Institute of Data

2022

Software Engineering

Module 9

API Development

© 2022 Institute of Data

Institute of Data

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

Institute of Data

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?

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.

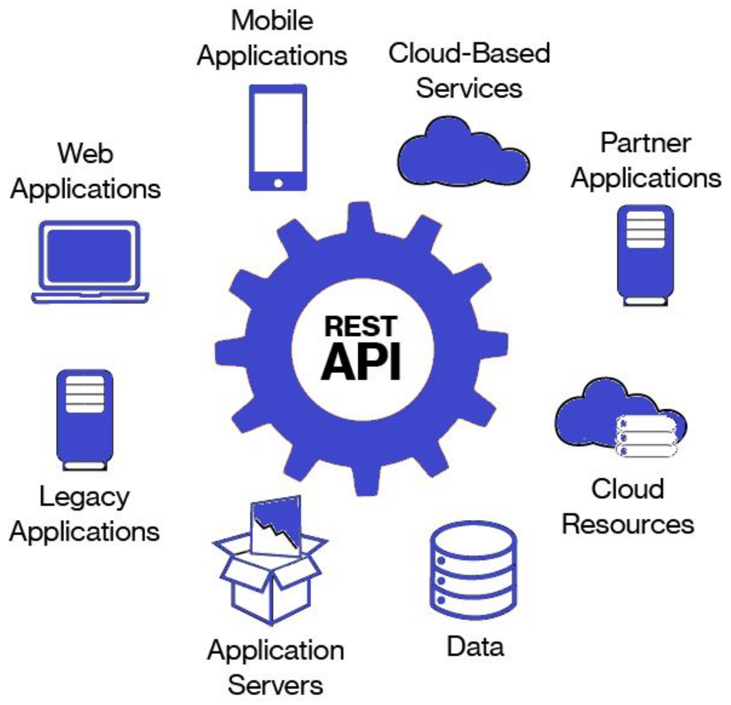
Any data (e.g., image, video, text, etc.) is treated as a resource by REST, which the client can fetch, edit, and delete. REST requires that a client be able to access a certain URL and send a request to complete the necessary function. After that, the server responds appropriately.

© 2022 Institute of Data

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.



© 2022 Institute of Data

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

© 2022 Institute of Data

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.

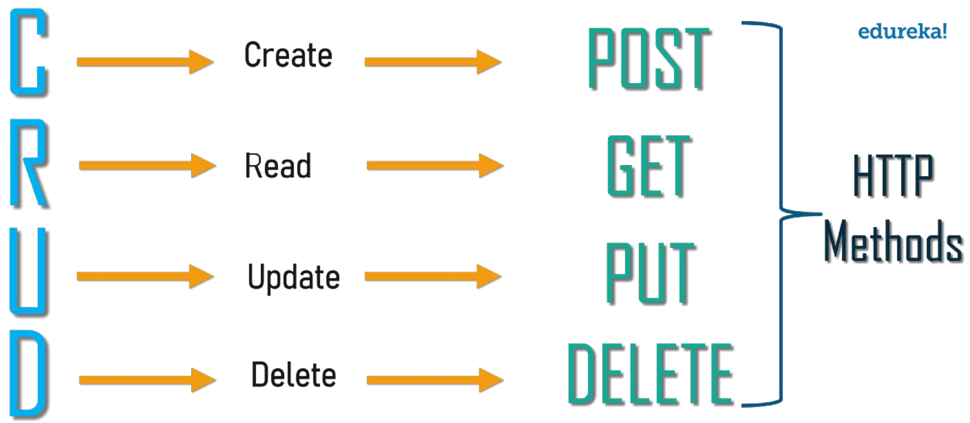
Layered System - By constraining component behaviour, the layered system architecture makes an application more reliable. Because components in each tier cannot interact beyond the next adjacent layer they are in, this architecture helps to improve the application's security. It also supports load balancing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

© 2022 Institute of Data

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.



© 2022 Institute of Data

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.

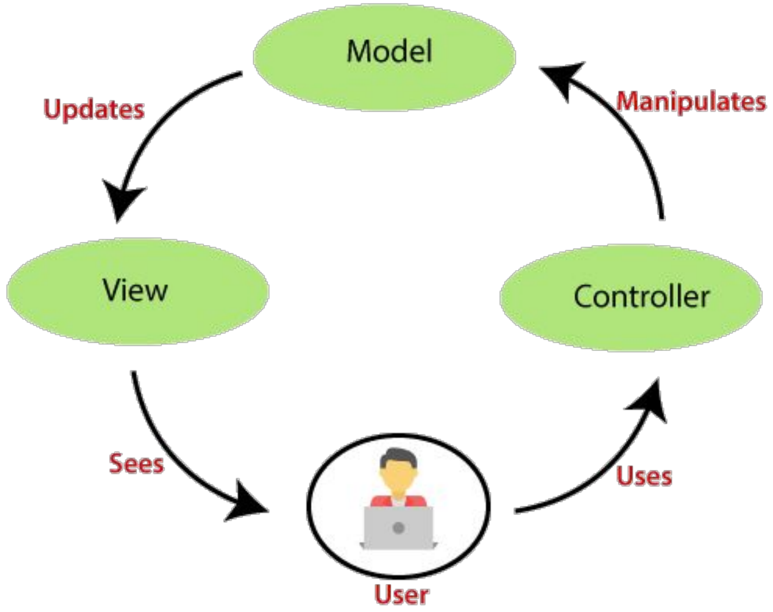
© 2022 Institute of Data

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.



© 2022 Institute of Data

MVC Structured 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.

© 2022 Institute of Data

MVC Structured 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, as 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 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 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.

© 2022 Institute of Data

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 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 will change depending on your local setup):

```
DB_USER=root
DB_PASSWORD=root
DB_DATABASE=classicmodels
DB_LOCAL_PORT=3307
```

© 2022 Institute of Data

Exercise 1

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

© 2022 Institute of Data

Section 3: REST API using MongoDB

In module 5 we 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 will try to connect it with a database and perform a change in data.

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.

© 2022 Institute of Data

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 dotenv`** to install the dot env 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.*

Next, install the mongoose module from npm in our application. For doing so we just need the command

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

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

© 2022 Institute of Data

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

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

(optional) Create a file called .env and store the mongodb connection string in there as follows:

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

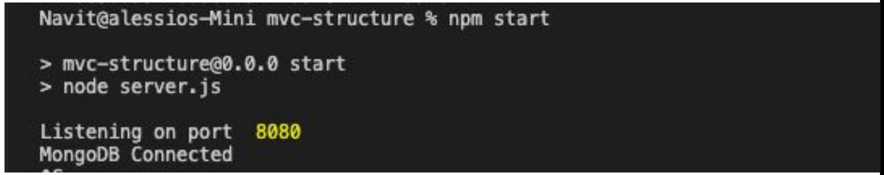
Once we have created dbConnect.js we need to import it in our server.js file. To do that we simply add (near the top) :

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

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

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

const mongooseOptions = {
    useNewUrlParser: true,
    useUnifiedTopology: true
};

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

// 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;
```

© 2022 Institute of Data

Now if we start our server (using **`npm start`**) 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. 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')
};
```

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

let 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);

//The "user" mentioned in the above line should be in the
//singular form ...whereas the actual collection name in
//mongodb will be in the plural form.
//Refer to the mongoose documentation for more details:
https://www.npmjs.com/package/mongoose
//const MyModel = mongoose.model('ModelName', mySchema);
//The first argument is the singular name of the collection your model
is for.
//Mongoose automatically looks for the lowercase plural version of
your model name.

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

© 2022 Institute of Data

REST API using Mongoose

Now that we have our Model ready and connected let's create an API 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 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 userRoutes = require('./routes/userRoutes')
app.use('/api/users', userRoutes)
```

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

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

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

module.exports = router;
```

© 2022 Institute of Data

REST API using Mongoose

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. Next we 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({}, {}, {}, (err, data) => {
        if (err) throw err;
        res.send({result: 200, data: data})
    });
}

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

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

© 2022 Institute of Data

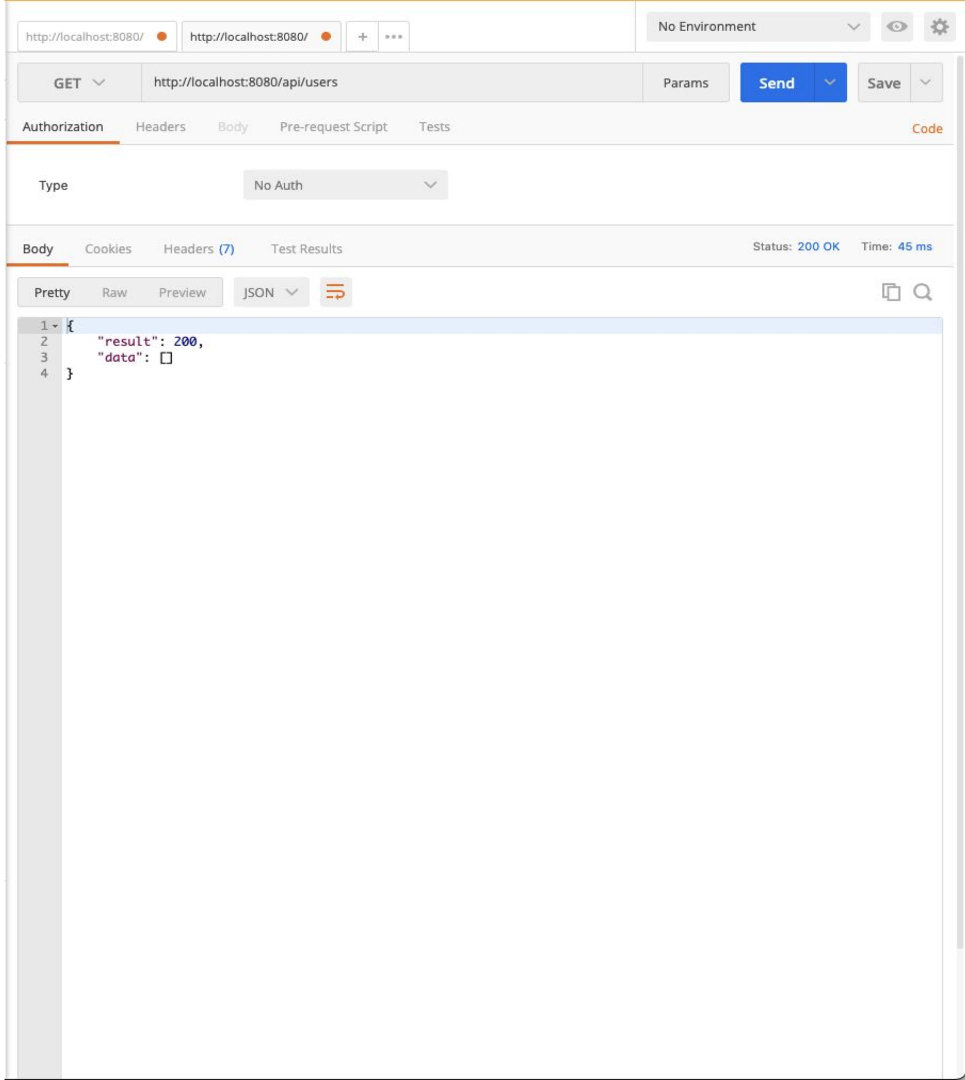
REST API using Mongoose

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. 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 API, using the right port (8080) and route endpoint (/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.



http://localhost:8080/api/users

Type: No Auth

Status: 200 OK Time: 45 ms

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

© 2022 Institute of Data

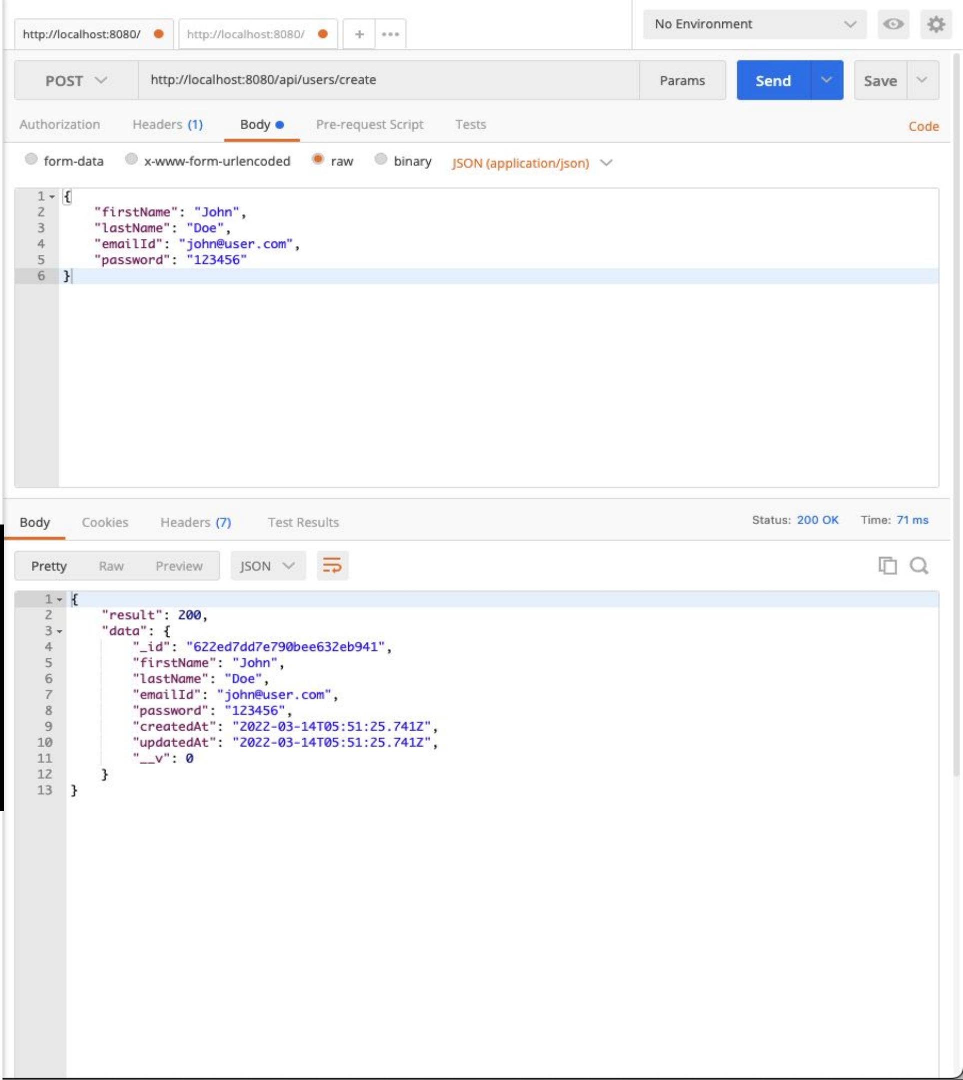
REST API using Mongoose

Now let us add a user to the database. For this we can use the createUser API and send the below JSON to our endpoint at /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.



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

Body: raw, JSON (application/json)

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

Status: 200 OK   Time: 71 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
    }
}
```

© 2022 Institute of Data

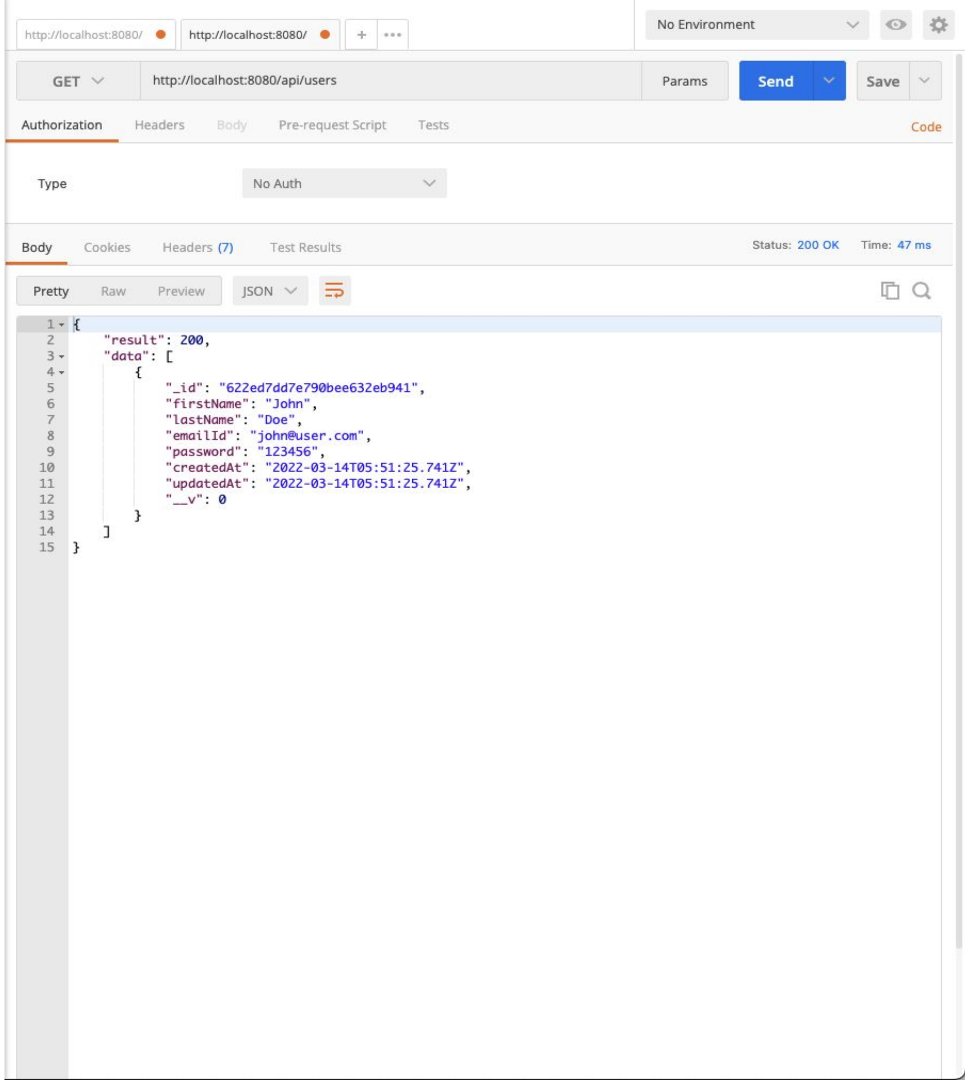
REST API using Mongoose

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

From the result you can see our user was successfully created, as the getUsers API returned our user.

Similarly we can create Update and Delete users API 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.



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

Type: No Auth

Status: 200 OK Time: 47 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
        }
    ]
}
```

© 2022 Institute of Data

Exercise 2

Try creating a express application for a Blog website using MongoDB. You can refer to your database model from 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.

© 2022 Institute of Data

Section 4: REST API using MySQL

So in previous modules we 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 will connect it with a database and perform a change 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.

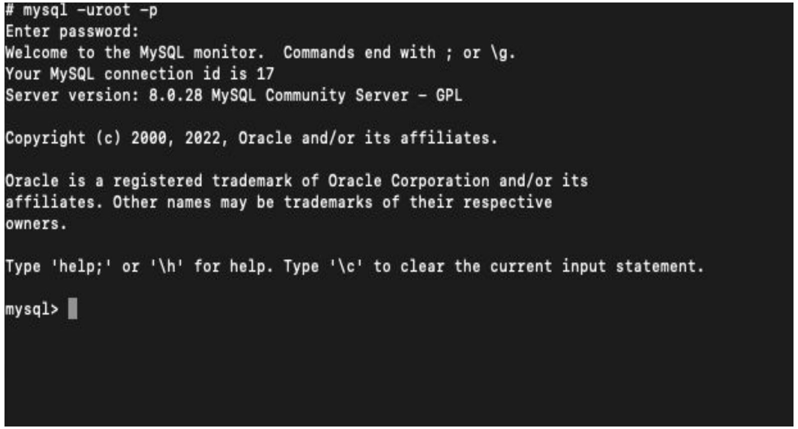
You can also refer here to learn in detail about Sequelize npm.

© 2022 Institute of Data

Prerequisite for using MySQL with Express

So before we create an express application to connect with MySQL, we need to run MySQL in terminal or Workbench, and actually 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.



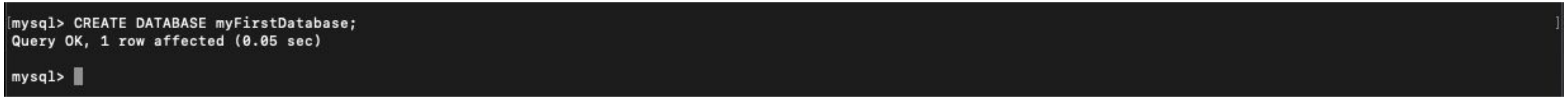
```
# mysql -uroot -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 17
Server version: 8.0.28 MySQL Community Server - GPL

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> 
```

© 2022 Institute of Data

Prerequisite for using MySQL with Express

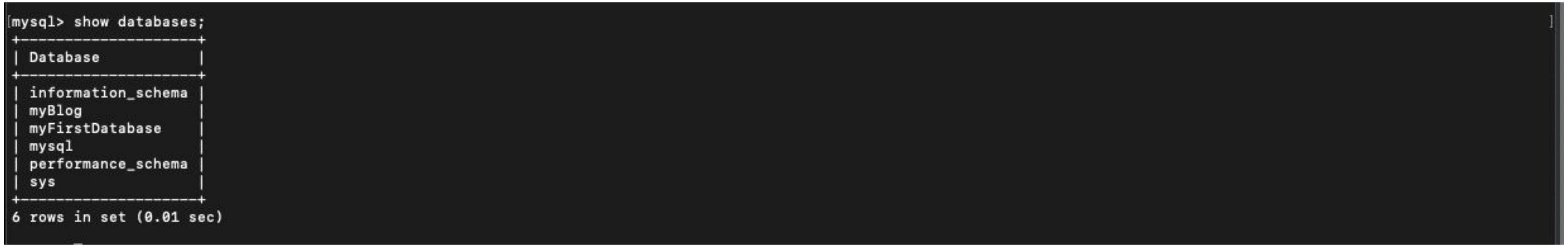
Let’s now create our first Database. For doing this we will simply type ‘ CREATE DATABASE <database name> ‘

```
> CREATE DATABASE myFirstDatabase;
```



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

mysql>
```

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)
```

© 2022 Institute of Data

Connect Using Sequelize

Now that we have mysql running in background and our database created, we need to create a new express application and then install the sequelize and mysql2 npm 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` and accepting all of the defaults. The file should have the same content as slide 17.

Next run these commands to add first express and dotenv, and then the sequelize and mysql2 packages:

```
> 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.

© 2022 Institute of Data

The contents of **dbConnect.js** should look something like this:

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

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

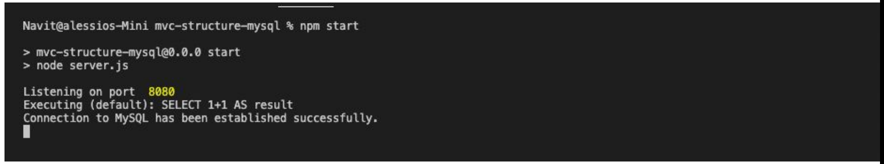
We also need to create a .env file with our database information (change as needed):

```
DB_NAME=myFirstDatabase
DB_USER=root
DB_PASSWORD=root
DB_HOST=localhost
DB_PORT=3307
```

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

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);
    }
}

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

© 2022 Institute of Data

Now 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 **user.js** and add this code 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({
     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;
```

© 2022 Institute of Data

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

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

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

async function init () {
    await User.sync();
};
init();

module.exports = {
    User
};
```

© 2022 Institute of Data

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 file in our routes folder called **userRoutes.js**. Once we have our routes 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 userRoutes = require('./routes/userRoutes')
app.use('/api/users', userRoutes)
```

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

© 2022 Institute of Data

REST API using Sequelize

Now once we are ready with our routes, 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 the content below to our **index.js** file in our controllers folder:

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

```
"use strict";

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

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

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

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

© 2022 Institute of Data

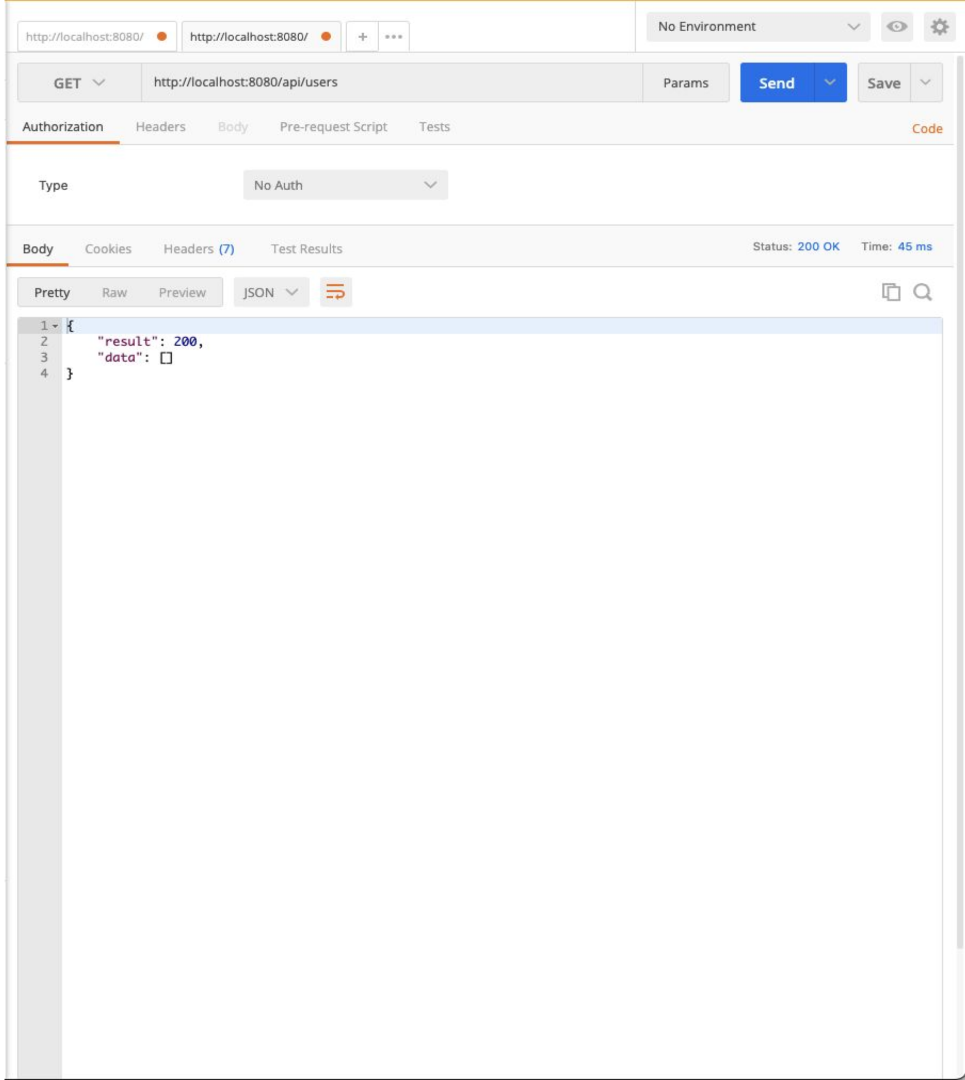
REST API using Sequelize

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. So let’s try our REST APIs that we created.

First we need to start the server using `npm start`.

Next access the getUsers API via the route endpoint /api/users. Initially it will be empty as we haven’t added any users.



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

Type: No Auth

Status: 200 OK Time: 45 ms

JSON

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

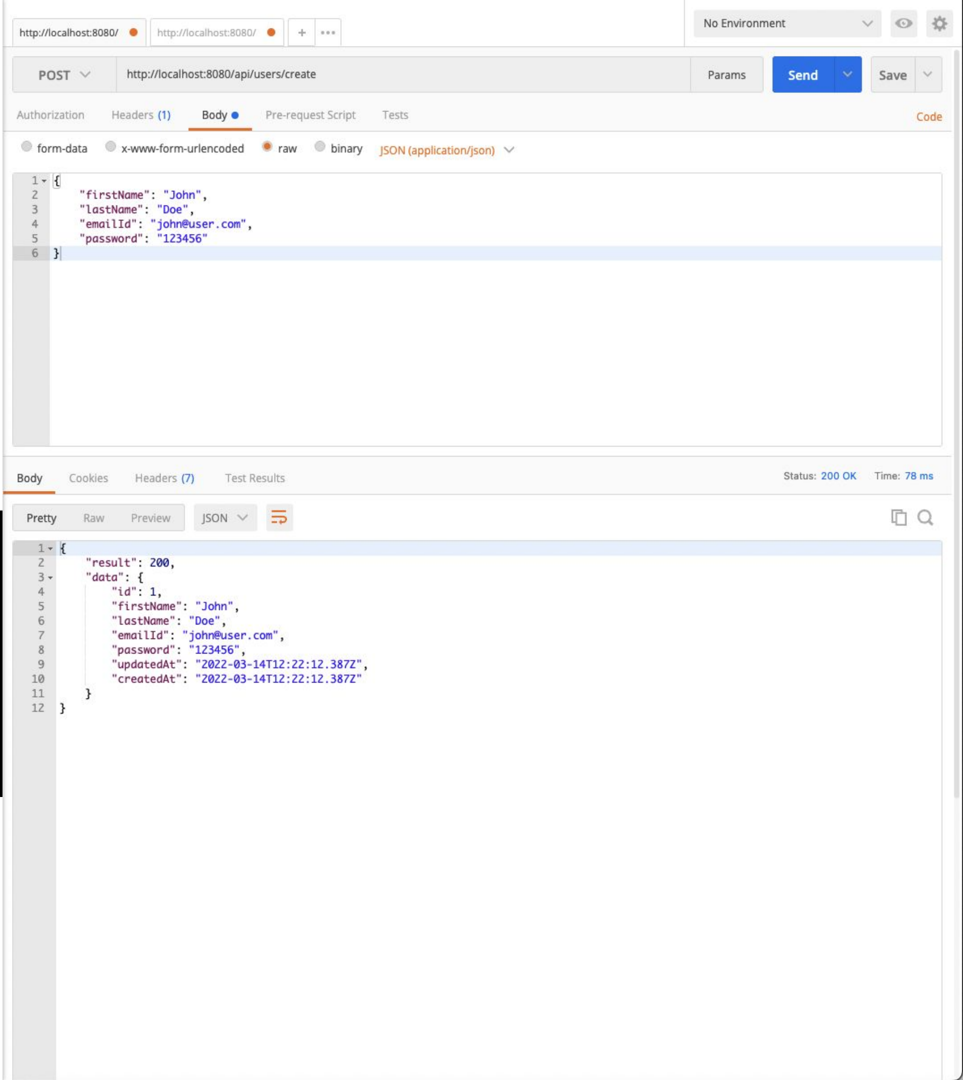
© 2022 Institute of Data

REST API using Sequelize

Now let us add a user to the database. For this we will use the createUser API, using the POST method via the /api/users/create endpoint, and we can send the below in the body of the request as 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

```
{
    "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"
    }
}
```

© 2022 Institute of Data

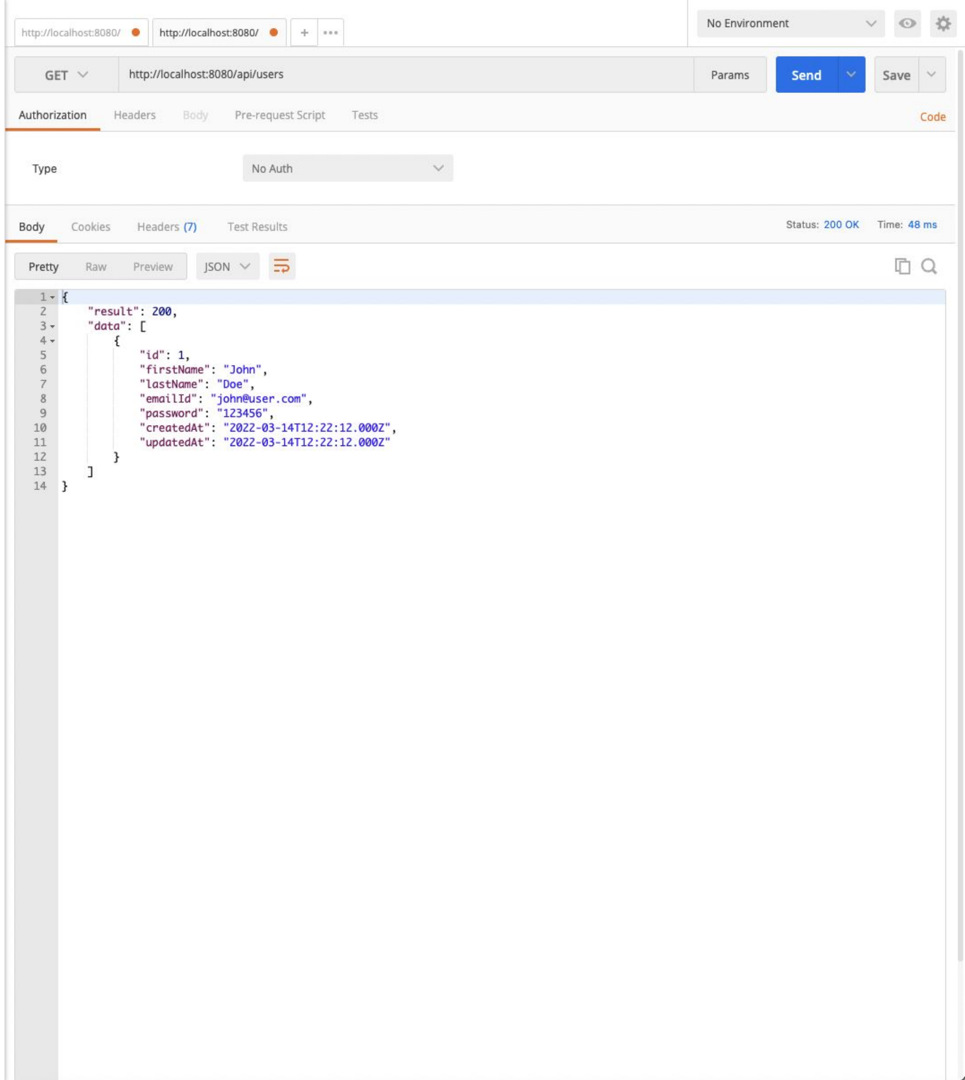
REST API using Sequelize

Now let's try to GET our users again and cross verify if our user was added successfully.

From the result you can see our user was successfully created and the getUsers API returned our new user.

Similarly we can also create Update and Delete users API endpoints also, using the PUT and DELETE methods with new route and controller functions.

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: 48 ms

```
{
    "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"
        }
    ]
}
```

© 2022 Institute of Data

Exercise 3

Try creating a express application for a Blog website using Sequelize. You can refer to Module 8 for the logical/physical models.

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.

© 2022 Institute of Data

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.

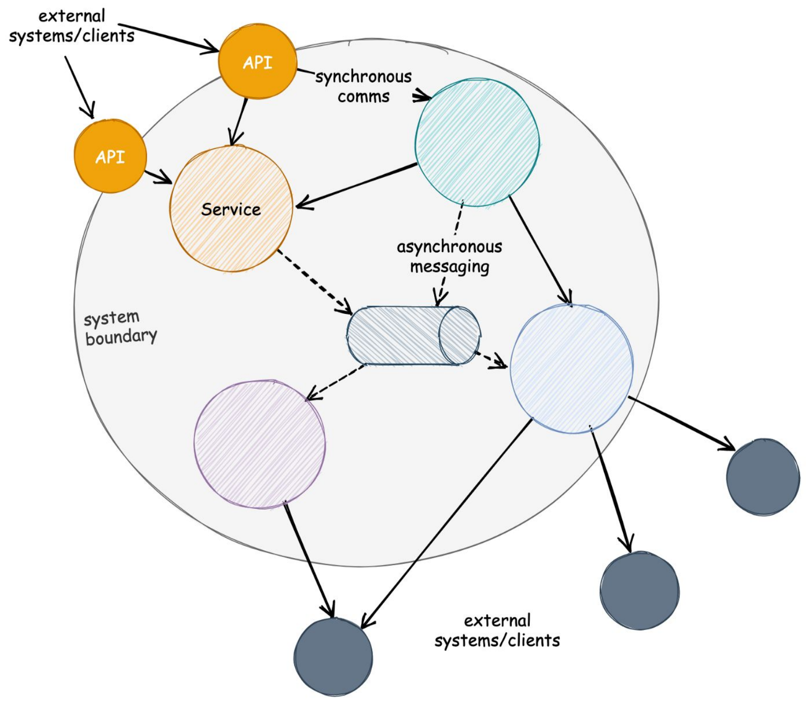
Modular, distributed programs have been around for a long time. Microservices aren't a novel concept in this context. However, it was the principles guiding how microservices were developed and consumed that made them popular. Microservices used open standards such as HTTP, REST, XML, and JSON to replace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

© 2022 Institute of Data

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.



© 2022 Institute of Data

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.

© 2022 Institute of Data

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.

© 2022 Institute of Data

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.

© 2022 Institute of Data

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.

© 2022 Institute of Data

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.

© 2022 Institute of Data

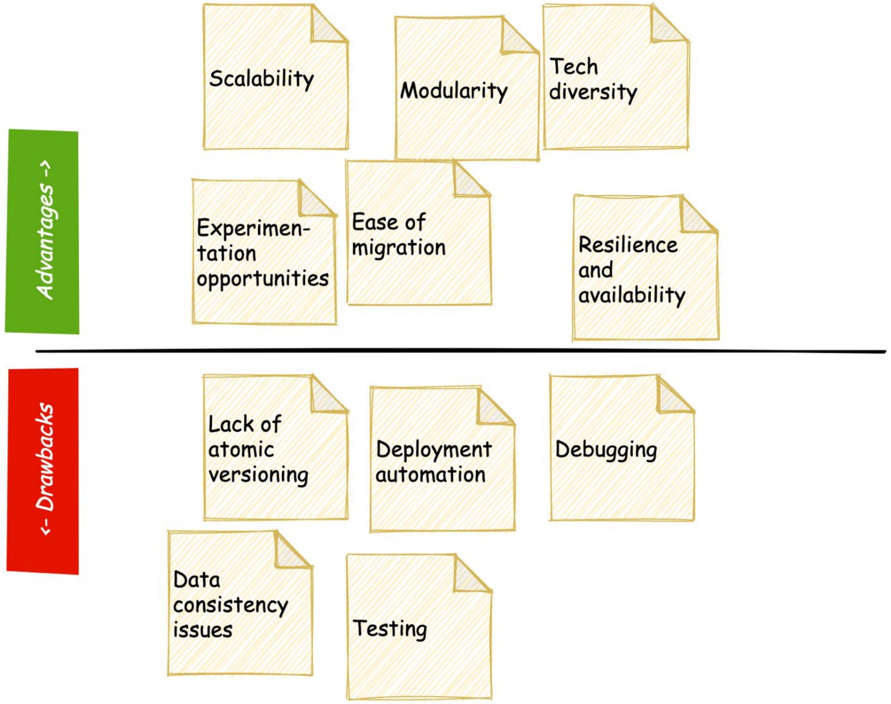
Limitations of Microservices

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.

© 2022 Institute of Data

Flashback on Microservices



© 2022 Institute of Data

Exercise 4

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

© 2022 Institute of Data

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

© 2022 Institute of Data

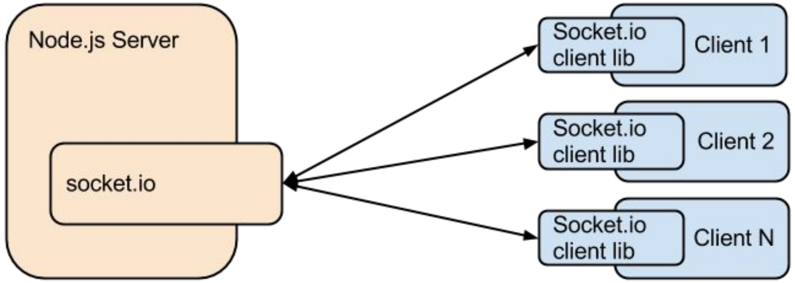
Sockets

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.



© 2022 Institute of Data

Adding Sockets to Nodejs Application

For creating the socket.io application, first create a new express app in a new folder (eg. socket-chat-app) with **index.js** as the main file. We then need to install both express and the socket.io npm package to our application and embed it into 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 https://socket.io/get-started/chat

© 2022 Institute of Data

So in order to add socket.io library to our index.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 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 always 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 help us listen to different event names and also perform different actions on the client side using those 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');
});
```

© 2022 Institute of Data

Adding Sockets to Client Application

Now that we have a socket server, let's try to 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 access the socket io library via 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>
```

© 2022 Institute of Data

Adding Sockets to Nodejs Client Application

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):

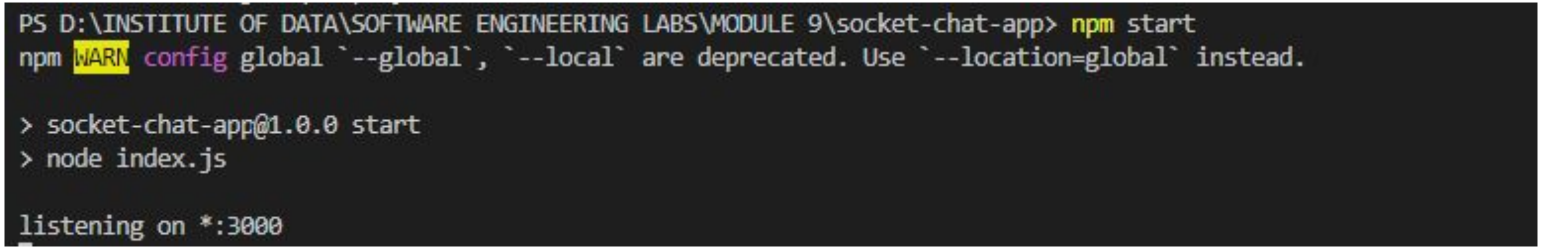
```
let socket = io();
socket.on('connection', (msg) => { console.log(msg) })
```

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

So we start our application using `npm start`

© 2022 Institute of Data

Adding Sockets to Nodejs Client 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.

© 2022 Institute of Data

Adding Sockets to Nodejs Client 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 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.

© 2022 Institute of Data

Exercise 5

Using the guide at https://socket.io/get-started/chat 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.

© 2022 Institute of Data

End of Presentation

© 2022 Institute of Data