GraphQL

## Overview

|  |  |
| --- | --- |
| Question | Details |
| What is GraphQL | GraphQL consists of a schema definition language and server-side implementations in multiple languages. Client use queries to query the server |
| Where does it beat REST (1)? | Each client can request a different subset of response fields preventing us sending unwanted data over the wire and wasting network resources. |
| Where does it beat REST (2)? | GraphQL is not bound to any transport but it is mostly used over HTTP. |
| Where does it beat REST (3)? | Multiple queries can be batched in a single request |
| Where does it beat REST (4)? | Clients are abstracted from where data comes from. The GraphQL provides an extra level of indirection. Clients get flexibility as to how they pull things together. |

## JavaScript (Apollo)

All sample in that follow are based on a template that can be found [here](https://bitbucket.org/kennyrnwilson/graphqlbasics/src/master/node/templateproject/).

### GraphQL Query and Resolvers

[Code](https://bitbucket.org/kennyrnwilson/graphqlbasics/src/master/node/templateproject/).

A GraphQL service consists of two parts.

* A schema that defines queries and types
* A set of resolvers that provide data when given a query that conforms to the schema.

In the below code I have colour coded the queries and the resolver function that map to them.

export const queryDefinition = `

    type Fruit {

        name:String!

        color:String!

    }

    type Car {

        make: String!

        model: String!

    }

    type Query {

**fruit: Fruit!**

**car: Car!**

    }

`;

export const resolvers = {

    Query : {

**fruit() {**

**return (**

**{**

**name: "Apple",**

**color: "Red"**

**}**

**);**

**},**

**car() {**

**return (**

**{**

**make: "Ford",**

**model: "Fiesta"**

**}**

**)**

**}**

    }

};

### Resolvers for properties

Resolvers do not only exist at the query level. Consider the following fragment. A resolve function is used to resolve a persons address.

export const queryDefinition = `

    type Address {

        street:String!

        number:Int!

    }

    type Person {

        first: String!

        second: String!

**address: Address!**

    }

    type Query {

        person: Person!

    }

`;

export const resolvers = {

    Query : {

        person: () => ({ first: "Ken",second: "Wilson" })

    },

    Person : {

**address: () => ({street:"South Road", number:37})**

    }

};

### Arguments

[Code](https://bitbucket.org/kennyrnwilson/graphqlbasics/src/master/node/arguments/)

The following code shows how to add arguments to queries and fields. We need to add arguments to the query definition and enhance the resolver function to handle them. For details on the arguments passed to a resolver using Apollo server see

<https://www.apollographql.com/docs/apollo-server/data/resolvers/>

export const queryDefinition = `

  type Address {

      type: String!

      street:String!

      number:Int!

  }

    type Person {

        first: String!

        second: String!

        age: Int!

        address(type:String): Address!

    }

    type Query {

        person(name:String!): Person

    }

`;

export const resolvers = {

    Query : {

        person: (parent,args,ctx,info) => {

            console.log(args.name);

            return people.find(x=> x.first === args.name)

        }

    },

    Person : {

        address: (parent,args,ctx,info) =>

addresses.find(x=>x.type === args.type)

    }

};

var people = [

        { first: "Ken",second: "Wilson", age:45 },

        { first: "John",second: "smith", age:30 }

]

var addresses = [

    { type: "Primary", street : "South Road", number: 12},

    { type: "Secondary", street : "Worple Road", number: 148}

]

### Data Sources

Apollo server has the concept of data sources. The resolver does not care how we do things. The data source encapsulates how we get the data. We need an abstraction called DataLoader. We create it as follows.

import { DataSource } from 'apollo-datasource';

class PersonDataSource extends DataSource {

    async getPerson(firstName: string) {

        return this.people.find(x => x.first == firstName);

    }

    private people = [

        { first: "Ken", second: "Wilson", age: 45 },

        { first: "John", second: "smith", age: 30 }

    ]

}

export const PersonAPI = new PersonDataSource();

Then we register all data sources with the ApolloServer.

const server:ApolloServer =

new ApolloServer({ schema, dataSources:()=> ({  personAPI: PersonAPI })  });

Finally, we adjust our resolver function to use the data source.

export const resolvers = {

    Query : {

        person: (parent,args,ctx,info) => {

            return ctx.dataSources.personAPI.getPerson(args.name);

        }

    },

### N+1 Problem

The N+1 problem describes a situation where we ask a database for a set of rows. Then for each of those we make another separate call to another service. This generates unnecessary network traffic. The key is to use a concept of a data loader. The data loader ensures we do not make a remote call more than necessary by caching and batching requests. In the below we do not have any remote call, but it shows the concept.

<https://github.com/graphql/dataloader>

class AddressSource extends DataSource {

    private dataLoader;

    constructor() {

        super();

        this.dataLoader = new DataLoader(async (addressIds:number[]) => await this.getAddresses(addressIds));

    }

    async getAddress(addressId: number)

    {

        return this.dataLoader.load(addressId);

    }

    async getAddresses(addressIds: number[]) {

        return Promise.resolve(this.addresses.filter(x=> addressIds.findIndex(y=>y===x.id) != -1));

    }

    private addresses = [

        { id: 1, street: "Worple Road", number: 21 },

        { id: 2, street: "Maple Road", number: 43 },

        { id: 3, street: "London Road", number: 14 }

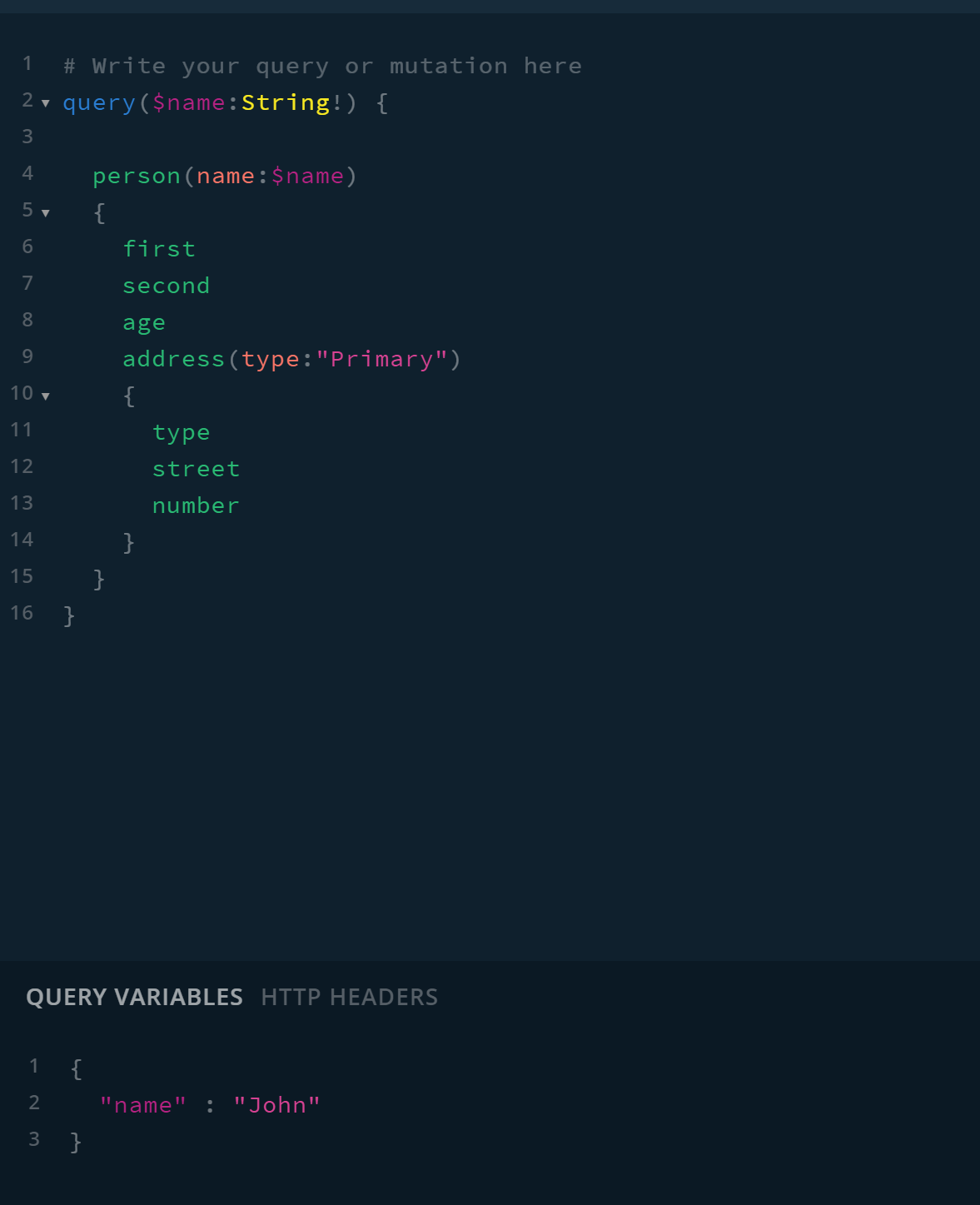
    ]

}

## Query Language

### Query Variables

Query variables enable a single query to be used with multiple input values.



### Aliases

Aliases enable us to make multiple requests in a single operation.

query {

first: person(name:"John")

{

first

second

age

}

second: person(name:"Ken")

{

first

second

age

}

}

The response looks like this

{

"data": {

"first": {

"first": "John",

"second": "smith",

"age": 30

},

"seccond": {

"first": "Ken",

"second": "Wilson",

"age": 45

}

}

}

### Fragments

Fragments prevent us cutting and pasting the same field sets.

fragment myfields on Person {

first,second,age

}

query {

first: person(name:"John")

{

...myfields

}

seccond: person(name:"Ken")

{

...myfields

}

}

## Appendices

### Node.JS

#### Dependencies

npm install graphql –save

|  |  |  |
| --- | --- | --- |
| Package | Link | Description |
| graphql | <https://graphql.org/graphql-js/> | JavaScript GraphQL package. |
| express | <https://expressjs.com/> | Node.js Web framework |
| apollo-server-express | <https://www.npmjs.com/package/apollo-server-express> | Express integration of GraphQL Server |
| graphql-voyager | https://github.com/APIs-guru/graphql-voyager | Visualize a schema |

#### Hello World

### DotNet Examples

### Simple Examples

Consider the following POCO class that we would want to expose in a query

public class Person

{

public int Age { get; set; }

public string FirstName { get; set; }

}

We now create a graph type that describes objects of type Person.

public class PersonType : ObjectGraphType<Person>

{

public PersonType()

{

Field(x=>x.Age);

Field(x=>x.FirstName);

}

}

We add a query that returns a person

public class PersonQuery : ObjectGraphType

{

public PersonQuery()

{

Field<PersonType>( "getPerson",

resolve: ctx => new Person() {

Age = 21,

FirstName = "Dave"

});

}

}

Now we need a schema that has a single top-level query.

var schema = new Schema {Query = new PersonQuery()};

Now we execute a query on the schema

var json = await schema.ExecuteAsync(\_ =>

{

\_.Query = @"

{

getPerson {

age,

firstName,

}

}";

\_.Root = schema;

});

The result is then given by

{  
  "data": {  
    "getPerson": {  
      "age": 21,  
      "firstName": "Dave"  
    }  
  }  
}

### List Examples

Let us extend our example from the previous section to include a second query that return multiple people.

public class PersonQuery : ObjectGraphType

{

public PersonQuery()

{

Field<PersonType>( "getPerson",

resolve: ctx => new Person() {

Age = 21,

FirstName = "Dave"

});

Field<ListGraphType<PersonType>>( "getPeople",

resolve: ctx => new List<Person>

{

new Person {Age=21, FirstName="John"},

new Person {Age=45, FirstName="Ken"},

});

}

}

Our query becomes

getPeople {

age,

firstName

}

Our result is then

{  
  "data": {  
    "getPeople": [  
      {  
        "age": 21,  
        "firstName": "John"  
      },  
      {  
        "age": 45,  
        "firstName": "Ken"  
      }  
    ]  
  }  
}

### Arguments

We can add arguments to queries

public class PersonQuery : ObjectGraphType

{

private List<Person> \_people = new List<UserQuery.Person>()

{

new Person {Age=21, FirstName="John"},

new Person {Age=45, FirstName="Ken"},

};

public PersonQuery()

{

Field<PersonType>( "getPerson",

arguments: new QueryArguments(

new QueryArgument<StringGraphType> {Name="name"}),

resolve: ctx =>

\_people.FirstOrDefault(x =>

x.FirstName == (string)ctx.Arguments["name"])

);

Field<ListGraphType<PersonType>>( "getPeople",

resolve: ctx => new List<Person>

{

new Person {Age=21, FirstName="John"},

new Person {Age=45, FirstName="Ken"},

});

}

}

Our query becomes

getPerson(name:""John"") {

age,

firstName,

}

And our result becomes

{  
  "data": {  
    "getPerson": {  
      "age": 21,  
      "firstName": "John"  
    }  
  }  
}