Leanpub

The Node Beginner Book

A comprehensive Node.js tutorial

Manuel Kiessling

The Node Beginner Book

A comprehensive Node.js tutorial

Manuel Kiessling

This book is for sale at http://leanpub.com/nodebeginner

This version was published on 2018-11-13

Tweet This Book!

Please help Manuel Kiessling by spreading the word about this book on Twitter!

The suggested hashtag for this book is #nodebeginner.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#nodebeginner

Also By Manuel Kiessling

Node入门

The Node Craftsman Book

El Libro Principiante de Node

Livro do Iniciante em Node

Beginning Mobile App Development with React Native

For Nora, Aaron and Melinda.

Contents

Special notice

Dear reader,

I would like to inform you about another book of mine which helps you to do even more with your newly learned JavaScript skills:

Beginning Mobile App Development with React Native

It is a comprehensive tutorial-style eBook that gets you from zero to native iOS mobile app development with JavaScript, HTML and CSS in no time.

Aimed at beginners in the field of mobile app programming, the book requires no prior knowledge in developing apps for iOS, and introduces everything that is needed to work with the React JavaScript framework in an easy-to-follow and comprehensive manner.

All that is required in order to dive into the book is a very basic understanding of how JavaScript code works. Everything else is introduced at just the right pace.

Now, the thing is that I have decided to make this a real steal: **It's only $5.99**. That's not a typo! Just go to

https://leanpub.com/beginning-mobile-app-development-with-react-native/

and download your PDF, ePub or MOBI copy for less than six dollars and start building mobile apps with JavaScript today!

About

The aim of this document is to get you started with developing applications for Node.js, teaching you everything you need to know about "advanced" JavaScript along the way. It goes way beyond your typical "Hello World" tutorial.

Status

You are reading the final version of this book, i.e., updates are only done to correct errors or to reflect changes in new versions of Node.js. It was last updated on June 5, 2017.

The code samples in this book are tested to work with both the Long Term Support version 6.10.3 as well as the most current 8.0.0 version of Node.js.

Intended audience

This document will probably fit best for readers that have a background similar to my own: experienced with at least one object-oriented language like Ruby, Python, PHP or Java, only little experience with JavaScript, and completely new to Node.js.

Aiming at developers that already have experience with other programming languages means that this document won't cover really basic stuff like data types, variables, control structures and the likes. You already need to know about these to understand this document.

However, because functions and objects in JavaScript are different from their counterparts in most other languages, these will be explained in more detail.

Structure of this document

Upon finishing this document, you will have created a complete web application which allows the users of this application to view web pages and upload files.

Which, of course, is not exactly world-changing, but we will go some extra miles and not only create the code that is "just enough" to make these use cases possible, but create a simple, yet complete framework to cleanly separate the different aspects of our application. You will see what I mean in a minute.

We will start with looking at how JavaScript development in Node.js is different from JavaScript development in a browser.

Next, we will stay with the good old tradition of writing a “Hello World” application, which is a most basic Node.js application that “does” something.

Then, we will discuss what kind of “real” application we want to build, dissect the different parts which need to be implemented to assemble this application, and start working on each of these parts step-by-step.

As promised, along the way we will learn about some of the more advanced concepts of JavaScript, how to make use of them, and look at why it makes sense to use these concepts instead of those we know from other programming languages.

JavaScript and Node.js

JavaScript and You

Before we talk about all the technical stuff, let's take a moment and talk about you and your relationship with JavaScript. This chapter is here to allow you to estimate if reading this document any further makes sense for you.

If you are like me, you started with HTML "development" long ago, by writing HTML documents. You came along this funny thing called JavaScript, but you only used it in a very basic way, adding interactivity to your web pages every now and then.

What you really wanted was "the real thing", you wanted to know how to build complex web sites - you learned a programming language like PHP, Ruby, Java, and started writing "backend" code.

Nevertheless, you kept an eye on JavaScript, you saw that with the introduction of jQuery, Prototype and the likes, things got more advanced in JavaScript land, and that this language really was about more than *window.open()*.

However, this was all still frontend stuff, and although it was nice to have jQuery at your disposal whenever you felt like spicing up a web page, at the end of the day you were, at best, a JavaScript *user*, but not a JavaScript *developer*.

And then came Node.js. JavaScript on the server, how cool is that?

You decided that it's about time to check out the old, new JavaScript. But wait, writing Node.js applications is the one thing; understanding why they need to be written the way they are written means - understanding JavaScript. And this time for real.

Here is the problem: Because JavaScript really lives two, maybe even three lives (the funny little DHMTL helper from the mid-90's, the more serious frontend stuff like jQuery and the likes, and now server-side), it's not that easy to find information that helps you to learn JavaScript the "right" way, in order to write Node.js applications in a fashion that makes you feel you are not just using JavaScript, you are actually developing it.

Because that's the catch: you already are an experienced developer, you don't want to learn a new technique by just hacking around and mis-using it; you want to be sure that you are approaching it from the right angle.

There is, of course, excellent documentation out there. But documentation alone sometimes isn't enough. What is needed is guidance.

My goal is to provide a guide for you.

A word of warning

There are some really excellent JavaScript people out there. I'm not one of them.

I'm really just the guy I talked about in the previous paragraph. I know a thing or two about developing backend web applications, but I'm still new to "real" JavaScript and still new to Node.js. I learned some of the more advanced aspects of JavaScript just recently. I'm not experienced.

Which is why this is no "from novice to expert" book. It's more like "from novice to advanced novice".

If I don't fail, then this will be the kind of document I wish I had when starting with Node.js.

Server-side JavaScript

The first incarnations of JavaScript lived in browsers. But this is just the context. It defines what you can do with the language, but it doesn't say much about what the language itself can do. 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 really is just another context: it allows you to run JavaScript code in the backend, outside a browser.

In order to execute the JavaScript you intend to run in the backend, it needs to be interpreted and, well, executed. This is what Node.js does, by making use of Google's V8 VM, the same runtime environment for JavaScript that Google Chrome uses.

Plus, Node.js ships with a lot of useful modules, so you don't have to write everything from scratch, like for example something that outputs a string on the console.

Thus, Node.js is really two things: a runtime environment and a library.

In order to make use of these, you need to install Node.js. Instead of repeating the process here, I kindly ask you to visit the official installation page[1]. Please come back once you are up and running.

"Hello World"

Ok, let's just jump in the cold water and write our first Node.js application: "Hello World".

Open your favorite editor and create a file called *helloworld.js*. We want it to write "Hello World" to STDOUT, and here is the code needed to do that:

[1] https://nodejs.org/en/download/

```
console.log("Hello World");
```

Save the file, and execute it through Node.js:

```
node helloworld.js
```

This should output *Hello World* on your terminal.

Ok, this stuff is boring, right? Let's write some real stuff.

A full blown web application with Node.js

The use cases

Let's keep it simple, but realistic:

- The user should be able to use our application with a web browser
- The user should see a welcome page when requesting http://localhost:8888/start which displays a file upload form
- By choosing an image file to upload and submitting the form, this image should then be uploaded to http://localhost:8888/upload, where it is displayed once the upload is finished

Fair enough. Now, you could achieve this goal by googling and hacking together *something*. But that's not what we want to do here.

Furthermore, we don't want to write only the most basic code to achieve the goal, however elegant and correct this code might be. We will intentionally add more abstraction than necessary in order to get a feeling for building more complex Node.js applications.

The application stack

Let's dissect our application. Which parts need to be implemented in order to fulfill the use cases?

- We want to serve web pages, therefore we need an **HTTP server**
- Our server will need to answer differently to requests, depending on which URL the request was asking for, thus we need some kind of **router** in order to map requests to request handlers
- To fullfill the requests that arrived at the server and have been routed using the router, we need actual **request handlers**
- The router probably should also treat any incoming POST data and give it to the request handlers in a convenient form, thus we need **request data handling**
- We not only want to handle requests for URLs, we also want to display content when these URLs are requested, which means we need some kind of **view logic** the request handlers can use in order to send content to the user's browser
- Last but not least, the user will be able to upload images, so we are going to need some kind of **upload handling** which takes care of the details

Let's think a moment about how we would build this stack with PHP. It's not exactly a secret that the typical setup would be an Apache HTTP server with mod_php5 installed.
Which in turn means that the whole "we need to be able to serve web pages and receive HTTP requests" stuff doesn't happen within PHP itself.

Well, with node, things are a bit different. Because 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 with Node.js, it's not.

Let's just start at the beginning and implement the first part of our stack, the HTTP server.

Building the application stack

A basic HTTP server

When I arrived at the point where I wanted to start with my first "real" Node.js application, I wondered not only how to actually code it, but also how to organize my code.
Do I need to have everything in one file? Most tutorials on the web that teach you how to write a basic HTTP server in Node.js have all the logic in one place. What if I want to make sure that my code stays readable the more stuff I implement?

Turns out, it's relatively easy to keep the different concerns of your code separated, by putting them in modules.

This 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.

So, let's create a main file which we use to start our application, and a module file where our HTTP server code lives.

My impression is that 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*.

Let's start with the server module. Create the file *server.js* in the root directory of your project, and fill it with the following code:

```
1  var http = require("http");
2
3  http.createServer(function(request, response) {
4    response.writeHead(200, {"Content-Type": "text/plain"});
5    response.write("Hello World");
6    response.end();
7  }).listen(8888);
```

That's it! You just wrote a working HTTP server. Let's prove it by running and testing it. First, execute your script with Node.js:

```
1  node server.js
```

Now, open your browser and point it at http://localhost:8888/. This should display a web page that says "Hello World".

That's quite interesting, isn't it. How about talking about what's going on here and leaving the question of how to organize our project for later? I promise we'll get back to it.

Analyzing our HTTP server

Well, then, let's analyze what's actually going on here.

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

We then call one of the functions the http module offers: *createServer*. This function returns an object, and 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.

Please ignore for a second the function definition that follows the opening bracket of *http.createServer*.

We could have written the code that starts our server and makes it listen at port 8888 like this:

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

var server = http.createServer();
server.listen(8888);
```

That would start an HTTP server listening at port 8888 and doing nothing else (not even answering any incoming requests).

The really interesting (and, if your background is a more conservative language like PHP, odd looking) part is the function definition right there where you would expect the first parameter of the *createServer()* call.

Turns out, this function definition IS the first (and only) parameter we are giving to the *createServer()* call. Because in JavaScript, functions can be passed around like any other value.

Passing functions around

You can, for example, do something like this:

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

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

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

Read this carefully! We pass the function *say* as the first parameter to the *execute* function. Not the return value of *say*, but *say* itself!

Thus, *say* becomes the local variable *someFunction* within *execute*, and execute can call the function in this variable by issuing *someFunction()* (adding brackets).

Of course, because *say* takes one parameter, *execute* can pass such a parameter when calling *someFunction*.

We can, as we just did, pass a function as a parameter to another function by its name. But we don't have to take this indirection of first defining, then passing it - we can define and pass a function as a parameter to another function in-place:

```
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*.

This is a first glimpse at what I like to call "advanced" JavaScript, but let's take it step by step. For now, let's just accept that in JavaScript, we can pass a function as a parameter when calling another function. We can do this by assigning our function to a variable, which we then pass, or by defining the function to pass in-place.

How function passing makes our HTTP server work

With this knowledge, let's get back to our minimalistic HTTP server:

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

http.createServer(function(request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello World");
  response.end();
}).listen(8888);
```

By now it should be clear what we are actually doing here: we pass the *createServer* function an anonymous function.

We could achieve the same by refactoring our code to:

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

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

http.createServer(onRequest).listen(8888);
```

Maybe now is a good moment to ask: Why are we doing it that way?

Event-driven asynchronous callbacks

To understand why Node.js applications have to be written this way, we need to understand how Node.js executes our code. Node's approach isn't unique, but the underlying execution model is different from runtime environments like Python, Ruby, PHP or Java.

Let's take a very simple piece of code like this:

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

Please ignore for now that we haven't actually talked about connecting to databases before - it's just an example. The first line queries a database for lots of rows, the second line puts "Hello World" to the console.

Let's assume that the database query is really slow, that it has to read an awful lot of rows, which takes several seconds.

The way we have written this code, the JavaScript interpreter of Node.js first has to read the complete result set from the database, and then it can execute the *console.log()* function.

If this piece of code actually was, say, PHP, it would work the same way: read all the results at once, then execute the next line of code. If this code would be part of a web page script, the user would have to wait several seconds for the page to load.

However, in the execution model of PHP, this would not become a "global" problem: the web server starts its own PHP process for every HTTP request it receives. If one of these requests results in the execution of a slow piece of code, it results in a slow page load for this particular user, but other users requesting other pages would not be affected.

The execution model of Node.js is different - 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.

To avoid this, JavaScript, and therefore Node.js, introduces the concept of event-driven, asynchronous callbacks, by utilizing an event loop.

We can understand this concept by analyzing a rewritten version of our problematic code:

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

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

In its previous form, our code was synchronous: *first* do the database query, and only when this is done, *then* write to the console.

Now, 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 and sends it to the database. 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()*."

Then, it immediately executes *console.log()*, and afterwards, it enters the event loop. Node.js continuously cycles through this loop again and again whenever there is nothing else to do, waiting for events. Events like, e.g., a slow database query finally delivering its results.

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 inefficent. If a second user requests the server while it is still serving the first request, that second request could only be answered after the first one is done - as soon as you have more than a handful of HTTP requests per second, this wouldn't work at all.

It's important to note that this asynchronous, single-threaded, event-driven execution model isn't an infinitely scalable performance unicorn with silver bullets attached. It is just one of several models, and it has its limitations. One being that as of now, Node.js is just one single process and it can run on only one single CPU core. Personally, I find this model quite approachable, because it allows you to write applications that have to deal with concurrency in an efficient and relatively straightforward manner.

You might want to take the time to read Felix Geisendoerfer's excellent post *Understanding node.js*[2] for additional background explanation.

Let's play around a bit with this new concept. Can we prove that our code continues after creating the server, even if no HTTP request happened and the callback function we passed isn't called? Let's try it:

[2] http://debuggable.com/posts/understanding-node-js:4bd98440-45e4-4a9a-8ef7-0f7ecbdd56cb

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

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.");
```

Note that I use *console.log* to output a text whenever the *onRequest* function (our callback) is triggered, and another text right *after* starting the HTTP server.

When we start this (*node server.js*, as always), it will immediately output "Server has started." on the command line. Whenever we request our server (by opening http://localhost:8888/ in our browser), the message "Request received." is printed on the command line.

Event-driven asynchronous server-side JavaScript with callbacks in action :-)

(Note that our server will probably write "Request received." to STDOUT two times upon opening the page in a browser. That's because most browser will try to load the favicon by requesting http://localhost:8888/favicon.ico whenever you open http://localhost:8888/).

How our server handles requests

Ok, let's quickly analyze the rest of our server code, that is, the body of our callback function *onRequest()*.

When the callback fires and our *onRequest()* function gets triggered, two parameters are passed into it: *request* and *response*.

Those are objects, and you can use their methods to handle the details of the HTTP request that occured and to respond to the request (i.e., to actually send something over the wire back to the browser that requested your server).

And our code does just that: Whenever a request is received, it uses the *response.writeHead()* function to send an HTTP status 200 and content-type in the HTTP response header, and the *response.write()* function to send the text "Hello World" in the HTTP response body.

At last, we call *response.end()* to actually finish our response.

At this point, we don't care for the details of the request, which is why we don't use the *request* object at all.

Finding a place for our server module

Ok, I promised we will get back to how to organize our application. We have the code for a very basic HTTP server in the file *server.js*, and I mentioned that it's common to have a main file called *index.js* which is used to bootstrap and start our application by making use of the other modules of the application (like the HTTP server module that lives in *server.js*).

Let's talk about how to make server.js a real Node.js module that can be used by our yet-to-be-written *index.js* main file.

As you may have noticed, we already used modules in our code, like this:

```
1 var http = require("http");
2
3 ...
4
5 http.createServer(...);
```

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, but we are free to choose whatever we like:

```
1 var foo = require("http");
2
3 ...
4
5 foo.createServer(...);
```

Fine, it's clear how to make use of internal Node.js modules. How do we create our own modules, and how do we use them?

Let's find out by turning our *server.js* script into a real module.

Turns out, we don't have to change that much. Making some code a module means we need to *export* those parts of its functionality that we want to provide to scripts that require our module.

For now, the functionality our HTTP server needs to export is simple: scripts requiring our server module simply need to start the server.

To make this possible, we will put our server code into a function named *start*, and we will export this function:

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

function start() {
  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.");
}

exports.start = start;
```

This way, we can now create our main file *index.js*, and start our HTTP there, although the code for the server is still in our *server.js* file.

Create a file *index.js* with the following content:

```
var server = require("./server");

server.start();
```

As you can see, we can use our server module just like any internal module: by requiring its file and assigning it to a variable, its exported functions become available to us.

That's it. We can now start our app via our main script, and it still does exactly the same:

```
node index.js
```

Great, we now can put the different parts of our application into different files and wire them together by making them modules.

We still have only the very first part of our application in place: we can receive HTTP requests. But we need to do something with them - 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()*. But as I said, let's add a bit more abstraction in order to make our example application a bit more interesting.

Making different HTTP requests point at different parts of our code is called "routing" - well, then, let's create a module called *router*.

What's needed to "route" requests?

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 (this "code to execute" is the third part of our application: a collection of request handlers that do the actual work when a request is received).

So, we need to look into the HTTP requests and extract the requested URL as well as the GET/POST parameters from them. It could be argued if that should be part of the router or part of the server (or even a module of its own), but let's just agree on making it part of our HTTP server for now.

All the information we need is available through the *request* object which is passed as the first parameter to our callback function *onRequest()*. But to interpret this information, we need some additional Node.js modules, namely *url* and *querystring*.

The *url* module provides methods which allow us to extract the different parts of a URL (like e.g. the requested path and query string), and *querystring* can in turn be used to parse the query string for request parameters:

```
                                url.parse(string).query
                                           |
           url.parse(string).pathname      |
                       |                   |
                       |                   |
                     ------ -------------------
http://localhost:8888/start?foo=bar&hello=world
                                  ---       -----
                                   |          |
                                   |          |
        querystring.parse(string)["foo"]    |
                                            |
                    querystring.parse(string)["hello"]
```

We can, of course, also use *querystring* to parse the body of a POST request for parameters, as we will see later.

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

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

function start() {
  function onRequest(request, response) {
        var pathname = url.parse(request.url).pathname;
        console.log("Request for " + pathname + " received.");
        response.writeHead(200, {"Content-Type": "text/plain"});
        response.write("Hello World");
        response.end();
  }

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

exports.start = start;
```

Fine. Our application can now distinguish requests based on the URL path requested - this allows us to map requests to our request handlers based on the URL path using our (yet to be written) router.

In the context of our application, it simply means that we will be able to have requests for the */start* and */upload* URLs handled by different parts of our code. We will see how everything fits together soon.

Ok, it's time to actually write our router. Create a new file called *router.js*, with the following content:

```
function route(pathname) {
  console.log("About to route a request for " + pathname);
}

exports.route = route;
```

Of course, this code basically does nothing, but that's ok for now. Let's first see how to wire together this router with our server before putting more logic into the router.

Our HTTP server needs to know about and make use of our router. We could hard-wire this dependency into the server, but because we learned the hard way from our experience with other programming languages, we are going to loosely couple server and router by injecting this dependency (you may want to read Martin Fowler's excellent post on Dependency Injection[3] for background information).

Let's first extend our server's *start()* function in order to enable us to pass the route function to be used by parameter:

[3] http://martinfowler.com/articles/injection.html

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

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

    route(pathname);

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

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

exports.start = start;
```

And let's extend our *index.js* accordingly, that is, injecting the route function of our router into the server:

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

server.start(router.route);
```

Again, we are passing a function, which by now isn't any news for us.

If we start our application now (*node index.js, as always*), and request an URL, you can now see from the application's output that our HTTP server makes use of our router and passes it the requested pathname:

```
bash$ node index.js
Request for /foo received.
About to route a request for /foo
```

(I omitted the rather annoying output for the /favicon.ico request).

Execution in the kingdom of verbs

May I once again stray away for a while and talk about functional programming again?

Passing functions is not only a technical consideration. With regard to software design, it's almost philosophical. Just think about it: 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*, and the server would have used this thing to *do* something. Hey, router thing, could you please route this for me?

But the server doesn't need the thing. It only needs to get something *done*, and to get something done, you don't need things at all, you need *actions*. You don't need *nouns*, you need *verbs*.

Understanding the fundamental mind-shift that's at the core of this idea is what made me really understand functional programming.

And I did understand it when reading Steve Yegge's masterpiece Execution in the Kingdom of Nouns[4]. Go read it now, really. It's one of the best writings related to software I ever had the pleasure to encounter.

Routing to real request handlers

Back to business. Our HTTP server and our request router are now best friends and talk to each other as we intended.

Of course, that's not enough. "Routing" means, we want to handle requests to different URLs differently. We would like to have the "business logic" for requests to */start* handled in another function than requests to */upload*.

Right now, the routing "ends" in the router, and 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.

Let's call these functions, where requests are routed to, *request handlers*. And let's tackle those next, because unless we have these in place there isn't much sense in doing anything with the router for now.

New application part, new module - no surprise here. Let's create a module called requestHandlers, add a placeholder function for every request handler, and export these as methods of the module:

[4] http://steve-yegge.blogspot.com/2006/03/execution-in-kingdom-of-nouns.html

```
function start() {
  console.log("Request handler 'start' was called.");
}

function upload() {
  console.log("Request handler 'upload' was called.");
}

exports.start = start;
exports.upload = upload;
```

This allows us to wire the request handlers into the router, giving our router something to route to.

At this point we need to make a decision: do we hard-code usage of the requestHandlers module into the router, or do we want a bit more dependency injection? Although dependency injection, like every other pattern, shouldn't be used only for the sake of using it, in this case it makes sense to loosely couple the router and its request handlers, and thus making the router really reusable.

This means we need to pass the request handlers from our server into our router, but this feels even more wrong, which is why we should go the whole way and pass them to the server from our main file, and passing it on to the router from there.

How are we going to pass them? Right now we have two handlers, but in a real application, this number is going to increase and vary, and we sure don't want to fiddle around mapping requests to handlers in the router anytime a new URL / request handler is added. And having some *if request == x then call handler y* in the router would be more than ugly.

A varying number of items, each mapped to a string (the requested URL)? Well, sounds like an associative array would be a perfect fit.

Well, this finding is slightly disappointed by the fact that JavaScript doesn't provide associative arrays - or does it? Turns out, it's actually objects that we want to use if we need an associative array!

There's a nice introduction to this at http://msdn.microsoft.com/en-us/magazine/cc163419.aspx, let me quote the relevant part:

> 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.

If JavaScript objects are just collections of name/value pairs, how can they have methods? Well, the values can be strings, numbers etc. - or functions!

Ok, now finally back to the code. We decided we want to pass the list of requestHandlers as an object, and in order to achieve loose coupling we want to inject this object into the *route()*.

Let's start with putting the object together in our main file *index.js*:

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

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

server.start(router.route, handle);
```

Although *handle* is more of a "thing" (a collection of request handlers), I propose we name it like a verb, because this will result in a fluent expression in our router, as we will see soon.

As you can see, it's really simple to map different URLs to the same request handler: by adding a key/value pair of *"/"* and *requestHandlers.start*, we can express in a nice and clean way that not only requests to */start*, but also requests to / shall be handled by the *start* handler.

After defining our object, we pass it into the server as an additional parameter. Let's change our *server.js* to make use of it:

```
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.");
}
```

```
19
20 exports.start = start;
```

We've added the *handle* parameter to our *start()* function, and pass the handle object on to the *route()* callback, as its first parameter.

Let's change the *route()* function accordingly, in our *router.js* file:

```
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;
```

What we do here is, we check if a request handler for the given pathname exists, and if it does, we simply call the according function. Because we can access our request handler functions from our object just as we would access an element of an associative array, we have this nice fluent *handle[pathname]();* expression I talked about earlier: "Please, *handle* this *pathname*".

Ok, that's all we need to wire server, router, and request handlers together! When starting our application and requesting http://localhost:8888/start in our browser, we can prove that the correct request handler was indeed called:

```
1 Server has started.
2 Request for /start received.
3 About to route a request for /start
4 Request handler 'start' was called.
```

And opening http://localhost:8888/ in our browser proves that these requests, too, are indeed handled by the *start* request handler:

```
1 Request for / received.
2 About to route a request for /
3 Request handler 'start' was called.
```

Making the request handlers respond

Beautiful. Now if only the request handlers could actually send something back to the browser, that would be even better, right?

Remember, the "Hello World" your browser displays upon requesting a page still comes from the *onRequest* function in our *server.js* file.

"Handling request" means "answering requests" after all, thus we need to enable our request handlers to speak with the browser just like our *onRequest* function does.

How to not do it

The straight-forward approach we - as developers with a background in PHP or Ruby - might want to follow is actually very deceitful: it works like a charm, seems to make a lot of sense, and then suddenly screws things up when we don't expect it.

What I mean by "straight-forward approach" is this: make the request handlers *return()* the content they want to display to the user, and send this response data in the *onRequest* function back to the user.

Let's just do this, and then see why it's not such an overly good idea.

We start with the request handlers and make them return what we would like to display in the browser. We need to change *requestHandlers.js* to this:

```
1  function start() {
2    console.log("Request handler 'start' was called.");
3    return "Hello Start";
4  }
5
6  function upload() {
7    console.log("Request handler 'upload' was called.");
8    return "Hello Upload";
9  }
10
11 exports.start = start;
12 exports.upload = upload;
```

Good. Likewise, the router needs to return to the server what the request handlers return to him. We therefore need to edit *router.js* like this:

```
function route(handle, pathname) {
  console.log("About to route a request for " + pathname);
  if (typeof handle[pathname] === 'function') {
    return handle[pathname]();
  } else {
    console.log("No request handler found for " + pathname);
    return "404 Not found";
  }
}

exports.route = route;
```

As you can see, we also return some text if the request could not be routed.

And last but not least, we need to refactor our server to make it respond to the browser with the content the request handlers returned via the router, transforming *server.js* into:

```
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.");

    response.writeHead(200, {"Content-Type": "text/plain"});
    var content = route(handle, pathname)
    response.write(content);
    response.end();
  }

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

exports.start = start;
```

If we start our rewritten application, everything works like a charm: requesting http://localhost:8888/start results in “Hello Start” being displayed in the browser, requesting http://localhost:8888/upload gives us “Hello Upload”, and http://localhost:8888/foo produces “404 Not found”.

Ok, then why is that a problem? The short answer: because we will run into problems if one the request handlers wants to make use of a non-blocking operation in the future.

Let’s take a bit more time for the long answer.

Blocking and non-blocking

As said, the problems will arise when we include non-blocking operations in the request handlers. But let's talk about blocking operations first, then about non-blocking operations.

And instead of trying to explain what "blocking" and "non-blocking" means, let's demonstrate ourselves what happens if we add a blocking operation to our request handlers.

To do this, we will modify our *start* request handler to make it wait 10 seconds before returning its "Hello Start" string. Because there is no such thing as *sleep()* in JavaScript, we will use a clever hack for that.

Please modify *requestHandlers.js* as follows:

```
function start() {
  console.log("Request handler 'start' was called.");

  function sleep(milliSeconds) {
    var startTime = new Date().getTime();
    while (new Date().getTime() < startTime + milliSeconds);
  }

  sleep(10000);
  return "Hello Start";
}

function upload() {
  console.log("Request handler 'upload' was called.");
  return "Hello Upload";
}

exports.start = start;
exports.upload = upload;
```

Just to make clear what that does: when the function *start()* is called, Node.js waits 10 seconds and only then returns "Hello Start". When calling *upload()*, it returns immediately, just like before.

(Of course, you should imagine that instead of sleeping for 10 seconds, there would be a real life blocking operation in *start()*, like some sort of long-running computation.)

Let's see what this change does.

As always, we need to restart our server. This time, I ask you to follow a slightly complex "protocol" in order to see what happens: First, open two browser windows or tabs. In the first browser window, please enter http://localhost:8888/start into the address bar, but do not yet open this url!

In the second browser window's address bar, enter http://localhost:8888/upload, and again, please do not yet hit enter.

Now, do as follows: hit enter on the first window ("/start"), then quickly change to the second window ("/upload") and hit enter, too.

What you will notice is this: The /start URL takes 10 seconds to load, as we would expect. But the /upload URL *also* takes 10 seconds to load, although there is no *sleep()* in the according request handler.

Why? Because *start()* contains a blocking operation. We already talked about Node's execution model - expensive operations are ok, but we must take care to not block the Node.js process with them. Instead, whenever expensive operations must be executed, these must be put in the background, and their events must be handled by the event loop.

And we will now see why the way we constructed the "request handler response handling" in our application doesn't allow us to make proper use of non-blocking operations.

Once again, let's try to experience the problem first-hand by modifying our application.

We are going to use our *start* request handler for this again. Please modify it to reflect the following (file *requestHandlers.js*):

```
1  var exec = require("child_process").exec;
2
3  function start() {
4    console.log("Request handler 'start' was called.");
5    var content = "empty";
6
7    exec("ls -lah", function (error, stdout, stderr) {
8      content = stdout;
9    });
10
11   return content;
12 }
13
14 function upload() {
15   console.log("Request handler 'upload' was called.");
16   return "Hello Upload";
17 }
18
19 exports.start = start;
20 exports.upload = upload;
```

As you can see, we just introduced a new Node.js module, *child_process*. We did so because it allows us to make use of a very simple yet useful non-blocking operation: *exec()*.

What *exec()* does is, it executes a shell command from within Node.js. In this example, we are going to use it to get a list of all files in the current directory ("ls -lah"), allowing us to display this list in the browser of a user requesting the */start* URL.

What the code does is straightforward: create a new variable *content* (with an initial value of "empty"), execute "ls -lah", fill the variable with the result, and return it.

As always, we will start our application, and visit http://localhost:8888/start.

Which loads a beautiful web page that displays the string "empty". What's going wrong here?

Well, as you may have already guessed, *exec()* does its magic in a non-blocking fashion. That's a good thing, because this way we can execute very expensive shell operations (like, e.g., copying huge files around or similar stuff) without forcing our application into a full stop as the blocking *sleep* operation did.

(If you would like to prove this, replace "ls -lah" with a more expensive operation like "find /").

But we aren't exactly happy with our elegant non-blocking operation, when our browser doesn't display its result, right?

Well, then, let's fix it. And while we are at it, let's try to understand why the current architecture doesn't work.

The problem is that *exec()*, in order to work non-blocking, makes use of a callback function.

In our example, it's an anonymous function which is passed as the second parameter to the *exec()* function call:

```
1  function (error, stdout, stderr) {
2    content = stdout;
3  }
```

And herein lies the root of our problem: our own code is executed synchronous, which means that immediately after calling *exec()*, Node.js continues to execute *return content;*. At this point, *content* is still "empty", due to the fact that the callback function passed to *exec()* has not yet been called - because *exec()* operates asynchronous.

Now, "ls -lah" is a very inexpensive and fast operation (unless there are millions of files in the directory). Which is why the callback is called relatively expeditious - but it nevertheless happens asynchronously.

Thinking about a more expensive command makes this more obvious: "find /" takes about 1 minute on my machine, but if I replace "ls -lah" with "find /" in the request handler, I still immediately receive an HTTP response when opening the /start URL - it's clear that *exec()* does something in the background, while Node.js itself continues with the application, and we may assume that the callback function we passed into *exec()* will be called only when the "find /" command has finished running.

But how can we achieve our goal, i.e. showing the user a list of files in the current directory?

Well, after learning how to *not* do it, let's discuss how to make our request handlers respond to browser requests the right way.

Responding request handlers with non-blocking operations

I've just used the phrase "the right way". Dangerous stuff. Quite often, there is no single "right way".

But one possible solution for this is, as often with Node.js, to pass functions around. Let's examine this.

Right now, our application is able to transport the content (which the request handlers would like to display to the user) from the request handlers to the HTTP server by returning it up through the layers of the application (request handler -> router -> server).

Our new approach is as follows: instead of bringing the content to the server, we will bring the server to the content. To be more precise, we will inject the *response* object (from our server's callback function *onRequest()*) through the router into the request handlers. The handlers will then be able to use this object's functions to respond to requests themselves.

Enough explanation, here is the step by step recipe on how to change our application.

Let's start with our *server.js*:

```
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);
  }

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

exports.start = start;
```

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.

Next comes *router.js*:

```
function route(handle, pathname, response) {
  console.log("About to route a request for " + pathname);
  if (typeof handle[pathname] === 'function') {
    handle[pathname](response);
  } else {
    console.log("No request handler found for " + pathname);
    response.writeHead(404, {"Content-Type": "text/plain"});
    response.write("404 Not found");
    response.end();
  }
}

exports.route = route;
```

Same pattern: instead of expecting a return value from our request handlers, we pass the *respond* object on.

If no request handler can be used, we now take care of responding with a proper "404" header and body ourselves.

And last but not least, we modify *requestHandlers.js*:

```
var exec = require("child_process").exec;

function start(response) {
  console.log("Request handler 'start' was called.");

  exec("ls -lah", function (error, stdout, stderr) {
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write(stdout);
    response.end();
  });
}

function upload(response) {
  console.log("Request handler 'upload' was called.");
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello Upload");
  response.end();
}

exports.start = start;
exports.upload = upload;
```

Our handler functions need to accept the response parameter, and have to make use of them in order to respond to the request directly.

The *start* handler will respond from within the anonymous *exec()* callback, and the *upload* handler still simply replies with "Hello Upload", but now by making use of the *response* object.

If we start our application again (*node index.js*), this should work as expected.

If you would like to prove that an expensive operation behind */start* will no longer block requests for */upload* from answering immediately, then modify your *requestHandlers.js* as follows:

```
var exec = require("child_process").exec;

function start(response) {
  console.log("Request handler 'start' was called.");

  exec("find /",
    { timeout: 10000, maxBuffer: 20000*1024 },
    function (error, stdout, stderr) {
      response.writeHead(200, {"Content-Type": "text/plain"});
      response.write(stdout);
      response.end();
    });
}

function upload(response) {
  console.log("Request handler 'upload' was called.");
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello Upload");
  response.end();
}

exports.start = start;
exports.upload = upload;
```

This will make HTTP requests to http://localhost:8888/start take at least 10 seconds, but requests to http://localhost:8888/upload will be answered immediately, even if /start is still computing.

Serving something useful

Until now, what we have done is all fine and dandy, but we haven't created any value for the customers of our award-winning website.

Our server, router, and request handlers are in place, thus now we can begin to add content to our site which allows our users to interact and walk through the use case of choosing a file, uploading

this file, and viewing the uploaded file in the browser. For the sake of simplicity we will assume that only image files are going to be uploaded and displayed through the application.

Ok, let's take it step by step, but with most of the techniques and principles of JavaScript explained by now, let's at the same time accelerate a bit. This author likes to hear himself talking way too much anyways.

Here, step by step means roughly two steps: We will first look at how to handle incoming POST requests (but not file uploads), and in a second step, we will make use of an external Node.js module for the file upload handling. I've chosen this approach for two reasons.

First, handling basic POST requests is relatively simple with Node.js, but still teaches us enough to be worth exercising it.
Second, handling file uploads (i.e., multipart POST requests) is *not* simple with Node.js, and therefore is beyond the scope of this tutorial, but using an external module is itself a lesson that makes sense to be included in a beginner's tutorial.

Handling POST requests

Let's keep this banally simple: We will present a textarea that can be filled by the user and submitted to the server in a POST request. Upon receiving and handling this request, we will display the content of the textarea.

The HTML for this textarea form needs to be served by our */start* request handler, so let's add it right away, in file *requestHandlers.js*:

```
function start(response) {
  console.log("Request handler 'start' was called.");

  var body = '<html>'+
    '<head>'+
    '<meta http-equiv="Content-Type" content="text/html; '+
    'charset=UTF-8" />'+
    '</head>'+
    '<body>'+
    '<form action="/upload" method="post">'+
    '<textarea name="text" rows="20" cols="60"></textarea>'+
    '<input type="submit" value="Submit text" />'+
    '</form>'+
    '</body>'+
    '</html>';

    response.writeHead(200, {"Content-Type": "text/html"});
    response.write(body);
```

```
19      response.end();
20  }
21
22  function upload(response) {
23    console.log("Request handler 'upload' was called.");
24    response.writeHead(200, {"Content-Type": "text/plain"});
25    response.write("Hello Upload");
26    response.end();
27  }
28
29  exports.start = start;
30  exports.upload = upload;
```

Now if this isn't going to win the Webby Awards, then I don't know what could. You should see this very simple form when requesting http://localhost:8888/start in your browser. If not, you probably didn't restart the application.

I hear you: having view content right in the request handler is ugly. However, I decided to not include that extra level of abstraction (i.e., separating view and controller logic) in this tutorial, because I think that it doesn't teach us anything worth knowing in the context of JavaScript or Node.js.

Let's rather use the remaining screen space for a more interesting problem, that is, handling the POST request that will hit our */upload* request handler when the user submits this form.

Now that we are becoming expert novices, we are no longer surprised by the fact that handling POST data is done in a non-blocking fashion, by using asynchronous callbacks.

Which makes sense, because POST requests can potentially be very large - nothing stops the user from entering text that is multiple megabytes in size. Handling the whole bulk of data in one go would result in a blocking operation.

To make the whole process non-blocking, Node.js serves our code the POST data in small chunks, callbacks that are called upon certain events. These events are *data* (a new chunk of POST data arrives) and *end* (all chunks have been received).

We need to tell Node.js which functions to call back to when these events occur. This is done by adding *listeners* to the *request* object that is passed to our *onRequest* callback whenever an HTTP request is received.

This basically looks like this:

```
request.addListener("data", function(chunk) {
  // called when a new chunk of data was received
});

request.addListener("end", function() {
  // called when all chunks of data have been received
});
```

The question arises where to implement this logic. We currently can access the *request* object in our server only - we don't pass it on to the router and the request handlers, like we did with the *response* object.

In my opinion, it's an HTTP servers job to give the application all the data from a requests it needs to do its job. Therefore, I suggest we handle the POST data processing right in the server and pass the final data on to the router and the request handlers, which then can decide what to do with it.

Thus, the idea is to put the *data* and *end* event callbacks in the server, collecting all POST data chunks in the *data* callback, and calling the router upon receiving the *end* event, while passing the collected data chunks on to the router, which in turn passes it on to the request handlers.

Here we go, starting with *server.js*:

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

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

    request.setEncoding("utf8");

    request.addListener("data", function(postDataChunk) {
      postData += postDataChunk;
      console.log("Received POST data chunk '"+
      postDataChunk + "'.");
    });

    request.addListener("end", function() {
      route(handle, pathname, response, postData);
    });

  }
```

```
23
24    http.createServer(onRequest).listen(8888);
25    console.log("Server has started.");
26  }
27
28  exports.start = start;
```

We basically did three things here: First, we defined that we expect the encoding of the received data to be UTF-8, we added an event listener for the "data" event which step by step fills our new *postData* variable whenever a new chunk of POST data arrives, and we moved the call to our router into the *end* event callback to make sure it's only called when all POST data is gathered. We also pass the POST data into the router, because we are going to need it in our request handlers.

Adding the console logging on every chunk that is received probably is a bad idea for production code (megabytes of POST data, remember?), but makes sense to see what happens.

I suggest playing around with this a bit. Put small amounts of text into the textarea as well as lots of text, and you will see that for the larger texts, the *data* callback is indeed called multiple times.

Let's add even more awesome to our app. On the /upload page, we will display the received content. To make this possible, we need to pass the *postData* on to the request handlers, in *router.js*:

```
1   function route(handle, pathname, response, postData) {
2     console.log("About to route a request for " + pathname);
3     if (typeof handle[pathname] === 'function') {
4       handle[pathname](response, postData);
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;
```

And in *requestHandlers.js*, we include the data in our response of the *upload* request handler:

```
function start(response, postData) {
  console.log("Request handler 'start' was called.");

  var body = '<html>'+
    '<head>'+
    '<meta http-equiv="Content-Type" content="text/html; '+
    'charset=UTF-8" />'+
    '</head>'+
    '<body>'+
    '<form action="/upload" method="post">'+
    '<textarea name="text" rows="20" cols="60"></textarea>'+
    '<input type="submit" value="Submit text" />'+
    '</form>'+
    '</body>'+
    '</html>';

    response.writeHead(200, {"Content-Type": "text/html"});
    response.write(body);
    response.end();
}

function upload(response, postData) {
  console.log("Request handler 'upload' was called.");
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("You've sent: " + postData);
  response.end();
}

exports.start = start;
exports.upload = upload;
```

That's it, we are now able to receive POST data and use it in our request handlers.

One last thing for this topic: what we pass on to the router and the request handlers is the complete body of our POST request. We will probably want to consume the individual fields that make up the POST data, in this case, the value of the *text* field.

We already read about the *querystring* module, which assists us with this:

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

function start(response, postData) {
  console.log("Request handler 'start' was called.");

  var body = '<html>'+
    '<head>'+
    '<meta http-equiv="Content-Type" content="text/html; '+
    'charset=UTF-8" />'+
    '</head>'+
    '<body>'+
    '<form action="/upload" method="post">'+
    '<textarea name="text" rows="20" cols="60"></textarea>'+
    '<input type="submit" value="Submit text" />'+
    '</form>'+
    '</body>'+
    '</html>';

    response.writeHead(200, {"Content-Type": "text/html"});
    response.write(body);
    response.end();
}

function upload(response, postData) {
  console.log("Request handler 'upload' was called.");
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("You've sent the text: "+
  querystring.parse(postData).text);
  response.end();
}

exports.start = start;
exports.upload = upload;
```

Well, for a beginner's tutorial, that's all there is to say about handling POST data.

Handling file uploads

Let's tackle our final use case. Our plan was to allow users to upload an image file, and display the uploaded image in the browser.

Back in the 90's this would have qualified as a business model for an IPO, today it must suffice to

teach us two things: how to install external Node.js libraries, and how to make use of them in our own code.

The external module we are going to use is *node-formidable* by Felix Geisendoerfer. It nicely abstracts away all the nasty details of parsing incoming file data. At the end of the day, handling incoming files is "only" about handling POST data - but the devil really *is* in the details here, and using a ready-made solution makes a lot of sense in this case.

In order to make use of Felix' code, the according Node.js module needs to be installed. Node.js ships with its own package manager, dubbed *NPM*. It allows us to install external Node.js modules in a very convenient fashion. Given a working Node.js installation, it boils down to issuing

```
npm install formidable
```

on our command line. If the following output ends similarly to this

```
added 1 package in 0.396s
```

then we are good to go.

The *formidable* module is now available to our own code - all we need to do is requiring it just like one of the built-in modules we used earlier:

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

The metaphor formidable uses is that of a form being submitted via HTTP POST, making it parseable in Node.js. All we need to do is create a new *IncomingForm*, which is an abstraction of this submitted form, and which can then be used to parse the *request* object of our HTTP server for the fields and files that were submitted through this form.

The example code from the node-formidable project page shows how the different parts play together:

```
var formidable = require('formidable'),
    http = require('http'),
    sys = require('sys');

http.createServer(function(req, res) {
  if (req.url == '/upload' && req.method.toLowerCase() == 'post') {
    // parse a file upload
    var form = new formidable.IncomingForm();
    form.parse(req, function(error, fields, files) {
      res.writeHead(200, {'content-type': 'text/plain'});
      res.write('received upload:\n\n');
```

```
12        res.end(sys.inspect({fields: fields, files: files}));
13      });
14      return;
15    }
16
17    // show a file upload form
18    res.writeHead(200, {'content-type': 'text/html'});
19    res.end(
20      '<form action="/upload" enctype="multipart/form-data" '+
21      'method="post">'+
22      '<input type="text" name="title"><br>'+
23      '<input type="file" name="upload" multiple="multiple"><br>'+
24      '<input type="submit" value="Upload">'+
25      '</form>'
26    );
27  }).listen(8888);
```

If we put this code into a file and execute it through *node*, we are able to submit a simple form, including a file upload, and see how the *files* object, which is passed to the callback defined in the *form.parse* call, is structured:

```
1  received upload:
2
3  { fields: { title: 'Hello World' },
4    files:
5     { upload:
6        { size: 1558,
7          path: '/tmp/1c747974a27a6292743669e91f29350b',
8          name: 'us-flag.png',
9          type: 'image/png',
10         lastModifiedDate: Tue, 21 Jun 2011 07:02:41 GMT,
11         _writeStream: [Object],
12         length: [Getter],
13         filename: [Getter],
14         mime: [Getter] } } }
```

In order to make our use case happen, what we need to do is to include the form-parsing logic of formidable into our code structure, plus we will need to find out how to serve the content of the uploaded file (which is saved into the */tmp* folder) to a requesting browser.

Let's tackle the latter one first: if there is an image file on our local hardrive, how do we serve it to a requesting browser?

We are obviously going to read the contents of this file into our Node.js server, and unsurprisingly, there is a module for that - it's called *fs*.

Let's add another request handler for the URL */show*, which will hardcodingly display the contents of the file */tmp/test.png*. It of course makes a lot of sense to save a real png image file to this location first.

We are going to modify *requestHandlers.js* as follows:

```
var querystring = require("querystring"),
    fs = require("fs");

function start(response, postData) {
  console.log("Request handler 'start' was called.");

  var body = '<html>'+
    '<head>'+
    '<meta http-equiv="Content-Type" '+
    'content="text/html; charset=UTF-8" />'+
    '</head>'+
    '<body>'+
    '<form action="/upload" method="post">'+
    '<textarea name="text" rows="20" cols="60"></textarea>'+
    '<input type="submit" value="Submit text" />'+
    '</form>'+
    '</body>'+
    '</html>';

    response.writeHead(200, {"Content-Type": "text/html"});
    response.write(body);
    response.end();
}

function upload(response, postData) {
  console.log("Request handler 'upload' was called.");
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("You've sent the text: "+
  querystring.parse(postData).text);
  response.end();
}

function show(response) {
  console.log("Request handler 'show' was called.");
  response.writeHead(200, {"Content-Type": "image/png"});
```

```
36    fs.createReadStream("/tmp/test.png").pipe(response);
37  }
38
39  exports.start = start;
40  exports.upload = upload;
41  exports.show = show;
```

We also need to map this new request handler to the URL */show* in file *index.js*:

```
1   var server = require("./server");
2   var router = require("./router");
3   var requestHandlers = require("./requestHandlers");
4
5   var handle = {};
6   handle["/"] = requestHandlers.start;
7   handle["/start"] = requestHandlers.start;
8   handle["/upload"] = requestHandlers.upload;
9   handle["/show"] = requestHandlers.show;
10
11  server.start(router.route, handle);
```

By restarting the server and opening http://localhost:8888/show in the browser, the image file saved at */tmp/test.png* should be displayed.

Fine. All we need to do now is

- add a file upload element to the form which is served at */start*,
- integrate node-formidable into the *upload* request handler, in order to save the uploaded file to */tmp/test.png*,
- embed the uploaded image into the HTML output of the */upload* URL.

Step 1 is simple. We need to add an encoding type of *multipart/form-data* to our HTML form, remove the textarea, add a file upload input field, and change the submit button text to “Upload file”. Let’s do just that in file *requestHandlers.js*:

```
var querystring = require("querystring"),
    fs = require("fs");

function start(response, postData) {
  console.log("Request handler 'start' was called.");

  var body = '<html>'+
    '<head>'+
    '<meta http-equiv="Content-Type" '+
    'content="text/html; charset=UTF-8" />'+
    '</head>'+
    '<body>'+
    '<form action="/upload" enctype="multipart/form-data" '+
    'method="post">'+
    '<input type="file" name="upload">'+
    '<input type="submit" value="Upload file" />'+
    '</form>'+
    '</body>'+
    '</html>';

    response.writeHead(200, {"Content-Type": "text/html"});
    response.write(body);
    response.end();
}

function upload(response, postData) {
  console.log("Request handler 'upload' was called.");
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("You've sent the text: "+
  querystring.parse(postData).text);
  response.end();
}

function show(response) {
  console.log("Request handler 'show' was called.");
  response.writeHead(200, {"Content-Type": "image/png"});
  fs.createReadStream("/tmp/test.png").pipe(response);
}

exports.start = start;
exports.upload = upload;
exports.show = show;
```

Great. The next step is a bit more complex of course. The first problem is: we want to handle the file upload in our *upload* request handler, and there, we will need to pass the *request* object to the *form.parse* call of node-formidable.

But all we have is the *response* object and the *postData* array. Sad panda. Looks like we will have to pass the *request* object all the way from the server to the router to the request handler. There may be more elegant solutions, but this approach should do the job for now.

And while we are at it, let's remove the whole *postData* stuff in our server and request handlers - we won't need it for handling the file upload, and it even raises a problem: we already "consumed" the *data* events of the *request* object in the server, which means that *form.parse*, which also needs to consume those events, wouldn't receive any more data from them (because Node.js doesn't buffer any data).

Let's start with *server.js* - we remove the postData handling and the *request.setEncoding* line (which is going to be handled by node-formidable itself), and we pass *request* to the router instead:

```
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, request);
  }

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

exports.start = start;
```

Next comes *router.js* - we don't need to pass *postData* on anymore, and instead pass *request*:

```
function route(handle, pathname, response, request) {
  console.log("About to route a request for " + pathname);
  if (typeof handle[pathname] === 'function') {
    handle[pathname](response, request);
  } else {
    console.log("No request handler found for " + pathname);
    response.writeHead(404, {"Content-Type": "text/html"});
    response.write("404 Not found");
    response.end();
```

```
  }
}

exports.route = route;
```

Now, the *request* object can be used in our *upload* request handler function. node-formidable will handle the details of saving the uploaded file to a local file within */tmp*, but we need to make sure that this file is renamed to */tmp/test.png* ourselves. Yes, we keep things really simple and assume that only PNG images will be uploaded.

There is a bit of extra-complexity in the rename logic: the Windows implementation of node doesn't like it when you try to rename a file onto the position of an existing file, which is why we need to delete the file in case of an error.

Let's put the pieces of managing the uploaded file and renaming it together now, in file *requestHandlers.js*:

```
var querystring = require("querystring"),
    fs = require("fs"),
    formidable = require("formidable");

function start(response) {
  console.log("Request handler 'start' was called.");

  var body = '<html>'+
    '<head>'+
    '<meta http-equiv="Content-Type" '+
    'content="text/html; charset=UTF-8" />'+
    '</head>'+
    '<body>'+
    '<form action="/upload" enctype="multipart/form-data" '+
    'method="post">'+
    '<input type="file" name="upload" multiple="multiple">'+
    '<input type="submit" value="Upload file" />'+
    '</form>'+
    '</body>'+
    '</html>';

    response.writeHead(200, {"Content-Type": "text/html"});
    response.write(body);
    response.end();
}

function upload(response, request) {
```

```
28    console.log("Request handler 'upload' was called.");
29
30    var form = new formidable.IncomingForm();
31    console.log("about to parse");
32    form.parse(request, function(error, fields, files) {
33      console.log("parsing done");
34
35      /* Possible error on Windows systems:
36         tried to rename to an already existing file */
37      fs.rename(files.upload.path, "/tmp/test.png", function(error) {
38        if (error) {
39          fs.unlink("/tmp/test.png");
40          fs.rename(files.upload.path, "/tmp/test.png");
41        }
42      });
43      response.writeHead(200, {"Content-Type": "text/html"});
44      response.write("received image:<br/>");
45      response.write("<img src='/show' />");
46      response.end();
47    });
48  }
49
50  function show(response) {
51    console.log("Request handler 'show' was called.");
52    response.writeHead(200, {"Content-Type": "image/png"});
53    fs.createReadStream("/tmp/test.png").pipe(response);
54  }
55
56  exports.start = start;
57  exports.upload = upload;
58  exports.show = show;
```

And that's it. Restart the server, and the complete use case will be available. Select a local PNG image from your hardrive, upload it to the server, and have it displayed in the web page.

Conclusion and outlook

Congratulations, our mission is accomplished! We wrote a simple yet full-fledged Node.js web application. We talked about server-side JavaScript, functional programming, blocking and non-blocking operations, callbacks, events, custom, internal and external modules, and a lot more.

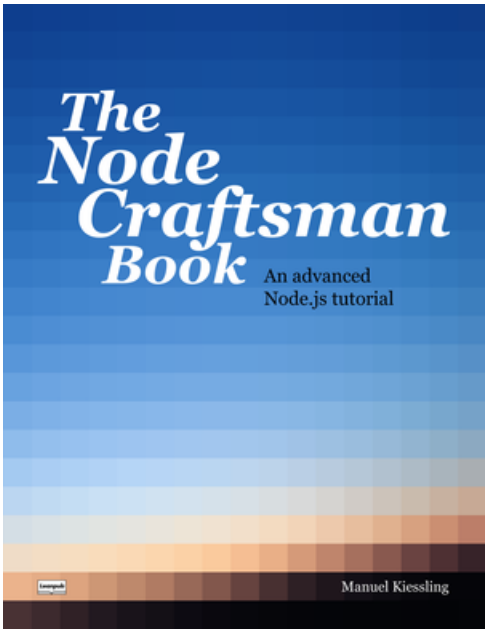
Of course, there's a lot of stuff we did not talk about: how to talk to a database, how to write unit tests, how to create external modules that are installable via NPM, or even something simple like how to handle GET requests.

But that's the fate of every book aimed at beginners - it can't talk about every single aspect in every single detail.

You can ignore the following info if you already own *The Node Craftsman Book*.

A lot of readers have asked for a sequel, a book that starts off where *The Node Beginner Book* ends, allowing them to dive deeper into Node.js development, learning about database handling, frameworks, unit tests, and much more.

This sequel book is now available. It is called *The Node Craftsman Book*, and it is available through Leanpub, too.



At 170 pages, it features the following chapters:

- Working with NPM and Packages
- Test-Driven Node.js Development
- Object-Oriented JavaScript
- Synchronous and Asynchronous operations explained
- Using and creating Event Emitters
- Optimizing code performance and control flow management using the async library
- Node.js and MySQL
- Node.js and MongoDB
- Building a complete web application with Node.js and AngularJS

The regular price of *The Node Craftsman Book* is $19.99, but as a reader and buyer of *The Node Beginner Book*, you have the opportunity to buy it for a discounted price of only **$9.99**.

If you would like to accept this limited offer, simply go to https://leanpub.com/nodecraftsman/c/nodereader.

Preview: The Node and React Beginner Book

Dear reader,

I have recently started to work on a new Node.js related book. It has a broader scope than "The Node Beginner Book", covering React *and* Node.js.

As a reader of "The Node Beginner Book", I would love to share the current draft and incomplete version with you, and I'm kindly asking for any feedback you have. The following really is only a rough cut, but I'm confident that it already contains a lot of useful stuff for Node.js beginners. It would be great if you gave it a try and send me your thoughts and criticism at manuel@kiessling.net[5].

Preface

About

Welcome to **The Node.js and React Beginner Book**. The aim of this book is to teach you everything you need to know to build full-blown web applications that are useful, reliable, and fast, using modern JavaScript features and tools.

We will start with the basics and build on these step-by-step. Everything is explained in detail and taught at just the right pace, making sure that everything sticks and can be understood easily.

All you need to bring to the table is a basic understanding of programming in general.

At the end of the book, you will have built a simple yet complete Node.js HTTP web server that provides a JSON-based REST API, and a simple yet complete client-side React single-page application that will use this API.

Together, the Node.js API and the React SPA will provide a web-based Todo management application - your first real Node.js and React application.

Status

You are reading the work-in-progress version of this book. It was last updated on November 13, 2018.

The code samples in this book are tested to work with version 10.x of Node.js.

[5] mailto:manuel@kiessling.net

Intended audience

This document will probably fit best for readers that have a background similar to my own: experienced with at least one object-oriented language like Ruby, Python, PHP or Java, only little experience with JavaScript, and completely new to Node.js and React.

Aiming at developers that already have experience with other programming languages means that this document won't cover really basic stuff - you should already know what a variable is, how an `if` statement works, what a `string` and an `integer` are, and how to run commands on the Command Line Interface of your computer.

However, because functions and objects in JavaScript are different from their counterparts in most other languages, these will be explained in more detail.

Getting started

JavaScript and You

Before we dive into all the technical stuff, let's take a moment and talk about you and your relationship with JavaScript. This chapter is here to allow you to estimate if reading this document any further makes sense for you.

If you are like me, you started with HTML "development" long ago, by writing HTML documents. You came across this funny thing called JavaScript, but you only used it in a very basic way, adding interactivity to your web pages every now and then.

What you really wanted was "the real thing", you wanted to know how to build complex web sites - you learned a programming language like PHP, Ruby, or Java, and started writing "backend" code.

Nevertheless, you kept an eye on JavaScript, and you saw that with the introduction of jQuery, things got more advanced in JavaScript land, that this language really was about more than `window.open()`.

However, it was all still very frontend-oriented, add-some-dynamic-sprinkles-to-a-web-page stuff, and although it was nice to have jQuery at your disposal whenever you felt like spicing up a web page, at the end of the day you were, at best, a JavaScript user, but not a JavaScript developer.

And then came Node.js. JavaScript on the server, how cool is that? And then React and friends: Finally building complex user interfaces with JavaScript doesn't seem like a crazy idea anymore. Also, JavaScript itself evolved as a language: ES6 and JSX look really promising, but also a bit intimidating - what exactly is "Babel", and what language features can be used where?

You decided that it's about time to check out the old, new JavaScript. But wait, writing Node.js and React applications is one thing; understanding why they need to be written the way they are written means - understanding JavaScript. And this time for real.

Here is the problem: Because JavaScript really lives two, maybe even three lives (the funny little DHTML helper from the mid-90's, the more serious frontend stuff like jQuery and the likes, and

now server-side, React, ES6, JSX etc.), it's not that easy to find information that helps you to learn JavaScript the "right" way, in order to write Node.js applications in a fashion that makes you feel you are not just using JavaScript, you are actually developing it.

Because that's the catch: you already are an experienced developer, you don't want to learn a new technique by just hacking around and mis-using it; you want to be sure that you are approaching it from the right angle.

There is, of course, excellent documentation out there. But documentation alone sometimes isn't enough. What is needed is guidance.

My goal is to provide this kind of guidance.

Setting up your software development environment

In order to be able to create JavaScript applications with Node.js and React, you need to have a collection of programs and tools installed on your computer, which together make up the Software Development Environment that allows you to work efficiently and build great stuff.

This environment consists of the following components:

- A powerful code editor, also called Integrated Development Environment (IDE)
- The Node.js interpreter and its supporting tools, used to run your application code and to manage additional code libraries used by your own code
- A command line interpreter, also called a terminal, which you will use to run important commands on
- A web browser, in order to use and test the web applications which you will build

The browser and the terminal

It's more than likely that you already have a terminal and a web browser on your system. It doesn't matter if you use Firefox, Chrome, Microsoft Edge, Safari, or Opera. Personally, I prefer Firefox, but every modern web browser will do.

If you are working on a Windows system, your command line interface or terminal is `cmd.exe`. Simply type `cmd` with the Start menu open, and hit enter. This will open a terminal window.

On macOS, the terminal application is simply called "Terminal". Simply launch it from folder "Utilities", which resides in folder "Applications". Alternatively, you can hit CMD+Space to bring up Spotlight Search, and enter "Terminal" in order to launch it.

Linux systems come in so many different forms and shapes that I can only assume you know how to open a terminal window. On Ubuntu systems, it is simply a matter of clicking the Ubuntu icon in the upper left corner of the screen and typing "Terminal" into the search form.

The code editor

There a many different code editors / IDEs available that enable you to write Node.js and React JavaScript code. Which one fits best for you really is a matter of taste.

Personally, I use *IntelliJ IDEA Ultimate Edition* from JetBrains, but this is just one possible option.

A very good IDE that runs on all three platforms (Windows, macOS, Linux), is available for free, has great JavaScript, Node.js, and React support, and is well-regarded in the JavaScript developer community is *Visual Studio Code* from Microsoft, often simply called *Code*. Visit https://code.visualstudio.com/ to download it for your platform.

Simply follow the installation instructions, and you are good to go. In the course of this book we will also install some extensions for Node.js and React, but for now the base install does the job.

Node.js

Last but not least, we need to install Node.js itself. While a code editor is all that is needed to write JavaScript code, we would not be able to do anything useful with it. In order to turn our code into a running Node.js application, we need the so-called Node.js interpreter. This is a program which reads our JavaScript source code, interprets its meaning, and executes it. It also provides *NPM*, the Node.js Package Manager, which we will use to install and manage third party JavaScript libraries, among other things.

No matter what platform you are on, the first step is to head over to https://nodejs.org/en/download/.

There, you are presented with a choice between the *LTS* version and the *Current* version of Node.js. The LTS version is older (version 8.x as of this writing), but more stable and with a longer support window. However, by now, its End-of-life date is already earlier than the one for the *Current* release version 10.x, so we don't really loose out by choosing the latest and greatest. Therefore, this book assumes an installation of Node.js 10. Have a look at https://github.com/nodejs/Release to always get an up-to-date release overview.

With this out of the way, switch to the "Current" tab on the Downloads page, and then choose the package that fits your platform. For Windows, you need to choose between 32-bit and 64-bit - it is very likely that your Windows is 64-bit. Also, choose the .msi Installer, not the .zip Binary.

For macOS, choose the 64-bit .pkg Installer. In case you are already using *Homebrew* on your Mac (see https://brew.sh for more details), you might want to use that to install Node.js and NPM instead, via `brew install node@10 npm`.

On Linux, again the installation heavily depends on your distribution. If you are running Ubuntu 18.04, a simple `apt-get install nodejs npm` is all you need - however, this will install Node.js 8.x, not 10.x. That's quite likely not a problem for the course of this book. If, however, you still want to install the recommended 10.x version, then please have a look at https://www.digitalocean.com/community/tutorials/how-to-install-node-js-on-ubuntu-18-04#installing-using-a-ppa and follow the steps described there.

If you decided to use an installer for Windows or macOS, then please just follow the installer instructions. Note that on Windows, the installer will set up a special Node.js Command Line Interface - always start this one when you want to work with Node.js on the terminal!

You can verify that all is well by running the following two commands in a terminal session after you have finished the installation:

```
node -v
npm -v
```

The output should look similar to the following:

```
$> node -v
v10.12.0

~$ > npm -v
6.4.1
```

This shows that Node.js version 10.12.0 and NPM version 6.4.1 have been installed successfully.

Ready to go

With this, we are equipped with everything we need to conquer worlds of Node.js and React.

The first part of this book focuses on ES6 JavaScript in general and how to use it to write Node.js code. By doing to, we will build the first part of our Todo management application: an HTTP-based REST API.

In the second part of the book, we will switch our focus to the client-side and build the React single-page application for the Todo app, integrate it with the REST API, and thus complete the app.

Some first experiments

"Hello, World"

Ok, let's finally jump in the cold water and write our very first Node.js application: "Hello, World".

To do so, launch your code editor and create a new file called *helloworld.js*.

We want our first application to write "Hello, World" on the console (to stdout[6], to be precise), and here is the code needed to do that:

[6] https://en.wikipedia.org/wiki/Standard_streams#Standard_output_(stdout)

```
console.log("Hello, World");
```

Save the file, and make Node.js read, interpret, and execute it:

```
$> node helloworld.js

Hello, World
```

It's probably obvious that you need to first `cd` to the directory which contains the *helloworld.js* file. If you try to invoke Node.js on a file that does not exist, it will print an error, like this:

```
~$ > node wrongname.js
internal/modules/cjs/loader.js:583
    throw err;
    ^

Error: Cannot find module '/Users/manuelkiessling/wrongname.js'
    at Function.Module._resolveFilename (internal/modules/cjs/loader.js:581:15)
    at Function.Module._load (internal/modules/cjs/loader.js:507:25)
    at Function.Module.runMain (internal/modules/cjs/loader.js:742:12)
    at startup (internal/bootstrap/node.js:279:19)
    at bootstrapNodeJSCore (internal/bootstrap/node.js:752:3)
```

Note how the error message reads *Cannot find module*, and not *Cannot find file*. We will talk about the concept of modules later.

For now, let's dissect our application and experiment a bit.

If we put our innocent line of code under the microscope, we see that it consists of three different parts.

We are invoking a function named `log`, which is defined on an object called `console`, with one string parameter that has the value `Hello, World`;

Everything else is just language syntax: an object and its functions are separated by a dot `.`, function call parameters are enclosed in parentheses `()`, string values are enclosed in double `"` quotes (single `'` quotes are fine, too, and there is also the backtick operator which we will properly introduce later), and lines end with a semicolon `;` - which optional in some cases and promptly sparked its own religious war[7]. In this book, we will stick to double quotes and semicolons.

Can we also print numbers instead of strings? You bet we can:

[7] https://www.theregister.co.uk/2018/01/12/javascript_technical_group_semicolons/

```
console.log(42);
```

```
$> node helloworld.js

42
```

Mh... let's get a little crazy. What if we try to print out an object? The only object we know so far is `console` itself, so let's try to print that:

```
console.log(console);
```

```
$> node helloworld.js

Console {
  log: [Function: bound consoleCall],
  debug: [Function: bound consoleCall],
  info: [Function: bound consoleCall],
  dirxml: [Function: bound consoleCall],
  warn: [Function: bound consoleCall],
  error: [Function: bound consoleCall],
  dir: [Function: bound consoleCall],
...
```

Interesting! Note that I have abbreviated the output to save valuable ebook paper, but what we see here is that `console.log` is able to print the outline of an object and its members, which happen to be mostly functions. This is certainly handy!

You might wonder where `console` comes from in the first place. We didn't define an object of this name ourselves, and yet it's at our disposal.

This shows that Node.js is not only an interpreter that turns JavaScript code into a running application. It also ships with its own bundle of ready-to-use JavaScript code, the so-called "lib". And the lib itself is nothing mysterious: it's just JavaScript code. For example, you can check out the code for the `console.log` function on GitHub at /lib/console.js[8].

Let's play around further. We can define our own objects:

[8] https://github.com/nodejs/node/blob/0f84120/lib/console.js#L198-L204

```
const myObject = {};

console.log(myObject);
```

This is an object without any members, so the output is not exactly mind-boggling.

Note the `const` keyword we used to declare our object. It's one of three ways to declare a variable in ES6 JavaScript - `var`, `let` an `const`. Previous versions of JavaScript only supported `var`.

While variables declared using *var* and *let* can be reassigned to a new value, this doesn't work for those declared with *const*:

```
var v = "this is a var";

let l = "this is a let";

const c = "this is a const";

console.log(v);
console.log(l);
console.log(c);


v = "this is a var, changed";

l = "this is a let, changed";

c = "this is a const, changed";

console.log(v);
console.log(l);
console.log(c);
```

```
$> node helloworld.js

this is a var
this is a let
this is a const

/Users/manuelkiessling/helloworld.js:19

c = "this is a const, changed";
  ^
```

```
TypeError: Assignment to constant variable.
...
```

However, there is an important subtlety when we say that variables declared with *const* cannot be reassigned to a new value. This can be shown with the following example:

```
const myObject = {};

console.log(myObject);

myObject.info = "this object has been changed";

console.log(myObject);
```

```
$> node helloworld.js

{}
{ info: 'this object has been changed' }
```

The subtlety here is that this in fact is not a reassignment - `myObject` still refers to the same object instance that was assigned on the first line. On line 5, the object itself changes, but it is not a new object instance - it simply gains a member.

This is different from the following:

```
const myObject = {};

console.log(myObject);

myObject = { info: "this object has been changed" };

console.log(myObject);
```

Here, we do try to assign a completely new object instance to the variable `myObject`, and because `myObject` has been declared a `const`, this fails:

```
$> node helloworld.js

{}
/Users/manuelkiessling/helloworld.js:5

myObject = { info: "this object has been changed" };
         ^
TypeError: Assignment to constant variable.
```

My recommendation regarding the use of *var*, *let* and *const* is as follows:

- Never use *var*, because it doesn't give you anything useful in comparison to *let*, except for some irritating scope issues. The main reason that *var* is still part of the language is to avoid breaking old code.
- Use *const* whenever possible, because it protects you from accidentally reassigning values to variables, which can prevent several types of subtle bugs in your code.
- Use *let* when you know that you really want to reassign values. You will see that this surprisingly seldom is the case.

A polite logger

Back to our object. Let's change and extend it so that it has a more useful name, and provides a function that allows to do console logging in a polite way:

```
const politeConsole = {
  log: function(text) {
    console.log("For your consideration: " + text);
  }
};

politeConsole.log("Hello, World");
```

The output now looks like this:

```
$> node helloworld.js

For your consideration: Hello, World
```

Just like we called the `log` function on `console` before, we can now call a function `log` on our own `politeConsole` (which in turn calls `console.log`).

Let's now take a look at something that is not typically possible or commonly done in "conventional" languages like PHP or Java, but is very natural in JavaScript.

Assume that while we want our keep our `politeConsole` object in charge of making log output more polite, we want more freedom regarding the "typography" of the text that is written out.

For example, we might sometimes want to write out a console message in ALL UPPERCASE. We could call the `politeConsole.log` function as follows:

```
const politeConsole = {
  log: function(text) {
    console.log("For your consideration: " + text);
  }
};

politeConsole.log("Hello, World".toUpperCase());
```

but this would result in

```
For your consideration: HELLO, WORLD
```

This is not what we want - we want the whole text to be written in all uppercase, like this:

```
FOR YOUR CONSIDERATION: HELLO, WORLD
```

We could, of course, extend `politeConsole.log` accordingly - for example, by adding a boolean parameter `uppercase` that defaults to `false`, but can be passed with `true` by a function caller.

While this is not inherently wrong, it is not very flexible. What if later on we want to be able to have our log message written out all lowercase? Or base64? Or have it transformed into a JSON representation?

We would have to touch `politeConsole` again and again, and the signature of function `politeConsole.log` would become more and more complex.

But there is another, more flexible solution. Instead of having `politeConsole` decide on text formatting, we generalize the fact that the "polite" message might be transformed. The transformation itself however is done outside of `politeConsole`.

Passing functions as values

This can be achieved with the following steps:

- extend `politeConsole.log` with a parameter `transform` that takes a function as its value
- instead of directly writing out the `"For your consideration: " + text` with `console.log`, write out the result of `transform("For your consideration: " + text)` if `transform` is a function

- when calling `politeConsole.log`, callers optionally pass a transform function

This is possible because in JavaScript, functions are a so-called "first class citizens". You can define and call functions like in other languages, but you are not limited to that. Functions can be passed around and passed as parameters to other functions, just like passing numbers or strings.

Here is how this looks in our example:

```
const politeConsole = {
  log: function(text, transform) {
    let politeText = "For your consideration: " + text;
    if (typeof(transform) === "function") {
      politeText = transform(politeText);
    }
    console.log(politeText);
  }
};

function upperCaseText(text) {
    return text.toUpperCase();
}

politeConsole.log("Hello, World", upperCaseText);

politeConsole.log("Hello, World");
```

When you run this, the output is as follows:

```
FOR YOUR CONSIDERATION: HELLO, WORLD
For your consideration: Hello, World
```

There's quite a lot of new stuff going on in a few lines of code, so let's dissect it step by step.

The `log` function got an additional parameter, `transform`. In the function body, we analyze the type of the parameter that has been passed to us under that name. We check if it is a function, using `typeof`. If it is indeed a function, and not `null` or `undefined` or a `string` etc., then we know we can call it.

Which is what we do in this case: We call the `transform` function that got passed to us, passing, in turn, our already "politelized" text. Note how `politeConsole.log` doesn't know anything about the passed function (other than the fact that it is, indeed, a function).

The result of running `politeText` through `transform` is then re-assigned back to `politeText`, which is possible because we defined it with `let` instead of `const`.

If `transform` is not a function (e.g. because the caller hasn't passed anything for this parameter, or passed something else, like a string), then `politeText` is not transformed. Transformed or not, the final `politeText` is finally printed to the screen with `console.log`.

On line 15, we make use of this new capability. We call `politeConsole.log` with two parameters: the text itself, and the name of the function that should be used to transform the output before it is written to the console.

In this case, it's a simple function declared on lines 11-13. It takes a parameter `text`, and returns the uppercase version of that text.

Note how on line 15, the second function call parameter is `upperCaseText`, not `upperCaseText()` - it is important to not add the parentheses here! If we would add those, like this:

```
1  politeConsole.log("Hello, World", upperCaseText());
```

then what would be passed to `politeConsole.log` as the second parameter `transform` is not the function `upperCaseText`, but instead the **result** of calling `upperCaseText()` - which, in this case, would result in `TypeError: Cannot read property 'toUpperCase' of undefined`, because line 12, `return text.toUpperCase()`, cannot work if nothing has been passed for parameter `text` of function `upperCaseText`.

In other words: Always make sure to pass the **function**, not the **function result**.

Anonymous functions

The code above works as intended, but can be improved significantly. For example, we don't actually need to explicitly declare and name the `upperCaseText` function - we can "inline" this declaration, like so:

```
1  const politeConsole = {
2    log: function(text, transform) {
3      let politeText = "For your consideration: " + text;
4      if (typeof(transform) === "function") {
5        politeText = transform(politeText);
6      }
7      console.log(politeText);
8    }
9  };
10
11 politeConsole.log(
12     "Hello, World",
13     function(text) { return text.toUpperCase(); }
14 );
```

This way, we declared an *anonymous function* right where we want to pass it. It doesn't exist outside of the function call to `politeConsole.log` - it is declared inline, passed, used, and then discarded.

And thanks to the new ES6 language features of JavaScript, we can further refactor the code and make it even more concise (I'm only showing the `politeConsole.log` call now):

```
politeConsole.log(
    "Hello, World",
    text => text.toUpperCase()
);
```

No `function` keyword needed - the new `=>`, or "arrow" operator makes this a function declaration.

No need to put the function parameter in `()` parentheses, as long as it's the only parameter.

No need to put the function body in `{}` brackets, as long as it contains only one line.

No need for a `return` statement - in a body without brackets, the result of the body is implicitly returned.

This syntax is not limited to anonymous function declarations - here are a couple of function declarations that are all valid:

```
function upper1(text) {
  return text.toUpperCase();
}

const upper2 = text => text.toUpperCase();

const lowerAndUpper =
  (lowertext, uppertext) =>
    lowertext.toLowerCase() + uppertext.toUpperCase();

const complexLowerAndUpper = (lowertext, uppertext) => {
  if (lowertext === "") {
    console.error("lowertext parameter is empty!");
    return "";
  } else {
    return lowertext.toLowerCase() + uppertext.toUpperCase();
  }
};

console.log(upper1("upper"));

console.log(upper2("upper"));
```

```
23
24 console.log(lowerAndUpper("lower", "upper"));
25
26 console.log(complexLowerAndUpper("lower", "upper"));
27
28 console.log(complexLowerAndUpper("", "upper"));
```

From here on, we will declare all functions, anonymous and named, using the new and concise arrow operator syntax, and named function variables will be declared as `const`.

That said, we can apply a final optimization to our code:

```
1  const politeConsole = {
2    log: (text, transform) => {
3      let politeText = "For your consideration: " + text;
4      if (typeof(transform) === "function") {
5        politeText = transform(politeText);
6      }
7      console.log(politeText);
8    }
9  };
10
11 politeConsole.log("Hello, World", function(text) { return text.toUpperCase(); });
```

With this, even the function declaration on line 2 uses the short form.

Further readings

- Stevey's Blog Rants: Execution in the Kingdom of Nouns[9]

Your first real Node.js application

So, this has been a nice first trip into Node.js land, but now it's time to get more serious. Let's write a first real app that does something useful.

While it is possible to write nearly any kind of application imaginable with Node.js, including 3D applications, Node.js isn't a good fit for every type of application.

One area that is very natural for Node.js software development is network servers. And Node.js comes with all the batteries included that make writing an HTTP web server relatively straightforward.

Thus, this is our next step: writing a Node.js server application that responds to HTTP requests.

[9] https://steve-yegge.blogspot.com/2006/03/execution-in-kingdom-of-nouns.html

The requirements

More specifically, we will create a very simple REST API server. This API will allow us to add, list, and delete Todo items - the API will thus serve as the backend for the React single-page application that we are going to build in the course of this book.

When finished, the API will allow the following operations:

- Sending a POST request to `/api/todos/` with a JSON object like `{"title": "Hello, World"}` will create a new Todo item
- Sending a GET request to `/api/todos/` will return a list of all Todo items, like this: `[ {"id": 1, "title": "Hello, World"}, {"id": 2, "title": "Foo bar"} ]`
- Sending a DELETE request to `/api/todos/?id=:id` will remove the Todo item with the given id

In the final version, it is the React application which will send these requests and handle the responses from the API provided by the web server. However, we will fully implement the API server first, and test it with a pure HTTP client like `curl`, before we even start working on the React app.

> A note on the *DELETE* operation: it's a bit unconventional to put the id of the item to be deleted into the query string via `?id=:id`. Typically, a REST API endpoint for this kind of operation would be designed as `/api/todos/:id`, putting the *id* value into the URL path. I've chosen to break with this convention, because following it would result in code that is, at least for our current learning purposes, more complex than needed.

The structure of Node.js applications

Before we dive into the actual codebase of such an application, let's take a moment and have a look at how codebases of Node.js applications can be organized.

Of course, for small experiments or single-purpose scripts, putting all the JavaScript code into a single file, like we did in *helloworld.js*, is perfectly fine. Even for huge and complex code bases, there is nothing that forbids putting all the tens of thousands of lines of code into just one file. But as in practically any other programming language out there, this wouldn't exactly make life easier.

The primary structural component of Node.js code bases is the *module*, and the module system allows to split large code bases into multiple files.

There are three categories of modules that we can use in our applications:

- Modules that ship with Node.js

- Modules that are provided by third parties and that we make available in our own code bases via dependency management tools like NPM
- Modules that we write ourselves

We will get in contact with all three types of modules in due time.

Creating and using modules ourselves is simple. The central idea behind the module system is that the code in one file *exports* elements (e.g. an object), and code in another file *imports* the exported element, which enables it to use that element as if it was declared in the same file.

We can transform the code file that defines `politeConsole` into a module by exporting the `politeConsole` object, and import and use that object in another file.

To do so, rename *helloworld.js* into *politeConsole.js* and edit its contents, resulting in the following:

```
const politeConsole = {
  log: (text, transform) => {
    let politeText = "For your consideration: " + text;
    if (typeof(transform) === "function") {
      politeText = transform(politeText);
    }
    console.log(politeText);
  }
};

module.exports = politeConsole;
```

To transform our code file into a module that can be imported elsewhere, we simply removed any code outside of the `politeConsole` definition, and we assigned the `politeConsole` object to the attribute `exports` on special object `module`. This object is what makes the module system tick in Node.js - whatever we assign to its `exports` attribute in one file can be imported into and used in other code files.

To illustrate this, again create a file *helloworld.js* (remember that we renamed the existing *helloworld.js* file to *politeConsole.js* before).

Put the following code into *helloworld.js*:

```
const politeConsole = require("./politeConsole");

politeConsole.log("Hello, World");
```

As you can see, we declare a const `politeConsole`, but instead of *defining* its value ourselves, we *assign* a value using the special function `require`. This function takes the source of a module as its parameter.

For internal Node.js modules and modules that we manage as external dependencies (more on this later), the source of a module is simply its name. But because we want to refer to our own module in file *politeConsole.js*, we pass the path to the file containing the module as its source. Because file *helloworld.js* and file *politeConsole* are located in the same folder, and because the file extension isn't required, `./politeConsole` does the job. Feel free to use the full path, `./politeConsole.js`, if you like. Throughout the book, we will leave the file extension out.

I would like to stress that really nothing special happens by exporting stuff in a module via `module.exports` and importing it via `require`. For example, instead of passing the whole object, you can pass only the function defined on its `log` attribute:

politeconsole.js:

```
const politeConsole = {
  log: (text, transform) => {
    let politeText = "For your consideration: " + text;
    if (typeof(transform) === "function") {
      politeText = transform(politeText);
    }
    console.log(politeText);
  }
};

module.exports = politeConsole.log;
```

helloworld.js:

```
const politeConsoleLog = require("./politeConsole");

politeConsoleLog("Hello, World");
```

Furthermore, you may want to export multiple things from a module if it defines more than one thing - no problem, `module.exports` can be an object with multiple attributes:

politeConsole.js:

```
const normalPoliteConsole = {
  log: (text, transform) => {
    let politeText = "For your consideration: " + text;
    if (typeof(transform) === "function") {
      politeText = transform(politeText);
    }
    console.log(politeText);
  }
};


const extremePoliteConsole = {
  log: (text, transform) => {
    let politeText = "For your consideration, your highness: " + text;
    if (typeof(transform) === "function") {
      politeText = transform(politeText);
    }
    console.log(politeText);
  }
};


module.exports = {
  normalPoliteConsole: normalPoliteConsole,
  extremePoliteConsole: extremePoliteConsole
};
```

helloworld.js:

```
const politeConsole = require("./politeConsole");

politeConsole.normalPoliteConsole.log("Hello, World");
politeConsole.extremePoliteConsole.log("Hello, World");
```

The version of *politeConsole.js* above presents the opportunity to introduce another ES6 nicety. Instead of explicitly declaring object attributes as key-value pairs, like so:

```
module.exports = {
  normalPoliteConsole: normalPoliteConsole,
  extremePoliteConsole: extremePoliteConsole
};
```

we can as well declare the object attributes using only the values:

```
module.exports = {
  normalPoliteConsole,
  extremePoliteConsole
};
```

This is called *Object Property Value Shorthand*, and for it to work as expected, naming things correctly obviously is important.

While in the traditional syntax something like

```
const a = "foo";

const obj = {
    b: a
};

console.log(obj.b);
```

works,

```
const a = "foo";

const obj = {
    a
};

console.log(obj.b);
```

cannot work because there is no way for the JavaScript interpreter to find out what you mean when accessing `obj.b`.

Even more sophisticated patterns of exporting and importing things via the module system are available, and we will get to those later in the book.

Launching an HTTP server using built-in Node.js modules

For our first real Node.js application, we will use what we learned about modularization and split our code base into multiple files, which will avoid that we end up with one large file full of spaghetti code.

Additionally, we will now also use internal Node.js modules. Because our mission is to write a Node.js server application that responds to HTTP requests, and because Node.js provides an internal module that allows to do just that, let's start on a blank canvas and create a new project folder *todos-app*, a subfolder *backend* within directory *todos-app*, and within with folder *backend*, a *server.js* file that imports and uses the `http` module:

```
// backend/server.js

const http = require("http");

http.createServer((request, response) => {
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("Hello, World");
    response.end();
}).listen(8000);
```

> Note that from here on, we assume that our project folder is *todos-app*, that on the command line, this is the current directory, and thus all file paths like *backend/server.js* are relative to the *todos-app* folder

As you can see, we declare a *const* named `http`, and assign it the value that results from calling `require("http")`, with *"http"* being the name of the internal Node.js module we want to use. In this case, it is not a file path name - it's just a name that Node.js knows how to resolve to this module. Of course, the code for the module does live in a file at the end of the day - have a look at /lib/http.js in the Node.js GitHub repository[10] if you are interested.

With this, `http` is now an object that provides the function needed to create an HTTP server - `createServer`. This function takes one parameter - a function that `createServer` will call whenever a client issues a new HTTP request against our server. When calling the function, `createServer` will pass two parameters, `request` and `response`. The `request` parameter is an object that provides information about the received request (but we don't use it yet). The `response` parameter is an object that provides functions which allow us to send an HTTP response to the retrieved request.

Here, we use it to set the HTTP response code to *200 OK*, set a *Content-Type* header, set the body of our response (again, a simple *"Hello, World"*), and to signal that our response is complete and shall be sent over the wire to the requesting client, via `response.end()`.

Note that `http.createServer()` alone isn't enough to build a fully working web server. `createServer()` returns an object on which we need to call the `listen()` function with a port number as the parameter, in order to bind our application to that port. This makes the operating system and in turn Node.js forward packets arriving at that port to our application.

After starting the web server application via `node server.js`, we can send HTTP requests to it - either by opening `http://127.0.0.1:8000/` in a browser, or by using the command line tool *curl*, like so:

[10] https://github.com/nodejs/node/blob/0f841208d2d89d91395536a3227c4b11e1bf2425/lib/http.js

```
~$ > curl -v http://127.0.0.1:8000/
*   Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to 127.0.0.1 (127.0.0.1) port 8000 (#0)
> GET / HTTP/1.1
> Host: 127.0.0.1:8000
> User-Agent: curl/7.54.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: text/plain
< Date: Tue, 09 Oct 2018 15:33:52 GMT
< Connection: keep-alive
< Transfer-Encoding: chunked
<
* Connection #0 to host 127.0.0.1 left intact
Hello, World
```

As expected, we get the string *Hello, World* as the response body, and the *Content-Type* header has been set to `text/plain`. Other headers are automatically set be the *http* module.

Let's proof that the anonymous function we passed as the parameter to `createServer` is indeed called anew every time we send an HTTP request; and while we are at it, let's include some information about the request:

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

http.createServer((request, response) => {
    console.log(`Received request for ${request.url}`);
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("Hello, World");
    response.end();
}).listen(8000);
```

Stop the running application (by hitting `CTRL + c`), and start it again after making the change.

Sending requests to the server will now result in output on the console where the application has been started:

```
$> curl http://127.0.0.1:8000/
$> curl http://127.0.0.1:8000/foo
$> curl http://127.0.0.1:8000/bar?a=b
```

results in

```
Received request for /
Received request for /foo
Received request for /bar?a=b
```

Event-driven asynchronous callbacks

Let's have a closer look at what is really going on here. We pass an (anonymous) function as the parameter to another function. We already discussed this in our first "Hello, World" experiments. But there's a difference - in our experiments, we passed a function that was then immediately called. Here, at first nothing happens with the function we pass. An external event - in this case, an HTTP request - is required to make the `http` module code call our passed function.

This is a very common pattern in Node.js (and JavaScript in general). It is called the *callback pattern*, because it's like giving someone your phone number and asking them to call you back whenever they have relevant information for you.

The pattern makes obvious that a lot is going on behind the scenes to make our simple web server work. Node.js is executing the code we feed it, but it also handles events that happen outside our code.

To understand why Node.js applications have to be written this way, we need to understand how Node.js executes our code. Node's approach isn't unique, but the underlying execution model is different from runtime environments like Python, Ruby, PHP or Java.

Let's take a very simple piece of code like this:

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

Please ignore for now that we haven't actually talked about connecting to databases before - it's just an example. The first line queries a database for lots of rows, the second line puts "Hello, World" to the console.

Let's assume that the database query is really slow, that it has to read an awful lot of rows, which takes several seconds.

The way we have written this code, the JavaScript interpreter of Node.js first has to read the complete result set from the database, and then it can execute the console.log() function.

If this piece of code actually was, say, PHP, it would work the same way: read all the results at once, then execute the next line of code. If this code would be part of an application that serves a web page, then the user would have to wait several seconds for that web page to load.

However, in the execution model of PHP, this would not become a "global" problem: the web server starts its own PHP process for every HTTP request it receives. If one of these requests results in the execution of a slow piece of code, it results in a slow page load for this particular user, but other users requesting other pages would not be affected.

The execution model of Node.js is different - 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.

To avoid this kind of *blocking* behaviour, JavaScript, and therefore Node.js, introduces the concept of event-driven, asynchronous callbacks, by utilizing an *event loop*.

We can understand this concept by analyzing a rewritten version of our problematic code:

```
1 let result;
2 database.query("SELECT * FROM hugetable", (rows) => result = rows);
3 console.log("Hello, World");
```

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

In its previous form, our code was *synchronous*, and therefore blocking: first do the database query, and only when this is done, then write to the console.

Now, Node.js can handle the database request asynchronously, and therefore non-blocking. Provided that `database.query()` is part of an asynchronous library, this is what Node.js does: just as before, it takes the query and sends it to the database. 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()."

This way, Node.js can immediately execute `console.log()`, and afterwards, it enters the event loop. Node.js continuously cycles through this loop again and again whenever there is nothing else to do, waiting for external events. Events like, e.g., a slow database query finally delivering its results.

This also explains why our HTTP server needs a function it can call upon each 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. If a second user requests the server while it is still serving the first request, that second request could only be answered after the first one is done - as soon as you have more than a handful of HTTP requests per second, this wouldn't work at all.

It's important to note that this asynchronous, non-blocking, single-threaded, event-driven execution model isn't an infinitely scalable performance unicorn with silver bullets attached. It is just one of several execution models, and it has its limitations. One being that as of now, a running Node.js application is just one single operating system process and it can run on only one single CPU core.

However, this execution model is quite approachable, because it allows us to write applications that deal with concurrency in an efficient and relatively straightforward manner.

Let's play around a bit with this new concept. Can we prove that our code continues after creating the server, even if no HTTP request happened and the callback function we passed isn't called? Let's give it a try:

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

http.createServer((request, response) => {
    console.log(`Received request for ${request.url}`);
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("Hello, World");
    response.end();
}).listen(8000);

console.log("Server has started.");
```

All we do here is adding another console log message at the end of our script. If it's true that `createServer().listen()` only registers the callback function for later use and immediately continues with the execution of our code, then we should immediately see the new message on the console, even if no HTTP request is handled:

```
$> node server.js
Server has started.
```

Sure enough, this is exactly what happens - event-driven asynchronous non-blocking server-side JavaScript in action.

Organizing the application

Ok, I promised we will get back to how to organize our application. We have the code for a very basic HTTP server in file *backend/server.js*. For now, this file *is* our application, but because the application will grow and we want to keep things tidy and organized, it's time to turn it into a module.

To do so, we need to make the server code export a function that allows us to start our HTTP server from another file:

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

module.exports.start = () => {
    http.createServer((request, response) => {
        console.log(`Received request for ${request.url}`);
        response.writeHead(200, { "Content-Type": "text/plain" });
        response.write("Hello, World");
        response.end();
    }).listen(8000);

    console.log("Server has started.");
};
```

With this, we can create our main backend application file, *backend/index.js*, which requires the exported function and uses it to start the web server:

```
const startHttpServer = require("./server").start;

startHttpServer();
```

Great! We can now put the different parts of our server application into different files and wire them together using the module system.

We still have only the very first part of our application in place: we can receive HTTP requests. But we need to do something with them - 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 to `createServer`. But we want a well-structured codebase, so we need to split our application logic into multiple modules.

Making different HTTP requests point at different parts of our code is called "routing" - well, then, let's create a module called *router* in the next chapter.

What's needed to "route" requests?

Routing requests to a dedicated piece of code is required because we want to handle a *GET* request for URL */todos/* different than a *POST* request to URL */todos/*. While the former should return a list of all todo item, the latter should create a new todo item.

What we need is a mapping that can be illustrated like this:

```
POST /api/todos/          -> Code that creates a new todo item
GET /api/todos/           -> Code that responds with a list of todos items
DELETE /api/todos/?id=:id -> Code that deletes the todo item with the given id
```

Keeping the code for these scenarios separated will result in a clean and readable code base.

Therefore, we need to be able to feed the HTTP method used (GET, POST, etc.) and the requested URL into our router, and based on these the router then needs to be able to decide which code to execute.

> This "code to execute" is the third part of our application: a collection of so-called request handlers that do the actual work when a request is received. We will create these in the next chapter.

We therefore need to look into the HTTP requests our server receives, and extract the method used and the requested URL.

> It could be argued whether this kind of extraction should be part of the router module or part of the server module (or even a module of its own), but let's just agree on making it part of our new *router* module. This way, the HTTP server which receives a request can pass the request object to the router component, which then encapsulates the complete routing logic based on that request.

Extracting the method from a request handled by the *http* module is straightforward, as we will see: its available on the `request.method` attribute. And getting the requested URL is simple, too: `request.url` provides it.

However, a URL consists of several different components, and for our routing concerns, we are only interested in some of them. We therefore need to also *parse* the URL to make sense of it.

When parsing a typical HTTP URL like `http://www.example.com:8000/foo?bar=1&baz=yes#main`, a parser needs to identify several different parts - e.g., the *host* part (www.example.com) or the *search* part (*?bar=1&baz=yes* - this is also called the *query string*).

While this can be achieved by working with the URL as a simple string, and by splitting this string into its different logical parts manually with substring operations or regular expressions, this is very cumbersome. A dedicated module that is integrated into Node.js makes this very easy.

This module, called *url*, implements the standardized WHATWG URL API, making the url-parsing code of Node.js work identical to the way that web browsers are parsing URLs.

To use it, we need to `require` the *url* module, and use the `URL` class it provides. We do so in a new file called *router.js*, again in folder *backend*:

```
1 const URL = require("url").URL;
2
3 const parsedUrl = new URL("http://www.example.com:8000/foo?bar=1&baz=yes#main");
4
5 console.log(parsedUrl);
```

Note how we not yet create it as a module, but merely play around with *url* for now.

What sticks out is the introduction of another JavaScript language construct - `URL` is a class, and we use the keyword `new` to create `parsedUrl` as an object instance of this class. This works mostly as expected, that is, it works like object orientation in classical OOP languages.

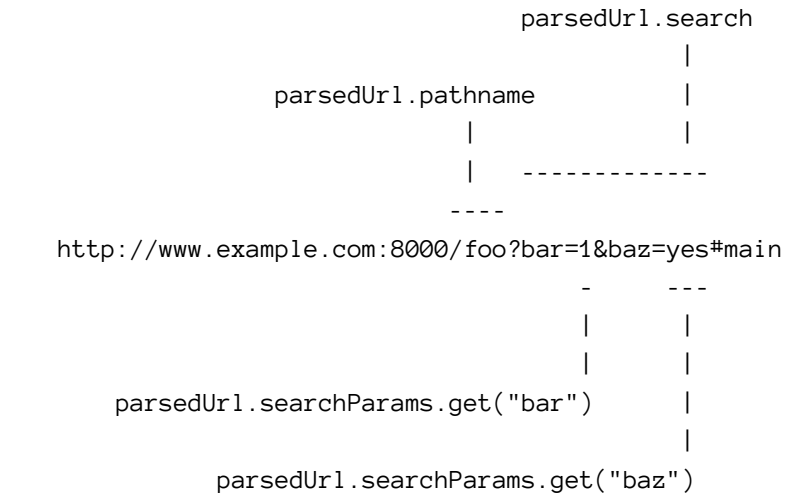
There are some subtleties behind the scenes, and ES6 introduced some syntactical sugar with regards to class creation, but we will discuss those topics later.

For now, it's sufficient to understand that a `URL` instance like `parsedUrl` is an object with attributes that allow us structured access to the different parts of an URL, as can be seen by the output of script *router.js*:

```
$> node router.js

URL {
  href: 'http://www.example.com:8000/foo?bar=1&baz=yes#main',
  origin: 'http://www.example.com:8000',
  protocol: 'http:',
  username: '',
  password: '',
  host: 'www.example.com:8000',
  hostname: 'www.example.com',
  port: '8000',
  pathname: '/foo',
  search: '?bar=1&baz=yes',
  searchParams: URLSearchParams { 'bar' => '1', 'baz' => 'yes' },
  hash: '#main' }
```

Note how this object provides another object on attribute `searchParams`, which allows to also access the different values of the *search* string by their keys. Here is an illustration on how to retrieve some of the parts of the URL:



Armed with this tool, we can now start to lay the foundation of HTTP request-to-code routing in our backend application.

It makes sense that our *router* offers two functions: one to register a piece of code to be executed when a certain type of request is received, and another one to actually route an incoming request, using the previously registered route-to-code mappings.

To do so, we implement *backend/router.js* as follows:

```
const URL = require("url").URL;

const getHandlerId = (method, pathname) => `${method} ${pathname}`;

const registeredHandlers = {};

module.exports = {
    register: (method, pathname, requestHandler) => {
        registeredHandlers[getHandlerId(method, pathname)] = requestHandler;
    },

    route: (request, response) => {
        const pathname = new URL(`http://localhost${request.url}`).pathname;
        const handlerId = getHandlerId(request.method, pathname);

        console.log(`About to route request for ${request.method} ${pathname}`);

        if (typeof(registeredHandlers[handlerId]) === 'function') {
            registeredHandlers[handlerId](request, response);
            return true;
        } else {
            console.log(`No request handler found for ${request.method} ${pathname}`\
);
            return false;
        }
    }
};
```

There's quite a lot going on in this file, so let's dissect it.

We still *require* class `URL` as before. Everything else is new.

We declare a function *getHandlerId* - this is just a local helper that returns a unique string for any given method and pathname combination. This is required for the actual request-to-code mapping. A request with method `GET` for path `/api/todos/` results in handler id `GET /api/todos/`. Using this id as a key on object `registeredHandlers` allows to map these requests to a function.

The `registeredHandlers` object, declared on the next line, thus acts as our routing table - it carries all the request-to-code mappings that we register. Note that although we change the attributes of this object later in the code, we can still declare it as a `const`, as described in chapter *"Hello, World.*

Next, we declare the exports of the router module. The outer world needs two functions from us to enable request routing: It needs to be able to register new request-to-code mappings, and it needs to be able to give us an incoming request which we then route.

We call these functions `register` and `route`, respectively.

`register` is very simple: It expects three parameters, a method and a pathname (to determine what request to route), and a so-called request handler - the code to trigger if a route matches. This allows to register routes like so:

```
1 register("GET", "/api/todos/", someFunction);
```

With this, *GET* requests to path */todos/* are registered to be handled by function *someFunction*.

The actual routing logic, defined in exported function `route`, is a bit more involved.

It will be called by the HTTP server whenever a new incoming request is received. The HTTP server will pass the `request` object for the request, and we need to inspect it to make a routing decision.

We first determine the path that has been requested, using the `URL` class as previously discussed.

The `request.url` value we use for this is always a relative URL (`/api/todos/`, and not `http://example.com/api/todo` but `URL` only works with absolute URLs. We therefore add `http://localhost` to the URL to satisfy this requirement.

Next, we calculate the handlerId for the request, again using the method and the pathname.

We now need to find out if `registeredHandlers` contains a matching handler. To do so, we inspect the type of the value stored at `registeredHandlers[handlerId]`. We check if the value is of type `function`, which implicitly verifies that there is a value at all - if no handler has been registered for this handler id, then the value is `undefined`, and `undefined` is not of type `function`.

If a matching handler is found, it is used to finally route the request to its target code. This is done by invoking the handler, which is simply a function after all. We also pass the `request` and `response` objects as parameters, because handler need `response` to send content back to the client and `end()` the response, and some handlers might

While we *call* the handler if we find a matching one, we also *return* - in this case, `true`. If we cannot find a matching handler, we log an error message, and return `false`. The code calling the `route` function can use the return value to correctly handle requests that don't have a handler - we will get to this in a moment.

Note how we use `===` (three equals signs, not two) to compare the output of `typeof` to the string `function`. This is the *strict equality operator*, and I recommend to always use this one instead of the normal equality operator `==` unless you know for sure that you don't want strict comparison.

The following illustrates why using the strict operator avoids a lot of ambiguity:

```
'' == 0       // true
'' === 0      // false

'1' == true   // true
'1' === true  // false

[] == 0       // true
[] === 0      // false


[null, null, null] == ",,"     // true
[null, null, null] === ",,"    // false
```

That's not to say that the strict operator protects you from each and every weirdness of the JavaScript type and comparison system - and there's a lot of weirdness: https://charlieharvey.org.uk/page/javascript_-the_weird_parts

Still, it should be preferred because it helps to prevents at least the worst types of errors.

Further readings

- https://github.com/nodejs/node/issues/12682
- Node.js: exports vs module.exports[11]
- Object initializer: New notations in ECMAScript 2015[12]
- Felix Geisendörfer: Understanding node.js[13]
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators#Identity
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Equality_comparisons_and_sameness

[11] https://www.hacksparrow.com/node-js-exports-vs-module-exports.html
[12] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Object_initializer#New_notations_in_ECMAScript_2015
[13] http://debuggable.com/posts/understanding-node-js:4bd98440-45e4-4a9a-8ef7-0f7ecbdd56cb