

Anthony Nandaa

3- Sep-2018

Beginning API Development with Node.js

Build highly scalable, developer-friendly APIs for
the modern web with JavaScript and Node.js



Packt>

Beginning API Development with Node.js

Build highly scalable, developer-friendly APIs for the modern web with JavaScript and Node.js

Anthony Nandaa



BIRMINGHAM - MUMBAI

Beginning API Development with Node.js

Copyright © 2018 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Acquisitions Editor: Koushik Sen

Content Development Editors: Tanmayee Patil, Rutuja Yerunkar

Production Coordinator: Ratan Pote

First published: **July 2018**

Production reference: 1230718

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78953-966-0

www.packtpub.com



mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why Subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

PacktPub.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the Author

Anthony Nandaa is a senior software developer with more than 7 years of professional programming experience. He was introduced to programming 3 years before his career as a developer began, working with Pascal and VB 6. In his career so far, he has worked with multiple languages, such as Python, PHP, Go, and **full-stack JavaScript**.

In his current role, he leads a team of engineers working with Node.js and React for frontend development. He considers himself a lifelong learner, and lately, he has been learning Haskell for fun and to gain some insight into pure functional programming.

About the Reviewer

Sam Anderson is an electronic engineer turned developer currently working for an award-winning creative digital agency based in Sheffield, England. Having moved from the world of hardware design, he is passionate about creating fast, beautiful, and efficient frontend applications. You can follow him on Twitter at [@andomain.](#)

Packt Is Searching for Authors like You

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Title Page

Copyright and Credits

Beginning API Development with Node.js

Packt Upsell

Why Subscribe?

PacktPub.com

Contributors

About the Author

About the Reviewer

Packt Is Searching for Authors like You

Preface

Who This Book Is For

What This Book Covers

To Get the Most out of This Book

Download the Example Code Files

Conventions Used

Get in Touch

Reviews

1. Introduction to Node.js

The Basics of Node.js

Applications of Node.js

Activity: Running Basic Node.js Code

The Module System

Application Modularization

Module Categories

Built-In Modules

[npm – Third-Party Module Registry](#)

[Scanning for node_modules](#)

[Handy npm Commands](#)

[Local Modules](#)

[Activity: Using a Third-Party Package for the Previous math.js Code](#)

[Asynchronous Programming with Node.js](#)

[Callbacks](#)

[Promises](#)

[Async/Await](#)

[Activity: Transforming a Text File Using an Async Function](#)

[Summary](#)

2. Building the API - Part 1

Building a Basic HTTP Server

Setting up Hapi.js

Exercise 1: Building a Basic Hapi.js Server

Using an API Client

Returning JSON Strings

Exercise 2: Returning JSON

Using nodemon for Development Workflow

Exercise 3: Using nodemon

Setting up the Logger

Exercise 4: Setting up the Logger

Understanding Requests

A Look at HTTP Request Methods

Exercise 5: Getting a List of Resources

Exercise 6: Getting a Specific Resource

Exercise 7: Creating a New Todo with POST

Exercise 8: Updating a Resource with PUT

Exercise 9: Updating with PATCH

Exercise 10: Deleting a Resource with DELETE

Request Validation

Exercise 11: Validating a Request

Summary

3. Building the API - Part 2

Working with the DB Using Knex.js

Exercise 12: Setting up the Database

Exercise 13: Connecting to the Database

Exercise 14: Creating a Record

Exercise 15: Reading from the Database

Exercise 16: Updating a Record

Exercise 17: Deleting a Record

Exercise 18: Cleaning up the Code

Authenticating Your API with JWT

Exercise 19: Securing All the Routes

Exercise 20: Adding User Authentication

Authentication versus Authorization

Exercise 21: Implementing Authorization

Testing Your API with Lab

Exercise 22: Writing Basic Tests with Lab

Summary

Other Books You May Enjoy

Leave a review - let other readers know what you think

Preface

Using the **same framework** to build both server and client-side applications saves you time and money. This book teaches you how you can use JavaScript and Node.js to build highly scalable APIs that work well with lightweight cross-platform client applications. It begins with the basics of Node.js in the context of backend development, and quickly leads you through the creation of an example client that pairs up with a fully authenticated API implementation.

~~This book balances theory and exercises, and contains multiple open-ended activities that use real-life business scenarios for you to practice and apply your newly acquired skills in a highly relevant context.~~

~~We have included over 20 practical activities and exercises across 9 topics to reinforce your learning. By the end of this book, you'll have the skills and exposure required to get hands-on with your own API development project.~~

Who This Book Is For

This book is ideal for developers who already understand JavaScript and are looking for a quick no-frills introduction to API development with Node.js. Though prior experience with other server-side technologies such as Python, PHP, ASP.NET, Ruby will help, it's not essential to have a background in backend development before getting started.

What This Book Covers

[chapter 1](#), *Introduction to Node.js*, covers a few fundamental concepts in Node.js, basic Node.js code and run it from the Terminal, module system, its categories, and asynchronous programming model that is at the heart of how Node.js works, and what actually makes Node.js tick.

[chapter 2](#), *Building the API – Part 1*, covers building a basic HTTP server, setting up Hapi.js, building basic API with [Hapi.js](#) Framework, and fundamental concepts of web applications.

[chapter 3](#), *Building the API – Part 2*, covers introduction to [Knex.js](#) and how we can use it to connect and use the database, essential CRUD database methods, API authentication using the [JWT mechanism](#), [CORS mechanism](#), testing the API using [Lab library](#), and [test automation using Gulp.js](#).

To Get the Most out of This Book

1. Prior experience with other server-side technologies, such as Python, PHP, ASP.NET, and Ruby will be beneficial but is not mandatory.
2. This book will require a computer system. The minimum hardware requirements are 1.8 GHz or higher Pentium 4 (or equivalent) processor, 4 GB RAM, 10 GB hard disk, and a stable internet connection.
3. The software required are Visual Studio Code (<https://code.visualstudio.com/>), Node.js (8.9.1) (<https://nodejs.org/en/>), MySQL Workbench 6.3 (<https://www.mysql.com/products/workbench/>), and MySQL (<https://dev.mysql.com/downloads/mysql/>).

Download the Example Code Files

You can download the example code files for this book from your account at www.packtpub.com. If you purchased this book elsewhere, you can visit www.packtpub.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packtpub.com.
2. Select the SUPPORT tab.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/TrainingByPackt/BeginningAPIDevelopmentwithNode.js>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Conventions Used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "After this setup is done, we then start the server using the `server.start` method."

A block of code is set as follows:

```
| handler: (request, reply) =>  
| {  
|   return reply({ message: 'hello, world' });  
| }
```

Any command-line input or output is written as follows:

```
| node server.js
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Change the request type to POST."

Activity: These are scenario-based activities that will let you practically apply what you've learned over the course of a complete section. They are typically in the context of a real-world problem or situation.



Warnings or important notes appear like this.

Get in Touch

Feedback from our readers is always welcome.

General feedback: Email feedback@packtpub.com and mention the book title in the subject of your message. If you have questions about any aspect of this book, please email us at questions@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/submit-errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packtpub.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packtpub.com.

Introduction to Node.js

3 - Sep - 2018

17 - May - 2019

This chapter is designed to cover a few fundamental concepts in Node.js, as we lay a foundation for our subsequent chapters on API development.

Let's start this first chapter with a quick dive into how Node.js works and where it's being used lately. We will then have a look at its module system and its asynchronous programming model. Let's get started.

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

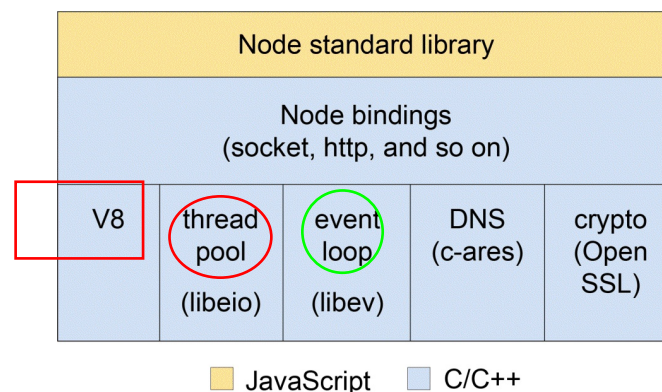
- Describe the basics of how Node.js works
- List the applications of Node.js in modern software development
- Describe the module system used by Node.js
- Implement basic modules for an application
- Explain the asynchronous programming basics in Node.js
- Implement a basic application using `async/await`

The Basics of Node.js

Node.js is an event-driven, server-side JavaScript environment. Node.js runs JS using the V8 engine developed by Google for use in their Chrome web browser. Leveraging V8 allows Node.js to provide a server-side runtime environment that compiles and executes JS at lightning speeds.

Node.js runs as a single-threaded process that acts upon *callbacks* and never blocks on the main thread, making it high-performing for web applications. A callback is basically a function that is passed to another function so that it can be called once that function is done. We will look into this in a later topic. This is known as the **single-threaded event loop model**. Other web technologies mainly follow the **multithreaded request-response** architecture.

The following diagram depicts the architecture of Node.js. As you can see, it's mostly C++ wrapped by a JavaScript layer. We will not go over the details of each component, since that is out of the scope of this chapter.



Node's goal is to offer an easy and safe way to build high-performance and scalable network applications in JavaScript.

Applications of Node.js

Node.js has the following four major applications:

- **Creating REST APIs:** We are going to look into this more in subsequent chapters
- **Creating real-time services:** Because of Node's asynchronous event-driven programming, it is well-suited to reactive real-time services
- **Building microservices:** Since Node.js has a very **lean core**, it is best suited to building microservices, since you will only add dependencies that you actually need for the microservices, as opposed to the glut that comes with other frameworks
- **Tooling:** For example, DevOps automations, and so on

Activity: Running Basic Node.js Code

Before You Begin

~~Open the IDE and the Terminal to implement this solution.~~

Aim

~~Learn how to write a basic Node.js file and run it.~~

Scenario

~~You are writing a very basic mathematical library with handy mathematical functions.~~

Steps for Completion

1. Create your project directory (folder), where all the code for this and other chapters will be kept. You can call it `beginning-nodejs` for brevity. Inside this directory, create another directory named `lesson-1`, and inside that, create another directory called `activity-a`. All this can be done using the following command:

```
|      mkdir -p beginning-nodejs/lesson-1/activity-a
```

2. Inside `activity-a`, create a file using `touch maths.js` command.
3. Inside this file, create the following functions:
 - `add`: This takes any two numbers and returns the sum of both, for example, `add(2, 5)` returns 7
 - `sum`: Unlike `add`, takes any number of numbers and returns their sum, for example, `sum(10, 5, 6)` returns 21
4. After these functions, write the following code to act as tests for your code:

```
|      console.log(add(10, 6)); // 16  
|      console.log(sum(10, 5, 6)); // 21
```

5. Now, on the Terminal, change directory to `lesson-1`. That's where we will be running most of our code from for the whole chapter.

6. To run the code, run the following command:

```
| node activity-a/math.js
```

The 16 and 21 values should be printed out on the Terminal.

Even though you can configure the IDE so that Node.js code be run at the click of a button, it's strongly recommend that you run the code from the Terminal to appreciate how Node.js actually works.



For uniformity, if you are using a Windows machine, then run your commands from the Git Bash Terminal.

For the reference solution, use the `math.js` file at `code/Lesson-1/activity-solutions/activity-a`.

The Module System

Let's have a look at Node's module system and the different categories of the Node.js modules.

Application Modularization

Like most programming languages, Node.js uses modules as a way of organizing code. The module system allows you to organize your code, hide information, and only expose the public interface of a component using `module.exports`.

Node.js uses the CommonJS specification for its module system:

- **Each file is its own module**, for instance, in the following example, `index.js` and `math.js` are both modules
- Each file has access to the current module definition using the `module` variable
- The export of the current module is determined by the `module.exports` variable
- To import a module, use the **globally** available `require` function

Let's look at a simple example:

```
// math.js file
function add(a, b)
{
  return a + b;
}
...
module.exports =
{
  add,
  mul,
  div,
};
// index.js file
const math = require('./math');
console.log(math.add(30, 20)); // 50
```



To call other functions such as `mul` and `div`, we'll use object destructuring as an alternative when requiring the module, for example, `const { add } = require('./math');`.

The code files for the section *The Module System* are placed at `Code/Lesson-1/b-module-system`.

Module Categories

We can place Node.js modules into three categories:

- **Built-in (native) modules:** These are modules that come with Node.js itself; you don't have to install them separately.
- **Third-party modules:** These are modules that are often installed from a package repository. npm is a commonly used package repository, but you can still host packages on GitHub, your own private server, and so on.
- **Local modules:** These are modules that you have created within your application, like the example given previously.

Built-In Modules

As mentioned earlier, these are modules that can be used straight-away without any further installation. All you need to do is to require them. There are quite a lot of them, but we will highlight a few that you are likely to come across when building web applications:

- `assert`: Provides a set of assertion tests to be used during unit testing
- `buffer`: To handle binary data
- `child_process`: To run a child process
- `crypto`: To handle OpenSSL cryptographic functions
- `dns`: To do DNS lookups and name resolution functions
- `events`: To handle events
- `fs`: To handle the filesystem
- `http` or `https`: For creating HTTP(s) servers
- `stream`: To handle streaming data
- `util`: To access utility functions like `deprecate` (for marking functions as deprecated), `format` (for string formatting), `inspect` (for object debugging), and so on

For example, the following code reads the content of the `lesson-1/temp/sample.txt` file using the in-built `fs` module:

```
const fs = require('fs');
let file = `${__dirname}/temp/sample.txt`;
fs.readFile(file, 'utf8', (err, data) =>
{
  if (err) throw err;
  console.log(data);
});
```

~~The details of this code will be explained when we look at asynchronous programming later in this chapter.~~

npm – Third-Party Module Registry

Node Package Manager (npm) is the package manager for JavaScript and the world's largest software registry, enabling developers to discover packages of reusable code.

~~To install an npm package, you only need to run the command `npm install <package-name>` within your project directory. We are going to use this a lot in the next two chapters.~~

Let's look at a simple example. If we wanted to use a package (library) like `request` in our project, we could run the following command on our Terminal, within our project directory: **npm install request**

To use it in our code, we require it, like any other module:

```
const request = require('request');
request('http://www.example.com', (error, response, body) =>
{
  if (error) console.log('error:', error); // Print the error if one occurred
  else console.log('body:', body); // Print the HTML for the site.
});
```

More details about npm can be found here: <https://docs.npmjs.com/>. Recently, a new package manager was released called **YARN** (<https://docs.npmjs.com/>), which is becoming increasingly popular.



~~When you run the `npm install <module-name>` command on your project for the first time, the `node_modules` folder gets created at the root of your project.~~

Scanning for node_modules

It's worth noting how Node.js goes about resolving a particular required module. For example, if a file `/home/tony/projects/foo.js` has a require call `require('bar')`, Node.js scans the filesystem for `node_modules` in the following order. The first `bar.js` that is found is returned:

- `/home/tony/projects/node_modules/bar.js`
- `/home/tony/node_modules/bar.js`
- `/home/node_modules/bar.js`
- `/node_modules/bar.js`

Node.js looks for `node_modules/bar` in the current folder followed by every parent folder until it reaches the root of the filesystem tree for the current file.



The module `foo/index.js` can be required as `foo`, without specifying `index`, and will be picked by default.

Handy npm Commands

Let's dive a little deeper into npm, by looking at some of the handy npm commands that you will often use:

- **npm init:** Initializes a Node.js project. This should be run at the root of your project and will create a respective `package.json` file. This file usually has the following parts (keys):
 - **name:** Name of the project.
 - **version:** Version of the project.
 - **description:** Project description.
 - **main:** The entry-point to your project, the main file.
 - **scripts:** This will be a list of other keys whose values will be the scripts to be run, for example, test, dev-server. Therefore, to run this script, you will only need to type commands such as `npm run dev-server`, `npm run test`, and so on.
 - **dependencies:** List of third-party packages and their versions used by the project. Whenever you do `npm install <package-name> --save`, this list is automatically updated.
 - **devDependencies:** List of third-party packages that are not required for production, but only during development. This will usually include packages that help to automate your development workflow, for example, task runners like gulp.js. This list is automatically updated whenever you do `npm install <package-name> --save-dev`.
 - **npm install:** This will install all the packages, as specified in the `package.json` file.
- **npm install <package-name> <options>:**
 - With the `--save` option, installs the package and saves the details in the `package.json` file.
 - With the `--save-dev` option, installs the package and saves the details in the `package.json`, under `devDependencies`.
 - With the `--global` option, installs the package globally in the whole

system, not only in the current system. Due to permissions, this might require running the command with administrator rights, for example,

```
sudo npm install <package-name> --global.
```

- `npm install <package-name>@<version>`, installs a specific version of a package. Usually, if a version is not specified, the latest version will be installed.
- `npm list`: Lists the packages that have been installed for the project, reading from what is installed in `node_modules`.
- `npm uninstall <package-name>`: Removes an installed package.
- `npm outdated`: Lists installed packages that are outdated, that is, newer versions have been released.

Local Modules

We have already looked at how local modules are loaded from the previous example that had `math.js` and `index.js`.

Since **JavaScript Object Notation (JSON)** is such an important part of the web, Node.js has fully embraced it as a data format, even locally. You can load a JSON object from the local filesystem the same way you load a JavaScript module. During the module loading sequence, whenever a `file.js` is not found, Node.js looks for a `file.json`.

```
See the example files in lesson-1/b-module-system/1-basics/load-json.js: const config
= require('./config/sample');
console.log(config.foo); // bar
```

Here, you will notice that once *required*, the JSON file is transformed into a JavaScript object implicitly. Other languages will have you read the file and perhaps use a different mechanism to convert the content into a data structure such as a map, a dictionary, and so on.



For local files, the extension is optional, but should there be a conflict, it might be necessary to specify the extension. For example, if we have both a `sample.js` and a `sample.json` file in the same folder, the `.js` file will be picked by default; it would be prudent to specify the extension, for example: `const config = require('./config/sample.json');`

When you run `npm install`, without specifying the module to install, npm will install the list of packages specified (under `dependencies` and `devDependencies` in the `package.json` file in your project). If `package.json` does not exist, it will give an error indicating that no such file has been found.

Activity: Using a Third-Party Package for the Previous math.js Code

Before You Begin

This activity will build upon the, *Running Basic Node.js* activity of this chapter.

Aim

If the argument is a single array, sum up the numbers, and if it's more than one array, first combine the arrays into one before summing up. We will use the `concat()` function from `lodash`, which is a third-party package that we will install.

Scenario

We want to create a new function, `sumArray`, which can sum up numbers from one or more arrays.

Steps for Completion

1. Inside `Lesson-1`, create another folder called `activity-b`.
2. On the Terminal, change directory to `activity-b` and run the following command:
3. This will take you to an interactive prompt; just press *Enter* all the way, leaving the answers as suggested defaults. The aim here is for us to get a `package.json` file, which will help us organize our installed packages.
4. Since we will be using `lodash`, let's install it. Run the following command:

```
| npm init
```

```
| npm install lodash --save
```

Notice that we are adding the `--save` option on our command so that the package installed can be tracked in `package.json`. When you open the

`package.json` file created in step 3, you will see an added dependencies key with the details.

5. Create a `math.js` file in the `activity-b` directory and copy the `math.js` code from *Activity, Running Basic Node.js* into this file.
6. Now, add the `sumArray` function right after the `sum` function.
7. Start with requiring `lodash`, which we installed in step 4, since we are going to use it in the `sumArray` function:

```
|     const _ = require('lodash');
```

8. The `sumArray` function should call the `sum` function to reuse our code. Hint: use the spread operator on the array. See the following code:

```
|     function sumArray()
|     {
|         let arr = arguments[0];
|         if (arguments.length > 1)
|         {
|             arr = _.concat(...arguments);
|         }
|         // reusing the sum function
|         // using the spread operator (...) since
|         // sum takes an argument of numbers
|         return sum(...arr);
|     }
| }
```

9. At the end of the file, export the three functions, `add`, `sum`, and `sumArray` with `module.exports`.
10. In the same `activity-b` folder, create a file, `index.js`.
11. In `index.js` file, *require* `./math.js` and go ahead to use `sumArray`:

```
|     // testing
|     console.log(math.sumArray([10, 5, 6])); // 21
|     console.log(math.sumArray([10, 5], [5, 6], [1, 3])) // 30
```

12. Run the following code on the Terminal:

```
|     node index.js
```

You should see 21 and 30 printed out.



The solution files are placed at `Code/Lesson-1/activitysolutions/activity-b`.

Asynchronous Programming with Node.js

Let's have a look at asynchronous programming model that is at the heart of how Node.js works.

Callbacks

Callbacks are functions that are executed asynchronously, or at a later time. Instead of the code reading top to bottom procedurally, asynchronous programs may execute different functions at different times based on the order and speed of earlier functions.

Since JavaScript treats functions like any other object, we can pass a function as an argument in another function and alter execute that passed-in function or even return it to be executed later.

We saw such a function previously when we were looking at the `fs` module in *The Module System* section. Let's revisit it:

```
const fs = require('fs');
let file = `${__dirname}/temp/sample.txt`;
fs.readFile(file, 'utf8', (err, data) =>
{
  if (err) throw err;
  console.log(data);
});
```



The code files for Asynchronous Programming with Node.js are placed at `Code/Lesson-1/c-async-programming`.

On line 3, we use a variable part of the globals, `__dirname`, which basically gives us the absolute path of the directory (folder) in which our current file (`read-file.js`) is, from which we can access the `temp/sample.txt` file.

Our main point of discussion is the chunk of code between lines 5 and 8. Just like most of the methods you will come across in Node.js, they mostly take in a callback function as the last argument.

Most callback functions will take in two parameters, the first being the error object and the second, the results. For the preceding case, if file reading is successful, the error object, `err`, will be null and the contents of the file will be returned in the data object.

Let's break down this code for it to make more sense:

```
const fs = require('fs');
let file = `${__dirname}/temp/sample.txt`;
const callback = (err, data) =>
{
  if (err) throw err;
  console.log(data);
};
fs.readFile(file, 'utf8', callback);
```

Now, let's look at the asynchronous part. Let's add an extra line to the preceding code:

```
const fs = require('fs');
let file = `${__dirname}/temp/sample.txt`;
const callback = (err, data) =>
{
  if (err) throw err;
  console.log(data);
};
fs.readFile(file, 'utf8', callback);
console.log('Print out last!');
```

See what we get as a print out:

```
Print out last!
hello,
world
```

How come `Print out last!` comes first? This is the whole essence of asynchronous programming. Node.js still runs on a single thread, line 10 executes in a non-blocking manner and moves on to the next line, which is `console.log('Print out last!')`. Since the previous line takes a long time, the next one will print first. Once the `readFile` process is done, it then prints out the content of file through the callback.

Promises

Promises are an alternative to callbacks for delivering the results of an asynchronous computation. First, let's look at the basic structure of promises, before we briefly look at the advantages of using promises over normal callbacks.

Let's rewrite the code above with promises:

```
const fs = require('fs');
const readFile = (file) =>
{
  return new Promise((resolve, reject) =>
  {
    fs.readFile(file, 'utf8', (err, data) =>
    {
      if (err) reject(err);
      else resolve(data);
    });
  });
}
// call the async function
readFile(`${__dirname}/../temp/sample.txt`)
  .then(data => console.log(data))
  .catch(error => console.log('err: ', error.message));
```

This code can further be simplified by using the `util.promisify` function, which takes a function following the common Node.js callback style, that is, taking an `(err, value) => ...` callback as the last argument and returning a version that returns promises:

```
const fs = require('fs');
const util = require('util');
const readFile = util.promisify(fs.readFile);
readFile(`${__dirname}/../temp/sample.txt`, 'utf8')
  .then(data => console.log(data))
  .catch(error => console.log('err: ', error));
```

From what we have seen so far, promises provide a standard way of handling asynchronous code, making it a little more readable.

What if you had 10 files, and you wanted to read all of them? `Promise.all` comes to the rescue. `Promise.all` is a handy function that enables you to run asynchronous functions in parallel. Its input is an array of promises; its output is a single promise that is fulfilled with an array of the results:

```
const fs = require('fs');
const util = require('util');
const readFile = util.promisify(fs.readFile);
const files = [
  'temp/sample.txt',
  'temp/sample1.txt',
  'temp/sample2.txt',
];
// map the files to the readFile function, creating an
// array of promises
const promises = files.map(file => readFile(`${__dirname}/${file}`, 'utf8'));
Promise.all(promises)
  .then(data =>
    {
      data.forEach(text => console.log(text));
    })
  .catch(error => console.log('err: ', error));
```

Async/Await

This is one of the latest additions to Node.js, having been added early in 2017 with version 7.6, providing an even better way of writing asynchronous code, making it look and behave a little more like synchronous code.

Going back to our file *reading* example, say you wanted to get the contents of two files and concatenate them in order. This is how you can achieve that with

```
async/await: const fs = require('fs');
const util = require('util');
const readFile = util.promisify(fs.readFile);
async function readFiles()
{
  const content1 = await readFile(`${__dirname}../temp/sample1.txt`);
  const content2 = await readFile(`${__dirname}../temp/sample2.txt`);
  return content1 + '\n - and - \n\n' + content2;
}
readFiles().then(result => console.log(result));
```

In summary, any asynchronous function that returns a promise can be *awaited*.

Activity: Transforming a Text File Using an Async Function

Before You Begin

You should have already gone through the previous activities.

Aim

Read the file (using `fs.readFile`), `in-file.txt`, properly case format the names (using the `lodash` function, `startCase`), then sort the names in alphabetical order and write them out to a separate file `out-file.txt` (using `fs.writeFile`).

Scenario

We have a file, `in-file.txt`, containing a list of peoples' names. Some of the names have not been properly case formatted, for example, `john doe` should be changed to `John Doe`.

Steps for Completion

1. In `Lesson-1`, create another folder called `activity-c`.
2. On the Terminal, change directory to `activity-c` and run the following command:

```
| npm init
```
3. Just like in the previous activity, this will take you to an interactive prompt; just press *Enter* all the way, leaving the answers as suggested defaults. The aim here is for us to get a `package.json` file, which will help us organize our installed packages.
4. Since we will be using `lodash` here too, let's install it. Run, `npm install lodash -save`.
5. Copy the `in-file.txt` file provided in the `student-files` directory into your `activity-c` directory.

6. In your `activity-c` directory, create a file called `index.js`, where you will write your code.
7. Now, go ahead and implement an `async` function `transformFile`, which will take the path to a file as an argument, transform it as described previously (under *Aim*), and write the output to an output file provided as a second parameter.
8. On the Terminal, you should indicate when you are reading, writing, and done, for example:
 - `reading file: in file.txt`
 - `writing file: out file.txt`
 - `done`



You should read the quick reference documentation on `fs.writeFile` since we haven't used it yet. However, you should be able to see its similarity with `fs.readFile`, and convert it into a promise function, as we did previously.

The solution files are placed at `code/Lesson-1/activitysolutions/activity-c`.

Summary

In this chapter, we went through a quick overview of Node.js, seeing how it looks under the hood.

We wrote basic Node.js code and ran it from the Terminal using the Node.js command.

We also looked at module system of Node.js, where we learnt about the three categories of Node.js modules, that is, in-built, third-party (installed from the npm registry), and local modules, and their examples. We also looked at how Node.js resolves a module name whenever you *require* it, by searching in the various directories.

We then finished off by looking at the asynchronous programming model that is at the heart of how Node.js works, and what actually makes Node.js tick. We looked at the three main ways you can write asynchronous code: using *callbacks*, *Promises*, and the new *async/await* paradigm.

The foundation is now laid for us to go ahead and implement our API using Node.js. Most of these concepts will crop up again as we build our API.

17 - May - 2019

Building the API - Part 1

This chapter is meant to introduce the students to API building using Node.js. We will start by building a basic HTTP server to gain an understanding of how Node.js works.

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

- Implement a basic HTTP server using the Node.js built-in `http` module
- Implement a basic `Hapi.js` setup for an API
- Describe the basic HTTP verbs and how they differ from each other
- Implement various routes for the API, making use of the different HTTP verbs
- Implement logging the web application
- Validating API requests

Building a Basic HTTP Server

Let's begin by looking at the basic building blocks of a Node.js web application. The built-in `http` module is the core of this. However, from the following example, you will also appreciate how basic this can be.

Save the following code in a file called `simple-server.js`:

```
const http = require('http');
const server = http.createServer((request, response) =>
{
  console.log('request starting...');
  // respond
  response.write('hello world!');
  response.end();
});
server.listen(5000);
console.log('Server running at http://127.0.0.1:5000');
```

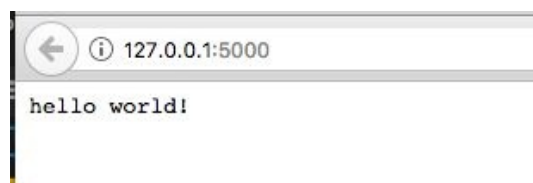


Use the `simple-server.js` file for your reference at [Code/Lesson-2](#).

Now, let's run the file:

```
| node simple-server.js
```

When we go to the browser and visit the URL in the example, this is what we get:



Setting up Hapi.js

Hapi.js (HTTP API), is a rich framework for building applications and services, focusing on writing reusable application logic. There are a number of other frameworks; notable among them is Express.js. However, from the ground up, Hapi.js is optimized for API building, and we will see this shortly when building our application.

Exercise 1: Building a Basic Hapi.js Server

In this exercise, we're going to build a basic HTTP server like the one before, but now with Hapi.js. You will notice how most of the things are done for us under the hood with Hapi.js. However, Hapi.js is also built on top of the `http` module.

For the rest of the exercises, from the first exercise of [chapter 3, Building the API – Part 2](#), we will be building on top of each exercise as we progress. So, we might need to go back and modify previous files and so forth:

1. In your `Lesson-2` folder, create a subfolder called `hello-hapi`.



Use the `exercise-b1` folder for your reference at `Code/Lesson-2`.

2. On the Terminal, change directory to the root of the `hello-hapi` folder.
3. Initialize it as a basic Node.js project and run the following command:

```
| npm init -y
```


4. Create a file, `server.js`.
5. Install Hapi.js by executing the following command:

```
| npm install hapi --save
```

6. In the file, write the following code:

```
const Hapi = require('hapi');
// create a server with a host and port
const server = new Hapi.Server();
server.connection
({
  host: 'localhost',
  port: 8000,
});
// Start the server
server.start((err) =>
{
  if (err) throw err;
  console.log(`Server running at: ${server.info.uri}`);
});
```



 Use the `server.js` file for your reference at `Code/Lesson-2/exercise-b1`.

Let us try to understand the code:

- We first start by requiring the Hapi.js framework that we just included.



Recall our subtopic, The Module System, in chapter 1, Introduction to Node.js? We looked at third-party modules—this is one of them.

- We then create a server by initializing the `Server` class, hence a new `Hapi.Server()`.
- We then bind that server on a specific host (`localhost`) and port (`8000`).
- After that, we create an example route, `/`. As you can see, for each route created, we have to specify three major things (as keys of an object passed to the `server.route` method):
 - `method`: This is the HTTP method for that route. We're going to look more deeply at the types of HTTP verbs in a later section. For our example, we're using `GET`. Basically, as the name suggests, this gets stuff/resources from the server.
 - `path`: This is the path on the server to the particular resource we are getting.
 - `handler`: This is a closure (anonymous function) that does the actual getting.



We're going to look at another extra key, called `config`, in our main project.

- After this setup is done, we then start the server using the `server.start` method. This method accepts a closure (callback function) that is called once the server has started. In this function, we can check whether any errors occurred while starting the server.

7. Run the server by going to the Terminal, and run the following command:

```
| node server.js
```

8. You should see this printed on the Terminal:

```
| Server running at: http://localhost:8000
```

You should see something similar to this at `http://localhost:8000`:



Open another Terminal, change directory to the same project folder, and run the same command, `node server.js`. We'll get this error: `Error: listen EADDRINUSE 127.0.0.1:8000`.



The reason we get this error is because we can only have one server running on a particular port of our host. Remember that the host IP `127.0.0.1` is what we refer to as `localhost`. If `(err) throw err;` is the line which throws the error.

We can fix this by changing the port number of our second server to something like `8001`. However, as best practice, other than keep changing the code, we can pass the port number as a Terminal argument, that is, running the app as, `node server.js <port-number>`, then changing our code (in the `port` section) to, `port: process.argv[2] || 8000,`

Here, we're saying, if the port is provided as the first argument of the script, use that, otherwise, use `8000` as the port number. Now, when you run: `node server.js 8002`, the server should run okay from `localhost:8002`.



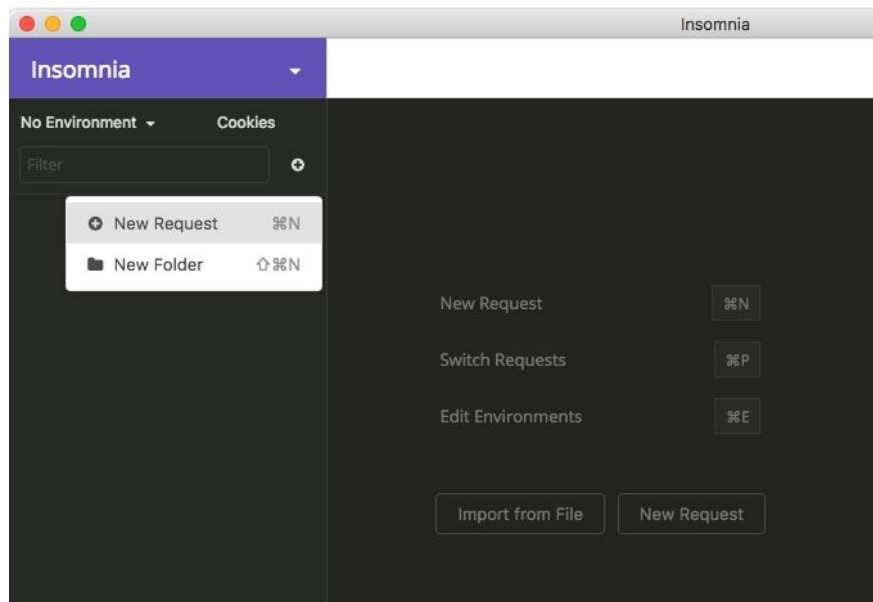
For the `process.argv` array, index `0` is the program running the script, `node` and index `1` is the script being run, `server.js`. Arguments passed to the script are therefore counted from index `2` onwards. You can read more about `process.argv` here later on.

Using an API Client

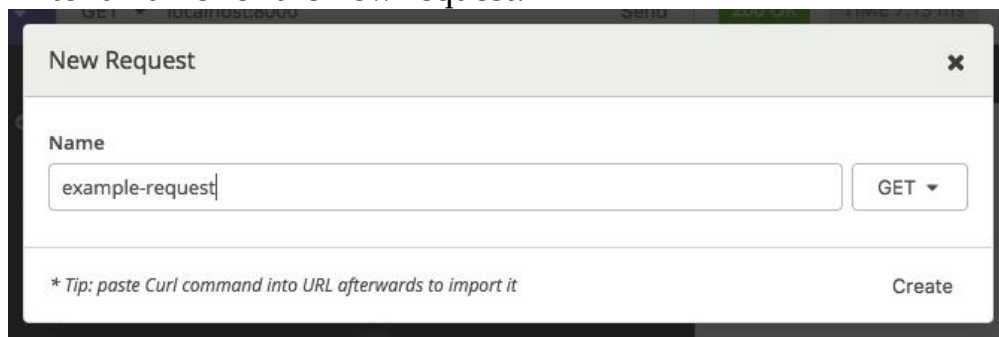
For us to utilize the client to the fullest, to be able to do all the request types (GET, POST, UPDATE, and so on), we will need to have an API client. There are a number out there, but we recommend either Postman (<https://www.getpostman.com/>) or Insomnia (<https://insomnia.rest/>). For our examples, we will be using Insomnia.

After installing Insomnia, add a GET request to `http://localhost:8000`:

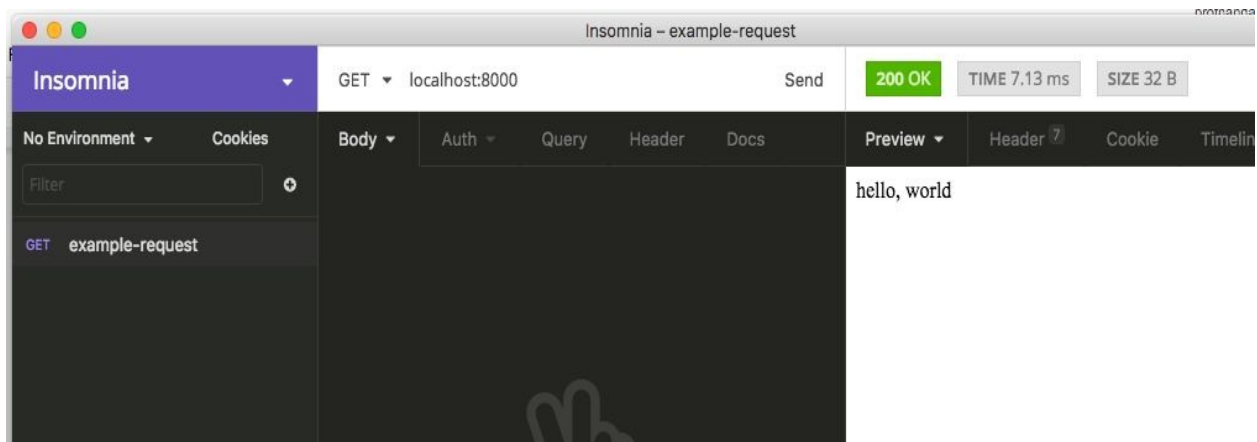
1. We will begin by creating a *request* page for Insomnia, where we will be making all of our requests:



Enter a name for the new request:



2. We will then make our request by typing the route and clicking on Send:



 When we change the type from *GET* to *POST*, and click on *Send*, we get a 404 error because, on our server, we currently have only the *GET* method defined for route */*.

Returning JSON Strings

As we are now building our API, we need a formal way of representing our data in our request, by sending or receiving it. **JavaScript Object Notation (JSON)** is the conventional data-interchange format for REST APIs.

One thing to note about JSON is that it started from JavaScript and is now widely adopted across other languages. So, when it comes to Node.js, you will see how using JSON becomes so easy and natural.

Exercise 2: Returning JSON

1. Let's go back to our `server.js` file from *Exercise 1, Building a Basic Hapi.js Server*.



Use the `exercise-b2` folder for your reference at `Code/Lesson-2`.

2. To return JSON for our `/` route, all we need to change is our returned string to an object:

```
handler: (request, reply) =>
{
  return reply({ message: 'hello, world' });
}
```

3. Stop the server by going to the Terminal where the server is running and pressing `Ctrl + C`. Then, start the server again to effect the changes by running the following command:

```
node server.js
```

4. Now go back to Insomnia and do another GET request. You can see that this is effectively changed into a JSON string:

```
{
  "message": "hello, world"
}
```

This comes out-of-the-box in Hapi.js, while with some frameworks, such as Express.js, you have to use a `json` function to do the conversion.

Using nodemon for Development Workflow

You will have noticed that, after making the changes in first exercise, we had to go back and stop the server and start over again. Doing this every time you make a change to your code becomes very cumbersome. Luckily, tooling comes to our rescue.

There is a Node.js package called `nodemon`, which can help restart the server automatically whenever there is a change in our files.

Exercise 3: Using nodemon

In this exercise, we're going to introduce a Node module known as `nodemon`, which we will be using to run our web server. This makes it possible for the server to automatically reload when we make changes to it, therefore avoiding the tediousness of stopping the server and starting it over again manually whenever we make changes to our server:

1. Go back to the Terminal and stop the server (press *Ctrl + C*), then run the following command.
2. We will need to install this package globally (remember that you might need some administrative rights, so in Unix systems, you need to run the command as `sudo`):

```
| npm install --global nodemon
```

3. Once installation is complete, we can run with `nodemon`:

```
| nodemon server.js
```

You should get something like this:

```
| [nodemon] 1.12.1
| [nodemon] to restart at any time, enter `rs`
| [nodemon] watching: *.*
| [nodemon] starting `node server.js`
| Server running at: http://localhost:8000
```

Setting up the Logger

Logging is a very important component of any web application. We need a way of preserving the history of the server so that we can come back any time and see how it was serving requests.

And, most of all, you don't want logging to be an afterthought, only being implemented after you come across a production bug that makes your web app crash when you are trying to figure out where the problem is exactly.

Hapi.js has a minimal logging functionality built in, but if you need an extensive one, a good example is called **good** (<https://github.com/hapijs/good>).

Exercise 4: Setting up the Logger

In this exercise, we're going to add a logging mechanism on the web server we have created, so that each request and server activity can be easily tracked through the logs:

1. Let's go back to our project from *Exercise 2: Returning JSON*.



Use the `exercise-b4` folder for your reference at `Code/Lesson-2`.

2. We first need to install a couple of packages that will help with our logging (`good` and `good-console`). Run the following command:

```
npm install --save good good-console
```



`good-console` is what we call a write stream. There are other write streams that work with `good`, but, for simplicity, we won't look at them. You can check <https://github.com/hapijs/good> for more information.

3. We will then modify our `server.js` code to configure our logging. First, by requiring `good` just after `Hapi.js`:

```
const Hapi = require('hapi');
const good = require('good');
```

4. Then, registering it with the server just before we start the server:

```
// set up logging
const options = {
  ops: {
    interval: 100000,
  },
  reporters: {
    consoleReporters: [
      { module: 'good-console' },
      'stdout',
    ],
  },
};
...
});
```



Use the `server.js` file for your reference at `Code/Lesson-2/exercise-b4`.

5. If you are still running the server with `nodemon`, by now, you will start seeing

the server logs being updated periodically on the Terminal; something similar to:

```
171102/012027.934, [ops] memory: 34Mb, uptime (seconds):  
100.387, load: [1.94580078125,1.740234375,1.72021484375]  
171102/012207.935, [ops] memory: 35Mb, uptime (seconds):  
200.389, load: [2.515625,2.029296875,1.83544921875]  
...
```

6. Now, go back to Insomnia and try to do another GET request on `localhost:8000/`. You will see an extra log has been created showing the time the request was made (`timestamp`), the route, the method (`get`), the status code (`200`), and the time taken for the request:

```
171102/012934.889, [response] http://localhost:8000: get /{} 200 (13ms)
```



The time taken comes in very handy when you are trying to optimize the performance of your server, seeing which requests take longer than expected to be served.

Understanding Requests

Let's have a look at the concept of request and the different HTTP request methods.

A Look at HTTP Request Methods

Having set up our server, we are ready to start building our API. The routes are basically what constitute the actual API.

We will first look at HTTP request methods (sometimes referred to as *HTTP verbs*), then apply them to our API using a simple *todo list* example. We will look at five major ones:

- **GET**: Requests a representation of the specified resource. Requests using `GET` should only retrieve data, and should not be used to make changes to resources.
- **POST**: Is used to submit an entry to a specified resource, often causing a change of state.
- **PUT**: Replaces all current representations of the target resource with the request payload.
- **DELETE**: Deletes the specified resource.
- **PATCH**: Used to apply partial modifications to a resource.

In the following exercises, we're going to rewrite our previous code where we had hardcoded our data so that we can work with real and dynamic data coming directly from the database.

Exercise 5: Getting a List of Resources

1. Let's go back to the project from *Exercise 4: Setting up the Logger*.



Use the `exercise-c1` folder for your reference at `Code/Lesson-2`.

2. Since we are going to have various routes, it would be prudent to now split our routes to a separate file for the sake of organization. Within the project, create a subfolder called `routes`.
3. Inside the created folder, create a file called `todo.js`. In `todo.js`, this is where we are going to have all our routes for the `todo` resource. This file (module) will export a list of routes.
4. Let's start by doing a simple route that returns a list of todos on a `GET` request:

```
const todoList = [
  {
    title: 'Shopping',
    dateCreated: 'Jan 21, 2018',
    list: [
      {
        text: 'Node.js Books', done: false },
      ...
    ]
  },
  {
  }
];
```



Use the `todo.js` file for your reference at `Code/Lesson-2/exercise-c1/routes`.

5. We then go back to our `server.js` file, require the `todo` route module, and register it with our server using the `server.route` method:

```
const routes = {};
routes.todo = require('./routes/todo')
// create a server with a host and port
const server = new Hapi.Server();
server.connection(
  {
    host: 'localhost',
```

```
    port: process.argv[2] || 8000,  
  });  
  server.route(routes.todo);
```



Use the `server.js` file for your reference at `Code/Lesson-2/exercise-c1`.

6. Using Insomnia, do a GET request to `http://localhost:8000/todo`. You should see this returned:

```
[  
  {  
    "title": "Shopping",  
    "dateCreated": "Jan 21, 2018",  
    "list": [  
      {  
        "text": "Node.js Books",  
        "done": false  
      },  
      {  
        "text": "MacBook",  
        "done": false  
      },  
      {  
        "text": "Shoes",  
        "done": false  
      }  
    ]  
  },  
  {  
    
```

Exercise 6: Getting a Specific Resource

1. Now, let's try and get a specific todo. Since we don't have a database with IDs, we will take the indices to be IDs, `[0]` being 1, and so on.



Use the `exercise-c1` folder for your reference at `Code/Lesson-2`.

2. Let's add a route for that. Notice that we use `{<parameter-key>}` as a way of passing request parameters to our `route` function, then get it through `request.params.id`:

```
module.exports = [  
  {  
    method: 'GET',  
    path: '/todo',  
    ...  
    handler: (request, reply) => {  
      const id = request.params.id - 1;  
      // since array is 0-based index  
      return reply(todoList[id]);  
    }  
  },  
];
```



Use the `todo.js` file for your reference at `Code/Lesson-2/exercise-c1/routes`.

3. Go to Insomnia and do a `GET` request to `http://localhost:8000/todo/1`. You should see this:

```
{
  "title": "Shopping",
  "dateCreated": "Jan 21, 2018",
  "list": [
    {
      "text": "Node.js Books",
      "done": false
    },
    {
      "text": "MacBook",
      "done": false
    },
    {
      "text": "Shoes",
      "done": false
    }
  ]
}
```

Exercise 7: Creating a New Todo with POST

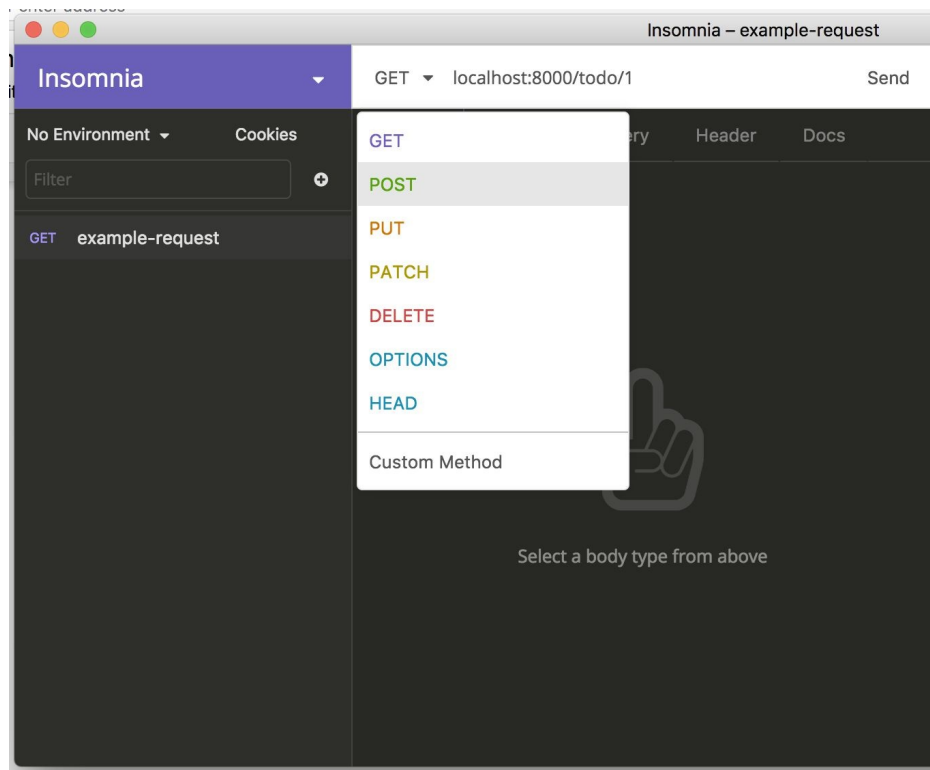
1. Now let's add a new todo. This is where `POST` comes in. A `POST` request should always come with a payload which is the data that is being *posted*. We will add a new route to handle this:

```
module.exports = [  
  // previous code  
  {  
    method: 'POST',  
    path: '/todo',  
    handler: (request, reply) => {  
      const todo = request.payload;  
      todoList.push(todo);  
      return reply({ message: 'created' });  
    },  
    ...  
  }  
];
```

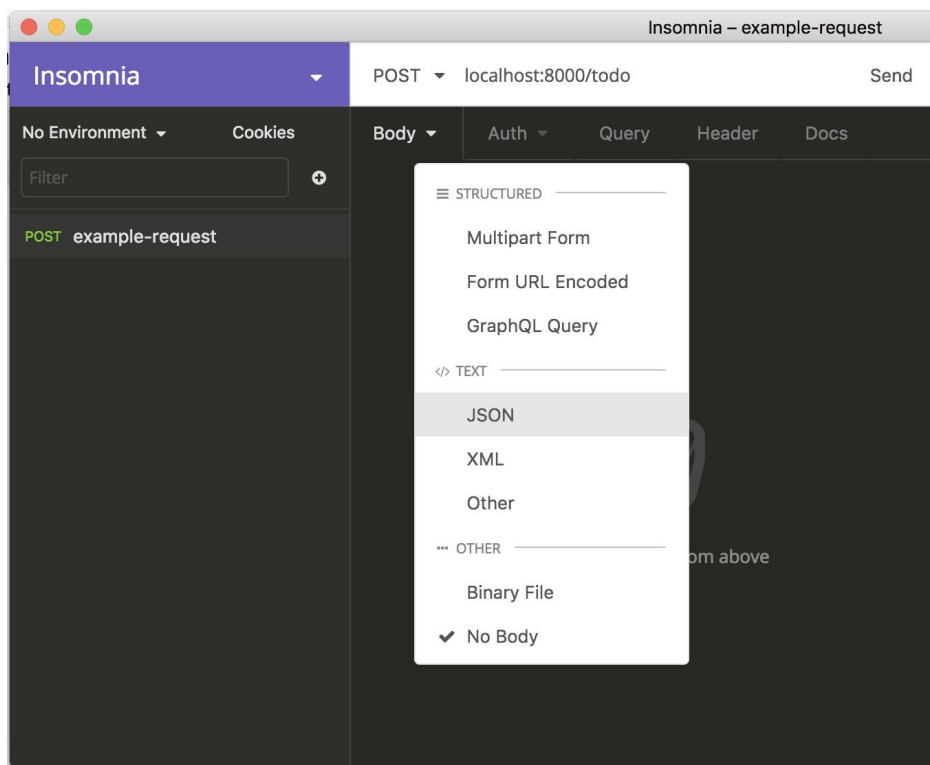


Use the `todo.js` file for your reference at `Code/Lesson-2/exercise-c1/routes`.

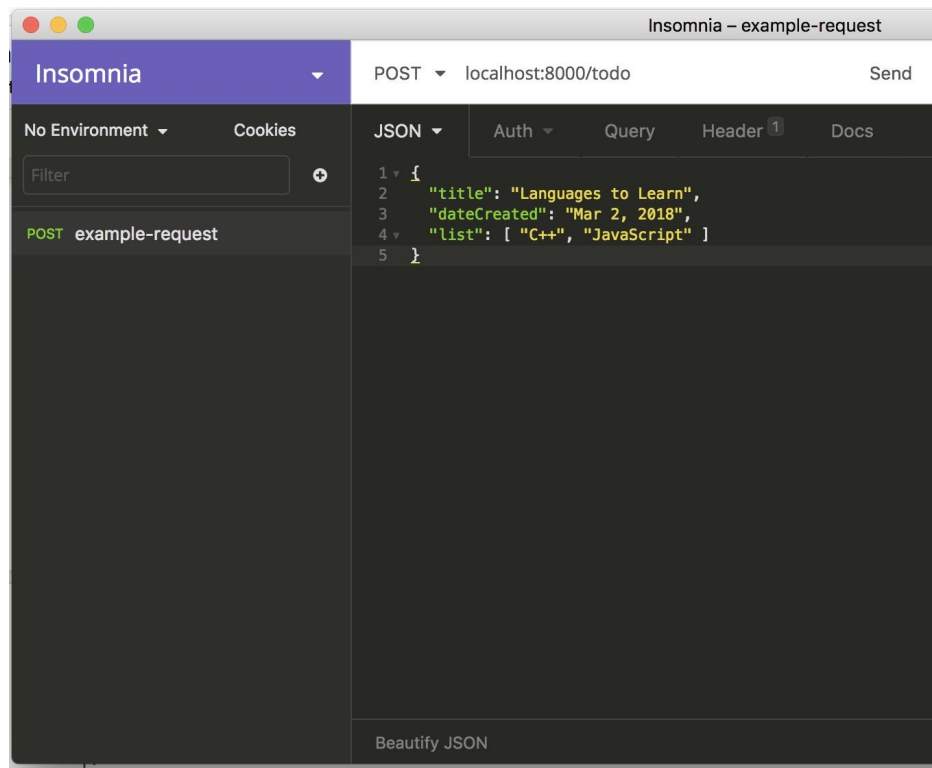
2. On Insomnia:
 1. Change the request type to POST:



2. Change the request body to JSON:



3. Add the request body and the URL appropriately:



3. When you post the request, you should see this as the response:

```
{
  "message": "created"
}
```

4. Now, when you do a GET request to <http://localhost:8000/todo>, you should see the newly created todo appear as part of the response:

```
[
  ...
  {
    "title": "Languages to Learn",
    "dateCreated": "Mar 2, 2018",
    "list": [
      "C++",
      "JavaScript"
    ]
  }
]
```


Exercise 8: Updating a Resource with PUT

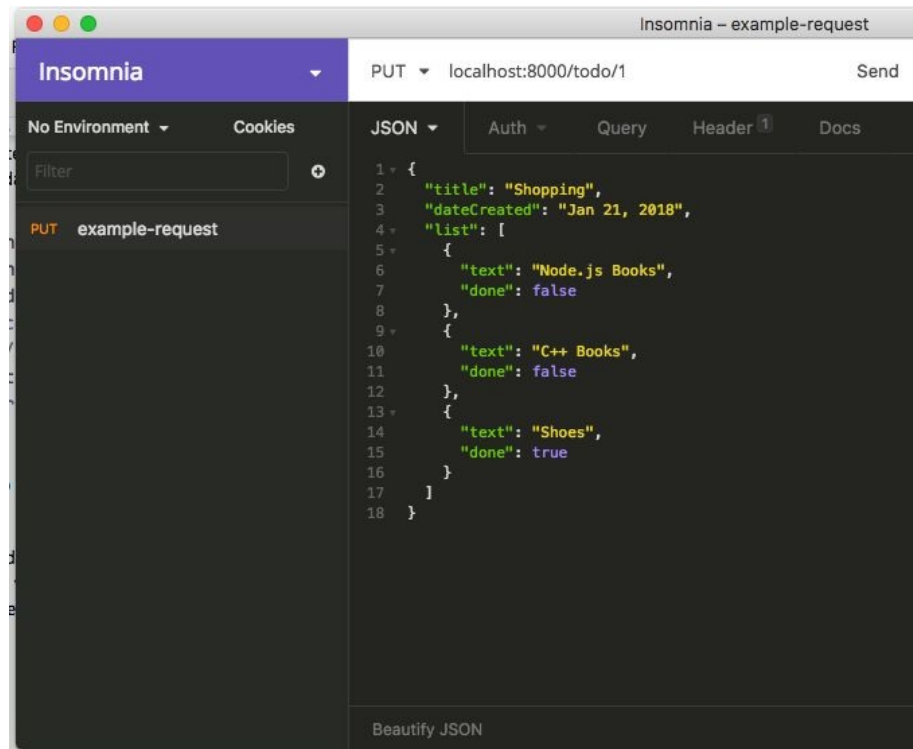
1. If we wanted to update, say, the first todo list, conventionally, `PUT` requires us to send the whole updated todo resource. Now let's create a `PUT` route:

```
{
  method: 'PUT',
  path: '/todo/{id}',
  handler: (request, reply) => {
    const index = request.params.id - 1;
    // replace the whole resource with the new one
    todoList[index] = request.payload;
    return reply({ message: 'updated' });
  }
}
```



Use the `todo.js` file for your reference at `Code/Lesson-2/exercise-c1/routes`.

2. Now go to Insomnia and make the request. Remember to change the request type to `PUT`:



3. You should see the following response:

```
{
  "message": "updated"
}
```

4. And when you do a GET on <http://localhost:8000/todo/1>, you should get the updated resource:

```
{
  "title": "Shopping",
  "dateCreated": "Jan 21, 2018",
  "list": [
    {
      "text": "Node.js Books",
      "done": false
    },
    {
      "text": "C++ Books",
      "done": false
    },
    {
      "text": "Shoes",
      "done": true
    }
  ]
}
```


Exercise 9: Updating with PATCH

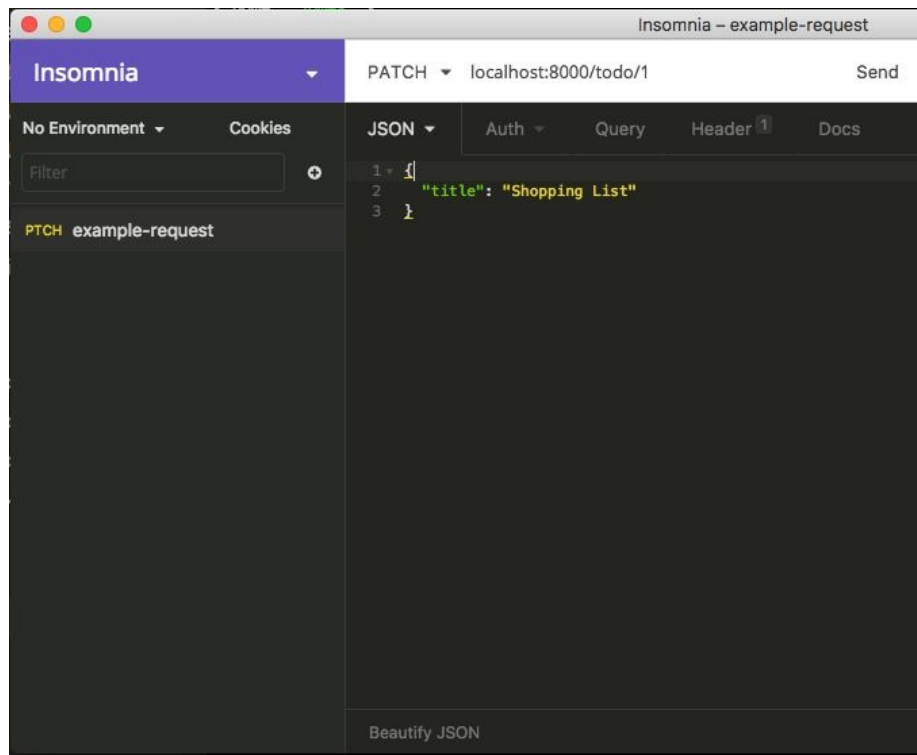
1. You will realize that, in our previous exercise, we had to post the whole resource just to change a part of it. A better way of doing this is using `PATCH`, so that the payload only contains what is required. Let's now create a `PATCH` route:

```
{
  method: 'PATCH',
  handler: (request, reply) =>
  {
    ...
    Object.keys(request.payload).forEach(key =>
    {
      if (key in todo)
      {
        todo[key] = request.payload[key];
      }
    })
    ...
    return reply({ message: 'patched' });
  },
}
```



Use the `todo.js` file for your reference at `code/Lesson-2/exercise-c1/routes`.

2. Now, you can provide any of the keys and their values, and they will be updated respectively. For example, make the following request, only changing the title of the first todo:



3. You should get the following response:

```
{
  "message": "patched"
}
```

4. And when you do a GET on <http://localhost:8000/todo/1>, you should get the updated resource:

```
{
  "title": "Shopping List",
  "dateCreated": "Jan 21, 2018",
  "list": [
    {
      "text": "Node.js Books",
      "done": false
    },
    {
      "text": "MacBook",
      "done": false
    },
    {
      "text": "Shoes",
      "done": true
    }
  ]
}
```


Exercise 10: Deleting a Resource with DELETE

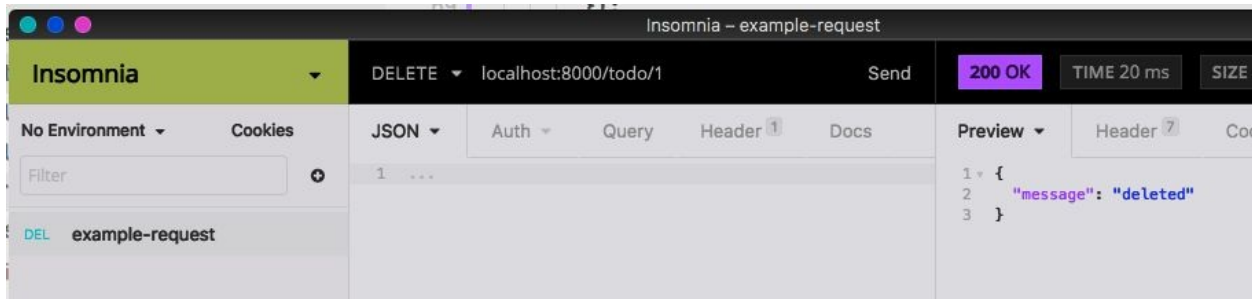
1. When we want to delete a resource, we use the `DELETE` method. Let's create a `DELETE` route:

```
{
  method: 'DELETE',
  path: '/todo/{id}',
  handler: (request, reply) => {
    const index = request.params.id - 1;
    delete todoList[index]; // replaces with `undefined`
    return reply({ message: 'deleted' });
  },
},
```



Use the `exercise-c1` folder for your reference at `Code/Lesson-2`.

2. Now go to Insomnia and test it—you should get this response:



3. Now try accessing the previously deleted resources—you should get a `404` error. However, in our previous `GET` route (in *Exercise 6: Getting a Specific Resource*), we did not cater for this, so let's go and make a modification to OUR `GET: /todo/{id}` route:

```
{
  method: 'GET',
  path: '/todo/{id}',
  handler: (request, reply) => {
    const id = request.params.id - 1;
    // should return 404 error if item is not found
    if (todoList[id]) return reply(todoList[id]);
    return reply({ message: 'Not found' }).code(404);
  },
},
```

```
}  
}
```

Use the `todo.js` file for your reference at `Code/Lesson-2/exercise-c1/routes`.



Don't worry about the status code, `404`, if you have never come across it. We are going to go through the major status codes in our last subsection of this section.

4. Remember, the server will reload this time, therefore, the deleted resource will still be brought back, so go back and repeat *step 2*.
5. When you now do a `GET` request to `http://localhost:8000/todo/1`, you should see this:

```
{  
  "message": "Not found"  
}
```



Short Closing Note on Insomnia

You should be able to access all your previous requests under *History*. Click on the *Time* icon in the top-right corner.

Request Validation

We will need to validate the incoming requests to make sure that they conform to what the server can handle.

This is one of the places I see Hapi.js shining above other frameworks. In Hapi.js, you hook in validation as a configuration object as part of the `route` object. For validation, we will use the Joi library, which works well with Hapi.js.

Exercise 11: Validating a Request

In this exercise, we are going to see the concept of *request validation* in action. We will write a validation for one of the routes as an example, but the same could be applied across the other routes:

1. For example, if we go back to the `POST` route from *Exercise 1: Building a Basic Hapi.js Server*, we can post an empty payload and still get status code `200`! Clearly, we need a way of validating this.
2. Let's start by installing Joi:

```
| npm install joi --save
```



Use the `exercise-c2` folder for your reference at `Code/Lesson-2`.

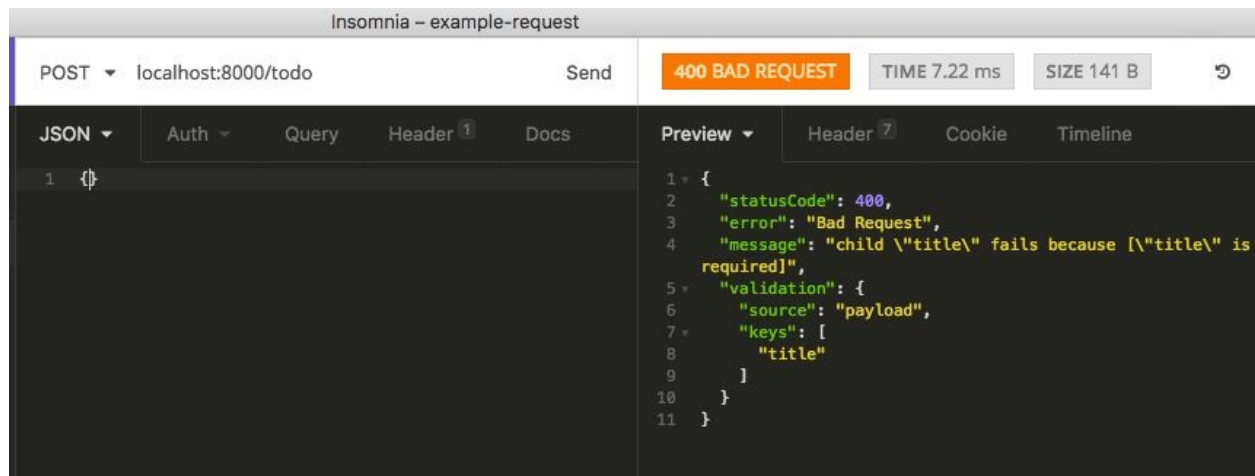
3. In the `routes/todo.js` file, we need to require Joi and then modify our post route by adding a `config.validate` key to the route object:

```
{
  method: 'POST',
  path: '/todo',
  handler: (request, reply) =>
  {
    const todo = request.payload;
    todoList.push(todo);
    return reply({ message: 'created' });
  },
  ...
},
```

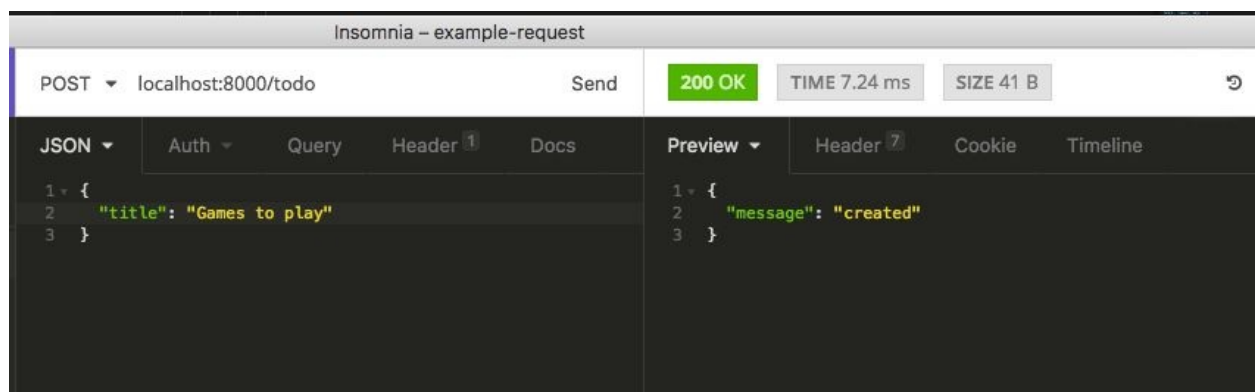


Use the `todo.js` file for your reference at `Code/Lesson-2/exercise-c1/routes`.

4. When we try to submit an empty payload, we now get error `400`:



5. That is, until we provide a title for the todo, since a title is required:



Joi is a full-fledged validation library with many options for how to use it. In this exercise, we just touched on a basic example.

You validate any part of the request by coming up with the respective key/value pair within the validate key and its respective type:

payload (for request payloads, as in the preceding exercise), params (for request params), and query (for query params).



For example, for the request, GET: /todo/:id, if we want to validate that the ID is an integer, we will add this config object:

```

config: {
  validate: {
    {
      params: {
        {
          id: Joi.number()
        }
      }
    }
  }
}

```

More details on Joi can be found here: <https://github.com/hapijs/joi>.

Summary

This chapter has covered initial part of building our API with Node.js. We started by looking at a basic HTTP server built with only the built-in HTTP module, for us to appreciate the basic building blocks of a Node.js web application. We then introduced doing the same thing with the Hapi.js framework.

We then went through various HTTP verbs (request methods) by example as we built our basic API with Hapi.js. Those were `GET`, `POST`, `PUT`, `PATCH`, and `DELETE`.

We also covered some fundamental concepts of web applications, such as logging, using good and request validation, and using Joi.

Building the API - Part 2

This chapter is intended to revisit the previous implementation, this time saving our data in a persistent storage (database). It will also cover authentication, and unit testing and hosting as additional good-to-know concepts (but not essential). It is therefore prudent to put more emphasis on working with the DB using knex.js and authenticating your API with JWT.

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

- Implement database connection with Knex.js
- Describe commonly used Knex.js methods
- Rewrite our previous implementation of the todo routes with Knex.js
- Implement API authentication with JWT
- Describe the importance of having unit tests for your API
- Implement basic testing of the API with Lab

Working with the DB Using Knex.js

In this section, we're going to go through the fundamental concepts of working with the database. We will continue with the step-by-step build-up from our previous todo project. You will have noticed that our last project, we were storing our information in computer memory, and that it disappears immediately once our server returns. In real-life, you will want to store this data persistently for later access.

So, what is Knex.js? It is a SQL query-builder for relational databases like PostgreSQL, Microsoft SQL Server, MySQL, MariaDB, SQLite3, and Oracle. Basically, with something like Knex, you can write one code that will easily work with any of the mentioned databases, with no extra effort, just switching the configurations.

Let's walk through the exercise as we explain the concepts.

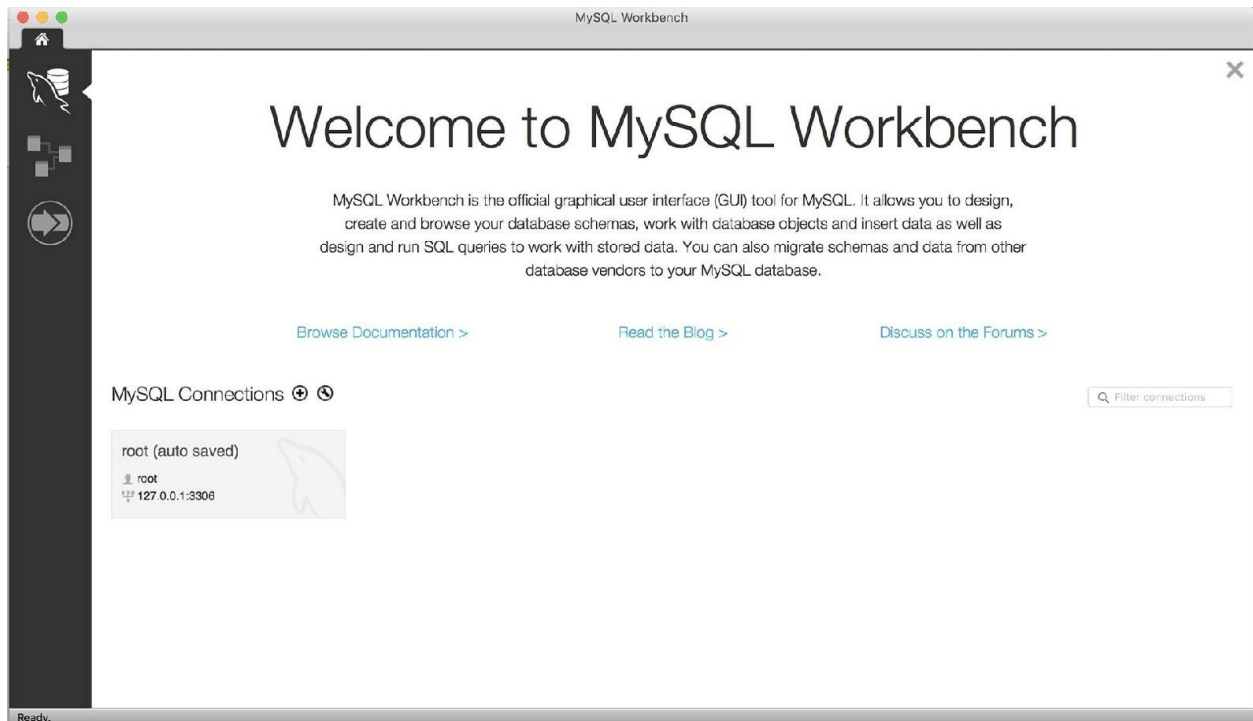
Exercise 12: Setting up the Database

Let's go back to where we left off in the *Exercise 11: Validating a Request* of [Chapter 2, Building the API – Part 1](#). In this example, we will be using MySQL as our database of choice. Make sure your machine is set up with MySQL and MySQL Workbench:

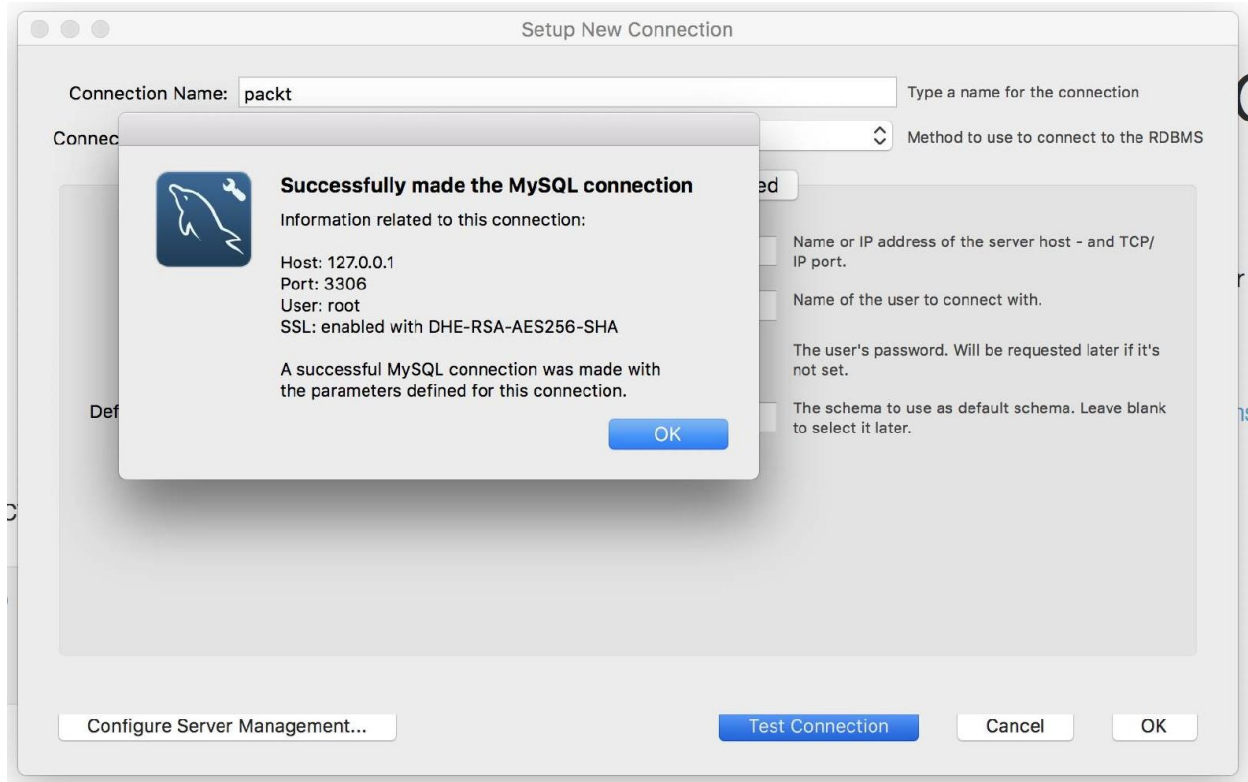


Use the `Code/Lesson-3/exercise-a` folder for your reference.

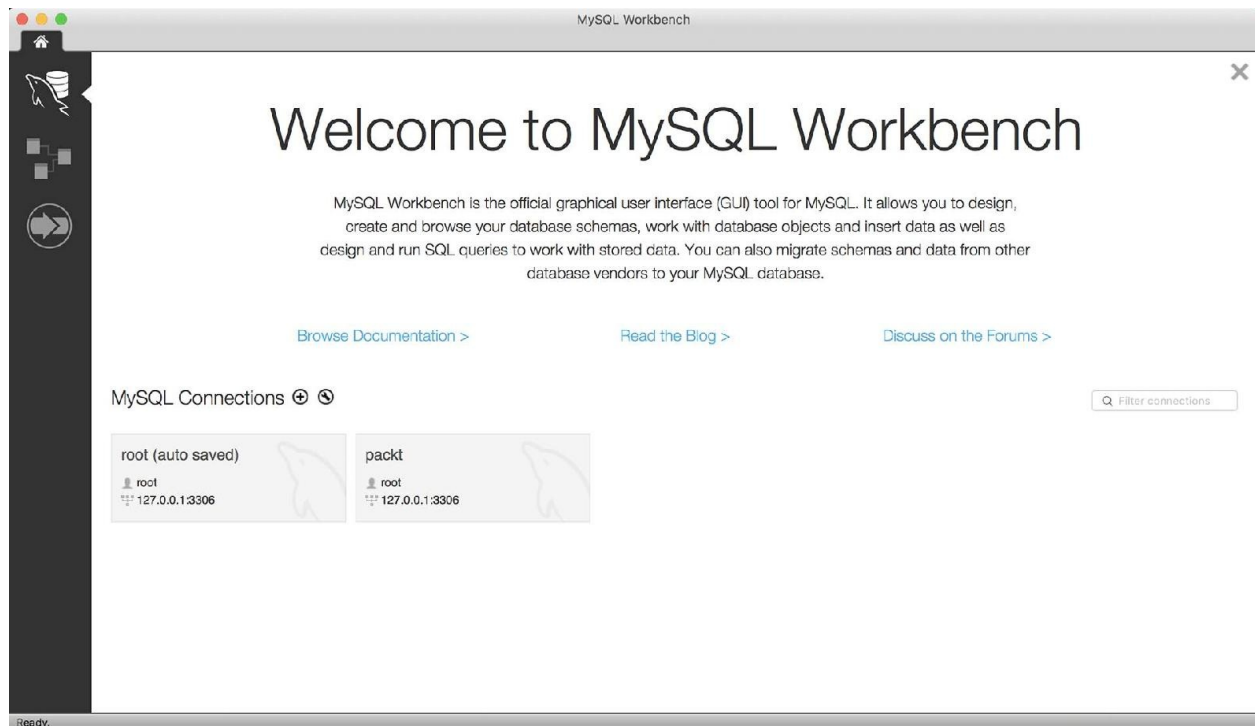
1. Open MySQL Workbench. Click on the + button to create a connection:



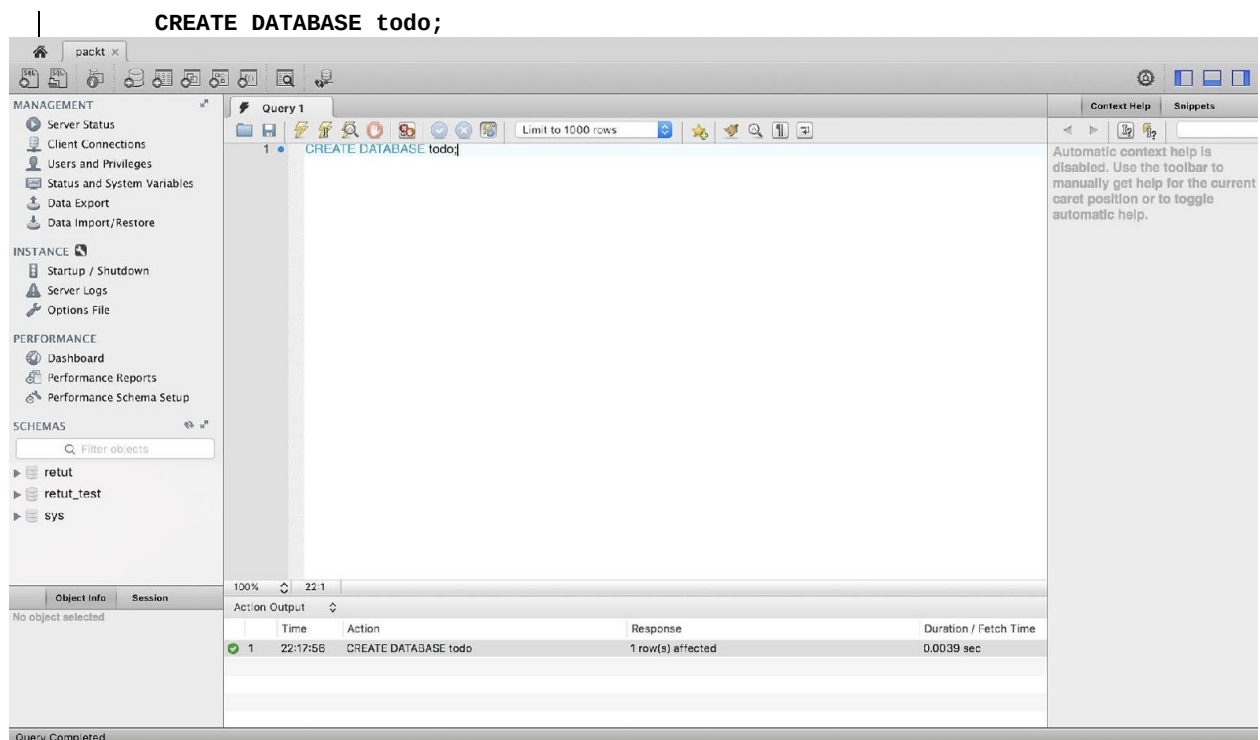
2. Add the connection name as `packt`, username as `root`, and the password (if any). Click on Test Connection to see if the connection is correct, then click on OK:



3. Click on OK to create the connection.
4. Now, click on the connection, packt:



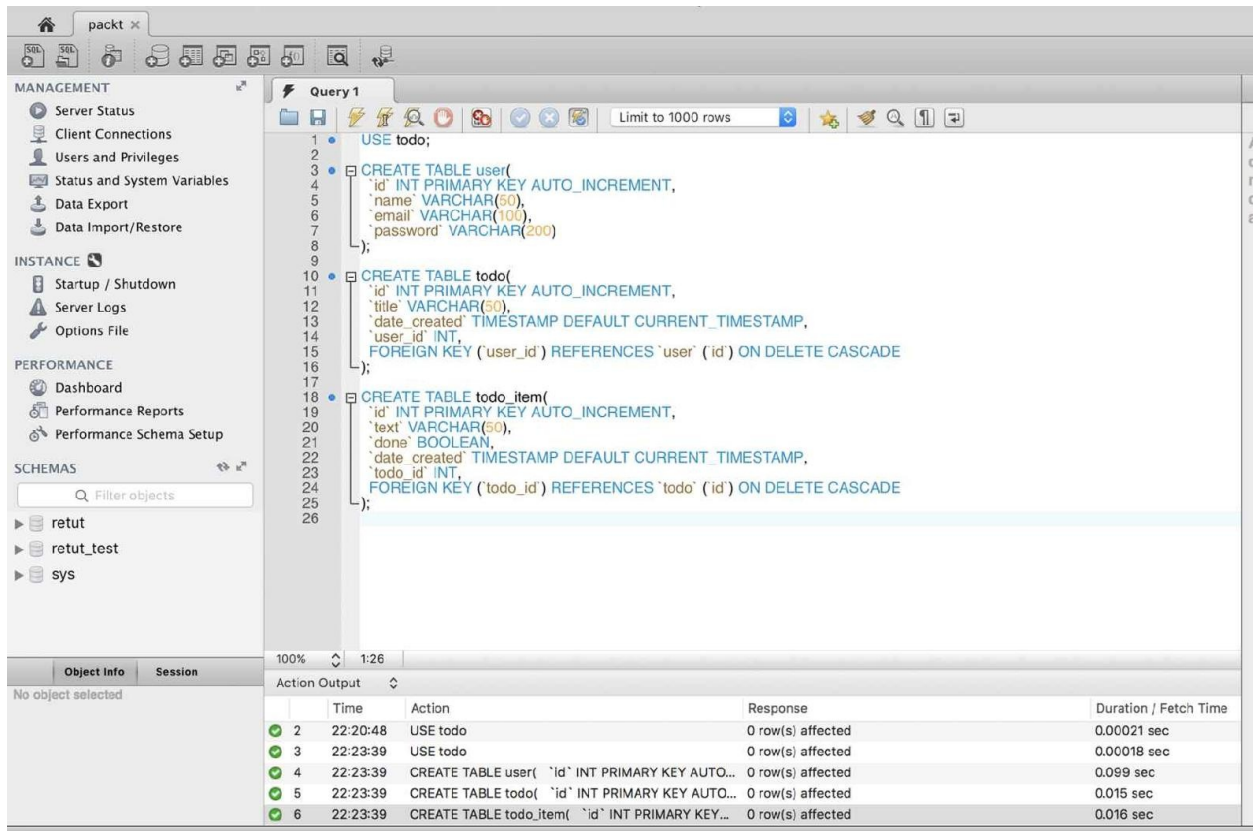
5. Create the todo database by running the following query, and click on the Execute icon:



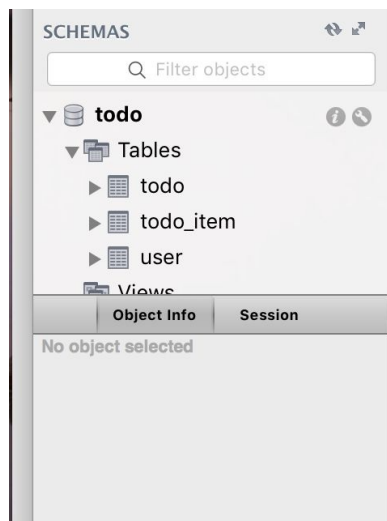
6. The chapter files come with a basic SQL schema for our todo example

project, almost similar to what we were using with the basic JavaScript array in the previous exercises:

1. In the `code/Lesson-3` folder, there is a file called `raw-sql.sql`. Open the file with your code editor and copy the contents of the file.
2. Then, go back to the MySQL Workbench.
3. Paste what you copied from the file in the textbox and click on the Execute icon:



- You should see the list of the created tables (todo, todo_item, user) as below, when you click on the Refresh icon to the right of the SCHEMAS label, and click on Tables:



Exercise 13: Connecting to the Database

Now that we have created our database, in this exercise we are going to connect our application to our database using the necessary npm packages, that is, `knex` and `mysql`:

1. On the Terminal, change directory to the root of our project, and run the following command:

```
| npm install mysql knex --save
```

2. Let's create a file `db.js` and add the following code to it, replacing the user and password appropriately if need be:

```
| const env = process.env.NODE_ENV || 'development';
| const configs =
| {
|   development:
|     {
|       client: 'mysql',
|       ...
|       const Knex = require('knex')(configs[env]);
|       module.exports = Knex;
```



You can find the complete code from the `db.js` file at `Code/Lesson-3/exercise-a`.

3. Let's test that we have our configurations right. We will create a `test-db.js` file:

```
| const Knex = require('./db');
| Knex.raw('select 1+1 as sum')
|   .catch((err) => console.log(err.message))
|   .then((res) => console.log('connected: ', res[0].sum));
```

4. Now, let's go to the Terminal and run the test file:

```
| node test-db.js
```

You should get the following printed:

| connected: 2

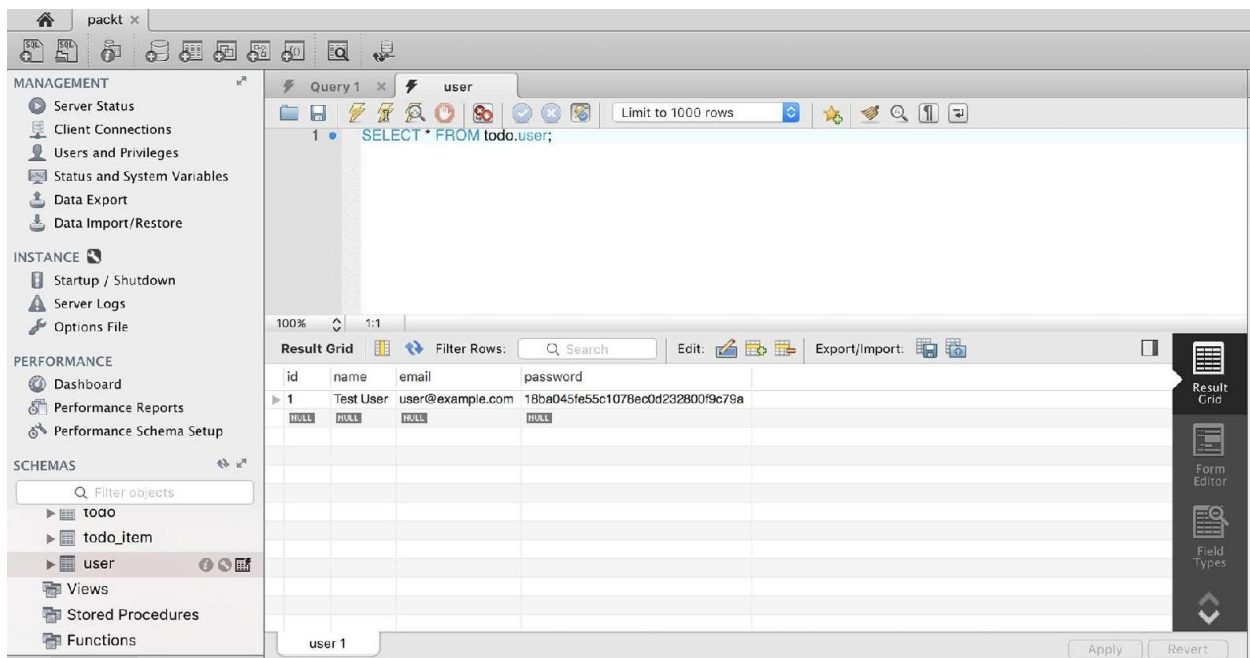
Exercise 14: Creating a Record

In this exercise, we're going to write code for saving a todo and its *items*. To start off, let's create a dummy user since we will hardcode the user ID for our code. Later, in *Exercise 19: Securing All the Routes*, we will have the ID picked from the authentication details:

1. Go back to MySQL Workbench.
2. Clear the previous query and paste the following query, and click on the Execute icon:

```
USE todo;  
INSERT INTO 'user' ('id', 'name', 'email', 'password')  
VALUES (NULL, 'Test User', 'user@example.com',  
MD5('u53rtest'));
```

3. When you click on the user table, you should see the following; our newly created user has an ID of 1:



4. Now, let's go to our routes file, `/routes/todo.js` and modify the code, for the POST: `/todo` route; change the code to be as follows (it's only the `handler` that is

changing, notice the change to `async` function):

1. Let's start by requiring our Knex instance that is in `../db.js`. Just after the line requiring Joi, add this:

```
const Knex = require('../db');
```



Notice the two dots, `../db.js`, since `db.js` is found in the parent folder. Recall our topic on requiring local modules in [Chapter 1, Introduction to Node.js](#).

2. Now, let's modify our handler for the `POST: /todo` route. Here, we are using the `Knex.insert` method, and adding an optional `.returning` method so that we get back the ID of `todo` we have added:

```
{
  method: 'POST',
  path: '/todo',
  handler: async (request, reply) => {
    const todo = request.payload;
    todo.user_id = 1; // hard-coded for now
    // using array-destructuring here since the
    // returned result is an array with 1 element
    const [ todoId ] = await Knex('todo')
      .returning('id')
      .insert(todo);
    ...
  },
},
```

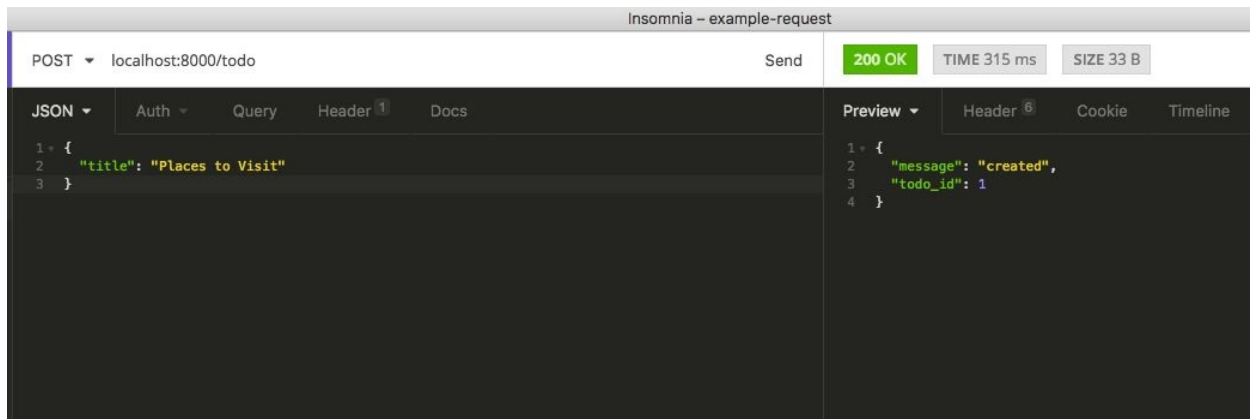


You can find the complete code from the `todo.js` file at [Code/Lesson-3/exercise-a/routes](#). Unlike our previous exercises in [Chapter 2, Building the API – Part 1](#), we will split our `POST: /todo` route into two, `POST: /todo`, for adding a todo list, and `POST: /todo/<id>/item` for adding items to the list.

5. Now, let's test our newly created endpoint. If you had stopped your server, go back to the Terminal and start it again, with `nodemon`:

```
nodemon server.js
```

6. Go to Insomnia and make the post request; you should get something like this (notice the `todo_id` returned, since we will use it in our next example):



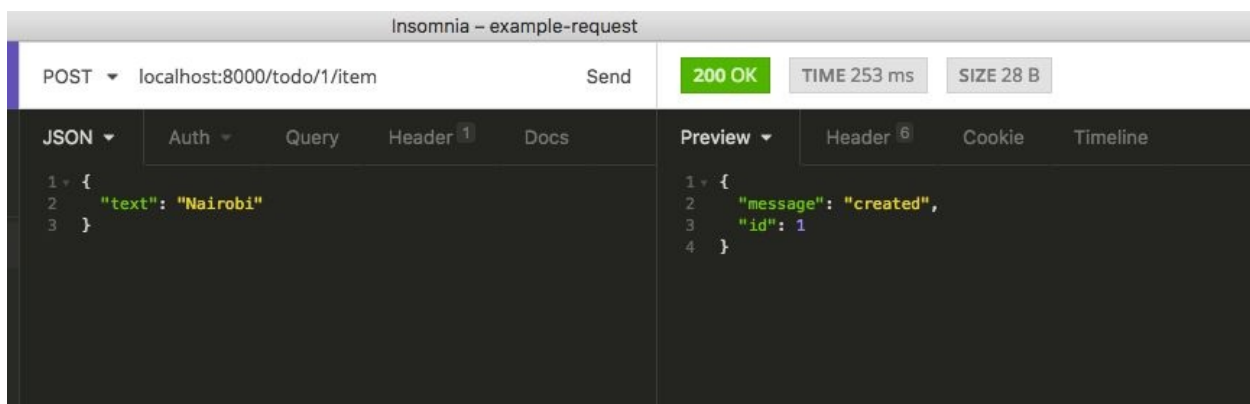
7. Now, let's add a route for adding todo items, POST: `/todo/<id>/item`; therefore, next to the previous route object, add this route object:

```
{
  method: 'POST',
  path: '/todo/{id}/item',
  handler: async (request, reply) => {
    const todoItem = request.payload;
    todoItem.todo_id = request.params.id;
    const [ id ] = await Knex('todo_item')
      .insert(todoItem);
    return reply({ message: 'created', id: id });
  },
  ...
},
```



You can find the complete code from the `todo.js` file at `Code/Lesson-3/exercise-a/routes`.

8. Now, let's test the route, `/todo/1/item`, 1 being the ID of `todo` we created in step 6:



Exercise 15: Reading from the Database

In this exercise, we're going to write the routes for:

- Listing all `todo` for a particular user
- Getting details for a single `todo` item
- Listing items for a particular `todo`

We will use a number `Knex` methods:

- `Knex('<table_name>')`, this is equivalent to `'SELECT * FROM <table_name>'`
- `.where()`, used for adding the where clause to the query

1. To get a list of all `todo`, we will modify our previous `GET: /todo` route. Here, you only want to list `todo` items for a particular authenticated user. For now, we will be using our hardcoded test user:

```
{
  method: 'GET',
  path: '/todo',
  handler: async (request, reply) =>
  {
    const userId = 1; // hard-coded
    const todos = await Knex('todo')
      .where('user_id', userId);
    return reply(todos);
  },
},
```

2. Let's modify the route for getting a single `todo` item, `GET: /todo/<id>`:

```
{
  method: 'GET',
  path: '/todo/{id}',
  ...
  .where({
    id: id,
    user_id: userId
  });
  if (todo) return reply(todo);
  return reply({ message: 'Not found' }).code(404);
},
```

| },



You can find the complete code from the `todo.js` file at `Code/Lesson-3/exercise-a/routes`.



We are using array destructuring here too, since the result, if any, will be an array of length 1, so we're getting the first and only element from the array with: `const [todo] = ...`

3. Now, let's add the route object for getting a list of items for a particular `todo`, preferably just after the route for adding a `todo` item that we did in *Exercise 14: Creating a Record*:

```
{
  method: 'GET',
  path: '/todo/{id}/item',
  handler: async (request, reply) => {
    const todoId = request.params.id;
    const items = await Knex('todo_item')
      .where('todo_id', todoId);
    return reply(items);
  },
},
```

4. Now, let's test the route:

The screenshot shows the Insomnia application interface. At the top, the title bar reads "Insomnia - example-request". Below it, the request details are shown: "GET" method, "localhost:8000/todo/1/item" path, and a "Send" button. To the right of the "Send" button, the response status is "200 OK" in a green box, with "TIME 248 ms" and "SIZE 93 B" in grey boxes. Below the request details, there are tabs for "JSON", "Auth", "Query", "Header", and "Docs". The "JSON" tab is selected, showing a truncated response "1 ...". To the right of the tabs, there are tabs for "Preview", "Header", "Cookie", and "Timeline". The "Preview" tab is selected, showing a JSON array of one object:

```
[
  {
    "id": 1,
    "text": "Nairobi",
    "done": null,
    "date_created": "2017-11-27T03:50:57.000Z",
    "todo_id": 1
  }
]
```

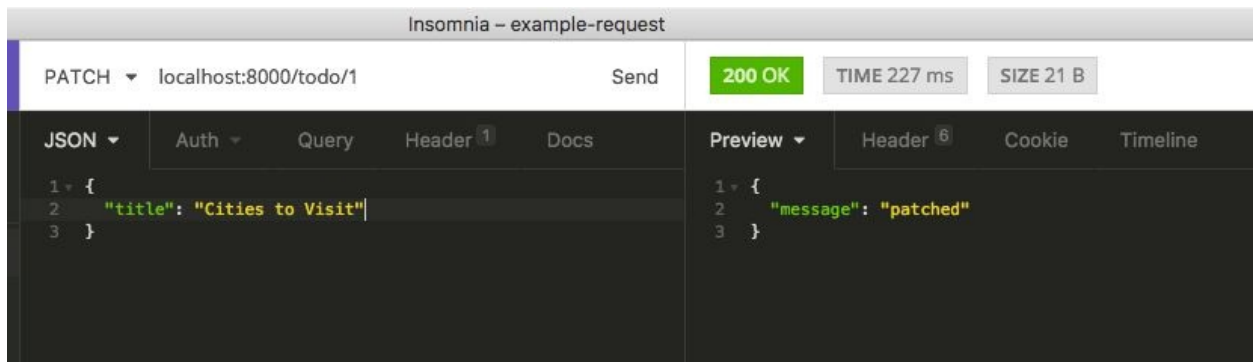
Exercise 16: Updating a Record

In this exercise, we're going to write routes for updating a todo title or a todo item, and here we will introduce a new Knex method, `.update()`:

1. Let's start by modifying our previous `PATCH: /todo/<id>` route. We have also added an extra validation to make sure that `title` is supplied as payload:

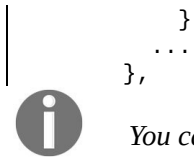
```
{
  method: 'PATCH',
  path: '/todo/{id}',
  ...
  title: Joi.string().required(),
}
},
```

2. Let's test the route:



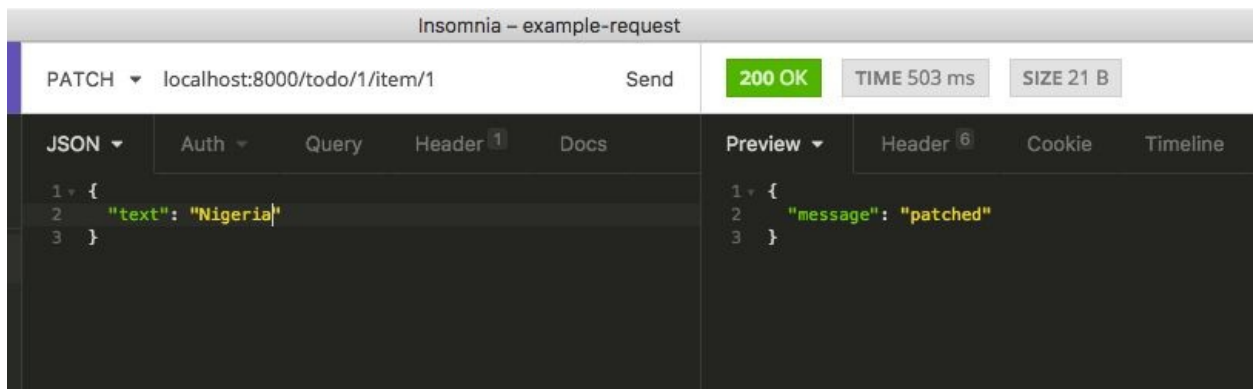
3. Now, let's add another `PATCH` route for `/todo/<id>/item`, this will help in editing a todo item's text and also marking a todo item as done or not done:

```
{
  method: 'PATCH',
  path: '/todo/{todo_id}/item/{id}',
  handler: async (request, reply) => {
    const itemId = request.params.id;
    ...
    payload: {
      text: Joi.string(),
      done: Joi.boolean(),
    }
  }
}
```

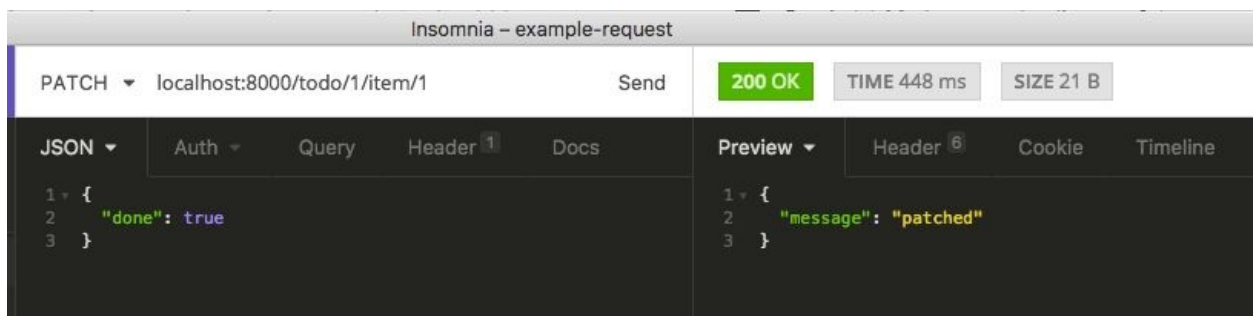



You can find the complete code from the `todo.js` file at [Code/Lesson-3/exercise-a/routes](#).

4. This route can take each of the payload items one at a time (which will be the most practical case, when using, for example, a web or mobile UI), or all at once:
 1. For instance, changing the item from Nairobi to Nigeria, or:



2. Marking the item as done:



5. When we list the items again through the `GET: /todo/<id>/item` route, you will see the updated item:

Insomnia – example-request

GET localhost:8000/todo/1/item

Send

200 OK

TIME 201 ms

SIZE 90 B

JSON

Auth

Query

Header 1

Docs

1 | ...

Preview

Header 2

Cookie

Timeline

```
1 [
2   {
3     "id": 1,
4     "text": "Nigeria",
5     "done": 1,
6     "date_created": "2017-11-27T03:50:57.000Z",
7     "todo_id": 1
8   }
9 ]
```

Exercise 17: Deleting a Record

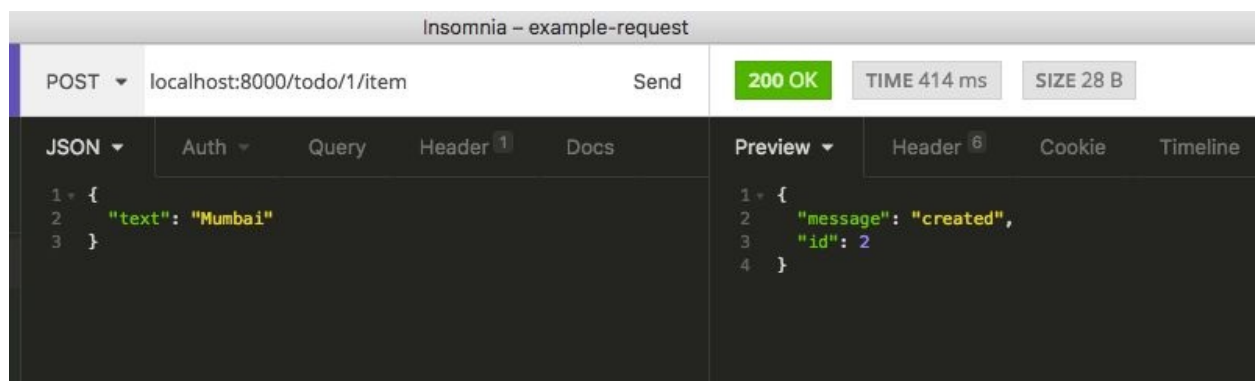
In this exercise, we will be introducing the last vital Knex method to complete our **Create, Read, Update, Delete (CRUD)** journey, `.delete()`:

1. Let's add a route for deleting a `todo` item:

```
{
  method: 'DELETE',
  path: '/todo/{todoId}/item/{id}',
  handler: async (request, reply) =>
  {
    const id = request.params.id;
    const deleted = await Knex('todo_item')
      .where('id', id)
      .delete();
    return reply({ message: 'deleted' });
  },
},
```

2. Now, let's add one more item on our previous `todo` (of ID `1`), then delete it:

1. Add item:



2. Now that we have its ID (`2`, in this case), delete it:

Insomnia – example-request

DELETE ▾

localhost:8000/todo/1/item/2

Send

200 OK

TIME 216 ms

SIZE 21 B

JSON ▾

Auth ▾

Query

Header 1

Docs

1 ...

Preview ▾

Header 6

Cookie

Timeline

1 {
2 "message": "deleted"
3 }

Exercise 18: Cleaning up the Code

Now that we have almost updated all our routes that we had from [Chapter 2, Building the API – Part 1](#), let's now remove all the code that is no longer needed:

1. Remove the previously hardcoded list of todos:

```
const todoList = [
  ...
];
```

2. Remove the PUT: /todo/<id> route object:

```
{
  method: 'PUT',
  path: '/todo/{id}',
  handler: (request, reply) =>
  {
    const index = request.params.id - 1;
    // replace the whole resource with the new one
    todoList[index] = request.payload;
    return reply({ message: 'updated' });
  },
},
```

3. Reimplement the DELETE: /todo/<id> route object, very similar to *Exercise 17: Deleting a Record*; the difference is just the route:

```
{
  method: 'DELETE',
  path: '/todo/{id}',
  handler: async (request, reply) =>
  {
    const id = request.params.id;
    const deleted = await Knex('todo')
      .where('id', id)
      .delete();
    return reply({ message: 'deleted' });
  },
},
```

Since our SQL query had this line that adds a constraint which is possible when a *todo* is deleted, all the items for that *todo* are also deleted:

```
CREATE TABLE todo_item(
  'id' INT PRIMARY KEY AUTO_INCREMENT,
  'text' VARCHAR(50),
  'done' BOOLEAN,
  'date_created' TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  'todo_id' INT,
  FOREIGN KEY (`todo_id`) REFERENCES `todo` (`id`) ON DELETE CASCADE
);
```



Authenticating Your API with JWT

So far, we have been using our API without any authentication. This means that if this API is hosted at a public place, anyone can access any of the routes, including deleting all our records! Any proper API needs authentication (and authorization). Basically, we need to know who is doing what, and if they are authorized (allowed) to do that.

JSON Web Tokens (JWT) is an open, industry standard method for representing claims securely between two parties. Claims are any bits of data that you want someone else to be able to read and/or verify but not alter.

To identify/authenticate users for our API, the user puts a standard-based token in the header (with the Authorization key) of the request (prefixing it with the word *Bearer*). We will see this practically in a short while.

Exercise 19: Securing All the Routes

In this exercise, we're going to secure all the `/todo/*` routes that we created so that no unauthenticated user can access them. In the *Exercise 21: Implementing Authorization*, we will differentiate between an *unauthenticated* and an *unauthorized* user:

1. We will first start by installing a Hapi.js plugin for JWT, `hapi-auth-jwt`. Go to the Terminal and run:

```
| npm install hapi-auth-jwt --save
```



Use the `Code/Lesson-3/exercise-b` for your reference.

2. We will modify the routes array that we get from `./routes/todo.js` in the `server.js` file:

1. First, begin by requiring the installed `hapi-auth-jwt` at the top of the file:

```
| const hapiAuthJwt = require('hapi-auth-jwt');
```

2. Then, replace the old line, `server.route(routes.todo)`, with this:

```
| server.register(hapiAuthJwt, (err) =>
| {
|   server.auth.strategy('token', 'jwt',
|   {
|     key: 'secretkey-hash',
|     verifyOptions:
|     {
|       algorithms: [ 'HS256' ],
|     ...
|     // add auth config on all routes
|     ...
|   });
```



You can find the complete code from the `server.js` file at `Code/Lesson-3/exercise-b`.

3. Now, try accessing any of the routes, for example, `GET: /todo`; you should get this:

GET ▼ localhost:8000/todo

Send

401 UNAUTHORIZED

TIME 4.46 ms

SIZE 76 B

JSON ▼

Auth ▼

Query

Header 1

Docs

1 | ...

Preview ▼

Header 7

Cookie

Timeline

1 {

2 "statusCode": 401,

3 "error": "Unauthorized",

4 "message": "Missing authentication"

5 }

Exercise 20: Adding User Authentication

Now that we have secured all our todo routes, we need a way to issue tokens to valid users to access the API. We will have the users send their email and password to a route (`/auth`), and our API will issue back an authentication token which will be used for each request:

1. In the `/routes` folder, create a file `auth.js`.
2. We will now need two more packages for this, `jsonwebtoken` for signing the authentication token, and `md5` for comparing the password since if you recall, we were using MySQL's `md5` function to store the user's password:

```
| npm install jsonwebtoken md5 --save
```

3. In the `auth.js` file, have the following code:

```
| const jwt = require('jsonwebtoken');
| const Joi = require('joi');
| const md5 = require('md5');
| const Knex = require('../db');
| module.exports =
|   {
|     method: 'POST',
|     path: '/auth',
|     ...
|   };
|
```



You can find the complete code from the `auth.js` file at `Code/Lesson-3/exercise-b/routes`.

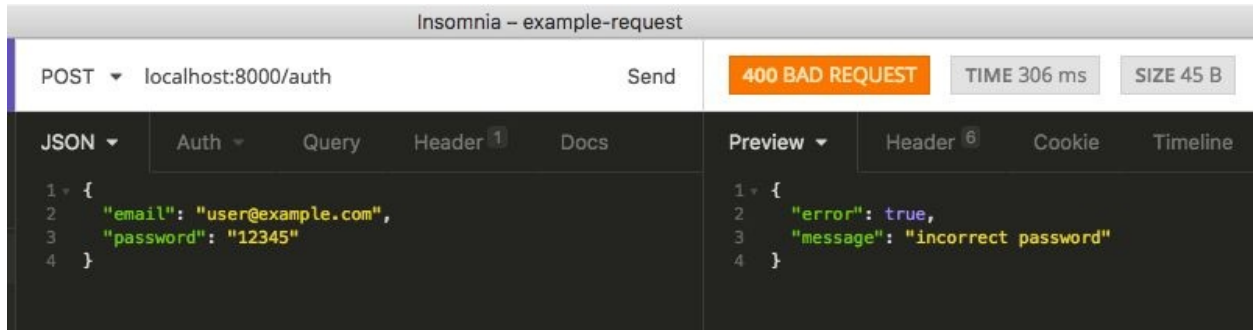
4. Now, let's register our `auth.js` route with the server. In `server.js`, after `routes.todo = ...`, add the following code:

```
| routes.auth = require('./routes/auth');
```

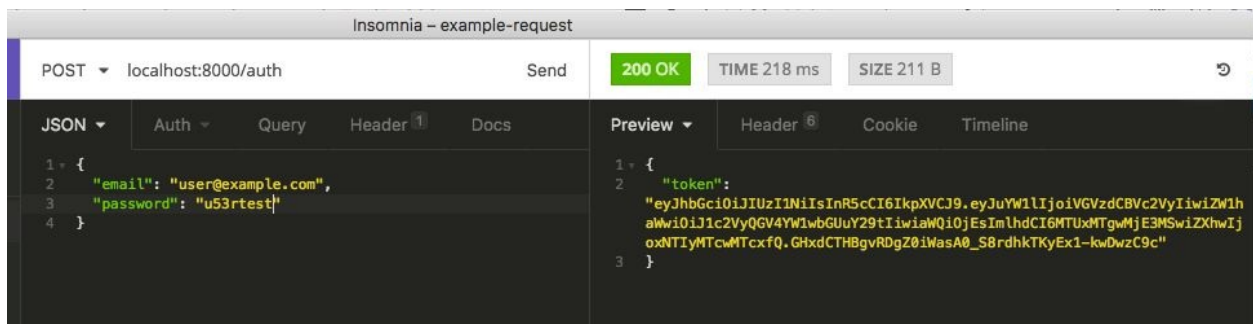
5. After the line initializing the server, we can add the route registration:

```
| server.route(routes.auth);
```

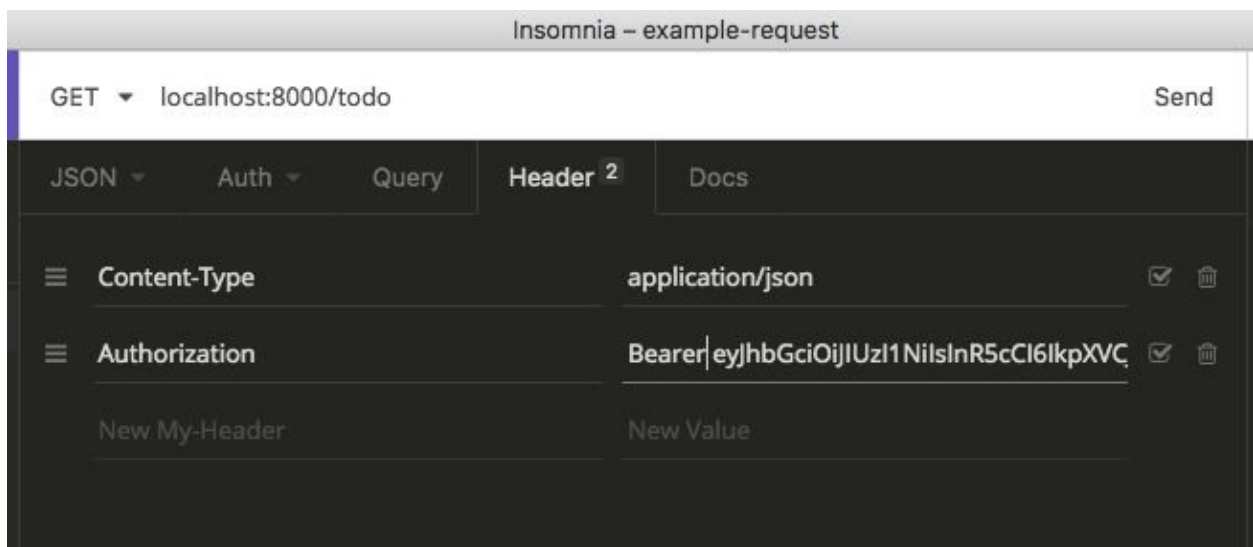
6. Now, let's try out our route, `POST: /auth`:
 1. First, with the incorrect email/password combination:



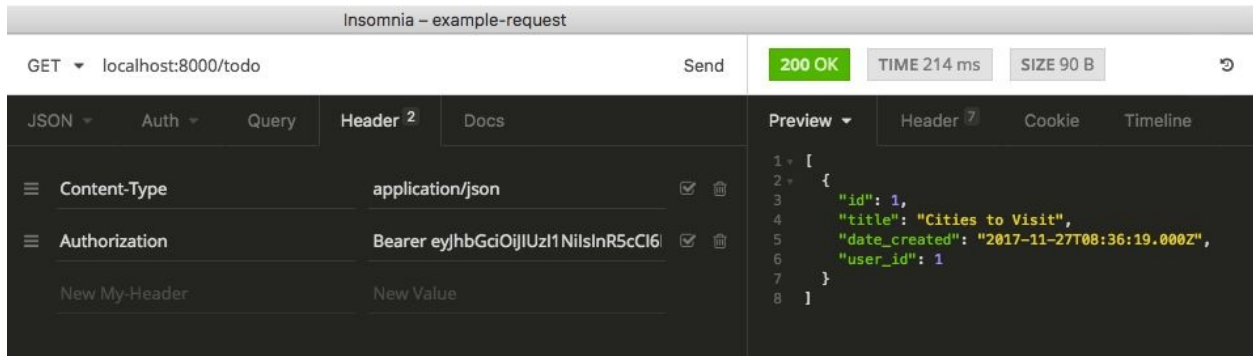
2. Then, with the correct password, remember *Exercise 14: Creating a Record, step 2* where we created the test user with the password:



7. Now, we can copy the generated token and use it for our subsequent requests, for example, GET: /todo, by adding an Authorization header. Thus remember, we start with the word `Bearer`, then space, then paste the token; that's the JWT convention:



8. And we can now access the route without getting the unauthorized responses, like in *step 6* of 20th exercise:



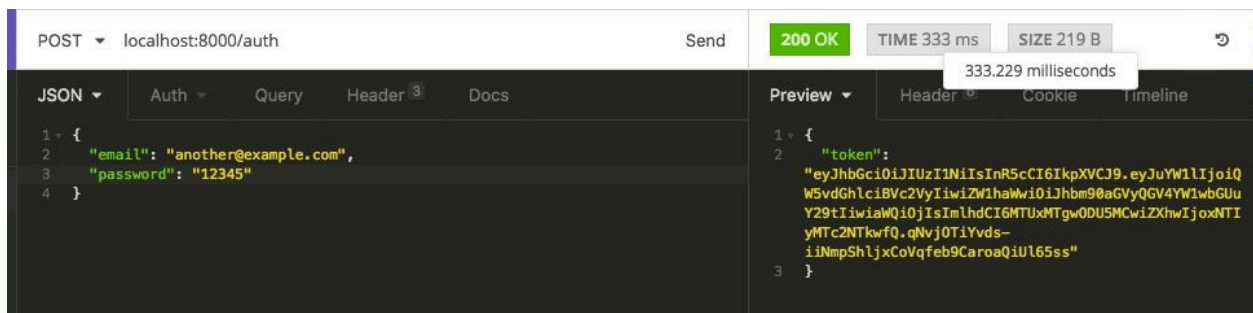
9. Now, let's go back to the places in our `./routes/todo.js` file where we were hardcoding the users, and now get them from the authentication object, that is:

```
const userId = request.auth.credentials.id;  
Recall in the preceding step 3, when we were signing our token, we provided the user details, that is, name, email, and id. This is where we get the .id in request.auth.credentials.id:  
jwt.sign(  
  {  
    name: user.name,  
    email: user.email,  
    id: user.id,  
    ...  
  },  
  ...  
);
```

10. Now, let's go back to our phpMyAdmin web interface and create another user, just like we did in *Exercise 14: Creating a Record, step 2*, and paste the following SQL in the SQL text area:

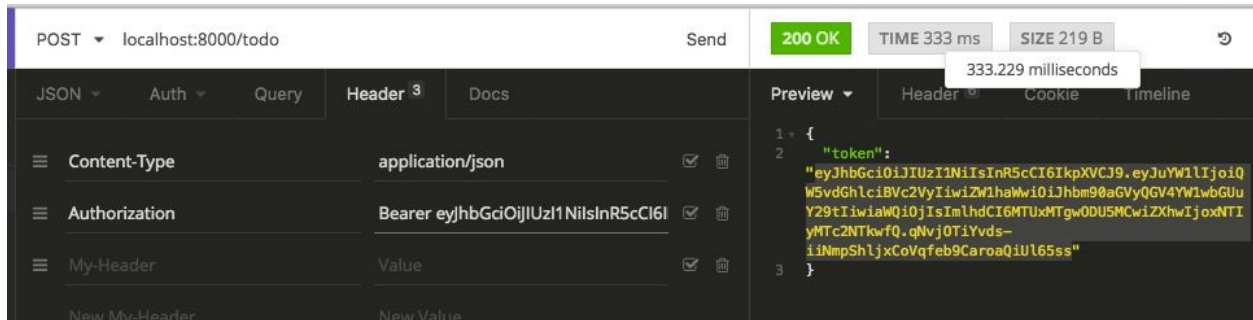
```
INSERT INTO 'user' ('id', 'name', 'email', 'password')  
VALUES (NULL, 'Another User', 'another@example.com',  
MD5('12345'));
```

11. Now, let's go and do another `POST: /auth` request with the new user and obtain the token:

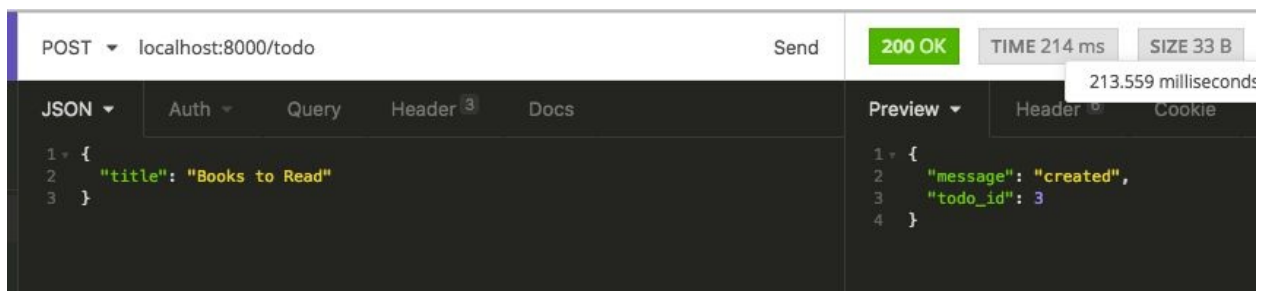


12. Let's use this new token to create another todo list by doing a POST: /todo request:

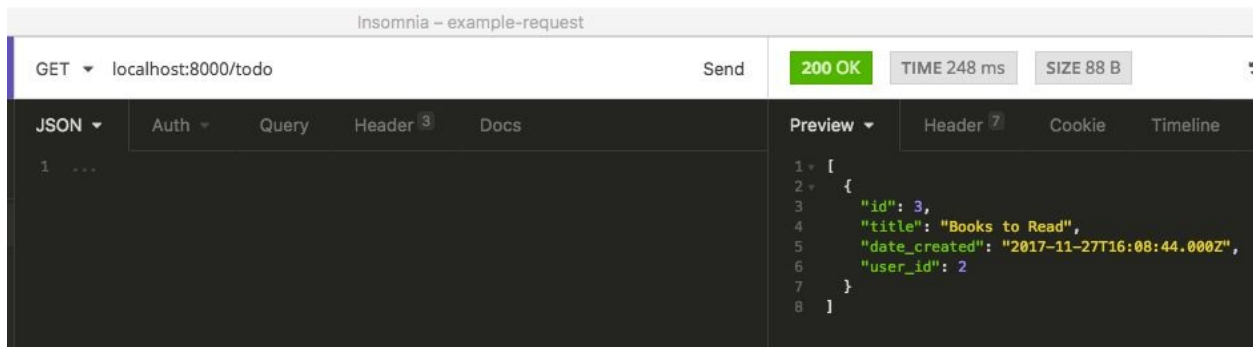
1. On Insomnia, go to the Header section, delete the previous Authorization header and replace it with the new one:



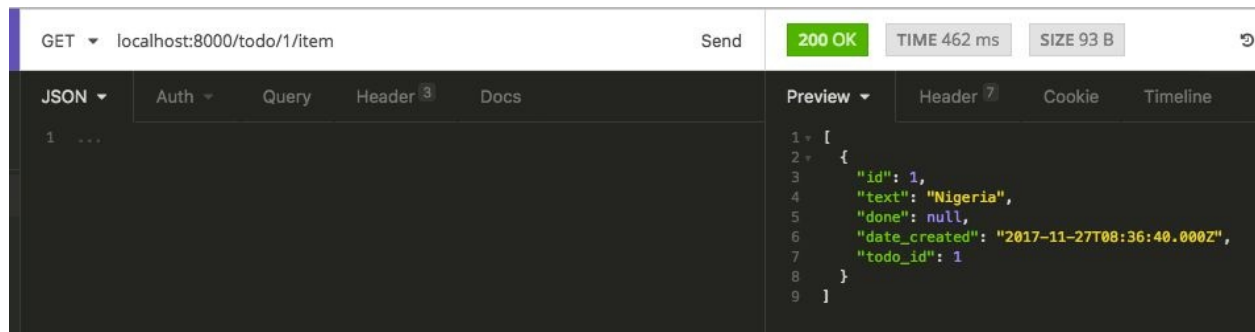
2. Now, let's make our request:



3. Let's see the new list of todos, by doing GET: /todo:



4. As you can see, the newly created user can only see what they have created. We have done a good job so far, as far as authorization is concerned. However, let's try and check the items for todo ID 1, which belonged to the first user:



Oops! We can see someone else's todo list items; this is a security flaw. This leads us to the final part of this topic, **authorization**.

Authentication versus Authorization

Through authentication, we get to know who is accessing our API; through authorization, we get to tell who can access what, within our API.

Exercise 21: Implementing Authorization

In this exercise, we are going to refine our API to make sure that users are only authorized to access their todos and todo items:

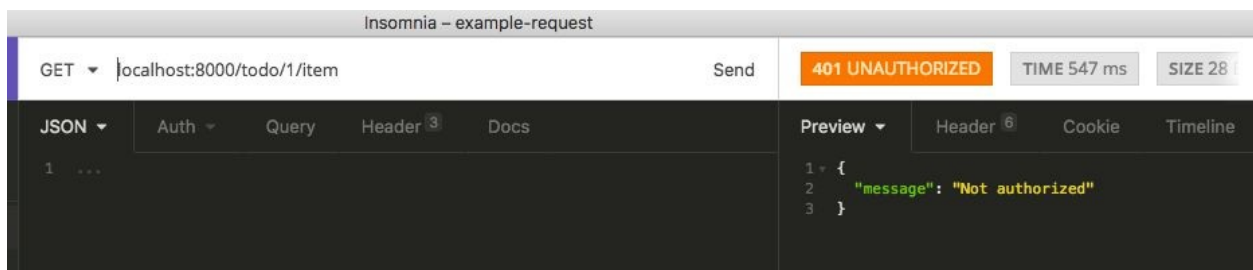
1. Let's first fix the flaw that we came across in *Exercise 20: Adding User Authentication, step 12*. So, we will modify the `GET: /todo/<id> item` route object in `/routes/todo.js`, by first checking if the todo belongs to the user before they can access its items:

```
{
  method: 'GET',
  path: '/todo/{id}/item',
  handler: async (request, reply) => {
    const todoId = request.params.id;
    ...
    return reply(items);
  },
},
```



You can find the complete code from the `todo.js` file at `Code/Lesson-3/exercise-b/routes`.

2. Now, when we go back to access `GET: /todo/1/item`, we get the right error message:



3. You can add extra authorization logic for the following routes:

- `POST: /todo/<id>/item`, to make sure that a user cannot add items to a todo that does not belong to them.

- PATCH: /todo/<id>, that a user cannot patch a todo that does not belong to them.
- PATCH: /todo/<todoId>/item/<id>, that a user cannot patch a todo item that does not belong to them.
- DELETE: /todo/<id>, that a user cannot delete a todo that does not belong to them.
- DELETE: /todo/<todoId>/item/<id>, that a user cannot patch a todo item that does not belong to them.

Cross-Origin Resource Sharing (CORS) is a mechanism that uses additional HTTP headers to let a user agent (browser) gain permission to access selected resources from a server on a different origin (domain) than the site currently in use. For instance, when you are hosting a web application frontend on another domain, because of browser restriction, you will not be able to access the API. We therefore need to explicitly state that our API will allow cross-origin requests. We will modify the `server.js` file, at the place we were initializing the server connection, to enable CORS:

```
server.connection({
  {
    host: 'localhost',
    port: process.argv[2] || 8000,
    routes:
      {
        cors: true,
      }
  }
});
```

Testing Your API with Lab

In this section, we will have a brief look at writing unit tests for Hapi.js APIs. Testing is a huge topic that perhaps requires a whole course on its own, but in this section, we will be introducing the essential parts to get you up and running.

Let's first underscore the importance of writing unit tests for your API:

- **Maintainability:** This is what I consider as the most important value of adding tests to your software. When you have tests, you can be confident to come back months later and, modify your code without worrying whether you broke anything by your update.
- **Requirement specifications:** Tests make sure that your code meets the requirements. For our example, we started off by implementing the requirements since we wanted to pass across some basic concepts. But in practice, it is way better to start off with the tests before implementing your routes.
- **Automated testing:** You realized that in our previous examples, we kept checking our API client (Insomnia) to see if our API was working correctly; this can be a little cumbersome. With tests, you don't have to worry about this once you have written correct tests.

Hapi.js conventionally uses **Lab** (<https://github.com/hapijs/lab>) as its testing framework. We're going to write a few tests for our API in the next exercise.

Exercise 22: Writing Basic Tests with Lab

In this exercise, we will introduce the concept of writing unit tests for the Hapi.js web API, mainly using the third-party `lab` module and the built-in `assert` module. Ideally, we should have a separate database for our tests, but for the sake of simplicity here, we will share our development database for tests too:

1. Let's first start by installing the necessary packages. Notice that we are using `--save-dev` since tests are not needed for production, therefore, they are *development dependencies*:

```
| npm install lab --save-dev
```



Use the `Code/Lesson-3/exercise-c` for your reference.

2. Create a `test` folder at the root of the project—that is where we will have our tests. Since our API is a simple one, we will only have one file with all our tests.
3. In `test`, create a file `test-todo.js`.
4. As a set up, `test/test-todo.js` requires the necessary modules that we need for our test:

```
| const assert = require('assert');  
  // lab set-up  
  const Lab = require('lab');  
  const lab = exports.lab = Lab.script();  
  // get our server(API)  
  const server = require('../server');
```

In the first line, we are requiring `assert`, which is an inbuilt module if you recall from [Chapter 1, Introduction to Node.js](#). Alternatively, you can use any other assertion libraries such as `chai` (<https://github.com/chaijs/chai>), `should.js` (<https://github.com/tj/should.js>), and others.



Lab test files must require the `lab` module and export a test script, as seen on line 4 prior. We will be getting the rest of the items from `lab` in the following line; we're going to see them in action shortly.

5. Since we are requiring the `server` in line 6 of our `test-todo.js` file, we need to go back to our `server.js` file and export the `server` object, on the last line:

```
module.exports = server;
```

6. For the DB configuration, let's modify our `db.js` file to include configurations for the test environment, pointing to the development configurations. Add this line right after the `configs` definition:

```
configs.test = configs.development;
```

7. Let's modify the server connection setup code so that the port for our testing server is set from the environment variables when running the tests. This allows us to have the test server run on a different port, while our development server is running:

```
server.connection({
  host: 'localhost',
  port: process.env.PORT || 8000,
  routes: {
    cors: true,
  }
});
```

8. There are a number of methods that we will use from the `lab` module; we will need to use object destructuring to get them. Add the following line in our `test-todo.js` file:

```
const
{
  experiment,
  test,
  before,
} = lab;
```

9. Let's start by writing a simple test that makes sure that the `GET: /` request goes through, and returns `{ message: 'hello, world' }` as defined:

```
experiment('Base API', () =>
{
  test('GET: /', () =>
  {
    const options =
    {
      ...
      assert.equal(response.result.message, 'hello, world');
    });
  });
});
```



You can find the complete code from the `test-todo.js` file at `Code/Lesson-3/exercise-c/test`.

We now see `experiment`, `test`, and `assert.equal` methods in action. `experiment` is basically a way of grouping together the tests, and the actual tests are written within the `test` method's

callback function (known as test cases). `assert.equal` here is just comparing the two values to make sure they are equal, and if not, an assertion error will be thrown.

10. Now, let's run our tests:

1. On the Terminal (open a new Terminal if you're running the API on one of them), navigate to the root of our project and run the following command:

```
PORT=8001 ./node_modules/lab/bin/lab test --leaks
```

We're adding an optional `--leaks` option to turn off memory leak detection since we don't need it now.



At the beginning of the command, we're adding `PORT=8001`; this is a way of passing an environment variable to our script, which is why we changed our code at step 7 previously. We're now running our test server on port `8001` while our development server is still running on port `8000`.

2. When you run the command, you should see something close to this:

```
→ activity-c git:(master) X PORT=8001 ./node_modules/lab/bin/lab test --leaks
Server running at: http://localhost:8001
171128/041815.596, [response] http://localhost:8001: get / {} 200 (12ms)

.

1 tests complete
Test duration: 38 ms
```

11. We can make our test command shorter by adding it as a script on our `package.json` file:

1. Replace the following line of code:

```
"test": "echo \"Error: no test specified\" && exit 1"
```

2. With the following line:

```
"test": "PORT=8001 ./node_modules/lab/bin/lab test --leaks"
```

3. Now, go back to the Terminal and just run:

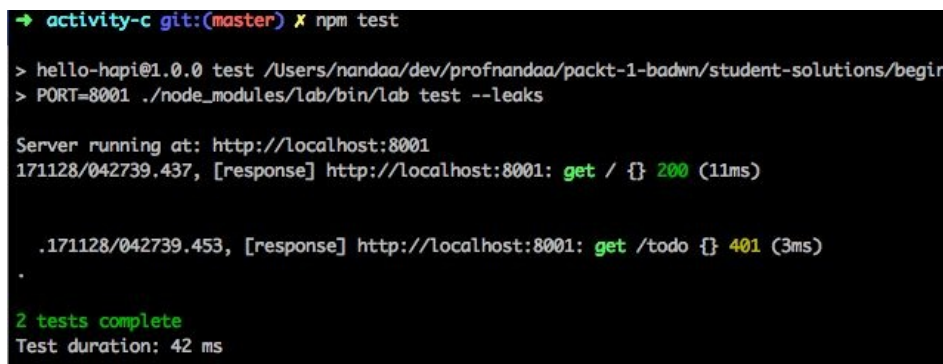
```
npm test
```

12. Now, let's test that our authentication is working correctly. Add the following segment after the previous one:

```
experiment('Authentication', () =>
```

```
{
  test('GET: /todo without auth', () =>
  {
    const options =
    {
      method: 'GET',
      url: '/todo'
    };
    server.inject(options, (response) =>
    {
      assert.equal(response.statusCode, 401);
    });
  });
});
```

13. Now, go back and run `npm test`. Both tests should be passing:



```
→ activity-c git:(master) X npm test
> hello-hapi@1.0.0 test /Users/nandaa/dev/profnandaa/packt-1-badwn/student-solutions/begin
> PORT=8001 ./node_modules/lab/bin/lab test --leaks

Server running at: http://localhost:8001
171128/042739.437, [response] http://localhost:8001: get / {} 200 (11ms)

.171128/042739.453, [response] http://localhost:8001: get /todo {} 401 (3ms)
.
2 tests complete
Test duration: 42 ms
```

14. You realize that we're having to go back to the Terminal every other time to run the tests. This is just as cumbersome as testing on the API client (Insomnia); we need some automation:

1. We will need `gulp.js` for this, and two other Gulp plugins. Let's install them:

```
install gulp gulp-shell gulp-watch --save-dev
```

2. Now, let's write a simple `gulpfile.js` at the root of our project to automate our testing task:

```
const gulp = require('gulp');
const shell = require('gulp-shell');
const watch = require('gulp-watch');
...
gulp.task('test', shell.task('npm test'));
```



You can find the complete code from the `gulpfile.js` file at [Code/Lesson-3/exercise-c](#).

3. Now, let's go to `package.json` and add another script option for our `gulp` task, next to the previous `test`:

```
"scripts":
{
  "test": "PORT=8001 ./node_modules/lab/bin/lab test --leaks",
  "test:dev": "./node_modules/.bin/gulp test:dev"
},
```

4. Now, go to the Terminal and instead of `npm test`, run the following:

```
npm run test:dev
```

5. The watch task will be fired up, and therefore, any changes made in any of the files within the `src` array in the preceding point, the test will automatically run. This means you can go on with your development work and periodically check that the tests are all good:



```
→ activity-c git:(master) X npm run test:dev
> hello-hapi@1.0.0 test:dev /Users/nandaa/dev/profnandaa/packt-1-badwn/student
> gulp test:dev

[08:17:14] Using gulpfile ~/dev/profnandaa/packt-1-badwn/student-solutions/be
[08:17:14] Starting 'test:dev'...
[08:17:14] Finished 'test:dev' after 2.41 ms
```

15. Let's now write a sample test for `GET: /todo` route. Remember that for all the authenticated routes, we need the token first, for us to make a successful request. We will therefore need a script to get us the token before any tests begin. This is where the `before` function that we got in step 8 kicks in. In our `test-todo.js` file, add the following segment:

```
experiment('/todo/* routes', () =>
{
  const headers =
  {
    Authorization: 'Bearer ',
  };
  before(() =>
  {
    const options =
    {
      method: 'POST',
      url: '/auth',
      ...
    };
  });
});
```



You can find the complete code from the `test-todo.js` file at `Code/Lesson-3/exercise-c/test`.

Summary

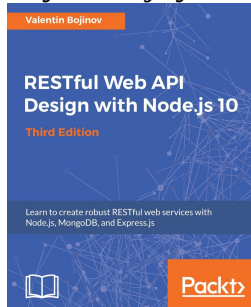
In this chapter, we have explored quite a lot. We started off with introducing Knex.js and how we can use it to connect and use the database. We went through the essential CRUD database methods. We then covered how we can authenticate our API and prevent it from unauthorized access, using the JWT mechanism. We also mentioned something important about CORS, how the browsers handle this and how we can enable this on our API. We finally finished off with covering concepts about testing our API, using the Lab library. We also covered, in passing, the concept of test automation using gulp.js.

In this book, we started off with learning how to implement the necessary modules to get simple applications up and running. We then moved on to implementing the `async` and `await` functions to handle asynchronous code efficiently. After a primer on Node.js (the application building aspect), we graduated to building an API using Node.js. To do this, we initially used the built-in module and then utilized the rich Hapi.js framework. We also understood the advantages of the Hapi.js framework. Later on, we learned how to handle requests from API clients and finally, we completed the book by covering interactions with databases.

This is a practical quick-start guide. To further your knowledge, you should consider building real-time applications with Node.js. We have recommended a few books in the next section, but ensure you check our website to find other books that may interest you!

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

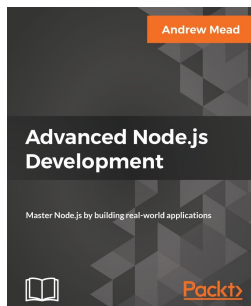


RESTful Web API Design with Node.js 10 - Third Edition

Valentin Bojinov

ISBN: 978-1-78862-332-2

- Install, develop, and test your own Node.js user modules
- Understand the differences between HTTP and RESTful applications
- Use self-descriptive URLs and set accurate HTTP status codes
- Eliminate third-party dependencies in your tests with mocking
- Implement automation tests for a REST-enabled endpoint with Mocha
- Secure your services with NoSQL database integration within Node.js applications
- Integrate a simple frontend using JavaScript libraries available on a CDN server



Advanced Node.js Development

Andrew Mead

ISBN: 978-1-78839-393-5

- Develop, test, and deploy real-world Node.js applications
- Master Node.js by building practical, working examples
- Use awesome third-party Node modules such as MongoDB, Mongoose, Socket.io, and Express
- Create real-time web applications
- Explore async and await in ES7

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!