

Angular Best Practices

User Group Talk Transcript

Talk given at **AngularJS MountainView** User Group on 11th December 2012



Video: <http://www.youtube.com/watch?v=ZhUv0spHCY>

Video Transcription by **Ian Smith** (<http://twitter.com/fastandfluid>)

Disclaimer: The original content is copyright of the original “free to download” video published as indicated by the links to the original source material above. Any mistakes, opinions or views in that content are those of the original presenter. Any mistakes in the actual transcription of that content are the fault of the transcriber. Please email ian.smith@fastandfluid.com if you spot any errors.

Contents

Angular Best Practices User Group Talk Transcript	1
Introduction	2
Directory Structure	3
Zen Moment 1: Dependency Injection	7
Script Loading.....	8
Zen Moment 2: Wrap Callbacks for 3 rd Party API into \$apply	13
Flash of Unstyled Content.....	14
Minification and Compilation	17
Zen Moment 3: Don't Fight the HTMLExtendIt	21
Templates.....	22
Zen Moment 4: Separate Presentation and Business Logic.....	26
Structuring Business Logic	27
Zen Moment 5: Treat scope as read-only in templates, write-only in controllers	29
Scope.....	30
Zen Moment 6: The \$watch getter function must always be fast and idempotent.....	33
Structuring Modules	34
Zen Moment 7: Develop with non-minified libraries	36
Deployment Techniques	37
Q&A.....	38

Introduction

Let's talk about Best Practices: What is "Best Practices" and why should you do it?

Best Practices is things that people kind of figure out over time as a good idea to do because it helps maintain the project. It helps when people come on board to figure out how the project is structured and also when you go to different projects it helps you to keep in sync and say "Yes. This is a good idea to do everywhere."


So "what are the kind of things that we have discovered over time that are good for Angular?" is a good question that we're trying to answer.

It's hard to have global statements that "this is always good" or "this is always bad" what I'm going to try to do is say "This is the kind of options that we think you have, and these are the kind of trade-offs that these options provide and so only you can make the best decision as to which one you think is best fit for yourself."

Directory Structure

Directory Structure

- Do you need it?
 - Not really, but...
 - Why invent the wheel when you can generate?
 - Working with others
 - Not forgetting important components
- Use what is there:
 - [Yeoman](#)
 - [angular-seed](#)



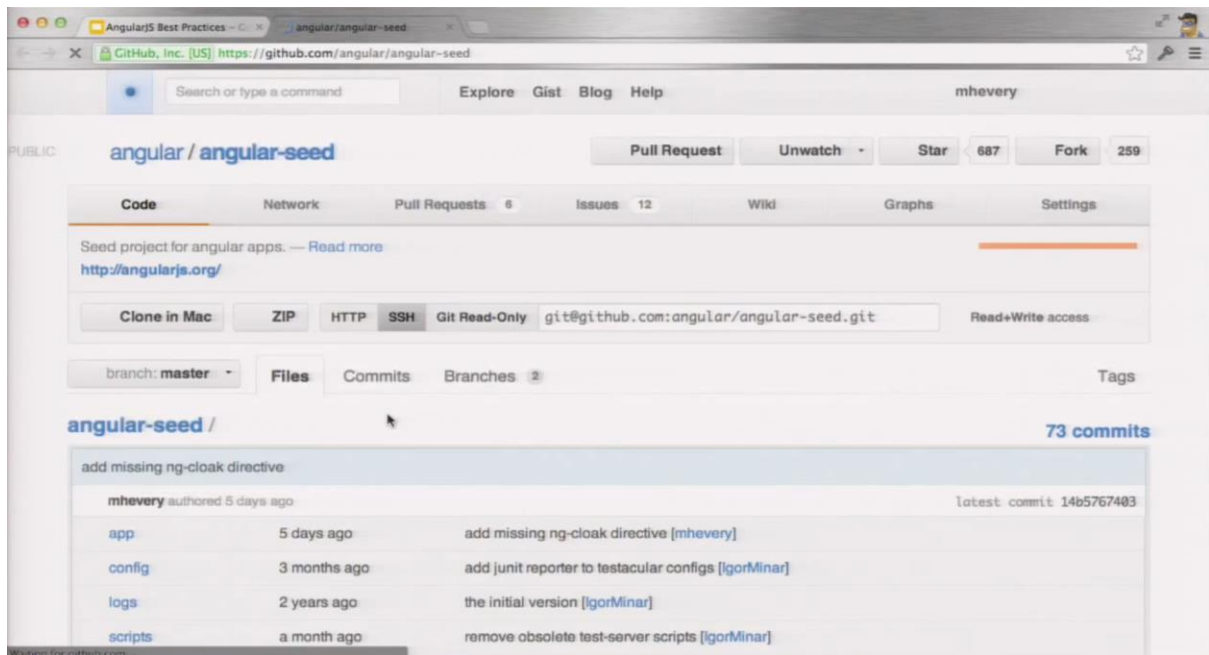
01:43

So the first question that oftentimes you want to answer is “How do I structure the application?”

So, first of all “Does it really matter the way you structure the application?”. Well to some degree it doesn’t. One structure is as good as another, but as I said once you structure it in a particular way and other people structure it the same way, it makes it easier to leave one project and join another one.

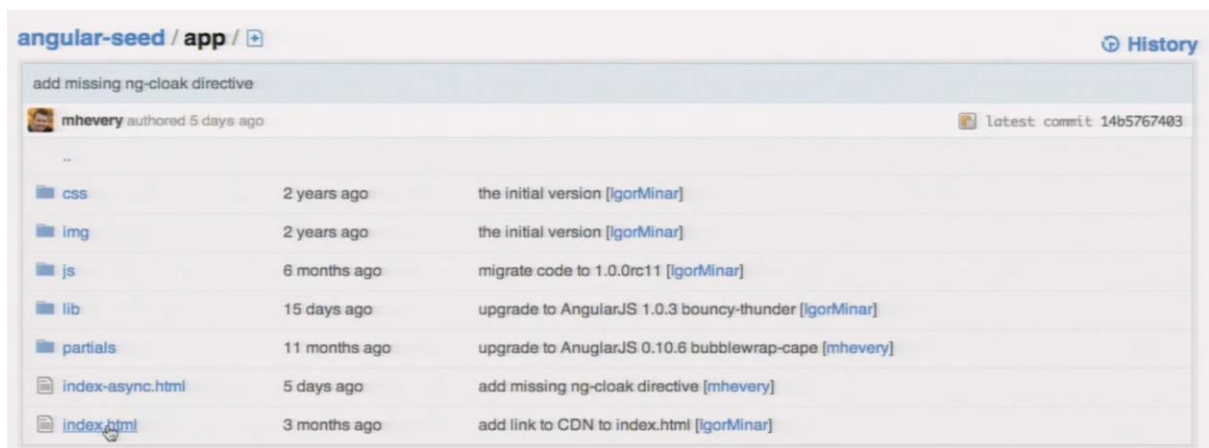
Convention over configuration is the big thing that Ruby brought to this world and we kind of believe in that as well.

We created a particular structure. It’s called **angular-seed**. It’s a GitHub project which you can go and check out. It’s at <http://github.com/angular/angular-seed>

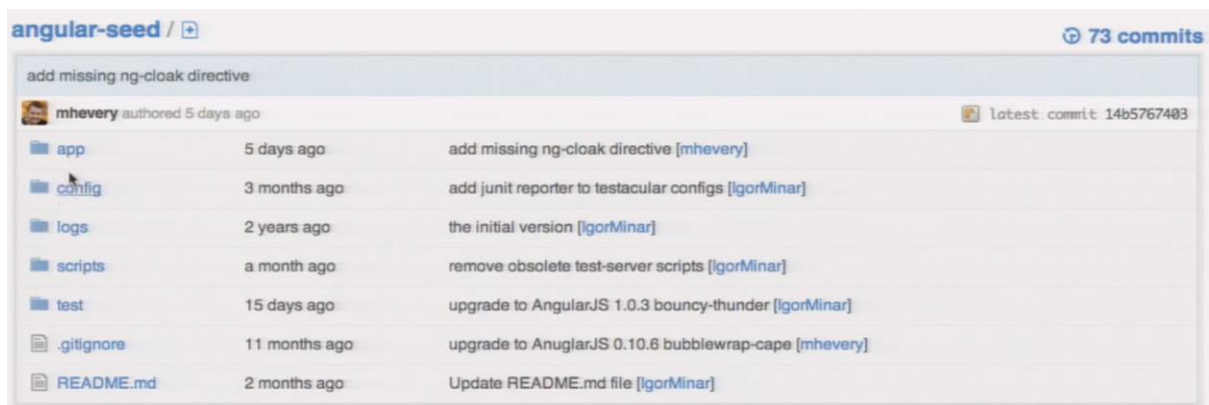


It basically sets up a directory structure where you have everything that you deploy in the **app** folder.

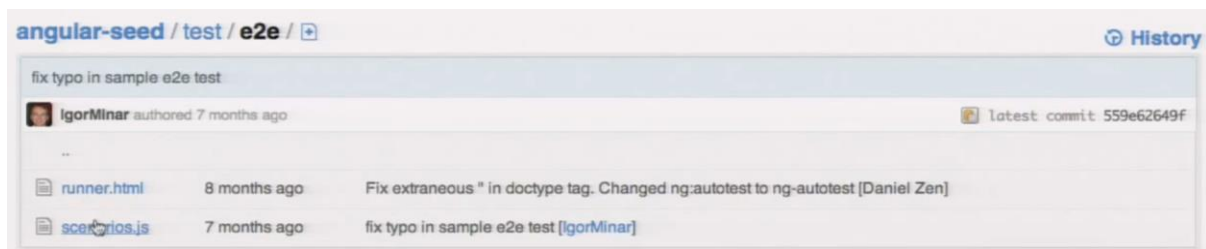
It has index.html both synchronous and asynchronous – we'll talk about the script loading later.



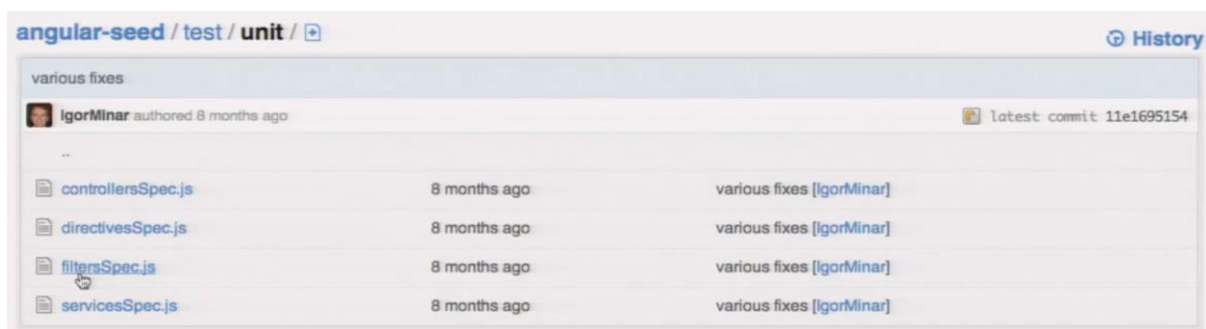
All the resources are pretty close to the directory structure, but more importantly it has the whole environment for development.



Anything outside the **app** folder is things that you need for development purposes but you don't actually deploy. The most important one of those is **test**. We have examples of end-to-end tests: how to write them and set them up.



Similarly we have an example of a unit test: how to set them up and run them etc.



We provide you with basically everything you need to get a basic “Hello world” application going and running.

So that's **Angular-seed** and we basically recommend that you clone the repo and go from there.

Angular-seed was actually created way before this other awesome project came along called **Yeoman**. **Yeoman** is a kind of replacement that we recommend people doing because **angular-seed** is a starting point. Once you clone it and start working with it it mutates and if you come up with a better practice you can't take the benefit of it because you can't go back in time and check out a different version of **angular-seed**.

Yeoman is kind of a dynamic directory structure.

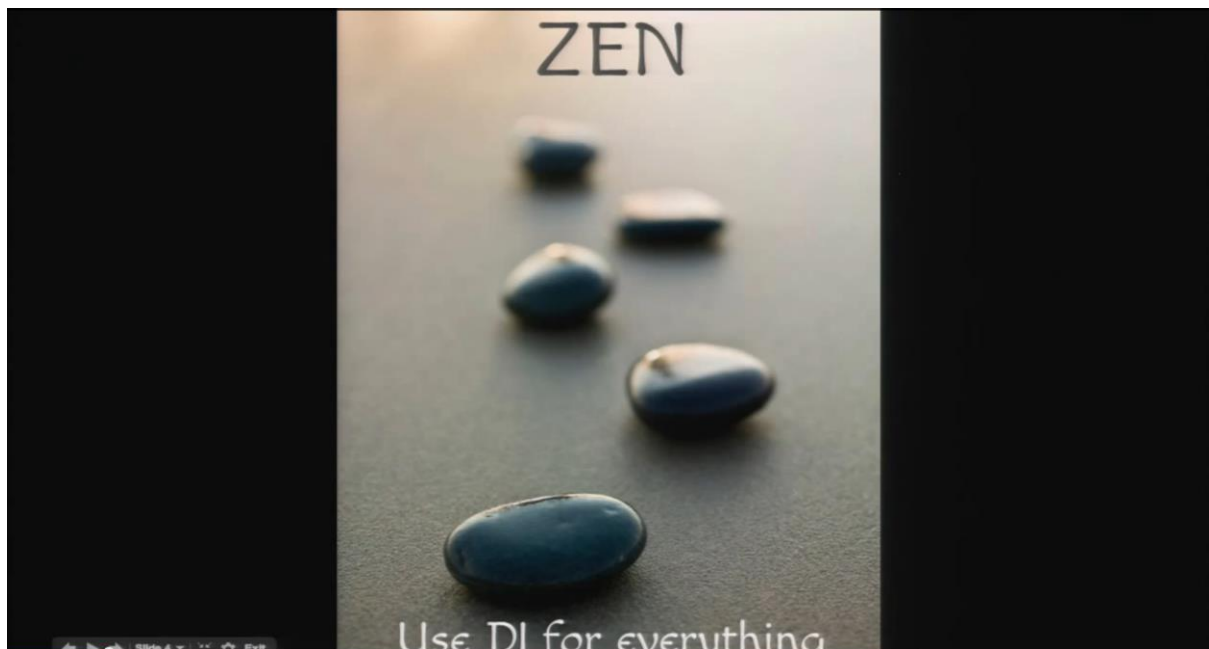


03:54

With **Yeoman** what you can do is... it first of all sets up your directory structure similar to **angular-seed**, but also it sets up dynamic commands that allow you to add controllers, remove controllers, add directives and all in a kind of automated fashion. They also allow you to set up your testing environment, run your tests, manage your dependencies, and I'm sure a bunch of other things in here that I'm forgetting in here as well.

We actually recommend Yeoman. A few months ago we had a technical talk, we discussed Yeoman, Brian did a wonderful job and that's available on the Angular channel. Check it out and find out more information about Yeoman and how to do it.

Zen Moment 1: Dependency Injection



Let's take a break for a moment, and have a little Zen moment over here.

One of the unique things about Angular is that we have **dependency injection**. It's the thing that actually assembles the application and makes so that you don't actually have to name everything.

This is a very unique point of view because what it allows you to do is it allows you to assemble the application in a different way for different purposes such as testing for example.


Or if you discover there's a service that you discover we've implemented in a different way not quite to your liking you can very easily with direct injection replace it with some other service that does something that you like better.

Dependency Injection is the core of everything and you should definitely embrace it and use it because it's what Angular is all about.

Script Loading

Script Loading

- Options:
 - <script> tags at the bottom
 - [ng-app](#) bootstrap
 - [builtwith.angularjs.org](#)
 - AMD (such as [require.js](#))
 - [manual](#) bootstrap



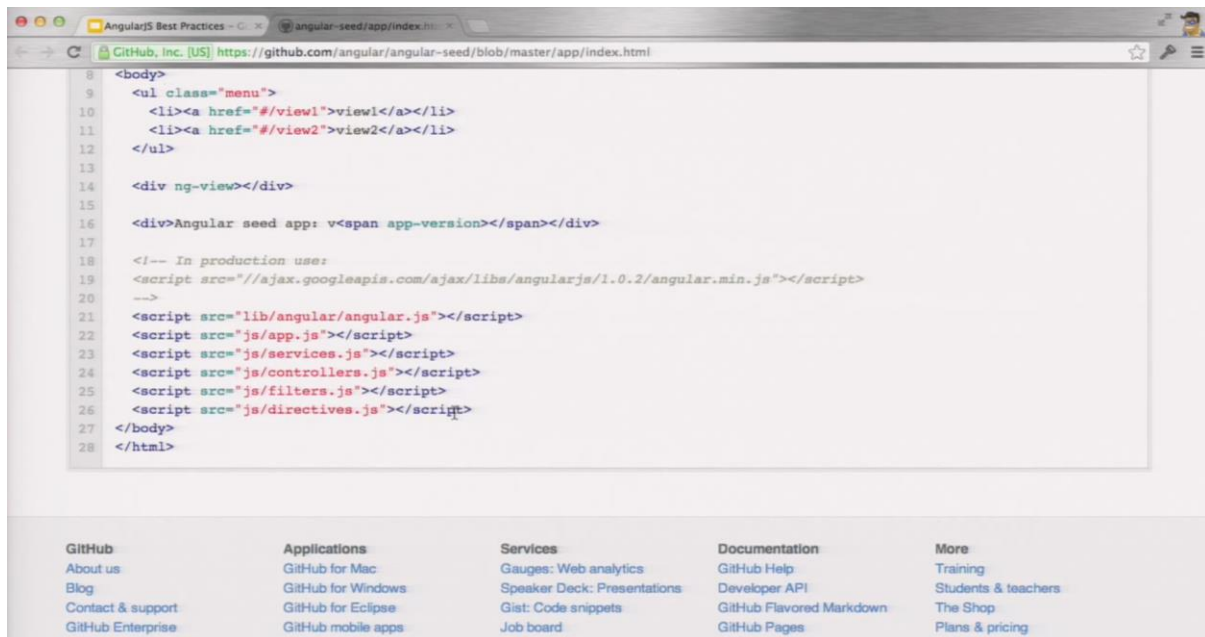
05:36

Let's talk about how we load scripts.

You have a couple of choices in loading the scripts. You can use the <script> tag and you can use something like a centralised module definition like **requireJS**. They both have their advantages and disadvantages.

Let's talk first about a simple way of using <script> tags.

If you check out <http://builtwith.angularjs.org> .. oops.. if you check out our seed the index.html has our <script> tags at the bottom of the page. The reason we want to put the <script> tags at the bottom is because <script> tag loading is blocking. If you have the <script> tags in <head> that actually prevents the browser from rendering the page until all the script tags are loaded.

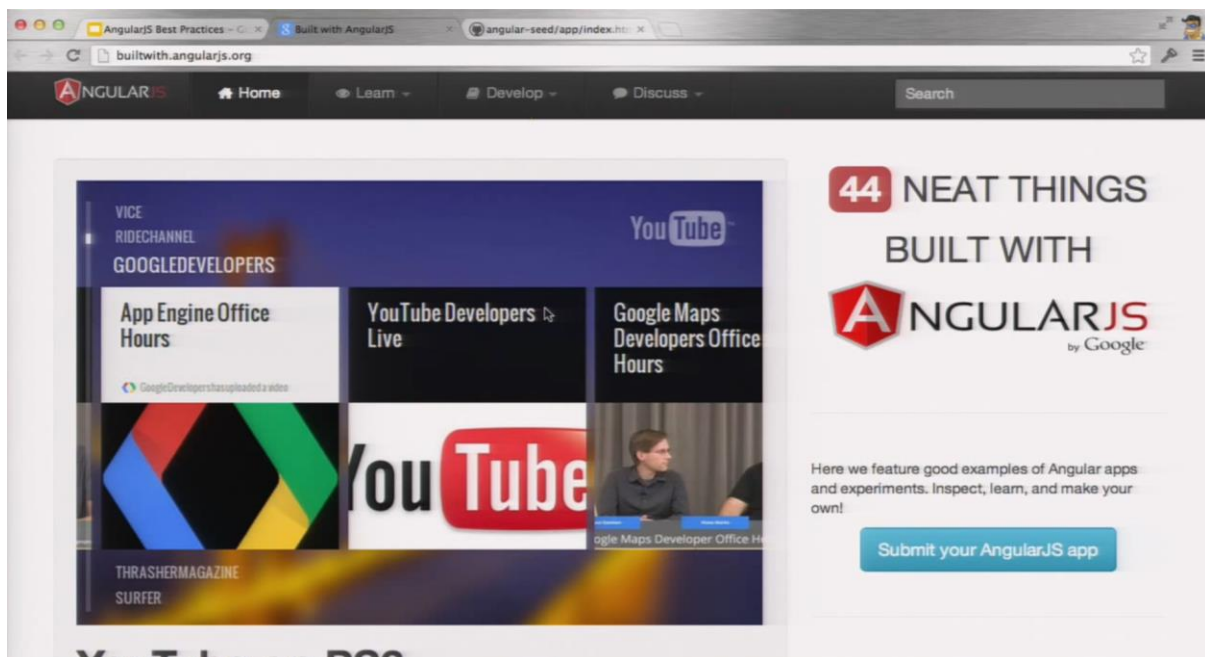


```
8 <body>
9   <ul class="menu">
10     <li><a href="#/view1">view1</a></li>
11     <li><a href="#/view2">view2</a></li>
12   </ul>
13
14   <div ng-view></div>
15
16   <div>Angular seed app: v<span app-version></span></div>
17
18   <!-- In production use:
19   <script src="//ajax.googleapis.com/ajax/libs/angularjs/1.0.2/angular.min.js"></script>
20   -->
21   <script src="lib/angular/angular.js"></script>
22   <script src="js/app.js"></script>
23   <script src="js/services.js"></script>
24   <script src="js/controllers.js"></script>
25   <script src="js/filters.js"></script>
26   <script src="js/directives.js"></script>
27 </body>
28 </html>
```

By putting it on the end you can show something useful to the user while he's waiting for the rest of the pages to come in.

With dynamic applications or AJAX applications like Angular the benefit of putting the script tags at the end is a little diminished because usually Angular is the thing that will render your page so you can't really display too much useful stuff while it's being loaded but there are a couple of things you can do. I'm going to show you an example of that.

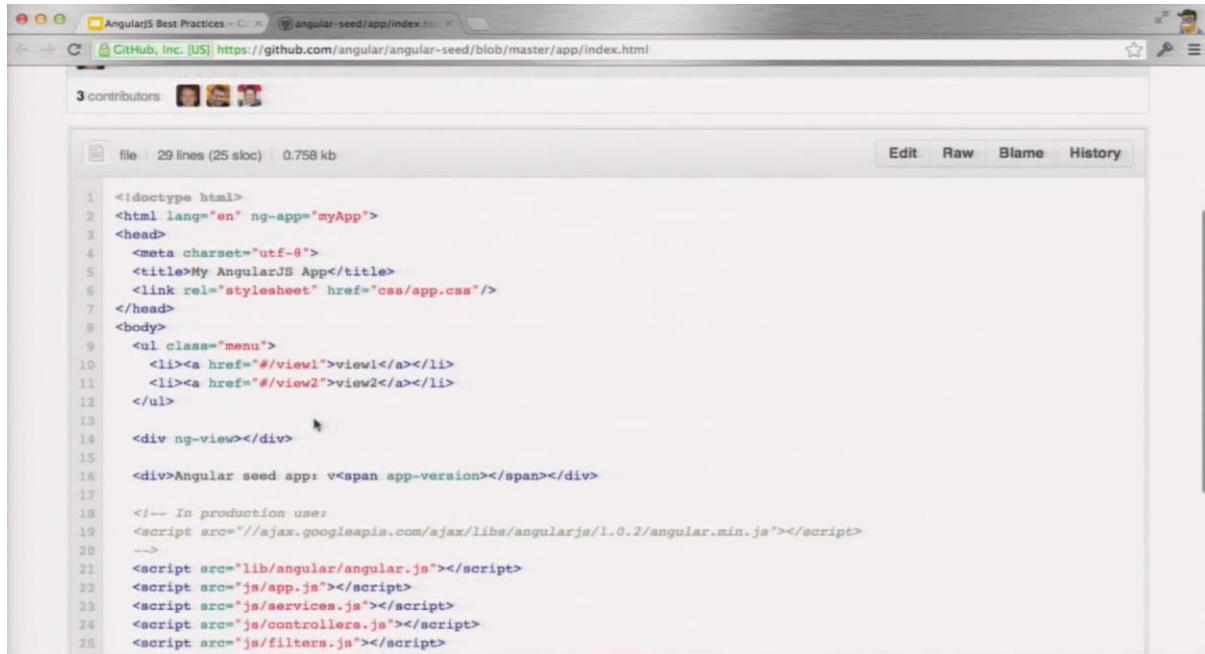
It's our <http://builtwith.angularjs.org> page.



If I refresh pay attention to this "44 neat things built with Angular". Notice it renders a question mark first, and then it renders the actual number.

What's happening is we load the template which just contains the "?" because we don't know what the number is, and at the bottom we load the `<script>` tag which is what bootstraps the whole process.

Going back to this...



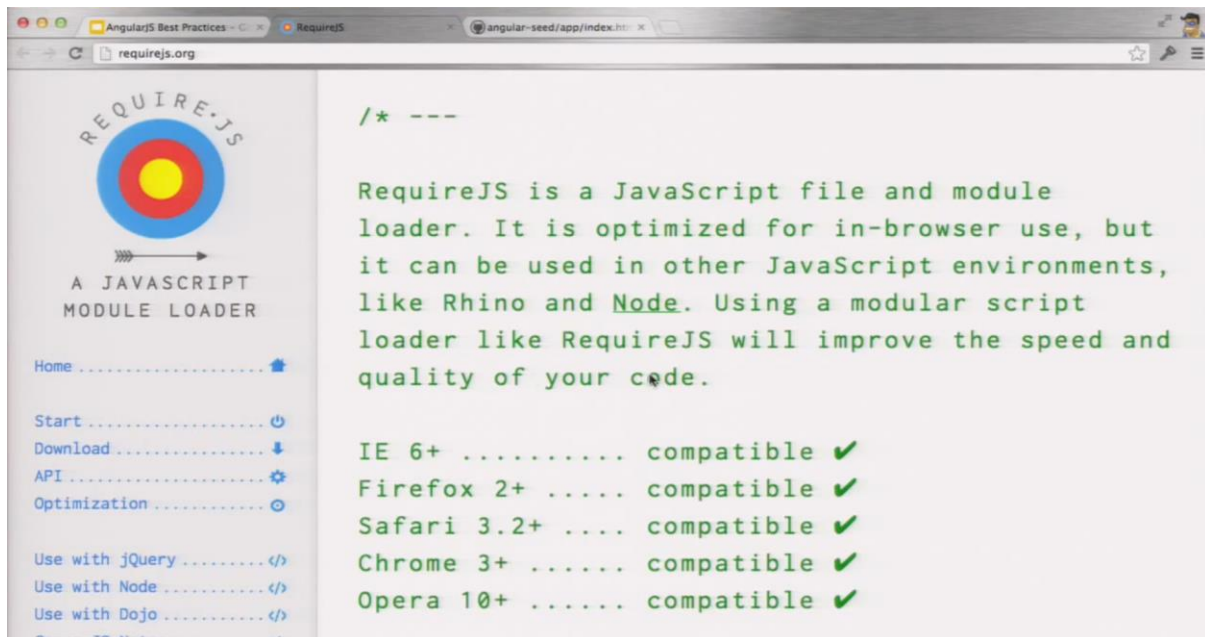
```
1 <!doctype html>
2 <html lang="en" ng-app="myApp">
3 <head>
4   <meta charset="utf-8">
5   <title>My AngularJS App</title>
6   <link rel="stylesheet" href="css/app.css"/>
7 </head>
8 <body>
9   <ul class="menu">
10    <li><a href="#/view1">view1</a></li>
11    <li><a href="#/view2">view2</a></li>
12  </ul>
13
14  <div ng-view></div>
15
16  <div>Angular seed app: v<span app-version></span></div>
17
18  <!-- In production use:
19  <script src="//ajax.googleapis.com/ajax/libs/angularjs/1.0.2/angular.min.js"></script>
20  -->
21  <script src="lib/angular/angular.js"></script>
22  <script src="js/app.js"></script>
23  <script src="js/services.js"></script>
24  <script src="js/controllers.js"></script>
25  <script src="js/filters.js"></script>
```

... this is called synchronous loading.

The way you bootstrap an application is something called **ng-app**, and this basically tells the Angular framework that once it discovers **ng-app** it can go ahead and start executing the application.

It's called synchronous for a reason because the browser blocks.

So sometimes it might be better to load these things asynchronously. For that purpose we have something like **requireJS**



07:48

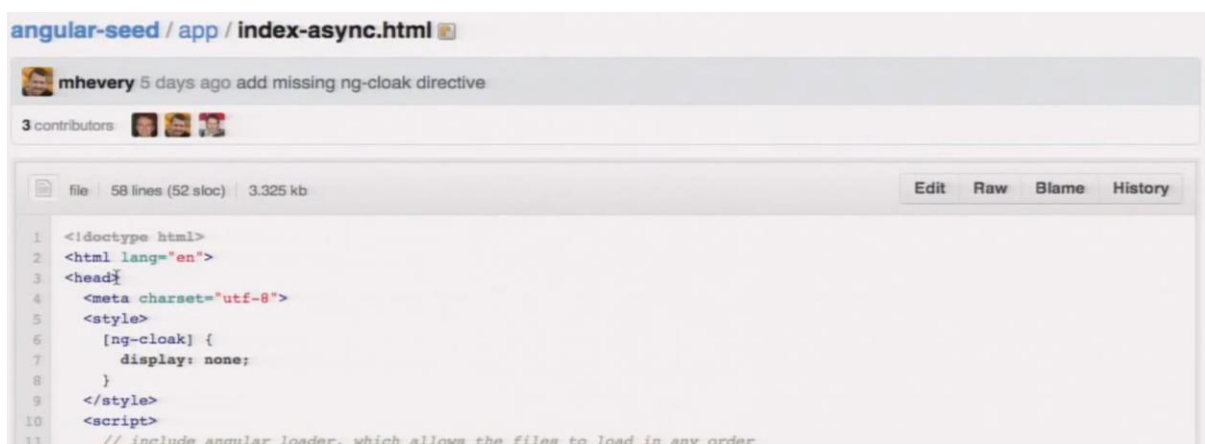
It's a popular way of loading things asynchronously. There's a lot of confusion people have between the **requireJS** module system and also the Angular module system. The only thing I have to say is that even though the names are very similar, it's actually a different concept.

Angular's modules do look at the configuration of the injector and the configuration of the injector defines how things are built at runtime and the building of things is something that happens throughout the lifetime of the application, whereas requireJS deals with how `<script>` tags are loaded into the browser. That only happens at the beginning and it happens only once.

So it's actually different goals that the two actually seek and they are actually very complementary.

But there's an important difference which is that if you have `async` in the `index.html` the biggest difference is that you can no longer have **ng-app**.

If we scroll to the top notice there's no longer **ng-app** present inside of your HTML.



The reason for that is the way that Angular bootstraps itself is it waits for the DOM “content ready”. In other words the browser says “OK. I’m finished loading your HTML file”, and then it looks for an ng-app directive to see if there is anything that needs to be bootstrapped.

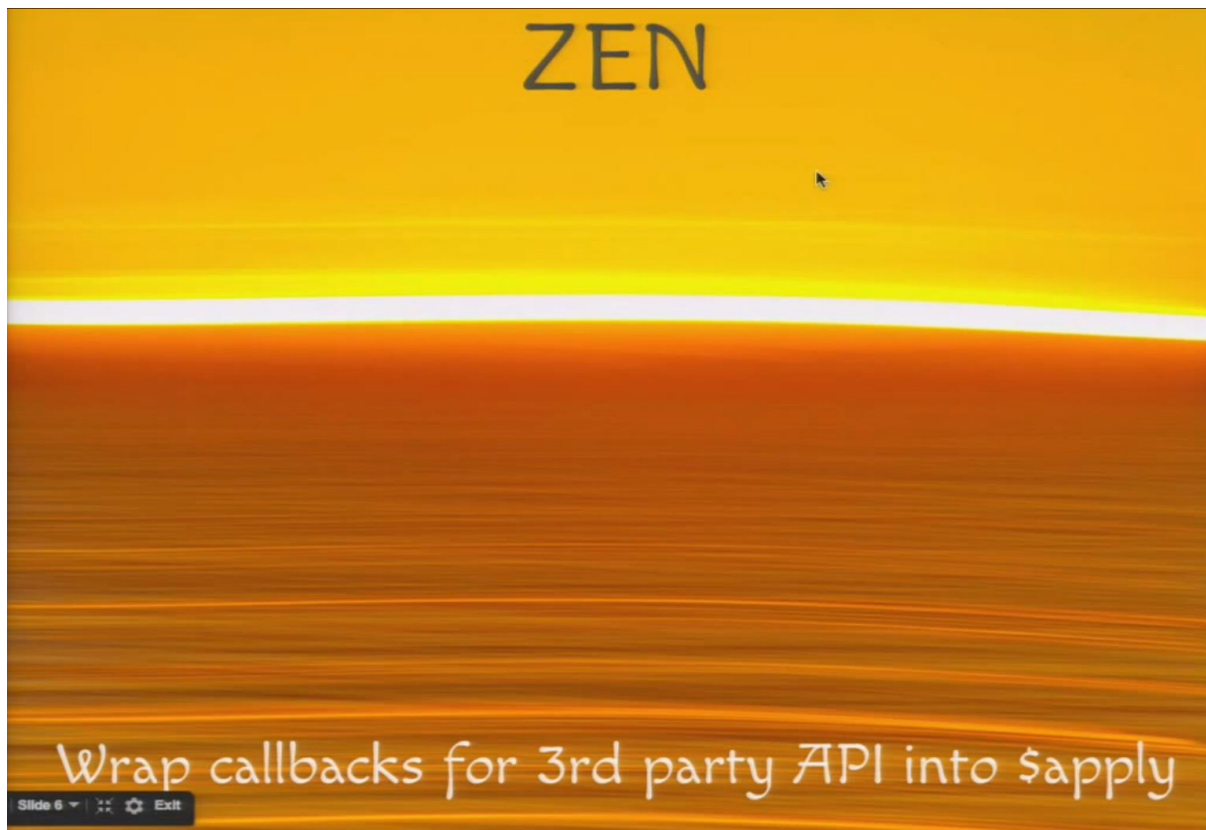
If you load it asynchronously, then the asynchronous can load the Angular after the “content ready” or, more importantly, it can load the Angular library before it loaded the application codebase so Angular might start up before the application’s been loaded and so there is no good way for Angular to know when it’s ready to actually run stuff.

For that purpose if you load it manually using requireJS or any other script loader you have to basically provide a manual bootstrap, which is essentially equivalent to saying “**ng-app = myApp**”.

```
30 // load all of the dependencies asynchronously.
31 $script([
32   'lib/angular/angular.js',
33   'js/app.js',
34   'js/services.js',
35   'js/controllers.js',
36   'js/filters.js',
37   'js/directives.js'
38 ], function() {
39   // when all is done, execute bootstrap angular application
40   angular.bootstrap(document, ['myApp']);
41 });
42 </script>
43 <title>My AngularJS App</title>
```

This thing says “Pretend that there’s an **ng-app** at the root of the document and the application to load is my application. Does that make sense?

Zen Moment 2: Wrap Callbacks for 3rd Party API into \$apply



10:00

All of the times you work with 3rd party applications you do things that are asynchronous. For example, you talk to a back-end server to get data and then when you update the model you're going to be surprised to discover that your UI doesn't update.

That's because you have changed the model outside the lifetime of Angular. Angular doesn't know that the model has changed and so it needs to be notified of it.

A good way to think about it is that just like a browser flip-flops between the render state and the JavaScript execution state you can subdivide the JavaScript execution state where it does regular JavaScript and then Angular JavaScript.

The way you enter the Angular world is through the **\$scope.\$apply** method which basically says "Go perform any operation to change the model that you might have to do and then when you're done performing the operation perform its normal lifecycle so that we can re-render and update the UI.

So the \$apply method is kind of your way out of most third party integrations.

Flash of Unstyled Content

Flash of Unstyled Content

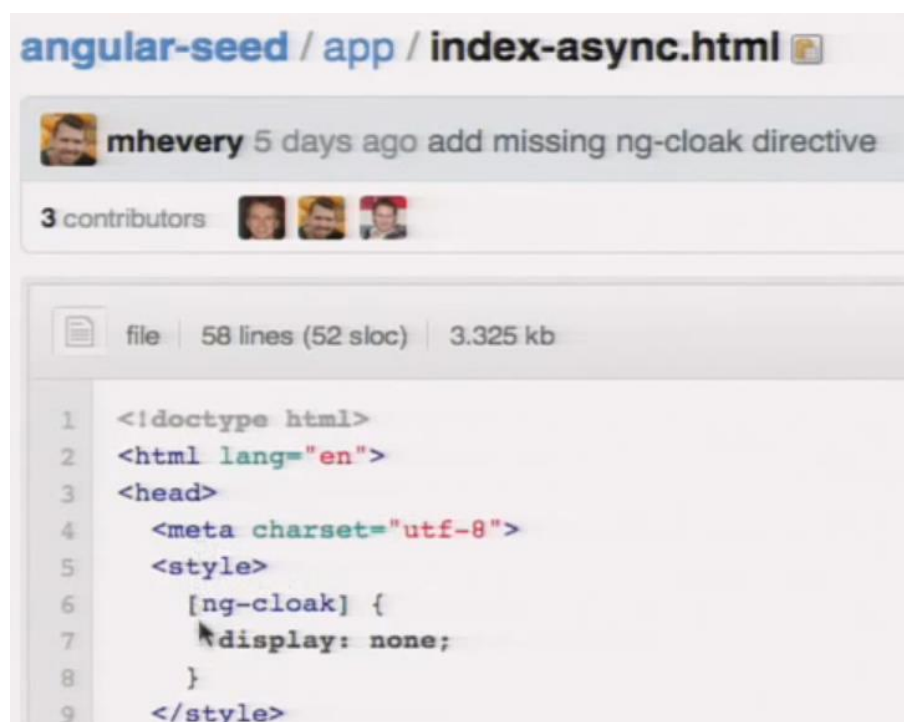
- Issue: User will see `{{}}` on `index.html`
- Solution:
 - hide it with [ng-cloak directive](#)
 - use [ng-bind](#) instead of `{{interpolation}}`

11:13

When you're loading your first page `index.html` you can have a flash of unstyled content that is the double brackets that you placed in your page – they can show up because the browser renders the `index.html` before Angular has a chance to look for them and then replace them with proper behaviour.

Obviously that's an undesirable behaviour so there's a couple of ways you can counteract it.

One is something called **ng-cloak**. This is by far the simplest.



What you're basically doing is putting a style that says "ng-cloak attribute – display: none", and then you place an ng-cloak attribute on the body:

```
<body ng-cloak>
  <ul class="menu">
    <li><a href="#/view1">view1</a></li>
    <li><a href="#/view2">view2</a></li>
  </ul>

  <div ng-view></div>

  <div>Angular seed app: v<span app-version></span></div>

</body>
```

What that basically does is tell the browser "Please do not render the body".

Angular has a directive called **ng-cloak** and what it does is basically removes itself from there. So the browser loads index.html, there is unstyled content in there but because of the style in **ng-cloak** the browser doesn't render anything. Once Angular boots you see the application without any kind of flash.

This is good. It kind of fixes the problem of the unstyled content but the drawback of this is that there's a delay before you actually render anything. You now have all the benefits of putting the `<script>` tags at the bottom of asynchronous loading are negated by not actually rendering anything to the user.

So a slightly better approach is to do what the built-in AngularJS has done.

Here's our example again, where I was showing you the number of things that are built with Angular. As you can see if I hit refresh it renders an unknown thing and then it puts the actual number.



The way that actually works is if we go to "View Page Source" we can see the string that says "Neat things" and we have literally placed the "?" inline in there.

```
<p class="bwa-neat">
  <span class="badge badge-important bwa-count" ng-bind="projects.length ||
  '?'></span> Neat things built with
  
</p>
```

Instead of using double-curly “{{{}}” we’re using the attribute **ng-bind**.

Obviously attributes are not rendered by the browser and so this allows us to ... this is essentially the equivalent of putting `projects.length` - putting this expression into `{{}}` inside of here. But the advantage here is that we don’t flash unstyled content.

So what we’re recommending is that for `index.html` you use **ng-bind** instead of `{{{}}`.

Do you have to use it in your whole application? No! Just in the `index.html` because once you get into individual views or directives or anything that you dynamically load into the browser that then gets fed to the compiler before it gets rendered so there’s no issue any more. This is purely just for the first page load.

Since the `index.html` is typically just to chrome the application they tend not to have a lot of `{{{}}` or bindings placed in there so it should be pretty straightforward to do.

Minification and Compilation

Minification and Compilation

- Do you need it?
 - Angular apps are smaller already
- Minification is an issue because...
 - Angular views use reflection to access model data.
 - basic minification only (no property rename)
 - Dependency injection uses reflection to assemble the application
 - Use proper [annotation](#).
- Never compile angular.min.js (it's already compiled)
 - We are working on better [jscompiler](#) support

14:30

The next thing you probably want to do is you want to minify and do compilation.

“Do you need to do minification?” is the first question.

Our experience is about 80% of what your application typically does if you’re building something using traditional methods like jQuery – about 80% of what you write is actually just DOM manipulation.

By using something like Angular you already are ahead on the size of your application because we can remove that 80% case of DOM manipulation.

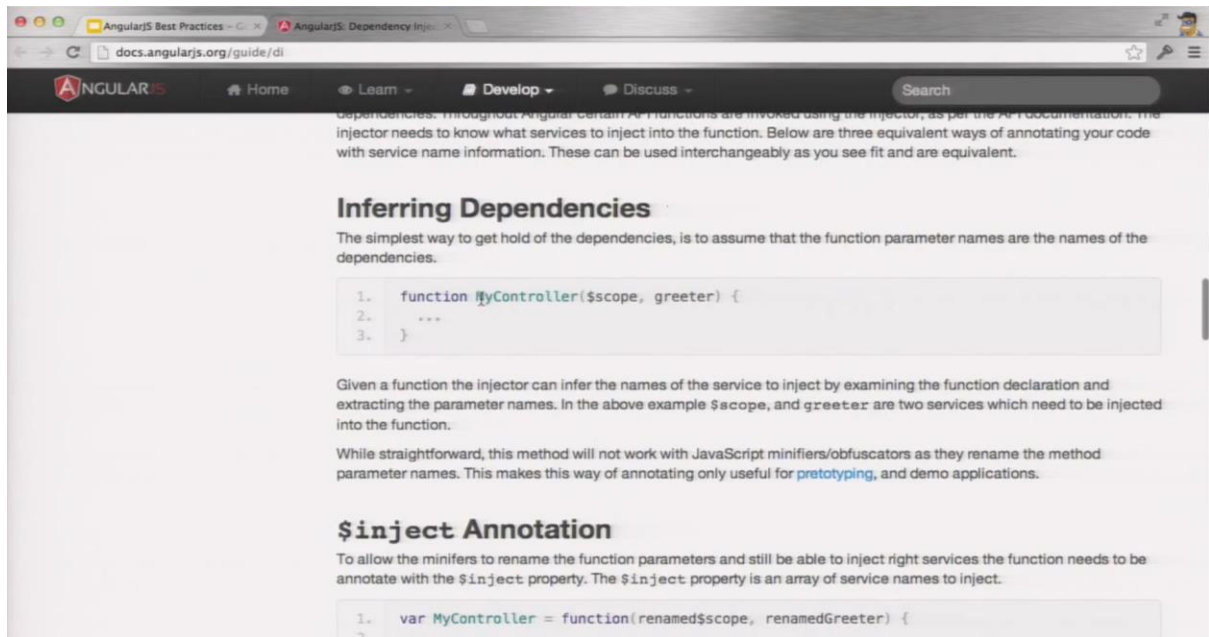
So that’s already a first benefit. I’m not necessarily saying that the benefits of minification of your code becomes less needed. But still there are reasons to minify your code. If you do minify your code you have to know a couple of things about Angular.

First of all, the views use reflection in order to access the model. So if you put `{{}}` curly and inside `{{}}` you put an expression like “username” then it needs to have a properly called username on an object somewhere inside of the model. If a minifier executes - the way it minifies is it renames long names to short names - it will most likely modify username for something shorter like “a” it will no longer work because the reflection no longer works.

For this reason when you use minifiers you have to disable property renaming of the code base. You still can gain a lot of benefit from variable and parameter renaming, also when we go into (indecipherable) and (indecipherable) and all the other stuff that minifiers do.

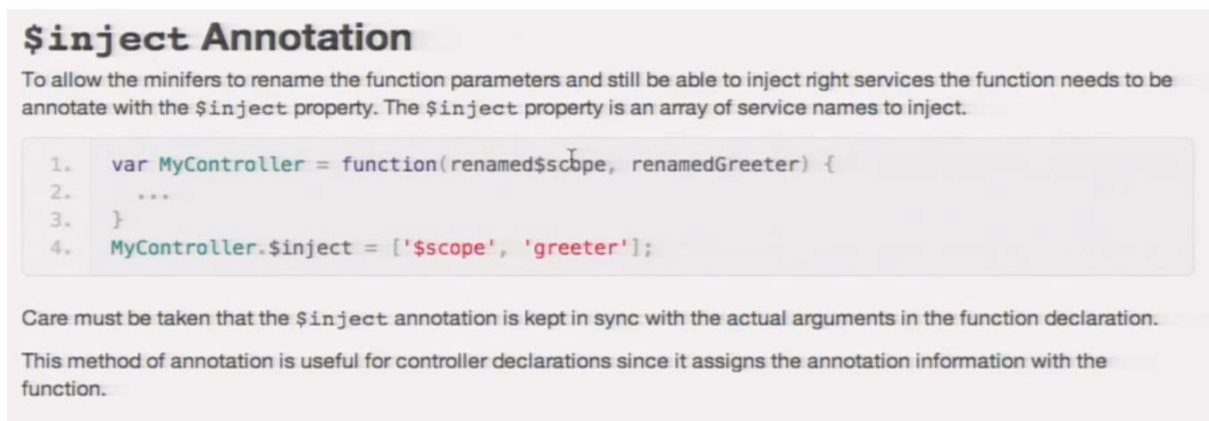
The added benefit of property renaming, depending on the style of the code base you write might or might not be that big, but most of the benefit – the majority of the benefit – is in renaming the names and attributes of the function. That’s called basic optimisation in jscompiler. By turning that

on you can get a lot of the benefits but those parameters are actually used for dependency injection so ... let's have a look...



Suppose you have a MyController. You can say “I need \$scope and a greeter – a service that’s defined somewhere else”.

What minifier essentially does is it renames those two parameters, so when Angular tries to figure out what to inject it will go and read the re-named name and obviously the service is not renamed and it must fail.



To help Angular out you can create a property called **\$inject**. You can place that property on your injectable functions and that basically says “Ignore the normal names and instead use the strings that are in here” and obviously minifier doesn’t change these.

What it means is if you know you’re going to have to do minification at some point in the future you need to place these annotations – these parameter annotations – properly inside your code base so you can take advantage of it.

Typically inside your tests there's no need to place these because we don't need to minify tests. This is just for the production code. But placing this annotation style can actually be cumbersome because let's say you were defining what a greeter is and you say "It's someModule.factory and it takes some other function called \$window).

```
1. someModule.factory('greeter', function($window) {
2.     ...;
3. });
```

Clearly the **\$window** is going to get renamed to something else like the case here using minification:

```
1. var greeterFactory = function(renamed$window) {
2.     ...;
3. };
4.
5. greeterFactory.$inject = ['$window'];
6.
7. someModule.factory('greeter', greeterFactory);
```

Then the issue will become that you can't register the function inline like you've seen it [first code example]. Instead you have to assign it to a local variable [shown above] so that you can assign a property.\$inject on it and then take that variable in here [as the parameter].

So you're getting a bit of an explosion of code and we don't like that in Angular so we actually have an inline style which is declared here...

```
1. someModule.factory('greeter', ['$window', function(renamed$window) {
2.     ...;
3. }]);
```

.. where you wrap the function essentially in an array and you put the parameter names in here.

With the inline style you basically get the same advantages as the **\$inject** property.

These are the kind of things we have to keep in mind when we are doing minification.

18:40

The other thing is we've already compiled **Angular.min.js** and twiddled all the right settings and all the optimisations that we could tweak into it. So don't compile it again because the chances are you're going to not hit all the right settings and you're going to break the thing. Instead just include it as a dependency and if you insist on having a single JavaScript file you can actually concatenate the

angular.min.js before or after your code base as part of your compilation process where things get minified.

Zen Moment 3: Don't Fight the HTML, Extend It



19:20

The big thing about Angular is we embrace HTML, JavaScript, CSS and all the technologies that are out there.

But most of those technologies are for static documents and so what we want to do is extend the vocabulary of those technologies for dynamic documents.

Don't pretend that HTML's not there and abstract it away into layers of some other stuff. Instead embrace it. What we do with Angular is give you the ability to augment the vocabulary of HTML through the concept of **directives**.

If you really miss your `<blink>` tag and you want it back Angular can give it back to you!

You just create a directive saying "If you come across a `<blink>` tag then do all this stuff to implement it".

In other words think of HTML as a bit of a DSL – a domain specific language. If you really wish you had a `tab` or a `pane` or a `zippy` or some special `hover` directive: a `hover` behaviour attribute so the browser would behave this particular way you can handle HTML. This takes a fundamental shift in how you build applications.

Templates

- Extend your HTML and turn it into DSL
 - `<my-component>`
 - `<div my-component>`
 - `<div class="my-component">`
 - `<!-- directive:my-component -->`
- Use a prefix
 - `<my-component>`
 - `<my:component>`
 - broken on [IE](#) so use: `<div my-component>`
- Optionally make it valid
 - `<div x-my-component>`
 - `<div data-my-component>`

20:36

There are limitations to what you can do with declaring these components.

First of all, one way to declare a component is as a component element name, which is what is shown right here [first bulleted item].

In other words, you can make the new thing like a tab or a pane or a zippy. We recommend you place a prefix in front of it – we’ll talk about that in a second or a bit later.

There are other ways of declaring it as well. So you can use it as an attribute. It doesn’t have to be an element name.

This gives you choices. We’re not saying one is better than the other. We’re just saying there are different ways of doing it.

Finally you can use it as part of a class. This can be a little bit surprising that all-of-a-sudden by adding a class you can actually trigger behaviour, but Angular allows you to do this as well.

Finally there are certain things that are not allowed in HTML like putting custom tags inside of table body’s `<tr>`. So for those purposes we have a comment because comments can be placed almost anywhere and the comment can be used to instantiate your components or directives.

You have these choices and it’s up to you to decide which of these choices are your preferred way of expressing yourself.

But – and there’s always a but! – there’s IE!

IE has a few quirks that you need to be aware of. Before we get to IE I want to point out that we strongly recommend putting a prefix in front of it, like clearly not “my” or “ng” as well – that’s our prefix! The prefix can be either as a dash or a colon. So if you wanted to use something like an XML

namespace – if you like that – you can do it as a colon. But if you say “forget the namespace” just use it as a dash as well.

In IE this always works, putting it as an attribute:

```
<div my-component>
```

But if you want to use it as an element there are a couple of surprises in IE.

Suppose you have an <html> tag which says <mytag>some text:

```
1. <html>
2.   <body>
3.     <mytag>some text</mytag>
4.   </body>
5. </html>
```

What you would expect – and this is IE8 specifically and below – when rendering you would expect to have an html document for the html and you have a body which is the child of html and you have a mytag which is the child of body, and mytag has a text element which has “some text”:

```
1. #document
2.   +- HTML
3.     +- BODY
4.       +- mytag
5.         +- #text: some text
```

This is the expected DOM tree.

What you actually get out of IE is you get html, body, mytag but notice that the text isn’t a child of mytag. Instead it’s a sibling!:

```
1.  #document
2.    +- HTML
3.      +- BODY
4.        +- mytag
5.          +- #text: some text
6.            +- /mytag
```

Basically IE treats mytag as a self-closing element, just like a `
`. It doesn't have children, or like `<hr>` there's no children allowed for it and there's no need to even put the trailing `"/"` in `
` because there's no other way of declaring it.

This is how IE treats unknown tags by default.

Notice it also treats the trailing or ending tag as a self-closing tag and inserting it with the invalid character `"/"` in there.

But there is a way round this. It turns out that if you just say **document.createElement** and specify a custom name then IE says "OK. Now it's a real thing. I'll behave the way I was supposed to."


```

1. <html xmlns:ng="http://angularjs.org">
2.   <head>
3.     <!--[if lte IE 8]>
4.       <script>
5.         document.createElement('ng-include');
6.         document.createElement('ng-pluralize');
7.         document.createElement('ng-view');
8.
9.         // Optionally these for CSS
10.        document.createElement('ng:include');
11.        document.createElement('ng:pluralize');
12.        document.createElement('ng:view');
13.      </script>
14.    <![endif]-->
15.  </head>
16.  <body>
17.    ...
18.  </body>
19. </html>

```

So you have two choices: you can either declare a namespace and then use the namespace and that fixes it, or you can basically do what we're doing here which is saying "If you are IE less than or equal to version 8 then...."

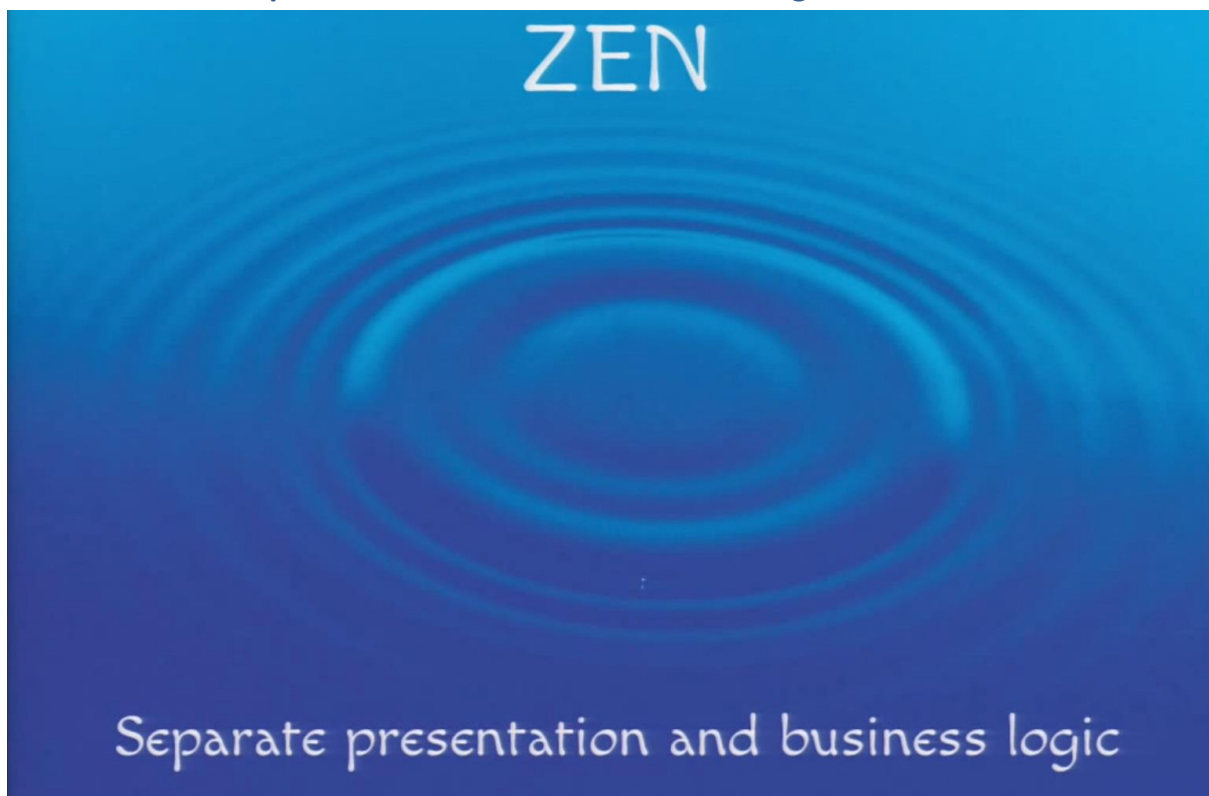
It means that you cannot place these things inside the bottom of your page because IE needs to execute the <script> tags BEFORE it sees the custom elements.

It also means you cannot do it asynchronously. This HAS to be at the top.

Also IE does not have certain things like JSON out of the box so we expect that for IE you get a polyfill to bring it up to modern standards. So you get a polyfill for JSON and also a polyfill for some APIs like string, toString etc.

Finally, some people feel very strongly that if I add a new element like myComponent or a new attribute that doesn't run through the HTML validators ... For those of you that feel strongly about that we give you the option of prefixing everything with either **x-** or **data-**. That should make the validators happy so the x- and data- prefixes are essentially ignored and stripped away and then you have a choice of using a dash or colon for your namespaces. That hopefully should give you plenty of freedom to decide which way you want to set up your templates.

Zen Moment 4: Separate Presentation and Business Logic



25:40

In Angular a lot of design was driven by separating the presentation and business logic.

We wanted to make sure that by separating the two pieces out you get a lot of benefits.

One is testing because if you have them mixed together it's very difficult to write a test that automatically requires a special DOM to be present.

The other thing is that by separating out presentation and business logic when you look at a controller you understand the behaviour of the application because all this stuff about "How do I render?" is subtracted away from it. It's just purely what the application does.

That's a great benefit that Angular brings to the table and a lot of design decisions, namely how we structure the scope and the view, and dependency injection were specifically done so that you can have this clean separation.

So please don't undo our work by putting DOM back into the controllers.

Structuring Business Logic

- Controllers
 - should not reference DOM
 - should have view behavior
 - What should happen if user does X
 - Where do I get X from?
- Services
 - should not reference DOM (mostly)
 - are singletons
 - have logic independent of view
 - Do X operation

PS: Do put DOM manipulation in Directives

26:40

Controllers and Services: When should you use which?

First of all, as I said earlier, controllers should not refer to DOM. If you really need to do DOM manipulation that's what directives are for. We'll talk about them separately.

Controllers should really have the behaviour for the view. In other words, "what should happen if I click X?" or "Where do I fix the data so the view can be rendered?"

Those are the kind of questions the controller is interested in.

Services, on the other hand, are interested with "How do I do X?"

Let me give you an example. Let's say you have an email application and you want it to mark a particular message as "read". Presumably the marking of the message "read" can be done in a lot of different ways in your application, and from a lot of different locations. So the behaviour of "How do I mark a message 'read'?" should be placed inside of a service.

The glue logic which glues the view to a particular service and then the logic that also fetches the right mail messages to be rendered from the view – that should be inside the controller.

There are other differences between services and controllers, namely that with controllers you get a new instance per view so when you navigate to the view you have a new instance whereas services are singletons so they are for the duration of the application lifetime which means that things that need to be cached and not destroyed between view navigations are the perfect place for services.

The other thing you want to keep in mind is that oftentimes people on the mailing list ask "I have one controller and I need to call a method in another controller. How do I get a reference to another controller?"

Well that's a dead giveaway that your logic should be inside of a service not inside of a controller because a controller's really meant to just hook things up to the view. If you actually have something that needs to be done from two different locations then service is your friend.

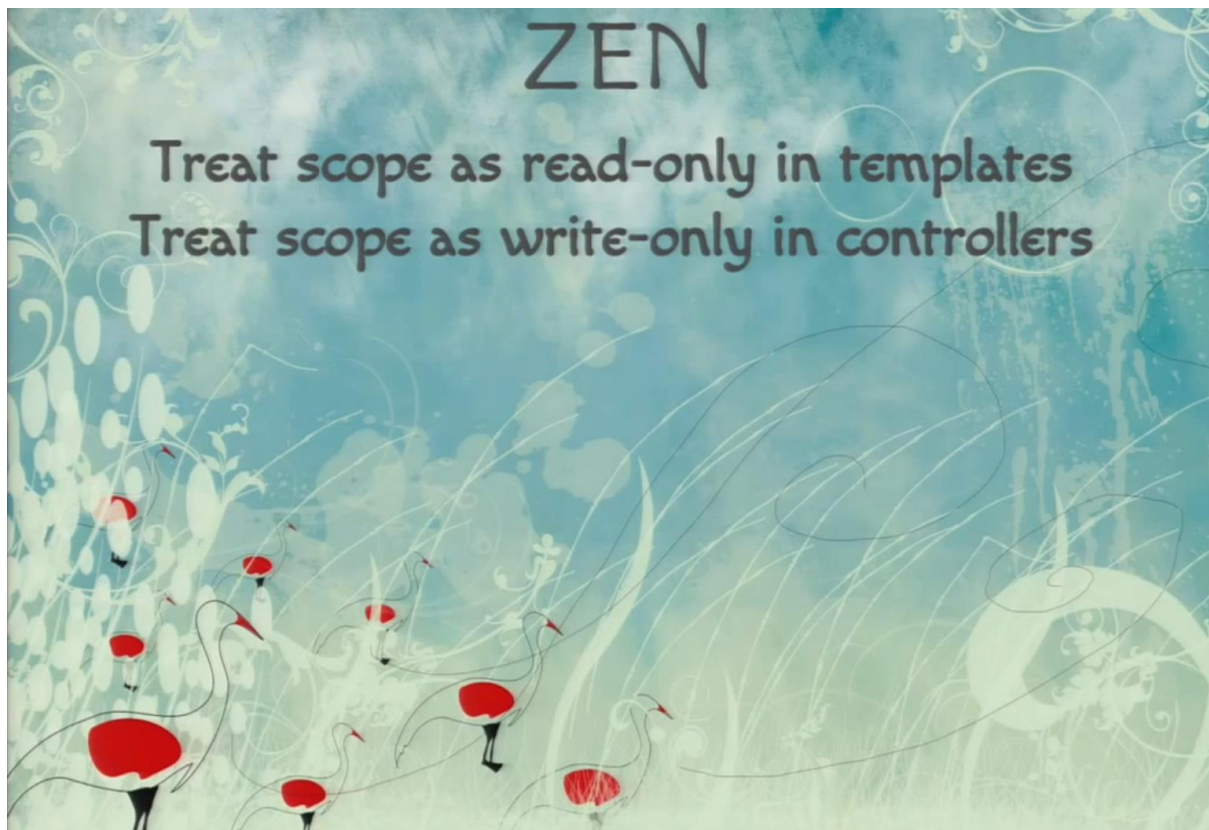
Now as I said, services should not refer to the DOM either, with a little caveat.

The caveat is that sometimes for example you need a service like a dialog box that is responsible for pulling a dialog template in and rendering it. A dialog box really needs to place a `<div>` inside the body of the application. It's not like a directive that you have to place somewhere. It's just something that needs to happen to get the job done.

In that case a very limited number of DOM references might be acceptable inside your service.

Keep in mind that every DOM reference you create could be a potential source of headache from a testing point of view.

Zen Moment 5: Treat scope as read-only in templates, write-only in controllers



29:20

The other thing to keep in mind when dealing with scopes is that `$scope` is the glue between the controller and the view. The view is what the template gets turned into when the template gets executed. A controller is what is responsible for collecting all the data.

From the point of view of a controller `$scope` should be write only.

In other words, the controller's job is to collect the data that needs to be rendered, place it in the scope and be done.

From the point of view of the view the scope is read-only. That is I'm only supposed to be reading properties out, not writing properties back into the scope. We'll talk more about the engine model and how that's important.

People oftentimes think that the scope is the model, and that's not the case. Scope has references to the model. So you create your own model object, and then you put a reference from the model object to the scope. In the view you say `model.<whatever property in the model you want to access>`.

Similarly if you have a form you should have a form model and then the scope basically says `model.propertyX` and then the view is updating the property on the model, not a property on the scope.

Scope

- Treat scope as read-only in templates & write-only in controllers
 - The purpose of the scope is to refer to model not to be the model.
 - The model is your javascript objects
- When doing bidirectional binding ([ng-model](#)) make sure you don't bind directly to the scope properties.
 - unexpected behavior in child scopes

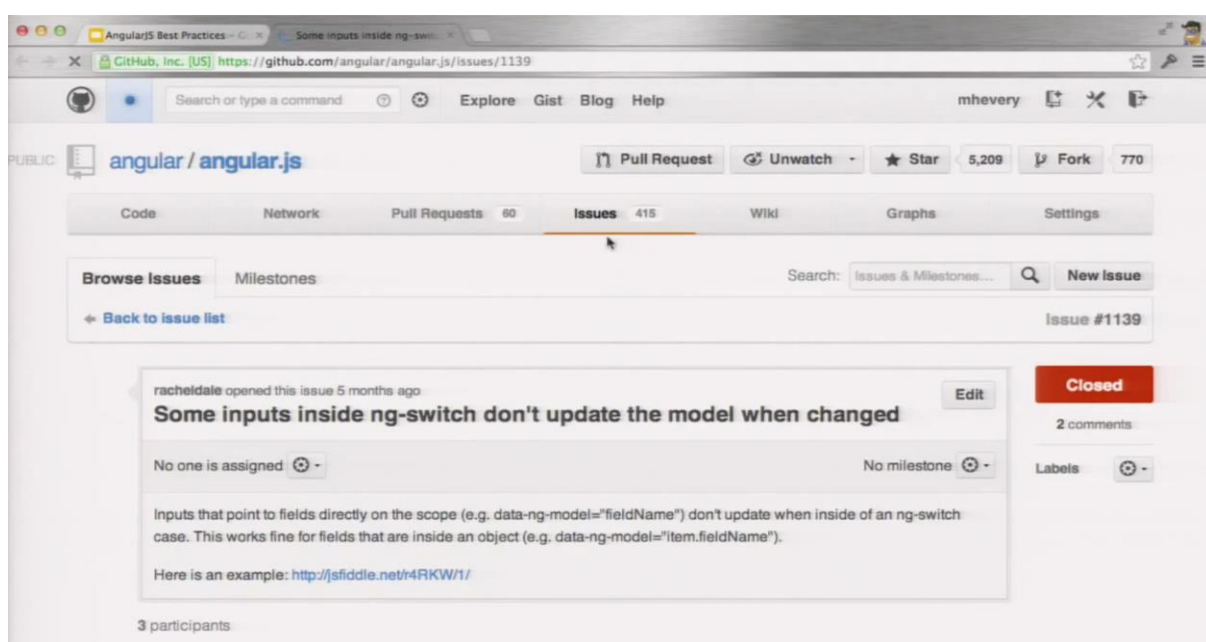
30:36

As I said earlier, the purpose of the scope is to refer to the model, not to be a model.

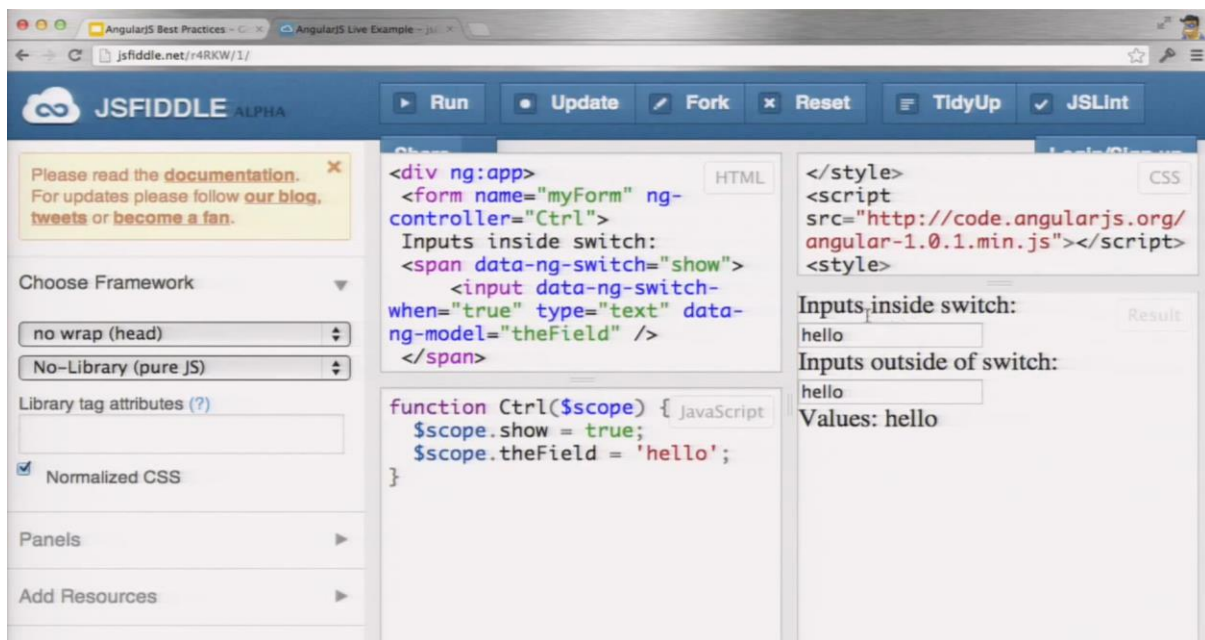
This is important because your model should be your own set of classes and objects that you create. Unlike most other frameworks we don't force you to inherit from any particular object or any particular class or any particular structures or special getter or setter methods.

You really should be free to declare any, or re-use any existing object, as your model for those purposes.

There's this tricky part about bi-directional data binding. I say tricky, because whenever you have **ng-model** sooner or later somebody files a bug like this with us.

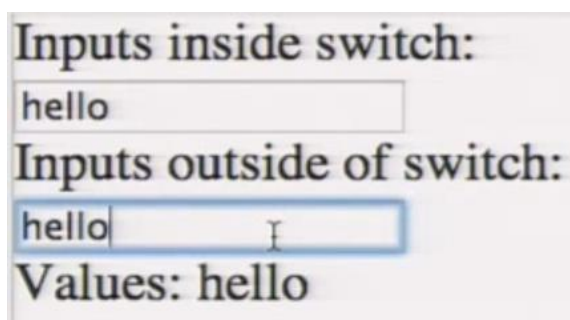


We get plenty of this. Let me demonstrate what this bug does.

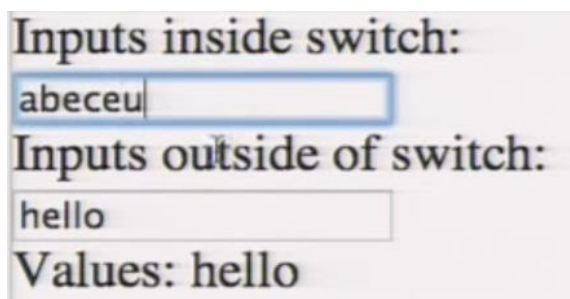


31:35

So here are two inputs. One is inside of a switch and one is outside of a switch. The fact that it's inside a switch is irrelevant. What matters is that the switch statement creates a new scope.



So here's an interesting behaviour. When I start typing here [in the lower box] the other form changes, but the moment I start typing here [the top input field] they become disconnected.



Inputs inside switch:
abeceu

Inputs outside of switch:
ueouoeuou

Values: ueouoeuou

Typing in the second one then no longer changes the other one.

There's got to be a bug filed every month about this type of thing. I know it's unexpected and it's strange behaviour, but if you think about it, it actually makes sense.

What's happening is that you're saying that you're binding to a property called "theField" and this isn't binding directly to the model. This is where we were saying you should be using \$scope purely as a read not as a write.

```
<div ng:app>
  <form name="myForm" ng-
controller="Ctrl">
    Inputs inside switch:
    <span data-ng-switch="show">
      <input data-ng-switch-
when="true" type="text" data-
ng-model="theField" />
    </span>
```

The strange behaviour happens because the parent \$scope has the field, and the child \$scope gets the property inherited. This is how particular inheritance works in JavaScript. Once you write to your child scope you're over-riding the parent. The parent no longer becomes visible and you essentially become disconnected from it.

The good rule-of-thumb is whenever you have **ng-model** there's got to be a "." in there somewhere. If you don't have a dot you're going to get it wrong.

Zen Moment 6: The \$watch getter function must always be fast and idempotent



33:10

The way Angular does watching a model is through basically a very fancy way of doing “isDirty” checking.

In order to do “isDirty” checking there is this function called **\$watch**.

\$watch has two functions that it takes. It takes what we call a **getter function**, and then it takes a **reaction function**.

A getter function is something we call all the time to figure out if a particular property has changed. That means, because we call it all the time, it has to have a couple of properties

- first of all it has got to be very fast so don’t do anything crazy inside of the **\$watch** function because it’s a sure way to make your application slow.
- the other thing is it should have no side effects. There shouldn’t be a counter or anything going on because we’re not guaranteeing how many times we call that function.

Finally it needs to be idempotent, which means every time I call it I should return the same value. The number of times I call this function should not determine in any way the value we get out of it.

This is the kind of thing to keep in mind when writing a **\$watch** function.

Structuring Modules

Structuring modules

- Why multiple modules?
- Usually one module for application
- One module per third-party reusable library
- Possibly for testing
 - create test modules which override services (mocks ngMock)
 - test portion of the app (*)
- Possibly for incremental code loading
- If you have multiple modules per app
 - Group by functionality / feature not by type
 - You should group by view since views will be lazy loaded in near future

34:20

Let's start with modules.

As I said, modules are a way of configuring the injector. We allow you to have multiple modules and then have modules that can be dependent on each other.

The primary reason for having multiple modules is to have third party libraries to provide something and then your application can depend on the third party library through the module system.

So **Angular-UI** by Beam will provide a module that you simply include in your application and then you have a dependency on it. People just take the module idea and then they run with it and go crazy and they create a module in their application and they chop it up into different modules. They have a module for all the controllers and then a module for all the directives and a module for all the services.

While that may look like a good idea it's actually self-defeating because the purpose of a module is to be able to instantiate a portion of the application. If you chop it up by type it's very unlikely that you can instantiate just the controllers without any kind of services or any kind of directives.

Usually things come in clusters, right? So a particular controller needs a particular set of services to get its job done and for rendering purposes you also need a particular set of directives.

A better way to structure the application if you wanted to take advantage of this is by feature or by things that go together rather than by type or the kind of things that you have.

Today there is very little benefit to actually splitting up your application into these chunks because all the module contains is configuration information. While the benefit of splitting up the application is minimal there is a benefit to having different modules for your tests.

If you have a scenario for your tests where you want to replace the back-end with a fake back-end then you should have a module for that which resets all the components in a particular way.

If you have another kind of test which tries to do it at a much higher level where the services are mocked up rather than the actual communication with the server then maybe have another module for testing purposes only that does this.

So you could use `$module` as a way of grouping related configurations for your tests together.

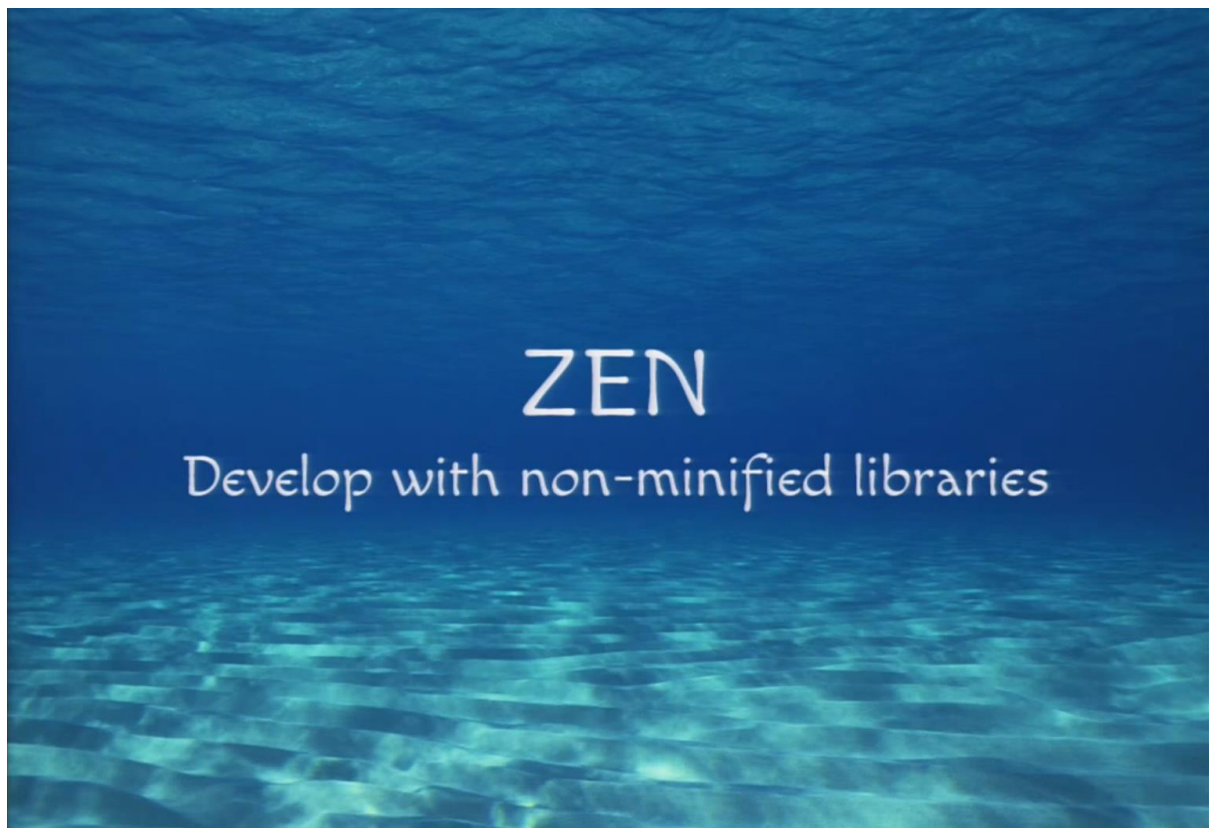
There's another reason to break up modules. While today it's very difficult to do lazy loading of your application – in other words lazy loading a portion of your application and the views – hopefully in the near future we are working on enabling so that you can lazy load the JavaScript code along with everything else.

What that's going to mean is your application should be broken up by modules by view.

Each view should have a separate module. The reason it's by view is because those are all the things that are related to each other and those are all the things that we'll need bringing together. The view is what we'll then be able to partially lazy load in the future.

So if you really insist on chopping things up then by view is a good way to go rather than by type.

Zen Moment 7: Develop with non-minified libraries



37:42

This should go without saying but it's surprising how many people send us bug reports with stack traces and it's minified stack frames on there.

The trouble with minified is it's heh! Fine for computers, but not so good for humans, right?

The line numbers are off, the function names are mangled. There's nothing to be got from a stack trace like that. So spend the time to set up your environment so that you can run in production with a minified version and in the developer world with a non-minified version, and it should be easy to switch back and forth to enable them.

The benefits of minification we've already mentioned. It's a smaller code base to be sent across. Inside of development you really want to see a full stack trace to see what's going on there: what line numbers and everything else.

Deployment Techniques

Deployment Techniques

- minify and concatenate your JS
- gzip enable your server
- index.html (non-cachable)
- cache by version:
 - views
 - code
 - images
 - css

38:38

Finally let's talk a little bit about deployment techniques.

As we said you want to minify and concatenate your JavaScript code although the concatenation... It used to be a clear yes, include the concatenation in order to have a performance impact.

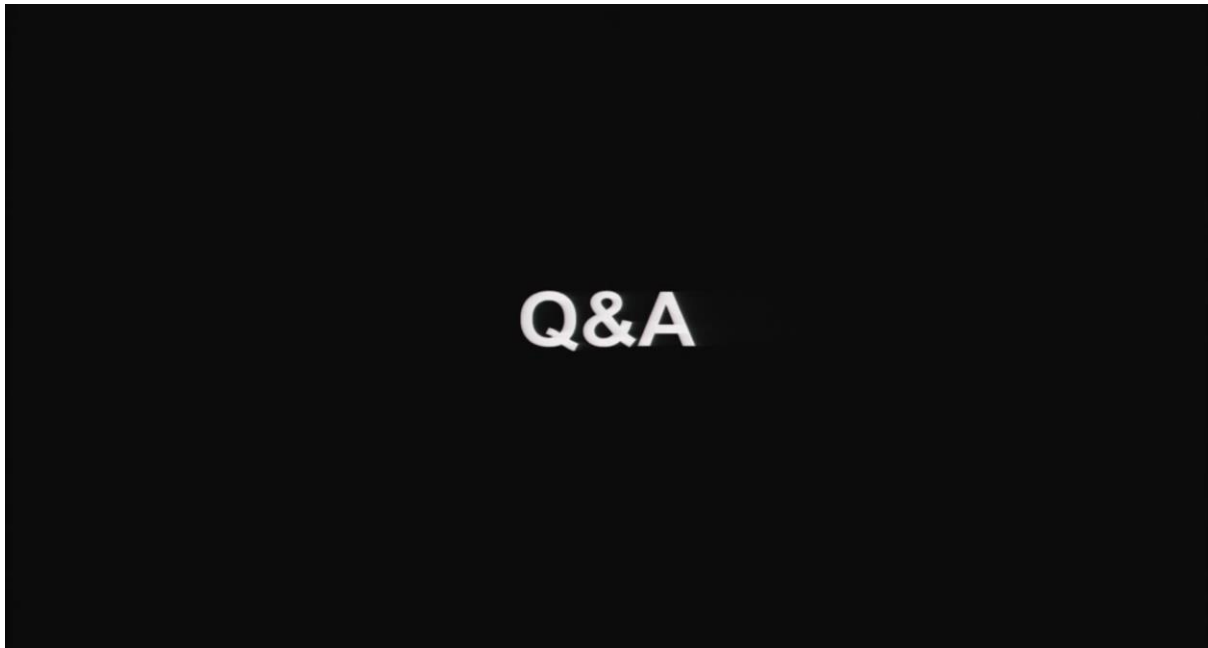
It turns out with modern browsers you should read some of the latest findings and blog posts people have and the experiments they've done. It turns out that most modern browsers are pretty good about loading things in parallel and executing them so it's not exactly clear that benefits of concatenating everything into one large `<script>` tag will gain you the benefits. But do your own tests, measure your own way. Don't take my word for it. Every app is different.

Secondly, enable gzip. It's a very simple thing to do and all of a sudden your download is almost twice as fast. Again a lot of server people will get to do that.

We think that HTML should be non-cachable in the browser so that every time you go to a particular application you get a brand new copy of index.html. The reason for that is you want to be able to easily change the version number of the libraries that are defined inside the index.html.

But the libraries themselves should have a version number encoded in the URL and those should be strongly cached. So you basically tell the browser go all out on these particular items because if you want to upgrade a particular version then you can just change the version in index.html and everything else is going to automatically load the proper versions for it.

Q&A



40:13

I think I've finished 5 minutes early so I would love to have some questions. I think there's good t-shirts to be gotten for good questions.

Q: When we deploy to production is it possible to de-compile the HTML because when we do ng-include it's going to be a string and is it possible to somehow do a compile on the back-end so when we get to the browser it's already ...

So your question is "Can you pre-compile the templates on the server?" [affirmation from audience member]

Not as of yet. That's a very complicated thing that you're asking and we're thinking about this problem but we don't think it's ever going to be a seamless solution. If we do something in the future it's not going to be a trivial setup because of the nature of this particular problem.

But it turns out that the compilation does not take that much time so you gain a lot more benefits by concatenating things together. So for example in the case of an Angular application if you have lots of small views and lots of small directives that have their own files you can benefit greatly by having the browser make a single request rather than lots of small ones. The way you can do that is Angular has a `$templateCache` service. You can auto-generate a module where you read the HTML file into a string and then you just put `$templateCache` and the URL and the content and then your application still looks like it's fetching from the server but actually everything's pre-cached in the browser in a single shot.

There's a bit of work you have to do on your side to generate this module but once you do you get great benefits of performance.

<indecipherable question expansion>

I'm saying take all the stuff that's related to a particular view, put it in a single module but the things that are HTML can be pre-cached by executing a piece of JavaScript that basically says "\$templateCache the URL where the HTML would have been and the content of the HTML".

43:00

Q: I have a question about lazy loading. You were talking about the HTML and the JavaScript. So we're using ng-include. We have two different approaches and I want to see what's the best practice. One of the ways we can do it is we can completely pre-load... let's say I'm abstracting the view like <div> and ng-controller and I put all the logic for the controller and the HTML object in this template on the server. Then I make ng-include based on demand and I'm loading HTML and loading controller and ng-controller together with it. It actually seems to be working and everything's good. There is no binding delays or anything. Another strategy that we have in the team and we're actually deviating a bit is What if we have a <div> with ng-controller and inside of this we're going to load the JavaScript for our controller as a part of our bundle. Inside of this div we're going to have ng-controller, and then controller on demand is going to decide when we're going to load a particular HTML for this. Is there any better strategy for this? Any best practices for doing this because in one way we're kind of unloading a bit of our JavaScript. So our initial bundle is getting smaller. In the other way everything's getting bootstrapped right away and then those types of things like shared data or like event handling is already there and taken care of because controller was there like from the beginning through the whole page cycle. We kind of have issues both ways. One is kind of lighter but it's more complex. Another way is less complex but heavier.

The only thing I can tell you is you've got to measure these things. It's hard to have an overall statement of "This is faster vs that one". It's something that you have to have tests for and figure out "OK. In these browsers these seems to perform better, or in those browsers ..." Just get some numbers to it. It's hard for me to say "This is better than the other way".

Are there any hoops we might encounter in one browser rather than another?

The only thing Angular provides, as I said earlier, is this \$templateCache, which allows you to pre-cache URLs with HTML responses for them so that you never have to change the way that you write your application but the browser will not incur an HTTP GET for that particular

46:40

Q: On one of your slides – it might even be relevant to go back to it - the \$switch with the input you said if you read from an input in a child scope like inside a \$switch if you mess with the input you'll break the relationship with the parent. You never explained how to fix it. One of the things I do is I do \$parent.<modelName> and I also do \$route but....

You could refer to the parent through \$parent. It's not recommended like we don't think that's a good practice. Instead the answer is actually I mentioned it in the slide: your engine model should have a dot in it. If it has a dot that means the property is not on the scope. Instead scope simply has a reference to the model and the property is on the model. This is a pretty cool way where the whole thing gets fixed.

So whilst \$parent is a way round this particular issue it's fragile because if somebody introduces another scope above you then you have to do \$parent \$parent whereas by having the model separate it always works no matter how many intermediate scopes there are inbetween.

47:07

Q: My name is Gene. I'm a software engineer working at ancestry.com. We manage this bunch of services that make AJAX calls. I'm curious about what the best way to get the data back from that service which is not synchronous. Right now we are broadcasting the results back to the controller and that controller implements a listener. Some of us think that's the best way but I've seen couple of different parameters such as returning a promise directly. Or you maintain reference to a \$scope variable so that Angular refreshes the data reference for you. Which way would you recommend?

I think promises are the preferred way in Angular and this is the way we're going to be moving in Angular. The reason we like promises is because it can synchronously return to you an object that you can keep reference to instead of you passing in like the object where eventually something's going to be placed on it, or putting everything in a callback function. The rest of the Angular system already knows about promises. When promises get resolved it automatically calls \$apply for you. They are getting resolved at the right point in the lifecycle of the application.

So all of these things are pushing you in the direction of promises. I think the promise is the proper way of dealing with this.

48:50

Q: I have a question regarding performance. One of the things I noticed is we're running into some performance issues with Angular just if you add lots of rows because it's update lots of models, run lots of filters and so on. But my question is slightly different. I want to know what happens if you have lots of different views like tabs in your app and your app is very deep and people move around it for a long time. A lot of stuff gets instantiated and then you update something on your \$rootScope because it gets used in lots of different places – it's the single source of truth that you want to be in one place. Are we going to see performance issues from that or how do I solve that basically?

Performance in Angular is directly dictated by two factors. One is how many bindings you have on a page. The second one is how expensive the \$getter functions are. As I said, the \$getter functions need to be really, really fast. Be careful if you are polling \$watch not to invoke a slow function in there because it can very quickly multiply.

As far as number of bindings on a page – it doesn't matter how big your application actually is, it only matters what's currently rendered. So things like ng-include or ng-view – they break that portion of the application away and it's no longer part of the redraw cycle. But things like ng-show and ng-hide do not break, so don't think that by hiding something you're actually losing those particular things.

The other thing is if you have something like a repeater you shouldn't have more than ... I don't know... something like a hundred rows because that's just a limitation of what a good UI is and I

don't want to see a thousand rows presented to me because as a human being I can't parse this particular thing. It's really calling for pagination or some sort of search etc etc.

When you're using filters sometimes filters can be expensive, especially when running them over and over again. The solution to that is actually to have an intermediary model. Rather than running the filter inline like `ng-repeat` item in items filter whatever, have a `$watch` on the filter property and then whenever the property changes re-filter and copy the values into the secondary model which you then use for `ng-repeat`.

Those are different strategies for dealing with it.

51:25

Q: We're actually implementing Angular on the front-end in a pre-existing Rails app and one of the things we've come across is that sometimes when we return ERB partials with ng- attributes spliced into them the ng- attributes come back dead. They just don't do anything. So the question's fairly simple. Can you maybe enlighten me as to why.

I'm guessing that the way you're putting these things in is you're using `innerHTML`?

<affirmative response>

Angular needs to know when the DOM changes. Specifically Angular wants to be in charge of the DOM changing. So what's happening is Angular thinks it's in charge and then you're going back to it [the DOM] and putting `innerHTML`. You can still do that but you have to notify Angular.

So the way you do that is there's a `$compile` service that you use to compile this particular thing and run it. We've kind of already done all those things for you so I think the answer is that you should be using `ng-include` rather than `innerHTML` for inserting these particular things in because `ng-include` does the proper compilation and does the proper insertion and the whole lifecycle and the whole scope management for you.

Back to the previous question, you were asking about all these things being instantiated. The other things `$scope`'s do is they worry about memory management so you don't have memory leaks. I would say you need to load your partials through `ng-include` rather than `innerHTML`.

Eventually browsers might implement a DOM mutation event. If they do then a simple act of `innerHTML` could fire an event that Angular could eventually listen on and do this but as of today browsers don't do this, those that do those APIs are deprecated but there's some work on new APIs that will enable this but that's not available on browsers today.

53:40

Q: My question's about the events API that's on the scope. I've got a lot of mileage out of using it as a sort of application event bus that allows me to have my components communicate with each other. But I haven't seen a lot of documentation or best practices on how one should go about doing that or even if it's a good idea. An example of the sort of problem I run into is because it's a sort of directional hierarchical bus you have to \$emit up and then re \$broadcast down if you're in a kind of leaf node... if you want siblings to talk to each other you have to go up and then come back

down. So you end up playing this game renaming your events because if you test the same event you throw from the route you just get a kind of...

Well you could just get a reference to the root node and then just fire it from the root if you want to broadcast it to everybody.

I suppose you could. I always feel that root is kind of grabbing global state and that feels a little dangerous to me but...

It's not exactly global state but finish your question.

... so the question is I was wondering if there were some best practices in dealing with the events, or what were the design goals you had in mind with it?

Events. What they do is they provide a communication channel between two parties that don't want to be too close to each other. They don't want to have references to each other. That has advantages and disadvantages.

The advantage is "Heh! I don't have to care whether you exist on the other side – I just fire off these events" which is great but it also makes them a very fragile API because if you change the name or if you fire before somebody registers .. There are all these edge events and this is generally true for all events not just for Angular.

So while certain things are well solved using events eg \$location changes – we can broadcast \$location changes and we can veto location changes and controllers to prevent you from navigating somewhere else, in general I think for apps the events are less useful, especially because Angular has data binding so many things events are used for is data binding and we've already done that for you so you don't have to use it for it.

So really the question around whether you should use events is how tightly coupled do you want your components to be? If you really want them to be at a distance then events might be the answer.

But most times I think injecting services and doing direct communications is probably preferred. It's a more robust way of dealing with these things. It depends on your case.

56:10

Q: This is from Dave, who I believe is the guy who made those beautiful Angular stockings for Christmas. If you guys haven't seen those check out our Google Plus stream. His question is what's on tap for Angular 2.0?

Wow! That's a loaded question. We are planning for where we'd like to take Angular 2.

There's a couple of things I'd like to attack. First of all we want to be able to do lazy loading of JavaScript. That's through modules that we've described. In order to do this we need to have a hierarchical injector like right now the injector is only one – we don't have the concept of hierarchies. There's a lot of different things that come with that and we can get into it later.

The other thing we want to be able to do is server-side pre-rendering. Somebody asked about that earlier. It's a very complicated question but the idea is that if you're somebody like a search engine and you want to see the actual HTML. Also in order to render the application very very fast it would be nice if the server could pre-render the original page and then Angular could somehow be able to attach to rendered state.

It sounds easy but in reality it's quite complicated because things like repeaters could not be present because you're repeating over zero items. Blocks could be hidden, and so on. So it's an interesting problem space.

Also we'd like to fix the directives API to simplify it. There's a lot of people that ask questions about the linking and compile functions so we figured out a way to get rid of the compile function and just have a linking function and actually turn all the directives into just plain old controllers that you're already familiar with. What else am I forgetting?

There's a lot of stuff. We're going to have a blog post soon...

Oh, animation! Animation is the big thing we wanted to be able to do so that if you have a repeater and you want to animate items coming in and out you should be able to just say "Animate text" or "Animate using full animation" and it should be able to just provide this in and out because right now animation is quite complicated.

Right now we act like the friendly giant and say "Farewell to our friends on YouTube. Thank you for joining us".

