Institute of Data

2022

Software Engineering

Module 9

API Development

Institute of Data

Agenda

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 REST?

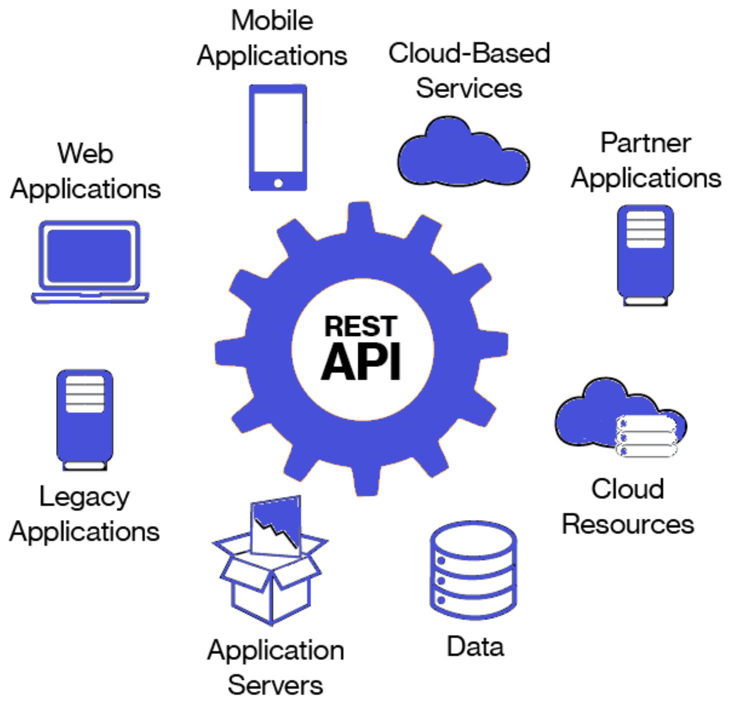
Representational State Transfer (REST) is an acronym for Representational State Transfer. It's a design style for creating networked applications (i.e apps that use some form a network to communicate). It is the most often used method for creating web APIs. When a REST API is developed, it follows a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

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

Stateless - The 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. Following the server's processing of the request, 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.

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

Principles of REST API

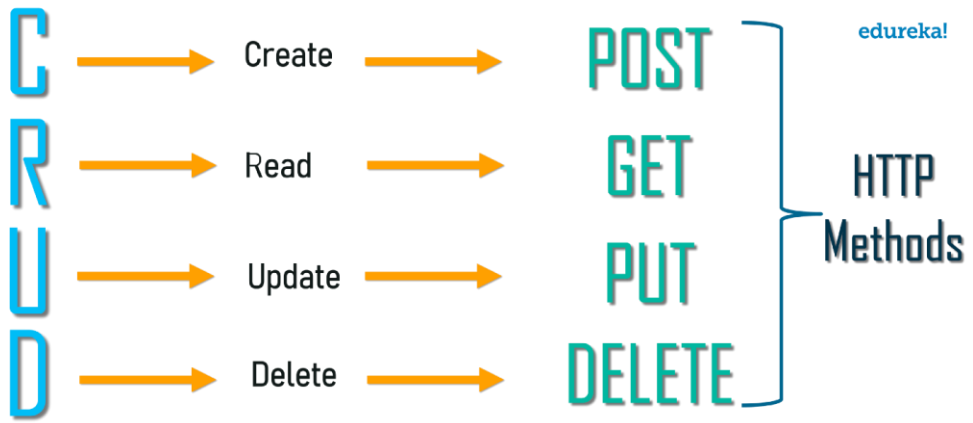
Cacheable - Applications are frequently made cacheable in order to improve performance. This is accomplished by either implicitly or explicitly identifying the server's answer as cacheable or non-cacheable. If the response is cacheable, the client cache can reuse the response data in the future for similar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

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 API

All of us who work with web technologies perform CRUD activities. When I mention CRUD operations, I'm referring to the 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.



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.

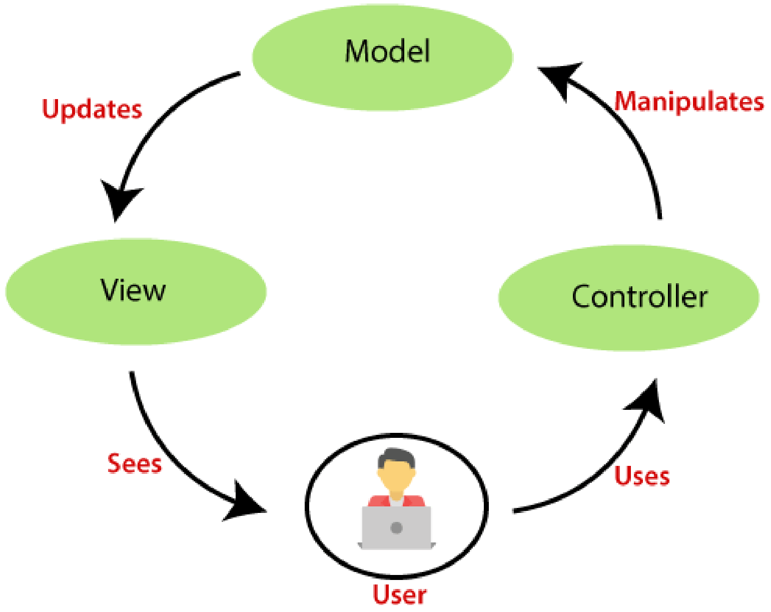
MVC architecture divides the whole application into three parts; the Model, the View and the Controller.

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 Structures Express App

Now let's try to understand why utilising an MVC 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). You should modify your current applications to 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 Structures Express App

Now that we have the MVC Structure ready lets understand what goes in which folder 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 is the applications main purpose is to serve as a server side application and is eventually going to be connected to another standalone frontend application.

Controller: Controllers are the parts that are responsible to handle the business logic of our application. When we call an endpoint from the user we send a bunch of data with that end point what needs to be done with that data is handle by our controllers. So when ever we create a route file we also create a corresponding controller file which handles all the business logics 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

Try modifying the current express applications that you have and try modifying it to a proper MVC structure.

Section 3: REST API using MongoDB

So in previous module you must have already created a very basic Rest API which has a route that takes some input from the user and then performs a business logic on that data which is written in the controller.

In this section we will do something similar but instead we would try to connect it with a database and perform a change in data.

For this section as we are using MongoDB we would 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.

Connect Using Mongoose

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

Now that we have mongodb running in background we need to install the mongoose npm in our application. For

```
> npm install mongoose --save
```

Now that we have mongoose install lets connect to our mongodb. For doing that we create a file called dbConnect.js in out root folder.

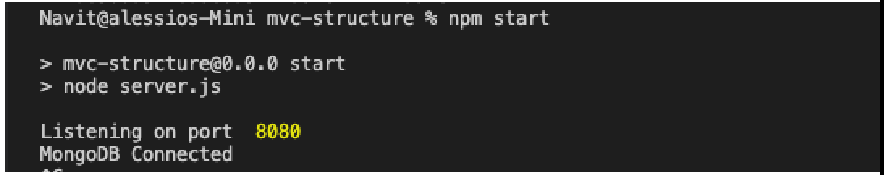
The contents of dbConnect.js should look something like this

Once we have written this file we need to import

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

```
'use strict';
var Mongoose = require('mongoose');

const uri = process.env.uri ||
"mongodb://localhost/myFirstDatabase";

const mongooseOptions = {
 useNewUrlParser: true,
 useUnifiedTopology: true
};
Mongoose.set('useCreateIndex', true);
Mongoose.set('useFindAndModify', false);



//Connect to MongoDB
Mongoose.connect(uri, mongooseOptions, function (err) {
    if (err) {
        console.log("DB Error: ", err);
        process.exit(1);
    } else {
        console.log('MongoDB Connected');
    }
});
exports.Mongoose = Mongoose;
```

No if we start our server we should get something like this in our console of application.



```
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. For doing that in our models folder we create a file called users.js and add this code to that file. Next we add this file to our index.js file in our models folder.

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

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

var user = 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", user);
```

REST API using Mongoose

Now that we have our Model ready and connected lets create an API to add a user to our database and also retrieve the users from our database.

For doing that lets create a in our routes folder called userRoute.js. Now that we have our rotes ready we also need to add it to our server.js file so the application knows that such routes exist for doing this we add this code to our server.js

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

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


router.get('/', (req, res) => {
   Controllers.userController.getUsers(res);
})

router.post('/create', (req, res) => {
   Controllers.userController.createUsers(req.body, res)
})


module.exports = router;
```

REST API using Mongoose

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

For doing that lets create a file called userController.js in our controllers folder.Next we add this file to our index.js file in our controllers folder.

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

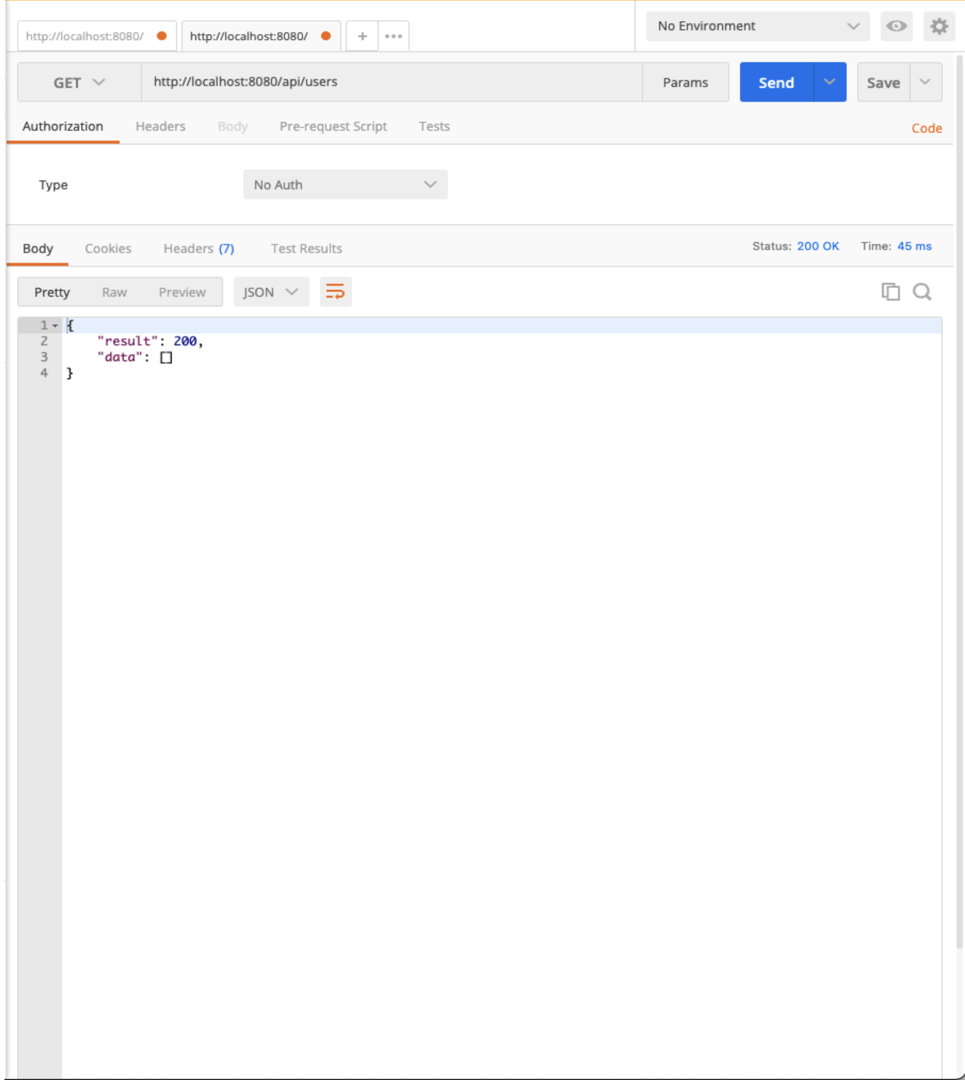
```
"use strict";

var Models = require("../models");

const getUsers = (res) => {
   Models.Users.find({}, {}, {}, (err,data) => {
       if (err) throw err;
       res.send({result: 200 , data: data})
   });
}

const createUsers = (data, res) => {
   new Models.Users(data).save((err,data) => {
       if(err) throw err
       res.send({ result: 200 , data: data})
   });
}

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

REST API using Mongoose

Now all our routes should be connected to their appropriate controller and we could test the API's.

In Order to test the api's i would suggest the use of Postman. Which is a google based tool to test HTTP Rest Api's. So lets try our Rest Api's created. First we need to start the server using npm start.

Let's First try our getUsers Api at start the api should respond with no users as we haven't added any yet



GET http://localhost:8080/api/users

Type: No Auth

Status: 200 OK Time: 45 ms

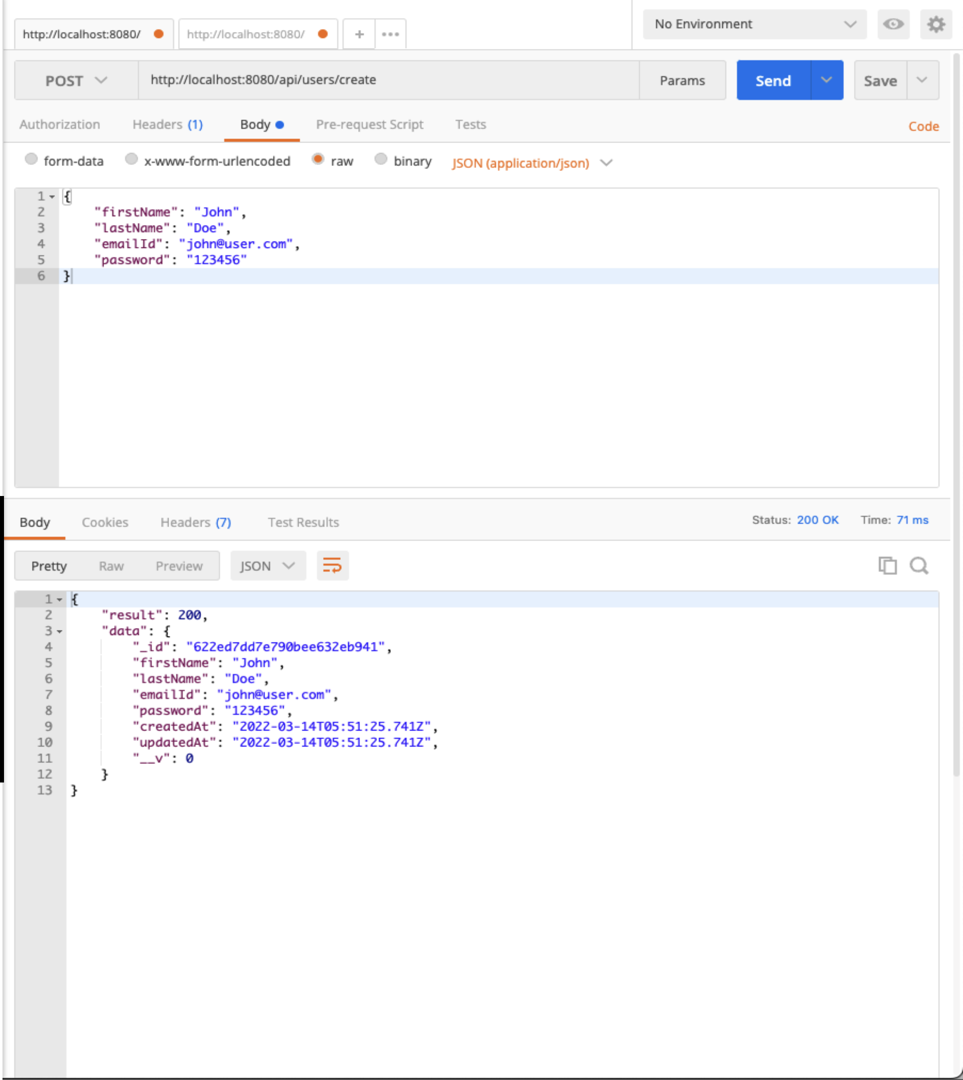
```
{
    "result": 200,
    "data": []
}
```

REST API using Mongoose

Now let us add a user to the database for this we would use the createUser API and we can use this as a sample data to save.

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

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



POST http://localhost:8080/api/users/create

Body: raw, JSON (application/json)

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

Body — Status: 200 OK Time: 71 ms — JSON

```
{
    "result": 200,
    "data": {
        "_id": "622ed7dd7e790bee632eb941",
        "firstName": "John",
        "lastName": "Doe",
        "emailId": "john@user.com",
        "password": "123456",
        "createdAt": "2022-03-14T05:51:25.741Z",
        "updatedAt": "2022-03-14T05:51:25.741Z",
        "__v": 0
    }
}
```

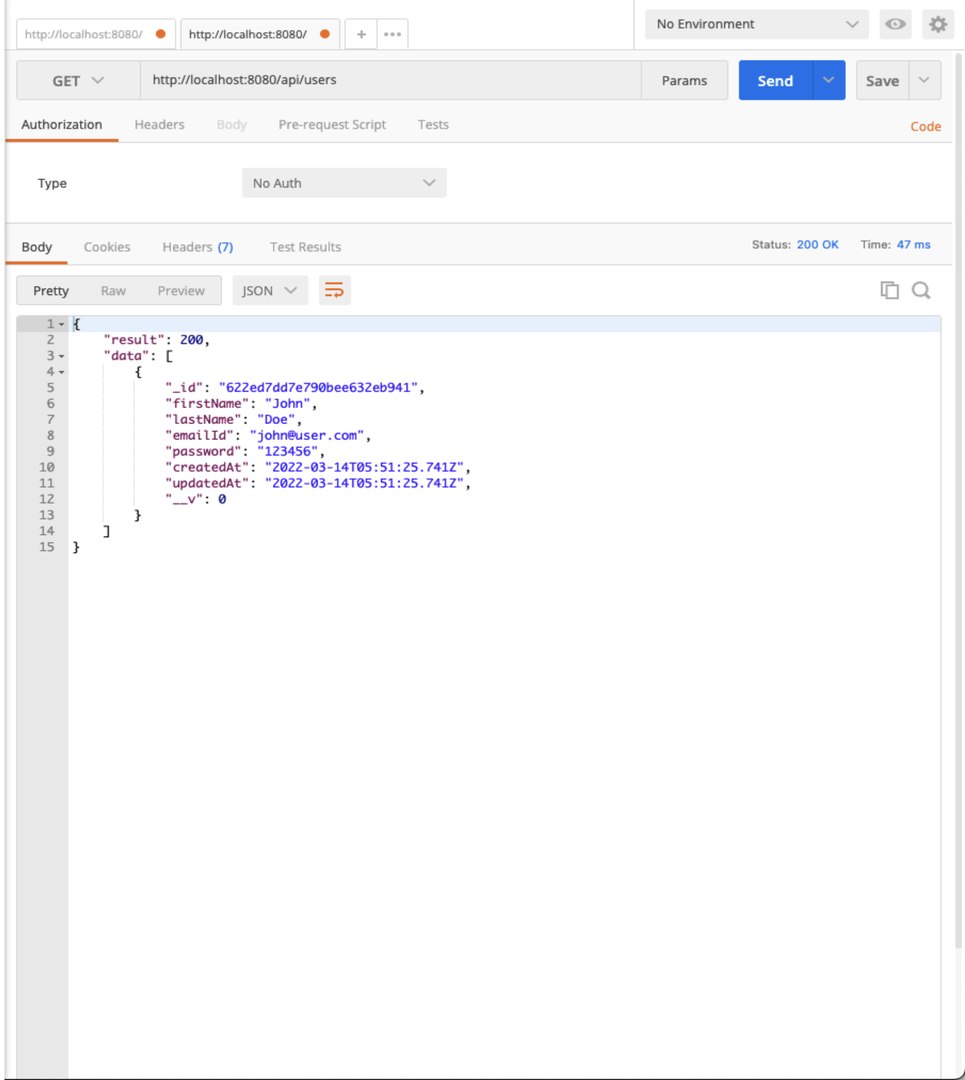
REST API using Mongoose

Now lets try our getUser api and cross verify if our user was added successfully.

From the result you can see our user was successfully created and the getUsers api also return our user.

Similarly we can also create Update and Delete users api also.

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



GET http://localhost:8080/api/users

Type: No Auth

Status: 200 OK Time: 47 ms

```
{
    "result": 200,
    "data": [
        {
            "_id": "622ed7dd7e790bee632eb941",
            "firstName": "John",
            "lastName": "Doe",
            "emailId": "john@user.com",
            "password": "123456",
            "createdAt": "2022-03-14T05:51:25.741Z",
            "updatedAt": "2022-03-14T05:51:25.741Z",
            "__v": 0
        }
    ]
}
```

Exercise 2

Try creating a express application for a Blog website using MongoDB. You can also refer to Module 8 for this.

Requirements

- You system should have a proper MVC Structure
- The system should be able to create Users.
- The users should be able to create multiple posts (Post 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

So in previous module you must have already created a very basic Rest API which has a route that takes some input from the user and then performs a business logic on that data which is written in the controller.

In this section we will do something similar but instead we would try to connect it with a database and perform a change in data.

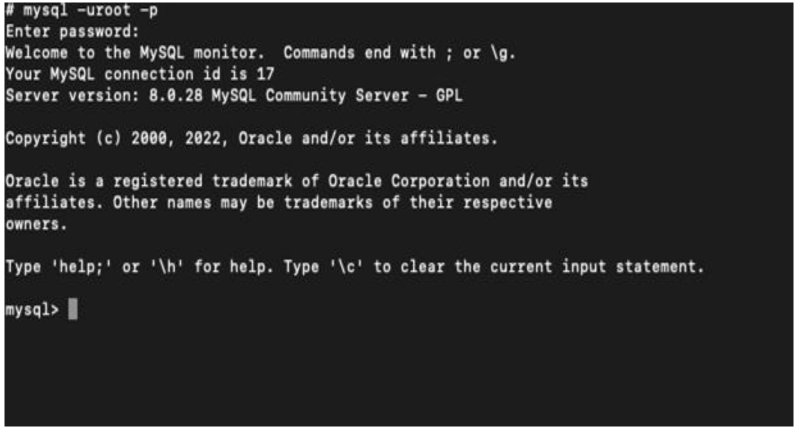
For this section as we are using MySQL we would be using a npm package called Sequelize to connect with MySQL through our Nodejs Application.

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

Prerequisite for using MySQL with Express

So before me connect to express application to connect with MySQL. We need to run the MySQL in terminal and actually create a database that we can use. This is one step that is very different from using MongoDB.

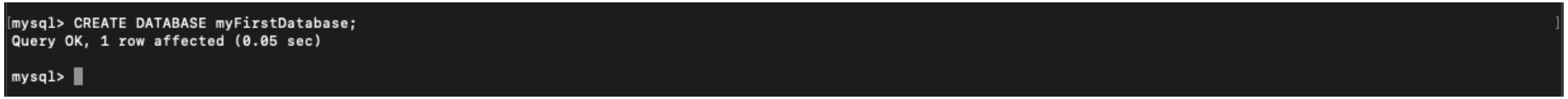
You should be having mysql service running in one of your terminal. It would look something like this:



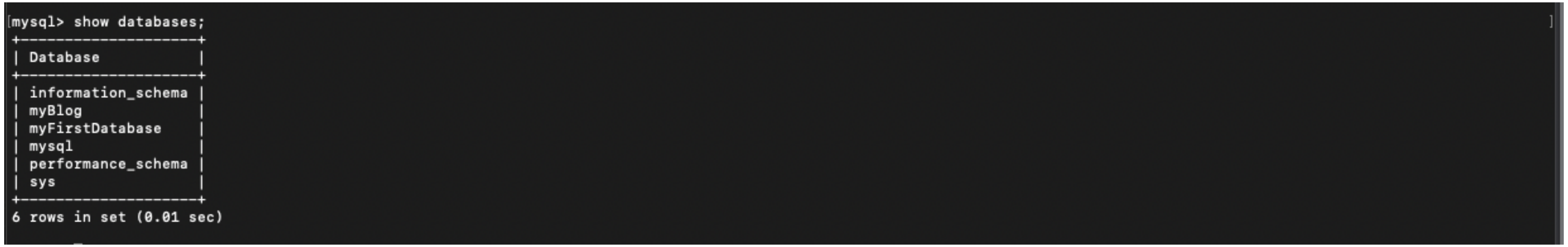
```
# mysql -uroot -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 17
Server version: 8.0.28 MySQL Community Server - GPL

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> 
```

Prerequisite for using MySQL with Express

```
> CREATE DATABASE myFirstDatabase;
```



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

mysql> 
```



```
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 background and our database created we need to install the sequelize and mysql2 npm in our application. For doing so we just need the command

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

Now that we have sequelize and mysql2 installed lets connect to our mysql. For doing that we create a file called dbConnect.js in out root folder.

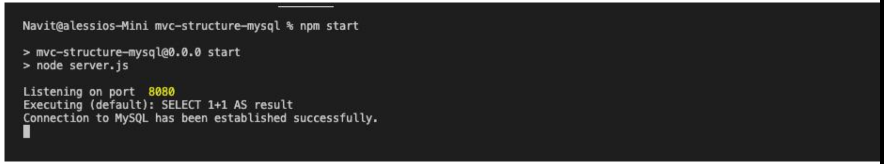
The contents of dbConnect.js should look something like this

Once we have written this file we need to import this file in our server.js file. To do that we simply add

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

dbConnect.connectMysql()
```

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

const sequelize = new Sequelize('myFirstDatabase', 'root',
'root', {
   host: 'localhost',
   dialect: 'mysql'
 });

const connectMysql = async () => {
   try {
       await sequelize.authenticate();
       console.log('Connection to MySQL has been established
successfully.');
     } catch (error) {
       console.error('Unable to connect to the MySQL database:',
error);
       process.exit(1);
     }
}

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

No if we start our server we should get something like this in our console of application.



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

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

Listening on port  8080
Executing (default): SELECT 1+1 AS result
Connection to MySQL has been established successfully.
```

Now let's create our first model. For doing that in our models folder we create a file called users.js and add this code to that file.

```
/**
* Created by Navit
*/
const { DataTypes, Model } = require("sequelize");
let dbConnect = require("../dbConnect");
const sequelizeInstance  = dbConnect.Sequelize;
class user extends Model {}
user.init({
     firstName: { type: DataTypes.STRING, allowNull: false,
required: true },
     lastName: { type: DataTypes.STRING, allowNull: false,
required: true },
     emailId: { type: DataTypes.STRING, allowNull: false,
required: true, unique: true },
     password: { type: DataTypes.STRING, allowNull: false,
required: true}
}, {sequelize: sequelizeInstance, modelName: 'user', timestamps:
true, freezeTableName: true})
module.exports = user;
```

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

```
/**
* Created by Navit
*/
'use strict'
const Users = require('./users')
async function init () {
  await Users.sync();
};
init();
module.exports = {
  Users
};
```

REST API using Sequelize

Now that we have our Model ready and connected lets create an API to add a user to our database and also retrieve the users from our database.

For doing that lets create a in our routes folder called userRoute.js. Now that we have our rotes ready we also need to add it to our server.js file so the application knows that such routes exist for doing this we add this code to our server.js

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

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


router.get('/', (req, res) => {
   Controllers.userController.getUsers(res);
})

router.post('/create', (req, res) => {
   Controllers.userController.createUsers(req.body, res)
})


module.exports = router;
```

REST API using Sequelize

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

For doing that lets create a file called userController.js in our controllers folder.Next we add this file to our index.js file in our controllers folder.

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

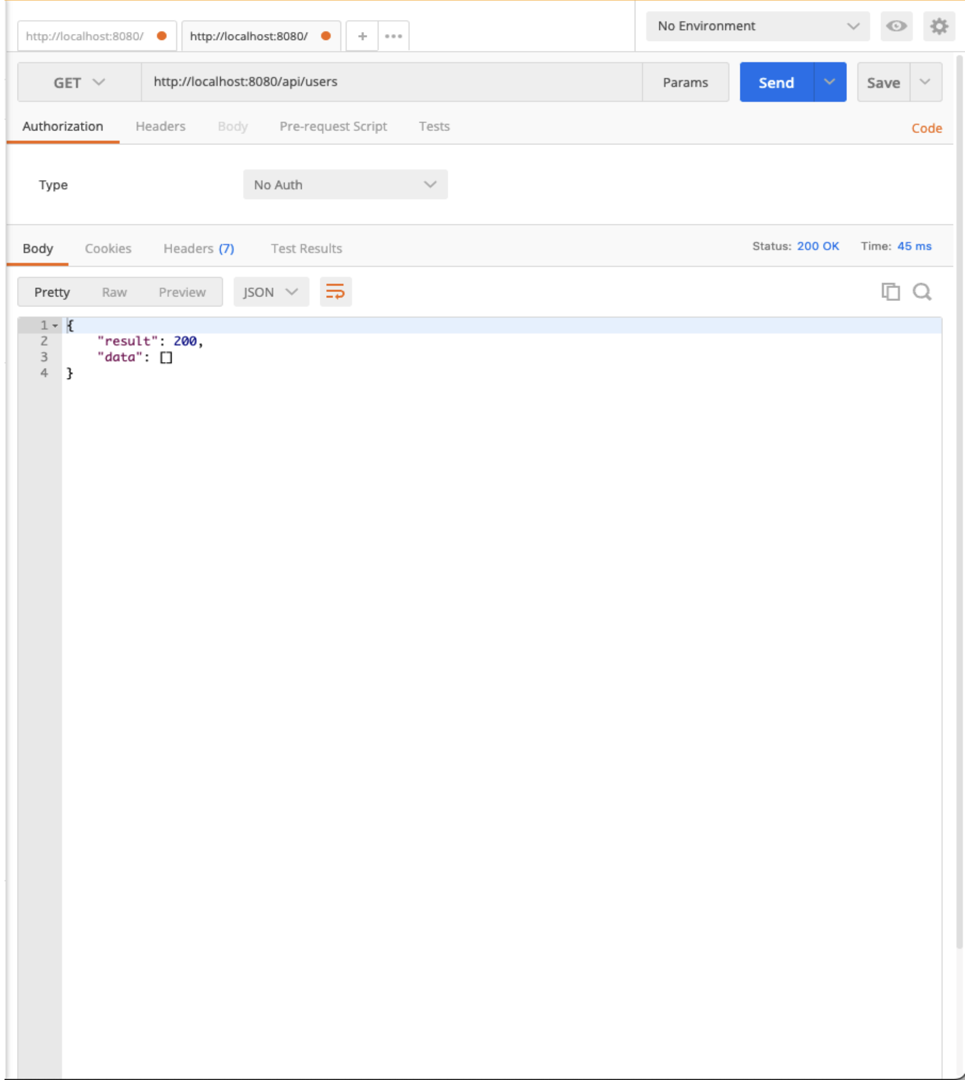
```
"use strict";

var Models = require("../models");

const getUsers = (res) => {
   Models.Users.findAll({}).then(function (data) {
       res.send({result: 200 , data: data})
   }).catch(err => {
       throw err
   })
}

const createUsers = (data, res) => {
   Models.Users.create(data).then(function (data) {
       res.send({ result: 200 , data: data})
   }).catch(err => {
       throw err
   })
}

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

REST API using Sequelize

Now all our routes should be connected to their appropriate controller and we could test the API's.

In Order to test the api's i would suggest the use of Postman. Which is a google based tool to test HTTP Rest Api's. So lets try our Rest Api's created. First we need to start the server using npm start.

Let's First try our getUsers Api at start the api should respond with no users as we haven't added any yet



GET http://localhost:8080/api/users

Type: No Auth

Status: 200 OK Time: 45 ms

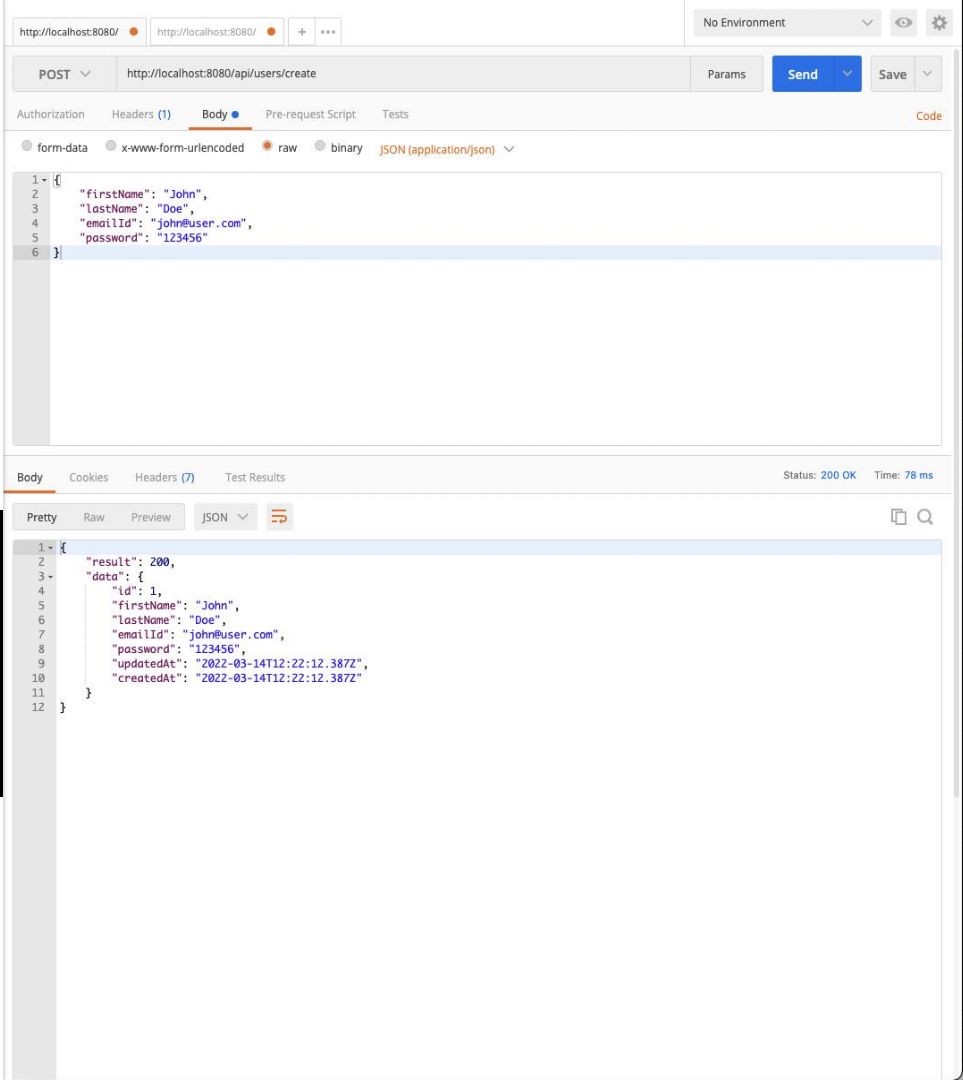
```
{
    "result": 200,
    "data": []
}
```

REST API using Sequelize

Now let us add a user to the database for this we would use the createUser API and we can use this as a sample data to save.

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

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



POST http://localhost:8080/api/users/create

JSON (application/json)

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

Status: 200 OK Time: 78 ms

```
{
    "result": 200,
    "data": {
        "id": 1,
        "firstName": "John",
        "lastName": "Doe",
        "emailId": "john@user.com",
        "password": "123456",
        "updatedAt": "2022-03-14T12:22:12.387Z",
        "createdAt": "2022-03-14T12:22:12.387Z"
    }
}
```

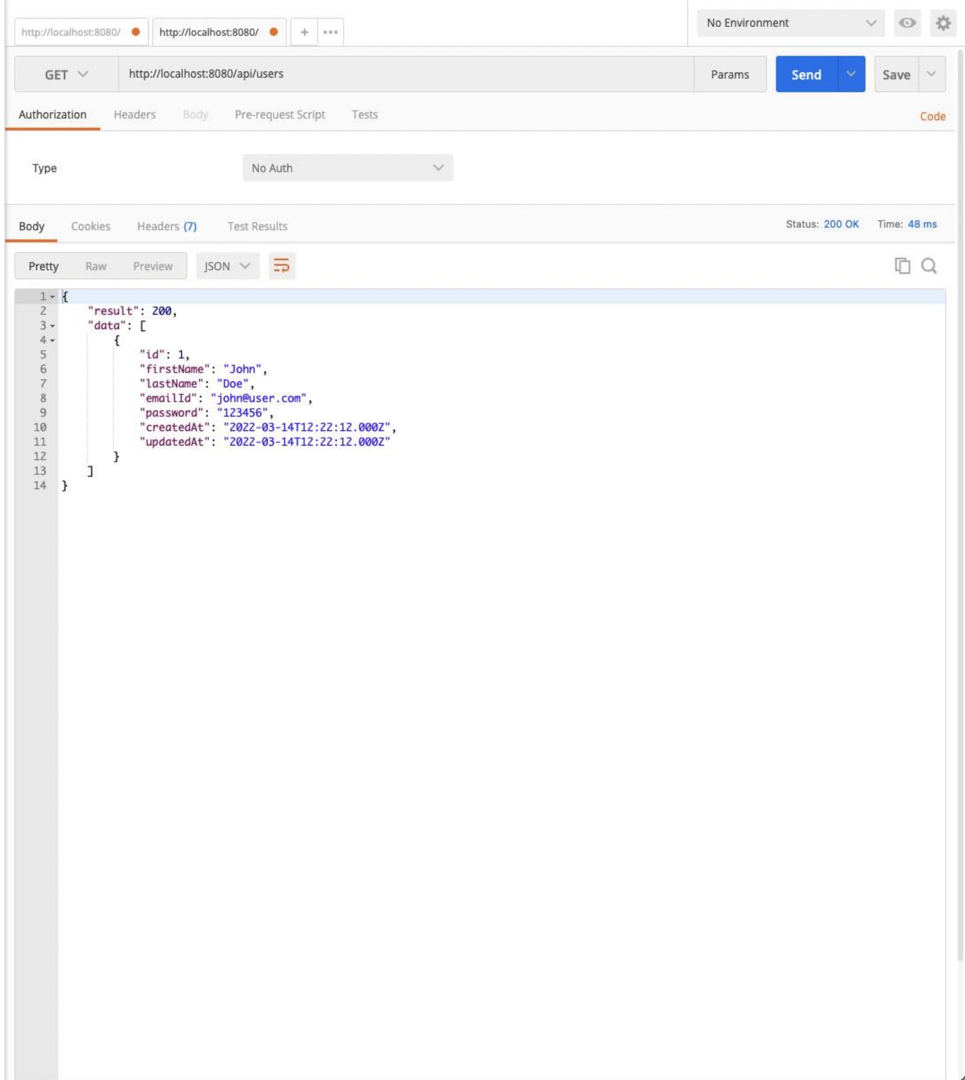
REST API using Sequelize

Now lets try our getUser api and cross verify if our user was added successfully.

From the result you can see our user was successfully created and the getUsers api also return our user.

Similarly we can also create Update and Delete users api also.

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



```
{
    "result": 200,
    "data": [
        {
            "id": 1,
            "firstName": "John",
            "lastName": "Doe",
            "emailId": "john@user.com",
            "password": "123456",
            "createdAt": "2022-03-14T12:22:12.000Z",
            "updatedAt": "2022-03-14T12:22:12.000Z"
        }
    ]
}
```

Exercise 3

Try creating a express application for a Blog website using Sequelize. You can also refer to Module 8 for this.

Requirements

- You system should have a proper MVC Structure
- The system should be able to create Users.
- The users should be able to create multiple posts (Post 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

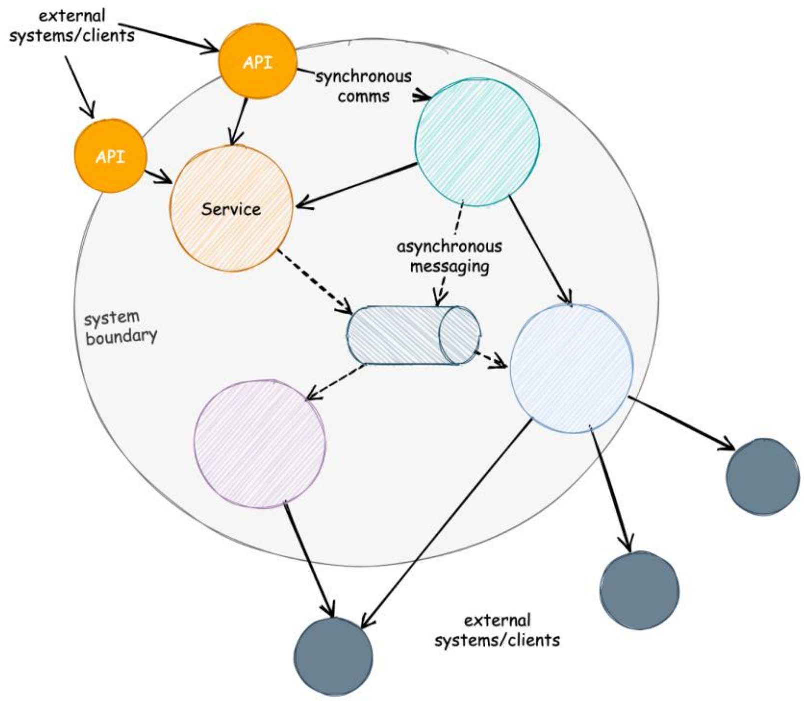
Dr. Peter Rodgers created the term microservices in 2005, and it was originally known as "micro web services." At the time, the fundamental motivation behind "micro web services" was to break up huge "monolithic" architectures into numerous independent components/processes, resulting in a more granular 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 uncommon.
- 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 adding more instances) (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. This indicates that, in comparison to a monolith, the technology choices underpinning microservices implementations are significantly less important; that is, the choices 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: We've all worked on enormous monolithic software systems that were based on two-decade-old technology and were extremely difficult and dangerous to update. In 2018, I recall working on a project where the team was stuck on Java 6 due to complex library dependencies, a lack of sufficient unit tests, and the accompanying migration risk. With microservices, this is almost never the case. Smaller codebases are 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, on the other hand, emphasises loose coupling, with services that are self-contained, completely own their dependencies, and avoid synchronous (blocking) communication. When a microservice fails, it will always disrupt some portions of the system and certain users, but it will 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.

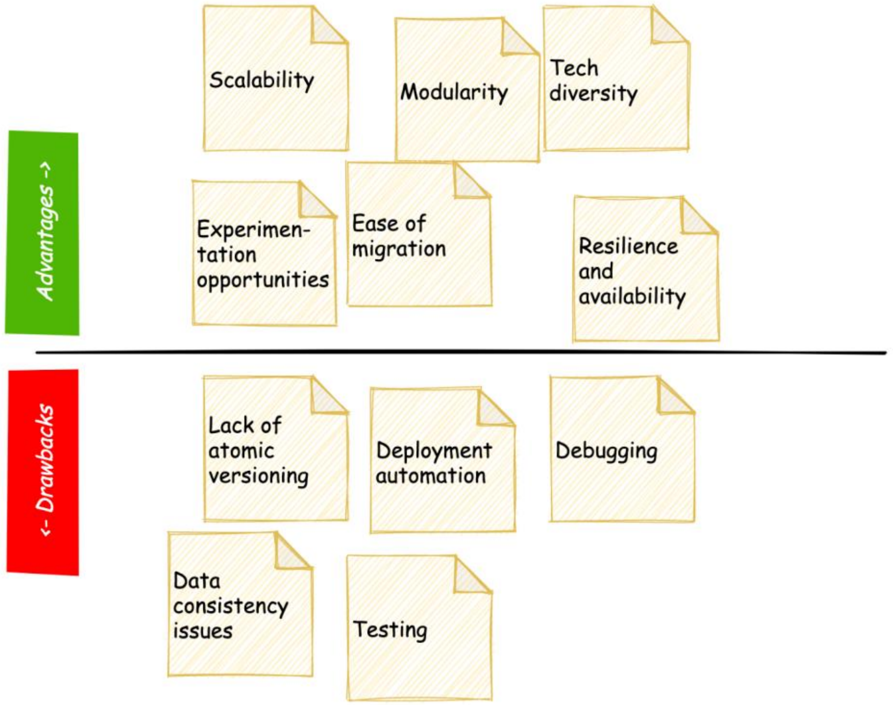
Debugging: When components of a system communicate in the same process, it's much easier to debug their interactions, especially when one component merely calls a method on another. Attaching a debugger to the process, stepping through the method calls, and observing variables is usually all that is required. In microservices, there is no simple comparable. Tracing and piecing together service messages is notoriously challenging, necessitating additional equipment, infrastructure, and complexity. In a microservices architecture, you don't want to be caught in the middle of a system-wide outage.

Limitations of Microservices

Data Consistency: ACID is a feature of a monolith that runs on a single relational database. To put it another way, transactions. In distributed systems, transactions take on a totally distinct shape. 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 prefered). 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

Try thinking of a third party microservice and how you can connect it to your current express application to add a new functionality to your application.

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 the 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 Library is an event-driven system, it emits and listens for certain events to be triggered.

Do have a look at it in more detail here

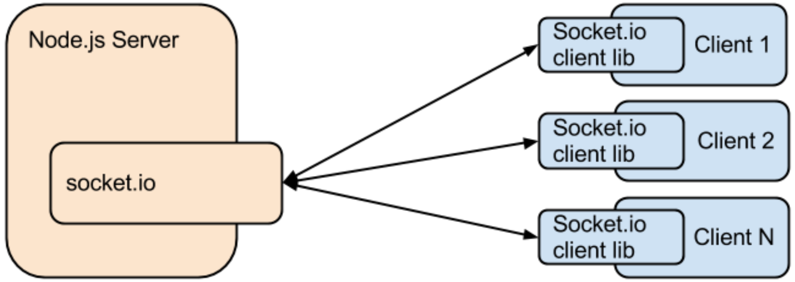
Sockets

So let's try to understand how our application design would look like when we add socket.io to our application.

From the image what we can understand is that the nodejs server hosts one instance of socket.io serv side package and every client host its own instance of socket.io client. So we can get to know that the server does not need multiple host 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.

While on the other hand the client can at one point only listen to one server side socket connection.



Adding Sockets to Nodejs Application

Now For creating the socket.io application, we need to install the socket.io npm to our application and embed it into our application so that we have a server side socket ready to use.

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

Now that we have the socket.io npm installed lets see what we need to add in our application in order to connect it to a client side application.

So in order to add socket.io library to our server.js file we need to add a little bit of code to our application.

Let's try to understand what is happening in that code.

So first we require the library in our application, and then we create a listener that whenever a user connects it prints in the console that the user has connected and also it has a set interval function that sends a random number on an event name called number to all the users that are connected to our socket server.

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

```
let io = require('socket.io')(http);
io.on('connection', (socket) => {
  console.log('a user connected');
  socket.on('disconnect', () => {
    console.log('user disconnected');
  });
  setInterval(()=>{
    socket.emit('number', parseInt(Math.random()*10));
  }, 1000);

});
```

Adding Sockets to Nodejs Client Application

Let's try to add the socket library to our client side.

If you see the index.html file of our client application you can see that first we need to add the socket library to our index.html file we can do that by adding this line of code.

```
<!-- socket IO Served by the server -->
    <script src="/socket.io/socket.io.js"></script>
```

We add the script which is reading the socket file from our socket.io folder installed using the npm and we will use this socket library in our env.js file. Now let's add the socket event to read the data from our socket library from the server.

Adding Sockets to Nodejs Client Application

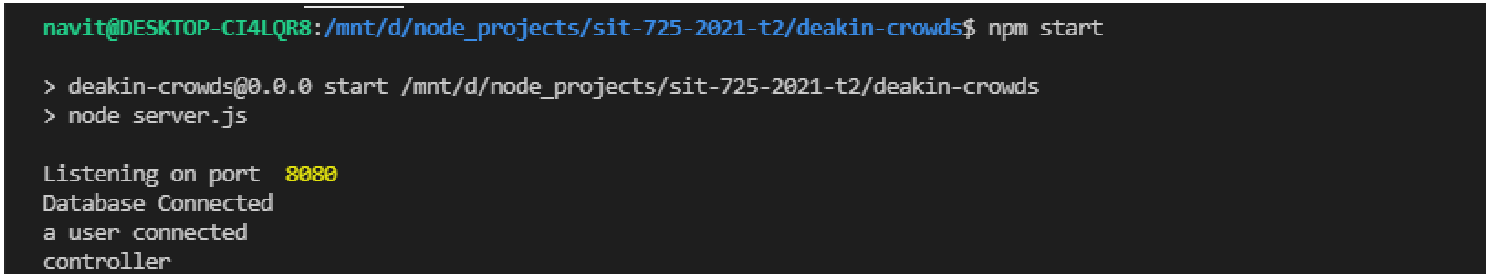
Next we update our env.js file to listen to events

```
// connect to the socket
let socket = io();
socket.on('number', (msg) => {
    console.log('Random number: ' + msg);
})
```

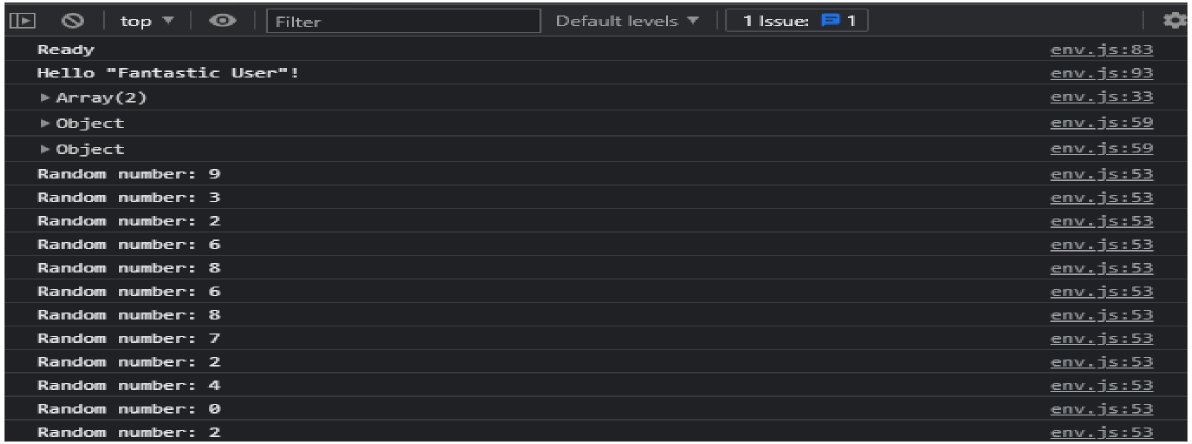
Here we create a socket variable using the io library and then we tell our application that the moment it connects to our socket start listening to the 'number' event and whatever msg you get from the server print it in the console.

Now that we have added the code both side lets see if our application behave as we expect. So we start our application.

Adding Sockets to Nodejs Client Application



```
navit@DESKTOP-CI4LQR8:/mnt/d/node_projects/sit-725-2021-t2/deakin-crowds$ npm start

> deakin-crowds@0.0.0 start /mnt/d/node_projects/sit-725-2021-t2/deakin-crowds
> node server.js

Listening on port  8080
Database Connected
a user connected
controller
```



```
Ready                                   env.js:83
Hello "Fantastic User"!                 env.js:93
 ▸ Array(2)                             env.js:33
 ▸ Object                               env.js:59
 ▸ Object                               env.js:59
Random number: 9                        env.js:53
Random number: 3                        env.js:53
Random number: 2                        env.js:53
Random number: 6                        env.js:53
Random number: 8                        env.js:53
Random number: 6                        env.js:53
Random number: 8                        env.js:53
Random number: 7                        env.js:53
Random number: 2                        env.js:53
Random number: 4                        env.js:53
Random number: 0                        env.js:53
Random number: 2                        env.js:53
```

Adding Sockets to Nodejs Client Application

So we can see that our application prints whenever we get a user connected to our application and also on the client side we are receiving an random number from the socket library that is being broadcasted by our server. This is the exact result we were hoping to see.

So now you have connected a socket system to your application.

You can also use more things from the socket library for doing that please read the document provided by the socket.io here.

Exercise 5

Try modifying the socket part of your express application to make use of sockets in a different way from the example we did.

End of Presentation