# SENG365 Lab 2
# Persisting Data, API Testing with Bruno, and Structuring Applications in Node with TypeScript

SENG365

Ben Adams Morgan English

19th February 2025

## Purpose of this Lab

Last week we had an introductory look at using JavaScript, Node, and Express to create a small micro-blogging API. This week's lab expands on these concepts in the domain of a chat app bringing in many new features and technologies. Due to the scope of this lab it is expected to take 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 create the chat app API, making use of proper application structure, TypeScript, validation, and a few other Node best practices. This will be followed by a look at how to test the API using Bruno. Finishing with a small introduction to API specifications, and leaving you to complete the rest of the application to the introduced specification.

## 1 Persisting Data with MySQL

### 1.1 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. Figure 1 shows the Entity Relationship Diagram[1] (ERD) for the chat application

### 1.2 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 `https://dbadmin.csse.canterbury.ac.nz/`. **Note:** to access phpMyAdmin you will go through two authentication steps; The first will be

[1] Students should be familiar with ERDs from previous courses, for a refresher, see `https://en.wikipedia.org/wiki/Entity-relationship_model`

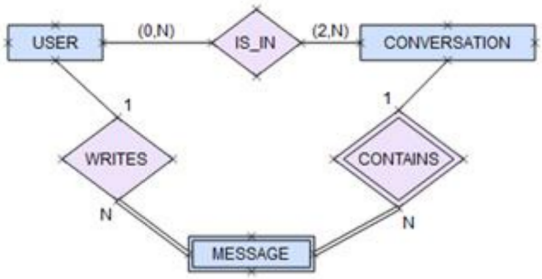



Figure 1: Entity relationship diagram for the chat app.

before the page loads and requires your UC username and password, the next will be when trying to log in through the admin portal, here is where you will input your user code and id as mentioned above.

Once you have logged into phpMyAdmin, you should be able to the databases starting with your user code for example *abc123_main*.

From here it is suggested you create a new database specifically for this lab, e.g. *abc123_s365_lab2* as seen in Figure 2.


Figure 2: Example creation of database for this lab.

## 1.3 Creating Tables

Now that we have access to our database, we will create a database using the *lab2_init.sql* script [2]. This database will have the tables and fields shown in Figure 3. Look through the SQL script for more detailed information on the foreign key constraints.

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
```

**Note** that you will need to make sure you update the user code to your own, and run the terminal from the same directory as the lab2_init.sql file.

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 though phpMyAdmin (make sure you've selected the database for this lab), paste the given script content and run it from there. See an example of what this should[3] look like in Figure 4.

Finally make sure you confirm the operation through the pop-up, making sure that everything looks correct first.

[2] You can find this alongside the lab handouts on Learn
[3] Software applications change frequently so it may have evolved from when this screenshot was taken

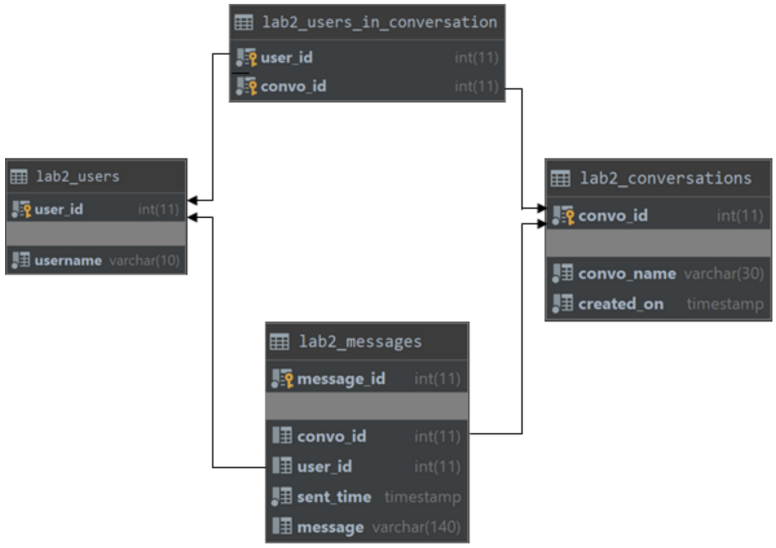



Figure 3: Database schema for chat app, derived from the ER diagram above.

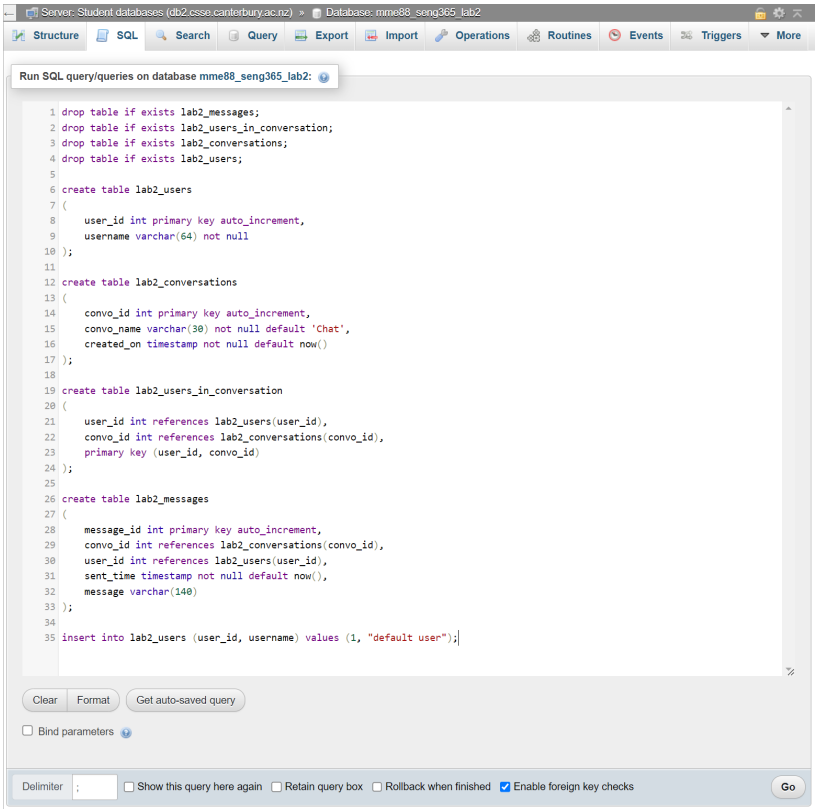



Figure 4: Example DBAdmin executing SQL script.

## 1.4 Reading Database Parameters from Environment Variables

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

To do this we will make use of the dotenv[4] module later in the lab when we get into actually coding the application. This works by including a *.env* file at the root of our project with *key=value* pairs.

**Notes:**

[4] an npm package available at `https://www.npmjs.com/package/dotenv`

- If you are working from home please follow the steps detailed in the document *Connecting to the University Database from Home* included alongside this handout on Learn. This will require you to modify the *.env* file and run an *ssh* command.
- Your *.env* file should never be version controlled, especially if it contains sensitive information like passwords. If you are using git to keep track of your completed labs it may pay to include a *.gitignore*[5] file at the root of your project.

# 2 Setting up our Node application with TypeScript

## 2.1 Installing Node Packages with a *package.json* File

All Node.js projects contain a file (usually[6] at the project root) named *package.json* that holds various metadata relevant to the project. The file is used to give information to *npm* that allows it to identify the project as well as handle the project's third-party dependencies. It can also contain other metadata such as a project description, the version of a project in a particular distribution, license information, and configuration data – all of which can be vital to both *npm* and the end-users of a package.

Now we will start creating a chat application similar to what we worked on last week, covering the new concepts introduced in this lab. This is quite a technical step up compared to last lab, so feel free to work with or discuss the new concepts with others or the tutors.

1. Create a new directory called *lab_2* (or something you prefer, though you may need to change this in some commands as well)
2. Inside this new directory, create a file called *package.json* and insert the code from Listing 1 into it.

```
{
    "name": "lab_2_combined",
    "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": "Morgan English",
    "license": "ISC",
    "dependencies": {
        "dotenv": "^16.0.3",
        "express": "^4.18.2",
        "mysql2": "^2.3.3",
        "winston": "^3.8.2",
        "ajv": "^8.11.0"
    },
    "devDependencies": {
        "@types/express": "^4.17.15",
        "@types/node": "^18.11.18",
        "tslint": "^6.1.3",
        "typescript": "^4.9.4"
    }
}
```

Listing 1: A basic *package.json* file for an express application with TypeScript

- 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 can affect the parsing. The best – and recommended – way is to type it out

[5] See an example at `https://github.com/github/gitignore/blob/main/Node.gitignore` though note this is a verbose example
[6] Sometimes there are several throughout a projects file structure

from scratch rather than copy-pasting.

- 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 specifically for packages only required for developing your application. In these dependencies you can set the necessary version for each required package using semantic versioning.
  - *scripts* are the commands that will run under the hood when you run your app with the command `npm run <script_name>`. Notice that we have introduced some TypeScript and linting in the (pre)building stages, a required step when working with TS. The most important of these scripts is ***npm run start*** which is used to run your application.
  - For further reading refer to `https://docs.npmjs.com/files/package.json`
- The keyword *main* refers to the primary entry point of your application, the exports from this file will be included if someone imports your package.

3. Now in your terminal[7], navigate to your project directory and run `npm install`. Node.js will create a new directory in your project folder named **node_modules** and will install the dependencies listed in the *package.json* file here. You may notice there are way more than you expected, these are mainly inherited dependencies from those we explicitly define.
4. Now we need to setup our TypeScript config[8] so that our *.ts* files accurately build to JavaScript with the build scripts we defined earlier. Create a new file *tsconfig.json* in your root directory and add the code in Listing 2.

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

Listing 2: A basic *tsconfig.json* file.

5. For this project we also make use of tslint[9], a linter for TypeScript. Linters allow for catching issues with coding practices before we run the application which helps to catch errors and keep our code uniform. Create a new file *tslint.json* in your root directory and add the code in Listing 3.

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

Listing 3: A basic *tslint.json* file.

6. Create a folder *src*, this is where we will be storing all of our code. Within this folder create a file *app.ts*. **Note:** This should be the same as the 'main' variable in our *package.json*, however when building with TypeScript our *app.ts* will be transformed into a JavaScript file in the build folder *dist*. Thus *src/app.ts* becomes *dist/app.js*.

[7] You can also simply use Webstorm's built in terminal, which defaults to the project root folder.
[8] If you want to find out more about this config see `https://www.typescriptlang.org/docs/handbook/tsconfig-json.html`.
[9] If you want to find out more about tslint and the configuration options see `https://palantir.github.io/tslint/usage/configuration/`.

We are now ready to start coding our API!

## 2.2 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 well with minimal refactoring required.

### 2.2.1 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 the presentation or the interface comes under the **view** section, and finally all the application 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 (within the src folder we created earlier) called *config* and *app*.
2. Inside the *app* directory, add three more directories *routes*, *models*, and *controllers*.
3. In the config directory, create a file called *db.ts* and copy the code from Listing 4. Notably this file exports two functions, one to create a connection with the database and another to get access to a pool of connections that we can run queries on.

```
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,
        ssl: {
            rejectUnauthorized: false
        }
    } );
    await state.pool.getConnection(); // Check connection
    Logger.info(`Successfully connected to database`)
    return
};

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

export {connect, getPool}
```

Listing 4: A file that defines how we connect to the MySQL database.

4. Create another file within your *config* directory called *express.ts* and copy the code from Listing 5. This will hold the details for configuring express, as well as being the starting point for our express API.

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

```
3
4  export default () => {
5      const app = express();
6      app.use((req, res, next) => {
7          res.header("Access-Control-Allow-Origin", "*");
8          res.header("Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type, Accept");
9          res.header("Access-Control-Allow-Methods", "GET, POST, PUT, DELETE");
10         next();
11     });
12     app.use( bodyParser.json() );
13     return app;
14 };
```

Listing 5: A file that defines our express configuration.

5. Create another file within your *config* directory called *logger.ts* and copy the code from Listing 6. This will hold the setup for logging using the Winston node package[10]. 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 easer, and to more verbosely show how the application is working.

```
1  import winston from 'winston'
2
3  const levels = {
4      error: 0,
5      warn: 1,
6      info: 2,
7      http: 3,
8      debug: 4,
9  }
10
11 const colors = {
12     error: 'red',
13     warn: 'yellow',
14     info: 'green',
15     http: 'magenta',
16     debug: 'white',
17 }
18 winston.addColors(colors)
19
20 const format = winston.format.combine(
21     winston.format.timestamp({ format: 'YYYY-MM-DD HH:mm:ss:ms' }),
22     winston.format.colorize({ all: true }),
23     winston.format.printf(
24         (info) => `${info.timestamp} ${info.level}: ${info.message}`,
25     ),
26 )
27
28 const transports = [
29     new winston.transports.Console(),
30     new winston.transports.File({
31         filename: 'logs/error.log',
32         level: 'error',
33     }),
34     new winston.transports.File({ filename: 'logs/ts_labs.log' }),
35 ]
36
37 const Logger = winston.createLogger({
38     level: 'debug',
39     levels,
40     format,
41     transports,
42 })
43
```

[10] See https://www.npmjs.com/package/winston.

```
export default Logger
```

Listing 6: A file that defines our logger configuration.

6. Now in our *app.ts* file copy Listing 7, which imports the two config files, initiates Express using the `express` function in the config file we have just written and connects to the database using the connect function imported from our database config file. If a connection to the database is successfully created, then it starts the server on the `SENG365_PORT` we define in our *.env* file or 3000 by default.

```
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 || 3000, () => {
            Logger.info('Listening on port: ' + process.env.SENG365_PORT||3000)
        });
    } catch (err) {
        Logger.error('Unable to connect to MySQL.')
        process.exit(1);
    }
}

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

Listing 7: Our main file that runs the express server

7. Finally we need to create a *.env* file in our root folder with the variables shown in Listing 8, ensuring that the values are updated with your own credentials[11] 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 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
```

Listing 8: example *.env* file.

8. Run your app.ts file using the command `npm run start` in the terminal. The application doesn't actually do anything yet, but we can test that it successfully connects to the database.

### 2.2.2 Debugging TypeScript

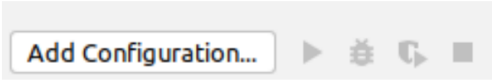
An important part of efficient coding is being able to quickly track down errors in your code. Generally as developers we all start with throwing print statements anywhere we can in our code, trying to work out the flow of an error, or printing out different variables to see what is happening to them. However as you have likely experienced, this technique is not very efficient and 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 take a quick look at how to use Webstorm's built-in debugger[12] (there is a little bit more setup when using TypeScript compared to plain old JavaScript).

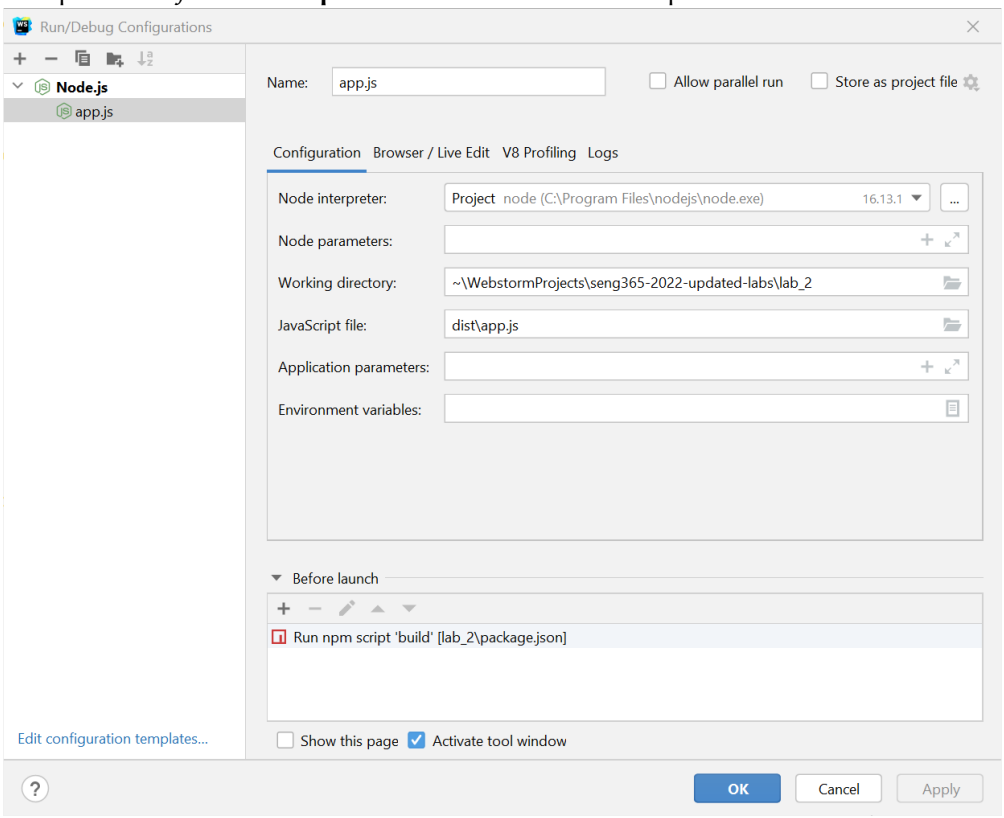
To start we need to create a run configuration in Webstorm.

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

[11] Make sure to remove the curly braces around them
[12] For more information see `https://www.jetbrains.com/help/idea/running-and-debugging-typescript.html`.



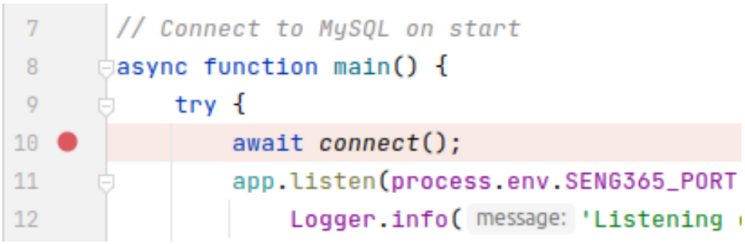

2. Use the "+" button to create a new configuration and select *Node.js*.
3. Set the working directory to your *lab_2* folder.
4. Set the "JavaScript file" to *dist/app.js*.
5. Set up a new *Before launch* **npm** task that runs the build script.



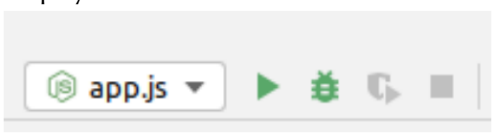

**Note:** In our *tsconfig.json* file we included the line `"sourceMap":true`, this tells the compiler we want a sourceMap to be generated which maps our TS code to the JS code in our *dist* folder. This is required for debugging.

Once we have created our configuration we can start debugging.

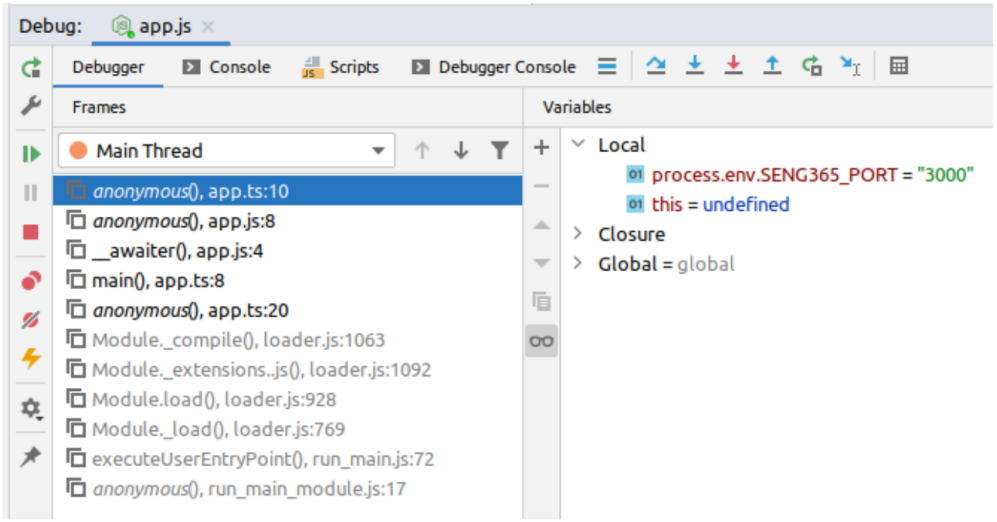
1. We can add breakpoints in our TS code by click to the right of the line number. For this example place a breakpoint in *app.ts* on the line `await connect()`.




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




3. Now you should see a debug toolbar open in the bottom of your window. Importantly on the right we can see all the variables and their current values which is immensely helpful when trying to track down a bug.




Important debugging actions:

- **Resume program** (green play/pause icon on left or F9): This allows us to continue execution as normal when we are paused.
- **Step over** (right angled arrow with underline or F8): This allows us to move over the current line of execution without going into the implementation (i.e. skipping over methods calls).
- **Step into** (down arrow with underline or F7): This continues execution like step over, however this will take you into the next execution (i.e. going deeper into methods). **Note:** You should use this 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 the hotkey F7) and you should be taken to the db.ts file, with the connect function highlighted.
3. If we 'step over' 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 using a debugger, in general we would often be paying more attention to the variables in our local scope (as these are generally the ones we define and which determine how the program functions) to understand what is causing an error.

## 2.3 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. A basic description of the API specification is shown in Table 1 below. **Note:** This API is properly defined in an API specification file discussed in Section 4.1, 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 |

Table 1: Basic API specification for user endpoints..

### 2.3.1 Boilerplate Code and Structure

1. In the *express.ts* config file, before we return the app variable add a line that imports a file called *user.server.routes* (the extension here is not specifically needed but often IDEs default to including it) from the */app/routes* directory. This file will take in the `app` variable as shown in Listing 9 below, add this directly before the `return app;` statement.

```
    require('../app/routes/user.server.routes.js')(app);
```

Listing 9: Adding routes to our express configuration.

2. Create a file *user.server.routes.ts* in the *routes* directory and add the code from Listing 10. This file will import a `users` controller and then define each of the relevant routes as outlined in the specification. Each HTTP method on a route references a function in our controller that will be ran when the endpoint is hit.
**Note:** we define url path parameters by prefixing them with `:` in this case `/:id`, however these will match to any value (including other valid routes) so in some cases we need to be careful of the ordering of our routes.

```
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 );
};
```

Listing 10: A file that defines our user routes.

3. Next we need to create the users controller; create a file in the *controllers* directory called *user.server.controller.ts* and add the code from Listing 11. This file will import the `users` model and contain the five functions that are called from our routes. For now we will simply add 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: Request, res: Response): Promise<void> => {
    return null;
}

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

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

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

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

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

Listing 11: Boilerplate code for user controller.

4. Finally, we need to add our model code. In the *models* directory, create a file called *user.server.model.ts* and add the code from Listing 12. This file imports the database config file so it can connect to it, and contains basic functions for interacting with the database inline with what the controller will need it to do.

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

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

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

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

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

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

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

Listing 12: Boilerplate code for user model.

### 2.3.2 Listing all users

Now we have the boilerplate code set up, we just need to fill in the functions that have been 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 users from the users table as Listing 13 shows. **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 in Listing 15).

```
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 );
    await conn.release();
    return rows;
};
```

Listing 13: User model getAll() function.

2. Now in the *user.server.controller.ts* file, update the `list()` function to call the models `getAll()` function. This function simply waits for and returns the result from the model function and handles the basic error flow as Listing 14 shows.

```
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}`);
    }
};
```

Listing 14: Get all users endpoint.

3. In our `getAll()` function we introduced a new type `User`, this is a custom type meaning that we have to provide our own definition for it. Create a new file at the root of your *app* directory called *user_types.d.ts*. **Note:** The file name doesn't matter but the *.d.ts* suffix is required. Finally add the following type definition in Listing 15.

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

Listing 15: Custom user type.

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 easily by simply navigating to `http://localhost:3000/api/users` in your browser. **Note:** Make sure you have some users in your database for testing before running this. These can easily be added through the phpMyAdmin web portal, under the insert tab.

### 2.3.3 Getting a Single User

1. In the *user.server.model.ts* file, update the `getOne()` function as shown in Listing 16 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 ] );
    await conn.release();
    return rows;
};
```

Listing 16: Getting a user from the database

2. In the controller, update the `read()` function as shown in Listing 17 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 *Not Found* status code if we **do not** find a marching user in 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 }` );
    }
};
```

Listing 17: Get user endpoint

3. Now re-run your application to test. Sending a GET request to */users/api/1* should result in one user being returned. You can do this quickly by going to `http://localhost:3000/api/users/1` in your browser. **Note:** Again make sure you have users in your database, and that the id you are checking for actually exists.

### 2.3.4 Creating New Users

1. In the model file, update the `insert()` function following Listing 18 so that it takes in the username as a parameter and queries the database to insert 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 ] );
    await conn.release();
    return result;
};
```

Listing 18: Adding a user to the database.

2. Now create the controller function following Listing 19. This function gets the username from the POST request body and passes it to the model function above. This function simply returns the result to the user. If there is no username field present in the request body we return a HTTP 400 *Bad Request* status code.

```
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 }` );
    }
};
```

Listing 19: Register user endpoint

3. Now re-run your app to test. This time we are working with a POST request and can not easily do this through our web browser like we did for the GET requests. Instead it is suggested you skip forward to Section 3 where we discuss API testing with Bruno.

## 2.4 Validating Data Easily with Ajv

As we have seen above we often want to validate a client's requests to our API to inform them if something is wrong. When posting a user we checked the body by hand to make sure it had a `username` field. However what happens if this field is empty? the wrong datatype? too long? Whilst we could check all of these cases by hand it becomes a lot of quite generic and repeated code that can lead to more bugs being accidentally introduced. Luckily other developers have already encountered this problem and there are many different techniques that we can implement

to make it easier.

In this short section we will look at using Ajv[13] a JSON schema validator to automatically validate the POST user request. **Note:** The first assignment will expect you to use some form of validation and will include JSON schemas you can use with Ajv if you want.

### 2.4.1 Understanding JSON Schema

We can define our schema using JSON, you can think of a schema as the rules that define the structure of some data, often used in the context of databases but still practical here. In Listing 20 we show the schema for registering a user.

```
1  {
2      "user_register": {
3          "type": "object",
4          "properties": {
5            "username": {
6              "type": "string",
7              "minLength": 1,
8              "maxLength": 64
9            }
10         },
11         "required": [
12           "username"
13         ],
14         "additionalProperties": false
15       }
16 }
```

Listing 20: Schema for register user

On Line 2 we define the schema named `user_register`. This schema is an object of one property a `username` detailed on lines 3-10, we also define a specific min and max length for the string. Finally on lines 11-13 we make sure the username property is required (it must be included) and on line 14 we don't allow other properties.

Whilst JSON schemas can be quite verbose they allow for very specific detailing of object constraints, that can be easily validated by a third-party library without us having to write code to check for each erroneous case.

### 2.4.2 Setting up a Validator

Listing 21 shows a basic `validate` function where we can pass in a JSON schema and an object we want to validate. If there are any issues these will be returned otherwise it will simply return `true`.

Within this file we can also create custom formats using regex[14], which will be helpful for the assignment but outside the scope of this lab.

```
1  import Ajv from 'ajv';
2  const ajv = new Ajv({removeAdditional: 'all', strict: false});
3
4  const validate = async (schema: object, data: any) => {
5      try {
6          const validator = ajv.compile(schema);
7          const valid = await validator(data);
8          if(!valid)
9              return ajv.errorsText(validator.errors);
10         return true;
11     } catch (err) {
12         return err.message;
13     }
14 }
15
```

[13] See `https://www.npmjs.com/package/ajv`

[14] See `https://ajv.js.org/guide/formats.html`, specifically the section "User-defined formats"

```
export {validate}
```

Listing 21: Validator setup for Ajv

### 2.4.3 Validating Requests

Now that we have the `validate()` function set up we can call this from our controller and use it to validate the request body. On line 1 we import the schema from a JSON file *schemas.json* (you may want to use a different path), as JSON is read as an object we can fetch any schema within our JSON file by name as seen on line 5. On lines 4-6 we validate the body against the `user_register` schema, checking on line 7 if there are any issues and return 400 as well as Ajv's (often quite useful) error messages.

Whilst this is a little more effort than the single check we had before, it scales far better allowing for the validation of many fields (as long as they are defined in the schema) without any additional code.

```
import * as schemas from '../resources/schemas.json';
const create = async (req: Request, res: Response): Promise<void> => {
    Logger.http(`POST create a user with username: ${req.body.username}`)
    const validation = await validate(
        schemas.user_register,
        req.body);
    if (validation !== true) {
        res.statusMessage = `Bad Request: ${validation.toString()}`;
        res.status(400).send();
        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}`);
    }
};
```

Listing 22: Using validation in /register

## 2.5 Finishing the Users Functionality

Now that we have updated the post user implementation to use Ajv validation the final two endpoints are up to you to complete. You may choose to use Ajv validation for editing as shown in POST user.

### 2.5.1 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 (it is suggested you also add validation with Ajv). **Note:** Think about what should happen if we don't find the user to edit.
3. Test using Bruno.

### 2.5.2 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 Bruno.

# 3 Testing APIs with Bruno

## 3.1 What is Bruno

Bruno[15] is an API testing suite that gives the developers the ability to create a collection of tests to run against their API. Bruno is pre-installed on the lab machines or can be downloaded for free (without any user account registration) if you are working from your own machine.

A collection of Bruno tests have 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 first 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 if you want extra coverage!).

## 3.2 A quick look at alternatives

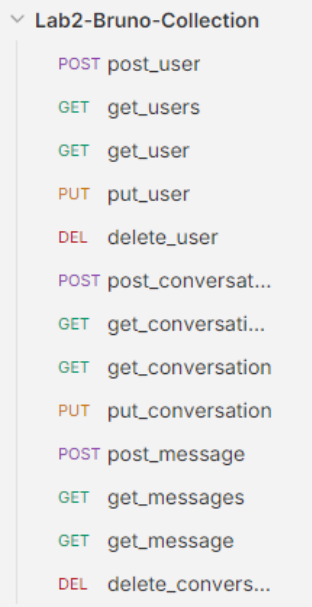
There are many similar API testing tools that occupy a similar niche as Bruno. The most well-known and widely used of these is Postman[16], though in recent years they have reduced the functionality they offer to free end users so open source alternatives like Bruno are gaining traction.

## 3.3 Getting Started with Requests

### 3.3.1 Importing our Bruno Collection

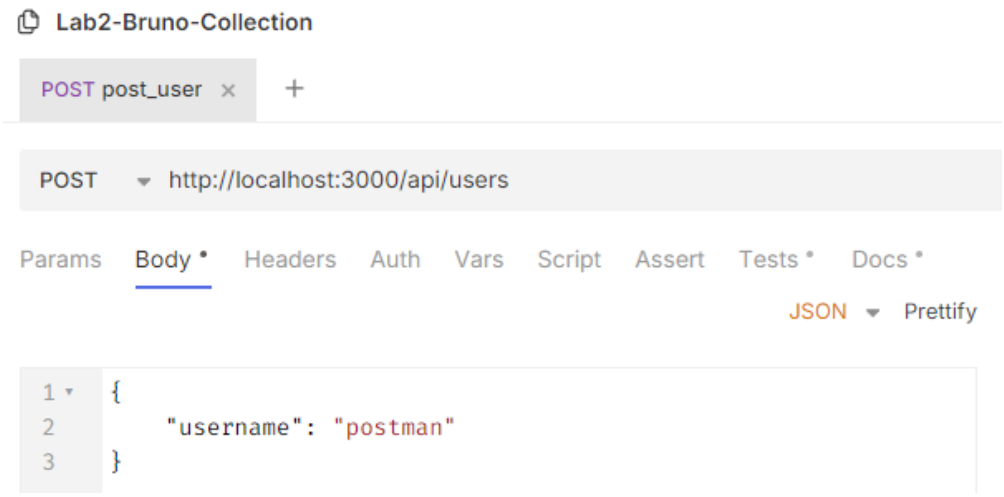
To start, lets load in the Bruno collection discussed above. This will allow us to have a look at some of the different request types, and to test our application so far.

1. To import a collection in Bruno we simply chose `Collection > Open Collection` from the menu bar at the top of our Bruno window.
2. Bruno collections are simply folders, so from the file picker that opened navigate to where you downloaded and unzipped the *Lab2-Bruno-Collection* collection from Learn.
3. Now that we have imported the collection we should see the request loaded in on the left hand side (making sure to expand the entry).




4. Lets run the request for our POST user endpoint implemented earlier.

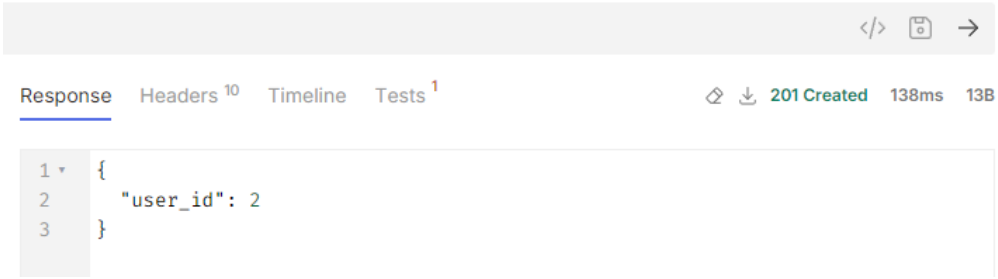
[15] See `https://www.usebruno.com/`
[16] See `https://www.postman.com/`




Importantly on this screen we have:

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

5. Once you click send, if the code is correct you should get a HTTP response with the status code *201 Created* and a JSON object containing our new user's id.




6. Finally feel free to have a look through the rest of the collection and try out the other requests once you have implemented the relevant endpoints.

### 3.3.2 Creating a New Request

Bruno allows for the creation of simple requests using HTTP methods, where you can specify headers, a request body, url parameters, and more. We will not discuss all of these in this lab however Bruno has good online documentation[17].

In the collection provided we only check for 'blue sky' flows, where everything works 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 endpoint you create a good start for more verbose testing is to create a test for each different status code that could be returned. Of course in larger applications there will places where we 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 possible status codes. 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 that matches the id supplied.

To get a 404 we need to provide an id that does not exist in the database, an easy way to do this is specifying a very large number (i.e., 999,999,999). Whilst theoretically we could have a user with this id, within the scope of this lab we aren't expecting to populate 1 billion users.

Now lets try creating a 404 request in Bruno:

1. Create a new request within the collection (you may want to name this something clear i.e., "get_user 404")

[17] See `https://docs.usebruno.com/get-started/bruno-basics/create-a-collection`

2. Make sure the request type is GET
3. Add the url `http://localhost:3000/api/users/999999999`
4. Run the request and check you get a 404 response

Another example of an erroneous status code is 400 *Bad Request*, this is commonly used for requests that provide data (such as POST, PATCH, and PUT) since it informs the client there is an error with the data supplied.

In the case of our application we return a 400 when an invalid user is created with the POST user endpoint.

1. Create a new request within the collection
2. Make sure the request type is POST
3. Add the url `http://localhost:3000/api/users`
4. Add a body of `raw > JSON` type with the content in Listing

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

Listing 23: Invalid POST user body.

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

You may have noticed that all of the endpoints also return a 500 *Internal Server Error* status code which we are not testing for. This is often a blanket status code for errors that are not expected and as such there is no meaningful response for the client. 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).

## 3.4 Getting Started with Bruno Tests

Now that we have learnt how to create requests, we can create proper tests where responses from our server are automatically validated instead of checking the results manually.

### 3.4.1 Running All Tests in a Collection

In the collection provided each request tests the endpoint as well, while these tests are ran 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) and select the `Run` option. This should open a new page with the total number of requests and a `Run Collection` button. Click this and check that the requests and tests run (and hopefully pass) as expected. Note that requests for end points you have not yet implemented will fail.
**Note:** If for some reason your version of Bruno does not match what we have discussed above ask a tutor for help or look online.

The results screen (after the tests have run) should look like Figure 5 below. **Note:** Unless you have completed the rest of the lab (conversations and messages) you should have ∼8 passing tests.

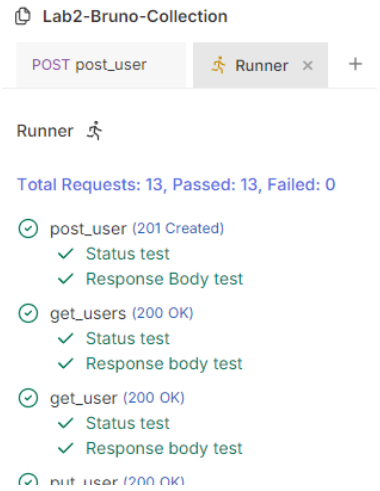



Figure 5: Result of running Bruno Collection.

### 3.4.2 Adding Tests to our Requests

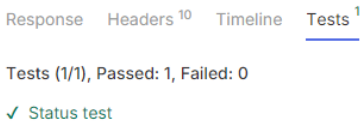
We are going to extend the requests we made in the section above so that they can automatically test the API and inform us if it acting as expected. The tests we create in the section will not be as advanced as those already in the collection, thus it is recommended that you look at the existing tests and online resources such as the Bruno documentation[18] to gain a deeper understanding of how these tests can be used.

1. Within the GET user 404 request created earlier navigate to the 'test' tab.
2. Add the following code from Listing 24 to the text area.

```
test("Status test", function () {
    expect(res.getStatus()).to.equal(404); // tests status is 404
})
```

Listing 24: Bruno 404 status test

3. Run your request again and now you should see the tab 'Test Results' in the response pane, with the 'Status test' passing.




4. A similar test can be added to our POST user 400 request shown in Listing 25.

```
test("Status test", function () {
    expect(res.getStatus()).to.equal(400); // tests status is 400
})
```

Listing 25: Bruno 400 status test

The tests we have created above are very basic, and only check the status code of the response, however Bruno allows for much more verbose testing functionality. To see this in action refer to the tests for the POST user request (provided in the collection). This request has 2 tests, one like those above checks for a specific status code (201 in this case), and a test that checks the body contains a non-empty object with a `user_id` field of type number is returned, it then sets this to a global variable `test_id`. If you look ahead to the GET, PATCH, and DELETE request for /users/id you can see this value being referenced in the url (surrounded by double curly braces, e.g., `http://localhost:3000/api/users/{{test_id}}`), this allows for these tests to reference the user that was 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

[18] See `https://docs.usebruno.com/testing/introduction`

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 the client can confirm everything is correct for themselves and have the required data to display it without needing to send a GET request.

# 4 Implementing the Rest of the API - Recommended

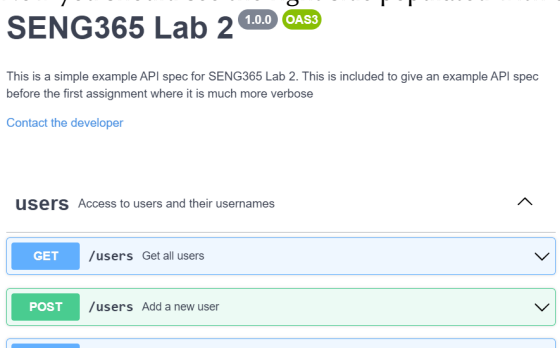
Like the last lab, this exercise is optional but **heavily** recommended due to the large number of concepts covered. Once you feel comfortable with the concepts you are ready to start working on the assignment.

## 4.1 Using an API Spec

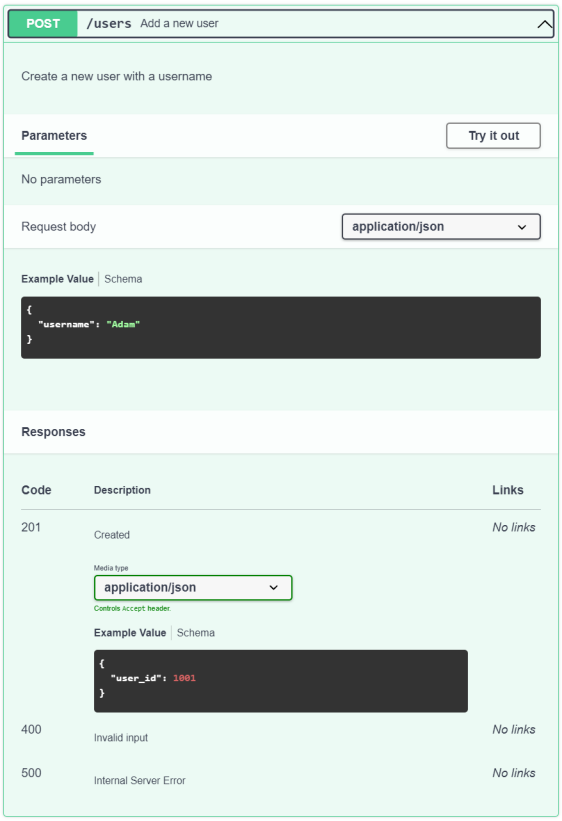
Commonly when creating an API a concrete specification is created beforehand so developers know all the endpoints that need to be created and what they should do. These specifications are also important after the API has been made, as the consumer of the API needs to understand how they can use each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

An API spec that details the functionality for the remaining endpoints (conversations and messages) can be found on Learn.

1. First copy the contents of this file into the left side of the swagger online editor `https://editor.swagger.io/`. Other editors and plugins for IDEs exist, however for simplicity only Swagger will be discussed here.
2. Now you should see the right side populated with content titled "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. We can click on any request to see more detailed information about the expected parameters, headers, request body, and more.




Here we see the POST user endpoint has no parameters, takes a body with a username, and returns a json body with a user_id and the status code 201, other possible status codes are 400 and 500.

5. Explore the rest of the API spec before moving on to gain more understanding of what is required to complete the rest of the Chat App API and where to start.
6. For further reading about OpenAPI 3.0.0, the specification used for this lab and the assignment refer to the specification on their website[19].

## 4.2 Completing the API

You now have all the skills required to implement the rest of the API to the specification introduced in Section 4.1. Doing so will provide you with great experience for starting the first assignment. Simple Bruno tests have been included so you can check that you are completing the endpoints as expected.

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

[19] See `https://spec.openapis.org/oas/v3.0.0`