LFW212 [NODE.JS SERVICES DEVELOPMENT](https://trainingportal.linuxfoundation.org/courses/nodejs-services-development-lfw212)

Course Introduction

**Welcome to Node.js Services Development (LFW212)!**

This course provides a deep dive into Node core HTTP clients and servers, web servers, RESTful services and web security essentials. With laboratory exercises and a pragmatic approach based on real world best practices this self-lead workshop will instill competency for day-to-day Enterprise Node.js development. With a major focus on Node.js services and security this content is an essential counterpart to the [Node.js Application Development (LFW211)](https://training.linuxfoundation.org/training/nodejs-application-development-lfw211/) course.

By taking this course, you will be fully prepared for the [OpenJS Node.js Services Developer Certification](https://training.linuxfoundation.org/certification/jsnsd/) and acquire pragmatic knowledge and core skills that accelerate both Node.js productivity and career growth.

# Course Learning Objectives

By the end of this course, you should be able to:

* Understand foundational essentials for creating web servers.
* Explain how HTTP works at a Node core API level.
* Understand and leverage ecosystem frameworks for rapid composability.
* Cover essential RESTful practices and gain practical working knowledge of implementing RESTful services.
* Develop the skill of server and service composition.

# Meet Your Instructor: David Mark Clements

David Mark Clements is a Principal Architect, technical author, public speaker and OSS creator specializing in Node.js and browser JavaScript.

David has been writing JavaScript since 1996 and has been working with, speaking and writing about Node.js since Node 0.4 (2011). He’s the author of various Open Source projects. Of note among them is Pino, the fastest Node.js JSON logger available and 0x, a powerful profiling tool for Node.js.

David is the technical lead and primary author of the official [OpenJS Node.js Application Developer Certification](https://training.linuxfoundation.org/certification/jsnad/) (JSNAD) and [OpenJS Node.js Services Developer Certification](https://training.linuxfoundation.org/certification/jsnsd/) (JSNSD).

David’s book about Node.js, called "Node Cookbook: Actionable Solutions for the Full Spectrum of Node.js 8 Development", is now in its third edition.

# Course Audience and Requirements

Audience

This course is designed for developers on their way to senior level who wish to master and demonstrate their Node.js knowledge and skills, in particular how to use Node with frameworks to rapidly and securely compose servers and services.

Knowledge Requirements

You should know how to use a command line terminal, and be familiar with JavaScript as well as knowledge domains of the OpenJS Node.js Application Developer (JSNAD) program.

System Requirements

MacOS, Windows or Linux platforms are supported.

# Course Resources

Resources for this course can be found online. Making updates to this course takes time. Therefore, if there are any changes in between updates, you can always access course updates, as well as the course resources online:

* Go to the Linux Foundation training website to obtain [Course Resources](https://training.linuxfoundation.org/cm/LFW212)
* The user ID is **LFtraining** and the password is **Penguin2014**.

# Course Support

One great way to interact with peers taking this course is via the [Class Forum](https://forum.linuxfoundation.org/categories/lfw212-class-forum). The forum can be used in the following ways:

* To introduce yourself to other peers taking this course.
* To discuss concepts, tools and technologies presented in this course, or related to the topics discussed in the course materials.
* To ask questions or report issues with labs or course content.
* To share resources and ideas related to Node.js.

The Class Forum will be reviewed periodically by the Linux Foundation staff, but it is primarily a community resource, not an 'ask the instructor' service.

If you have questions regarding to your course enrollment, you can reach out to us via our [Customer Support system](http://trainingsupport.linuxfoundation.org/). You will be required to login with your LF Account, which will help us to quickly locate your account and respond to your request. This will also allow you to track your support request through to resolution, and create an ongoing record of your support requests.

The Linux Foundation Training & Certification Customer Support system also offers enhanced functionality, such as:

* Knowledge Base Articles - to help you find a quick response to your commonly asked questions
* Service Request Forms - asking the right questions so that you can get the right answers.

# Course Timing

This course is entirely self-paced; there is no fixed schedule for going through the material. You can go through the course at your own pace, and you will always be returned to exactly where you left off when you come back to start a new session. However, we still suggest you avoid long breaks in between periods of work, as learning will be faster and content retention improved.

**You have unlimited access to this course for 12 months from the date you registered, even after you have completed the course.**

The chapters in the course have been designed to build on one another. It is probably best to work through them in sequence; if you skip or only skim some chapters quickly, you may find there are topics being discussed you have not been exposed to yet. But this is all self-paced, and you can always go back, so you can thread your own path through the material.

# The OpenJS Foundation

[The OpenJS Foundation](https://openjsf.org/) was formed in March 2019 after the merger of the Node.js Foundation and JS Foundation. It is the home of a wide range of JavaScript projects including Appium, Dojo, jQuery, Node.js, webpack, and many more.

Its goal is to support the healthy growth of the JavaScript and web ecosystem, and to share its learnings around technical governance, project hosting, and running diverse and welcoming communities over a broader section of the ecosystem.



### **SETTING UP**Introduction

# Chapter Overview

Node works on many platforms, and there are many ways to install Node on each platform. This short precursory chapter covers best-practice setup of Node.js on macOS, Linux and Windows machines.

# Learning Objectives

By the end of this chapter, you should be able to:

* Understand the best way to set up and use Node on various platforms.
* Understand what executables are installed.
* Manage multiple Node versions.

# How Not to Install Node

Often Node.js can be installed with a particular Operating System's official or unofficial package manager. For instance apt-get on Debian/Ubuntu, Brew on macOs, Chocolatey on Windows. It is strongly recommended against using this approach to install Node. Package managers tend to lag behind the faster Node.js release cycle. Additionally the placement of binary and config files and folders isn't standardized across OS package managers and can cause compatibility issues.

Another significant issue with installing Node.js via an OS package manager is that installing global modules with Node's module installer (**npm**) tends to require the use of **sudo** (a command which grants root privileges) on non-Windows systems. This is not an ideal setup for a developer machine and granting root privileges to the install process of third-party libraries is not a good security practice.

Node can also be installed directly from the Node.js website. Again on macOS and Linux it predicates the use of **sudo** for installing global libraries. Whether Windows, macOS or Linux, in the following sections we'll present a better way to install Node using a version manager.

It's strongly recommended that if Node is installed via an Operating System Package Manager or directly via the website, that it be completely uninstalled before proceeding to the following sections.

# Installing Node.js on macOS and Linux

In this section, we'll look at installing Node on macOS and Linux. For Windows users feel free to skip to the next section, unless using Windows Subsystem for Linux v2 in which case this section may also be relevant.

The recommended way to install Node.js on macOS and Linux is by using a Node version manager, in particular [**nvm**](https://github.com/nvm-sh/nvm). See <https://github.com/nvm-sh/nvm> for full details.

We're going to install **nvm** and then use it to install Node.

The current **nvm** version is v0.39.1 (as of January 2022), so the install process will contain this version in the URL, if a greater version is out at time of reading, replace v0.39.1 with the current **nvm** version. For this installation process we assume that Bash, Sh, or Zsh is the shell being used, Fish is not supported but see the **nvm** readme for alternatives.

The way to install **nvm** is via the install script at <https://github.com/nvm-sh/nvm/blob/v0.39.1/install.sh>. If **curl** is installed (it usually is) a single command can be used to install and setup **nvm**:

**curl -o- ht‌tps://raw.githubusercontent.com/nvm-sh/nvm/v0.39.1/install.sh | bash**

If using **zsh** (e.g. on newer macOS releases) the **bash** part of the command can be replaced with **zsh**.

Alternatively the file can be downloaded and saved, and then easily executed like so:

**cat install.sh | bash**

Again **bash** can be replaced with **zsh**. To check that the installation was successful execute the following in the terminal:

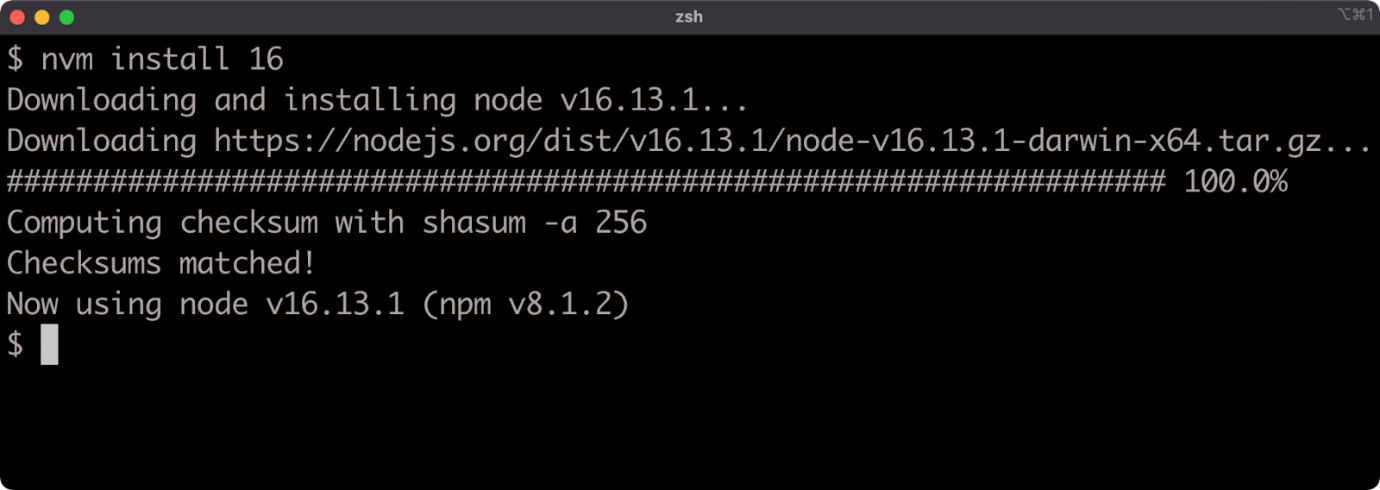
**command -v nvm**

It should output **nvm**. If this fails on Linux, close and reopen the terminal (or SSH session) and try running the command again. On macOS see <https://github.com/nvm-sh/nvm#troubleshooting-on-macos> for in depth troubleshooting.

Now that we have a version manager, let's install the Node version we'll be using on this course:

**nvm install 16**

This will install the latest version of Node 16:



In this case, the command installed Node v16.13.1. It doesn't matter if the right-most numbers are higher for this course, as long as the major number (the first number) is 16.

We can verify that Node is installed, and which version, with the following command:

**node -v**

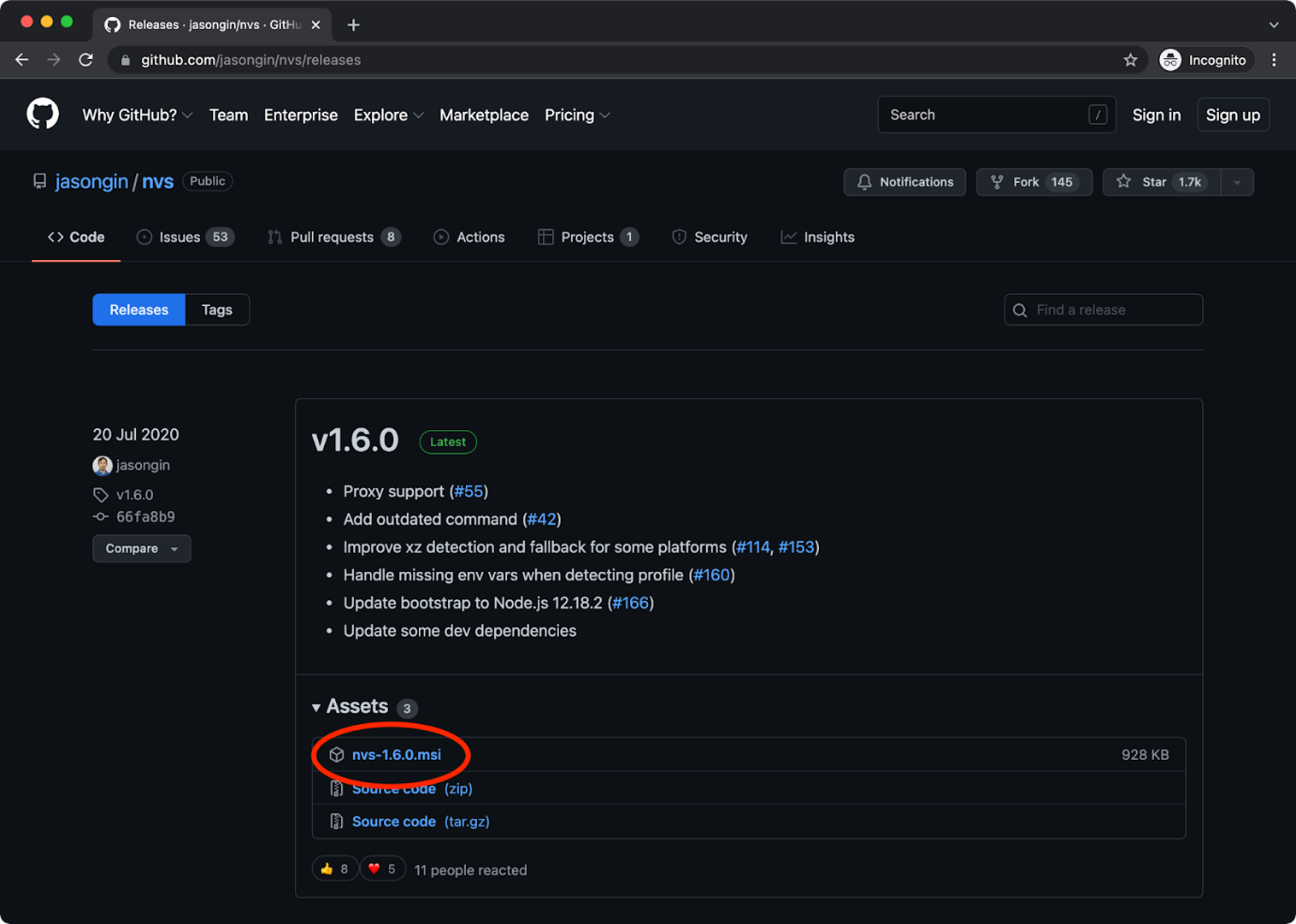
Congratulations, you now have the right setup on your macOS or Linux machine to proceed with the course.

# Installing Node.js on Windows

In this section, we'll look at installing Node.js on Windows 10 and up. As a non-Windows user feel free to skip this section.

While **nvm** is recommended for macOS and Linux, and there is an unaffiliated **nvm-windows** version manager the recommended version manager for Windows is [**nvs**](https://github.com/jasongin/nvs). The **nvs** version manager is actually cross-platform so can be used on macOS and Linux but **nvm** is a lot more popular.

To install **nvs** on Windows go to the release page here, <https://github.com/jasongin/nvs/releases> and download the MSI installer file of the latest release:



If a later release than v1.6.0 is available, download the MSI for that release. Once downloaded, run the installer and follow the steps to install. After it's installed open a **cmd.exe** or powershell prompt and run the following:

**nvs add 16**

This should result in the latest version of Node 16 being installed:

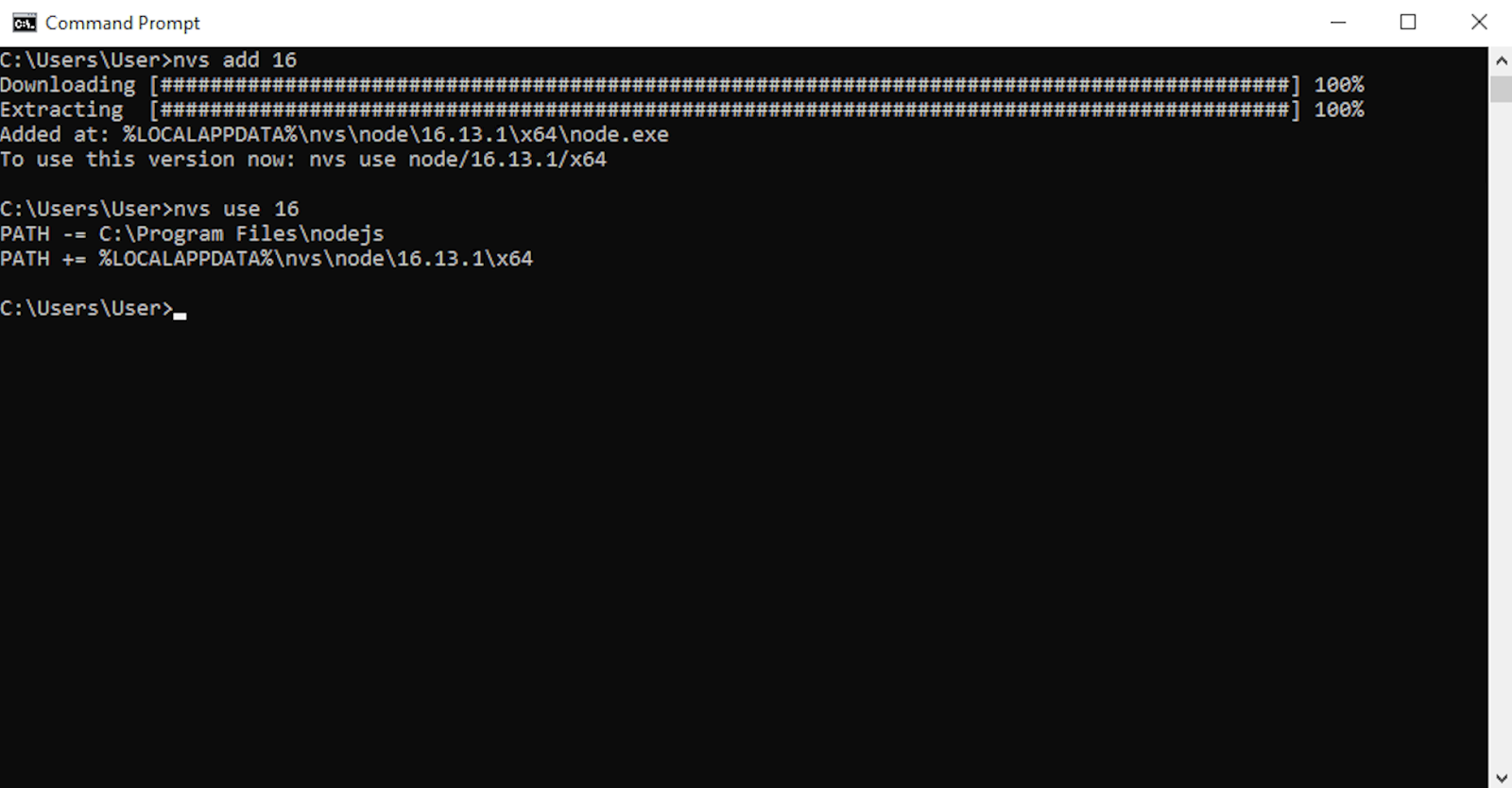


In this case the command installed Node v16.13.1, it doesn't matter if the right-most numbers are higher for this course, as long as the major number (the first number) is 16.

To activate the newly installed version we also need to run the following command:

**nvs use 16**

This should result in output similar to the following:



We can verify that Node is installed, and which version, with the following command:

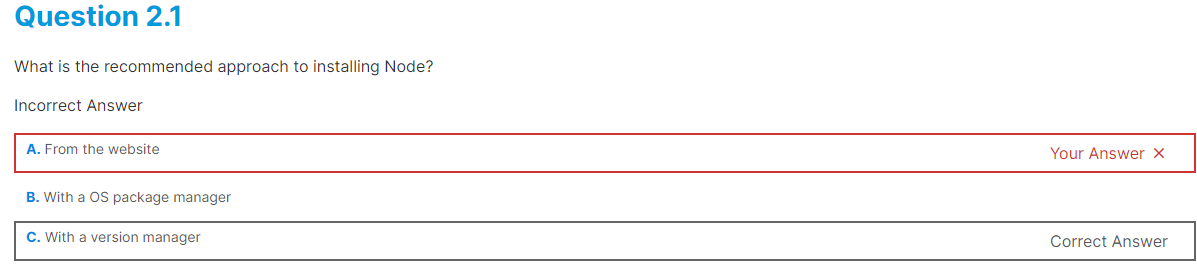
**node -v**

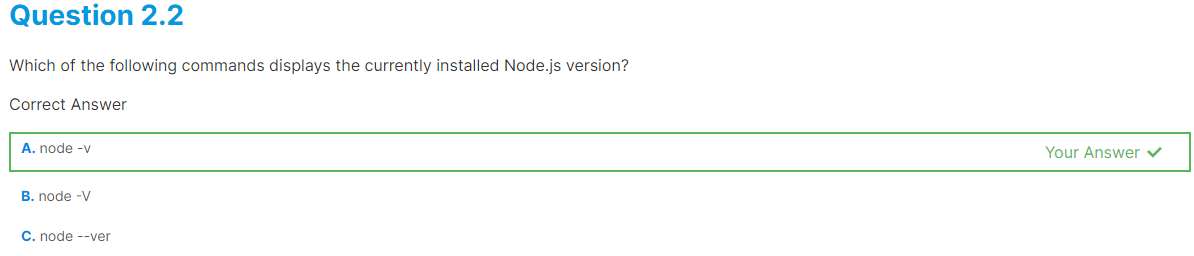
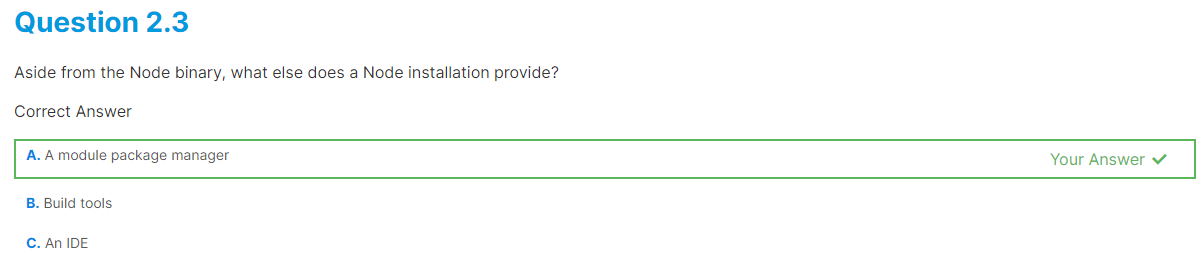
Congratulations, you now have the right setup on your Windows machine to proceed with the course.

# Lab 2.1 - Node Check

# Lab 2.2 - NPM Check

### Knowledge Check



### **CREATING A WEB SERVER**Introduction

# Chapter Overview

The focus of this course and the JSNSD examination centers on RESTful/HTTP services as key knowledge for almost every contemporary scenario involving Node.js and services. There is a great deal of overlap between an HTTP service and a web server. In order to make an HTTP-based service, the first step is to create an HTTP server. In this chapter we'll explore different approaches to creating a web server and perform a tour de force of selected web frameworks while we're at it. This chapter and indeed the entire course assumes prior knowledge of Node.js, including the module system and Node.js streams and Node CLI flags. See the [Node.js Application Development (LFW211)](https://training.linuxfoundation.org/training/nodejs-application-development-lfw211/) course for instruction in these fundamentals.

# Learning Objectives

By the end of this chapter, you should be able to:

* Create a minimal web server with Node core APIs.
* Create a minimal web server with the Express framework.
* Create a minimal web server with the Fastify framework.

# Creating a Web Server with Node Core (1)

Generally speaking, attempting to create a web server or service with just the Node core **http** (or **https**) module is not recommended. However, for learning purposes we'll put together a basic web server using the core **http** module in order to better understand the value that a web framework can bring and how a web framework is actually operating under the hood.

Let's define what we expect from a minimum viable web server:

* Responds to HTTP requests based on a given HTTP verb (for instance GET).
* Responds to requests based on a given route.
* Responds with 404 HTTP Status code if a route isn't found.
* Sets appropriate headers, such as Content-Type.

To meet this criteria we're going to take an iterative approach and build in layers.

**Before getting started, be aware that we'll be stopping and starting servers a lot throughout this chapter and the entire course. Make sure that any previous processes from prior sections have been terminated, especially when attempting the exercises.**

To get started, we can create a folder called **http-web-server** with the following commands:

**node -e "fs.mkdirSync('http-web-server')"  
cd http-web-server**

Note, throughout this course **node** with the **-e** (evaluate) flag will be used for cross-platform/cross-shell administrative commands (like creating folders).

Now, let's create a file called **server.js** with the following initial code:

**'use strict'  
const http = require('http')  
const PORT = process.env.PORT || 3000**

**const hello = `<html>  
  <head>  
    <style>  
     body { background: #333; margin: 1.25rem }  
     h1 { color: #EEE; font-family: sans-serif }  
    </style>  
  </head>  
  <body>  
    <h1>Hello World</h1>  
  </body>  
</html>`**

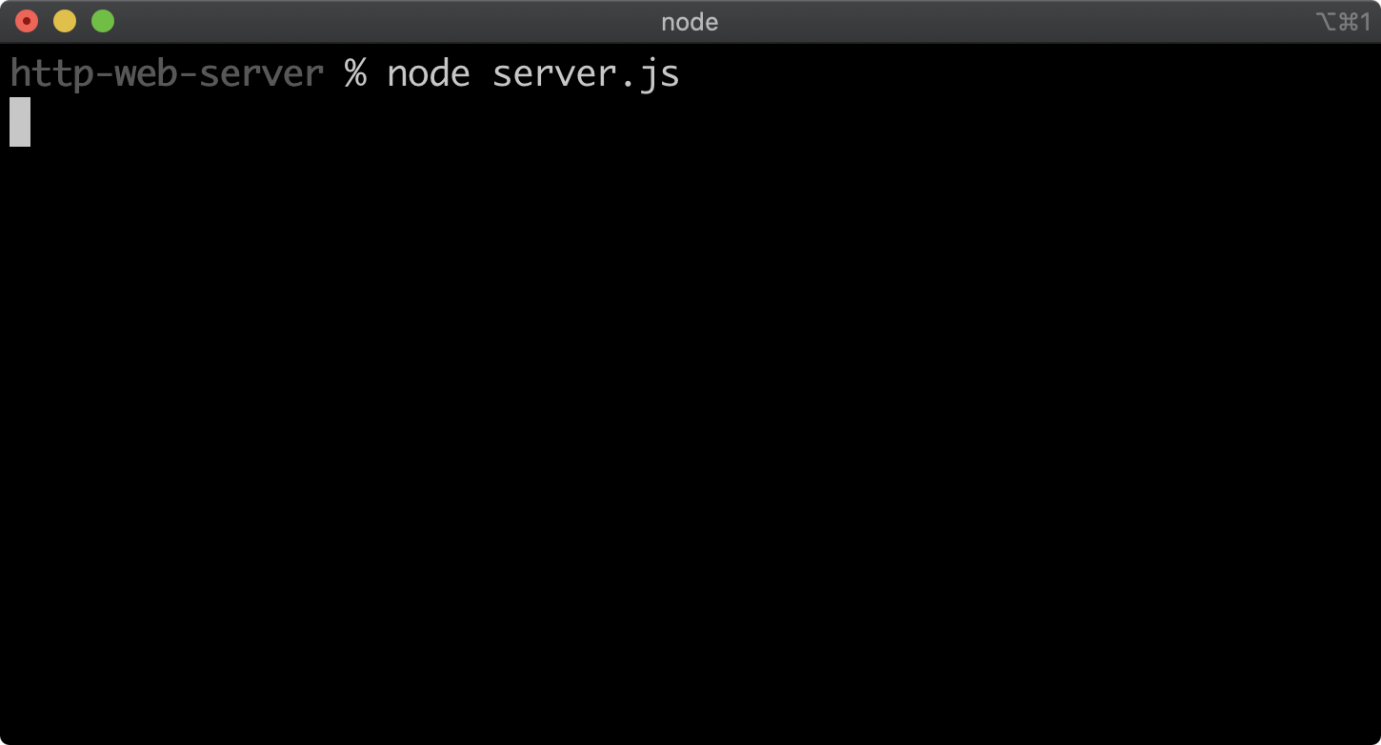
**const server = http.createServer((req, res) => {  
  res.setHeader('Content-Type', 'text/html')  
  res.end(hello)  
})**

**server.listen(PORT)**

This code can be executed with the following command:

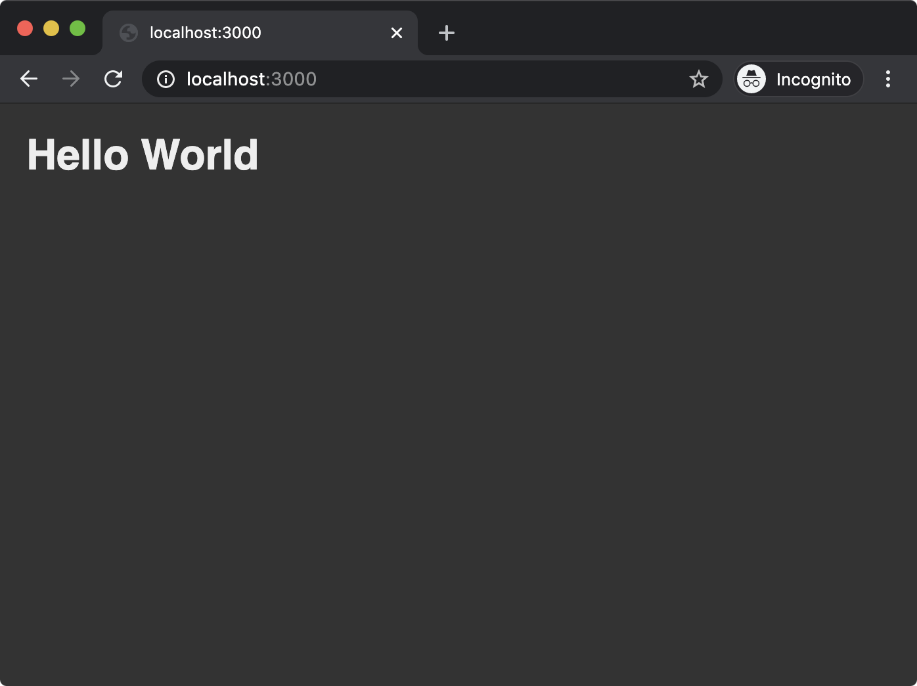
**node server.js**

If we run this code the process will not exit by itself:



This is because the created server is keeping the process open.

If we then navigate in a browser to ht‌‌tp://localhost:3000, we should see something like the following:



So far we've used the Node core **http** module to create a server with the **createServer** method. We've also observed that a function is passed to **createServer**. This function is called every time the HTTP server receives a request. The function passed to **createServer** is passed two objects: the request object and the response object. These objects are created for every request and then passed to the function we supply to **createServer**.

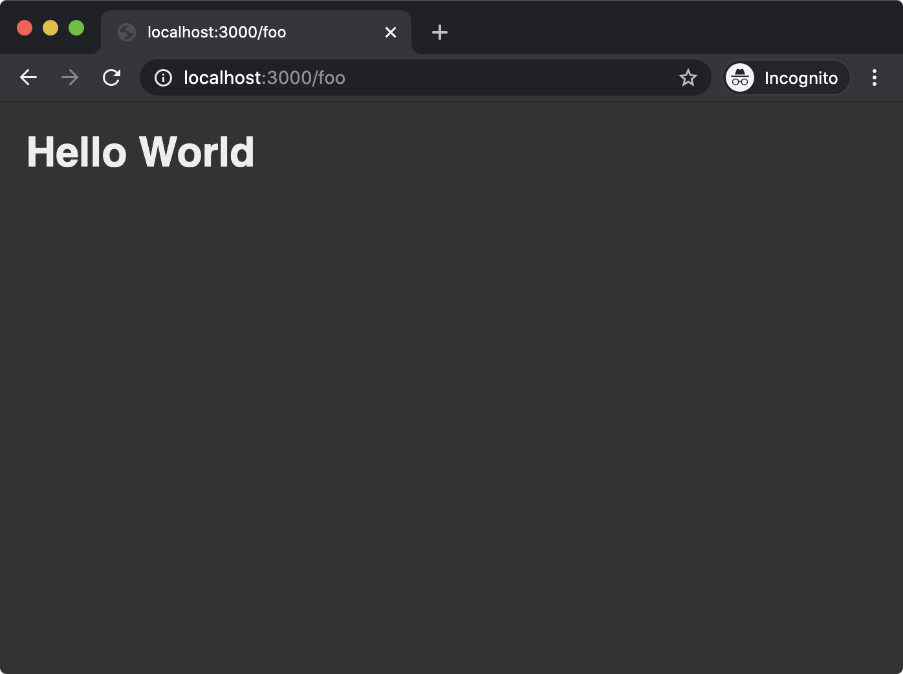
In the function we passed to **createServer** we use the **setHeader** and the **end** methods of the response object (**res**) to set the Content-Type header to text/html, and to send the string of HTML (assigned to the **hello** constant) while also closing the connection. The **res** object inherits from **http.ServerResponse** which in turn inherits from **http.OutgoingMessage** (a Node core internal constructor) which then inherits from **stream.Stream**. For all practical purposes the **res** object is a writable stream, which is why calling **end** writes our content and also closes the connection.

The **createServer** method also returns an object which represents the server. We use the **listen** method to bind to a port. In our case, by default, we bind to port 3000. Our server can instead bind to a different port by setting the **PORT** environment variable.

Cont'd on the next page.

# Creating a Web Server with Node Core (2)

Our implementation does not yet meet our criteria. We can navigate to any route and the response will be the same. For instance, ht‌tp://localhost:3000/foo:



Regardless of whether we use a POST, GET or any other HTTP verb, we will always get the same response.

Let's update our **server.js** code to the following:

**'use strict'  
const url = require('url')  
const http = require('http')  
const PORT = process.env.PORT || 3000  
const { STATUS\_CODES } = http**

**const hello = `<html>  
  <head>  
    <style>  
     body { background: #333; margin: 1.25rem }  
     h1 { color: #EEE; font-family: sans-serif }  
   </style>  
  </head>  
  <body>  
    <h1>Hello World</h1>  
  </body>  
</html>`**

**const root = `<html>  
<head>  
  <style>  
   body { background: #333; margin: 1.25rem }  
   a { color: yellow; font-size: 2rem; font-family: sans-serif }  
  </style>  
</head>  
<body>  
  <a href='/hello'>Hello</a>  
</body>  
</html>  
`**

**const server = http.createServer((req, res) => {  
  res.setHeader('Content-Type', 'text/html')  
  if (req.method !== 'GET') {  
    res.statusCode = 405  
    res.end(STATUS\_CODES[res.statusCode] + '\r\n')  
    return  
  }  
  const { pathname } = url.parse(req.url)  
  if (pathname === '/') {  
    res.end(root)  
    return  
  }  
  if (pathname === '/hello') {  
    res.end(hello)  
    return  
  }  
  res.statusCode = 404  
  res.end(STATUS\_CODES[res.statusCode] + '\r\n')  
})**

**server.listen(PORT)**

We can stop the currently running **server.js** process with Ctrl+C and start our modified **server.js** file with the following command:

**node server.js**

Cont'd on the next page.

# Creating a Web Server with Node Core (3)

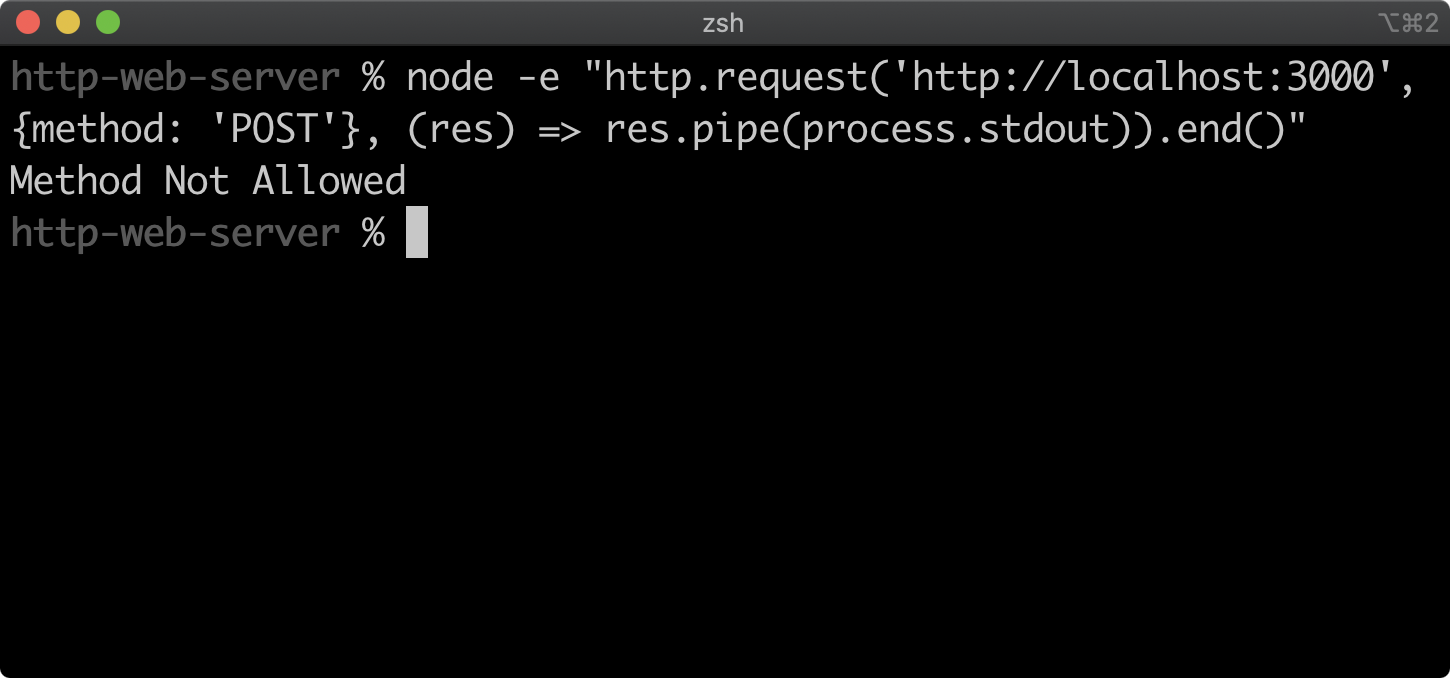
At the top of our **server.js** file we've added the additional Node core **url** module, and we've destructured the **STATUS\_CODES** object, which contains key-values of status codes to HTTP status messages, from the **http** module. Just before creating our server we've also added a **root** constant which contains an HTML string with an anchor tag linking to the **/hello** route.

We've updated the function passed to **createServer** with some new logic. We check the incoming requests HTTP verb by accessing the **req.method** property. If this is not set to GET we set the **statusCode** of the **res** object to **405** (Method Not Allowed) and end the response with an appropriate status message.

We can check whether this works by running the following command in another terminal window:

**node -e "http.request('http://localhost:3000', {method: 'POST'}, (res) => res.pipe(process.stdout)).end()"**

This command uses the **http** module to submit a POST request to our server and then prints the result to the terminal:



res.pipe(process.stdout)).end()" Terminal window with "Method Not Allowed" printed as the result of the command.">

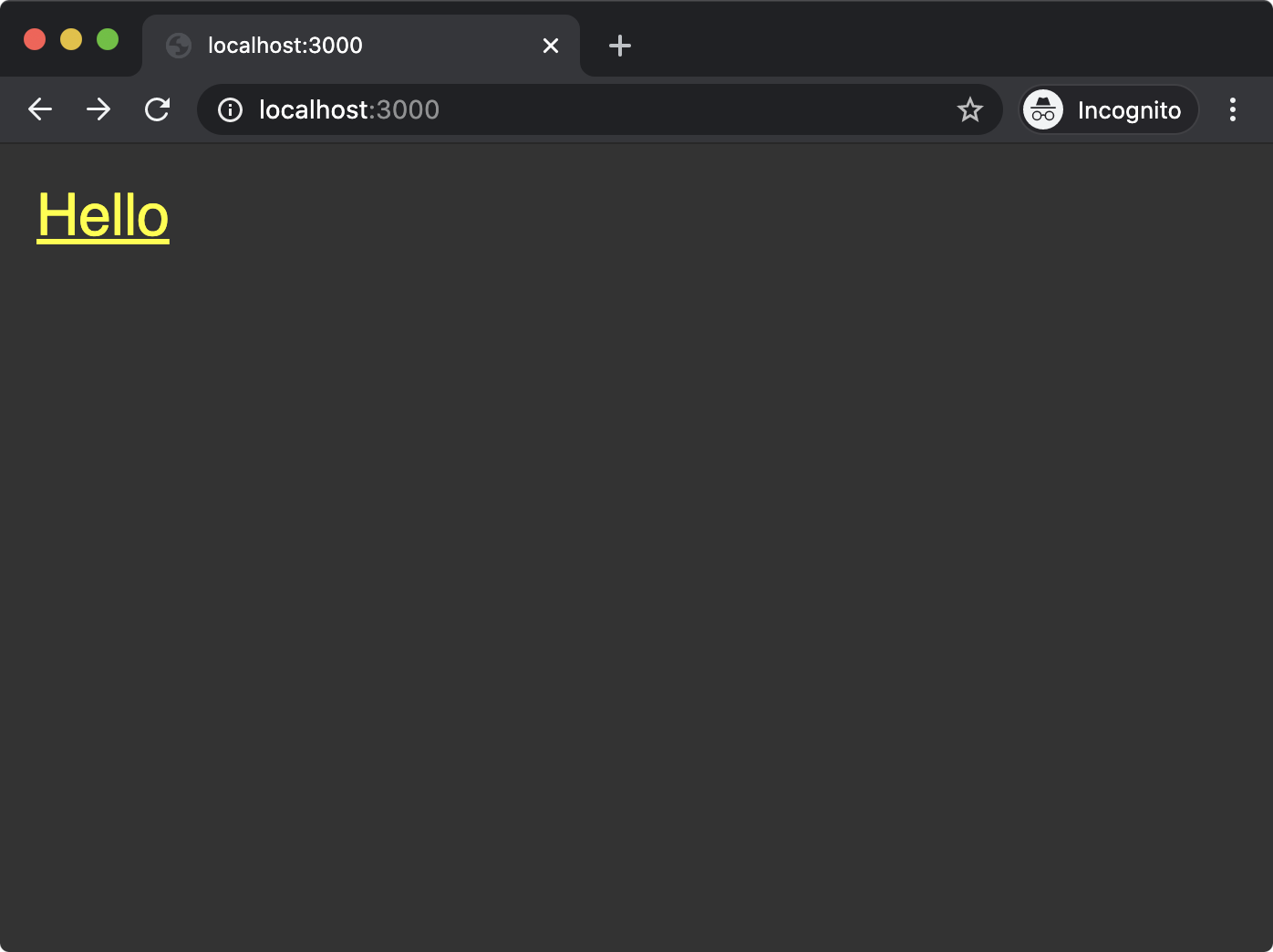
The Node core **url** module has a **parse** method which turns a URL string into an object containing various segments of the URL, such as **host**, **protocol** and **pathname**. See the [Node.js Documentation](https://nodejs.org/dist/latest-v12.x/docs/api/url.html#url_url_parse_urlstring_parsequerystring_slashesdenotehost) to learn more.

The **req.url** property has a slightly misleading name. It does not hold the entire URL of an incoming request, only the relative path after the host portion. For instance a request to http://localhost:3000/hello will result in a **req.url** of **/hello**. The reason we pass **req.url** to **url.parse** is to separate any potential query string from the URL. Now, let's consider a request to http://localhost:3000/hello?foo=1. It would result in a **req.url** value of **/hello?foo=1**. Passing such a string to **url.parse** will result in an object with a **pathname** property of **/hello**.

If the **pathname** is **/** then we end the response with the contents of **root** and exit the function early with a **return** keyword. In this case, there is no need to set **res.statusCode** because the default **res.statusCode** is 200 (OK).

For more information on status codes see the [MDN web docs](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status).

If we navigate to ht‌tp://localhost:3000 in the browser we should see the following:

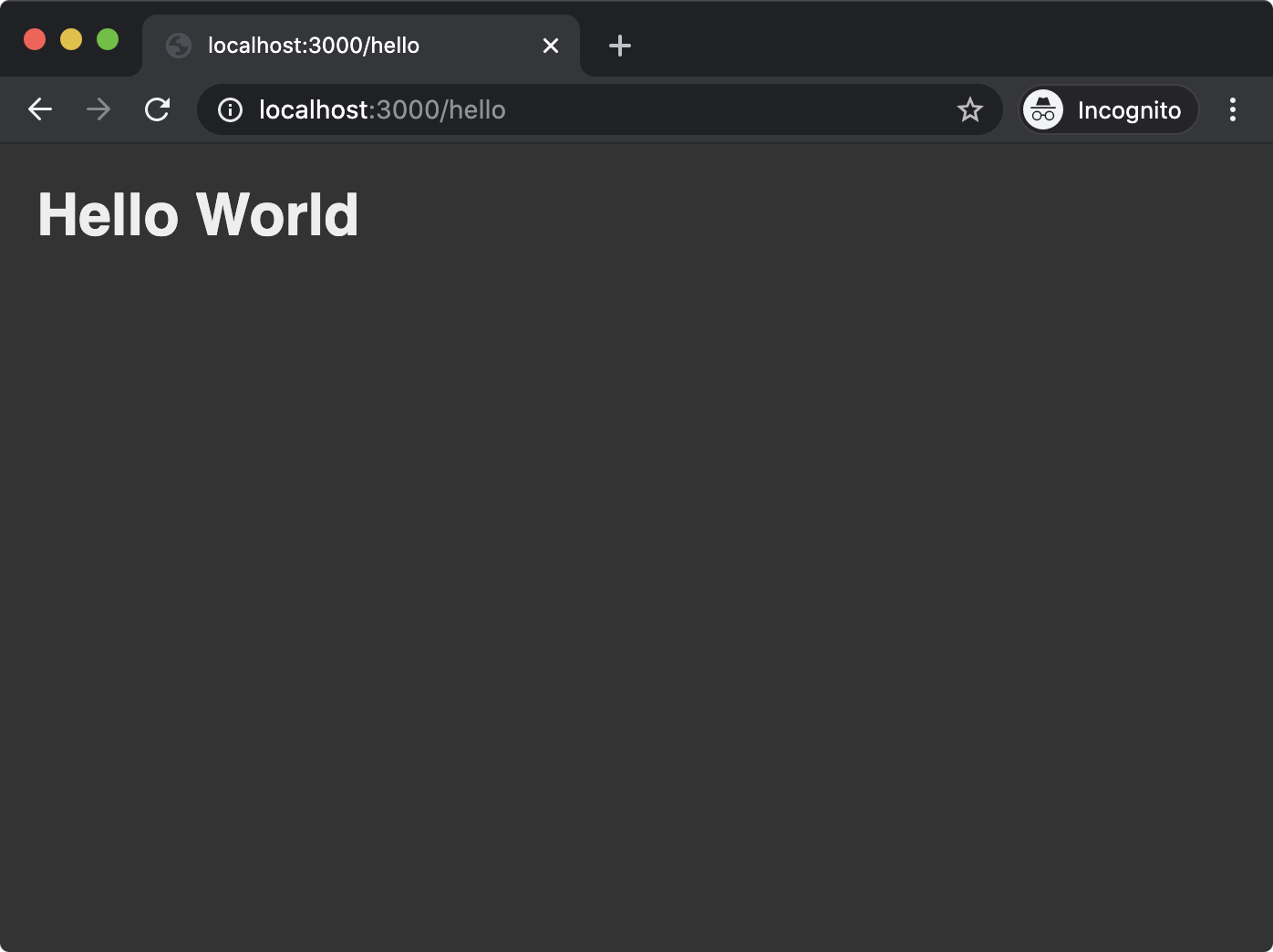


Cont'd on the next page.

# Creating a Web Server with Node Core (4)

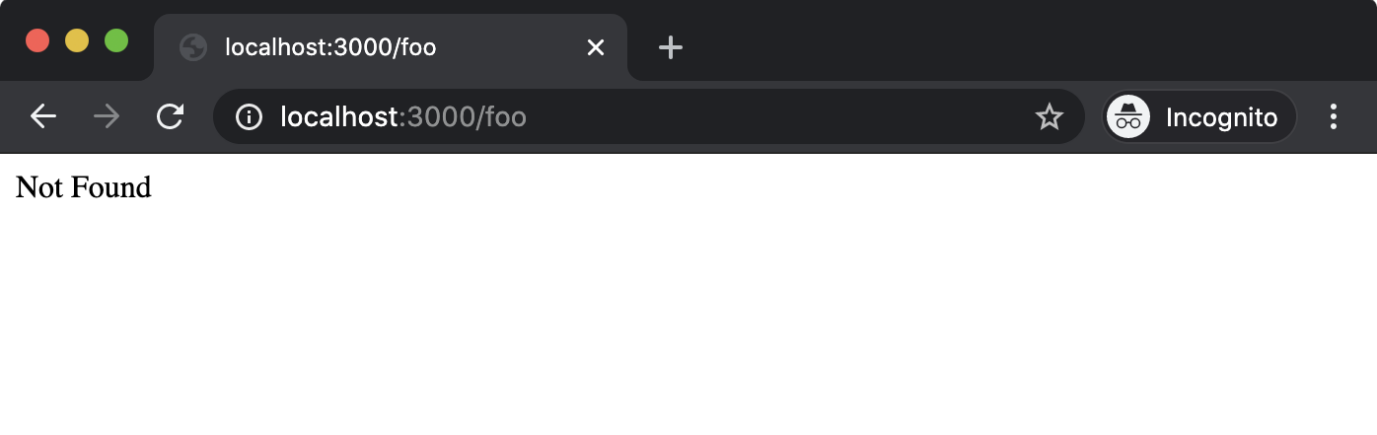
If the **pathname** is **/hello** then we end the response with the contents of **hello** and exit the function. Again, no need to set the **res.statusCode** property.

In the browser, upon clicking the link or manually navigating to http://localhost/hello we should see the following:



If **pathname** is neither **/hello** nor **/** the end of the function is reached, where the **res.statusCode** property is set to **404** and the response is ended with the corresponding status message (Not Found).

Navigating to http://localhost:3000/foo in the browser should result in the following:



We have now created a very basic web server. This procedural approach can become very rigid and unwieldy if we were to attempt to extend functionality over time. In the next sections we'll learn how to use the Express and Fastify frameworks to achieve the same results in a more flexible, declarative manner.

# Creating a Web Server with Express (1)

Now that we understand what it takes to create a rudimentary web server using Node.js core **http** and **url** modules, let's turn our attention to expediting the process of creating, extending and maintaining Node.js web servers.

Express is one of the most widely used Node.js frameworks. More so when it comes to generating and delivering HTML dynamically, as opposed to delivering RESTful JSON content as a service.

The core development efforts of the Express project have been stagnant since around 2017. Despite renewed interest in modernizing the framework from the OSS community and technology industry circa 2020, it has remained in maintenance mode. In this course, we'll be focusing on Express 4, which, although it was first released 8 years ago, is the latest major version at the time of writing (February 2022).

Even after the release of the next major version (Express 5, last alpha was two years ago, ETA unknown), understanding version 4 is essential from a pragmatic perspective since so many legacy code bases have been built using Express 3 and 4, which are fairly similar to each other.

Let's explore Express 4. First, create:

* an **express-web-server** folder with an **app.js file**
* a **routes** folder with **index.js** and **hello.js** files, and
* a **bin** folder with a **www** file:

**node -e "fs.mkdirSync('express-web-server')"  
cd express-web-server  
node -e "fs.mkdirSync('routes')"  
node -e "fs.mkdirSync('bin')"  
node -e "fs.openSync('app.js', 'w')"  
cd routes  
node -e "fs.openSync('index.js', 'w')"  
node -e "fs.openSync('hello.js', 'w')"  
cd ../bin  
node -e "fs.openSync('www', 'w')"  
cd ..**

Next, we'll generate a **package.json** file for the project and then install Express and another package called **http-errors**:

**npm init -y  
npm install express@4 http-errors@2**

The generated **package.json** file should look similar to the following:

**{  
  "name": "express-web-server",  
  "version": "1.0.0",  
  "description": "",  
  "main": "app.js",  
  "scripts": {  
    "test": "echo \"Error: no test specified\" && exit 1"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "dependencies": {  
    "express": "^4.17.2",  
    "http-errors": "^2.0.0"  
  }  
}**

We need to modify the **scripts** object in the **package.json** file to the following:

**"scripts": {  
    "test": "echo \"Error: no test specified\" && exit 1",  
    "start": "node ./bin/www"  
  },**

Here we've added a **start** script, so when we run **npm start** the **bin/www** file will be executed with Node.

We're using this folder structure for the sake of familiarity: it's a conventional Express structure. A tool called [**express-generator**](https://www.npmjs.com/package/express-generator) generates this structure with some other additions. We'll use the generator in the next chapter, for now we're concentrating on the bare bones to create the equivalent of our minimum viable web server that we implemented in the previous section.

Cont'd on the next page.

# Creating a Web Server with Express (2)

We're going to build the **app.js** file iteratively. For our first iteration **app.js** should look as follows:

**'use strict'  
const express = require('express')**

**const app = express()**

**module.exports = app**

All we're doing here is instantiating an Express instance and exporting it from the **app.js** file as a module.

The **bin/www** file is the entry point for the application, and is responsible for starting the server. It should look as follows:

**#!/usr/bin/env node  
'use strict'**

**const app = require('../app')  
const http = require('http')**

**const PORT = process.env.PORT || 3000**

**const server = http.createServer(app)**

**server.listen(PORT)**

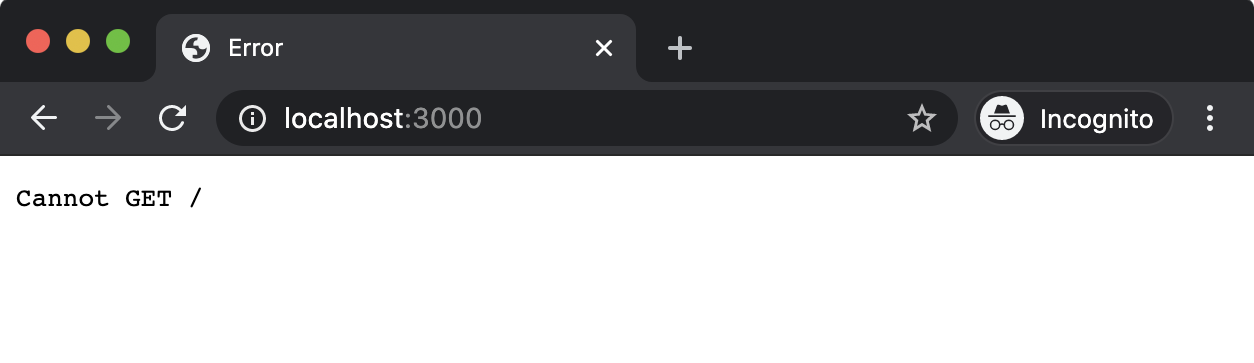
In the previous section we learned about the **http.createServer** method. It takes a function which is passed a request and response object. In **bin/www** we import the **app.js** file we just wrote, assign it to the **app** constant and then pass that to **http.createServer**. An Express instance on the other hand is actually a function that accepts a request object and a response object, that's why we pass the Express instance to **http.createServer**. In the next steps, we'll configure our Express **app** instance, and this will change how the function that is passed to **http.createServer** behaves.

We should now be able to get our server up and running with the following command:

**npm start**

This command will look up the **package.json** **scripts.start** field and then run the command (**node ./bin/www**). The **bin/www** file will create a server with the Node core **http.createServer** method, but use the **app.js** file to generate the function which is passed to **createServer**. Like our HTTP server in the previous section, once the process has started, it will not output anything.

If we navigate in a browser to http://localhost:3000, we should see the following output:



We haven't defined any routes, so the generated Express function will default to this behavior. Before we create our routes, let's align the error handling with our HTTP server from the previous section by modifying **app.js** to the following:

**'use strict'  
const express = require('express')  
const createError = require('http-errors')**

**const app = express()**

**app.use((req, res, next) => {  
  if (req.method !== 'GET') {  
    next(createError(405))  
    return  
  }  
  next(createError(404))  
})**

**app.use((err, req, res, next) => {  
  res.status(err.status || 500)  
  res.send(err.message)  
})**

**module.exports = app**

Cont'd on the next page.

# Creating a Web Server with Express (3)

Configuring an Express servers' behavior is almost always performed with **app.use** (where **app** is the Express instance). The **app.use** method takes a function which is very similar to the function that is passed to **http.createServer**. The function will be called for every incoming request, and it will be passed a request object and a response object.

The difference between the function passed to **app.use** and the function passed to **http.createServer** is that it can also have a third parameter called **next**. This is an error-first callback function that is called when the function passed to **app.use** has completed any tasks and is ready to handover to the subsequent function registered via **app.use**. So this means that instead of passing one big function to **http.createServer**, multiple functions can be registered via **app.use**. They will be called in order of registration, each one handing over to the following one when it's done processing. If the **next** function is not called, then the request handling ends there and none of the ensuing registered functions are called for that request. This approach is known as the middleware pattern. The building blocks for configuring an Express server are middleware functions.

In our case we've configured two pieces of middleware. The first middleware function we registered should always be the second-to-last middleware. Essentially, if this middleware has been reached then we can assume that no routes were matched. Therefore we generate a 404 error using the **http-errors** module (part of the Express ecosystem). The **http-errors** module will generate an appropriate message for any HTTP status code passed to it. We then pass this error object as the first argument to the **next** callback function, which lets Express know that an error has occurred. We may also pass a 405 (Method Not Allowed) error instead, if we find that the **req.method** property does not have the value of GET. This matches the functionality in our HTTP server implementation from the first section. Currently, we have no routes registered, so a 404 error is the default for any HTTP GET requests.

The very last piece of middleware in our modified **app** file should always be the final piece of middleware. This registered middleware specifies four parameters instead of the usual three. This makes Express recognize the middleware as the final error handling middleware and passes the error object that we pass to **next** in the prior middleware as the first argument of this special error-handling middleware function. From there we can grab the HTTP status code from the error object and use it to set the response status code. Notice that we use a **res.status** function instead of the **res.statusCode** property. Similarly, we can use **res.send** instead of **res.end** to write and end the response. This is another method added by Express that will detect the Content-Type from the input, and potentially perform additional operations. For instance, if an object was passed to **res.send** that object would be serialized to JSON and the response Content-Type would automatically be set to **application/json**.

Even though the **req** and **res** objects are generated by the **http** module and have all of the same functionality, Express decorates the **req** and **res** objects with additional functionality. We could not have used **res.status** or **res.send** in the previous section because these functions did not exist. Some, including this author, view Express' decorator approach on core APIs as a mistake. By conflating Node core APIs with Express APIs on the same objects the principles of least surprise and separation of concerns are violated, while also causing performance issues. However, so much legacy code has been written with Express it's important to understand its APIs.

Let's move onto creating our routes, the **routes/index.js** file should look as follows:

**'use strict'  
const { Router } = require('express')  
const router = Router()**

**const root = `<html>  
<head>  
  <style>  
   body { background: #333; margin: 1.25rem }  
   a { color: yellow; font-size: 2rem; font-family: sans-serif }  
  </style>  
</head>  
<body>  
  <a href='/hello'>Hello</a>  
</body>  
</html>  
`**

**router.get('/', (req, res) => {  
  res.send(root)  
})**

**module.exports = router**

Note that inlining HTML like this is uncommon and it was used here only for demonstration purposes. Usually, HTML would be dynamically generated with a template language or maybe a Server Side Rendering of a frontend framework such as React. We'll look briefly at templates in the next chapter.

Cont'd on the next page.

# Creating a Web Server with Express (4)

To add a route we create an instance of the Express router, and then use its **get** method to define a GET route. The router supports all HTTP verbs (e.g. POST, PUT and so forth). In each case when a router HTTP verb method is called the first argument passed to it is a string declaring the path for the route. In this case we just use **/**, since this is our root route. The second argument is a route middleware function. In our case we simply send our HTML string (**root**) to the response. The router instance is exported from the **routes/index.js** file, and we'll later import it into **app.js** in order to register it with the Express application instance.

Now, let's create our **/hello** route. The **routes/hello.js** file should contain the following:

**'use strict'  
const { Router } = require('express')  
const router = Router()**

**const hello = `<html>  
  <head>  
    <style>  
     body { background: #333; margin: 1.25rem }  
     h1 { color: #EEE; font-family: sans-serif }  
    </style>  
  </head>  
  <body>  
    <h1>Hello World</h1>  
  </body>  
</html>`**

**router.get('/', (req, res) => {  
  res.send(hello)  
})**

**module.exports = router**

This is very similar to the root route. Note that we define the route path as **/** in this case as well, instead of **/hello**. This is because we'll be mounting this router at the **/hello** route path in **app.js** instead. This pattern allows for easy renaming of routes at the top level.

Finally, let's register our routes in **app.js**, by modifying **app.js** to the following:

**'use strict'  
const express = require('express')  
const createError = require('http-errors')  
const indexRoutes = require('./routes')  
const helloRoutes = require('./routes/hello')**

**const app = express()**

**app.use('/', indexRoutes)  
app.use('/hello', helloRoutes)**

**app.use((req, res, next) => {  
  if (req.method !== 'GET') {  
    next(createError(405))  
    return  
  }  
  next(createError(404))  
})**

**app.use((err, req, res, next) => {  
  res.status(err.status || 500)  
  res.send(err.message)  
})**

**module.exports = app**

Cont'd on the next page.

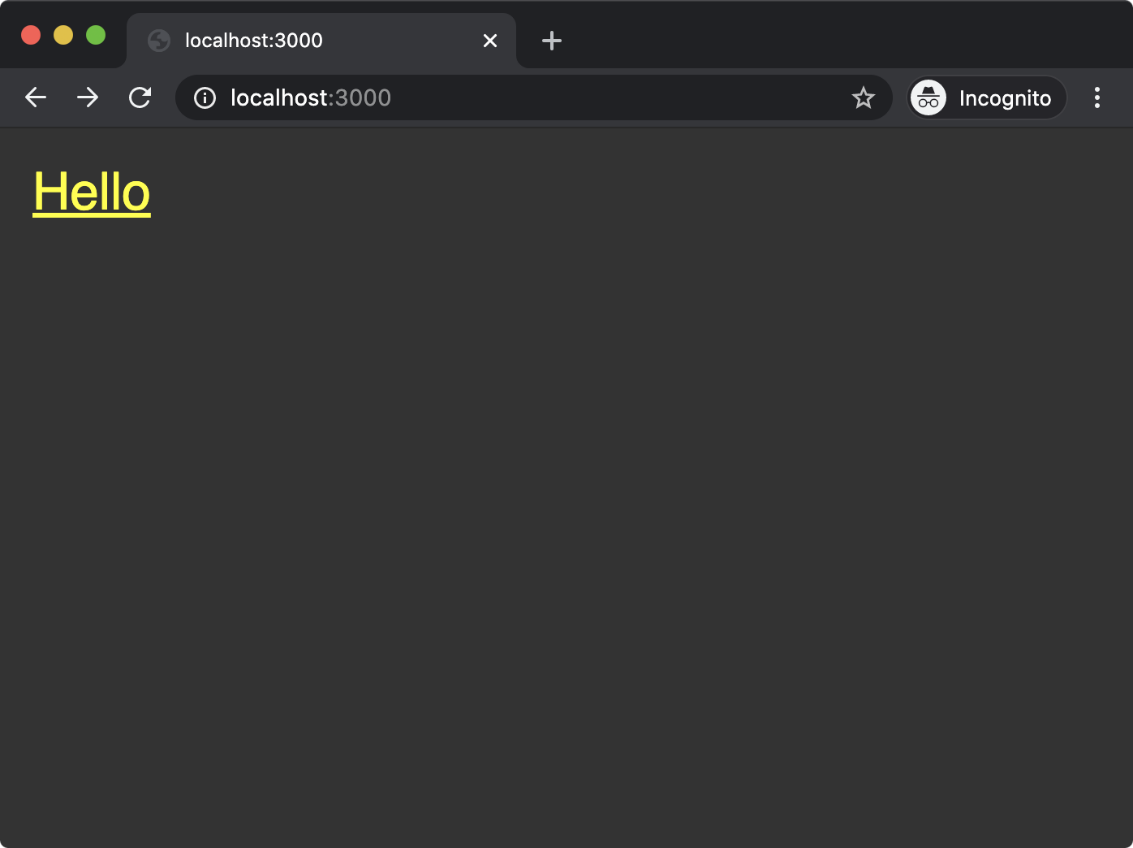
# Creating a Web Server with Express (5)

We imported both **routes/index.js** and **routes/hello.js**, and assigned them to **indexRoutes** and **helloRoutes** respectively. Then above our 404/405 error handler middleware, we registered the imported **express.Router** instances with **app.use**. The **app.use** function can optionally take a mount point, which is a string representing a route path. This means it will only apply the registered middleware (the **express.Router** instances are also middleware functions) when incoming requests match that path. We mount the **indexRoutes** at the **/** route path and the **helloRoutes** at the **/hello** route path.

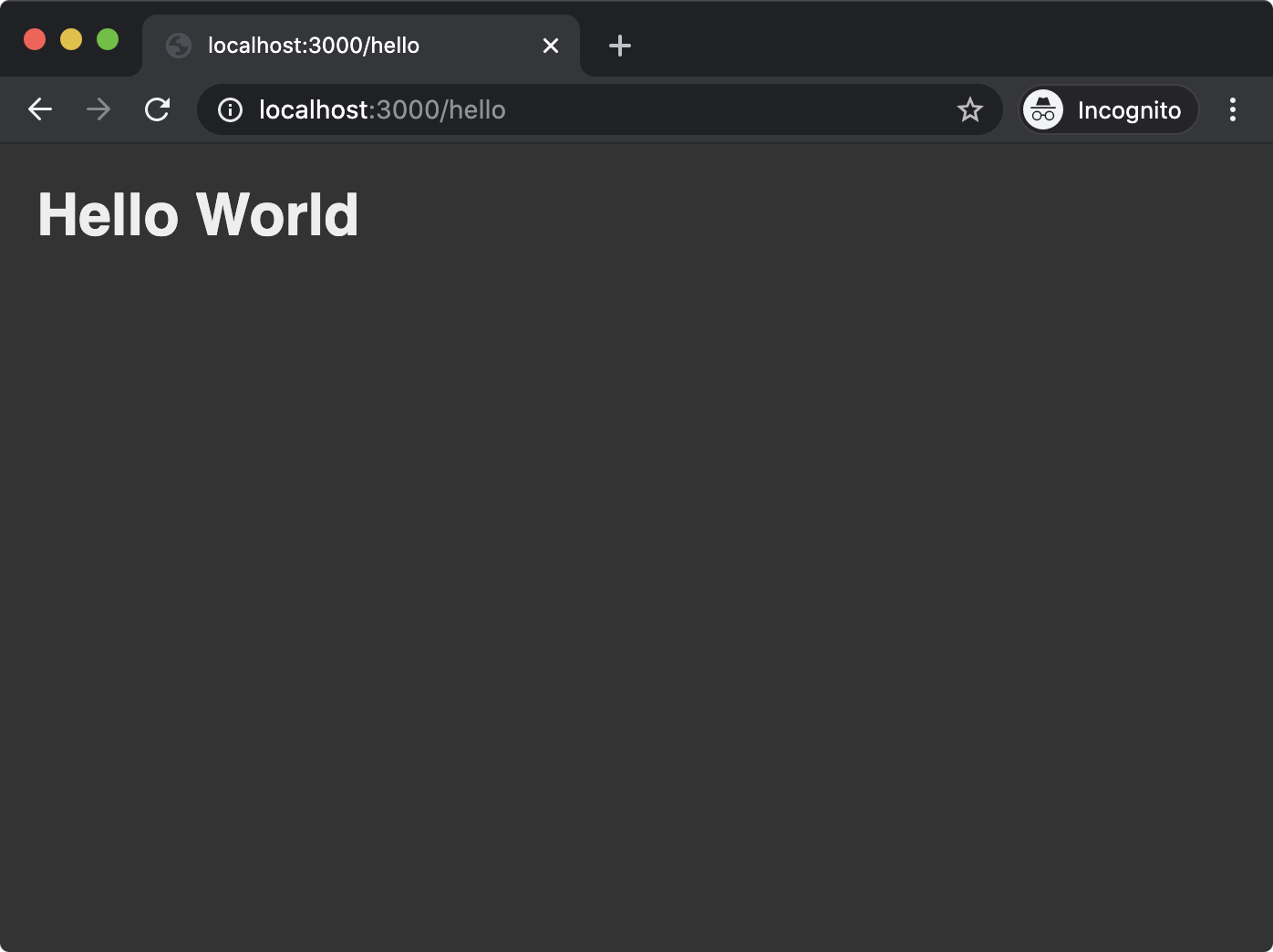
We need to start a server with our new changes. If our server is still running, we can stop it with Ctrl+C and then start it again with the following command:

**npm start**

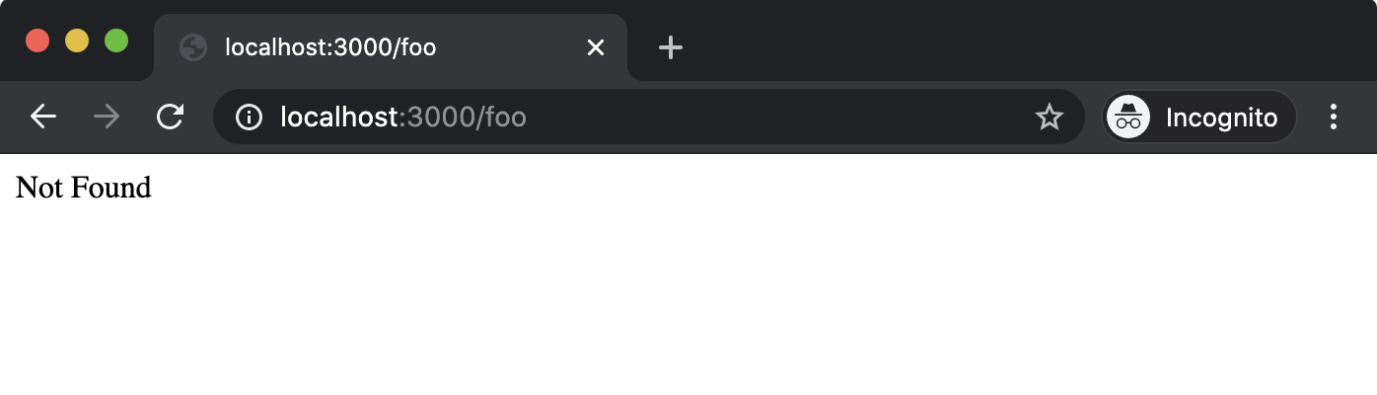
Now, if we navigate in a browser to http://localhost:3000, we should see the following:



If we click the link (or navigate to http://localhost:3000/hello), we should see th-e following:



If we navigate to a non-existent route, we should see the following



This section has been a deep-end dive into Express fundamentals. Throughout the course we'll continue to explore Express and retrace key topics.

# Creating a Web Server with Fastify (1)

Fastify is an up-and-coming framework in the Node.js ecosystem. It's specifically geared towards creating RESTful JSON services but can also be used for serving HTML as in our example implementation in the previous two sections.

Instead of middleware, Fastify supports a plugin-based pattern which provides full code isolation and encapsulation. We'll explore this more in this section.

Fastify explicitly supports newer language features (such as async/await), has a focus on modern developer experience and is the most performant framework in the Node.js ecosystem. Not only that but Fastify also provides full Express integration via the [**fastify-express**](https://trainingportal.linuxfoundation.org/learn/course/nodejs-services-development-lfw212/creating-a-web-server/ADDWEBSITEADDRESS) plugin. This means that the vast Express ecosystem can be used with Fastify (often at higher requests per second than using the same middleware with Express!), and entire Express projects can be encapsulated in a Fastify plugin and used as part of a Fastify project.

In this section, however, we'll implement the same server created in the previous two sections but with Fastify. Let's start by making a folder:

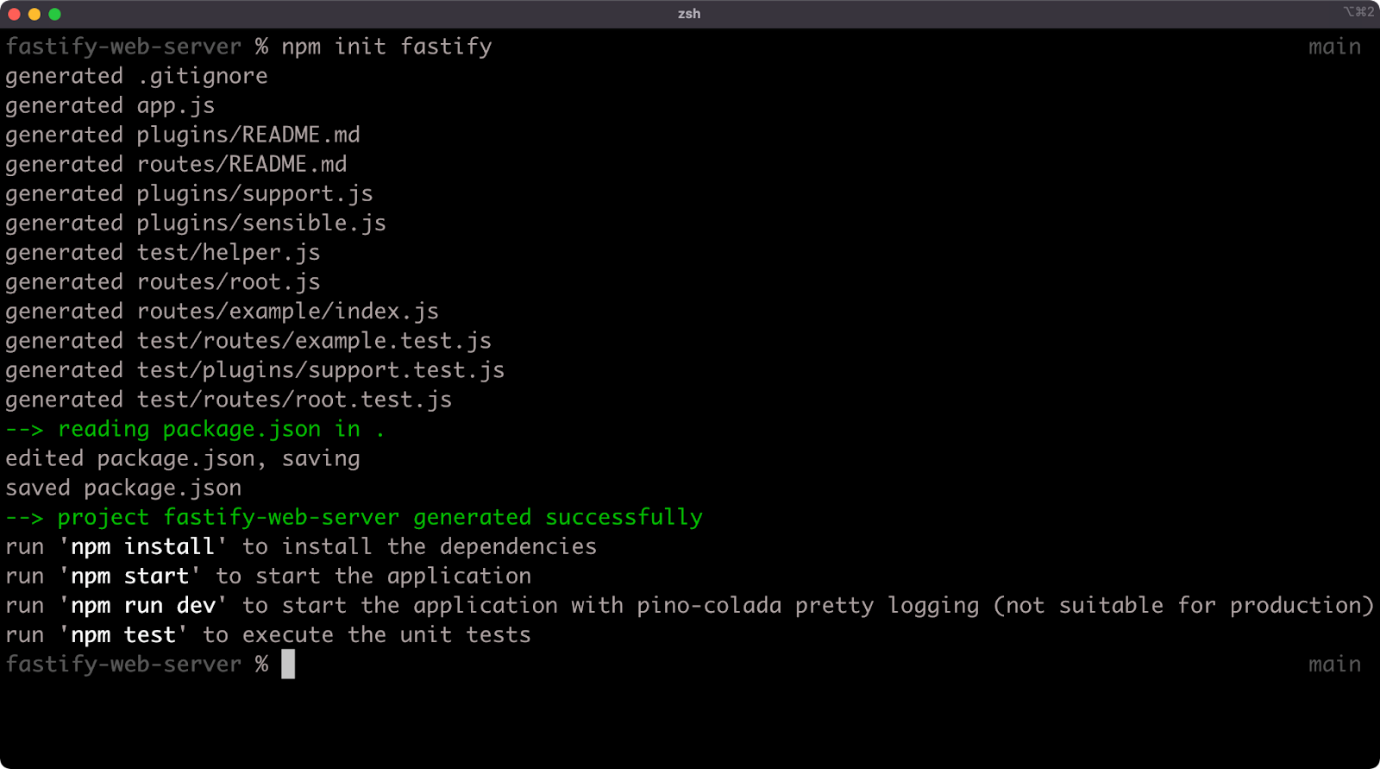
**node -e "fs.mkdirSync('fastify-web-server')"  
cd fastify-web-server**

Now, we can run the following command to bootstrap a Fastify project:

**npm init fastify**

It's important to ensure that this command is executed within the **fastify-web-server** folder otherwise files will be added to unintended locations.

This command should produce output similar to the following:



As we can see from the output, the **npm init fastify** command creates some files in the current working directory. The following files and folders are generated:

* **.gitignore**
* **package.json**
* **app.js**
* **plugins/README.md**
* **test/helper.js**
* **routes/README.md**
* **plugins/support.js**
* **routes/root.js**
* **test/routes/example.test.js**
* **test/plugins/support.test.js**
* **routes/example/index.js**
* **test/routes/root.test.js**

We'll only need to edit the **app.js**, **routes/root.js** and **routes/example/index.js** files to create the same implementation as in the previous two sections.

Cont'd on the next page.

# Creating a Web Server with Fastify (2)

Before doing anything else, let's make sure that the project dependencies are installed. In the same folder, let's run the following command:

**npm install**

While that's running, let's take a look at the contents of the **app.js** file (any comments have been stripped):

**'use strict'**

**const path = require('path')  
const AutoLoad = require('fastify-autoload')**

**module.exports = async function (fastify, opts) {  
  fastify.register(AutoLoad, {  
    dir: path.join(\_\_dirname, 'plugins'),  
    options: Object.assign({}, opts)  
  })**

**fastify.register(AutoLoad, {  
    dir: path.join(\_\_dirname, 'routes'),  
    options: Object.assign({}, opts)  
  })  
}**

The **app.js** file is the entry-point for the project and exports an **async** function. A Fastify plugin is a function that accepts a server instance and options as parameters. It may accept a third parameter, a **next** callback or it may return a promise (which is what an **async** function does). So the **app.js** file is actually exporting a Fastify plugin.

The server instance that is passed as the first argument to this function is named **fastify**. Additional plugins are registered with the **registered** method. In this case, a single plugin is registered twice. The **fastify-autoload** plugin automatically loads folders of plugins, so all **app.js** is doing is setting up a convenient way for us to define and work with plugins and routes. In both cases where **fastify.register** is called, the **fastify-autoload** plugin (**AutoLoad**) is passed as the first parameter and an object is passed as the second parameter. This second parameter is the options for the **AutoLoad** plugin. The **dir** option in each case points the **fastify-autoload** plugin to a **plugins** folder and a **routes** folder. The **options** option in each case specifies options that would be passed to all plugins that are autoloaded. It's essentially shallow merging the options passed to the **app.js** plugin function with an empty object.

The **package.json** added by the **npm init fastify** command should look something like the following:

**{  
  "name": "bah",  
  "version": "1.0.0",  
  "description": "This project was bootstrapped with Fastify-CLI.",  
  "main": "app.js",  
  "directories": {  
    "test": "test"  
  },  
  "scripts": {  
    "test": "tap \"test/\*\*/\*.test.js\"",  
    "start": "fastify start -l info app.js",  
    "dev": "fastify start -w -l info -P app.js"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "dependencies": {  
    "fastify": "^3.0.0",  
    "fastify-autoload": "^3.10.0",  
    "fastify-cli": "^2.15.0",  
    "fastify-plugin": "^3.0.0",  
    "fastify-sensible": "^3.1.2"  
  },  
  "devDependencies": {  
    "tap": "^15.1.6"  
  }  
}**

Cont'd on the next page.

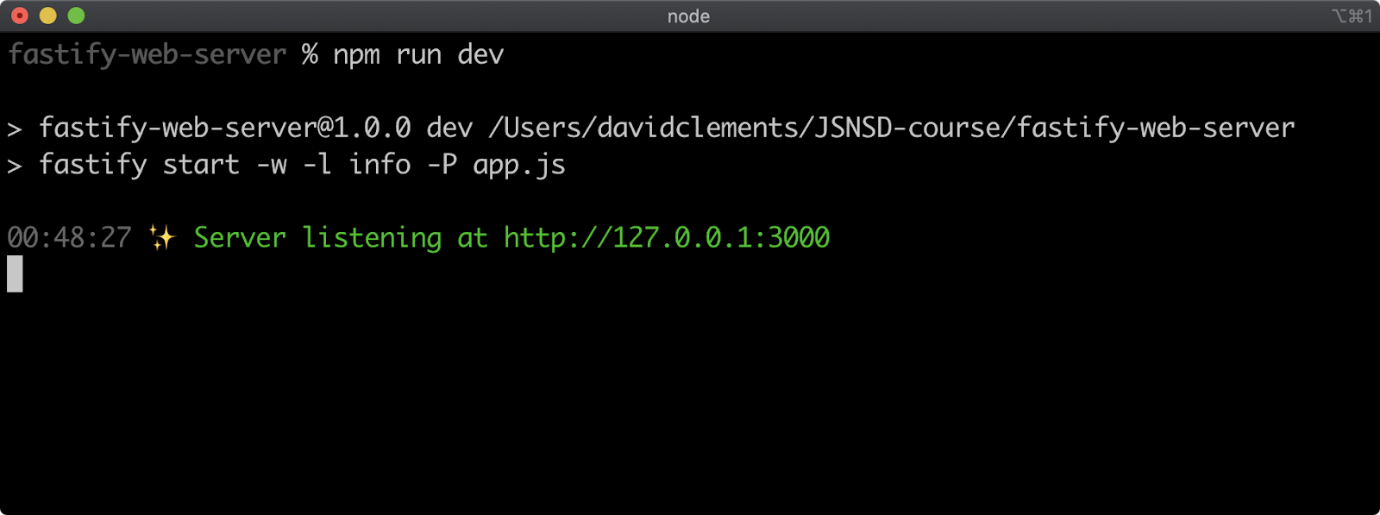
# Creating a Web Server with Fastify (3)

In the **scripts** field of the **package.json** we can see a **start** field and a **dev** field. In each case the **app.js** file is booted with the **fastify start** command. This works because the **fastify-cli** dependency (which can be seen in the **dependencies** field of the **package.json**) provides a CLI named **fastify** which is accessible to **npm** when running **package.json** scripts. The **fastify start** command automatically starts the server (performing the same role as the **bin/www** file in the Express example). Notice we have no defined port, this is because **fastify start** defaults to port 3000, it could be configured with the **-p** flag (lowercase) if desired.

In the **dev** script there are two additional flags: **-w** and **-P** (uppercase). The **-P** flag means "prettify the log output", which would otherwise be newline delimited JSON logs. The **-w** flag means “watch and reload the project as we work on it”, so we can go ahead and run the following to start our server:

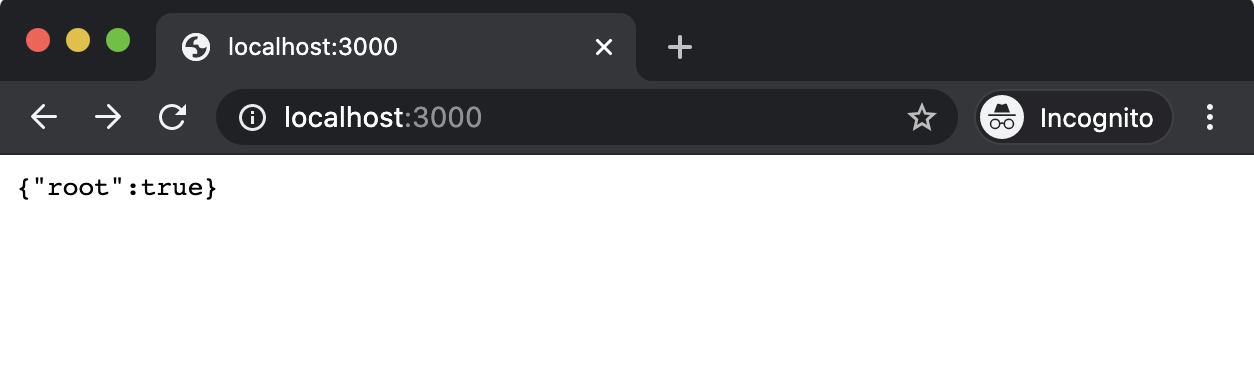
**npm run dev**

This should display something like the following:



We shouldn't need to start and stop the server as we make changes, that will happen automatically.

We can check the server is running correctly by navigating to [http://localhost:3000](http://localhost:3000/) in the browser. We should see something like the following:



In Fastify, everything is a plugin. The distinction between plugins and routes is mostly convention-setting to help us reason about a server or service's functionality. The files in the **routes** folder are actually plugins (exported functions that return promises or use a **next** callback). The files in the **plugins** folder are also plugins, but they are more commonly de-encapsulated plugins, meaning that the functionality that they provide can be accessed by sibling plugins. Think of the **plugins** folder like a **lib** folder, but where a strict and enforceable common interface is used for every exported piece of functionality. Use of the **plugins** folder will be explored more in later sections. The entry point is a plugin. Routes are plugins. Plugins (local libraries) are plugins.

Cont'd on the next page.

# Creating a Web Server with Fastify (4)

A key difference between Express middleware and Fastify plugins is that Express middleware is executed for every request (if reachable) but Fastify plugins are called only at initialization time. Fastify plugins are always asynchronous (either with a callback or a returned promise) to allow for asynchronous initialization of every plugin.

Now, let's focus on the **routes** folder. We just saw that the root route (**/**) responds with **{"root":true}**. Let's take a look at the code in **routes/root.js**:

**'use strict'**

**module.exports = async function (fastify, opts) {  
  fastify.get('/', async function (request, reply) {  
    return { root: true }  
  })  
}**

The **routes/root.js** file exports an **async** function that accepts the **fastify** instance and an options argument. The **routes/root.js** file exports a Fastify plugin. A Fastify plugin is a function that takes the server instance (**fastify**).

Within the plugin function, **fastify.get** is called. This registers an HTTP GET route. The first argument is a string containing a forward slash (**/**), indicating that the route being registered is the root route (**/**). All HTTP verbs can be called as methods on the **fastify** instance (e.g. **fastify.post**, **fastify.put** and so on).

The second argument passed to **fastify.get** is an **async** function, the route handler, which accepts the **request** and **reply** objects. The **request** and **reply** objects have the same objective as the **http** and Express **req** and **res** objects but they have a different (and separate API). To learn more see the ["Fastify: Request"](https://www.fastify.io/docs/v3.27.x/Reference/Request/) and ["Fastify: Reply"](https://www.fastify.io/docs/v3.27.x/Reference/Reply/) Documentation.

The **fastify.get** method can accept a normal synchronous function or an async function. Whatever is returned from the function or async function is automatically processed and sent as the content of the HTTP response.

Alternatively the **reply.send** method can be used (e.g. **reply.send({root: true})**), which is similar to the **res.send** method of Express. This can be useful when working with nested callback APIs.

Since an object is returned, Fastify converts it to a JSON payload before sending it as a response.

Let's edit the **routes/root.js** file to the following:

**'use strict'**

**const root = `<html>  
<head>  
  <style>  
   body { background: #333; margin: 1.25rem }  
   a { color: yellow; font-size: 2rem; font-family: sans-serif }  
  </style>  
</head>  
<body>  
  <a href='/hello'>Hello</a>  
</body>  
</html>  
`**

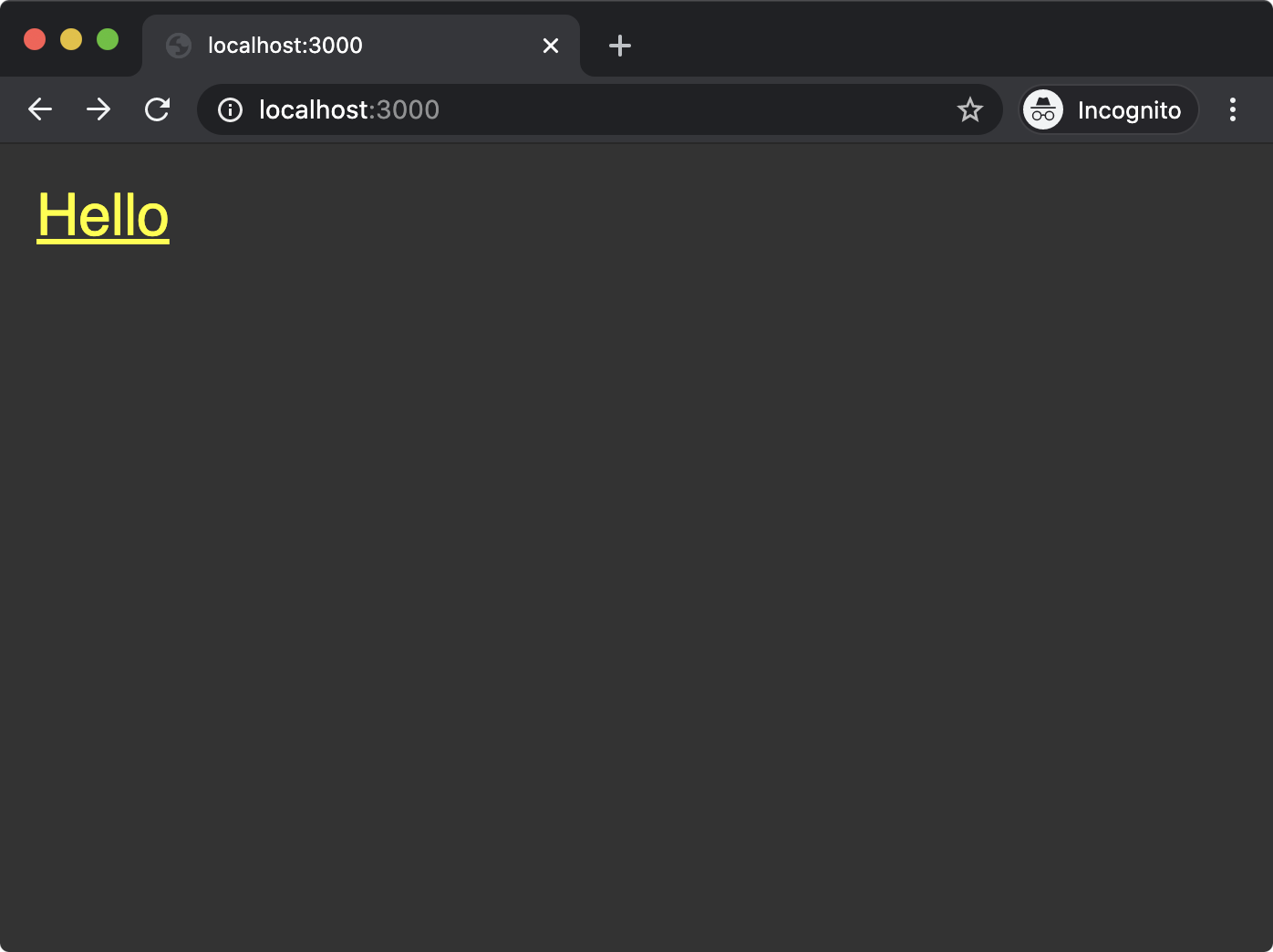
**module.exports = async function (fastify, opts) {  
  fastify.get('/', async function (request, reply) {  
    reply.type('text/html')  
    return root  
  })  
}**

We've added the now familiar **root** string of HTML as a constant to the **routes/root.js** file and then we return root from the **async** function passed to **fastify.get** instead of returning an object. We've also used the Fastify API method **reply.type** to set the Content-Type header to **text/html**.

Cont'd on the next page.

# Creating a Web Server with Fastify (5)

If we now navigate in the browser to http://localhost:3000 we should see the following:



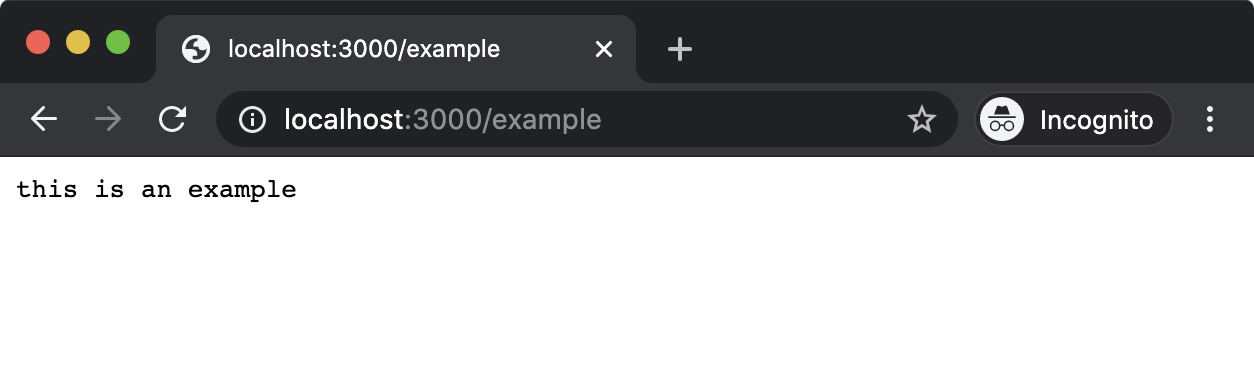
The other defined route is in **routes/example/index.js**, let's see what that looks like:

**'use strict'**

**module.exports = async function (fastify, opts) {  
  fastify.get('/', async function (request, reply) {  
    return 'this is an example'  
  })  
}**

This code is very similar to the original **routes/root.js** route. Again, we have an exported **async** function that accepts the server instance (**fastify**) and options (**opts**). And again **fastify.get** is used to register a route, where the second parameter is an **async** function that is passed **request** and **reply** objects representing the incoming request and the outgoing response. However, notice that the defined route, the first argument passed to **fastify.get**, is also **/** (not **/example**).

Let's navigate to http://localhost:3000/example in the browser:



When a route is defined in a subfolder, by default, the **fastify-autoload** plugin will register that route prefixed with the name of the subfolder. So the example route is at **routes/example/index.js** and registers a route at **/**. This causes **fastify-autoload** to register the server route at **/example**. If the route passed to **fastify.get** in **routes/example/index.js** had been **/foo** then **fastify-autoload** would have registered that route at **/example/foo**.

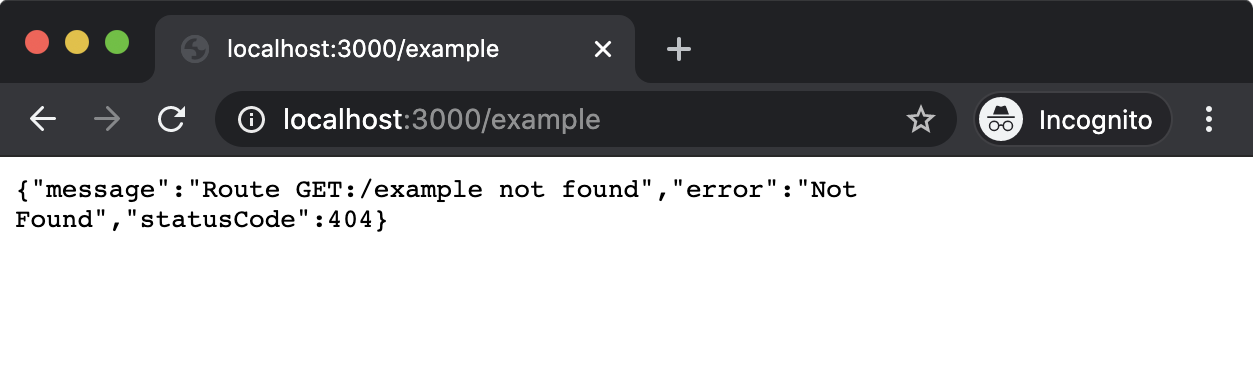
Cont'd on the next page.

# Creating a Web Server with Fastify (6)

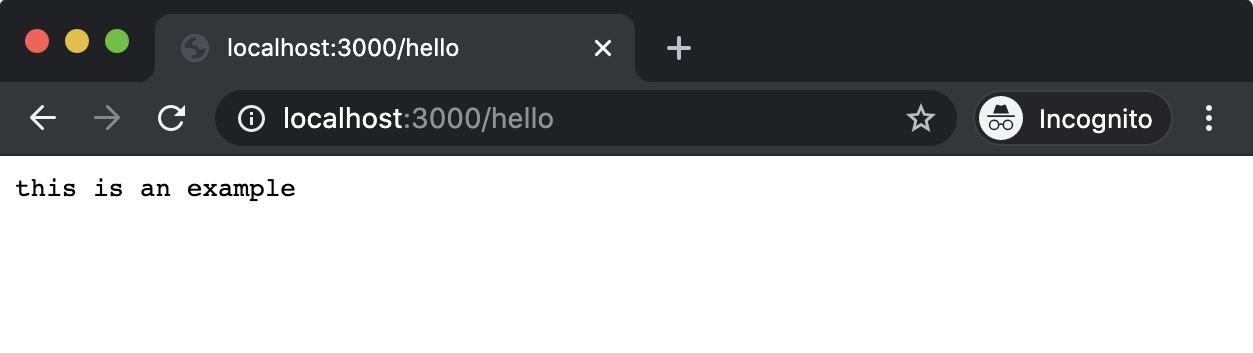
We need a **/hello** route, so let's rename the **routes/example** folder to **routes/hello**. Leaving the **npm run dev** command running, use another terminal with the working directory set to our **fastify-web-server** folder to run the following:

**cd routes  
node -e "fs.renameSync('example', 'hello')"  
cd ..**

Attempting to load http://localhost:3000/example in the browser will now result in the following:



This is the default Fastify 404 handling behavior. We'll modify this later to align with our server implementations, but for now we can see that the **/example** route no longer exists. However, if we navigate to http://localhost:3000/hello, we should see the following:



Now, let's modify **routes/hello/index.js** to contain the following code:

**'use strict'**

**const hello = `<html>  
  <head>  
    <style>  
     body { background: #333; margin: 1.25rem }  
     h1 { color: #EEE; font-family: sans-serif }  
    </style>  
  </head>  
  <body>  
    <h1>Hello World</h1>  
  </body>  
</html>`**

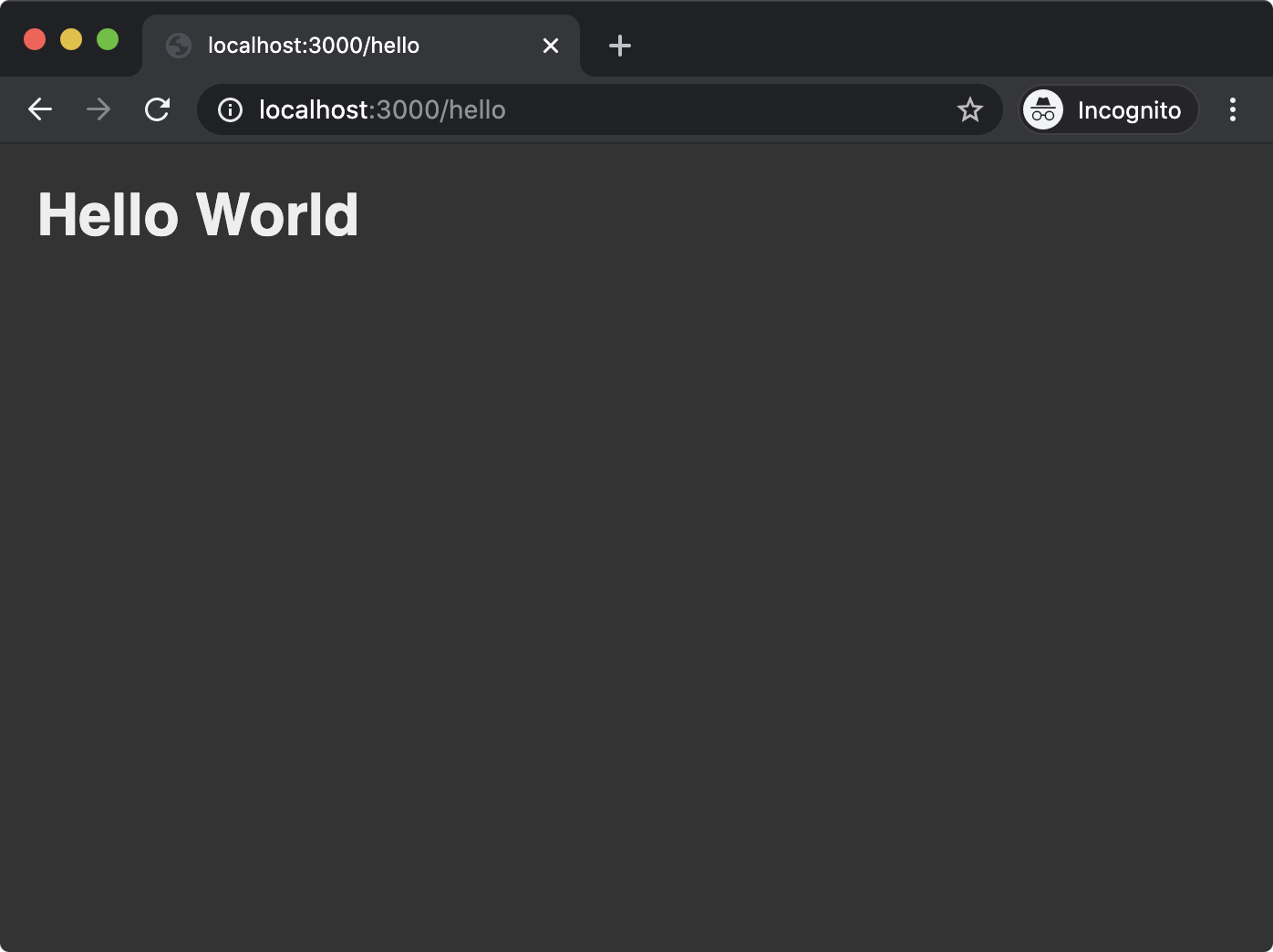
**module.exports = async function (fastify, opts) {  
  fastify.get('/', async function (request, reply) {  
    reply.type('text/html')  
    return hello  
  })  
}**

Cont'd on the next page.

# Creating a Web Server with Fastify (7)

Again, we see the familiar **hello** constant, which is returned from the route handler function passed as the second argument to **fastify.get**. At this point we can see that the code is somewhat repetitive. This is a good thing, Fastify is providing a strong declarative structure allowing us to focus on what we actually want to do instead of how to do it.

If we now navigate to http://localhost:3000/hello we should see the following:



Finally, to fully align this implementation with the **http** and Express implementations we need to modify the Not Found and Method Not Allowed behavior by making our **app.js** file look as follows (comments have been stripped):

**'use strict'**

**const path = require('path')  
const AutoLoad = require('fastify-autoload')**

**module.exports = async function (fastify, opts) {**

**fastify.register(AutoLoad, {  
    dir: path.join(\_\_dirname, 'plugins'),  
    options: Object.assign({}, opts)  
  })**

**fastify.register(AutoLoad, {  
    dir: path.join(\_\_dirname, 'routes'),  
    options: Object.assign({}, opts)  
  })**

**fastify.setNotFoundHandler((request, reply) => {  
    if (request.method !== 'GET') {  
      reply.status(405)  
      return 'Method Not Allowed\n'  
    }  
    return 'Not Found\n'  
  })**

**}**

The only addition is the **fastify.setNotFoundHandler** method call. This method accepts a function with the same criteria as the route handler function passed to **fastify.get** (and **fastify.post**, **fastify.put** and so on). In our case, we use a normal function, inspect the HTTP method and if it is not GET we set the HTTP status code to 405 and then return the associated message (Method Not Allowed). Otherwise we return the 404 message (Not Found). Fastify will call this function and use its output in cases where a route cannot be found (which includes routes that haven't been registered with the requested HTTP verb).

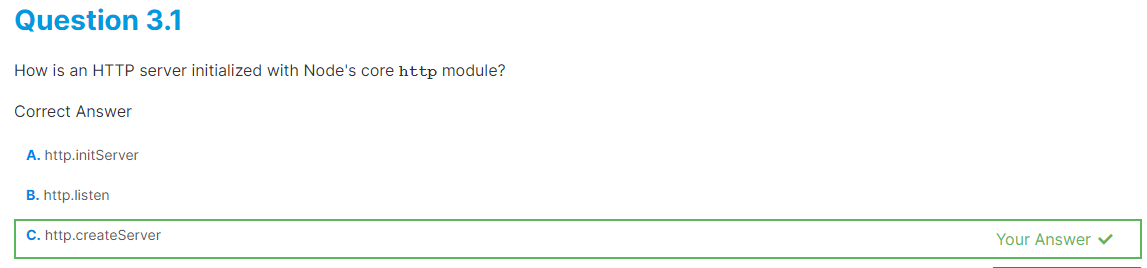
This has been an intense primer on Fastify. In the following chapters we'll be using it more and learning more about it as we go forward.

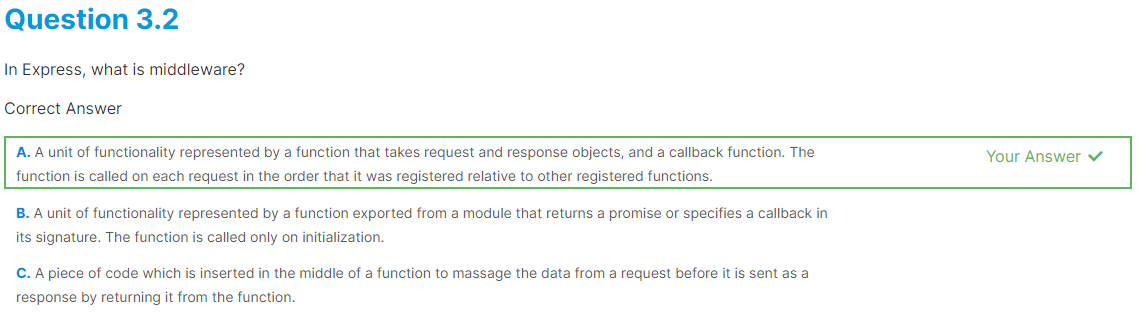
### Lab Exercises

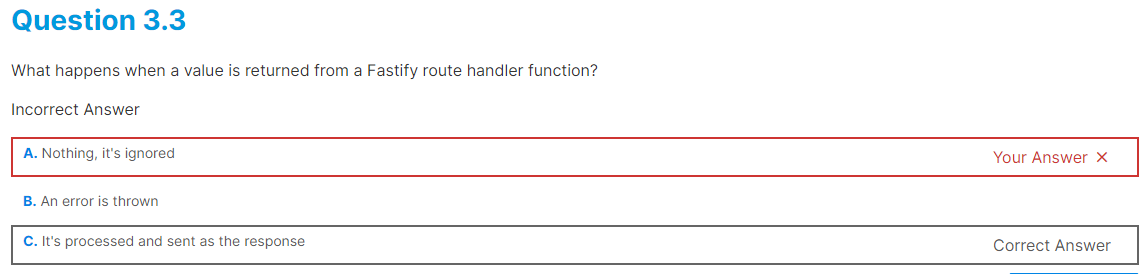
# Lab 3.1 - Deliver Data from a Library API

# Lab 3.2 - Implement a Status Code Response

### Knowledge Check







### **SERVING WEB CONTENT**Introduction

# Chapter Overview

Generally speaking, static assets (content that does not change very often) should not be served by Node. Static content should be delivered via a CDN and/or a caching reverse proxy that specializes in static content such as [NGINX](https://www.nginx.com/) or [Varnish](https://varnish-cache.org/). However, there are cases where serving static content with Node.js is useful. One case would be when working on a Node.js server locally without access to deployment infrastructure. Another is perhaps in independent development scenarios (as in "indie development") where investing time, effort and energy into additional infrastructure projects makes less sense than serving content from a Node.js process. At a stretch, Node.js could serve static content for applications with very small user bases that have a very low growth potential. In enterprise or scalable-startup scenarios, static content should be handled outside of Node.js. Where Node.js shines however, is dynamic content. Using Node.js as a mediator for gathering data from multiple sources and rendering some output is perfect for such an evented language and non-blocking I/O platform. In this chapter, we'll be exploring serving static content and dynamic content and also looking into streaming content with both Fastify and Express frameworks.

# Learning Objectives

By the end of this chapter, you should be able to:

* Learn how to serve static content with Fastify and Express.
* Understand the benefits of streaming and how to use it with Fastify and Express.
* Generate dynamic content with template engines in Fastify and Express.

# Serving Static Content with Fastify (1)

In the previous chapter, we created a Fastify server in the Creating a Web Server with Fastify section. In this section, we're going to work on the same code base as we left off in the last chapter.

Currently the **package.json** of our code looks as follows:

**{  
  "name": "bah",  
  "version": "1.0.0",  
  "description": "This project was bootstrapped with Fastify-CLI.",  
  "main": "app.js",  
  "directories": {  
    "test": "test"  
  },  
  "scripts": {  
    "test": "tap \"test/\*\*/\*.test.js\"",  
    "start": "fastify start -l info app.js",  
    "dev": "fastify start -w -l info -P app.js"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "dependencies": {  
    "fastify": "^3.0.0",  
    "fastify-autoload": "^3.10.0",  
    "fastify-cli": "^2.15.0",  
    "fastify-plugin": "^3.0.0",  
    "fastify-sensible": "^3.1.2"  
  },  
  "devDependencies": {  
    "tap": "^15.1.6"  
  }  
}**

We need to add a new Fastify plugin that will handle static content for us. Making sure that our current working directory is the **fastify-web-server** folder we created in the previous chapter let's run the following command:

**npm install --save-dev fastify-static**

This will automatically update the **devDependencies** section of our **package.json** to look as follows:

**"devDependencies": {  
    "fastify-static": "^4.5.0",  
    "tap": "^15.1.6"  
  }**

We've deliberately installed **fastify-static** as a development dependency. It's generally bad practice to use Node.js for static file hosting in production. We need to think of this as a local development convenience only in most cases so we're going to apply constraints to ensure this isn't used in production.

The **app.js** file currently looks as follows (any comments are removed):

**'use strict'**

**const path = require('path')  
const AutoLoad = require('fastify-autoload')**

**module.exports = async function (fastify, opts) {**

**fastify.register(AutoLoad, {  
    dir: path.join(\_\_dirname, 'plugins'),  
    options: Object.assign({}, opts)  
  })**

**fastify.register(AutoLoad, {  
    dir: path.join(\_\_dirname, 'routes'),  
    options: Object.assign({}, opts)  
  })**

**fastify.setNotFoundHandler((request, reply) => {  
    if (request.method !== 'GET') {  
      reply.status(405)  
      return 'Method Not Allowed\n'  
    }  
    return 'Not Found\n'  
  })**

**}**

We need to register and configure **fastify-static** but not in production. Let's make our **app.js** look as follows:

**'use strict'**

**const path = require('path')  
const AutoLoad = require('fastify-autoload')**

**const dev = process.env.NODE\_ENV !== 'production'**

**const fastifyStatic = dev && require('fastify-static')**

**module.exports = async function (fastify, opts) {  
  if (dev) {  
    fastify.register(fastifyStatic, {  
      root: path.join(\_\_dirname, 'public')  
    })  
  }  
  fastify.register(AutoLoad, {  
    dir: path.join(\_\_dirname, 'plugins'),  
    options: Object.assign({}, opts)  
  })**

**fastify.register(AutoLoad, {  
    dir: path.join(\_\_dirname, 'routes'),  
    options: Object.assign({}, opts)  
  })  
  fastify.setNotFoundHandler((request, reply) => {  
    if (request.method !== 'GET') {  
      reply.status(405)  
      return 'Method Not Allowed\n'  
    }  
    return 'Not Found\n'  
  })  
}**

Cont'd on the next page.

**Serving Static Content with Fastify (2)**

It is a typical convention when deploying an application to set an environment variable called **NODE\_ENV** to **'production'**. This would be managed by deployment infrastructure and is outside of scope here, other than to acknowledge that it's the convention most often used to determine whether a Node.js process is running in development mode or is deployed to production (or staging). By checking that **NODE\_ENV** is not set to **production** we assume development mode, which makes our **dev** constant **true**.

We conditionally load **fastify-static** into the process if **dev** is **true**. Since **fastify-static** is a development dependency, if we didn't do this the server would throw on initialization due to an attempt to load a missing dependency.

Within the root plugin (the exported async function of **app.js**), we also conditionally register **fastify-static** with the **fastify.register** method. The first argument passed to **fastify.register** is the **fastify-static** plugin (**fastifyStatic**). The second argument is the options for the plugin. We set the **root** option to point to a folder named **public** in our project dir. This instructs **fastify-static** to only serve files from that folder, and not allow any files above that folder to be accessible.

We'll need to create this **public** folder next. Making sure that **fastify-web-server** is our current working directory, let's run the following in the terminal:

**node -e "fs.mkdirSync('public')"  
cd public  
node -e "fs.openSync('index.html', 'w')"  
node -e "fs.openSync('hello.html', 'w')"  
cd ..**

We'll also be replacing our routes with static HTML so let's delete **routes/hello.js** and **routes/root.js**:

**cd routes  
node -e "fs.unlinkSync('root.js')"  
node -e "fs.rmdirSync('hello', {recursive: true})"  
cd ..**

The project file and folder structure should now be as follows:

* **.gitignore**
* **app.js**
* **public/index.html**
* **public/hello.html**
* **routes/README.md**
* **plugins/README.md**
* **plugins/support.js**
* **test/helper.js**
* **test/routes/example.test.js**
* **test/plugins/support.test.js**
* **test/routes/root.test.js**

Our final step is to add the contents of the **index.html** and **hello.html** files in the **public** folder.

The **index.html** file should contain the following content:

**<html>  
<head>  
  <style>  
   body { background: #333; margin: 1.25rem }  
   a { color: yellow; font-size: 2rem; font-family: sans-serif }  
  </style>  
</head>  
<body>  
  <a href='/hello.html'>Hello</a>  
</body>  
</html>**

Note that the contents of **index.html** differ from the string of HTML in our root route from the previous section in one key place: the anchor link (**<a/>**) points to **/hello.html** instead of **/hello**.

The **hello.html** file should contain the following content:

**<html>  
  <head>  
    <style>  
     body { background: #333; margin: 1.25rem }  
     h1 { color: #EEE; font-family: sans-serif }  
    </style>  
  </head>  
  <body>  
    <h1>Hello World</h1>  
  </body>  
</html>**

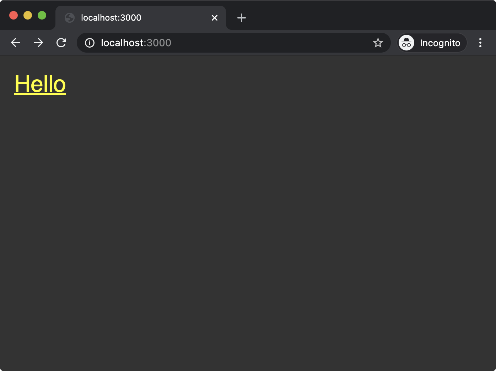
Now we can start our server:

**npm run dev**

*Cont'd on the next page.*

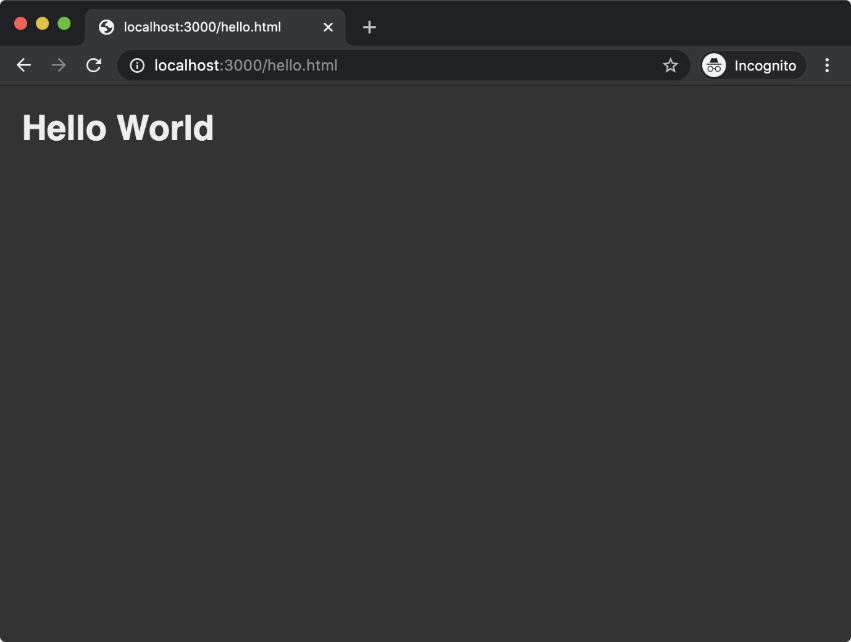
# Serving Static Content with Fastify (3)

If we navigate in a browser to http://localhost:3000 we should see something like the following:

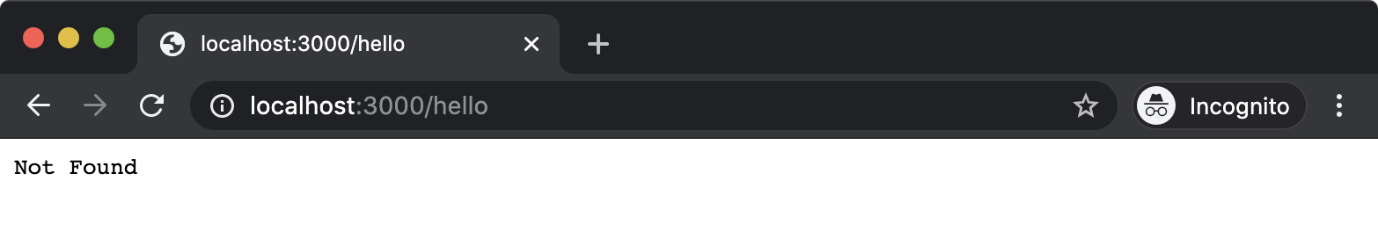


This root route delivers the same HTML as in the previous chapter, the difference is that **fastify-static** is loading **public/index.html** instead of us defining a route and manually sending the content. If we navigate to http://localhost:3000/index.html we'll get the same outcome because the root route (**/**) is special-cased to load an **index.html** file.

If we click the link we'll navigate to http://localhost:3000/hello.html, which should result in something similar to the following:



Note that the path of the URL is **/hello.html** whereas previously it was **/hello**. This is because **fastify-static** is loading **hello.html** from the **public** folder whereas before we had defined a **/hello** route which responded with a string of HTML. If we attempt to load http://localhost:3000/hello we would see our 404 Not Found page:



The **fastify-static** module also decorates the **reply** object with **sendFile** method. We can use this to create a route that manually responds with the contents of **hello.html** if we wanted to alias **/hello.html** to **/hello**.

Let's finish this section off by doing just that. We need to recreate the **hello** folder in the **routes** directory with an **index.js** file. Let's run the following commands to create the desired structure:

**cd routes  
node -e "fs.mkdirSync('hello')"  
cd hello  
node -e "fs.openSync('index.js', 'w')"  
cd ..  
cd ..**

Let's write the following code into **routes/hello/index.js**:

**'use strict'**

**module.exports = async (fastify, opts) => {  
  fastify.get('/', async (request, reply) => {  
    return reply.sendFile('hello.html')  
  })  
}**

Adding **routes/hello/index.js** automatically causes **fastify-autoload** to mount any routes registered in that file at the **/hello** URL path. We register a **GET /** route (which is therefore the **/hello** route since the route path is set by the folder path) and we call **reply.sendFile('hello.html')**. This causes **fastify-static** to respond to the request with contents of the **public/hello.html** file. The **sendFile** method knows how to load the **hello.html** file from the **public** folder because we configure the **root** option passed to **fastifyStatic** in **app.js** to point to the **public** folder. If we restart the server (**npm run dev**) and navigate to http://localhost:3000/hello, we should now see the same page as http://localhost:3000/hello.html.

# Using Templates with Fastify (1)

While the primary and original focus of Fastify was for building data services, view rendering capability is available with a little bit of set up.

Building off our example Fastify server in the prior section, let's add template rendering for dynamic content generation.

In the terminal, with **fastify-web-server** as the current working directory let's run the following command in order to install a template engine and Fastify's view rendering plugin:

**npm install point-of-view handlebars**

Handlebars is one of the template engines that **point-of-view** supports. See more about Handlebars at <http://handlebarsjs.com>.

Now we can set up and configure view rendering, we need to modify our **app.js** file to look as follows:

**'use strict'**

**const path = require('path')  
const AutoLoad = require('fastify-autoload')**

**const pointOfView = require('point-of-view')  
const handlebars = require('handlebars')**

**module.exports = async function (fastify, opts) {**

**fastify.register(pointOfView, {  
    engine: { handlebars },  
    root: path.join(\_\_dirname, 'views'),  
    layout: 'layout.hbs'  
  })**

**fastify.register(AutoLoad, {  
    dir: path.join(\_\_dirname, 'plugins'),  
    options: Object.assign({}, opts)  
  })**

**fastify.register(AutoLoad, {  
    dir: path.join(\_\_dirname, 'routes'),  
    options: Object.assign({}, opts)  
  })  
  fastify.setNotFoundHandler((request, reply) => {  
    if (request.method !== 'GET') {  
      reply.status(405)  
      return 'Method Not Allowed\n'  
    }  
    return 'Not Found\n'  
  })**

**}**

We've removed **fastify-static** which we introduced in the prior section, and with it the **dev** constant which we won't need for this case because our server will now be performing on-the-fly dynamic rendering.

We loaded the two modules that we installed and then at the top of the exported async function we use **fastify.register** to register **point-of-view** (referenced as **pointOfView**). In the options object passed to **fastify.register** we set the **engine** to **handlebars**. Note that the **engine** option expects an option with the key being the name of the engine and the value being the engine library itself. We used a shorthand property **{ handlebars }**, which creates an object with the shape **{ handlebars: handlebars }**. We set the **root** option to **path.join(\_\_dirname, 'views')**; we'll be creating a **views** folder in the project folder shortly. We've also set a **layout** option assigned to **layout.hbs**, we're also going to create a layout template in the **views** folder.

Let's create a **views** folder by running the following command:

**node -e "fs.mkdirSync('views')"**

We'll also create three files in the **views** folder: **index.hbs**, **hello.hbs**, and **layout.hbs**:

**cd views  
node -e "fs.openSync('index.hbs', 'w')"  
node -e "fs.openSync('hello.hbs', 'w')"  
node -e "fs.openSync('layout.hbs', 'w')"  
cd ..**

We'll also delete the **public** folder since we won't be using that any more:

**node -e "fs.rmdirSync('public', {recursive: true})"**

The **views/layout.hbs** file should contain the following:

**<html>  
  <head>  
    <style>  
     body { background: #333; margin: 1.25rem }  
     h1 { color: #EEE; font-family: sans-serif }  
     a { color: yellow; font-size: 2rem; font-family: sans-serif }  
    </style>  
  </head>  
  <body>  
    {{{ body }}}  
  </body>  
</html>**

Cont'd on the next page.

# Using Templates with Fastify (2)

We've mixed the styles together from the **index.html** and **hello.html** files we created in the previous section and we interpolate a special template local called **body** inside of the **<body>** opening and closing tags. Using three braces to denote an interpolation point is Handlebars syntax that instructs the template engine to conduct raw interpolation. In other words, if the **body** template local contains HTML syntax the content will not be escaped whereas using two braces would cause HTML syntax to be escaped (for instance **<** would be escaped to **&lt;**). This should never be used when interpolating (uncleaned) user input into templates but when building a layout we need to inject raw HTML. The **body** local is created automatically by **point-of-view** when rendering a view because we specified the **layout** option.

In **views/index.hbs** we'll add the following content:

**<a href='/hello'>Hello</a><br>  
<a href='/hello?greeting=Ahoy'>Ahoy</a>**

The **views/hello.hbs** file should contain the following:

**<h1>{{ greeting }} World</h1>**

Finally, we need to set up our routes to render our views. Let's recreate the **routes/root.js** that we removed in the last section:

**cd routes  
node -e "fs.openSync('root.js', 'w')"**

The **routes/root.js** file should contain the following:

**'use strict'**

**module.exports = async (fastify, opts) => {  
  fastify.get('/', async (request, reply) => {  
    return reply.view('index.hbs')  
  })  
}**

The **point-of-view** plugin that we registered in **app.js** decorated the **reply** instance with a **view** method. When we registered **point-of-view**, we set the **root** option to the **views** folder. Therefore, when we pass **'index.hbs'** to **reply.view** it knows to look for **index.hbs** in the **view** folder. Similarly, the **layout** option that we set to **'layout.hbs'** indicates to **point-of-view** that the layout template can be found in **views/layout.hbs**. So when we use **reply.view** here **point-of-view** first renders both the **views/index.hbs** file and then interpolates the rendered output into **views/layout.hbs** and sends the final rendered output of both files combined as the response. The return value of the **reply.view** method must be returned from the async function passed as the route handler so that Fastify knows when the route handler has finished processing the request.

For our hello pages, the **routes/hello/index.js** file should be updated to contain the following:

**'use strict'**

**module.exports = async (fastify, opts) => {  
  fastify.get('/', async (request, reply) => {  
    const { greeting = 'Hello '} = request.query  
    return reply.view(`hello.hbs`, { greeting })  
  })  
}**

The **reply.view** method can take a second parameter, an object which sets the values of the template locals. Recall that **views/hello.hbs** contains a **greeting** template local, we pass an object with a property called **greeting** and a value defaulting to **'Hello'** or else the value of a URL query string key named **greeting**. For instance, a request to /hello?greeting=Ahoy would result in the **greeting** constant being set to **'Ahoy'** for that request and so the object passed as the second argument to **reply.view** would contain a property named **greeting** with a value of **'Ahoy'**. This in turn would make **reply.view** render **views/hello.hbs** with **Ahoy World** text instead of **Hello World**.

For the purposes of understanding we're using a query string key-value and sending it back to the client as content. As discussed, the template engine will automatically clean the input (because we interpolate **greeting** with just two braces) however always exercise caution when handling user input. If there's another way to achieve a goal without directly reflecting content back to the client that is a more secure approach. See ["Cross Site Scripting (XSS)"](https://owasp.org/www-community/attacks/xss/) by OWASP for more information.

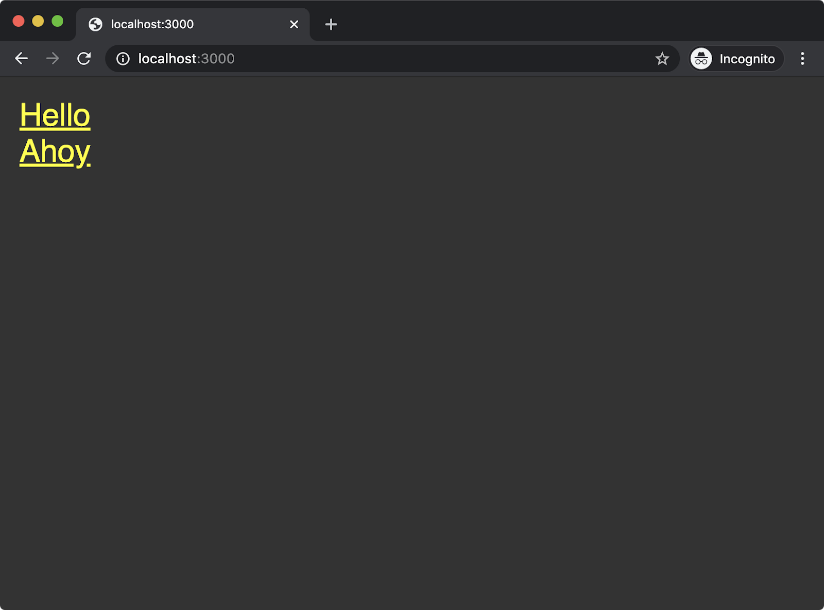
Cont'd on the next page.

# Using Templates with Fastify (3)

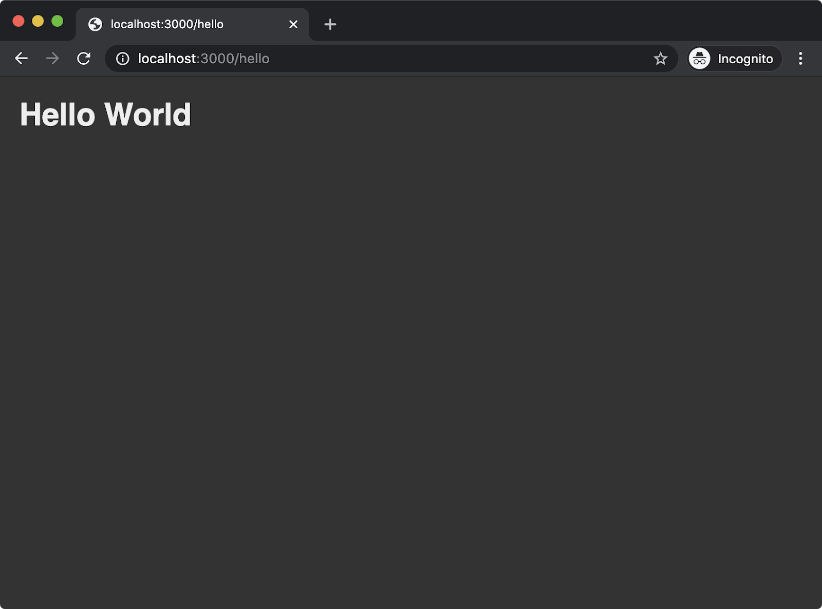
Let's try it out. If everything went according to plan we should be able to run the following to successfully start the server:

**npm run dev**

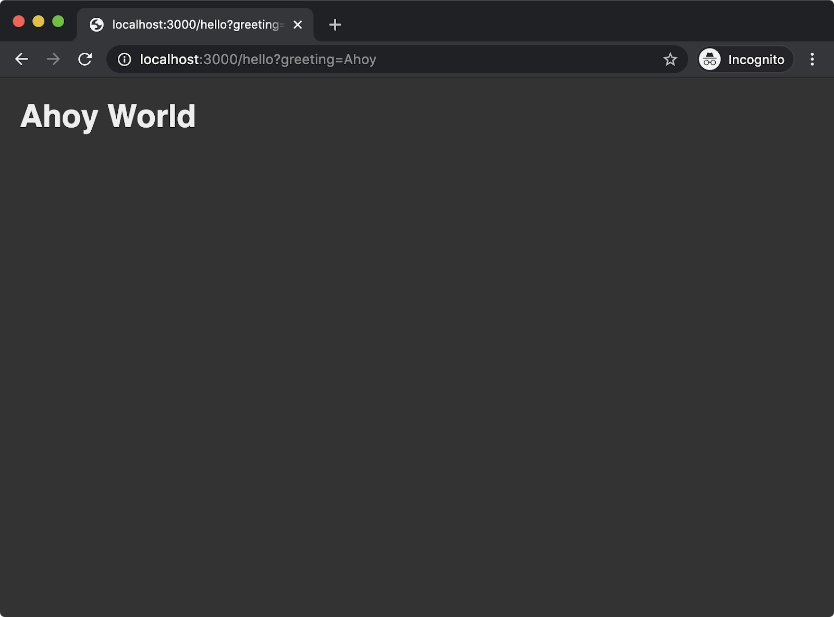
Once the server has started, we should be able to navigate to http://localhost:3000 and see something similar to the following:



Clicking the **Hello** link should take us to the following screen:



Going back to http://localhost:3000 and clicking the **Ahoy** link should display the following:



We've now served dynamic content with Fastify. For more details on the **point-of-view** plugin see <https://github.com/fastify/point-of-view>.

**Serving Static Content and Using Templates with Express (1)**

The original focus of the Fastify framework was on building RESTful JSON services, whereas Express is more geared towards template rendering (and static serving static content). Therefore Express has these pieces built into its core whereas in Fastify template rendering is an add-on.

In this section we'll use the **express-generator** command-line utility to generate a new project with view rendering and static asset serving preconfigured.

Let's install **express-generator**:

**npm install -g express-generator@4**

This will install a **globally** available command-line executable named **express**. We can use this executable to generate an express project.

Let's run the following command:

**express --hbs express-web-server**

This will generate a folder named **express-web-server** with an Express project that is set up to render Handlebar templates from the **views** folder.

Let's change our working directory to **express-web-server** and install the project dependencies:

**cd express-web-server**  
**npm install**

The **express-generator** generated the following files and folders:

* **app.js**
* **package.json**
* **routes/index.js**
* **routes/users.js**
* **public/images**
* **public/javascripts**
* **public/stylesheets/style.css**
* **views/error.hbs**
* **views/index.hbs**
* **views/layout.hbs**

Let's take a look at the top 20 lines of the **app.js** file:

**var createError = require('http-errors');  
var express = require('express');  
var path = require('path');  
var cookieParser = require('cookie-parser');  
var logger = require('morgan');**

**var indexRouter = require('./routes/index');  
var usersRouter = require('./routes/users');**

**var app = express();**

**// view engine setup  
app.set('views', path.join(\_\_dirname, 'views'));  
app.set('view engine', 'hbs');**

**app.use(logger('dev'));  
app.use(express.json());  
app.use(express.urlencoded({ extended: false }));  
app.use(cookieParser());  
app.use(express.static(path.join(\_\_dirname, 'public')));**

On the last line of this snippet from **app.js** we can see that the **express.static** method is called, being passed the directory path to the **public** folder. The result of this is immediately passed to **app.use**. So the **express** instance has a method named **static** which returns Express middleware that will serve requests that match up with any files in the **public** folder. Notably this will serve files both in development and production environments which is recommended against. To reinforce this point, let's alter the last line in our snippet from **app.js** to the following:

**if (process.env.NODE\_ENV !== 'production') {  
  app.use(express.static(path.join(\_\_dirname, 'public')));  
}**

Now static hosting will only occur in development and production static hosting is left as a deployment infrastructure problem.

Around the middle of our code snippet from **app.js** we can see the view engine configuration:

**// view engine setup  
app.set('views', path.join(\_\_dirname, 'views'));  
app.set('view engine', 'hbs');**

In Express the **app.set** method can be used to store state as key-values. In this specific case the **'views'** key and the **'view engine'** key are both special-cased key names that instruct Express to load views from a particular path and use a particular view engine respectively.

We already have a **views/index.hbs** and a **views/layout.hbs**, we just need to create a **views/hello.hbs** file. Let's create it now:

**cd views  
node -e "fs.openSync('hello.hbs', 'w')"  
cd ..**

*Cont'd on the next page.*

# Serving Static Content and Using Templates with Express (2)

Let's alter the **views/layout.hbs** file to contain the following:

**<html>  
  <head>  
    <style>  
     body { background: #333; margin: 1.25rem }  
     h1 { color: #EEE; font-family: sans-serif }  
     a { color: yellow; font-size: 2rem; font-family: sans-serif }  
    </style>  
  </head>  
  <body>  
    {{{ body }}}  
  </body>  
</html>**

The **views/index.hbs** should be altered to contain the following code:

**<a href='/hello'>Hello</a><br>  
<a href='/hello?greeting=Ahoy'>Ahoy</a>**

The **views/hello.hbs** file should contain the following:

**<h1>{{ greeting }} World</h1>**

Finally, we need to configure our routes. Let's rename **routes/users.js** to **routes/hello.js**:

**cd routes  
node -e "fs.renameSync('users.js', 'hello.js')"  
cd ..**

The **routes/index.js** file should have the following content:

**var express = require('express');  
var router = express.Router();**

**/\* GET home page. \*/  
router.get('/', function(req, res, next) {  
  res.render('index');  
});**

**module.exports = router;**

This is exactly as it was when generated by **express-generator** except we've removed the template locals object **({ title: 'Express' })** from being passed as the second argument to the **res.render** invocation. Express has **res.render** built-in to its core and it works in essentially the same way as **reply.view** added by the **point-of-view** plugin when registered in a Fastify server - although at the time of writing Express v4 renders at about half the speed of Fastify's **point-of-view** in production.

The **routes/hello.js** file should have the following content:

**var express = require('express');  
var router = express.Router();**

**router.get('/', function(req, res, next) {  
  var greeting = 'greeting' in req.query ?  
    req.query.greeting :  
    'Hello';  
  res.render('hello', { greeting: greeting });  
});**

**module.exports = router;**

Before we can start the server we now need to go back to **app.js** and configure the application to use **routes/hello.js** and to stop using **routes/users.js** which no longer exists.

Underneath the registration of **express.static** middleware we can find the routing setup:

**app.use('/', indexRouter);  
app.use('/users', userRouter);**

Let's change the **/users** mount point to **/hello** and rename **userRouter** to **helloRouter**. Those two lines should look as follows:

**app.use('/', indexRouter);  
app.use('/hello', helloRouter);**

Cont'd on the next page.

# Serving Static Content and Using Templates with Express (3)

At the top of **app.js** the last two **require** statements are:

**var indexRouter = require('./routes/index');  
var userRouter = require('./routes/user');**

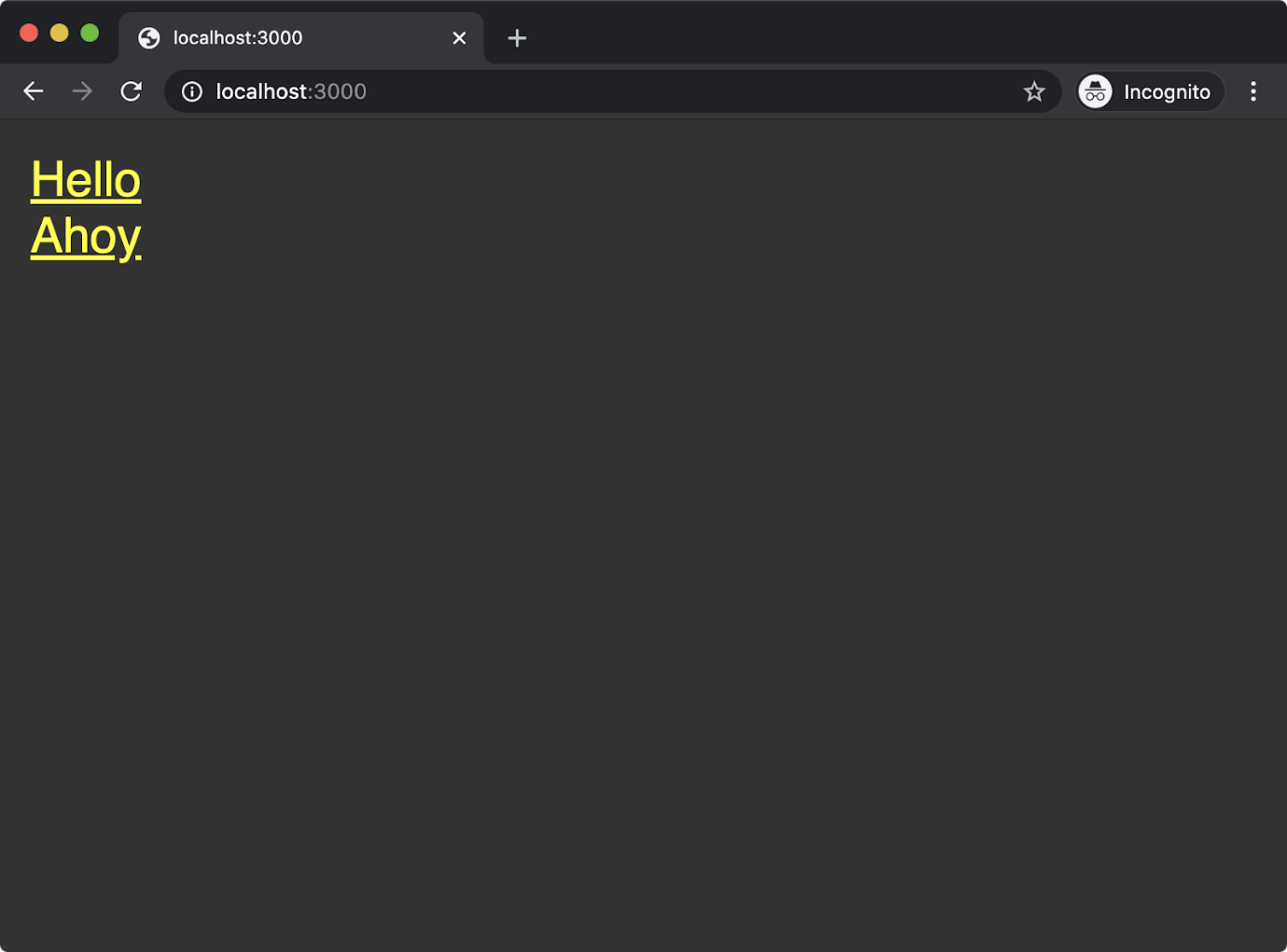
We also need to update that to load **routes/hello.js** and update the variable name. Those two lines should look as follows:

**var indexRouter = require('./routes/index');  
var helloRouter = require('./routes/hello');**

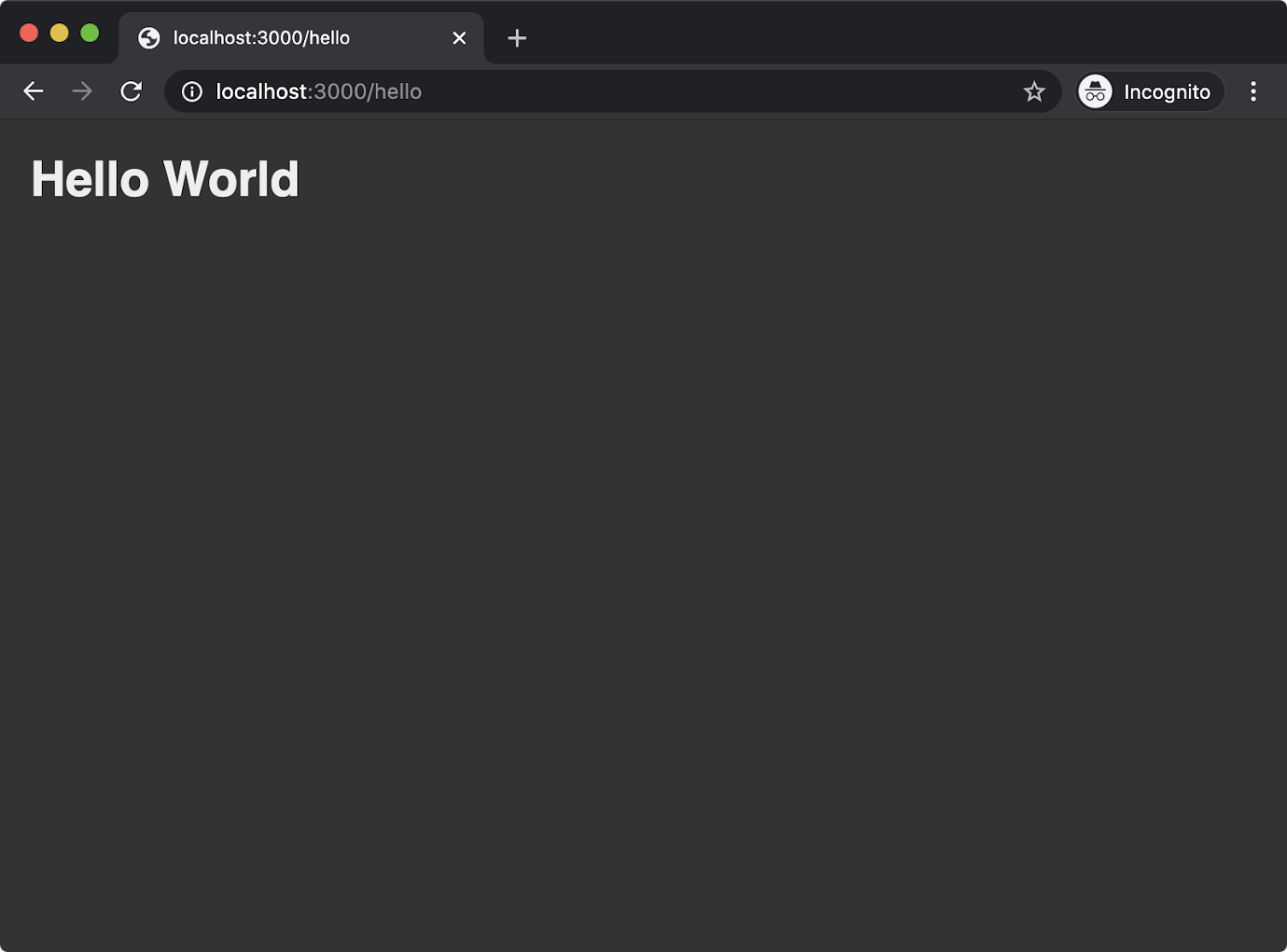
If everything was completed correctly, we should now be able to start our Express server with the following command:

**npm start**

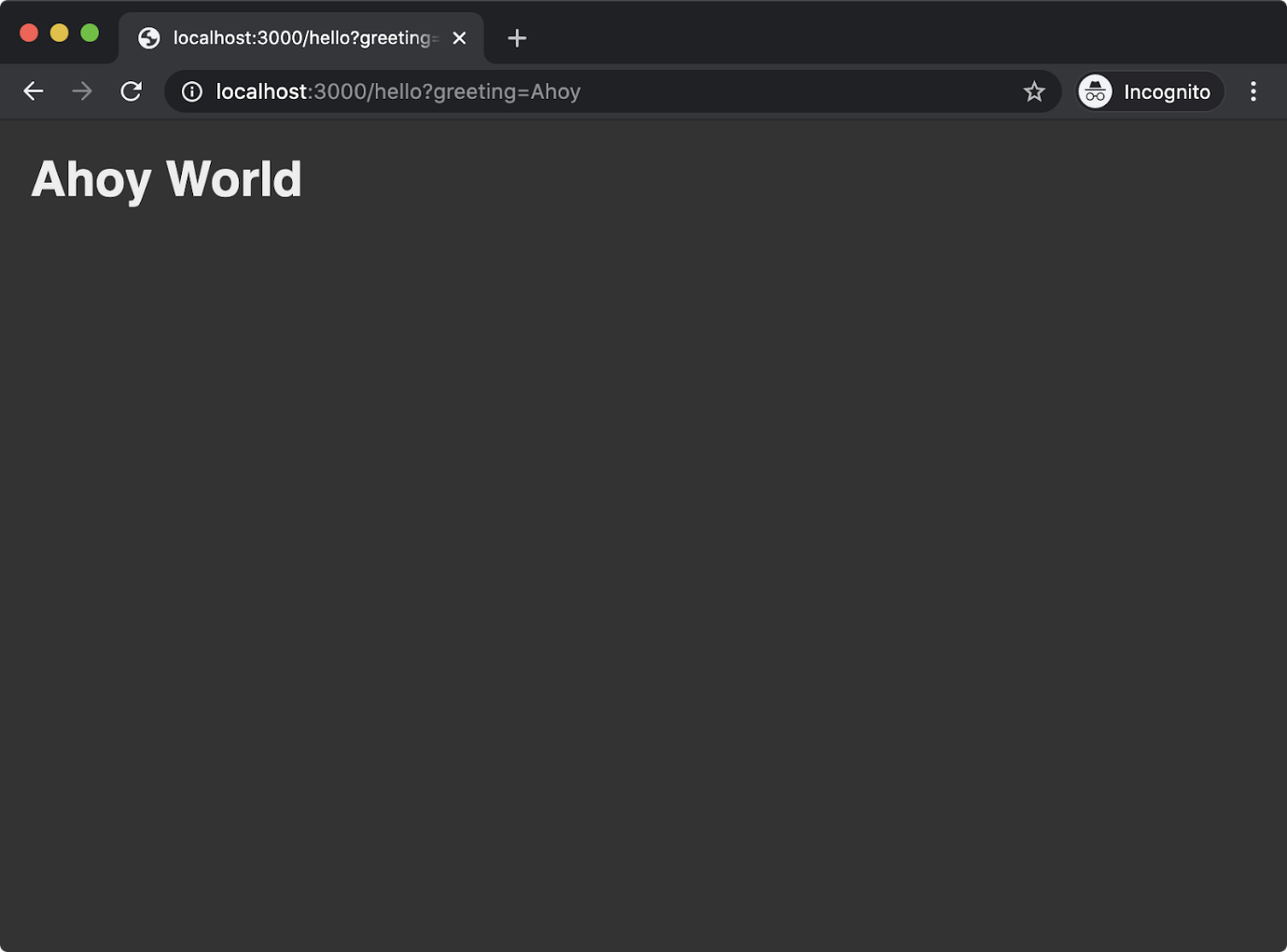
This will start the process and keep it open. Navigating to ht‌tp://localhost:3000 in the browser should show the following:



Clicking the **Hello** link should take us to the following screen:



Going back to ht‌tp://localhost:3000 and clicking the **Ahoy** link should display the following:



We've now achieved development-only static-hosting and served dynamic content with Express.

# Streaming Content with Fastify (1)

The HTTP specification has a header called [Transfer-Encoding](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Transfer-Encoding) which can be set to chunked. This means that chunks of data can be sent over HTTP and in many cases browser-clients can begin parsing immediately. Node.js Streams also allow for chunked reading, processing and writing of data. This affinity between Node.js Streams means we can serve content in a highly efficient way: instead of waiting for the server to prepare and process all data and then sending the response, the client can begin parsing some HTML (or theoretically any structured data) we've sent before the server has even finished preparing it for sending.

For this example we'll be using a package that provides a stream of Hacker News content, called **hn-latest-stream**.

Let's work in our **fastify-web-server** folder from where we left off in the "Using Templates with Fastify" section.

Making sure that **fastify-web-server** is the current working directory, run the following command to install **hn-latest-stream**:

**npm install hn-latest-stream**

Now we'll create a new routes folder called **routes/articles** and add an **index.js** file to it:

**cd routes  
node -e "fs.mkdirSync('articles')"  
cd articles  
node -e "fs.openSync('index.js', 'w')"  
cd ..  
cd ..**

The contents of **routes/articles/index.js** should be as follows:

**'use strict'**

**const hnLatestStream = require('hn-latest-stream')**

**module.exports = async (fastify, opts) => {  
  fastify.get('/', async (request, reply) => {  
    const { amount = 10, type = 'html' } = request.query**

**if (type === 'html') reply.type('text/html')  
    if (type === 'json') reply.type('application/json')  
    return hnLatestStream(amount, type)  
  })  
}**

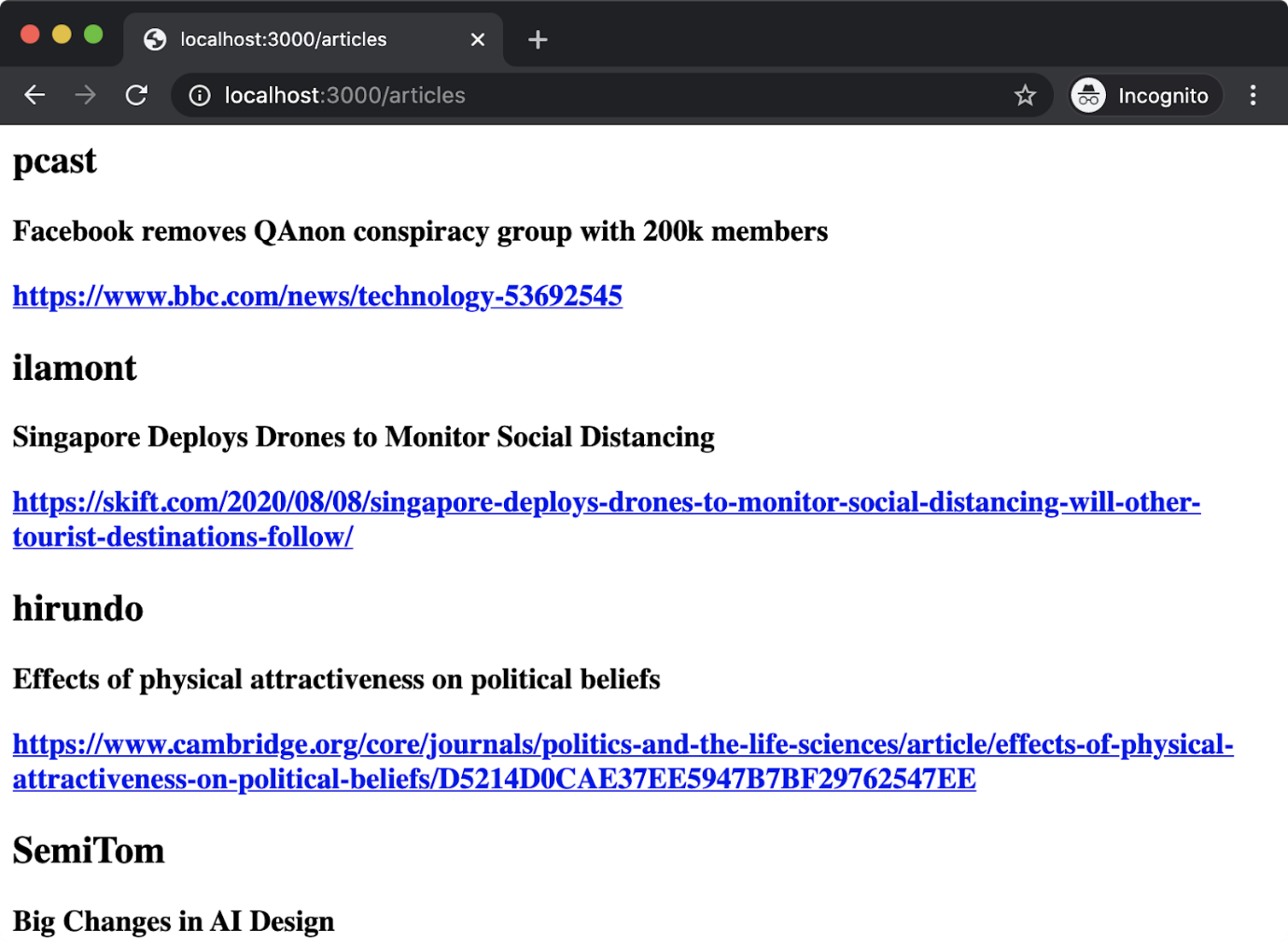
The **hnLatestStream** function accepts two parameters, **amount** and **type** and then returns a Node.js Stream. The **amount** is the number of most recent Hacker News articles we want to load and the **type** describes whether the stream should send HTML chunks or JSON chunks. In our route handler, we set a default amount of 10 and a default type of `'html'` while also allowing these to be overridden by query string arguments. Depending on the desired type we also use the **reply.type** method to set the correct **Content-Type HTTP** header for the content.

Returning the stream (the result of calling **hnLatestStream**) from the route handler instructs Fastify to safely pipe the stream to the response. The **reply.send** method can also be passed a stream and Fastify behaves in the same way - by piping the stream as the HTTP response.

Let's see it in action. First we need to start the server with the following command:

**npm run dev**

Next we can navigate to ht‌tp://localhost:3000/articles and we should see output similar to but not the same as, the following:

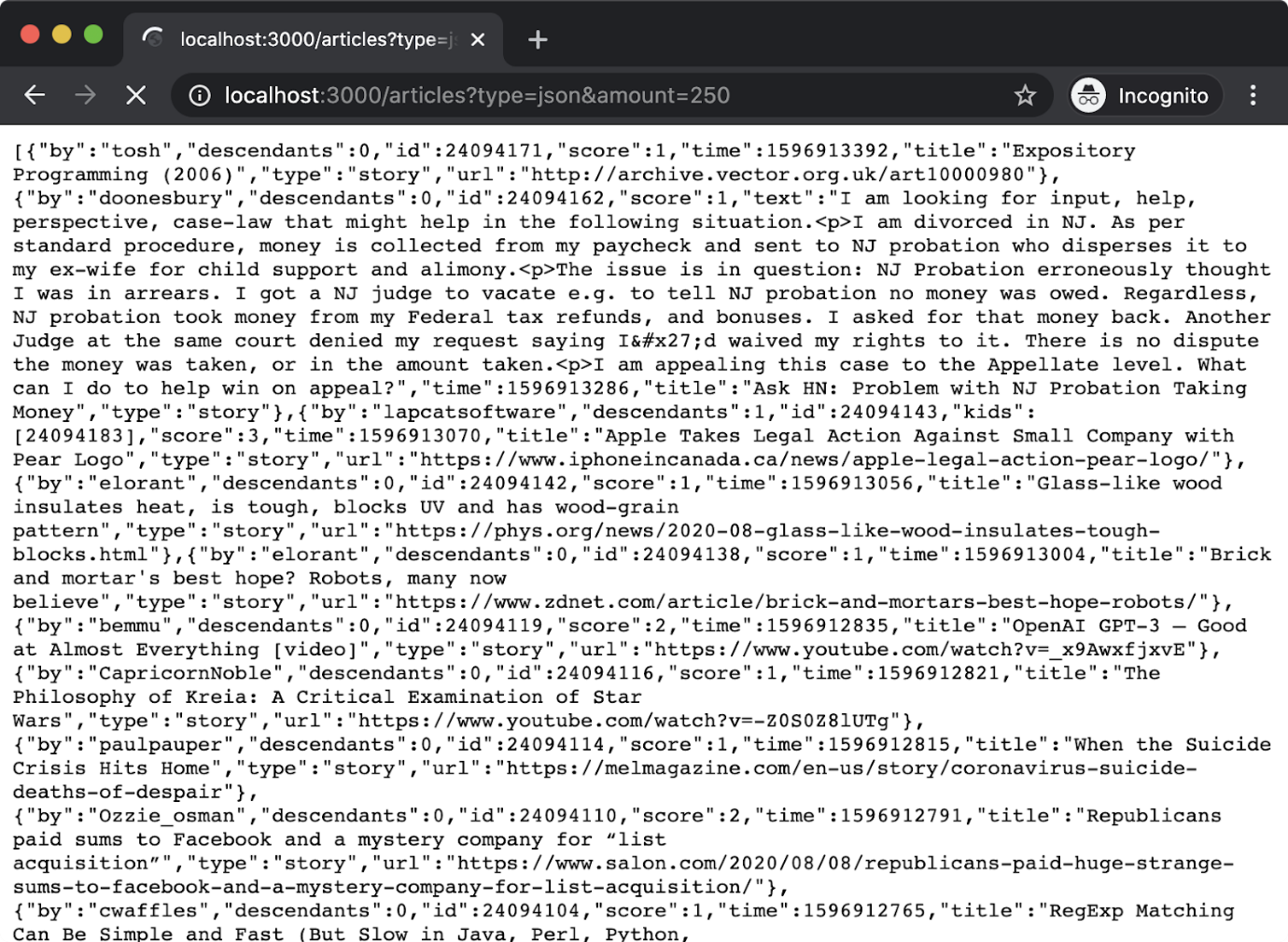


Cont'd on the next page.

# Streaming Content with Fastify (2)

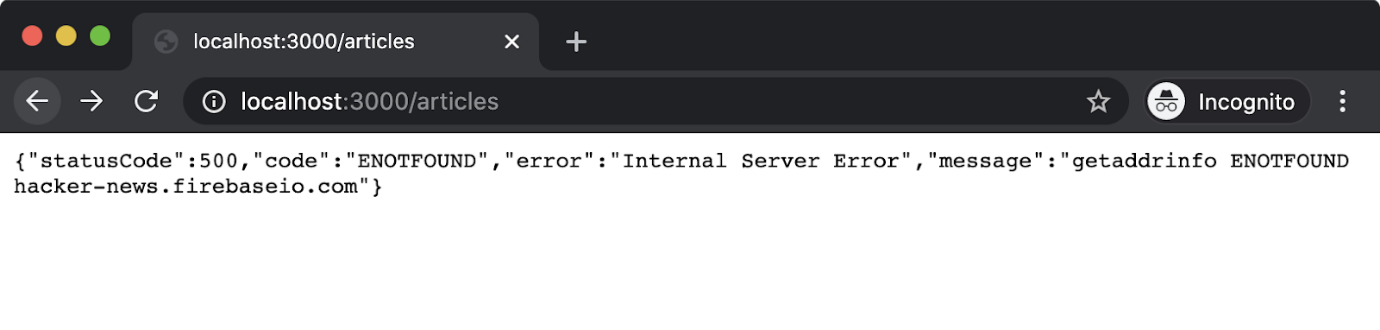
This will load different articles each time and there should be ten articles in total. The **hn-latest-stream** module uses the Hacker News API to fetch the content. It has to first lookup the latest story IDs and then for each ID it has to make a separate HTTP request to fetch the article and then push either JSON or HTML content to the stream that it returns. As such, it should be easy to observe the content being parsed and rendered by the browser incrementally in that there's a visible delay between each article rendering in the browser. This shows the power of streams in action for long running tasks. The server hasn't retrieved all the data yet, but we can still fill the above-the-fold (the part of the page that's first seen when a page loads) with the latest articles while more articles continue to load on the server, and then sent to the client to be displayed beneath the fold.

Let's try out the query string parameters as well. In the browser let's try navigating to the URL: http://localhost:3000/articles?type=json&amount=250. This will load the JSON data for the latest 250 Hacker News stories. We should again be able to observe short delays between each JSON object being received by the browser. When navigating to this URL we should see something similar to the following, but with different content:



In the next chapter, we'll look at creating JSON services in more detail, this section was more to illustrate how streams can be a useful tool in constrained scenarios and how to use streams with the Fastify framework.

One final note about error handling before wrapping up. Due to Fastify handling the stream for us, any errors in the stream will be handled and propagated. If we disconnect from the Internet and then attempt to access http://localhost:3000/articles, we'll see something like the following:



We caused an error in the stream, which Fastify then handled and responded to our request with a 500 status code along with information about the error.

In the next and final section, we'll discuss how to use streams with Express, which will require more glue around error handling.

Node.js Streams is a large topic. A chapter is dedicated to streams in the course for the companion certification to Node.js Services Development - the [Node.js Application Development (LFW211)](https://training.linuxfoundation.org/training/nodejs-application-development-lfw211/) course.

# Streaming Content with Express (1)

Express does not have native support for streams in the same way that Fastify does. However Express decorates the native HTTP **OutgoingMessage** object (the response object, **res**). The **res** object is a writable stream, so this means we can essentially roll our own streaming response. However there are some complexities around this. For instance, when one stream is piped to another it will automatically end the destination stream when the source stream has ended. But in the case of an error, we don't want to end the response, we need to instead propagate the error back into Express so it can handle it centrally according to the configuration in **app.js**.

Making sure that our current working directory is the **express-web-server** folder as we left in the "Serving Static Content and Using Templates with Express" section, let's install **hn-latest-stream**:

**npm install hn-latest-stream**

Now, we'll create a new route:

**cd routes  
node -e "fs.openSync('articles.js', 'w')"**

We need to register the router we'll be creating for **routes/articles.js** in the **app.js** file.

Near the top of the file there's currently two routers imported:

**var indexRouter = require('./routes/index');  
var helloRouter = require('./routes/hello');**

We need to add an **articlesRouter** to this, that small section of **app.js** should be updated to look as follows:

**var indexRouter = require('./routes/index');  
var helloRouter = require('./routes/hello');  
var articlesRouter = require('./routes/articles');**

Around the middle of **app.js** the routers are registered. This currently looks as follows:

**app.use('/', indexRouter);  
app.use('/hello', helloRouter);**

We need to add the **articlesRouter** mounted at the **/articles** route. That section should be updated to look like so:

**app.use('/', indexRouter);  
app.use('/hello', helloRouter);  
app.use('/articles', articlesRouter);**

The content of **routes/articles.js** should be as follows:

**var express = require('express');  
var router = express.Router();  
var hnLatestStream = require('hn-latest-stream')  
var finished = require('stream').finished**

**router.get('/', function(req, res, next) {  
  const { amount = 10, type = 'html' } = req.query**

**if (type === 'html') res.type('text/html')  
  if (type === 'json') res.type('application/json')**

**const stream = hnLatestStream(amount, type)**

**stream.pipe(res, {end: false})**

**finished(stream, (err) => {  
    if (err) {  
      next(err)  
      return  
    }  
    res.end()  
  })**

**});**

**module.exports = router;**

The logic of the router handler is the same as in the previous section. We have a default **amount** of 10 and a default **type** of **'html'** although this can be overridden by the query string of the incoming request URL. We also set the **Content-Type** HTTP header based on the **type**.

From there the code is slightly more complicated because we have to manually propagate errors and handle the dance of sending data without ending the response too soon.

The **stream.pipe(res, {end: false})** line tells the **stream** (our Hacker News stream) to write all data it receives to the **res** object (which is also a stream). The second parameter, an object with a property named **end** set to **false** prevents pipe from performing its default behavior of endings the destination stream (**res**) when the source stream (**stream**) has ended. This is important because without this, if there is an error in the source stream then **res** will be ended before our server can send an appropriate error response.

Cont'd on the next page.

# Streaming Content with Express (2)

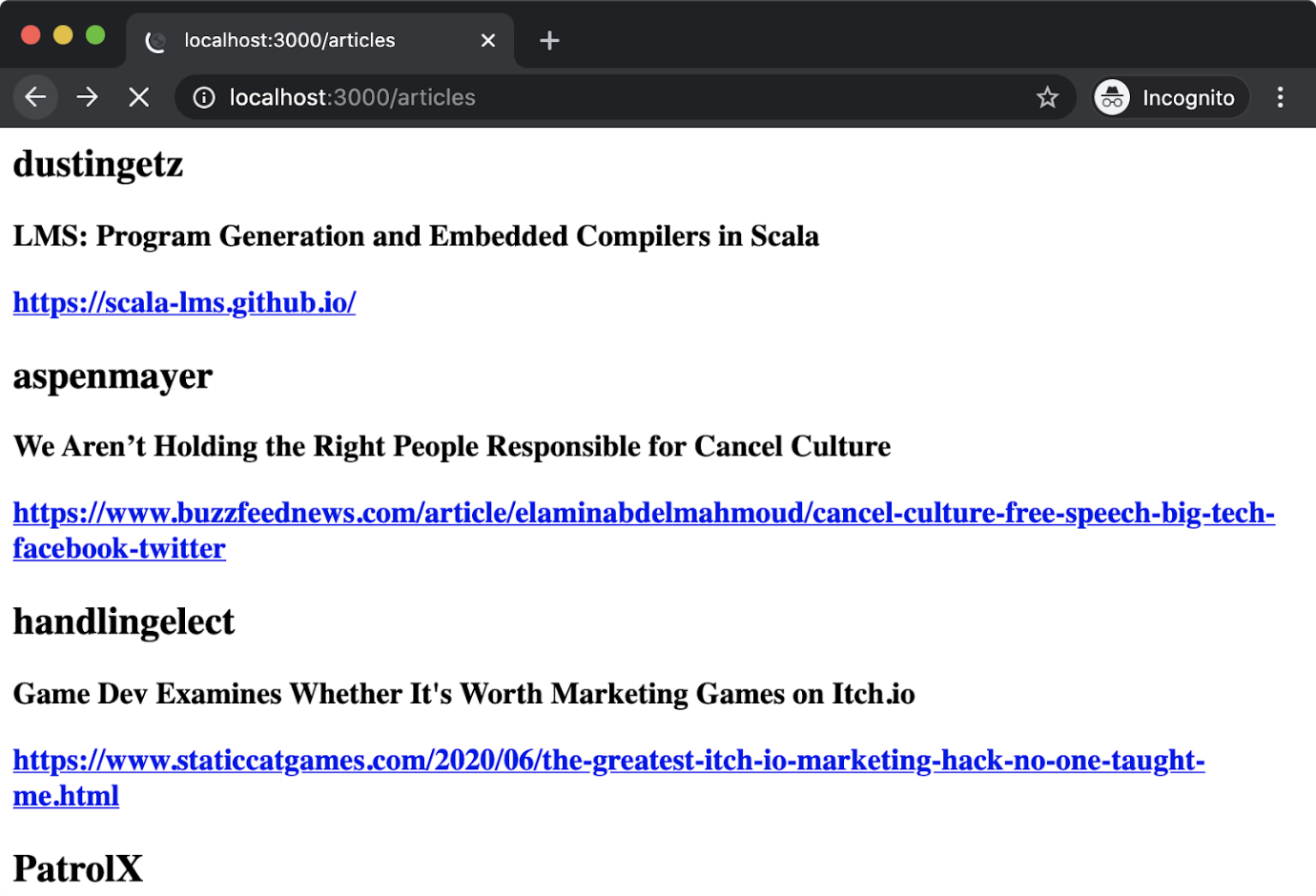
We use the **finished** function from the core **stream** module to determine when our Hacker News stream (**stream**) has ended. Using **finished** saves a lot of code, otherwise we would have to listen to the **end**, **close** and **error** events (plus handle some other obscure scenarios in older versions of Node). The first argument passed to **finished** is the stream, the second is a callback function which is called when the **stream** has ended in some way. If there was an error in **stream** the callback will be passed an error object (the **err** parameter). So if the **err** parameter is truthy (e.g. not **null** or **undefined**) then we call **next** and pass it the **err**. The **next** function is the third parameter of the route handler, and passes control back to Express allowing it to handle the error as per the configuration in **app.js**.

One final note. Express applications tend to be legacy applications which means they may be built against older versions of Node and for various reasons may be stuck on that outdated version. In this case the core **stream** module may not have a **finished** function. However the [**readable-stream**](https://www.npmjs.com/package/readable-stream) module can be installed into old Node versions to provide the latest core **stream** functionality.

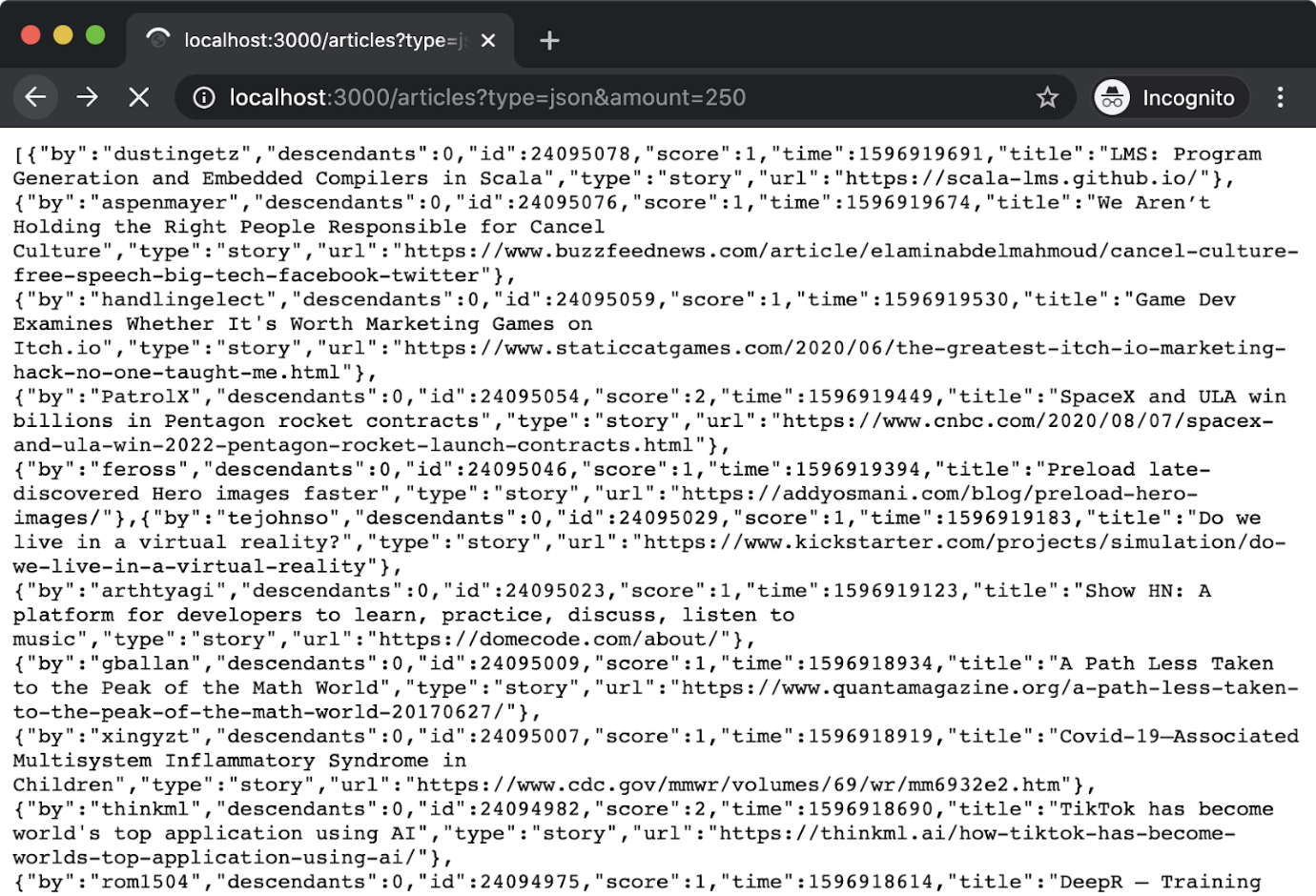
Let's try out our implementation. We can start the server using the following command:

**npm start**

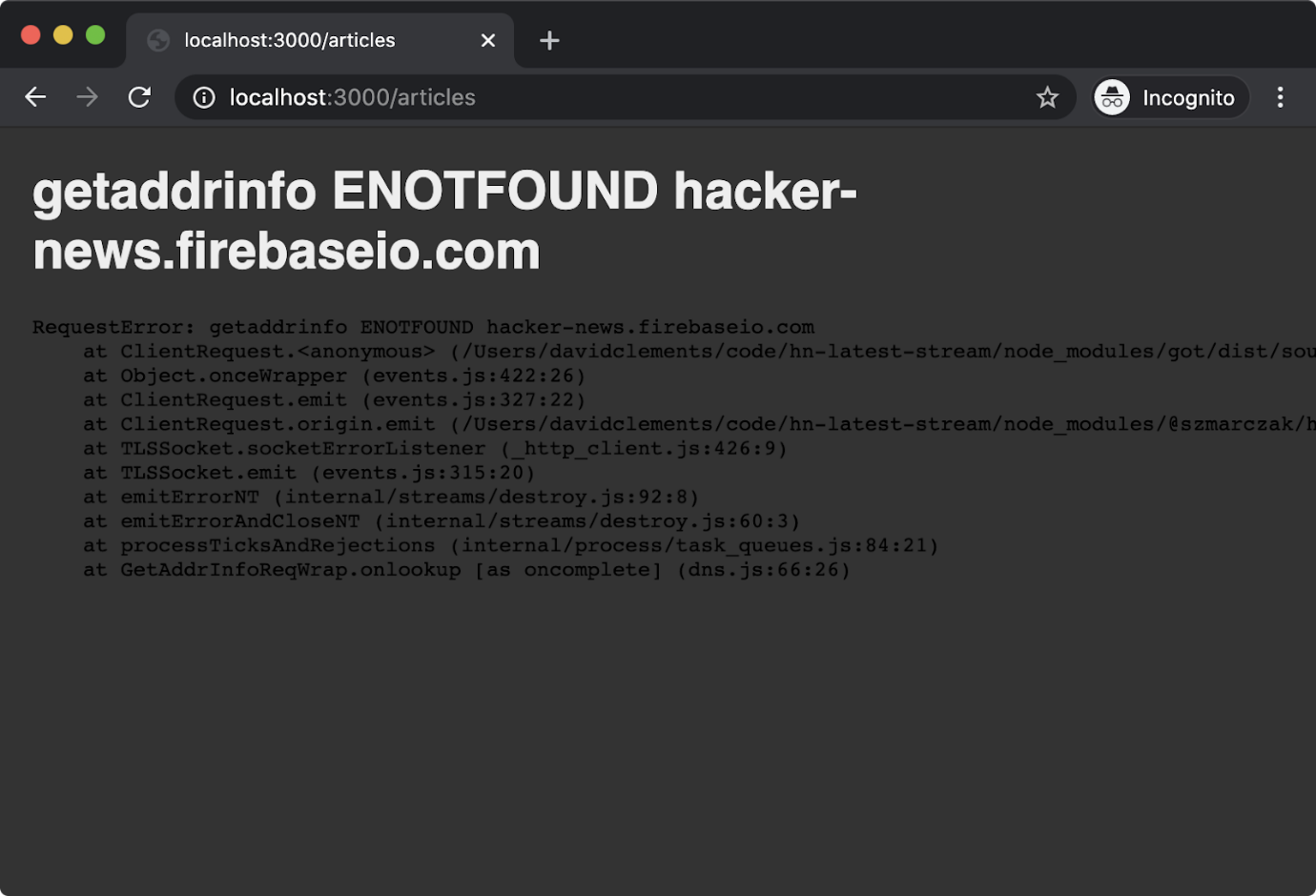
Now we can navigate to http://localhost:3000/articles and we should see output similar to but not the same as, the following:



We can also, as in the prior section, try navigating to the URL http://localhost:3000/articles?type=json&amount=250 to see some streamed JSON output:



Finally, let's make sure stream errors are propagated and handled as appropriately as server errors. If we disconnect Internet access on our local machine and then navigate to http://localhost:3000/articles we should see something similar to the following:

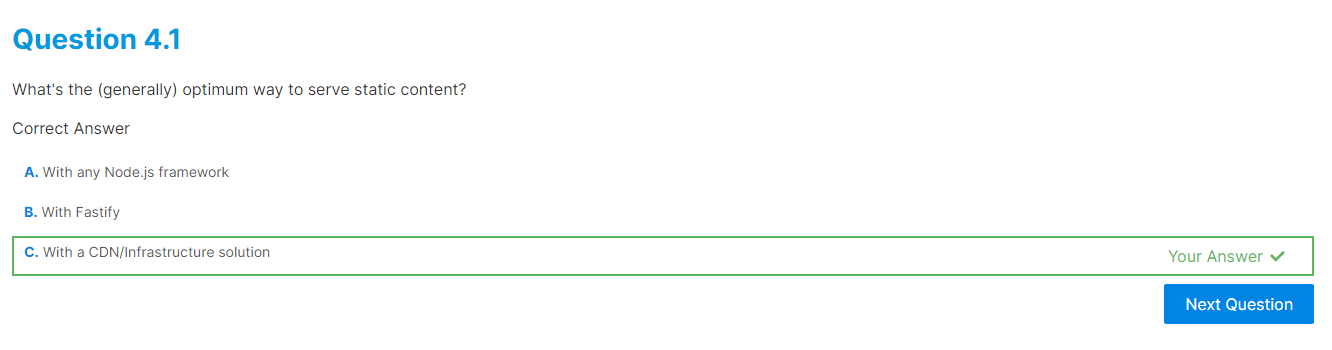


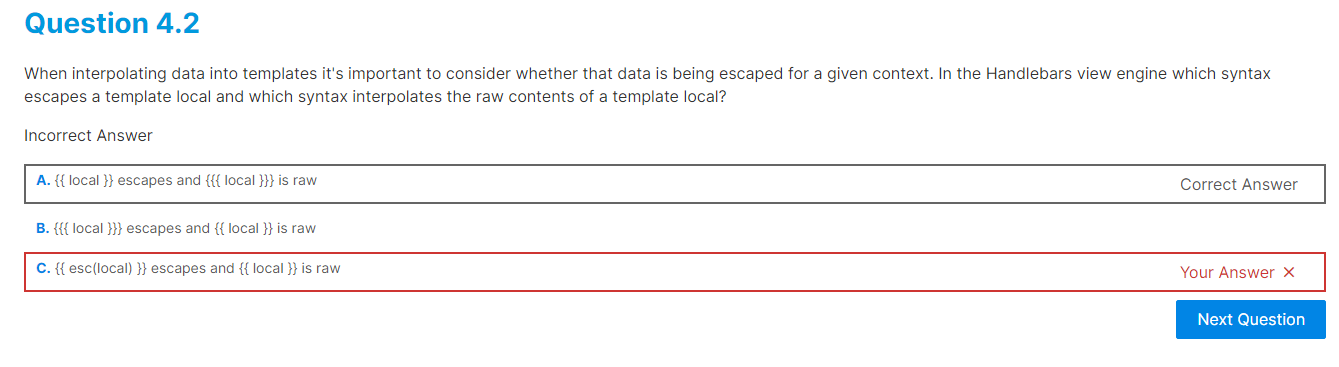
Notice the difference between the default way that Express displays errors versus Fastify. Fastify sets the HTTP status code to 500 and outputs some JSON that describes the error. Express likewise sets an HTTP 500 status code, and then renders a view in order to respond with HTML describing the error. In development it includes the stack trace but when the **NODE\_ENV** environment variable is set to **production** it only outputs the error message.

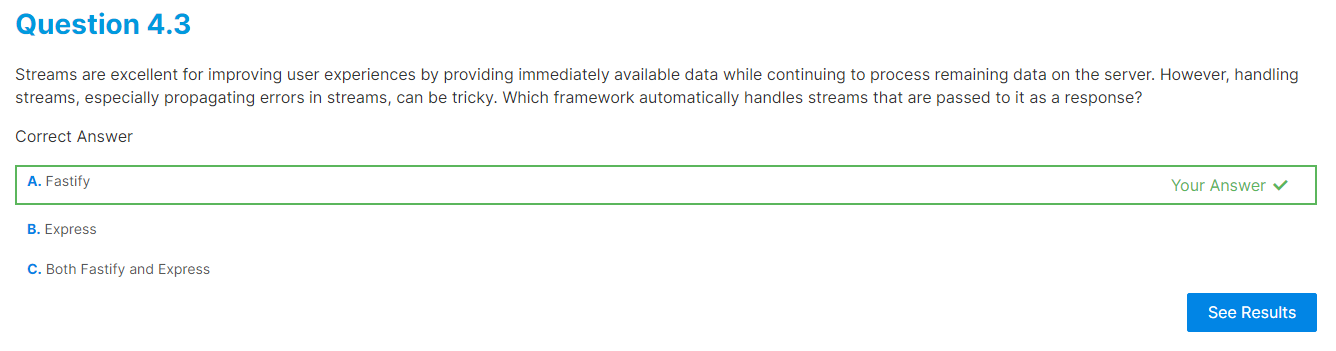
# Lab 4.1 - Render a View

# Lab 4.2 - Stream Some Content

### Knowledge Check







**Creating RESTful JSON Services**

# Chapter Overview

There are many ways to construct a service, and various ways to implement how those services communicate. Interoperability between services and between services and clients is a question of architectural style. This training and the Node Services Developer Certification Examination itself focuses on one of the most commonly used architectural styles for web-facing systems: RESTful services. It also focuses on one of the most common data interchange formats: JSON.

This chapter is about implementing RESTful JSON services, a basic understanding of REST and JSON is assumed. For more information see the [JSON specification](https://www.json.org/json-en.html) and the original dissertation ["Architectural Styles and the Design of Network-based Software Architectures"](https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm) (in particular Chapter 5) that outlined the REST architectural style. Alternatively, you can review the more easily digestible Wikipedia article - ["Representational State Transfer"](https://en.wikipedia.org/wiki/Representational_state_transfer).

**Learning Objectives**

By the end of this chapter, you should be able to:

* Understand common conventions for building and deploying RESTful JSON services.
* Ascertain how to implement a service that can handle a JSON GET to specific resources.
* Identify the fundamental minimum functional criteria that a RESTful JSON service should satisfy.

# Conventions

When writing a Node.js service it is typical to default to listening on a well-known port (say 3000) while also allowing that port to be overridden with the **PORT** environment variable. This can be accessed with **process.env.PORT** and then passed to a server's **listen** method.

The **npm init fastify** command which generates a Fastify project uses **fastify-cli** to start the application, which automatically allows the **PORT** environment variable to specify the port.

The **express** command-line executable that's installed by the **express-generator** module when globally installed generates a **bin/www** file that similarly allows the port to be specified via a **PORT** environment variable.

Another frequently occurring convention that is important to the deployment of Node.js services is a standard approach to starting the Node.js process. The standard approach is to have a **start** field in the **scripts** object of the package.json file so that the process can be started with **npm start**. The **npm start** command will execute the contents of the **start** field as a shell command.

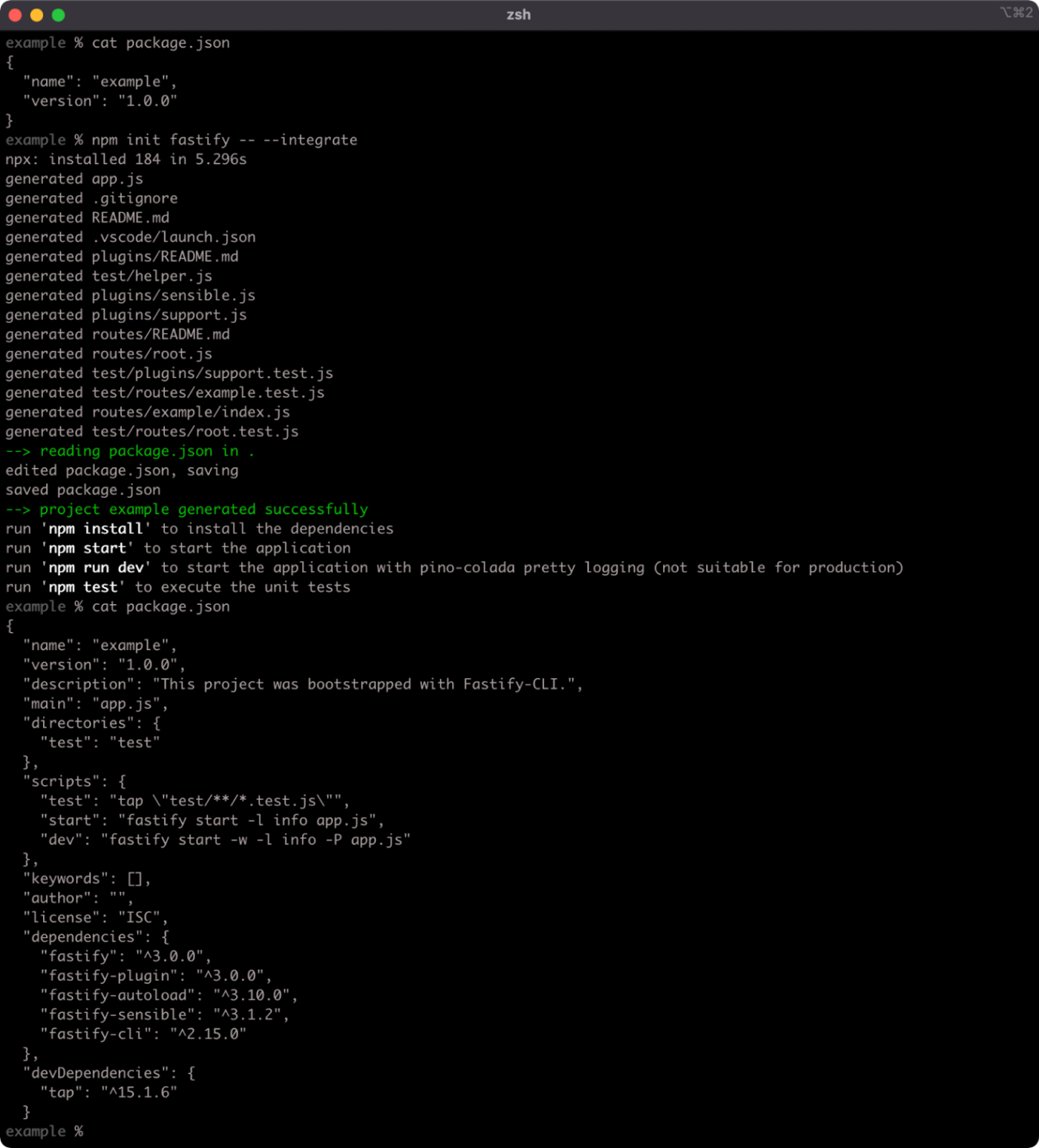
For instance in Express the **start** field would usually be:

**"scripts": {  
  "start": "node ./bin/www"  
}**

In Fastify the **--integrate** flag can be executed in a directory with a package.json file to generate a project and also update the preexisting package.json file:

**npm init fastify -- --integrate**

If this command fails, ensure that the installed npm version (**npm -v**) is 8.1.2 or higher. In this case the **start** field in the **scripts** object will be updated automatically and all the usual files (such as **app.js**, the **routes** folder and so on) will be added into that directory:



In the following sections we'll be using Fastify and Express to implement a RESTful JSON service.

**Implementing a RESTful JSON GET with Fastify (1)**

REST stands for REpresentational State Transfer, and it's an architectural style that seeks to make the most of the features of HTTP/1.1. Data is communicated via HTTP response bodies, metadata is communicated through HTTP headers, and operation outcomes are communicated with HTTP status codes. The State Transfer part of REST is about shuffling state from clients to server-backends. A REST service should be stateless, an intermediate layer between a browser and a database and it should boil down to performing one or more CRUD operations (Create, Read, Update, Delete).

In this section we'll focus on implementing a Read, which maps to the HTTP GET method.

Let's create a folder called **my-service** and make it our current working directory:

**node -e "fs.mkdirSync('my-service')"  
cd my-service**

Now let's generate a new Fastify project:

**npm init fastify**

Once the project is generated we need to install dependencies:

**npm install**

One of the included dependencies in a generated Fastify project is **fastify-sensible**. The **fastify-sensible** plugin adds some useful "sane defaults" to Fastify, including the convenience functions for HTTP status codes and messages. This plugin is not registered in the **app.js** file but the generated project contains a **plugins/sensible.js** file that registers **fastify-sensible**. The **plugins/sensible.js** file looks as follows:

**'use strict'**

**const fp = require('fastify-plugin')**

**/\*\*  
 \* This plugins adds some utilities to handle http errors  
 \*  
 \* @see ht‌tps://github.com/fastify/fastify-sensible  
 \*/  
module.exports = fp(async function (fastify, opts) {  
  fastify.register(require('fastify-sensible'), {  
    errorHandler: false  
  })  
})**

The **app.js** file loads all plugins in the **plugins** folder, using this piece of code:

**fastify.register(AutoLoad, {  
    dir: path.join(\_\_dirname, 'plugins'),  
    options: Object.assign({}, opts)  
  })**

So the **plugins/sensible.js** file contains a method call of **fastify.register** with the **fastify-sensible** module being passed as the first argument. The second argument sets the options for **fastify-sensible**, **errorHandler** is set to **false** by default. In production mode (for instance, when **process.env.NODE\_ENV** is set to **production**) it makes sense to set this to **true** as the **fastify-sensible** error handler produces lower-information error messages ("Something went wrong") which is better for public facing services.

A fairly typical situation would be to integrate a RESTful service with some kind of database such as MariaDB, Redis, Postgres, MongoDB and so forth. In every case the service is reading data from an external data source. In our case we're going to contrive a model that conceptually plays the role of a database but that stores state in-process. It's important to acknowledge that this actually violates the Statelessness constraint of REST, in that state shouldn't be held in the process that handles the HTTP request. However the model we'll create is a placeholder for teaching purposes that should be thought of in an abstract sense as a database connection.

Let's create a file which we'll call **model.js**:

**node -e "fs.openSync('model.js', 'w')"**

The contents of **model.js** should be as follows:

**'use strict'**

**module.exports = {  
  bicycle: bicycleModel()  
}**

**function bicycleModel () {  
  const db = {  
    1: { brand: 'Veloretti', color: 'green' },  
    2: { brand: 'Batavus', color: 'yellow' }  
  }**

**return {  
    read  
  }**

**function read (id, cb) {  
    if (!(db.hasOwnProperty(id))) {  
      const err = Error('not found')  
      setImmediate(() => cb(err))  
      return  
    }  
    setImmediate(() => cb(null, db[id]))  
  }  
}**

Not only is the code in **model.js** contrived, the error handling is subpar. Ideally there would be a **code** property on the errors and a map of error constants to check against. However the point here is to emulate more real-world scenarios where integrating with libraries can be messy and less than ideal. Note that the **read** function uses **setImmediate**; this is to simulate asynchronous operations. I/O operations should always be asynchronous when dealing with requests.

Our GET route must:

* Respond with a valid JSON payload.
* Respond with an **application/json Content-Type** header.
* Respond with 200 status code when successful.
* Respond with a 404 status when a requested resource is not available. This would be when the **read** function in **model.js** responds with an error with message **'not found'**.
* Respond with a 400, 404 or 405 for unsupported methods. For instance a POST to our server should respond with one of these codes, it doesn't matter which as the specification is ambiguous on these points and it can come down to implementation goals or wider policy.
* Respond with a 500 status for unknown errors.

*Cont'd on the next page.*

# Implementing a RESTful JSON GET with Fastify (2)

Let's create a route called **bicycle**. In Fastify all we need to do is create a directory in **routes** called **bicycle** and place an **index.js** file in it:

**cd routes  
node -e "fs.mkdirSync('bicycle')"  
cd bicycle  
node -e "fs.openSync('index.js', 'w')"  
cd ..  
cd ..**

The fact that the API we need to work with is callback-based allows us an opportunity to explore the various ways to integrate with callback-based API's (as opposed to promise-based) in Fastify.

While the response can be returned from an async function (which is to say, a promise can be returned which resolves to the response), the **reply.send** method can be used in a callback-based approach instead.

Let's make our **routes/bicycle/index.js** file look as follows:

**'use strict'**

**const { bicycle } = require('../../model')**

**module.exports = async (fastify, opts) => {  
  fastify.get('/:id', (request, reply) => {  
    const { id } = request.params  
    bicycle.read(id, (err, result) => {  
      if (err) {  
        if (err.message === 'not found') reply.notFound()  
        else reply.send(err)  
      } else reply.send(result)  
    })  
  })  
}**

Now, if we start our server (**npm start** or **npm run dev**) and navigate to http://localhost:3000/bicycle/1 in the browser we should see the following output:

**{"brand":"Veloretti","color":"green"}**

However, if we navigate to http://localhost:3000/bicycle/3 we should see:

**{"statusCode":404,"error":"Not Found","message":"Not Found"}**

We set the route up with **fastify.get('/:id', …)**. The **:id** part of the route creates a parameter, which can then be accessed on **request.params.id**. So a request to **/bicycle/1** will mean that **request.params.id** is 1. Recall that folder name is causing the route to mount at **/bicycle** which is why the route is specified as **/:id** and not **/bicycle/:id**.

This **id** is passed to **bicycle.read** which calls the callback provided to it passing either an **err** object as the first argument or a **result** object as the second argument. If the **err** argument is falsy (e.g. **null**) then **reply.send(result)** is called. The **result** argument would hold a JavaScript object, so Fastify would know to set the response **Content-Type** header to **application/json**.

If there is an error, the **err** object will be populated. If the message is **'not found'** then **reply.notFound** is called. This is a method added by the **fastify-sensible** plugin, it sets the response status code to 404 and generates some JSON output describing the Not Found error. If the error is something else, this is assumed to be a server error, and the **err** object is passed to **reply.send**. This causes Fastify to generate a 500 response and output the error message.

Note that the function passed as the route handler to **fastify.get** is not an async function. This is because an async function would return a promise that would resolve immediately and the route would fail as it tries to process a response value of **undefined**. Or if we did return something from it, that would be the response and the call to **reply.send** would be too late and result in an error regarding writing to a response that has ended.

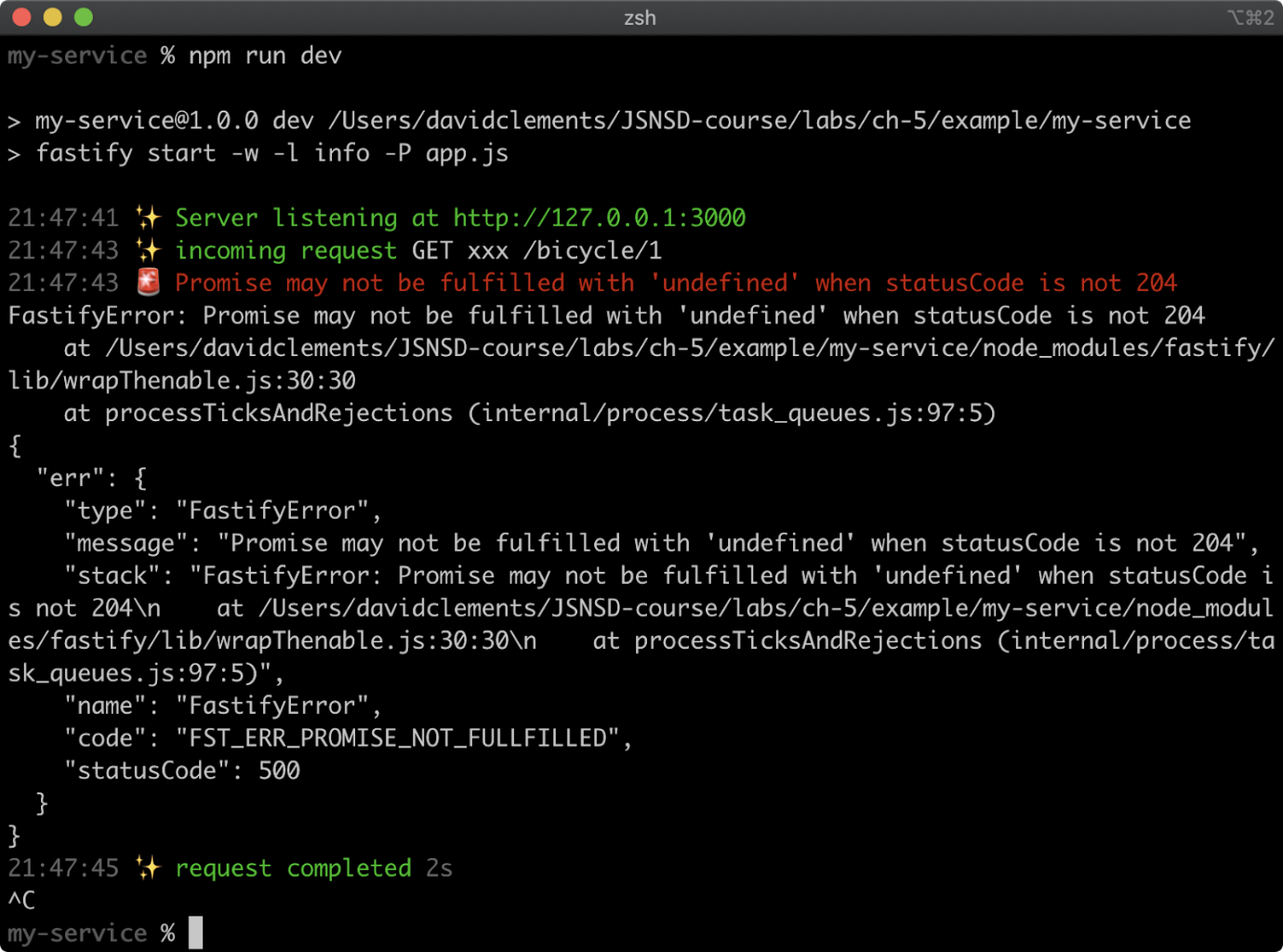
If we wanted to use a callback API inside an async route handler the following approach could instead be taken. Let's modify **routes/bicycle/index.js** to the following:

**'use strict'**

**const { bicycle } = require('../../model')**

**module.exports = async (fastify, opts) => {  
  fastify.get('/:id', async (request, reply) => {  
    const { id } = request.params  
    bicycle.read(id, (err, result) => {  
      if (err) {  
        if (err.message === 'not found') reply.notFound()  
        else reply.send(err)  
      } else reply.send(result)  
    })  
    await reply  
  })  
}**

This can be a useful approach when mixing callback-based API's and promise-based API's in a route handler. If the **await reply** line is removed it results in an error message:



Cont'd on the next page.

# Implementing a RESTful JSON GET with Fastify (3)

The other approach to using callback-based APIs in an async function is to promisify the API.

Let's wrap up by altering **index.js** again to look as follows:

**'use strict'  
const { promisify } = require('util')  
const { bicycle } = require('../../model')  
const read = promisify(bicycle.read)**

**module.exports = async (fastify, opts) => {  
  const { notFound } = fastify.httpErrors**

**fastify.get('/:id', async (request, reply) => {  
    const { id } = request.params  
    try {  
      return await read(id)  
    } catch (err) {  
      if (err.message === 'not found') throw notFound()  
      throw err  
    }  
  })  
}**

This will yield the same behavior again. Passing **bicycle.read** to **util.promisify** causes a new function to be returned, which we assign to **read**. When called this will return a promise that resolves with the result or rejects with the error depending on the outcome.

Note that to generate a 404 Not Found HTTP Status we throw **fastify.httpErrors.notFound** instead of using **reply.notFound**. We also throw the caught **err** instead of passing **err** to **reply.send**. This is extremely useful as any unexpected throw in an async route handler will result in 500 status.

All three forms of **routes/bicycle/index.js** behave the same way, all of them meet the requirements for our GET route.

Let's check that our GET route is operating according to our requirements.

We know that navigating to /bicycle/{id} with an existing ID provides a JSON payload, but we have not yet validated if the **Content-Type** header is correct. With our server running in one terminal (**npm start** or **npm run dev** to start the server), in another terminal we can use the following command to check the response headers:

**node -e "http.get('http://localhost:3000/bicycle/1', ({headers}) => console.log(headers))"**

This should output something similar to the following:



We can see that there is a **'content-type'** property in the response headers, with a value of **'application/json; charset=utf-8'**. The Fastify framework has detected that the response is JSON and set the headers appropriately.

Now let's verify that the status codes are appropriate. The following command can be used to verify a 200 response:

**node -e "http.get('http://localhost:3000/bicycle/1', ({statusCode}) => \  
console.log(statusCode))"**

This should output: **200**.

We can check a 404 with the same but with a non-existent ID:

**node -e "http.get('http://localhost:3000/bicycle/9', ({statusCode}) => \  
console.log(statusCode))"**

This should output: **404**. For this case the **model.js** **read** function responded with an error with the message **'not found'** and the **fastify-sensible notFound** method was used to generate a 404 response.

Let's try a POST request to a valid GET route and see what happens:

**node -e "http.request('http://localhost:3000/bicycle/1', { method: 'post'}, ({statusCode}) \  
=> console.log(statusCode)).end()"**

**Implementing a RESTful JSON GET with Fastify (4)**

For Fastify the default behavior in this scenario is to respond with a 404 as well, so this will output: **404**. Other acceptable response status codes would be 405 Method not Allowed and 400 Bad Request. The reason that a 405 is not the default for this scenario is that a 404 gives less information than a 405, so for public facing services this is a more secure approach.

Finally, let's check whether the server responds with a 500 status code for an unknown error. We'll have to modify the **models.js** file for this one. Let's temporarily alter the **read** function in **models.js** to look as follows:

**function read (id, cb) {  
  setImmediate(() => cb(Error()))  
}**

To make sure this change is applied, restart the server (Ctrl+C and then **npm run dev** or **npm start**), then in another terminal run the following command:

**node -e "http.get('http://localhost:3000/bicycle/1', ({statusCode}) => console.log(statusCode))"**

The route now has an error that doesn't relate to an ID not existing, so the output of this command should now be: **500**. In the async function route example, any error that doesn't have the message **'not found'** is re-thrown inside the **catch** block. This propagates the error so that it's handled by Fastify, which auto-generates a 500 Server Error status code. In the callback-based examples of the route handler, any error that doesn't have the message **'not found'** is passed to **reply.send** which recognizes that it's been passed an error object and from there generates a 500 Server Error status code.

The criteria for our GET route is:

* Respond with a valid JSON payload
* Respond with an **application/json Content-Type** header
* Respond with 200 status code when successful
* Respond with a 404 status when a requested resource is not available. This would be when the **read** function in **model.js** responds with an error with message **'not found'**
* Respond with a 400, 404 or 405 for unsupported methods. For instance a POST to our server should respond with one of these codes, it doesn't matter which as the specification is ambiguous on these points and it can come down to implementation goals or wider policy.
* Respond with a 500 status for unknown errors

We've now verified that our implementation matches all of the criteria we set for our RESTful JSON GET route. In the next section we're going to do the same again with Express.

**Implementing a RESTful JSON GET with Express (1)**

For the sake of comparison, we'll concisely reimplement our RESTful JSON GET route in Express.

Recall that the requirements for our GET route are as follows:

* Respond with a valid JSON payload
* Respond with an **application/json Content-Type** header
* Respond with 200 status code when successful
* Respond with a 404 status when a requested resource is not available. This would be when the **read** function in **model.js** responds with an error with message **'not found'**
* Respond with a 400, 404 or 405 for unsupported methods. For instance a POST to our server should respond with one of these codes, it doesn't matter which as the specification is ambiguous on these points and it can come down to implementation goals or wider policy.
* Respond with a 500 status for unknown errors

In a previous chapter we installed **express-generator** package globally so an **express** command-line executable on our machine, if not we can install it with **npm install -g express-generator**.

Let's generate an Express project with the following command:

**express my-express-service**

Next, we'll set our working folder to **my-express-service** and install the project dependencies:

**cd my-express-service  
npm install**

Now, let's add the **model.js** file to the project root and create a **routes/bicycle.js** file:

**node -e "fs.openSync('model.js', 'w')"  
cd routes  
node -e "fs.openSync('bicycle.js', 'w')"  
cd ..**

The contents of the **model.js** file should be:

**'use strict'**

**module.exports = {  
  bicycle: bicycleModel()  
}**

**function bicycleModel () {  
  const db = {  
    1: { brand: 'Veloretti', color: 'green' },  
    2: { brand: 'Batavus', color: 'yellow' }  
  }**

**return {  
    read  
  }**

**function read (id, cb) {  
    if (!(db.hasOwnProperty(id))) {  
      const err = Error('not found')  
      setImmediate(() => cb(err))  
      return  
    }  
    setImmediate(() => cb(null, db[id]))  
  }  
}**

The **routes/bicycle.js** file should look as follows:

**var express = require('express');  
var router = express.Router();  
var model = require('../model');**

**router.get('/:id', function(req, res, next) {  
  model.bicycle.read(req.params.id, (err, result) => {  
    if (err) {  
      if (err.message === 'not found') next();  
      else next(err);  
    } else {  
      res.send(result);  
    }  
  });**

**});**

**module.exports = router;**

Finally, we need to modify the **app.js** file to include our **routes/bicycle.js** router. Around line 9 of **app.js** we need to add the following:

**var bicycleRouter = require('./routes/bicycle');**

Around line 25 of **app.js**, above the error handling middleware we need to register the router at the **/bicycle** route like so:

**app.use('/bicycle', bicycleRouter);**

*Cont'd on the next page.*

# Implementing a RESTful JSON GET with Express (2)

Now if we start the service with npm start and navigate to http://localhost:3000/bicycle/1 the response body should be:

**{"brand":"Veloretti","color":"green"}**

In a separate terminal to the where the server is running, we can use the following command to check the response headers:

**node -e "http.get('http://localhost:3000/bicycle/1', ({headers}) => console.log(headers))"**

This should output something similar to the following:



We can see that there is a 'content-type' property in the response headers, with a value of 'application/json; charset=utf-8'. The Express framework has detected that the response is JSON because **res.send** was passed an object, and set the headers appropriately.

The following command can be used to verify a 200 response:

**node -e "http.get('http://localhost:3000/bicycle/1', ({statusCode}) => console.log(statusCode))"**

This should output: **200**.

We can check a 404 with the same but with a non-existent ID:

**node -e "http.get('http://localhost:3000/bicycle/9', ({statusCode}) => console.log(statusCode))"**

This should output: **404**. For this case the **model.js** **read** function responded with an error with the message **'not found'** and the **next** function in the route handler was called without writing anything to the response. This caused Express to move to the next piece of middleware in the **app.js** file which is:

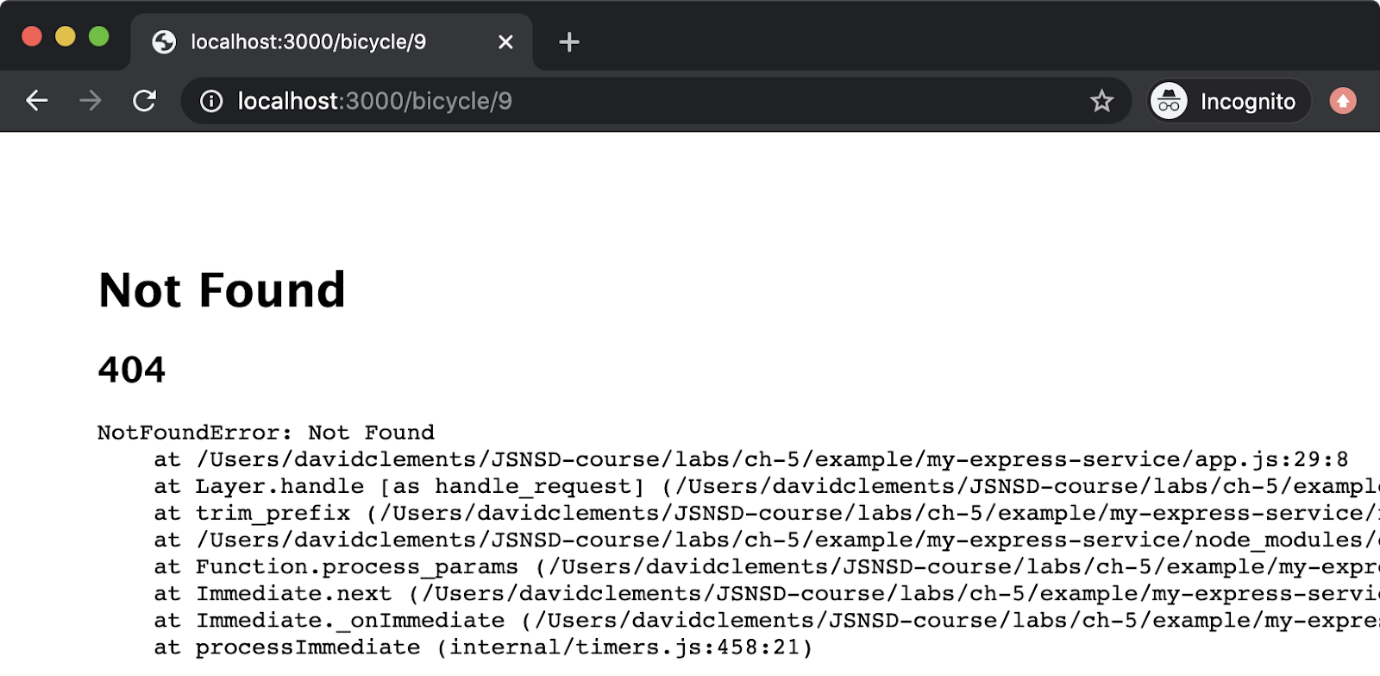
**app.use(function(req, res, next) {  
  next(createError(404));  
});**

If any route calls the **next** callback without an error a 404 error is generated by this small middleware function in **app.js**. Since this middleware passes this error to its own **next** callback, this will fall through to the error handling middleware in **app.js** which is:

**app.use(function(err, req, res, next) {  
  // set locals, only providing error in development  
  res.locals.message = err.message;  
  res.locals.error = req.app.get('env') === 'development' ? err : {};**

**// render the error page  
  res.status(err.status || 500);  
  res.render('error');  
});**

While this meets our criteria by responding with a 404 status code, this error-handling middleware does not output JSON, it outputs HTML. We can see this HTML rendered in the browser by navigating to http://localhost:3000/bicycle/9:



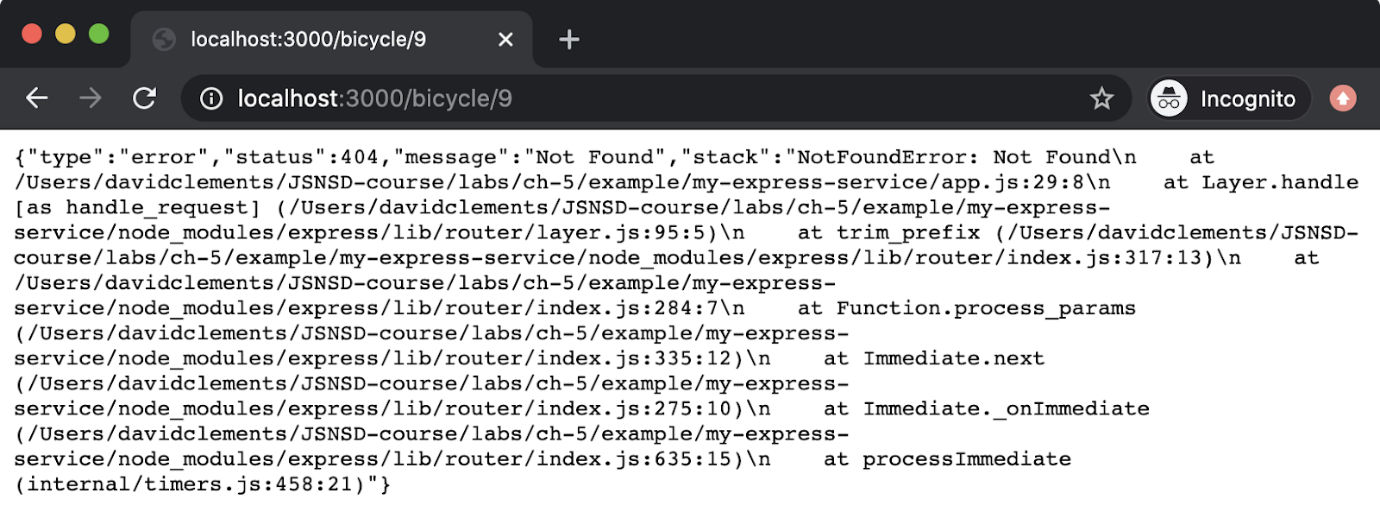
Cont'd on the next page.

**Implementing a RESTful JSON GET with Express (3)**

While Fastify is geared towards building RESTful services, Express defaults to serving HTML as it was primarily for dynamic content generation. The error handling middleware can be modified to the following to produce a JSON response when errors occur:

**app.use(function(err, req, res, next) {  
  res.status(err.status || 500);  
  res.send({  
    type: 'error',  
    status: err.status,  
    message: err.message,  
    stack: req.app.get('env') === 'development' ? err.stack : undefined  
  });  
});**

Restarting the server and navigating to http://localhost:3000/bicycle/9 will now produce something like the following:



Let's try a POST request to a valid GET route and see what happens:

**node -e "http.request('http://localhost:3000/bicycle/1', { method: 'post'}, ({statusCode}) => console.log(statusCode)).end()**"

For Express the default behavior in this scenario is to respond with a 404 as well, so this will output: **404**.

Finally let's check whether the server responds with a 500 status code for an unknown error. Let's temporarily alter the **read** function in **models.js** to look as follows:

**function read (id, cb) {  
  setImmediate(() => cb(Error()))  
}**

To make sure this change is applied, restart the server (Ctrl+C and then **npm start**), then in another terminal run the following command:

**node -e "http.get('http://localhost:3000/bicycle/1', ({statusCode}) => console.log(statusCode))"**

The route now has an error that doesn't relate to an ID not existing, so the output of this command should now be: **500**.

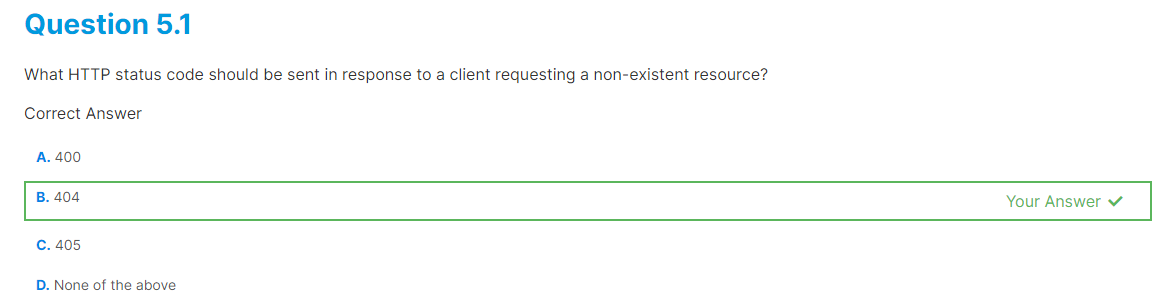
The criteria for our GET route is:

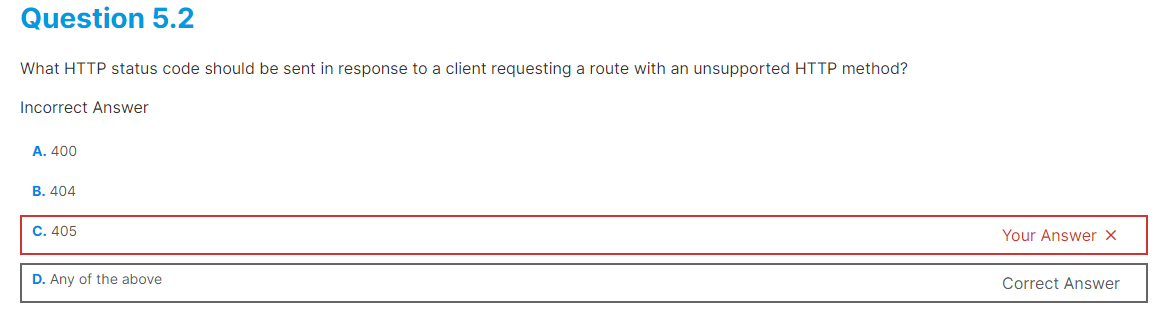
* Respond with a valid JSON payload
* Respond with an **application/json Content-Type** header
* Respond with 200 status code when successful
* Respond with a 404 status when a requested resource is not available. This would be when the **read** function in **model.js** responds with an error with message **'not found'**
* Respond with a 400, 404 or 405 for unsupported methods. For instance a POST to our server should respond with one of these codes, it doesn't matter which as the specification is ambiguous on these points and it can come down to implementation goals or wider policy.
* Respond with a 500 status for unknown errors

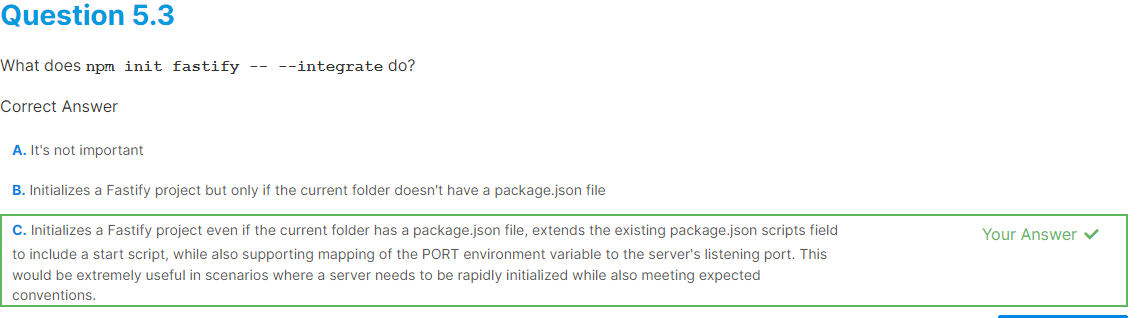
We've now verified that our implementation matches all of the criteria we set for our RESTful JSON GET route.

# Lab 5.1 - Implement a RESTful JSON GET

### Knowledge Check







**Manipulating Data with RESTful Services**

# Chapter Overview

In the prior chapter we covered some conventions to use when creating Node.js services and set up a JSON **GET** route. In this chapter we're going to extend the services we built, with support for **POST**, **PUT** and **DELETE** HTTP methods.

**Learning Objectives**

By the end of this chapter, you should be able to:

* Learn how to implement **POST** request functionality for creating and updating data entries.
* Understand how to implement **PUT** request for creating and updating data entries as well as the differences between **POST** and **PUT**.
* Learn how to implement **DELETE** requests for removing data entries.

# Implementing POST, PUT and DELETE with Fastify (1)

In the previous chapter we created a folder called **my-service** and generated a Fastify project in that folder using the **npm init fastify** command. In addition, we added a **model.js** file with a **read** method, which we'll adapt in a moment and the **routes/bicycle/index.js** file which ended up looking as follows:

**'use strict'  
const { promisify } = require('util')  
const { bicycle } = require('../../model')  
const read = promisify(bicycle.read)**

**module.exports = async (fastify, opts) => {  
  const { notFound } = fastify.httpErrors**

**fastify.get('/:id', async (request, reply) => {  
    const { id } = request.params  
    try {  
      return await read(id)  
    } catch (err) {  
      if (err.message === 'not found') throw notFound()  
      throw err  
    }  
  })  
}**

We're going to add new routes to integrate with new model capabilities. Let's update the **model.js** file to the following:

**'use strict'**

**module.exports = {  
  bicycle: bicycleModel()  
}**

**function bicycleModel () {  
  const db = {  
    1: { brand: 'Veloretti', color: 'green' },  
    2: { brand: 'Batavus', color: 'yellow' }  
  }**

**return {  
    create, read, update, del, uid  
  }**

**function uid () {  
    return Object.keys(db)  
      .sort((a, b) => a - b)  
      .map(Number)  
      .filter((n) => !isNaN(n))  
      .pop() + 1 + ''  
  }**

**function create (id, data, cb) {  
    if (db.hasOwnProperty(id)) {  
      const err = Error('resource exists')  
      setImmediate(() => cb(err))  
      return  
    }  
    db[id] = data  
    setImmediate(() => cb(null, id))  
  }**

**function read (id, cb) {  
    if (!(db.hasOwnProperty(id))) {  
      const err = Error('not found')  
      setImmediate(() => cb(err))  
      return  
    }  
    setImmediate(() => cb(null, db[id]))  
  }**

**function update (id, data, cb) {  
    if (!(db.hasOwnProperty(id))) {  
      const err = Error('not found')  
      setImmediate(() => cb(err))  
      return  
    }  
    db[id] = data  
    setImmediate(() => cb())  
  }**

**function del (id, cb) {  
    if (!(db.hasOwnProperty(id))) {  
      const err = Error('not found')  
      setImmediate(() => cb(err))  
      return  
    }  
    delete db[id]  
    setImmediate(() => cb())  
  }  
}**

Cont'd on the next page.

# Implementing POST, PUT and DELETE with Fastify (2)

Along with the **read** function there's now **create**, **update** and **del** functions and a function for calculating the next ID for an entry. Recall that the implementation is contrived, it's not meant to be real. However the interface (**create**, **read**, **update**, **del**) does conceptually reflect the sort of operations we would perform on a database.

The difference between a **POST** and **PUT** is nuanced. Both involve sending data from a client to a server but they are supposed to behave differently. The key difference is idempotency, which means that multiple identical operations should lead to the same result. **POST** is not idempotent whereas **PUT** is idempotent. So multiple identical **POST** requests would, for instance, create multiple entries with identical data, contrariwise multiple **PUT** requests should overwrite the same entry with the same data. This does not mean that **PUT** can't be used to create entries, or that **POST** can't be used to update, it's just that expected behavior is different. A **POST** request can be used to create an entry without supplying an ID, whereas a **PUT** request could be used to create an entry where a specific ID is desired. Using **POST** to update should be an explicitly separate route for updating versus creating whereas the ability to update or create with **PUT** can exist on the same route. With all that in mind let's implement **POST** functionality by updating **routes/bicycle/index.js** to look as follows:

**'use strict'  
const { promisify } = require('util')  
const { bicycle } = require('../../model')  
const { uid } = bicycle  
const read = promisify(bicycle.read)  
const create = promisify(bicycle.create)  
const update = promisify(bicycle.update)**

**module.exports = async (fastify, opts) => {  
  const { notFound } = fastify.httpErrors**

**fastify.post('/', async (request, reply) => {  
    const { data } = request.body  
    const id = uid()  
    await create(id, data)  
    reply.code(201)  
    return { id }  
  })**

**fastify.post('/:id/update', async (request, reply) => {  
    const { id } = request.params  
    const { data } = request.body  
    try {  
      await update(id, data)  
      reply.code(204)  
    } catch (err) {  
      if (err.message === 'not found') throw notFound()  
      throw err  
    }  
  })**

**fastify.get('/:id', async (request, reply) => {  
    const { id } = request.params  
    try {  
      return await read(id)  
    } catch (err) {  
      if (err.message === 'not found') throw notFound()  
      throw err  
    }  
  })  
}**

We've now added two **POST** routes: **POST /bicycle** and **POST /bicycle/:id/update**. Let's study each in isolation. The first POST route is configured with the following route handler:

**fastify.post('/', async (request, reply) => {  
    const { data } = request.body  
    const id = uid()  
    await create(id, data)  
    reply.code(201)  
    return { id }  
  })**

Cont'd on the next page.

# Implementing POST, PUT and DELETE with Fastify (3)

This route allows a new entry to be created by using the **uid** method exported from **model.js** to get a new ID and then passes that ID along with an expected **data** property in the request **POST** payload to the **create** method.

Note how there is no explicit error handling here, since the only known error would be regarding the resource already existing and since the **uid** function provides a new ID that won't be an issue. Any error therefore would be an unknown error, if **create** throws for any reason this will cause the async function route handler to throw and then be handled as a 500 Server Error by Fastify.

A successful request will respond with a 201 Created status code and send back a JSON object containing an **id** property with a value of the ID for the new entry.

By default Fastify supports **application/json POST** requests. The **fastify-multipart** plugin can be used to support **multipart/formdata** requests and **fastify-formbody** can be used to support **application/x-www-form-urlencoded POST** requests. Our goal is to support **application/json** so the core functionality is all we need.

Making sure that the server has been started in another terminal (**npm run dev** or **npm start**), the following command will perform a **POST** request to **/bicycle** and output the status code and the response body:

**node -e "http.request('http://localhost:3000/bicycle', { method: 'post', headers: {'content-type': 'application/json'}}, (res) => res.setEncoding('utf8').once('data', console.log.bind(null, res.statusCode))).end(JSON.stringify({data: {brand: 'Gazelle', color: 'red'}}))"**

The first time this is executed the output should be: **201 {"id":"3"}**.

If it is executed again with the same data, the output would be: **201 {"id":"4"}** because **POST** requests are meant to be non-idempotent.

We can check that the entry was added by hitting the **GET** route with the new ID:

**node -e "http.get('http://localhost:3000/bicycle/3', (res) => res.setEncoding('utf8').once('data', console.log))"**

This should output: **{"brand":"Gazelle","color":"red"}**. This is the **data** property of the object that we sent to the service in the **POST** request. Of course we could also just navigate to http://localhost:3000/bicycle/3 in a browser to see the same output.

Let's take a look at the handler for the second **POST** route:

**fastify.post('/:id/update', async (request, reply) => {  
    const { id } = request.params  
    const { data } = request.body  
    try {  
      await update(id, data)  
      reply.code(204)  
    } catch (err) {  
      if (err.message === 'not found') throw notFound()  
      throw err  
    }  
  })**

This allows the client to make a **POST** request to, for example, **/bicycle/3/update** and the entry with an ID of 3 will be updated and the response will contain no data and have a status code of 204 No Content since there's nothing we really need to send back but the request was successfully processed. As a side note, this is the only case where a JSON service may respond without the **application/json Content-Type** header since there is no content, so there is no content-type.

If an entry for a given ID does not exist then a 404 response will result.

Let's try it out with the following command:

**node -e "http.request('http://localhost:3000/bicycle/3/update', { method: 'post', headers: {'content-type': 'application/json'}}, (res) => console.log(res.statusCode)).end(JSON.stringify({data: {brand: 'Ampler', color: 'blue'}}))"**

This should output **204**. We can then see if the update worked with another GET request to http://localhost:3000/bicycle/3:

**node -e "http.get('http://localhost:3000/bicycle/3', (res) => res.setEncoding('utf8').once('data', console.log))"**

This command should now output: **{"brand":"Ampler","color":"blue"}**.

The following shows these requests and responses all together:



Cont'd on the next page.

# Implementing POST, PUT and DELETE with Fastify (4)

Typically a service would either use **POST** or **PUT** for both creation and updates, or sometimes less-advisedly, **POST** for creation and **PUT** for updates. However, for understanding purposes, we'll implement **PUT** for updating and creating entries as well.

Let's update **routes/bicycle/index.js** to the following:

**'use strict'  
const { promisify } = require('util')  
const { bicycle } = require('../../model')  
const { uid } = bicycle  
const read = promisify(bicycle.read)  
const create = promisify(bicycle.create)  
const update = promisify(bicycle.update)**

**module.exports = async (fastify, opts) => {  
  const { notFound } = fastify.httpErrors**

**fastify.post('/', async (request, reply) => {  
    const { data } = request.body  
    const id = uid()  
    await create(id, data)  
    reply.code(201)  
    return { id }  
  })**

**fastify.post('/:id/update', async (request, reply) => {  
    const { id } = request.params  
    const { data } = request.body  
    try {  
      await update(id, data)  
      reply.code(204)  
    } catch (err) {  
      if (err.message === 'not found') throw notFound()  
      throw err  
    }  
  })**

**fastify.get('/:id', async (request, reply) => {  
    const { id } = request.params  
    try {  
      return await read(id)  
    } catch (err) {  
      if (err.message === 'not found') throw notFound()  
      throw err  
    }  
  })**

**fastify.put('/:id', async (request, reply) => {  
    const { id } = request.params  
    const { data } = request.body  
    try {  
      await create(id, data)  
      reply.code(201)  
      return {}  
    } catch (err) {  
      if (err.message === 'resource exists') {  
        await update(id, data)  
        reply.code(204)  
      } else {  
        throw err  
      }  
    }  
  })**

**}**

The only addition is the **fastify.put** call at the bottom of the exported function. Let's take a close look:

**fastify.put('/:id', async (request, reply) => {  
    const { id } = request.params  
    const { data } = request.body  
    try {  
      await create(id, data)  
      reply.code(201)  
      return {}  
    } catch (err) {  
      if (err.message === 'resource exists') {  
        await update(id, data)  
        reply.code(204)  
      } else {  
        throw err  
      }  
    }  
  })**

Cont'd on the next page.

# Implementing POST, PUT and DELETE with Fastify (5)

This one route allows for both creation and updates of entries but the ID has to be specified for creation. The **model.js** **create** function will respond with an error when trying to create an entry with an ID that already exists, so our **PUT** route handler can simply try to create an entry and if that fails update the entry instead. Any other error is re-thrown, which would then be handled by Fastify as a 500 Server Error response.

If the entry is created, the response status code is set to 201 Created. The only legitimate case for responding with no data is when the status code is 204 No Content but since 201 Created applies far more strongly in the case of entry creation we send an empty object in response. If the entry is updated, we do respond with 204 No Content status code to indicate that the entry was updated but that there's no data to return.

We can try out our **PUT** route with the following command:

**node -e "http.request('http://localhost:3000/bicycle/99', { method: 'put', headers: {'content-type': 'application/json'}}, (res) => console.log(res.statusCode)).end(JSON.stringify({data: {brand: 'VanMoof', color: 'black'}}))"**

This command should output: **201**. Since there won't be an entry with an ID of 99 this will create a new entry, which we can retrieve with a **GET** request to http://localhost:3000/bicycle/99:

**node -e "http.get('http://localhost:3000/bicycle/99', (res) => res.setEncoding('utf8').once('data', console.log))"**

This command should output: **{"brand":"VanMoof","color":"black"}**.

We can now hit the same route with different data to update it:

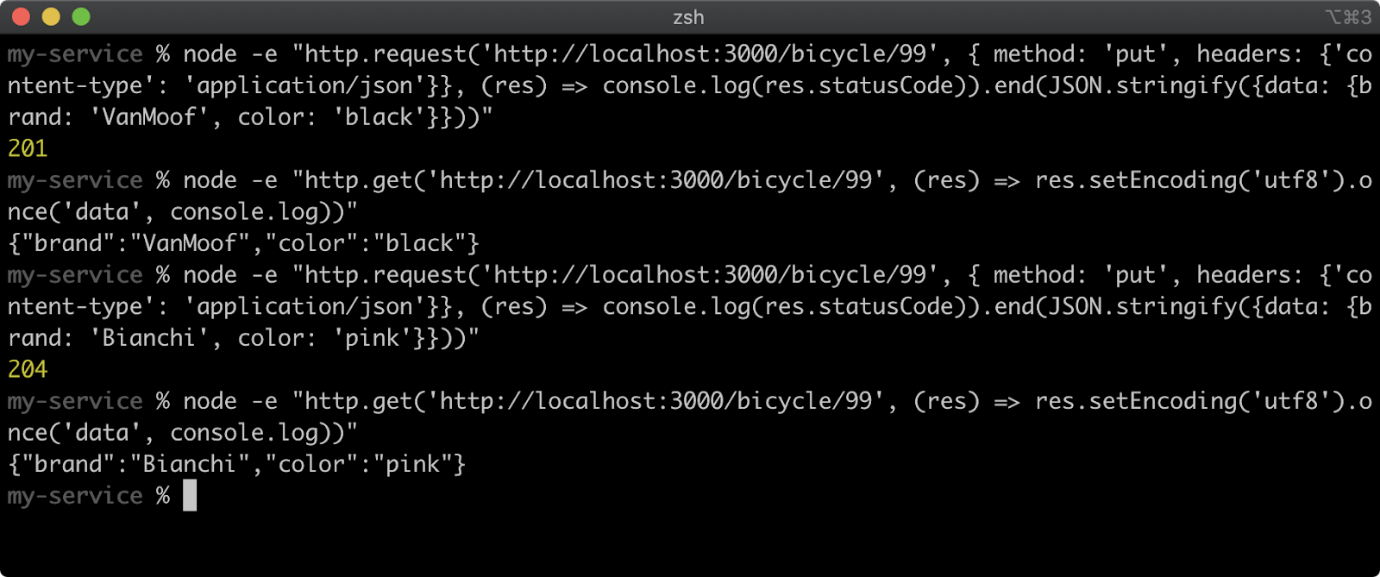
**node -e "http.request('http://localhost:3000/bicycle/99', { method: 'put', headers: {'content-type': 'application/json'}}, (res) => console.log(res.statusCode)).end(JSON.stringify({data: {brand: 'Bianchi', color: 'pink'}}))"**

This command should output: **204**. We can verify that the update occurred with the following:

**node -e "http.get('http://localhost:3000/bicycle/99', (res) => res.setEncoding('utf8').once('data', console.log))"**

This should output: **{"brand":"Bianchi","color":"pink"}**.

The following shows these interactions with our **PUT** route:



Finally, let's implement a **DELETE** route by altering **routes/bicycle/index.js** to the following:

**'use strict'  
const { promisify } = require('util')  
const { bicycle } = require('../../model')  
const { uid } = bicycle  
const read = promisify(bicycle.read)  
const create = promisify(bicycle.create)  
const update = promisify(bicycle.update)  
const del = promisify(bicycle.del)**

**module.exports = async (fastify, opts) => {  
  const { notFound } = fastify.httpErrors**

**fastify.post('/', async (request, reply) => {  
    const { data } = request.body  
    const id = uid()  
    await create(id, data)  
    reply.code(201)  
    return { id }  
  })**

**fastify.post('/:id/update', async (request, reply) => {  
    const { id } = request.params  
    const { data } = request.body  
    try {  
      await update(id, data)  
      reply.code(204)  
    } catch (err) {  
      if (err.message === 'not found') throw notFound()  
      throw err  
    }  
  })**

**fastify.get('/:id', async (request, reply) => {  
    const { id } = request.params  
    try {  
      return await read(id)  
    } catch (err) {  
      if (err.message === 'not found') throw notFound()  
      throw err  
    }  
  })**

**fastify.put('/:id', async (request, reply) => {  
    const { id } = request.params  
    const { data } = request.body  
    try {  
      await create(id, data)  
      reply.code(201)  
      return { }  
    } catch (err) {  
      if (err.message === 'resource exists') {  
        await update(id, data)  
        reply.code(204)  
      } else {  
        throw err  
      }  
    }  
  })**

**fastify.delete('/:id', async (request, reply) => {  
    const { id } = request.params  
    try {  
      await del(id)  
      reply.code(204)  
    } catch (err) {  
      if (err.message === 'not found') throw notFound()  
      throw err  
    }  
  })**

**}**

Cont'd on the next page.

# Implementing POST, PUT and DELETE with Fastify (6)

At the top of the file we added **const del = promisify(bicycle.del)** to our other promisified **model.js** functions, and then at the bottom of the exported function we've added the **fastify.delete** call. Let's look at that in isolation:

**fastify.delete('/:id', async (request, reply) => {  
    const { id } = request.params  
    try {  
      await del(id)  
      reply.code(204)  
    } catch (err) {  
      if (err.message === 'not found') throw notFound()  
      throw err  
    }  
  })**

This enables an HTTP **DELETE** request for a given ID to remove the entry from the data set. If there is no entry, the **del** method in **model.js** will cause an error indicating that the entry is not found. The route handler catches this error and rethrows the **fastify-sensible notFound** error so that a 404 error is generated. Any unknown error is rethrown so that it propagates as a 500 status response.

As we know there is an entry with an ID of 1, let's fetch that entry with a **GET** request to start out with:

**node -e "http.get('http://localhost:3000/bicycle/1', (res) => res.setEncoding('utf8').once('data', console.log))"**

This should output **{"brand":"Veloretti","color":"green"}**.

Now let's make a **DELETE** request to the same route:

**node -e "http.request('http://localhost:3000/bicycle/1', { method: 'delete', headers: {'content-type': 'application/json'}}, (res) => console.log(res.statusCode)).end()"**

This command should output the status code **204**, which indicates the record was successfully deleted.

If we attempt the same **GET** request we'll get a 404 response:

**node -e "http.get('http://localhost:3000/bicycle/1', (res) => res.setEncoding('utf8').once('data', console.log))"**

This should output: **{"statusCode":404,"error":"Not Found","message":"Not Found"}** which is the response body generated by the **fastify-sensible notFound** error.

If we attempt the same **DELETE** request again we'll likewise get a 404 response:

**node -e "http.request('http://localhost:3000/bicycle/1', { method: 'delete', headers: {'content-type': 'application/json'}}, (res) => console.log(res.statusCode)).end()"**

This will output **404** (because this command outputs the status code not the response body).

We now have a fully functioning RESTful service that performs backend Create, Read, Update and Delete (CRUD) operations.

One important thing to note: we haven't handled unexpected input. For instance, what if the **POST** body does not contain a **data** property? This scenario comes under the auspice of user input validation which is extremely important from a security perspective. We'll be revisiting our service in Chapter 9. Web Security: Handling User Input in order to apply user input validation to our routes.

In the next section we'll build our service again in Express.