Building HTML5/CSS3/JavaScript Applications an Introduction

Adrian Pomilio
Twitter: @adrianpomilio
Blog: uiandtherest.com

Table of Contents

Summary

DON'T LEARN HTML5! Learn how HTML/CSS/JavaScript combine together to build applications.

This hands on session is for beginners and those interested in creating applications using the standards stack. It is beyond the scope of this session to dive deep into all of the techniques and features of the stack.

The application we will be building is simple, yet key to understanding how to develop applications using JavaScript, HTML, and CSS. At the end of the session you will have created an application that can keep track of the license plates you see on your adventures.

Core concepts will be data entry, data retrieval, maintain state, and basic MVC practices within a browser.

Setup

IDE

You may utilize any IDE of your choice that will allow you to create and edit JavaScript, CSS, and HTML files.

Server

You may use any web server (e.g. Apache) that you have the ability to publish to the server. You may use a remote server (keep in mind conference wifi) or a locally installed server.

Browser

You may use any modern browser, but for best results we highly recommend the most recent version of Google Chrome.

Dev Tools

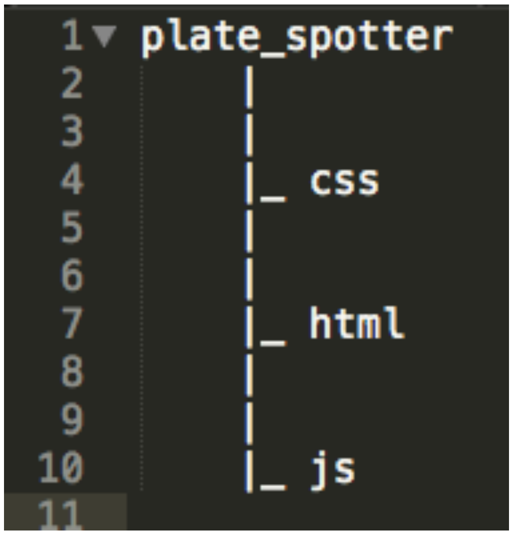
Outside of your IDE, and browser, you will need to be able to debug your JavaScript and inspect elements within your browser. Be sure to have the appropriate developer tools installed for your browser.

Course Resources

We will be able to provide the course code via a thumb drive, or you may pull it from GIT at: https://github.com/adrianpomilio/plate_spotter

Directory Structure

You will want to create the following directory structure for your application, see figure 1.



```
plate_spotter
      |
      |
      |_ css
      |
      |
      |_ html
      |
      |
      |_ js
```

Figure - 1

Section 1 - The Markup

HTML markup is one of the underrated pieces of the puzzle. Markup can help make, or break, application development and maintenance. Poor markup leads to interface bugs as well as a pattern of "hacking" around a problem with more bad markup.

We are going to create the HTML skeleton for our application.

Step 1 – Creating index.html

Create a new, blank/empty, document and name it **index.html,** save the document to the **html** directory.

Step 2 – Adding Basic Markup

We are going to be leveraging some of the **HTML5** features so we must ensure we start off with the proper **DOCTYPE**. The doctype gives instructions to the browser on how to process the html file. Enter the code seen in **figure 2** in to the **index.html** file.

```
<!DOCTYPE html>
<html>
	<head>
		<meta http-equiv="Content-Type" content="text/html; charset=UTF-
	8" />
		<meta name="viewport" content="width=device-width, initial-
	scale=1.0" />
		<title>Plate Spotter</title>
	</head>
	<body>
	</body>
</html>
```

Figure – 2

NOTE – You will notice a **<meta name="viewport"...>** tag. This tag gives instructions to mobile browsers to set the initial scale within the viewport.

Step 3 – Application Markup

Our interface is going to contain four sections. Using **HTML5** syntax you will be adding the following tags to your document: **<header>, <section>, <article>, <footer>**. These tags do not offer any "new" functionality, but rather serve as a more semantic way to **describe** your application.

Inside of **index.html**, between the **<body>...</body>** tags add the markup seen in **figure 3.**

```
<header class="header">
        <h1>Plate Spotter</h1>
</header>
<section>
        <article>Your Current Location:
                <div id="latLonDisplay"></div>
        </article>
</section>
<section id="spotFormPanel" class="panel">
        <header class="sub-heading"><h2>Select a State</h2></header>
</section>
<section class="panel">
        <header class="sub-heading"><h2>My Plates</h2></header>

        <div id="mySpotsPanel"  >
        </div>
        <div class="clear"></div>
</section>
<footer>
</footer>
```

Figure – 3

NOTE - You will see that there are attributes of **class** and **id** on some of the elements. These attributes will be used in later sections to manipulate the elements visually and logically.

Yes, there are still lovely **<div>** tags in our markup. That's okay.

At this point your index.html page should look like **figure 4.**

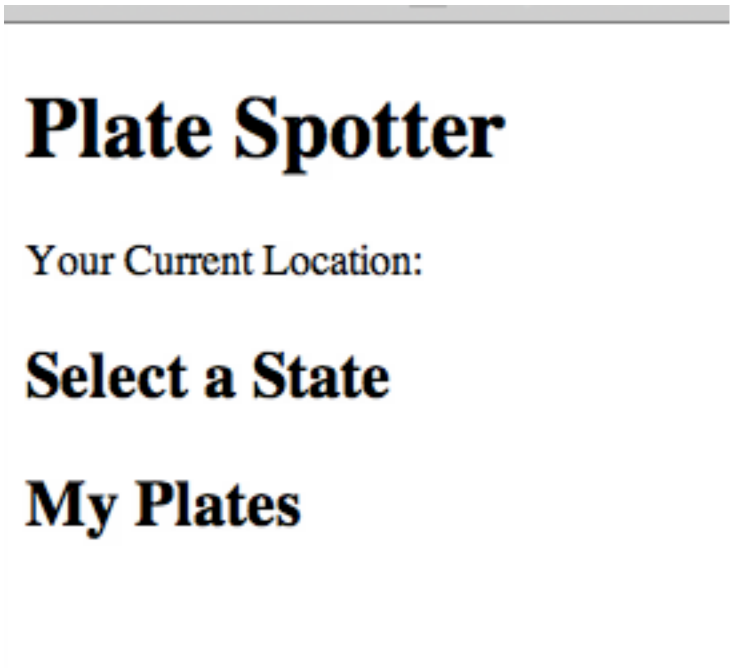


Plate Spotter

Your Current Location:

Select a State

My Plates

Figure – 4

Not very impressive, but structurally sound for our next section…

Section 2 – CSS

CSS allows you to control the layout, style, and in some cases interactions. Properly written **CSS** coupled with proper markup can make life very easy if you decide to change the layout or styling.

Step 1 – Create app.css

Create a new, blank/empty, document and name it **app.css,** save the document to the **css** directory.

Step 2 – Link CSS

Open **index.html** and place a link to the **app.css** as seen in **figure 5** following the **<title>** tag.

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

Figure -5

You have now linked your **app.css** file to your **index.html** document. If you view your application you will not see any changes at this time.

Step 3 – Add Base Styles

We are going to add basic styling for our tags. Inside of your **app.css** file add the code seen in **figure 6.**

```
html, body {margin:0;
        padding:0;
        font-family:Helvetica;
        font-size:1em;
        background-color:#09F;}

section {margin:10px auto;
                padding:10px;
                width:500px;
                border-radius:10px;
                background:#fff;
        }

section > div {margin:5px auto;}

select {margin:5px 20px;
                padding:5px;
                border:1px solid #333;
        }
```

Figure – 6

At this point you have added a foundational style to HTML elements. These styles will be applied directly to any corresponding html element (e.g. the **section** tags will all have a **width** of 500px).

Step 4 – Add License Plate Markup

Let's add the markup structure for a license plate. Inside of your **index.html** file add the code found in **figure 7** after the opening **<div>** tag with the **id** of **mySpotsPanel** but before the closing **</div>.** Your code should look like **figure 7.**

```
<div id="mySpotsPanel"  >

        <div class="license-plate" id="plateId" >
                <div class="license-plate-date">sample</div>
                <div class="license-plate-state">state</div>
                <div class="remove-plate-icon" data-plateId="data">x</div>
        </div>

</div>
```

Figure – 7

View your **index.html** file in your browser. If everything is correct you should see something like **figure 8.**

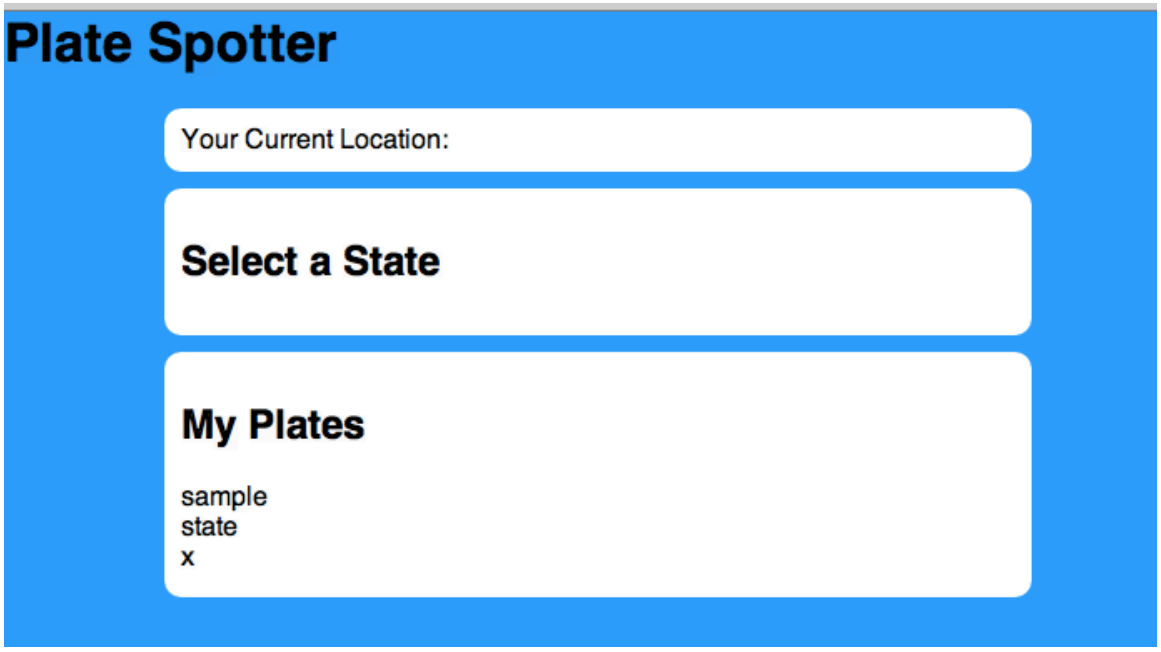


Plate Spotter

Your Current Location:

Select a State

My Plates

sample
state
x

Figure - 8

Okay, there is some basic styling, but the license plate markup doesn't look right, and the header seems out of whack. In our next step we are going to add the appropriate CSS to even things out.

Step 5 – Add Basic Classes

Open your **app.css** file and add the code seen in **figure 9** after previous declarations (at the end of the file).

```
.header {
        margin:20px auto;
                padding:10px;
                width:500px;
                border-radius:10px;
                background:#fff;
                color:#09F;
        }

.sub-heading {margin:0;
                padding:5px;
                border-bottom:1px solid #09F;
        }
```

```
.sub-heading > h2 {margin:0;
                padding:0;
                font-size:1.2em;
        }
```
Figure - 9

Save your changes and view your **index.html** file. You should see something just like **figure 10.**

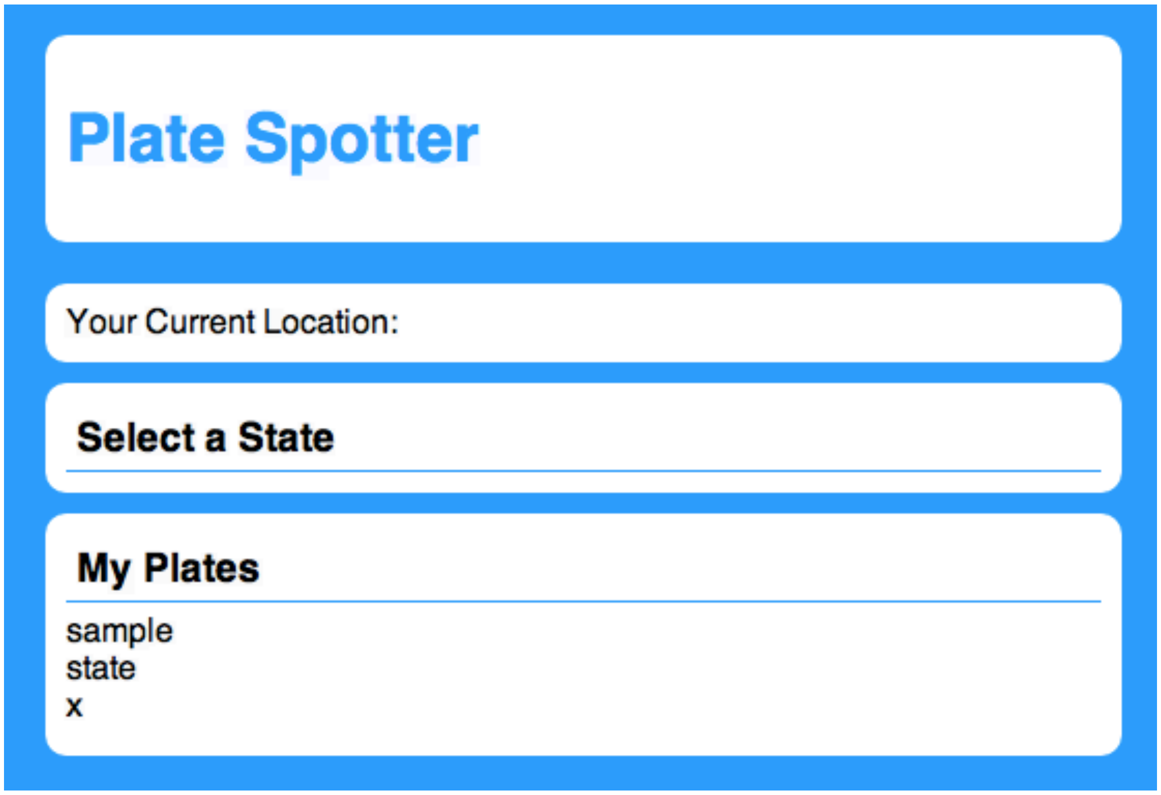


Figure -10

So what happened? Well you now added **class declarations** to your CSS file. The class **.header** corresponds to the html tag with a class of **header.** That means that any element that uses **class="header"** will inherit the style defined in your **app.css.**

NOTE - Classes are denoted by a preceding **.** before the text. Classes can be used many times over within the same document. IDs are denoted by a preceding **#** before the text. IDs should only be used once and are mainly used to get a "handle" on an element or to enforce extreme specificity.

Now every thing looks okay except for the license plate code, it just looks like text in the **My Plates** section, as seen in **figure 10.** Let's fix that in the next step.

Step 6 – Add License Plate Styling

Let’s fix our license plate by adding the code in **figure 11** to the end of your **app.css** file.

```
.license-plate {
	position:relative;
	float:left;
	display:inline-block;
	width:220px;
	height: 80px;
	border:4px solid #000;
	border-radius:10px;
	text-align: center;
	margin:10px ;
	overflow:hidden;
	background: #deefff;
	background: -moz-linear-gradient(top,  #deefff 0%, #98bede 100%);
	background: -webkit-gradient(linear, left top, left bottom, color-
stop(0%,#deefff), color-stop(100%,#98bede));
	background: -webkit-linear-gradient(top,  #deefff 0%,#98bede 100%);
	background: -o-linear-gradient(top,  #deefff 0%,#98bede 100%);
	background: -ms-linear-gradient(top,  #deefff 0%,#98bede 100%);
	background: linear-gradient(to bottom,  #deefff 0%,#98bede 100%);
	filter: progid:DXImageTransform.Microsoft.gradient( startColorstr='#deefff',
endColorstr='#98bede',GradientType=0 );

}

.license-plate-state {
	position:absolute;
	left:0;
	right:0;
	bottom:2px;
	height:15px;
}

.license-plate-number{
	position:absolute;
	left:0;
	right:0;
	top:15px;
	height:15px;
}
.remove-plate-icon {
	position:absolute;
```

```
        bottom:0;
        right:0;
        border-radius: 50%;
        border:1px solid red;
        color:white;
        background-color:red;
        cursor:pointer;
        height:20px;
        width:20px;
        opacity:.60;
}
.remove-plate-icon:hover {
        opacity:1;
}
```

Figure – 11

There is a lot going on here. We are using 5 different classes to describe the look of the license plate. We are also using positioning, gradients, and a pseudo-class. For now, save your **app.css** and take view the **index.html** file. If everything went well you should see something exactly like **figure 12.**

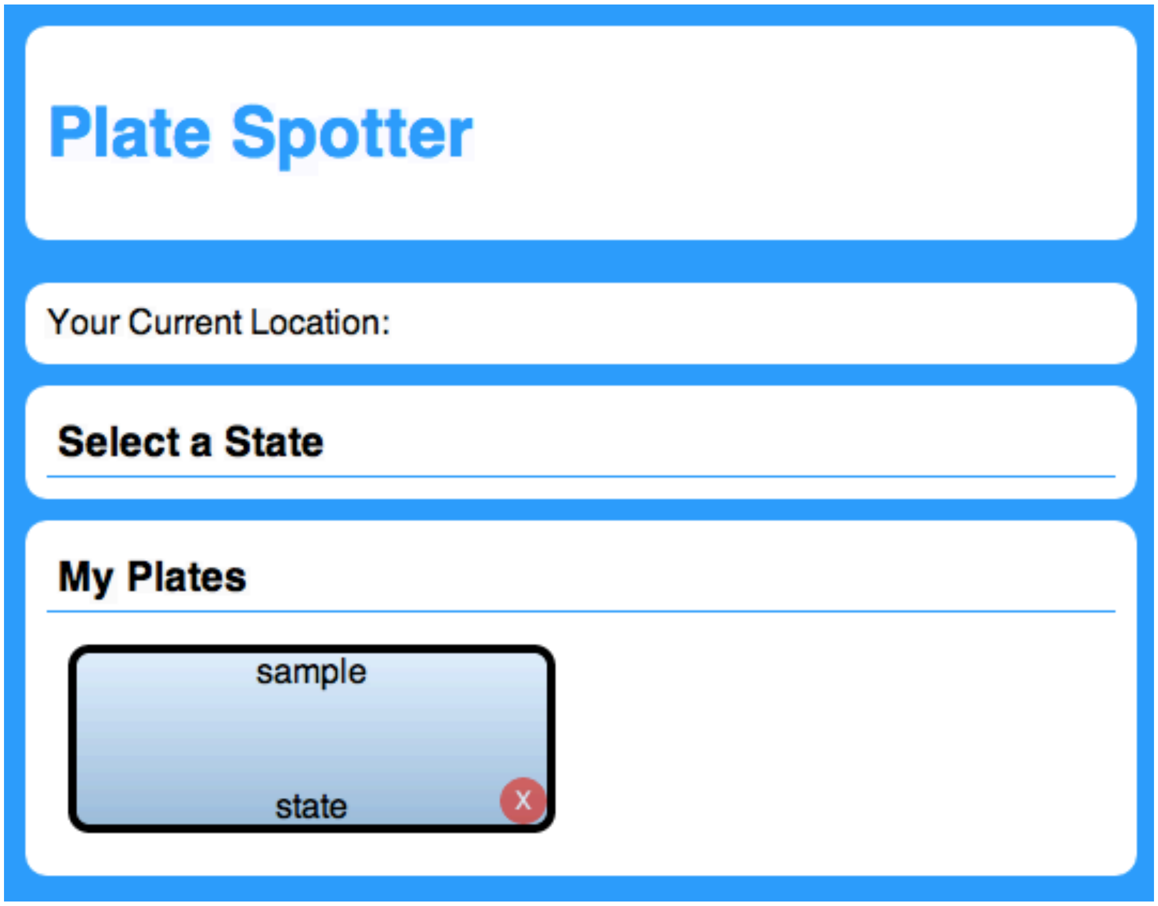


Figure – 12

Now we have something resembling a license plate. Right now you have the overall structure of the visual pieces of your application. You might be wondering how you select a state, well we will cover that in the next section – JavaScript

Section 3 – JavaScript

Now it's time to add functionality to your application. We will do this entirely through JavaScript. This section is the final stage but it is also the most complex. We will be covering: external libraries, templating, and a module pattern. On top of all that we will be listening for events, storing/removing data, updating the view, and finding out where you are.

It seems complex, but it isn't.

Step 1 – Create app.js

Create a new, blank/empty, document and name it **app.js,** save the document to the **js** directory.

Step 2 – Create spotter.js

Create a new, blank/empty, document and name it **spotter.js,** save the document to the **js** directory.

Step 3 – Copy states.js

For simplicity sake you will need to copy the **states.js** file from the **js** directory of the **final** project to your **js** directory.

Step 4 – Copy lib Directory

You will want to copy the **lib** directory and its contents to the same place you created the **plate_spotter** directory. Do not place this inside your **plate_spotter** directory, keep it at the same level. (I did it this way so we would not have to keep copying library files over and over for each section).

Step 5 – Link JavaScript

Open **index.html** and enter the code that is in **bold** in **figure 13,** after the closing **</footer>** tag and before the closing **</body>** tag. The order of the files does matter.

```
</footer>

<script src="../../lib/jquery-1.7.1.min.js" type="text/javascript"></script>
<script src="../../lib/handlebars-1.0.0.beta.6.js"
type="text/javascript"></script>
<script src="../js/states.js" type="text/javascript"></script>
<script src="../js/app.js" type="text/javascript"></script>
<script src="../js/spotter.js" type="text/javascript"></script>

</body
```

Figure – 13

What we have done is imported/linked 5 different JavaScript files to your **index.html** file. Here is a brief description of each of them.

1. jQuery – this is the first external library. jQuery allows us to manipulate the DOM in a consistent manner across different browsers.
2. Handlebars – this is the second external library. Handlebars will allow us to create client side templates that can be rendered by JavaScript. Templating is a powerful tool.
3. States – this is the third JavaScript file. This is simply an array of objects containing information about States found in the US. This is just a data file, which could easily be swapped out for a service. You won't be doing much inside this file, but take a look anyway.
4. App – this is the fourth JavaScript file. This is where we will place "generic" JavaScript and initialize our application, as well as set up event handlers.
5. Spotter – this is the fifth and final JavaScript file. This is where we will spend most of our time. This is a file that uses a modular pattern and is the brains of this application.

NOTE - External libraries are used to assist you in your development; there are many options out there that do what these do. Technically you could use different libraries with minimal change to your code to make the application work.

Step 6 – Initializing jQuery

Now that we have our core JavaScript files in place we need to get rolling with some boilerplate stuff. We are going to initialize jQuery for use in our application. Instead of placing JavaScript code in our **index.html** file we will work out of **app.js** for this step. By doing this we are keeping a clear separation of logic and presentation.

Open **app.js** and add the code in **figure 14** and save the file.

```
$(document).ready(function(){

});
```

Figure – 14
This is a crucial piece of code that leverages jQuery. This function will execute any JavaScript once the DOM is ready, which is different than when the document has completed loading everything. Remember, jQuery is a library that helps us manipulate the DOM.

Templating
Templating is a very powerful technique that allows us to separate our markup from our JavaScript. The next three steps will focus on creating simple templates for us to use in later steps. We are leveraging the Handlebars templating library which you have already included in previous steps.

Step 7 – Creating the States Template
Open **index.html** and place the code found in **figure 15** after the last **<script>** tag and before the closing **</body>** tag. You may add the templates anywhere, but I find it easier to place them at the bottom of the document.

```
<script id="spot-create-template" type="text/x-handlebars-template">
	<div id="spotCreatePanel">
		<select id="statePicker">
		<option value="Choose State">Choose State</option>
			{{#each states}}
				<option value="{{name}}">{{name}} {{abbr}}</option>
			{{/each}}
		</select>
	</div>
</script>
```

Figure – 15

There are some important concepts to grasp with this template, which will apply to the other templates as well.

- The **<style>** tag has an **id** attribute is assigned a value, this will be used to reference this template within your code
- The **<style>** tag has a **type** attribute with a value of **"text/x-handlebars-template"** which means that a browser will completely ignore this **<script>** tag and it's contents. But since you have included the Handlebars library we will be able to interact with the template later on.
- You will notice a **{{somevalue}}** syntax within the template, these are placeholders for values from an object when you implement the template.

Otherwise this is simply HTML markup, and very easy to read. All of this will apply to our other templates as well.

Step 8 – Creating the License Plate Template

Open **index.html** and place the code found in **figure 16a** after the **spot-create-template.**

```
<script id="spot-view-template" type="text/x-handlebars-template">
        {{#each plates}}
        <div class="license-plate" id="{{id}}" >
                <div class="license-plate-number">{{date}}</div>
                <div class="license-plate-state">{{state}}</div>
                <div class="remove-plate-icon" data-plateId="{{id}}">x</div>
        </div>
        {{/each}}
</script>
```

Figure – 16a

Now you are probably wondering why we made a template for something that we already have in our markup, the license plate. That was a place holder, so you will need to remove the contents of **mySpotsPanel** so that it looks like **figure 16b.** You will no longer need that markup.

```
<div id="mySpotsPanel"  >
</div>
```

Figure – 16b

Step 9 – Creating the Location Template

Open **index.html** and place the code found in **figure 17** after the **spot-view-template.**

```
<script id="location-view-template" type="text/x-handlebars-template">
                <label>Lat:</label>{{coords.latitude}}
                <label>Lon:</label>{{coords.longitude}}
</script>
```

Figure – 17

You have now put in place three templates that you will interact with from your module. You can save your **index.html** file and see that there are no real changes, nothing is happening yet. This is just the setup for the fun stuff.

Step 10 – Creating a Module

A Module is simply a type of design pattern for writing your JavaScript. A module allows you to encapsulate all of your component, or applications needs without polluting the global namespace. This is a large topic on its own, so I highly recommend doing additional research on Modules.

NOTE – for more information on Modules I recommend the following blog post: http://www.adequatelygood.com/2010/3/JavaScript-Module-Pattern-In-Depth

For our purposes we are going to create one module called **SPOTTER** and it is going to be created as an anonymous selfcalling function that we pass in two arguments.

Open the **spotter.js** file you created previously and enter the code as seen in **figure 18.**

```
var SPOTTER = ( function(_s, $){
	"use strict";

return _s;

}(SPOTTER || {},  jQuery))
```

Figure - 18

Now there are a few things taking place here.

- **SPOTTER** will be the **namespace** that the rest of our application will interact with our module.
- The var **SPOTTER** is actually a self calling function, which is denoted by the **(** prior to the **function** , and you will see a **)** at the end to close it all of.
- We are also passing in two arguments. The first is **SPOTTER || {}** and the second is **jQuery.**
- The first argument says that if **SPOTTER** doesn't exist then create it.
- The second argument is allowing us to pass in jQuery and use it within the module.
- You will notice a **return _s;** statement at the end as well. This will make methods available for outside calls (e.g **SPOTTER.getSomething()**).
    - JavaScript scope is kept within the Function

NOTE - If you are lost, please ask questions. If you are reading this at a later date please review the blog link in the preivous note.

Step 11 – Initializing the Module

We have created a module but we need to **initialize** the module by calling it from our application once the DOM is loaded, or not, we are doing so just to be safe.

The **init()** method in the module sets up our application for us to use. So lets set up a few things within our **spotter.js** module.

We need to add some local variables to keep track of things in the future, and we need to have an initialize method to kick things off. To do this lets open **spotter.js** and set up or module to look like **figure 19.**

```
var SPOTTER = ( function(_s, $){
        "use strict";
        var _locObj, _mySpots = {plates:[]}, _watchPositionId;
        _s.init = function(){
                initializeLocation();
                initializeStorage();
                reportPlateView(states);
                displaySpotsView();
        };
        return _s;

}(SPOTTER || {},  jQuery))
```

Figure -19

Open **app.js** and inside of the jQuery ready function place the code found in **figure 20 ,** and save the file.

```
SPOTTER.init();
```

Figure 20

We aren't doing anything yet, but we now have:

- An **init** method to intitialize the module
- Three local variables to keep track of some important information
- Only one global variable was added, so we are not polluting the global scope.

NOTE – You will get errors at this point as you are calling functions in the init function that do not exist yet. Ignore those for now.

Step 12 – Add a Plate

Let's start to build the basic logic to add a plate when a user selects a state. This won't work until we add the event listeners in upcoming steps, but it will get us setup.

Get ready to roll up your sleeves for a lot of "boilerplate."

In **figure 21** we have the **_r.setSpot()** method. You will want to add this code inside of your **SPOTTER** function/module and save your **spotter.js** file.

```
_s.setSpot = function(state){
                var _obj = {};
                // bad way to set an ID but for us it will work
                _obj.id = Math.floor(Math.random() * (1 - 10000 + 1) + 10000);
                _obj.state = state;
                _obj.yourLocation = _locObj;
                _obj.date = new Date();

                _mySpots.plates.push(_obj);

                setLocalStorageData('my_plates',_mySpots);
                displaySpotsView();
        };
```

Figure 21

What this does:

- The **_s.** prefix allows us to access this function outside of the **SPOTTER** function.
- The **_s.setSpot** function takes an argument of a **state**
- We create a simple object literal with **var _obj = {};**
- We then assign values to various properties of the **_obj** (some of these values are obtained with code in later steps).
- We then **push** the **_obj** object in to an object called **_mySpots.plates** array property (we set this up at the very beginning of the module in **figure 19.**
- We then call **setLocalStorageData** (which we haven't created yet)
- We then call **displaySpotsView()** (which we haven't created yet)

Head spinning yet? No worries, it is pretty simple, just ask questions and check the code samples.

Step 13 – Remove a Plate

If we are going to add plates then we should be able to remove plates, so lets set that method up real quick. It is a little different than adding a plate, but similar principles.

In **figure 22** we have the **_r.removeSpot()** method. You will want to add this code inside of your **SPOTTER** function/module and save your **spotter.js** file.

```
_s.removeSpot = function(id){
                var i = 0, len = _mySpots.plates.length;
                for(i; i < len; i = i + 1){
                        if(id == _mySpots.plates[i].id){
                                _mySpots.plates.splice(i,1);
                                setLocalStorageData('my_plates',_mySpots);
                                displaySpotsView();
                                break;
                        }
                }
        };
```
Figure – 22

We again have a **_s.removeSpot()** method that takes an id and simply loops through the **_mySpots.plates** array and removes the item. It also updates our **localStorage** and updates the view as well.

Okay, so we have some add and remove things in place, lets jump out and add some event listeners and take a break from this file.

Step 14 – Adding Event Listeners

Our application has some functionality but it can't be used until we listen for a couple of events. We need a catalyst to get things moving so we will add two event handlers.

The first event handler listens for when there is a change made to the State Picker component. When the state of the select element changes a change event is fired. The listener in turn calls the **setSpot** method in our module.

The second event handler listens for a click on any element with the **class** attribute of **.remove-plate-icon** and then calls the **removeSpot** method.

You will notice in **figure** that both methods take an argument. But for now, place the code from **figure 23** inside of the **document.ready** function inside of **app.js.**

```
    $('#statePicker').live('change',function(e){
                    if( $('#statePicker').val() !== "Choose State") {
                            SPOTTER.setSpot($('#statePicker').val());
            }
    });

    $('.remove-plate-icon').live('click', function(e){
                    $('#'+e.currentTarget.parentElement.id).hide('slow');
                    SPOTTER.removeSpot(e.currentTarget.dataset.plateid);
    });
```
Figure - 23

NOTE - Every time a user clicks, types, moves the mouse, etc.. an event is fired. JavaScript has a standards way of listening for events, however Microsoft does it a little differently, so we are using jQuery to smooth that out for us. You can easily write your own cross browser event handler, but we will go with jQuery for now.

Step 15 – Template Handler

Thinking back to our templates we mentioned that you could manipulate them and pass data to them, so lets write a function to handle this situation. In **figure 24a** we have code that you will need to add to your **spotter.js** file inside the **SPOTTER** function. Go ahead and add it right after your **_s.removeSpot()** function.

```
function templateCall(data, tmpl){
                var _data = data, _template = tmpl;
                var source   = $(_template).html();
                var template = Handlebars.compile(source);
                var html = template(_data);

                return html;
}
```

Figure – 24a

What this does:

- We pass in two arguments, the data we want the template to have, and the id of the template we want to work with.
- We then use Handlebars to access the template and pass in the data.
- In the end we **return html;** This will be the template complete with data.

In **figure 24b** we see how we will populate the **States** template. Place this code right after the **function templateCall**

```
function reportPlateView(states){

                var _html = templateCall(states, '#spot-create-template');

                $('#spotFormPanel').append(_html).slideDown('slow');

        };
```
Figure – 24b

What this does:

- What this is doing is simply calling **function templateCall** passing in the **States** and the template id.
- The **templateCall** function return the template to **var _html.**
- We then use jQuery to append the **_html** template to the **spotFormPanel** in our view.

Rinse and repeat. **Figures 24c and 24d** do the same thing. You can place those right after your **reportPlateView** function

```
function displaySpotsView(){
        var _html = templateCall(_mySpots, '#spot-view-template');
        $('#mySpotsPanel').html(_html).slideDown('slow');
};
```
Figure – 24c

```
function myLocationView(obj){
                var _html = templateCall(obj, '#location-view-template');
                $('#latLonDisplay').html(_html).show('slow');

        };
```
Figure – 24d

Things are pretty hectic right now, but we are almost there. Two more items and we can take this for a test drive.

Step 16 – Persisting Data

In your **_s.init** function you called i**nitializeStorage();** but it doesn't exist. Let's take care of that right now.

We are going to use the HTML5 localStorage functionality to persist data between sessions (meaning that the data remains even after the user closes the browser).

LocalStorage stores values in a browser sandbox as **"key:value"** pairs for a specific domain. All values must be strings, but that's okay because of **JSON.stringify** and **JSON.parse** methods built into the browsers.

NOTE – JSON.stringify takes any JavaScript object and marshals it into a string. **JSON.parse** takes any string and attempts to marshal it into a JavaScript object. So by using these features we can take our **_mySpots** object and turn it into a string and store it in localStorage, and then pull it back out and turn it into an object. Very handy feature. As you will see in **figure 25b** and **25c.**

To initialize localStorage we want to first call it in our **_s.init** function. The good part is you have already done that. You can place the **initializeStorage** function , as seen in **figure 25a,** right after the **myLocationView** function.

```
	function initializeStorage(){
		if (!window.localStorage.getItem('my_plates')){
			setLocalStorageData('my_plates', _mySpots)
		}else {
			_mySpots = getLocalStorageData('my_plates');
		}
	};
```
Figure – 25a

What this does:

- We check to see if localStorage has an item called **my_plates**
- If it does not then we create the key **my_plates** and then pass in the data of **_mySpots**
- If the key **my_plates** exists then we set **mySpots** to the value of ***my*plates**

We could have simply used the native **getItem** and **setItem** methods of the localStorage object but we want to abstract it out a layer. Down the road this can give you greater freedom if you have to manipulate the data or use **JSON.stringify/parse.** This is why we have created the **getLocalStorageData** and **setLocalStorageData** functions, as seen in **figures 25b** and **25c** respectively.

You can add the code in **figures 25b and 25c** right after the **initializeStorage** function.

```
function getLocalStorageData(key){
		var _obj = JSON.parse(localStorage.getItem(key));
		return _obj;

	};
```

Figure – 25b

```
	function setLocalStorageData(key, val){
		localStorage.setItem(key,JSON.stringify(val));
	};
```

Figure – 25c

Step 17 – Getting The Location

In your **_s.init** function you called **initializeLocation();** but it doesn't exist. Let's take care of that right now.

We are going to use the HTML5 geoLocation functionality to watch for a person's location and store the information. The code in **figure 26** can be added right after the **setLocalStorageData** function.

```
function initializeLocation(){
 // makes a single call for location
// navigator.geolocation.getCurrentPosition(setLocationHandler, errorHandler);

 // watches postion
watchPositionId = navigator.geolocation.watchPosition(setLocationHandler,
errorHandler, {enableHighAccuracy:false, timeout:6000, maximumAge:5000});
};
```

Figure – 26a

What this does:
The **initializeLocation** function uses the **navigator.geolcation** object to get a location. There are two ways to do this, one is a one shot call, the other is a watch call that checks every so often to see if the position has changed.
On success the **setLocationHandler** in **figure 26b** is called.

If everything went well then the **setLocationHandler** is called and a **location** object is passed into it. As seen in **figure 26b.** You can place this code right after the **initializeLocation** function.

```
function setLocationHandler(pos){
                _locObj = pos;
                myLocationView(_locObj)

        };
```
Figure – 26b

What this does:

- This function sets our **_locObj** value.
- Then the **myLocationView** function is called, which in turn updates the view.

But if things go wrong we need an error handler, as seen in **figure 26c.** You may add this code right after the **setLocationHandler** function.

```
function errorHandler(str){
                console.log(str);
};
```
Figure – 26c

What this does:

- It receives and error obj or string that will write to the developer console. This is not ideal, you would want something more robust.

Well, if everything went well we should be able to take it for a test spin.

Conclusion:

There you have it, a whirlwind tour of building an application using only HTML/CSS/JavaScript. We covered a lot of different features and organization techniques.

Keep in mind that there are frameworks, testing, IDEs, and about a million ways to do something in JavaScript. This was just one approach.

Please feel free to pull the code from Git Hub any time. If you have questions please feel free to contact me via email at adrianpomilio@gmail.com or on twitter @adrianpomilio