Extra Lab 1: Introduction and hands on **GraphQL**[1]

1. Purpose of this lab

In the previous set of labs, we learned about Node JS and wrote a simple – yet flexible and robust – Web Server using `ExpressJS` based on the REST architecture. Today, we are going to recreate that same API using `GraphQL`. This will allow you to better understand the `GraphQL` standard and its core difference when compared with the *REST API*.

The exercises in this lab are based on the conceptual model presented in Lab 2 (the chat application) and it is assumed that you already have the database set up. Please, go back to Labs 1, 2, and 3 if you need to recap concepts around the previous labs or talk with one of your tutors.

2. What is GraphQL?

"`GraphQL` is a new API standard that provides a more efficient, powerful and flexible alternative to REST. It was developed and open-sourced by Facebook and is now maintained by a large community of companies and individuals from all over the world.

At its core, `GraphQL` enables declarative data fetching *where a client can specify exactly what data it needs from an API. Instead of multiple endpoints that return fixed data structures, a `GraphQL` server only exposes a single endpoint and responds with precisely the data a client asked for."* (HOW TO GRAPHQL-> Basics Tutorial – Introduction: *https://www.howtographql.com/basics/0-introduction/*)

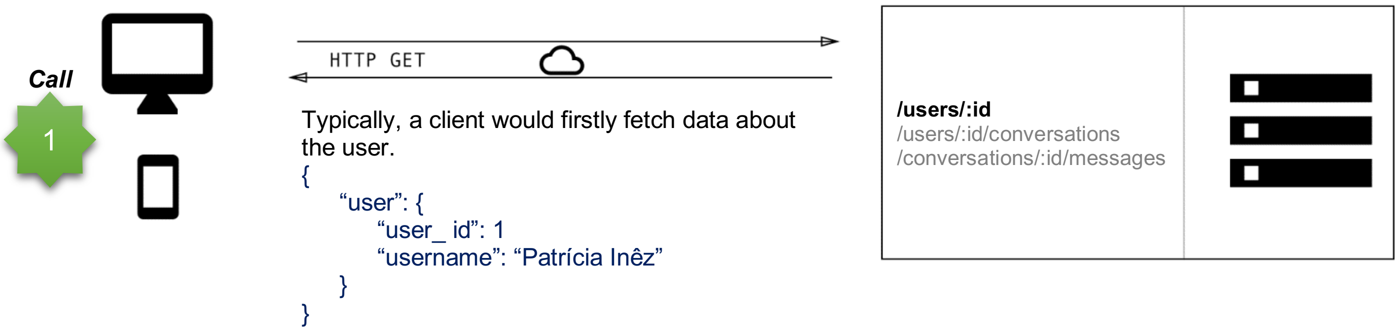
2.1. GraphQL x REST

Over the years, REST has become the de-facto standard for software architecture when designing web APIs, and although it has great features, it is less flexible when dealing with recurring changes in requirements.

Images 1 and 2 illustrate the major differences between the two approaches, simulating that the client wants to *fetch all the messages from the conversations he is engaged with*.

2.1.1. Fetching with REST API

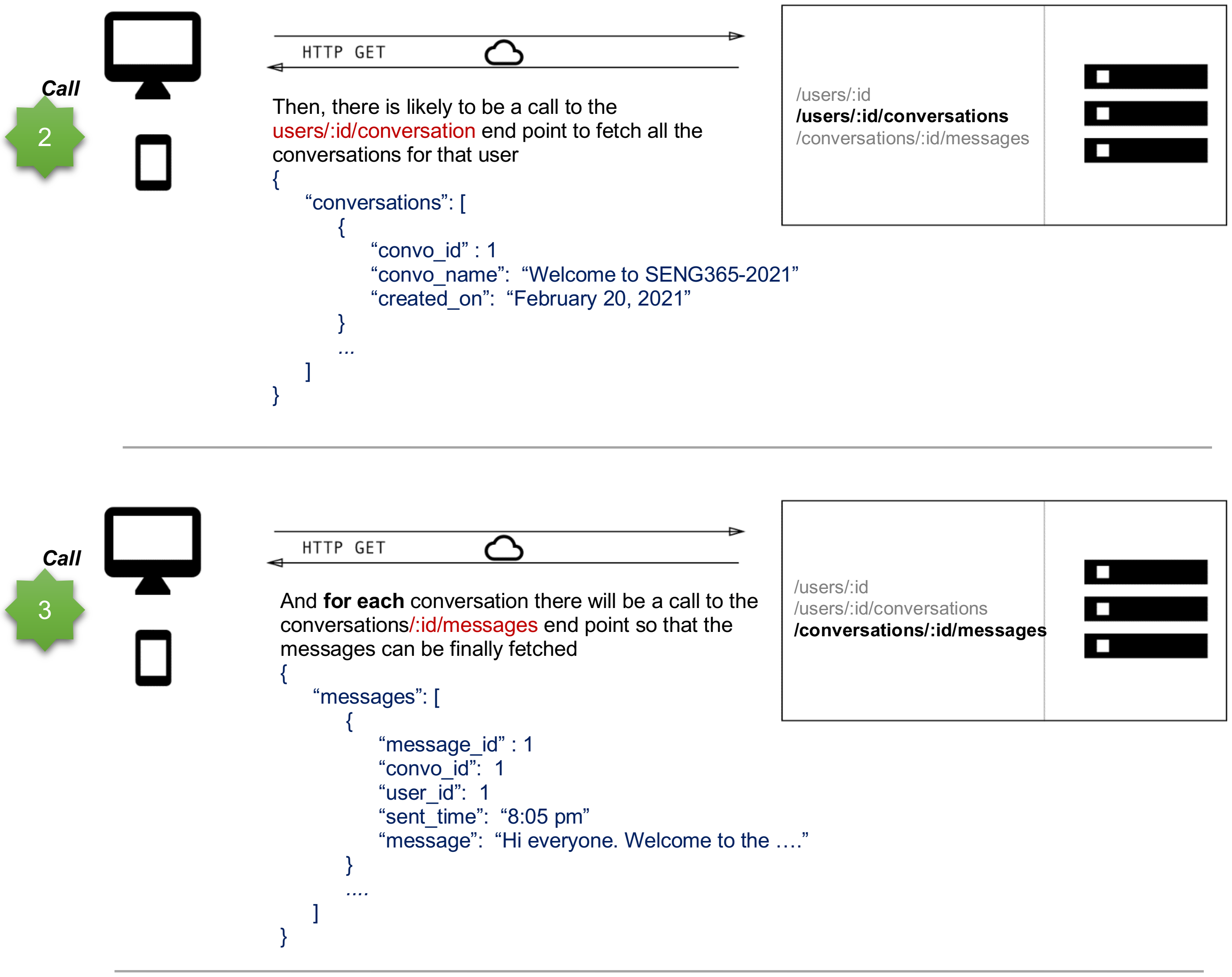
With a REST API server, considering the user is engaged in only a single conversation, there will likely be at least three calls from the client to the server as depicted below:



Typically, a client would firstly fetch data about the user.

```
{
    "user": {
        "user_ id": 1
        "username": "Patrícia Inêz"
    }
}
```

[1] This handout is based on a collection of sources about the matter, especially those available at '**HOW TO GRAPHQL**' website (https://www.howtographql.com/), which you are strongly recommended to look at.


Call
2
HTTP GET
/users/:id
/users/:id/conversations
/conversations/:id/messages

Then, there is likely to be a call to the users/:id/conversation end point to fetch all the conversations for that user

```
{
    "conversations": [
        {
            "convo_id" : 1
            "convo_name":  "Welcome to SENG365-2021"
            "created_on":  "February 20, 2021"
        }
        ...
    ]
}
```

Call
3
HTTP GET
/users/:id
/users/:id/conversations
/conversations/:id/messages

And **for each** conversation there will be a call to the conversations/:id/messages end point so that the messages can be finally fetched

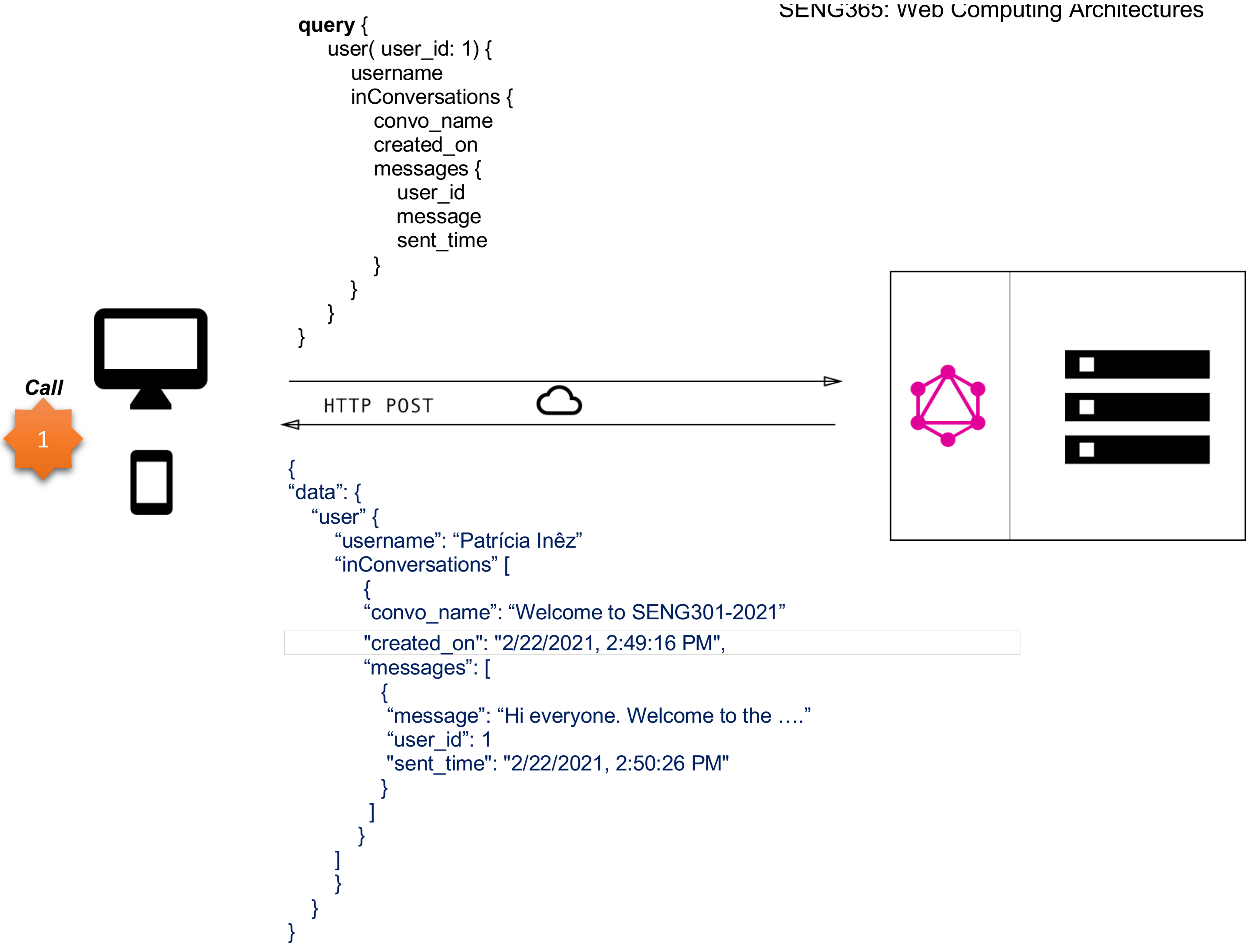
```
{
    "messages": [
        {
            "message_id" : 1
            "convo_id":  1
            "user_id":  1
            "sent_time":  "8:05 pm"
            "message":  "Hi everyone. Welcome to the ...."
        }
        ....
    ]
}
```

Image 1: Fetching with REST API

2.1.2. Fetching with GraphQL API

For APIs implemented using `GraphQL` standard, the same data could be fetched with a single call to the server, which would then respond with `JSON` with all the data requested. This means:

- No more over/under fetching as the `GraphQL` eliminates fixed data structures being returned by endpoints.
- Less networking traffic: one call is likely to be enough to fetch all the needed data, as long as it is in the API schema (we will learn about schemas further in this tutorial).
- More flexibility and autonomy on the client's side, with the possibility to customize the response object with the properties needed, again, as long as it is in the API schema, which also implies in rapid iterations.
- Thanks to the flexible nature of `GraphQL`, changes on the client-side can be made without refactoring the server side, which as a side-effect, incurs in higher stability.



```
query {
    user( user_id: 1) {
        username
        inConversations {
            convo_name
            created_on
            messages {
                user_id
                message
                sent_time
            }
        }
    }
}
```

```
{
"data": {
    "user" {
        "username": "Patrícia Inêz"
        "inConversations" [
            {
            "convo_name": "Welcome to SENG301-2021"
            "created_on": "2/22/2021, 2:49:16 PM",
            "messages": [
              {
                "message": "Hi everyone. Welcome to the ...."
                "user_id": 1
                "sent_time": "2/22/2021, 2:50:26 PM"
              }
            ]
          }
        ]
      }
    }
}
```

Image 2: Fetching with GraphQL

3. GraphQL architecture & core concepts

3.1. Types & schemas

"GraphQL uses a strong type system to define the capabilities of an API. All the types that are exposed in an API are written down in a schema using the GraphQL Schema Definition Language (SDL). This schema serves as the contract between the client and the server to define how a client can access the data. Once the schema is defined, the teams working on frontend and backend can do their work without further communication since they both are aware of the definite structure of the data that's sent over the network." (HOW TO GRAPHQL -> GraphQL is the better REST: *https://www.howtographql.com/basics/1-graphql-is-the-better-rest/*)

Snippet 1 shows how a type declaration looks like:

```
const UserType = new GraphQLObjectType( {
    name: 'User',
    fields: {
        user_id: { type: GraphQLID },
        username: { type: GraphQLString },
        inConversations: {
            type: GraphQLList(ConversationType),
            resolve: ( root, _, context) => {
                return context.convRepository.findByUser( root.user_id );
            },
        },
    },
} );
```

Snippet 1: `UserType` type definition

Note that:

1. The SDL has five scalar (primitive) types: **`Int`**, **`Float`**, **`String`**, **`Boolean`** and **`ID`.** All other types need to be created by you. Please, refer to https://www.prisma.io/blog/graphql-sdl-schema-definition-language-6755bcb9ce51 for further information.
2. A type definition per se does not expose any data to client applications; it simply defines the structure for that specific type. In order to expose the data to the API, we need to add fields to the root types of the GraphQL schema: **`Query`**, **`Mutation`**, and **`Subscription`** – which define the *entry points* for a GraphQL API.

Snippet 2 illustrates a schema.

```
const schema = new GraphQLSchema({
   query: new GraphQLObjectType({
      name: 'Query',
      fields: {
          user: {
              type: UserType,
              args: {
                  user_id: {type: GraphQLID},
              },
              resolve: (_, {user_id}, context) => {
                  return context.userRepository.findById(user_id)
              },
          },
      },
   }),
});
```

Snippet 2: `Schema` type definition

In the above snippet, an **entry point** named **`user`** is being defined for the **root type `Query`**, which returns a **`UserType`** object *(defined in snippet 1)* and expects a **`GraphQLID`** parameter named **`id`**. Assuming the function **`findById`** is already implemented and returns a **`UserType`** instance *(a JSON object with id and name fields)*, the resolve function now enables execution of the schema.

3.2. Resolvers

Resolvers define the concrete implementation of each of the schema and type fields. Without a resolver, the entry point would have no use for the client, as nothing would be performed. A resolver function has the following three main arguments:

- **`root`**: Also called as parent, the **`root`** carries the object from the previous resolve, allowing one resolver to have access to the context of its predecessor, that's why it is given the name root or parent. Note that in the above example, the root object is null (first entry point in the chain), therefore it was omitted.
- **`args`**: This object carries the set of parameters for the query formatted as a JSON object, thus, in the above example, the early extraction of its pair (key-value) `id`.

- `context`: An object that gets passed through the resolver chain and that each resolver can write to and read from (basically a means for resolvers to communicate and share information).

Note that resolvers are not needed where the GraphQL can infer the returned value, such as in the `id` and `name` fields in the `UserType` type definition.

3.3. Data fetching

As discussed in topic 2.1 (`GraphQL` x `REST`), `GraphQL` has a different approach to loading data. Instead of having multiple endpoints that return fixed data structures, GraphQL APIs typically expose a single entry point (usually named as '**graphql'**) and the structure of the response object is completely up to the client. This "*means that the client needs to send more* information *to the server to express its data needs - this information is called a* query.*"* (HOW TO GRAPHQL -> Core Concepts: https://www.howtographql.com/basics/2-core-concepts/)

To illustrate this, considering the type and schema definitions from previous sections, a client could – at its own discretion – customize a query payload (fields to be fetched) in response to the query.

```
query {
  user(user_id: 1)
  {
     username
  }
}
```

```
query {
  user(user_id: 1)
  {
     user_id
     username
  }
}
```

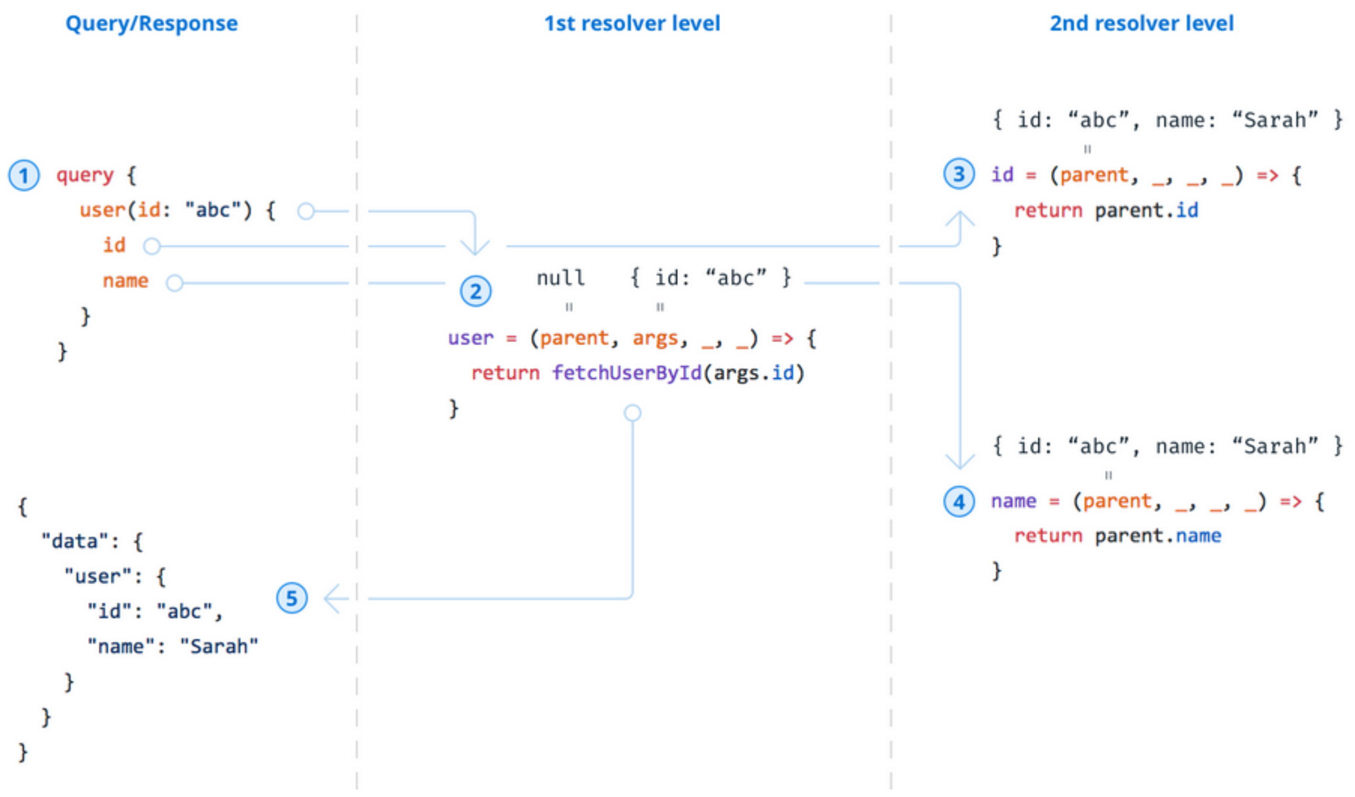
```
query {
  user(user_id: 1) {
     user_id
     username
     inConversations{
        convo_name
     }
  }
}
```

```
query {
  user(user_id: 1) {
     user_id
     username
     inConversations{
        convo_name
        created_on
        messages {
           message
           sent_time
           user_id
        }
     }
  }
}
```

Image 3: Four different queries fetching data from the same entry point

The beauty of it is that with a **single entry point** exposed by the API, the client can fetch many different data structures without any drawback to the server. Image 4 illustrates how a query is executed, helping one to understand how data is collected and parameters and other objects are passed along the chain of resolvers.

The concepts summarized up to this point are thoroughly explained at GraphQL Server Basics: GraphQL Schemas, TypeDefs & Resolvers Explained *(https://www.prisma.io/blog/graphql-server-basics-the-schema-ac5e2950214e)*. Unless you are very confident about everything that was discussed, you are highly encouraged to take time to at least skim it before getting you into the hands-on sections.



```
query {
  user(id: "abc") {
    id
    name
  }
}
```

```
null    { id: "abc" }
user = (parent, args, _, _) => {
  return fetchUserById(args.id)
}
```

```
{ id: "abc", name: "Sarah" }
id = (parent, _, _, _) => {
  return parent.id
}
```

```
{ id: "abc", name: "Sarah" }
name = (parent, _, _, _) => {
  return parent.name
}
```

```
{
  "data": {
    "user": {
      "id": "abc",
      "name": "Sarah"
    }
  }
}
```

Image 4: A query execution illustration *(https://www.prisma.io/blog/graphql-server-basics-the-schema-ac5e2950214e)*

4. Guided hands-on

Through this section, you will implement all the needed code so that the queries from **Image 3** get the expected result.

4.1. Project setup and basic structure (architecture)

1. Create a folder for this lab, and initiate a new NPM project (`` `npm init` ``) inside this new folder. *(Refer to Lab 3 if you are unsure about how to do this).*

2. Using the knowledge gained from Lab 2, create a basic **`Express Server`**, with **`Mysql`** support, configure it to listen on port `3000`.

   There is no need though to create the **`routes`**, **`models`** and **`controllers`** folders as the structure in a **GraphQL** project is a bit different from a REST project. At this stage, all you need is a very basic implementation of the **`express server`**; thus, your implementation could be as simple as the one shown below:

```
const path = require( 'path' );
require( 'dotenv' ).config( { path: path.join( __dirname, 'app/config/.env' ) } );

const db = require( './app/config/db' );
const port = process.env.SENG365_PORT || 3000;
const express = require( 'express' );

const app = express();
app.get('/', (req, res) => {
    res.send('Hello SENG365-2021 team');
});

// Test connection to MySQL on start-up
async function testDbConnection() {
    try {
        await db.connect();
    } catch( err ) {
        console.error( ` Unable to connect to MySQL: ${ err.message }` );
        process.exit( 1 );
    }
}

testDbConnection()
    .then( function() {
        console.log( '. Successfully connected to the database' );
        app.listen( port, function() {
            console.log( `.Listening on port: ${ port }\n` );
        } );
    } );
```

Snippet 3: A very basic express web-server implementation

Note that in the above snippet, the `.env` file was put under the `app/config` folder, meaning either you would need to use the same folder suggestion or change the path to the correct location where your file is.

3. We will be using the same conceptual model used in Lab 2, so doublecheck that the database exists and that the connection test is passing.

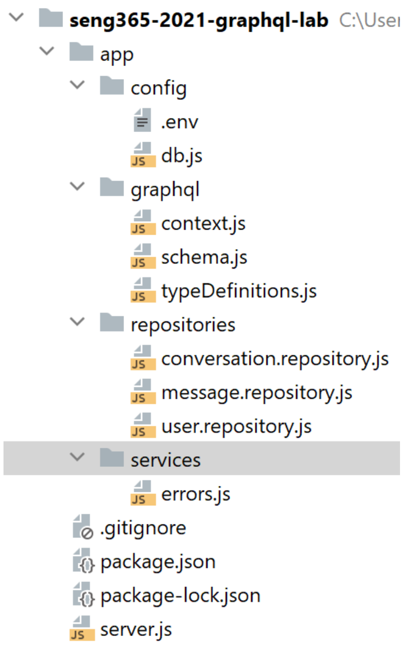
   You may also want to try accessing the endpoint created `'/'` and see if it is responding as expected.

4. Add the following dependencies to your `package.json` file and run `npm install`:

```
"dependencies": {
  "express-graphql": "^0.12.0",
  "graphql": "^15.5.0",
  "graphql-scalars": "^1.7.0",
}
```

Snippet 4: List of dependencies to be added in your project

5. The project structure should be organized as illustrated on image 5.



```
seng365-2021-graphql-lab  C:\User
  app
    config
      .env
      db.js
    graphql
      context.js
      schema.js
      typeDefinitions.js
    repositories
      conversation.repository.js
      message.repository.js
      user.repository.js
    services
      errors.js
  .gitignore
  package.json
  package-lock.json
  server.js
```

Image 5: Project structure

In regards to the files under the `graphql` folder:

- The `context.js` module defines the `context` object passed through the resolver chain;
- The `schema.js` module defines the functionalities that will be exposed to the clients, that is, the API entry points; and
- The `typeDefinitions.js` module defines the structure for that specific types handle by the server, which in our case are `UserType`, `ConversationType`, `MessageType` and `GraphQLDate`.

The `repositories` folder is where the code to handle the database objects will be stored.

4.2. Schema and type definitions

1. Start by implementing the functionalities to retrieve the application conceptual model. For the sake of organization and favouring the Single Responsibility Principle, there we will be one module for each entity:

```
const db = require( '../config/db' );
const errors = require( '../services/errors' );

const findByConversation = async function( convId ) {
    const sqlStatement = 'SELECT * FROM lab2_messages WHERE `convo_id` = ?';
    try {
        const [ messages ] = await db.getPool().query( sqlStatement, [ convId ] );
        return messages;
    } catch( err ) {
        errors.logAndThrowSqlError( 'Error accessing the database: ' + err );
    }
};
module.exports = { findByConversation }
```

Snippet 5: `message.repository` module

```
const db = require( '../config/db' );
const errors = require( '../services/errors' );

const findByUserId = async function( userId ) {
    const sqlStatement = 'SELECT * FROM lab2_users_in_conversation WHERE user_id = ?';
    try {
        const [ usersInConversations ] = await db.getPool().query( sqlStatement, [ userId
] );
        const conversations = []
        if( usersInConversations.length >= 1 ) {
            usersInConversations.forEach( row => {
                conversations.push( findById( row.convo_id ) );
            } );
        }
        return conversations;
    } catch( err ) {
        errors.logAndThrowSqlError( "Error accessing the database: " + err );
    }
};

const findById = async function( convId ) {
    const sqlStatement = 'SELECT * FROM lab2_conversations WHERE convo_id = ?';
    try {
        const [ conversations ] = await db.getPool().query( sqlStatement, [ convId ] )
        if( conversations.length < 1 ) {
            return null;
        } else {
            return conversations[ 0 ];
        }
    } catch( err ) {
        errors.logAndThrowSqlError( "Error accessing the database: " + err );
    }
};

module.exports = { findByUserId, findById }
```

Snippet 6: `conversation.repository` module

```
const db = require( '../config/db' );
const errors = require( '../services/errors' );

const findAll = async function() {
    const sqlStatement = 'SELECT * FROM `lab2_users`';
    try {
        const [ users ] = await db.getPool().query( sqlStatement );
        return users;
    } catch( err ) {
        errors.logAndThrowSqlError( 'Error accessing the database: ' + err );
    }
};
```

```
const findById = async function( id ) {
    const sqlStatement = 'SELECT * FROM `lab2_users` WHERE user_id = ?';
    try {
        const [ users ] = await db.getPool().query( sqlStatement, id );
        if( users.length < 1 ) {
            return null;
        } else {
            return users[ 0 ];
        }
    } catch( err ) {
        errors.logAndThrowSqlError( 'Error accessing the database: ' + err );
    }
};

module.exports = { findAll, findById }
```

Snippet 7: `user.repository` module

2. The next step is to implement the type definitions so that **`GraphQL`** knows how to handle the system types. This file is in essence built based on the concept model:

```
const { GraphQLID, GraphQLObjectType, GraphQLString, GraphQLList } = require('graphql');
const { GraphQLScalarType } = require( 'graphql' );
const { Kind } = require ('graphql/language');

const GraphQLDate = new  GraphQLScalarType({
    name       : 'GraphQLDate',
    description: 'Date custom scalar type',
    parseValue(value) {
        return new Date(value); // value from the client
    },
    serialize(value) {
        return value.toLocaleString(); // value sent to the client
    },
    parseLiteral(ast) {
        if (ast.kind === Kind.INT) {
            return new Date(ast.value) // ast value is always in string format
        }
        return null;
    },
});

const MessageType = new GraphQLObjectType({
    name: 'Message',
    fields: {
        convo_id: { type: GraphQLID },
        user_id: { type: GraphQLID },
        message: { type: GraphQLString },
        sent_time: { type: GraphQLDate }
    },
});

const ConversationType = new GraphQLObjectType( {
    name: 'Conversation',
    fields: {
        convo_id: { type: GraphQLID },
        convo_name: { type: GraphQLString },
        created_on: { type: GraphQLDate },
        messages: {
            type: GraphQLList(MessageType),
            resolve: ( root, _, context) => {
                return context.messageRepository.findByConversation( root.convo_d );
            },
        }
    },
} );

const UserType = new GraphQLObjectType( {
    name: 'User',
    fields: {
        user_id: { type: GraphQLID },
        username: { type: GraphQLString },
        inConversations: {
            type: GraphQLList(ConversationType),
            resolve: ( root, _, context) => {
                return context.convRepository.findByUserId( root.user_id );
            },
        },
    },
} );
module.exports = { UserType, ConversationType, MessageType };
```

Snippet 8: `typeDefinitions` module

3. Then, the schema definition, with the functionalities we want to expose, which at this point consist of a single query: **`user`**.

```
const { GraphQLSchema, GraphQLObjectType, GraphQLID } = require( 'graphql' );
const { UserType } = require( '../graphql/typeDefinitions' );

const schema = new GraphQLSchema({
    query: new GraphQLObjectType({
        name: 'Query',
        fields: {
            user: {
                type: UserType,
                args: {
                    user_id: { type: GraphQLID },
                },
                resolve: (_, {user_id}, context) => {
                    return context.userRepository.findById(user_id);
                },
            },
        },
    }),
});

module.exports = { schema };
```

Snippet 9: the `schema` module

4. Then, the context definition, which ties together all the repositories into a single context module.

```
const userRepository = require( '../repositories/user.repository' );
const convRepository = require( '../repositories/conversation.repository' )
const messageRepository = require( '../repositories/message.repository' )
const { UserType } = require( '../graphql/typeDefinitions' );

module.exports = { userRepository, convRepository, messageRepository };
```

Snippet 10: the `context` module

5. Then, the errors definition, which allows us to log SQL errors to the console with greater detail.

```
exports.logAndThrowSqlError = function( err ) {
    console.error( `An error occurred when executing: \n${ err.sql } \nERROR: ${ err.sqlMessage }` );
    err.hasBeenLogged = true;
    throw err;
};
```

Snippet 11: the `errors` module

4.3. Starting express-graphql

The last step is to update express server to work with GraphQL:

1. Import the following **`graphqlHTTP`** dependency adding the following line to your express module (**`server.js`**):

```
const { graphqlHTTP } = require( 'express-graphql' );
```

Snippet 10: `graphqlHTTP` import line.

2. Create the **`context`** and **`schema`** objects, as they will be used to set up the **`graphqlHTTP`** server.

```
const context = require( './app/graphql/context' );
const {schema} = require( './app/graphql/schema' );
```

Snippet 11: `context` and `schema` objects declaration

3. Instantiate a **`graphqlHTTP`** object passing the above objects as options, and set up the express server to use it.

   To do so, replace the code below

```
app.get('/', (req, res) => {
    res.send('Hello SENG365-2021 team');
});
```

Snippet 12: `'/'` express endpoint setup

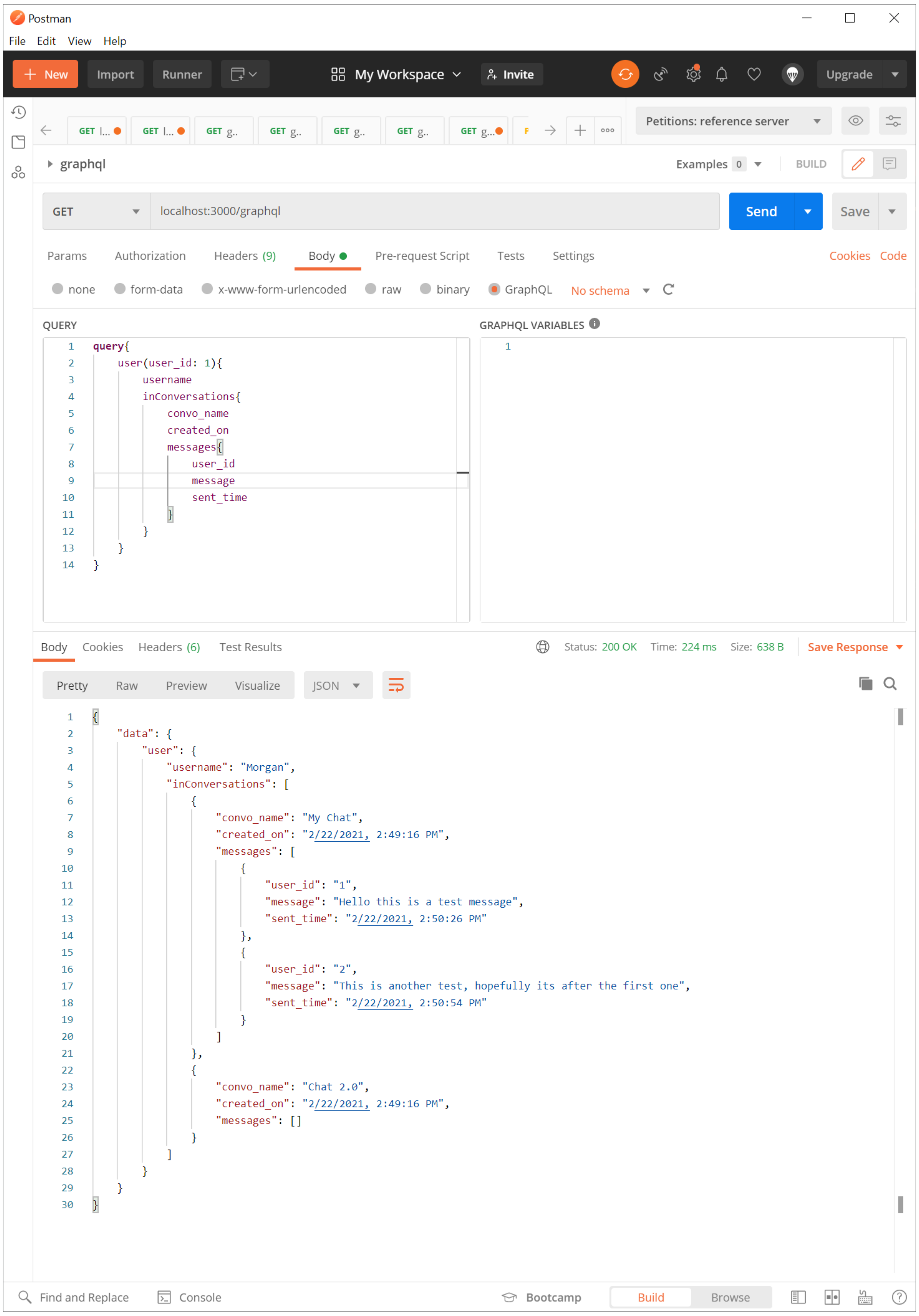
By this one

```
app.use( '/graphql', graphqlHTTP( {
    context,
    schema,
} ) );
```

Snippet 13: `graphqlHTTP` instantiation and express setup

4. Start the server, open the **Postman** application and carry out the queries from image 3.

   If all went well, you should get results as the one illustrates by image 5. If there is no data returning ensure that your database is populated.


Postman
File Edit View Help
New
Import
Runner
My Workspace
Invite
Upgrade
Petitions: reference server
graphql
Examples 0
BUILD
Send
Save
Params Authorization Headers (9) Body Pre-request Script Tests Settings
Cookies Code
Pretty Raw Preview Visualize JSON
Save Response
Find and Replace Console Bootcamp Build Browse

GET localhost:3000/graphql

Body: none form-data x-www-form-urlencoded raw binary [✓ GraphQL] No schema

QUERY

```
query{
    user(user_id: 1){
        username
        inConversations{
            convo_name
            created_on
            messages{
                user_id
                message
                sent_time
            }
        }
    }
}
```

GRAPHQL VARIABLES

Body Cookies Headers (6) Test Results Status: 200 OK Time: 224 ms Size: 638 B

```
{
    "data": {
        "user": {
            "username": "Morgan",
            "inConversations": [
                {
                    "convo_name": "My Chat",
                    "created_on": "2/22/2021, 2:49:16 PM",
                    "messages": [
                        {
                            "user_id": "1",
                            "message": "Hello this is a test message",
                            "sent_time": "2/22/2021, 2:50:26 PM"
                        },
                        {
                            "user_id": "2",
                            "message": "This is another test, hopefully its after the first one",
                            "sent_time": "2/22/2021, 2:50:54 PM"
                        }
                    ]
                },
                {
                    "convo_name": "Chat 2.0",
                    "created_on": "2/22/2021, 2:49:16 PM",
                    "messages": []
                }
            ]
        }
    }
}
```

Image 5: Using **Postman** with **graphql**

The above image also illustrates how to use Postman with GraphQL root types. Please, refer to the GraphQL Support page *(at www.postman.org/graphql)* for further information on that.

5. On your own time

Expand your knowledge about the GraphQL root types and implement mutation entry points to register new users, conversations, and messages.

6. Reference

- GraphQL: https://graphql.org/
- The Fullstack Tutorial for GraphQL: https://www.howtographql.com/
- Basics Tutorial – Introduction: https://www.howtographql.com/basics/0-introduction/
- GraphQL is the better REST: https://www.howtographql.com/basics/1-graphql-is-the-better-rest/
- Schema Definition Language: https://www.prisma.io/blog/graphql-sdl-schema-definition-language-6755bcb9ce51
- Core Concepts: https://www.howtographql.com/basics/2-core-concepts/
- GraphQL Server Basics: GraphQL Schemas, TypeDefs & Resolvers Explained: https://www.prisma.io/blog/graphql-server-basics-the-schema-ac5e2950214e
- Schemas and Types: https://graphql.org/learn/schema/#the-query-and-mutation-types
- Postman GraphQL Support page: www.postman.org/graphql