

# Contents

<b>Book Todo</b>	<b>4</b>
<b>Tips</b>	<b>4</b>
<b>Introduction to Web Development</b>	<b>4</b>
Hello World? . . . . .	4
How does a web page actually get to me? . . . . .	5
What is the HTTP Protocol? . . . . .	5
Delving into Dynamic Web Pages . . . . .	6
Moving Application Logic to the Client . . . . .	6
HTML Pages Expanded . . . . .	8
Styling with CSS . . . . .	9
Submitting New Data Using HTTP POST and Forms . . . . .	10
ADD IMAGE HERE OF REQUESTS . . . . .	10
Modern Conventions . . . . .	11
Adjustments . . . . .	11
<b>JavaScript</b>	<b>12</b>
JavaScript Background . . . . .	12
Variables . . . . .	13
Arrays . . . . .	13
Immutability and Concat . . . . .	13
The Map Expression . . . . .	14
Destructuring Assignment Expression . . . . .	14
Objects and JavaScript Object Notation (JSON) . . . . .	14
Referencing Properties . . . . .	15
Adding Properties . . . . .	15
Object Methods . . . . .	16
INSERT INFORMATION ABOUT OBJECT METHODS IN JAVASCRIPT . . . . .	17
Constructor Functions . . . . .	17
Class Syntax . . . . .	18
<b>JAVASCRIPT PROTOTYPES</b>	<b>19</b>
JavaScript Functions . . . . .	19
Function Expressions and Declarations (Regular Functions) . . . .	20
JavaScript Nuances and Tips . . . . .	21
Using Strict Mode . . . . .	21
Closures . . . . .	21
Hoisting . . . . .	22
Spread Syntax . . . . .	22
<b>Let's React</b>	<b>22</b>
Components . . . . .	23
Adding Complexity . . . . .	25

Components With Data (Props) . . . . .	26
Dealing With Component State . . . . .	28
Component Helper Functions . . . . .	29
Destructuring in JavaScript . . . . .	29
Component Re-Renders . . . . .	31
Stateful Components . . . . .	31
Event Handlers . . . . .	33

ver-  
image:  
cover.jpg  
eator:  
role:  
au-  
thor  
text:  
“Joseph  
Sem-  
rai”  
scrip-  
tion:  
“An  
in-  
tro-  
duc-  
tion  
to  
mod-  
ern  
full-  
stack  
de-  
vel-  
op-  
ment  
in  
2021.”  
nguage:  
en-  
US  
ghits:  
Com-  
mer-  
cial  
ylesheet:  
epub.css  
tle:

text: "Write for Anything: Modern Full-Stack Development"  
type: main  
[TOC]

## Book Todo

- Add Images/Diagrams written with the iPad Pro for
  - Document Object Model
  - HTTP Requests (Submitting New Data Using HTTP POST and Forms)
- Add section about style guides, linting, etc.
- Replace mentions of ‘I’ in first person
- Explain what the global object is in the JavaScript section

## Tips

As a beginner, whenever you encounter an error message, you should always take a second to understand the cause of the error message. With this, you should write your code advancing in very small steps in order to avoid writing a lot of code that ends up not working. This way, you will only have a few lines that can be problematic at a given time and can take the time to understand why these lines are throwing errors, etc.

## Introduction to Web Development

This book was written originally to take someone from knowing nothing to knowing a little bit of almost everything but not exactly everything that one would need to know to do something useful with web development. Looking at this now, you should probably have some programming experience or at least some background on how the magic that is the internet works, but if you don’t, you’ll probably be fine, just Google the things that aren’t super clear to you. Have trust in the system.

## Hello World?

So, before we actually get into things, I just want to introduce you to some words that are super common in the world of web development and beyond.

Also, please note that the explanations in Chapter 0 follow subpar programming practices coupled with mediocre systems design. These are just examples to be looked at as a high level overview of how things may work while giving you a taste at what these Application Programming Interfaces (APIs) can do. Additionally, there are some omitted blocks of code that are not relevant to your understanding of these concepts. As a result, the code mentioned in Chapter 0 will not create a working app, but rather serve as a reference and a device for conceptual learning.

## How does a web page actually get to me?

Let's talk about web development in a very high level manner. Basically, when you visit a website in your browser, you are making a **HTTP GET** request to a web server somewhere that is connected to the internet.

### What is the HTTP Protocol?

HTTP is a protocol that allows for the fetching of resources as indicated by the recipient through requests. There is no link that is created, meaning that it is stateless. It also allows for extensibility with HTTP headers allowing for a simple agreement between a client and a server to create new header semantics.

There are various HTTP methods used for various purposes.

---

HTTP Methods	What does it do?
GET	Requests data from a specified resource
POST	Sends data to a server to update or create something
PUT	Sends data to a server to update or create something, but is <i>idempotent</i> (calling the same PUT request will always produce the same result, but multiple POST requests may produce duplicate results)
HEAD	Pretty much identical to GET, but does not include the response body (does not return the contents of the response - often used for checking what a GET request will return before actually downloading it)
DELETE	Deletes the specified resources
PATCH	Update/Changes a resource. PATCH requests only contain the changes to the resource, not the entire resource. Thus, they should be in a patch language (JSON Patch, XML Patch, etc.) (not safe or idempotent on its own)
OPTIONS	Describes/Outlines the communication options for the resource that is being targeted.

---

These HTTP methods all have their own response codes which we'll talk about later. Also be aware that these methods all have different behaviors when cached, reloaded, called upon via the user pressing the back button, etc.

Now, visit a website and open up your developer tools panel (this book will reference the Chrome Development Tools) and click on the 'Network' tab. Upon refreshing or loading the page, you will see everything that the browser has requested. Click one of these requests to see more information about the request and response. Here you can see information such as the Content-Type of the page which indicates the format and type of the document returned.

If you take a look at the response tab, you can see what was actually returned from the server from that GET request.

Further along the requests, you may see references to images that may have been loaded after receiving a HTML response with tags (such as the tag) telling the browser to request other resources. Upon receiving an initial response from the server, a browser will make additional requests to load any additional resources that may be specified in the HTML.

## Delving into Dynamic Web Pages

Here we'll take a look at a code snippet for a dynamic web page that makes use of the Express framework for Node.js. We'll talk more about this later, so it's okay if you have a vague idea of what's going on here.

```
const getHtml = (listLength) => {
  return `
    <!DOCTYPE html>
    <html>
      <head>
      </head>
      <body>
        <div class='container'>
          <h1>This is an example</h1>
          <p>number of notes created ${listLength}</p>
        </div>
      </body>
    </html>
  `;
};

app.get("/", (req, res) => {
  const page = getHtml(list.length);
  res.send(page);
});
```

Here is an example of a dynamic web page that we generate on the server side. We take advantage of this fact by passing the page some data to display that we would not be able to do traditionally with a static file. Upon a request to the server, the server will take note of the length of a list stored in memory and return a HTML file with the value inserted in place of `${listLength}`.

```
<p>Number of notes created ${listLength}</p>
```

Important Note: For management purposes, it is typically not a good idea to write HTML in the middle of code.

## Moving Application Logic to the Client

We don't have to run everything on the server and return a "static" HTML file. We can also incorporate client-side logic in the form of JavaScript files that make

requests to the server and do their own processing.

We can mention a reference to a JavaScript file in our HTML in order to execute it and add client-side logic to our application.

```
<script type="text/javascript" src="pathToFile.js"></script>
```

To dynamically load content in our application from the client side, we can take advantage of the XMLHttpRequest API based on HTTP which is primarily used to exchange data between a client and a service. The more modern Fetch API with a more powerful feature set will be discussed later after we gain our initial bearings.

**With code blocks such as these, please read along the code comments to follow what is going on**

```
var xhttp = new XMLHttpRequest()

xhttp.onreadystatechange = function() {
  if (this.readyState == 4 && this.status == 200) { // Checks if the operation is complete
    const data = JSON.parse(this.responseText) // Parses the body of the response
    console.log(data)

    var todoList = document.createElement('ul') // Creates an unordered list
    todoList.setAttribute('id', 'list') // Adds the class 'unordered' to the unordered list

    data.forEach(function(item) { // Creates a list item, adds the text content of each data
      var li = document.createElement('li')

      todoList.appendChild(li)
      li.appendChild(document.createTextNode(item.content))
    })

    document.getElementById('list').appendChild(todoList)
  }
}

xhttp.open('GET', '/data.json', true) // Defines a GET request to the serverAddress + /data
xhttp.send() // Sends the request
```

Now, you might have noticed that we write the code for handling the response before we even send the request for the response. This is because

```
xhttp.onreadystatechange = function () {
  // Anything in here
};
```

is an event handler defined on the xhttp object involved with the request that is called when the state of the object changes. Thus, if we assign a function to this event handler, it will be called when the state of this object changes.

When the request is completed, the state changes and calls this function. From within the function, we check if `readyState == 4 && this.status == 200` to see if the operation is complete (readyState 4) and if the HTTP status code of the response is 200 (HTTP Status OK).

**This style of invoking event handlers is something that you will see a lot of in JavaScript.** The functions that are assigned to these event handlers are called ‘**callback functions**’ where the code that you write does not invoke the function but rather the runtime environment (the browser in this case) will invoke the function when the event has occurred.

## HTML Pages Expanded

**Document Object Model** The Document Object Model, or DOM for short, is an API that allows for the editing of element trees programatically. What this means is that you can edit your page (like how we did in the above example) through the use of JavaScript by traversing the element tree that is a web page.

To have a glimpse at this tree-like structure, take a look at the ‘Elements’ tab in your console. Here you can see the various elements of a page nested within other elements, with the ‘branches’ extending out from their parents. This concept allows us to traverse from levels of branches from any point on the tree.

You can also create elements and add them to the DOM and other objects in the DOM as we did earlier.

```
var todoList = document.createElement('ul')
var li = document.createElement('li')
var li2 = document.createElement('li')
todoList.appendChild(li)
todoList.appendChild(li2)
```

We can also select elements programatically.

```
li.setAttribute("class", "list-item");
todoList.setAttribute("id", "list");

document.getElementById("list").innerHTML = "example";
document.getElementsByClassName("list-item")[0].innerHTML = "example";
```

Here, `getElementById()` returns the one element with that ID of an element’s child elements. The `document` object is the topmost node in the hierarchy of the DOM tree. As we are looking in the document’s child elements, we are looking through all elements.

`getElementsByClassName()` returns a collection of an element’s child elements with the specified class name. In the above example, we adjust the `innerHTML` of the first and only element in that collection. `innerHTML` relates to the HTML content of an element (the content within element tags).



## Styling with CSS

In order to provide our web page styling information, we can take advantage of Cascading Style Sheets (CSS for short) which allow us to define the appearance of web pages using a markup language.

Within the head element of the HTML code, we can create a `<link>` tag to fetch a CSS file from a particular address as such.

```
<link rel="stylesheet" type="text/css" href="main.css" />
```

Now, this element refers to a file called “main.css” in the same directory as this HTML file.

Within “main.css” we can write the following to define some styling information.

```
.container {  
  padding: 5em;  
  border: 1px solid;  
}  
  
.todoList {  
  color: red;  
}  
  
#list {  
  color: blue;  
}
```

Here we define the styling information for two classes that we can specify as an attribute to an HTML element. A class selector definition in CSS always starts with a period and contains the actual name of the class.

You can write CSS rules to apply styling information to elements that contain these classes. Similarly, to use ID selectors, you would write `#class-name` as seen above with the ‘list’ ID.

### ###Combining CSS, HTML, and JavaScript

When you include CSS links and JavaScript files that make requests to the server, the browser will roughly follow this order (assuming the web application is laid out similarly to the one described above).

1. Browser fetches the HTML for the application through a HTTP GET request
2. Browser realizes that there is a link to a CSS file and a link to a JavaScript file
3. Browser fetches the CSS file
4. Browser fetches the JavaScript file
5. Browser executes the JavaScript code which has XMLHttpRequest object that makes a HTTP GET request in an attempt to retrieve JSON data.

6. Browser receives the JSON data to which the browser executes the event handler dealing with the XMLHttpRequest state change.
7. The callback function referenced by the event handler utilizes the DOM API in order to add and render the list.

## Submitting New Data Using HTTP POST and Forms

We can use HTML form elements in order to submit data to a server using the POST method. The syntax for doing so is as follows.

```
<form action="/new_todo" method="POST">
  <input type="text" name="todo" />
  <br />
  <input type="submit" value="Save" />
</form>
```

Inserting this in our page gives us a text box and a “Save” button that allows the user to type in a todo and save it. Please note how the form tag takes in the attributes of `action` and `method` allowing us to specify the HTTP method and the address to point to.

When the button is clicked, the browser will send the values in the inputs (the textbox) to the server using a HTTP POST method directed at the “/new\_todo” endpoint.

## ADD IMAGE HERE OF REQUESTS

Now, if you take a look at the Network tab when you submit this form, you might be surprised that it causes five HTTP requests. The reason for this being that the first event is the HTTP POST request to the server with the data to be submitted to which the server responds with the 302 HTTP status code that describes a URL redirect. This URL redirect tells the browser to do a new HTTP GET request to the address mentioned in the header of the response; as a result, the browser redirects to the location and loads the page bringing along more HTTP GET requests for each of the HTML, CSS, JavaScript, and raw data files (data.json that is fetched as part of the client side logic).

Now, how might we handle this HTTP POST request on the server side of things?

On the server, we would write a few lines of code to handle this route.

```
app.post("/new_todo", (req, res) => {
  todos.push({
    content: req.body.todo,
    date: new Date(),
  });
});
```

```
    return res.redirect("/todos");  
  });
```

Here, we handle a POST request to the `/new_todo` route. From this, we create a new todo object that contains the content of the object and the date. We then push this object to the todos array. After this is done, we put an instruction to redirect to `/todos` in the response.

Saving objects to memory does not permanently store it. A simple restart of the web server will cause everything to be lost as it is not saved to a database. We'll discuss how to save objects to a database later in this book.

## Modern Conventions

Over time, these once traditional methods of how we just dealt with AJAX and our submission form became unacceptable. We have evolved from doing everything on the server side with HTML-code being generated entirely by the server to moving to what is called a Single Page App (SPA) with ever more logic being handled client side.

Traditional Web Apps (Fetching HTML from the web server) -> AJAX fetching data -> Modern APIs following a RESTful approach (We'll talk about this later in the book)

The example application that we just went through, by modern conventions, should not have fetched JSON data from a URL, but we'll talk about this more in depth later on. The application also utilized the traditional way of submitting a form to send data to our server. We can get closer to our goal of a SPA by changing a few things around.

## Adjustments

In order to prevent a redirect (against the goal of having a single-page app), we can write our form tag without any action or method attributes as follows.

```
<form id="todos_form">  
  <input type="text" name="todo" />  
  <br />  
  <input type="text" value="Save" />  
</form>
```

Here, you'll notice that we got rid of the action and method attributes while also adding an ID to the form so that we can programatically select it using the DOM API.

Now, to send the form data the "SPA way", we must write some more JavaScript on the client side.

```
var form = document.getElementById("todos_form");  
form.onsubmit = function (e) {
```

```

e.preventDefault(); // Overrides the default behavior of some browsers that may lead to a

var todo = {
  content: e.target.elements[0].value, // Gets the first input of the form and it's value
  date: new Date(),
};

todos.push(todo); // Adds the todo to the todos list so we can use it to rerender the page
e.target.elements[0].value = ""; // Clears the form text box
redrawTodos(); // Function that will cause all of the todos to be rerendered
sendToServer(todo); // Function that will send the new todo to be sent to the server using
};

```

Notice how we are creating functions that provide reusable functionality to make our code easier to maintain and read, especially as our projects continue to grow. Now, we'll go over the implementation of the `sendToServer()` method.

```

var sendToServer = function (todo) {
  var xhttpForPost = new XMLHttpRequest();

  xhttpForPost.open("POST", "/new_todo", true);
  xhttpForPost.setRequestHeader("Content-type", "application/json");
  xhttpForPost.send(JSON.stringify(todo));
};

```

In this method, we create a new `XMLHttpRequest` and set its request header to match the type of data that we are sending. In this case, we are sending json objects, so we update the request header and use `JSON.stringify()` to convert our JavaScript `todo` object into a JSON string that can be sent to the server.

## JavaScript

This is just going to be a quick overview of some features of the JavaScript language and tips that will prove useful during development. This section will teach you everything that you will need to know to continue in this book, but by no means should be used to learn JavaScript. For the purpose of learning JavaScript, I heavily recommend the *You Don't Know JavaScript* book series.

### JavaScript Background

You should have experience with some language prior to going through this book; if that language was Java, great, you'll recognize some of the syntax. Do not mistake JavaScript as being similar to Java, however, as they are completely different in how they are used. Some may try to use JavaScript as if it were Java with regard to design patterns and features, but this is highly disapproved.

Another thing to know about JavaScript is that it is often transpiled down to

older versions as browsers may not support all of the new features in newer versions of JavaScript. This is accomplished by using transpilers such as Babel which transpile JavaScript written in newer versions to older versions that are more compatible. When we break into creating our first React project with `create-react-app`, Babel will automatically be configured to transpile our code.

JavaScript is widely known as a client-side language that enables interactivity on websites, but it can also run on anything from embedded machines to servers. Node.js, a JavaScript runtime environment based off of Chrome's V8 engine, allows for JavaScript to be executed outside of a browser.

## Variables

### Arrays

You can store an array in a const as follows

```
const exampleArray = [5, 0, 3];
```

If you would like to add an item to the array, you can use

```
exampleArray.push(6);
```

If you would like to execute a function for each item in an array, you can iterate through items in an array by using the `forEach()` function as follows

```
exampleArray.forEach((valueFromArray) => {  
  console.log(valueFromArray); // prints 4 lines containing 5, 0, 3, and 6  
});
```

Here, you can see something called an 'arrow function' being defined and passed to the `forEach()` function. Upon receiving this function as a parameter, `forEach()` will execute that function for each of the items in the array, making sure to pass the individual element (for the current iteration) as a parameter for arrow function that we passed to it.

### Immutability and Concat

Now, when we eventually visit React, you'll soon find that a common feature is the use of immutable data structures. Thus, if we were to ever need to add another item to an array, we wouldn't add it to the same array reference but rather create a new array with our new value by using the `concat()` function.

```
const newArray = exampleArray.concat(5);  
console.log(exampleArray); // [5, 0, 3] is printed  
console.log(newArray); // [5, 0, 3, 5] is printed
```

Here, you can see how using the `concat()` method did not mutate our original array in any way but rather created and returned a new array with the function argument added on to it.

## The Map Expression

Expanding on this concept of immutability, when we want to modify or pass every value from an array into a function, the `map()` function (which will be used quite extensively in React) returns the results of an iterable (array, list, etc.) after applying a given function. What this means for our arrays is that we can create completely new arrays where the function given as a parameter is used to create them based off of our original array. This is done without mutating our original array as it returns a completely new array.

```
const exampleMap = [5, 3, 1];

const newExample = exampleMap.map((value) => value * 2);
console.log(newExample); // [10, 6, 2] is printed
```

## Destructuring Assignment Expression

You can assign individual items in an array to variables by using **destructuring assignment**.

```
const toBeDeconstructed = [5, 6, 7, 8, 9];

const [firstItem, secondItem, ...everythingElse] = toBeDeconstructed;

console.log(firstItem, secondItem); // 5, 6 is printed
console.log(everythingElse); // [7, 8, 9] is printed
```

Here, you can see that the variables `firstItem` and `secondItem` receive the first two integers of the array. Then the rest of the values get collected into a new array which is then assigned to the variable `everythingElse`. You can use any variable name and any amount of variables to get the values that you desire from the array.

## Objects and JavaScript Object Notation (JSON)

JavaScript contains objects that follow the JavaScript object notation. One way to create these objects is by using *object literals* in which you literally write the object in your code as follows

```
const examplePersonObject = {
  name: {
    first: "ExampleFirst",
    last: "ExampleLast",
  },
  jobs: ["Teacher", "Software Engineer"],
  workplace: "Microsoft",
};
```

Within an object you have things called properties. These are essentially the key values that you would use in order to get the values of the properties. The syntax is, as you can see above, **property: value**. The values of these properties can be anything ranging from integers to full blown objects.

## Referencing Properties

In order to use values within objects, we can use dot notation or brackets to reference these properties and receive these values back.

```
console.log(examplePersonObject.workplace); // Using dot notation to reference the property,
console.log(examplePersonObject["workplace"]); // Using brackets to reference the property,
```

With brackets, we also gain the ability to reference properties based off of a variable instead of having to rely on string literals or hardcoded properties.

```
const exampleField = "workplace";
console.log(examplePersonObject[exampleField]); // Using bracket notation and a variable to
```

In addition, we can use the concepts from above to dive deeper into our nested object. Recall that our **name** property within our **examplePersonObject** object has an object of its own. We can access properties of that object as follows

```
const requestedName = "last";
console.log(examplePersonObject["name"][requestedName]); // Using bracket notation and a variable
console.log(examplePersonObject.name.first); // Using 'stacked' dot notation, we reference
```

Here, you'll notice that we can 'stack' brackets or the use of dot notation multiple times to delve within objects. It may also be helpful to think of your objects as any file structure that you would find on your computer and the dot notation/brackets as the file path.

Please note, you *cannot* use variables with dot notation as of ES9. You also cannot reference properties with spaces or create new properties with spaces in their names when using dot notation as in the following example.

## Adding Properties

We can also add properties to any object after the fact by using brackets or dot notation as well.

```
examplePersonObject.age = 204;
examplePersonObject["favorite color"] = "Blue";
```

Here, you'll notice, as mentioned earlier, we must use brackets to create the 'favorite color' property as the name has a space in it.

Whenever we encounter these odd cases for property names, we can use string syntax when assigning our object literals as follows.

```
const exampleObject2 = {
  "some random property": 25,
};
```

Here, you'll notice that we wrapped the property name in quotes in order to accomplish this.

Do not use awful, nondescript property names as the one illustrated above.

## Object Methods

You can also create methods for objects as well. You can simply add a function as the value to a property.

```
exampleObject2.exampleFunction = function () {
  console.log("I'm called inside an object method!")
}

const exampleObject3 = {
  name: 'Some Random Value'
  bindDemo: function(e) {
    console.log(this.name)
  }
  talk: function(e) {
    console.log("Called from within an object with the argument: " + e)
  }
}
```

Now, to call these functions, we can use the usual dot notation or brackets, but we can also store a method reference in a variable and call that method through the variable as follows

```
const referenceToTalk = exampleObject3.talk;
referenceToTalk("Example"); // Prints "Called from within an object with the argument: Example"
```

If you use the `this` keyword in any of your object methods, please note that `this` is defined based on how the method is called with these regular functions (explained in further detail below in the 'JavaScript Functions' section). Thus, if you call the function via reference, `this` becomes the global object and can lead to many errors.

For example, if we tried calling the `bindDemo` method from a reference, we will get an error

```
const referenceToBindDemo = exampleObject3.bindDemo;
referenceToBindDemo(); // Error printed as there is no property in the global object
```

We can create new functions with `this` bound manually regardless of what is calling the method.



```
const boundFunction = exampleObject3.bindDemo.bind(exampleObject3); // Bound to exampleObject3
```

With this, it is important to always keep track of **this** when writing JavaScript as it can lead to a plethora of issues.

## INSERT INFORMATION ABOUT OBJECT METHODS IN JAVASCRIPT

### Constructor Functions

The use of a constructor may remind you of classes coming from a Java background or any of the many other languages that make use of this mechanism. I wouldn't get my hopes up too much, though, as JavaScript doesn't really have classes that behave in the same way as these other languages.

We can write constructors as follows

```
function Person(first, last, age, occupation) {  
  this.firstName = first;  
  this.lastName = last;  
  this.age = age;  
  this.occupation = occupation;  
  this.changeAge = function (age) {  
    this.age = age;  
  };  
}
```

Here, you can see the constructor function for the creation of a **Person** object for which we can use to create many objects of the **Person** "type". You'll also notice the use of a keyword **this** which is a substitute for the new object which is to be created. Upon the creation of this object, this binds itself to that object and the appropriate properties are created.

We can also add object methods to our constructor which will be applied to the objects that are created. You'll also notice how we use **this** to reference the object that we created using the constructor so that we can just call **changeAge()** in order to update a given **Person** object's age.

It is recommended to name constructor functions with an upper-case first letter.

```
var examplePerson = new Person("Jake", "Flannn", 25, "Software Engineer");
```

The above is how we would go about creating a object of the **Person** "type". If we wanted to go ahead and change the age of our **Person** object, we can call the object method stored in one of its properties.

```
examplePerson.changeAge(25);
```

Here, the object method mentioned in the constructor has bound **this** to the **examplePerson** object, to which we are able to reference the object's age by

`this.age` in the method and update it.

## Class Syntax

With version ES6 came the introduction of the *class syntax* which can help us structure objects and object-oriented classes. Now, this is where things might get familiar to those with a background in languages such as Java. Please keep in mind, although they behave similarly to Java objects, at their core, they are still JavaScript objects that are based on prototype inheritance.

**Class Declarations** We can define a class by using a *class declaration* as follows

```
class Person {
  constructor(firstName, lastName, age, occupation) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
    this.occupation = occupation;
  }
}
```

There can only be one `constructor` method for a class. When dealing with JavaScript inheritance and subclasses, you can use the `super` keyword to call the constructor of the super class. Additionally, you can not add a new property to an existing object constructor outside of the constructor function.

Also, please be sure to mention class declarations in your code before you actually attempt to create instances of these classes as class declarations are not hoisted unlike function declarations.

If you care to read more about hoisting, I suggest you take a look at the ‘JavaScript Nuances and Tips’ section at the end of the chapter.

The body of a class is automatically put into “strict mode” for increased performance and safer guidelines (see ‘JavaScript Nuances and Tips’ for an in-depth explanation of what it does).

**Class Expressions** Class expressions are also another way to define a class in JavaScript versions past ES2015.

```
var Square = class {
  constructor(dimension) {
    this.height = dimension;
    this.width = dimension;
  }
  area() {
    return this.height * this.width;
  }
}
```

```
};
```

```
console.log(new Square(5).area()); // Prints 25
```

You'll notice that we omitted the class name here, which you cannot do with class declarations. The constructor property is also optional for these classes with redeclaration being possible.

As mentioned earlier, the body of a class is automatically put into “strict mode”.

**Static Methods in Classes** When using the `static` keyword to define a method for a class, we create a static method which can be called without instantiating the class (creating an instance of the class/the object created after using the constructor to create a new object). These methods **cannot** be called by class instances.

```
class Example {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
  static add(a, b) {
    return a + b;
  }
}
const exampleInstance = new Example(5, 5);
console.log(Example.add(5, 5)); // Prints 10
```

Notice how we called the `add()` static method from the actual class declaration rather than the `exampleInstance` instance that we have instantiated from the class. Static functions are used for utility functions that do not depend on properties of specific instances.

There are many more special behaviors with JavaScript classes, but this should suffice for most web development purposes. When the time comes to learn JavaScript behaviors, features, and syntax more deeply, it is recommended to read the ‘MDN web docs’ or the ‘You Don’t Know JavaScript’ book series.

## JAVASCRIPT PROTOTYPES

Every object in JavaScript has an internal property known as ‘prototype’

### JavaScript Functions

Now let’s talk about one of the most important concepts that you’ll find in JavaScript when dealing with web development, especially when dealing with React.

Nowadays, the most common way to define functions is by defining an arrow function. Arrow functions can be written as follows

```
const add = (a, b) => {  
  return a + b;  
};  
  
// To call our function  
const sum = add(5, 5);  
console.log(sum); // Prints 10
```

There are also shortcuts that we can take. If we just have one parameter that we need for our function, we can get rid of the parentheses around our parameters as follows

```
const square = (d) => {  
  return d * d;  
};
```

Going even further, if there's just one expression inside the body of our function, we can keep everything in one line and get rid of the brackets.

```
const square = (d) => d * d;
```

This shortened syntax often finds its use when dealing with the map method as it looks quite clean and can fit everything into one line.

```
const toBeSquared = [5, 2, 3];  
const squaredNumbers = toBeSquared.map((d) => d * d);
```

Arrow functions behave differently than regular functions. For example, they bind the keyword `this` to the lexical scope instead of binding it based on context. Usually, regular functions bind `this` to the object that calls the function (the context), while arrow functions take the value of `this` in the scope that the function was defined. In other words, arrow functions bind the value of `this` to the value of `this` where it was defined. This makes them unsuitable for being used as an object method.

The differences between arrow functions and regular functions is something that you should definitely read up on later in your career, but for now, just go with the general rule that you must use function expressions for object methods while arrow functions are often better when dealing with many callbacks or methods such as `map()`, `reduce()` or `forEach()`.

## Function Expressions and Declarations (Regular Functions)

Before arrow functions, there were function expressions and function declarations. When we give a regular function a name, it is called a function declaration.

```
function add(a, b) {  
  return a + b;  
}
```

```
}  
const sum = add(2, 9);
```

When we do not give a function a name, it is called a function expression:

```
const add = function (a, b) {  
    return a + b;  
};  
const sum = add(2, 9);
```

Please do not worry about remembering all of this, we will revisit most relevant concepts later on when we encounter their actual real-world uses.

## JavaScript Nuances and Tips

### Using Strict Mode

With the release of ES5 came the introduction of a “strict mode” to the language. In general, the restrictions that this strict mode imposes keeps the code adhering to a set of safer guidelines and allows for further optimizations to be made by the JavaScript engine.

You can opt some or all of your code in to strict mode by writing

```
"use strict";
```

“Strict mode” follows function scope, so you can declare it for a specific function or the entire global scope.

It is recommended to enable strict mode on the global scope as they solve common problems such as implicit auto-global variable declaration when you omit the ‘var’ declaration. Turning on strict mode might lead to encountering more errors or buggy behavior of code, but if this occurs, that is almost always a sign that there is something wrong with your program, thus you should keep strict mode on in most cases.

### Closures

To explain closures as simply as possible:

When defining a function inside another function, that inner function is able to access variables declared in the outer function. Thus, you can share variables between function calls without making the variables global. Then, when you call the inner function, it will have access to the variables defined in the outer function.

If you look online for some explanations, they will explain the technical reasons as to why this is in JavaScript, but if you just want a simple explanation for now, this will do for this book .

## Hoisting

## Spread Syntax

## Let's React

Now, there are a variety of JavaScript libraries for building user interfaces. React is just the most popular (as of writing) with a massive community and many supporters. To get started, we're going to make our first simple app using React as as the “view” layer of our web application.

The easiest way to get started is by using a tool called **create-react-app** which you can install using npm, the default package manager of Node. If you do not already have Node installed, feel free to install that now.

Please note: I did not say that **create-react-app** was the best way to get started. The reason for this being that **create-react-app** is quite bloated with a huge bundle size and is quite messy when ejecting. This is because ‘create-react-app’ comes with many tools as a one-size-fits-all approach.

To create your first React application, run the following commands in a terminal.

Make sure not to write the \$ in your commands as that simply denotes that the following line is to be typed into a terminal.

```
$ npx create-react-app project1
$ cd project1
```

This will create your app in a directory called project1 and configure it accordingly. Then, you must change your working directory to this new “project1” directory with the `cd` command. After this, you can finally start your application by running

```
$ npm start
```

Upon running this command, the application should run at the address `https://localhost:3000`

Chrome should have opened automatically, so you should now open your developer tools so we can get started.

For all intents and purposes with regard to learning React, we can simplify our `index.js` file and our project directory quite significantly.

Please replace your `index.js` file with the contents below.

```
import React from "react";
import ReactDOM from "react-dom";

const App = () => (
  <div>
```

```

    <p>My first React app!</p>
  </div>
);

```

```
ReactDOM.render(<App />, document.getElementById("root"));
```

You can also go ahead and delete `App.js`, `App.css`, `App.test.js`, `logo.svg`, and `serviceWorker.js` as they are not needed for this project.

## Components

Within our `index.js` file, you'll find an item called a React-component with the name of `App`.

There is also a line that renders this component into view at the bottom of the file as follows

```
ReactDOM.render(<App />, document.getElementById("root"));
```

This line renders the component into the `div`-element with the `id` of 'root' that was specified in the `public/index.html` file.

Now, you can always write standard HTML into that file, and it will appear, but with React applications, we try to write everything defined as React components. Who would've known?

Let's take a dive into our `App` component. Looking at the code, you can see that the component places a `<p>` tag containing the text 'My first React app!' inside a `div` tag. Moving on to the JavaScript aspect, the component is an anonymous JavaScript function (a function that is not bound to an identifier or more simply, a function that does not have a name) with no parameters. This function is then stored inside the constant variable `App`.

This function syntax also follows something new that was introduced in ECMAScript 6 (ES6), arrow functions. We have also used a shorthand that returns the value of the expression within the parentheses,

`const App = () => ()`, where the evaluation of the parentheses will be returned.

Omitting the shorthand, we can write the same block as

```

const App = () => {
  return (
    <div>
      <p>My first React app!</p>
    </div>
  );
};

```

Where we now explicitly return the component that we want to render. Upon doing this, we can now render dynamic content as we can execute any JavaScript statement before the return statement.

For example, we can now write

```
const App = () => {
  const now = new Date();
  const a = 50;
  const b = 50;

  return (
    <div>
      <p>My first React clock! It is {now.toString()}</p>
      <p>
        {a} plus {b} is {a + b}
      </p>
    </div>
  );
};
```

Here, you can see us execute JavaScript statements before the return statement which allows us to store values within variables to be referenced in our component. We reference these variables in our component by enclosing them within curly braces. In our return statement, we can put JavaScript code within curly braces which will be evaluated and inserted into our component.

Now, you might have noticed that the code that we are writing in our return statements looks like an awful lot like HTML markup. In reality, we are dealing with straight JavaScript using JSX. This JSX that will be returned will be returned as pure JavaScript. Now, this compiling from JSX to JS is done by something called Babel and is automatically set up when you use **create-react-app**. We will talk about this more later in the book once we dive into how everything works.

With JSX being compiled to JS anyway, you'll notice that you can write React using pure JavaScript without using JSX, but this is not recommended as JSX simplifies things dramatically with its declarative style and more.

This compiled JSX will look like

```
var App = function App() {
  var now = new Date();
  var a = 50;
  var b = 50;
  return React.createElement(
    "div",
    null,
    React.createElement(
```



```

    "p",
    null,
    "My first React clock! It is ",
    now.toString()
  ),
  React.createElement("p", null, a, " plus ", b, " is ", a + b)
);
};

```

in pure JavaScript. If you were to write this component in pure JavaScript, you would have to write this.

JSX isn't exactly like HTML though, it is "XML-like", meaning that every tag written needs a closing tag.

For example, when writing HTML, you can write a line break element as follows

```
<br />
```

However, when writing JSX, you must have a closing tag as follows

```
<br />
```

or when dealing with components that may require items between the starting and closing tags

```
<h1>I need a closing tag!</h1>
```

**As you begin to write your own components, please note that a React-component must have a name that starts with a capital letter.** The reason for this being that the component use will be treated as a built-in element (such as a span or div) if the component does not start with a capital letter.

## Adding Complexity

We can include components inside components, allowing us to increase modularity and maintainability across our components.

```

const ExampleComponent = () => {
  return (
    <div>
      <p>I will be used inside another component!</p>
    </div>
  );
};

const App = () => {
  return (
    <div>
      <h1>Here's my React App!</h1>
      <ExampleComponent />
    </div>
  );
};

```

```

        <ExampleComponent />
        <ExampleComponent />
      </div>
    );
  };

```

```
ReactDOM.render(<App />, document.getElementById("root"));
```

Here, you can see that we mentioned `<ExampleComponent />` multiple times in our `App` component. This allows us to break our project into many components designed for a purpose that can be reused in other components, allowing us to keep our project maintainable as it grows larger.

## Components With Data (Props)

Now, let's combine the concept of using JavaScript in our JSX and the modular structure of React. Doing so requires the use of *props* which allow us to pass data to components.

For example, we can create the following components that pass data to each other and use it in a meaningful way.

```

const Welcome = (props) => {
  return (
    <div>
      <p>
        Welcome, {props.title}
        {props.name}!
      </p>
    </div>
  );
};

const App = () => {
  const josephTitle = "Mr. ";
  const josephAge = 37;

  return (
    <div>
      <h1>Members: </h1>
      <Welcome name="Grace" title="Ms. " age={5 + 20} />
      <Welcome name="Joseph" title={josephTitle} age={josephAge} />
    </div>
  );
};

```

Here, you can see that we make use of the `Welcome` component twice in our `App` component. Within our calls to the `Welcome` component, we pass in the

prop `name` as a string parameter. We can also pass the result of a JavaScript expression by wrapping it in curly braces as you can see with our references to the `josephTitle` and `josephAge` variables as well as our evaluation of the expression `5+20`. When you take a look at our `Welcome` component, you can see how we reference prop values via dot notation.

When writing your React components, keep in mind that the content of a React component needs to contain one root element or return an array of components. You cannot write the `App` component as follows

```
const App = () => {
  const josephTitle = 'Mr. '
  const josephAge = 37

  return (
    <h1>Members: </h1>
    <Welcome name="Grace" title="Ms. " age={5+20}/>
    <Welcome name="Joseph" title={josephTitle} age={josephAge}/>
  )
}
```

Here, without the `<div>` tag wrapping the three inner components/tags, we will get an error message relating to our code being unable to be compiled.

... Parsing error: Adjacent JSX elements must be wrapped in an enclosing tag. Did you want a JSX fragment `<>...</>`?

This is because we are no longer only returning the root `<div>` tag, but are instead trying to return 3 adjacent JSX elements.

As we have discussed earlier, we can also return an array of components as follows

```
const App = () => {
  const josephTitle = 'Mr. '
  const josephAge = 37

  return [
    <h1>Members: </h1>
    <Welcome name="Grace" title="Ms. " age={5+20}/>
    <Welcome name="Joseph" title={josephTitle} age={josephAge}/>
  ]
}
```

This should be avoided, however, as it is quite unsightly and may violate some style guides.

With our first solution of wrapping multiple JSX elements in one enclosing tag, you might notice that it creates extraneous `div`-elements that pollute our

DOM-tree. We can avoid this by using ‘fragments’ that were mentioned in the error message above.

```
const App = () => {
  const josephTitle = "Mr. ";
  const josephAge = 37;

  return (
    <>
      <h1>Members: </h1>
      <Welcome name="Grace" title="Ms. " age={5 + 20} />
      <Welcome name="Joseph" title={josephTitle} age={josephAge} />
    </>
  );
};
```

Here, the fragment’s opening tag is `<>` with `</>` as the closing tag. Fragments are essentially empty elements that we can use to wrap the elements to be returned by the component. Doing this allows us to compile our code successfully without making it unsightly or adding extra div-elements to our DOM-tree.

## Dealing With Component State

Let’s first start off by writing a component that greets a person with text inside a div.

```
const Greet = (props) => {
  return (
    <div>
      <p>
        Welcome {props.name}, we have your age as: {props.age}
      </p>
    </div>
  );
};

const App = () => {
  const exampleName = "Jake";
  const exampleAge = 15;

  return (
    <div>
      <Greet name="Jason" age={36 + 10} />
      <Greet name={exampleName} age={exampleAge} />
    </div>
  );
};
```

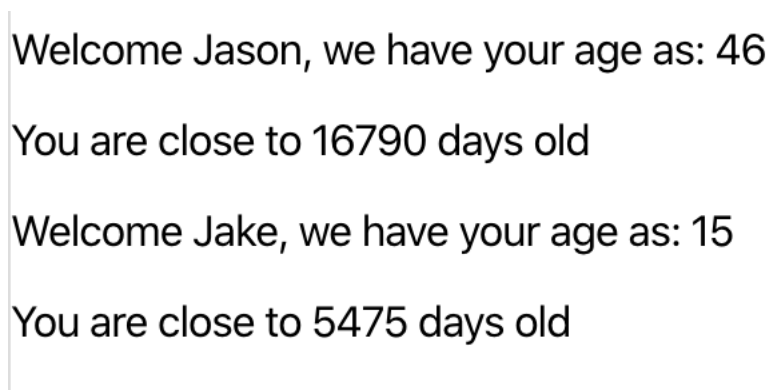
## Component Helper Functions

Now, if we wanted to write a component that also wrote about how many days the person has been alive based off of their age, we could make use of what is called a **component helper function**.

```
const Greet = (props) => {  
  const ageDays = () => {  
    return props.age * 365;  
  };  
  
  return (  
    <div>  
      <p>  
        Welcome {props.name}, we have your age as: {props.age}  
      </p>  
      <p>You are close to {ageDays()} days old</p>  
    </div>  
  );  
};
```

As the function is defined within the component function, it has access to all the props passed to the components. Thus, we do not need to pass the age prop to the function. We then use the function that we defined within the **Greet** component in order to get, roughly, the amount of days that the user has been alive for.

The result of this component should look like this:



Welcome Jason, we have your age as: 46

You are close to 16790 days old

Welcome Jake, we have your age as: 15

You are close to 5475 days old

Figure 1: image-20190915002753027

## Destructuring in JavaScript

We can streamline our references to our props by using something called destructuring to extract the values of an object's properties into separate variables

without needing to reference them via brackets or dot notation.

```
const Greet = (props) => {
  const { name, age } = props; // Assigns the name property to the variable 'name' and the

  const ageDays = () => {
    return age * 365;
  };

  return (
    <div>
      <p>
        Welcome {name}, we have your age as: {age}
      </p>
      <p>You are close to {ageDays()} days old</p>
    </div>
  );
};
```

Here, you'll see that we no longer have to reference these properties through dot notation/brackets upon the object, we can just simply reference the properties by their names as they have been assigned to constants when they were destructured.

In order to further streamline this, we can destructure the props into the variables in the argument list when we define our arrow function.

```
const Greet = ({ name, age }) => {
  // The props are destructured in the argument list
  const ageDays = () => {
    return age * 365;
  };

  return (
    <div>
      <p>
        Welcome {name}, we have your age as: {age}
      </p>
      <p>You are close to {ageDays()} days old</p>
    </div>
  );
};
```

Here, instead of passing the entire `props` object into the component function, we destructure the props into the two variables `name` and `age` which are then passed into our component without passing in the entire `props` object. Thus, any references to the `props` object will lead to a 'Failed to compile' error. With this, it is important to remember that you will not be able to reference the `props` object and will have to destructure any new props that you pass into your

component.

## Component Re-Renders

Let's bring some interactivity into our components.

```
const App = (props) => {
  const { count } = props;
  return (
    <div>
      <p>The current count is: {count}</p>
    </div>
  );
};

let count = 1;

const refresh = () => {
  ReactDOM.render(<App count={count} />, document.getElementById("root"));
};

setInterval(() => {
  refresh();
  count += 1;
}, 1000);
```

Now, you'll see here that we have a function called **refresh** that will call upon **ReactDOM.render** to render our component to the root element in our document. Within this **refresh** function, we pass the current value of the variable **count** to our **App** component which will then be rendered and added to the root element.

Moving down to the bottom of our snippet, you'll also notice that we have a **setInterval** method that calls our **refresh** function and then adds 1 to the **count** variable every 1000 milliseconds (1 second). This effectively renders our component every second: first with the **count** variable as 1, second with the **count** variable as 2, and so on.

This actually isn't how things are typically done with React though, we shouldn't need to call **ReactDOM.render** in order to re-render components. We'll discuss better ways to re-render our components next.

## Stateful Components

React recently had the addition of 'React Hooks' in React 16.8. We'll now be taking advantage of React's state hook. React Hooks allow us to add state to function components without converting it to a class.

Let's rewrite our above app to make use of **useState**.

```
import React, { useState } from "react";
import ReactDOM from "react-dom";

const App = (props) => {
  const [count, setCounter] = useState(0);

  setTimeout(() => setCounter(count + 1), 1000);

  return (
    <div>
      <p>The current count is: {count}</p>
    </div>
  );
};
```

```
ReactDOM.render(<App />, document.getElementById("root"));
```

There are quite a few new concepts and changes to this application, so let's take a look into what has changed and what these new functions do.

You'll first notice that we imported the `useState` function from 'react' in order to take advantage of React's state hook.

Next, draw your attention to the following line

```
const [count, setCounter] = useState(0);
```

Here, we make use of the destructuring syntax to assign the elements of the array that `useState` returns to the constants `count` and `setCounter`. When you call `useState`, it creates this 'state variable' which will be preserved by React and not 'disappear' (due to the scope of our variables) when the function exists unlike normal variables. It also creates a function that we can use to update our state.

The first element in the array that is returned from the `useState` function call is the 'state variable'. The second element in the array that is returned from the call is our function that we can use to update our state. We then utilize array destructuring (the bracket syntax) in order to store the first element in the `count` constant and the update function `setCounter` constant.

One common point of confusion is simply just passing an expression to update the state in a `setTimeout` function or anywhere. You typically will have to create a function to change state and pass that to any function which will update your state. For example, in our application above, we wrote

```
setTimeout(() => setCounter(count + 1), 1000);
```

In this snippet, we call the `setTimeout` function and pass it an arrow function (with shortcut syntax) to increment the counter variable in our state and with a timeout duration of 1000 milliseconds. Every time this state modifying function



is called, React will go ahead and re-render the component. When dealing with functional components, the entire function's body will be re-executed upon a re-render. Upon calling `setCounter`, the `count` variable will be updated and we can reference it as such.

Just to provide some contrast, if we were to not use React Hooks and convert it to a class, our component would look something like this.

```
import React from "react";
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0,
    };
  }

  render() {
    return (
      <div>
        <p>The current count is {this.state.count}</p>
      </div>
    );
  }

  componentDidMount() {
    setInterval(() => {
      this.setState({
        count: this.state.count + 1,
      });
    }, 1000);
  }
}
```

Notice how things can be quite verbose and how we have to employ slightly different practices such as referring to our state variables with dot notation and `this.state`.

## Event Handlers

Event Handlers are properties of methods that manage how a particular element reacts to events. Upon an event, the handler (a function that is invoked by an event listener) will be executed.

In React, button elements have a variety of mouse events. We can use the click event to make something happen as a result of pressing a button as follows

```
class EventHandlerButton extends React.Component {
```

```
handleClick = () => {  
  console.log("A button has been clicked!");  
};  
  
render() {  
  return <button onClick={this.handleClick}>Click Me!</button>;  
}  
}
```