Libraries, Frameworks and Template engines

Advanced JavaScript for Web Sites and Web Applications

What a complex web we weave...

Libraries, Frameworks and Template engines

- There are many third party libraries and frameworks available for JavaScript
  - jQuery, Angular, Mustache, Handlebars, React, Ember, Zepto, Underscore, Lodash, Backbone, Knockout... and many, many more!
- Why is this?

History

- In the early 2000s, writing code in JavaScript was painful!
  - Browser differences
  - Limitations of the language
- The first generation of frameworks and libraries aimed to make the process a lot easier

First generation libraries

- Libraries such as *Prototype JS*, *Mootools*, *Scriptaculous* and *jQuery* were created to:
  - Patch the differences between the browsers (E.g.: Ajax requests)
  - Provide convenience methods to perform otherwise long and convoluted tasks (E.g.: iterating collections)
  - Provide UI enhancements that were not possible with CSS at the time (E.g.: animations)

First generation libraries

- These libraries revolutionised web development and sowed the seeds for the modern web we work with today
  - Ajax-driven websites
  - Client-side web applications
  - A gentle learning curve

The current state of play

- The JavaScript we use today is very different to that which we used in the early 2000s
  - Browsers are more compliant with standards
  - ECMA script has added new features to simplify common tasks
  - CSS has many new features
- For this reason, the 1st gen libraries have fallen out of favour with some developers
  - But probably aren't going to disappear just yet!

Modern libraries and frameworks

- The libraries and frameworks we see today are very different to the 1st gen
- They are also very different to each other
  - Each focuses on one particular aspect of the development/programming process
- **Libraries:** collections of functions that we can use within our apps
- **Frameworks:** Same as libraries but also provide ready made code structures and patterns

Libraries

- Many libraries provide generalised functional programming helpers
  - Array tools (sorting, filtering, searching), Iteration helpers, Validation tools
- Generally, they don’t tend to dictate the way we use their functions
  - So we can use them within our own program structure
- E.g.: Underscore, Lodash

Underscore

"[...] provides a whole mess of useful functional programming helpers [...]"

- Underscore is a tool kit that helps you work with and manipulate data.
- It simplifies a lot of JavaScript functionality that is otherwise tedious to code.
- Documentation

Underscore

- For example, Underscore provides a useful `_.each()` method which allows us to loop over collections.
- We simply pass it a collection and a callback function
- It will iterate the collection, passing each item to the callback function

Underscore _.each() example

```
// Function is called for each item in collection
// Current item is passed to function as argument
function logData(c_item) {
    console.log(c_item);
}

// Data to iterate
var data = [1, 2, 3];

// Pass data and callback to _.each
_.each(data, logData);
```

A loop for every collection...

- To iterate JavaScript arrays, we use a `for` loop.
- To iterate object properties, we use a `for in` loop.
- To iterate a jQuery collection, we use the jQuery `each()` method
- ... which can make life difficult for us!

Underscore - type agnostic loops

- With Underscore's `_.each()` method, you can iterate arrays, object properties, nodeLists, jQuery collections…
- … all with the same function!

Underscore example

- In the Underscore folder, have a look at `underscore.js` where some collection variables have been defined.
- We can use the `_.each()` function to loop over each collection and log the *items*.
  - Note, the *item* will not always be of the same *type*

Underscore

- This is just one example of Underscore providing a simple interface for a common task.
- Underscore contains many similar functions which can greatly simplify your code
- More info

JavaScript and HTML

- As we have seen, creating HTML from within our JavaScript code is tedious.
- Whether we *concatenate* or use the *array/join* technique the resulting code is hard to read/maintain/modify
- Additionally, it becomes difficult to reuse our scripts because the HTML output is hard-coded

Solving the HTML problem

- JavaScript *Template engines* offer a solution to this problem

Template engines

- *Template engines* provide ways to easily transform *data* into html
  - *data* can be: objects, arrays, functions, JSON strings etc.
- They usually work by introducing template *tags* that can be used directly in the JavaScript or HTML code.

Mustache

- One popular template framework is **Mustache**
  - Documentation
- It has been implemented in many programming languages, including JavaScript.
- Given a *template*, Mustache will search for template *tags* within it, and replace them with corresponding data.

Mustache templates

- Templates are strings that contain Mustache tags.
  - usually combined with HTML or text.
- A Mustache tag consists of a *label*, enclosed in double curly braces:
  - E.g.: `{{label}}`

Mustache example - the template

- From the Mustache documentation…
- A *template* will look something like this:

```
// A Mustache template:
var tpl = "<p>{{title}} spends {{calc}}</p>";
```

- `{{title}}` and `{{calc}}` are placeholders, which will be replaced with data

Mustache example - the data

- The data source will be an object, containing the data and/or logic to display that data:

```
var myData = {
    title: "Joe",
    calc: function () {
        return 2 + 4;
    }
};
```

- Notice how the property names match the template placeholders?

Mustache example - data meets template

- To merge the *data* with the *template*, use Mustache's `render()` method, passing it the template and the data object:

```
// Render the template
var output = Mustache.render(tpl, myData);
```

Mustache example - using the result

- `render()` returns a string, containing the template HTML, with all placeholders replaced with data
- We can insert this string into the DOM with `insertAdjacentHTML` or similar

Mustache loops

- If the data contains a list of multiple items you can iterate them using Mustache *enumerable sections*
- *enumerable sections* start with the list identifier, preceded by a #
- They end with the list identifier, preceded by a \
- Template code within the *enumerable section* is rendered once for each item in the list

Mustache loops example - the data

```
// The "list" is stored under "people"
// Each item in the list is an object
// Each object has one property: "name"
var myData = {
    "people": [
        { "name": "John" },
        { "name": "Paul" },
        { "name": "George" }
    ]
};
```

Mustache loops example - the template

```
// Template with enum section for "people" list.
// Each item in list has a "name" property.
// The inner template "<p>{{name}}</p>" will
// be applied to each item in list.
var tpl = "{{#people}}<p>{{name}}</p>{{/people}}";

// Render the template
var output = Mustache.render(tpl, myData);
```

Mustache loops example - the result

- After calling `render()`, the `output` variable will contain:

```
<p>John</p>
<p>Paul</p>
<p>George<p>
```

Mustache loops and arrays

- Consider this array of data:

```
var data = ["apple", "banana", "pear"];
```

- Unlike the object we saw earlier, the array and it's elements have no "labels"
- To reference elements of this type with Mustache, we use a dot (.).

Mustache loops and arrays

```
// The JavaScript:
data = ["apple", "banana", "pear"];
tpl = "{{#.}} <p>{{.}}</p> {{/.}}";
output = Mustache.render(tpl, data);
```

```
<!-- The Result: -->
<p>apple</p>
<p>banana</p>
<p>pear</p>
```

Mustache loops exercise

- Download the exercises document from Moodle and do *Exercise 1*

Template storage

- In the examples, we have placed the template code in a regular string within our JavaScript
- But, we can also store it within the HTML…

Unknown elements

- When the browser encounters a `script` tag with an unknown `type` attribute, it will not attempt to execute it.
- But it will add it to the DOM, allowing us to access it via JavaScript

Template storage - in the HTML

```
<html>
<body>
    <!-- Normal HTML code here... -->

    <!-- Browser ignores this: -->
    <script id="myTpl" type="x-tmpl-mustache">
        Hello {{name}}!
    </script>

</body>
</html>
```

The template code

- To use the template in our script, we retrieve it with regular DOM manipulation techniques:

```
var tplTag = document.getElementById('myTpl');
var tpl = templateWrapper.innerHTML;
```

Dynamically loading templates

- You can also store your templates in separate files, and load them when needed.
- E.g.: Using an Ajax call with the content type set to `html` or `text`
- Of note, templates are just *strings*, so anything that can produce a string can produce a template!

Other template engines

- Several other template engines exist for JavaScript
  - E.g. Handlebars, JSRender
- Many libraries and frameworks also have a template engine built in to them
  - E.g. Underscore, jQuery, Angular

Native templating

- However, recent versions of ECMA script support template literals
- So it may be that, in the future, we will not need third party libraries for this task either!

Frameworks

- Frameworks solve the issues you run into when working with complex applications.
- They let you easily manage complicated interactions between your data and the DOM (elements and events).
- Frameworks implement common *design patterns* and provide a means for us to give structure to our code easily.

Frameworks

- A lot of frameworks are based on the concept of MVC (Model, View, Controller)…

MVC - In simplistic terms...

- The **model** is the core data of the application, and the functions that act upon it (sorting/filtering/etc.)
- The **view** is the user interface for the application (normally, HTML)
- The **controller** is the code that ties the two together

MVC

- The *controller* reacts to user actions and changes in application *state*
- It tells the *model* about these events, so it can update it's data
- The *view* then updates itself, to reflect changes in the *model*
- Note, there are many differing opinions as to how these elements should work together!

Why frameworks are useful

- Remember the Ajax product listing application we built previously?
- When dealing with web pages that involve a lot of interaction with the user, where:
  - the data on the page is updated quickly and regularly,
  - you need to watch a lot of events,
  - you need to generate/update HTML
- … writing code as we did before quickly becomes a nightmare!

What is in a framework?

- Most major frameworks will include:
  - data manipulation tools (similar to Underscore)
  - a template engine (similar to Mustache)
- And concepts of:
  - views, event management, data modeling, routing

Popular frameworks

- Some popular frameworks:
  - Backbone
  - Angular
  - Ember
  - React
  - and many, many, many, many more….

Angular

- We will take a brief look at Angular which is a popular choice for single page web apps

Angular concepts:

- **Models**: hold information to be displayed by a view. ie. The data
- **Directives**: special tags in Angular that bind application code to a section of the page.
- **View**: The data output on the page (DOM)
- **Controller**: logic that sits between the data and the view. (functionality for a section of the page)

Angular example

- Add this HTML to `angular-test.html` from the workshop files, somewhere within the body element:

```
<div ng-app="">
    {{1 + 1}}
    <br />
    {{"Hello" + " World"}}
    <br />
    {{3 * 3}}
</div>
```

- View the page in your browser...

Angular example

- `ng-app=""` is a *directive* that tells angular that this div contains an Angular app.
- The `{{ }}` symbols represent template tags
- The data enclosed in `{{ }}` are *expressions*, similar to normal JavaScript expressions

Angular example 2

- Remove the previous snippet of html and add this instead:

```
<div ng-app="">
    <input type="text" ng-model="data.message">
    <h2>{{data.message}}</h2>
</div>
```

- View the page in your browser and type something in the input field

Angular example 2

- In the previous example, the `<input />` and `<h2>` elements are bound to the `data.message` property of the application

Angular

- You'll notice that we have used Angular without writing any JavaScript code!
- We are loading the Angular library in our page, but are using HTML attributes and Angular syntax to enable the functionality.
- As we can see, Angular uses its own templating system, which is fairly similar to Mustache.

Angular example 2+

- We can enhance our previous example…
- Replace the previous snippet of HTML with the code on the next slide.
- Then view the page in your browser and type something in the input fields

Angular example 2+

```
<div ng-app="">
    <label>Name</label>
    <input type="text" ng-model="data.name">
    <label>Job:</label>
    <input type="text" ng-model="data.job">
    <h2>Name is: {{data.name}},
        Job is: {{data.job}}</h2>
</div>
```

Angular exercise

- Now do *Exercise 2* from the exercises document

Angular Templates and Directives

- What we have just seen in Angular are *Templates* and *Directives*.
- **ng-app=“”** is a directive that triggers the Angular app.
- **ng-model=“”** is a directive that links the input field to the *model*.
- The **template** is our html, augmented with the directives above.

Angular Controller

- Add this snippet of HTML to **angular-test.html**:

```
<div ng-app="">
    <div ng-controller="myController">
        <h1>Hello {{data.name}}!</h1>
        <p>You are {{data.age}} years old
        and you work as a {{data.job}}</p>
    </div>
</div>
```

- You have just set up an Angular Controller...

Angular Controller

- Now add this snippet of code to **angular.js**:

```
function myController($scope){
    $scope.data = {
        name: "Joe Bloggs",
        age: 36,
        job: "Programmer"
    };
}
```

- Now view angular.html in your browser

Angular Controller

- In the JavaScript code, `$scope` represents the application model
- The function is named `myController`, which is the value of the `ng-controller` attribute in the HTML
  - This binds our function to the angular app

Angular Controller

- In our example, the data (“Joe Bloggs”, 36, “Programmer”) comes from our controller function.
- We have hard coded the data in the function, but it could be coming from:
  - an Ajax call to an api
  - The result of another function
  - etc.

Summing up

- Using frameworks and libraries can assist us in building complex applications quickly and with a minimum of stress.
- But there is a price to pay…

Summing up

- Each library we use causes at least one more *request* to be made by the browser
  - to load the library code
- The overall *size* of our web pages increases with every library we use
  - jQuery = 80-150kb, jQuery UI = ~500kb, Angular = ~80kb

Summing up

- We can resolve some of these issues by employing various techniques
  - Code minification, Placing all the libraries in one file
- But we still need to be vigilant in order to keep things under control.