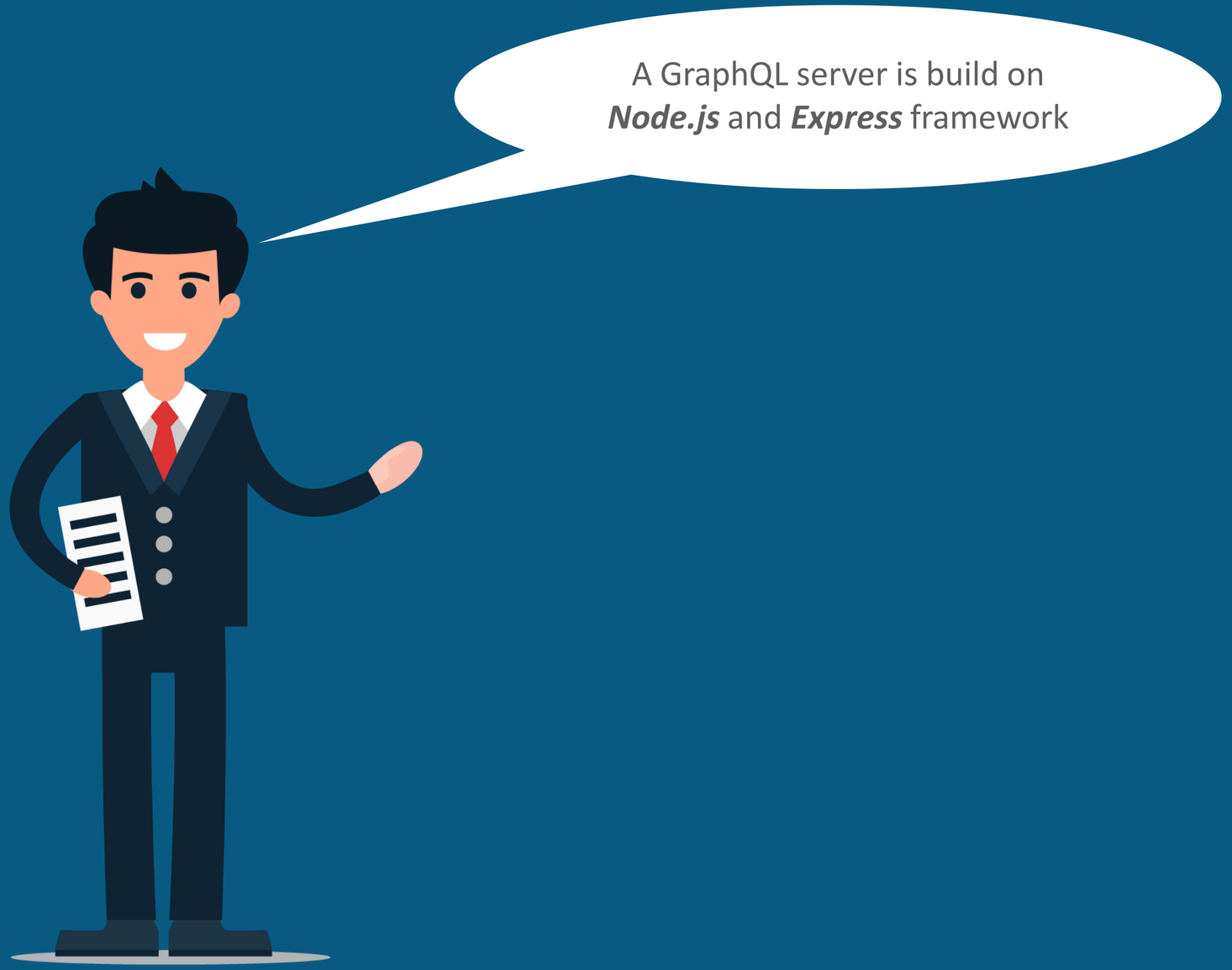
GraphiQL TooL

GraphiQL is a query tool used by the client to make queries to the server.
You add it to your application using: ***npm install --save graphiql***

Click here to execute the query

Query Section

Response Section


A GraphQL server is build on
Node.js and ***Express*** framework

Express

Express is a web framework which behaves like a middleware to help manage servers and routes.

- ***Web Framework*** is used to perform the ***tasks*** of accepting a http request from browser and sending back a HTML response from a server to browser
- Express application uses a call-back function whose parameters are ***request*** and ***response objects***
- ***Request Object*** – The request object represents the HTTP request made by any browser
- ***Response Object*** – The response object represents the HTTP response sent by Express when it gets an HTTP request

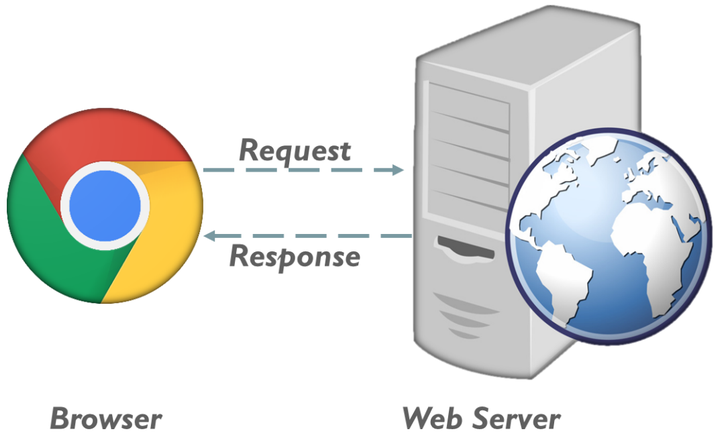


Fig: Working of Express Framework

How To Integrate Express And GraphQL?

Below is the setup used to implement express with GraphQL:

```
const express = require('express');
const graphqlHTTP = require('express-
graphql');

const app = express();

app.use('/graphql', graphqlHTTP({
    schema,
    graphiql: true,
  }),
);

app.listen(4000);
```

An HTTP request method

Path of server

An object of express

The function executed when the route is matched

Executes data fetching and data manipulation

Enables execution of queries from GraphiQL tool

Starts server on port 4000

NPM

NPM

The major ***npm libraries*** being used to get data using GraphQL are:

express

It is used to create web server.

Installation: ***npm i express***

graphql

It installs GraphQL and other core libraries that enables user to leverage GraphQL.

Installation: ***npm i express***

express-graphql

This library enables us to bind together ***graphql*** and ***express***.

Installation: ***npm i express-graphql***

NPM Packages

The major ***npm libraries*** being used to get data using GraphQL are:

axios

This library is used to ***fetch data*** from remote server.

Installation: ***npm i axios***

concurrently

This library is used to run both backend and
frontend on ***single port.***

Installation: ***npm i concurrently***

lodash

This library helps to load an ***array, numbers, objects, strings*** and more.

Installation: ***npm i lodash***

Demo 1: How To Run A GraphQL Server And Build GraphQL API

Demo: Installation Of Packages

Create an folder ***graphqlapi*** and generate package.json file in it using ***npm init.***

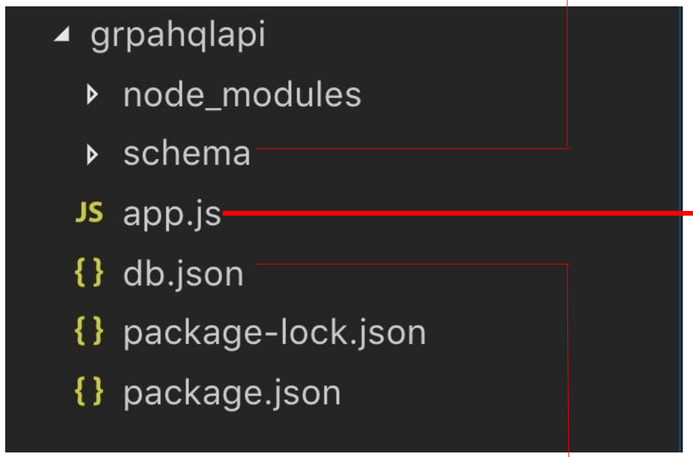
```
Avyaans-MacBook-Pro:module7 avi$ clear

Avyaans-MacBook-Pro:module7 avi$ cd grpahqlapi/
Avyaans-MacBook-Pro:grpahqlapi avi$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

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

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

Press ^C at any time to quit.
package name: (grpahqlapi) graphlqlapi
version: (1.0.0)
description: node with garphql
entry point: (index.js)
test command:
git repository:
keywords: Graphql Nodejs
author: Edureka
license: (ISC)
About to write to /Users/avi/Desktop/folder/EdurekaApp/module7/grpahqlapi/package.json:

{
  "name": "graphlqlapi",
  "version": "1.0.0",
  "description": "node with garphql",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "Graphql",
    "Nodejs"
  ],
  "author": "Edureka",
  "license": "ISC"
}


Is this OK? (yes) yes
Avyaans-MacBook-Pro:grpahqlapi avi$
```

Install the packages: ***express, express-graphql, graphql, axios*** and ***lodash***

```
Avyaans-MacBook-Pro:grpahqlapi avi$ npm install express express-graphql graphql axios lodash
```

Demo: Folder Structure

Create the folder structure as shown below:



Contains GraphQL object types and static data

Main application folder which contains integration of express and GraphQL schemas

Contains the json data to be displayed

Demo: Add nodemon

We need to start application in Production and Dev mode. Install nodemon using ***npm i nodemon***. Add the below scripts in ***package.json*** file.

```
pp.js        {} db.json        {} package.json ×
{
  "name": "graphlqlapi",
  "version": "1.0.0",
  "description": "node with garphql",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "dev":"nodemon app.js",
    "start":"node app.js"
  },
  "keywords": [
    "Graphql",
    "Nodejs"
```

Demo: app.js

In ***app.js*** file add the below snippet, setup the ***express*** framework to run the GraphQL server.

JS app.js × {} package.json

```javascript
const express = require('express');
const expressGraphQL = require('express-graphql');
const port = 4800;

const app = express();


app.use('/graphql', expressGraphQL({
    graphiql:true
}))

app.listen(port,() => {
    console.log(`listing to port ${port}`)
})
```

Configured port to run server

Single endpoint to get GraphQL data

Enable execution of query from GraphiQL tool

Starts the server on specified port

Demo: Start The Server

To start the server execute the command: ***npm start*** in terminal.
On checking the browser you may get the below error:

localhost:4800/graphql

```
// 20190521095209
// http://localhost:4800/graphql

{
  "errors": [
    {
      "message": "GraphQL middleware options must contain a schema."
    }
  ]
}
```

Demo: Schema.js

Define the object type in ***schema.js*** file.

JS app.js | JS schema.js × | {} db.json

```
1  const graphql = require('graphql');
2  const _ = require('lodash');
3
4  const {
5      GraphQLObjectType,
6      GraphQLString,
7      GraphQLInt,
8      GraphQLSchema} = graphql;
```

Demo: Integration Of Schema And Express

Add ***schema*** in ***app.js*** file.

app.js | db.json | package.json

```js
const express = require('express');
const expressGraphQL = require('express-graphql');
//Add Schema
const schema = require('./schema/schema')
const port = 4800;

const app = express();

// Middle ware route to use graphl Playgorund
app.use('/graphql', expressGraphQL({
    schema,
    graphiql:true
}))

app.listen(port,() => {
    console.log(`listing to port ${port}`)
})
```

Demo: Root Query

Create an object for the ***user***, where we define the data type of keys using ***GrpahQL Object type***.

JS app.js | JS schema.js × | JS schema_bk.js

schema > JS schema.js > ...

```js
// Create Object for user
const UserType = new GraphQLObjectType({
    name:'User',
    fields:{
        id: {type: GraphQLString},
        firstName: {type: GraphQLString},
        age:{type: GraphQLInt}
    }
})


const RootQuery = new GraphQLObjectType({
    name:'RootQueryType',
    fields:{
        user:{
            type:UserType,
            args: {id:{type:GraphQLString}},
            resolve(parentValue, args){
                return _.find(users,{id:args.id})
            }
        }
    }
})
```

User object, defined to validate the query

Root query defines the user data and allows to create the query

Resolver function to handle the query

Demo: Export Root Query

Define a ***export*** object to export the ***RootQuery*** in order to import it in app.js file.

```
const RootQuery = new GraphQLObjectType({
    name:'RootQueryType',
    fields:{
        user:{
            type:UserType,
            args: {id:{type:GraphQLString}},
            resolve(parentValue, args){
                return _.find(users,{id:args.id})
            }
        }
    }
})



module.exports = new GraphQLSchema({
    query: RootQuery
})
```

Demo: db.json

In order to build an API, collect the JSON data (sample data) in db.json file.

JS app.js | JS schema.js | {} db.json

{} db.json > ...

```
1  {
2    "users": [
3      {
4        "id": "23",
5        "firstName": "Bill",
6        "age": 20,
7        "companyId": "1"
8      },
9      {
10       "id": "47",
11       "firstName": "John",
12       "age": 22,
13       "companyId": "2"
14     },
15     {
16       "id": "48",
17       "firstName": "Andy",
18       "age": 11,
19       "companyId": "2"
20     },
```

Demo: JSON Server

Install a JSON server using***: npm i json-server.***

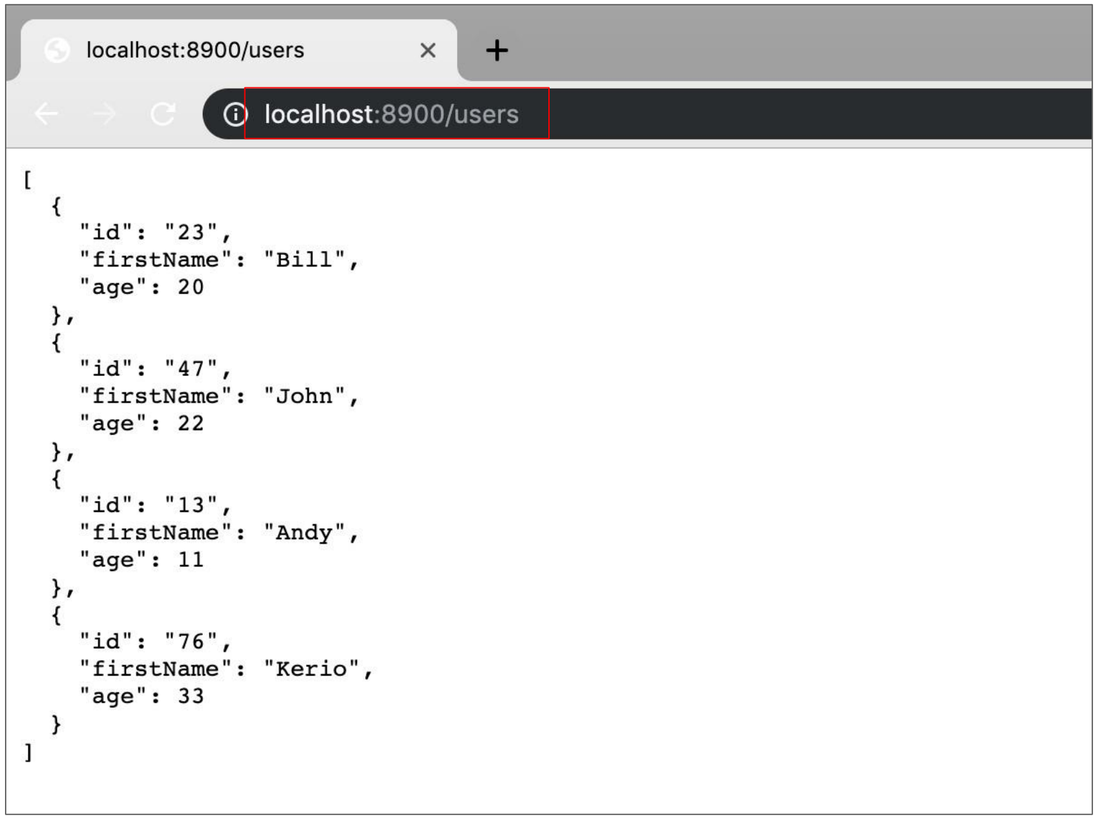
```
Avyaans-MacBook-Pro:grpahqlapi avi$ sudo npm install -g json-server
```

By using '***json-server --watch db.json --port 8900***' build an API with user route.

```
Avyaans-MacBook-Pro:grpahqlapi avi$ json-server --watch db.json --port 8900

  \{^_^}/ hi!

  Loading db.json
  Done

  Resources
  http://localhost:8900/users

  Home
  http://localhost:8900

  Type s + enter at any time to create a snapshot of the database
  Watching...
```

Demo: Test API

Verify the working of the API by running the ***url*** in browser.



Demo: API Consumption Via GraphQL

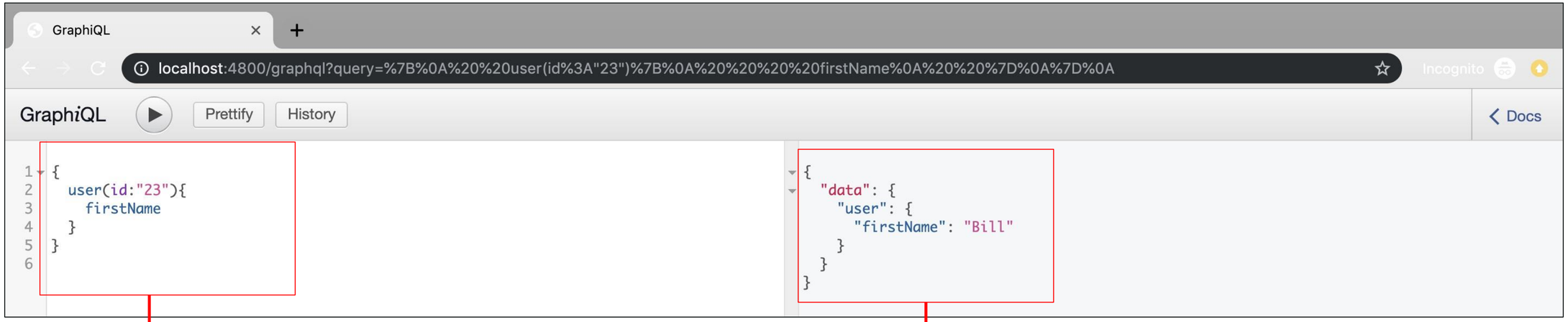
Make a call to the API using ***Axios,*** add the ***API link*** in the resolver function.

JS app.js | JS schema_bk.js | JS schema.js × | {} db.json | {} package.json

```
21
22
23
24
25
26
27  // Getting data from API
28  const RootQuery = new GraphQLObjectType({
29      name:'RootQueryType',
30      fields:{
31          user:{
32              type:UserType,
33              // taking user search id
34              args: {id:{type:GraphQLString}},
35              resolve(parentValue, args){
36                  return axios.get(`http://localhost:8900/users/${args.id}`)
37                  .then(resp => resp.data)
38              }
39          }
40      }
41  })
42
```

Demo: Check The Queries In GraphiQL Tool

Execute the application code using ***npm start,*** open GraphiQL tool in browser using: ***'loaclhost:4800/graphql'***



Run the query

Check the output

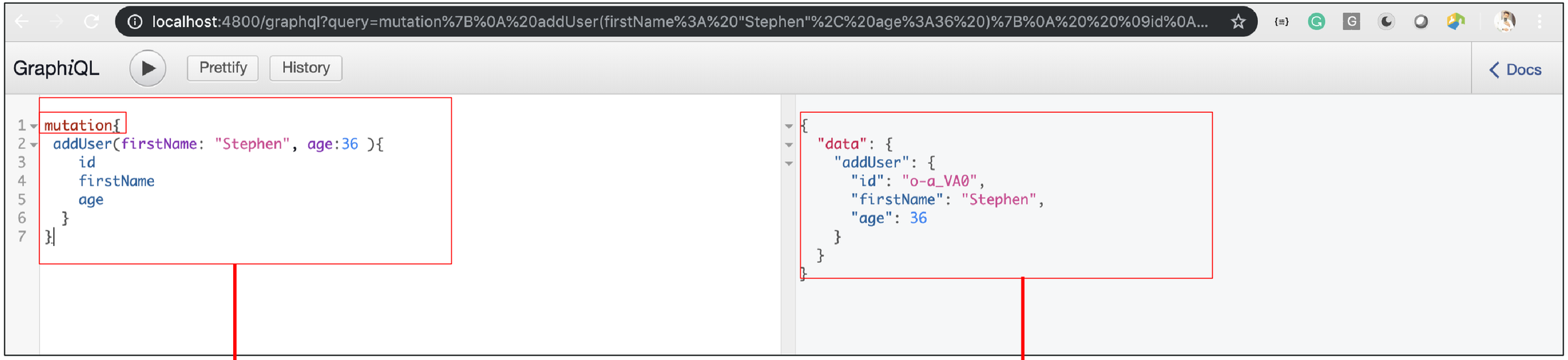
Demo: Mutation

In order to edit the data present at API link add ***mutation.***

```
// Mutation
const mutation = new GraphQLObjectType({
    name:'Mutation',
    fields: {
        addUser: {
            type: UserType,
            args: {
                firstName: {type: new GraphQLNonNull(GraphQLString)},
                age:{type: new GraphQLNonNull(GraphQLInt)},
                companyId: {type: GraphQLString}
            },
            resolve(parentValue, {firstName, age}) {
                return axios.post('http://localhost:8900/users' ,{firstName, age})
                .then(res => res.data)
            }
        }
    }
})

module.exports = new GraphQLSchema({
    query: RootQuery,
    mutation: mutation
})
```

Demo: Query To Edit Data

Now by using ***mutation*** we can make a post call to API to add the data.



Query to send data

Output

Demo: Verify Addition Of Data At The API Link

Run the API link in browser to check the addition of data.

localhost:8900/users


```
// 20190525210753
// http://localhost:8900/users

[
  {
    "id": "23",
    "firstName": "Bill",
    "age": 20,
    "companyId": "1"
  },
  {
    "id": "47",
    "firstName": "John",
    "age": 22,
    "companyId": "2"
  },
  {
    "id": "48",
    "firstName": "Andy",
    "age": 11,
    "companyId": "2"
  },
  {
    "id": "49",
    "firstName": "Kerio",
    "age": 33,
    "companyId": "3"
  },
  {
    "firstName": "Stephen",
    "age": 36,
    "id": "o-a_VA0"
  }
```

Apollo Client

Apollo Client

Apollo client is a client-side library that leverages the power of a GraphQL API to handle data fetching from a GraphQL sever.

- The client is designed to help developer ***quickly build*** an UI, that ***fetches*** the data with GraphQL and can be used with any JavaScript frontend technology

- ***Caching*** is one of the major features of Apollo client

- Apollo Client takes care of ***requesting*** and ***caching*** application data, as well as updating application UI

NPM

NPM

The major ***npm libraries*** being used to configure Apollo client with GraphQL server are:

react-apollo

Provides necessary modules to fetch data from GraphQL server.

Installation: ***npm install react-apollo***

apollo-boost

This Library contains the required modules to setup Apollo Client.

Installation: ***npm install apollo-boost***

cors (cross-origin resource sharing)

It makes an API open to cross-site requests.

Installation: ***npm install cors***

Implement: ***app.use(cors());***

ApolloClient and ApolloProvider

ApolloClient: creates an instance and connect it to GraphQL server.

ApolloProvider: wraps our application code and sends to the Apollo client.

How To Setup Apollo Client?

How To Setup Apollo Client?

01 Import the ***packages***

```
import gql from 'graphql-tag’;
import ApolloClient from 'apollo-boost’;
import { Query } from 'react-apollo';
```

Here, ***gql*** is a template literal tag, which allows us to create a schema (as per ES6 rules)

02 Create ***Apollo Client Instance***, ànd pass URI to connect to the server

```
const client = new ApolloClient({uri: '/graphql'});
```

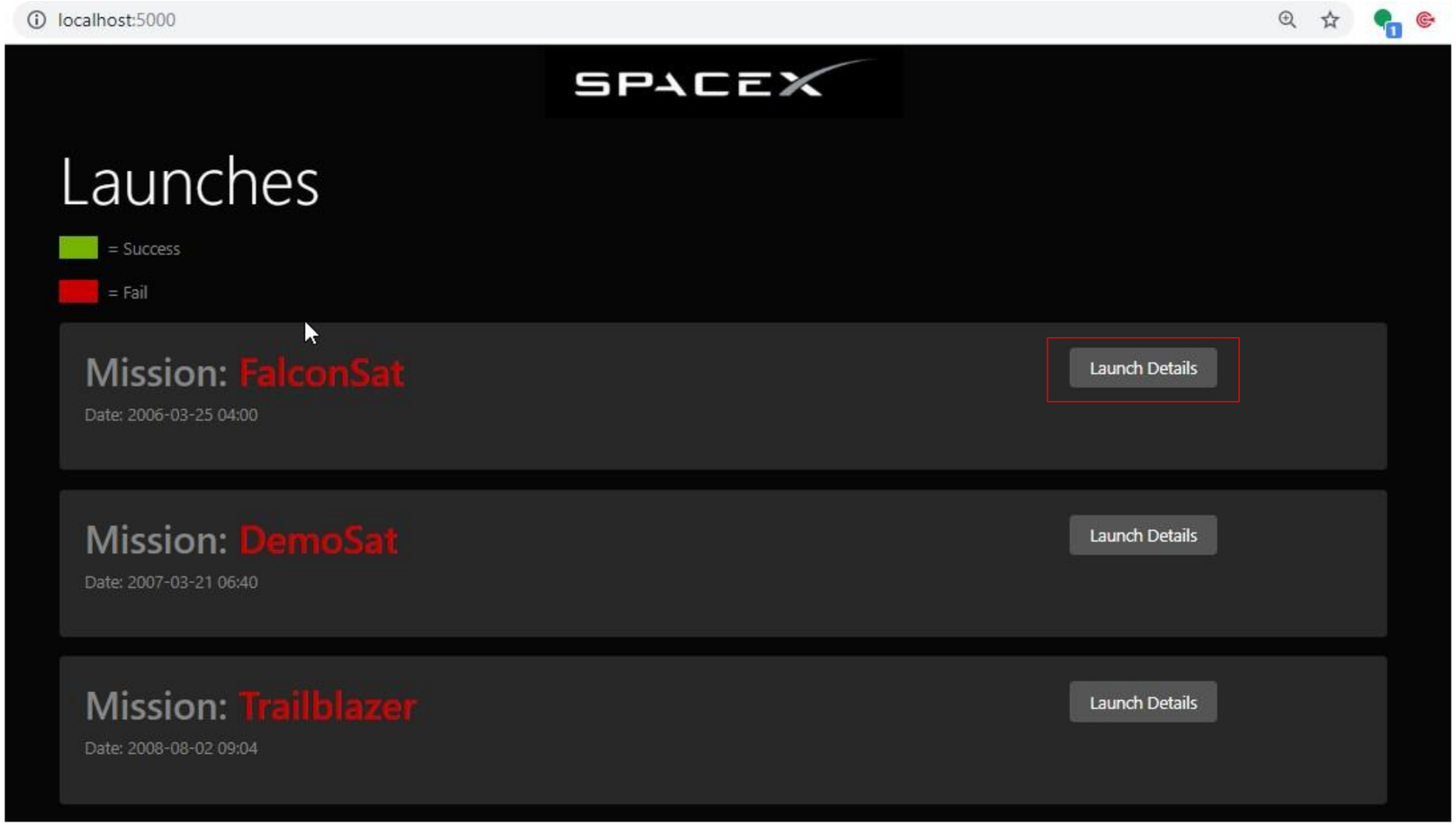
03 Execute the ***query*** with client instance, pass query as an object. The query should be a string parsed by the gql tag

```
const LAUNCH_QUERY = gql`
  query LaunchQuery($flight_number: Int!) {

    launch(flight_number: $flight_number) {
      flight_number
      mission_name
      launch_year
      launch_success
      launch_date_local

      rocket {
        rocket_id
        rocket_name
        rocket_type
      }}}
`;
```

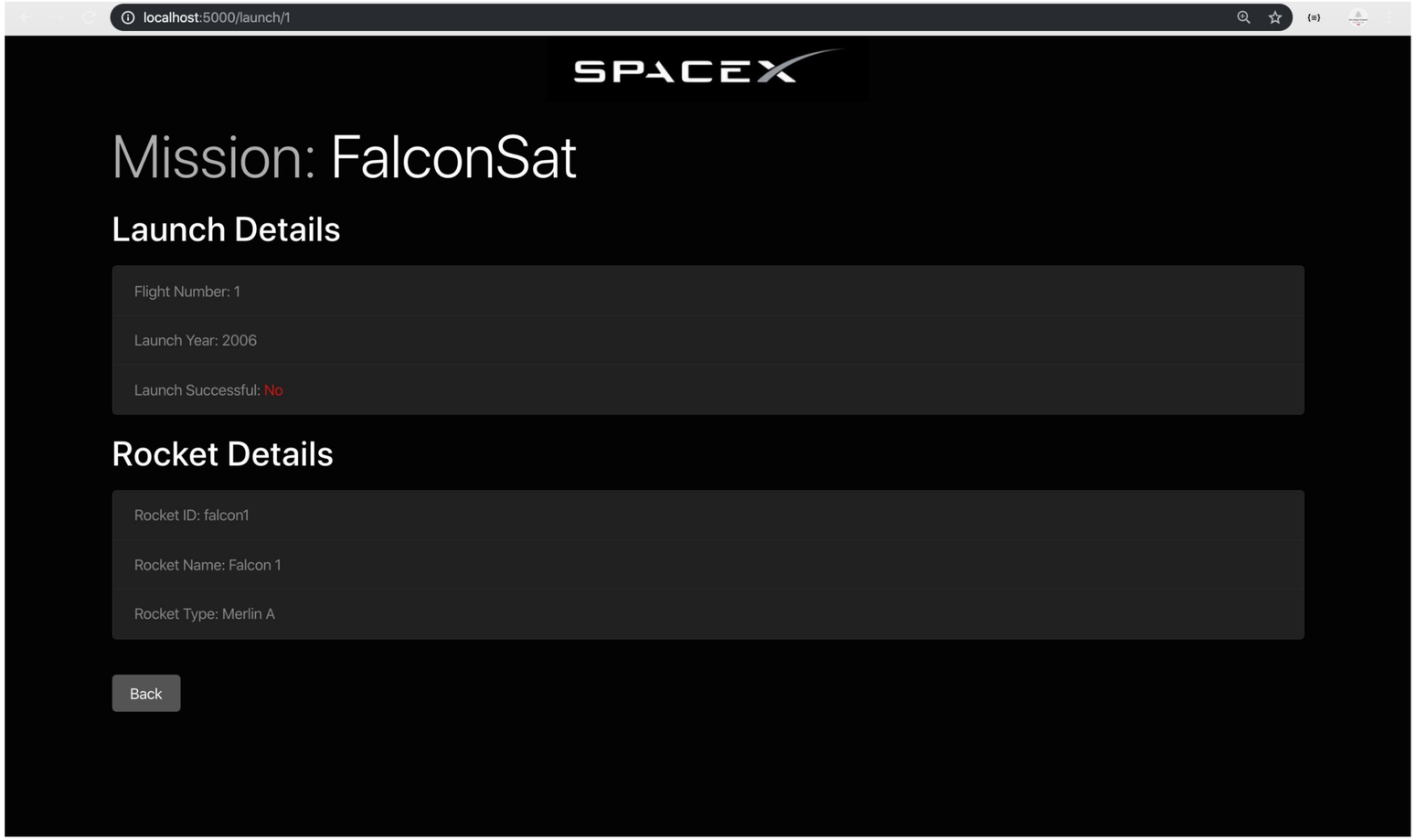
Demo 2: Fetch Space Launch Data Using Apollo-GraphQL


In this demo, you will learn how to ***connect Apollo Client to the GraphQL server*** and execute queries.

Demo: Output Of Launch Component



Demo: Output Of Details Page



Questions


Comments
Ratings
Suggestions
FEEDBACK
Survey
Likes
Ideas