# Case Study: GraphQL

CMPU4023 - Enterprise Application Development

# What is GraphQL?

- GraphQL (from Facebook) is a library that provides an API specification and dynamic query language for building self-documenting service APIs
- The interfaces produced from GraphQL allow the API client to ask for just what it needs and nothing more
- A GraphQL interface specification is organised as a set of container data types and member fields rather than endpoints (a la REST)
- This allows multiple resources to be fetched and updated in a single API request
- GraphQL specifications can be evolved in a forwardly and backwardly compatible way without the need for a formal versioning scheme

# Reference Implementation

- Although GraphQL is an API specification and query language, it must be incarnated in some service runtime to be actually useful
- By runtime, we mean some service language and execution environment that binds the API to some real-world function or capability
- The Facebook reference implementation, as of 2017, is in Javascript and NodeJS although bindings exist for many popular enterprise languages such as Java, C#, Go, and C/C++
- In the following overview, we will consider the key GraphQL features through the reference implementation
- Ref: http://graphql.org/graphql-js

# Queries and Mutations

- GraphQL allows for the specification of both reads (called queries) and updates (called mutations)
- In both scenarios, the client interfaces with an abstraction in the form of container objects and member fields
- How these query and mutation operations are actually carried out is an service-side implementation detail which is independent from the API specification
- It is the responsibility of the service to perform the internal mapping of the GraphQL-specified operations
- GraphQL is service-agnostic in this sense

# Schema

- A GraphQL schema is a specification of the supported API operations
- A schema specifies the operation type (i.e. query or mutation) and the objects to be read or written by that operation type
- The objects also have a specified type which can be one of the built-in types like `String` or a user-defined composite type (i.e. object)
- Each object operation is implemented as a resolver which is provided as an accessor or mutator function
- The resolver argument and return types must match those of the corresponding schema definitions

# Simple Query

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

// Construct a schema, using GraphQL schema language
const schema = buildSchema(`
  type Query {
    hello: String
  }
`);

// The root provides a resolver function for each API endpoint
const root = {
  hello: () => {
    return 'Hello world!';
  },
};

// Run the GraphQL query '{ hello }' and print out the response
graphql(schema, '{ hello }', root).then((response) => {
  console.log(response);
});
```

# Executing the Query

- The example can be run in NodeJS

```
npm init
npm install graphql --save
node server.js
```

- The output will show the following

```
{ data: { hello: 'Hello world!' } }
```

# GraphQL in a HTTP Server

- A GraphQL server can be embedded within an express server as a middleware, optionally exposing interactive query composer

```
const express = require('express');
const graphqlHTTP = require('express-graphql');
.
.
const app = express();
app.use('/graphql', graphqlHTTP({
  schema: schema,
  rootValue: root,
  graphiql: true,
}));
app.listen(4000);
console.log('Running a GraphQL API server at localhost:4000/graphql');
```

# HTTP Client

- With the GraphQL interface responding over HTTP, any HTTP client can execute the defined query or mutation operations

```
$ curl -X POST -H "Content-Type: application/json" -d '{"query": "{
hello }"}' http://localhost:4000/graphql
```

```
fetch('http://localhost:4000/graphql', {
    method: 'POST',
    headers: new Headers({
        'Accept': 'application/json',
    }),
    body: JSON.stringify({
        query: '{ hello }'  // Note: query is a structured string
    })
});
```

# Interactive Query Composer

- An interactive query composer is enabled with `graphql` set to **`true`** on the Express server startup


- The developer can author and test queries and mutations through this interface before deploying to a real-world production environment

# Types

- GraphQL includes built-in data types such as ID, Int, Float, String and Boolean - can be marked are required or allowed to be null
- User-defined composite types can be built from native types, composite types and arrays of either

```
type Product {
  id: ID!
  name: String!
}

type Order {
  customerId: ID!
  products: [Product]!
}
```

# Query Parameters

- Queries can be parameterised using one or more typed arguments

```
const schema = buildSchema(`
  type Query {
    getProduct(id ID!): Product
  }
`);

// Resolvers
const root = {
  getProduct: function ( { id } ) {
    // Logic for querying product into result ...
    return result;
  }
};
```

# Mutations

- Mutations are how updates can be specified and performed

```
const schema = buildSchema(`
  type Mutation {
    createProduct(product Product!): ID!
  }
`);

// Resolvers
const root = {
  createProduct: function ({product}) {
    // Logic for creating a new product record, returning an id ...
    return id;
  }
};
```

# GraphQLSchema Constructor

- In the examples so far, we've seen the schema and types constructed using the GraphQL DSL within the `buildSchema()` function
- While fine for small interface definitions, larger projects would need to manage many queries and types across dozens or even hundreds of source files
- For this reason, GraphQL offers a rich set of corresponding native language data types

# Native Schema Construction

```
const productType = new graphql.GraphQLObjectType({
  name: 'Product',
  fields: {
    id: { type: graphql.GraphQLID },
    name: { type: graphql.GraphQLString },
  }
});
const queryType = new graphql.GraphQLObjectType({
  name: 'Query',
  fields: {
    product: {
      type: productType,
      args: {
        id: { type: graphql.GraphQLString }
      },
      resolve: function (_, {id}) {
        // Logic to fetch the product ...
      }
    }
  }
});
const schema = new graphql.GraphQLSchema({query: queryType});
```

# Self-Documenting Interfaces

- Using the native schema construction approach allows the developer to document each of the supported operations with details of the objects and data types returned
- A snapshot static API specification document can be generated automatically or dynamically through the online GraphQL composer when authoring and testing queries
- The GraphQL design model encourages the coupling of living documentation to the API specification

# Summary

- GraphQL is a library that provides an API specification and dynamic query language for building self-documenting service APIs
- A GraphQL schema specifies the operation type (i.e. query or mutation) and the objects to be read or written by that operation type
- A GraphQL server can be embedded within a HTTP server, optionally exposing interactive query composer
- GraphQL includes built-in data types and supports user-defined composite types
- Schema can be authored using the GraphQL DSL or with native host language data types for better organisation and project scalability

# Case Study: GraphQL

CMPU4023 - Enterprise Application Development