Lesson 7

GRAPHQL

PROBLEM WITH REST

Fetching posts

GET /posts

```
[
  {
    "title":"Cool post",
    "subtitle": ...
    "date":"07/09/2020"
    "authorid":"122"
  },
    {
    "title":"Cooler post",
    "subtitle": ...
    "date":"06/05/2020"
     "authorid":"198"
  },
]
```

What if we also want the author's name for every post?

Option 1: under-fetching

- Fetch the authors from another resource

GET /posts

```
[
  {
    "title":"Cool post",
    "subtitle": ...
    "date":"07/09/2020"
    "authorid":"122"
  },
    {
    "title":"Cooler post",
    "subtitle": ...
    "date":"06/05/2020"
     "authorid":"198"
  },
]
```

GET /author/122

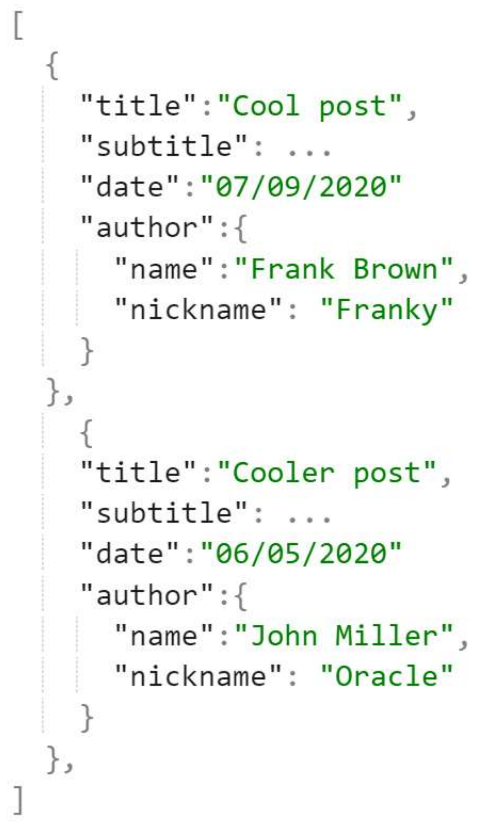
```
{
  "id":"122",
  "name":"Frank Brown",
  "nickname": "Franky"
  "birthdate":"07/02/1996"
}
```

We need to call different endpoints in order to get all data we want to show on the UI

Option 2: over-fetching

- Modify the resource to also return the author data

GET /posts



```
[
  {
   "title":"Cool post",
   "subtitle": ...
   "date":"07/09/2020"
   "author":{
     "name":"Frank Brown",
     "nickname": "Franky"
   }
  },
   {
   "title":"Cooler post",
   "subtitle": ...
   "date":"06/05/2020"
   "author":{
     "name":"John Miller",
     "nickname": "Oracle"
   }
  },
]
```

We may not always need the author's name and author's nickname

We may fetch too much data

Option 3: new endpoint

- Create a new endpoint according the needs of the client

GET /posts

```
[
  {
    "title":"Cool post",
    "subtitle": ...
    "date":"07/09/2020"
    "authorid":"122"
  },
   {
    "title":"Cooler post",
    "subtitle": ...
    "date":"06/05/2020"
     "authorid":"198"
  },
]
```

GET /postsWithAuthor

```
[
  {
    "title":"Cool post",
    "subtitle": ...
    "date":"07/09/2020"
    "author":{
      "name":"Frank Brown",
      "nickname": "Franky"
    }
  },
   {
    "title":"Cooler post",
    "subtitle": ...
    "date":"06/05/2020"
    "author":{
      "name":"John Miller",
      "nickname": "Oracle"
    }
  },
]
```

We end up with many (related) endpoints.

We need to write many endpoints. Slows down development.

Hard to maintain when changes happen

Documenting your API

- You started with a relative simple API for a front-end client
  - But it evolved over time to a complex API
- Over time:
  - Other clients also need to use your API
  - New developers join the project
- Problem
  - What resources are available
  - What parameters are accepted
  - Which ones are required, which ones are not?

We did not document our API

GRAPHQL

GraphQL

- Query language for APIs
- Client can specify exactly the data that is needed from the API
- Developed by Facebook

request

```
query {
  user {
    name
    age
  }
}
```

Specify the data you need

response

```
{
  "user": {
    "name": "Johnathan Joestar",
    "age": 27
  }
}
```

GraphQL

- GraphQL is an alternative to REST, not a replacement
- GraphQL is a standard that is implemented in almost all languages.

Why graphQL? No under-fetching

REST

GET /posts

GET /author/122

Multiple requests

GraphQL

```
query {
  posts {
    title
    subtitle
    date
    author {
      name
    }
  }
}
```

One requests

Why graphQL? No over-fetching

REST

```
[
  {
    "title":"Cool post",
    "subtitle": ...
    "date":"07/09/2020"
    "author":{
      "name":"Frank Brown",
      "nickname": "Franky"
    }
  },
    {
    "title":"Cooler post",
    "subtitle": ...
    "date":"06/05/2020"
    "author":{
      "name":"John Miller",
      "nickname": "Oracle"
    }
  },
]
```

We sometimes fetch too much data

GraphQL

```
query {
  posts {
    title
    subtitle
    date
  }
}
```

We only fetch what we need

Why graphQL? No new endpoints

REST

GET /posts

GET /postsWithAuthor

GET /postsWithAuthorAndNumberOfViews

New endpoint for every client's need

We need to write many endpoints. Slows down development.

Hard to maintain when changes happen

GraphQL

```
query {
  posts {
    title
    subtitle
    date
  }
}
```

```
query {
  posts {
    title
    subtitle
    date
    author {
      name
    }
  }
}
```

Query specifies what is needed

Faster development

Easy to maintain

Why graphQL? API documentation

- GraphQL uses a schema

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

type Post {
  title: String!
  subtitle: String!
  body: String!
  date: String!
  author: User!
}

type Query {
  users: [User!]!
  user(name: String!): User!
  posts: [Post!]!
  post(title: String!): Post!
}

type Mutation {
  createUser(name: String!, age: Int):User!
  createPost(title: String!, subtitle: String!, body: String!): Post!
}
```

What resources are available?

What parameters are accepted?

Which parameters are required, which ones are not?

GRAPHQL EXAMPLE

Dependencies

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
  <groupId>com.graphql-java</groupId>
  <artifactId>graphql-spring-boot-starter</artifactId>
  <version>5.0.2</version>
</dependency>
<dependency>
  <groupId>com.graphql-java</groupId>
  <artifactId>graphql-java-tools</artifactId>
  <version>5.2.4</version>
</dependency>
<dependency>
  <groupId>com.graphql-java</groupId>
  <artifactId>graphiql-spring-boot-starter</artifactId>
  <version>5.0.2</version>
</dependency>
```

Web application running in Tomcat

GraphQL

Parses the graphql schema

GraphQL client

Graphql schema: customer.graphqls

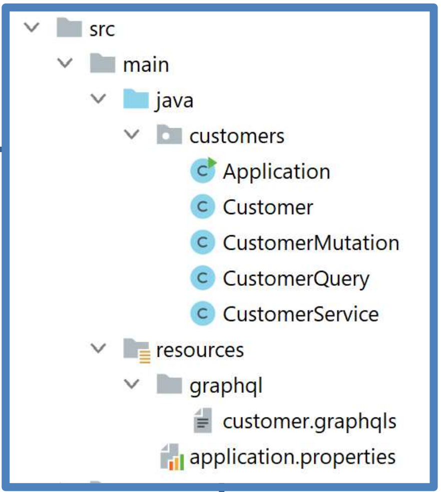
```
type Customer {
customerNumber : Int!
name: String!
email: String!
phone: String!
}

type Query {
customers(count: Int!):[Customer]
customer(customerNumber: Int!):Customer
}

type Mutation {
createCustomer(customerNumber: Int!, name: String!, email: String!, phone: String!):Customer
}
```

Fields of a customer and their types

! means required

Query methods:
Parameters and their types
Return type

Mutation methods:
Parameters and their types
Return type



```
src
  main
    java
      customers
        Application
        Customer
        CustomerMutation
        CustomerQuery
        CustomerService
    resources
      graphql
        customer.graphqls
      application.properties
```

Customer and CustomerService

```
public class Customer {
  private int customerNumber;
  private String name;
  private String email;
  private String phone;
```

```
@Service
public class CustomerService {
   Map<Integer, Customer> customers = new HashMap<>();

   public List<Customer> getAllCustomers(int count) {
      List<Customer> customerList  = customers.values().stream().collect(Collectors.toList());
      return customerList.subList(0,count);
   }

   public Optional<Customer> getCustomer(int customerNumber) {
      return  Optional.of(customers.get(customerNumber));
   }

   public Customer createCustomer(int customerNumber, String name, String email, String phone) {
      Customer customer = new Customer(customerNumber, name, email, phone);
      customers.put(customerNumber, customer);
      return customer;
   }
}
```

Query and Mutation class

```java
@Component
public class CustomerQuery implements GraphQLQueryResolver {

   @Autowired
   private CustomerService customerService;

   public List<Customer> getCustomers(final int count) {
      return customerService.getAllCustomers(count);
   }

   public Optional<Customer> getCustomer(final int customerNumber) {
      return customerService.getCustomer(customerNumber);
   }
}
```

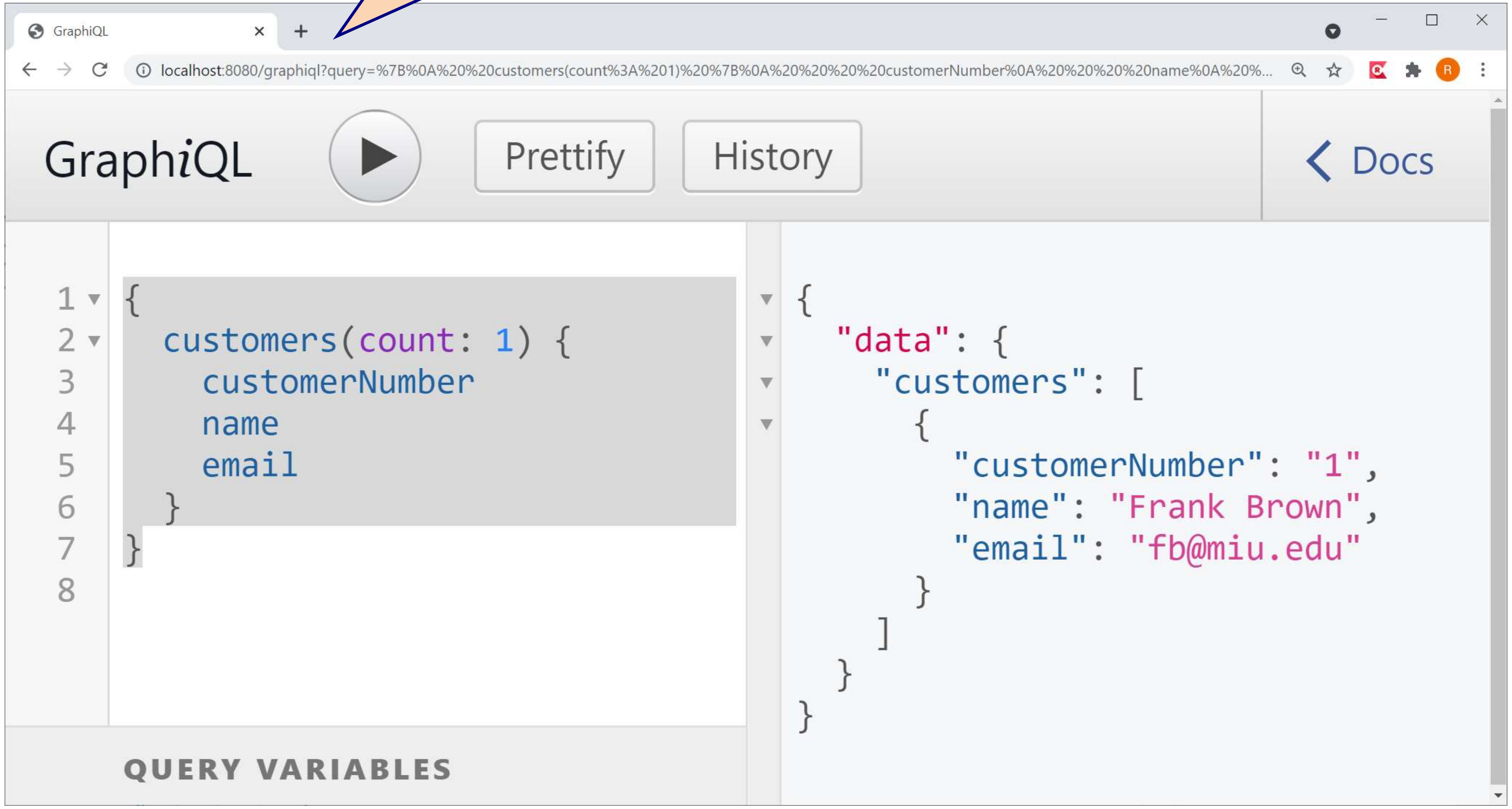
```graphql
type Query {
customers(count: Int):[Customer]
customer(customerNumber: ID):Customer
}
```

```graphql
type Mutation {
createCustomer(customerNumber: Int!, name: String!, email: String!, phone: String!):Customer
}
```

```java
@Component
public class CustomerMutation implements GraphQLMutationResolver {

   @Autowired
   private CustomerService customerService;

   public Customer createCustomer(final int customerNumber, final String name, final String email, final String phone) {
      return customerService.createCustomer(customerNumber, name, email, phone);
   }
}
```

GraphiQL library

URL: localhost:8080/graphiql


GraphiQL
localhost:8080/graphiql?query=%7B%0A%20%20customers(count%3A%201)%20%7B%0A%20%20%20%20customerNumber%0A%20%20%20%20name%0A%20%...
GraphiQL
Prettify
History
Docs
QUERY VARIABLES

```
{
  customers(count: 1) {
    customerNumber
    name
    email
  }
}
```

```
{
  "data": {
    "customers": [
      {
        "customerNumber": "1",
        "name": "Frank Brown",
        "email": "fb@miu.edu"
      }
    ]
  }
}
```

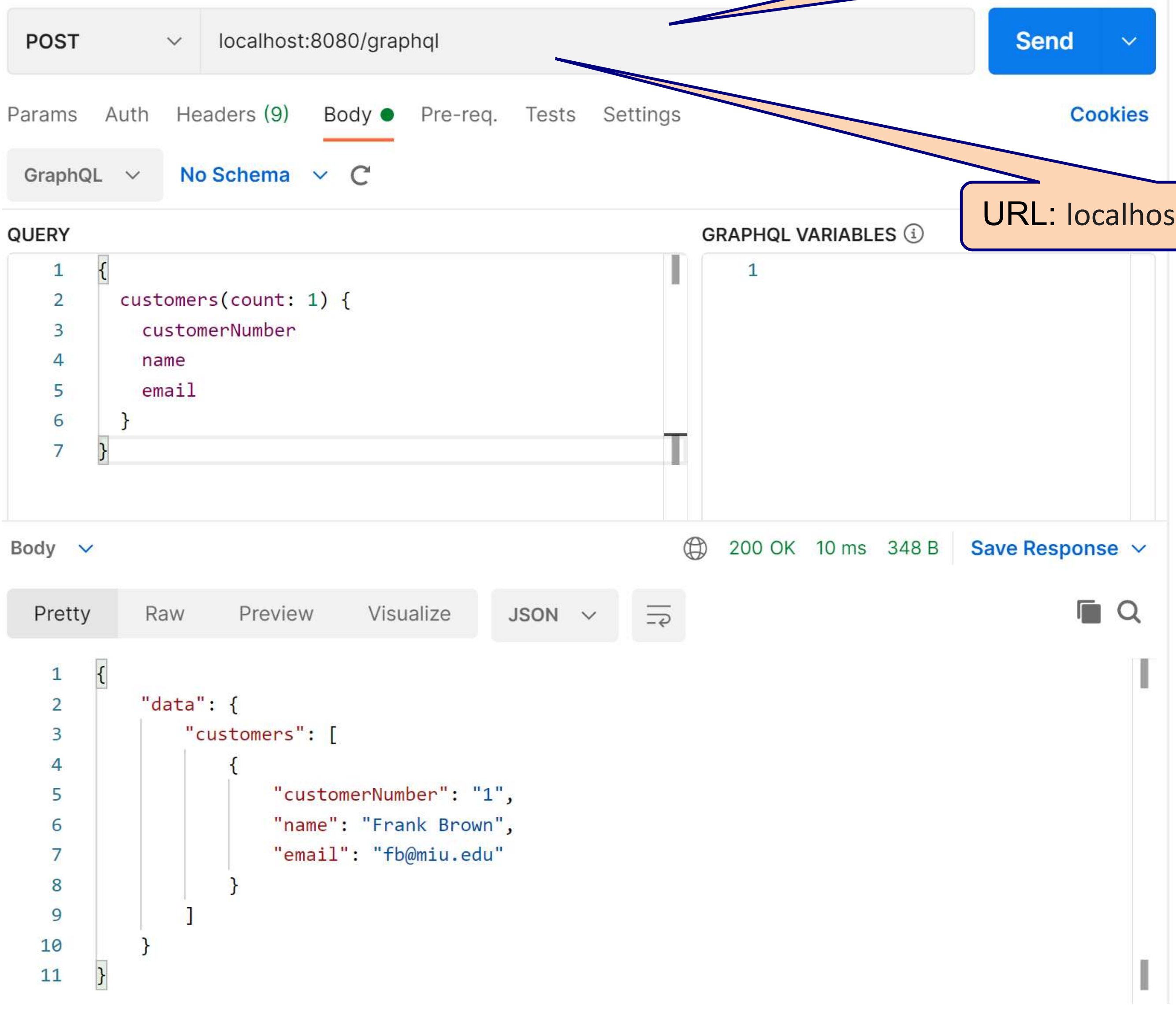
Postman

There is only one endpoint with graphql

POST

GraphQL

URL: localhost:8080/graphql



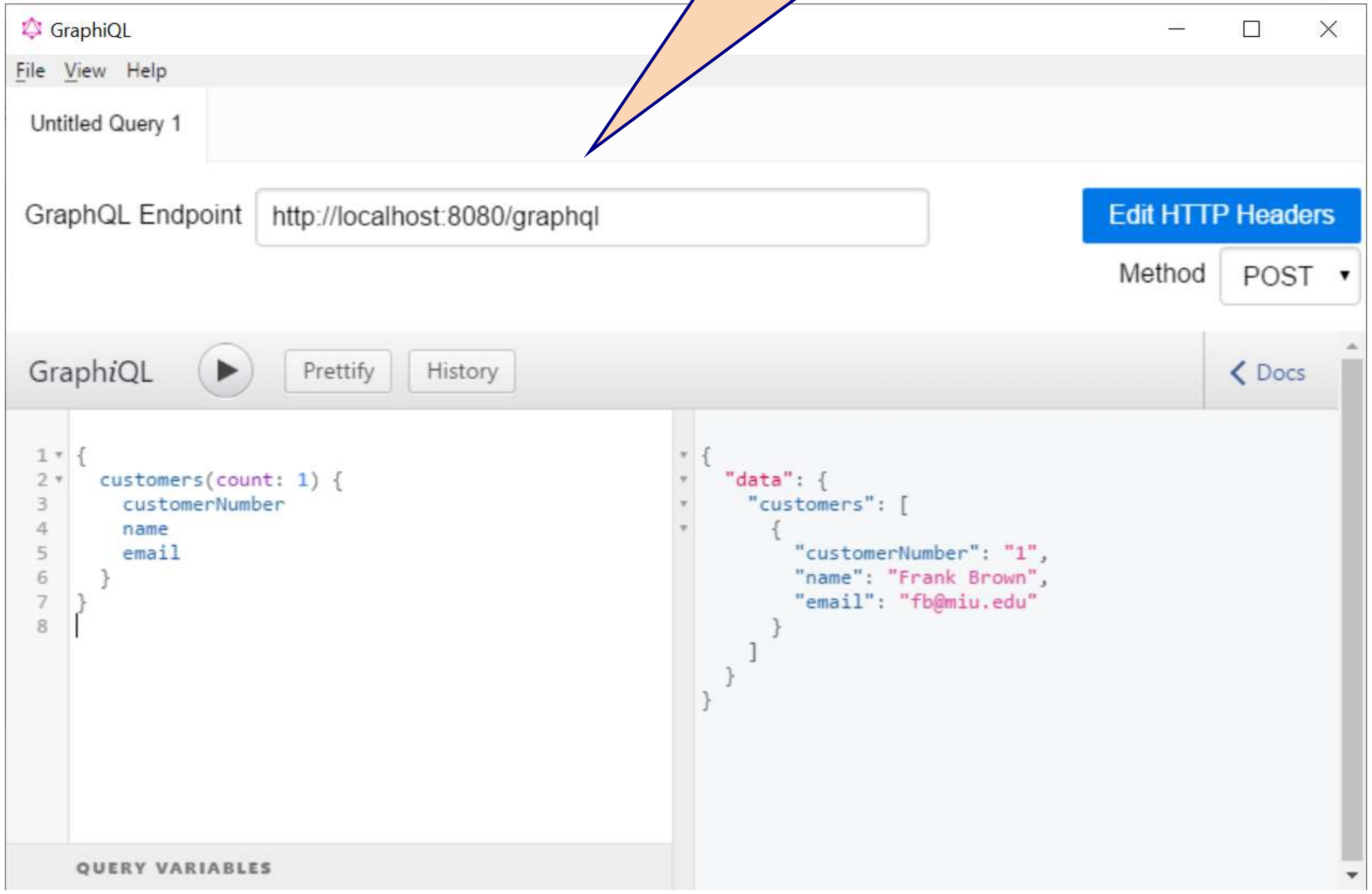
```
{
  customers(count: 1) {
    customerNumber
    name
    email
  }
}
```

```
{
    "data": {
        "customers": [
            {
                "customerNumber": "1",
                "name": "Frank Brown",
                "email": "fb@miu.edu"
            }
        ]
    }
}
```

GraphQL tool

URL: localhost:8080/graphql

Standalone tool like postman for GraphQL



GraphQL Endpoint http://localhost:8080/graphql

Method POST

```
{
  customers(count: 1) {
    customerNumber
    name
    email
  }
}
```

```
{
  "data": {
    "customers": [
      {
        "customerNumber": "1",
        "name": "Frank Brown",
        "email": "fb@miu.edu"
      }
    ]
  }
}
```

GRAPHQL SCHEMA

Graphql schema: customer.graphqls

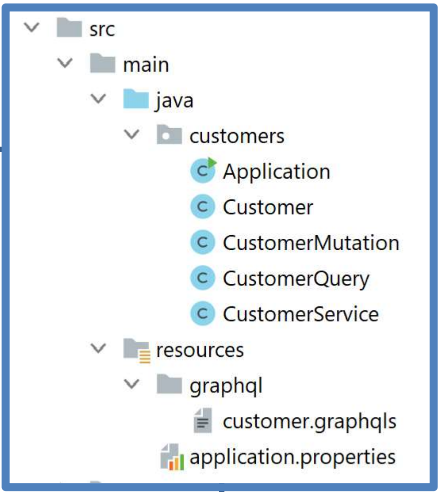
```
type Customer {
customerNumber : Int!
name: String!
email: String!
phone: String!
}

type Query {
customers(count: Int!):[Customer]
customer(customerNumber: Int!):Customer
}

type Mutation {
createCustomer(customerNumber: Int!, name: String!, email: String!, phone: String!):Customer
}
```

Fields of a customer and their types

! means required

Query methods:
Parameters and their types
Return type

Mutation methods:
Parameters and their types
Return type



GraphQL schema

GRAPHQL TYPE	SERIALIZED AS
Int	Signed 32-bit integer
Float	Signed double-precision floating-point value
String	UTF-8 character sequence
Boolean	true or false
ID	String

GRAPHQL MARKER	EQUIVALENT
`<type>!`	Not Null
`[<type>]`	List
`[<type>!]`	List of Not Null Elements
`[<type>]!`	Not Null list
`[<type>!]!`	Not Null list of Not Null Elements

```
type Hotel {
    id: ID!
    # Hotel name
    name: String!
    # Hotel address
    address: String!
    # Date of the hotel registry creation
    creationDate: String!
    # List of rooms for a particular hotel
    room: [Room]!
}
```

Graphql schema: customer.graphqls

```
type Customer {
customerNumber : Int!
name: String!
email: String!
phone: String!
address : Address!
}

type Address {
street: String!
city: String!
zip: String!
customer: Customer!
}

type Query {
customers(count: Int!):[Customer]
customer(customerNumber: Int!):Customer
customer(street: String!, city: String!, zip:String!):[Customer]
address(customerNumber: Int!):Address
}

type Mutation {
createCustomer(customerNumber: Int!, name: String!, email: String!, phone: String!, street: String!, city: String!,
zip:String!):Customer
}
```

Customer and Address

```
public class Customer {
  private int customerNumber;
  private String name;
  private String email;
  private String phone;
  private Address address;
```

```
public class Address {
  private String street;
  private String city;
  private String zip;
```

CustomerService

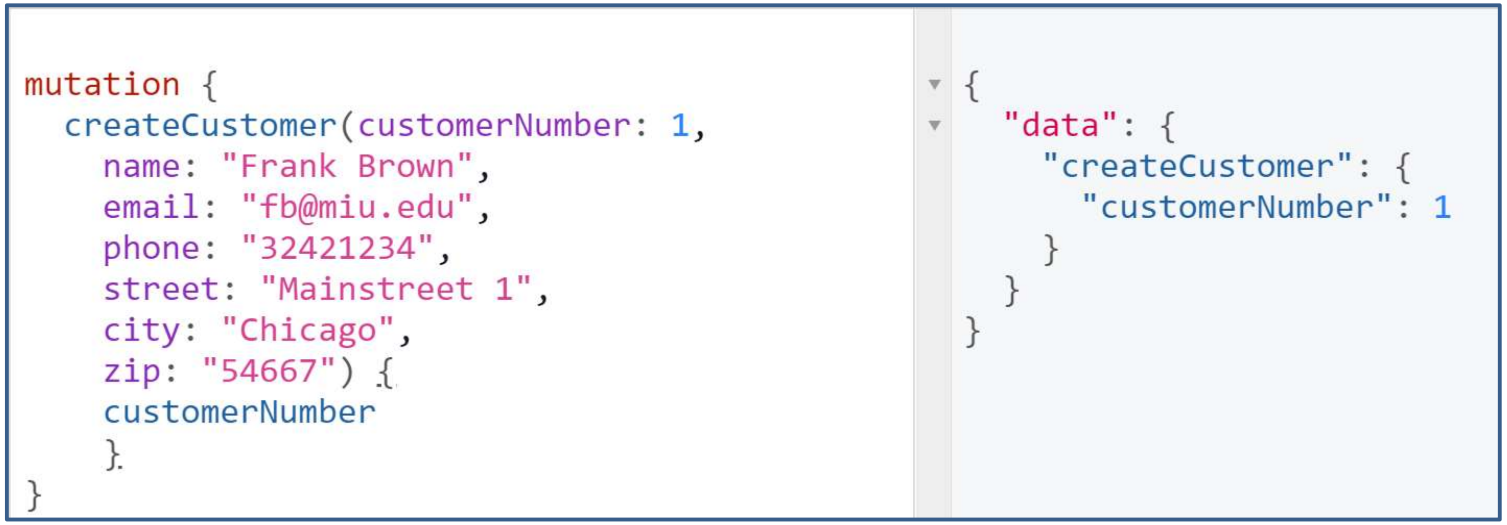
```
@Service
public class CustomerService {
   Map<Integer, Customer> customers = new HashMap<>();

   public List<Customer> getAllCustomers(int count) {
      List<Customer> customerList  = customers.values().stream().collect(Collectors.toList());
      return customerList.subList(0,count);
   }

   public Optional<Customer> getCustomer(int customerNumber) {
      return  Optional.of(customers.get(customerNumber));
   }

   public Customer createCustomer(int customerNumber, String name, String email, String phone, String street,
String city, String zip) {
      Customer customer = new Customer(customerNumber, name, email, phone);
      Address address = new Address(street, city, zip);
      customer.setAddress(address);
      customers.put(customerNumber, customer);
      return customer;
   }

   public List<Customer> getCustomersWithAddress(String street, String city, String zip) {
      List<Customer> customerList  = customers.values().stream()
            .filter(c-> c.getAddress().getStreet().equals(street))
            .filter(c-> c.getAddress().getCity().equals(city))
            .filter(c-> c.getAddress().getZip().equals(zip))
            .collect(Collectors.toList());
      return customerList;
   }
}
```

Mutation class

```
@Component
public class CustomerMutation implements GraphQLMutationResolver {

   @Autowired
   private CustomerService customerService;

   public Customer createCustomer(final int customerNumber, final String name, final String email, final String phone,
final  String street, final String city, final String zip) {
      return customerService.createCustomer(customerNumber, name, email, phone, street, city, zip);
   }
}
```

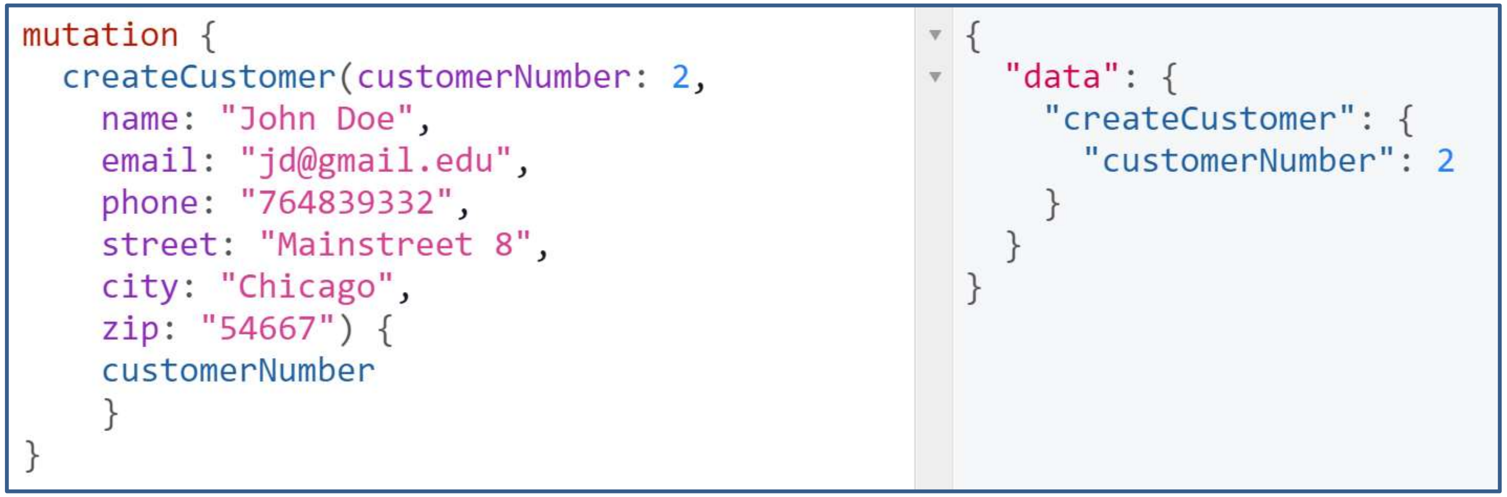
Query class

```
@Component
public class CustomerQuery implements GraphQLQueryResolver {

  @Autowired
  private CustomerService customerService;

  public List<Customer> getCustomers(final int count) {
    return customerService.getAllCustomers(count);
  }

  public Optional<Customer> getCustomer(final int customerNumber) {
    return customerService.getCustomer(customerNumber);
  }

  public List<Customer> getCustomer(final String street, final String city, final String zip) {
    return customerService.getCustomersWithAddress(street, city, zip);
  }

  public Optional<Address> getAddress(final int customerNumber) {
    Optional<Customer> customerOpt = customerService.getCustomer(customerNumber);
    if (customerOpt.isPresent())
      return Optional.of(customerOpt.get().getAddress());
    else
      return Optional.of(null);
  }
}
```



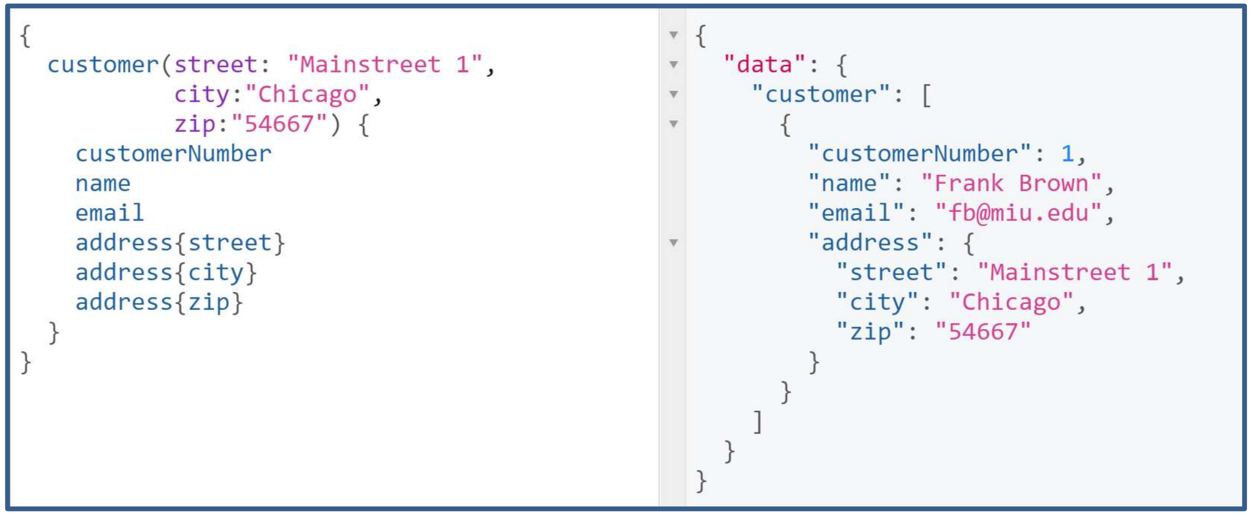
```
mutation {
  createCustomer(customerNumber: 1,
    name: "Frank Brown",
    email: "fb@miu.edu",
    phone: "32421234",
    street: "Mainstreet 1",
    city: "Chicago",
    zip: "54667") {
    customerNumber
    }
}
```

```
{
  "data": {
    "createCustomer": {
      "customerNumber": 1
    }
  }
}
```



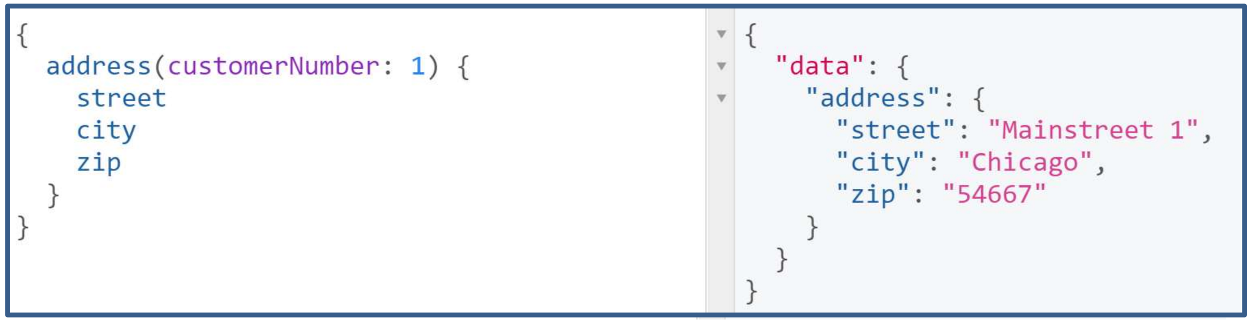
```
mutation {
  createCustomer(customerNumber: 2,
    name: "John Doe",
    email: "jd@gmail.edu",
    phone: "764839332",
    street: "Mainstreet 8",
    city: "Chicago",
    zip: "54667") {
    customerNumber
    }
}
```

```
{
  "data": {
    "createCustomer": {
      "customerNumber": 2
    }
  }
}
```



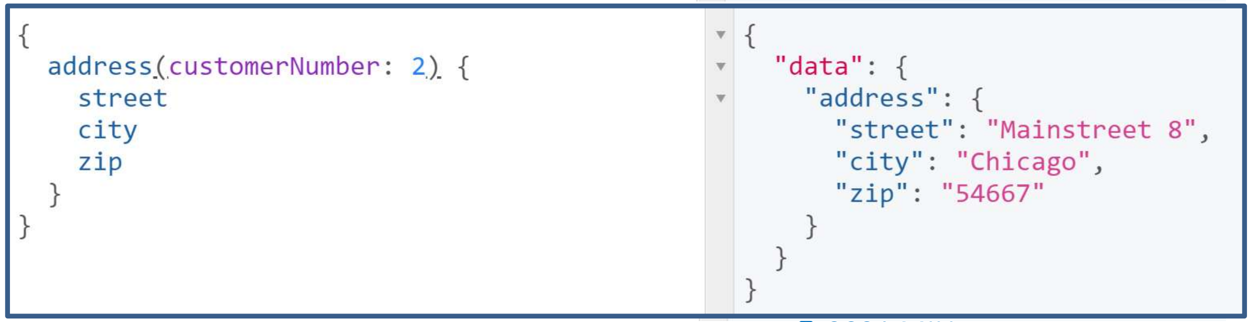
```
{
  customer(street: "Mainstreet 1",
           city:"Chicago",
           zip:"54667") {
    customerNumber
    name
    email
    address{street}
    address{city}
    address{zip}
  }
}
```

```
{
  "data": {
    "customer": [
      {
        "customerNumber": 1,
        "name": "Frank Brown",
        "email": "fb@miu.edu",
        "address": {
          "street": "Mainstreet 1",
          "city": "Chicago",
          "zip": "54667"
        }
      }
    ]
  }
}
```



```
{
  address(customerNumber: 1) {
    street
    city
    zip
  }
}
```

```
{
  "data": {
    "address": {
      "street": "Mainstreet 1",
      "city": "Chicago",
      "zip": "54667"
    }
  }
}
```



```
{
  address(customerNumber: 2) {
    street
    city
    zip
  }
}
```

```
{
  "data": {
    "address": {
      "street": "Mainstreet 8",
      "city": "Chicago",
      "zip": "54667"
    }
  }
}
```

GraphQL disadvantages

- Error handling is more complex
  - HTTP response code is always 200
- Caching is simpler with REST because of multiple endpoints

Main point

- With graphQL you only have one URL to which you can send queries. *TM is a simple technique that allows you to access and experience pure consciousness, the source of all the laws of nature.*

Connecting the parts of knowledge with the wholeness of knowledge

1. The main characteristic of REST is that it is resource oriented which can lead to over-fetching, under-fetching or the need to send multiple requests.
2. GraphQL solves these problems by defining a standard query language over HTTP

---

3. **Transcendental consciousness** is the field of all intelligence.
4. **Wholeness moving within itself:** In Unity Consciousness, all of the intelligence and structure at the basis of the universe is realized as the lively qualities of one's own inner intelligence.