SIT737 Prac 2

# Creating a web service

Last week we went through the basics of creating web service, we really just created something extremely small to make sure everyone was at the same level.

This week we will instead, start exploring the actual technologies ,

Learn how to “satisfy a need” by creating web-server to serve static webpages

Basic HTTP requests, GET, POST, PUT, DELETE

How to use Postman

Make an http call from your browser

Mastering this should lead to having enough knowledge to build on bigger applications

Static Web Server

Last week we created a web server in order to test some extremely basic browser functionalities, this week we will work on creating a proper, modern web server to serve web pages.

To do so, we need two files to begin with, our packaga.json file, in order to keep track of our application description, and our web-server.js

Here is my package.json

{

"name": "web-server",

"version": "0.1.0",

"description": "Web server for SIT 737",

"dependencies": {

"express": "\*",

"log": "1.4.0",

"body-parser": "\*"

},

"author": "Alessio Bonti",

"license": "MIT"

}

You can notice I now have three packages available, express, log and body-parser. We are adding complexity to our service right now.

Here is the code for the application instead, create a new file.

var express = require("express")

var bodyParser=require('body-parser')

app = express();

// we set the port programmatically, in case we need to change it later

var port = 3000;

//this is where we are going to getch our html from

var root = '/public'

//tell express to use the static middleware,

app.use(express.static(\_\_dirname + root));

//start the app and listen to the port

app.listen(port);

console.log("Listening on port ", port);

Try to understand the code, for example, google what express.static is.

You can see that we are looking for our static content inside the public folder, therefore , create a folder called public, this is where we are going to be serving all of our content from.

Refer to week 1 if you are not sure about the steps so far.

Create a simple html file to serve and call it index.html , as per default, this file is served if no specific file is requested.

Test your application.

We have now created some very basic server. We are now going to create a set of endpoints that we can use to test our advanced capabilities.

Add the following code, we have now constructed to more endpoints, they are both sharing the same url, but referencing different request mo

/\*\* bodyParser.urlencoded(options)

\* Parses the text as URL encoded data (which is how browsers tend to send form data from regular forms set to POST)

\* and exposes the resulting object (containing the keys and values) on req.body

\*/

app.use(bodyParser.urlencoded({

extended: true

}));

app.use(bodyParser.json());

app.use(express.static(\_\_dirname + '/public'));

app.get("/test",function(request,response){

var param=request.query.username

console.log('Get requested by '+param)

response.send('Thank you for requesting our Get Service')

})

app.post('/test',function(request,response){

console.log(request.body)

var data=request.body;

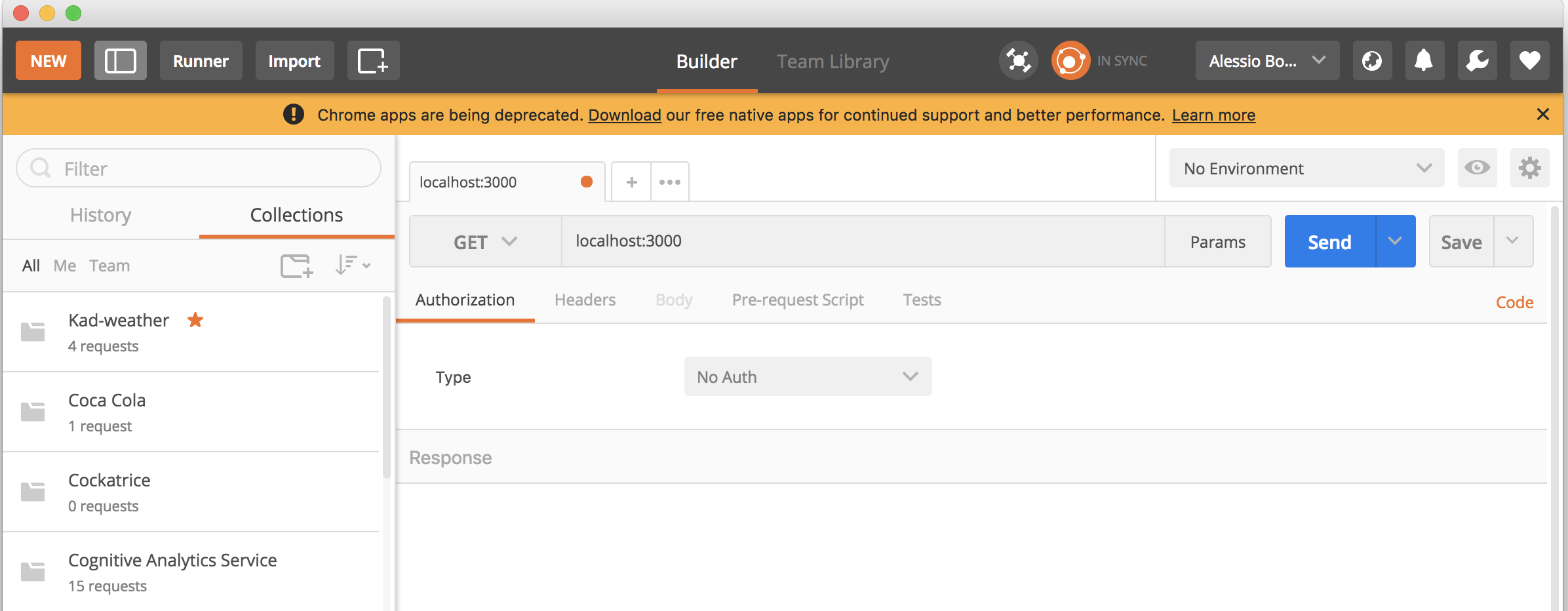
console.log('Post requested, here is the data :'+data)

response.send('Thank you for requesting our Post Service')

})

As we mentioned in week 1, the browser normally operates on a get when making a url request, therefore we will not be able to test the post.  **Make sure app.listen is at the bottom!**

It is time to move forward and learn another vastly used industry tool, Postman. Google it and it should come up straight away.



In the box next to get, you can insert your URL, in our case, localhost:3000 , and you can also choose the action type (GET or POST). If you press send you can now see the results in the bottom section.

**GET**

Postman is made so that we can more easily test our web services, so test our new endpoints, /test . This endpoint, if you read the code, requires a **parameter**. GET parameters are passed together with the request, in Postman, you can add the parameters by clicking on Params. **Parameters** in GET requests are passed as data in the url, in a key value structure, add one called **username**, and see how the url changes.

Because of this, it is generally not a good way to send important parameters through GET, or also from a manageability point of view, they are harder to trace, it is generally indicated for something small such as passing a parameter for database query.

**POST**

Post is generally the preferred choice to send big chuncks of data, such as complex objects, or heavy objects such as pictures, or complete files.

The data is obtained through parsing the body part of the request, and because it is encoded, we need to parse it, we do so using a body-parser package (reason why we included this).

We have set the web-server to automatically do this, therefore it can decode both urlencoded or json.

Create a new Tab in Postman, and this time change the Action to POST. To add some data, we need to add a body to it, therefore click on **Body** choose **raw** and then type **JSON (Application/json)** from where it initially says text, this is the type. Try to add some well formatted json object in the area below.

Such as this

{

    "name":"alessio",

    "lastname":"bonti"

}

Send it and have a look at the console of the server.

You should see the content of your function.

You now know the basic for constructing a well formed web service, complete the next two exercises, working together with the person next to you.

**Logging into a file**

Use the log functionality, this is not the console.log, this is a proper package we have downloaded. This is a much higher level of logging, which is more similar to what is used in the industry.

This time, you will need to explore how the package works by reading its npm documentation. You will be logging the server startup and the calls made to it into a file, which you will then show your tutor.

**Create A simple calculator**

Create an endpoint with a new name, using a GET action, to pass three parameters, two numbers and an operator. Internally, you can use a switch to understand logically the operation being called, and execute that operation.

Return the result to the user.

Once again, work together!