Institute of Data

2022

Software Engineering

Module 5

Introduction to Back End Development

Institute of Data

Agenda

Introduction

Backends are sometimes seen as the engine of everything. This is where the magic really happens, where the date gets routed, where all the users 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, backends 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.

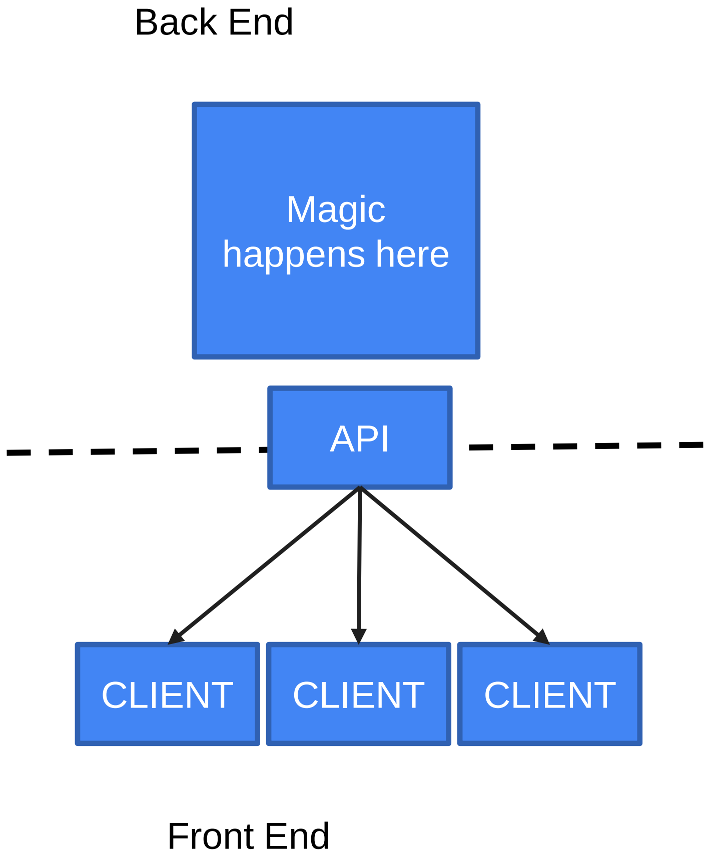
Whether you are working on your side hustle or a client project, you should always have a good plan. In web development, the plan start with the **Design**.

What is a Backend?

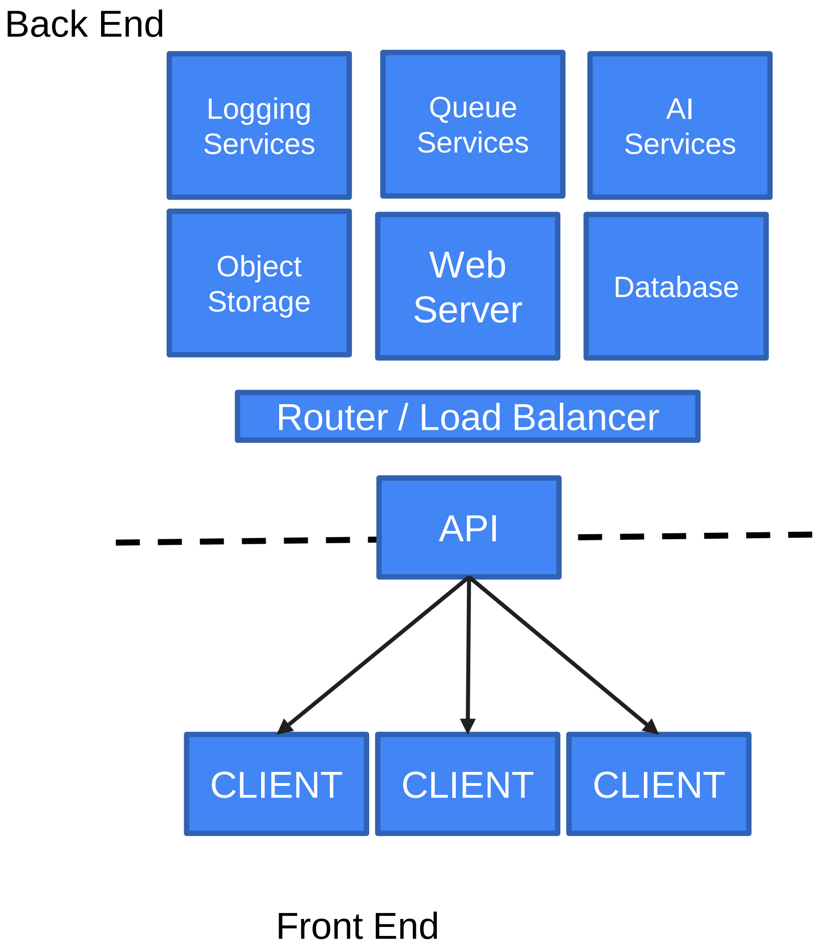
As the name implies, it is something that happens at the back, like a kitchen in the restaurant. All the orders go in, and for those waiting outside, it is just a magic box, or a “black box” if we want to stay in the field.

The backend 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)



If we break down the back end, make it visible, we can see there is a lot happening here.
In this course, we will look at the simple Client-Server applications, but backends can become extremely complicated, especially if third part services are added.
To keep things simple, we will focus on simple CRUD operation, which are data insertions and retrieval, which make up for more than 90% of the web at the moment.



Asynchronous environments

To better understand, 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 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.

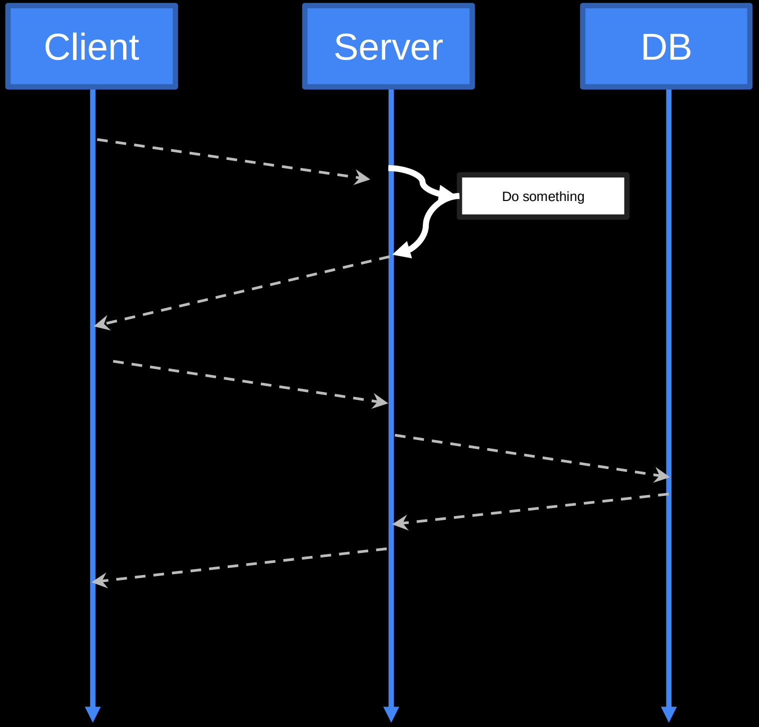
In this case, the answer is 1,3 and after 500ms, 2. This is because the environment does not wait.

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.

Everything about the web is Async, if it wasn’t, it would have been a terrible problem, from a resources point of view, it would have been 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.

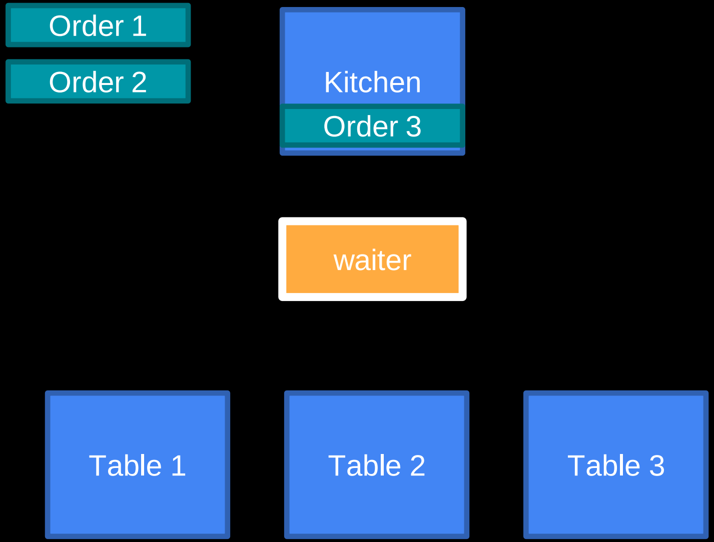


This is a multi-tier design, which extends the classic client – server design. When designing for the web, you always need to understand what exactly happens. You may need to make decisions. For example, what if you were using a system like this for live videos? It would be quite slow.

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 go into the folder (if you are using windows

```
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. 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 –save
```

This will install the express package. The –save keyword tells the system to add it to the 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"
  }
}
```

The package.json 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, 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.

Remove everything from the index.js 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 localhost:3000 , you should get 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}`)
})
```

The following is a breakdown. The first line is requiring the express package, this is the same 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.

```javascript
//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
```

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

Finally, this function activates the web server, until now, it has not been. This reads “Let the webserver listen on the provided port (3000), once the server starts, if successful (hidden test) send out this message to the console”

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

Exercise 1

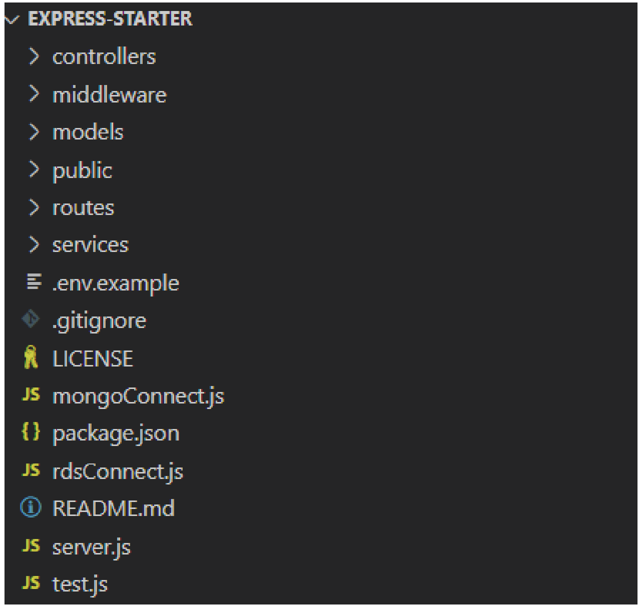
Create a system with multiple web servers running on different ports.

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 will look like. It doesn't matter how easy or complex your application is, your tree should look like. This is what we normally refer to as an MVC "inspired" structure.



```
EXPRESS-STARTER
  > controllers
  > middleware
  > models
  > public
  > routes
  > services
  .env.example
  .gitignore
  LICENSE
  mongoConnect.js
  package.json
  rdsConnect.js
  README.md
  server.js
  test.js
```

The MVC structure reflects what we technical call the “separation of concerns”. An application is made of many independent pieces of software 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.

Let’s look at a real example. Let’s say that a user wants to create a new resource, wants to create a new post on his facebook timeline.

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.

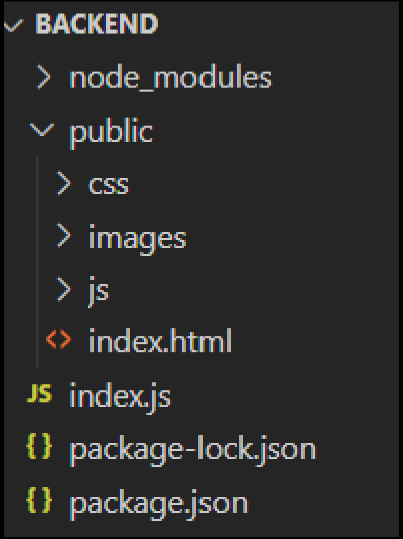
The Controller will have the internal logic to handle the creation of a new post and will call the TimeLinePosts model (which refers to a database table) to update the Database.

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.

Serve Static Content

The most common and simplest thing to do, 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, create a folder css, images and js.
Create a simple html file with a few lines of code just to test things out.

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



```
BACKEND
  > node_modules
  v public
    > css
    > images
    > js
    index.html
  index.js
  package-lock.json
  package.json
```

Should look like this

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

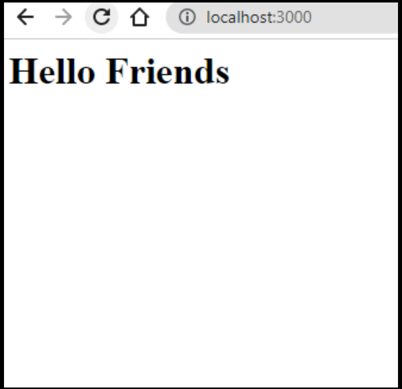
Anywhere in your code, after your app declaration, simply add the following.

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

This will tell the system to fetch pages from the public directory. Remember to change the previous endpoint, it will not work now, simply add a prefix to it.

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

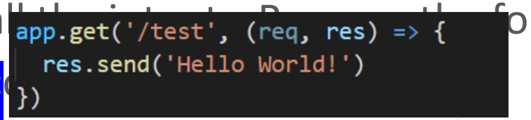
Stop your node server using control – c (or command C) and then restart. Go to your localhost:3000 and you should be able to see your new page being hosted. Remember, index.html is a special page, it does not require to be called specifically. But if you want to create a page with a different name, you will need to call it specifically in the URL, for example, **localhost:3000/users**


localhost:3000

Hello Friends

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

Start by creating a folder called **routes** and create a file called **myTestRoute.js**. We will move the test route in here. Let's create another one as well just to show a[illegible] following c[illegible]



```
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.

```
var express = require('express');
var 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. If we didn't export it, we wouldn't be able to import it.

We now need to import the route. This has to be done for each route we create.

Once is complete, restart your node server again.

At the top of the index.js file, import the route.

```
var testRoute =
require('./routes/myTestRoute');
```

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

Add the following

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

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**

Institute of Data

Section 3 : Design a backend Service

Tips and Tricks

You may have noticed by now, that you need to restart everytime you want to make a change. There are a few tricks to overcome this. You can use a package called **nodemon . **Nodemon restarts your server everytime you save.

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

Then, instead of running node index.js run nodemon index.js (you can change your start script)

A Server Side Calculator

Client-Server architecture is not just a way to make things “more connected” , but also a way to make things securely. All users 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, be able to do that on the client side.
So all we do, 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 kept 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

The 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, it only has one function, to add 2 numbers, so the user will need to pass these as arguments.

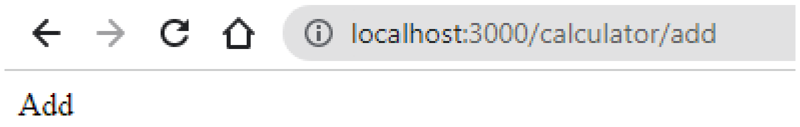
Passing data between client and server

There are two ways of passing data (actually there is more, but we will focus on these two) , through the URL or though the body. Normally URL is less secure, but easier to implement.
To do that, we make a get call to a service. Remember, an endpoint has two objects bound to the function, **req** and **res**. req contains all the parameters passed by the client.

Let's create a new route and call it **calculatorRoute**, bint it to the top level **calculator** URL in the index.js, and create an **add** route.

Routes

You should now have a simple URL that you can use to call the add page.


localhost:3000/calculator/add

Add

```
calculatorRoute.js
router.get('/add', (req, res) => {
  res.send('Add')
})

index.js
var calculatorRoute =
require('./routes/calculatorRoute');

app.use('/calculator',calculatorRoute)
```

Routes (contd)

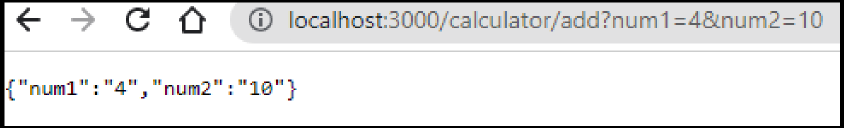
It is time to get some data out of this URL.
Use the following this time.
http://localhost:3000/calculator/add?num1=4&num2=10

This time you are sending data through the URL, **?** Indicates the beginning of query parameters. Then following are the parameters, **num1** and **num2**, these are separated by the **&**

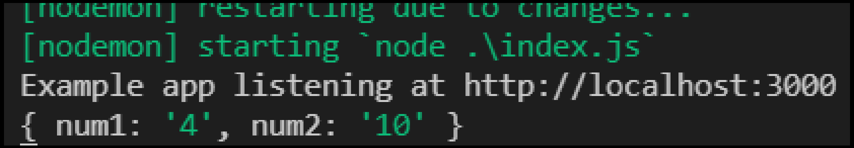
Doing this will not cause any problems, the server is not going to handle the data, simple. To see the data we need to make some changes to the route.
Just add the following line to the route runction.

```
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.



```
{"num1":"4","num2":"10"}
```



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

Pass data

This is how we access GET data. Now we can use that data to achieve our goal.
Let’s change the route so that we can handle this properly.

Complete this, and try the new outcomes.

```
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 used this to send data in json format instead of just strings.

Fetch Data from client

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

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

```
<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>
```

You can now change your html page to test that you can make the request to the server.

Exercise 2

Using the code from the previous slide as your starting point, create a full calculator that is able to do the 4 different operations.

Exercise 3

Create other routes to manage different operations.

Increasing the complexity

As you can see, it 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. This will be done in future modules.

Controllers

Our separation of concern is not complete. We mentioned before that routes should only do “routing” and have no logic involved. They take data and pass it 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 left, we have moved all the logic out of the route, and into the controller.

Why is it callback hell? Because the data is passed from one to another until the end. This is still quite small.

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
}
```

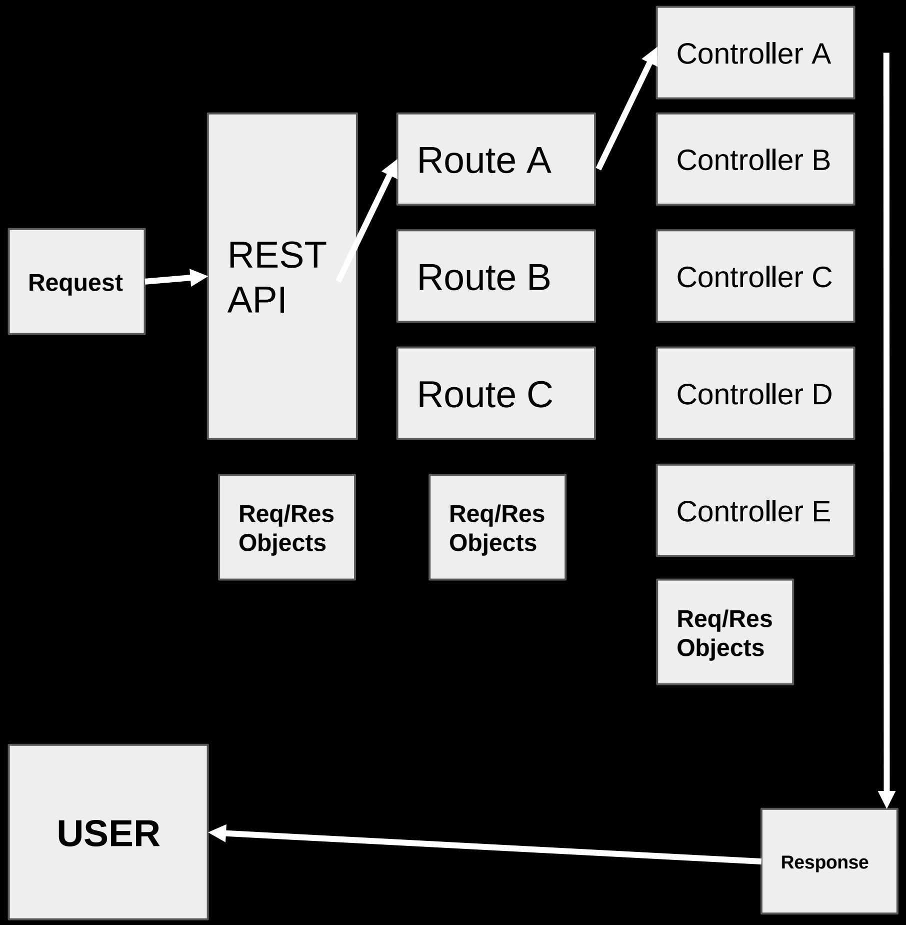
calculatorRoute.js

```
var express = require('express');
var calculatorController =
require('../controllers/calculatorController')
var 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 it is sent back to the user.

As you can see from the diagram , data is always passed down as a Req/Res Object. Finally, the controller sends a response through the Response object when operations are complete.



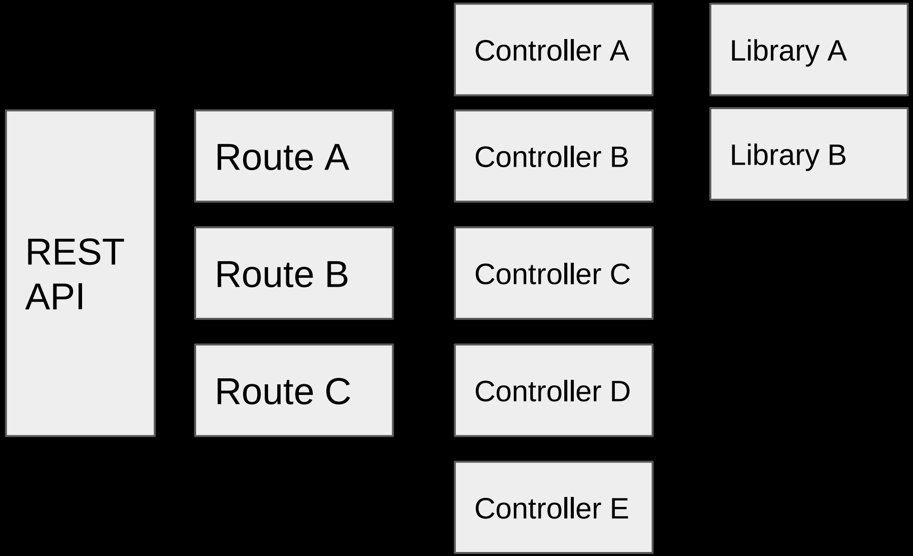
Exercise 4

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

Section 4 : Object-Oriented Development

Libraries

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

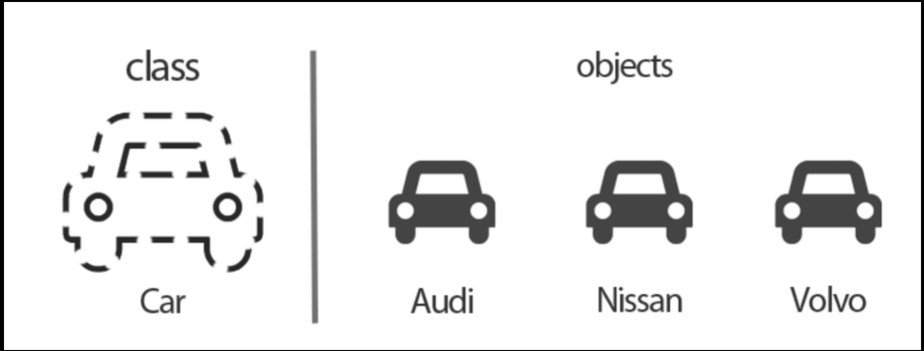
This time we step up the game, and start developing “real” applications, not just “web structures”. In this part we go back 30 years and we build a small library that can be utilized by anyone. It is independent and has no knowledge of the web!

Let’s review what a library is.

A library is simply a program.

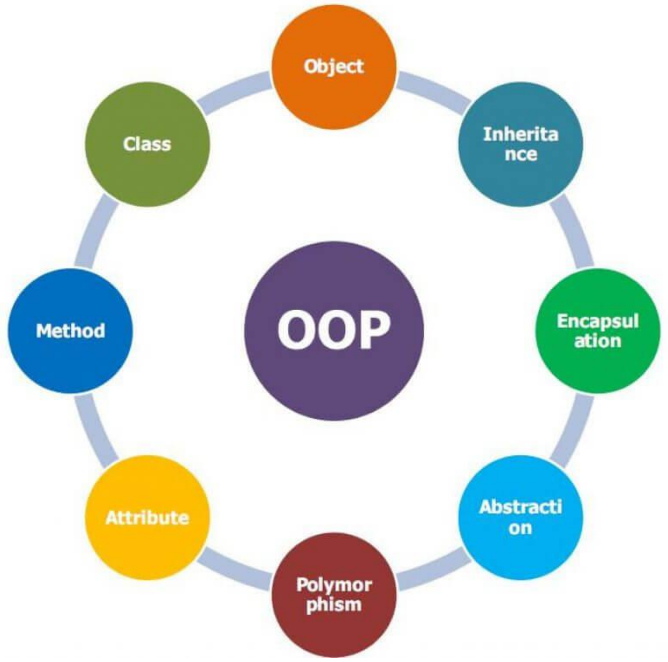
We use “object-oriented” development style to build libraries. This means that we use classes, also known as blueprints, to create objects.

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).



Notes for the student

The concept of class is 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!
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 simplest thing the class has are “a name, methods and properties” - and just to make things more fun, “everything in Javascript is an object, so methods and properties are all objects and seen in the same way”

A library is simply a program.

Generic Class

ClassName
List Of properties
List of Methods

Car Class

Car
Wheels Color Engine Range
Accelerate Break

Object of Type Car

Car : MX5
Wheels : 4 Color : Black Engine : 2000 Range :500
Accelerate Break

Object of Type Car

Audi A4
Wheels : 4 Color : Red Engine : 1500 Range : 600
Accelerate Break

Visibility

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

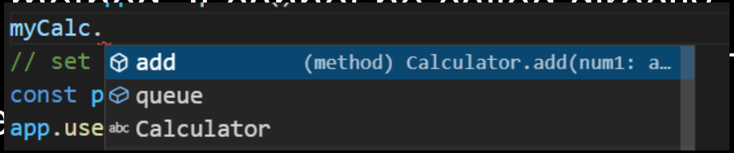
A **public** method can call a **private** method!

```
class Calculator {
    constructor(contents = []) {
      this.queue = [...contents];

    }
    #log=()=>{
        console.log('test')
    }
    add(num1, num2) {
        this.#log()
        const value=num1+num2
        return value;
    }
  }
```

Take this as an example, the # indicates a private

but the n

in the e



```
myCalc.
// set   add        (method) Calculator.add(num1: a…
const p  queue
app.use  Calculator
```

Create a class

Javascript is not hard typed, meaning it has no forced rules, because rules came after JS, therefore we have “conventions”

A class has a constructor, which is a method called when we want to create an instance (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 standard, 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
    }
  }
```

Structure

As an example, this is the Calculator Class. We have a constructor that takes 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, when the operation is complete, it will call the #log (private) to console the result.
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.

```
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 is used to get the epoc time, we can use this as
an ID, simply but effective, as long as things are generated
at least one second apart.
```

Instantiating

Our blueprint is ready, we can now make as many 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)
```

Integration

We now have the class, we have the controller, 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.

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, 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, 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 5

Part 1 : Expand your application to use a 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.

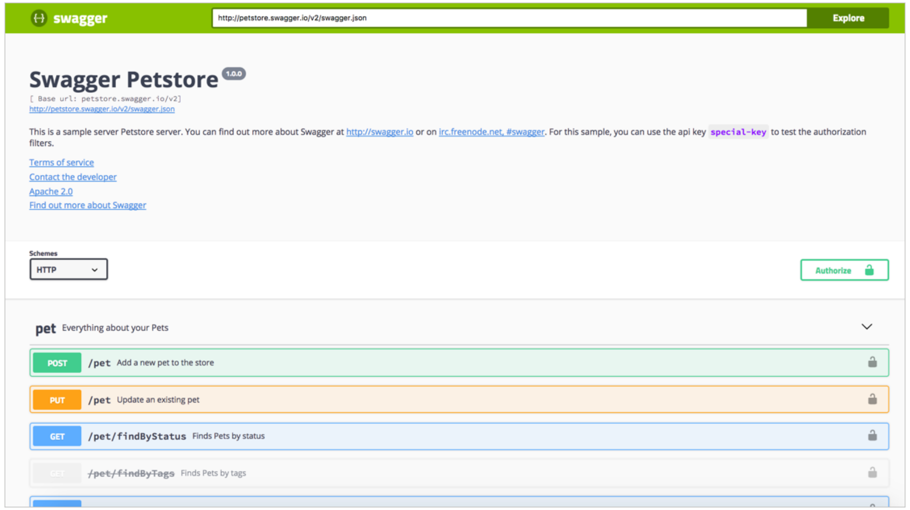
Part 3 : Create a generic library for logging - pass a message to be logged, this will contain at least The ID of the caller, and the result. Log to the console every call made.

Section 5 : Swagger

Time to SWAG

In order to step up, 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.



Swagger Petstore 1.0.0

[Base url: petstore.swagger.io/v2]
http://petstore.swagger.io/v2/swagger.json

This is a sample server Petstore server. You can find out more about Swagger at http://swagger.io or on irc.freenode.net, #swagger. For this sample, you can use the api key `special-key` to test the authorization filters.

Terms of service
Contact the developer
Apache 2.0
Find out more about Swagger

Schemes: HTTP

Authorize

pet Everything about your Pets

- POST `/pet` Add a new pet to the store
- PUT `/pet` Update an existing pet
- GET `/pet/findByStatus` Finds Pets by status
- GET ~~`/pet/findByTags`~~ Finds Pets by tags

Apply Swagger

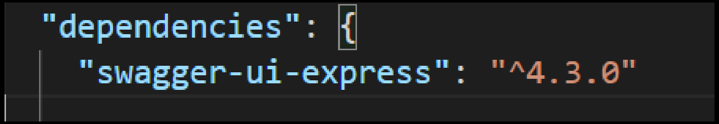
Swagger is very simple, but it can grow exponentially in complexity. We are going to make a very simple integration.

The first thing to do is get swagger as a dependency.

Use the following to add swagger as your dependency.

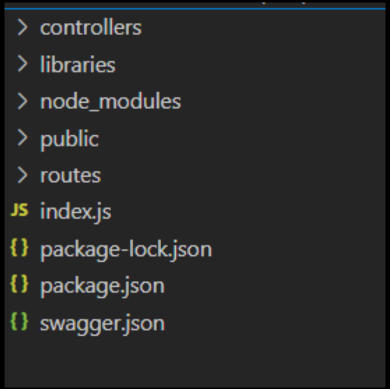
```
npm i swagger-ui-express -S
```

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.



```
> controllers
> libraries
> node_modules
> public
> routes
JS index.js
{} package-lock.json
{} package.json
{} swagger.json
```

Swagger Settings

Copy and paste the following into the swagger.json

This is your Swagger definition, it describes all the services and the documentation.

You can customise it as you like.

```
 {
    "swagger": "2.0",
    "info": {
      "version": "1.0.0",
      "title": "My User Project CRUD",
      "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"]
   }
```

Modify your server

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 swagger document and create a rout specifically for it.

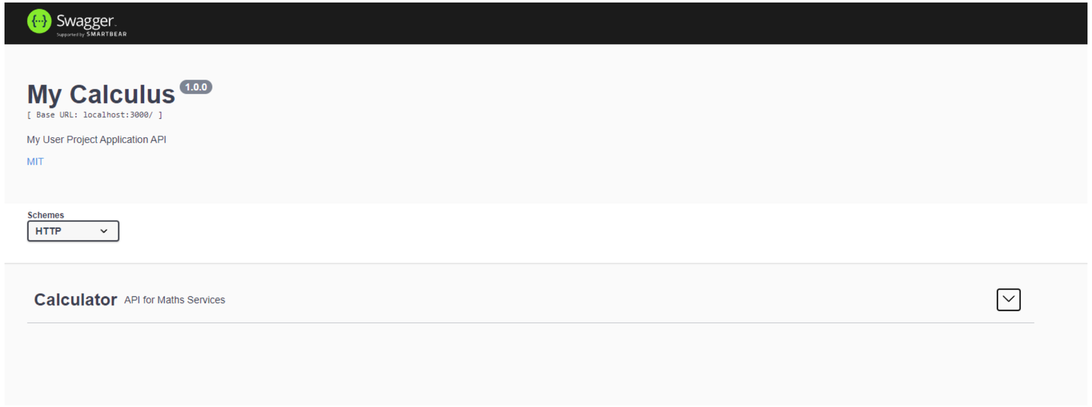
```
const swaggerUi = require('swagger-ui-express');
swaggerDocument = require('./swagger.json');
app.use(
  '/api-docs',
  swaggerUi.serve,
  swaggerUi.setup(swaggerDocument)
);
```

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

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 this part yet.

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



My Calculus 1.0.0

[Base URL: localhost:3000/]

My User Project Application API

MIT

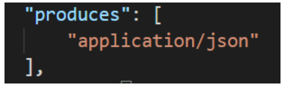
Schemes: HTTP

Calculator API for Maths Services

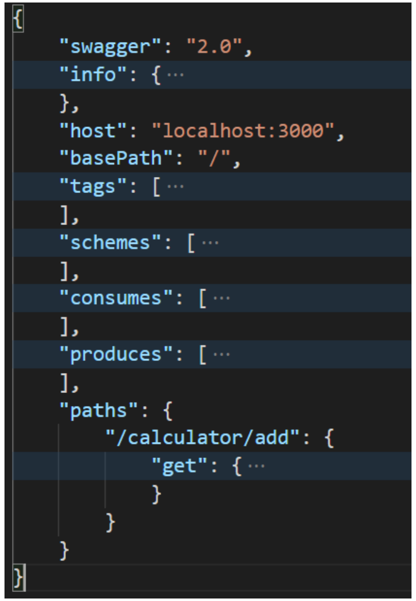
Define Paths

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

It should look like this, i have closed all of the code snippets.



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



```
{
    "swagger": "2.0",
    "info": {…
    },
    "host": "localhost:3000",
    "basePath": "/",
    "tags": […
    ],
    "schemes": […
    ],
    "consumes": […
    ],
    "produces": […
    ],
    "paths": {
        "/calculator/add": {
            "get": {…
            }
        }
    }
}
```

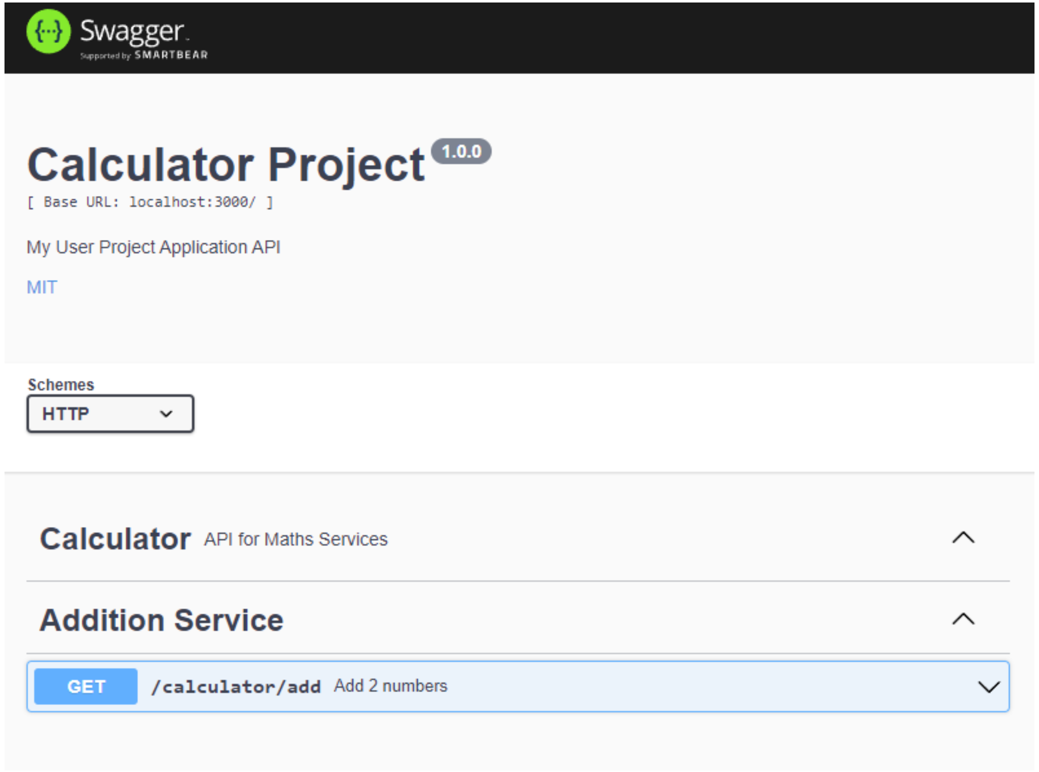
This is the code that defines the paths.

```
"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"
                    }
                }
            }
        }
    }
```

Result 1 / 3

You can now see the “paths”, you should have all your paths here.

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


Calculator Project 1.0.0

[Base URL: localhost:3000/]

My User Project Application API

MIT

Schemes: HTTP

Calculator API for Maths Services

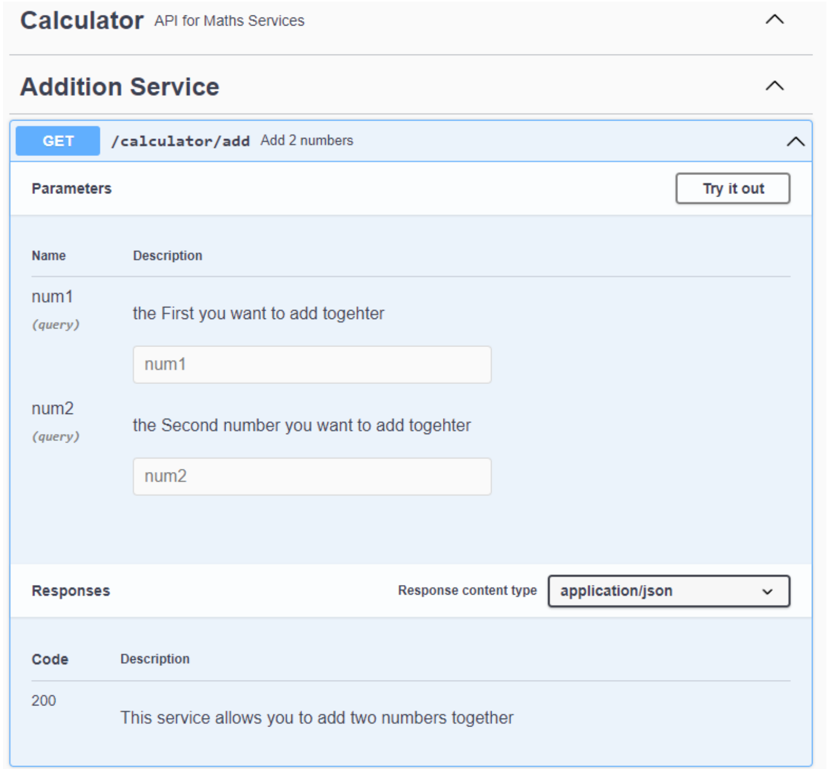
Addition Service

GET /calculator/add Add 2 numbers

Result 2 / 3

If you open the path, you can see test the actual API, this makes it incredibly useful in a real-world environment, This way, anyone can test it easily. Write proper documentation and your team members will thank you forever, or if you go back to your project 5 years from today.

Next **Try it Out**!



/calculator/add Add 2 numbers

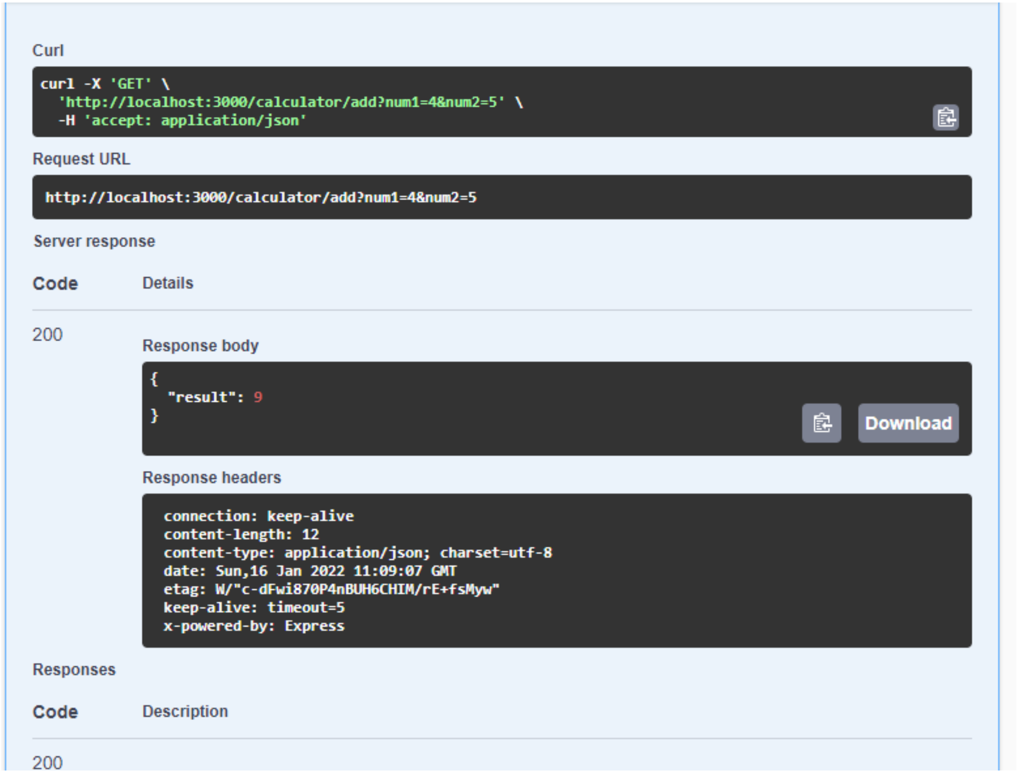
Name	Description
num1 *(query)*	the First you want to add togehter num1
num2 *(query)*	the Second number you want to add togehter num2

Response content type: application/json

Code	Description
200	This service allows you to add two numbers together

Result 3 / 3

Fill in the parameters and press execute, here you can see everything, from the call to the actual result.



Curl

```
curl -X 'GET' \
  'http://localhost:3000/calculator/add?num1=4&num2=5' \
  -H 'accept: application/json'
```

Request URL

```
http://localhost:3000/calculator/add?num1=4&num2=5
```

Server response

Code Details

200

Response body

```
{
  "result": 9
}
```

Response headers

```
 connection: keep-alive
 content-length: 12
 content-type: application/json; charset=utf-8
 date: Sun,16 Jan 2022 11:09:07 GMT
 etag: W/"c-dFwi870P4nBUH6CHIM/rE+fsMyw"
 keep-alive: timeout=5
 x-powered-by: Express
```

Responses

Code Description

200

Exercise 6

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 Social media services that you have created in module 4, this time, on the server. We will use them again later on.
Make sure to make a clean MVC Structure and use the Swagger to test things out, and make a documentation that is easy to read and test.

End of Presentation