Intro to web development

# Key points to cover

* JS on server side

Day 1

Day 2

Day 3

Day 4

Day 5

* NodeJS
* Install and setup env
* Familiarize node and npm, package.json
* Web servers. Node web server
* Serving a website, intro KOA
* Intro REST API, Build an api in KOA

# NodeJS

Node.js is a server side JavaScript built on Google’s V8 JavaScript engine. V8 is the software component that runs javascript in Google chrome.

It is an open source framework. NodeJS is a framework and not a language. Javascript is the language, V8 is the engine that executes the JS code (Or in general terms it compiles and executes the JS code). NodeJS takes the JS part from chrome and adds other functionality to perform system level actions. Node makes use of V8 to run JS code outside a browser.

It allows us to build scalable network applications, and is very fast when compared with other server side programming languages because it is written in C and the non-blocking I/O model.

We’ll discuss the io model later.

Install node. Follow instructions here:

<https://nodejs.org/en/download/>

Exercise

Open terminal (in linux/mac) or cmd or powershell (in windows)

Type in node and hit enter.

This will start a new terminal. Its called node repl. **REPL** stands for Read-Eval-Print-Loop

Node repl is a node feature that allows you to type in js code and get output right away

this new terminal starts with a ‘>’ Meaning the REPL is ready to take commands, type in console.log(“Hello World”)

Hit enter. This will print Hello World in the next line.

Press ctrl+c twice to exit REPL.

Now put console.log(“Hello World”) in a new file hello.js

Go to terminal and type in node {path-to}/hello.js and hit enter. You’ll see that this does the same output but without repl

Repl is like your browser console. You can try live js code in it.

We are not using REPL in this tutorial. But if you want to try something out, you can use it.

Node.js files must be initiated in the "Command Line Interface" program of your computer.

How to open the command line interface on your computer depends on the operating system. For Windows users, press the start button and look for "Command Prompt", or simply write "cmd" in the search field.

There are multiple versions of Node and with each new release, new language features are introduced.

We are using Nodejs v8.11 as its an LTS release which means it’s a stable release and Support and updates are guarantied for an extended period than regular releases.

Before we start take a look at the official documentation for nodejs here <https://nodejs.org/en/docs/>

Next thing you need is a text editor to edit your code. You could use notepad if you want, but there are better tools to do that. I recommend visual studio code. Sublime text3 or atom are other capable alternatives.

NodeJS is a framework built on top of the concept of non-blocking IO. Most IO operations in Node are asynchronous which means an IO operation might not result immediately but the js code will continue executing without waiting for the IO operation to complete. Example of IO operations are file-read, file –write, network-operations etc.

Our objective with this tutorial is to create a web server with nodejs. Unlike many stacks NodeJS comes with a n HTTP server in-built.

One more tool we’ll be using from here on is **git** . You can download the latest version from <https://git-scm.com/> . For most linux distributions and MacOs git can be installed through the respective package managers. On windows, you should download the installer from git website and run it. (Git experience might not be very smooth on windows )

Git is a version control system for tracking changes in computer files and coordinating work on those files among multiple people. It keeps the entire edit history in a nice organized manner. You’ll be wondering how git does that. Well, git doesn’t do that by itself, the developer does that using git.

So this is how it goes… You are editing a bunch of source files. Whenever you feel like your files are in a state that is complete in itself, or worth keeping a marker to come back to, you make a commit there. A commit is a reference to the current state of files + a commit message so that you can recognize what it was. In this tutorial however, you will not have to commit anything. We give you a codebase which has all the commits already done. Each step in this tutorial will be a commit. You’ll have to switch through them to progress through the tutorial. This process is called **checkout** in git terms.

So to begin, first we should get a copy of the git repository. Create a folder to keep your project files. Then open a CLI window at that location. Now, to get a copy of the git repository, run the following command in CLI

git clone {git-repo-url}

You will now have an index.js file inside your project directory. This is empty for now. To get the code for our web server run:

git checkout ex2.1

Note that now the index.js contains some code. We’ll come to explanation in a bit. Before that we’ll run the web server to make sure its working for everyone.

node index.js

Now open the link <http://localhost:8080> in your browser. If you get a Hello World! In your browser, then its working. (Tip: if not working, try changing the port number or check the firewall options )

The code looks like this:

**var** http = require('http');

This is how you require a module in nodeJS. In node, every unit of functionality is modeled as a module. This makes it convenient to compose applications with just what you need. The http module is used to create a web server or make http requests from node.

http is a built-in module which means it is there in every node installation. But in case you need some extra functionality that is not built in, you can probably find it on npm. We’ll get to that in a bit.

**var** reqHandler = **function** (req, res) {

res.write('Hello World!'); *//write a response to the client*

res.end(); *//end the response*

};

Here reqHandler is a function that handles each http request. One interesting thing you might notice is that here a function is being assigned to a variable which is not very common in languages like Java or C++. That’s because in JavaScript, functions are first-class objects, because they can have properties and methods just like any other object. What distinguishes them from other objects is that functions can be called. In brief, they are Function objects.

*//create a server object:*

**var** server = http.createServer(reqHandler);

The createServer method returns a server object.

server.listen(8080); *//the server object listens on port 8080*

the number 8080 denotes the port number at which the server will be listening

opening the link <http://localhost:8080> in your browser sends an http request to our node server.

This request will be handed over to reqHandler which writes “Hello World” as response body and then ends the response.

Now let us make it a little more interesting.   
In reqHandler, make the response body to following

res.write('Hello World! you are on page'+req.url); *//write a response to the client*

req is the request object. It contain quiet a few properties relating to request.

req.url is the url path that the page is at. So instead of <http://localhost:8080> if we visit <http://localhost:8080/some-path> then req.url will contain the value “/some-path”.

# Intro to KOA

So creating a web server is very easy in node. But at the same time, this model is very difficult to use when your app is very complex with a large number of pages. That is when we rely on web application frameworks. We will be following KoaJS for this tutorial

Koa is a bunch of node modules that bring in extra functionality to node web servers.

But unfortunately Koa is not a built-in module. You have to install it from npm registery.

Npm=> node package manager.

You can search for and find packages for virtually anything on <https://www.npmjs.com/>

But to install them we rely on a cli tool called npm that comes bundled with node.

To install koaJS you can run following:

npm install koa

This command will create a node\_modules folder and download some files from npm registry into it.

But we are not doing any of that. Instead we will use some git magic to get everything ready for next step

git checkout ex2.2

You’ll see that there are a couple of new files added there. We’ll talk about that in a bit. Before that we’ll run the app and make sure its working

npm install  
node .

npm install will install all the dependency modules in package.json. So what exactly is this package.json file. It is a file that holds various metadata relevant to the project. When you install a module, add a -S flag to add that module to package.json as a dependency

npm install -S koa

node index.js

to run the app. This will again give the same Hello world message wjen you open <http://localhost:8080> in your browser. We will try the same exercise we did in previous example, i..e display the url path in page

TO do that change line 5 to

ctx.body = 'Hello World. You are @ '+ ctx.req.url;

Here ctx is the container object for request and response objects.

Now run

node index.js

and open <http://localhost:8080/your_name> in your browser.

# Routing in koa

Now this is cool and all, but this is not enough to make a website or an api server. For that we have to make our web server respond differently to different urls thrown at it. For example  
<http://localhost:8080/contact> and <http://localhost:8080/about> should show different content. To get the code for next exercise

git checkout ex2.2

This process of mapping a path to a handler function is generally called routing. To do that we’ll add a router module to koa.

npm install -S koa-router

Koa is a bare minimum framework. It handles the very basics of http request. Everything else is accomplished by a middleware. A middleware is a node module that plugs in to koa request handler chain their by adding more functionality. Koa-router is such a middleware.

Now if you look at the code, you’ll notice there are some changes.

Just like we included the Koa module, now in include koa-router module

**var** Router = require('koa-router');

In line 7, this code initializes a router middleware object. All of our routes (paths) will be configured on to this object

**const** router = new Router();

Then you’ll see this

router.get('/', (ctx, next) **=>** {

ctx.body = "Welcome"

});

This code tells router to print welcome if the path is / . What you see below is another way to add multiple route definitions in a continuous fashion. This is called chaining.

router

.get('/about', (ctx, next) **=>** {

ctx.body = "We are team irisind. To know more, go to contact page"

})

.get('/contact', (ctx, next) **=>** {

ctx.body = "Contact us at info@irisind.com"

});

We also changed something with the app.use line. Now it looks like this.

app

.use(router.routes())

.use(router.allowedMethods());

So what is happening here is simple. Before, we were handling the request with a custom handler function defined by us. Now, the request is handled by a handler function included in koa-router. That handler takes the burden of choosing which handler function should be called for each path it gets.

Finally there is this tiny little message added to let us know when the server starts. Remember PORT is a variable defined at the top of the code which contains the port number.

app.listen(PORT, ()**=>** console.log(`Server running on port ${PORT}`));

# Splitting into sub modules

Our web server is working fine. We have three routes and all display different messages as required. You might also notice that our code is growing longer and getting difficult to understand. What if we have 44 routes and each perform much complex tasks than printing a message. It will be a mess.

As a beginning step to organizing our code nice and tidy, we’ll split our routes as a sub module. So how do you do it. Simple, create a new file, move all routing code into it and export the router object.

Exercise

git checkout ex2.4

**Make students do it manually. Use ex2.4 for instructor reference only**

The code looks much cleaner in index.js right?

Routes.js is exactly the same code that you did in previous exercise. Except the last line

module.exports = router;

Also you’ll see this at line 6 in index.js

**const** router = require('./routes'); *// requires router config from ./routes.js*

What this does is imports whatever you give to module.exports in the source file.

Now run the code and see that everything is working fine as before. We’ll be splitting more code into submodules later on.