



NODE.JS

Part I

JAVASCRIPT

- ❑ The first incarnations of JavaScript lived in browsers
 - ❑ JavaScript is a “complete” language: you can use it in many contexts and achieve everything with it you can achieve with any other “complete” language
 - ❑ Node.js allows you to run JavaScript code in the backend, outside a browser.
 - ❑ It was built on top of the Google Chrome V8 engine
-
- ❑ `console.log("Hello World");`
-
- ❑ **node helloworld.js**
-
- ❑ Node.js is really two things: a runtime environment and a library.

MODULAR PROGRAMMING WITH NODE.JS

- ❑ With Node.js, we not only implement our application, we also implement the whole HTTP server
- ❑ In fact, our web application and its web server are basically the same
- ❑ This might sound like a lot of work, but we will see in a moment that it is not so
- ❑ It allows you to have a clean main file, which you execute with Node.js, and
- ❑ Clean modules that can be used by the main file and among each other
- ❑ It's more or less a standard to name your main file *index.js*.
- ❑ It makes sense to put our server module into a file named *server.js*.

WRITING HTTP SERVER

The first line *requires* the *http* module that ships with Node.js and makes it accessible through the variable *http*

- `var http = require('http');`

We then call one of the functions the *http* module offers: *createServer*

This function returns an object

This object has a method named *listen*, and takes a numeric value which indicates the port number our HTTP server is going to listen on.

- `var server = http.createServer(<function as argument>);`
- `server.listen(8888);`

Because in JavaScript, functions can be passed around like any other value.

FUNCTIONAL PROGRAMMING IN JAVASCRIPT

```
function say(word) {  
  console.log(word);  
}
```

```
function execute(someFunction, value) {  
  someFunction(value);  
}
```

```
execute(say, "Hello");
```

```
function execute(someFunction, value) {  
  someFunction(value);  
}
```

```
execute(function(word){ console.log(word) }, "Hello");
```

- ❑ We define the function we want to pass to *execute* right there at the place where *execute* expects its first parameter.
- ❑ This way, we don't even need to give the function a name, which is why this is called an *anonymous function*.

CALLBACKS

```
var result = database.query("SELECT * FROM hugetable");  
console.log("Hello World");
```

The JavaScript interpreter of Node.js first has to read the complete result set from the database

Then it can execute the *console.log()* function

According to the execution model of Node.js - there is only one single process

If there is a slow database query somewhere in this process, this affects the whole process - everything comes to a halt until the slow query has finished execution.

```
1 database.query("SELECT * FROM hugetable", function(rows) {  
2     var result = rows;  
3 });  
4 console.log("Hello World");
```

CALLBACKS

```
1 database.query("SELECT * FROM hugetable", function(rows) {  
2     var result = rows;  
3 });  
4 console.log("Hello World");
```

Node.js can handle the database request asynchronously.

Provided that *database.query()* is part of an asynchronous library, this is what Node.js does: just as before, it takes the query

Then sends it to the database.

Here, instead of expecting *database.query()* to directly return a result to us, we pass it a second parameter, an anonymous function.

But instead of waiting for it to be finished, it makes a mental note that says

- “When at some point in the future the database server is done and sends the result of the query, then I have to execute the anonymous function that was passed to *database.query()*.”

After printing to the console log, it goes to an event loop

Node.js continuously cycles through this loop again and again whenever there is nothing else to do, waiting for events.

CALLBACKS

```
function onRequest(request, response) {  
  console.log("Request received.");  
  response.writeHead(200, {"Content-Type": "text/plain"});  
  response.write("Hello World");  
  response.end();  
}  
  
http.createServer(onRequest).listen(8888);  
  
console.log("Server has started.");
```


EVENT LOOP IN HTTP SERVER

This also explains why our HTTP server needs a function it can call upon incoming requests

if Node.js would start the server and then just pause, waiting for the next request, continuing only when it arrives, that would be highly inefficient

Multi client support

- ❑ It's important to note that this asynchronous, single-threaded, event-driven execution model isn't an infinitely scalable performance option

Node.js is just one single process, and it can run on only one single CPU core.

Node.js supports cluster module that enables creation of child processes to be executed on separate cores of a multi-core machine

SCALING FOR MULTI-CORE CPUS

For scaling throughput on a webservice, you should run multiple Node.js servers on one box, one per core and split request traffic between them.

This provides excellent CPU-affinity and will scale throughput nearly linearly with core count.

```
if (cluster.isMaster) {  
  // Fork workers.  
  for (var i = 0; i < numCPUs; i++) {  
    cluster.fork();  
  }  
}  
else {  
  http.Server(function(req, res) { ... }).listen(8000); }
```

CREATING MODULES WITH NODE.JS

In C++ or C#, when we're talking about objects, we're referring to instances of classes or structs. Objects have different properties and methods, depending on which templates (that is, classes) they are instantiated from. That's not the case with JavaScript objects. In JavaScript, objects are just collections of name/value pairs - think of a JavaScript object as a dictionary with string keys.

- ❑ Somewhere within Node.js lives a module called "http", and we can make use of it in our own code
- ❑ By requiring it and assigning the result of the require to a local variable
- ❑ This makes our local variable an object that carries all the public methods the *http* module provides.
- ❑ It's common practice to choose the name of the module for the name of the local variable

```
var http = require("http");
```

CREATING MODULES WITH NODE.JS

❑ We can put the different parts of our application into different files and wire them together by making them modules

❑ The functionality our HTTP server needs to export is simple: scripts requiring our server module simply need to start the server.

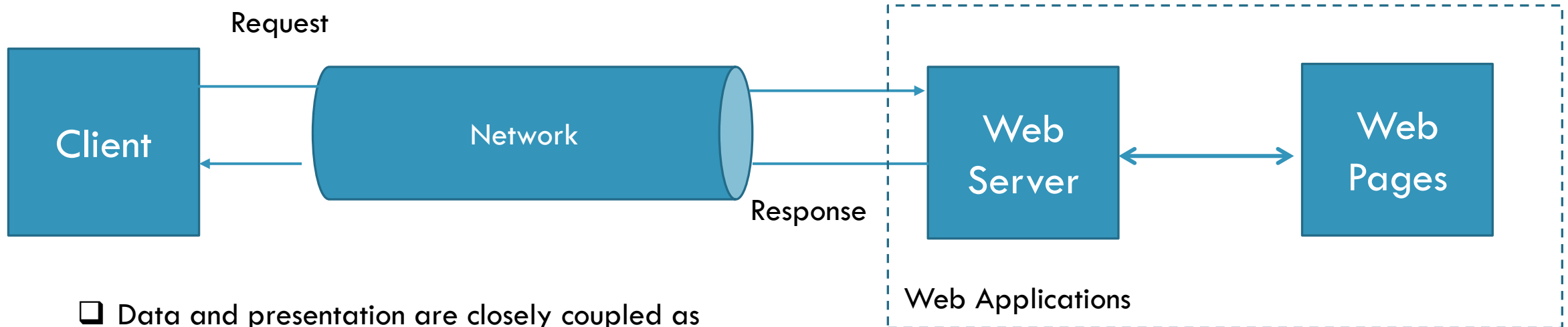
```
1  var server = require("./server");
2
3  server.start();
```

```
1  var http = require("http");
2
3  function start() {
4    function onRequest(request, response) {
5      console.log("Request received.");
6      response.writeHead(200, {"Content-Type": "text/plain"});
7      response.write("Hello World");
8      response.end();
9    }
10
11    http.createServer(onRequest).listen(8888);
12    console.log("Server has started.");
13  }
14
15  exports.start = start;
```

ROUTER

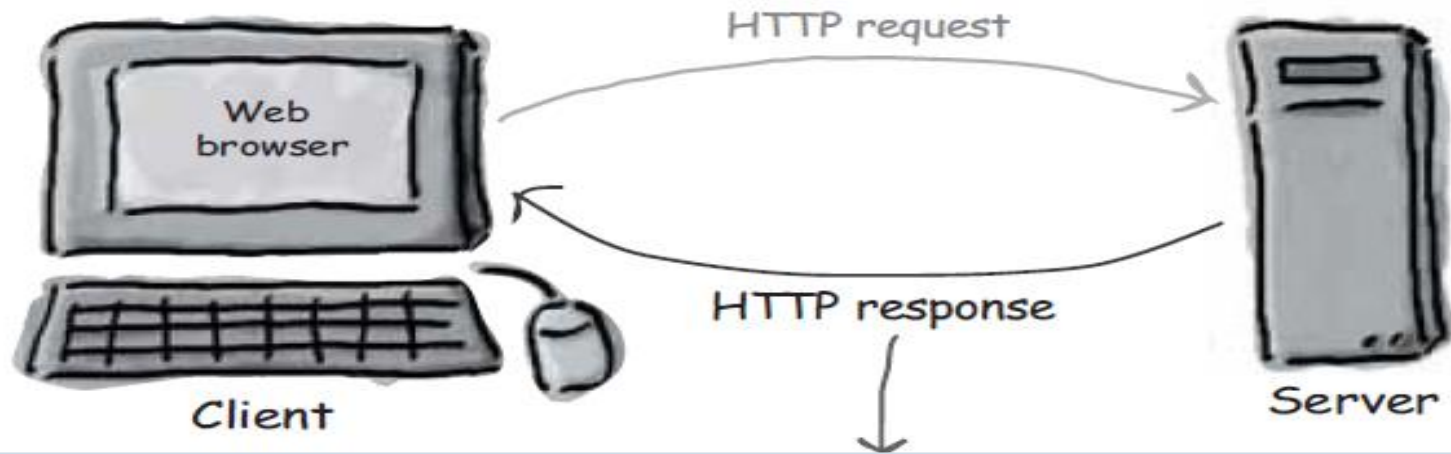
- ❑ Depending on which URL the browser requested from our server, we need to react differently
- ❑ For a very simple application, you could do this directly within the callback function *onRequest()*
- ❑ Making different HTTP requests point at different parts of our code is called “routing” – so, let’s create a module called *router*.
- ❑ We need to be able to feed the requested URL and possible additional GET and POST parameters into our router, and based on these the router then needs to be able to decide which code to execute
- ❑ To interpret the request object, we need two additional Node.js modules, namely *url* and *querystring*.
- ❑ *Querystring* can be used to parse the query string or the body of a POST request for parameters

WEB APP ARCHITECTURE-WEB 1.0



- ❑ Data and presentation are closely coupled as web pages are static

HTTP AND HTML



HTTP REQUEST

Every request has a method and a resource path

HTTP GET

- The total amount of characters in a GET is really limited (depending on the server)
- The data you send with the GET is appended to the URL up in the browser bar, so whatever you send is exposed
- Because of this, the user can bookmark a form submission if you use GET

HTTP POST

- The data is included in the request body
- More data can be sent
- General purpose sending of data

HTTP METHODS

Put

- Asking the server to store some Data

Delete

- Remove some information from the server

GET	PATH + Resource
POST	
PUT	
DELETE	

REQUEST LINE

GET /com/Kolkata/Home.html HTTP/ 1.1

Method

<path+resource>

The response may be stored by *any* cache, even if the response is normally non-cacheable. However, the stored response *MUST always* go through validation with the origin server first before using it

method

path

protocol

GET /tutorials/other/top-20-mysql-best-practices/ HTTP/1.1

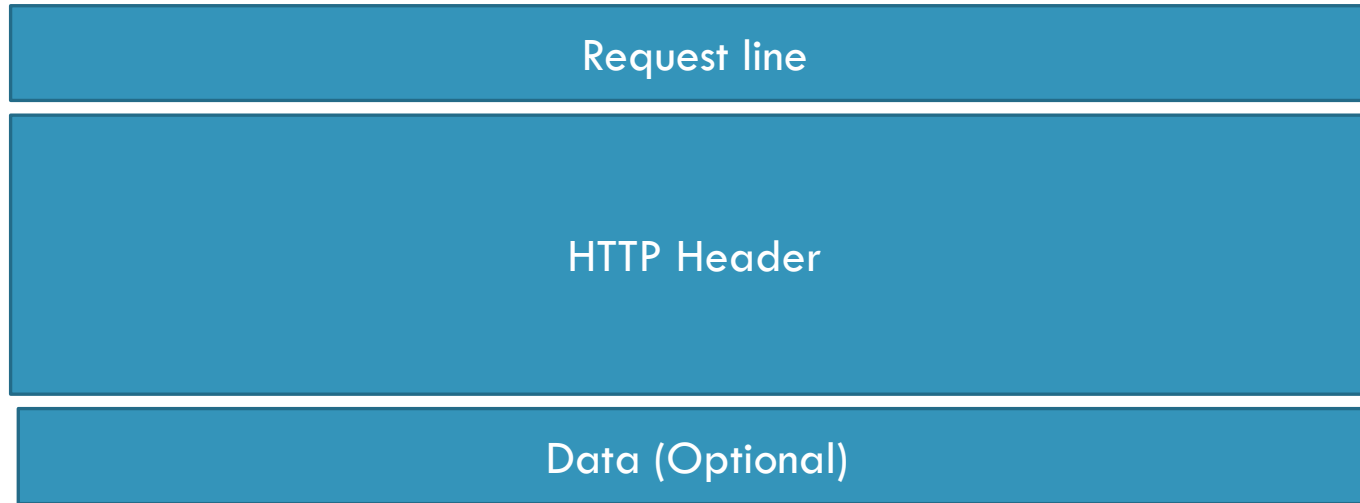
```
Host: net.tutsplus.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US; rv:1.9.1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Cookie: PHPSESSID=r2t5uvjq435r4q71b3vtdjq120
Pragma: no-cache
Cache-Control: no-cache
```

The server **MUST NOT** use a cached copy when responding to such a request.

HTTP headers as Name: Value

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers>

HTTP HEADERS



Headers are meta information but body of a HTTP message contains pure data

- These are the extra information that the client is giving the server to help it complete that Request.

If message body is sent without the header then the server may process the request

- May not send the response in the expected format

When body of a message is missing when it was required, the relevant information would not be processed

UNIFORM RESOURCE LOCATOR

`https://wishnet.in/home/login.html`

The way resources are identified is called a URL

`http://<host name>:<port number>/path/resource?key1=value1&key2=value2`

Using the query parameters we can pass extra information about a specific aspect of a resource to the server

URL encoding encodes any character that is not allowed in the query param spec

For dynamically constructed URLs with data, it is better to encode all URLs as data may not follow the spec

It is good to provide the correct file extension in the encoded URLs but not a requirement

PARSING A URL

Let's now add to our *onRequest()* function the logic needed to find out which URL path the browser requested

```
1  var http = require("http");
2  var url = require("url");
3
4  function start() {
5      function onRequest(request, response) {
6          var pathname = url.parse(request.url).pathname;
7          console.log("Request for " + pathname + " received.");
8          response.writeHead(200, {"Content-Type": "text/plain"});
9          response.write("Hello World");
10         response.end();
11     }
12
13     http.createServer(onRequest).listen(8888);
14     console.log("Server has started.");
15 }
16
17 exports.start = start;
```

ROUTER

```
1  function route(pathname) {  
2      console.log("About to route a request for " + pathname);  
3  }  
4  
5  exports.route = route;
```

Dependency Injection

```
var server = require("./server");  
var router = require("./router");  
  
server.start(router.route);
```

- ☐ For this case, the routing “ends” in the router
- ☐ The router is not the place to actually “do” something with the requests, because that wouldn’t scale well once our application becomes more complex

REQUEST HANDLING

In C++ or C#, when we're talking about objects, we're referring to instances of classes or structs. Objects have different properties and methods, depending on which templates (that is, classes) they are instantiated from. That's not the case with JavaScript objects.

In JavaScript, objects are just collections of name/value pairs - think of a JavaScript object as a dictionary with string keys.

- ❑ Value can be data or functions!
- ❑ A list of request handlers will be added based on different URL portions
- ❑ In index.js, the mapping of the handle to appropriate handlers have been done

```
var handle = {};  
    handle["/"] = requestHandlers.start;  
    handle["/start"] = requestHandlers.start;  
    handle["/upload"] = requestHandlers.upload;
```

- ❑ Thus, a handle is a collection of request handlers

SERVER FUNCTIONS

- ❑ Passing the list of handlers to the router
- ❑ the server would now need the handlers
`server.start(router.route, handle);`
- ❑ Passing the requested resource extracted from the URL to the router

```
route(handle, pathname);
```

```
var http = require("http");
var url = require("url");

function start(route, handle) {
  function onRequest(request, response) {
    var pathname = url.parse(request.url).pathname;
    console.log("Request for " + pathname + " received.");

    route(handle, pathname);

    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("Hello World");
    response.end();
  }

  http.createServer(onRequest).listen(8888);
  console.log("Server has started.");
}
```


ROUTER FUNCTIONS

```
1  function route(handle, pathname) {
2    console.log("About to route a request for " + pathname);
3    if (typeof handle[pathname] === 'function') {
4      handle[pathname]();
5    } else {
6      console.log("No request handler found for " + pathname);
7    }
8  }
9
10 exports.route = route;
```

Server → router → requestHandler

Server→router→requestHandler

HANDLING REQUESTS

Server→router→requestHandler→router→server

```
function start() {  
  console.log("Request handler 'start' was called.");  
  return "Hello Start";  
}  
  
function upload() {  
  console.log("Request handler 'upload' was called.");  
  return "Hello Upload";  
}  
  
exports.start = start;  
exports.upload = upload;
```

NONBLOCKING REQUEST HANDLING

Instead of expecting a return value from the *route()* function, we pass it a third parameter, our *response* object

Furthermore, we removed any *response* method calls from the *onRequest()* handler, because we now expect *route* to take care of that

```
1  function route(handle, pathname, response) {
2    console.log("About to route a request for " + pathname);
3    if (typeof handle[pathname] === 'function') {
4      handle[pathname](response);
5    } else {
6      console.log("No request handler found for " + pathname);
7      response.writeHead(404, {"Content-Type": "text/plain"});
8      response.write("404 Not found");
9      response.end();
10   }
11 }
12
13 exports.route = route;
```

WRITING A SEPARATE RESPONSE

```
function upload(response) {
  console.log("Request handler 'upload' was called.");
  response.writeHead(200, {"Content-Type": "text/html"});
  fs.readFile('./ChristmasPredicates.html', function
(error, data) {
    if (error) {
      response.writeHead(404);
      response.write('file not found');
    } else {
      response.write(data);
    }
    response.end();
  });
}
```

DEPENDENCY INJECTION

- ❑ Dependency Injection is a technique to wire an application together without any participation by the code that requires the dependency.
- ❑ The code usually exposes setter methods so that the framework may inject any needed dependencies.
- ❑ The code also becomes much more testable.

DEPENDENCY INJECTION

Index → Server → router → requestHandler

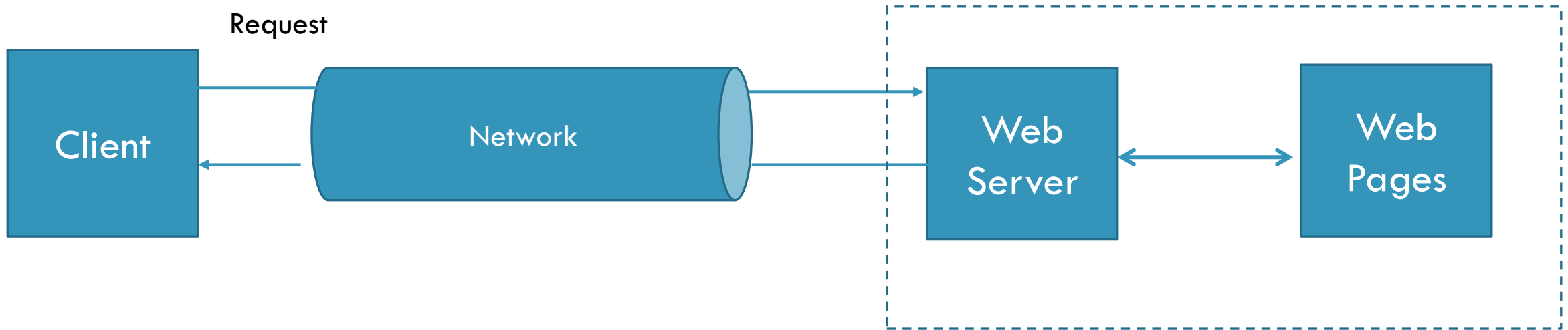
- ▶ it's time to actually write our router
- ▶ We could hard-wire this dependency into the server
- ▶ A better way is to loosely couple server and router by injecting this dependency
- ▶ In our index file, we could have passed the *router* object into the server, and the server could have called this object's *route* function.
- ▶ This way, we would have passed a *thing (an object)*, and the server would have used this thing to *do* something (a function).
- ▶ But the server doesn't need the thing (object). It only needs to get something *done (function)*

```
var server = require("./server");  
var router = require("./router");  
  
server.start(router.route);
```

Index → Server → router → requestHandler

```
app.get("/", function (req, res) {  
    res.send("Hello World!");  
});
```

```
app.get('/', (req, res) => {  
    res.sendFile(path.join(__dirname, 'views/show1.html'));  
});
```



PATH TO GET REQUEST

- User goes to `/` (ie. performs a **GET request** to `localhost:8000/`) => respond with **show.html**
- User goes to `/edit` (ie. performs a **GET request** to `localhost:8000/edit`) => respond with **edit.html**

```
1  const express = require('express');
2  const http = require('http');
3  const path = require('path');
4  const app = express();
5  const server = http.Server(app);

// Configuration

server.listen(process.env.PORT || 8000, () => {
  console.log(`[ server.js ] Listening on port ${server.address().port}`);
});

// Routes

app.get('/', (req, res) => {
  res.sendFile(path.join(__dirname, 'views/show.html'));
});

app.get('/edit', (req, res) => {
  res.sendFile(path.join(__dirname, 'views/edit.html'));
});
```