CREATING IONIC APPLICATIONS WITH STENCILJS

JOSHUA MORONY

Table of Contents

Example: Weather App

18. Lesson 1: Introduction & Requirements (Weather not included in preview) *(not included in preview)*
19. Lesson 2: Getting Ready (Weather not included in preview) *(not included in preview)*
20. Lesson 3: Creating the User Interface *(not included in preview)*
21. Lesson 4: Implementing Geolocation & Saving Data *(not included in preview)*
22. Lesson 5: Handling User Input & Settings *(not included in preview)*
23. Lesson 6: Integrating the Weather API *(not included in preview)*
24. Lesson 7: Improving User Experience and Styling *(not included in preview)*
25. Weather App Conclusion *(not included in preview)*

Reusable Web Components

26. Lesson 1: Why Build Reusable Web Components *(not included in preview)*
27. Lesson 2: Creating a Collection of Web Components *(not included in preview)*
28. Lesson 3: Publishing and Using a Collection of Web Components *(not included in preview)*

Example: Chat App

34. Lesson 1: Introduction & Requirements (Live Chat not included in preview) *(not included in preview)*
35. Lesson 2: An Introduction to Firebase and NoSQL *(not included in preview)*
36. Lesson 3: Getting Ready (Live Chat not included in preview) *(not included in preview)*
37. Lesson 4: Setting up Firebase *(not included in preview)*
38. Lesson 5: Implementing the User Interface *(not included in preview)*
39. Lesson 6: Adding Authentication *(not included in preview)*

40. Lesson 7: Protecting Routes *(not included in preview)*
41. Lesson 8: Creating and Reading Data with Firestore *(not included in preview)*
42. Lesson 9: Implementing Firestore Security Rules *(not included in preview)*
43. Lesson 10: Improving Styling *(not included in preview)*
44. Chat App Conclusion *(not included in preview)*

State Management

43. Lesson 1: What is State Management? *(not included in preview)*
44. Lesson 2: Stencil State Tunnel *(not included in preview)*
45. Lesson 3: An Introduction to Redux *(not included in preview)*

Example: Storefront App

51. Lesson 1: Introduction & Requirements (Storefront not included in preview) *(not included in preview)*
52. Lesson 2: Getting Ready *(not included in preview)*
53. Lesson 3: Implementing the User Interface *(not included in preview)*
54. Lesson 4: Integrating Redux *(not included in preview)*
55. Lesson 5: Loading Products *(not included in preview)*
56. Lesson 6: Implementing Searching and Filtering *(not included in preview)*
57. Lesson 7: Creating a Shopping Cart *(not included in preview)*
58. Lesson 8: Persisting State *(not included in preview)*
59. Lesson 9: Styling and User Experience *(not included in preview)*
60. Storefront App Conclusion *(not included in preview)*

Testing and Debugging

59. Testing & Debugging *(not included in preview)*

Building and Submitting

60. Preparing Assets *(not included in preview)*
61. Building for iOS and Distributing to the Apple App Store *(not included in preview)*
62. Building for Android and Distributing to Google Play *(not included in preview)*
63. Creating iOS Certificates on Windows *(not included in preview)*
64. Publishing as a PWA (Progressive Web Application) with Netlify *(not included in preview)*

Conclusion

66. Conclusion *(not included in preview)*

Basics

StencilJS & Ionic Basics

The first section of this book is dedicated to understanding the basic concepts and syntax you will need to build Ionic applications with StencilJS. The main goal is to introduce you to the basic theory, and then we will implement that theory with real-world examples later in the book. You are welcome to implement the code from the basics section in your own test application as we go if you like, but it is not required. Again, the main goal is just to start getting you familiar with the basic concepts.

Depending on your learning style, you may want to approach this section differently. It is rather long and contains a lot of theory (although there is a lot of theory to consume, it is important to understand). You may want to go through the Basics section in a linear fashion from start to finish before tackling any example applications, or you may prefer to read bits and pieces of the Basics section as required and instead jump ahead to the "fun" stuff before finishing all the theory. It is totally up to you, but if you do want to skip ahead, I would recommend reading the Basics section up until at least the lesson on **Navigation**.

We will walk through many examples and scenarios in the Basics section, and it will likely serve as a good point of reference for when you are building your own applications in the future.

Lesson 1: Basic Building Blocks

Before we even get into the basics of generating a new project and the various files and folders that an Ionic/StencilJS project contains, I wanted to cover a few important concepts.

Your projects will mostly consist of a bunch of different **web components**. Each individual web component will represent a small chunk of your application (like an individual page), but together the web components will power your entire application.

Since the vast majority of what you will be doing to build Ionic applications with StencilJS is creating these components, I think it is important to understand the basic building blocks of these components right out of the gate. If we were to be working on the component for the **home** page, the file that is responsible for generating the page/component would be:

- src/components/app-home/app-home.tsx

You will find some other files inside of **app-home**, but this is the important one. We are building applications with JavaScript, so you might expect that we would be working with an **app-home.js** file not an **app-home.tsx** file. There is a good chance you may not have even seen a **.tsx** file before.

We are going to get into all the specifics of this in just a moment, but first, let's take a look at what is actually in that file. A basic Ionic/StencilJS component might look something like this:

```
import { Component, h } from "@stencil/core";

@Component({
  tag: "app-home",
  styleUrl: "app-home.css"
})
export class AppHome {
  public firstName: string = "Josh";

  componentDidLoad() {
    // perform logic when the component loads
  }

  someMethod() {
    // a method that could be triggered by another function
    // or bound to some action (e.g. a click) in the template
  }

  render() {
    return [
      <ion-header>
        <ion-toolbar>
          <ion-title>Example</ion-title>
        </ion-toolbar>
      </ion-header>,

      <ion-content class="ion-padding">
        <p>Hi, {this.firstName}</p>
```

```
      </ion-content>
    ];
  }
}
```

All three important "building blocks" that we are going to discuss in this lesson and the next are present here: **Classes**, **JSX**, and **TypeScript**. We are going to discuss each of these in-depth, but to give you a quick over:

- **Classes** provide the structure/logic/methods/variables for our component
- **JSX** is the syntax we use to define our templates and display data
- **TypeScript** is an extension to JavaScript that allows us to add "types" to our code (and it also allows us to use newer JavaScript features that are not yet supported by browsers, by transpiling our "modern" code into code that today's browsers do understand)

This is where the `.tsx` extension comes from. We would typically use a `.js` extension for standard JavaScript, a `.jsx` extension for code that uses JSX, and a `.ts` extension for code that uses TypeScript. Therefore, if we are using JSX and TypeScript we use a `.tsx` extension.

In this lesson, we are going to focus on the concept of **Classes** and in the following lesson, we will cover **JSX** and **TypeScript**. Classes are a rather generic concept to programming in general, and depending on your previous experience (whether with JavaScript or another language) you may already be quite familiar with some of these concepts.

If you are already comfortably familiar with these concepts:

- Classes/Objects
- Scope and the `this` keyword
- Modules/Importing/Exporting

You can safely **skip** straight to the **Classes in Ionic/StencilJS** section in this lesson, which will give you a brief introduction to the basics of creating a component in StencilJS before moving on to the next lesson.

Classes

If you are not already familiar with classes, let's take a step back first and explain what a class is as a general programming concept, as it is not something that is specific to Ionic, StencilJS, or even JavaScript.

Classes are a concept from Object Oriented Programming (OOP) and they essentially behave as blueprints for creating "objects". An object would provide variables and methods/functions related to a particular "thing".

You can define a class, and then using that class you can create, or "instantiate", objects from it. In general, our program/code/application would consist of a bunch of different "objects". To use a non-programming analogy, we could have a class for generating humans. That class might specify properties like `eyeColor` and `height` and perhaps provide methods like `eat()` and `sleep()`. An object would be an individual instance of that class: a human. We might create a human object with `green` eyes and a height of `150cm`, but we could also create another individual human/object from that same class

with `brown` eyes and a height of `200cm`.

These analogies usually aren't all that great for explaining how classes are actually used in a programming context, but it does help to explain the general concept of classes/objects. In an Ionic/StencilJS context, our classes will generally be used to generate the components used within our application (but also sometimes other things like helpers/services).

If classes are a completely new concept to you, it would be worth doing a little bit of your own research before continuing, but let's take a look at a simple example.

```
class Person {

  constructor(name, age){
    this.name = name;
    this.age = age;
  }

  setAge(age){
    this.age = age;
    return true;
  }

  getAge(){
    return this.age;
  }

  setName(name){
```

```
    this.name = name;
    return true;
  }

  getName(){
    return this.name
  }

  canDrive(){
    return this.age > 16;
  }
}
```

This class defines a **Person** object. The `constructor` is run whenever we create an instance of this class (an object is an instance of a class), and it takes in two values: `name` and `age`. These values are used to set the **member variables** (or **class members**) of the class, which are `this.name` and `this.age`.

These values can be accessed from anywhere within the object by using the `this` keyword. The `this` keyword in JavaScript refers to the current **scope** of wherever `this` is used, so what it evaluates to depends on where you use it, but if you use it within a class (and not within a callback function or anything else which would change the scope) this will refer to the class/object itself.

If you imagine yourself as `this` and your location in the physical world as the **scope**, consider the following example: if you are in a house, your "scope" may include the room you are in, all of the areas of the house, and even the entire world. These are areas you are

free to explore and access. In a programming sense, if you are inside of a function you can access things inside of that function (your room), other properties/methods of the class that contains that function (other rooms/items in the house), or the entire global scope of the application (the world around the house). When you are anywhere inside of the house, `this` would be a reference to the house (except in special circumstances). Although you can access anything inside of your own house, you can not access things in other peoples houses (unless you have permission to do so).

In the analogy above, we could use the `this` keyword to access other methods/variables of the class (rooms/items of the house). If we were inside of one function, but want to trigger another in the same class, we would do something like: `this.someOtherFunction()`.

If you're not familiar with the `this` keyword, I'd recommend reading this. It is a tricky concept to get your head around, fortunately, this is one of those things that isn't critical to building applications with Ionic. Context and understanding always help you improve as a developer, but it's not going to stop you from progressing. In general, you will only use `this` when you need to refer to other methods or variables of the class that you are in. But it is important to understand that you can't just freely access variables and methods of *other* classes.

Once we have our class defined which acts as a blueprint for creating objects, we could create a new **Person** object like this:

```
let john = new Person('John', 32);
```

The two values I've supplied here will be passed into the `constructor` of the **Person** class to set up the **member variables**. Now if I were to run the following code:

```
console.log(john.getName());
```

John's name would be logged to the console. Similarly, we could also call the `getAge` function to retrieve his age or we could even change his name or age using the set functions. Getters and setters are very common for classes, but we've also defined a more interesting function here which is `canDrive()`. This will return true if the **Person** is over 16 years old, which is the case for John.

Perhaps the most important concept to remember is that the class is just a "blueprint", an object is kind of like an individual copy of a class. We can have multiple objects created from the same class, e.g:

```
let john = new Person('John', 32);
let louise = new Person('Louise', 14);
let david = new Person('David', 22);

console.log(john.canDrive());
console.log(louise.canDrive());
console.log(david.canDrive());
```

In the code above, John, Louise, and David are all individual objects of the **Person** class

and maintain their values separately. If we ran the code above, it would only return **false** for Louise (since she is under 16 years old).

Classes in Ionic/StencilJS

With a basic understanding of classes/objects in general, let's now consider what we use classes for in Ionic/StencilJS applications. We took a sneak peek at a typical Ionic/StencilJS page/component earlier in this lesson - let's take another quick look at an even more simplified version, just focusing on the basic structure:

```
import { Component, h } from "@stencil/core";
import { MyService } from "../../services/my-service"

@Component({
  tag: "app-home",
  styleUrl: "app-home.css"
})
export class AppHome {

}
```

The first thing you will notice is the `import` statements. Anything that is required by the class that you are creating will generally be **imported**. In this case, we are importing **Component** from **@stencil/core** which allows us to use the **@Component** decorator.

We are also importing **MyService** which is a object of our own creation (i.e. not something

provided to us by StencilJS/Ionic). The path for this simply follows the directory structure of your project, in this case, we have the **MyService** object defined inside of a folder called **services** which is one two levels above the current file. The import should link to wherever the file is for the class/object, but it is not necessary to include the file extension. If the file is in the same folder as the file you are coding in, you can reference it with `./the-file`. If you need to go up folders to link to the file you just use `../`, so if the file was up three folders, and then inside of a folder called `cool-stuff` you would do this: `../../../cool-stuff/the-file`.

Next up we have the decorator, which we use to define the "selector" or "tag" for the component (i.e. the name this component will have in our DOM (Document Object Model)) and the URL to the file we are using for CSS styles. This decorator is important, as it is what will allow us to actually "use" this component in our application - i.e. it will take the functionality we define in our class and bundle it up into a custom HTML element called `<app-home></app-home>`. We are going to talk about decorators in more detail in another lesson.

Once we get past the decorator, we finally arrive at the class itself:

```
export class AppHome {

}
```

Everything inside of this class is what will make up the template and functionality of our components, but we will be looking more into that in the next lesson.

Notice, though, that the class is preceded by the **export** keyword. The **export** keyword works in tandem with the **import** keyword - we **export** classes that we want to **import** somewhere else. One final thing that you may find missing from this class is the `constructor`. As we discussed before, the `constructor` initialises an object from a class and is often used to do "setup" style work, but you will find that we don't often use a `constructor` inside of most of our StencilJS components. Typically, if we want to do some "work" right away we would use the "lifecycle hooks" that StencilJS provides, e.g:

```
import { Component, h } from "@stencil/core";

@Component({
  tag: "app-home",
  styleUrl: "app-home.css"
})
export class AppHome {

  componentWillLoad(){
      // do something automatically when component is about to
load
  }

  componentDidLoad() {
      // do something automatically after component loads
  }

}
```

Lifecycle hooks are another thing we are going to touch on more a little later.

With what we have discussed in this lesson, we should have a basic understanding of a class and how they can be used to create components in an Ionic/StencilJS application. However, at this stage, our components wouldn't be anything more than empty shells - we need to give them a template and some logic to actually be of any use.

In the next lesson, we are going to discuss JSX and TypeScript which play a critical role in building StencilJS components for our Ionic applications.

Lesson 2: JSX and TypeScript

With a basic understanding of how classes are used as the basis for constructing our components, we get to the pair that makes up our `.tsx` extension: **JSX + TypeScript**.

JSX is a syntax extension to standard ECMAScript (JavaScript) that allows XML style syntax in JavaScript code - or in easier to understand terms, it allows you to write HTML directly in your JavaScript (not as strings, but as literal HTML syntax). It was created by Facebook and popularised by its usage in the React framework, but JSX can also be used outside of React. If you are already familiar with React then you are going to have a big head start here, but we are going to walk through many aspects of JSX in-depth in this lesson.

TypeScript is also an extension to standard ECMAScript (JavaScript) that adds the ability to add "types" to our code. A type basically just enforces that a particular variable or method has a particular "type" (e.g. `number`, `string`, `boolean`). As I mentioned, TypeScript also handles transpiling "modern" code that isn't supported by browsers yet into code that is, but we won't be focusing on that aspect since it is all handled automatically for us.

JSX

For the most part, using JSX is about mixing JavaScript into your HTML for templates. As you will see from the demonstrations in this section, anything we want to achieve with our templates - whether that is looping over data, conditionally rendering elements, invoking functions, and so on - we do with *mostly* regular JavaScript (rather than framework specific

methods for performing these operations).

The *weird* thing about JSX is that, of course, we are mixing HTML into it and that's not really normal. This, for example, would look wrong to most people unless they were familiar with JSX:

```
render() {
  return <div><p>Hello</p></div>;
}
```

We are attempting to return a chunk of HTML, and this just looks like it is going to cause an error - surely that should be returning a string and is missing its quotations? However, this is perfectly valid syntax with JSX.

With that basic understanding, let's briefly go over how we would do all the typical template stuff with JSX, and then we will move on to discussing TypeScript.

A Basic Template

When building an application with StencilJS, we will have various components that make up our application. These components will each define a `render` function that specifies the template for the component. When necessary (e.g. when certain data has been updated), the `render` function will be calling again to recreate our template (evaluating any JavaScript in the process).

The render function simply needs to `return` the template (using JSX) that we want to display for that component. This will look something like this:

```
import { Component, h } from '@stencil/core';

@Component({
    tag: 'app-home',
    styleUrl: 'app-home.css'
})
export class AppHome {

    render(){
        return (
            <div>
                <p>Hello</p>
            </div>
        )
    }

}
```

Whatever we `return` is what is used for the template, but since we are in JavaScript land we can do more than just return simple HTML. We could build whatever logic we like in here. For example, we could render local or member variables inside of the template:

```
import { Component, h } from '@stencil/core';

@Component({
    tag: 'app-home',
    styleUrl: 'app-home.css'
})
export class AppHome {

    private firstName: string = "Josh";

    render(){

        const myMessage = <p>welcome to my app!</p>;

        return (
            <div>
                <p>Hi {this.firstName}, {myMessage}</p>
            </div>
        )
    }

}
```

and the resulting template would be:

```
<div>
```

```
    <p>Hi Josh, welcome to my app!</p>
</div>
```

There is *a lot* more we could do, and we are going to cover some of those now.

Multiple Root Nodes

The render function must return a single root node, that means that this is will work:

```
return (
  <div></div>
)
```

and this will work:

```
return (
  <div>
    <p>Hello</p>
    <div>
      <p>there.</p>
    </div>
  </div>
)
```

but this **will not**:

```
return (
  <div>root one</div>
  <div>root two</div>
)
```

To address this scenario you can either wrap the two nodes inside of a single node:

```
return (
  <div>
    <div>Root one</div>
    <div>Root two</div>
  </div>
)
```

or you can return an **array** of nodes instead, like this:

```
return ([
  <div>Root one</div>,
  <div>Root two</div>
```

```
])
```

Notice the use of square brackets to create an array, and that each node would be followed by a comma as in an array. Although it may look a little strange to use HTML syntax in an array like this, it is the same idea as doing this:

```
return ([
  'array element 1',
  'array element 2'
])
```

Expressions

Expressions allow us to do things like execute logic inside of our templates or render out a variable to display on the screen. We've already seen this in the code above, but we can do more than just render out variables, for example:

```
render(){
    return (
        <div>
            <p>{ 1 + 1 }</p>
        </div>
    )
```

```
}
```

This example would execute the `1 + 1` operation, and display the result inside of the paragraph tag. In this case, it would, therefore, render the following out to the DOM:

```
<div>
  <p>2</p>
</div>
```

We could also use expressions to make function calls:

```
calculateAddition(one, two){
  return one + two;
}

render(){
    return (
        <div>
            <p>{this.calculateAddition(1,5)}</p>
        </div>
    )
}
```

and much more, some of which we will touch on later.

Styles

If you attempt to add a style to an element in JSX like this:

```
render(){
    return (
        <div style="background-color: #f6f6f6; padding: 20px;">
            <p>Hello</p>
        </div>
    )
}
```

You will be met with an error that reads something like this:

```
Type 'string' is not assignable to type '{ [key: string]: string;
}'.
```

But what's the deal with that? Aren't we allowed to use standard HTML syntax? Not quite, there are a few differences. With JSX, inline styles must be supplied as an object, where the properties of that object define the styles you want:

```
render(){
    return (
        <div style={{
          backgroundColor: `#f6f6f6`,
          padding: `20px`
        }}>
            <p>Hello</p>
        </div>
    )
}
```

We are assigning an expression (which we just discussed) to style that contains an object representing our style properties. That is why there are two curly braces - one set to contain the expression and one set to contain the object we are creating. You will notice that we are using `camelCase` - so instead of using the usual hyphenated properties like `background-color` or `list-style-type` we would use `backgroundColor` and `listStyleType`.

Conditionals

There are different ways to achieve conditionally displaying data/elements with JSX (just as there are many ways to achieve things with JavaScript in general), so let's take a look at a few. We covered before how you could make a function call inside of an expression in a template, and that is one way that you could conditionally display an element:

```
import { Component, h } from '@stencil/core';

@Component({
    tag: 'app-home',
    styleUrl: 'app-home.css'
})
export class AppHome {

    private loggedIn: boolean = false;

    getWelcomeMessage(){
        if(this.loggedIn){
            return 'Welcome back!';
        } else {
            return 'Please log in';
        }
    }

    render(){
        return (
            <div>
                <p>{this.getWelcomeMessage()}</p>
            </div>
        )
    }

}
```

You could also achieve the same effect by building some logic directly into the expression in the template, rather than having a separate function:

```
render(){
    return (
        <div>
            <p>{this.loggedIn ? 'Welcome back!' : 'Please log
in.'}</p>
        </div>
    )
}
```

You could render entirely different chunks of your template using `if/else` statements (remember, whatever we `return` is what is used for the template):

```
import { Component, h } from '@stencil/core';

@Component({
    tag: 'app-home',
    styleUrl: 'app-home.css'
})
export class AppHome {

    private loggedIn: boolean = false;
```

```
    render(){

        if(this.loggedIn){
            return (
                <div>
                    <p>Welcome back!</p>
                </div>
            )
        } else {
            return (
                <div>
                    <p>Please log in.</p>
                </div>
            )
        }

    }

}
```

You can completely remove any rendering to the DOM by returning `null` instead of an element:

```
import { Component, h } from '@stencil/core';

@Component({
    tag: 'app-home',
```

```
    styleUrl: 'app-home.css'
})
export class AppHome {

    private loggedIn: boolean = false;

    render(){

        if(this.loggedIn){
             return (
                <div>
                    <p>Welcome back!</p>
                </div>
            )
        } else {
             return null
        }

    }

}
```

If you don't want to return entirely different templates depending on some condition (this could get messy in some cases) you could also render entirely different chunks just by using a ternary operator inside of your template like this:

```
private loggedIn: boolean = false;
```

```
render(){
    return (
        <div>
            {
                this.loggedIn
                ?
                <div>
                    <h2>Hello</h2>
                    <p>This is a message</p>
                </div>
                :
                <div>
                    <h2>Hello</h2>
                    <p>This is a different message</p>
                </div>
            }
        </div>
    )

}
```

If you aren't familiar with the ternary operator, we are basically just looking at this:

```
this.loggedIn ? 'logged in' : 'not logged in';
```

which is a simplified version of:

```
if(this.loggedIn){
  return 'logged in';
} else {
  return 'not logged in';
}
```

We can also simplify the ternary operator more if we don't care about the else case. For example, if we only wanted to show a message to a user who was logged in, but we didn't care about showing anything to a user who isn't, we could use this syntax:

```
private loggedIn: boolean = true;

render(){
    return (
        <div>

            {
                this.loggedIn &&

                <div>
                    <h2>Hello</h2>
                    <p>This is a message</p>
                </div>
```

```
            }

        </div>
    )

}
```

This will only render out the message if loggedIn is true. The methods I have mentioned here should be enough to cover most circumstances, but there are still even more you can use. In the end it's all just JavaScript, there is no specific "framework" way of doing something (because we aren't using a framework!).

Looping Data

We will often want to create templates dynamically based on some array of data like an array of `todos` or `posts` and so on. Once again, with StencilJS and JSX, we just use standard JavaScript syntax/logic embedded directly into the template to achieve this.

We can use the `map` method on an array of data to loop through the data and render out a part of the template for each iteration. Let's take a look at an example from the documentation:

```
render() {
  return (
    <div>
      {this.todos.map((todo) =>
```

```
        <div>
          <div>{todo.taskName}</div>
          <div>{todo.isCompleted}</div>
        </div>
      )}
    </div>
  )
}
```

In this example, we would have a class member variable named `todos` that we are looping over. Notice once again that we have our curly braces for expression surrounding the JavaScript we want to execute. The `map` method of the `todos` array will iterate over each element in the array, and for each todo we will render this out:

```
<div>
  <div>{todo.taskName}</div>
  <div>{todo.isCompleted}</div>
</div>
```

The `{todo.taskName}` and `{todo.isCompleted}` expressions used here will be executed and the values for the particular todo in that iteration of the map method will be used. The end result would be a template that might look something like this:

```
<div>
    <div>
        <div>Get apples</div>
        <div>true</div>
    </div>
    <div>
        <div>Clean shed</div>
        <div>false</div>
    </div>
    <div>
        <div>Exercise</div>
        <div>true</div>
    </div>
</div>
```

In order for StencilJS to be able to perform as efficiently as possible, it is important that if you intend to change this data (e.g. you can add/remove `todos`) that you give each todo that is rendered out a unique `key` property. You can attach that key to the root node of whatever you are rendering out for each iteration, e.g:

```
render() {
  return (
    <div>
      {this.todos.map((todo) =>
        <div key={todo.id}>
```

```
          <div>{todo.taskName}</div>
          <div>{todo.isCompleted}</div>
        </div>
      )}
    </div>
  )
}
```

Event Binding

The last thing we are going to cover is event binding, which we will use mostly to handle users clicking on buttons or other elements. We can handle DOM events by binding to properties like `onClick`. For example, if we wanted to run a method that we created called `handleClick` when the user clicks a button we might do something like this:

```
<ion-button onClick={this.handleClick(event)}>Click me</ion-button>
```

This is fine and would invoke our `handleClick` method, but the downside of this approach is that it won't maintain the scope of `this`. That means that if you were to try to reference a member variable like `this.loggedIn` inside of your `handleClick` method it would not work - because `this` would no longer refer to the class of your component.

You can solve this issue in either of the following ways. You could manually bind the

function to the correct scope like this:

```
<ion-button onClick={this.handleClick(event).bind(this)}>Click
me</ion-button>
```

or you could use an arrow function like this (which is the more popular approach):

```
<ion-button onClick={(event) => this.handleClick(event)}>Click
me</ion-button>
```

You can use different approaches depending on the scenario if you like, but in general, we will always just use the last example. Even though it may not be required in every circumstance (e.g. we might not need `this` to refer to the current class) I think it helps just to keep things nice and consistent.

As well as `onClick` you can also bind to other DOM events like `onChange` and `onSubmit`. With Ionic, you can also bind to any of the events that the Ionic web components emit.

That brings us to the end of our primer on JSX, but we will, of course, get to practical examples later. For now, let's move on to TypeScript!

TypeScript

For the most part, the way in which we actually use TypeScript in our projects doesn't really add anything "functional". Unlike JSX, where we need to understand the concepts in order to effectively achieve our goals, you could just about ignore the existence of TypeScript and still be able to build an application. The goal of this section is to convince you that you should pay attention to TypeScript, and try to use it effectively in your projects.

TypeScript is something that seems like more of a pointless annoyance to begin with, until you've been using it for a while and start appreciating the benefits it offers. In a nutshell, using TypeScript means adding **types** to our code. For example, instead of declaring a member variable like this:

```
public firstName;
```

we would instead do this:

```
public firstName: string;
```

This gives the `firstName` variable a type of `string` meaning that we can only ever assign `string` values to the variable. If we attempted to set the `firstName` variable to the number `1` it would cause an error.

We are going to talk a lot more about what TypeScript is exactly, but in my mind, the two

biggest benefits of paying attention to using TypeScript properly (i.e. adding appropriate "types" to your variables/methods) are:

- It will catch many bugs/issues before you do
- When using a code editor that supports TypeScript and code completion, you can save a huge amount of time with type information at your fingertips

If our variables have appropriate types, if we run into a situation in our code where we accidentally assign the wrong type of data to a variable we are going to know right away (rather than having to let our code fail before going back and fixing the issue). Sometimes the error you run into without types might be obvious and quick enough to fix, but sometimes just having an appropriate type set up could save you a lot of time trying to debug an issue.

Also, consider a situation where you are calling a method you created and using it in some way in your code. Maybe the purpose of this method is to return a list of characters in a show: `getCharacters()`. Perhaps you try to use this method like this:

```
let characters = someService.getCharacters();
console.log(characters);
```

But what if `getCharacters()` doesn't just return the characters directly, it's an asynchronous function that returns a promise that resolves with the characters (we will talk more about asynchronous code and promises soon). Our code wouldn't work in this case, and again we wouldn't know about it until we ran our code and it failed. If we had given our

`getCharacters()` an appropriate return type, then we would have known immediately. Even if we don't make the mistake, and instead take the time to look up what `getCharacters()` returns, TypeScript still saves us time because if we are using a code editor with TypeScript support (e.g. Visual Studio Code) we can just hover over the service or method to see what it returns.

Let's look at the different types we might use in a typical application and where we can use them. First of all, let's consider the sorts of types we have. Commonly used types include:

- `any` (can be useful in some situations, but best avoided where possible as it doesn't add any value)
- `number`
- `string`
- `boolean`
- `Function`
- `Object`

Then we have some slightly more advanced types like:

- `Promise<boolean>`
- `string[]`

These might look a little more intimidating, but it is still pretty straight-forward. The `Promise<boolean>` type just means it is a `Promise` that resolves with a `boolean` value. You can just change the `<boolean>` part to reflect whatever type the `Promise` will resolve with. The `string[]` type indicates an array of strings, e.g: `['apple', 'banana', 'grape']`. It is an array filled with string values. Again, you can change this

to reflect whatever type of data will be in your array, e.g. `number[]`.

We can also use **interfaces** to create our own custom types. For example, let's say we had an array filled with these objects:

```
this.myArray = [
    {title: 'apple', amount: 1},
    {title: 'banana', amount: 1},
    {title: 'grape', amount: 1},
    {title: 'milk', amount: 1}
];
```

I could give `this.myArray` a type of `Object[]` indicating that it is an array of objects:

```
public myArray: Object[];
```

But, like our `any` type, this doesn't really add any value. We could still add data to that array that we don't want there. For example, this would be perfectly valid:

```
this.myArray = [
    {title: 'apple', amount: 1},
    {title: 'banana', amount: 1},
```

```
    {direction: 'north' time: 3929013, finished: false}
];
```

However, if we were to create a custom `interface` like this:

```
interface GroceryItem {
    title: string;
    amount: number
}
```

and we applied that custom type to our array:

```
public myArray: GroceryItem[];
```

This would now ensure that `myArray` only contained grocery items that fit the structure of the type we defined. Now, let's consider where we might use our types.

As we have already seen, we can attach types to our member variables:

```
public firstName: string;
public myArray: GroceryItem[];
```

We can also attach types to any other variables, for example within a function:

```
getCharacters(){
    let characters: string[] = [];
}
```

We could attach types to functions to specify the type of data they should `return`:

```
getCharacters(): Promise<string[]>{

}
```

This would mean that `getCharacters` should return a `Promise` that resolves with an `array` of `strings`. The function as it is above now would cause an error because it isn't returning anything yet (and it needs to return that Promise).

Often, a function doesn't need to return anything, in that case, we can use the `void` type:

```
doSomething(): void {

}
```

Even the parameters inside of functions can have a type applied to them:

```
getCharacters(showTitle: string): Promise<string[]>{

}
```

This method expects a `showTitle` to be passed in which much be a `string`, it will then `return` a `Promise` that resolves with an `array` of `strings`.

You could assign a type to just about everything if you like, and technically speaking the more type information you provide the better off you will be. How much type information you actually provide is ultimately up to you (or your team). Personally, I mostly just focus on making sure I have types for member variables and return types for methods. A lot of the time I don't use types for function parameters, but this is partly to make my examples look simpler/less intimidating. For example, this:

```
getCharacters(showTitle){
    let characters = [];
}
```

Looks quite simple and obvious, whereas this:

```
getCharacters(showTitle: string): Promise<string[]>{
```

```
    let characters: string[] = [];
}
```

would be far more confusing/intimidating to somebody not familiar with types and TypeScript (even though the function is exactly the same). Nonetheless, the second example will provide a far greater benefit than the first. It will prevent you from making errors in your code, and it will make it easier to look up the methods you have created.

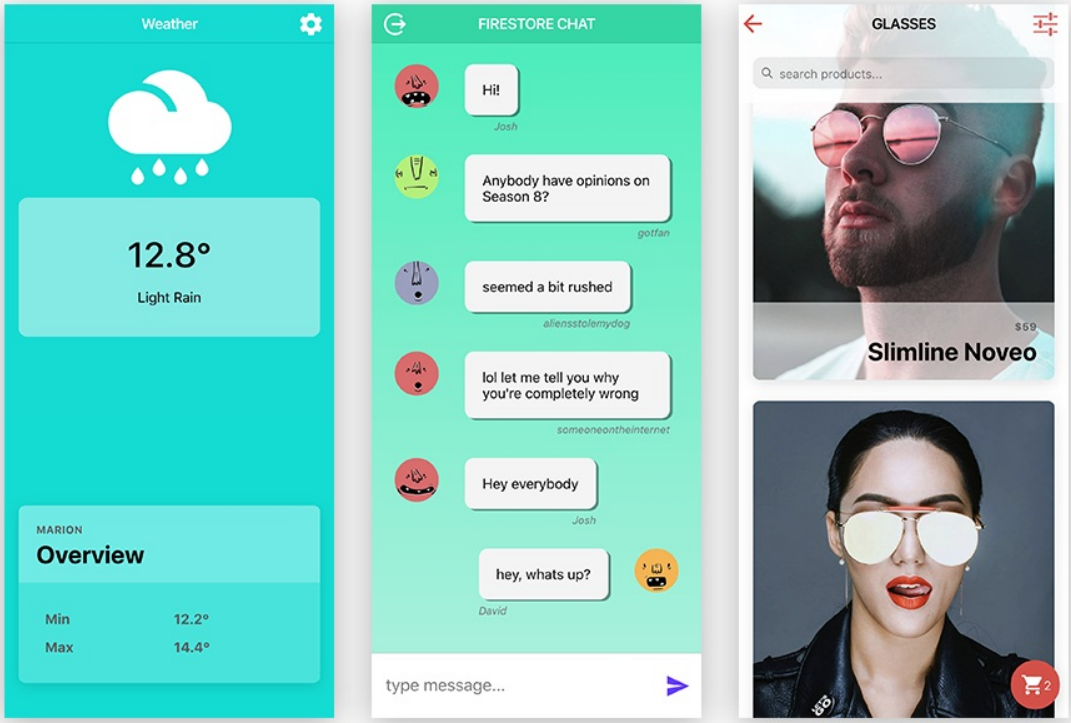
As we make our way throughout this book, we will encounter plenty more examples and have chances to flex our JSX and TypeScript muscles.

Continue reading...

This PDF provided a preview of two of the lessons from [Creating Ionic Applications with StencilJS](#).

The complete book, as well as its resources, serve as an all-in-one resource for learning how to use StencilJS to build Ionic applications. It is also a useful reference to continually come back to as you build applications.

As well as covering all of the basic theory, the complete book also covers building three example applications:



Each of these example applications varies in degree of difficulty and concepts covered.

For more details about the rest of the book, and if you would like to purchase the book, please click here.