Implementing schema resolvers

LEARNING VOYAGE

Table of Contents

- **Running the development environment**
- Setting up the GraphQL runtime
- Communicating over HTTP
- Building a schema using constructor objects
- Generating SDL text from object-based schemas
- Working with asynchronous functions

Running the development environment

- We will use this repository in lessons 5-10.
- It has the skeleton for both the API server (which we're going to build in lessons 5-8) and the skeleton for the web server (which we'll build in lessons 9 and 10).
- 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

```
{
  "name": "az.dev",
  "version": "0.0.1",
  "private": true,
  "scripts": {
  "scripts": {
    "start-dbs": "docker-compose -f dev-dbs/docker.yml up",
    "api-server": "(cd api && nodemon -r esm src/server.js)",
    "web-server": "(cd web/src && rimraf .cache dist && parcel
index.html)",
    "start-blank-dbs": "docker-compose -f dev-dbs/docker-blank.yml
up"
  },
  },
  ·-·-·
 }
```

Node.js packages

- To implement the GraphQL API server, we need two new packages.

```
$ npm install graphql express
```

Table of Contents

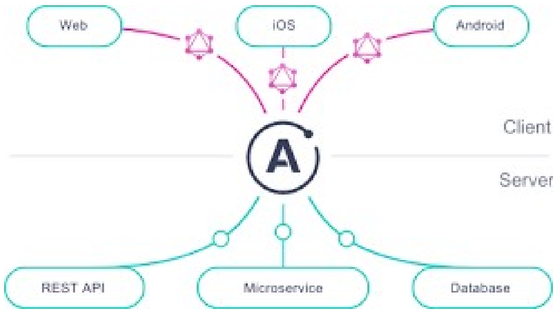
- Running the development environment
- **Setting up the GraphQL runtime**
- Communicating over HTTP
- Building a schema using constructor objects
- Generating SDL text from object-based schemas
- Working with asynchronous functions

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.

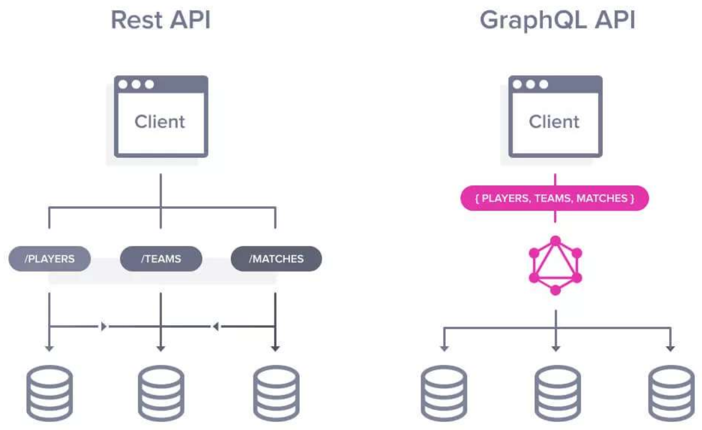
```
{
  currentTime
}
```



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

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

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

Creating the schema object

- Schema text for the simple example schema we’re building.

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

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);
  },
};
```

Executing requests

- We can test that in api/src/server.js. Add the following import line.

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

- 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.

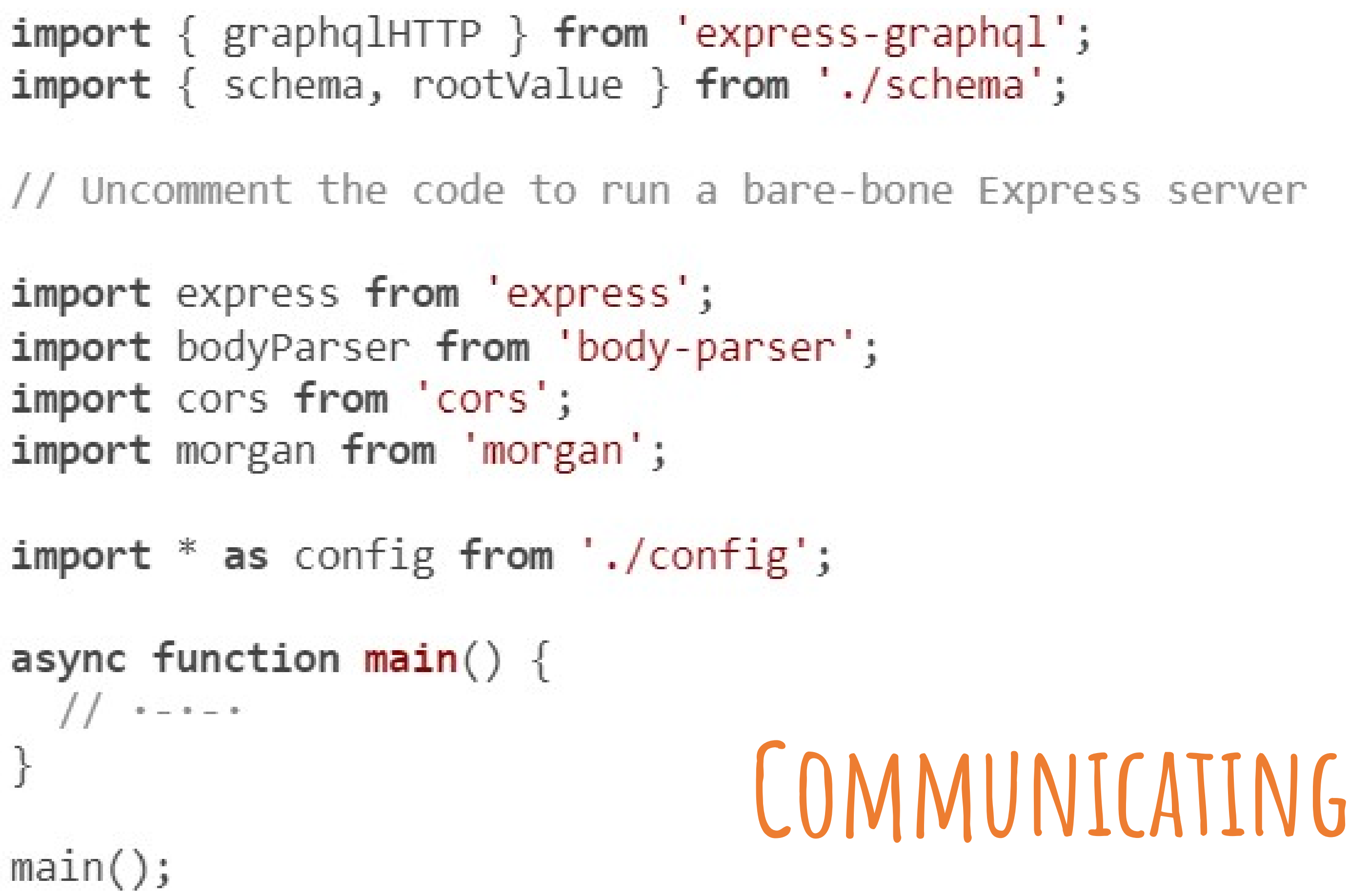
```
$ node -r esm api/src/server.js "{ currentTime }"
```

Executing requests

```
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]);
// •-•-•
```

Table of Contents

- Running the development environment
- Setting up the GraphQL runtime
- **Communicating over HTTP**
- Building a schema using constructor objects
- Generating SDL text from object-based schemas
- Working with asynchronous functions



```
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();
```

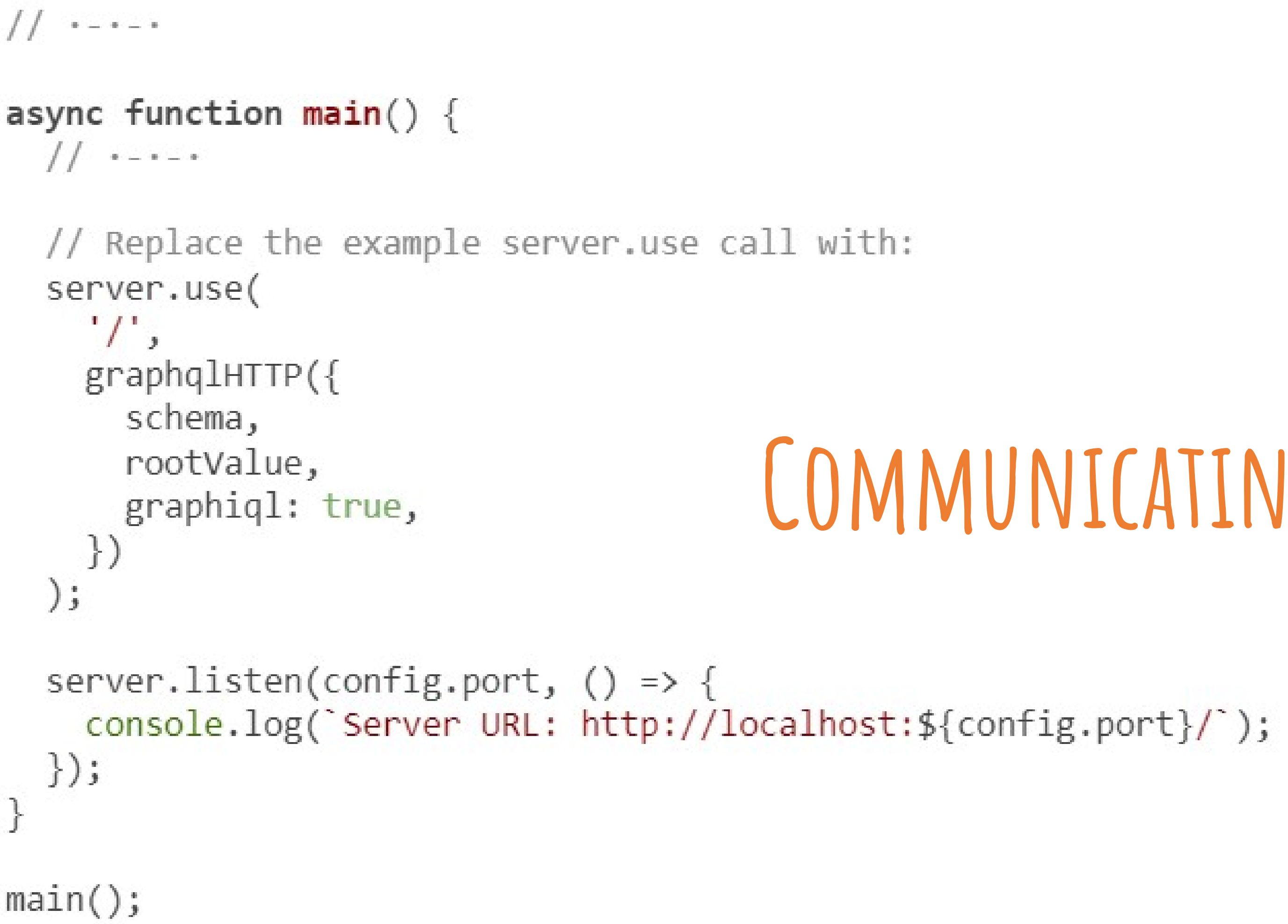
Communicating over HTTP

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
});
```

Communicating over HTTP



```
// ·-·-·

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.

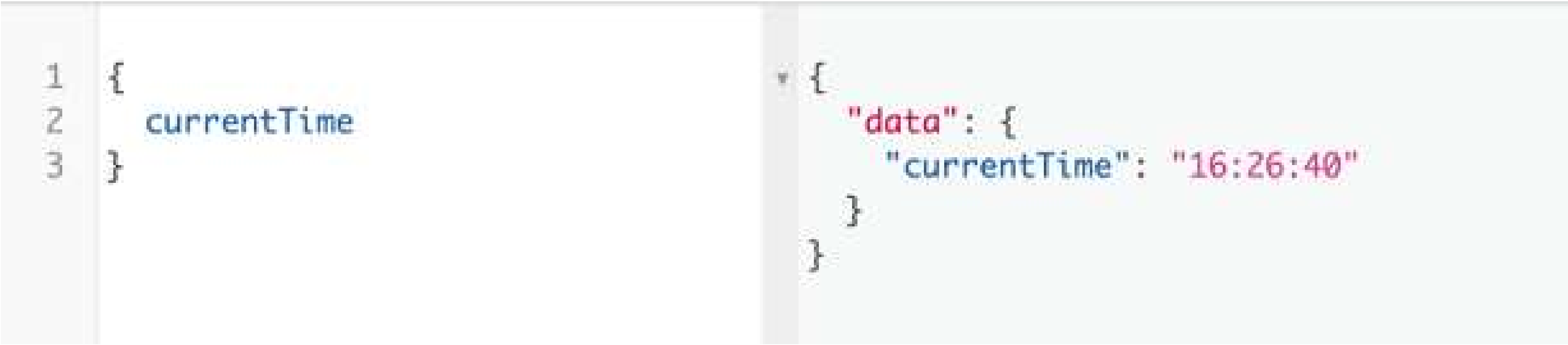
```
$ npm run api-server
```

- You should see this message:

```
Server URL: http://localhost:4321/
```

Communicating over HTTP

- You should be able to test the currentTime field query in it, as shown in figure



```
{
  currentTime
}
```

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

Table of Contents

- Running the development environment
- Setting up the GraphQL runtime
- Communicating over HTTP
- **Building a schema using constructor objects**
- Generating SDL text from object-based schemas
- Working with asynchronous functions

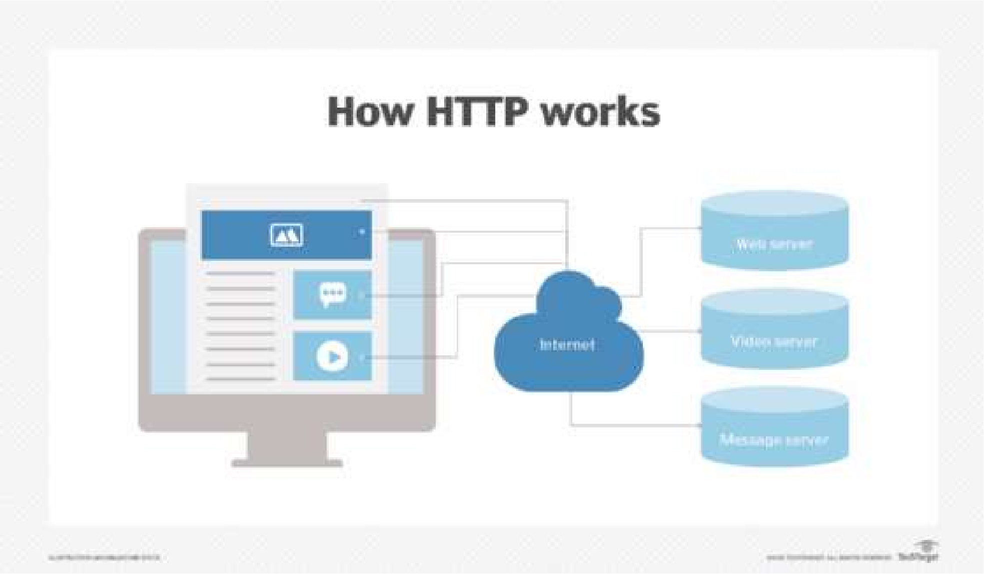
Building a schema using constructor objects

- GraphQL.js has another format that can be used to create a GraphQL schema and its various types.
- Instead of text written with the schema language, you can use JavaScript objects instantiated from calls to various constructor classes.

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';
```

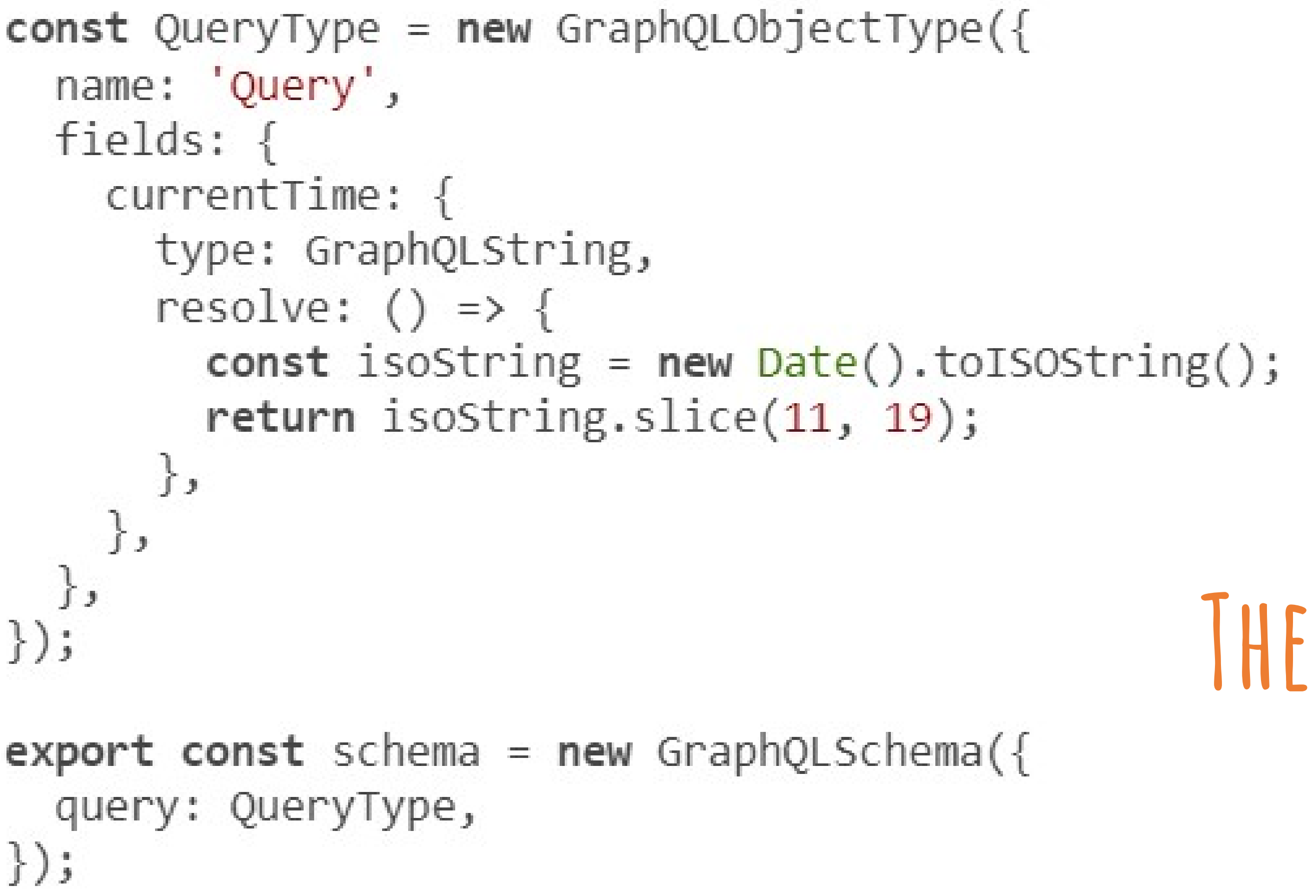


The Query type

- Type-based objects are designed to work together to help us create a schema.
- For example, to instantiate a schema object, you just do something like this.

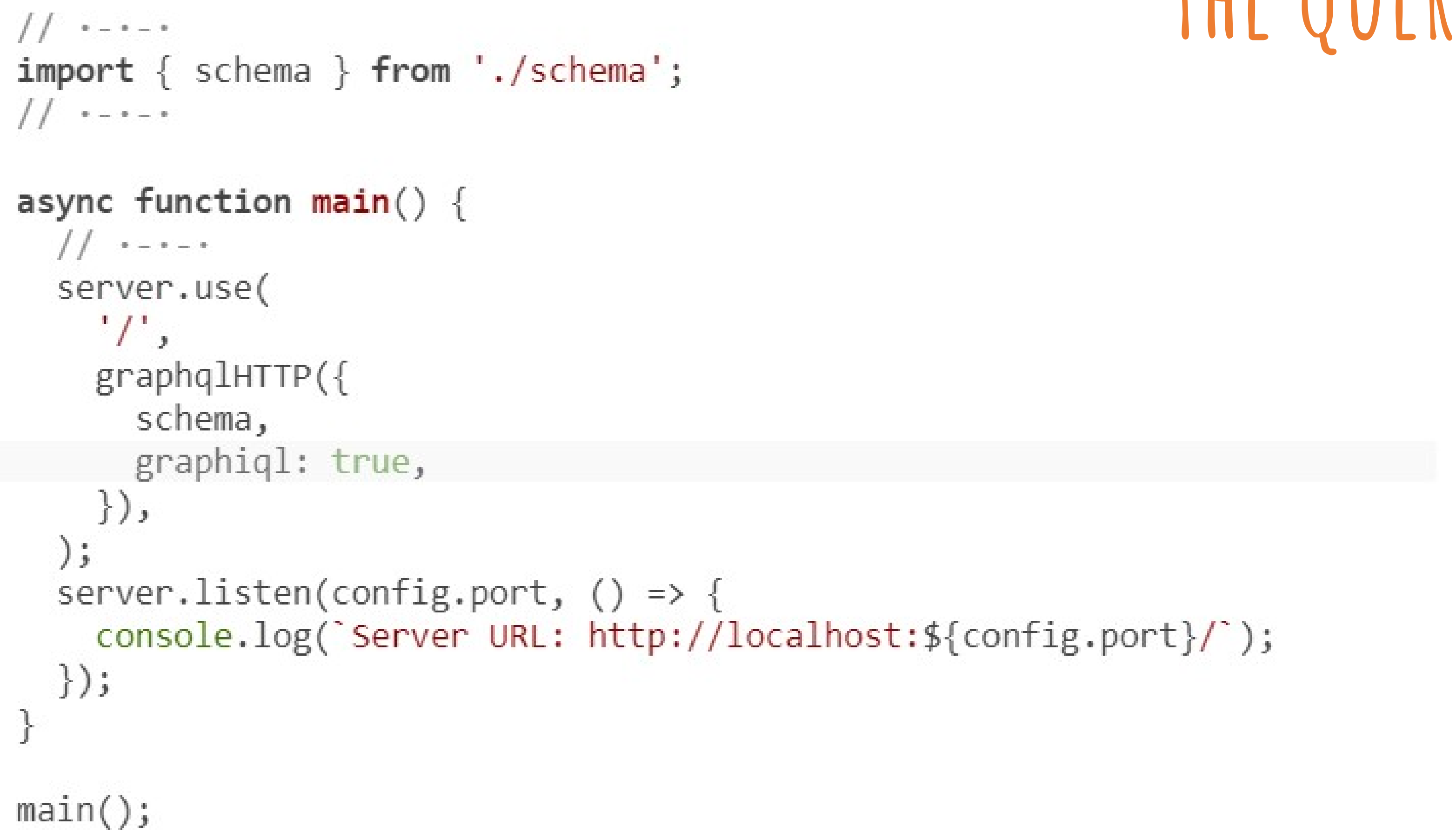
```
const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'Query',
    fields: {
      // Root query fields are defined here
    }
  }),
});
```

The Query type



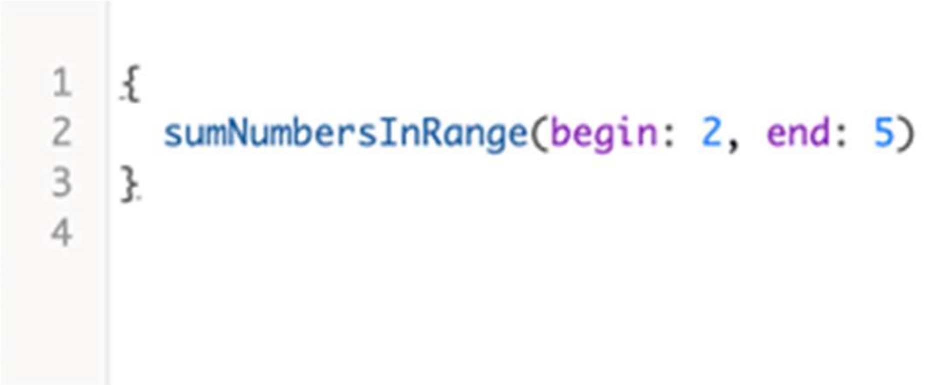
```
const QueryType = new GraphQLObjectType({
  name: 'Query',
  fields: {
    currentTime: {
      type: GraphQLString,
      resolve: () => {
        const isoString = new Date().toISOString();
        return isoString.slice(11, 19);
      },
    },
  },
});

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

The Query type

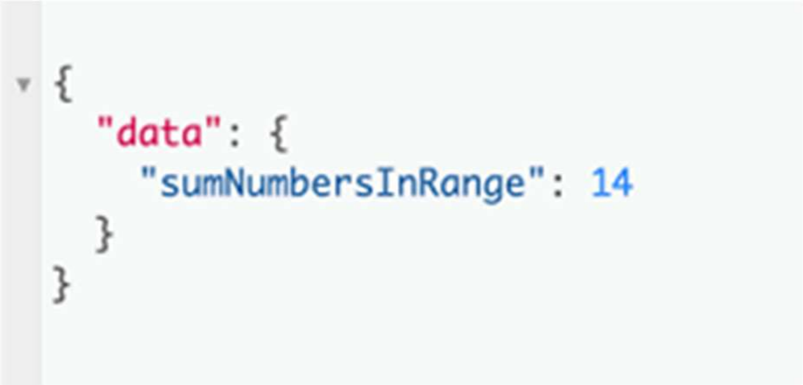


```
// ·-·-·
import { schema } from './schema';
// ·-·-·

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

main();
```

API support a sumNumbersInRange field that accepts two arguments

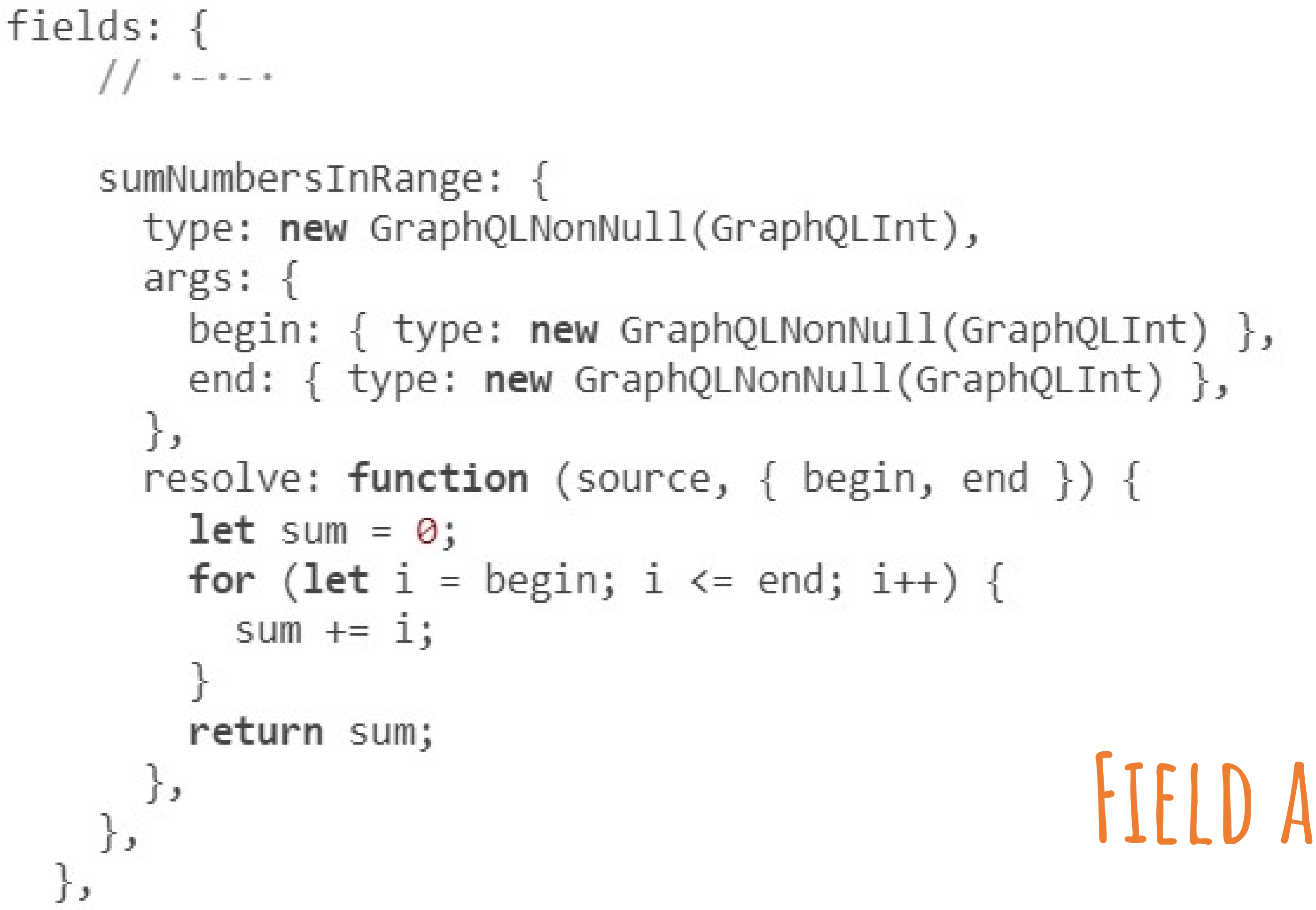


```
{
   sumNumbersInRange(begin: 2, end: 5)
}
```



```
{
  "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.

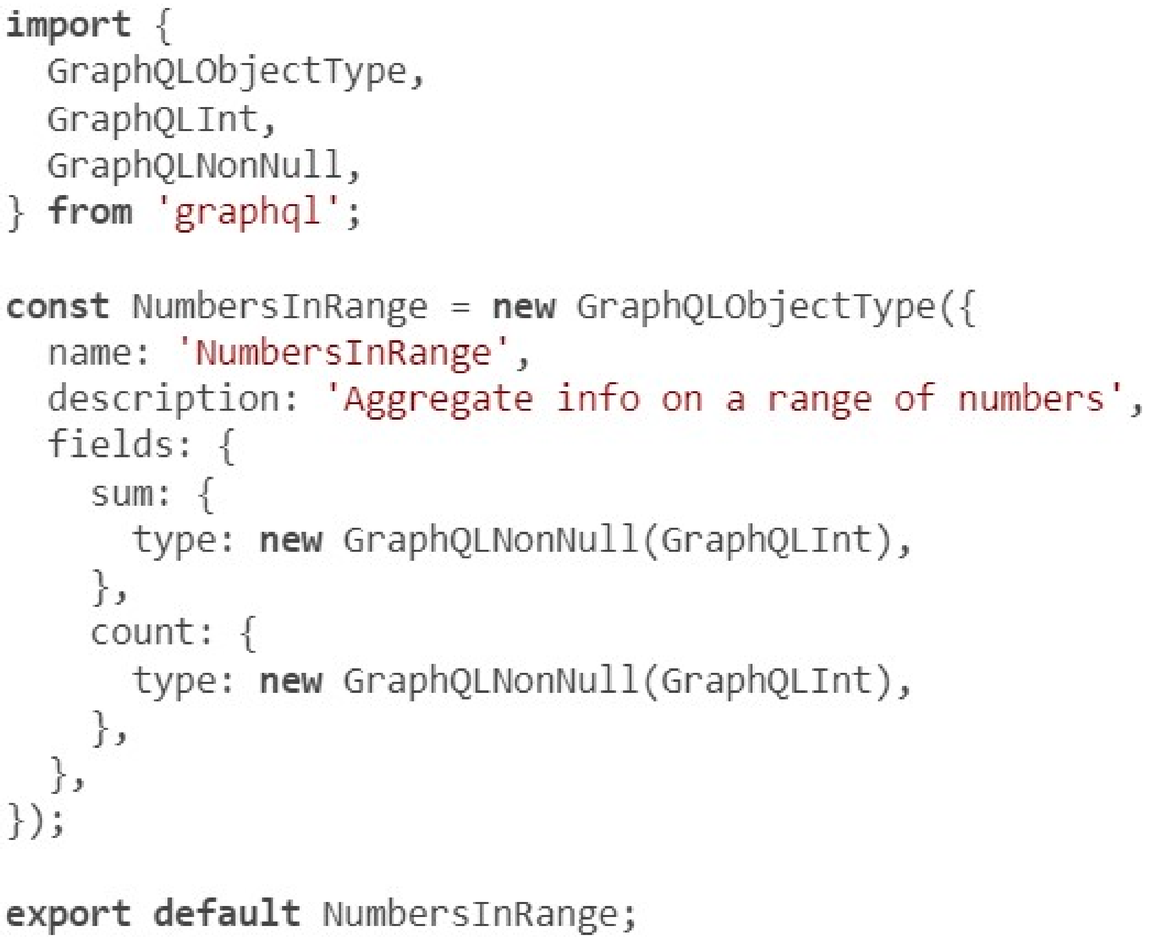
```
{
  sumNumbersInRange(begin: 2, end: 5)
}
```

Custom object types

- How the new numbersInRange field will be queried.

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

Custom object types

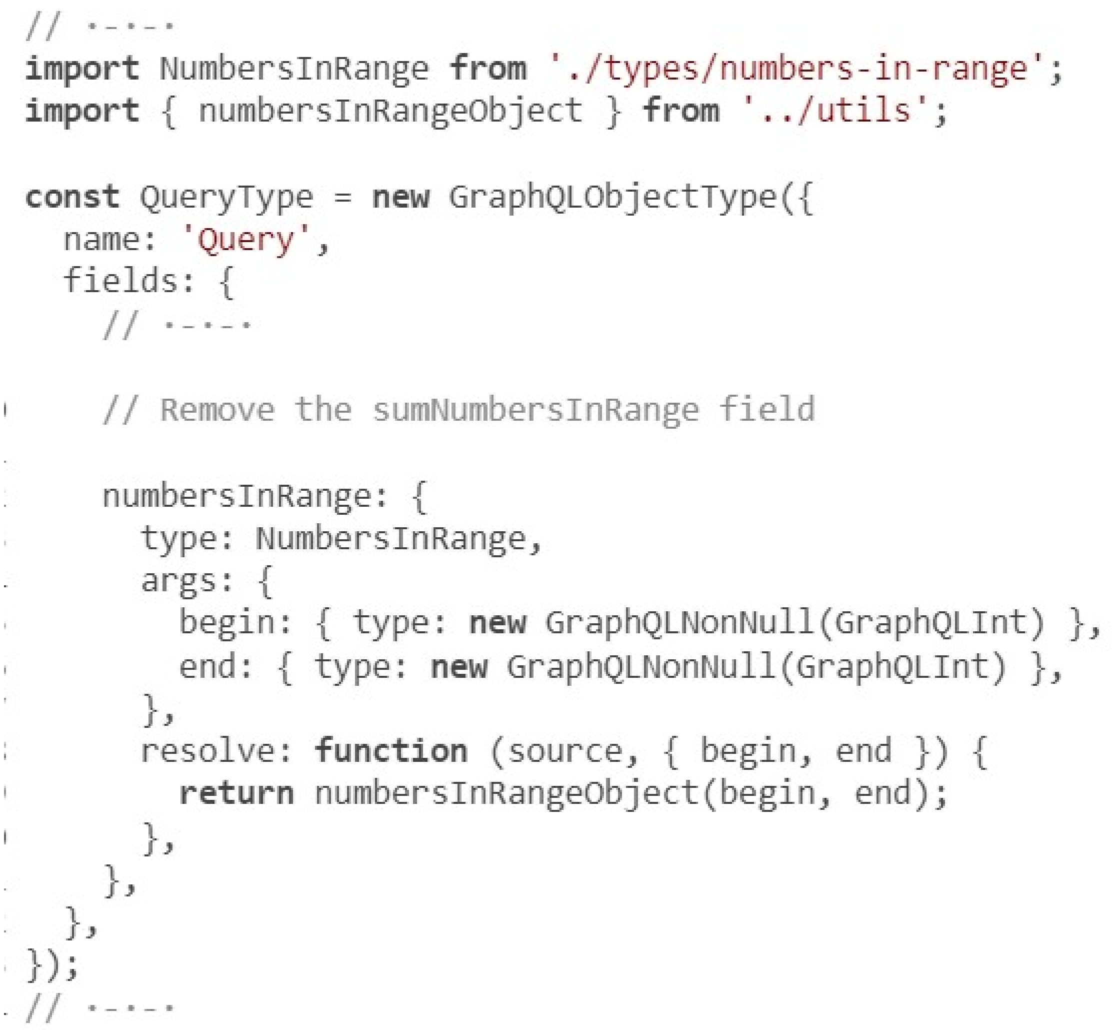


```
import {
  GraphQLObjectType,
  GraphQLInt,
  GraphQLNonNull,
} from 'graphql';

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 };
};
```

Custom object types



```
// .....
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);
      },
    },
  },
});
// .....
```

Custom object types

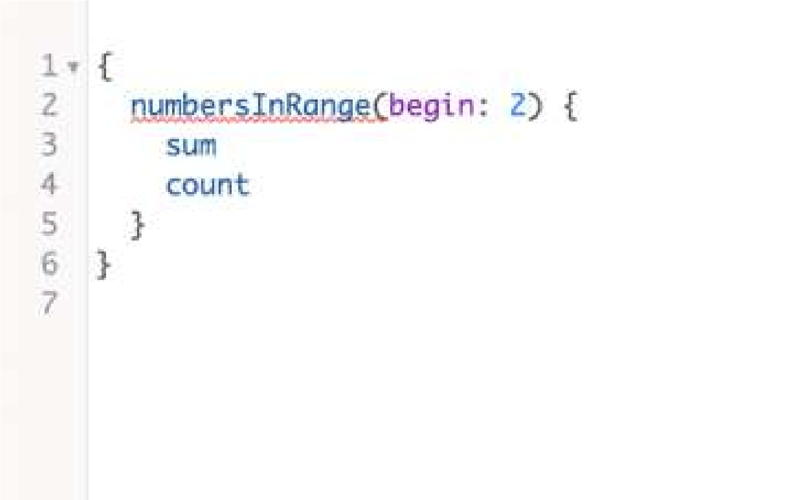
- If you test the API now, you should be able to execute a query like the following:

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

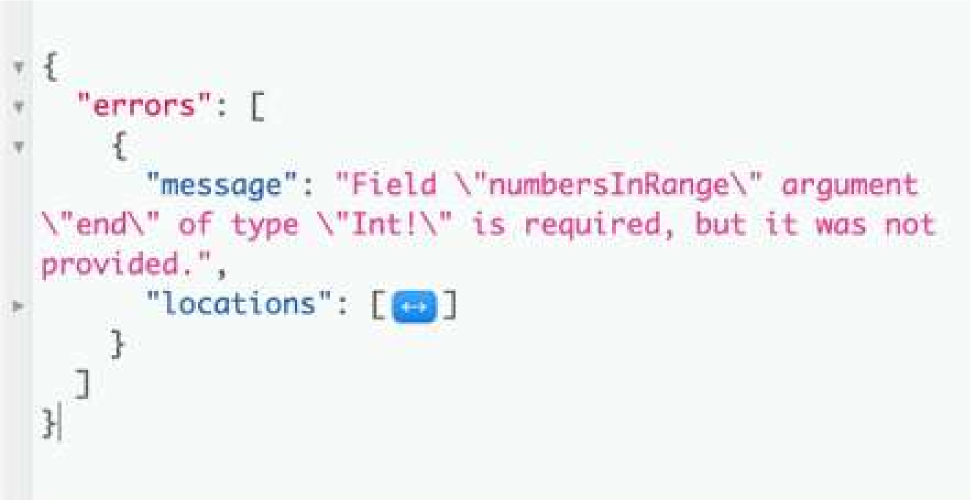
- And you will get this response:

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

Custom errors

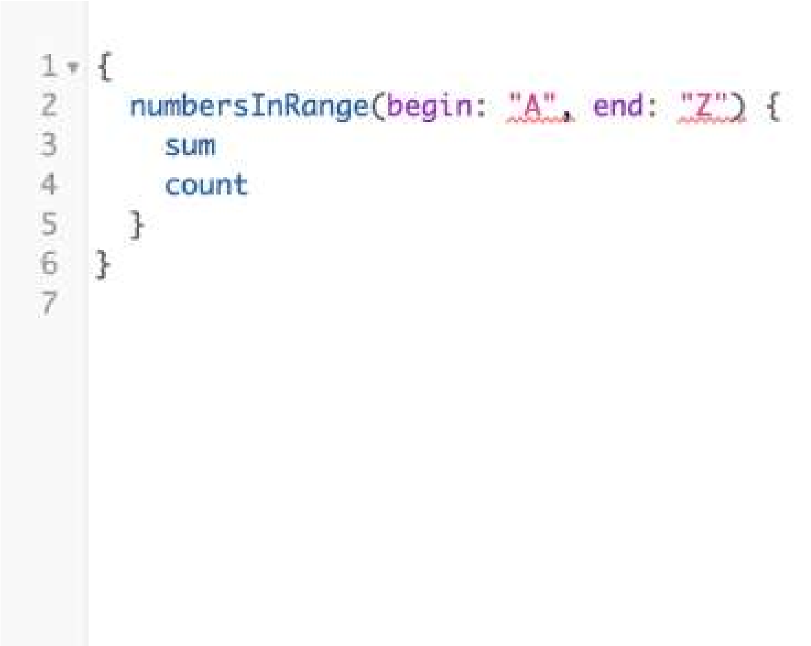


```
{
  numbersInRange(begin: 2) {
    sum
    count
  }
}

```

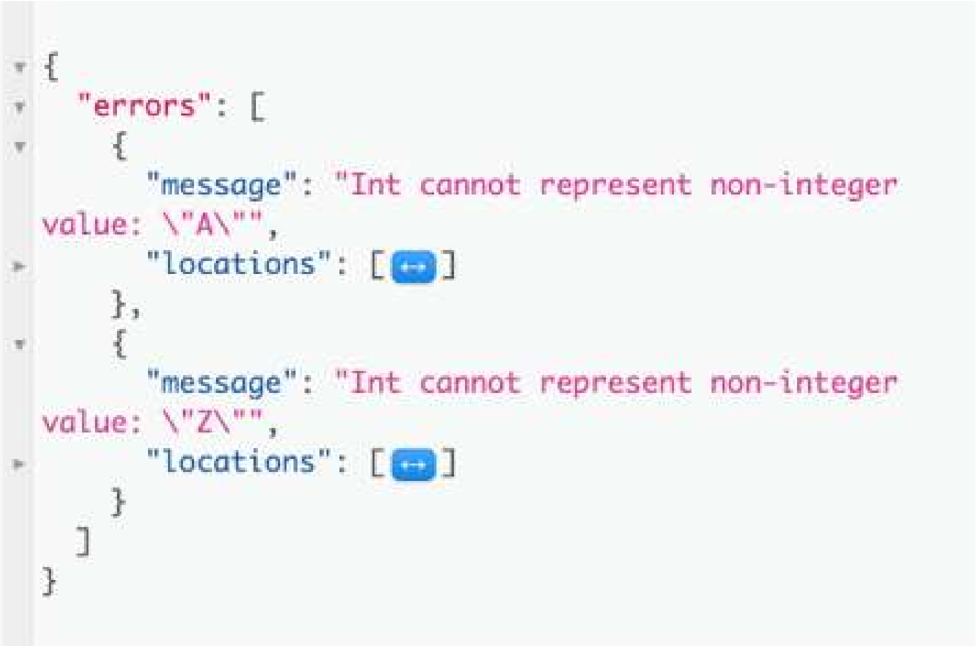


```
{
  "errors": [
    {
      "message": "Field \"numbersInRange\" argument \"end\" of type \"Int!\" is required, but it was not provided.",
      "locations": [↔]
    }
  ]
}
```

Custom errors

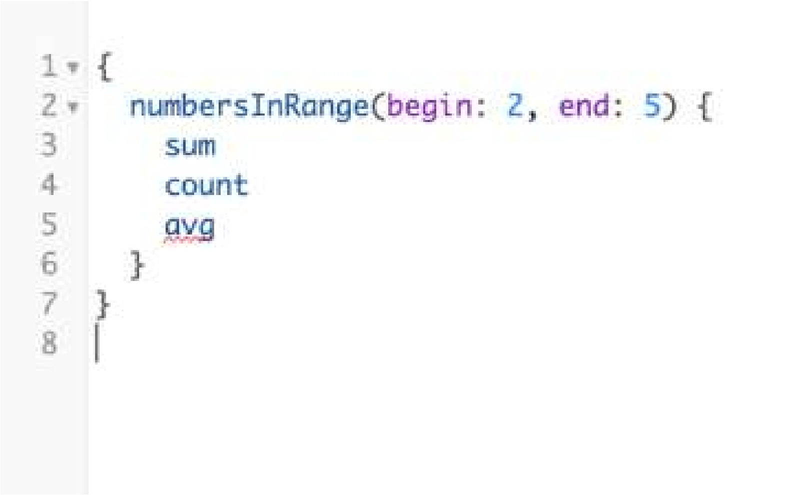


```
{
  numbersInRange(begin: "A", end: "Z") {
    sum
    count
  }
}
```

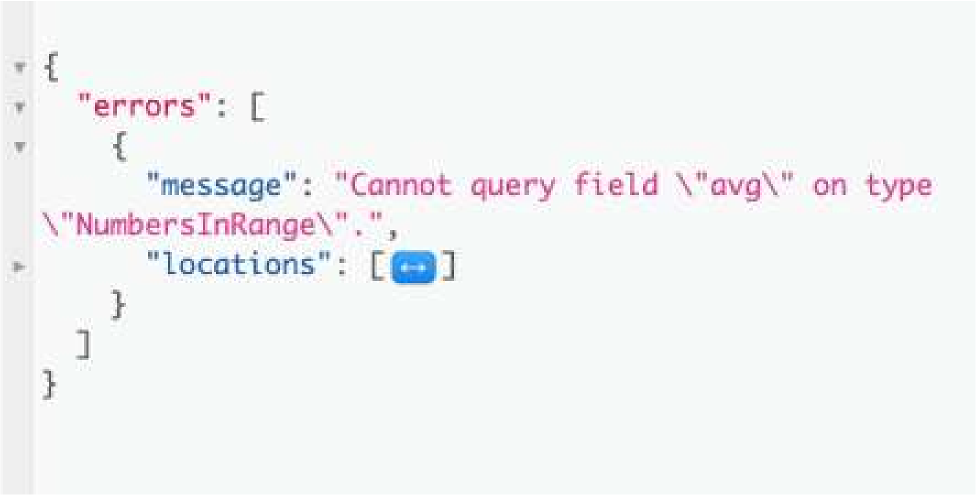


```
{
  "errors": [
    {
      "message": "Int cannot represent non-integer
value: \"A\"",
      "locations": [↔]
    },
    {
      "message": "Int cannot represent non-integer
value: \"Z\"",
      "locations": [↔]
    }
  ]
}
```

Custom errors



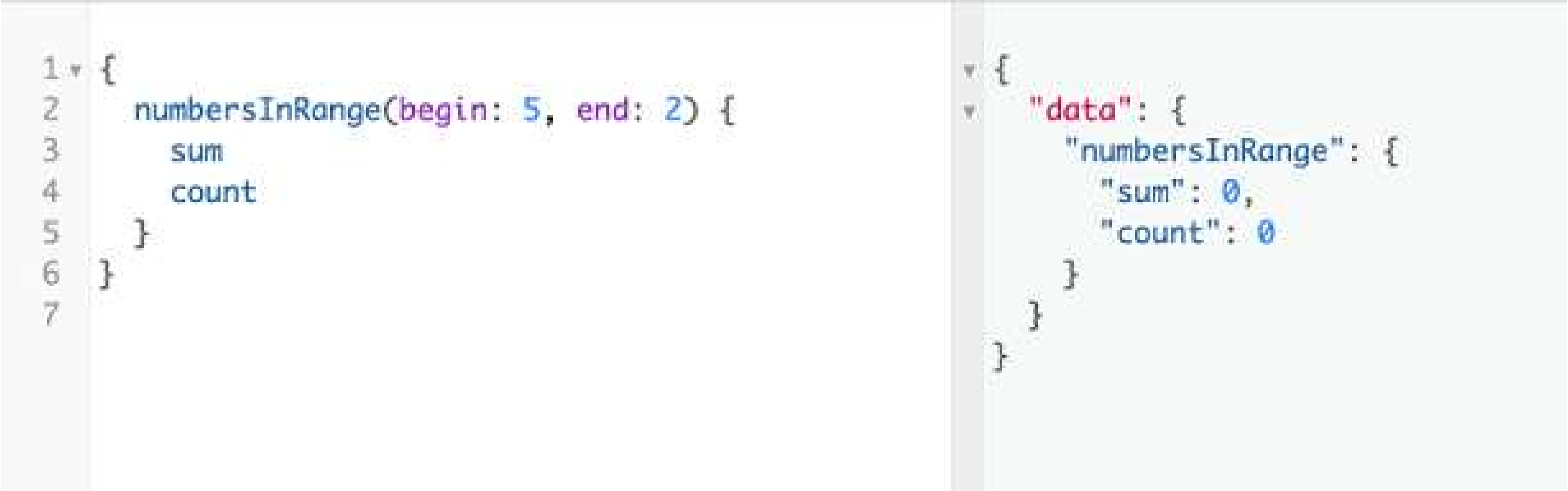
```
{
  numbersInRange(begin: 2, end: 5) {
    sum
    count
    avg
  }
}
```



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

Custom errors

- The API currently ignores this case and just returns zeros, as shown in figure



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

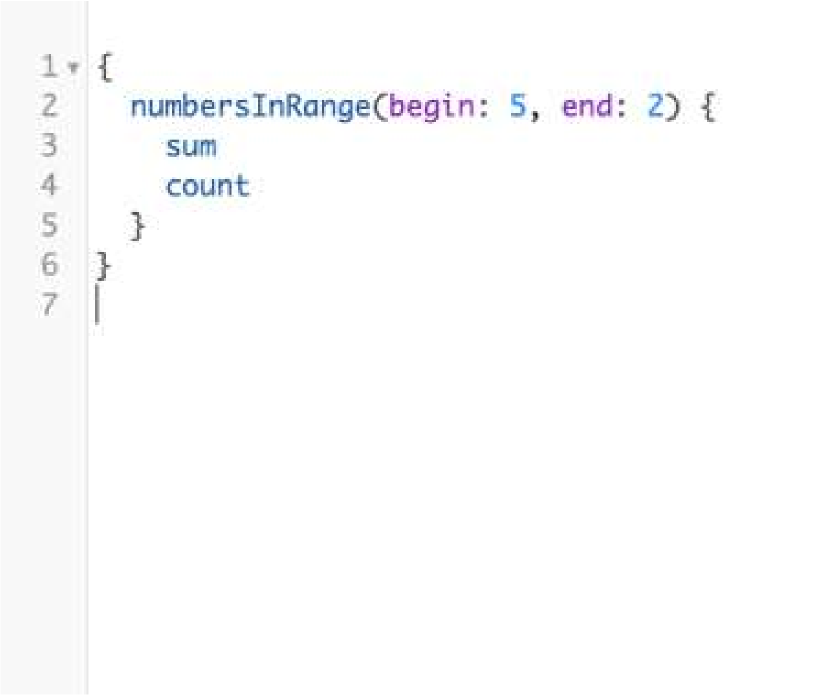
```
{
  "data": {
    "numbersInRange": {
      "sum": 0,
      "count": 0
    }
  }
}
```

Custom errors

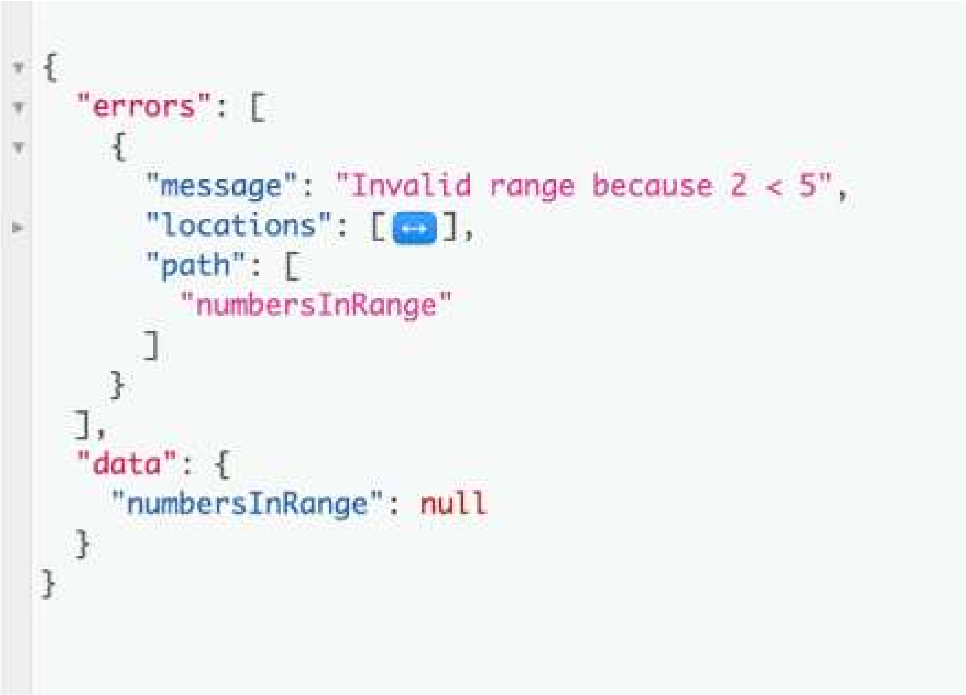
- We do the check in the resolver function for the numbersInRange field and throw an error with our custom message.

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

Custom errors

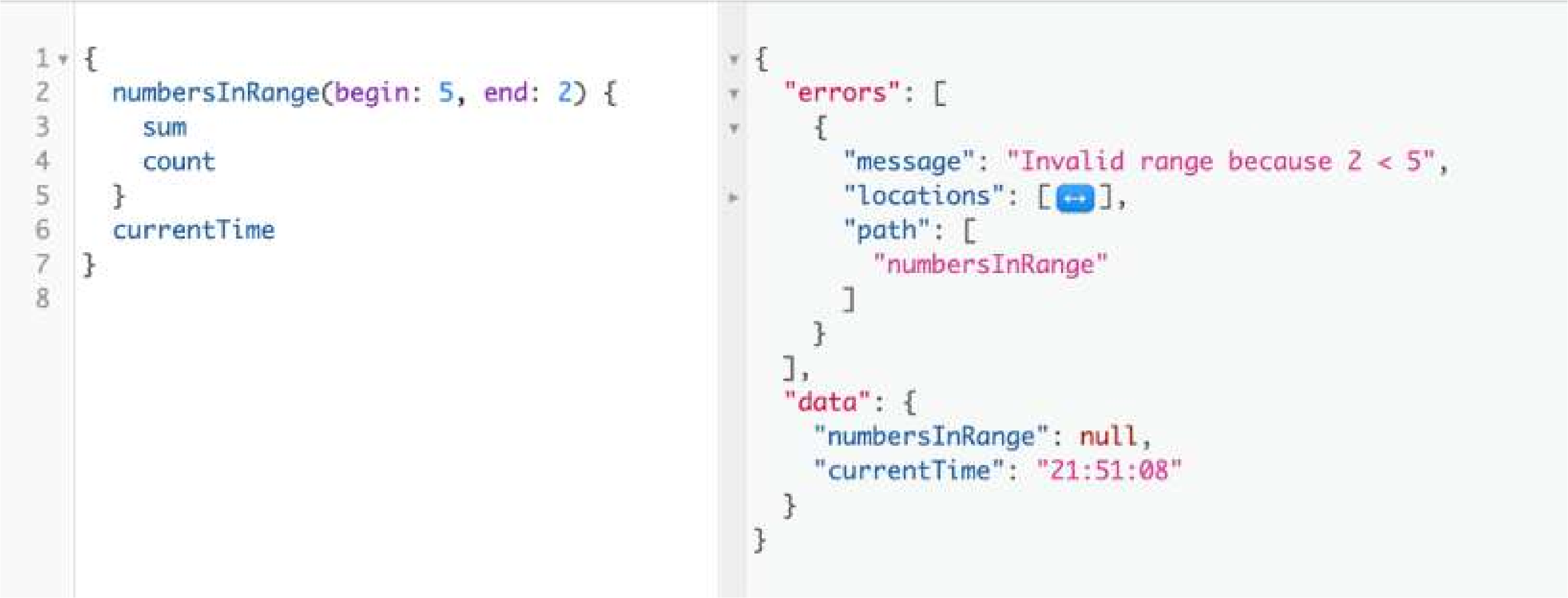


```
{
  numbersInRange(begin: 5, end: 2) {
    sum
    count
  }
}
```



```
{
  "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"
  }
}
```

Table of Contents

- Running the development environment
- Setting up the GraphQL runtime
- Communicating over HTTP
- Building a schema using constructor objects
- **Generating SDL text from object-based schemas**
- Working with asynchronous functions

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

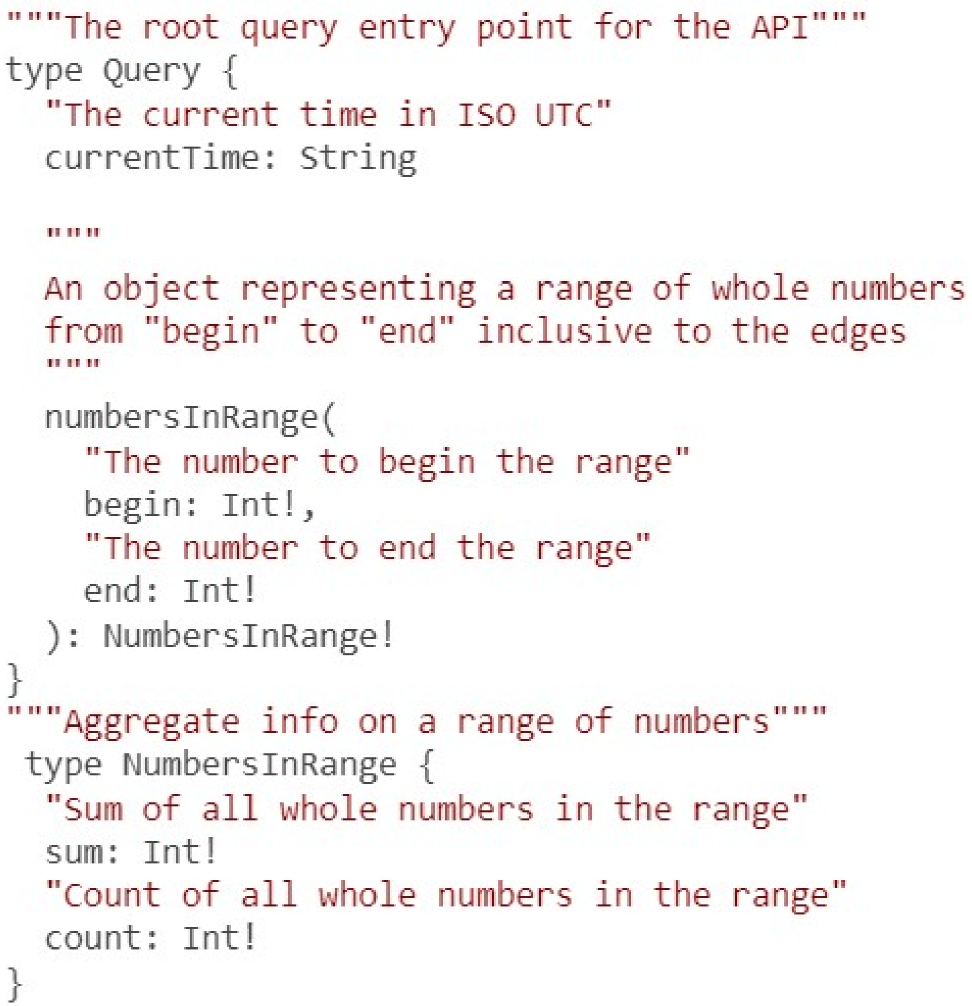
- My favorite part about this conversion is how the arguments to the numbersInRange field are defined in the schema language format:

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

- Compare that with:

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

Generating SDL text from object-based schemas



```
"""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!
}
```

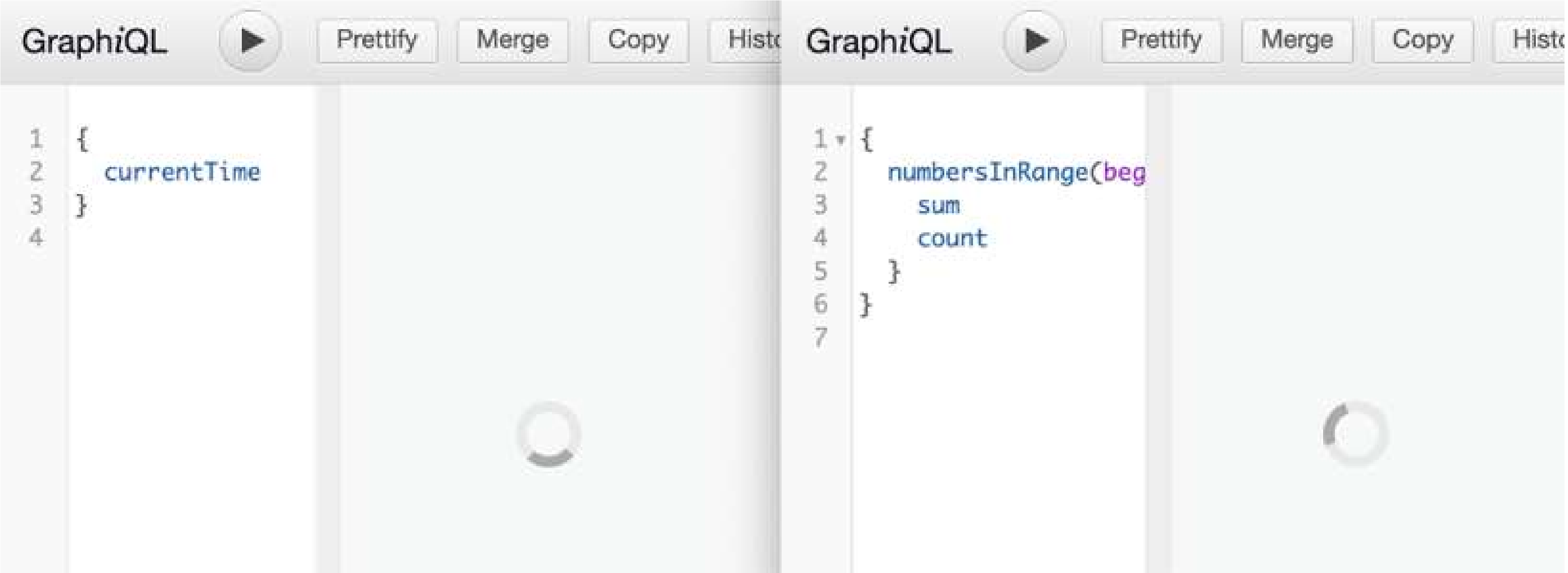
Table of Contents

- Running the development environment
- Setting up the GraphQL runtime
- Communicating over HTTP
- Building a schema using constructor objects
- Generating SDL text from object-based schemas
- **Working with asynchronous functions**

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



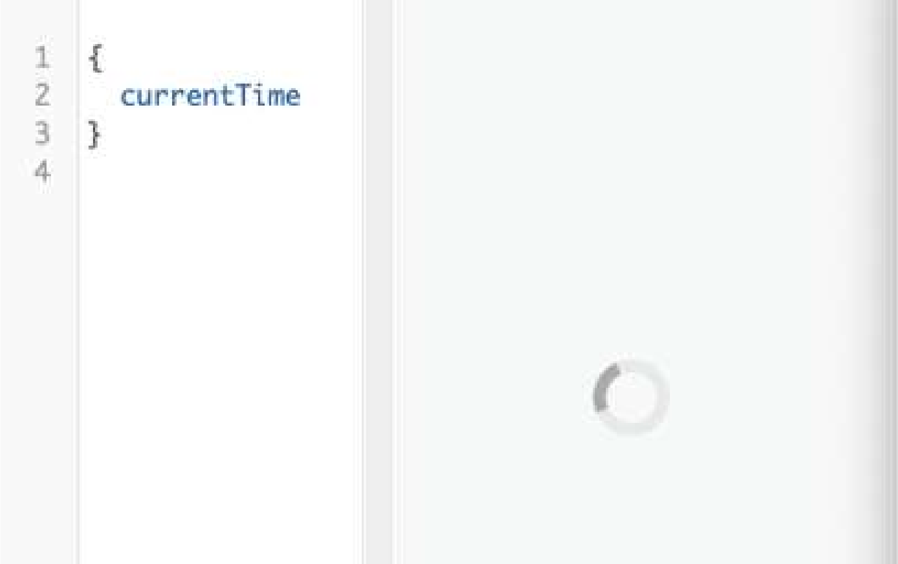
```
{
  currentTime
}
```

```
{
  numbersInRange(beg
    sum
    count
  }
}
```

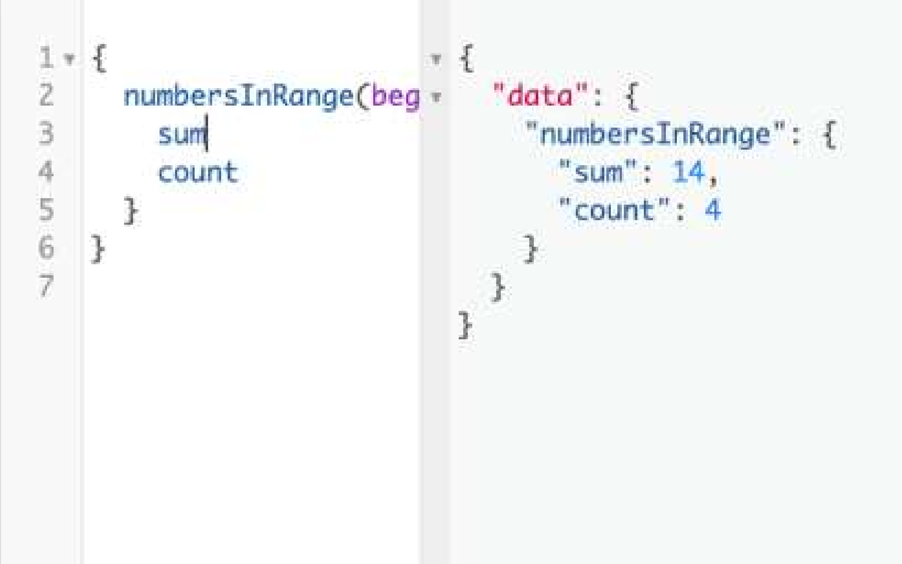
Working with asynchronous functions

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

Working with asynchronous functions



```
{
  currentTime
}
```



```
{
  numbersInRange(beg
    sum
    count
  }
}
```

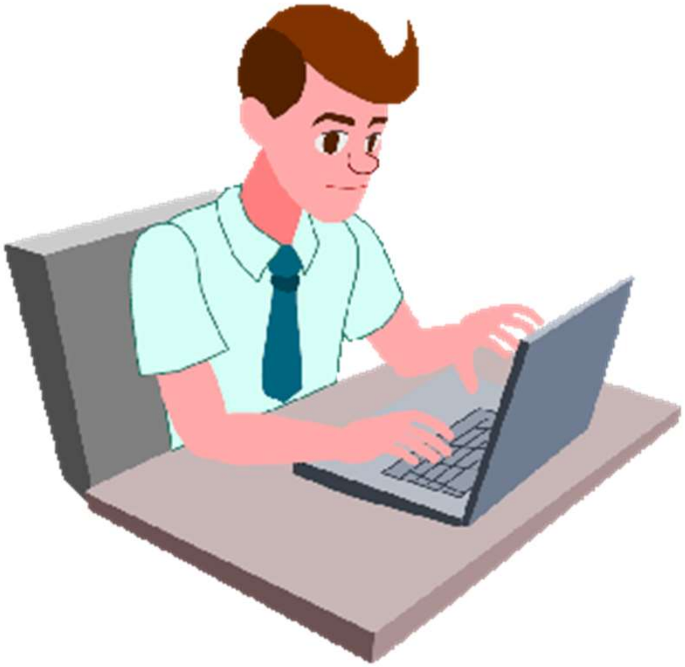
```
{
  "data": {
    "numbersInRange": {
      "sum": 14,
      "count": 4
    }
  }
}
```

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.



"Complete Lab"