**Front and Back**

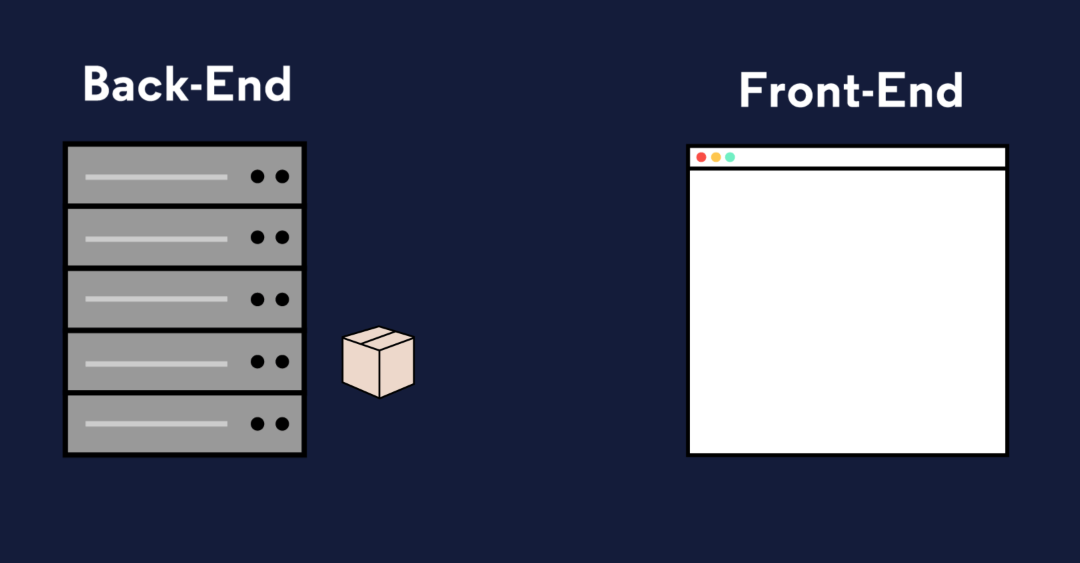
In this lesson, we’ll explain what makes up the back-end of a web application or website. The back-end can feel very abstract, but it becomes clearer when we explain it in terms of the front-end! To oversimplify a bit, the front-end is the parts of a webpage that a visitor can interact with and see.

Various tools and frameworks can be used to make a webpage, but, at its core, the front-end is composed of JavaScript, CSS, HTML, and other *static assets*, such as images or videos. Static assets are files that don’t change. When a visitor navigates to a webpage, these assets are sent to their browser.

Visiting a simple website is like ordering delivery from a restaurant: we place an order for our meal, and, once it’s delivered to us, we have it entirely in our possession. In this analogy, we can think of the front-end as everything that’s dropped off with the delivery: the containers, the utensils, and the food itself.

You’ll sometimes hear front-end development referred to as *client-side* development. Our instinct might be to think of the client as the human visitor or user of a website, but when referring to the client in web development, we’re usually referring to the non-human requester of content. In the case of visiting a website, the client is the browser, but in other circumstances, a client might be another application, a mobile device, or even a “smart” appliance!

While the front-end is the part of the website that makes it to the browser, the back-end consists of all the behind-the-scenes processes and data that make a website function and send resources to clients.



# So What is the Back-end?

When a user navigates to google.com, their request specifies the URL but not the filename for today’s [Google Doodle](https://en.wikipedia.org/wiki/Google_Doodle). The web application’s back-end will need to hold the logic for deciding which assets to send. Moreover, modern web applications often cater to the specific user rather than sending the same files to every visitor of a webpage. This is known as dynamic content.

When we eat at a restaurant, we’ll order to our tastes, make substitutions, etc; the result is a dining experience unique to us. Aside from that, there’s a lot happening behind the scenes to make a restaurant work: ingredients are ordered from suppliers, new menus are designed, and employees’ schedules are created. Similarly, to make a web application that runs smoothly, the back-end is doing a lot more than simply sending assets to browsers.

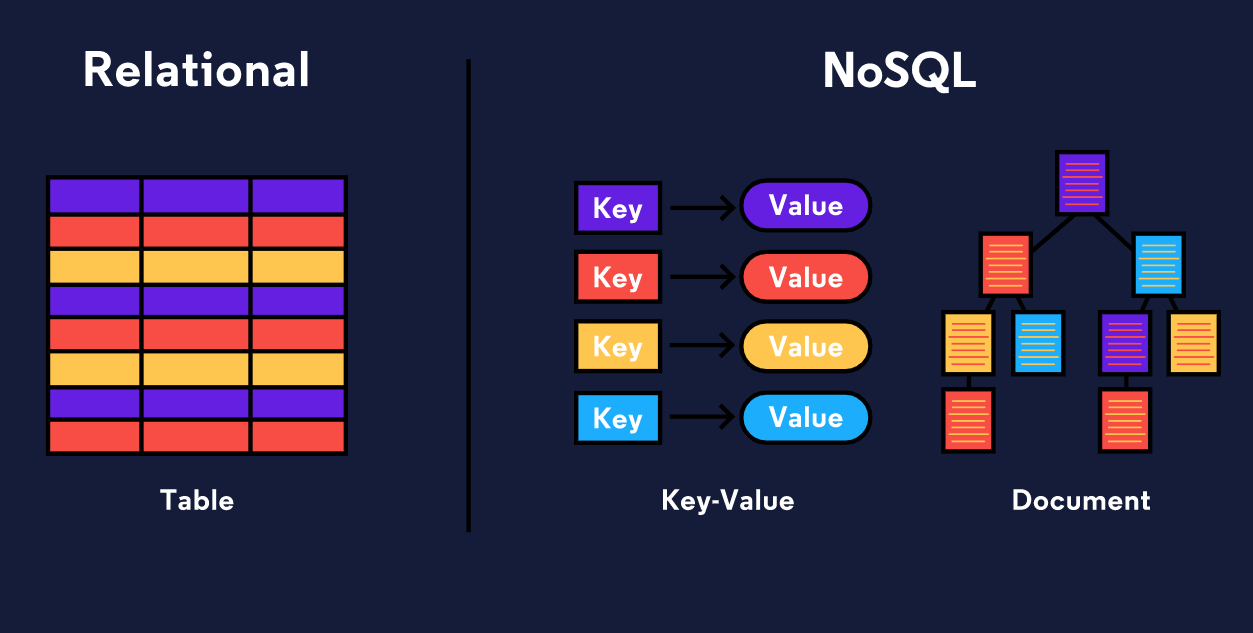
The collection of programming logic required to deliver dynamic content to a client, manage security, process payments, and myriad other tasks is sometimes known as the “application” or application server. The application server can be responsible for anything from sending an email confirmation after a purchase to running the complicated algorithms a search engine uses to give us meaningful results.

# Storing Data

You’ve probably heard that data is a big deal. By some measures, 90% of the world’s data has been generated in just the past two years! From a stored credit card number on an e-commerce site to the timestamp when you hit pause on Netflix, modern web applications collect a lot of data. For that data to be useful, it has to be organized and stored somewhere.

The back-ends of modern web applications include some sort of database, often more than one. Databases are collections of information. There are many different databases, but we can divide them into two types: [relational databases](https://www.codecademy.com/articles/what-is-rdbms-sql) and [non-relational databases (also known as NoSQL databases)](https://en.wikipedia.org/wiki/NoSQL). Whereas relational databases store information in tables with columns and rows, non-relational databases might use other systems such as key-value pairs or a document storage model. SQL, **S**tructured **Q**uery **L**anguage, is a programming language for accessing and changing data stored in relational databases. Popular relational databases include [MySQL](https://www.mysql.com/) and [PostgreSQL](https://www.postgresql.org/) while popular NoSQL databases include [MongoDB](https://www.mongodb.com/) and [Redis](https://redis.io/).

In addition to the database itself, the back-end needs a way to programmatically access, change, and analyze the data stored there.



# What is an API?

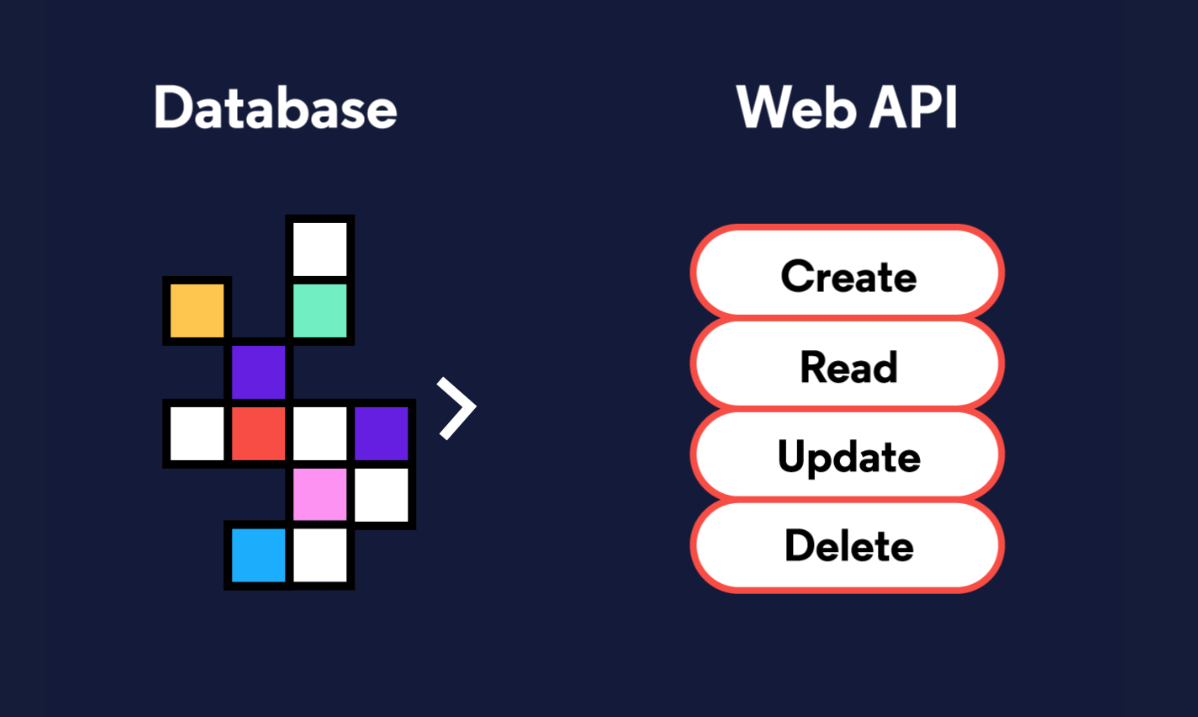
When a user navigates to a specific item for sale on an e-commerce site, the price listed for that item is stored in a database, and when they purchase it, the database will need to be updated with the correct inventory for that item type. In fact, much of what the back-end entails is reading, updating, or deleting information stored in a database.

In order to have consistent ways of interacting with data, a back-end will often include a web API. API stands for **A**pplication **P**rogram **I**nterface and can mean a lot of different things, but a web API is a collection of predefined ways of, or rules for, interacting with a web application’s data, often through an HTTP request-response cycle. Unlike the HTTP requests a client makes when a user navigates to a website’s URL, this type of request indicates how it would like to interact with a web application’s data (create new data, read existing data, update existing data, or delete existing data), and it receives some data back as a response.

Let’s walk through the example of making an online purchase again, but this time, we’ll imagine how the application’s web API might be used. When a user presses the button to submit an order, that will trigger a request to send them to a different view of the website, an order confirmation page, but an additional request will be triggered from the front-end, unseen by the user, to the web API so that the database can be updated with the information from the order.

Some web APIs are open to the public. [Instagram](https://www.instagram.com/developer/), for example, has an API that other developers can use to access some of the data Instagram stores. Others are only used by the web application internally— Codecademy, for example, has a web API that employees use to accomplish internal tasks.

## Image



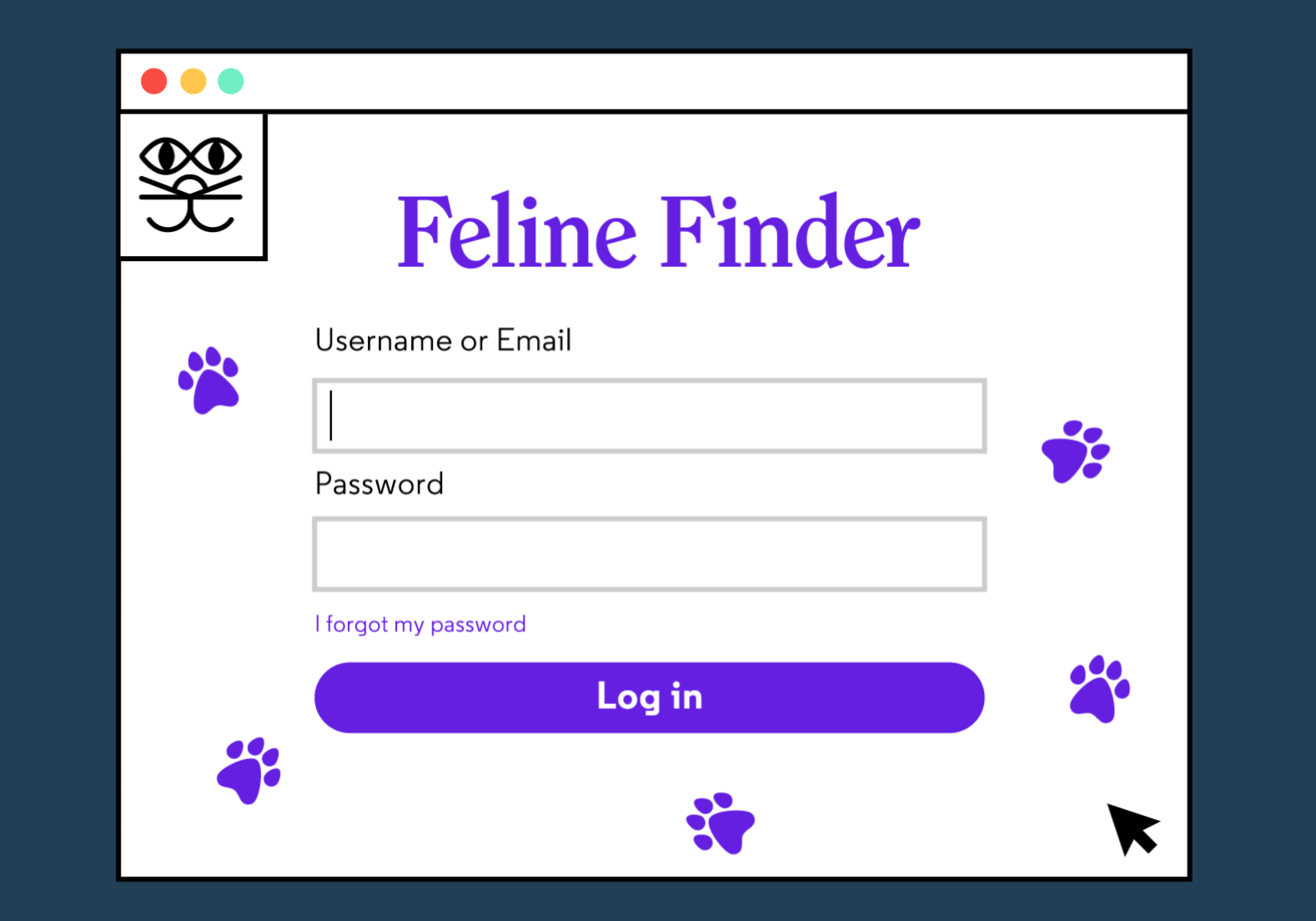
# Authorization and Authentication

Two other concepts we’ll want our server-side logic to handle are authentication and authorization.

Authentication is the process of validating the identity of a user. One technique for authentication is to use logins with usernames and passwords. These credentials need to be securely stored in the back-end on a database and checked upon each visit. Web applications can also use external resources for authentication. You’ve likely logged into a website or application using your Facebook, Google, or Github credentials; that’s also an authentication process.

Authorization controls which users have access to which resources and actions. Certain application views, like the page to edit a social media personal profile, are only accessible to that user. Other activities, like deleting a post, are often similarly restricted.

When building a robust web application back-end, we need to incorporate both authentication (Who is this user? Are they who they claim to be?) and authorization (Who is allowed to do and see what?) into our server-side logic to make sure we’re creating secure, personalized, and dynamic content.



# Different Back-end Stacks

Unlike the front-end, which must be built using HTML, CSS, and JavaScript, there’s a lot of flexibility in which technologies can be used in order to create the back-end of a web application. Developers can construct back-ends in many different languages like PHP, Java, JavaScript, Python, and more.

You don’t need to reinvent the wheel to create a robust back-end. Instead, most developers make use of frameworks which are collections of tools that shape the organization of your back-end and provide efficient ways of accomplishing otherwise difficult tasks.

There are numerous [back-end frameworks](https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Web_frameworks#A_few_good_web_frameworks) from which developers can choose. Here are a few examples:

| **Framework** | **Language** |
| --- | --- |
| [Laravel](https://laravel.com/) | [PHP](http://www.php.net/) |
| [Express.js](https://expressjs.com/) | [JavaScript](https://developer.mozilla.org/en-US/docs/Web/JavaScript) (runs in the [Node environment](https://nodejs.org/en/)) |
| [Ruby on Rails](https://rubyonrails.org/) | [Ruby](https://www.ruby-lang.org/en/) |
| [Spring](https://spring.io/) | [Java](https://www.oracle.com/java/) |
| [JSF](https://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html) | [Java](https://www.oracle.com/java/) |
| [Flask](http://flask.pocoo.org/) | [Python](https://www.python.org/) |
| [Django](https://www.djangoproject.com/) | [Python](https://www.python.org/) |
| [ASP.NET](https://dotnet.microsoft.com/learn/aspnet/what-is-aspnet) | [C#](https://dotnet.microsoft.com/learn/csharp) |

The collection of technologies used to create the front-end and back-end of a web application is referred to as a stack. This is where the term full-stack developer comes from; rather than working in either the front-end or the back-end exclusively, a full-stack developer works in both.

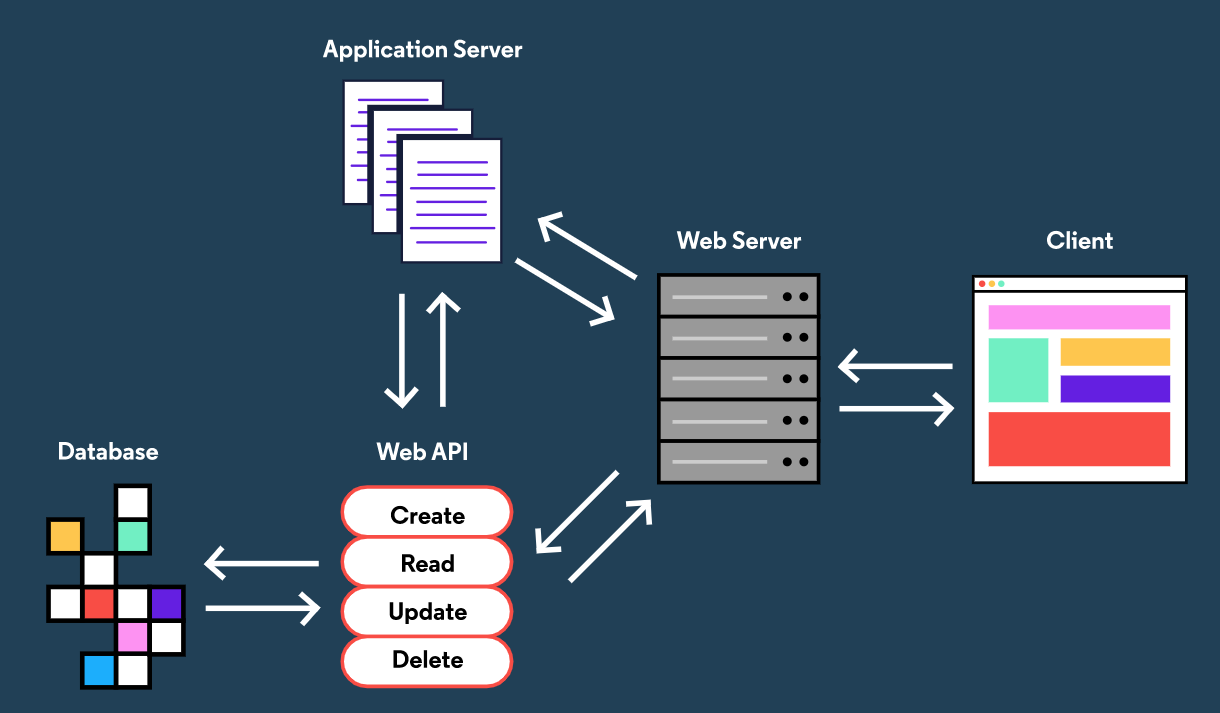
For example, [the MEAN stack](https://en.wikipedia.org/wiki/MEAN_(software_bundle)) is a technology stack for building web applications that uses **M**ongoDB, **E**xpress.js, **A**ngularJS, and **N**ode.js: MongoDB is used as the database, Node.js with Express.js for the rest of the back-end, and Angular is used as a front-end framework. While the [LAMP Stack](https://en.wikipedia.org/wiki/LAMP_(software_bundle)), sometimes considered the archetypal stack, uses **L**inux, **A**pache, **M**ySQL, and **P**HP.

# Review

In order to deliver the front-end of a website or web application to a user, a lot needs to happen behind the scenes on the back-end! Understanding what makes up the back-end can be overwhelming because the back-end has a lot of different parts, and different websites or web applications can have dramatically different back-ends. We covered a lot in this lesson, so let’s review what we learned:

* The front-end of a website or application consists of the HTML, CSS, JavaScript, and static assets sent to a client, like a web browser.
* A web server is a process running on a computer somewhere that listens for incoming requests for information over the internet and sends back responses.
* Storing, accessing, and manipulating data is a large part of a web application’s back-end
* Data is stored in databases which can be relational databases or NoSQL databases.
* The server-side of a web application, sometimes called the application server, handles important tasks such as authorization and authentication.
* The back-end of web application often has a web API which is a way of interacting with an application’s data through HTTP requests and responses.
* Together the technologies used to build the front-end and back-end of a web application are known as the stack, and many different languages and frameworks can be used to build a robust back-end.

Now that you have a sense for server-side web development and what the back-end is, you’re ready to dive in and learn about the different parts in more depth!



# Introduction

For a long time, the browser was the only place JavaScript code could be executed. Web developers had to use a different programming language on the front-end than the back-end. It also meant that, even as JavaScript evolved into a more robust and powerful language, it remained a front-end only language.

Though multiple attempts to create off-browser JavaScript environments have been attempted, [Node.js](https://nodejs.org/en/), invented by Ryan Dahl in 2009, found unprecedented popularity and is currently being used by numerous top-tier companies including Netflix, Uber, Paypal, and eBay. Node.js is a JavaScript runtime, or an environment that allows us to execute JavaScript code outside of the browser. A “runtime” converts code written in a high-level, human-readable, programming language and compiles it down to code the computer can execute.

Though Node was created with the goal of building web servers and web applications in JavaScript, it can also be used for creating command-line applications or desktop applications. In this lesson, we’ll explore some features of Node so you start to feel comfortable with running JavaScript in the Node environment and gain some familiarity with features unique to Node. For more advanced development, Node can be combined with any number of robust frameworks like the [Express.js framework](https://expressjs.com/) for creating effective web application back-ends.

There’s more to learn about Node than we could ever fit in one lesson. We’ll try to point to great resources like [MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript) and the [Node.js documentation](https://nodejs.org/api/). Take your time exploring and use the documentation.

**Instructions**

**1.**

In this lesson, we’ll be providing you a terminal with Node.js already installed. If you’d like to download Node on your local machine and follow along, check out [this article](https://www.codecademy.com/content-items/c4fe3060dbc61fc82d810c4ea06c29a8).

Let’s see what version of Node we have installed. Type node -v in the terminal and then press and hit enter (or return).

Node -v

# The Node REPL

[REPL](https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop) is an abbreviation for **r**ead–**e**val–**p**rint **l**oop. It’s a program that **l**oops, or repeatedly cycles, through three different states: a **r**ead state where the program **r**eads input from a user, the **e**val state where the program **e**valuates the user’s input, and the **p**rint state where the program **p**rints out its evaluation to a console. Then it **l**oops through these states again.

When you install Node, it comes with a built-in JavaScript REPL. You can access the REPL by typing the command node (with nothing after it) into the terminal and hitting enter. A > character will show up in the terminal indicating the REPL is running and prompting your input. The Node REPL will evaluate your input line by line.

By default, you indicate the input is ready for eval when you hit enter. If you’d like to type multiple lines and then have them evaluated at once you can type .editor while in the REPL. Once in “editor” mode, you can type CONTROLD when you’re ready for the input to be evaluated. Each session of the REPL has a single shared memory; you can access any variables or functions you define until you exit the REPL.

A REPL can be extremely useful for performing calculations, learning a language, and developing code. It’s a place where you can explore language features and try things out while receiving immediate feedback. Figuring out how to do this outside of the browser or a website can be really empowering.

The Node environment contains a number of Node-specific global elements in addition to those [built into the JavaScript language](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects). Every Node-specific global property sits inside the [the Node global object](https://nodejs.org/api/globals.html). This object contains a number of useful properties and methods that are available anywhere in the Node environment.

Let’s try out the Node REPL. This will be a good way for you to explore the Node global object!

**Instructions**

**1.**

Let’s enter the Node REPL. Type node in the terminal and press enter.

Checkpoint 2 Passed

**2.**

Experiment on your own within the REPL to get a better sense of it. Here are some suggestions for things to try:

* Access the global object. You can console.log(global) or, since the REPL displays the return of each evaluated line, simply type global and then enter.
* Woah… it looks huge. A lot of that is because of the global.process object. Check out an easier to read list of the properties on the global object with Object.keys(global).
* The global object has a lot of useful properties and methods, and it’s not common to add any to it. However, it is just an object, so we can! Add a property to the global object, eg. global.cat = 'meow!'.
* Now print or return the property you just added:

> console.log(global.cat)

'meow!'

* If you’re familiar with running JavaScript on the browser, you’ve likely encountered the Window object. Here’s one major way that Node differs: try to access the Window object (this will throw an error). The Window object is the JavaScript object in the browser that holds the DOM, since we don’t have a DOM here, there’s no Window object.

You’ll learn more about the global object as you explore Node, but remember that, at its core, it’s just a JavaScript object

# Running a Program with Node

Node was designed with server-side web development in mind and has a lot of thoughtful functionality towards that end. At its most simple, however, it provides the ability to run JavaScript programs on our own computers instead of just in the browser’s console or embedded in HTML.

In this lesson, we’ll explore some of the functionality and properties specific to the Node environment, but first, let’s see how we run a program.

We’ll need to create a file with a .js extension. We’ll call ours **myProgram.js**. Next, we’ll open that file with a text editor and add our code:

// Inside myProgram.js

console.log('Hello World');

Our code is complete! Now, we want to execute it. We’ll open our terminal and navigate to the directory that contains **myProgram.js**. Finally, we’ll type the command node myProgram.js into our terminal.

$ node myProgram.js

The results of our program will print to the terminal.

Hello World

Let’s write a program and run it in Node.

**Instructions**

**1.**

We’ve written a silly sentence JavaScript program in the **app.js** file. There are a number of variables assigned the string '\_\_\_\_'. Replace each of them with words of the designated type:

let adjective = 'silly';

When you’re ready to move on to the next step, click the “Check Work” button.

Checkpoint 2 Passed

**2.**

Let’s run the program in the terminal so we can see its output. Type node app.js in the terminal and press enter and then press “Check Work”. You should see the output of the program in the terminal!

let noun1 = 'girl';

let adjective = 'silly';

let noun2 = 'boy';

let verb = 'fu\*k';

let noun3 = 'dick';

console.log(`The world's first ${noun1} was a very ${adjective} ${noun2} who loved to ${verb} while eating ${noun3} for every meal.`);

# Accessing the Process Object

In computer science, a process is the instance of a computer program that is being executed. You can open Task Manager if you’re on a Windows machine or Activity Monitor from a Mac to see information about the various processes running on your computer right now. Node has a global process object with useful methods and information about the current process.

The process.env property is an object which stores and controls information about the environment in which the process is currently running. For example, the process.env object contains a PWD property which holds a string with the directory in which the current process is located. It can be useful to have some if/else logic in a program depending on the current environment— a web application in a development phase might perform different tasks than when it’s live to users. We could store this information on the process.env. One convention is to add a property to process.env with the key NODE\_ENV and a value of either production or development.

if (process.env.NODE\_ENV === 'development'){

console.log('Testing! Testing! Does everything work?');

}

The process.memoryUsage() returns information on the CPU demands of the current process. It returns a property that looks similar to this:

{ rss: 26247168,

heapTotal: 5767168,

heapUsed: 3573032,

external: 8772 }

Heap can mean different things in different contexts: a heap can refer to [a specific data structure](https://en.wikipedia.org/wiki/Heap_(data_structure)), but it can also refer to the a block of [computer memory](https://en.wikipedia.org/wiki/Memory_management). process.memoryUsage().heapUsed will return a number representing how many bytes of memory the current process is using.

The process.argv property holds an array of command line values provided when the current process was initiated. The first element in the array is the absolute path to Node, which ran the process. The second element in the array is the path to the file that’s running. The following elements will be any command line arguments provided when the process was initiated. Command line arguments are separated from one another with spaces.

node myProgram.js testing several features

console.log(process.argv[3]); // Prints 'several'

We’ve only covered a few of the properties of the process object, so make sure to check out the [documentation on the process object](https://nodejs.org/api/process.html) to learn more about it and explore some of its other methods and properties.

Let’s get some practice using the process object!

**Instructions**

**1.**

We want the program in **app.js** to store the starting amount of memory used (heapUsed), perform an operation, and then compare the final amount of memory used to the original amount. Right now, the initialMemory variable is assigned to null. Change this line, so that initialMemory is instead assigned the value of the heapUsed property on the object returned from invoking the process.memoryUsage() method.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

We want the user of the program to be able to fill in their own word when they run the program. Right now word is assigned to null. Change the program so that when a user initiates the program with an additional command line argument, word will be assigned that value. For example, running the program with the command: node app.js Codecademy would result in word being assigned the value 'Codecademy'

Checkpoint 3 Passed

Hint

When we initiate the program with the command node app.js Codecademy, the first element in the process.argv array is the file path to Node, the second is the file path to the location program file, and the third is the string 'Codecademy'. In **app.js**, we’ll want to assign word to this third element in the array:

let word = process.argv[2];

**3.**

Awesome! Now let’s run the program. Type node app.js followed by any word you like and then hit enter.

Checkpoint 4 Passed

Hint

Initiate the process with any additional command line argument you like. For example, you can type node app.js Codecademy and then press enter.

let initialMemory = process.memoryUsage().heapUsed;

let word = process.argv[2];

console.log(`Your word is ${word}`)

// Create a new array

let wordArray = [];

// Loop 1000 times, pushing into the array each time

for (let i = 0; i < 1000; i++){

  wordArray.push(`${word} count: ${i}`)

}

console.log(`Starting memory usage: ${initialMemory}. \nCurrent memory usage: ${process.memoryUsage().heapUsed}. \nAfter using the loop to add elements to the array, the process is using ${process.memoryUsage().heapUsed - initialMemory} more bytes of memory.`)

# Core Modules and Local Modules

Modularity is a software design technique where one program has distinct parts each providing a single piece of the overall functionality. These separate modules come together to build a cohesive whole. Modularity is essential for creating scalable programs which incorporate libraries and frameworks and separate the program’s concerns into manageable chunks. Essentially, a module is a collection of code located in a file. Instead of having an entire program located in a single file, code is organized into separate files and combined through requiring them where needed using the require() function.

To save developers from having to reinvent the wheel each time, Node has several modules included within the environment to efficiently perform common tasks. These are known as the core modules. The core modules are defined within Node.js’s source and are located in the lib/ folder. Core modules are required by passing a string with the name of the module into the require() function:

// Require in the 'events' core module:

let events = require('events');

We can use the same require() function to require modules of our own creation. To handle these different tasks, the require() function includes some interesting logic “under the hood.” The require() function will first check to see if its argument is a core module, if not, it will move on to different attempts to locate it. Check out the [Node Modules documentation](https://nodejs.org/api/modules.html#modules_modules) to learn more about how require() works.

Let’s walk through the process of requiring a local module:

// dog.js

module.exports = class Dog {

constructor(name) {

this.name = name;

}

praise() {

return `Good dog, ${this.name}!`;

}

};

Above, in the **dog.js** file, we assign the Dog class as the value of module.exports. Each JavaScript file in the Node environment has a special JavaScript object called module.exports. It holds everything in that file, or module, that’s available to be required into a different file.

// app.js

let Dog = require('./dog.js');

const tadpole = new Dog('Tadpole');

console.log(tadpole.praise());

In our **app.js** file we assign the variable Dog to the module.exports object of our **dog.js** file by invoking the require() function. Unlike when we require core modules which are required in with the name of the module as a string, local modules are required by passing in the path to the module. The require() function has some other quirks, like assuming file extensions if none are provided; this means we could have written let Dog = require('./dog'); in place of let Dog = require('./dog.js'); in the code above, and the require() function would have still correctly located and required in **dog.js**.

**Instructions**

**1.**

We wrote a program where a Dog can fight a Cat, but it doesn’t work yet because we haven’t properly connected our code. We created two modules: **cat.js**, which contains our Cat class, and **dog.js**, which contains our Dog class. But we need to assign these modules to their module.exports. Let’s start with **cat.js**. Inside **cat.js** assign the Cat class as the value of module.exports.

Checkpoint 2 Passed

Hint

There are a couple ways to do this. For example, your **cat.js** file could look like this:

module.exports = class Cat {

constructor(name, clawStrength) {

this.name = name;

this.clawStrength = clawStrength;

}

};

**2.**

We’ll also need to export our Dog class. Navigate to **dog.js** and assign the Dog class as the value of module.exports.

Checkpoint 3 Passed

Hint

There are a couple ways to do this. For example, your **dog.js** file could look like this:

module.exports = class Dog {

constructor(name, toothStrength) {

this.name = name;

this.toothStrength = toothStrength;

}

};

**3.**

Great! Now our Dog and Cat classes are being exported, but we still need to require them into **app.js** for our function to work properly. At the top of the **app.js** file, create a variable Dog and assign as its value invoking the require() function with the relative path of the **dog.js** file. Next, create a Cat variable and assign as its value invoking the require() function with the relative path of the **cat.js** file.

Checkpoint 4 Passed

Hint

The top of your **app.js** file should have these two lines:

let Cat = require('./cat.js');

let Dog = require('./dog.js');

**4.**

Let’s see this program in action! Use the node command to run **app.js** from the terminal.

module.exports = class Cat {

  constructor(name, clawStrength) {

    this.name = name;

    this.clawStrength = clawStrength;

  }

};

// Require modules in:

let Cat = require('./cat.js');

let Dog = require('./dog.js');

let fight = (dog, cat) => {

    if (dog.toothStrength > cat.clawStrength) {

        console.log(`${dog.name} wins!`);

    }

    else if (dog.toothStrength < cat.clawStrength) {

        console.log(`${cat.name} wins!`);

    }

    else {

        console.log(`${dog.name} and ${cat.name} are equally skilled fighters!`);

    }

}

const myDog = new Dog('Rex', Math.random());

const myCat = new Cat('Tabby', Math.random());

fight(myDog, myCat);

module.exports = class Dog {

  constructor(name, toothStrength) {

    this.name = name;

    this.toothStrength = toothStrength;

  }

};

# Node Package Manager

In addition to local modules and core modules, we can take advantage of third-party modules. Using libraries created by other developers is an essential aspect of production; we don’t have to reinvent the wheel each time we want to include new functionality into our applications. NPM, which stands for **N**ode **P**ackage **M**anager, is an online collection, or registry, of software. Developers can share code they’ve written to the registry or download code provided by other developers.

When we download Node, the npm command-line tool is downloaded as well, which enables us to interact with the registry via our terminal. There are hundreds of thousands of packages of re-usable code in the NPM registry including powerful and popular frameworks like express and react. You can explore the collection at [the npm website.](https://www.npmjs.com/)

One package we like is [nodemon](https://www.npmjs.com/package/nodemon). It’s a powerful tool for development in Node that watches all the files in a project you’re working on, and automatically restarts your application when any of them change.



# Event-Driven Architecture

Node is often described as having event-driven architecture. Let’s explore what that means.

In traditional imperative programming, we give the computer a series of instructions to execute in a pre-defined order. In contrast, when we write web applications, we often need to write logic to handle situations without knowing exactly when they’ll occur. For example, when programming a website, we might provide functionality for a click event without knowing when a user will trigger it. When Node was created, it applied this same concept of event-driven principles to the back-end environment.

Node provides an EventEmitter class which we can access by requiring in the events core module:

// Require in the 'events' core module

let events = require('events');

// Create an instance of the EventEmitter class

let myEmitter = new events.EventEmitter();

Each event emitter instance has an .on() method which assigns a listener callback function to a named event. The .on() method takes as its first argument the name of the event as a string and, as its second argument, the listener callback function.

Each event emitter instance also has an .emit() method which announces a named event has occurred. The .emit() method takes as its first argument the name of the event as a string and, as its second argument, the data that should be passed into the listener callback function.

let newUserListener = (data) => {

console.log(`We have a new user: ${data}.`);

};

// Assign the newUserListener function as the listener callback for 'new user' events

myEmitter.on('new user', newUserListener)

// Emit a 'new user' event

myEmitter.emit('new user', 'Lily Pad') //newUserListener will be invoked with 'Lily Pad'

Let’s create an event emitter!

**Instructions**

**1.**

In **app.js**, we’ve required in the events core module and written a function listenerCallback which expects to be passed data and will log a string to the console which incorporates that data. Now it’s time to create an event emitter. Create a new variable, myEmitter and assign as its value a new instance of the event emitter class.

Checkpoint 2 Passed

Hint

You’ll do this by invoking EventEmitter():

let myEmitter = new events.EventEmitter();

**2.**

Invoke myEmitter‘s .on() method passing in 'celebration' as the event name and listenerCallback as the listener callback function.

Checkpoint 3 Passed

Hint

An event emitter’s .on() method takes as its the name of the event as a string as its first argument and, the listener callback function as its second argument.

someEmitter.on('name of event', callbackFunction);

**3.**

Let’s emit a 'celebration' event! Invoke myEmitter‘s .emit() method passing in 'celebration' as the event name and a string of your choice as the second argument.

Checkpoint 4 Passed

Hint

If you like to celebrate good times, you might want to pass in 'good times, come on!' as your second argument:

myEmitter.emit('celebration', 'good times, come on!');

**4.**

Let’s see this program run! Use the node command to run **app.js** from the terminal.

// Here we require in the 'events' module and save a reference to it in an events variable

let events = require('events');

let listenerCallback = (data) => {

    console.log(`Celebrate ${data}`);

}

// Here we create an instance of the EventEmitter class

let myEmitter = new events.EventEmitter();

// Here we subscribe to 'celebration' events and provide a callback function which will be passed the event's data

myEmitter.on('celebration', listenerCallback);

// Here we emit an event, we pass the event type, 'celebration', as the first argument, and the event data as the second

myEmitter.emit('celebration', 'good times, come on!');

# Asynchronous JavaScript with Node.js

In server-side development, we often perform time-consuming tasks such as reading files or querying a database. Instead of halting the execution of our code to await these operations or using multiple threads like other back end environments, Node was designed to use an [event loop](https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/) like the one used in browser-based JavaScript execution. The event-loop enables asynchronous actions to be handled in a [non-blocking](https://nodejs.org/en/docs/guides/blocking-vs-non-blocking/) way.

Node provides a number of APIs for performing asynchronous tasks which expect callback functions to be passed in as arguments. Under the hood, these APIs trigger the subscription to and emitting of events to signal the completion of the operation. When the operation completes, the callback function is added to a queue, or line, of tasks waiting for their turn to be executed. When the current stack, or list, or synchronous tasks finish executing, the operations on the queue will be performed.

This means if synchronous tasks never end, operations waiting in the event-queue would never have the chance to run. Take a look at the following example code using the asynchronous Node setTimeout() API which asynchronously executes a provided callback function after a given delay:

let keepGoing = true;

let callback = () => {

keepGoing = false;

};

setTimeout(callback, 1000); // Run callback after 1000ms

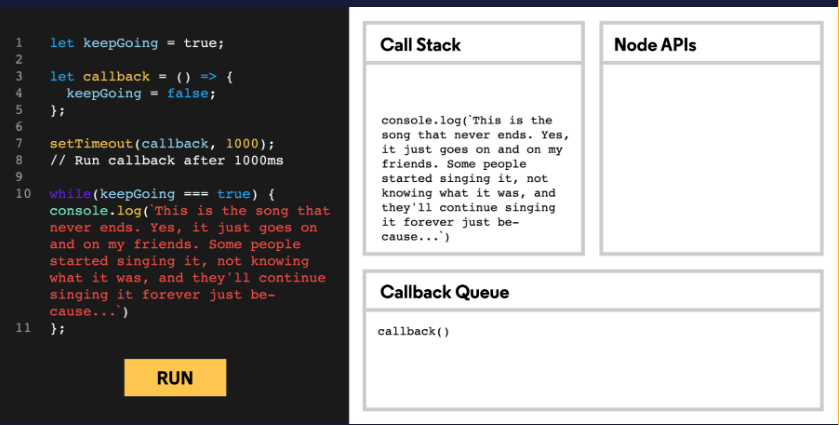
while(keepGoing === true) {

console.log(`This is the song that never ends. Yes, it just goes on and on my friends. Some people started singing it, not knowing what it was, and they'll continue singing it forever just because...`)

};

This while-loop will continue forever! Even though the callback changing the keepGoing variable to false is added to the event queue after 1 second, it will never have a chance to run— the synchronous code from the loop will always fill the stack! If we wanted to avoid the infinite loop, we could replace the while-loop with an asynchronous function— for example, the Node setInterval() API.

Note: The modern way of handling asynchronous tasks is through JavaScript Promises (developers also favor the newer async...await syntax). If you’re not familiar with these topics, check out [our lessons on them](https://www.codecademy.com/learn/asynchronous-javascript). Newer versions of Node (v8.0.0 and later) provide a collection of the traditional Node asynchronous APIs formatted for promises instead of callbacks. This can be found on util.promisify. Many contemporary 3rd party libraries also favor promise-based patterns over traditional callbacks.



# User Input/Output

If you’ve worked with JavaScript before, you’re likely familiar with the concept of input/output even if you haven’t heard it called that. At its most abstract, output is any data or feedback that a computer provides (like to a human user), while input is data provided to the computer. When we use console.log() we prompt the computer to output information to the console. In the Node environment, the console is the terminal, and the console.log() method is a “thin wrapper” on the .stdout.write() method of the process object. stdout stands for standard output.

In Node, we can also receive input from a user through the terminal using the stdin.on() method on the process object:

process.stdin.on('data', (userInput) => {

let input = userInput.toString()

console.log(input)

});

Here, we were able to use .on() because under the hood process.stdin is an instance of EventEmitter. When a user enters text into the terminal and hits enter, a 'data' event will be fired and our anonymous listener callback will be invoked. The userInput we receive is an instance of [the Node Buffer class](https://nodejs.org/api/buffer.html#buffer_buffer), so we convert it to a string before printing.

**Instructions**

**1.**

We’ve gotten started making a number guessing game. We’ve written a function playGame() which expects a line of user input corresponding to a guess. Input read through the terminal is received as a Buffer object with a new line character at the end, so we’ve converted it to a string and trimmed off the unnecessary new line character. We then feed the processed input into the testNumber() function required in from **game.js**.

Read through the code and try to figure out what has been done and what is left to do to make this an interactive command line program (**game.js** includes the core game logic, but reading through it is optional).

Checkpoint 2 Passed

**2.**

Everything about the game is complete except for reading input from the user. Based on what you’ve learned about getting user input from the terminal, complete the game.

Checkpoint 3 Passed

Hint

We wrote the playGame() function, but it’s never invoked! Where should this happen?

We need to assign playGame() as the listener callback function so that it’s invoked any time process.stdin emits a 'data' event.

process.stdin.on('name of event', listenerCallbackFunction);

**3.**

Awesome! Let’s play! Run **app.js** and play the game.

let {testNumber} = require('./game.js');

process.stdout.write("I'm thinking of a number from 1 through 10. What do you think it is? \n(Write \"quit\" to give up.)\n\nIs the number ... ");

let playGame = (userInput) => {

  let input = userInput.toString().trim();

  testNumber(input);

};

process.stdin.on('data', playGame);

# Errors

The Node environment has all the standard JavaScript errors such as EvalError, SyntaxError, RangeError, ReferenceError, TypeError, and URIError as well as the JavaScript Error class for creating new error instances. Within our own code, we can generate errors and throw them, and, with synchronous code in Node, we can use [error handling](https://www.codecademy.com/learn/javascript-errors-debugging/modules/errors-and-error-handling) techniques such as try...catch statements.

Many asynchronous Node APIs use error-first callback functions: callback functions which have an error as the first expected argument and the data as the second argument. If the asynchronous task results in an error, it will be passed in as the first argument to the callback function. If no error was thrown, the first argument will be undefined.

const errorFirstCallback = (err, data) => {

if (err) {

console.log(`There WAS an error: ${err}`);

} else {

// err was falsy

console.log(`There was NO error. Event data: ${data}`);

}

}

**Instructions**

**1.**

In order to understand why Node uses error-first callbacks in many of its asynchronous APIs, let’s demonstrate that traditional try...catch statements won’t work for errors thrown during asynchronous operations.

In **naiveAsyncErr.js**, we require in the local **api.js** module which contains the api.naiveErrorProneAsyncFunction() method. This asynchronous method throws an error whenever it is passed the input 'problematic input'. We would want the try...catch statement in **naiveAsyncErr.js** to catch this error, but it cannot since the error is thrown asynchronously.

In the terminal, execute the **naiveAsyncErr.js** file. You’ll see that the intended output, Something went wrong. ${err}\n, wasn’t logged— meaning that the error was never caught.

Checkpoint 2 Passed

Hint

Type node naiveAsyncErr.js in the terminal and press enter.

**2.**

Now take a look at **app.js**. Here we wrote an error-first callback function. This callback does the error handling so there’s no need for a try...catch statement. This is how most Node asynchronous APIs are set up to handle errors. In **app.js** we require in the **api.js** module. This time, we’ll use an asynchronous method, api.errorProneAsyncApi(), designed to work like the asynchronous methods in Node.

At the botton of **app.js**, invoke the api.errorProneAsyncApi() method with 'problematic input' as the first argument and the error-first callback as the second.

Checkpoint 3 Passed

Hint

We’ve provided errorFirstCallback for you:

api.errorProneAsyncApi('problematic input', errorFirstCallback);

**3.**

Let’s run **app.js** in the terminal so we can see how it properly handles the error created during the asynchronous operation.

Checkpoint 4 Passed

Hint

Type node app.js in the terminal and press enter.

const api = require('./api.js');

// Not an error-first callback

let callbackFunc = (data) => {

   console.log(`Something went right. Data: ${data}\n`);

};

try {

  api.naiveErrorProneAsyncFunction('problematic input', callbackFunc);

} catch(err) {

  console.log(`Something went wrong. ${err}\n`);

}

# Filesystem

All of the data on a computer is organized and accessed through a filesystem. When running JavaScript code on a browser, it’s important for a script to have only limited access to a user’s filesystem. This technique of isolating some applications from others is known as sandboxing. Sandboxing protects users from malicious programs and invasions of privacy.

In the back-end, however, less restricted interaction with the filesystem is essential. The Node fs core module is an API for interacting with the **f**ile **s**ystem. It was modeled after the [POSIX](https://en.wikipedia.org/wiki/POSIX) standard for interacting with the filesystem.

Each method available through the fs module has a synchronous version and an asynchronous version. One method available on the fs core module is the .readFile() method which **read**s data from a provided **file**:

const fs = require('fs');

let readDataCallback = (err, data) => {

if (err) {

console.log(`Something went wrong: ${err}`);

} else {

console.log(`Provided file contained: ${data}`);

}

};

fs.readFile('./file.txt', 'utf-8', readDataCallback);

Let’s walk through the example above:

* We required in the fs core module.
* We define an error-first callback function which expects an error to be passed as the first argument and data as the second. If the error is present, the function will print Something went wrong: ${err}, otherwise, it will print Provided file contained: ${data}.
* We invoked the .readFile() method with three arguments:
  1. The first argument is a string that contains a path to the file **file.txt**.
  2. The second argument is a string specifying the file’s [character encoding](https://en.wikipedia.org/wiki/Character_encoding) (usually ‘utf-8’ for text files).
  3. The third argument is the callback function to be invoked when the asynchronous task of reading from the file system is complete. Node will pass the contents of **file.txt** into the provided callback as its second argument.

**Instructions**

**1.**

We’ve created a devious treasure hunt for you! Your task is to use fs.readFile() to figure out the secret word and assign that value to the secretWord variable in **app.js**. Here’s your first clue, found scratched into walls of an abandoned castle: **fileOne.txt**.

There are many ways to complete this treasure hunt! You could write one program in **app.js** to solve the puzzle or you can run **app.js** to gain new insight and then change the program based on what you’ve figured out.

If you want some direction, but aren’t quite ready to check out the hint: we suggest you use the fs.readFile() method to print the contents of **fileOne.txt**.

Checkpoint 2 Passed

Hint

We created the following error-first callback function to start:

let readDataCallback = (err, data) => {

if (err) {

console.log(`Something went wrong: ${err}`);

} else {

console.log(`Provided file contained: ${data}`);

}

};

Next we ran:

fs.readFile('./fileOne.txt', 'utf-8', readDataCallback);

This gave us the clue that the next file was called **anotherFile.txt**, so we ran:

fs.readFile('./anotherFile.txt', 'utf-8', readDataCallback);

That gave us the clue that the next file was called **finalFile.txt**, so we ran:

fs.readFile('./finalFile.txt', 'utf-8', readDataCallback);

That gave us the secret word!

const fs = require('fs');

let secretWord = null;

let readDataCallback = (err, data) => {

  if (err) {

    console.log(`Something went wrong: ${err}`);

  } else {

    console.log(`Provided file contained: ${data}`);

  }

};

//fs.readFile('./fileOne.txt', 'utf-8', readDataCallback);

//fs.readFile('./anotherFile.txt', 'utf-8', readDataCallback);

fs.readFile('./finalFile.txt', 'utf-8', readDataCallback);

//secretWord = "cheeseburgerpizzabagels"

**Readable Streams**

In the previous exercise, we practiced reading the contents of entire files into our JavaScript programs. In more realistic scenarios, data isn’t processed all at once but rather sequentially, piece by piece, in what is known as a *stream*. Streaming data is often preferable since you don’t need enough RAM to process all the data at once nor do you need to have all the data on hand to begin processing it.

One of the simplest uses of streams is reading and writing to files line-by-line. To read files line-by-line, we can use the .createInterface() method from the readline core module. .createInterface() returns an EventEmitter set up to emit 'line' events:

const readline = require('readline');

const fs = require('fs');

const myInterface = readline.createInterface({

input: fs.createReadStream('text.txt')

});

myInterface.on('line', (fileLine) => {

console.log(`The line read: ${fileLine}`);

});

Let’s walk through the above code:

* We require in the readline and fs core modules.
* We assign to myInterface the returned value from invoking readline.createInterface() with an object containing our designated input.
* We set our input to fs.createReadStream('text.txt') which will create a stream from the **text.txt** file.
* Next we assign a listener callback to execute when line events are emitted. A 'line' event will be emitted after each line from the file is read.
* Our listener callback will log to the console 'The line read: [fileLine]', where [fileLine] is the line just read.

Let’s practice making a readable stream.

**Instructions**

**1.**

You’re going to create a program that reads each item off of a shopping list (located in **shoppingList.txt**) and prints it to the console. Let’s take it one step at a time.

Create a myInterface variable. Assign myInterface the value returned from invoking readline.createInterface().

You’ll want to invoke readline.createInterface() with an object with a key of input and a value of fs.createReadStream(). Remember that fs.createReadStream() expects the file (as a string) from which it should read.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Great work. Let’s create a listener callback function to use in the next step. Name this function printData. printData() should expect to receive some data (we named our parameter data) and it should log that data to the console in the format: Item: [data], where [data] is the argument passed into the function.

Checkpoint 3 Passed

Hint

Here’s our printData() function looks:

const printData = (data) => {

console.log(`Item: ${data}`);

};

**3.**

We’re nearly there! Remember that a 'line' event will be emitted after each line from the file is read. Let’s assign our printData() function to execute whenever a 'line' event is emitted by using myInterface‘s .on() method.

Checkpoint 4 Passed

Hint

myInterface.on('line', printData);

**4.**

Sweet! Let’s run the program in the terminal. Type node app.js in the terminal and press enter. If everything worked, each item from the shopping list should be printed to the terminal.

const readline = require('readline');

const fs = require('fs');

const myInterface = readline.createInterface({

  input: fs.createReadStream('shoppingList.txt')

});

const printData = (data) => {

  console.log(`Item: ${data}`);

};

myInterface.on('line', printData);

# Writable Streams

In the previous exercise, we were reading data from a stream, but we can also write to streams! We can create a writeable stream to a file using the fs.createWriteStream() method:

const fs = require('fs')

const fileStream = fs.createWriteStream('output.txt');

fileStream.write('This is the first line!');

fileStream.write('This is the second line!');

fileStream.end();

In the code above, we set the output file as **output.txt**. Then we .write() lines to the file. Unlike a readable stream, which ends when it has no more data to read, a writable stream could remain open indefinitely. We can indicate the end of a writable stream with the .end() method.

Let’s combine our knowledge of readable and writable streams to create a program which reads from one text file and then writes to another.

**Instructions**

**1.**

We’re going to create a writeable stream. We want to write to a file named **shoppingResults.txt**. Create a variable fileStream and assign as its value the writable stream.

Checkpoint 2 Passed

Hint

The fs core module has already been required in so we have access to the fs.createWriteStream() method:

const fileStream = fs.createWriteStream('shoppingResults.txt');

**2.**

Great work. Let’s create a listener callback function to use in the next step. Name this function transformData. transformData should expect to receive some data (we named our parameter line) and it should write() to the writable stream (fileStream) in the format They were out of: [line]\n, where [line] is the argument passed into the function.

Checkpoint 3 Passed

Hint

Here’s how our transformData() function looks:

let transformData = (line) => {

fileStream.write(`They were out of: ${line}\n`);

}

**3.**

We’re nearly there! Let’s assign our transformData function to execute whenever a 'line' event is emitted on the myInterface stream.

Checkpoint 4 Passed

Hint

We’ll want to use myInterface‘s on() method:

myInterface.on('line', transformData);

**4.**

Sweet! Let’s run the program in the terminal. Type node app.js in the terminal and press enter. If everything worked, you should be able to open the **shoppingResults.txt** file and see that it has the correct contents.

const readline = require('readline');

const fs = require('fs');

const myInterface = readline.createInterface({

  input: fs.createReadStream('shoppingList.txt')

});

const fileStream = fs.createWriteStream('shoppingResults.txt');

let transformData = (line) => {

 fileStream.write(`They were out of: ${line}\n`);

}

myInterface.on('line', transformData);

# Create an HTTP Server

Node was designed with back end development needs as a top priority. One of these needs is the ability to create web servers, computer processes that listen for requests from clients and return responses. A Node core module designed to meet these needs is the http module. This module contains functions which simplify interacting with HTTP and streamline receiving and responding to requests.

The http.createServer() method returns an instance of an http.server. An http.server has a method .listen() which causes the server to “listen” for incoming connections. When we run http.createServer() we pass in a custom callback function (often referred to as the requestListener). This callback function will be triggered once the server is listening and receives a request.

Let’s break down how the requestListener callback function works:

* The function expects two arguments: a request object and a response object.
* Each time a request to the server is made, Node will invoke the provided requestListener callback function, passing in the request and response objects of the incoming request.
* Request and response objects come with a number of properties and methods of their own, and within the requestListener function, we can access information about the request via the request object passed in.
* The requestListener is responsible for setting the response header and body.
* The requestListener must signal that the interaction is complete by calling the response.end() method.

const http = require('http');

let requestListener = (request, response) => {

response.writeHead(200, {'Content-Type': 'text/plain' });

response.write('Hello World!\n');

response.end();

};

const server = http.createServer(requestListener);

server.listen(3000);

Let’s walk through the above code:

* We required in the http core module.
* We created a server variable assigned to the return value of the http.createServer() method.
* We invoked http.createServer() with our requestListener callback. This is similar to running the .on() of an EventEmitter: the requestListener will execute whenever an HTTP request is sent to the server on the correct port.
* Within the requestListener callback, we make changes to the response object, response, so that it can send the appropriate information to the client sending the request. The status code 200 means that no errors were encountered. The header communicates that the file type is text, rather than something like audio or compressed data.
* The last line starts the server with the port 3000. Every server on a given machine specifies a unique port so that traffic can be correctly routed.

You could run the above code on your local machine, and access it by visiting http://localhost:3000/ from your browser. “localhost” is used to refer to the same computer that’s running the current Node process.

In our example web server, we showed you a handful of the methods available on response objects. Be sure to check out [the documentation](https://nodejs.org/api/http.html) to learn other methods and properties available on response and request objects.

**Instructions**

**1.**

In the example code earlier, we started a web server which only served the string ‘Hello World!\n’, now we want to serve some HTML. You can see the HTML for the website we’re going to serve in the **myWebsite.html** file. We’ve written a requestListener for you and required it into **app.js**. You can look at the code for requestListener in the **callbackFile.js** file.

Our requestListener expects to take in request and response objects (req and res). It invokes the fs.readFile() method with **myWebsite.html**. If there’s an error it will write the error while attempting to read the file to the response object, otherwise, it will write the contents of the file.

When you feel like you have a sense for the code base, move on to the next step!

Checkpoint 2 Passed

**2.**

It’s time to make the web server! In **app.js**, create a variable server and assign to it the value of invoking the http.createServer(). Remember, you’ll need to pass a callback function in as the argument to http.createServer(). We’ve already required in the function requestListener from the **callbackFile.js** file.

Checkpoint 3 Passed

Hint

You should invoke http.createServer() with the requestListener function we provided for you:

const server = http.createServer(requestListener);

**3.**

Awesome! Now that we have the server we need it to start listening for incoming requests! We supplied a PORT variable at the top of **app.js**. Invoke the server.listen() method passing in PORT as the argument.

Checkpoint 4 Passed

**4.**

Let’s get this server up and running! Type node app.js in the terminal and press enter. If everything worked, you should be able to navigate to http://localhost:4001 in the browser and see the rendered HTML.

Feel free to make changes and experiment. You can stop the server by typing controlc, make whatever changes you’d like, and then start the server by typing node app.js in the terminal again and pressing enter.

const http = require('http');

let {requestListener} = require('./callbackFile.js');

const PORT = process.env.PORT || 4001;

const server = http.createServer(requestListener);

server.listen(PORT)

# Review

Awesome work! You’ve learned a lot about Node.js including:

* Node.js is a JavaScript runtime, an environment that allows us to execute our JavaScript code by converting it into something a computer can understand.
* REPLs are processes that **r**ead, **e**valuate, **p**rint, and repeat (**l**oop), and Node.js comes with its own REPL we can access in our terminal with the node command.
* We run JavaScript programs with Node in the terminal by typing node followed by the file name (if we’re in the same directory) or the absolute path of the file.
* Code can be organized into separate files, modules, and combined through requiring them where needed using the require() function.
* In addition to core modules, modules included within the environment to efficiently perform common tasks, we can also create our own modules using module.exports and the require() function.
* We can access NPM, a registry of hundreds of thousands of packages of re-usable code from other developers, directly through our terminal.
* Node has an event-driven architecture.
* We can make our own instances of the EventEmitter class and we can subscribe to listen for named events with the .on() method and emit events with the .emit() method.
* Node uses an event loop which enables asynchronous actions to be handled in a non-blocking way by adding callback functions to a queue of tasks to be executed when the callstack is empty.
* In order to handle errors during asynchronous operations, provided callback functions are expected to have an error as their first parameter.
* Node allows for both output, data/feedback to a user provided by a computer, and input data/feedback to the computer provided by the user.
* The Node fs core module is an API for interacting with the **f**ile **s**ystem.
* Streams allow us to read or write data piece by piece instead of all at once.
* The Node http core module allows for easy creation of web servers, computer processes that listen for requests from clients and return responses.

Woah, that was a lot… And there’s even more to Node that we didn’t cover in this lesson, but don’t panic! Learning Node isn’t about memorizing every aspect of the environment. The best way to get comfortable with Node is just to practice making things in it. Your imagination is the limit! If you haven’t already, [download Node on your local machine](https://www.codecademy.com/articles/setting-up-node-locally). You can start by recreating some of the programs you built in this lesson— put your own spin on a guessing game, for example. If you’re eager to build web application back-ends, we recommend you start [learning the awesome Express.js](https://www.codecademy.com/learn/learn-express) web framework.

Great work! We’re excited to see what you build!