

Lab 2: Persisting Data, Postman Testing, and Structuring Applications in Node with TypeScript

1 Purpose of this lab

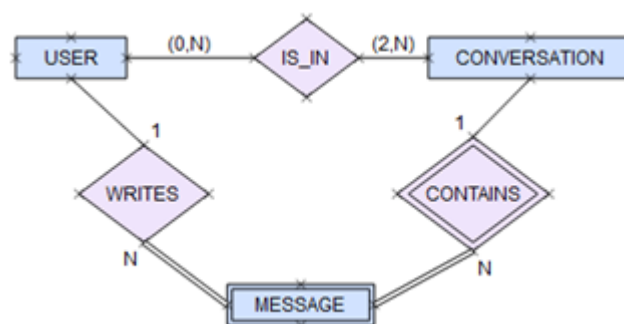
Last week we had an introductory look at using JavaScript, Node, and Express to create a small chat app API. This week's lab expands on these concepts bringing in many new features and technologies. Due to the scope of this lab it may take you more than one lab session to complete, however once completed you should have the necessary skills and knowledge to begin working on your first assignment.

This lab is broken up into several segments. Firstly we will look into data persistence using MySQL. This is followed by the largest and most important section where we will re-create the chat app API from lab 1, this time making use of proper application structure, TypeScript, and a few other node best practices. This will be followed by a look at how to test the API using Postman. Finishing with a small introduction to API specifications, and leaving you to complete the rest of the application to the introduced specification.

2 Persisting Data with MYSQL

2.1 Conceptual Modelling

In the chat application, we will have multiple users that can talk to each other in conversations. Each conversation can contain multiple messages. Each conversation must have at least two users. There is no upper limit on the number of users that can participate in a conversation. Here is an Entity Relationship Diagram (ERD) for the chat application:



(Students should be familiar with ERDs from previous courses; for a refresher, see [Entity-relationship model](#))

2.2 Connecting to the MySQL server

Each student enrolled in SENG365 has a user account on the course's MySQL server. You can access the database both on and off campus.

The connection details are as follows:

Hostname: **db2.csse.canterbury.ac.nz**

Username: your student user code (e.g. **abc123**)

Password: your **student ID** (e.g. **12345678**)

You can manage your database using phpMyAdmin. To access the control panel, go to: dbadmin.csse.canterbury.ac.nz.

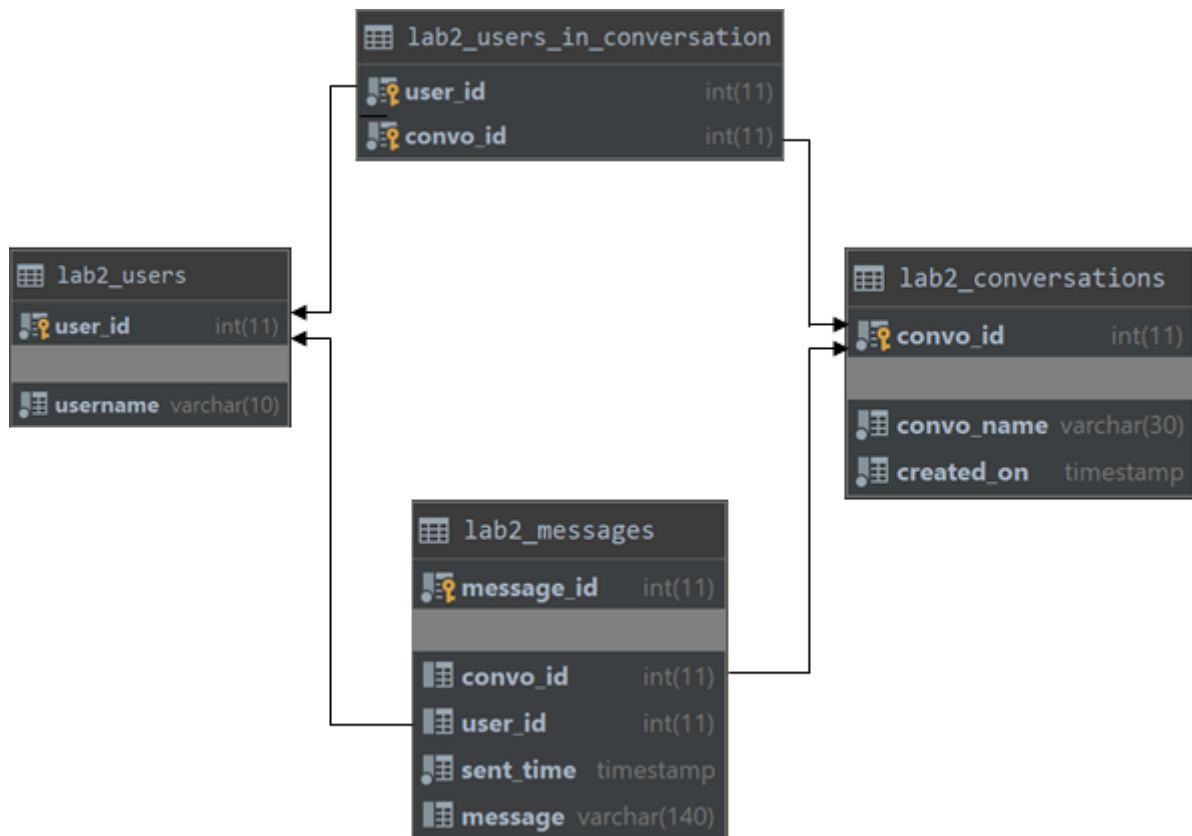
Note: to access the phpMyAdmin you will need two authentications: use your normal UC login and password for the first one, then, the above one.

Once you've logged into phpMyAdmin, you should be able to see the databases beginning with your username, e.g. **abc123_main**. Create a database that you will use solely for this lab, e.g. **abc123_s365_lab2**.



2.3 Creating Tables

Now that we have access to our database, we will create a database using the '**lab2_init.sql**' script (**available on Learn**). This database will have the tables and fields shown below*. Look through the SQL script for more detailed information on the foreign key constraints.

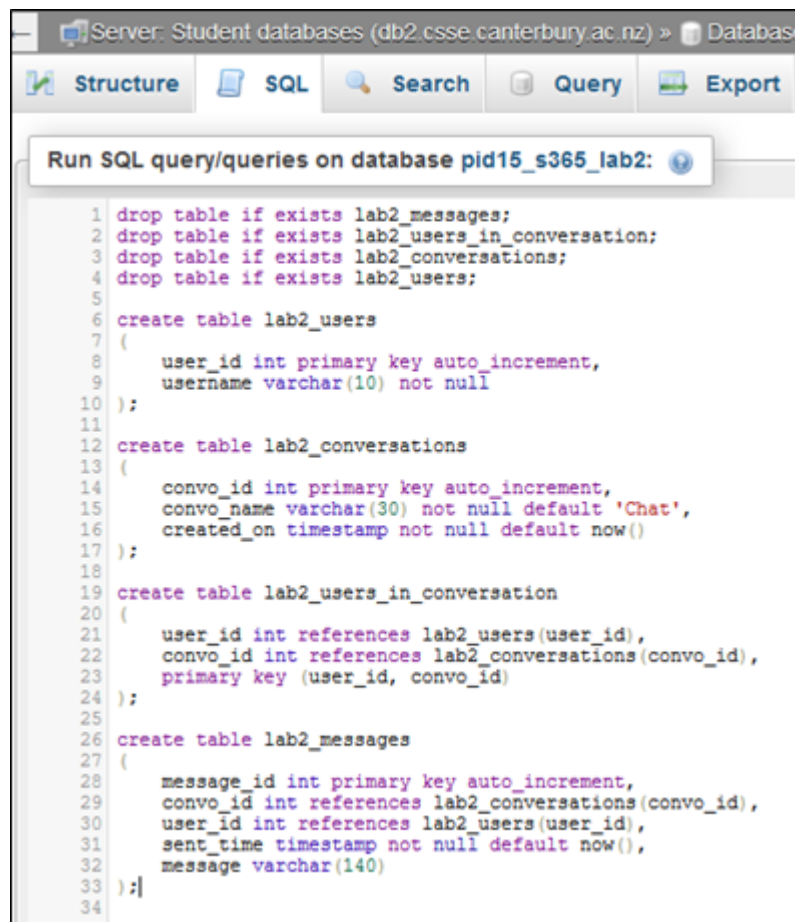


**This database schema is derived from the ER diagram shown above*

We can run the entire SQL script on our database by opening a terminal (*on your local machine*) and running the following command:

```
mysql -h db2.csse.canterbury.ac.nz -u abc123 -D abc123_s365_lab2 -p <
lab2_init.sql
```

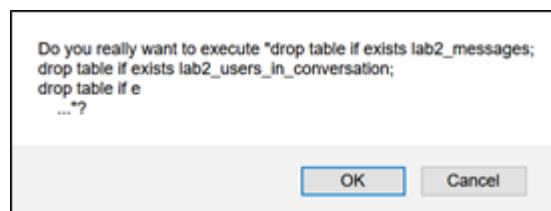
Alternatively, in case you don't have *mysql* installed in your home computer (all labs computers have it installed), you can open a SQL terminal inside the **phpMyAdmin** (make sure you've selected the database for this lab), past the given script content and run it from there:



The screenshot shows a database management interface with a menu bar (Structure, SQL, Search, Query, Export) and a title bar indicating the server is 'Student databases (db2.csse.canterbury.ac.nz)' and the database is 'pid15_s365_lab2'. A text area contains the following SQL code:

```
1 drop table if exists lab2_messages;
2 drop table if exists lab2_users_in_conversation;
3 drop table if exists lab2_conversations;
4 drop table if exists lab2_users;
5
6 create table lab2_users
7 (
8     user_id int primary key auto_increment,
9     username varchar(10) not null
10 );
11
12 create table lab2_conversations
13 (
14     convo_id int primary key auto_increment,
15     convo_name varchar(30) not null default 'Chat',
16     created_on timestamp not null default now()
17 );
18
19 create table lab2_users_in_conversation
20 (
21     user_id int references lab2_users(user_id),
22     convo_id int references lab2_conversations(convo_id),
23     primary key (user_id, convo_id)
24 );
25
26 create table lab2_messages
27 (
28     message_id int primary key auto_increment,
29     convo_id int references lab2_conversations(convo_id),
30     user_id int references lab2_users(user_id),
31     sent_time timestamp not null default now(),
32     message varchar(140)
33 );
34 ;|
```

Press 'ctrl + enter' and confirm the operation.



2.4 Reading database params from environment variables

We do not want to be putting our confidential username and password into code (especially if we put that code under version control!). Therefore, we need a way to inject the necessary details into our application when it runs. This is where **environment variables** come to the rescue!

To do this we will make use of the *dotenv* (<https://www.npmjs.com/package/dotenv>) module later in the lab when we structure our application. This is done by creating a '.env' file at the root of our project with key=value pairs.

Notes

1. **If you are working from home** please follow the steps detailed in the document "Connecting to the University Database from Home" under the labs section on **Learn**, as you will have to modify the .env file and run an ssh command.

2. This file should *never* be version-controlled, i.e. if you're using Git then add it to your `.gitignore`.

3 Setting up our Node application with TypeScript

3.1 Installing node packages with 'package.json'

All Node.js projects contain a file, usually located at the root directory, named '**package.json**' that holds various metadata relevant to the project. This file is used to give information to `npm` that allows it to identify the project as well as handle the project's dependencies. It can also contain other metadata such as a project description, the version of the project in a particular distribution, license information, even configuration data - all of which can be vital to both `npm` and to the end users of the package.

Let's re-create the chat application from last week using the concepts that we learn in this lab.

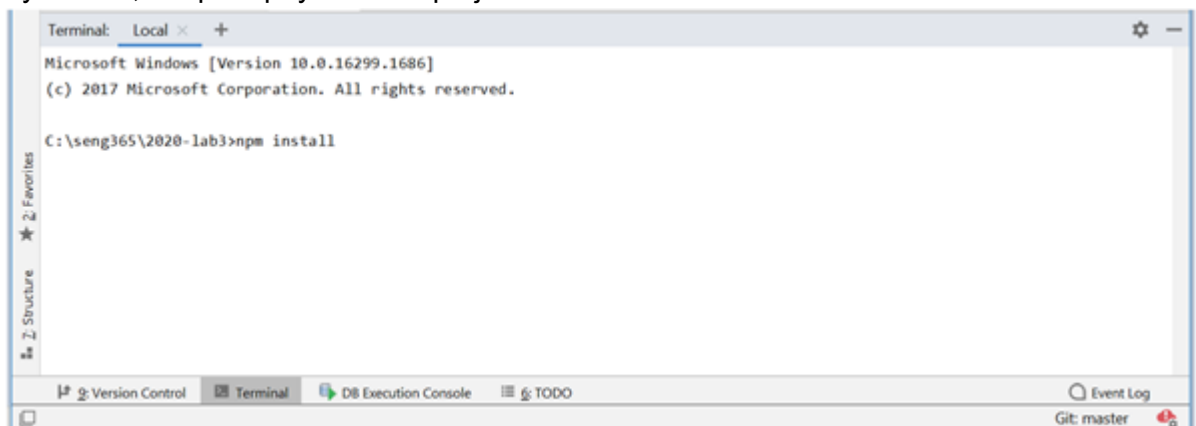
1. Create a new directory called 'lab_2' (or similar).
2. Inside this new directory, create a file called 'package.json' and insert the following code into it.

```
{
  "name": "lab_2",
  "version": "1.0.0",
  "description": "Combined persistence, architecture, and
typescript labs for 2022",
  "main": "dist/app.js",
  "scripts": {
    "prebuild": "tslint -c tslint.json -p tsconfig.json --fix",
    "build": "tsc",
    "prestart": "npm run build",
    "start": "node .",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Your Name",
  "license": "ISC",
  "dependencies": {
    "dotenv": "^10.0.0",
    "express": "^4.17.1",
    "mysql2": "^2.3.2",
    "winston": "^3.3.3"
  },
  "devDependencies": {
    "@types/express": "^4.17.13",
    "@types/node": "^16.11.6",
    "tslint": "^6.1.3",
    "typescript": "^4.4.4"
  }
}
```

Note:

- If you copy the above code to paste in your project, make sure there are no extra empty spaces between keywords and values as it would affect the parse of the values. The best – and **recommended** – way is to type from scratch rather than copy and paste.
 - We have introduced a lot in this file.
 - “dependencies” tell npm which packages are required to build and run your app, with
 - “devDependencies” being similar but with these packages only needed for development purposes.
 - “scripts” are the commands that will run under the hood when we run our app with the command “npm run <script_name>”, notice that we have introduced some typescript and linting in the (pre)building stages. The most important of these scripts is ‘npm run start’ that we will use to run our application.
 - For further reading refer to <https://docs.npmjs.com/files/package.json>.
 - The keyword ‘main’ refers to the file that will be included if someone does a ‘require’ of your package. In the dependencies you can set the necessary version for each required package using semantic versioning.
3. Now in your terminal, navigate to your project directory and run ‘npm install’. Node.js will create a new directory under your project folder named ‘node_modules’ and install the dependencies listed in the ‘package.json’ file in this new directory.

As an alternative to the system terminal, you can use the *Webstorm* terminal, which by default, will prompt you in the project’s root folder:



4. Now we need to set up our typescript config file so that our ‘.ts’ files accurately build to JavaScript with the build scripts we defined earlier. If you want to find out more about what is going on please refer to <https://www.typescriptlang.org/docs/handbook/tsconfig-json.html>. Create a new file “tsconfig.json” in your root directory and add the code below.

```
{
  "compilerOptions": {
    "module": "commonjs",
    "target": "es6",
    "rootDir": "./src/",
    "outDir": "dist",
    "esModuleInterop": true,
```

```
"noImplicitAny": true,  
"sourceMap": true  
}  
}
```

5. For this project we are also making use of tslint, a linter for TypeScript. A linter allows for catching issues with coding practices before we run the application which helps to catch errors and keep our code uniform. If you want to find out more about what is going on please refer to <https://palantir.github.io/tslint/usage/configuration/>. Create a new file "tslint.json" in your root directory and add the code below.

```
{  
  "defaultSeverity": "error",  
  "extends": [  
    "tslint:recommended"  
  ],  
  "jsRules": {},  
  "rules": {  
    "trailing-comma": [ false ]  
  },  
  "rulesDirectory": []  
}
```

6. Create a folder called 'src' this is where we will be storing all of our code. Within this file create a file 'app.ts'.
Note: this should be the same name as the 'main' variable in package.json, however when building with type script our 'app.ts' in the 'src' folder will be turned into a js file 'app.js' in the build folder 'dist' (thus app.ts becomes dist/app.js).
7. We are now ready to start coding our API.

3.2 Structuring large applications and MVC

Now that we have our modules installed, we can begin to develop our app. We want to break up the structure of our application to make it easier to understand for developers and ensure our application scales with minimal code refactoring required.

3.2.1 Model, View, Controller Architecture (MVC)

MVC is an architectural pattern used to break up the structure of an application into its conceptual counterparts. Anything relating to domain elements or interactions with databases comes under the 'model' section, anything that relates to presentation or the interface comes under the 'view' section and all the application's logic is stored under the 'controller' section.

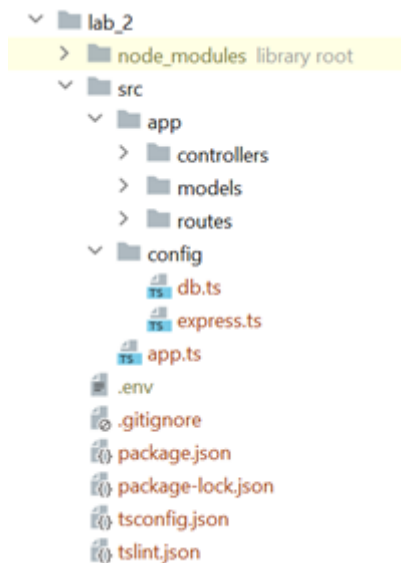
In our application, we store each part of the MVC structure in its own directory. As our API has no 'view' we use a 'routes' directory to provide the definition of the endpoints of our API.

1. Create two directories in your project called 'config' and 'app.'
2. Inside the 'app' directory, add three more directories called 'routes', 'models', and 'controllers.'
3. In your config directory, create a file called db.ts. This will hold all the details for connecting and configuring the database. Here is how the directory structure should

look

in

Webstorm:



4. Inside **db.ts**, we have two functions; 'connect' that connects to the database and 'getPool' which returns the connection pool. Copy/Type the below code into **db.ts**.

```
import mysql from 'mysql2/promise';
import dotenv from 'dotenv';
import Logger from '../logger';
dotenv.config();

const state = {
  // @ts-ignore
  pool: null
};

const connect = async (): Promise<void> => {
  state.pool = await mysql.createPool( {
    host: process.env.SENG365_MYSQL_HOST,
    user: process.env.SENG365_MYSQL_USER,
    password: process.env.SENG365_MYSQL_PASSWORD,
    database: process.env.SENG365_MYSQL_DATABASE,
  } );
  await state.pool.getConnection(); // Check connection
  Logger.info(`Successfully connected to database`)
  return
};

const getPool = () => {
  return state.pool;
};

export {connect, getPool}
```

5. Create another file in your **config** directory called '**express.ts**.' This will hold all the details for configuring express, as well as being the starting point for our express API.
6. Inside '**express.ts**' we have one function. This function simply initiates express, sets up body-parser and then returns the app.


```
import express from "express";
import bodyParser from "body-parser"

export default () => {
  const app = express();
  app.use( bodyParser.json() );
  return app;
};
```

7. Now in our **'app.ts'** file, import the two *config* files, initiate express using the express function in the *config* file that we have just written and connect to the database using the connect function in the imported database *config* file. If a connection to the database is successfully created, then start the server.

```
import { connect } from './config/db';
import express from './config/express';
import Logger from './config/logger'

const app = express();

// Connect to MySQL on start
async function main() {
  try {
    await connect();
    app.listen(process.env.SENG365_PORT, () => {
      Logger.info('Listening on port: ' +
process.env.SENG365_PORT)
    });
  } catch (err) {
    Logger.error('Unable to connect to MySQL.')
    process.exit(1);
  }
}

main().catch(err => Logger.error(err));
```

8. In your config directory create a file **'logger.ts'**. This will hold the setup for logging using the Winston node package <https://www.npmjs.com/package/winston>. Logging is common in industry where littering your code with `'console.log()'` calls is poor practice. **Note:** This logger prints to the console (alongside a file) which can be considered bad practice, but is done to make implementation easier, and to more verbosely show how our application is working. Copy the following code into your new **'logger.ts'** file.

```
import winston from 'winston'

const levels = {
  error: 0,
  warn: 1,
  info: 2,
  http: 3,
  debug: 4,
```

```
}

const colors = {
  error: 'red',
  warn: 'yellow',
  info: 'green',
  http: 'magenta',
  debug: 'white',
}
winston.addColors(colors)

const format = winston.format.combine(
  winston.format.timestamp({ format: 'YYYY-MM-DD HH:mm:ss:ms' }),
  winston.format.colorize({ all: true }),
  winston.format.printf(
    (info) => `${info.timestamp} ${info.level}:
${info.message}`,
  ),
)

const transports = [
  new winston.transports.Console(),
  new winston.transports.File({
    filename: 'logs/error.log',
    level: 'error',
  }),
  new winston.transports.File({ filename: 'logs/ts_labs.log' }),
]

const Logger = winston.createLogger({
  level: 'debug',
  levels,
  format,
  transports,
})

export default Logger
```

9. Finally create a **‘.env’** file in our root folder with the following environment variables, ensuring that the values are updated with your own credentials for database connection.

Note: Remember that if you are working from home you must set up ssh tunneling and set `SENG365_MYSQL_HOST=localhost`. Also remember to set the database to the correct name that you created it with earlier in the lab.

```
SENG365_MYSQL_HOST=db2.csse.canterbury.ac.nz
SENG365_MYSQL_USER={your user code}
SENG365_MYSQL_PASSWORD={your student id}
SENG365_MYSQL_DATABASE={your user code}_s365_lab2
SENG365_PORT=3000
```

10. Run your `app.ts` file using the command **‘npm run start’** in the terminal. The application can’t do anything at the moment but we can test that the database successfully connects.

3.2.2 Debugging Typescript

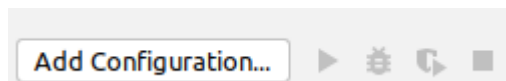
An important part of efficient coding is being able to quickly track down errors in your code. Generally as developers we all start with throwing print statements anywhere we can in our code, trying to work out the flow of the error, or printing out different variables to see what is happening to them. However as you have likely experienced, this technique is not very efficient, and it can make tracking down bugs a nightmare.

Thus instead of this we should use proper tools designed to help developers find errors. In this section we will look at how to use WebStorm's built-in debugger (there is a little more setup when using typescript compared to regular js).

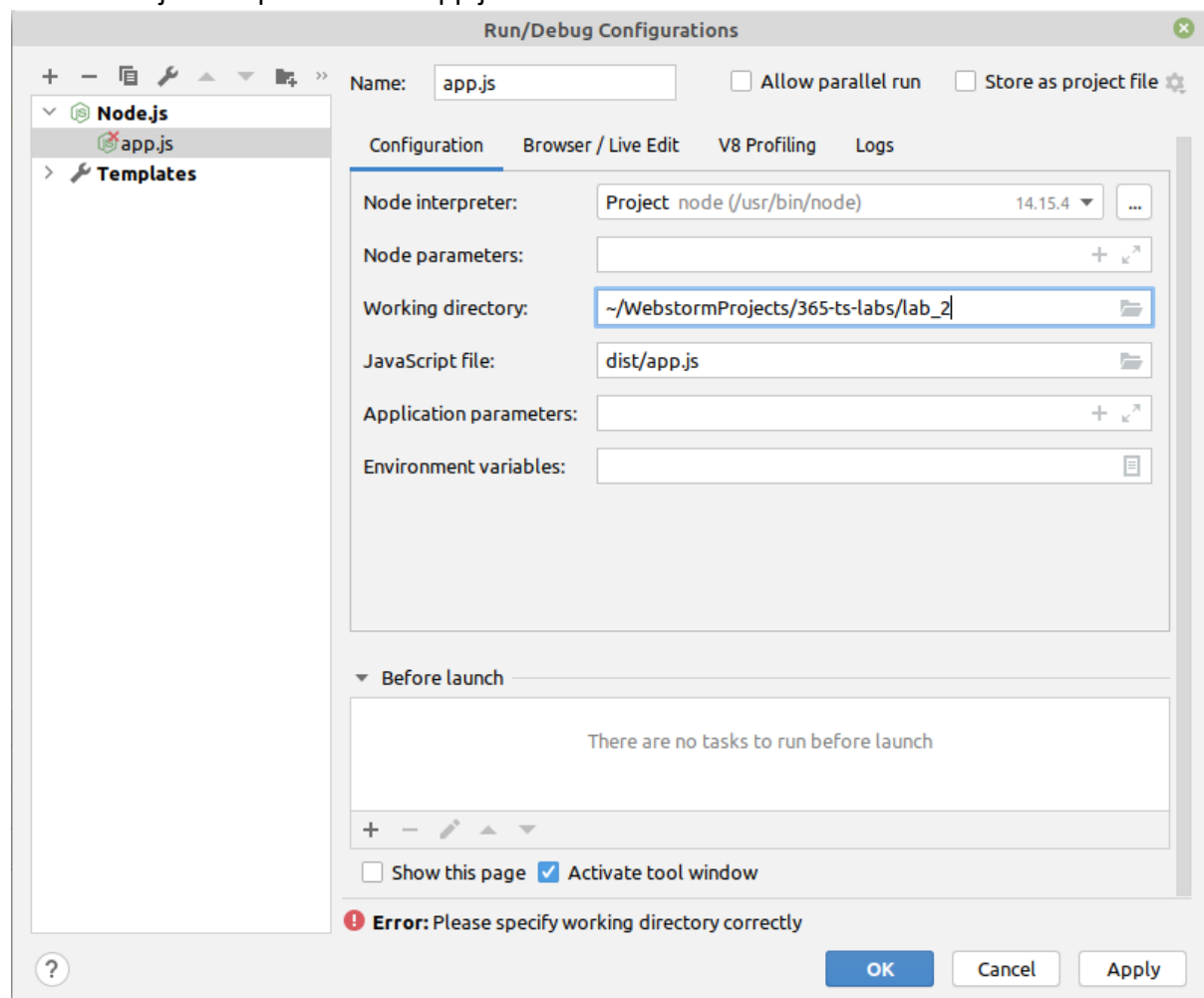
Note: For further reading refer to: https://www.jetbrains.com/help/idea/running-and-debugging-typescript.html#ws_ts_run_debug_server_side

To start we need to create a run configuration in WebStorm

1. Click on the "add Configuration..." button to open the Run/Debug configuration window in WebStorm.



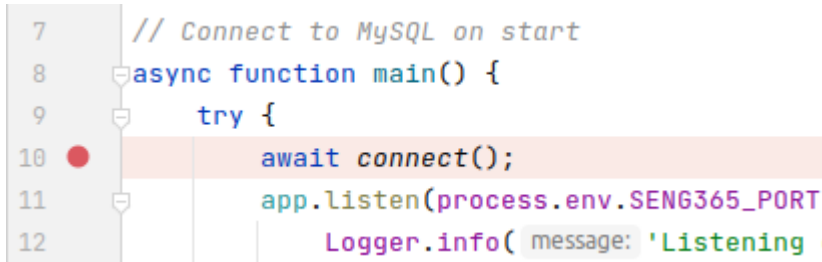
2. Use the '+' button to create a new configuration and select Node.js
3. Set working directory to your lab_2 folder
4. Set javascript file to dist/app.js



Note: In our tsconfig.json we included "sourceMap": true, this tells our compiler we want a sourceMap to be generated, which maps our ts code to the js code in our dist folder. This is required for debugging.

Once we have created our configuration we can start debugging

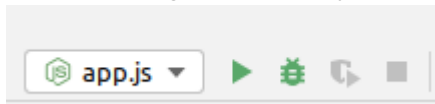
1. We can add breakpoints in our TS code by clicking to the right of the line number. For this example place a breakpoint in app.ts on the line "await connect()"



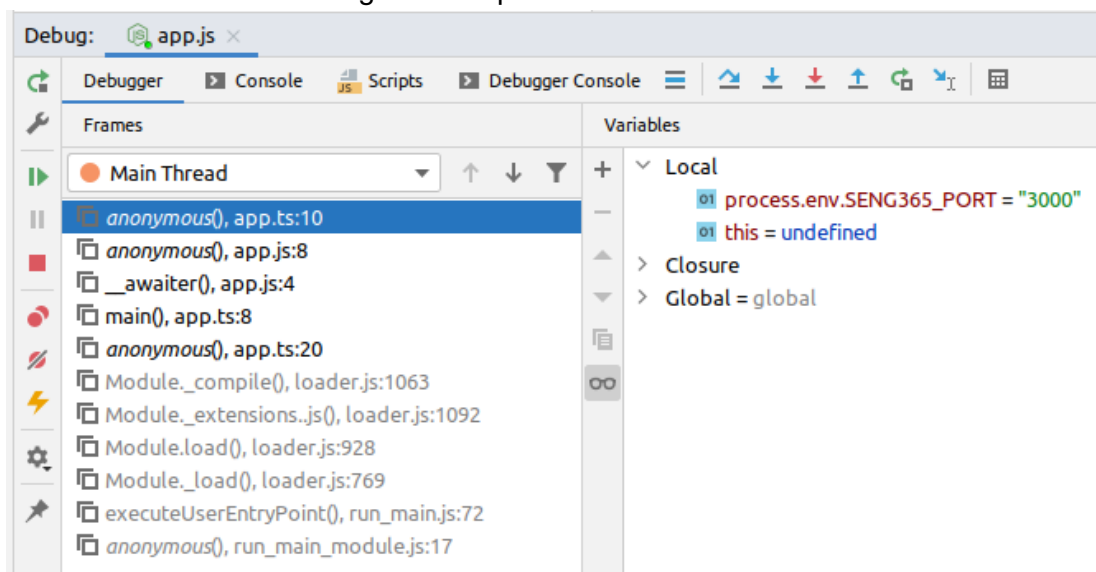
```

7 // Connect to MySQL on start
8 async function main() {
9   try {
10    await connect();
11    app.listen(process.env.SENG365_PORT
12    Logger.info( message: 'Listening
  
```

2. With our Run/Debug configuration created above selected, click on the small bug icon to the right of the play button.



3. Now we should see a debug toolbar open in the bottom of our window



Note: Importantly here we can see the variables tab, this is likely where you will do the most of your debugging

Important debugging actions:

- Resume program (green play/pause icon on left) (F9):
This allows us to continue from a breakpoint.
- Step over (right angled arrow with underline on top) (F8):
This allows us to move over the current line of execution. For example if it called a method to set a variable, we would not go into that method.
- Step into (down arrow with underline on top) (F7):
This is similar to step over, but in this case specifies that we should go into

whatever execution is next (this should be used sparingly as you can quite quickly end up in code that isn't yours)

Lets try some debugging,

1. Hopefully you are still stopped at the `await connect()` line, if not restart your debugger.
2. Click the 'step into' button (or hotkey) and you should be taken to the `db.ts` file, with the `connect` function highlighted
3. If we step through this function we will eventually be taken back to `app.ts` and the rest of the code

This was a very simple introduction to how we can step through code as we debug, in general we would often be paying more attention to the variables in our local scope (as these are generally the ones we have defined) to understand what is causing an error.

3.3 Adding the Users functionality to the API

Now that our boilerplate code for the application is working, we can add some functionality to the API. Like in last week's lab, this exercise will run through the Users functionality and then you will create the rest of the API on your own. This is the API specification for the Users functionality from lab 2 (we have changed the URL's slightly).

Note: This API is properly defined in an API specification file discussed in Section 5, feel free to look ahead now if you want a deeper understanding of the functionality we are about to implement.

URI	Method	Action
/api/users	GET	List all users
/api/users/:id	GET	List a single user
/api/users	POST	Add a new user
/api/users/:id	PUT	Edit an existing user
/api/users/:id	DELETE	Delete a user

3.3.1 Boilerplate code and structure

1. In our '**express.ts**' config file, before we return the `app` variable, add a line that imports a file called '`user.server.routes.js`' from the '`/app/routes`' directory. This file will take in the '`app`' variable as shown in the code below.

```
import express from "express";
import bodyParser from "body-parser"
```

```
export default () => {
  const app = express();
  app.use( bodyParser.json() );
  require('../app/routes/user.server.routes.js')(app);
  return app;
};
```

2. Create the **'users.server.routes.ts'** file in the routes directory. This file will import a 'users' controller and then define each of the relevant routes as outlined in the specification. Each route calls a function in our controller. Another controller function is used for retrieving a user from the ID that is specified in the URL.

```
import {Express} from "express";
import * as users from '../controllers/user.server.controller';

module.exports = ( app: Express ) => {

  app.route( '/api/users' )
    .get( users.list )
    .post( users.create );

  app.route( '/api/users/:id' )
    .get( users.read )
    .put( users.update )
    .delete( users.remove );

};
```

3. Next we need to create the users controller, create a file in the controllers directory called **'user.server.controller.ts'** This file will import the 'users' model and contain the five functions that are called by the routes file. For now, add them as functions that just return null.

```
import * as users from '../models/user.server.model';
import Logger from "../../config/logger";
import {Request, Response} from "express";

const list = async (req:any, res:any) : Promise<any> => {
  return null;
};

const create = async (req:any, res:any) : Promise<any> => {
  return null;
};

const read = async (req:any, res:any) : Promise<any> => {
  return null;
};

const update = async (req:any, res:any) : Promise<any> => {
  return null;
};

const remove = async (req:any, res:any) : Promise<any> => {
  return null;
};

export { list, create, read, update, remove }
```

4. Finally, we need to add our model code. In the models directory, create a file called **'user.server.model.ts'**. This file should import the database config file and contain the following functions.

```
import { getPool } from "../../config/db";
import Logger from "../../config/logger";
import { ResultSetHeader } from "mysql2";

const getAll = async () : Promise<any> => {
    return null;
};

const getOne = async (id: number) : Promise<any> => {
    return null;
};

const insert = async (username: string) : Promise<any> => {
    return null;
};

const alter = async (id: number, username: string) : Promise<any>
=> {
    return null;
};

const remove = async (id: number) : Promise<any> => {
    return null;
};

export { getAll, getOne, insert, alter, remove }
```

3.3.2 Listing all users

Now we have the boilerplate code set up, we just need to fill in the functions that we have left blank. First we will look at listing all users.

1. Within the **user.server.model.ts** file, update the **getAll()** function to get a database connection and run an asynchronous query to select all from the users table. **Note:** We specify a return type of **Promise<User[]>**, importantly this is a promise that resolves to a list of User objects (which are defined below).

```
const getAll = async () : Promise<User[]> => {
    Logger.info(`Getting all users from the database`);
    const conn = await getPool().getConnection();
    const query = 'select * from lab2_users';
    const [ rows ] = await conn.query( query );
    conn.release();
    return rows;
};
```

2. Now in the **user.server.controller.ts** file, update the list function. The **list()** function calls the models **'getAll()'** async function. This function simply waits for and returns the result from the model function and handles a basic error flow.

```
const list = async (req: Request, res: Response) : Promise<void> =>
{
```

```

    Logger.http(`GET all users`)
    try {
        const result = await users.getAll();
        res.status( 200 ).send( result );
    } catch( err ) {
        res.status( 500 )
            .send( `ERROR getting users ${ err }` );
    }
};

```

3. In our `getAll` function we introduced a new type “User”, this is a custom type meaning that we have to provide a definition for it. Create a new ts file at the root of your app directory called **‘user_types.d.ts’** (note the name doesn’t matter but the suffix **‘.d.ts’** is required. Finally, add the following type definition:

```

type User = {
    /**
     * User id as defined by the database
     */
    user_id: number,
    /**
     * Users username as entered when created
     */
    username: string
}

```

4. Now run your app for testing. Sending a GET request to **/api/users** should result in all the users being returned. You can do this quickly by simply typing `localhost:300/api/users` into your browser.
Note: Make sure you have some users in your database for testing before running this. These can be easily added through the phpMyAdmin web portal, under the insert tab.

3.3.3 Getting a single user

1. In the **user.server.model** file, edit the **‘getOne()’** function so that it takes in the `userId` as a parameter. Like the previous functions, run the query and return the results.

```

const getOne = async (id: number) : Promise<User[]> => {
    Logger.info(`Getting user ${id} from the database`);
    const conn = await getPool().getConnection();
    const query = 'select * from lab2_users where user_id = ?';
    const [ rows ] = await conn.query( query, [ id ] );
    conn.release();
    return rows;
};

```

2. In the controller, edit the **read** function to retrieve the `id` from the url and call the **‘getOne’** function and return the result to the client. Notably here, we return a 404 status code if we do not find a match from the database.

```

const read = async (req: Request, res: Response) : Promise<void> =>
{
    Logger.http(`GET single user id: ${req.params.id}`)
    const id = req.params.id;
    try {

```



```
const result = await users.getOne( parseInt(id, 10) );
if( result.length === 0 ){
    res.status( 404 ).send('User not found');
} else {
    res.status( 200 ).send( result[0] );
}
} catch( err ) {
    res.status( 500 ).send( `ERROR reading user ${id}: ${ err }`
);
}
};
```

3. Now run your app for testing. Sending a GET request to **/api/users/1** should result in one user being returned (make sure that the returned user matches what is expected). You can do this quickly by simply typing localhost:300/api/users/1 into your browser. **Note:** Make sure you have some users in your table for testing before running this, you may also find there is no user with user_id=1, in that case pick a user_id from the get all users response.

3.3.4 Creating new users

1. In the model file, edit the insert function so that it takes in the username as a parameter. Query the database so that it inserts a new record into the users table.

```
const insert = async (username: string) : Promise<ResultSetHeader>
=> {
    Logger.info(`Adding user ${username} to the database`);
    const conn = await getPool().getConnection();
    const query = 'insert into lab2_users (username) values ( ? )';
    const [ result ] = await conn.query( query, [ username ] );
    conn.release();
    return result;
};
```

2. Now create the controller function. This function gets the username from the POST data and then input it to the models insert function. This function simply returns the result to the user. If there is no username field present in the request body we will return a HTTP 400 status code (Bad Request).

```
const create = async (req: Request, res: Response) : Promise<void>
=> {
    Logger.http(`POST create a user with username:
    ${req.body.username}`)
    if (! req.body.hasOwnProperty("username")){
        res.status(400).send("Please provide username field");
        return
    }
    const username = req.body.username;
    try {
        const result = await users.insert( username );
        res.status( 201 ).send({"user_id": result.insertId} );
    } catch( err ) {
        res.status( 500 ).send( `ERROR creating user ${username}: ${
err }` );
    }
};
```

```
}  
};
```

3. Now run your app.js file for testing. This time we are working with a POST request and can not easily do this through our browser like we can for GET. Instead you may want to check out Section 4 where we discuss proper API testing with Postman.

3.3.5 Altering a user

1. Create the model function, using the previous tasks as a template.
2. Create the controller function, using the previous tasks as a template.
Note: Think about what should happen if we don't find the user to alter, or what should happen if the request doesn't provide a username to change to.
3. Test using Postman (again looking ahead to section 4).

3.3.6 Deleting a user

1. Create the model function, using the previous tasks as a template.
2. Create the controller function, using the previous tasks as a template.
Note: Think about what should happen if we don't find the user to delete.
3. Test using Postman (again looking ahead to section 4).

4 Testing APIs with Postman

4.1 What is Postman

Postman (<https://www.postman.com/>) is an API testing suite that gives developers the ability to easily and automatically test their API's. Postman is pre-installed on the lab machines, or can be downloaded for free (with a registration) if you are working on your own machine.

A suite of postman tests has been provided alongside this lab **on Learn** so that you can have a play around with the functionality and test your API. This is important as going into the assignment we will provide a much more in depth test suite you can use to verify your API is working as expected (you can even add your own tests to this if you want more coverage).

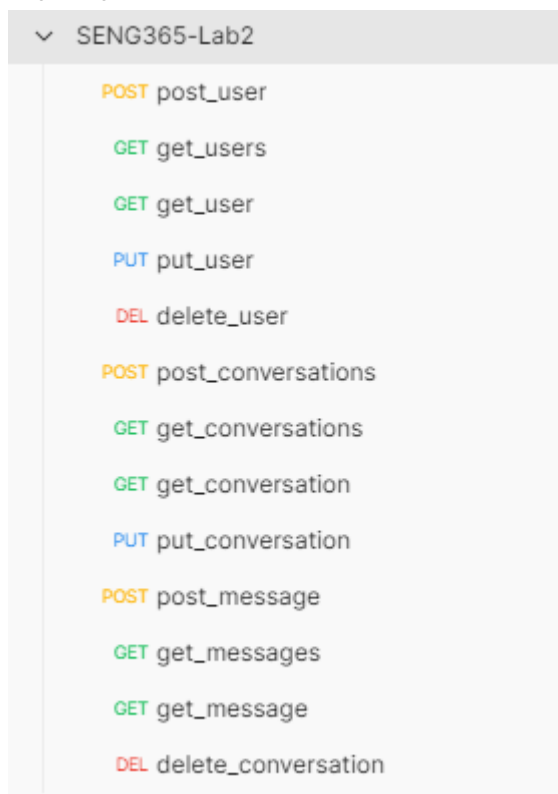
4.2 Getting started with requests

4.2.1 Importing Postman requests collection

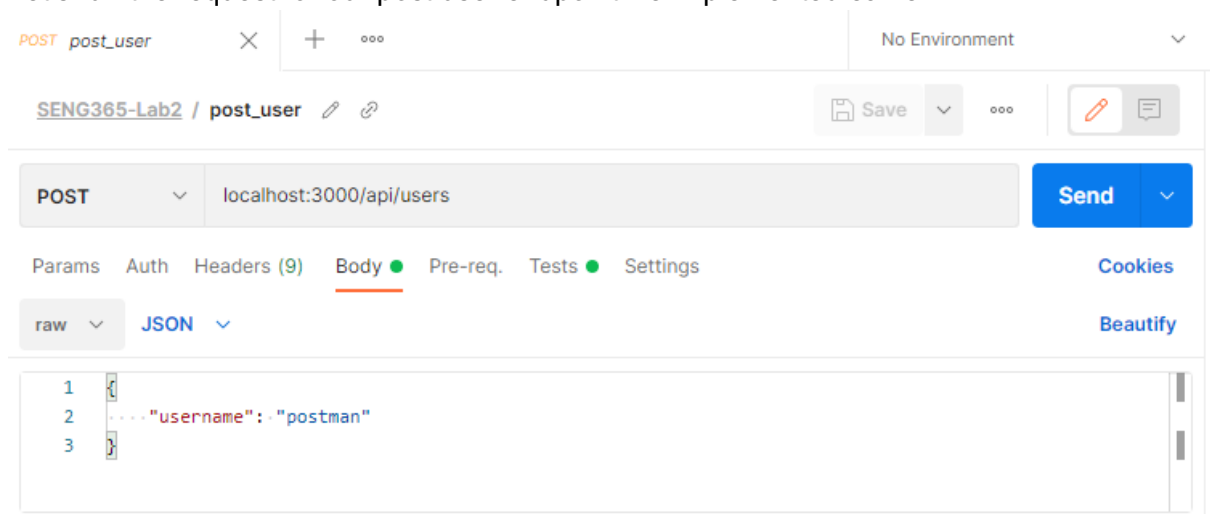
To start with postman we will load in the collection discussed above. This will allow us to have a look at some of the different request types, and to test our application so far.

1. To import a collection, navigate to the 'My Workspace' > 'Collections' tab on the left side of the window
2. Here there should be a button 'Import', we can click this to open a window which will allow us to drag and drop, or select a file. Use either option to load in the postman collection file.

3. Now that we have imported the collection we should be able to see the requests within it.



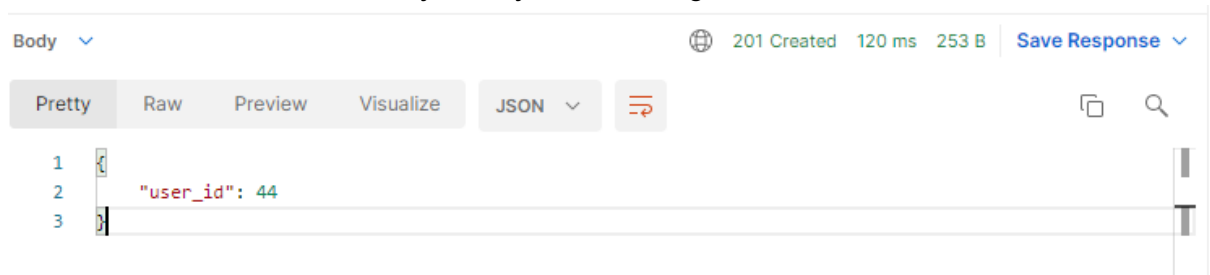
4. Let's run the request for our post user endpoint we implemented earlier.



Importantly on this screen we have:

- The HTTP method, in this case a POST request
- The url 'localhost:3000/api/users'
- Our request body (of raw JSON type) that contains our new user object

- Once we click send, if all of our code is correct we should get a HTTP response with status code **201 Created** and a json object containing our new users' id.



- Finally, feel free to have a look through the rest of the collection, and try out the other requests for functionality such as updating and deleting a user.

4.2.2 Creating a new request

Postman allows for the creation of simple requests using all of the HTTP methods, where you can specify headers, a request body, url parameters, and more. We will not discuss all of these in this lab so feel free to read <https://learning.postman.com/docs/sending-requests/requests/> for more detail.

In the collection given we only check for 'blue sky' or expected flows, where everything functions as expected. However, any good testing regime requires the testing of alternate and erroneous flows as well. The HTTP protocol provides many different status codes, for each end point a good start for more verbose testing is to create a test for each different status code an endpoint could return. Of course in larger applications we may need to test several flows that lead to the same status code being returned.

The API spec we are using is not overly verbose, however we can see some endpoints have several different return types excluding the expected flow. A common one is 404 'Not Found', this is a status you have likely seen when browsing the web and finding a page that no longer exists. In the case of an API we can use this to signal that the data requested was not found. For example the GET user/:id endpoint should return a 404 if there is no user with the id we supply.

To get a 404 we need to provide an id that does not exist in the database, an easy way to do this is specify a very large number such as 999,999,999. Whilst theoretically we could have a user with this id, within the scope of this lab we do not expect to populate 1 billion users.

- Create a new request within the collection
- Make sure the request type is GET
- Add the url "localhost:3000/api/users/999999999"
- Run the request and check that you get a 404 response

Another example is the POST user endpoint, which should return a 400 if the data we are providing in the body is incorrect.

To get a 400 we have to provide an incorrect body, notably this means we must not have the username field.

- Create a new request within the collection

2. Make sure the request type is POST
3. Add the url "localhost:3000/api/users"
4. Add a body of raw JSON type with the following value

```
{
  "not_a_real_field": "not_a_real_value"
}
```

5. Run the request and check that you get a 400 response

You may have noticed that many of our endpoints have code to return status code 500 'Internal Server Error' which we have no tests for. This is often a blanket status code for errors that are not expected, such as errors occurring within the database. Due to this it is hard to create proper tests for them, as any clear erroneous flow should return a specific and related status code (such as the 4XX codes we discussed above).

4.3 Getting started with Postman tests

Now that we have learnt how to create requests, we can create proper tests where we check what is being returned from our server automatically instead of doing so manually.

4.3.1 Running all tests in a collection

In the collection provided each request tests the endpoint as well, while these tests run when we send individual requests, it is much more useful running all the requests (and by extension, the tests) together. For some tests this is required as they are dependent on earlier requests (such as deleting a user we previously created).

To run a collection right click on the collection, and select the options button (horizontal triple dots on right hand side) and select the 'Run' or 'Run Collection' option. This should open a new page where all the requests in the collection are listed and we can (de)select them and change the ordering. **Deselect any tests that are not for the users endpoint**, otherwise the collection will break when reaching conversations and messages endpoints as that we have not yet implemented. However, do not change the order of tests as there are some tests that rely on others to run before them. This page should have a blue 'Run <collection_name>' button, click on this and check that the tests run (and pass) as expected.

Note: There are some small differences in how to reach the Collection Runner screen depending on the Postman version. If for some reason your version does not match one of the ways we discussed above, ask a tutor for help or look online.

The results screen (after the tests have run) should look like the image below.

SENG365-Lab2 No Environment, just now

[View Summary](#) [Run Again](#) [New](#) [Export Results](#)

All Tests Passed (32) Failed (0)

Iteration 1

POST	post_user	localhost:3000/api/users	/ post_user	201 Created	24 ms	253 B
Pass	Object test					
Pass	Object test					
Pass	Status test					
Pass	Body test					
GET	get_users	localhost:3000/api/users	/ get_users	200 OK	18 ms	798 B
Pass	Status test					
Pass	Body test					
Pass	Array test					
GET	get_user	localhost:3000/api/users/{{user_id}}	/ get_user	200 OK	18 ms	270 B
Pass	Status test					
Pass	Body test					
Pass	Object test					

Note: Unless you have completed the rest of the lab (conversations and messages) you should have 11 passing tests (only the user related tests).

4.3.2 Adding tests to our requests

We are going to extend the requests we made in the section above so that they can automatically test the API and inform us if it is acting as expected. The tests we create in this section will not be as advanced as those already in the collection (though we will discuss some more advanced concepts), thus it is recommended that you look at these tests and online resources such as <https://learning.postman.com/docs/writing-scripts/test-scripts/> to gain a deeper understanding of how these tests can be used.

1. Within the GET user 404 request you created, navigate to the 'test' tab
2. Add the following code to the text area

```
pm.test("Status test", function () {  
    pm.response.to.have.status(404); // tests status is 404  
})
```

- Run your request again, and now you should see the option 'Test Results' in the bottom pane with the 'Status test' passing.

Body Cookies Headers (7) Test Results (1/1)

All Passed Skipped Failed

PASS Status test

- A similar test can be added to our POST user 400 request, by simply changing the expected response status.

```
pm.test("Status test", function () {  
    pm.response.to.have.status(400); // tests status is 400  
})
```

The tests we created above are very basic, and only check the status code of the response, however Postman allows for much more verbose testing functionality. To see this in action refer to the POST user request (provided to you in the collection). This request has 3 separate tests, one like those above checks for a specific status (in this case 201), another checks that a json body is returned. The final test checking the body contains a non-empty object with a user_id field of type number and then sets the global variable 'test_id' to this value. If you look ahead to the GET, PATCH, and DELETE requests for users/:id you should see this value being referenced in the url (surrounded by double curly braces, e.g. 'localhost:3000/api/users/{{test_id}}'), this allows for these tests to reference the user we just created.

Note: This is a perfect example of why post requests should return at least the id of any object created. In a real world application once we create something, if we don't have a reference to it we have no way to easily retrieve it from the API. (Some approaches like to return the whole object as it has been created, so that a client can confirm everything is correct themselves, and have the required data to display it without needing to send a GET request).

5 Implementing the rest of the API - Recommended

Like the last lab, this exercise is optional but recommended due to the large number of concepts covered. Once you feel comfortable with the concepts you are ready to start working on the assignment.

5.1 Using an API spec

Commonly when creating an API a concrete specification is created beforehand so developers know all the endpoints that need to be implemented and what they should do. These specifications are also important after the API has been made, as the consumer of the API needs to understand how they can use each individual endpoint to achieve their goal.

We have created an API spec for this lab **on Learn** that details the functionality for the remaining endpoints relating to conversations and messages.

1. The first step is to copy the contents of this file into the left side of the Swagger online editor <https://editor.swagger.io/> (other editors exist such as plugins for WebStorm, however this lab will assume the use of the Swagger editor).
2. Now you should see the right side populated with content title “SENG365 Lab 2”
3. Here we can see each of the different API sections (known as tags): ‘users’, ‘conversations’, and ‘messages’.
4. Under each of these tags are the related HTTP requests (shown below)

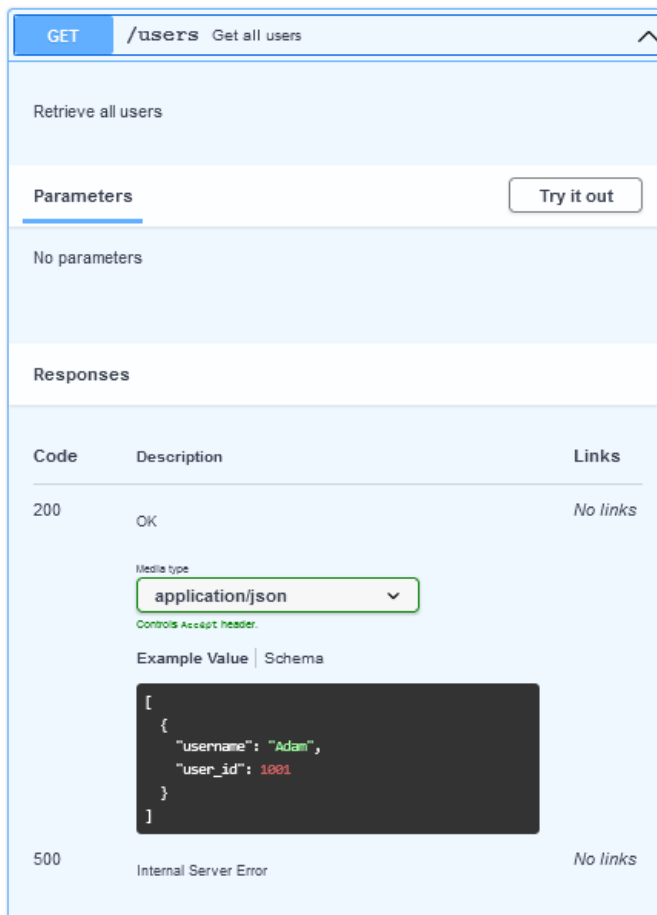
SENG365 Lab 2 1.0.0 OAS3

This is a simple example API spec for SENG365 Lab 2. This is included to give an example API spec before the first assignment where it is much more verbose

[Contact the developer](#)

users Access to users and their usernames ^		
GET	/users	Get all users
POST	/users	Add a new user
GET	/users/{id}	Find user by ID
PUT	/users/{id}	Update an existing user
DELETE	/users/{id}	Deletes a user

5. Within each of these requests we can see more detailed information about the expected parameters, headers, request body, and more.



Here we see the 'Get all users' endpoint has no parameters and either returns a 200 'OK' with a list of User objects, or 500 'Internal Server Error' if something goes wrong.

Note: You may want to have a look at the possible HTTP status codes to understand what they represent https://en.wikipedia.org/wiki/List_of_HTTP_status_codes.

6. Explore the rest of the API spec before moving on to give more understanding of what is required to complete the rest of the Chat App API. This should also give you more experience understanding these specifications before moving onto the assignment.
7. For further reading about OpenAPI 3.0.0, the specification used for this lab and the assignment, refer to <https://spec.openapis.org/oas/v3.0.0>.

5.2 Completing the API

You now have all the skills required to implement the rest of the API to the specification we introduced in Section 5.1. Doing so will provide you with great experience for starting the first assignment. Simple Postman tests have been included so you can check that you are completing the endpoints as expected.

That concludes the server-side labs. You should now have the knowledge you need to complete the first assignment. In the next lab (next term), we begin to look at client applications.