
Implementing
schema resolvers
LEARNING
VOYAGE

Implementing schema resolvers

This lesson covers

- Using Node.js drivers for PostgreSQL and MongoDB
- Using an interface to communicate with a GraphQL service
- Making a GraphQL schema executable
- Creating custom object types and handling errors

Running the development environment

- To let you focus on the GraphQL skills in this course's project, I prepared a Git repository that contains all the non-GraphQL things that you need to follow up with the project.
- We will use this repository in lessons 5–10. It has the skeleton for both the API server and the skeleton for the web server .
- Clone that repo

git clone https://az.dev/gia-repo graphql

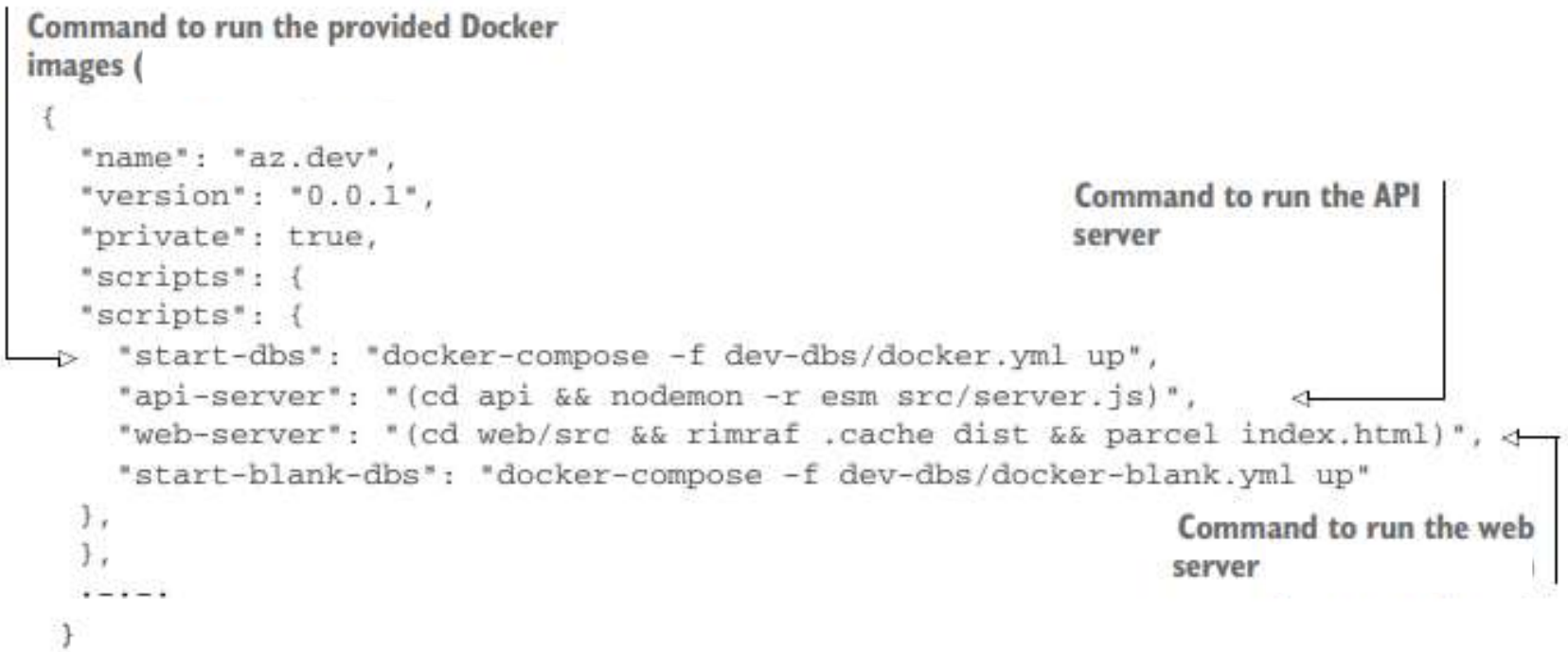
Running the development environment

- Cloning the repo creates the graphql directory under your current working directory.

- There, the first step is to install the initial packages that are used by the repo.

```
$ cd graphql
$ npm install
```

Running the development environment



Node.js packages

- For a web server to host the project's API endpoint, we will use Express.js.
- There are a few other Express-related packages that we need.
- All these packages are already installed in the repo's starting point.
- To implement the GraphQL API server, we need two new packages.

```
$ npm install graphql express-graphql
```

Environment variables

- Under the api directory is a .env file that contains the default environment variables we need in this project.
- If you do not plan to use any of the project's defaults, you'll need to change these variables.
- This file is automatically loaded, and its variables are exported in api/src/config.js.

Setting up the GraphQL runtime

- Suppose we are creating a web application that needs to know the exact current time the server is using (and not rely on the client's time).
- We would like to be able to send a query request to the API server as follows

```
{
        currentTime
}
```

Setting up the GraphQL runtime

- To respond to this query, let's make the server use an ISO UTC time string in the HH:MM:SS format.

```
{
        currentTime: "20:32:55"
}
```

Creating the schema object

For the very first GraphQL.js example, we need to use two of the functions exported by the graphql package:

- The buildSchema function that builds a schema from a schema language text.
- The graphql function to execute a GraphQL query against that generated schema. To avoid confusion, I'll refer to it as the graphql executor function.

Creating the schema object

- Create a schema directory under api/src, and put the following index.js file in it.

```
import { buildSchema } from 'graphql';
```

Creating the schema object

- Here's the schema text for the simple example schema we're building.

```
export const schema = buildSchema(`
  type Query {
    currentTime: String!
  }
`);
```

Creating resolver functions

- We have a schema, and we can validate any request against it if we need to, but we have not told the GraphQL service what data to associate with the currentTime field in that schema.
- If a client asks for that field, what should the server response be? This is the job of a resolver function.
- Each field defined in the schema needs to be associated with a resolver function.

Creating resolver functions

- Let's create an object to hold the many resolver functions we will eventually have.
- Here's one way to implement the currentTime resolver logic.

```
export const rootValue = {
  currentTime: () => {
    const isoString = new Date().toISOString();
    return isoString.slice(11, 19);
  },
};
```

The ISO format is fixed. The 11-19 slice is the time part.

Executing requests

- The graphql executor function can be used for this purpose.
- We can test that in api/src/server.js, Add the following import line.

```
import { graphql } from 'graphql';
```

Executing requests

- ThE graphql executor function accepts a list of arguments: the first is a schema object, the second is a source request (the operation text), and the third is a rootValue object of resolvers.

- Here's an example of how you call it.

```
graphql(schema, request, rootValue);
```

Executing requests

- In JavaScript, we can access the resolved value of this promise by putting the keyword await in front of it and wrapping the code with a function labeled with the async keyword.

```
async () => {
  const resp = await graphql(schema, request, rootValue);
};
```

Executing requests

- The request text is something the clients of this API server will supply.
- They'll do that eventually over an HTTP(S) channel, but for now, we can read it directly from the command line as an argument.
- We'll test the server.js file this way

```
$ node -r esm api/src/server.js "{ currentTime }"
```

This command will work after you implement the next code change. The -r esm part enables working with ECMAScript modules on older versions of Node.js.

Executing requests

- Here's the full code snippet in api/src/server.js that we can use to carry out this test.

```
import { graphql } from 'graphql';
import { schema, rootValue } from './schema';

const executeGraphQLRequest = async request => {
  const resp = await graphql(schema, request, rootValue);
  console.log(resp.data);
};

executeGraphQLRequest(process.argv[2]);
// .-.-.
```

Executing requests

- This example is complete! You can test it with the command and you should see the server report the time in UTC:

```
$ node -r esm api/src/server.js "{ currentTime }"
[Object: null prototype] { currentTime: '18:35:10' }
```

Communicating over HTTP

- We're going to use the express package to create an HTTP server and the expressgraphql package to wire that server to work with the GraphQL service that we have so far
- Remove the executeGraphQLRequest function and the graphql executor function (in api/src/server.js).
- Instead, import the graphqlHTTP named export from the expressgraphql package.

```
import { graphqlHTTP } from 'express-graphql';
import { schema, rootValue } from './schema';

// Uncomment the code to run a bare-bone Express server

import express from 'express';
import bodyParser from 'body-parser';
import cors from 'cors';
import morgan from 'morgan';

import * as config from './config';

async function main() {
  // .-.-.
}

main();
```

Communicating over HTTP

- The provided main function has an example of a server.get call.
- Here is the signature of the server.VERB methods and an example of what you can do within it.

```
server.use('/', (req, res, next) => {
  // Read something from req
  // Write something to res
  // Either end things here or call the next function
});
```

```
// ·-·-·

async function main() {
  // ·-·-·

  // Replace the example server.use call with:
  server.use(
    '/',
    graphqlHTTP({
      schema,
      rootValue,
      graphiql: true,
    })
  );

  server.listen(config.port, () => {
    console.log(`Server URL: http://localhost:${config.port}/`);
  });
}

main();
```

Communicating over HTTP

- Let's test. Start the API server with the following command.

$ npm run api-server

- You should see this message:

Server URL: http://localhost:4321/

Communicating over HTTP

```
{
  currentTime
}
```

```
{
  "data": {
    "currentTime": "16:26:40"
  }
}
```

Building a schema using constructor objects

- The GraphQL schema language is a great programming-language-agnostic way to describe a GraphQL schema.

- It's a human-readable format that's easy to work with, and it is the popular, preferable format for describing your GraphQL schemas, However, it has some limitations.

The Query type

- To create a GraphQL schema using this method, we need to import a few objects from the graphql package, as follows.

```
import {
  GraphQLSchema,
  GraphQLObjectType,
  GraphQLString,
  GraphQLInt,
  GraphQLNonNull,
} from 'graphql';
```

The Query type

- For example, to instantiate a schema object, you just do something like this.

```
const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'Query',
    fields: {
      // Root query fields are defined here
    }
  }),
});
```

```
const QueryType = new GraphQLObjectType({
  name: 'Query',
  fields: {
    currentTime: {
      type: GraphQLString,
      resolve: () => {
        const isoString = new Date().toISOString();
        return isoString.slice(11, 19);
      },
    },
  },
});

export const schema = new GraphQLSchema({
  query: QueryType,
});
```

- To test this code, we need to remove the rootValue concept from api/src/server.js.

```
// ...-...
import { schema } from './schema';
// ...-...

async function main() {
  // ...-...
  server.use(
    '/',
    graphqlHTTP({
      schema,
      graphiql: true,
    }),
  );
  server.listen(config.port, () => {
    console.log(`Server URL: http://localhost:${config.port}/`);
  });
}

main();
```

Remove the rootValue object

Field arguments

```
{
  sumNumbersInRange(begin: 2, end: 5)
}
```

```
{
  "data": {
    "sumNumbersInRange": 14
  }
}
```

Field arguments

```
fields: {
  // ...

  sumNumbersInRange: {
    type: new GraphQLNonNull(GraphQLInt),
    args: {
      begin: { type: new GraphQLNonNull(GraphQLInt) },
      end: { type: new GraphQLNonNull(GraphQLInt) },
    },
    resolve: function (source, { begin, end }) {
      let sum = 0;
      for (let i = begin; i <= end; i++) {
        sum += i;
      }
      return sum;
    },
  },
},
```

Field arguments

- The resolver function simply loops over the range, computes the sum, and returns it.
- Use the following query to test the new field this API now supports.

```
{
        sumNumbersInRange(begin: 2, end: 5)
}
```

Custom object types

- let's make it support two leaf fields for the sum and count of the whole numbers in the range.
- Here's how the new numbersInRange field will be queried.

```
{
  numbersInRange(begin: 2, end: 5) {
    sum
    count
  }
}
```

- Create a new directory api/src/schema/types, and create a numbers-in-range.js file there to implement the NumbersInRange type.

```
import {
  GraphQLObjectType,
  GraphQLInt,
  GraphQLNonNull,
} from 'graphql';

const NumbersInRange = new GraphQLObjectType({
  name: 'NumbersInRange',
  description: 'Aggregate info on a range of numbers',
  fields: {
    sum: {
      type: new GraphQLNonNull(GraphQLInt),
    },
    count: {
      type: new GraphQLNonNull(GraphQLInt),
    },
    },
  });

  export default NumbersInRange;
```

Custom object types

```
// .-.-.

export const numbersInRangeObject = (begin, end) => {
  let sum = 0;
  let count = 0;
  for (let i = begin; i <= end; i++) {
    sum += i;
    count++;
  }
  return { sum, count };
};
```

```
// ----
import NumbersInRange from './types/numbers-in-range';
import { numbersInRangeObject } from '../utils';

  const QueryType = new GraphQLObjectType({
    name: 'Query',
    fields: {
      // ----

      // Remove the sumNumbersInRange field

      numbersInRange: {
        type: NumbersInRange,
        args: {
          begin: { type: new GraphQLNonNull(GraphQLInt) },
          end: { type: new GraphQLNonNull(GraphQLInt) },
        },
        resolve: function (source, { begin, end }) {
          return numbersInRangeObject(begin, end);
        },
      },
    },
  });
```

That's it. If you test the API now, you should be able to execute a query like the following:

```
{
  numbersInRange(begin: 2, end: 5) {
    sum
    count
  }
}
```

And you will get this response:

```
{
  "data": {
    "numbersInRange": {
      "sum": 14,
      "count": 4
    }
  }
}
```

Custom errors

```
{
  numbersInRange(begin: 2) {
    sum
    count
  }
}
```

```
{
  "errors": [
    {
      "message": "Field \"numbersInRange\" argument \"end\" of type \"Int!\" is required, but it was not provided.",
      "locations": [ ↔ ]
    }
  ]
}
```

Custom errors

```
{
  numbersInRange(begin: "A", end: "Z") {
    sum
    count
  }
}
```

```
{
  "errors": [
    {
      "message": "Int cannot represent non-integer value: \"A\"",
      "locations": [↔]
    },
    {
      "message": "Int cannot represent non-integer value: \"Z\"",
      "locations": [↔]
    }
  ]
}
```

Custom errors

```
{
  numbersInRange(begin: 2, end: 5) {
    sum
    count
    avg
  }
}
```

```
{
  "errors": [
    {
      "message": "Cannot query field \"avg\" on type \"NumbersInRange\".",
      "locations": [ ↔ ]
    }
  ]
}
```

Custom errors

```
{
  numbersInRange(begin: 5, end: 2) {
    sum
    count
  }
}
```

```
{
  "data": {
    "numbersInRange": {
      "sum": 0,
      "count": 0
    }
  }
}
```

Custom errors

- We do the check in the resolver function for the numbersInRange field and throw an error with our custom message.

```
export const numbersInRangeObject = (begin, end) => {
  if (end < begin) {
    throw Error(`Invalid range because ${end} < ${begin}`);
  }
  // ----
};
```

Custom errors

```
{
  numbersInRange(begin: 5, end: 2) {
    sum
    count
  }
}
```

```
{
  "errors": [
    {
      "message": "Invalid range because 2 < 5",
      "locations": [↔],
      "path": [
        "numbersInRange"
      ]
    }
  ],
  "data": {
    "numbersInRange": null
  }
}
```

Custom errors

```
{
  numbersInRange(begin: 5, end: 2) {
    sum
    count
  }
  currentTime
}
```

```
{
  "errors": [
    {
      "message": "Invalid range because 2 < 5",
      "locations": [↔],
      "path": [
        "numbersInRange"
      ]
    }
  ],
  "data": {
    "numbersInRange": null,
    "currentTime": "21:51:08"
  }
}
```

Generating SDL text from object-based schemas

```
import {
  // ....
  printSchema,
} from 'graphql';
// ....

export const schema = new GraphQLSchema({
  query: QueryType,
});
console.log(printSchema(schema));
```

Generating SDL text from object-based schemas

Here's what you'll see.

```
type Query {
  currentTime: String
  numbersInRange(begin: Int!, end: Int!): NumbersInRange
}
"""Aggregate info on a range of numbers"""
type NumbersInRange {
  sum: Int!
  count: Int!
}
```

Generating SDL text from object-based schemas

- My favorite part about this conversion is how the arguments to the numbersInRange field are defined in the schema language format:

```
(begin: Int!, end: Int!)
```

- Compare that with:

```
args: {
        begin: { type: new GraphQLNonNull(GraphQLInt) },
        end: { type: new GraphQLNonNull(GraphQLInt) },
}
```

```
"""The root query entry point for the API"""
type Query {
  "The current time in ISO UTC"
  currentTime: String

  """
  An object representing a range of whole numbers
  from "begin" to "end" inclusive to the edges
  """
  numbersInRange(
    "The number to begin the range"
    begin: Int!,
    "The number to end the range"
    end: Int!
  ): NumbersInRange!
}
"""Aggregate info on a range of numbers"""
 type NumbersInRange {
   "Sum of all whole numbers in the range"
   sum: Int!
   "Count of all whole numbers in the range"
   count: Int!
 }
```

The schema language versus the object-based method

- The schema language enables front-end developers to participate in designing the API and, more important, start using a mocked version of it right away.
- The frontend people on your team will absolutely love it.
- It enables them to participate in designing the API and, more important, start using a mocked version of it right away.
- The schema language text can serve as an early version of the API documentation.

Working with asynchronous functions

- Both fields we have so far in this example are mapped to a normal synchronous resolver.
- However, if a field needs to do a lot of work to resolve its data, it should use an asynchronous resolver because, otherwise, the entire API service will be blocked and unable to serve other requests.
- To demonstrate this problem, let's fake a delay in processing the currentTime field.

Working with asynchronous functions

```
currentTime: {
  type: GraphQLString,
  resolve: () => {
    const sleepToDate = new Date(new Date().getTime() + 5000);
    while (sleepToDate > new Date()) {
      // sleep
    }
    const isoString = new Date().toISOString();
    return isoString.slice(11, 19);
  },
},
```

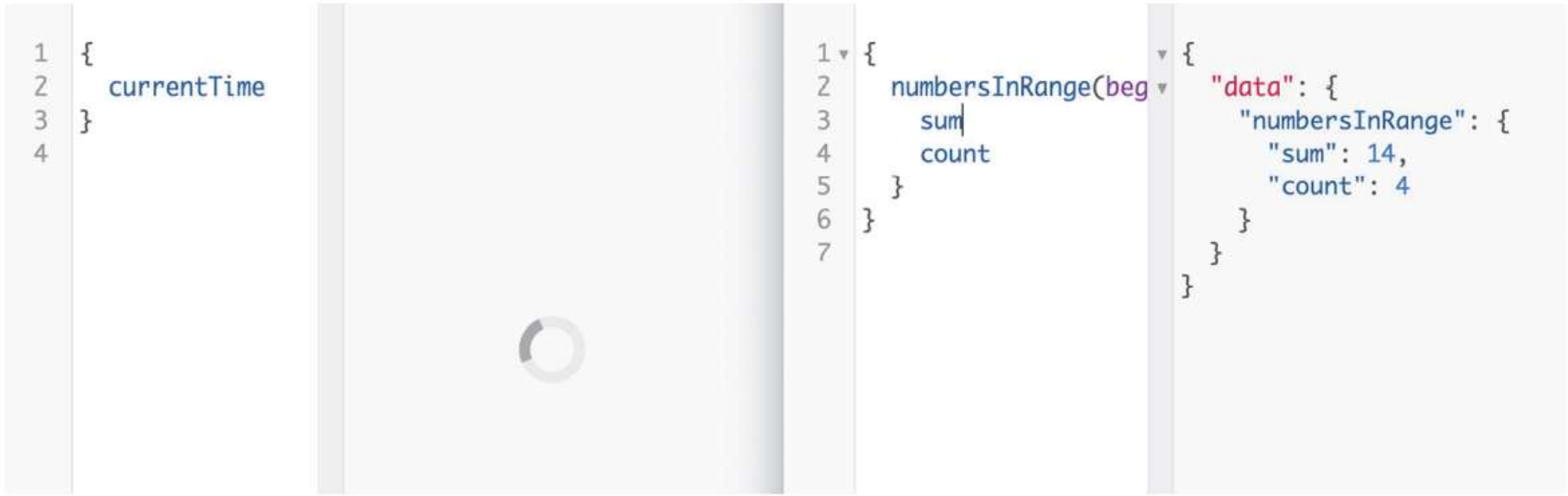
Working with asynchronous functions

Working with asynchronous functions

```
currentTime: {
  type: GraphQLString,
  resolve: () => {
    return new Promise(resolve => {
      setTimeout(() => {
        const isoString = new Date().toISOString();
        resolve(isoString.slice(11, 19));
      }, 5000);
    });
  },
};
```

Resolver functions support returning a promise object.

Working with asynchronous functions



Summary

- A GraphQL service is centered around the concept of a schema that is made executable with resolver functions.
- A GraphQL implementation like GraphQL.js takes care of the generic tasks involved in working with an executable schema.
- You can interact with a GraphQL service using any communication interface.
- HTTP(S) is the popular choice for GraphQL services designed for web and mobile applications.

"Complete Lab"