FITPED

Frameworks for backend application development (NodeJS, Laravel)

Wojciech Baran
Dominik Halvoník

www.fitped.eu

2021

Co-funded by the Erasmus+ Programme of the European Union

Work-Based Learning in Future IT Professionals Education
(Grant. no. 2018-1-SK01-KA203-046382)

Frameworks for Backend Application Development (NodeJS, Laravel)

Published on

November 2021

Authors

Wojciech Baran | Pedagogical University of Cracow, Poland

Dominik Halvoník | Constantine the Philosopher University in Nitra, Slovakia

Reviewers

Jozef Kapusta | Pedagogical University of Cracow, Poland

Peter Švec | Teacher.sk, Slovakia

Eugenia Smyrnova-Trybulska | University of Silesia in Katowice, Poland

Piet Kommers | Helix5, Netherland

Graphics

Ľubomír Benko | Constantine the Philosopher University in Nitra, Slovakia

David Sabol | Constantine the Philosopher University in Nitra, Slovakia

Erasmus+ FITPED

Work-Based Learning in Future IT Professionals Education

Project 2018-1-SK01-KA203-046382

Co-funded by the
Erasmus+ Programme
of the European Union

ISBN 978-80-558-1793-4

Table of Contents

NodeJS

Introduction

Chapter 1

1.1 What is Node.js?

🕮 1.1.1

The course is intended for people who are just starting to learn Node.js.

However, to understand the topic and the presented examples, you should already be familiar with JavaScript.

You should also know the object-oriented approach, understand classes and actions on objects.

To work with the code, you need a code editor, eg VisualStudioCode, and a program that will handle the command line.

🕮 1.1.2

Node.js is an open source runtime runtime environment that is used to execute JavaScript code. Another runtime environment may be a browser.

Node.js makes it easier to create web applications. It is also a tool for creating server side applications, all kinds of scripts and libraries. It can be an alternative to PHP or other language frameworks.

🕮 1.1.3

Node.js is very useful for frontend developers who are familiar with JavaScript. They can write server-side code in the same language as the client-side code. Node.js also allows you to use the new ECMAScript standards.

📝 1.1.4

Node.js runs on ...?

- server
- client
- server and client

🕮 1.1.5

Both your browser JavaScript and Node. js use JavaScript as their programming language. However, building applications that run in the browser is quite a different thing than building a Node.js application.

Working in the browser is mainly about interacting with the DOM or other Web Platform APIs. They don't exist in Node.js. There is also no window, document and other objects provided by the browser. In the browser, we don't have all the APIs that Node.js provides through its modules.

Another difference is that in Node.js you control the environment. If you are not developing an open source application, you know which version of Node.js you will run your application in. Compared to a browser environment where you don't have the choice of which browser your users will use. This allows you to use the latest versions of ECMAScript from Node.js. Unlike browsers where updates can be a bit slow, so you'll have to consider older versions of ECMAScript at times.

🕮 1.1.6

Node.js is written in C ++ and uses the V8 engine on which the Chrome browser is also built. He is mainly responsible for compiling JavaScript code into machine code and for executing this code.

Firefox, Safari, Edge have their own JavaScript engine. For example firefox has SpiderMonkey.

1.1.7

Which of the followings are valid languages for Node.js?

- JavaScript
- C#
- Python

1.1.8

What language is Node.js written in?

- C++
- JavaScript
- Java

📖 1.1.9

Node.js is single-threaded. This means that all requests to the server are executed on the same thread (they are not run in separate processes that could be executing simultaneously). Such a model is very efficient in terms of speed of execution and use of server resources, but it causes that if a function that takes a long time to execute is called synchronously, it will block not only the current request, but also all other requests handled by the application at that time.

📝 1.1.10

Is Node.js single-threaded? Yes or No?

📖 1.1.11

Node.js uses an event-based model with non-blocking I/O to maximize the use of a single CPU and computer memory (asynchronicity). If an event occurs, the code responsible for a given action is executed in response, otherwise the program waits and does not load the processor.

📖 1.1.12

Node.js is highly scalable. Thanks to the built-in modules in Node.js, we are able to easily scale our application and adapt it to a greater load. We can create additional processes. Although Node uses only one processor by default, we can easily scale it to the number of processors we have in the computer.

📖 1.1.13

Node.js is great for creating single-page applications, JSON API-based applications, data streaming applications and data intensive real-time applications . However,

Node.js is not recommended for CPU-intensive applications.

📝 1.1.14

Is Node.js asynchronous?

- Yes
- No

🕮 1.1.15

Node.js has a standard package manager called npm. The main task of npm is the installation of modules and their possible configuration. At www.npmjs.com you will find thousands of free packages to download and use. Npm is installed on your computer along with the Node.js installation, which you will learn about in the next section.

1.2 Environment setup and first program

🕮 1.2.1

Node.js is supported by i.a. Microsoft Azure, Amazon Web Services or Google Cloud Platform.

It is a cross-platform runtime environment, i.e. it can run on Linux, MacOS or Windows.

1.2.2

Which system can node.js run on?

- Windows
- Linux
- MacOS
- On all mentioned

🕮 1.2.3

To check if you have Node.js installed on your computer, start the console and type the command:

node -v or node --version

In response, we should receive the version that is installed, if we do not have it installed, an error will appear. Together with the Node.js environment, the npm

manager is also installed. We can test the correctness of the installation in the same way:

npm -v or npm --version

📝 1.2.4

Is nmp installed with Node.js?

- Yes
- No

🕮 1.2.5

If it turns out that we do not have Node.js installed, we must choose the easiest installation method for a given system. For Windows and MacOS, just use the installer. The installer can be downloaded from https://nodejs.org/en/download/. We can choose the latest version or a slightly older version, recommended for most users. After downloading the file, just follow the instructions.

🕮 1.2.6

On Linux systems, use the terminal. For example, for Ubuntu, the installation can be done with two commands:

curl -fsSL https://deb.nodesource.com/setup_15.x | sudo -E bash -

sudo apt-get install -y nodejs

🕮 1.2.7

REPL (Read Eval Print Loop) is a computer environment, such as a Windows console or Unix / Linux shell, where you enter commands and the system responds by displaying the output. Node.js is packaged with the REPL environment. From the expanded name of the environment, you can read the tasks for which it is responsible:

Read - reads user input

Eval - retrieves and evaluates the data structure.

Print - prints the result

Loop - loops the command until the user presses ctrl-c twice.

🕮 1.2.8

Let's take a look at the most important REPL commands:

help – list of all commands

Up/Down Keys – see command history and modify previous commands

tab Keys – list of current commands

save filename – save the current Node REPL session to a file

load filename – load file content in current Node REPL session

1.3 Introduction (Exercises)

1.3.1

Node.js runs on ...?

- server
- client
- server and client

1.3.2

Which of the followings are valid languages for Node.js?

- C#
- JavaScript
- Python

1.3.3

What language is Node.js written in?

- JavaScript

- Java
- C++

1.3.4

Is Node.js single-threaded? Yes or No?

1.3.5

Is Node.js asynchronous?

- Yes
- No

1.3.6

Is nmp installed with Node.js?

- Yes
- No

Event Control

Chapter 2

2.1 Events in Node.js

🕮 2.1.1

Events in Node.js

Every action on a computer is an event. Like when a connection is made or a file is opened.

Objects in Node.js can fire events, like the readStream object fires events when opening and closing a file:

Example

```
var fs = require('fs');
var rs = fs.createReadStream('./demofile.txt');
rs.on('open', function () {
 console.log('The file is open');
});
```

🕮 2.1.2

Events Module

Node.js has a built-in module, called "Events", where you can create-, fire-, and listen for- your own events.

To include the built-in Events module use the **require()** method. In addition, all event properties and methods are an instance of an EventEmitter object. To be able to access these properties and methods, create an EventEmitter object:

```
var events = require('events');
var eventEmitter = new events.EventEmitter();
```

2.2 EventEmitter

🕮 2.2.1

Node.js EventEmitter

Node.js allows us to create and handle custom events easily by using events module. Event module includes EventEmitter class which can be used to raise and handle custom events.

📖 2.2.2

The following example demonstrates EventEmitter class for raising and handling a custom event.

Example: Raise and Handle Node.js events

```
// get the reference of EventEmitter class of events module
var events = require('events');

//create an object of EventEmitter class by using above
reference
var em = new events.EventEmitter();

//Subscribe for FirstEvent
em.on('FirstEvent', function (data) {
    console.log('First subscriber: ' + data);
});

// Raising FirstEvent
em.emit('FirstEvent', 'This is my first Node.js event emitter
example.');
```

In the above example, we first import the 'events' module and then create an object of EventEmitter class. We then specify event handler function using on() function. The on() method requires name of the event to handle and callback function which is called when an event is raised.

The emit() function raises the specified event. First parameter is name of the event as a string and then arguments. An event can be emitted with zero or more arguments. You can specify any name for a custom event in the emit() function.

📖 2.2.3

You can also use addListener() methods to subscribe for an event as shown below.

Example: EventEmitter

```
var emitter = require('events').EventEmitter;

var em = new emitter();

//Subscribe FirstEvent
em.addListener('FirstEvent', function (data) {
```

```
    console.log('First subscriber: ' + data);
});

//Subscribe SecondEvent
em.on('SecondEvent', function (data) {
    console.log('First subscriber: ' + data);
});

// Raising FirstEvent
em.emit('FirstEvent', 'This is my first Node.js event emitter
example.');

// Raising SecondEvent
em.emit('SecondEvent', 'This is my second Node.js event
emitter example.');
```

🕮 2.2.4

The following lists all the important methods of EventEmitter class.

emitter.addListener(event, listener) - Adds a listener to the end of the listeners array for the specified event. No checks are made to see if the listener has already been added.

emitter.on(event, listener) - Adds a listener to the end of the listeners array for the specified event. No checks are made to see if the listener has already been added. It can also be called as an alias of emitter.addListener()

emitter.once(event, listener) - Adds a one time listener for the event. This listener is invoked only the next time the event is fired, after which it is removed.

emitter.removeListener(event, listener) - Removes a listener from the listener array for the specified event. Caution: changes array indices in the listener array behind the listener.

emitter.removeAllListeners([event]) - Removes all listeners, or those of the specified event.

emitter.setMaxListeners(n) - By default EventEmitters will print a warning if more than 10 listeners are added for a particular event.

emitter.getMaxListeners() - Returns the current maximum listener value for the emitter which is either set by emitter.setMaxListeners(n) - or defaults to EventEmitter.defaultMaxListeners.

emitter.listeners(event) - Returns a copy of the array of listeners for the specified event.

emitter.emit(event[, arg1][, arg2][, ...]) - Raise the specified events with the supplied arguments.

emitter.listenerCount(type) - Returns the number of listeners listening to the type of event.

🕮 2.2.5

Common Patterns for EventEmitters

There are two common patterns that can be used to raise and bind an event using EventEmitter class in Node.js.

1. Return EventEmitter from a function
2. Extend the EventEmitter class

🕮 2.2.6

Return EventEmitter from a function

In this pattern, a constructor function returns an EventEmitter object, which was used to emit events inside a function. This EventEmitter object can be used to subscribe for the events. Consider the following example.

Example: Return EventEmitter from a function

```
var emitter = require('events').EventEmitter;

function LoopProcessor(num) {
    var e = new emitter();

    setTimeout(function () {

        for (var i = 1; i <= num; i++) {
            e.emit('BeforeProcess', i);

            console.log('Processing number:' + i);

            e.emit('AfterProcess', i);
        }
```

```
    }
    , 2000)

    return e;
}
var lp = LoopProcessor(3);

lp.on('BeforeProcess', function (data) {
    console.log('About to start the process for ' + data);
});

lp.on('AfterProcess', function (data) {
    console.log('Completed processing ' + data);
});
```

Output:

About to start the process for 1

Processing number:1

Completed processing 1

About to start the process for 2

Processing number:2

Completed processing 2

About to start the process for 3

Processing number:3

Completed processing 3In the above LoopProcessor() function, first we create an object of EventEmitter class and then use it to emit 'BeforeProcess' and 'AfterProcess' events. Finally, we return an object of EventEmitter from the function. So now, we can use the return value of LoopProcessor function to bind these events using on() or addListener() function.

🕮 2.2.7

Extend EventEmitter Class

In this pattern, we can extend the constructor function from EventEmitter class to emit the events.

Example: Extend EventEmitter Class

```
var emitter = require('events').EventEmitter;

var util = require('util');

function LoopProcessor(num) {
    var me = this;

    setTimeout(function () {

        for (var i = 1; i <= num; i++) {
            me.emit('BeforeProcess', i);

            console.log('Processing number:' + i);

            me.emit('AfterProcess', i);
        }
    }
    , 2000)

    return this;
}

util.inherits(LoopProcessor, emitter)

var lp = new LoopProcessor(3);

lp.on('BeforeProcess', function (data) {
    console.log('About to start the process for ' + data);
});

lp.on('AfterProcess', function (data) {
    console.log('Completed processing ' + data);
});
```

Output:

About to start the process for 1

Processing number:1

Completed processing 1

About to start the process for 2

Processing number:2

Completed processing 2

About to start the process for 3

Processing number:3

Completed processing 3In the above example, we have extended LoopProcessor constructor function with EventEmitter class using **util.inherits()** method of utility module. So, you can use EventEmitter's methods with LoopProcessor object to handle its own events.

In this way, you can use EventEmitter class to raise and handle custom events in Node.js.

2.3 Event

2.3.1

By default EventEmitters will print a warning if more than 10 listeners are added for a particular event.

- emitter.setMaxListeners(n)
- emitter.getMaxListeners()
- emitter.once(event, listener)

2.3.2

Returns the number of listeners listening to the type of event.

- emitter.emit(event[, arg1][, arg2][, ...])
- emitter.on(event, listener)
- emitter.listenerCount(type)

2.3.3

What can the EventEmitter class in the Event module be used for?

Modules

Chapter 3

3.1 Modules

🕮 3.1.1

Node.js Module

Module in Node.js is a simple or complex functionality organized in single or multiple JavaScript files which can be reused throughout the Node.js application.

Each module in Node.js has its own context, so it cannot interfere with other modules or pollute global scope. Also, each module can be placed in a separate .js file under a separate folder.

Node.js implements CommonJS modules standard. CommonJS is a group of volunteers who define JavaScript standards for web server, desktop, and console application.

🕮 3.1.2

Node.js Module Types

Node.js includes three types of modules:

1. Core Modules
2. Local Modules
3. Third Party Modules

🕮 3.1.3

Node.js Core Modules

Node.js is a light weight framework. The core modules include bare minimum functionalities of Node.js. These core modules are compiled into its binary distribution and load automatically when Node.js process starts. However, you need to import the core module first in order to use it in your application.

The following table lists some of the important core modules in Node.js.

http - http module includes classes, methods and events to create Node.js http server.

url - url module includes methods for URL resolution and parsing.

querystring - querystring module includes methods to deal with query string.

path - path module includes methods to deal with file paths.

fs - fs module includes classes, methods, and events to work with file I/O.

util - util module includes utility functions useful for programmers.

🕮 3.1.4

Loading Core Modules

In order to use Node.js core or NPM modules, you first need to import it using require() function as shown below.

```
var module = require('module_name');
```

As per above syntax, specify the module name in the require() function. The require() function will return an object, function, property or any other JavaScript type, depending on what the specified module returns.

The following example demonstrates how to use Node.js http module to create a web server.

Example: Load and Use Core http Module

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

var server = http.createServer(function(req, res){

  //write code here

});

server.listen(5000);
```

In the above example, require() function returns an object because http module returns its functionality as an object, you can then use its properties and methods using dot notation e.g. http.createServer().

In this way, you can load and use Node.js core modules in your application. We will be using core modules throughout these tutorials.

3.2 Locacl modules

🕮 3.2.1

Node.js Local Module

Local modules are modules created locally in your Node.js application. These modules include different functionalities of your application in separate files and folders. You can also package it and distribute it via NPM, so that Node.js community can use it. For example, if you need to connect to MongoDB and fetch data then you can create a module for it, which can be reused in your application.

🕮 3.2.2

Writing Simple Module

Let's write simple logging module which logs the information, warning or error to the console.

In Node.js, module should be placed in a separate JavaScript file. So, create a Log.js file and write the following code in it.

Log.js

```
var log = {
            info: function (info) {
                console.log('Info: ' + info);
            },
            warning:function (warning) {
                console.log('Warning: ' + warning);
            },
            error:function (error) {
                console.log('Error: ' + error);
            }
    };

module.exports = log
```

In the above example of logging module, we have created an object with three functions - info(), warning() and error(). At the end, we have assigned this object to module.exports. The module.exports in the above example exposes a log object as a module.

The *module.exports* is a special object which is included in every JS file in the Node.js application by default. Use module.exports or exports to expose a function, object or variable as a module in Node.js.

Now, let's see how to use the above logging module in our application.

🕮 3.2.3

Loading Local Module

To use local modules in your application, you need to load it using require() function in the same way as core module. However, you need to specify the path of JavaScript file of the module.

The following example demonstrates how to use the above logging module contained in Log.js.

app.js Copy

```
var myLogModule = require('./Log.js');

myLogModule.info('Node.js started');
```

In the above example, app.js is using log module. First, it loads the logging module using require() function and specified path where logging module is stored. Logging module is contained in Log.js file in the root folder. So, we have specified the path './Log.js' in the require() function. The '.' denotes a root folder.

The require() function returns a log object because logging module exposes an object in Log.js using module.exports. So now you can use logging module as an object and call any of its function using dot notation e.g myLogModule.info() or myLogModule.warning() or myLogModule.error()

Run the above example using command prompt (in Windows) as shown below.

C:\> node app.js

Info: Node.js startedThus, you can create a local module using module.exports and use it in your application.

3.3 Export Module

🕮 3.3.1

Export Module in Node.js

The **module.exports** is a special object which is included in every JavaScript file in the Node.js application by default. The **module** is a variable that represents the

current module, and **exports** is an object that will be exposed as a module. So, whatever you assign to **module.exports** will be exposed as a module.

🕮 3.3.2

Export Literals

As mentioned above, **exports** is an object. So it exposes whatever you assigned to it as a module. For example, if you assign a string literal then it will expose that string literal as a module.

The following example exposes simple string message as a module in Message.js.

Message.js

```
module.exports = 'Hello world';
```

Now, import this message module and use it as shown below.

app.js

```
var msg = require('./Messages.js');

console.log(msg);
```

Run the above example and see the result, as shown below.

C:\> node app.js

Hello World Note:

You must specify **./** as a path of root folder to import a local module. However, you do not need to specify the path to import Node.js core modules or NPM modules in the **require()** function.

🕮 3.3.3

Export Object

The **exports** is an object. So, you can attach properties or methods to it. The following example exposes an object with a string property in **Message.js** file.

Message.js

```
exports.SimpleMessage = 'Hello world';

//or

module.exports.SimpleMessage = 'Hello world';
```

In the above example, we have attached a property **SimpleMessage** to the exports object. Now, import and use this module, as shown below.

app.js

```
var msg = require('./Messages.js');

console.log(msg.SimpleMessage);
```

In the above example, the **require()** function will return an object **{ SimpleMessage : 'Hello World'}** and assign it to the msg variable. So, now you can use **msg.SimpleMessage**.

Run the above example by writing **node app.js** in the command prompt and see the output as shown below.

C:\> node app.js

Hello WorldIn the same way as above, you can expose an object with function. The following example exposes an object with the **log** function as a module.

Log.js

```
module.exports.log = function (msg) {
    console.log(msg);
};
```

The above module will expose an object- **{ log : function(msg){ console.log(msg); } }** . Use the above module as shown below.

app.js

```
var msg = require('./Log.js');

msg.log('Hello World');
```

Run and see the output in command prompt as shown below.

C:\> node app.js

Hello WorldYou can also attach an object to **module.exports**, as shown below.

data.js

```
module.exports = {
    firstName: 'James',
    lastName: 'Bond'
}
```

app.js

```
var person = require('./data.js');
console.log(person.firstName + ' ' + person.lastName);
```

Run the above example and see the result, as shown below.

C:\> node app.js

James Bond

🕮 3.3.4

Export Function

You can attach an anonymous function to exports object as shown below.

Log.js

```
module.exports = function (msg) {
    console.log(msg);
};
```

Now, you can use the above module, as shown below.

app.js

```
var msg = require('./Log.js');

msg('Hello World');
```

The **msg** variable becomes a function expression in the above example. So, you can invoke the function using parenthesis **()**. Run the above example and see the output as shown below.

C:\> node app.js

Hello World

🕮 3.3.5

Export Function as a Class

In JavaScript, a function can be treated like a class. The following example exposes a function that can be used like a class.

Person.js

```
module.exports = function (firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.fullName = function () {
        return this.firstName + ' ' + this.lastName;
    }
}
```

The above module can be used, as shown below.

app.js

```
var person = require('./Person.js');

var person1 = new person('James', 'Bond');

console.log(person1.fullName());
```

As you can see, we have created a **person** object using the **new** keyword. Run the above example, as shown below.

C:\> node app.js

James BondIn this way, you can export and import a local module created in a separate file under root folder.

Node.js also allows you to create modules in sub folders. Let's see how to load module from sub folders.

🕮 3.3.6

Load Module from the Separate Folder

Use the full path of a module file where you have exported it using **module.exports**. For example, if the log module in the log.js is stored under the **utility** folder under the root folder of your application, then import it, as shown below.

app.js

```
var log = require('./utility/log.js');
```

In the above example, **.** is for the root folder, and then specify the exact path of your module file. Node.js also allows us to specify the path to the folder without specifying the file name. For example, you can specify only the utility folder without specifying **log.js**, as shown below.

app.js

```
var log = require('./utility');
```

In the above example, Node.js will search for a package definition file called **package.json** inside the utility folder. This is because Node assumes that this folder is a package and will try to look for a package definition.
The **package.json** file should be in a module directory. The **package.json** under utility folder specifies the file name using the **main** key, as shown below.

./utility/package.json

```
{
    "name" : "log",
    "main" : "./log.js"
}
```

Now, Node.js will find the **log.js** file using the **main** entry in **package.json** and import it.

Note:

If the **package.json** file does not exist, then it will look for index.js file as a module file by default.

3.4 Modules (Exercises)

3.4.1

module includes methods for URL resolution and parsing.

3.4.2

module includes classes, methods and events to create Node.js http server -

3.4.3

module includes classes, methods, and events to work with file I/O -

3.4.4

module includes methods to deal with file paths -

3.4.5

module includes methods to deal with query string -

3.4.6

module includes utility functions useful for programmers -

3.4.7

In Node.js, module should be placed in a separate JavaScript file

- True
- False

3.4.8

The export is not an object.

- True
- False

3.4.9

In JavaScript, a function can be thought of as?

Module Manager – NPM

Chapter 4

4.1 What is NPM?

🕮 4.1.1

npm is two things: first and foremost, it is an online repository for the publishing of open-source Node.js projects; second, it is a command-line utility for interacting with said repository that aids in package installation, version management, and dependency management. A plethora of Node.js libraries and applications are published on npm, and many more are added every day. These applications can be searched for on https://www.npmjs.com/. Once you have a package you want to install, it can be installed with a single command-line command.

Let's say you're hard at work one day, developing the Next Great Application. You come across a problem, and you decide that it's time to use that cool library you keep hearing about - let's use Caolan McMahon's async as an example. Thankfully, **npm** is very simple to use: you only have to run **npm install async**, and the specified module will be installed in the current directory under **./node_modules/**. Once installed to your **node_modules** folder, you'll be able to use **require()** on them just like they were built-ins.

Let's look at an example of a global install - let's say **coffee-script**. The npm command is simple: **npm install coffee-script -g**. This will typically install the program and put a symlink to it in **/usr/local/bin/**. This will then allow you to run the program from the console just like any other CLI tool. In this case, running **coffee** will now allow you to use the coffee-script REPL.

Another important use for npm is dependency management. When you have a node project with a package.json file, you can run **npm install** from the project root and npm will install all the dependencies listed in the package.json. This makes installing a Node.js project from a git repo much easier! For example, **vows**, a Node.js testing framework, can be installed from git, and its single dependency, **eyes**, can be automatically handled:

Example:

```
git clone https://github.com/cloudhead/vows.git
cd vows
npm install
```

After running those commands, you will see a **node_modules** folder containing all of the project dependencies specified in the package.json.

4.2 JSON

🕮 4.2.1

JavaScript Object Notation, or JSON, is a lightweight data format that has become the defacto standard for the web. JSON can be represented as either a list of values, e.g. an Array, or a hash of properties and values, e.g. an Object.

```
// a JSON array
["one", "two", "three"]

// a JSON object
{ "one": 1, "two": 2, "three": 3 }
```

🕮 4.2.2

Encoding and Decoding

JavaScript provides 2 methods for encoding data structures to json and encoding json back to JavaScript objects and arrays. They are both available on the **JSON** object that is available in the global scope.

JSON.stringify takes a JavaScript object or array and returns a serialized string in the JSON format.

```
const data = {
  name: "John Doe",
  age: 32,
  title: "Vice President of JavaScript"
}

const jsonStr = JSON.stringify(data);

console.log(jsonStr);

// prints '{"name":"John Doe","age":32,"title":"Vice President
of JavaScript"}'
```

JSON.parse takes a JSON string and decodes it to a JavaScript data structure.

```
const jsonStr = '{"name":"John Doe","age":32,"title":"Vice
President of JavaScript"}';

const data = JSON.parse(jsonStr);

console.log(data.title);

// prints 'Vice President of JavaScript'
```

🕮 4.2.3

What is valid JSON?

There are a few rules to remember when dealing with data in JSON format. There are several gotchas that can produce invalid JSON as well.

- Empty objects and arrays are okay
- Strings can contain any unicode character, this includes object properties
- **null** is a valid JSON value on it's own
- All object properties should always be double quoted
- Object property values must be one of the following: String, Number, Boolean, Object, Array, null
- Number values must be in decimal format, no octal or hex representations
- Trailing commas on arrays are not allowed

These are all examples of valid JSON.

```
{"name":"John Doe","age":32,"title":"Vice President of
JavaScript"}

["one", "two", "three"]

// nesting valid values is okay
{"names": ["John Doe", "Jane Doe"] }

[ { "name": "John Doe"}, {"name": "Jane Doe"} ]

{} // empty hash

[] // empty list

null

{ "key": "\uFDD0" } // unicode escape codes
```

These are all examples of bad JSON formatting.

```
{ name: "John Doe", 'age': 32 } // name and age should be in
double quotes

[32, 64, 128, 0xFFF] // hex numbers are not allowed

{ "name": "John Doe", "age": undefined } // undefined is an
invalid value

// functions and dates are not allowed
{ "name": "John Doe",
  "birthday": new Date('Fri, 26 Jan 2019 07:13:10 GMT'),
  "getName": function() {
```

```
        return this.name;
    }
}
```

Calling **JSON.parse** with an invalid JSON string will result in a SyntaxError being thrown. If you are not sure of the validity of your JSON data, you can anticipate errors by wrapping the call in a try/catch block.

Notice that the only complex values allowed in JSON are objects and arrays. Functions, dates and other types are excluded. This may not seem to make sense at first. But remember that JSON is a data format, not a format for transferring complex JavaScript objects along with their functionality.

🕮 4.2.4

JSON Validators

As JSON has become the most widely used data formate with well-defined rules to abide by, there are many validators available to assist your workflow:

- Online Validators: If you are just playing around with JSON or checking someone's JSON (without IDEs/editors) then online validators could be of great help. For instance: jsonlint.com is a good online JSON validator and reformatter.
- npm Packages: If you are working with a team and want JSON Validation baked into your project or simply like to automate validation in your workflow then the large collection of npm packages are at your disposal. For instance: jsonlint is a pure JavaScript version of the service provided at **jsonlint.com**.
- Plugins for IDEs/editors: There are many plugins/extensions available for most of the IDEs/editors which validate JSON for you. Some editors like **VS Code** come with JSON IntelliSense & Validation out of the box.

WEB Server

Chapter 5

5.1 Node.js Web Server

🕮 5.1.1

To access web pages of any web application, you need a web server. The web server will handle all the http requests for the web application e.g IIS is a web server for ASP.NET web applications and Apache is a web server for PHP or Java web applications.

Node.js provides capabilities to create your own web server which will handle HTTP requests asynchronously. You can use IIS or Apache to run Node.js web application but it is recommended to use Node.js web server.

🕮 5.1.2

Create Node.js Web Server

Node.js makes it easy to create a simple web server that processes incoming requests asynchronously.

The following example is a simple Node.js web server contained in server.js file.

server.js

```
var http = require('http'); // 1 - Import Node.js core module

var server = http.createServer(function (req, res) {   // 2 - creating server

    //handle incomming requests here..

});

server.listen(5000); //3 - listen for any incoming requests

console.log('Node.js web server at port 5000 is running..')
```

In the above example, we import the http module using require() function. The http module is a core module of Node.js, so no need to install it using NPM. The next step is to call createServer() method of http and specify callback function with request and response parameter. Finally, call listen() method of server object which was returned from createServer() method with port number, to start listening to incoming requests on port 5000. You can specify any unused port here.

Run the above web server by writing **node server.js** command in command prompt or terminal window and it will display message as shown below.

C:\> node server.js

Node.js web server at port 5000 is running..This is how you create a Node.js web server using simple steps. Now, let's see how to handle HTTP request and send response in Node.js web server.

🕮 5.1.3

Handle HTTP Request

The http.createServer() method includes request and response parameters which is supplied by Node.js. The request object can be used to get information about the current HTTP request e.g., url, request header, and data. The response object can be used to send a response for a current HTTP request.

The following example demonstrates handling HTTP request and response in Node.js.

server.js

```
var http = require('http'); // Import Node.js core module

var server = http.createServer(function (req, res) {
//create web server
    if (req.url == '/') { //check the URL of the current
request

        // set response header
        res.writeHead(200, { 'Content-Type': 'text/html' });

        // set response content
        res.write('<html><body><p>This is home
Page.</p></body></html>');
        res.end();

    }
    else if (req.url == "/student") {

        res.writeHead(200, { 'Content-Type': 'text/html' });
        res.write('<html><body><p>This is student
Page.</p></body></html>');
        res.end();

    }
    else if (req.url == "/admin") {

        res.writeHead(200, { 'Content-Type': 'text/html' });
```

```
        res.write('<html><body><p>This is admin
Page.</p></body></html>');
        res.end();

    }
    else
        res.end('Invalid Request!');

});

server.listen(5000); //6 - listen for any incoming requests

console.log('Node.js web server at port 5000 is running..')
```

In the above example, req.url is used to check the url of the current request and based on that it sends the response. To send a response, first it sets the response header using writeHead() method and then writes a string as a response body using write() method. Finally, Node.js web server sends the response using end() method.

Now, run the above web server as shown below.

C:\> node server.js

Node.js web server at port 5000 is running..To test it, you can use the command-line program curl, which most Mac and Linux machines have pre-installed.

curl -i http://localhost:5000

You should see the following response.

HTTP/1.1 200 OK

Content-Type: text/plain

Date: Tue, 8 Sep 2015 03:05:08 GMT

Connection: keep-alive

This is home page.

For Windows users, point your browser to *http://localhost:5000*

The same way, point your browser to http://localhost:5000/student

📖 5.1.4

Sending JSON Response

The following example demonstrates how to serve JSON response from the Node.js web server.

server.js

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

var server = http.createServer(function (req, res) {

    if (req.url == '/data') { //check the URL of the current
request
            res.writeHead(200, { 'Content-Type':
'application/json' });
            res.write(JSON.stringify({ message: "Hello
World"}));
            res.end();
    }
});

server.listen(5000);

console.log('Node.js web server at port 5000 is running..')
```

So, this way you can create a simple web server that serves different responses.

5.2 WEB Server (Exercises)

5.2.1

The http.createServer() method includes ... and ... parameters which is supplied by Node.js

5.2.2

Node.js provides capabilities to create your own web server which will handle HTTP requests asynchronously

- True
- False

File System

Chapter 6

6.1 Node.js File System

🕮 6.1.1

Node.js includes fs module to access physical file system. The fs module is responsible for all the asynchronous or synchronous file I/O operations.

🕮 6.1.2

Reading File

Use fs.readFile() method to read the physical file asynchronously.

Signature:

```
fs.readFile(fileName [,options], callback)
```

Parameter Description:

- filename: Full path and name of the file as a string.
- options: The options parameter can be an object or string which can include encoding and flag. The default encoding is utf8 and default flag is "r".
- callback: A function with two parameters err and fd. This will get called when readFile operation completes.

The following example demonstrates reading existing TestFile.txt asynchronously.

Example: Reading File

```
var fs = require('fs');

fs.readFile('TestFile.txt', function (err, data) {
                    if (err) throw err;

    console.log(data);
});
```

The above example reads TestFile.txt (on Windows) asynchronously and executes callback function when read operation completes. This read operation either throws an error or completes successfully. The err parameter contains error information if any. The data parameter contains the content of the specified file.

The following is a sample TextFile.txt file.

TextFile.txt

```
This is test file to test fs module of Node.js
```

Now, run the above example and see the result as shown below.

C:\> node server.js

This is test file to test fs module of Node.jsUse fs.readFileSync() method to read file synchronously as shown below.

Example: Reading File Synchronously

```
var fs = require('fs');

var data = fs.readFileSync('dummyfile.txt', 'utf8');
console.log(data);
```

🕮 6.1.3

Writing File

Use fs.writeFile() method to write data to a file. If file already exists then it overwrites the existing content otherwise it creates a new file and writes data into it.

Signature:

```
fs.writeFile(filename, data[, options], callback)
```

Parameter Description:

- filename: Full path and name of the file as a string.
- Data: The content to be written in a file.
- options: The options parameter can be an object or string which can include encoding, mode and flag. The default encoding is utf8 and default flag is "r".
- callback: A function with two parameters err and fd. This will get called when write operation completes.

The following example creates a new file called test.txt and writes "Hello World" into it asynchronously.

Example: Creating & Writing File

```
var fs = require('fs');

fs.writeFile('test.txt', 'Hello World!', function (err) {
                        if (err)
        console.log(err);
                        else
```

```
        console.log('Write operation complete.');
});
```

In the same way, use fs.appendFile() method to append the content to an existing file.

Example: Append File Content

```
var fs = require('fs');

fs.appendFile('test.txt', 'Hello World!', function (err) {
                        if (err)
        console.log(err);
                        else
        console.log('Append operation complete.');
});
```

🕮 6.1.4

Open File

Alternatively, you can open a file for reading or writing using fs.open() method.

Signature:

```
fs.open(path, flags[, mode], callback)
```

Parameter Description:

- path: Full path with name of the file as a string.
- Flag: The flag to perform operation
- Mode: The mode for read, write or readwrite. Defaults to 0666 readwrite.
- callback: A function with two parameters err and fd. This will get called when file open operation completes.

Flags

The following table lists all the flags which can be used in read/write operation.

r Open file for reading. An exception occurs if the file does not exist.

r+ Open file for reading and writing. An exception occurs if the file does not exist.

rs Open file for reading in synchronous mode.

rs+ Open file for reading and writing, telling the OS to open it synchronously. See notes for 'rs' about using this with caution.

w Open file for writing. The file is created (if it does not exist) or truncated (if it exists).

wx Like 'w' but fails if path exists.

w+ Open file for reading and writing. The file is created (if it does not exist) or truncated (if it exists).

wx+ Like 'w+' but fails if path exists.

a Open file for appending. The file is created if it does not exist.

ax Like 'a' but fails if path exists.

a+ Open file for reading and appending. The file is created if it does not exist.

ax+ Like 'a+' but fails if path exists.

The following example opens an existing file and reads its content.

Example:File open and read

```
var fs = require('fs');

fs.open('TestFile.txt', 'r', function (err, fd) {

                            if (err) {
                            return console.error(err);
    }

                            var buffr = new Buffer(1024);

    fs.read(fd, buffr, 0, buffr.length, 0, function (err,
bytes) {

                            if (err) throw err;

                            // Print only read bytes to avoid
junk.
                            if (bytes > 0) {
            console.log(buffr.slice(0, bytes).toString());
        }

                            // Close the opened file.
        fs.close(fd, function (err) {
                            if (err) throw err;
        });
    });
});
```

📖 6.1.5

Delete File

Use fs.unlink() method to delete an existing file.

Signature:

```
fs.unlink(path, callback);
```

The following example deletes an existing file.

Example:File Open and Read

```
var fs = require('fs');

fs.unlink('test.txt', function () {

    console.log('write operation complete.');

});
```

📖 6.1.6

Important method of fs module

fs.readFile(fileName [,options], callback) Reads existing file.

fs.writeFile(filename, data[, options], callback) Writes to the file. If file exists then overwrite the content otherwise creates new file.

fs.open(path, flags[, mode], callback) Opens file for reading or writing.

fs.rename(oldPath, newPath, callback) Renames an existing file.

fs.chown(path, uid, gid, callback) Asynchronous chown.

fs.stat(path, callback) Returns fs.stat object which includes important file statistics.

fs.link(srcpath, dstpath, callback) Links file asynchronously.

fs.symlink(destination, path[, type], callback) Symlink asynchronously.

fs.rmdir(path, callback) Renames an existing directory.

fs.mkdir(path[, mode], callback) Creates a new directory.

fs.readdir(path, callback) Reads the content of the specified directory.

fs.utimes(path, atime, mtime, callback) Changes the timestamp of the file.

fs.exists(path, callback) Determines whether the specified file exists or not.

fs.access(path[, mode], callback) Tests a user's permissions for the specified file.

fs.appendFile(file, data[, options], callback) Appends new content to the existing file.

6.2 File System

6.2.1

Renames an existing directory.

- fs.stat(path, callback)
- fs.rmdir(path, callback)
- fs.exists(path, callback)

6.2.2

Tests a user's permissions for the specified file.

- fs.access(path[, mode], callback)
- fs.appendFile(file, data[, options], callback)
- fs.chown(path, uid, gid, callback)

6.2.3

Open file for reading and writing. The file is created (if it does not exist) or truncated (if it exists).

6.2.4

Open file for reading in synchronous mode.

6.2.5

Like 'a+' but fails if path exists.

Laravel

Setting up the Development Environment and Relationships Between Applications

Chapter 7

7.1 Introduction

📖 7.1.1

There are multiple ways how to create a local development environment when you want to start developing PHP application with the Laravel framework. The fastest one is with applications that compile a set of necessary applications like WampServer or XAMPP. WAMP, also known as WAMPserver is a free localhost server stack comprising Apache, MySQL and PHP for Windows. It is ideal for learning, testing and developing websites without having to use a remote web server. The main benefit of this approach is rapid local env setup and easy configurable local web server. On the other hand, it is quite simple and not suited for a large project for multiple reasons.

Another approach is the usage of virtual machines (VM). This approach is much more complex and requires multiple additional software like VirtualBox or VMware. There are also helper applications, like Vagrant, which help create and manage these virtual operating systems, representing the Virtual Private Server (VPS) on our local dev env. The main advantage of this approach is that at larger developers teams share the same dev env so it will not happen that one member of your team will develop some functionality with PHP extension that is no longer supported with the PHP version which is used by you. The problem with this approach is performance (because basically, you are hosting an independent operating system for your application) and configuration issues (matrix of hell).

One of the best possible ways how to develop PHP application under your local env is using Docker. This approach is not so performance extensive as Vagrant and it is also helpful regarding compatibility issues that might occur in large teams. For simplicity, we will use the WampServer approach because this course's main goal is to learn Laravel, no other technologies that help improve your application quality.

7.1.2

What is Vagrant for?

- Manage VPS for local dev
- Tool for WampServer setup
- Tool for Docker installation

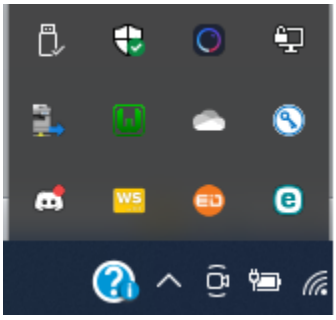
📖 7.1.3

Installing WampServer is quite easy. Just go to this link(https://www.wampserver.com/en/), and download the correct application for

you. If you do not have the operating system Windows, you can go to Xampp page(https://www.apachefriends.org/index.html) and download the correct version for your OS. For the sake of this tutorial, we will assume that we have OS Windows. After successful installation of WAMP stack, we can check if everything is correctly installed. By „everything“ we mean 5 applications:

1. Wamp Manager – an application that covers all other applications under one management tool
2. Apache Web Server – a web server for PHP applications
3. MySQL database – SQL database application that stores your application data
4. MariaDB database - SQL database application that stores your application data
5. PHP – an application that interprets your PHP application to your web server

First of all, we need to check if Wamp Manager is running correctly. What we should see is a small green icon on the bottom right section of our screen



7.1.4

What applications are installed together with WampServer?

- Apache, MySQL, MariaDB, PHP
- PHP, Java, Ruby, MySQL
- MariaDB, MS SQL, Tomcat, Apache

7.1.5

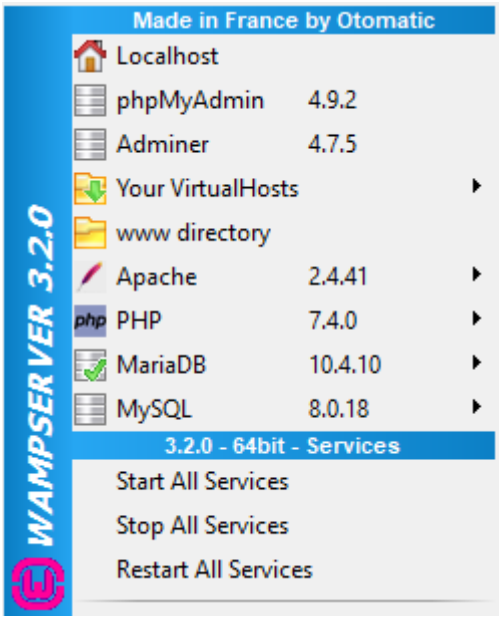
If our Wamp Manager is correctly running that means most of our applications should be also correctly installed. For example, we can check what version of PHP we have currently set as default by opening command line and typing the following command:

```
php -v
```

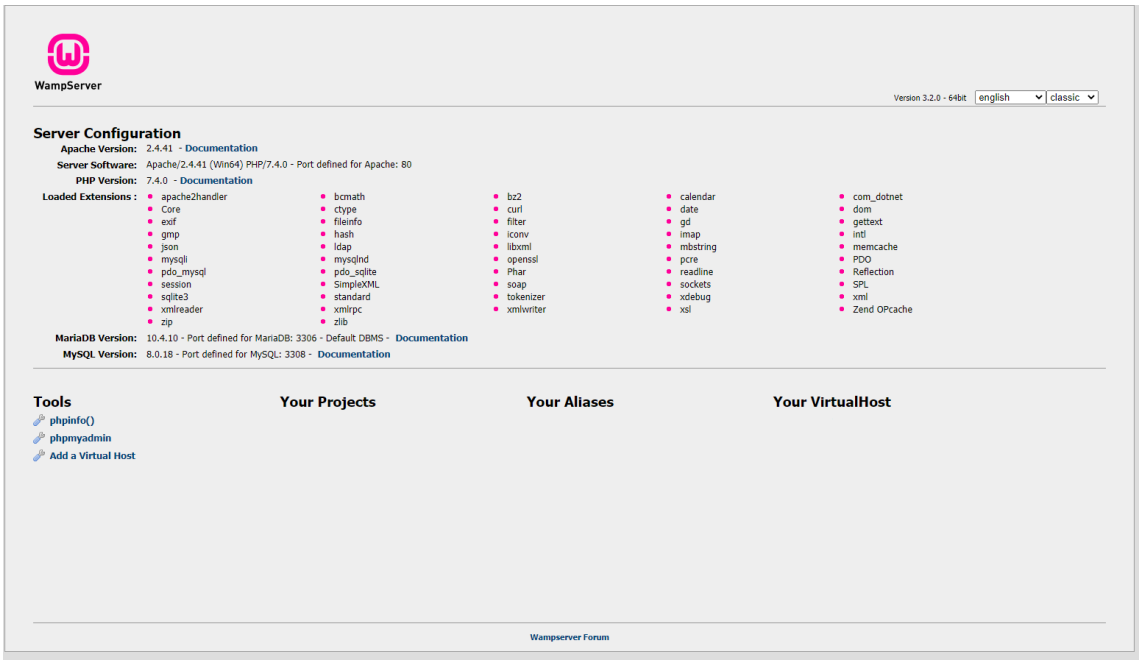
Something similar should come out as output:

```
PHP 7.4.0 (cli) (built: Nov 27 2019 10:14:18) ( ZTS Visual C++
2017 x64 )
Copyright (c) The PHP Group
Zend Engine v3.4.0, Copyright (c) Zend Technologies
```

You can check if the version in your command line is the same as the one attached to your web server. You see all of this information by left clicking on the WAMP icon.



If that is true, then we can open our web browser and type the following URL: http://localhost . After that, you should see the Wamp welcome page.



7.1.6

What is the command for getting information of PHP version from command line interface?

- php -i
- php –v
- php –m

7.1.7

This means that every application that is part of WAMP stack is correctly configured and is up and running. That means that we can now install the last key software for PHP development, which is Composer. Based on the documentation: **“Composer is a tool for dependency management in PHP. It allows you to declare the libraries your project depends on and it will manage (install/update) them for you.”**

This tool is basically responsible for downloading and loading external libraries/packages and even whole applications into your project. Simply follow the instructions on the Composer(https://getcomposer.org/download/) page and install Composer. After you will install Composer check if it is correctly installed by typing the following command to your command line:

```
composer --version
```

Something similar should come out as output:

```
Composer version 2.0.2 2020-10-25 23:03:59
```

If you can see the version of your composer we can now start with the development of our first Laravel application.

7.1.8

What is Composer for?

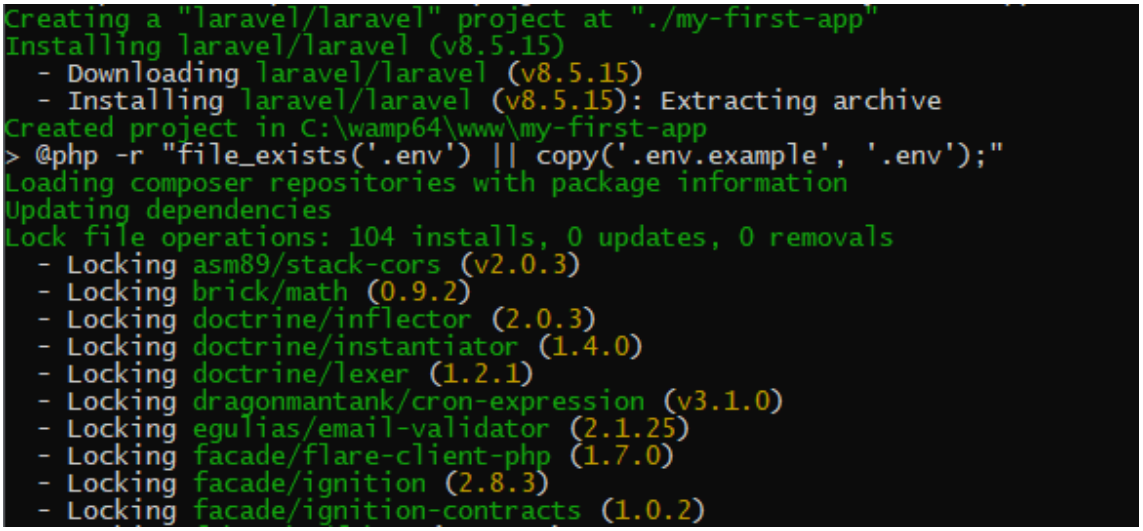
- Composer is a tool for dependency management in PHP
- Composer is a Laravel helper for installation
- Composer is platform independent installing wizard

🕮 7.1.9

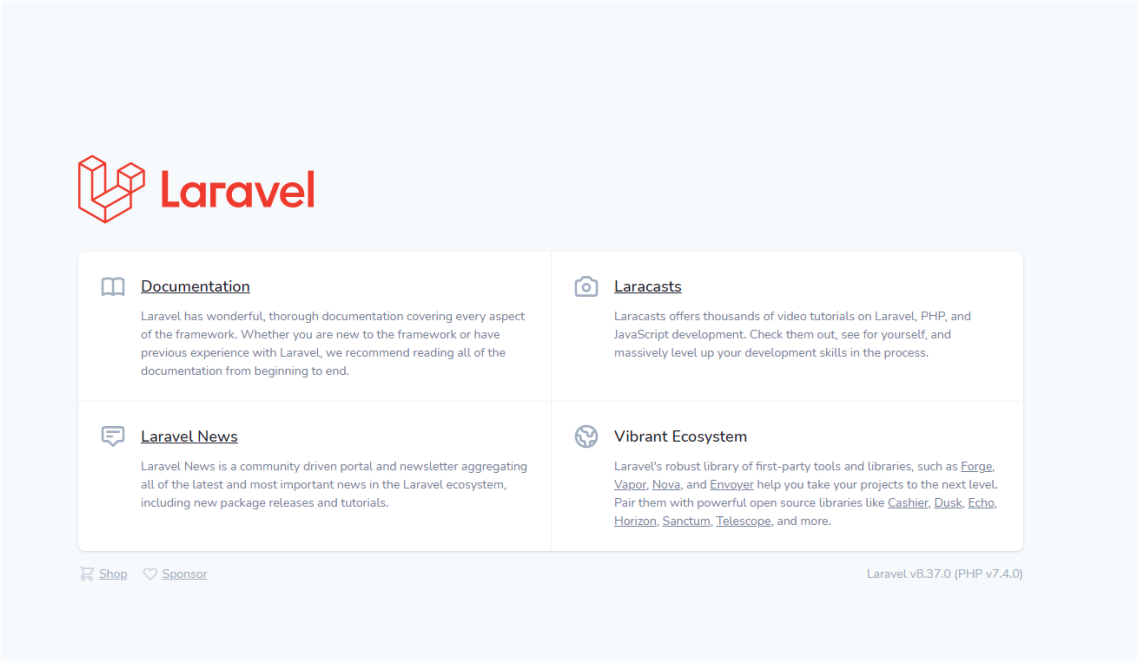
First of all, open your command-line interface and navigate yourself to www folder of Wamp server. This is usually somewhere like C:\wamp64\www. Apache as a web server application is configured to look into this folder and server content presented in this main folder. What we will do now is to install our fresh Laravel application by typing the following command:

```
composer create-project laravel/laravel my-first-app
```

After clicking the enter button you should see multiple download/install operations running in the command line.



This means that Composer is downloading each mandatory package defined in Laravel composer.json file and also in each downloaded package composer.json file. After this operation is finished we can go to our web browser and type http://localhost/my-first-app/ . What we will see is a list of files and folder but no Laravel application output. Why? Because Laravel is pre-configured to look into /public directory, where we can found index.php which is the entry point to our application. So when we will type http://localhost/my-first-app/public , we should see the Laravel welcome screen which in version 8.* looks like this.



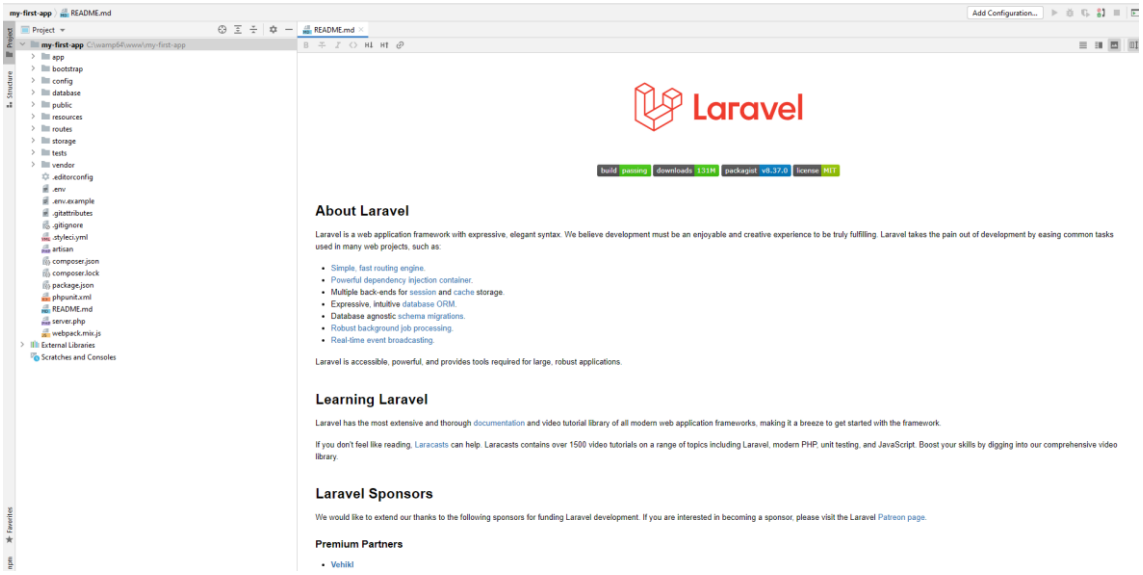
7.1.10

What is the main entry point to Laravel application?

- public/index.html
- public/index.php
- public/robots.txt

7.1.11

Great! Now we have our Laravel application prepared for development. But first, we need to be able to edit the code and also understand which files belong to the correct folders. So you need to download something called IDE (Integrated Development Environment). There are plenty of options like Visual Studio Code, Netbeans, WebStrom etc. We prefer PHPStorm because we are developing mainly PHP application and PHPStorm is an IDE that offers all the functionality that we need at the moment. So open C:\wamp64\www\my-fist-app at PHPStorm. What you should is something like this



By default, PHPStorm will open the file called README.md which contains a basic introduction to Laravel documentation. We can close this file. On the left side, you should see a list of folders. Each of these folders contains important Laravel files. However each folder contains different, we can say specific, files. Each folder is important but at this moment the only folders that we are interested in are:

- app
- routes
- public

As we said public folder is the place that is publically available for users to see. So all images, CSS or JavaScript files need to be stored here. Routes folder contains a list of all available application routes. This means, that whatever we have behind http://localhost/my-first-app/public/ need to be defined in these files. The app folder contains all “application” files. This means that each Controller, Model, Middleware need to be placed over here. We will cover these files later. Now let’s finally code!

7.1.12

Which folder contains all files defining all application routes?

- routes
- public
- app

🕮 7.1.13

Open file routes/web.php. This file should has similar content:

```
use Illuminate\Support\Facades\Route;

/*
|--------------------------------------------------------------------------
| Web Routes
|--------------------------------------------------------------------------
|
| Here is where you can register web routes for your application. These
| routes are loaded by the RouteServiceProvider within a group which
| contains the "web" middleware group. Now create something great!
|
*/

Route::get('/', function () {
    return view('welcome');
});
```

This file currently handles the main route of our application, so you can see that the Laravel welcome screen is just a view called welcome which is printed after we access our application. So you can see that the route definition is a minister by an object called Route. This object has a static method called “get” which contains 2 main arguments. First is the shape of URI. In this place, we define how the URI should look like (/test, /home etc.). The second argument is the callback function and defines the behaviour directly in the route definition. We will see how we can configure this better later in the course. Let’s comment on line 17 and under that line lets type:

```
echo "Hello is anybody home?";
```

So now, the route definition should looks like this:

```
Route::get('/', function () {
    //return view('welcome');
    echo "Hello is anybody home?";
});
```

Please save the file and now when you go to the web browser and refresh the page http://localhost/my-first-app/public/ you should see our message instead of Laravel welcome screen.

7.1.14

What is the purpose of the word get in the following code: **Route::get**

- Representation of HTTP method used in Laravel Router
- Method used in Laravel Router for getting list of routes
- Definition of URL which will contains the word get

Controllers and Routing

Chapter 8

8.1 Basic connection

🕮 8.1.1

All Laravel routes are defined in your route files, which are located in the routes directory. These files are automatically loaded by your application's App\Providers\RouteServiceProvider. The routes/web.php file defines routes that are for your web interface. These routes are assigned the web middleware group, which provides features like session state and CSRF protection. The routes in routes/api.php are stateless and are assigned the api middleware group.

For most applications, you will begin by defining routes in your routes/web.php file. The routes defined in routes/web.php may be accessed by entering the defined route's URL in your browser. For example, you may access the following route by navigating to http://localhost/my-first-app/public/user in your browser:

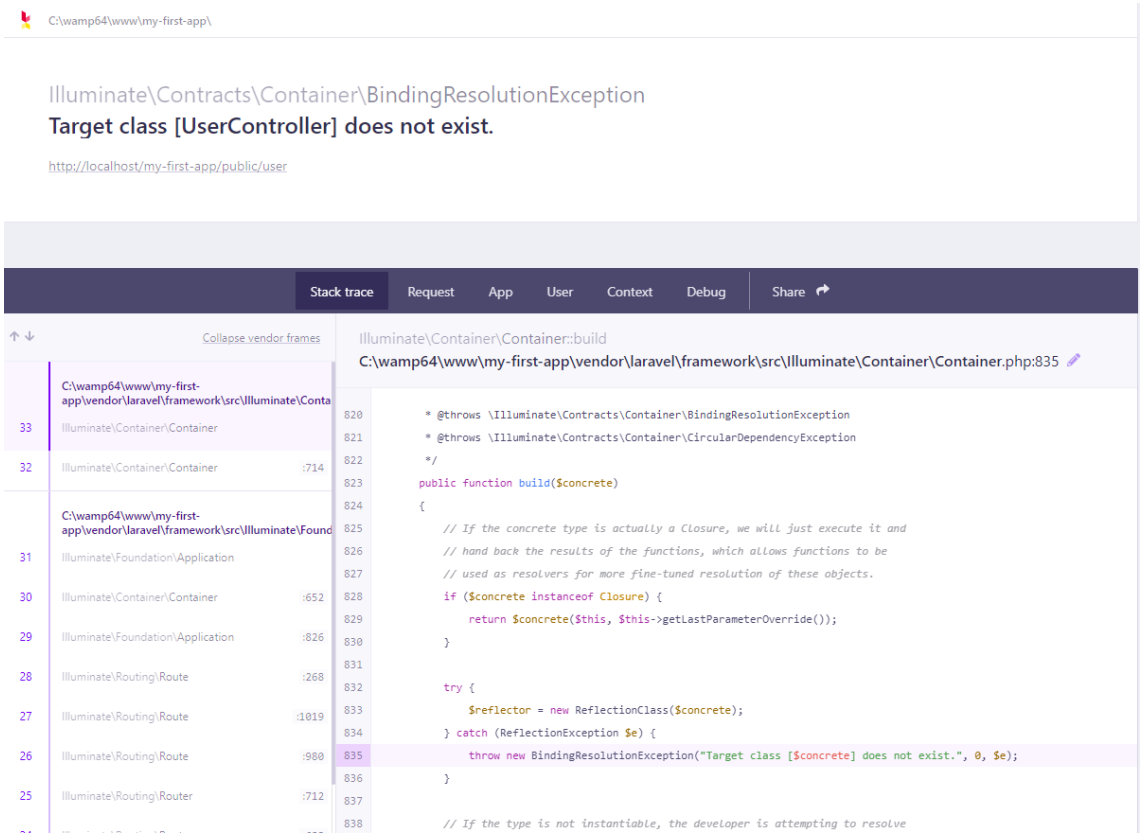
```
use Illuminate\Support\Facades\Route;

/*
|--------------------------------------------------------------------------
| Web Routes
|--------------------------------------------------------------------------
|
| Here is where you can register web routes for your application. These
| routes are loaded by the RouteServiceProvider within a group which
| contains the "web" middleware group. Now create something great!
|
*/

Route::get('/', function () {
    //return view('welcome');
    echo "Hello is anybody home?";
});

Route::get('/user/{id?}', [UserController::class, 'index']);
```

The problem is that when we try to enter this URL we will get this, instead of what we originally wanted:



The reason why this happened is that we defined a route that is assigned to a specific controller and method in that controller. This is a correct approach but the problem is that we did not create any controller with that specific method. So now we need to do exactly that. We need to create a new file with the name UserController in the folder app/Http/Controllers. This file will contain only one class with the same name - UserController. This class need to extend functionality from the main Laravel Controller class - Illuminate\Routing\Controller.

```
<?php


namespace App\Http\Controllers;

use Illuminate\Routing\Controller;

class UserController extends Controller
{
    public function index()
    {
        echo "My first controller";
    }
}
```

At this moment the functionality on UserController is defined correctly. The last thing that we need to do, is to say in our routers/web.php file that we want to use that specific UserController in our application. That need to be done by adding this line on the top of the file:

```
use App\Http\Controllers\UserController;
```

So the file will look like this

```
use Illuminate\Support\Facades\Route;
use App\Http\Controllers\UserController;

/*
|--------------------------------------------------------------------------
| Web Routes
|--------------------------------------------------------------------------
|
| Here is where you can register web routes for your application. These
| routes are loaded by the RouteServiceProvider within a group which
| contains the "web" middleware group. Now create something great!
|
*/

Route::get('/', function () {
    //return view('welcome');
    echo "Hello is anybody home?";
});

Route::get('/user/{id?}', [UserController::class, 'index']);
```

After saving all files and refreshing the page the output should be: My first controller. For the sake of completeness, we would like to mention that routes defined in the routes/api.php file are nested within a route group by the RouteServiceProvider. Within this group, the /api URI prefix is automatically applied so you do not need to manually apply it to every route in the file. You may modify the prefix and other route group options by modifying your RouteServiceProvider class.

8.1.2

What is the main Laravel controller class.

- Illuminate\Routing\Controller

- Illuminate\Core\Controller
- Illuminate\Controllers\Controller

🕮 8.1.3

In the previous lesson, we defined functionality that was specifically binded to a specific method in a specific controller. This however happens only when you access the defined URL via HTTP GET method. When you will try to send POST or DELETE request to that URL you will receive error. Laravel supports these explicit HTTP methods:

```
Route::get($uri, $callback);
Route::post($uri, $callback);
Route::put($uri, $callback);
Route::patch($uri, $callback);
Route::delete($uri, $callback);
Route::options($uri, $callback);
```

Sometimes you may need to register a route that responds to multiple HTTP verbs. You may do so using the match method. Or, you may even register a route that responds to all HTTP verbs using any method:

```
Route::match(['get', 'post'], '/post-get-methods',
[UserController::class, 'save']);

Route::any('/anything-you-want', [UserController::class,
'anything']);
```

Also do not forget to define the requested methods in your controller

```
<?php


namespace App\Http\Controllers;

use Illuminate\Routing\Controller;

class UserController extends Controller
{
    public function index()
    {
        echo "My first controller";
    }

    public function save()
    {
        echo "Data has been saved";
    }
```

```
    public function anything()
    {
        echo "You can send anything here";
    }
}
```

Now when you enter http://localhost/my-first-app/public/anything-you-want using any HTTP method or http://localhost/my-first-app/public/post-get-methods using either POST or GET method you should see the correct output.

8.1.4

What are the available HTTP methods that Laravel Router supports?

- match,any,get,post,patch
- get,post,put,patch,delete,options
- match,any,redirect,view

8.1.5

Sometimes you will need to capture segments of the URI within your route. For example, you may need to capture a user's ID from the URL. You may do so by defining route parameters. Let's re-build our /user URL which is binded to index method. Let's say, that we want to show the only user with a specific ID. We need to define that ID into the URL. So now we will change the current route definition like this:

```
Route::get('/user/{id}', [UserController::class, 'index']);
```

Now when we enter URL http://localhost/my-first-app/public/user/1 we will see the same output as before. Great! It is working. On the other hand, it is not doing what we were expecting. The reason is that we did not modify our index method to accept {id} argument from URL. So now, we will. We need to simply add id as an argument variable to the defined method.

```
public function index($id)
{
    echo "My first controller for user with ID ".$id;
}
```

After this modification, you should be able to see the different output after changing the last part of URL: http://localhost/my-first-app/public/user/1 or http://localhost/my-first-app/public/user/158 .

Now let's say that we want slightly different behaviour. What we want now, is to define ID as an optional argument instead of mandatory and when we will get a specific ID we will show different output that when ID will not be present. So now, we need to change our route definition as follow:

```
Route::get('/user/{id?}', [UserController::class, 'index']);
```

Now Laravel will know, that the URL can have different formats like http://localhost/my-first-app/public/user/158 or http://localhost/my-first-app/public/user and it is still the same. Based on our requirements the index method can be changed like this

```
public function index($id = null)
{
    if(empty($id)) {
        echo "Show all users in this system";
    } else {
        echo "My first controller for user with ID ".$id;
    }
}
```

At this specific moment, things are looking great, and we can be happy that we did everything correctly. Or did we? In most systems ID of anything is a number, usually integer. But what will happen in our Laravel application when we will add this to our web browser: http://localhost/my-first-app/public/user/hacking. What we missed is validated our URL argument to a specific data format based on regular expression constraints. Laravel offers us a very easy way how to fix this issue. The only thing that we need to change is the route definition:

```
Route::get('/user/{id?}', [UserController::class, 'index'])->where('id', '[0-9]+');
```

Now when someone will try to add ID argument in incorrect data format(different than integer) Laravel will simply respond with HTTP 404 - Page not Found response.

8.1.6

What is the correct definition of the route with mandatory GET argument in URL?

- Route::get('/user/{id?}', [UserController::class, 'index'])
- Route::get('/user/{id}', [UserController::class, 'index'])
- Route::get('/user/id', [UserController::class, 'index'])

Working with Database

Chapter 9

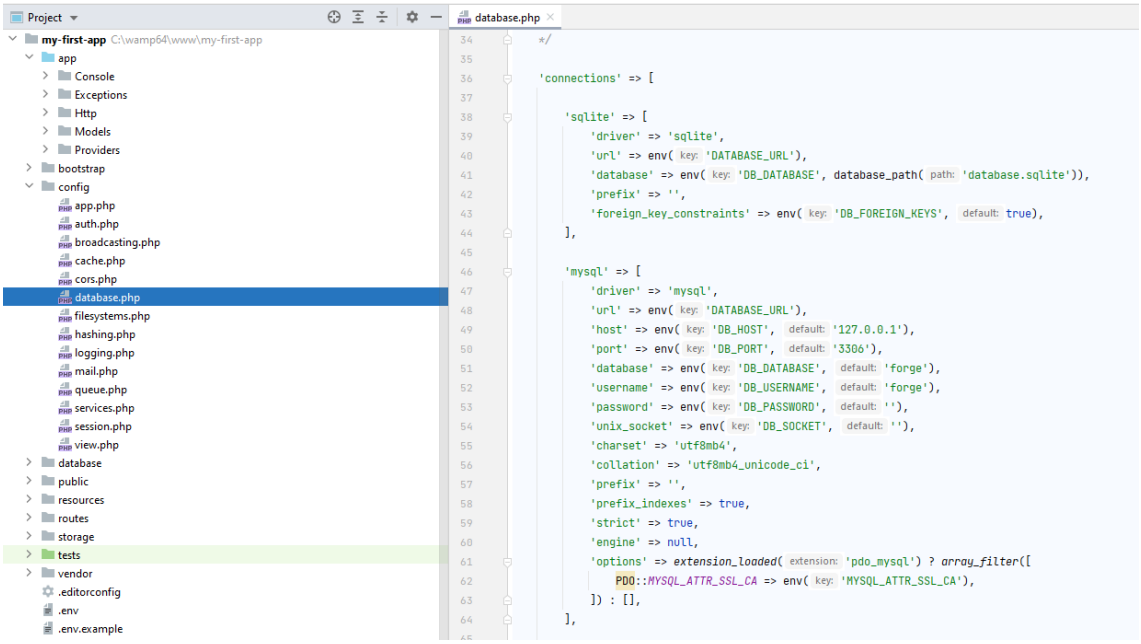
9.1 Setting up a connection and creating a schema

🕮 9.1.1

Almost every modern web application interacts with a database. Laravel makes interacting with databases extremely simple across a variety of supported databases using raw SQL, a fluent query builder, and the Eloquent ORM. Currently, Laravel provides first-party support for four databases:

- MySQL 5.7+
- PostgreSQL 9.6+
- SQLite 3.8.8+
- SQL Server 2017+

The configuration for Laravel's database services is located in your application's config/database.php configuration file. In this file, you may define all of your database connections and specify which connection should be used by default. Most of the configuration options within this file are driven by the values of your application's environment variables. Examples for most of Laravel's supported database systems are provided in this file.



As you can see many configuration options are not hardcoded but function env() is used. What this function does is that it looks at the file called .env placed in the root folder of your application and extract the values based on defined keys which is the first argument of the function. The second argument is the default value which should be used if the key is not presented in the .env file. When you open .env file you will see a list of environment variables defined for your local development. You can have multiple .env files with different names like .env.production or .env.testing

and each of these files will be renamed to .env after the application will be deployed to a specific environment. Configuration options that are related to the database starting with the prefix DB_ . You need to modify these options to those which suits you (name of your database, the password for your database user etc.).

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=my-first-app
DB_USERNAME=app-user
DB_PASSWORD=1f6sd5fsd561f6sda16asf5
```

After this modification, we are ready to use Laravel with MySQL database. So the first thing that we need to do is to create at least 1 database table. Laravel offers us an easy tool for this called Migrations. Migrations are PHP classes that represent the structure of a defined SQL table. Laravel has a console interface called artisan which can be used for generating multiple files like controllers, models and also migrations. So now we will create new migration which will store our posts. You need to open any command-line application like cmd.exe on Windows, navigate to root folder of your application and type:

```
php artisan make:migration create_post_table
```

Now Laravel will automatically create a new file with the unique name under folder database/migrations. In our case, the file is called 2021_06_17_064334_create_post_table.php. For sake of this tutorial, we can delete all the rest of the migrations. When we do that we also need to delete the following file: app/Models/User.php. Now we have clear Laravel installation regards to database configuration. So what we want, is to define how our post table will look like. In this example, we will add only 2 additional columns: title and content. So the file should look like this.

```
 <?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class CreatePostTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('post', function (Blueprint $table) {
            $table->id();
            $table->string('title',255);
```

```
            $table->text('content');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('post');
    }
}
```

As you can see our migration has 2 methods:

1. up() - responsible for creating the database table
2. down() - responsible for removing it

Now what we want is to apply this migration so we will have the database table created in our database. It is very easy, just type to your command-line following command:

```
php artisan migrate
```

When you open your database, you will see 2 tables:

- post - table created by our migration
- migrations - system table which Laravel use for storing all applied migration. So when you will call the command 2x and the table is already there, it will not show any error(table with that name already exists) because Laravel already knows that that migrations have been applied.

Now we have our database structure ready.

9.1.2

What is the configuration option in .env file that specifies the database name?

- DB_DATABASE
- DB_NAME
- DB_DATABASE_NAME

🕮 9.1.3

Laravel includes the ability to seed your database with test data using seed classes. All seed classes are stored in the database/seeders directory. By default, a DatabaseSeeder class is defined for you. From this class, you may use the call method to run other seed classes, allowing you to control the seeding order. To generate a seeder, execute the make:seeder Artisan command. All seeders generated by the framework will be placed in the database/seeders directory. Let's say we want to create some posts in our database this way. First, we need to create a seeder itself:

```
php artisan make:seeder PostSeeder
```

What Laravel does is that it creates the seeder file and prepared the default structure. So now we can go to database/seeders and we can open the file called PostSeeder. Because we do not have any model at the moment we will use DB facade for data manipulation. So we will modify our seeder that it will add 1 record to our post table.

```
namespace Database\Seeders;

use Illuminate\Database\Seeder;
use Illuminate\Support\Facades\DB;

class PostSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        DB::table('post')->insert([
            'title' => 'New post',
            'content' => 'This is the content of our post',
            'created_at' => now(),
            'updated_at' => now()
        ]);
    }
}
```

As you can see we called method table() from DB faced which contains multiple addition methods. In our case, we used method insert() responsible for inserting data stored as an array to table based on the key/value logic. Now we need to register our seeder, so when we will call the command for seed Laravel will know that this specific seeder should be included. Open database/seeders/DatabaseSeeder.php and add call() method to its run method.

```
namespace Database\Seeders;

use Illuminate\Database\Seeder;

class DatabaseSeeder extends Seeder
{
    /**
     * Seed the application's database.
     *
     * @return void
     */
    public function run()
    {
        $this->call([
            PostSeeder::class
        ]);
    }
}
```

Now we can test if all is working correctly and seed our database by following this command in our command line:

```
php artisan db:seed
```

When you open your database table called post you should see a new record in it. Keep in mind that seeders do not work the same way as migrations. Laravel does not store any information on which seeders have been applied so when you run this command again, you will have 2 records in your database.

9.1.4

What is the main class responsible for containing a list of seeders that need to be applied?

- PostSeeder
- DatabaseSeeder
- DatabaseSeederManager

9.1.5

Laravel includes Eloquent, an object-relational mapper (ORM) that makes it enjoyable to interact with your database. When using Eloquent, each database table has a corresponding "Model" that is used to interact with that table. In addition to retrieving records from the database table, Eloquent models allow you to insert,

update, and delete records from the table as well. But first, we need to create one. The easiest way how to do it is via command-line call:

```
php artisan make:model Post
```

What we will get is a clear Post model without any additional properties or methods. The Model will be created in app/Models folder under the name Post.php. After glancing at the example above, you may have noticed that we did not tell Eloquent which database table corresponds to our Post model. By convention, the "snake case", plural name of the class will be used as the table name unless another name is explicitly specified. So, in this case, Eloquent will assume the Post model stores records in the posts table. If your model's corresponding database table does not fit this convention, you may manually specify the model's table name by defining a table property on the model which is exactly our case. So we will define $table attribute with value post. There are plenty of configuration attributes in Eloquent models but for sake of this tutorial, we will use only this one. So now our model should look like this:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    use HasFactory;

    /**
     * The table associated with the model.
     *
     * @var string
     */
    protected $table = 'post';
}
```

At this stage, our model is ready for use.

9.1.6

What is the Artisan command that creates a model?

- php artisan make:model MODEL_NAME
- php artisan create:model MODEL_NAME
- php artisan add:model MODEL_NAME

🕮 9.1.7

It is very rare that the database of any web application contains only 1 table. From the point when databases contain multiple tables, we can assume that there are relations between these tables. For example, in our database, there is a post table. In most newspaper sites you have the ability to comment on any article presented on the page. To add this ability to our project, we need to do the following:

1. Create comment table
2. Add foreign key to post table so we can create the relation between post and comment
3. Create Comment model
4. Create a relation between Comment and Post model

As we did when we created the post table we will call the command responsible for creating new migration:

```
php artisan make:migration create_comment_table
```

Now we need to modify the migration to contain all attributes that we want.

```
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class CreateCommentTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('comment', function (Blueprint $table)
{
            $table->id();
            $table->string('content', 255);
            $table->unsignedInteger('post_id')-
>index('fk_post_idx');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
```

```
    {
        Schema::dropIfExists('comment');
    }
}
```

As you can see the table will contains 2 main attributes:

- content - attribute that will contain the comment it self
- post_id - id of post which is related to that comment

Now we need to create second migration which will add the foreine key on MySQL level. So again we will call the command but now it will not be for creating a new table but altering existing one.

```
php artisan make:migration alter_comment_table
```

At the moment we have a clear migration without any predefined functionality. What we need to do is to add the foreign key definition i it so we will use Schema::table instead of Schema::create function.

```
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class AlterCommentTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::table('comment', function (Blueprint $table) {
            $table->foreign('post_id', 'fk_post_idx')-
>references('id')->on('post');
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::table('comment', function (Blueprint $table) {
            $table->dropForeign('fk_post_idx');
        });
    }
```

```
}
```

At this stage we have the schema prepared so we can apply our migrations. As we know, we can do it by running the following command:

```
php artisan migrate
```

Because we want to use Eloquent ORM we need to define second model which will represent the comment table. What we will do is to execute the set of same steps as we done with Post model. Only difference will be that additionaly we will add one method which will represent the relation between Post and Comment. In this case it is One-to-Many relation. A one-to-many relationship is used to define relationships where a single model is a parent to one or more child models. For example, a blog post may have an infinite number of comments. Like all other Eloquent relationships, one-to-many relationships are defined by defining a method on your Eloquent model:

```
namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    use HasFactory;

    /**
     * The table associated with the model.
     *
     * @var string
     */
    protected $table = 'post';

    /**
     * Get the comments for the blog post.
     */
    public function comments()
    {
        return $this->hasMany(Comment::class);
    }
}
```

Remember, Eloquent will automatically determine the proper foreign key column for the Comment model. By convention, Eloquent will take the "snake case" name of the parent model and suffix it with _id. So, in this example, Eloquent will assume the foreign key column on the Comment model is post_id. Once the relationship method has been defined, we can access the collection of related comments by accessing the comments property. Remember, since Eloquent provides "dynamic relationship properties", we can access relationship methods as if they were defined as properties on the model.

Now that we can access all of a post's comments, let's define a relationship to allow a comment to access its parent post. To define the inverse of a hasMany relationship, define a relationship method on the child model which calls the belongsTo method:

```
namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class Comment extends Model
{
    use HasFactory;

    /**
     * The table associated with the model.
     *
     * @var string
     */
    protected $table = 'comment';

    /**
     * Get the post that owns the comment.
     */
    public function post()
    {
        return $this->belongsTo(Post::class);
    }
}
```

Once the relationship has been defined, we can retrieve a comment's parent post by accessing the post "dynamic relationship property".

9.1.8

What is the correct function for altering table in migrations?

- Schema::create()
- Schema::table()
- Schema::alter()

9.1.9

In this stage, our application can store data and has the ability to present them via controllers. Let connect our controllers and models so we can see how we can use

our models in our controllers. Go to your routes/web.php file and define a new route that will return data for a specific post:

```
Route::get('/post/{id}', [PostController::class, 'index'])->where('id', '[0-9]+');
```

Do not forget to create a new controller with that specific name PostController the same way how we created UserController. Now when we have that controller created we need to add method index($id) which will show all the data for a specific post based on its ID.

```
namespace App\Http\Controllers;

use App\Models\Post;
use Illuminate\Routing\Controller;

class PostController extends Controller
{
    public function index($id)
    {
        $post = Post::find($id);
        echo "Post data: <br>";
        echo $post->id . "<br>";
        echo $post->title . "<br>";
        echo $post->content . "<br>";
        echo "<br><br>Comments: <br><br>";
        foreach ($post->comments as $comment) {
            echo $comment->id . "<br>";
            echo $comment->content . "<br>";
        }
    }
}
```

After you will access URL http://localhost/my-first-app/public/post/1 you should see the data related to post with ID 1 on your screen. When you will try to change the URL argument from 1 to 2 you will get the error: Trying to get property 'id' of non-object . The reason is that you are trying to get attribute id from the variable post but that variable is not an instance of Model but it is of type null. When you want to fix this you can replace the predefined method find($id) with predefined method findOrFail($id):

```
$post = Post::findOrFail($id);
```

Now when you will add ID which is not in the database you will get HTTP 404.

9.1.10

What is the predefined Eloquent Model method find() for?

- Search in related table for record with provided ID
- Search in related table for record with multiple provided criteria
- Search in selected collection for record with multiple provided criteria

Used View Files and Blade Templates

Chapter 10

10.1 Creating forms

🕮 10.1.1

Blade is the simple, yet powerful templating engine that is included with Laravel. Unlike some PHP templating engines, Blade does not restrict you from using plain PHP code in your templates. In fact, all Blade templates are compiled into plain PHP code and cached until they are modified, meaning Blade adds essentially zero overhead to your application. Blade template files use the .blade.php file extension and are typically stored in the resources/views directory.

So the first step that we need to do is to create a simple view file in Blade engine that will show a very simple form used for inserting posts in our application. So we will create a new file named insert_post.blade.php in the correct folder which is resources/views. In this file, we will define a form that will send the provided data to the method responsible for processing these input data.

```
<form action="{{ route('insert') }}" method="post">
    Title: <br>
    <input type="text" name="title" value="" placeholder="Post
title"><br>
    Content: <br>
    <textarea name="content" placeholder="Post
content"></textarea><br>
    <br><br>
    <input type="hidden" name="_token" value="{{ csrf_token()
}}">
    <input type="submit" name="submit" value="Submit">
</form>
```

We added HTML form definition, which tells us that the form will use HTTP POST method and the data will be processed in a method that has been binded to route with name insert. We also added 4 inputs:

- Text - post title
- Text area - post content
- Hidden - form CSRF token for security reasons
- Submit - submit button

Now we need to define 2 methods in our PostController. One which will show the form and one that will process the data.

```
/**
 * @return
\Illuminate\Contracts\Foundation\Application|\Illuminate\Contr
acts\View\Factory|\Illuminate\Contracts\View\View
 */
public function getAddPostForm()
{
```

```
    return view('insert_post');
}

/**
 * @param Request $request
 * @return \Illuminate\Http\Response
 */
public function insertPost(Request $request)
{
    $title = $request->input('title');
    $content = $request->input('content');

    $newPost = new Post();
    $newPost->title = $title;
    $newPost->content = $content;
    $newPost->save();

    return response()->view('insert_post');
}
```

First method with name getAddPostForm is quite easy. The only operation that this method do is to render the view file which in our case is of type Blade. Second method is more complex and it is responsible for obtaining the HTTP request. After that it will extract the 2 main input values from request body and it will create new record in post table with provided data. In the end it will render the same form again so you will not see the post that you saved but you will have the ability to save another post. Now we need to define new routes that will connect the router of our Laravel application and our method in controller.

```
Route::get('/post/add', [PostController::class,
'getAddPostForm'])->name('add');

Route::post('/post/insert', [PostController::class,
'insertPost'])->name('insert');
```

As you can see we added 2 new routes, one of type GET and one of type POST. In both of the we used new feature of Laravel routing which is called Route naming. Insted of using the full controller@method convetion or specific URL usage we can add a specific name to our route and than via the route() helper function we can get the full URL of that functionality. Now when everything is ready we can try to enter the form screen by entering the following URL to our browser http://localhost/my-first-app/public/post/insert

10.1.2

Which Laravel helper method we use when we want to print URL based on the named route?

- route()
- action()
- url()

🕮 10.1.3

Blade provides simple directives for working with PHP's loop structures. Again, each of these directives functions identically to their PHP counterparts:

```
@for ($i = 0; $i < 10; $i++)
  The current value is {{ $i }}
@endfor

@foreach ($users as $user)
  <p>This is user {{ $user->id }}</p>
@endforeach

@forelse ($users as $user)
  <li>{{ $user->name }}</li>
@empty
  <p>No users</p>
@endforelse

@while (true)
  <p>I'm looping forever.</p>
@endwhile
```

Let's say, that in our application we want to have the ability to see post and all comments that are stored in our system related to that specific post. We already have a method in PostController that provides this function. It is index(). The problem with this is that it is against MVC architecture. Because we want to produce the correct code we need to remove presentation logic from the controller and move it to view. So what we will do now is that we will rebuild the index method.

```
/**
 * @param $id
 * @return
\Illuminate\Contracts\Foundation\Application|\Illuminate\Contr
acts\View\Factory|\Illuminate\Contracts\View\View
 */
public function index($id)
{
    $post = Post::findOrFail($id);
    return view('post', ['post' => $post]);
}
```

The changes that we are quite big. We removed the printing and data presentation functionalities and replace it with simple data extraction and passing the data to a related view file. So instead of returning the only result of view() method, we added a second argument which is of type array. The provided array contains:

1. key - the name of the variable in view file
2. value - the variable value, can be any data type supported by PHP (string, object, array etc.)

Now we will create a new blade file with the name post.blade.php. This file will contain the presentation logic of our application.

```
<table>
    <tr>
        <td>
            Post title:   {{ $post->title }}
        </td>
        <td>
            Post content: {{ $post->content }}
        </td>
    </tr>
    <tr>
        @foreach ($post->comments as $comment)
            <td>
                {{ $comment->content }}
            </td>
        @endforeach
    </tr>
</table>
```

In post.blade.php file contains a simple table that presents the post data + it shows all the comments that are related to that post. Blade's {{ }} echo statements are automatically sent through PHP's htmlspecialchars function to prevent XSS attacks. As we can see the Blade loop syntax is a bit different than the one in PHP. The main difference is that instead of using { } to define where is the beginning and end of the specified loop we use words @foreach / @endforeach. The is the same for all loop types.

10.1.4

Which PHP loop is not integrated into Blade?

- while
- foreach
- do - while
- for

🕮 10.1.5

Now when we have our basic structure let's say, that we want to show different output based on the data provided by our controller. You may construct if statements using the @if, @elseif, @else, and @endif directives. These directives function identically to their PHP counterparts:

```
@if (count($records) === 1)
  I have one record!
@elseif (count($records) > 1)
  I have multiple records!
@else
  I don't have any records!
@endif
```

Our application does not support any functionality that can show all the post titles on one screen. So we will implement it now. First, as always, we need to define a method in our controller that will extract the data from the database.

```
/**
 * @return
\Illuminate\Contracts\Foundation\Application|\Illuminate\Contr
acts\View\Factory|\Illuminate\Contracts\View\View
 */
public function showAll()
{
    $posts = Post::all();
    return view('show_all_posts', ['posts' => $posts]);
}
```

Now we will add a route that will connect this functionality to our router.

```
Route::get('/post/show', [PostController::class, 'showAll'])-
>name('show');
```

And now we will create a blade template with name show_all_posts.blade.php which will based on provided data which different results.

```
@if (count($posts) === 1)
    There is only 1 post in database and that is: {{
$posts[0]->title }} <br>
@elseif (count($posts) > 1)
    @foreach($posts as $post)
        {{ $post->title }} <br>
        @empty($post->comments)
            This post does not have any comments
        @endempty
    @endforeach
@else
    There are no posts in database!
```

```
@endif
```

Now when you will enter URL http://localhost/my-first-app/public/post/show you will see the list of post that are stored in your database. Laravel out of the box is configured quite performance-friendly. So it might happen that while you were testing your application it will cache the configuration of your application(list of routes etc.) and you might get non-sence errors. In most cases you need to type this command to your command line, which will remove all cached and pre-compiled files:

```
php artisan optimize
```

10.1.6

What is the correct syntax for printing variables in Blade?

- {{ $variable }}
- {! $variable !}
- {{{ $variable }}}

Testing

Chapter 11

11.1 Introduction to testing

📖 11.1.1

Laravel is built with testing in mind. In fact, support for testing with PHPUnit is included out of the box and a phpunit.xml file is already set up for your application. The framework also ships with convenient helper methods that allow you to expressively test your applications.

By default, your application's tests directory contains two directories: Feature and Unit. Unit tests are tests that focus on a very small, isolated portion of your code. In fact, most unit tests probably focus on a single method. Tests within your "Unit" test directory do not boot your Laravel application and therefore are unable to access your application's database or other framework services.

Feature tests may test a larger portion of your code, including how several objects interact with each other or even a full HTTP request to a JSON endpoint. Generally, most of your tests should be feature tests. These types of tests provide the most confidence that your system as a whole is functioning as intended.

To run our tests we need to run the following command:

```
php artisan test
```

When running tests, Laravel will automatically set the configuration environment to testing because of the environment variables defined in the phpunit.xml file. Laravel also automatically configures the session and cache to the array driver while testing, meaning no session or cache data will be persisted while testing.

You are free to define other testing environment configuration values as necessary. The testing environment variables may be configured in your application's phpunit.xml file, but make sure to clear your configuration cache using the config:clear Artisan command before running your tests!

In addition, you may create a .env.testing file at the root of your project. This file will be used instead of the .env file when running PHPUnit tests or executing Artisan commands with the --env=testing option. So what we will do now, we will create a file with the name .env.testing in our root directory and we will copy the same content that we currently have in .env file.

Now we are ready for testing!

📝 11.1.2

What type of tests Laravel supports?

- Feature and Unit
- Feature and Integration
- Integration and Unit

🕮 11.1.3

To create a new test case, use the make:test Artisan command. By default, tests will be placed in the tests/Feature directory:

```
php artisan make:test PostTest
```

If you would like to create a test within the tests/Unit directory, you may use the --unit option when executing the make:test command. Once the test has been generated, you may define test methods as you normally would using PHPUnit. Now, delete both example test cases defined in Unit and Feature folders in tests/ folder. So at the moment, we should have only 1 test which is defined in file test/Feature/PostTest.php . What we will do we will add a testing scenario that will show us if when we will enter URL http://localhost/my-first-app/public/post/show the HTTP response will be 200.

```
namespace Tests\Feature;

use Tests\TestCase;

class PostTest extends TestCase
{
    /**
     * A basic feature test example.
     *
     * @return void
     */
    public function test_example()
    {
        $response = $this->get('/post/show');

        $response->assertStatus(200);
    }
}
```

Now when we run the command for test execution: **php artisan test** we should get something like this:

```
C:\wamp64\www\my-first-app>php artisan test
Warning: TTY mode is not supported on Windows platform.

  PASS  Tests\Feature\PostTest
  ✓ example

  Tests:  1 passed
  Time:   0.19s
```

This means that our test executed HTTP request on URL http://localhost/my-first-app/public/post/show and the HTTP response Code was 200 which is exactly what we're expecting. Now, let's try to "break" our test so we can see what is the output when the test failed. We can achieve this in multiple ways but the easiest way is to add an invalid URL. For example, instead of /post/show we can add /post/show/4564456 which should return 404 instead of 200.

```
C:\wamp64\www\my-first-app>php artisan test
Warning: TTY mode is not supported on Windows platform.

  FAIL  Tests\Feature\PostTest
  × example

  ---

  • Tests\Feature\PostTest > example
  Expected status code 200 but received 404.
  Failed asserting that 200 is identical to 404.

  at C:\wamp64\www\my-first-app\tests\Feature\PostTest.php:18
     14 |     public function test_example()
     15 |     {
     16 |         $response = $this->get('/post/show/4564456');
     17 |
  ➜  18 |         $response->assertStatus(200);
     19 |     }
     20 | }
     21 |

  1   C:\wamp64\www\my-first-app\vendor\phpunit\phpunit\phpunit:61
      PHPUnit\TextUI\Command::main()


  Tests:  1 failed
  Time:   0.21s
```

11.1.4

What is the purpose of method TestCase::assertStatus() ?

- Compare HTTP body of HTTP response with provided status
- Compare HTTP status of HTTP response with provided status
- Compare predefined application status with provided status

PRISCILLA

priscilla.fitped.eu