Week 2: REST API

1: REST API

What is REST API?

- **REST** stands for **RE**presentational **S**tate **T**ransfer
- **REST** is an architectural style, or design pattern, for APIs (API: Application Programming Interface)

Important Definitions

Server: a computer or computer program which manages access to a centralized resource or service in a network.

Client: a person or software who uses the API.

Resource: can be any object the API can provide information about.

HTTP (Hypertext Transfer Protocol): Protocol for communication between servers

---

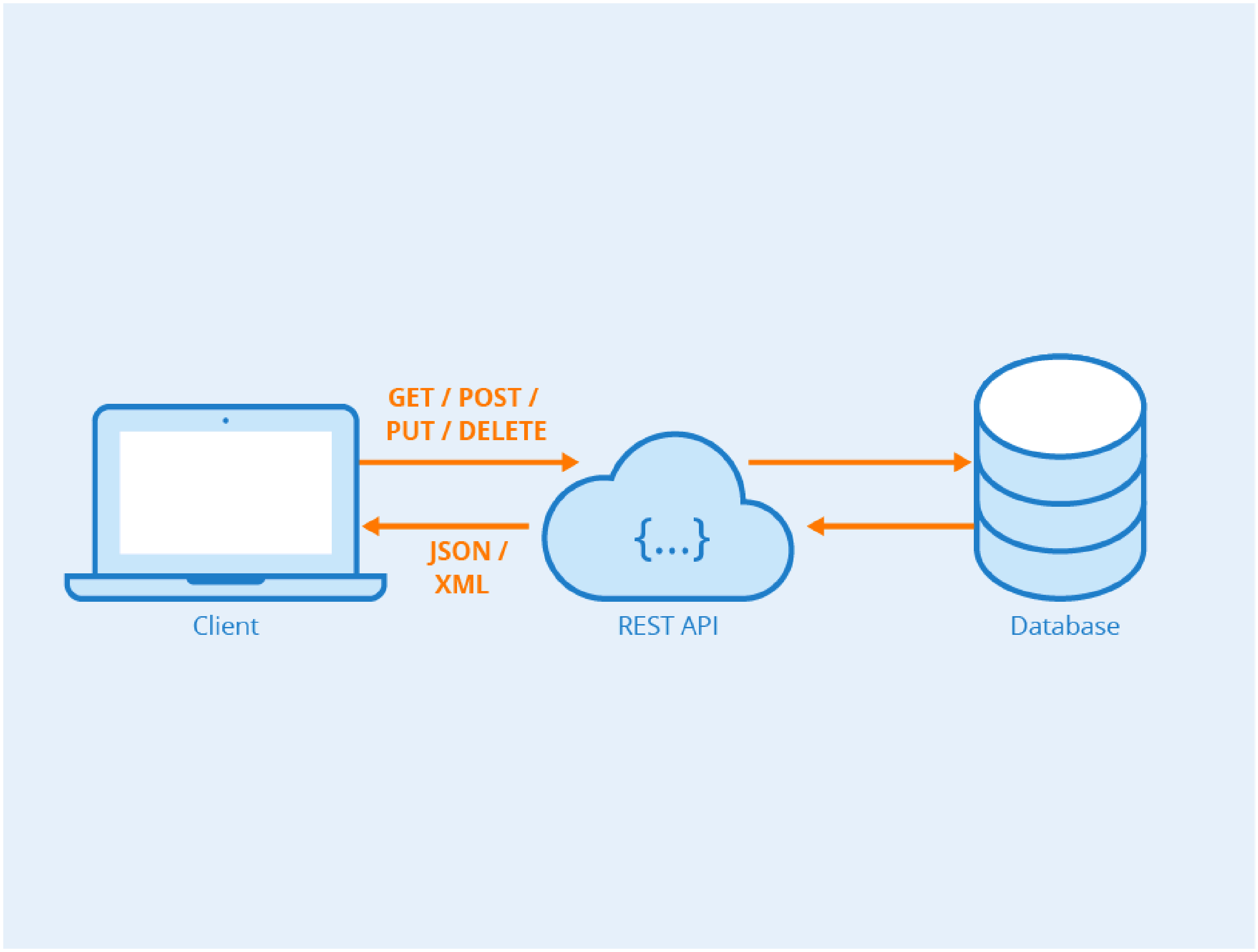
Key Idea:
When a **REST API** is called, the server will transfer to the client a representation of the state of the requested resource.

Types of HTTP methods

GET - Retrieves resource from the server (should only retrieve resource and should have no other effect).
POST - This HTTP request type is usually used for creating a resource. Once the request is successfully created, an id of the newly created resource is returned as part of the response to this HTTP request.
PUT - Similar to POST, but used to update an existing resource. You pass the id of existing resource along with the request.
PATCH - Applies partial modifications to a resource.

DELETE - Removes the resource from the server. You need to pass the id of the resource to be deleted.

Visualization:



2: JavaScript Objects

Motivation: Strings, Numbers, Arrays, Functions are not enough to represent more conceptual concepts.

For example, let's say you want to represent a car. You can try to do the following:

```javascript
// car properties
let numberOfWheels = 4
let speed = 0
let beep = function() { console.log('beep') }
let carMark = 'Toyota'
```

This is not as simple to keep track of, and can get very messy. So we can try using JavaScript Objects, since Car is conceptually an object.

We can do it like so:

```
const car = {
    numberOfWheels: 4,
    speed: 0,
    beep: function() { console.log('beep') },
    carMark: 'Toyota'
}
```

and then, to access the properties, we can do it like this:

```
car.numberOfWheels
car['numberOfWheels]

car.speed
car['speed']

car.beep
car['beep']

// you can call the function like this:
car.beep()
car['beep']()

car.carMark
car['carMark']
```

JSON (JavaScript Object Notation)
JSON is a syntax for storing and exchanging data, and it is a Standard across many web technologies.

Motivation: We can convert any JSON received from the server into JavaScript objects, natively (without much effort, since it's already part of the JavaScript language).

This way we can work with the data as JavaScript Objects, and not worry about various data parsing, and formatting

Note:

- For JSON, a String must use these quotations: (" "), as the other ones, are not part of the Standard, and simply won't be understood.
- Functions are not classified as data, so trying to pass a function would not work. (You can theoretically pass a function as a String)

3: Initializing our project

npm (node package manager)

What is npm? it's an online repository with its own command tool interface.

1. Let's open a new folder in Visual Code, I am calling it `/webdev`
2. Then we open the **Terminal** (Ctrl ~ if you are on Windows/Linux and Control ~ if you are on Mac), and type in `npm init -y` and ENTER

This will initialize a Node.js project in `/webdev` folder, with the default values that you can later modify in `package.json` that is now created.

4: Creating our first REST API route

Now we will create our very first REST API Route, using `express.js`

But, firstly, what is **express** and why is it important?

express

- it is a framework that is built on top of node.js for primarily back-end development of web applications.
- it allows us to create our own web routes that we can use in our application

1. Create a new file called: `index.js`
2. in the **Terminal** write `npm install --save express` and ENTER

This installs express.js for our project, `--save` is used for saving it as a dependency of our project. You can check high level dependencies in `package.json`

3. There we need to import `express` to be able to use it; we can import it like so: `const express = require('express')`

4. Now, we use express to actually create a server
   `const app = express()`
5. Having a server is definitely nice, but it doesn't know what to do. So we will assign it a port that it will own on our computer, and we will be able to communicate with our server.

To do that, we add this to `index.js`

```
app.listen(3000, function() {
    console.log('server running on port: 3000')
})
```

The function is there as a callback function: which means that the function will get executed after the server assigns itself to a port.

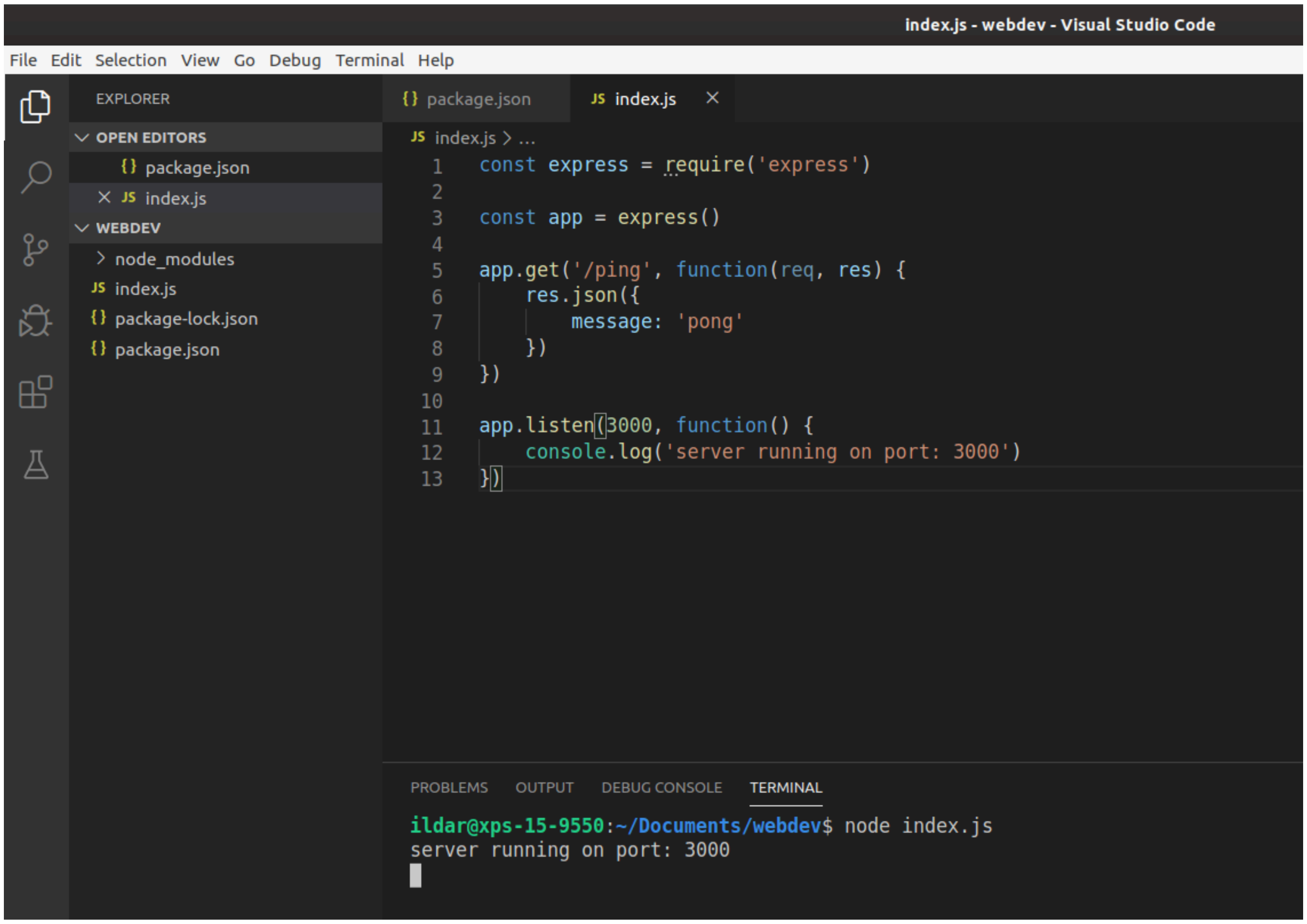
6. Now we can add the following code, which will be our very first REST Route.

```
app.get('/ping', function(req, res) {
    res.json({
        message: 'pong'
    })
})
```

what this code means:

- early we defined a variable named `app` which is the server object
- `app.get` is a call to a function that defined a REST route for us: `GET localhost:3000/ping` where `/ping` is the URL and GET is the HTTP method that we talked about earlier
- the second part of the function call `app.get` is a callback function which is called by express when we make a request to `localhost:3000/ping`
- the parameters of the function must be: **req**, which stands for request (that is the information that is sent to an express server by the client) and **res**, which stands for response (that is the information that express sends back to the client)
- `res.json` sends our JavaScript Object information with the message: 'pong' (after parsing it to JSON)

Now we can run `node index.js` and we get the following:


index.js - webdev - Visual Studio Code
File Edit Selection View Go Debug Terminal Help
EXPLORER
OPEN EDITORS
package.json
index.js
WEBDEV
node_modules
index.js
package-lock.json
package.json
index.js > ...
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
const express = require('express')

const app = express()

app.get('/ping', function(req, res) {
    res.json({
        message: 'pong'
    })
})

app.listen(3000, function() {
    console.log('server running on port: 3000')
})
```

```
ildar@xps-15-9550:~/Documents/webdev$ node index.js
server running on port: 3000
```

Looking at the **Terminal** we see that the `server running on port: 3000`, so we can now test our route.

5: Testing our Route

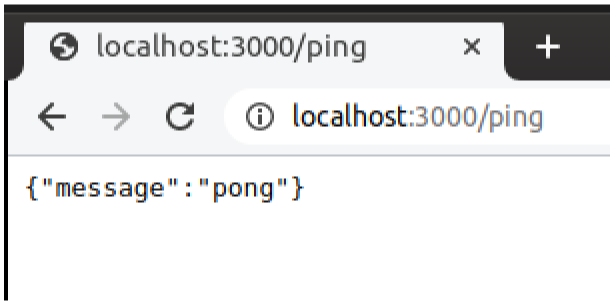
So, the cool part is that going to a typical website, e.g. youtube, google, facebook, etc. on your browser is actually a GET route. For example: GET facebook.com

Chrome/Firefox/Safari

So similarly, we can go to Chrome and type in:

`localhost:3000/ping` and press ENTER.

and we get our response!



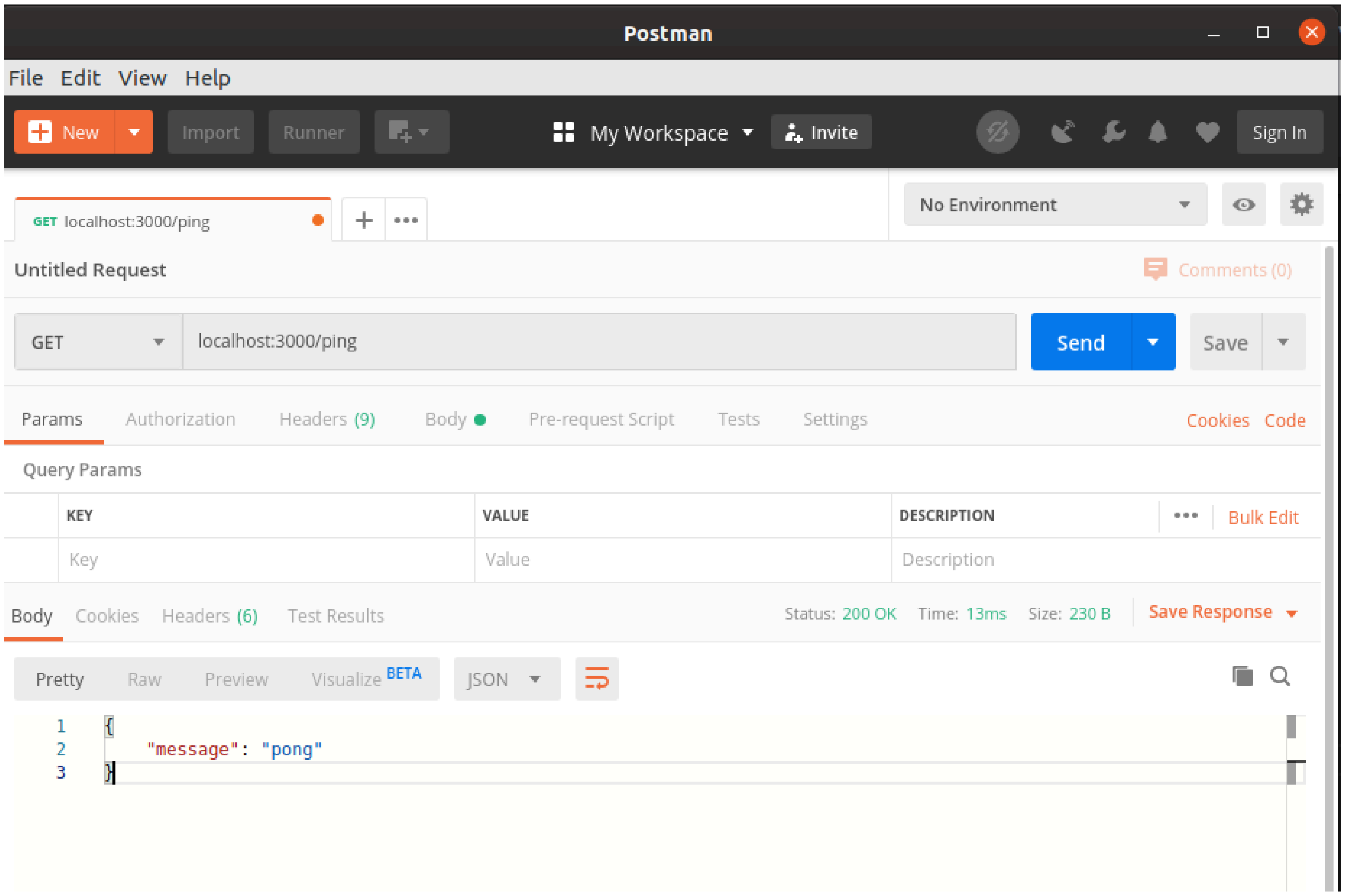
```
{"message":"pong"}
```

Using Postman

a different way to test our routes now, and in the future, is to use Postman.

You can similarly type in `localhost:3000/ping` in the url part of Postman (make sure you select GET in the HTTP methods part) and press SEND.

Then we get our response here as well!



```
{
    "message": "pong"
}
```

Note that it says Status: 200OK
This means that everything went okay. This is also where you will potentially see such errors as 404, 503, or a 403.

Also, note Body contains the response that we sent from the express server.

We can ignore the top part for now

6: Implementing GET Item routes

As we can see we can now communicate with our server using Postman.

- So, let's try to keep track of items as we want to for our project!

let's create an array to keep track of the items, we can add it right after creation of our server, putting it on line #4

`const items = []`

and then we make `GET /items` route to be able to retrieve currently created items, like so

```
app.get('/items', function(req, res) {
    res.json(items)
})
```

We can stop our server by pressing CTRL C in **Terminal**, and restarting the server with `node index.js`.

And we can test our route like we did with `/ping`
However, this will be a pretty boring route, for now...

7: Implementing POST Item route

Now, let's add some values to our `items` array.

Recall that we can use the `push` function to add an element. So let's define what the item object would look like first:

```
const item = {
    title: 'Some Movie Title',
    completed: true, // or false?
    type: 'Movie',
    createdAt: Date.now(), // gets the time of this instant
    id: 1 // some unique way to identify an object
}
```

and then we can just use `items.push(item)` and then the list will have our item added to it. But, let's give the ability to specify fields such as: 'title', 'completed', and 'type' to the user (we will handle 'id' and 'createdAt' ourselves)

To do that we need to be able to understand, and use the data that the user sends us.

By design, a **GET** request cannot accept packaged data, usually referred to as a *payload*.

That is in fact, the main difference between a **POST** request and a **GET**. This should make sense, since if you recall POST is meant to create a new resource, and the way it will do that is by using user's input.

---

we will force the client to send us **JSON** and just not understand non-JSON data.

To do that, at the top of the file (after definition of `app`), please add:

```
app.use(express.json())
```

This is called middleware, which is named like that because it is a function that gets executed after express receives a request from the client, and then right after that you get a parsed request body (`req.body`) to use in your route.

Note: `req.body` will have the data that we will send to the server from Postman.

So, we can define the route as follows:

```
let currId = 1

app.post('/items', function(req, res) {
    const item = {
        title: req.body.title,
        type: req.body.type,
        completed: false, //since you why would you add something to watch
                          // that you already watched
        createdAt: new Date(),
        id: currId
    }
    items.push(item)
    res.json(item)
    currId = currId + 1
})
```

We can stop our server by pressing CTRL C in **Terminal**, and restarting the server with `node index.js`.

Now we can **POST** new items, and **GET** our existing items.

8: Implementing GET of certain item

So, a certain thing we would like to overcome is the ability to get a certain item, rather than the entire list of items. To do this, we will make a route that queries for an id of an item, and returns a valid item if it finds it.

But, how do we pass the id to the server, since there is no body in a GET request?

We can pass the id in the URL, e.g. `items/314` where 314 is the id of the item, then we use that *parameter* (`req.params` is where you can access all the parameters passed in the request) to query the items.

Note: in express we need to specify that 314 is a parameter, otherwise it will just think it's a normal url without any parameters. To do that we make specify id as: `:id`, to get the full URL as: `items/:id`

We then can implement our function like so:

```
app.get('/items/:id', function(req, res) {
    const id = req.params.id

    for (let i = 0; i < items.length; i++) {
        const item = items[i]
        if (item.id == id) {
            res.json(items[i])
            return
        }
    }
    res.status(404).json({})
})
```

What the function does:
It looks at every item that we currently have in `items`, and tries to find one with the id we have provided as a parameter. If we do find one, we want to send the function, and terminate our search and the function since the id is unique, so you won't find a copy.

Note: we can specify that status of our response as 404, if we don't find anything.

9: Implementing DELETE Item route

Now we would like to delete an item that is in the list, but we don't want it to be there for some reason.

Aside
To remove an element in the array in JavaScript:

```
array.splice(indexOfElement, 1)
```

So, similarly, to GET item route, we would like to specify the id of the item we would like to DELETE.

[this is simply because we are following best practices, as it doesn't make too much sense to send a body for a DELETE request, although you can send a body request, as opposed to parameter, nobody really stops you in Express. But you might be judged, or unable to do so in other frameworks]

Here is the code that does what we would like to accomplish:

```
app.delete('/items/:id', function(req, res) {
    const id = req.params.id

    for (let i = 0; i < items.length; i++) {
        const item = items[i]
        if (item.id == id) {
            res.json(items.splice(i, 1))
            return
        }
    }
    res.status(404).json({})
})
```

10: Implementing PATCH Item route

Patching an item requires you to modify an existing item. For this, you will be implementing your own route.

Hint, this is very similar to a POST request, except you are not creating a new item, but modifying an existing one. So, try using the things we learned while implementing the other routes.

Note:

Your patch route should be able to:

- find an item by id
- be able to modify the following properties: 'type', 'completed', and 'name' of the existing item
- should return an updated item, if you find one with the right id.
  if you can't, return an empty object with status 404.

We will go over the solution next week.