Institute of Data

Software Engineering

Module 5 Introduction to Back-End Development

Agenda: Module 5

- Section 1 : Introduction to Web services and Javascript applications
- Section 2 : Develop the MVC Structure
- Section 3 : Design a Back-end Service
- Section 4 : Unit Testing
- Section 5 : Object-Oriented Development
- Section 6 : Swagger

Introduction

Back ends are sometimes seen as the engine of everything. This is where the magic really happens, where the data gets routed, where all the users and their data eventually are stored, where all the major computational tasks are taking place.

Some may say, that this is for those more introverts' developers that care more about numbers, and are just software mechanics, but in reality, back ends are the art of performance and technical engineering.

The personal satisfaction of refactoring a function to achieve smaller wait times and less memory usage is the guilty pleasure of those who decide to spend hours and hours improving their code, knowing that no one will ever see it. Of course, this was written by a Software Engineer.

Section 1 :

Introduction to JS Applications

As you start your journey as a web developer, you will be tempted to improvise and hope for the best. While this is a common practice for beginners, soon you will discover that planning is the most important thing you can do in this industry.

Whether you are working on your side hustle or a client project, you should always have a good plan. In web development, the plan starts with the **Design**.

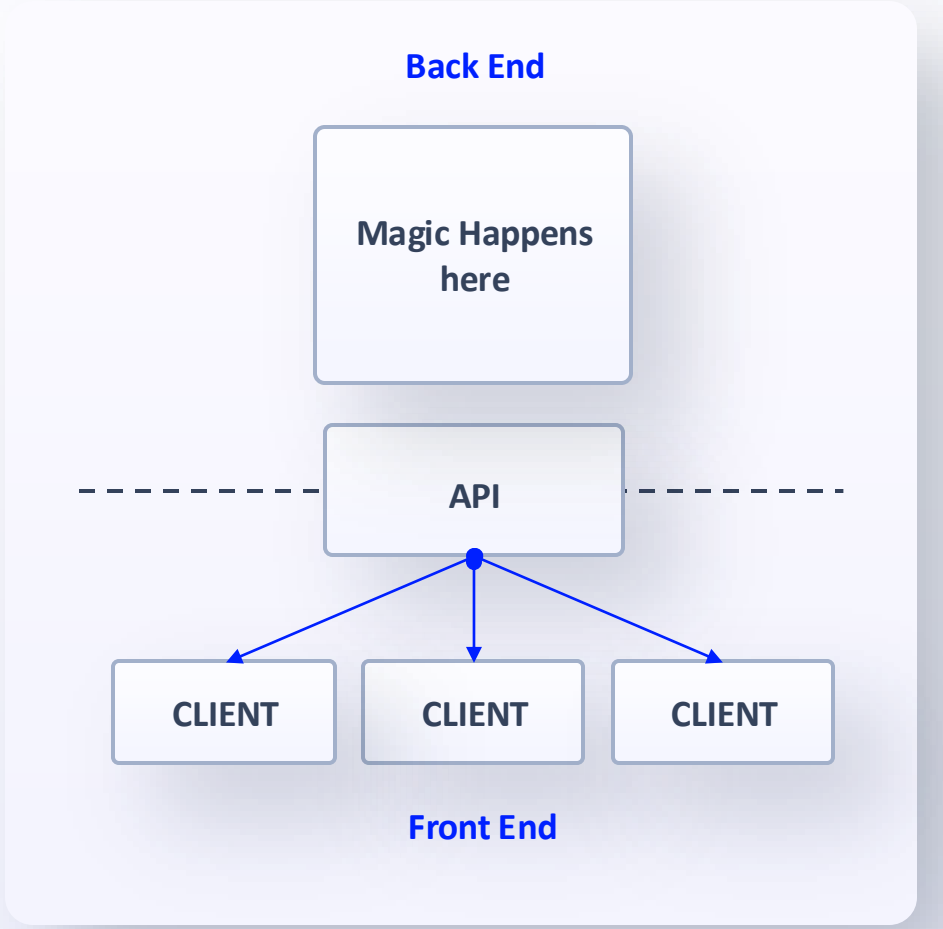
In order to design an effective **back-end** application, first we must understand how they work, both internally and as they interact with other systems.

What is a Back End?

As the name implies, it is something that happens at the back, like a kitchen in the restaurant. All the orders go in, and for those waiting outside, it is just a magic box, or a “black box” if we want to stay in the field.

The back end of a web application is generally the web server (or servers) and the many other services which are abstracted from the user.

This abstraction normally happens through the API (normally REST Interfaces)

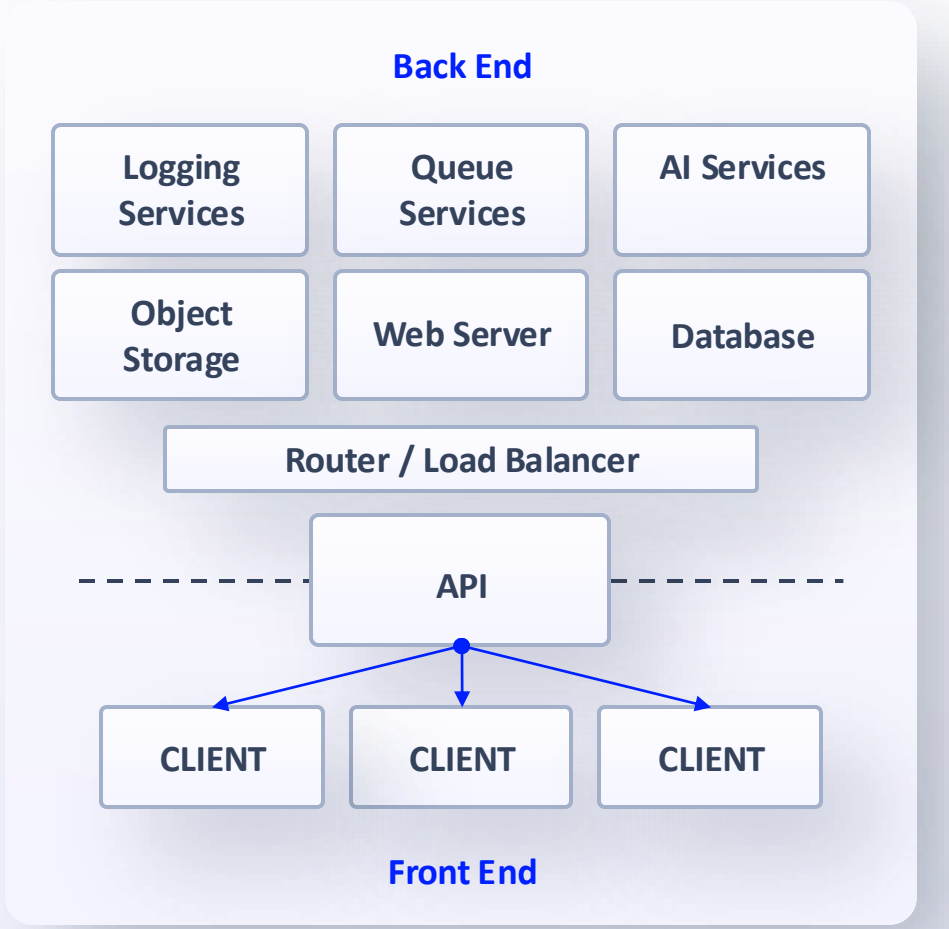


Back-End Architecture

If we break down the back end, and make it visible, we can see there is a lot happening here.

In this course, we will look at simple Client-Server applications, but back ends can become extremely complicated, especially if third party services are added.

To keep things simple, we will focus on simple CRUD operation, which are data insertions and retrieval, and which make up more than 90% of the web at the moment.



Asynchronous Environments

To better understand web-based back-ends, you need to shift your focus towards thinking “asynchronously” . What does that mean ?

In a normal environment, we expect that once you call a function, the system will execute that function and immediately return the result. Look at the example.

```
Example
console.log(1)
console.log(2)
console.log(3)
```

In this case, our result will be 1,2,3 – this what expect and this is what we get. What happens if we include some “waiting” time?

```
console.log(1)
setTimeout(function(){
    console.log(2)
},500)
console.log(3)
```

Some will say, 1, wait 500ms, 2 and then 3.
WRONG! In async environments, this will not happen, it will happen in Java, or C, because are sync environments, but the web is not a nice place where resources are waiting for you.

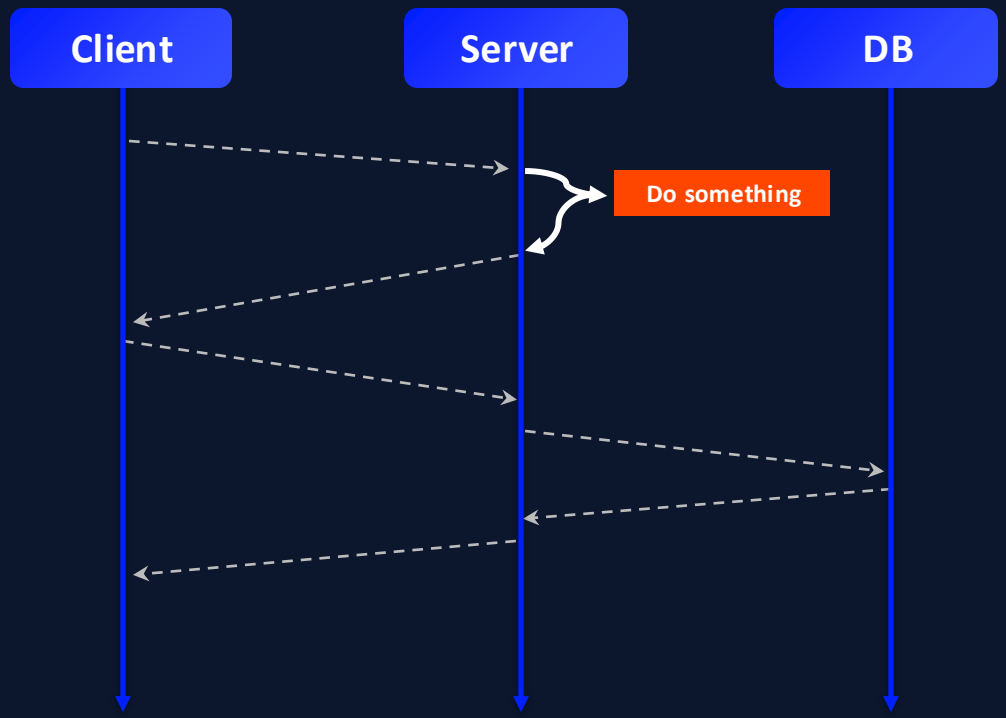
In this case, the answer is 1,3 and after 500ms, 2. This is because the environment does not wait.

Why is this? Because by its design, the internet is a “best-effort model” – translated – you make a request, you hit a web page, there is no guarantee you will get something back, and you don’t want to have your system hanging – which is why we have timeouts.

Round Trip Requests

Everything about the web is Async. If it wasn't, performance would be a terrible problem, and from a resources point of view, it would be difficult to maintain a global network.

We identify with Round Trip Time (RTT), the time it takes for a request to be executed and return to the client. RTT is a common metric for websites effectiveness.

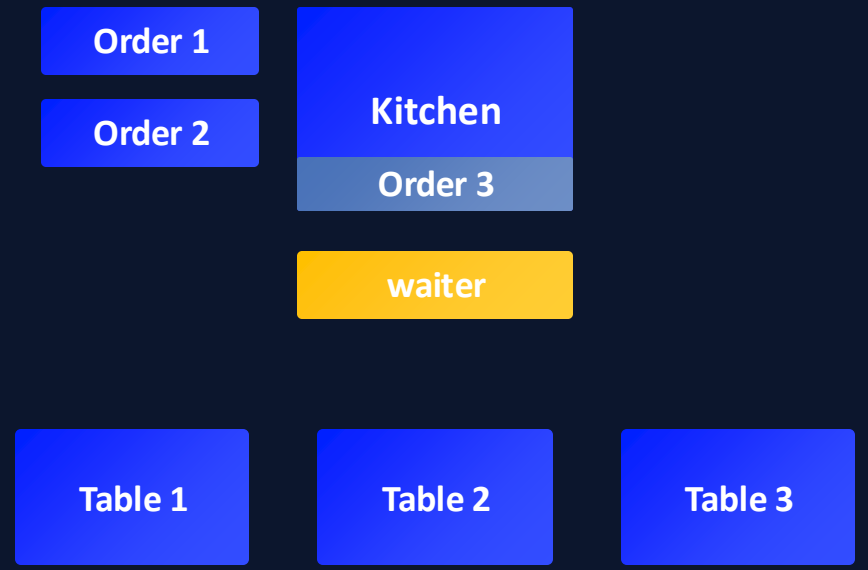


This is a multi-tier design, which extends the classic client – server design. When designing for the web, you always need to understand what exactly happens. You may need to make decisions. For example, what if you were using a system like this for live videos? It would be quite slow.

Multithreaded vs Single-Threaded

To make things even more complex, we coined terms single threaded and multithreaded. A thread is a single process. Remember, at the end of the day, a computer CPU can only do one thing, which is manipulating 1s and 0s. With time, we have created multithreaded environments, but these did not work on the web as we hoped. NodeJS, our language of choice, is single threaded. If we go back to the restaurant analogy, imagine there are many tables, one chef and one waiter.

In this scenario, the waiter keeps running between the kitchen and the tables to pick up the orders. But regardless, the kitchen can only handle one order at a time. Also, if that order cannot be completed in a defined allocated time, it will be removed and paused, to let another order go in. This concept is the event loop. This is not going to trouble you for a few more years, but it is good to know.



Express

Express is one of the most common frameworks used for the web, because it allows you to use Javascript everywhere, in the front end and in the back end.

This is very practical, and it helped in delivering strong full stack designs, because developers do not need to learn two languages.

We will start with the simplest form.

Create a directory for your app and change into the folder:

```
mkdir myapp
cd myapp
```

Then you can initialize it using npm init. This will generate a basic structure for any NodeJS application. It will ask questions, feel free to answer as you like or just hit Enter to accept the defaults. You will learn later what they all are, but make sure you leave index.js as the entry point.

```
npm init
```

Next run

```
npm install express
```

This will install the express package as a dependency for your app.

package.json

Your package.json should look like this. As you can see, you have added express as a dependency. This tells the program to download express in order to work. The number next to it **4.17.1** refers to the version. You can use * , this means “download the latest” which is something you should never do. The package may be updated one day, and not be compatible with past versions. For this reason we **NEVER** use * .

```
{
  "name": "backend",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test
specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.17.1"
  }
}
```

package.json

The package.json file is extremely important. There are a lot of things you can do here, but the most important is the dependency tracking. We don't ever pass modules to clients - instead we always pass the package.json file, which knows what to download.

Let's make a small modification. Create an **index.js** file and add anything to it. A `console.log('hello world')` will do.

```
{
  "name": "backend",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "node index.js",
    "test": "echo \"Error: no test specified\" &&
exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.17.1"
  }
}
```

Modify the package.json to add the start script. This allows you to build a list of automated commands.

When you run the command "npm start" it will execute the script instead.

index.js

- Remove everything from the **index.js** file and copy the following.
- This is how much you need to run a complete web server.
- Save it and run using **npm start**
- Open your browser and go to http://localhost:3000 - you should see the “Hello World!” message.

```
const express = require('express')
const app = express()
const port = 3000

app.get('/', (req, res) => {
  res.send('Hello World!')
})

app.listen(port, () => {
  console.log(`Example app listening at
http://localhost:${port}`)
})
```

index.js

The following is a breakdown. The first line is requiring the express package, this is the same syntax for any **npm package**.

Next, create an app using the **express()** function, which is a **constructor** or initiator function. You can possibly create as many as you want, just give different names.

Last, **3000** is the port we are binding it to. You can set any open port on your system. Try to use port **80**, and simply go to localhost without passing the port, see what happens.

```
//require the express package
const express = require('express')

//create an app using the express
// package
const app = express()

// set the port to 3000
const port = 3000
```

index.js

App.get is a binding for an **endpoint**.

This reads as “bind a **get** endpoint to the object **app** using the **/** URL (root). When called, call this function passing two arguments, req and res. The function contains a call to **res.send** which will send the message in the brackets to the requestor ”

Req stands for **request**, res stands for response. The **response** object has the ability to send messages back.

```
app.get('/', (req, res) => {
  res.send('Hello World!')
})
```

In normal terms, a web browser request is always a GET (when you insert a URL) and it hits an endpoint. In this case, this is bound to localhost:3000.

```
app.get('/test', (req, res) => {
  res.send('This is a test')
})
```

Add this new endpoint and this time, on your web browser, request localhost:3000/test

index.js

Finally, this **listen** function activates the web server; until now, it has not been.

This reads “Let the webserver listen on the provided port (3000), and once the server starts, if successful print out this message to the console”.

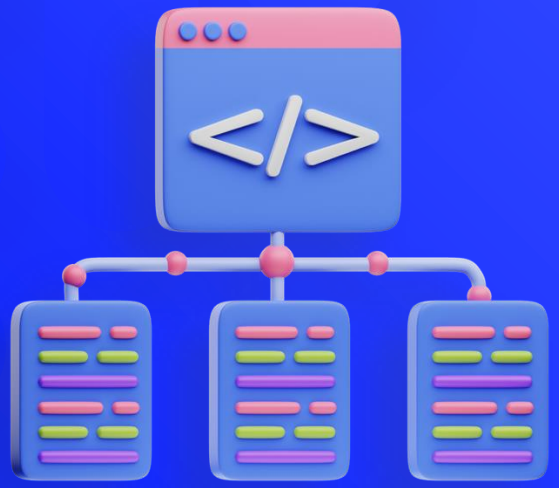
```
app.listen(port, () => {
  console.log(`Example app listening at
http://localhost:${port}`)
})
```

Exercise 1

- Create a basic back-end application with multiple web servers running on different ports.

Section 2 :

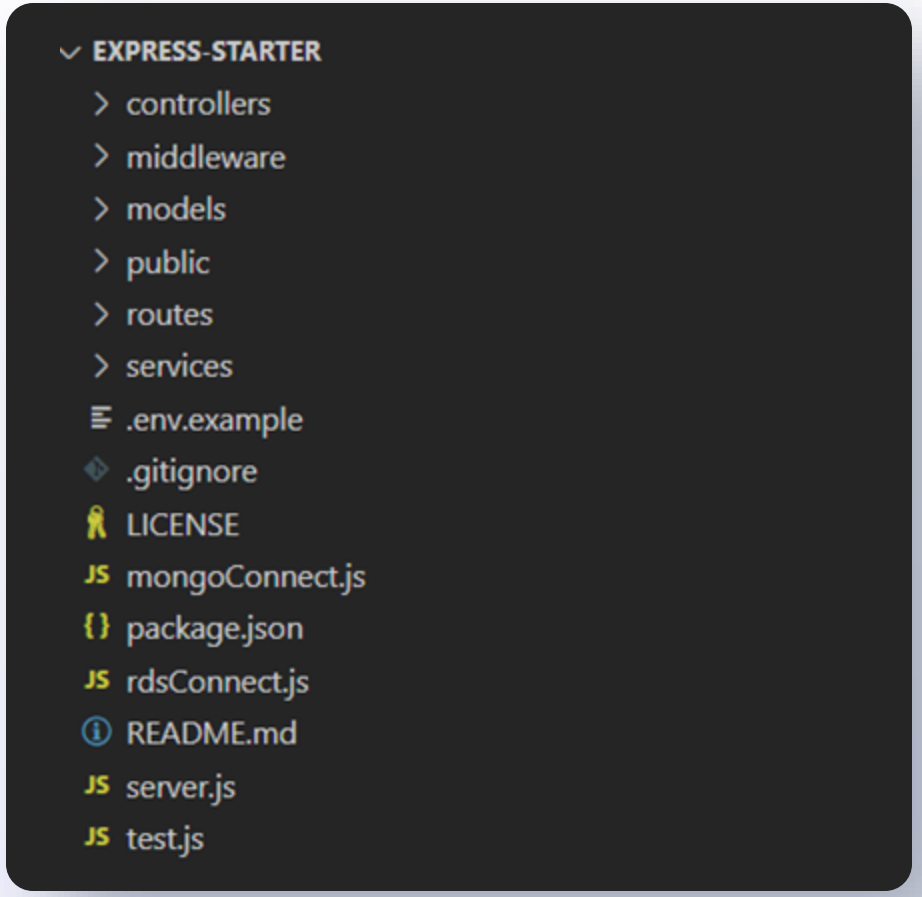
Develop the MVC Structure



MVC Structure in The Real World

So far it has been extremely basic. Let's begin to do things a little more seriously, and let's start with a proper structure.

On the right you can see what a real production-like structure may look like. It doesn't matter how easy or complex your application is, your tree should look similar to this. This is what we normally refer to as an MVC "inspired" structure.



MVC Structure

The MVC structure reflects what we technically call the “separation of concerns”.

An application is made up of many independent pieces of code that talk to each other to accomplish a given task.

- **Controllers** – handle the business logic, for example “adding a user” or “retrieving a resource”
- **Middleware** – are part of the application which support the operations, they are normally, generic. For example, routes management.
- **Models** – reflect the data models, at a database level (but not limited).
- **Public** – is where the static pages are hosted, e.g. HTML pages
- **Routes** – is where the endpoints are mapped.
- **Services** – is where third party libraries are kept, for example, controllers handle the logic of the user management, but ultimately, use a DB service to update the database.

MVC Structure

Let's look at a more real-world example.

Let's say that a user wants to create a new post on his Facebook timeline.

The back-end API will accept the request and match it to a **route**, from where the data will be passed on to the **controller**, which handles processing via a **model**.

The user will hit the **/timeline** endpoint (**ROUTE**) with a post message containing the content.

The route will pass the content to a Controller, in this case the **timelineManagement** controller.

The Controller will have the internal logic to handle the creation of a new post and will call the TimeLinePosts model (which refers to a database table) to update the Database.

As you can see, each part is very specific. This allows the system to be easily debugged. Also, it is not unusual to physically separate functions as well. Routing for example, is very light weight, so it can be handled on one small server.

Serving Static Content

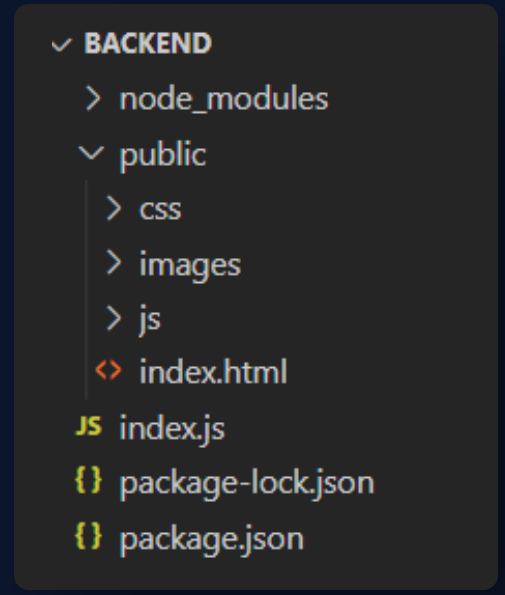
The most common and simplest thing to do in a back end is to serve static content. This is no brainer, and requires the least effort.

Create a folder in your express project, and simply call it **public**. Keep things tidy by creating folders for css, images and js inside the public folder.

Create a simple **index.html** file with a few lines of code just to test things out.

```
<html>
    <title>My first page</title>
    <body>
        <h1>Hello Friends</h1>
    </body>
</html>
```

Folder structure should look like this



Serving Static Content

Now let's add our first middleware, which will deploy the static content.

Anywhere in your code, after your app declaration, simply add the following:

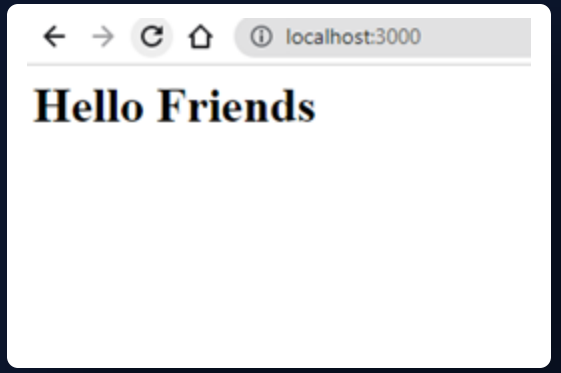
```
app.use('/', express.static('public'))
```

This will tell the system to fetch pages from the public directory on the root endpoint. Change the previous root endpoint mapping, it will not work now. Simply add a prefix to it:

```
app.get('/test', (req, res) => {
        res.send('Hello World!')
})
```

Stop your node server using control – c (or command C) and then restart. Go to your localhost:3000 and you should be able to see your new page being hosted.

Remember, index.html is a special page, it does not require to be called specifically. But if you want to create a page with a different name, you will need to call it specifically in the URL, for example, **localhost:3000/users**



Routes

Next we are going to move the routes outside index.js. It is convenient now, but once you have hundreds of routes, or even 10, it will be impossible to manage. This is why we need a specific place to handle the **routes**.

Start by creating a folder called routes and create a file called **myTestRoutes.js**.

We will move the test route in here. Let's create another one as well just to show all the intents. Remove the following code from the index.js file:

```
app.get('/test', (req, res) => {
        res.send('Hello World!')
})
```

A route is a single express component, and it is created out of the express Router module.

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

router.get('/test', (req, res) => {
  res.send('Hello World!')
})

router.get('/test2', (req, res) => {
  res.send('Second test')
})

module.exports = router;
```

As you can see, we don't use app anymore, but router, because this is a router object. Last, we export it. Exporting is how we make a particular object available to other code. If we didn't export it, we wouldn't be able to import it.

Routes

We now need to import the routes, and then bind them to the app using a unique prefix.

This has to be done for each set of routes that we create.

Once the changes are complete, restart your node server again.

At the top of the index.js file, import the routes:

```
const testRoutes =
require('./routes/myTestRoutes');
```

This will import the route, but has not been bound yet, you need to go through the middleware.

Add the following:

```
app.use('/mytest', testRoutes);
```

The route is now bound to the app. /mytest in app.use is a prefix, you can change it as you like. If you want to call those routes, you now need to call:

localhost:3000/mytest/test or

localhost:3000/mytest/test2

Section 3 :

Design a Back-End Service

Tips and Tricks

You may have noticed by now that you need to stop and restart every time you make a change. There are a few tricks to overcome this. You can use a package called **nodemon**. Nodemon monitors your code for changes and restarts your server every time you save.

To run nodemon, first install the package by doing **npm install -g nodemon**

Then, instead of running **node index.js** to start your app, run nodemon index.js instead (you should change your start script in package.json).

Another best practice step is to include a **.gitignore** file at the top level of your app. This is a config file that tells Git to ignore certain files from your repository. We want to use it to ignore everything in the **node_modules** folder, since these are the third-party dependencies for our app that we haven't created ourselves.

A Server-Side Calculator

Client-Server architecture is not just a way to make things “more connected”, but also a way to make things work securely. All users (clients) have full access to their machines, meaning that they can change code at run time. For example, you will never want an application transferring money, to be able to do that on the client side.

So all non-trivial tasks, we do on the server through a transaction. The simplest example is time. Imagine that you have bidders on your eBay Air Jordan shoes. If the time was controlled on the client side, it would be very simple to cheat. Instead, the server is the only ground truth.

So let’s make a calculator service, following some of the work we did in Module 4.

A Server-Side Calculator

Our back-end Calculator is designed as a service, this means that it is completely independent of the client side implementation. Many different websites can use it, each with their own skin or themes. Because the service is only data, it doesn't care about who will utilize it, and on the other side, the client only sees a URL and a list of requirements.

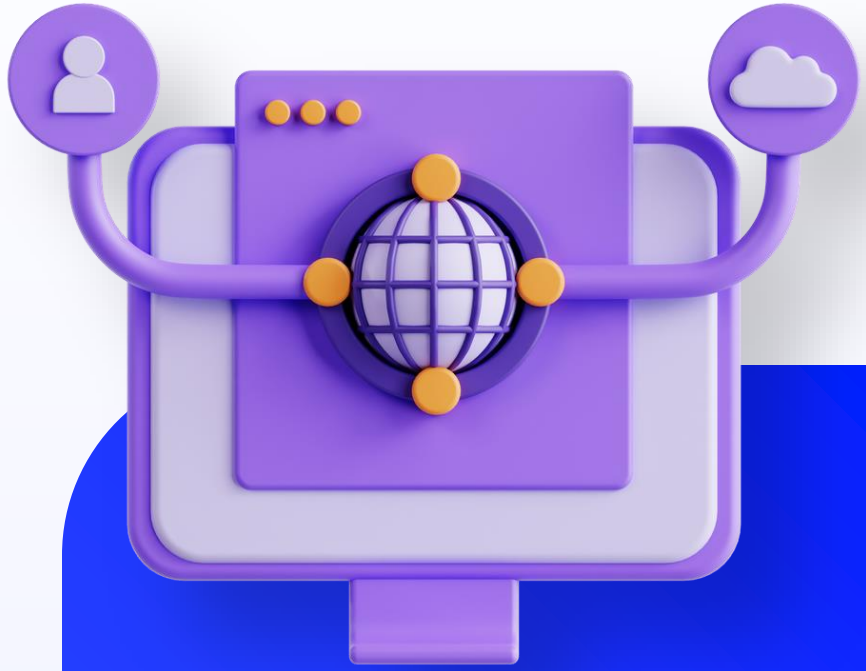
Our calculator is going to be very simple to begin with. It only has one function, which is to add 2 numbers, so the user will need to pass these as arguments, or parameters, for the request.

Passing Data Between Client and Server

There are two ways of passing data via a HTTP Request (actually there are more, but we will focus on these two): through the **URL** or through the **body**. Normally the URL is less secure as the parameter values are easily visible, but it is easier to implement.

To send a request with data, we make a GET call to a back-end service. Remember, a back-end endpoint has two objects bound to the function: **req** and **res**. req is the HTTP Request, which contains all the parameters passed by the client.

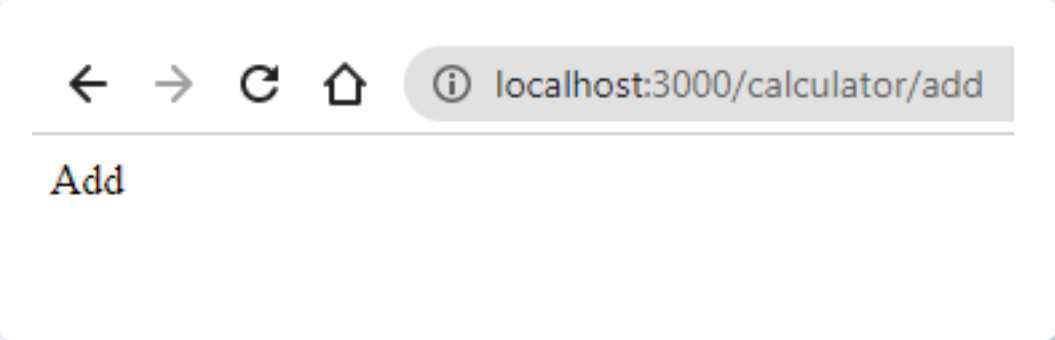
Let's create a new set of routes and call it **calculatorRoutes**, bind it to the top level **/calculator** URL in index.js, and create an **add** route.



Routes

Add the code on the right to **calculatorRoutes.js** and **index.js**.

You should now have a simple add route, accessed via the http://localhost:3000/calculator/add URL, that we can use to start building functionality for our server-side calculator.



calculatorRoutes.js :

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

// new route for adding two numbers
router.get('/add', (req, res) => {
    res.send('Add')
})

module.exports = router;
```

index.js :

```
// import all calculator routes (up the top)
const calculatorRoutes =
require('./routes/calculatorRoutes');


// map the calculator routes to our app
app.use('/calculator', calculatorRoutes);
```

Routes

It is time now to get some data out of this route.

Use the following URL this time, to send data:

http://localhost:3000/calculator/add?num1=4&num2=10

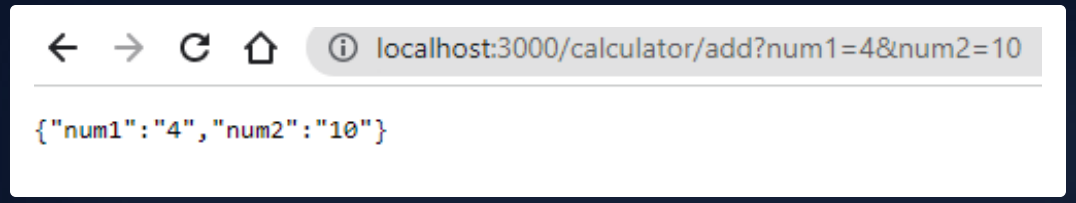
- **?** indicates the beginning of **query parameters**.
- Next are the parameters, **num1** and **num2**, with their values after the =.
- Multiple parameters are separated by **&**

This request will not cause any problems, the server is simply not going to handle the data. To see the data we need to make some changes to the route.

Modify the route function to send back the query parameters instead of the 'Add' string:

```
router.get('/add', (req, res) => {
    console.log(req.query)
    res.send(req.query)
})
```

Refresh the page, you can now see the results in the page and also in the server console.




```
[nodemon] restarting due to changes...
[nodemon] starting `node .\index.js`
Example app listening at http://localhost:3000
{ num1: '4', num2: '10' }
```

Routes

req.query is how we access GET data sent via the request query parameters.

Now we can use that data to achieve our addition goal.

Let's change the route using the code to the right, so that we can handle this operation properly.

Test the updated route with several different values for num1 and num2.

```
// functional add route performing addition on
request parameters
router.get('/add', (req, res) => {
    let number1 = parseInt(req.query.num1);
    let number2 = parseInt(req.query.num2);
    let sum = number1 + number2
    console.log(sum)
    res.status(200)
    res.json({result:sum})
})
```

A few extras.

parseInt – by default, get arguments are strings. In order to add two numbers, they need to be numbers. This is why we parse them.

res.status - we use this to set the value to 200 (Complete), this is used to give extra info to developers.

res.json – instead of send, we use this to send data in json format instead of just strings.

Front-end Fetch

In module 4, you created a simple calculator. Here is some very basic code to show how you can make a request to our backend calculator service using the fetch method.

Fetch is an inbuilt browser JS method for getting data from a webserver.

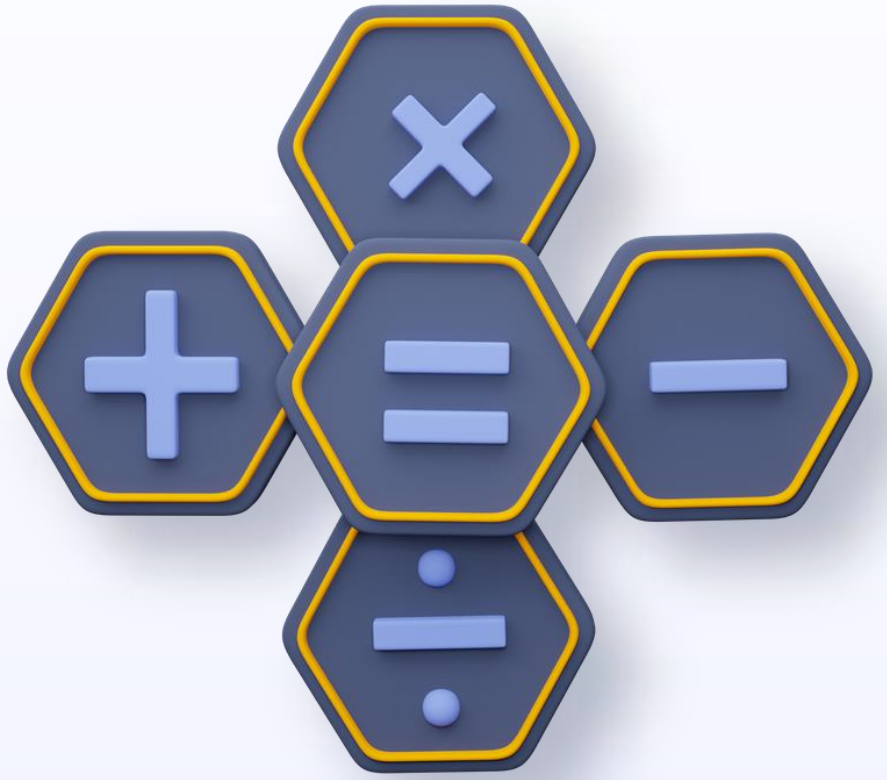
*Save the code as **calculator.html** in your **public** folder, and view it using http://localhost:3000/calculator.html.*

```
<html lang="en">
<head>
  <title>Calculator Example</title>
</head>
<body>
    <div>Result
        <span id="result"></span>
    </div>
    </body>
    <script>
        let num1 = 2;
        let num2 = 4;
        fetch(`/calculator/add?num1=${num1}&num2=${num2}`)
        .then(response => response.json())
        .then(data => {
            document.getElementById("result").innerHTML =
data.result;
        })
      </script>
</html>
```

Update this html page to use input fields for num1 and num2 and an ‘Add’ button to run the fetch and display the result from the server.

Exercise 2

- Based on the ‘add’ route demonstrated in the slides, create routes for ‘subtract’, ‘divide’, and ‘multiply’ to manage the four core mathematical operations.



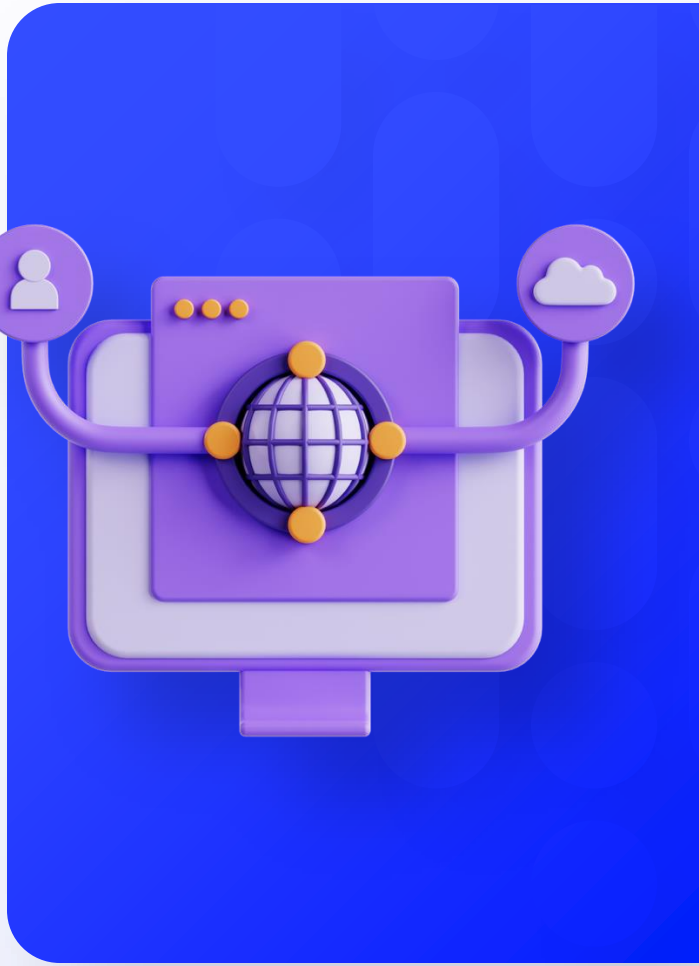
Exercise 3

- Extend your calculator.html so that it allows the user to access all 4 server-side calculator routes from Exercise 2 (Add, Multiply, Divide, Subtract) via a basic front-end calculator interface.

Passing Data Between Client and Server

So far we have seen how to pass data from the client to the server through the request via the **query parameters**. The HTTP Request can also support **dynamic request parameters**, as well as data sent via the request headers and the **request body**.

- **Dynamic request parameters** are normally used to identify and request a specific item by its unique ID. For example, we may want detailed data from the server about a specific product from a list of products, or a specific user from a list of users.
- **Request headers** contain more information about the resource to be fetched, or about the client requesting the resource. They can be used for authentication and caching.
- The **request body** is used to send more complex or large data to the server via a **POST** request, for example after a form submit, in order to create a new resource.



Dynamic Params

In this example we are adding support for a dynamic request parameter representing a user id.

When we send a request such as http://localhost/users/3, the '3' is dynamically matched against this route, and stored in a param called id. *Try it out using different ids!*

```
// import all user routes (up top in index.js)
const userRoutes = require('./routes/userRoutes');

// map the user routes to our app
app.use('/users', userRoutes);
```

```
// Add to new file - routes/userRoutes.js
const express = require("express");
const router = express.Router();

const users = [
    {id: 1, name: 'Anthony Albanese', country: 'AU'},
    {id: 2, name: 'Joe Biden', country: 'US'},
    {id: 3, name: 'Chris Hipkins', country: 'NZ'},
    {id: 4, name: 'Lee Hsien Loong', country: 'SG'}
]

// Dynamic request param endpoint - get the user matching
the specific ID ie. /users/3
router.get('/:id', (req, res) => {
    console.log(req.params)

    let userId = req.params.id; // 'id' will be a value
matching anything after the / in the request path
    let user = users.find(user => user.id == userId)

    user ? res.status(200).json({result: user})
         : res.status(404).json({result:
              `User ${userId} not found`})
})

module.exports = router;
```

Request Headers

The request headers can also be accessed from the req parameter in a route mapping.

req.headers is an object that includes information about the request, including session data, cookies, the type of agent sending the request, any particular caching protocol, and potentially also things like authentication tokens.

Add the new route into userRoutes.js ABOVE the previous dynamic path and test it out.

Why does it work differently if moved below?

```
// Add this new route into userRoutes.js:

// get information about this request from
the headers
router.get('/headers', (req, res) => {
    console.log(req.headers)

    res.json(req.headers)
})
```

Access this route via http://localhost:3000/users/headers from several different browsers (ie. Chrome, Safari, Firefox) and see the slightly different headers in the returned response.

Request Body

To send more complex or complete data from the client to the server, we need to use a **POST** call instead of GET, in order to populate the **body** of the request.

Express supports methods that correspond to all HTTP request methods: get, post, and so on.

In order to parse json data sent in the request body, we need to use the built-in **express.json()** method in **index.js** BEFORE mapping any routes files.

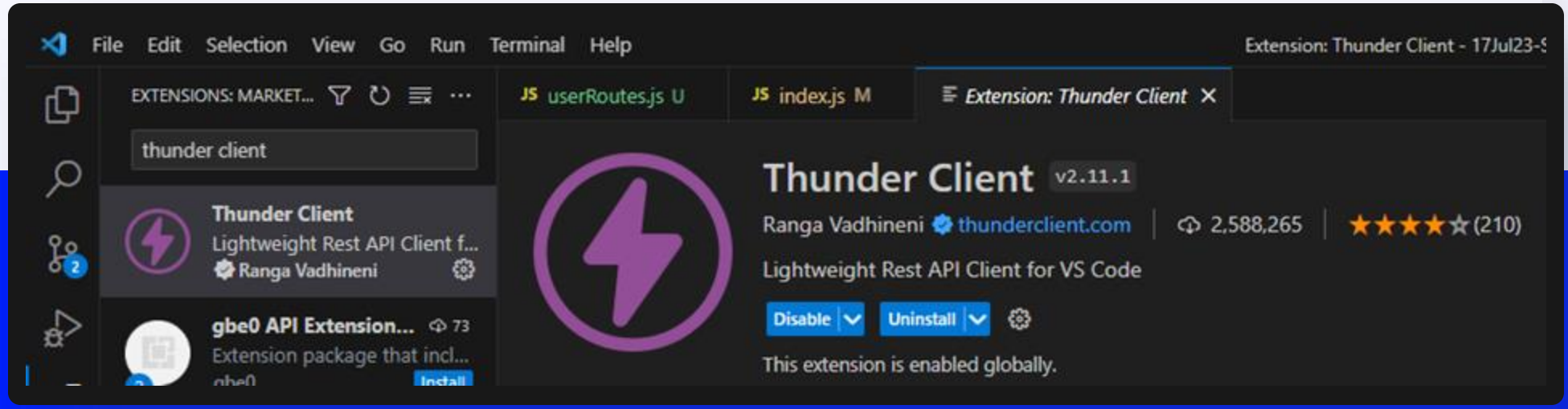
```
// Add this new route into userRoutes.js:

// a POST request with data sent in the body of the request,
representing a new user
router.post('/', (req, res) => {
    let newUser = req.body; // first update index.js
    console.log(newUser)

    // we can add some validation as well
    if (!newUser.name || !newUser.country) {
        res.status(500).json({error: 'User must contain a
name and country'});
        return;
    }
    else if (!newUser.id) { // if no ID, generate one
        newUser.id = users.length + 1;
    }

    // if the new user is valid, add them to the list
    users.push(newUser)
    res.status(200).json(newUser) // return the new user
});


// ADD TO index.js ABOVE ALL app.use CALLS
// parse requests of content-type - application/json
app.use(express.json());
```

Passing Data Between Client and Server

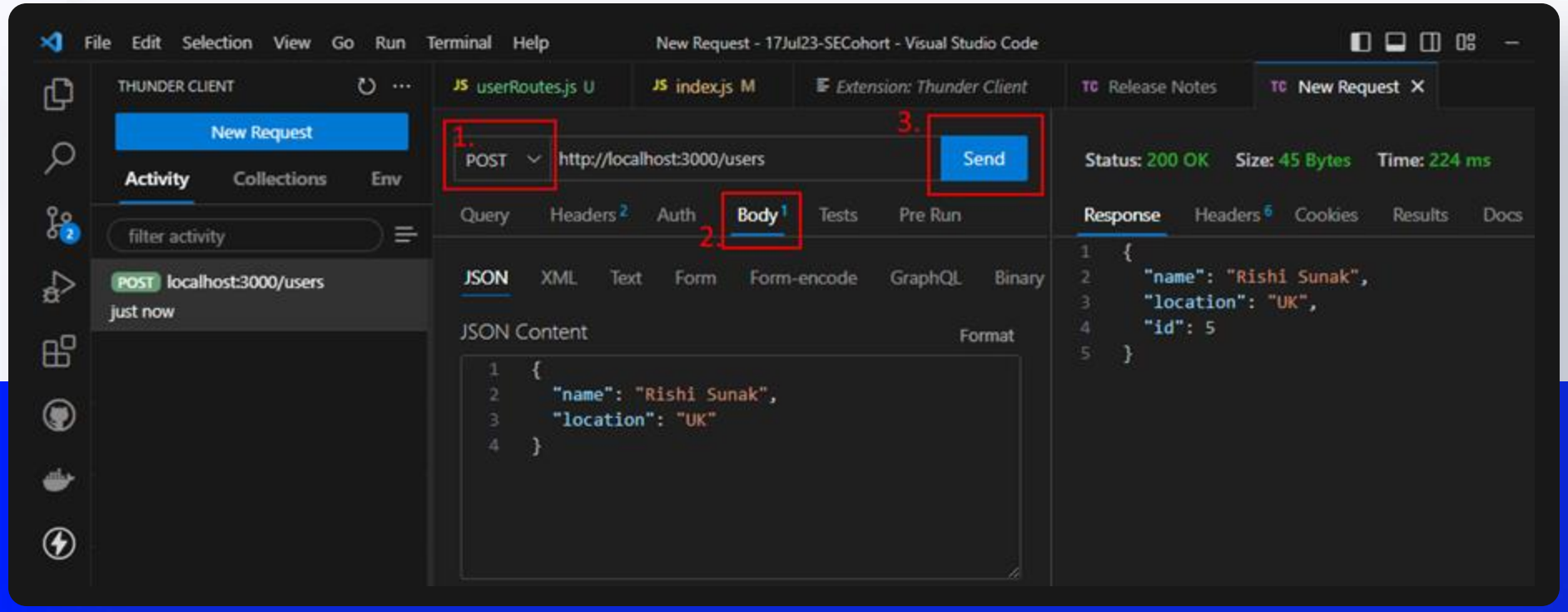
So far we have been using a browser to send GET requests to our back-end server by simply typing in various URLs. POST requests are more complex, however, and need to be sent differently. Some common options include:

- A **HTML form** with `<form action="/users" method="POST">` and input fields corresponding to the fields in the body
- Using **axios** or **fetch** to send a POST request to the server including the request body
- Using a tool such as **Postman** or the **Thunder Client** extension in VSCode.

Thunder Client is the easiest to setup. Simply install it from the extensions menu:



Passing Data Between Client and Server



Now we can easily change the request method to **POST**, configure the data to be sent in the **Body**, and **send** our request to the server. The response is then displayed to the right.

Exercise 4

Use the Express App template in **Exercise4/m5lab4_expressapp** from Google Drive. The friendRoutes.js file details 4 tasks to give more understanding on setting up different types of requests via routes:

- Part 1: Add support to the 'filter' endpoint for a new query parameter 'letter' which filters friends by starting letter
- Part 2: Modify the 'info' route to only return the **user-agent**, content-type and **accept** header data
- Part 3: Modify the dynamic GET route to return a single friend object matching the dynamic 'id' request parameter
- Part 4: Complete the PUT route which will update data for an existing friend

Test each of the above with different values, and include some basic data validation.

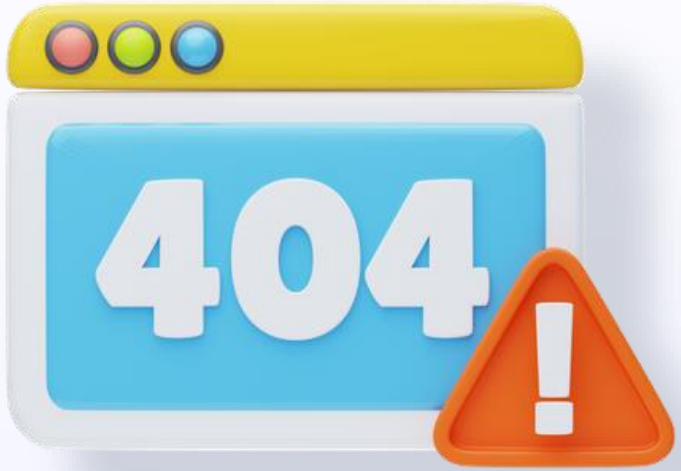
Error Handling

As you can see, our calculator is now getting more complex, and we are still doing extremely simple things.

Now, try to send something which is not a number, see what happens?

This is because we have not done any error handling. Error handling should check to make sure that all information sent in the request is valid and conforms to any rules, such as mandatory fields and correct data types. Messages regarding any invalid data should be sent back in the response so that the client is aware and can make fixes on their end before re-sending a valid request.

We will look into this in more detail in future modules.



Controllers

Our separation of concerns is not yet complete. We mentioned before that routes should only do “routing” and have no logic involved. They simply define the available routes, then pass the data and actual route operation on to the controllers.

Similar to our previous work, let’s create a folder called **controllers** and then create a **calculatorController**.

Note

We are moving into chained calls. This is something very complex, and understanding will come with experience. At a later stage, we will look into better ways of managing chains. For now we have to live in the “**callback hell**”.

Basic Controller

As you can see on the right, we have moved all the addition logic out of the route, and into the controller.

Why is it callback hell? Because the same request/response data is passed from one function to another until the end. This is still quite small, but it can easily grow in complexity.

calculatorController.js

```
const addNumbers = (req, res) => {
    let number1 = parseInt(req.query.num1);
    let number2 = parseInt(req.query.num2);
    let sum = number1 + number2
    console.log(sum)
    res.status(200)
    res.json({result:sum})
}

module.exports = {
    addNumbers
}
```

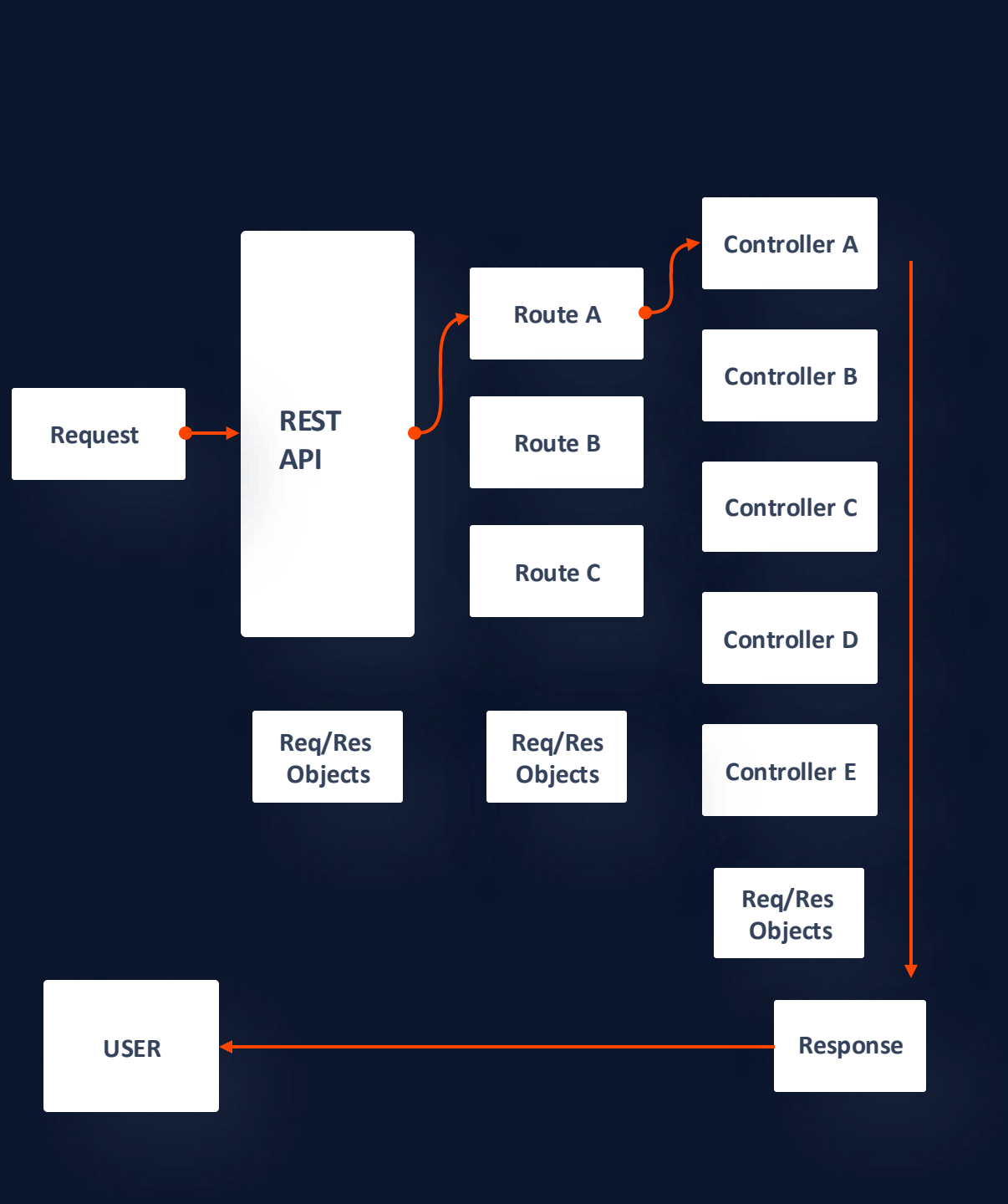
calculatorRoute.js

```
const express = require('express');
const calculatorController =
require('../controllers/calculatorController')
const router = express.Router();

router.get('/add', (req, res) => {
    calculatorController.addNumbers(req,res)
})

module.exports = router;
```

Anatomy of Controllers

Controllers take care of the business logic. This is where data should be “treated” before a response is sent back to the user.

As you can see from the diagram, data is always passed down as req/res objects. After the request data is parsed and acted on, the controller sends a response through the Response (res) object when operations are complete.



Exercise 5

- Expand on the previous exercises and update your application to use controllers instead.

Section 4 :

Unit Testing

There are quite a number of testing frameworks designed to automate and streamline unit testing across a codebase. These may approach testing from different perspectives, eg: **behaviour-driven**, **error-driven** or **test-driven** testing, but all aim to ensure reliability by finding problems early.

In this section we will look at how to set up some automated tests for our backend API using **Jest**, a popular test-driven JS testing library.

Unit Testing

Unit testing is the first stage of testing for any application. It is designed to verify the correctness of each individual unit of code, whether that be a function, object or API endpoint.

If you've written some code and checked its performance by running it, or used console messages to verify data or results, you've done a basic form of unit testing. Beginners often rely on console.log for this, which is fine initially but becomes impractical as code complexity increases.

Unit testing automates this process by writing and running a series of tests that verify the output of each unit of code. Test-driven development focuses on writing the test for each unit of code while or before writing the unit itself. Any code improvements (or refactoring) can then be easily checked to ensure the changes have not introduced any errors or bugs, as long as the tests still pass.

Testing Concepts

Most testing frameworks make use of some common patterns and concepts. Typically we want to define or describe a number of individual tests, and the expected values that each test should produce. To pass, the test needs to produce the expected result - otherwise it fails.

Creating a unit test therefore requires a few key steps:

1. **Describing** the feature/unit our test is calling (so that we can track, identify and fix any failures)
2. **Executing** the code unit
3. **Comparing** the **actual** result to the **expected** result.

The testing framework will then handle tracking all of the test results and reporting a summary of the passes and failures. Often we also need to simulate (or **mock**) some third-party dependencies that the code unit relies on, in order to test our code in isolation.

JS Testing Framework - Jest

Jest is a widely-used JavaScript testing framework designed to ensure correctness in any JavaScript codebase. It allows you to write tests with an approachable, well-documented and feature-rich API that gives you results quickly.

Those tests can then be automated to ensure that your code continues to function as expected every time a change is made or a new feature is added.

Basic Jest Example

To get this example working, we just need to install Jest as a dependency from the NPM:

```
npm install jest
```

The test and expect functions are globally imported when Jest is installed, so we don't need to explicitly require them. Just create the two files with the contents as shown (right).

To run it, add a new test script to package.json:

```
"scripts": {
   "test": "jest",
   "start": "node index.js"
 },
```

```
// Unit to be tested - the square function.
// Stored in a file called square.js - from here we can
// both test and use the same function, by exporting it

function square(a) {
  return a * a;
}
module.exports = { square };

-----------------------------------------------------

// Unit test - stored in square.test.js
// First we import the square function
const { square } = require('./square');

// Then we test it by describing the test, running the
// code, and comparing expected vs. actual results
test('square 5 to get 25', () => {
  expect(square(5)).toBe(25);
});

// ++ Try creating a second function in square.js called
// squareRoot, then test that too!
```

Basic Jest Example

After adding the **test** script to **package.json**, we can now run our tests using:

```
npm test
```

```
PS D:\IOD\SE\MODULE 5\myapp> npm test

> myapp@1.0.0 test
> jest

 PASS  ./square.test.js
  √ square 5 to get 25 (2 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.407 s, estimated 1 s
Ran all test suites.
```

How does Jest know what's a test file?

First: any files in a directory with the name **`__test__`** are considered a test. If you put a JavaScript file in one of these folders, Jest will try to run it.

Second: Jest will recognize any file with the suffix **`.spec.js`** or **`.test.js.`** It will search your entire repository for any files matching this pattern, and include them when the **`npm test`** command is run.

Jest - Expectations & Matchers

The `expect` function in our basic example is an essential part of unit testing in Jest. It returns an **expectation object** which is most commonly used to call a **matcher** function - in our case, `toBe`.

The matcher function will record either a pass or fail result by **comparing** the value passed to `expect` with the value passed to the matcher, according to the type of comparison we need to do. In addition to `toBe`, which performs an **exact equality** test, there are a number of others:

Equality	toBe, toEqual (recursively compares objects/arrays)
Truthiness	toBeNull, toBeUndefined, toBeDefined, toBeTruthy, toBeFalsy
Numbers	toBeGreaterThan, toBeGreaterThanOrEqual, toBeLessThan, toBeLessThanOrEqual, toBeCloseTo
Strings	toMatch
Arrays	toContain
Exceptions	toThrow

API Testing with Supertest

The basic unit tests used so far are ideal for testing derived data functions. Modern back-end systems are commonly more complex than this, and often use **HTTP APIs**.

Supertest simplifies testing these HTTP requests, eliminating the need for a running server in a Node.js application, to test requests such as GET, PUT, and POST with only a few lines of code. Supertest is a Node.js library that allows developers and testers to write automated tests for routes and endpoints. It is designed to work within a testing framework such as Jest.

```
npm install supertest
```

Run the above command to install Supertest as another dependency in our package.json file.

API Test Example

Before we can add tests to our calculator API routes, we need to make some changes to index.js, to support our Express app being used in both **test** and **development** modes.

First create a new file at the top level called **app.js**, which will handle creating and setting up the Express app with all routes.

index.js will then only need to import this app and start it on the right port, as all other setup is already handled in app.js.

No new code has been introduced yet, just some refactoring to isolate the app for testing.

```
// app.js - new file at top level
const express = require('express')
const app = express()
const port = 3000

// map all routes to the express app
const calculatorRoutes =
require('./routes/calculatorRoutes');
app.use('/calculator', calculatorRoutes);

// export the app
module.exports = app;
```

```
// index.js - updated version
// import the app
const app = require('./app');
const port = 3000

// start the app to listen on the right port
app.listen(port, () => {
    console.log(`Example app listening at
http://localhost:${port}`)
})
```

API Test Example

Next create a new file called **app.test.js** at the top level, using the code on the right.

This unit test will simulate a HTTP request to the GET /add endpoint of our server-side calculator (using **Supertest**), and expect the response to fulfil several conditions (using **Jest**).

There are several important things happening here, and we will break down each one on the following slides.

First, rerun the **npm test** command to see the code in action.

```
// app.test.js
// import supertest and the express app
const request = require('supertest');
const app = require('./app');

describe('Calculator Routes', () => {

  // generate some random numbers to test the calculator
  let number1 = Math.floor(Math.random() * 1_000_000);
  let number2 = Math.floor(Math.random() * 1_000_000);

  test('GET /calculator/add => sum of numbers', () => {
    return request(app)

.get(`/calculator/add?num1=${number1}&num2=${number2}`)

      .expect('Content-Type', /json/)
      .expect(200)

      .then((response) => {
        expect(response.body).toEqual({
            result: (number1 + number2)
        });
      });

  });

});
```

API Test Example

First, we import the Supertest library and store it in a constant variable (a **function**) called **request**. After we also import the express **app**, we can then make requests to the app's API endpoints (routes). This allows us to test our routes without having to actually start the app.

The **describe** function groups multiple unit tests together into a **test suite**, named using the string passed as the first parameter.

For the first step in the test suite, we generate two random **numbers** that we will use as **request parameters** when calling the various calculator operations.

```
// app.test.js
// import supertest and the express app
const request = require('supertest');
const app = require('./app');

describe('Calculator Routes', () => {

  // generate some random numbers to test the calculator
  let number1 = Math.floor(Math.random() * 1_000_000);
  let number2 = Math.floor(Math.random() * 1_000_000);
```

describe is automatically imported from Jest (the same as **test** and **expect**). The function passed as the second parameter typically contains one or more unit tests as well as any variables needed to run those tests.

API Test Example

The test function takes two parameters:

- A **string** describing what is being tested
- A **function** that runs the test and returns a pass/fail result.

Inside the test, we make a request to the app using `.get()` to simulate a GET HTTP request (`.post()` etc is also available). The string passed to `.get` should match the expected format, including request parameters.

We then use **expect** to check the content type and status code of the response.

```
// run a test with the given description
test('GET /calculator/add => sum of numbers', () => {

  return request(app)
.get(`/calculator/add?num1=${number1}&num2=${number2}`)
      .expect('Content-Type', /json/)
      .expect(200)
```

Aspects of the HTTP response, such as the content type and status code, can be tested before the full asynchronous operation has resolved with the results of the request.

API Test Example

Once the GET request has completed, we can also test the response sent back from the API to check the expected format and content.

We use the toEqual matcher for this, as it does a deep comparison between objects.

The object we pass to it needs to match the object sent from the controller via res.json for the test to pass.

Any unexpected responses that do not match will cause the test to fail.

```
.then((response) => {
        expect(response.body).toEqual({
            result: (number1 + number2)
        });
    });
```

This is a simple test, checking the response using equality. We can also build more complex tests, to test if the response is an object containing certain properties, and check property values using string/number/array matchers, and many more.

Check the Supertest documentation for more info.

Exercise 6

- Add unit tests for each of the calculator operations, and run them to ensure all routes are working successfully and returning the expected response.

Section 5 :

Object-Oriented Development

In object-oriented programming (OOP), we organise our code around **objects**, which are like containers that model both information (data) and actions (behaviour).

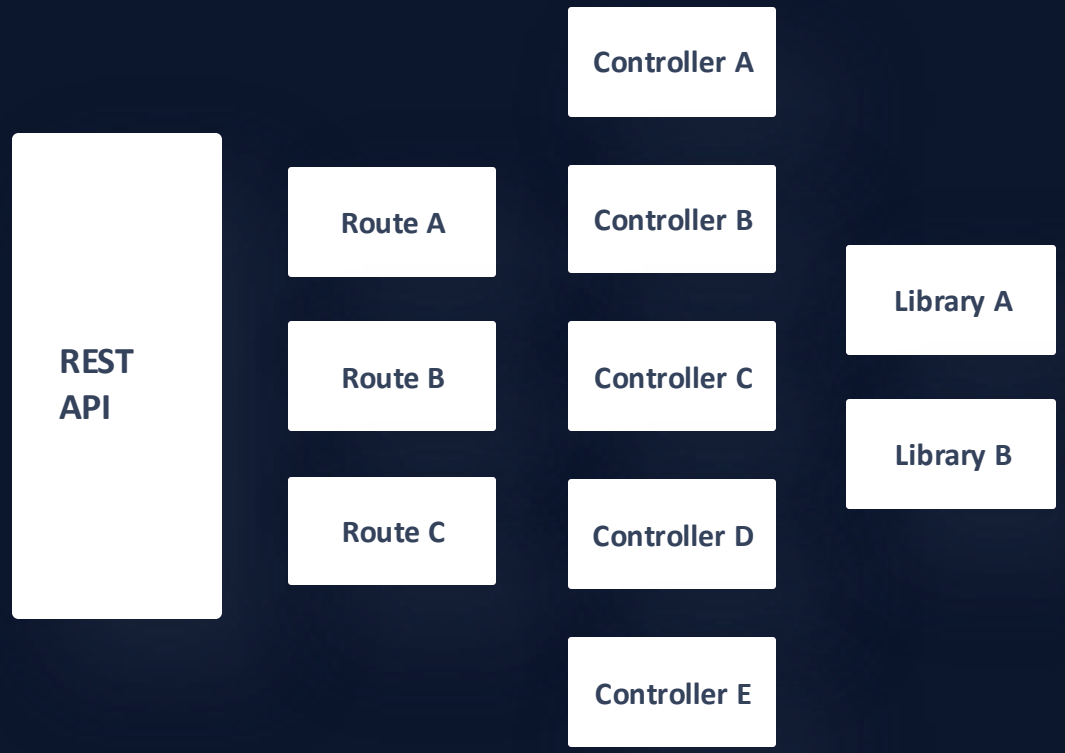
Instead of thinking about functions and steps to follow, we think about the objects we want to work with and what they can do.

Libraries

If we dig down deeper, we discover that even using controllers, we really shouldn't have the "how", rather the "what" only.

As the application grows, manageability is one key component, and that is done through the creation of libraries. Libraries are just small and independent pieces of code.

Let's make it more practical.



The new controller will now have no knowledge on how to run the operation, instead, it will make use of a library!

What are Libraries?

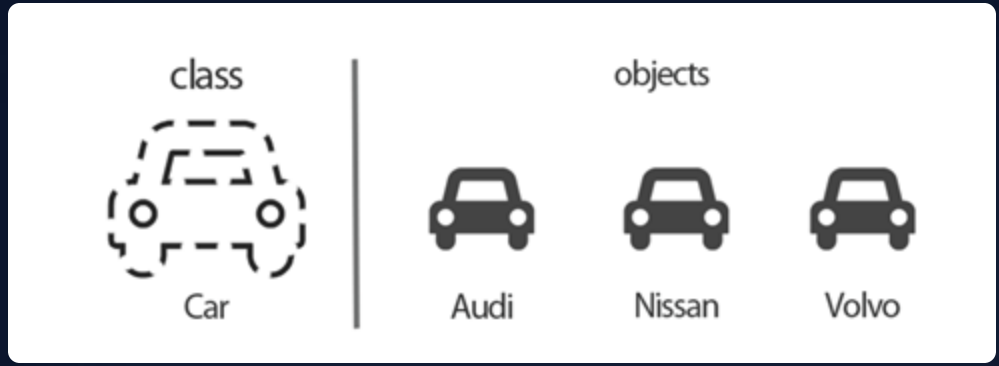
This time we step up the game, and start developing “real” applications, not just “web structures”. In this part we go back 30 years and we build a small calculator library that can be utilized by anyone. It is independent and has no knowledge of the web!

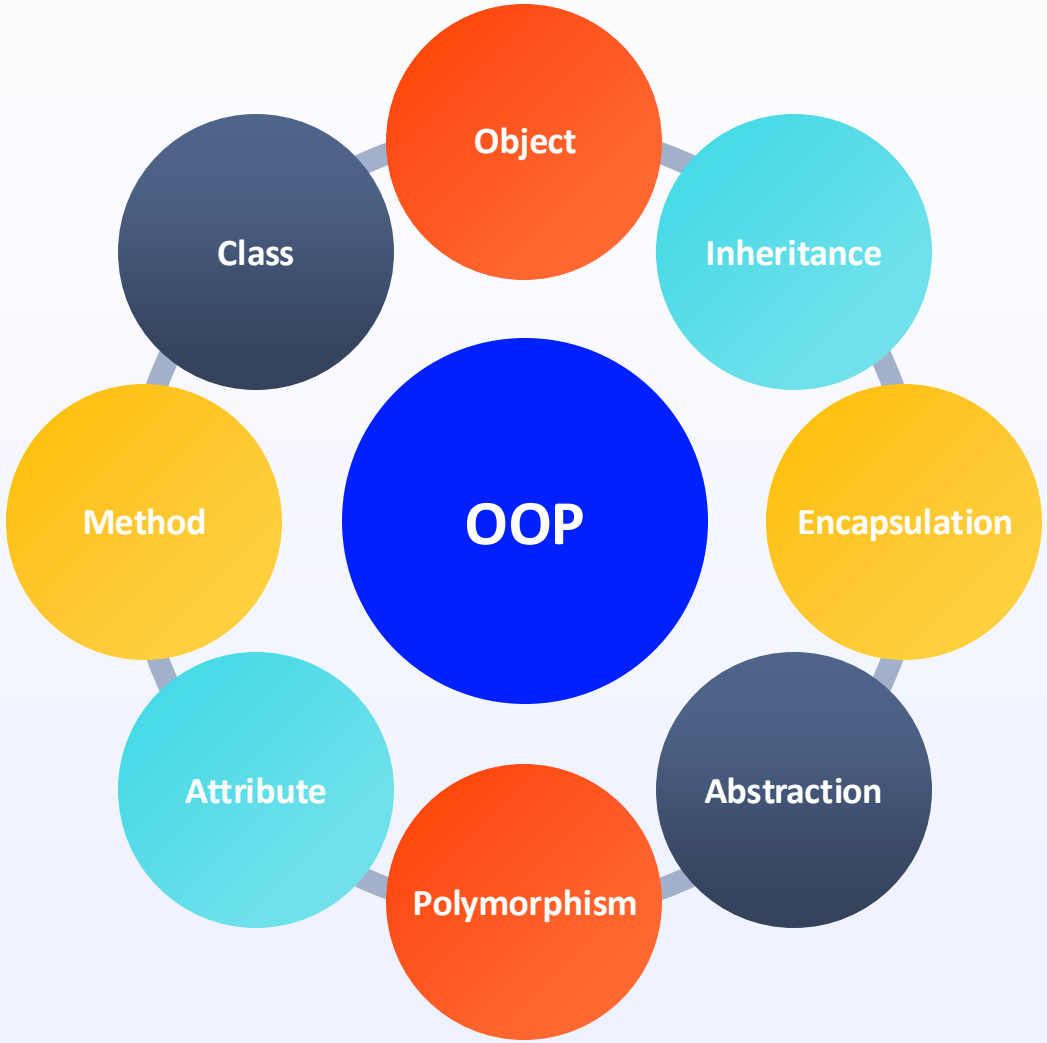
Let’s review what a library is.

A library is simply a program.

We use “object-oriented” development style to build libraries. This means that we use classes, also known as blueprints, to create objects.

For example, we design a blueprint of a car (class), but we can’t ride the blueprint. Instead we use that blueprint to create the car, we call this an object (instance of the class).





Object-Oriented Programming (OOP)

The details of object-oriented programming are too broad for this course. Concepts like polymorphism or advanced class design are topics which are part of larger courses of studies. Normally, one entire unit at university level is just about Object-Oriented Programming!

You can find out more by researching it, but fundamentally OOP is centred around data and object representation. We define templates and create instances of them which group both data and related methods together.

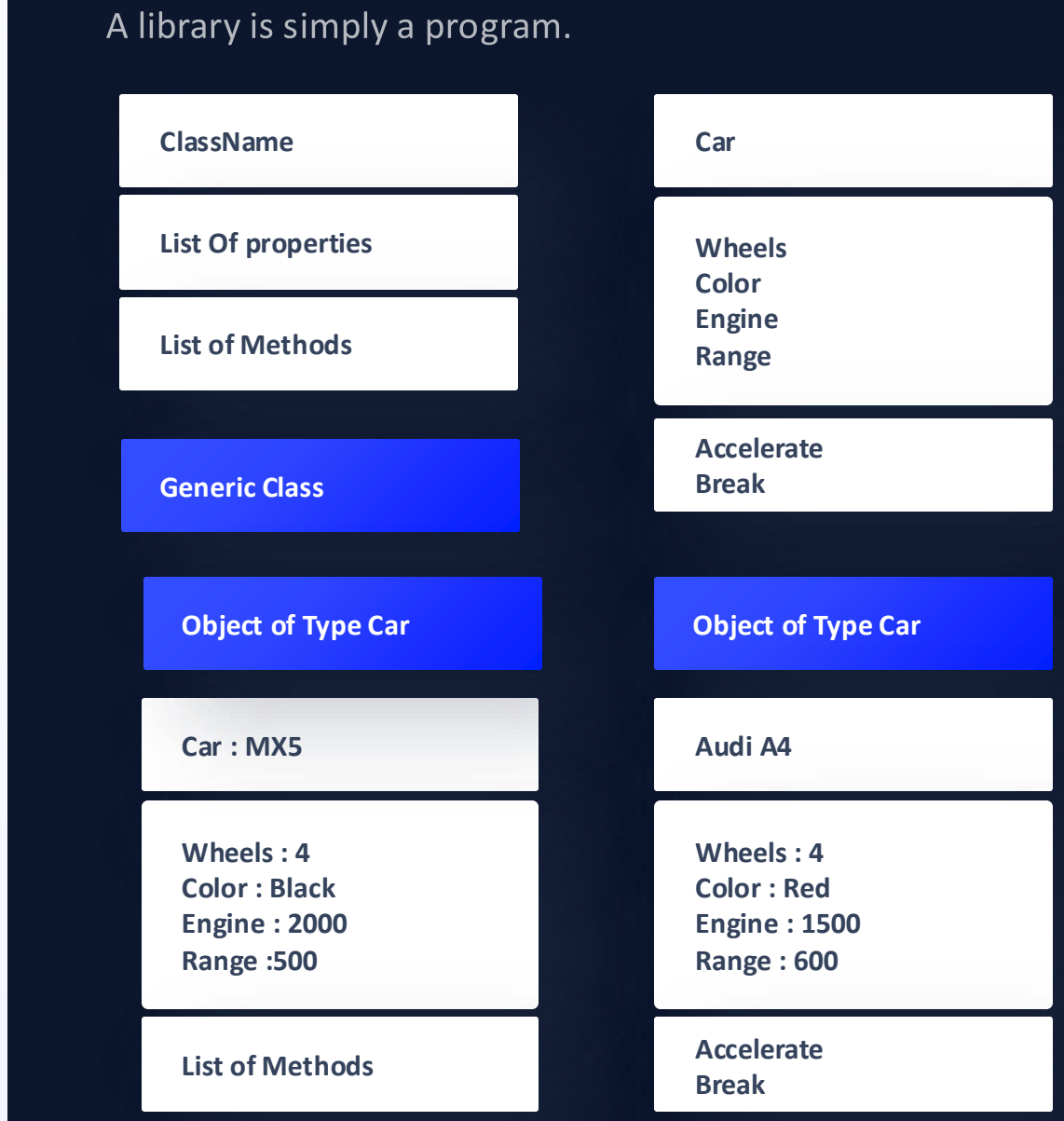
We will focus on simple libraries you can use everyday!

What are Libraries

An object is made out of a class: as mentioned, a class is a blueprint.

The most fundamental elements of a class are its **name**, **methods** and **properties**.

Just to make things more fun, “everything in Javascript is an object”, so methods and properties are all parts of the object and seen in the same way.

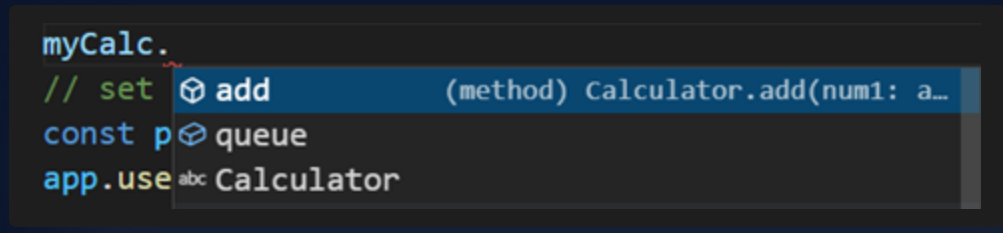


Class Visibility

Classes are also defined by their visibility. We don't want to show everything to user, so we create some methods and properties which are visible inside but not outside the class. Those that are visible outside are called **public**, those not visible, **private**. When we say invisible, it means they cannot be called directly.

A **public** method can call a **private** method internally.

```
class Calculator {
    constructor() {
    }

    #log = () => { // private method
        console.log('test')
    }

    add(num1, num2) {
        this.#log(); // public method calling private method
        const value = num1 + num2
        return value;
    }
}
```



Take this as an example. The # indicates a private method. It cannot be called directly, but the add method can instead call it. Even in the editor, the #log function is not visible.

Class Creation

JavaScript is not hard typed, meaning it has no forced rules around data types (TypeScript adds these rules). Instead of rules, JS uses several “conventions”.

A class has a constructor, which is a method called when we want to create an instance of that class (object).

Properties are defined in the Constructor and methods are made available in the rest of the code.

A class has a name, and by convention, starts with a Capital letter, and has the same name as the file holding it.

```
class ClassName {
    constructor(data) {
      this.data = data;
     this.someProperty;
    }
    #testMethod() {
      // private method
    }
    testMethod() {
      // public method
    }
  }
```

Class Structure

As an example, this is the Calculator Class.

We have a constructor that creates a unique ID based on a timestamp (of the moment it was created), then there are two functions, one private and one public. We pass data to the public one, and when the operation is complete, it will call the #log method (private) to log the result to the console.

Also, the module.exports at the bottom is to “make it available” as an ES6 importable, this is simply a requirement, there is no great magic behind it.

libraries/Calculator.js

```
class Calculator {
    constructor() {
        this.id = Date.now()
    }

    #log = (value) => {
    console.log(`[Calculator :${this.id}]:${value}`)
    }

    add(num1, num2) {
        const value = num1 + num2
        this.#log(value)
        return value;
    }
}

module.exports = Calculator
```

The Date.now() method is used to get the epoch time, we can use this as a simple but effective ID, as long as things are generated at least one millisecond apart.

Class Instantiation

Our blueprint is ready, so we can now make as many calculator library objects as we want.

We use the **new** keyword followed by the class, this reads as “create an object named **myCalc** from the class **Calculator**”, meaning that we can potentially go and create many more objects just changing the name.

As per every library, similarly to an NPM package, we need to import it.

```
const Calculator =
require('./libraries/Calculator');
let myCalc = new Calculator()
myCalc.add(3,4)
```

Potentially we could do this :

```
const Calculator =
require('./libraries/Calculator');
let myCalc1 = new Calculator()
let myCalc2 = new Calculator()
myCalc1.add(3,4)
myCalc1.add(4,2)
myCalc2.add(5,4)
```

Class Integration

We now have the class, we have the controller, and we have the route. Let's simply put them together.

You can see on the right how this has taken place, you may think "this is actually more work" - it is - but as engineers we need the holistic view, and we think about the future.

This separation of concerns is important for supporting greater complexity and maintainability as our app grows.

We went from this :

```
const addNumbers = (req, res) => {
    let number1 = parseInt(req.query.num1);
    let number2 = parseInt(req.query.num2);
    let sum = number1 + number2
    console.log(sum)
    res.status(200)
    res.json({result:sum})
}
```

To this :

```
const Calculator = require('../libraries/Calculator');
let myCalc = new Calculator()

const addNumbers = (req, res) => {
    let number1 = parseInt(req.query.num1);
    let number2 = parseInt(req.query.num2);
    let sum = myCalc.add(number1,number2)
    res.status(200)
    res.json({result:sum})
}
```

Uncle Bob says …

The difference between good and bad developers is not whether or not they can solve the problem. The difference is whether or not, people can read their code.

If you don’t apply separation of concerns, it becomes extremely difficult to try and track where problems are or add new features, especially if you are reading someone else’s code, or your code from years ago.

Be nice to future you, write good code!

If you want to know more about this, watch any video online about legend **Uncle Bob and Good Code!**

Exercise 7

- **Part 1** : Expand your application to use a Calculator library that takes care of the calculations and integrate it in your code.
- **Part 2** : Change the library so that you can generate a random number to be used as the ID, instead of the time. This way it will be almost impossible to have two of the same objects with the same ID.
- **Part 3** : Create a generic library for logging - pass a message to be logged, this will contain at least the ID of the caller, and the result. Log to the console every call made.

Section 6 :

Swagger

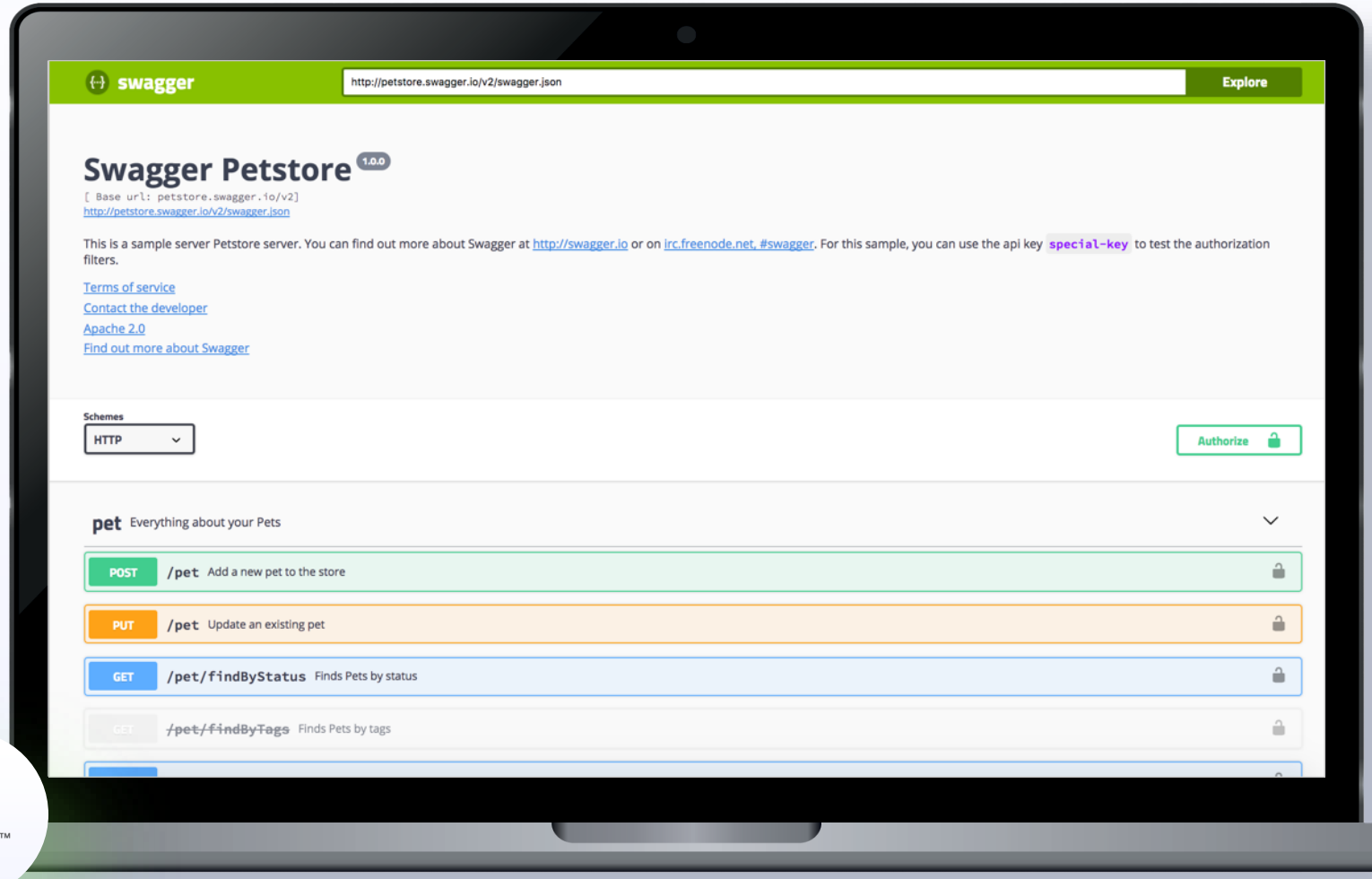
Swagger is a powerful yet easy-to-use suite of API developer tools for teams and individuals, enabling development across the entire API lifecycle, from design and documentation, to test and deployment.

In this section we will see how it can be used to take the manual work out of API documentation, by generating and visualizing API docs.

Time to Swag …

In order to step up and make our back end service easily available to others, we need to learn to “present”. Presenting in IT means to expose our work. One great way is Swagger!

Swagger is an open-source software framework backed by a large ecosystem of tools that help developers design, build, document, and consume RESTful web services.



Apply Swagger

Swagger is very simple, but it can grow exponentially in complexity. We are going to make a very simple integration.

The first thing to do is install swagger as a dependency. Then we will need to configure it.

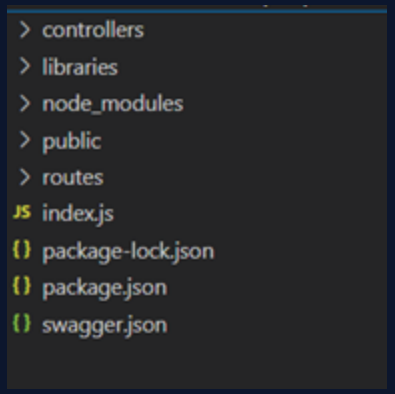
Use the following to add swagger as your dependency.

```
npm i swagger-ui-express -S
```

This will add swagger as a dependency, if you now check your package.json you should see the following.

```
"dependencies": {
  "swagger-ui-express": "^4.3.0"
```

Create a new file and call it **swagger.json** in the root of your project.



Swagger Settings

Copy and paste the following into swagger.json.

This is your Swagger definition, it describes all the services available in your back-end web service and generates the documentation.

You can customise it as you like - try changing the title and description to begin with.

```
{
   "swagger": "2.0",
   "info": {
     "version": "1.0.0",
     "title": "My Calculator Project",
     "description": "My User Project Application API",
     "license": {
       "name": "MIT",
       "url": "https://opensource.org/licenses/MIT"
     }
   },
   "host": "localhost:3000",
   "basePath": "/",
   "tags": [
     {
       "name": "Calculator",
       "description": "API for Calculus in the system"
     }
   ],
   "schemes": ["http"],
   "consumes": ["application/json"],
   "produces": ["application/json"]
}
```

Swagger Setup

Next we need to add the Swagger service to our server, so let's modify our index.js to include it.

Somewhere at the top of your code, add the following. This will import swagger, create a swagger document and create a route specifically for it.

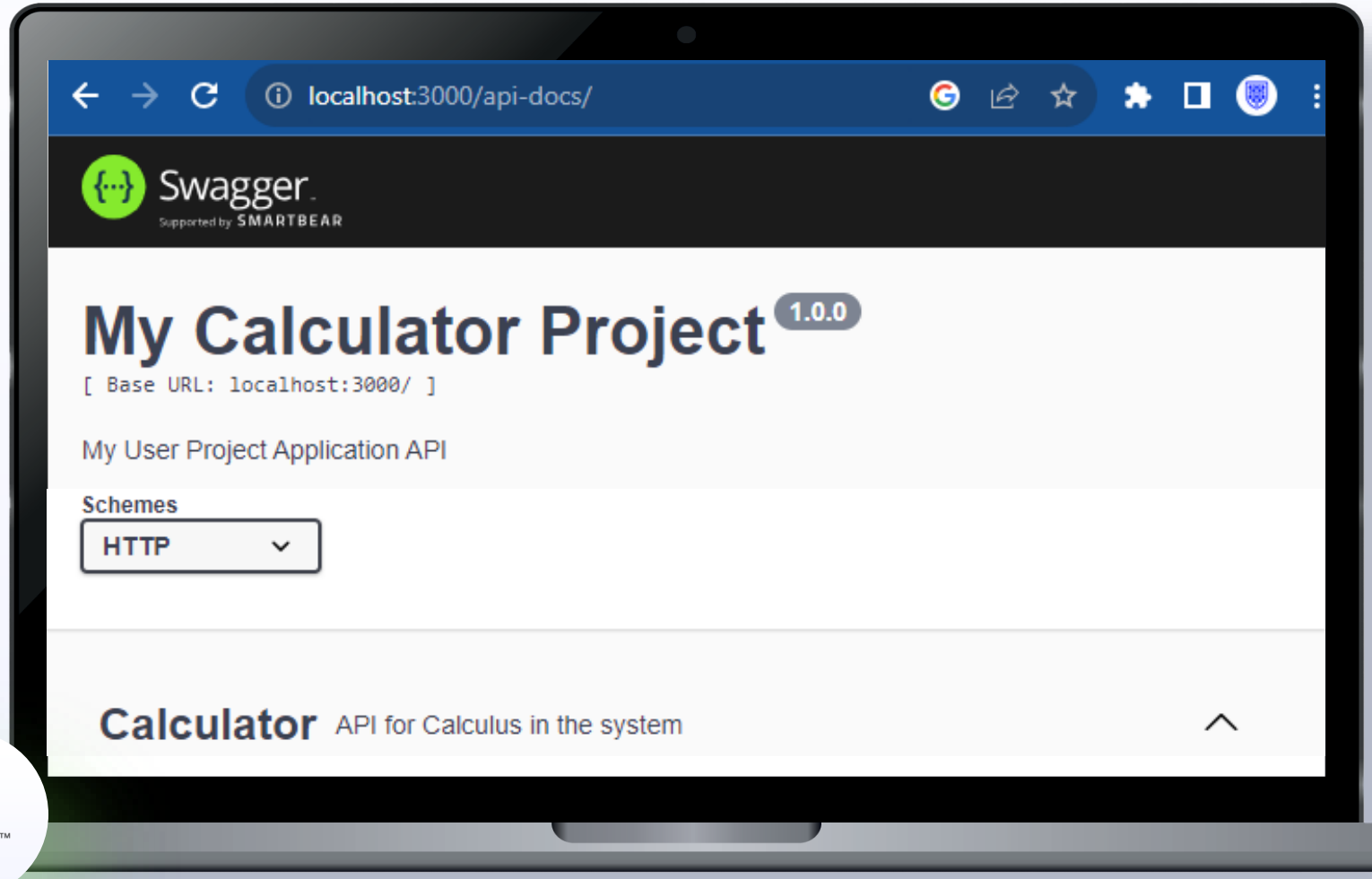
```
const swaggerUi = require('swagger-ui-express');
swaggerDocument = require('./swagger.json');
app.use(
  '/api-docs',
  swaggerUi.serve,
  swaggerUi.setup(swaggerDocument)
);
```

Reload your page and go to localhost:3000/api-docs

Swagger Result

If everything went well, this is what it should look like. You can try it, but it will not be very exciting. We have not configured the API requests yet.

Remember, this is only “documentation”, we are not writing any code here.

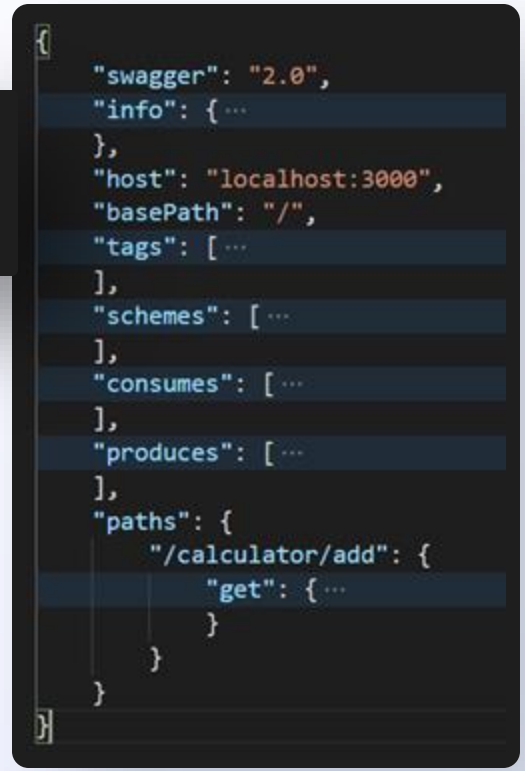


Swagger Paths

Add the “paths” code on the right to swagger.json. Right after “produces”.

It should look like this (code snippets are closed).

```
"produces": [
    "application/json"
],
```



This is the code that defines the paths.

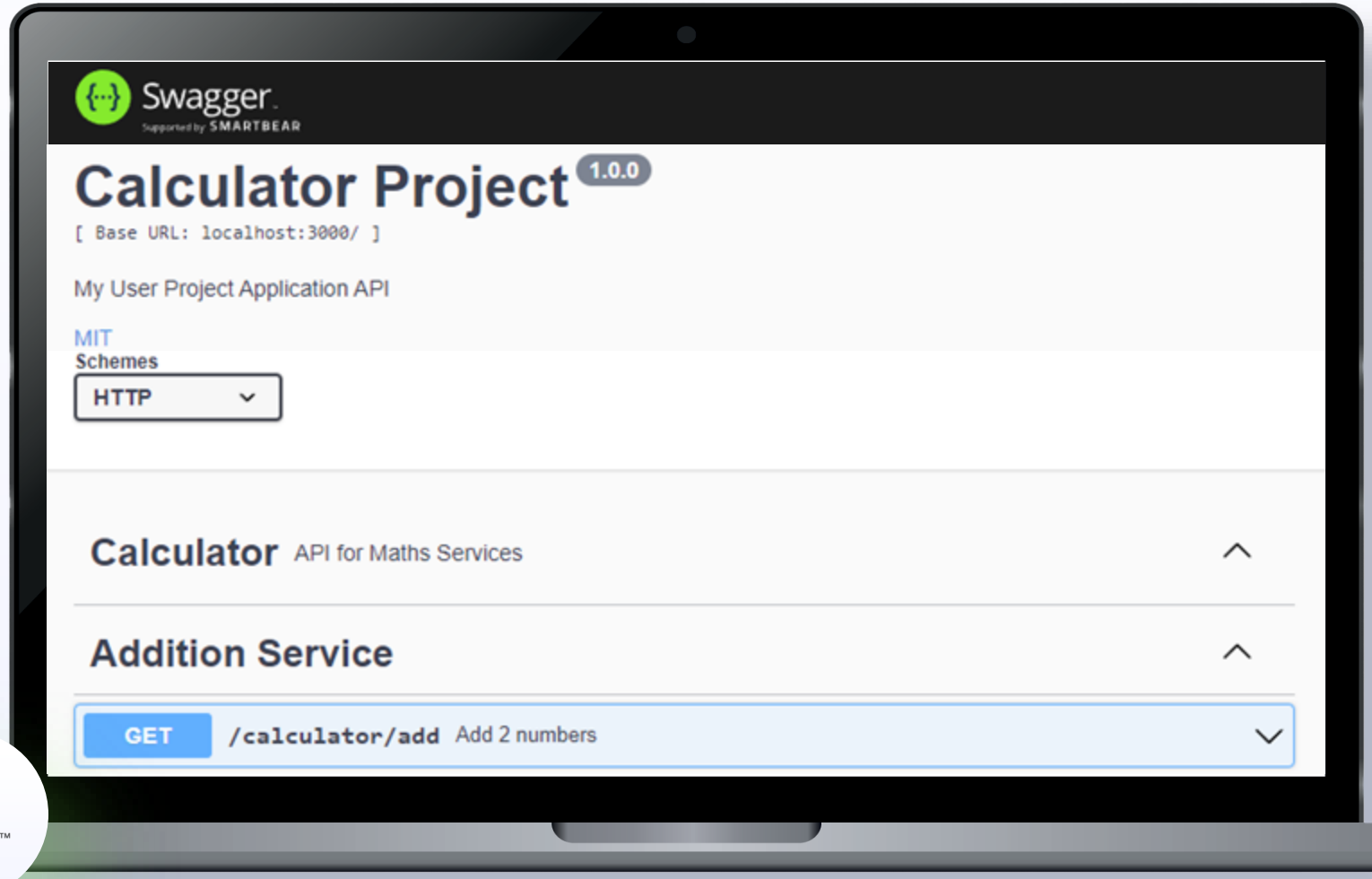
```
"paths": {
        "/calculator/add": {
            "get": {
                "tags": [
                    "Addition Service"
                ],
                "summary": "Add 2 numbers",
                "parameters": [
                    {
                        "name": "num1",
                        "in": "query",
                        "description": "The First Number"
                    },
                    {
                        "name": "num2",
                        "in": "query",
                        "description": "The Second number"
                    }
                ],
                "responses": {
                    "200": {
                        "description": "This service allows
you to add two numbers together"
                    }
                }
            }
        }
    }
```

Swagger Result 1/3

You can now see the “paths” from our previous configuration step in the Swagger UI.

Swagger is a great tool, with a very big ecosystem.

You can even use the API maker to define the documentation, and auto-generate the actual code for the API.



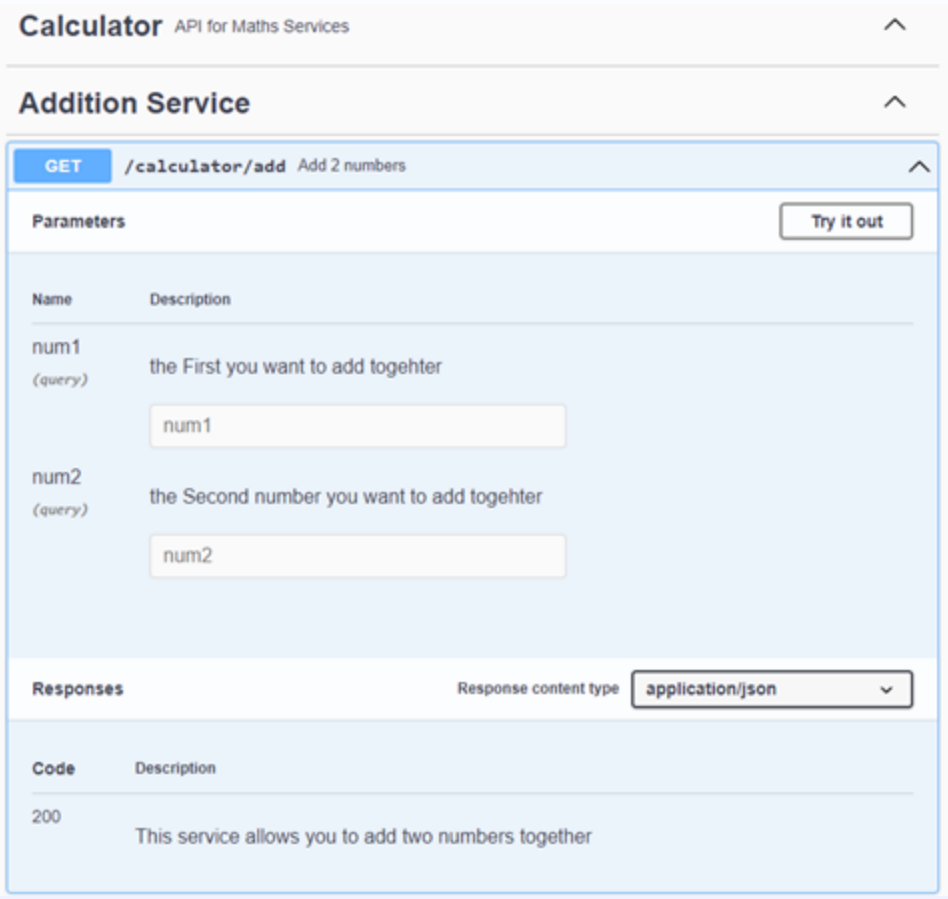
Swagger Result 2/3

If you open the path, you see that we can test the actual API. This makes it incredibly useful in a real-world environment, as anyone can test it easily.

Try out this real-world Swagger example here: Cat Facts API

Write proper documentation and your team members will thank you forever, or you will thank yourself if you go back to your project 5 years from today.

Next Try it Out!

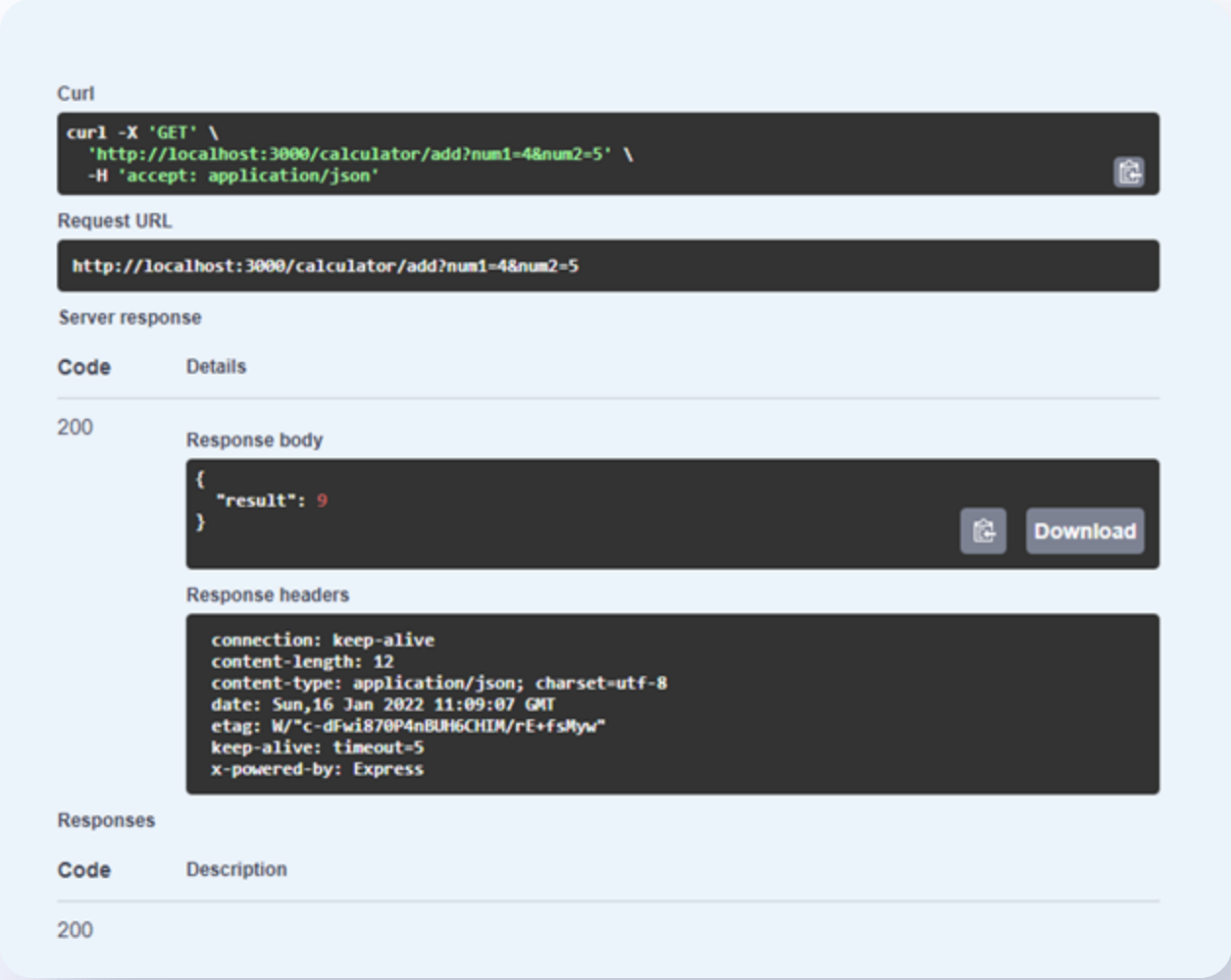


Swagger Result 3/3

Fill in the request parameters and press Execute.

Swagger then shows the request and response details, including the headers.

We can easily see all of the details, including the result.



Exercise 8

- **Part 1** : Write the Swagger specification for your entire project so far!
- **Part 2 : Final Module Project**

  Using what you learnt in this module, recreate the Fake eCommerce Store that you have created in Module 4. Move your front-end files to the public folder, and create an Express back-end to handle all third-party data fetching using axios. Your front end should then fetch from your back-end instead of the fake store API directly.

  Make sure to make a clean MVC Structure and use Swagger to test things out, and make a documentation that is easy to read and test.