ADVANCED JAVASCRIPT FOR WEB SITES AND WEB APPLICATIONS

SESSION 8 - FRAMEWORKS, TOOLS AND BEST PRACTICE

Avoid inline JavaScript

- not flexible enough
- danger of polluting the global name space
- bad separation of content and behaviour

Include html script tags at the end of the page

- doesn't block page rendering early on
- good for front end code performance

Use as few JavaScript files as possible

- limits the number of resources the browser needs to load and the number of http request
- limits the blocking of loading each file separately

Minify your code

- using a tool like **jscompress**
- reduces the size of the files the browser had to download

FILE ORGANISATION

When building a large application, there are different ways to organise your scripts and load them.

The aim is to have as few files as possible for the browser to load and to optimise how much the browser has to load for each page.

Lazy Load example

Using a tool such as require.js, you can load extra scripts dynamically, after the DOM has finished loading.

With this method, you can put the essential scripts required to create the interface intially in a main script file.

All other, non essential scripts can be loaded dynamically after the DOM is loaded.

Your initial file is smaller for the browser to load. The user souldn't notice the files being loaded later on, for components that they will not interact with straight away.

Repository Structure

```
- project
  -- index.html
  -- css
  -- scripts
      -- common.js
      -- vendor
          -- jquery.js
          -- underscore.js
      -- modules
          -- event.js
          -- tabs.js
          -- home.js
          -- carousel.js
          -- widgets.js
```

`common.js` would be loaded on all pages.

`event.js`, `tabs.js`, `home.js`, `carousel.js`, `widgets.js` would be loaded dynamically, after the DOM load.

Tools for this: `yepnope` (deprecated), `requirejs`.

The global object method

You create a global object, and you add methods to this object in each modules/files you build.

The code is broken down into modules and easy to organise, but if you do not add concatenation or module loading etc. the files still need to be loaded through script tags

This method is also sensitive to the order the code needs to execute in.

This doesn't require any server side tools or module loader tools.

The one file method

With this method, your final code is pushed and concatenated into one file, that contains every script needed for the entire application/site.

Easy to implement but the file can become very large, and every script is loaded on every page of the site, even when not needed. Ideally though, the file is cached by the browser.

Tools for this: `gulp`, `Browserify` etc.

Repository Structure

```
- project
  -- index.html
  -- css
  -- scripts
      -- common.js
      -- vendor
          -- jquery.js
          -- underscore.js
      -- modules
          -- event.js
          -- tabs.js
          -- home.js
          -- carousel.js
          -- widgets.js
```

Server side, use a tool like `gulp` or `Browserify` to merge all your files into one.

The many files method

With this method, your final code is pushed into one file, but this file is different for different sections of the site.

For example, you would have one version of the file made up of the core scripts and extra script for one section and another version with core scripts and scripts for another section.

The advantage of this method: one JavaScript file but smaller in size. The file can't be cached though.

Server side, use a tool like `gulp`, `Browserify`, or `requirejs` to merge all your files into different "final" files that mix and match different modules.

Module organisation

In the previous methods, your code is broken down into different files or modules, which is good, but the code is still dependant on the oder the files are included in.

You can go further and have the different modules loaded by a module manager, such as `requirejs` or `Browserify`.

The advantage of doing this is that your modules aren't loaded by concatenation or by script tags in the oroder they are included.

Modules are loaded based on their dependencies and on when they are called by other parts of the app.

The module loader will go through the entire config, establish dependencies and use that to generate your script file.

You still end up with one bundle JavaScript file, but when writing the modules, the load order doesn't matter as much.

As well as organising your files and concatenate them into a single file or multiple files, most tools will optimise this process.

Some common optimisation tasks include:

- minifying the code
- linting the code
- checking the synthax
- reporting errors

A tool such as gulp is very efficient at doing this.

Quick gulp example

- task / build system
- good at automating routine development tasks
- relies heavely on plugins
- ....and node

It needs to be installed on the machine or server it's running on (using npm or other package manager).

It works by creating a `gulp.js` file, that tells the service what to run and what to do when exectuted.

This file then gets run when you run the command `gulp` on that folder.

I can add this in my gulp gulp file:

```
var gulp = require("gulp"),
   uglify = require("gulp-uglify");

gulp.task("minify", function () {
   gulp.src("/js/CITY_R.js")
      .pipe(uglify())
      .pipe(gulp.dest("build"))
});
```

And run the file by calling `gulp` and the task you want to run, in this case `minify`:

```
gulp minify
```

You can add more tasks to the gulp file, by simply declaring them:

```
gulp.task("concat", function () {
   return gulp.src("js/*.js")
      .pipe(concat("app.js"))
      .pipe(gulp.dest("build"));
});
```

And making sure you call the dependent plugin:

```
concat = require("gulp-concat")
```

...or you can use gulp streams to do all this in one step:

```
var gulp = require("gulp"),
    uglify = require("gulp-uglify"),
    concat = require("gulp-concat");

gulp.task("minify-concat", function () {
   gulp.src("js/*.js")
      .pipe(concat("app.js"))
      .pipe(uglify())
      .pipe(gulp.dest("build"))
});
```

JAVASCRIPT FRAMEWORKS

- code organisation
- usefull for large sites, apps, app-like pages
- more performant at doing repetitive things like DOM manipulation
- not dependant on jQuery or other libraries, can replace them in some cases

Underscore

"provides a whole mess of useful functional programming helpers"

documentation

HTML templating / rendering

The **Mustache example**

From the Mustache documentation:

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

var template = "{{title}} spends {{calc}}";

var output = Mustache.render(template, view);
```

In this example, `render` takes two arguments, a template and a view object that holds the data and logic behind the template.

Templates are JavaScript strings that contain Mustache tags (" {{}} ") and HTML or text.

If the view is a list of multiple item you can loop over it using this synthax in Mustache:

```
var people = {
  "people": [
    { "name": "Rik" },
    { "name": "Larry" },
    { "name": "Gilles" },
    { "name": "Aris" }
  ]
};

var template =
  "{{#people}}
    {{name}}
  {{/people}}";
```

Try to create a simple script to loop over this object:

```
var people = {
  "people": [
    { "name": "Rik", "age": "35" },
    { "name": "Larry", "age": "35" },
    { "name": "Gilles", "age": "34" },
    { "name": "Aris", "age": "33" }
  ]
};
```

and add the following html to the DOM:

```
<ul>
  <li>name: Rik, age: 35</li>
  <li>name: Larry, age: 35</li>
  <li>name: Gilles, age: 34</li>
  <li>name: Aris, age: 33</li>
</ul>
```

```
var people = {
  "people": [
    { "name": "Rik", "age": "35" },
    { "name": "Larry", "age": "35" },
    { "name": "Gilles", "age": "34" },
    { "name": "Aris", "age": "33" }
  ]
};

var template = "{{#people}}<li>name: {{name}}, age: {{age}}</li>{{/people

var output = [];

output.push("<ul>");
output.push(Mustache.render(template, people));
output.push("</ul>");
output = output.join("");

document.querySelectorAll(".header")[0].insertAdjacentHTML("afterend", ou
```

Angular

The best way to get started with Angular and how it works or what it does is with an example.

Add this HTML to the angular.html file, in the body element:

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

`ng-app=""` tells the page that this div wraps an Angular app, and that inside this div, we'll be able to use Angular.

Now let's do something slightly more interesting.

Remove the previous bit of html and add this instead:

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

You'll notice that we haven't written any JavaScript ourselves. We're loading the Angular library in the HTML, but only using HTML attributes and Angular synthax.

Angular uses its own templating system, fairly similar to Mustache.

We can make the previous example more complicated:

```
<div ng-app="">
  <input type="text" placeholder="Your name" ng-model="data.name">
  <input type="text" placeholder="Your job title" ng-model="data.job">
  <h2>Your name is: {{data.name}}, your job is: {{data.job}}</h2>
</div>
```

In this page, there are a 3 classes in the css called "red", "blue" and "yellow". Each assigns a background colour to an element.

Can you try to give a box a different colour based on the text added to the input field?

You can use this HTML for the box:

```
<div class="box">
  I've got a border,
  <br />
  give me a background colour!
</div>
```

```
<div ng-app="">
  <input type="text" placeholder="enter a colour" ng-model="data.colour">
  <div class="box {{data.colour}}">
    I've got a border, <br /> give me a background colour!
  </div>
</div>
```

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 interacts with the value of the input field.

The template is our html, augmented with the directives above.

Remove the previous HTML from angular.html and add this:

```
 <div ng-app="">
  <div ng-controller="myController">
    <h1>{{data.message + " world"}}</h1>
  </div>
</div>
```

We have introduced an Angular `Controller`, that needs to be added to our script.

This time, in angular.js, add this bit of code:

```
function myController($scope){
  $scope.data = {message: "Hello"};
}
```

In the previous example, the data ("Hello world") doesn't come from the user input, but from the JavaScript, through our controller function.

here the data is hard coded in the script, but it could be coming from an Ajax call to an api, it could be generated through another function etc.

More Angular fun:

Egghead

You can also have a look at:

- **Backbone.js**
- **Knockout**
- **Ember**
- **Maria**