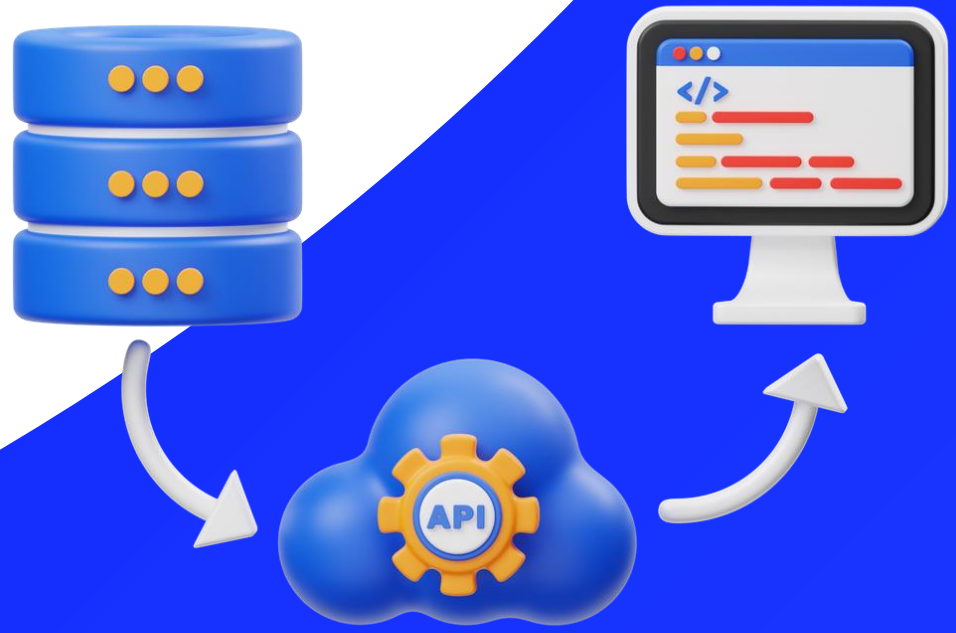
Institute of Data

Software Engineering

Module 9 API Development



Agenda

- Section 1 — Introduction to REST
- Section 2 — MVC Structure
- Section 3 — REST API using MongoDB
- Section 4 — REST API using MySQL
- Section 5 — Micro Services
- Section 6 — Sockets

Section 1:

Introduction to REST

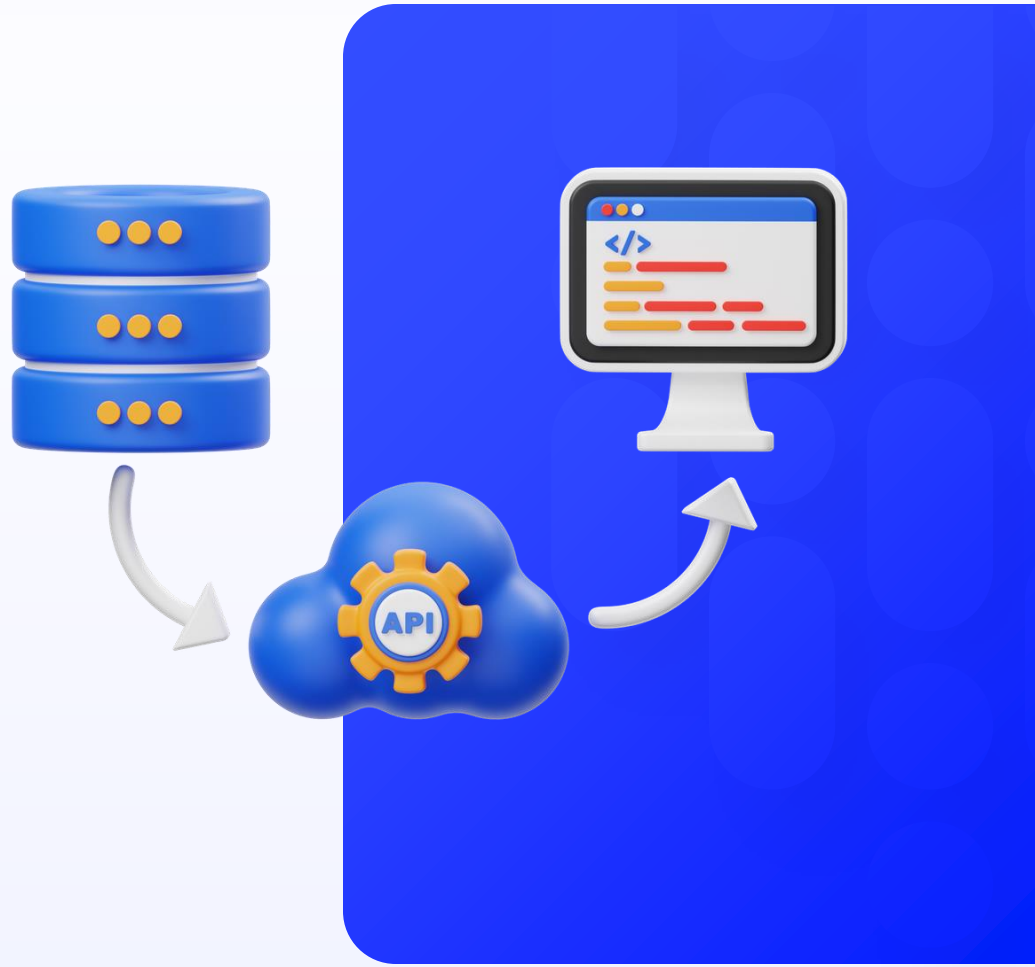
It is natural for programs to be able to communicate with one another in order to create interactive and scalable applications. An API (short for Application Programming Interface) is a collection of rules that allows various programs to communicate with one other. These programs could be a software library (for example, the Python API), an operating system, or a web server (a web API).

One of the most significant advantages of an API is that the requester and responder do not need to be familiar with each other's software. This enables services employing various technologies to communicate in a consistent manner.

What is a REST ?

REST is an acronym for **Representational State Transfer**. It's a design style for creating networked applications (i.e. apps that use some form of a network to communicate). It is the most often used method for creating web APIs. When a REST API is developed, it follows a set of rules that determine the API's requirements.

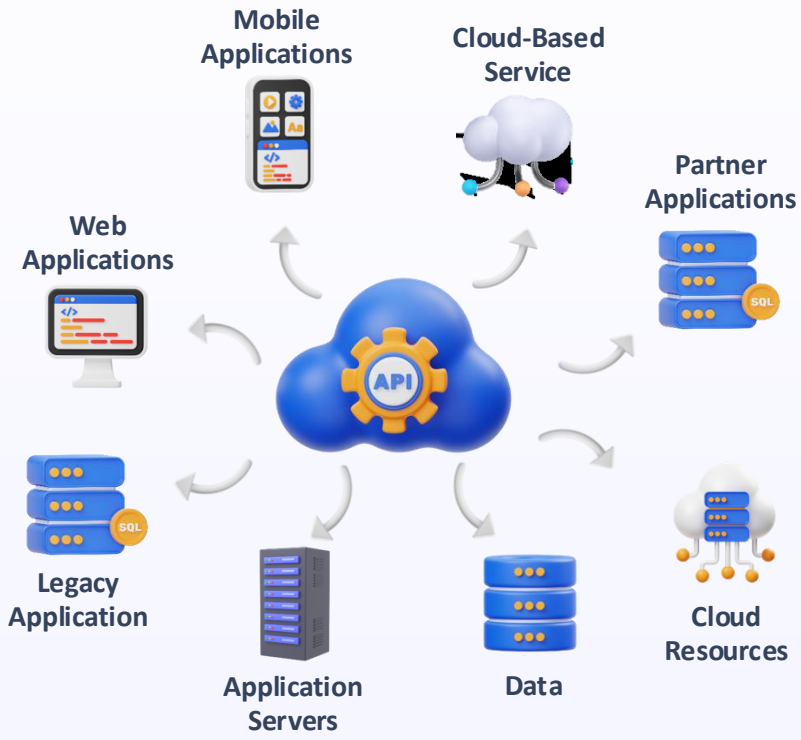
Any data (e.g., image, video, text, etc.) is treated as a resource by REST, which the client can fetch, edit, and delete. REST requires that a client be able to access a certain URL and send a request to complete the necessary function. After that, the server responds appropriately.



What is REST?

Consider it a contract between the two programs: the client (requester) and the responder (server). The responder will give the requester Y if the requester sends X to the responder. X and Y are given in the API documentation and explained in the contract between the two parties.

REST is **stateless**, which means that each client request must include all of the information needed for the server to interpret it. For example, the client cannot presume that the server recalls their previous request.



Principles of REST APIs

Dr. Fielding, who was the one who defined the REST API design in 2000, established six fundamental principles.

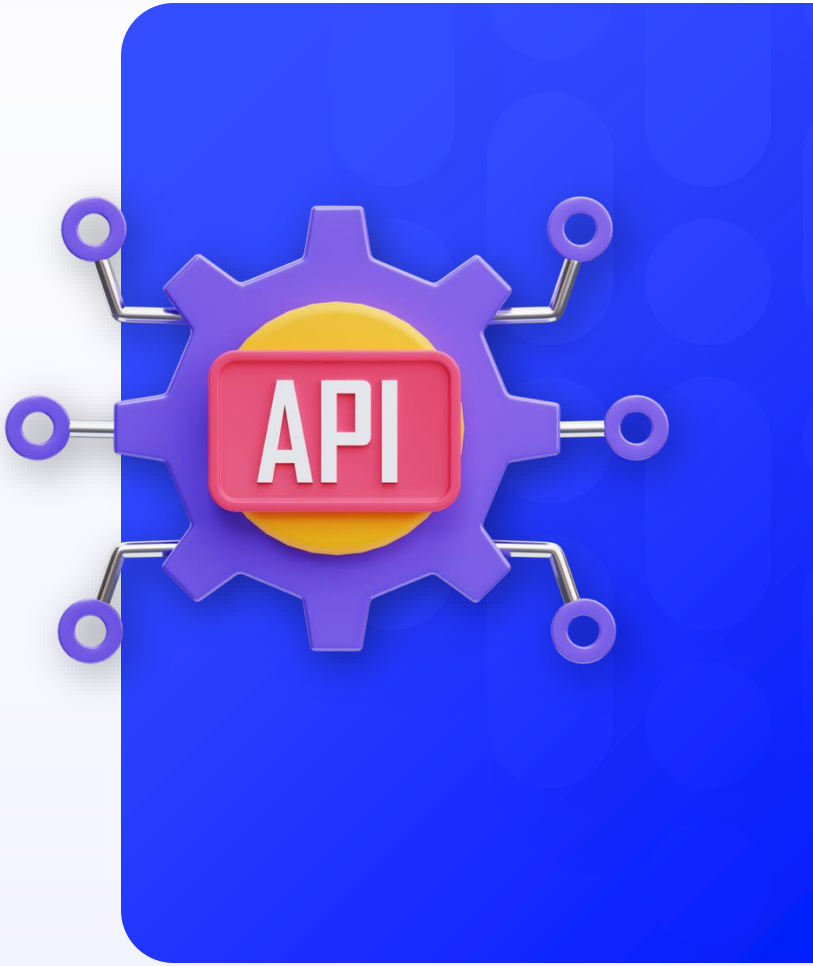
Stateless - Requests sent from a client to a server will include all of the necessary information for the server to interpret the client's requests. This could be in the URL, query-string arguments, the body, or even the headers. The body contains the state of the requesting resource, whereas the URL is used to uniquely identify it. After the request is processed, the client receives a response in the form of body, status, or headers.

Client-Server - The client-server design allows for a consistent user interface while also separating clients from servers. This improves the server components' portability across many platforms as well as their scalability.

Principles of REST APIs

Uniform Interface - REST contains the following four interface requirements to achieve uniformity throughout the application:

- Resource identification
- Resource manipulation using representations
- Self-descriptive messages
- Hypermedia as the engine of application state

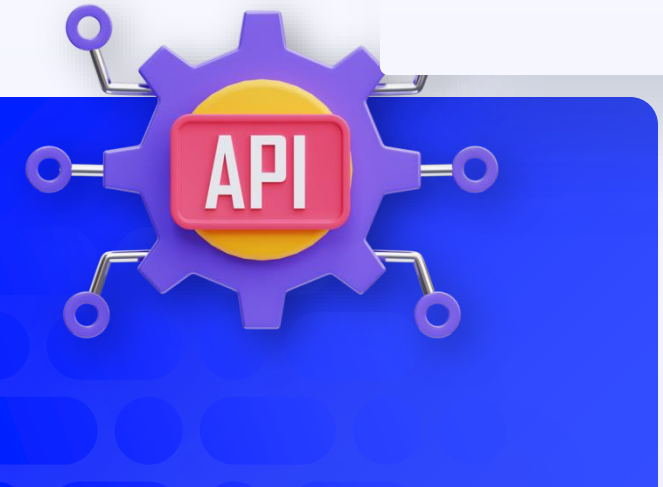


Cacheable - Applications are frequently made cacheable in order to improve performance. This is accomplished by either implicitly or explicitly identifying the server's answer as cacheable or non-cacheable. If the response is cacheable, the client cache can reuse the response data in the future for similar responses.

Principles of REST APIs

Layered System - By constraining component behaviour, the layered system architecture makes an application more reliable. Because components in each tier cannot interact beyond the next adjacent layer they are in, this architecture helps to improve the application's security. It also supports load balancing and offers shared caches to help with scalability.

Code On Demand - This is an optional requirement that is rarely utilised. It enables the download and use of a client's code or applets within the program. In essence, it makes clients' lives easier by developing a smart program that isn't reliant on its own coding structure.



Methods of REST APIs

All of us who work with web technologies perform CRUD activities.

These refer to the core operations of **creating**, **reading**, **updating**, and **deleting** resources.

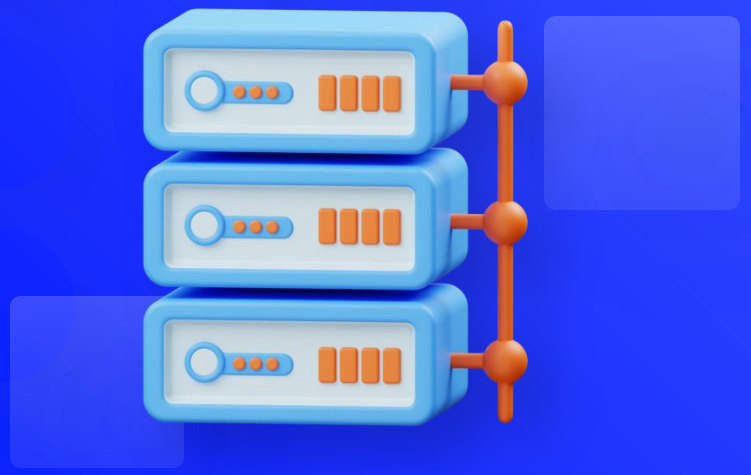
To carry out these tasks, you can use HTTP methods, which are nothing more than REST API methods.

REST API example for Posts

Creating:	/posts	→	POST new resource
Reading:	/posts/123	→	GET resource id 123
Updating:	/posts/123	→	PUT updated resource id 123
Deleting:	/posts/123	→	DELETE resource id 123
Read All:	/posts	→	GET all posts
Front end route:	/posts/new	→	GET page with form to create
	/posts/123/edit	→	GET page with form to edit
Read post comments:	/posts/123/comments	→	GET comments for post 123

Section 2:

MVC Structures



Any software development project's success hinges on good architecture. This enables not just smooth development processes among teams, but also the application's scalability. It ensures that developers will have little trouble refactoring various portions of the code whenever new modifications are required.

In diverse languages, architecture patterns such as MVT in Python, MVVM in Android, and MVC in JavaScript apps exist.

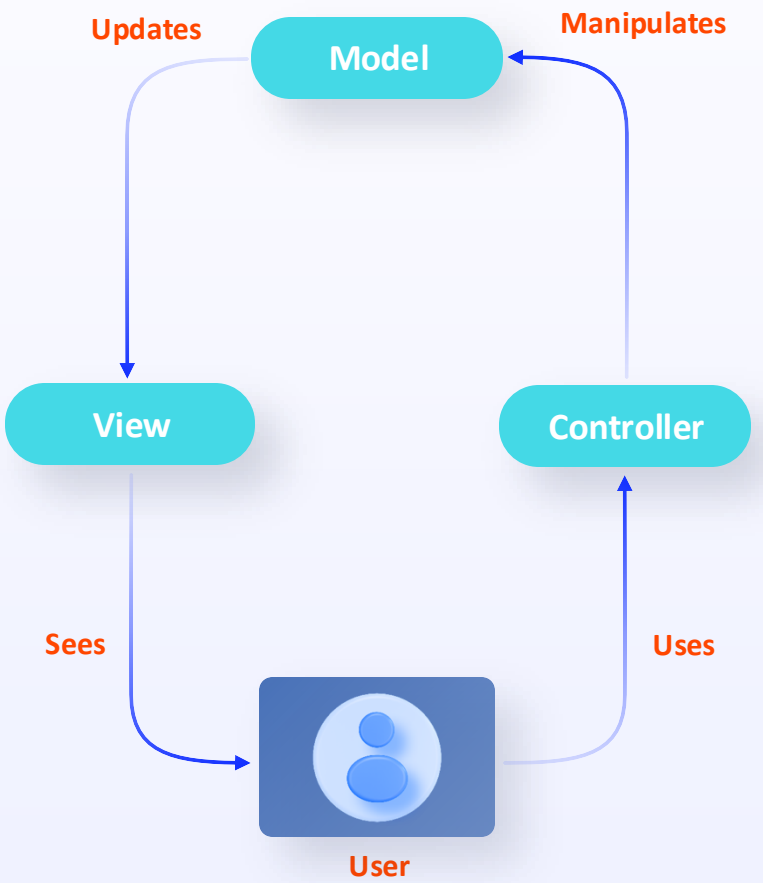
MVC architecture divides the whole application into three parts; the **Model**, the **View** and the **Controller**.

MVC Structures

Model : Our data is defined in this section. It's where we save our schemas and models, or the blueprint for our app's data.

View : This includes templates as well as any other interaction the user has with the app. It is here that our Model's data is given to the user.

Controller : This section is where the business logic is handled. This includes database reading and writing, as well as any other data alterations. This is the link between the Model and the View.

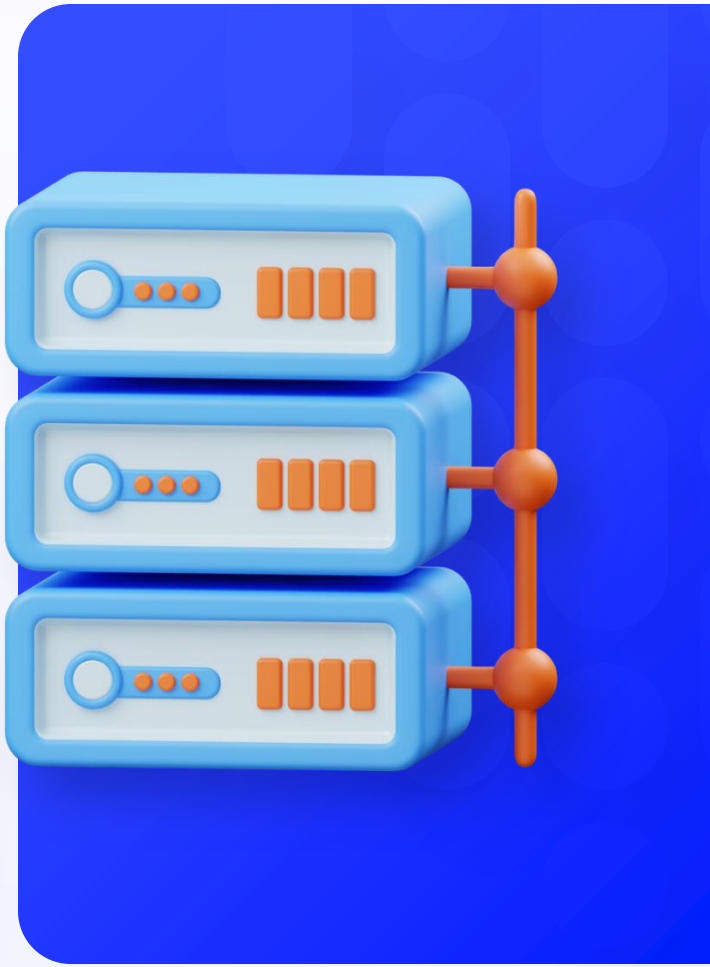


MVC Structured Express App

Now let's try to understand why utilising an MVC structure is beneficial and how it may transform your normal Express/Node.js application into a high-level application just by wrapping it in an MVC Architecture.

Since the architecture is made up of three key components (Controller, Model and View), MVC applications should have a folder structure similar to something like this:

```
controllers/
models/
views/
routes/
server.js
```



You can also reference here to see the folder structure for a MVC Application.

MVC Structured Express App

Let’s understand what goes in each of these MVC folders for our application.

Model: In this section, we define our data. It's where we save our schemas and models, as well as the data blueprint for our app.

View: This includes templates as well as any other interactions with the app that the user has. Our Model's data is shown to the user here. Some applications may not have this folder, if the application’s main purpose is to serve as a server-side application and is eventually going to be connected to another standalone frontend application.

Controller: Controllers handle the business logic of our application. When we call an endpoint from the user we send a bunch of data to that end point, and what needs to be done with that data is handled by our controllers. So whenever we create a route file, we also create a corresponding controller file which handles all the business logic for all the different route endpoints in that file.

Routes: Routes are the files that are responsible to serve the endpoints of our application to the user. We try to create separate route files for separate functionalities in our application.

Exercise 1

Review your Calculator application from Module 5 to make sure that it uses a proper MVC structure.

Section 3:

REST API using MongoDB

In module 5 we created a very basic Rest API which has routes that take some input from the user, and then performs business logic on that data which is written in the controller.

In this section we will do something similar, but we will connect our app with a database and perform CRUD operations.

For this section as we are using MongoDB, we will be using a npm package called **Mongoose** to connect with MongoDB through our Nodejs Application.

You can also refer here to learn in detail about Mongoose npm package.

Connect Using Mongoose

First create a new express application using Node.js. To do this:

1. Create a new folder for your app, called **mongodb-app** or similar
2. Within this folder create a **server.js** file, with the starter content as shown on the right
3. From within the new folder, run **npm init** and accept the defaults to create a package.json file and initialise your new app
4. Run **npm install express** to install the express module.
5. Run **npm install mongoose** to install the Mongoose package.

The next step is to enable our app to connect to a MongoDB database.

Before we connect our application using mongoose, please make sure your mongodb server is running in the background.

```
const express = require("express");
const app = express();
require("dotenv").config();

// parse requests of content-type - application/json
app.use(express.json());

app.get("/", (req, res) => {
    res.json({ message: "Welcome to my MongoDB
application." });
});

// set port, listen for requests
const PORT = process.env.PORT || 8080;
app.listen(PORT, () => {
    console.log(`Server is running on port ${PORT}.`);
});
```

Mongoose

Now that we have mongoose installed, let's use it to connect to our mongodb. For doing that, we will create a file called dbConnect.js in our root folder.

The contents of dbConnect.js should look like the code on the right. **You may have to change the URI value to match your mongodb connection string, but you don't need to create myFirstDatabase beforehand.**

Once we have created dbConnect.js we need to **import it in our server.js file**. To do that we simply add (near the top) :

```
let dbConnect = require("./dbConnect");
```

```
'use strict';
const Mongoose = require('mongoose');
// if the connection fails, try 127.0.0.1 instead of
localhost below
const uri = process.env.DB_URI ||
"mongodb://localhost/myFirstDatabase";

// Connect to MongoDB
Mongoose.connect(uri)
    .then(() => console.log('MongoDB Connected'))
    .catch(error => console.log('MongoDB Error:
'+error.message));

// Get the default connection
const db = Mongoose.connection;

// Bind connection to error event (to get notification of
connection errors)
db.on("error", console.error.bind(console, "MongoDB
connection error:"));

exports.Mongoose = Mongoose;
```

Environment Variables

When working with databases in express, we usually need to store connection details such as database URIs, usernames, passwords, ports, etc. It is not considered best practice to hardcode these values directly into the code, both for security purposes and for ease of deploying the app in multiple environments (development, staging, production, etc).

Instead we use an npm package called **dotenv**, which reads from special configuration files called `.env` which contain all of the environment variables needed by the app.

Simply run `npm install dotenv` in your express app and create a file that looks something like the right (values may change depending on your local setup), with uppercase constant names assigned to appropriate values:

```
DB_URI=mongodb://localhost/myFirstDatabase
PORT=8080
```

Models

Now if we start our server (using npm `start`) we should get something like this in our console:

```
Navit@alessios-Mini mvc-structure % npm start

> mvc-structure@0.0.0 start
> node server.js

Listening on port  8080
MongoDB Connected
```

Now let's create our first model. First create a folder called models with two files: **`user.js`** and **`index.js`**.

Add the code on the right to **`user.js`** to create the schema.

Next we add the below code to **`index.js`**:

```
'use strict'
module.exports = {
   User: require('./user')
};
```

```
const mongoose = require("mongoose");
const Schema = mongoose.Schema;

const userSchema = new Schema({
  firstName: { type: String, trim: true, required: true },
  lastName: { type: String, trim: true, required: true },
  emailId: { type: String, trim: true, required: true,
unique: true },
  password: { type: String },
  createdAt: { type: Date, default: Date.now },
  updatedAt: { type: Date, default: Date.now }
});

module.exports = mongoose.model("user", userSchema);

// The "user" mentioned in the above line should be in
// lowercase singular form ..whereas the actual collection
// name in mongodb will be in the plural form.
// Refer to mongoose documentation for more:
https://www.npmjs.com/package/mongoose
// The first argument is the singular name of your
collection.
// Mongoose automatically looks for the lowercase plural
version. For example, if you use

// const MyModel = mongoose.model('Ticket', mySchema);
// Then MyModel will use the tickets collection, not the
Ticket collection.
```

Routes

Now that we have our Model ready and connected, let's create some API endpoints to add a user to our database and also retrieve the users from our database.

First create a routes folder containing a file called `userRoutes.js`, using the content on the right.

Now that we have our routes ready, we also need to add them to our server.js file so the application knows that such routes exist. To do this, we add this code to our server.js (before `app.listen`):

```
let userRoutes = require('./routes/userRoutes');
app.use('/api/users', userRoutes);
```

```
let express = require("express");
let router = express.Router();
let Controllers = require("../controllers"); // index.js

// Adds a GET route to return all users
router.get('/', (req, res) => {
   Controllers.userController.getUsers(res);
})

// Adds a POST route to create a new user
router.post('/create', (req, res) => {
   Controllers.userController.createUser(req.body, res);
})

module.exports = router;
```

Controllers

Now once we are ready with our routes, we need to create the appropriate controllers to handle the business logic of these requests.

First create a new folder called `controllers`.

Next, create two files called `userController.js` and `index.js` in our controllers folder.

Add the content on the right to **userController.js** (including a function for each controller operation) and the below content to **index.js**:

```
module.exports = {
    userController: require('./userController')
}
```

```
"use strict";
let Models = require("../models"); // matches index.js

const getUsers = (res) => {
    // finds all users
    Models.User.find({})
        .then(data => res.send({result: 200, data: data}))
        .catch(err => {
            console.log(err);
            res.send({result: 500, error: err.message})
        })
}

const createUser = (data, res) => {
    // creates a new user using JSON data POSTed in request body
    console.log(data)
    new Models.User(data).save()
        .then(data => res.send({result: 200, data: data}))
        .catch(err => {
            console.log(err);
            res.send({result: 500, error: err.message})
        })
}

module.exports = {
    getUsers, createUser
}
```

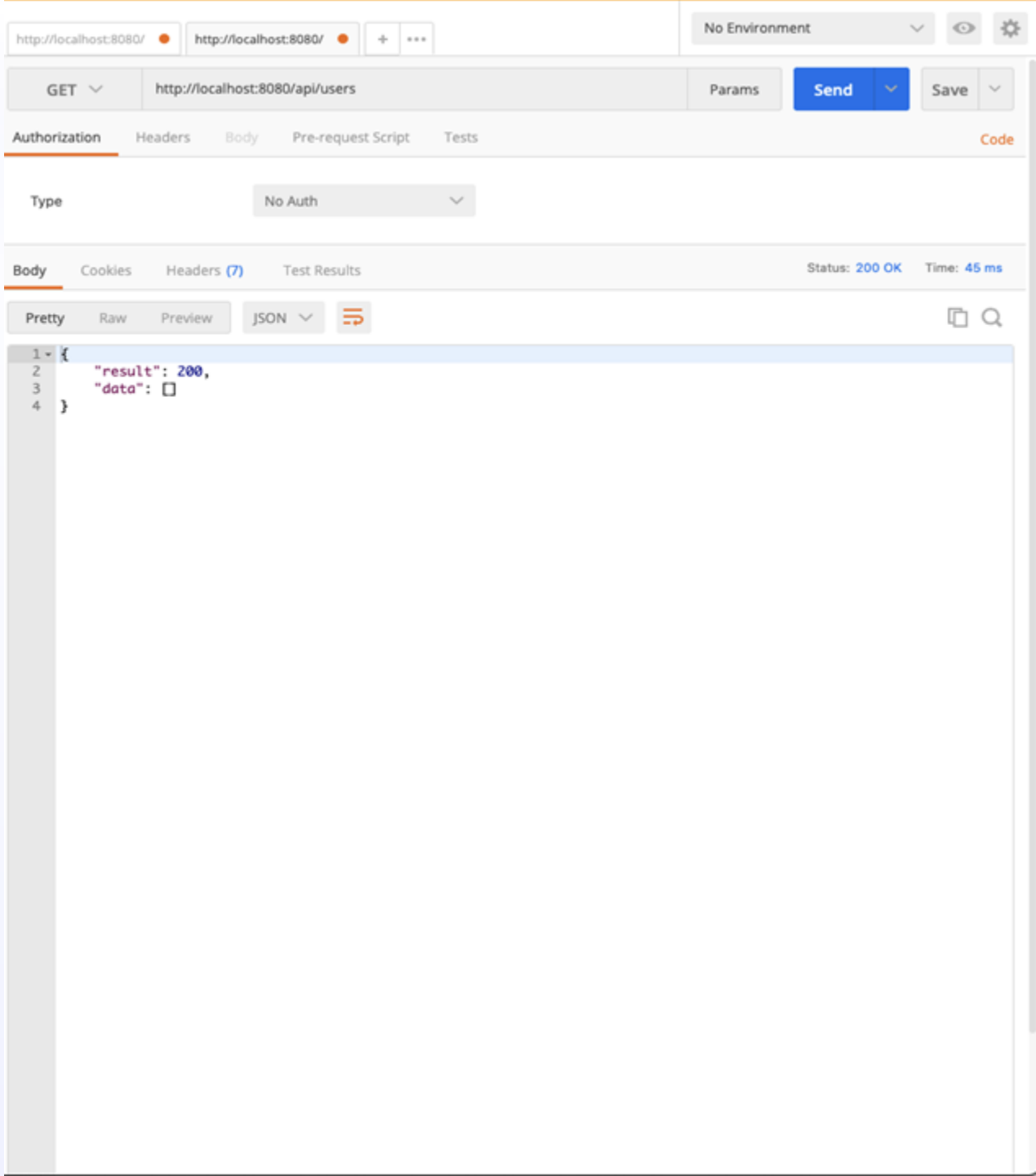
Mongo REST API

Now all our routes should be connected to their appropriate controller and we can test the APIs.

In order to test the APIs I would suggest the use of Postman, which is a google-based tool to test HTTP REST APIs. (You can also use a VSCode extension called Thunder Client.) So let's try the REST API we created. First we need to start the server using **`npm start`**.

Let's first try our **getUsers** endpoint, using the right port (8080) and route path (/api/users/). At the start the API should respond with no users, as we haven't added any yet.

If you get an error, check the application console and make sure your DB URI and route path is correct.



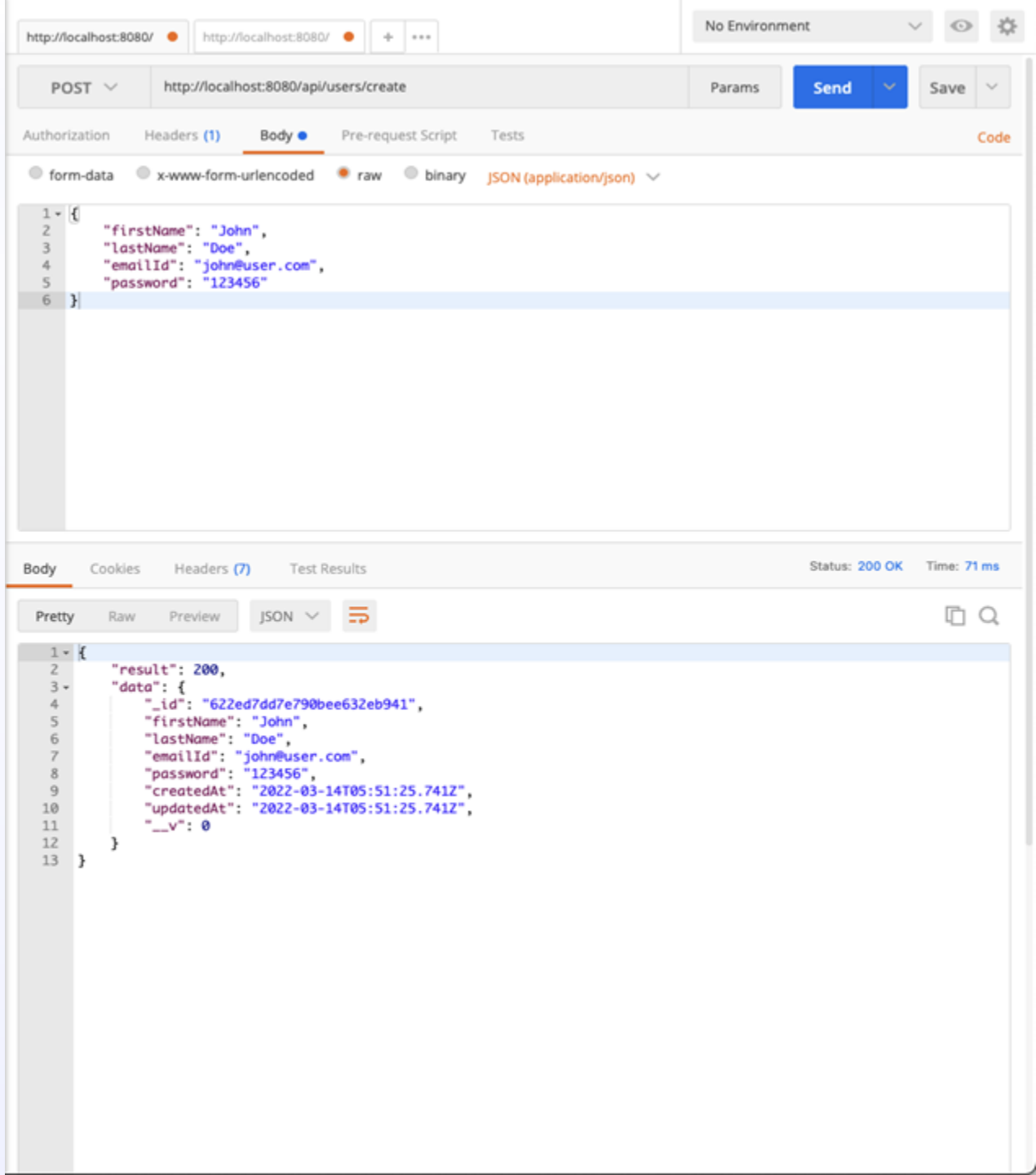
Mongo REST API

Now let us add a user to the database. For this we can use the **`createUser`** endpoint and send the below JSON to /api/users/create as sample data to save.

In Postman, choose the **POST** operation and copy the below to the **Body** using **raw** data, encoded as **JSON**:

```
{
        "firstName": "John",
        "lastName": "Doe",
        "emailId": "john@user.com",
        "password": "123456"
}
```

So we can see here our user was successfully added to our database.



Mongo REST API

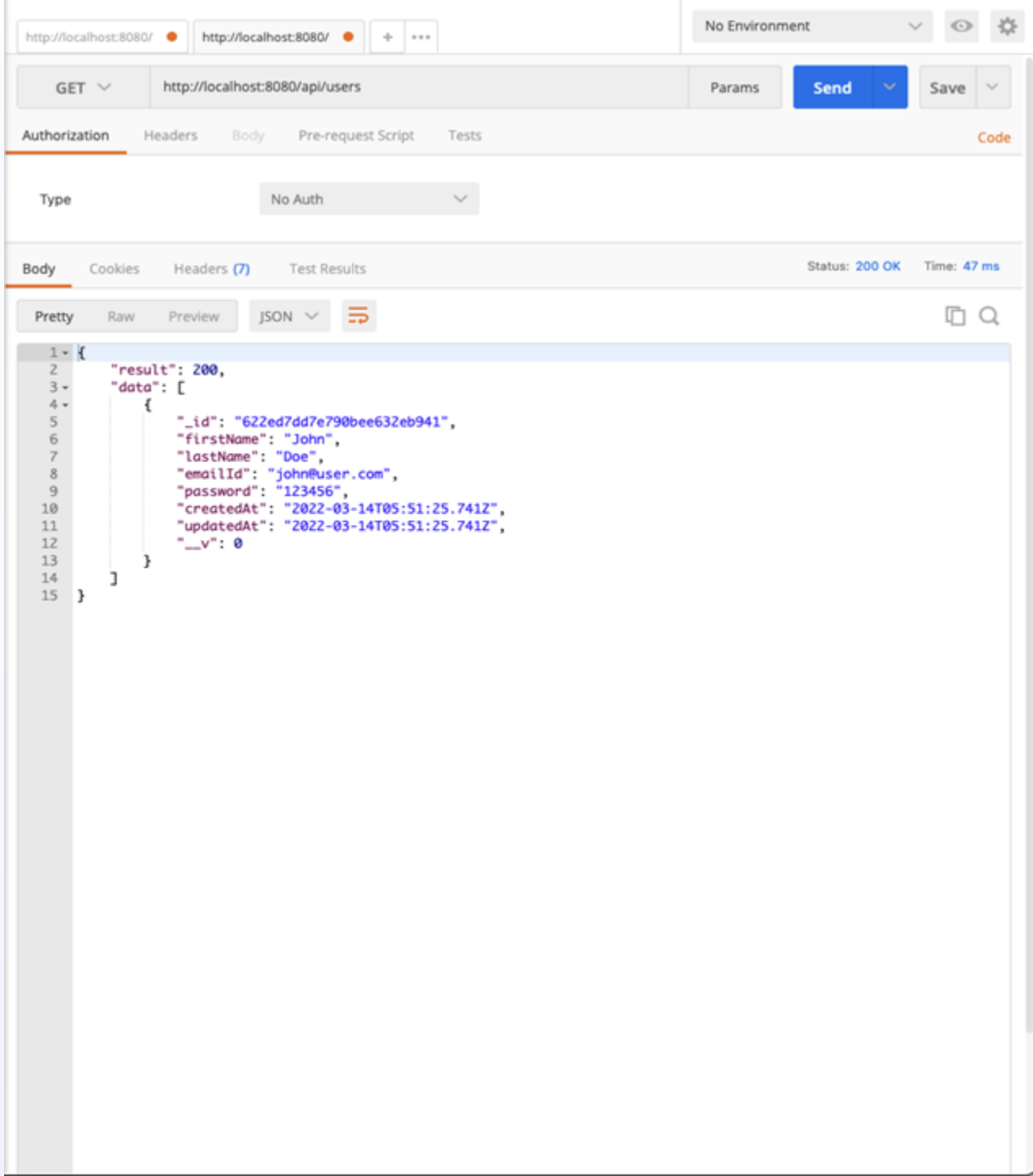
Now let's try our getUser API and verify if our user was added successfully.

From the result, you can see our user was successfully created, as the getUsers API endpoint now returns an array containing our new user.

Try adding a second user using your own details and verify that there are now two users.

Similarly we can create Update and Delete users API endpoints also (using the PUT and DELETE methods).

You can also refer to the github repo here if you face any issues regarding the code or the folder structures.



Mongo REST API

To update and delete users, use the following example code and add the below to `userRoutes.js` and the controller functions on the right to `userController.js`.

```
router.put('/:id', (req, res) => {
    Controllers.userController.updateUser(req, res)
})

router.delete('/:id', (req, res) => {
    Controllers.userController.deleteUser(req, res)
})
```

The /:id syntax captures everything after the / and stores it in a request param of that name - in this case id.

We can therefore update users using a request URL such as http://localhost:8080/api/users/1234, where 1234 is the matching user ID, which will be stored in the `req.params.id` property.

```
const updateUser = (req, res) => {
    // updates the user matching the ID from the param using JSON
data POSTed in request body
    console.log(req.body)
    Models.User.findByIdAndUpdate(req.params.id, req.body, { new:
true })
        .then(data => res.send({result: 200, data: data}))
        .catch(err => {
            console.log(err);
            res.send({result: 500, error: err.message})
        })
}

const deleteUser = (req, res) => {
    // deletes the user matching the ID from the param
    Models.User.findByIdAndDelete(req.params.id)
        .then(data => res.send({result: 200, data: data}))
        .catch(err => {
            console.log(err);
            res.send({result: 500, error: err.message})
        })
}

// ++ Test updating and deleting a user using Postman
```

Foreign Keys

Although MongoDB itself does not enforce a schema, constraints or foreign key relationships, the Mongoose package does allow us to do this, mainly so that we can combine records from multiple related collections in a single query.

In the model, the field that references an ID from another collection should be defined like:

```
// foreign key needs ObjectId type and a 'ref' to the referenced schema
 userId: { type: mongoose.Schema.Types.ObjectId, ref: 'user' },
```

..where the value of 'ref' should match a collection name passed to `mongoose.model`.

See the Mongoose documentation for more information and examples.

This technique can be used to implement the foreign keys for your blogging data model in Exercise 2.

Foreign Keys

Once a foreign key has been defined in the model using a `ref`, the `populate` function in Mongoose can be used to combine data from multiple related collections in a single find.

*The **`path`** passed to **`populate`** should match the field name defined as a **`ref`** in the model.*

```
const getUserPosts = (req, res) => {
    // finds all posts for a given user and populates with user details
    Models.Post.find({userId: req.params.uid}).populate({path: 'userId'})
      .then((data) => res.send({ result: 200, data: data }))
      .catch((err) => {
        console.log(err);
        res.send({ result: 500, error: err.message });
      });
};
```

The above controller function will (after first setting up a Post model) find all posts for a given user id, and include both post fields and user fields in the combined result.

Exercise 2

Create an express back-end application for a Blog website using MongoDB. You should refer to your database model from Module 8 for this, ensuring that your app matches the model.

Requirements:

- Your system should have a proper MVC Structure
- The system should be able to create users.
- The users should be able to create multiple posts (posts should be very basic with title, description and image)
- Other users should be able to like the posts and comment on the posts.

Section 4:

REST API using MySQL

In this section we will again be creating an MVC structured express app to set up API endpoints using routes, models and controllers. We will also connect it with a database in order to perform changes in data.

For this section we are using **MySQL**, using a npm package called **Sequelize** to connect with MySQL through our Nodejs Application.

Sequelize is a promise-based **Object Relational Mapping (ORM)** package that supports not only MySQL but also Postgres, MSSQL and more. Like Mongoose, it offers extra tools and abstraction so you don't need to write complex SQL queries directly. If you prefer to create and work with SQL directly, the **mysql2** npm package is a better promise-based option.

You can also refer here to learn in detail about Sequelize npm.

Prerequisites for using MySQL with Express

So **before** we create an express application to connect with MySQL, we need to run the MySQL Server, and use terminal or Workbench to **create a database** that we can use. This is one step that is very different from using MongoDB.

The mysql service needs to be running on your computer. You can check this using a terminal as shown below, or by using MySQL Workbench.


```
# mysql -uroot -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 17
Server version: 8.0.28 MySQL Community Server - GPL

Copyright (c) 2000, 2022, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

Prerequisites for using MySQL with Express

```
> CREATE DATABASE myFirstDatabase;
```

```
mysql> CREATE DATABASE myFirstDatabase;
Query OK, 1 row affected (0.05 sec)

mysql>
```

Let's now create our first Database. For doing this we will simply type CREATE DATABASE <database name> '

This would have successfully created a new database for use with the name **myFirstDatabase**. You can also use WorkBench to do this step.

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| myBlog             |
| myFirstDatabase    |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
6 rows in set (0.01 sec)
```

Connect Using Sequelize

Now that we have mysql running in the background and our database created, we need to create a new express application and then install the required packages in our application.

Create a new folder called **`mysqldb-app`** or similar, and create a file called **`server.js`** inside it. Then initialise the app with npm by running **`npm init`** from the new folder and accepting all of the defaults. *The server.js file should initially have the same content as slide 18.*

Next run these commands to install **express** and **dotenv**, and then **sequelize** and **mysql2**:

```
> npm install express
> npm install dotenv
> npm install sequelize --save
> npm install mysql2 --save
```

Now that we have sequelize and mysql2 installed into our app, let's connect to our mysql database by creating another file called **`dbConnect.js`** in our root folder.

Sequelize

The contents of **dbConnect.js** should look something like this (see right):

Once we have created this file we need to import it in our **server.js**. To do that we simply add (after requiring dotenv):

```
let dbConnect = require("./dbConnect");
```

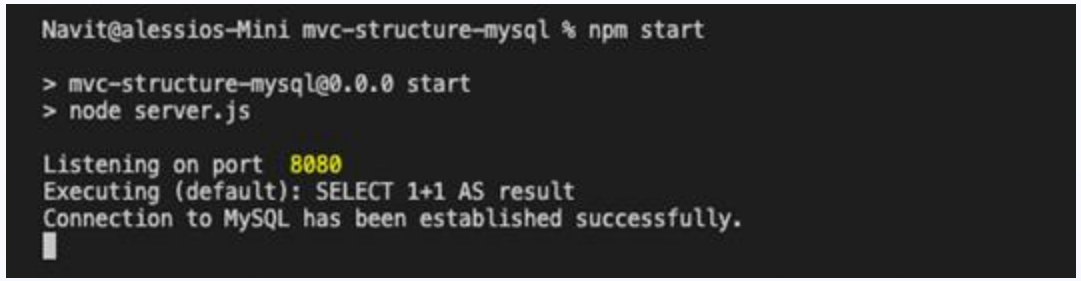
We also need to create a **.env** file with our database information (change as needed):

```
DB_NAME=myFirstDatabase
DB_USER=root
DB_PASSWORD=root
DB_HOST=localhost
DB_PORT=3306
PORT=8081
```

```
'use strict';
const { Sequelize } = require('sequelize');

// Sequelize is a package that abstracts out the need to write
// SQL queries, relying instead on their models to do it for you

const sequelize = new Sequelize(process.env.DB_NAME,
    process.env.DB_USER, process.env.DB_PASSWORD, {
    host: process.env.DB_HOST,
    dialect: 'mysql'
});

const connectMysql = async () => {
    try {
        await sequelize.authenticate();
        console.log(`Successful connection to MySQL Database
${process.env.DB_NAME}`);
    } catch (error) {
        console.error('Unable to connect to MySQL database:',
error);
        process.exit(1);
    }
}

connectMysql();

module.exports = {
    Sequelize: sequelize
}
```

Models

Now if we start our server we should get something like this in our console of application.



If your server fails to connect, try replacing DB_HOST=localhost with DB_HOST=127.0.0.1.

Now let's create our first model. For doing that in our **models** folder we create a file called **`user.js`** and add this code (see right) to that file.

```
const { DataTypes, Model } = require("sequelize");
let dbConnect = require("../dbConnect");

const sequelizeInstance = dbConnect.Sequelize;

class User extends Model { }

// Sequelize will create this table if it doesn't exist on startup
User.init({
    id: {
        type: DataTypes.INTEGER, allowNull: false, autoIncrement:
true, primaryKey: true
    },
    firstName: {
        type: DataTypes.STRING, allowNull: false
    },
    lastName: {
        type: DataTypes.STRING, allowNull: false
    },
    emailId: {
        type: DataTypes.STRING, allowNull: false, unique: true
    },
    password: {
        type: DataTypes.STRING, allowNull: false
    }},
    {
        sequelize: sequelizeInstance, modelName: 'users', // use
lowercase plural format
        timestamps: true, freezeTableName: true
    }
)
module.exports = User;
```

Models

Next we add this content to our **index.js** file in the **models** folder.

Using an index.js file like this just allows us to import and initialise all models at once.

If we need to create relationships between models, we can also do that here after syncing them via the init function.

When adding a new model, make sure to:

- require it,
- sync it, and
- export it.

```
'use strict'
const User = require('./user') //require the model

async function init() {
    await User.sync(); // sync the model
    // also sync any extra models here
};

init();

module.exports = {
    User, // export the model
    // also export any extra models here
};
```

Routes

Now that we have our Model ready and connected, let's create API endpoints to add a user to our database and also retrieve the users from our database.

First, create a file in our **routes** folder called **userRoutes.js** using the content on the right.

Once we have our routes ready, we also need to map them to **server.js** so the application knows that such routes exist. Add this code to **server.js** (before `app.listen`):

```
let userRoutes = require('./routes/userRoutes');
app.use('/api/users', userRoutes);
```

```
const express = require("express");
const router = express.Router();
const Controllers = require("../controllers");

// matches GET requests sent to /api/users
// (the prefix from server.js)
router.get('/', (req, res) => {
   Controllers.userController.getUsers(res);
})

// matches POST requests sent to /api/users/create
router.post('/create', (req, res) => {
   Controllers.userController.createUser(req.body,
res)
})

module.exports = router;
```

Controllers

Now we are ready with our routes, we need to create the appropriate controllers to handle the business logic of these requests.

First create a file **userController.js** in a **controllers** folder, using content on the right.

Next we add the content below to an **index.js** file in the same **controllers** folder:

```
module.exports = {
    userController: require('./userController')
}
```

```
"use strict";
const Models = require("../models");

// finds all users in DB, then sends array as response
const getUsers = (res) => {
  Models.User.findAll({}).then(data => {
      res.send({result: 200 , data: data});
  }).catch(err => {
     console.log(err);
     res.send({ result: 500, error: err.message });
  })
}

// uses JSON from request body to create new user in DB
const createUser = (data, res) => {
  Models.User.create(data).then(data => {
      res.send({ result: 200 , data: data});
  }).catch(err => {
     console.log(err);
     res.send({ result: 500, error: err.message });
  })
}

module.exports = {
    getUsers, createUser
}
```

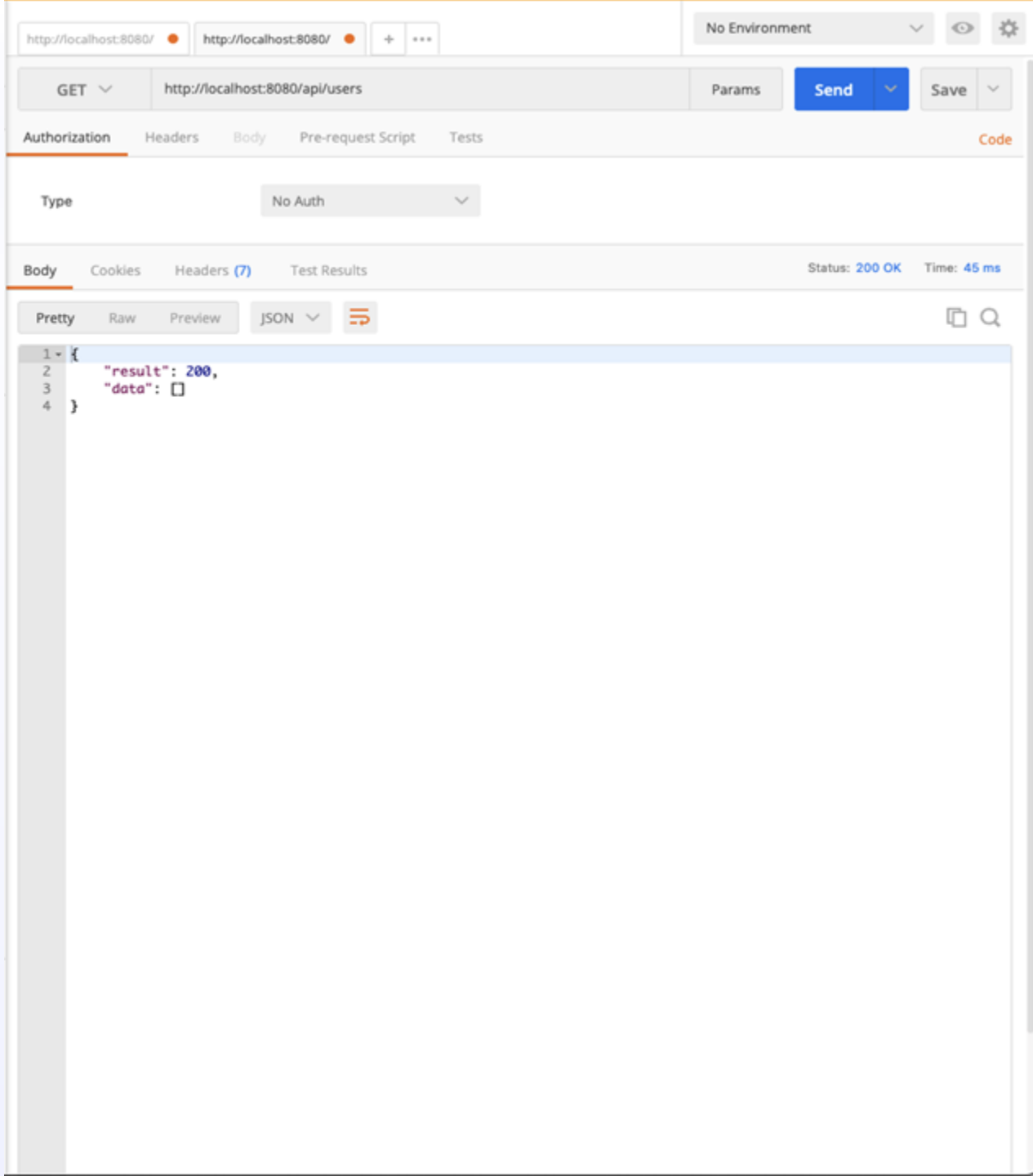
MySQL REST API

Now all our routes should be connected to their appropriate controller and we can test the APIs.

In order to test the APIs I would suggest the use of Postman, which is a google-based tool to test HTTP REST APIs. (You can also use a VSCode extension called Thunder Client.) So let's try the REST API we created. First we need to start the server using **`npm start`**.

Let's first try our **getUsers** endpoint, using the right port (8081) and route path (/api/users/). At the start the API should respond with no users, as we haven't added any yet.

If you get an error, check the application console and make sure your DB URI and route path is correct.



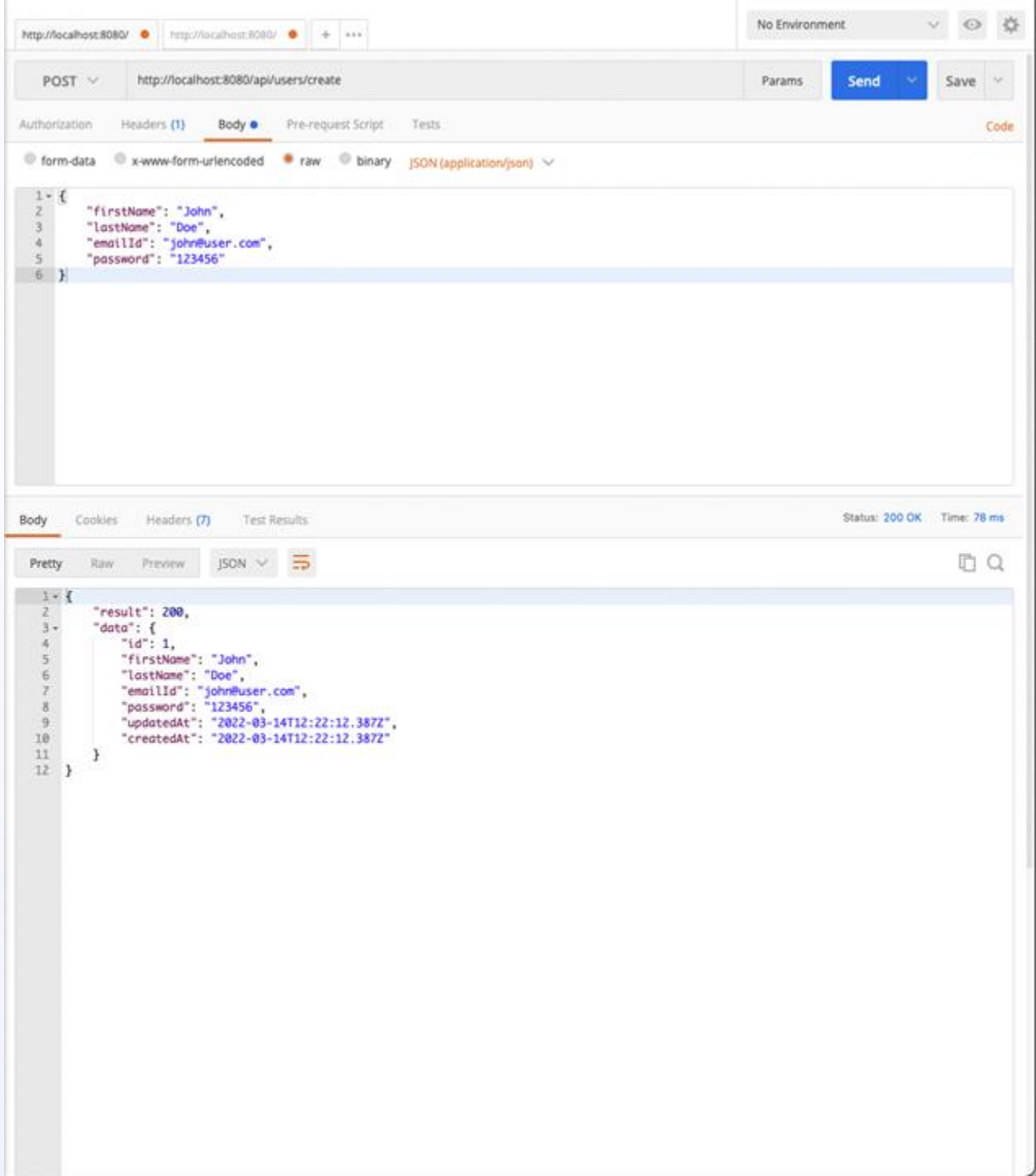
MySQL REST API

Now let us add a user to the database. For this we can use the **createUser** endpoint and send the below JSON to /api/users/create as sample data to save.

In Postman, choose the **POST** operation and copy the below to the **Body** using **raw** data, encoded as **JSON**:

```
{
        "firstName": "John",
        "lastName": "Doe",
        "emailId": "john@user.com",
        "password": "123456"
}
```

So we can see here our user was successfully added to our database.



MySQL REST API

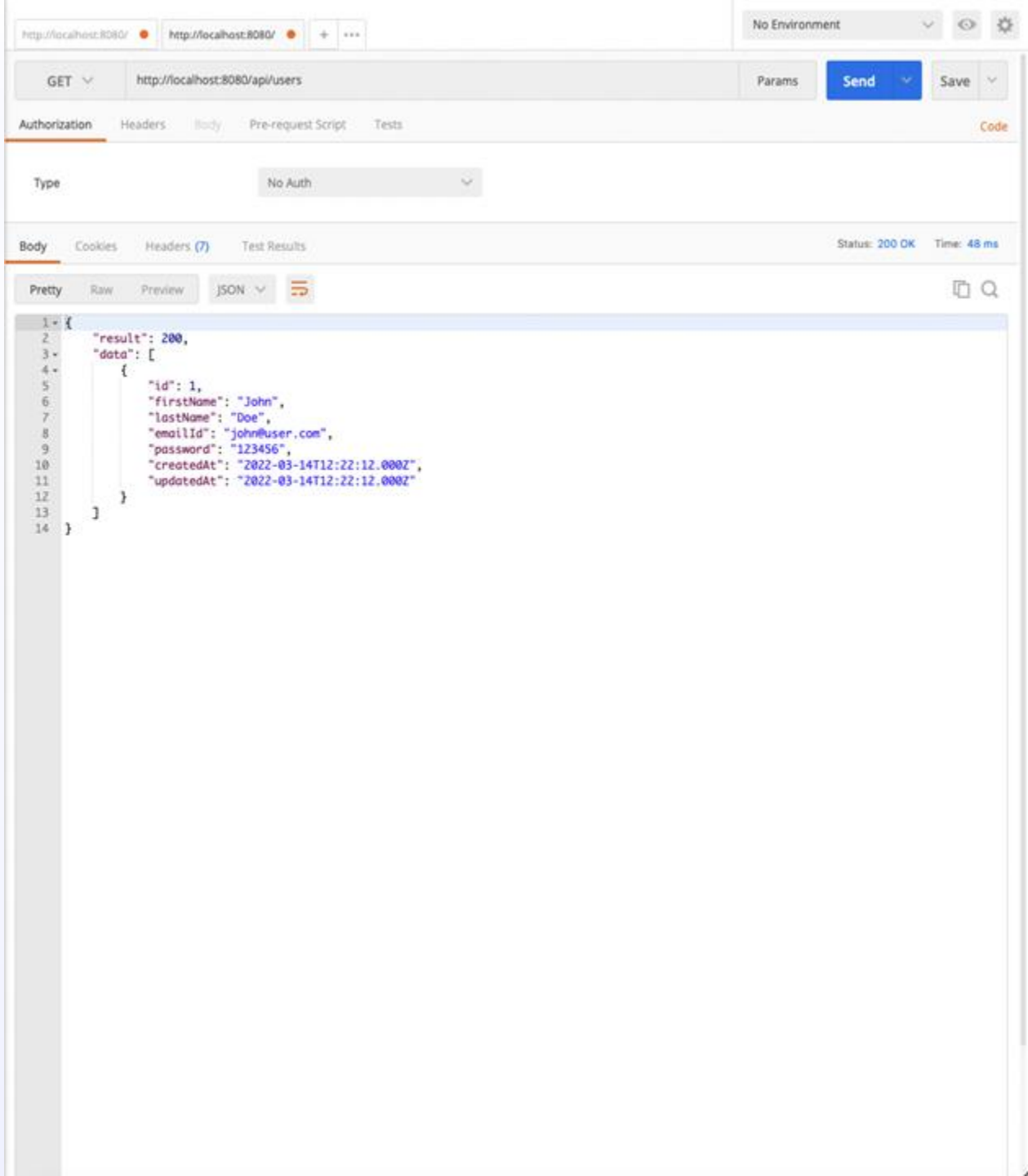
Now let's try our getUser API and verify if our user was added successfully.

From the result, you can see our user was successfully created, as the getUsers API endpoint now returns our new user.

Try adding a second user using your own details and verify that there are now two users.

Similarly we can create Update and Delete users API endpoints also (using the PUT and DELETE methods).

You can also refer to the github repo here if you face any issues regarding the code or the folder structures.



MySQL REST API

To update and delete users, use the following example code and add the below to **`userRoutes.js`** and the controller functions on the right to **`userController.js`**.

```
// matches PUT requests to /api/users/123 (stores 123 in id
param)
router.put('/:id', (req, res) => {
  Controllers.userController.updateUser(req, res)
})

// matches DELETE requests to /api/users/123 (123 in id param)
router.delete('/:id', (req, res) => {
  Controllers.userController.deleteUser(req, res)
})
```

The /:id syntax captures everything after the / and stores it in a request param of that name - in this case id.
We can therefore update users using a request URL such as http://localhost:8080/api/users/1234, where 1234 is the matching user ID which will be stored in the **`req.params.id`** property.

```
// uses JSON from request body to update user ID from params
const updateUser = (req, res) => {
  Models.User.update(req.body, { where: { id: req.params.id },
returning: true })
    .then(data => {
      res.send({ result: 200, data: data });
    }).catch((err) => {
      console.log(err);
      res.send({ result: 500, error: err.message });
    });
};

// deletes user matching ID from params
const deleteUser = (req, res) => {
  Models.User.destroy({ where: { id: req.params.id } })
    .then(data => {
      res.send({ result: 200, data: data });
    }).catch((err) => {
      console.log(err);
      res.send({ result: 500, error: err.message });
    });
};

module.exports = {
  getUsers, createUser, updateUser, deleteUser
};
// ++ Test updating and deleting a user using Postman
```

Foreign Keys

Since MySQL is designed to rely heavily on foreign keys to maintain relationships between tables, the Sequelize library also supports several ways to define and manage these associations.

Since the order of these associations is important, defining them in the `models/index.js` file **after** importing and syncing them to the database is usually recommended.

```
// Sequelize will auto-generate foreign key column names based on the table names
Post.belongsTo(User);
User.hasMany(Post);
```

See the Sequelize documentation for more information and examples. A second options argument, eg. `{ foreignKey: 'user_id' }` can be passed to these association methods to control the name of the foreign key column instead of auto-generating it, and to override various other defaults.

This technique can be used to implement the foreign keys for your blogging data model in Exercise 3.

Foreign Keys

Once one or more associations have been defined between Models, the **`include`** property in Sequelize can be used to combine data from multiple related tables in a single `findAll`.

*The value passed to **`include`** should be a reference to the associated model.*

```
const getUserPosts = (req, res) => {
    // finds all posts for a given user and includes matching user details
    Models.Post.findAll({ where: { userId: req.params.uid }, include: Models.User })
        .then((data) => res.send({ result: 200, data: data }))
        .catch((err) => {
            console.log(err);
            res.send({ result: 500, error: err.message });
        });
};
```

The above controller function will (after first setting up a Post model) find all posts for a given user id, and include both post fields and user fields in the combined result.

Exercise 3

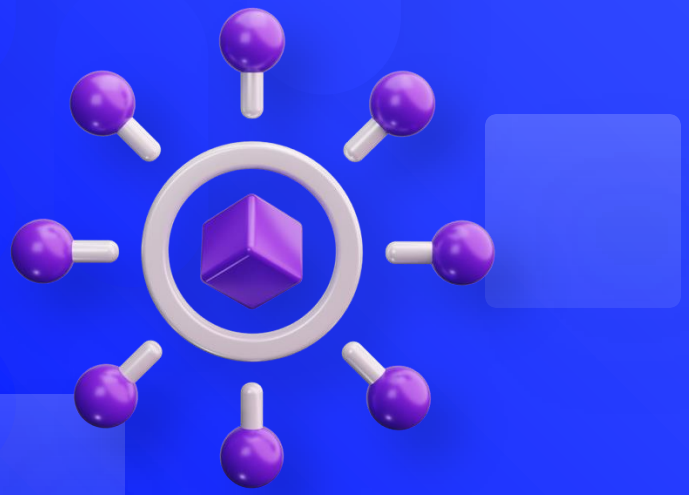
Create an express back-end application for a Blog website using MySQL. You should refer to your database model from Module 8 for this, ensuring that your app matches the model.

Requirements:

- Your system should have a proper MVC Structure
- The system should be able to create users.
- The users should be able to create multiple posts (posts should be very basic with title, description and image)
- Other users should be able to like the posts and comment on the posts.

Section 5:

Micro Services



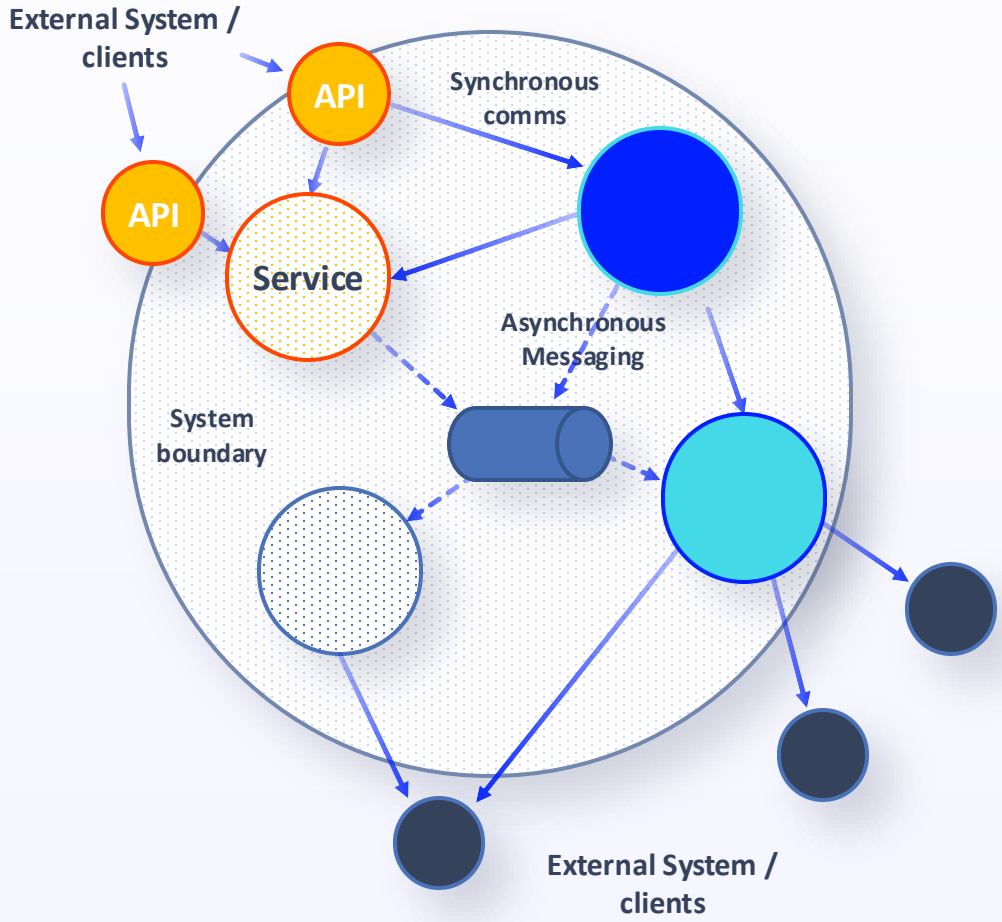
Dr. Peter Rodgers created the term microservices in 2005, and it was originally known as "micro web services." At the time, the fundamental motivation behind "micro web services" was to break up huge "monolithic" architectures into numerous independent components/processes, resulting in a more granular and manageable codebase.

Modular, distributed programs have been around for a long time. Microservices aren't a novel concept in this context. However, it was the principles guiding how microservices were developed and consumed that made them popular. Microservices used open standards such as HTTP, REST, XML, and JSON to replace proprietary communications protocols used by traditional distributed systems at the time.

Micro Services

A microservice is a distributed service that is tiny and loosely connected. It's part of a larger microservices architecture, which consists of a collection of loosely coupled microservices that work together to achieve a shared purpose. A system can be thought of as a collection of microservices.

Let's have a look at a super-simple microservices architecture reference model. It depicts a hypothetical system made up of numerous granular services that communicate synchronously (through internal API calls) or asynchronously (by message forwarding via a message broker). The deployment as a whole is contained within a fictitious system boundary. External systems (users, applications, B2B partners, and so on) can only connect with the microservices via an API gateway, which is a set of externally-facing APIs. Within the boundary, services can freely consume external services as needed.



Reasons for Building Microservices

Consider the common issues that monolithic programs face, to see why we would go down this path.

- Regardless matter how minor or significant the update is, the **complete application must be rebuilt and redeployed**. For large applications, construction times of 15–30 minutes are fairly common.
- A slight modification in one element of an application can cause the **entire system to fail**.
- As the application's size grows, its components become **increasingly entangled**, making the codebase more complex to understand and manage.
- Huge applications have a proportionally **large resource footprint**, consuming more memory and requiring more processing power. As a result, they need to be housed on big servers with plenty of resources. Their ability to scale is also hampered as a result of this.
- They also have a **long starting time**, which is inconvenient considering that even minor modifications necessitate a complete redeployment. They're not well adapted to the Cloud and can't easily use ephemeral computing, such as spot instances.

Reasons for Building Microservices

- The entire application is implemented using a **single technology**, which is typically a compromise between generality and the needs of certain application areas. Java and.Net are likely candidates for monoliths because they are among the greatest "all-rounder" languages, rather than because they excel at specific tasks.
- A huge codebase automatically **limits the scalability of a team**. The more complicated the program (in terms of internal dependencies), the more difficult it is to accommodate large teams of developers comfortably without tripping on each other's toes.

Benefits of Microservices

Scalability: In a microservices design, individual operations can scale to meet their demands. Smaller applications can scale horizontally (by adding more instances) and vertically (by increasing the resources available to each instance).

Modularity: The physical isolation between processes encourages you to handle coupling at the forefront of your design, which is an obvious benefit of smaller, independent applications. Each application is responsible for fewer duties, resulting in a code that is more compact and cohesive. Monoliths can (and should) be developed in a modular form, with coupling and coherence in mind; but, because monoliths are in-process, developers are free to short-circuit "soft" boundaries within the program and violate encapsulation, frequently without realising it.

Tech Diversity: Microservices are self-contained apps that communicate via open protocols. In comparison to a monolith, the technology choices underpinning microservices implementations are important for the microservice in question, but not for the rest of the system. A single microservices architecture is frequently built using a mix of technologies, such as Java and Go for business logic, Node.js for API gateways and presentation concerns, and Python for reporting and analytics.

Benefits of Microservices

Opportunities for Experimentation: A microservice is self-contained and can be designed and developed independently of its peers. It will have a different database from the rest. It can be written in a language that is most appropriate for the task at hand. This autonomy allows the team to safely experiment with new technologies, approaches, and procedures while being constrained to a single service. If one of the experiments fails, the mitigation costs are generally low.

Eases Migration: Monolithic software systems can often be based on decades-old technology because they are extremely difficult and dangerous to update, due to complex library dependencies, a lack of sufficient unit tests, and the accompanying migration risk. Smaller microservice codebases are much easier to refactor, and even badly designed individual microservices don't slow down the system as a whole.

Resilience and availability: When a monolith fails, the business comes to a halt. Of course, the same might be said for microservices that are poorly built, tightly connected, and have complicated interdependencies. Good microservices architecture emphasises loose coupling, with services that are self-contained, completely own their dependencies, and avoid synchronous (blocking) communication. When a microservice fails, it will disrupt some portions of the system and certain users, but usually allow other elements of the system to continue to function.

Limitations of Microservices

Atomic Versioning: Since the codebase, together with any related tags and branches, is stored in a single repository, versioning a monolith is rather simple. You may be quite certain that all components are compatible and can be safely deployed together when you check out a version. Microservices are often developed independently and stored in their own repositories. They must, however, communicate. When services are not co-versioned, keeping track of versions and ensuring compatibility becomes much more difficult. The same issues that plague code also plague configuration.

Deployment Automation: When you're deploying one application to a pair of servers once a month, you might be able to get away with manually moving WAR or EAR files to an Application Server in a data centre. Microservice architectures will suffer as a result of this approach. Manual processes will not suffice when your team manages a fleet of several dozen microservices that are constantly changing. Microservices necessitate a well-developed DevOps concept, as well as CI/CD methods and infrastructure.

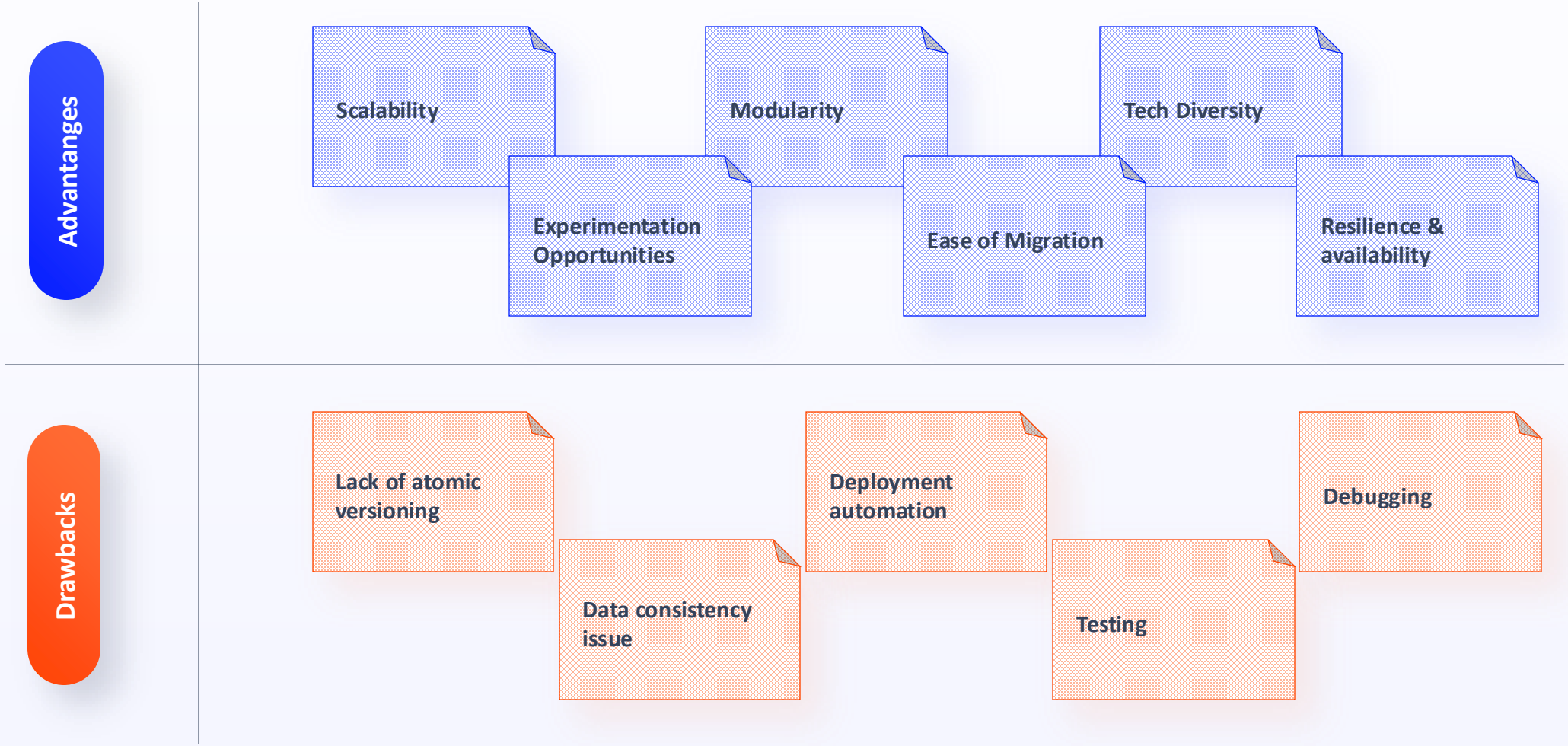
Limitations of Microservices

Debugging: When components of a system communicate in the same process, it's much easier to debug their interactions, especially when one component merely calls a method on another. Attaching a debugger to the process, stepping through the method calls, and observing variables is usually all that is required. In microservices, there is no simple comparable. Tracing and piecing together log messages is challenging, necessitating additional infrastructure and complexity.

Data Consistency: ACID is a feature of a monolith that runs on a single relational database. To put it another way, transactions are pure and reliable. In distributed systems, consistency is more difficult to ensure in general, particularly considering the types of errors that occur in distributed systems.

Testing: A monolith may be easily tested as a whole system, either manually or via an automated test suite (which is preferred). Testing a single microservice does not provide a complete picture: just because a service meets its requirements does not mean the entire system will perform as expected. Running more comprehensive integration and end-to-end (or acceptance) tests is the only way to be sure.

Flashback on Microservices



Exercise 4

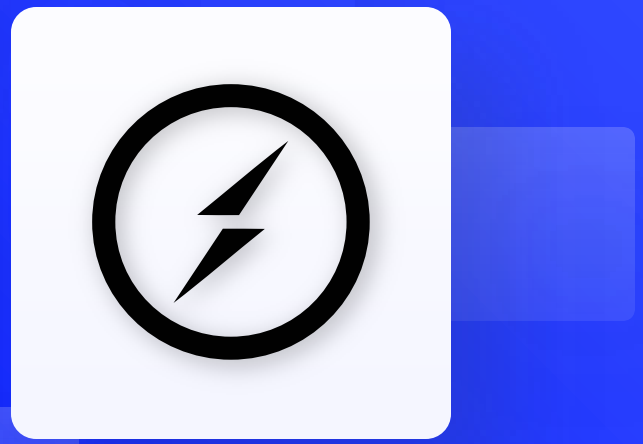
Choose a third-party microservice (see the list of free JSON APIs) and connect it to your current express application (or a new one) with its own routes and controller, to add new functionality to your application.

Try to include support for either or both types of request parameters:

- via the query - eg. GET http://localhost:8080/api/users?page=1
- via the params - eg. PUT http://localhost:8080/api/users/1234

Section 6:

Sockets



While it is a wrapper around WebSockets for Node.js, **Socket.io** is a Library for connecting a Client(s) to a Server utilising a Client/Server Architecture. It is incredibly easy and simple to use, especially when dealing with chat messages or real-time data.

A socket is a single connection between a client and a server that allows both the client and the server to send and receive data at the same time. Because the Socket.io library is an event-driven system, it emits and listens for multiple custom events to be triggered.

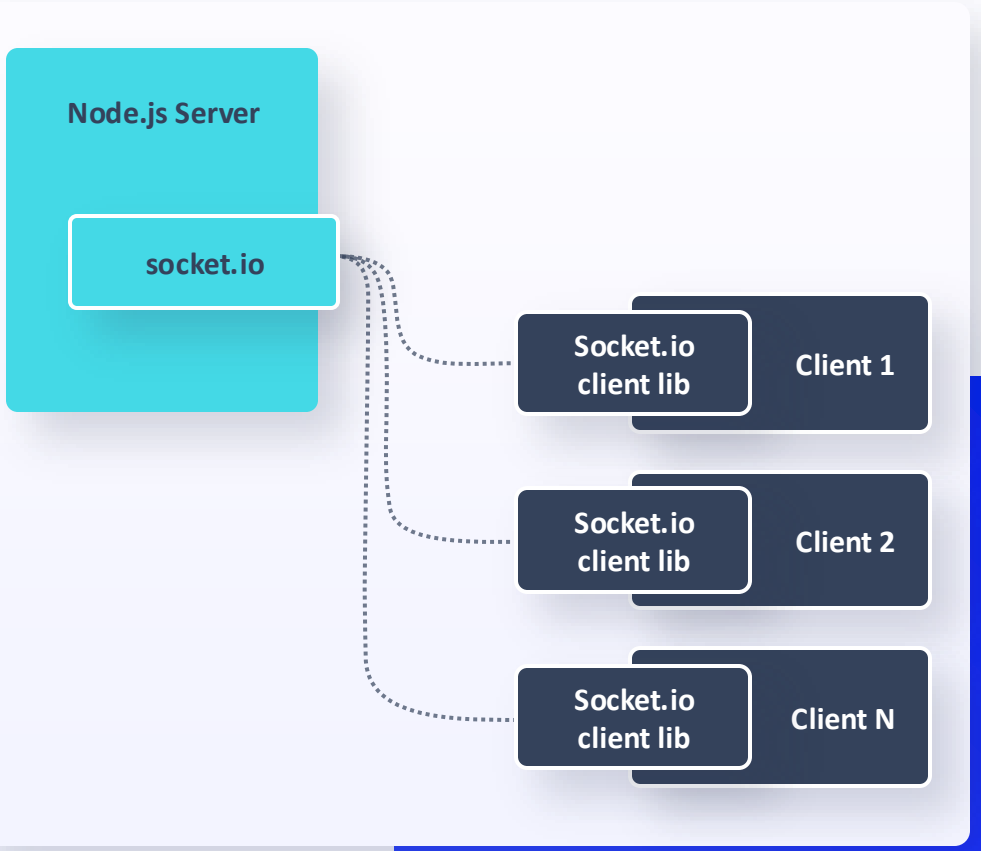
Do have a look at it in more detail here

Sockets

From the architecture diagram, we can understand that the node.js server hosts one instance of the socket.io server-side package, and every client hosts its own instance of the socket.io client. So the server does not need multiple hosts to handle multiple clients.

This means that it is a **one-to-many relation**, where one server-side socket.io can handle multiple client-side socket connections.

The client, however, can only listen to one server-side socket connection. To communicate with other clients, all requests must first be sent back to the server.



Adding Sockets to Nodejs Application

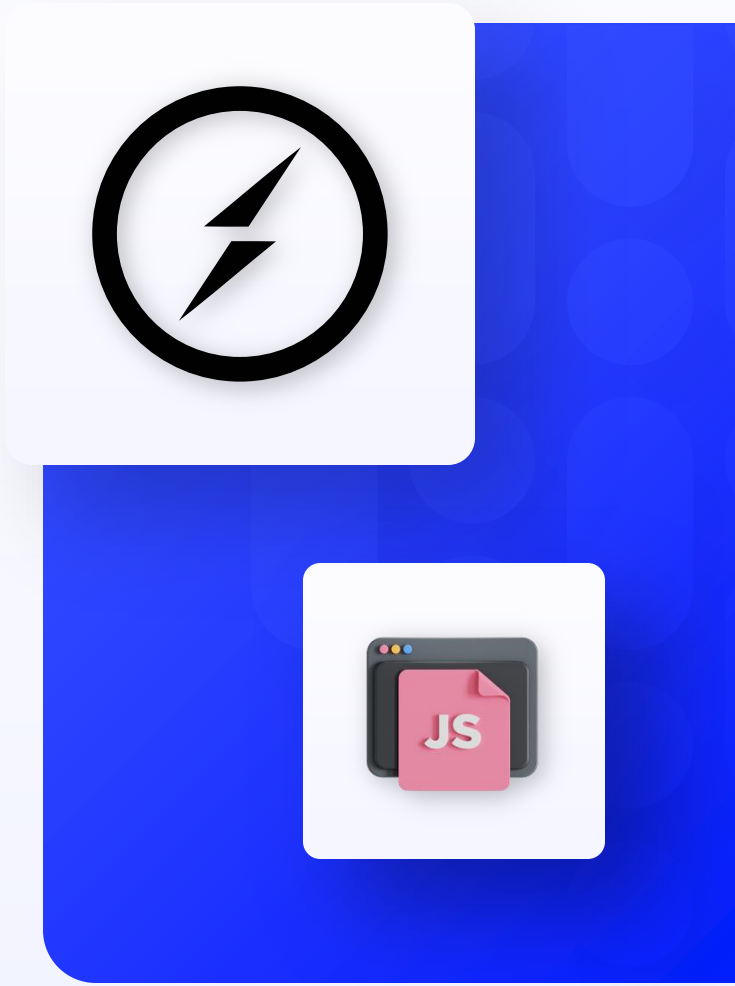
To create a socket.io application, first create a new express app in a new folder (eg. **`socket-chat-app`**) with **`index.js`** as the main file, then run **`npm init`** to initialise it.

We then need to install both **express** and the **socket.io** npm package to our application so that we have a server-side socket ready to use.

```
npm install express
npm install socket.io --save
```

Now that we have the socket.io package installed, let's see what we need to add in our application in order to connect it to a client-side application.

More info can be found at Socket.IO



Socket.io Server

Use the **index.js** code on the right to get started.

First we create an express app and a http server. By requiring the socket.io library in our application, we create a server that listens whenever a user connects to it and can send messages to all registered clients (sockets).

We then serve a static index.html file on the default endpoint, and create our first event called 'connection'. When the server receives an event (with the 'on' function) with this name from **one** client, it sends out (emits) an event to **all** clients so that they know what has happened.

A point to notice is that the socket library works on the basis of **events**. The data is always sent on event name and read using event names on the client side. This gives the ability to listen to different event names and perform different actions on the client side using those custom event names.

```
const express = require('express');
const app = express();
const http = require('http');
const { Server } = require("socket.io");

const server = http.createServer(app);
const io = new Server(server);

app.get('/', (req, res) => {
    res.sendFile(__dirname + '/index.html');
});

io.on('connection', (socket) => {
    io.emit('connection', 'a user connected');
});

server.listen(3000, () => {
  console.log('listening on *:3000');
});
```

Socket.io Client

Now that we have a socket server, let's add the socket library to our **client** side.

For this we need an **index.html** file, which will be loaded in each client browser, potentially many times, and act as the **client** application. To link with our socket **server**, we need to add the socket library to our **index.html** file by including the **socket.io.js** file.

We can then use the **io()** function exposed by this script and begin to use it to listen for events from the server (with the 'on' function) and send events back (with the 'emit' function).

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Socket Chat App</title>
</head>
<body>
    <h2>Welcome to Socket Chat</h2>
    <script src="/socket.io/socket.io.js"></script>
    <script>
      let socket = io();
      socket.on('connection', msg => console.log(msg))
    </script>
</body>
</html>
```

Socket.io Client

This piece of code in **`index.html`** tells our client to connect to our socket server, and to listen for the 'connection' event. When it happens, it will run the arrow function which takes the data sent from the server and simply logs it to the console (but instead we could display it on the page using the JS DOM functions):

```
let socket = io();
socket.on('connection', msg => console.log(msg))
```

Now that we have added code to both our server and client side, let's see if our application behaves as we expect.

To do this, we need to start our application using **npm start**.

(First make sure you have a 'start' script in package.json.)

Socket.io Express Application

```
PS D:\INSTITUTE OF DATA\SOFTWARE ENGINEERING LABS\MODULE 9\socket-chat-app> npm start
npm WARN config global `--global`, `--local` are deprecated. Use `--location=global` instead.

> socket-chat-app@1.0.0 start
> node index.js

listening on *:3000
```

Once the server is running, open http://localhost:3000/ in a browser and open the Dev Inspector. The console should include our 'a user connected' message for each client that opens this URL and connects to our server.

Try opening multiple browser tabs to the same URL and watching the console in the first one as each new client connects.

Socket.io Express Application

We can do much more interesting things between the server and its clients using this same structure.

A few important things to note:

- The server can emit (send) events in **three** possible ways:
  1. To **all** possible clients, using **`io.emit`**
  2. To **only** the socket/client that sent the event, using **`socket.emit`**
  3. To every socket/client **except** the one that sent the event, using **`socket.broadcast.emit`**.
- Each **`emit`** function takes two parameters: the first is a **string** indicating the name of the event, and the second is the **`data`** which should be sent. It can be a string, a number or an object containing multiple properties.
- The client can emit events in only one way - using **`socket.emit`** to send the name of the event and any associated data back to the server
- Clients also emit a special 'disconnect' event which automatically fires when the browser closes or refreshes and causes the connection to break.

Exercise 5

Using the guide at Socket.IO as a helper, try to implement a basic chat app which includes one of their suggested extensions (or come up with your own!):

- Broadcast a message to connected users when someone connects or disconnects.
- Add support for nicknames.
- Don’t send the same message to the user that sent it. Instead, append the message directly as soon as he/she presses enter.
- Add “{user} is typing” functionality.
- Show who’s online.