

AngularJS in 60 Minutes



by

Dan Wahlin

Transcription and Arrangement

by Ian Smith

AngularJS Fundamentals in 60-ish Minutes

Dan Wahlin

The Wahlin Group
A Division of Wahlin Consulting



Video Length: 01:10:49

So you've heard about **AngularJS**, but you're not exactly sure how to get started with it?

This video's for you.

AngularJS Fundamentals in 60-ish Minutes is going to go through all of the key fundamentals you need to know about the *AngularJS* SPA framework.

Video: <http://www.youtube.com/watch?v=i9MHigUZKEM>

Disclaimer: *The original content is copyright of the original "free to download" video published as indicated by the link 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.*

Contents


AngularJS in 60 Minutes.....	1
Introduction	4
Module 1: Getting Started	15
Single Page Application (SPA)	16
The Challenge With SPAs	17
AngularJS.org	21
Download AngularJS	22
Module 2: Directives, Filters and Data Binding.....	24
What are Directives?.....	25
Using Directives and Data Binding Syntax	26
Data-Binding Example using AngularJS Directives.....	28
Iterating with the ng-repeat Directive	31
ng-Repeat Example	33
The AngularJS API Reference for Directives.....	35
ngRepeat Documentation.....	36
Using Filters.....	37
Using Filters Demo	39
Module 3: Views, Controllers and Scope	44
Creating a View and Controller	47
Module 4: Modules, Routes and Factories	53
Modules are Containers.....	56
Creating a Module.....	58
Creating a Controller in a Module.....	60
The Role of Routes	65
Defining Routes.....	66
Defining Routes Demo	68
Using Factories and Services.....	80
The Role of the Factory	82
Factory Demo.....	84
Wrap-Up Demo: Pulling It All Together	89
Summary	98
Download the Sample Code.....	100
Resources	101

Introduction

Dan Wahlin



Blog
<http://weblogs.asp.net/dwahlin>



Twitter
@DanWahlin

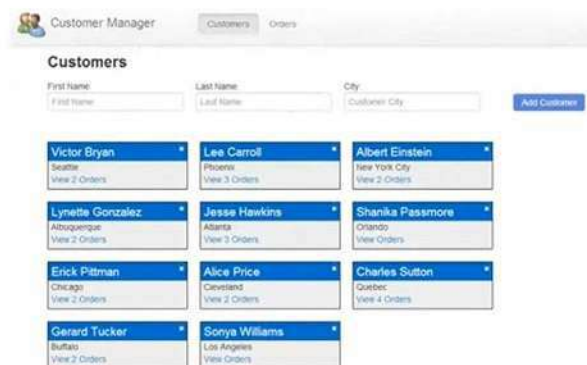
00:00:15

My name's **Dan Wahlin** and I'll be walking you through some of the key fundamentals, some of the things I really like.

My blog can be found here at <http://weblogs.asp.net/dwahlin> and if you're on Twitter feel free to follow me @danWahlin and keep in touch that way.

Download the sample code:

<http://tinyurl.com/AngularJSDemos>



00:00:30

To get started we're going to be focussing 100% on AngularJS.

The demo's I'm going to be showing throughout this are going to be found here

<http://tinyurl.com/AngularJSDemos>

I have some real simple demo's to start things out, and then we'll build up and then by the end we'll kind of put all the main pieces together.

I'm not going to have time in around 60-ish minutes to go through an entire full-scale line-of-business type app but you can absolutely build those types of apps with *AngularJS*.



<http://angularjs.org>

00:01:03

If you're new to Angular and you really haven't read much about it you can go to <http://angularjs.org> to get all the information. This includes documentation, demonstrations, tutorials – all that fun stuff.

The first time I went there I was really excited about it. I heard about it from a friend. I really hadn't been on the SPA (Single Page Application) bandwagon because I just felt it was too much of a mess. There's too many scripts involved and you have all of these different things you need to deal with.

So I was really excited about Angular because it really was, as you'll see, kind of a SPA framework.

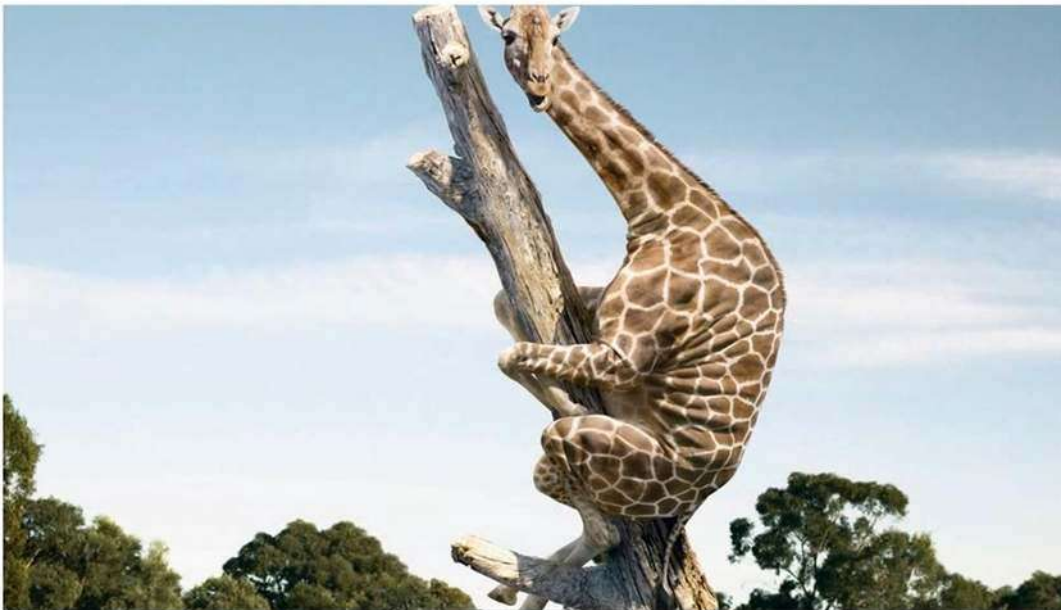
The first time I went and looked at the docs I have to say not a lot of light bulbs went off.



00:01:45

It felt a little bit strange.

The more I got into it, it looked a little bit stranger in parts.



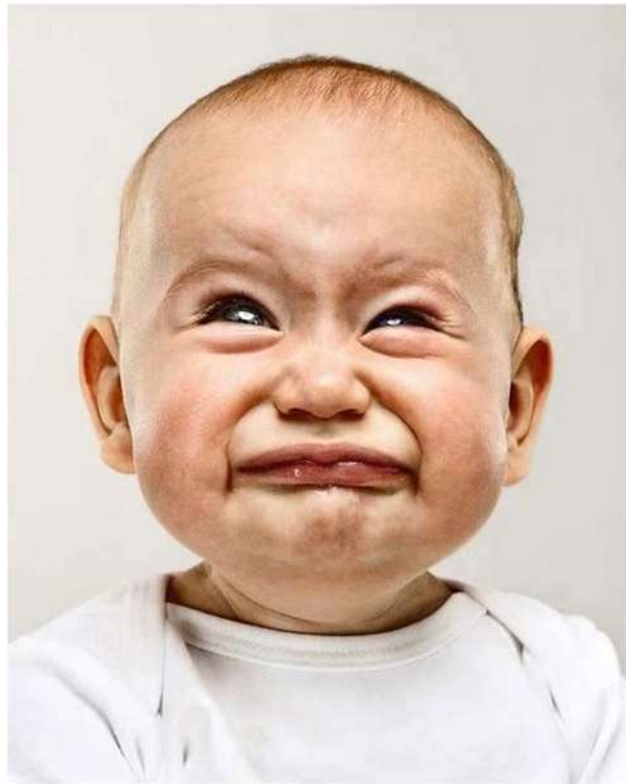


00:01:50

I saw all these different words like “transclusion” and “scope” and “directive” and I said “What the heck is this?”

In fact I wasn’t even sure if transclusion was an actual word, but it turns out it is.

I kind of felt like this guy.



I won't say I shed a tear, but I was a little frustrated because I've been using JavaScript since the 90's and usually I pick up on stuff pretty quickly so I figured "OK. Maybe I'm stupid!"

It turns out I just wasn't thinking about it the right way, and about how to approach it.



00:02:20

Once I took a step back and relaxed a little bit and said “Heh! This can’t be as hard as I’m making it.” I chilled out a little bit...





00:02:31

... and I realised “Wow! This is awesome”.

AngularJS truly in my opinion is an awesome framework and so a lot of light bulbs went off. Once that started happening all the pieces fitted in and made total sense to me.

I think it’s like anything. With a new framework: sometimes you catch it instantly when you learn it, and other times you don’t.

With this it was more a matter of I hadn’t taken the time to be really honest to research the different pieces. I was kind of learning little titbits here and there.

What I’m going to do throughout this video is walk you through all the key things that I wish I would have understood more about upfront and hopefully jump-start your *AngularJS* development process.



00:03:10

Once you get done you're going to have superpowers, just like this kid here.

Maybe not "force" powers, but super-SPA powers – Single Page Application powers

Agenda

- **AngularJS Features**
- **Getting Started**
- **Directives, Filters and Data Binding**
- **Views, Controllers and Scope**
- **Modules, Routes and Factories**



00:03:22

The agenda is...

We're going to start off with some of the key features **AngularJS** offers and I'll kind of introduce the challenge with writing SPAs from scratch.

Anyone who knows me knows I do not recommend writing them from scratch. I just think that in the long term it is, when it comes to maintenance, a nightmare.

There's too many scripts involved and I'm worried about version dependencies and scripts changing and things breaking.

So we're going to talk about that and how Angular addresses it.

Then we're going to get started with some of the framework fundamental features that Angular provides.

Then I'll go into some of those key features that you've really got to start off with, kind of the A-B-Cs of Angular if you will, so Directives, Filters and two-way Data Binding which is just awesome.

I'm a big fan of some of the other scripts out there – like **KnockoutJS** as an example - but you're going to see that Angular is a true framework. It's not just a library that does maybe one or two things: it actually can do a LOT of different things.

Once we get through the Directives, Filters and Data Binding we're going to talk about Views, Controllers and Scope.

And then we'll wrap up with Modules, and we'll talk about how all this other stuff fits into modules, and then we'll get into some SPA concepts like Routes and even Factories for sharing data and using data.

Module 1: Getting Started



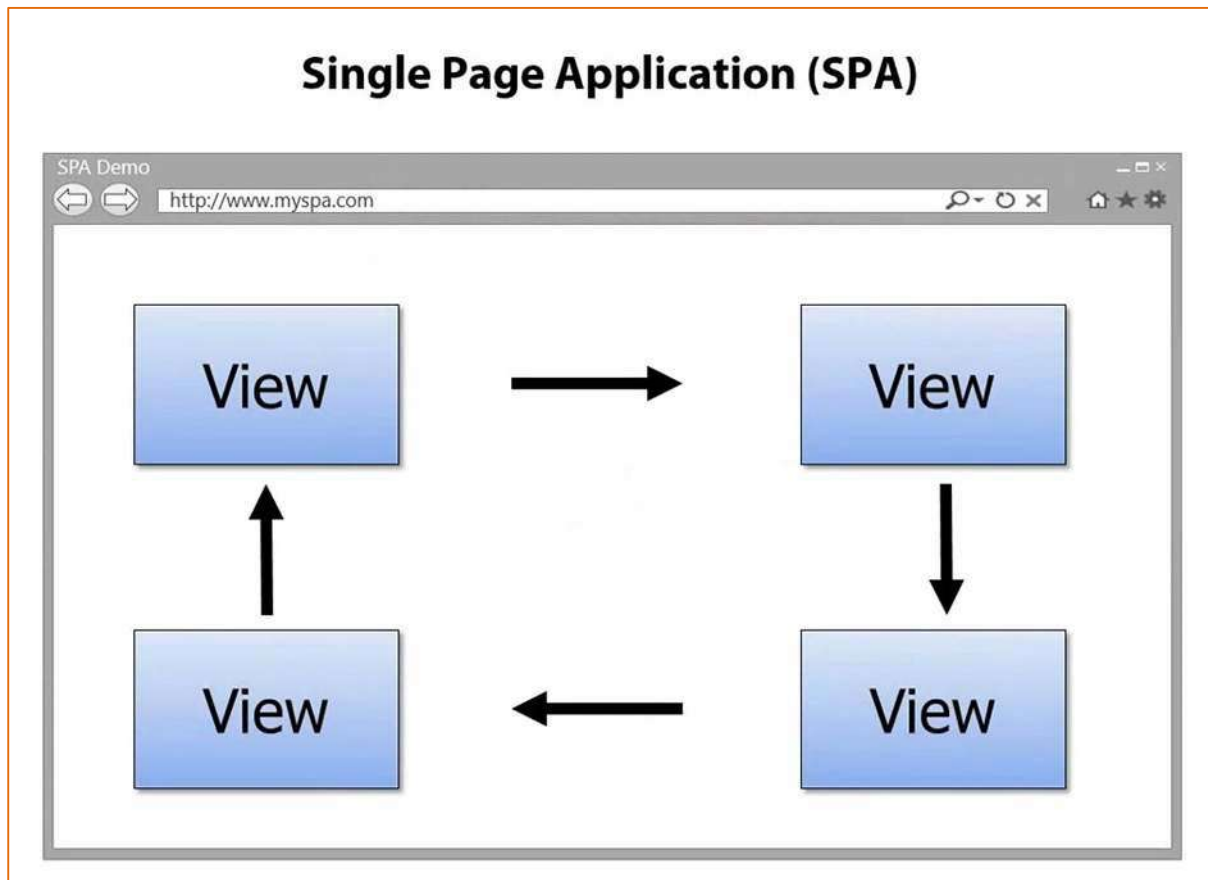
Getting Started

00:04:28

So how do you get started with *AngularJS* and why do you want to get started? I mean what features does it offer that are that compelling?

Well we're going to talk about that in this particular section.

Single Page Application (SPA)



00:04:49

First off, a Single Page Application is one in which we have a shell page and we can load multiple views into that.

So a traditional app, as you know you typically blink and load everything again. It's not very efficient on the bandwidth, especially in the mobile world.

In a SPA we can load the initial content upfront and then the different views or the little kind of mini-web pages can be loaded on the fly and embedded into the shell.

AngularJS, as we're going to see, is a very good SPA framework, but it's not just for that. You don't have to load these dynamic views with it. In fact if you wanted you could just use it for some of the cool separation of code that I'll show you and data binding, but I will focus on SPAs here.

The Challenge With SPAs

The Challenge with SPAs

DOM Manipulation History Module Loading
Routing Caching Object Modeling
Data Binding Ajax/Promises View Loading



00:05:34

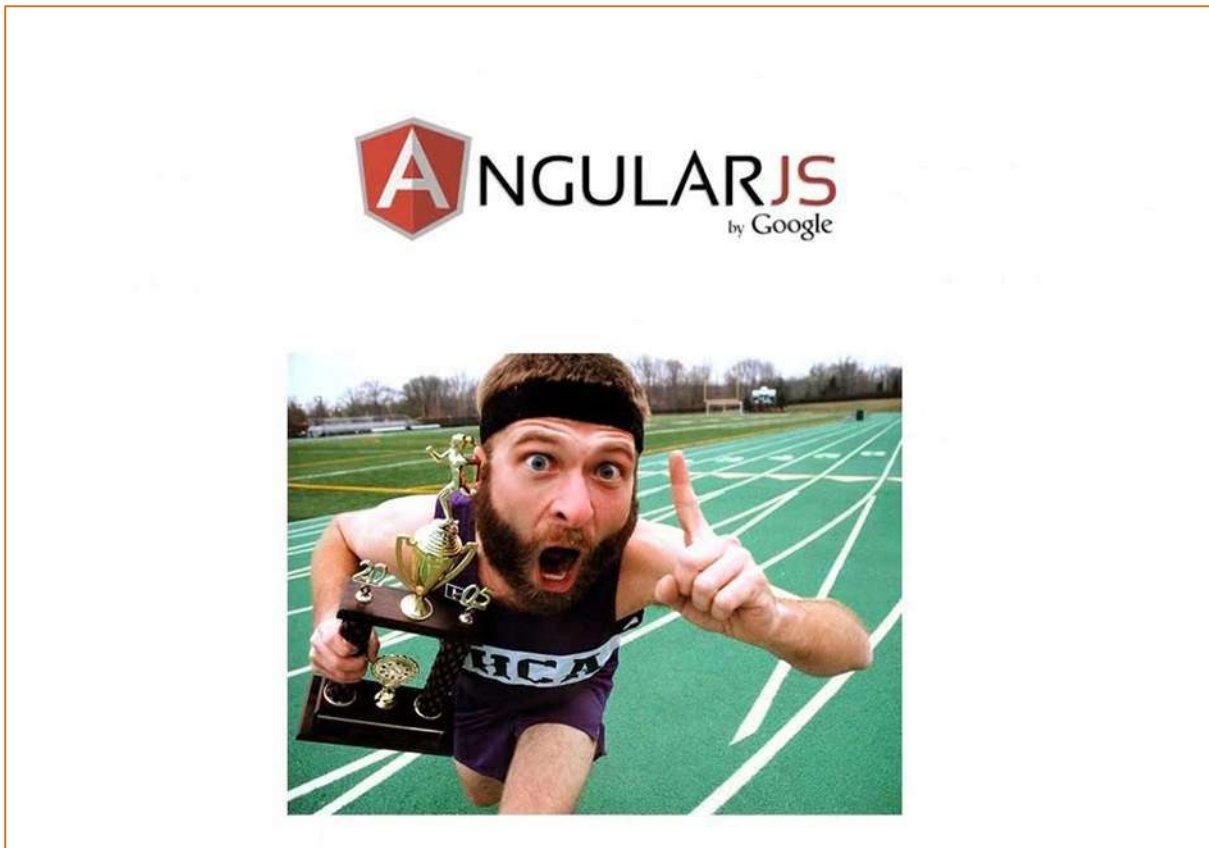
The challenge with building SPAs from scratch is there's a lot of different issues to deal with: DOM manipulation and history and how do you dynamically load modules and how do you deal with promises when you make async calls and things like that.

Routing becomes a huge issue because you have to have some way to track "Where are we? And where are we going?"

All of this type of stuff you're going to see is built into Angular. Now we can certainly do all this with different scripts out there. We could use sammyJS and jQuery and historyJS and requireJS. For AJAX we can use Q and there's a lot of different options.

But Angular, you're going to see, provides a lot of cool features.

A Full-featured SPA Framework



00:06:10

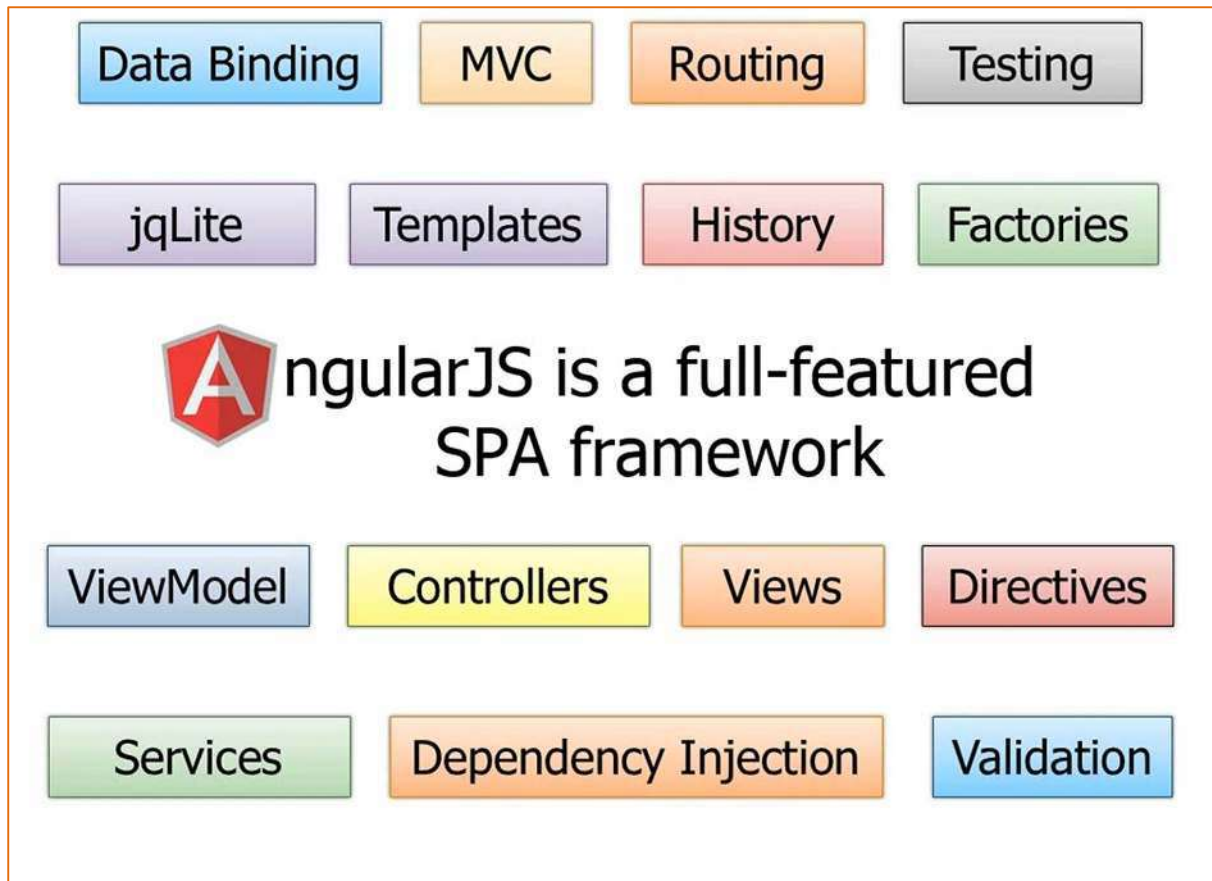
AngularJS is one core library.

I really like that concept because I don't have to rely on a million other scripts and worrying about those different script versions playing nicely into the future.

If you work on a development team then maintenance should be some type of a goal for you, especially if you do the maintenance.

Angular, in my view, gives a nice solid core that you can build on top of.

Now what are some of the features?



00:06:37

As mentioned it [Angular] is really a full-featured SPA framework.

It does all kinds of good stuff.

We have two-way data binding. We have the Model-View-Controller concept. Routing of the Views I mentioned into the shell pages is done through built-in routing support and I'll show how to do that in this video.

Testing was designed right from the beginning so you can build very robust tests if you'd like, which is obviously recommended.

For DOM manipulation jqLite is built-in which is kind of like the Mini-Me of jQuery. If you want to use more advanced stuff you can even use jQuery and they play really nice together: Angular and jQuery.

When it comes to data binding we have full support for templates. History's built in. We can share code through factories and services and other things.

Then there's even more. We have the concept of data-binding with View Models. Directives I'm going to be talking about in the next section, which is a way to teach HTML new tricks. Validation. Dynamically injecting different features at run time through dependency injection and much much more.

These are just some of the core features and these features will satisfy some of the others that I mentioned when it comes to building a SPA.

Now building that SPA from scratch can be a challenge – not so hard here.

AngularJS.org



00:07:55

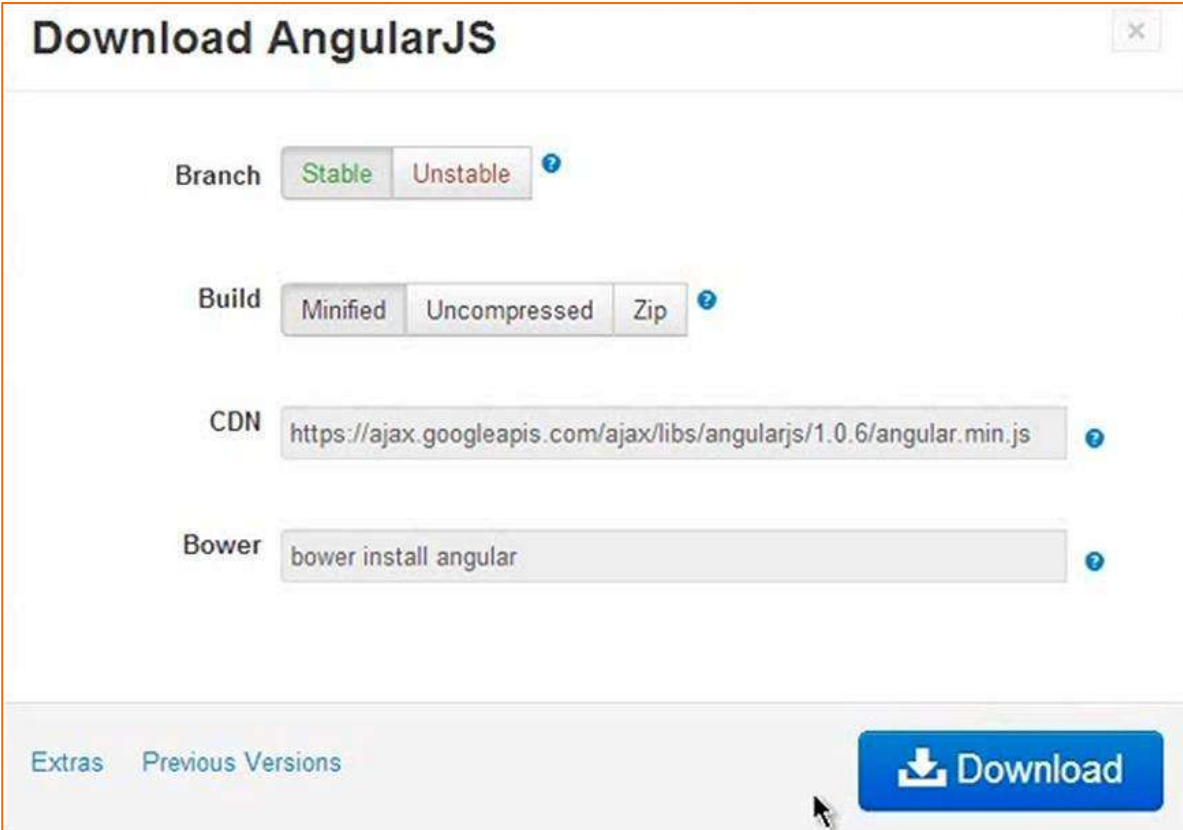
Let's go and run off to the AngularJS page real fast and I'll show you how we can get started with it.

Then in the next section we'll jump into some of the key starting features.

To get started with AngularJS just head over to <http://angularjs.org>

You'll notice here that I can go to GitHub and I can actually get to all of the scripts there, or I can just hit "Download" which is very simple.

Download AngularJS



There are two different options. I can go with the “stable” or the “unstable”.

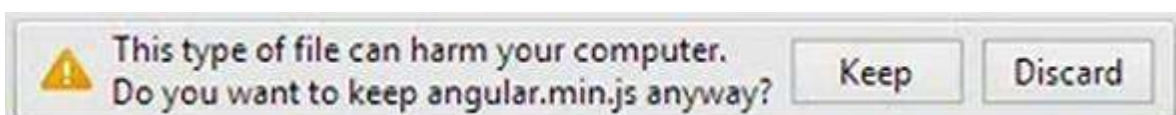
If you want cutting edge go with the “unstable”. I actually use it all the time and I’ve had really good luck with that but Stable would be the official, recommended release if we’re building a production application.

Then we can go with the minified, uncompressed or zipped version.

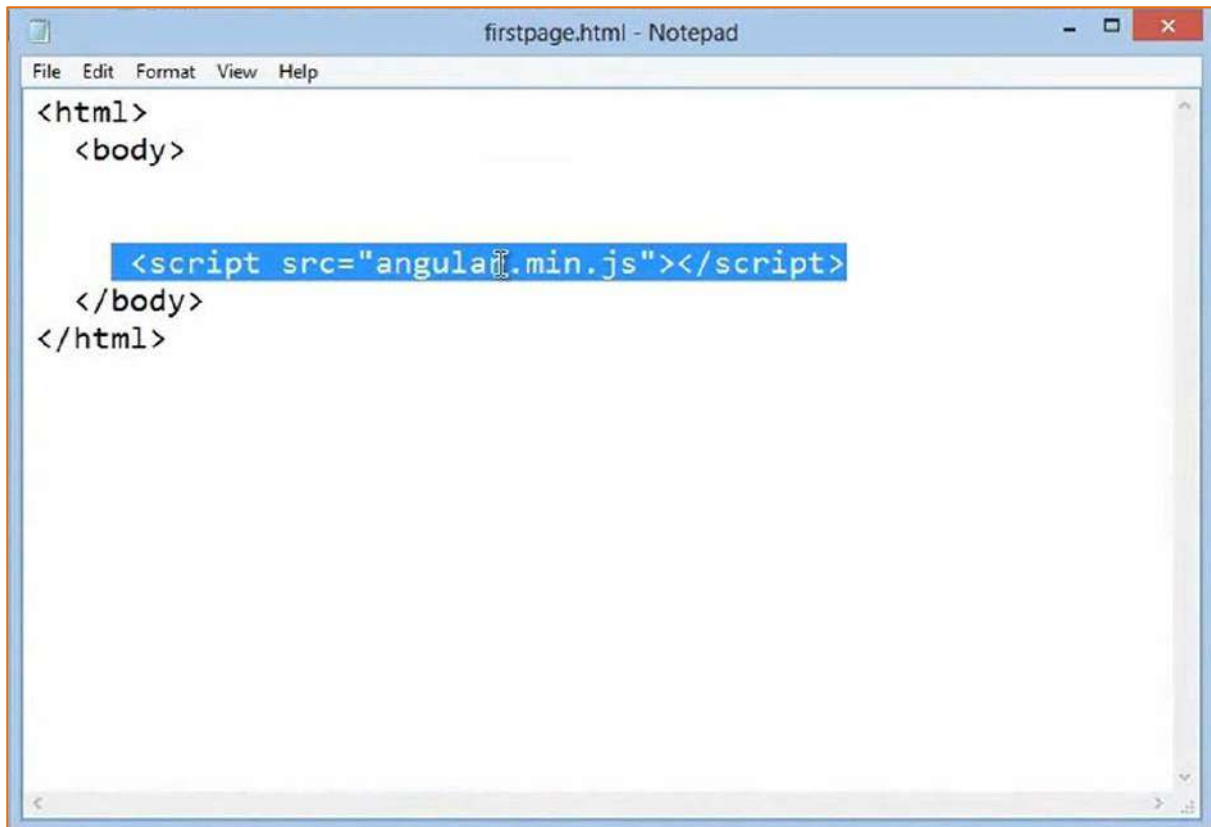
You also have access to a CDN. The CDN will give you access around the world to regional data centres that in this case, Google host.

Then we also potentially get some caching benefits, some parallelism benefits of browsers being able to load different scripts because they’re different domains. There’s actually some good benefits by going with the CDN if you want it.

But for now I’m just going to go for stable and minified and download it. This is just going to give me this **angular.min.js** that you’ll see.



Now what I want to do from here is just plop it into a web page and that’s all I have to do.



```
<html>
  <body>

  <script src="angular.min.js"></script>
</body>
</html>
```

00:09:09

You'll notice I've already added that in [above] and we're ready to go. It's not going to do much right now but that's going to lead us in nicely to the next section which will be **Directives, Data Binding and Filters**.

Module 2: Directives, Filters and Data Binding



Directives, Filters and Data Binding

00:09:26

Once you've added the *AngularJS* script into a page now you're ready to start using it and the first thing we're going to talk about is something called **Directives**.

They're very, very critical and a kind of core concept in the AngularJS framework.

From there we're going to talk about filtering data and we'll talk about data binding, so a lot of cool stuff in this particular section.

What are Directives?

What are Directives?

They teach HTML new tricks!

00:09:48

To start off, what is a directive?

Well I mentioned this earlier. A directive is really a way to teach HTML new tricks.

The web when it first came out was really just designed to display static pages. As we all know it's become very dynamic and we've dealt with that pretty well. **jQuery** came out many years ago and it provided a way to do it. Even before then we could use raw, vanilla JavaScript.

Angular takes it up a whole notch and allows us to extend HTML very easily by simply adding attributes, elements or comments.

Using Directives and Data Binding Syntax

Using Directives and Data Binding Syntax

```
<!DOCTYPE html>
<html ng-app>
<head>
  <title></title>
</head>
<body>
  <div class="container">
    Name: <input type="text" ng-model="name" /> {{ name }}
  </div>

  <script src="Scripts/angular.js"></script>
</body>
</html>
```

The diagram illustrates the code with three callouts:

- A blue callout box labeled "Directive" points to the `ng-app` attribute in the `<html>` tag.
- A blue callout box labeled "Directive" points to the `ng-model="name"` attribute in the `<input>` tag.
- A blue callout box labeled "Data Binding Expression" points to the `{{ name }}` expression in the text.

00:10:20

Here's an example of using a very basic, but important, Angular directive.

Notice at the top we have **ng-app**. Any time you see **ng-** that is an Angular directive. It's a built-on directive. You can also write custom ones. You can get third party ones and things like that.

This particular directive is very important because the script that's now loaded [at the bottom] is going to kick off and this will initialise the Angular app. Right now we don't have any particular module associated or any other code but we can still do stuff just by adding **ng-app**.

So for example, this is an example of another directive called **ng-model**.

What **ng-model** does is behind the scenes it's going to add a property up in the memory called "name" into what's called "the **scope**".

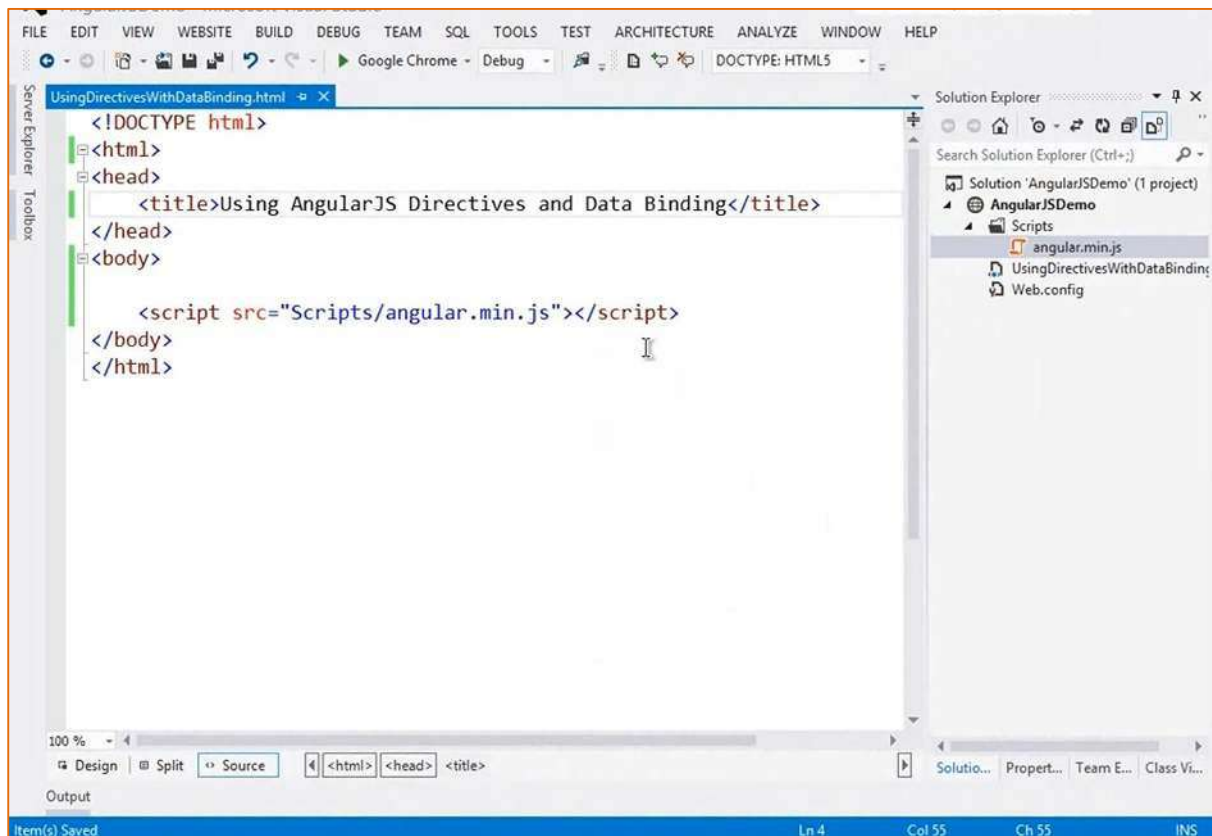
If you've ever dealt with the concept of a View's model called a ViewModel - **Knockout** and some other libraries have this concept – then what this is really doing behind the scenes is making an empty ViewModel but then filling it with a **name** property. Now if I want to write out that value then I can simply come over and add a data binding expression.

Expressions are really cool because if I wanted to put "1 + 1" and try to write out the result I could do that. You can't put conditional logic in here because you shouldn't be putting that type of conditional logic in your views. But out of the box, just by adding the **ng-app** and **ng-model** with a

property as they type into this text box I can actually bind to that value and that provides a very cool little feature.

So let's go ahead and look at a demonstration of that.

Data-Binding Example using AngularJS Directives



00:12:02

So I have a pretty simple web page. You'll see I already have Angular included.

Let's go ahead and start off by saying we'll allow you to type your name. We'll bump this down:

```
<body>
  Name:
  <br />
  <input type="text" />
  <script src="Scripts/angular.min.js"></script>
</body>
```

We're just going to do an `<input type="text" />` and we'll leave it at that right now.

```
Name:
<br />
<input type="text" />
<script src="Scripts/angular.min.js"></script>
```

Obviously if I were to run this we're not going to see much happen. We're going to see a textbox and as I type nothing's going to happen. Let's say that as they type we're going to write out the value live as they type.

The first thing I need to do is come in and add the **ng-app** directive.

```
<!DOCTYPE html>
<html ng-app>
<head>
  <title>Using AngularJS Directives and Data Binding</title>
</head>
```

00:12:39

If you feel more comfortable, and I do to be really honest, adding **data-** on these then you can do it. In fact I could even add **data-ng-app=""** and this would still work.

```
<!DOCTYPE html>
<html data-ng-app="" >
```

Then it will validate against some of the different validators out there. I'll leave that up to you but you don't have to put **data-** if you don't want to.

I'm going to come down a little bit and say **ng-model="name"**.

This is the name of the property. I could have said "foo" or I could have said "fo", "fum" or whatever I wanted but we're going to do "name" here, and again I'm going to add **data-** just because it makes me feel all warm and fuzzy.

```
<body>
  Name:
  <br />
  <input type="text" data-ng-model="name" />

  <script src="Scripts/angular.min.js"></script>
</body>
```

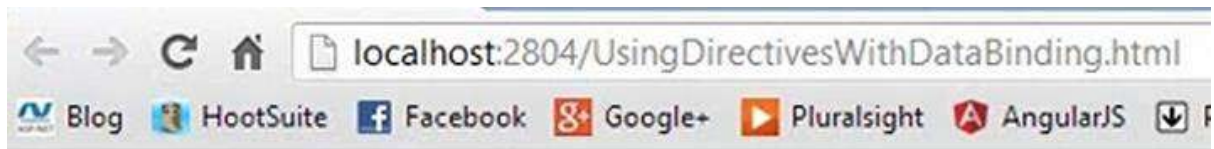
Next what I'm going to do is as they type I would like to bind to the value that they type. Because we now, behind the scenes, made a property up in memory called "name" I can now bind to that, and we do that using the double brackets "{{". We're using the kind of handlebars or moustache -type style of data-binding if you've used those script libraries before.

```
<input type="text" data-ng-model="name" /> {{ name }}
```

That'll simply type out name as we type it.

Let's go ahead and run this and you're going to see that although it won't be super, super impressive it should work for us and that's all we have to do to get started with **Angular**.

We'll go ahead and type the name and there we go...



Name:

Dan Dan

You can see that as I type it automatically binds it, and that's pretty damned easy, right?

- Include the **ng-app**
- Include the **ng-model**
- Bind to that model.

This is pretty primitive and we're going to go much deeper here, but that's how we can get started.

Iterating with the ng-repeat Directive

Iterating with the ng-repeat Directive

```
<html data-ng-app="">
...
  <div class="container"
    data-ng-init="names=['Dave','Napur','Heedy','Shriva']">

    <h3>Looping with the ng-repeat Directive</h3>
    <ul>

      </ul>
  </div>

...
</html>
```

00:14:07

The next thing we can do is we can actually iterate through data.

So I have another directive here called **ng-init** and this isn't one I use a lot in real life apps because we're going to get into controllers and things like that later in the video, but this is going to give me some initialisation data that I want to actually bind to and display so we can come in and use another directive in Angular called **ng-repeat**.

```
<div class="container"
  data-ng-init="names=['Dave','Napur','Heedy','Shriva']">

  <h3>Looping with the ng-repeat Directive</h3>
  <ul>
    <li data-ng-repeat="name in names">{{ name }}</li>
  </ul>
</div>
```

We're going to say **ng-repeat** and then I'm going to give a variable here. For each name in the names variable write out that name.

In this case "name" is not the same thing as I just demonstrated: "name" is just a variable. If I put "foo" here then I would bind to "foo" here.

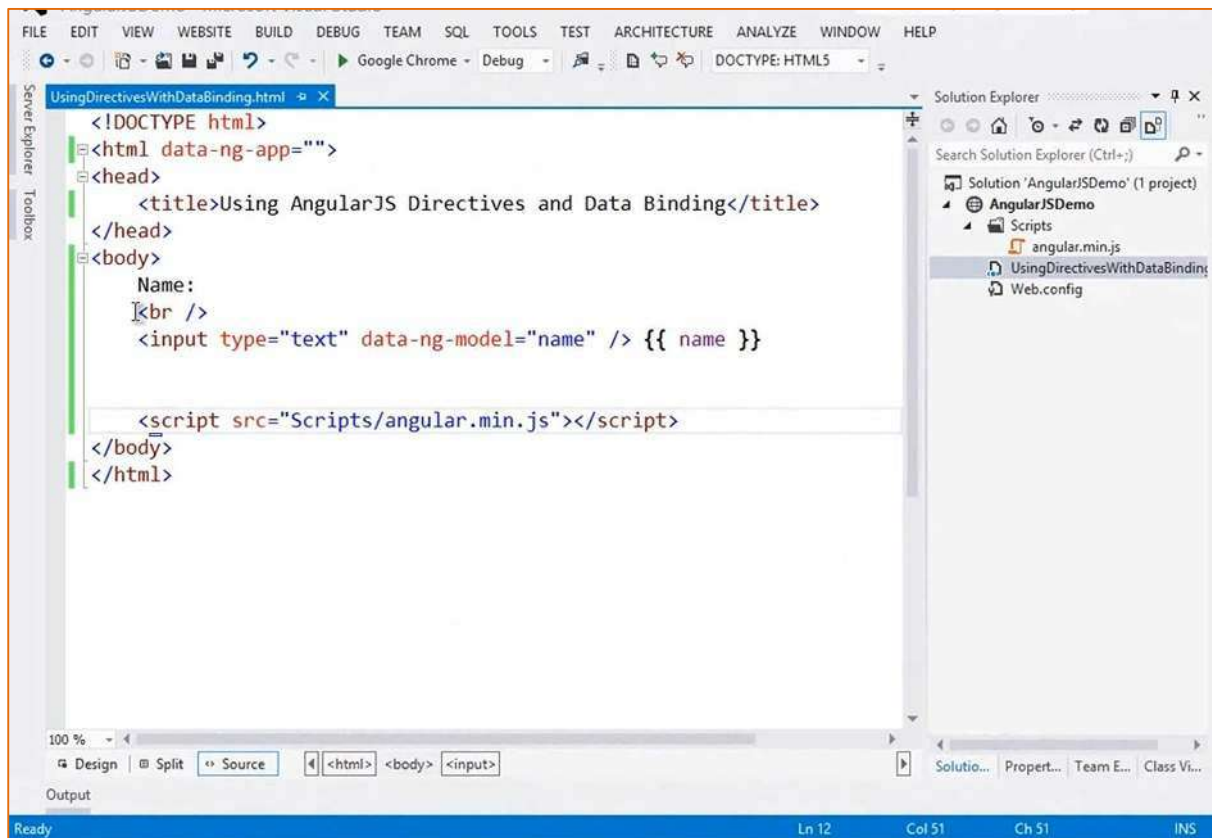
This provides a very easy way to duplicate ****s.

In this case we have four names and so we would get four ``s with the name written out automatically.

So again, we have the **ng-app**, the **ng-init**: these are two directives. Then the third is **ng-repeat** which will simply loop through all the names, and then data-bind or apply the value into the ``.

Let's look at an example of that really quickly.

ng-Repeat Example



00:15:20

So we can come back into our web page and I'm going to do the **ng-init**.

```
<body data-ng-init="names">
```

I'm going to give it an array with a couple of names.

```
<body data-ng-init="names=['John Smith', 'John Doe', 'Jane Doe']">
```

This is a primitive way to initialise some variables with data.

If we come down [below the `input`] I can do a `` and `` and do an **ng-repeat** – yet another directive – for each name in `names`. I've already used "name" here:

```
<input type="text" data-ng-model="name" /> {{ name }}
```

So I'm going to call it something different. Let's say for each `personName` in `names`:

```
<ul>
  <li data-ng-repeat="personName in names"></li>
</ul>
```

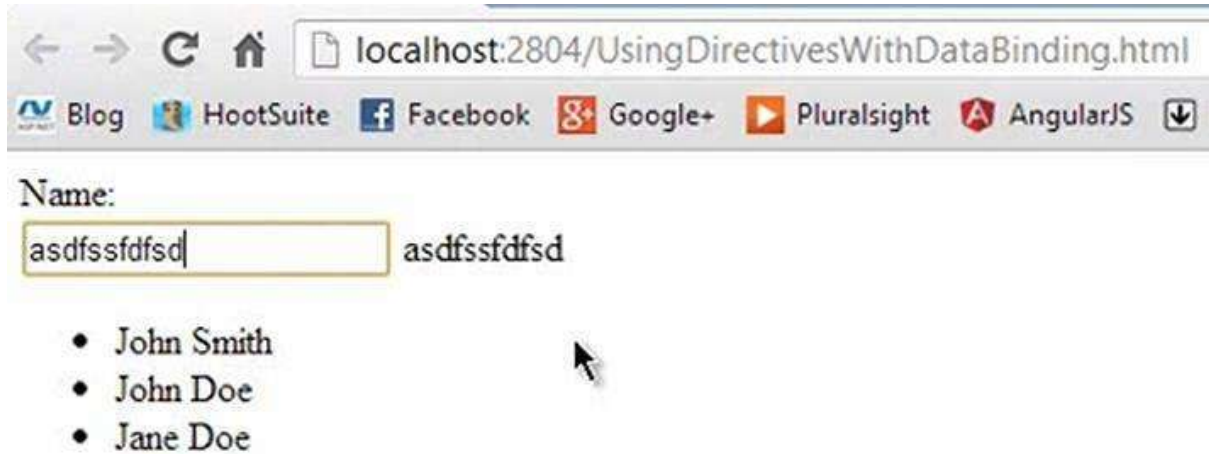
Let's go ahead and bind to `personName`:

```
<li data-ng-repeat="personName in names">{{ personName }}</li>
```

Very easy. Very simple. So ...

- We've now initialised our data with the **ng-init**.
- We're going to iterate through our data with the **ng-repeat**
- We simply give it the name and it's going to put that name into the variable when we bind to it

If we go ahead and run this you'll see that we just get a nice little list written out, nothing too fancy, but it does work, and if the name binding at the top still works as well.

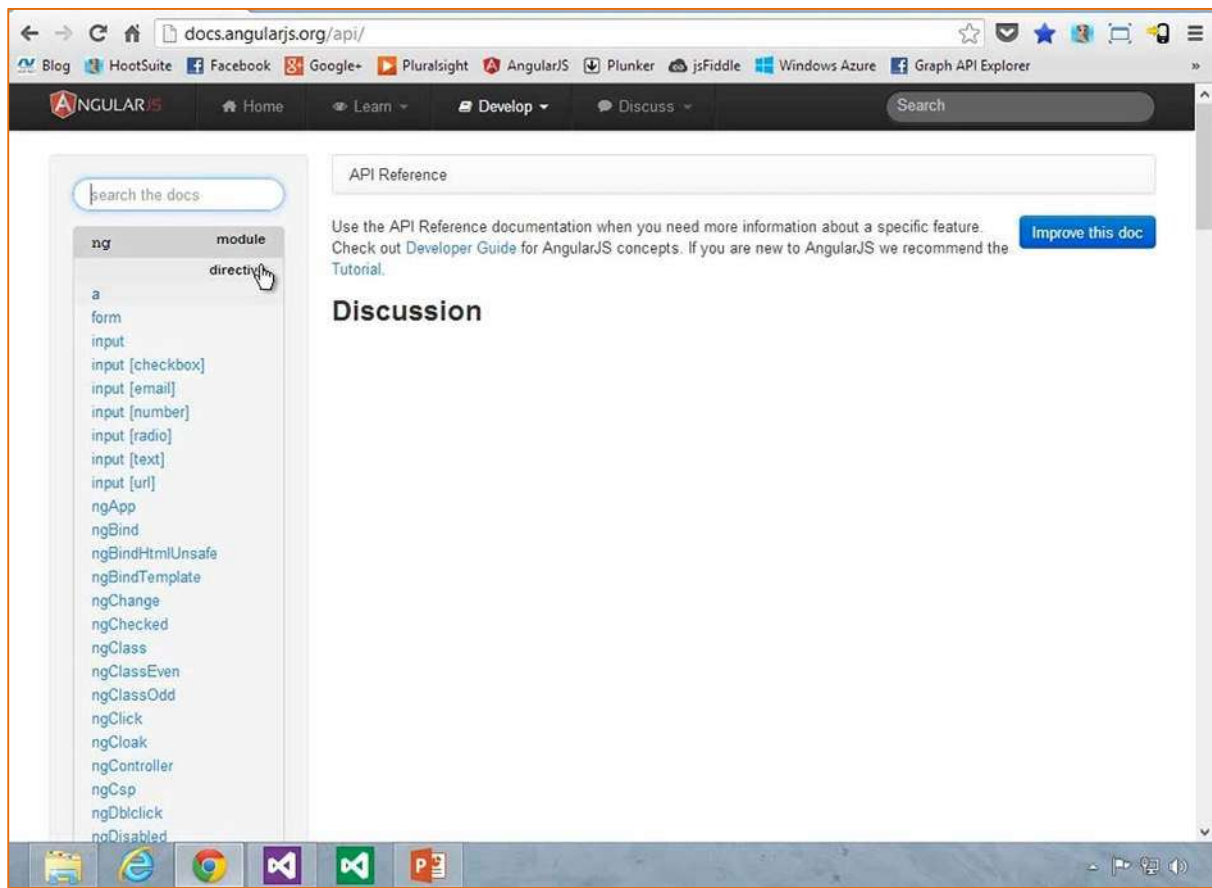


When you use directives one of the nice things you can do is go off to the documentation. One of the best things you need to know about is go to “Develop”...



.. and select “API Reference”.

The AngularJS API Reference for Directives



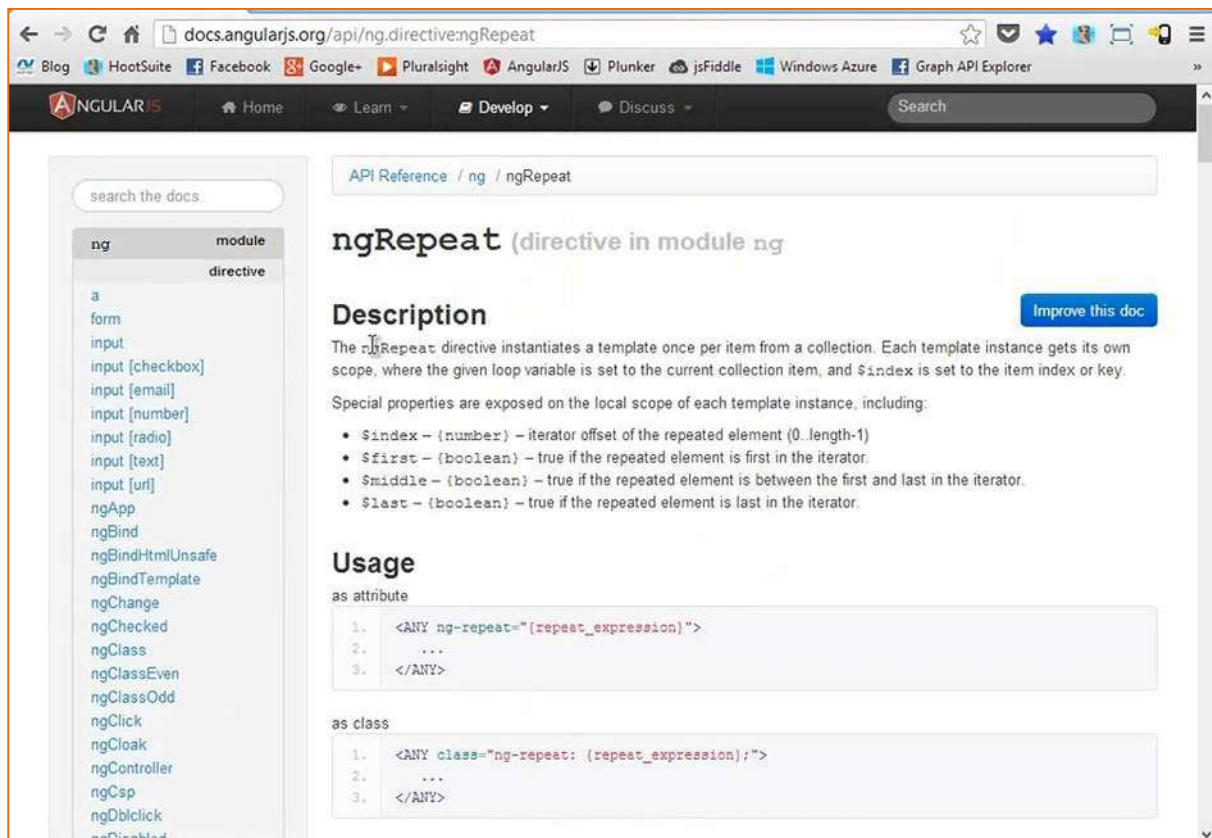
00:16:50

In the API Reference you'll see that right at the top we have different directives.

There's a whole bunch of these. I'm only showing a very small subset of what's available. I'll show some others as we move along here.

So for instance if we want to know more about **ngRepeat** we can click on that.

ngRepeat Documentation



The screenshot shows the AngularJS documentation page for the `ngRepeat` directive. The browser address bar shows `docs.angularjs.org/api/ng.directive:ngRepeat`. The page title is "API Reference / ng / ngRepeat". The main heading is "ngRepeat (directive in module ng)". Below this is a "Description" section with a blue "Improve this doc" button. The description states: "The `ngRepeat` directive instantiates a template once per item from a collection. Each template instance gets its own scope, where the given loop variable is set to the current collection item, and `$index` is set to the item index or key. Special properties are exposed on the local scope of each template instance, including:" followed by a bulleted list of properties: `$index`, `$first`, `$middle`, and `$last`. Below the description is a "Usage" section with two code blocks. The first is "as attribute" showing a template snippet:

```
1. <ANY ng-repeat="(repeat_expression)">
2.   ...
3. </ANY>
```

 The second is "as class" showing another template snippet:

```
1. <ANY class="ng-repeat: (repeat_expression);">
2.   ...
3. </ANY>
```

 A left sidebar contains a search bar and a list of directives under the "ng" module, including `form`, `input`, `input [checkbox]`, `input [email]`, `input [number]`, `input [radio]`, `input [text]`, `input [url]`, `ngApp`, `ngBind`, `ngBindHtmlUnsafe`, `ngBindTemplate`, `ngChange`, `ngChecked`, `ngClass`, `ngClassEven`, `ngClassOdd`, `ngClick`, `ngCloak`, `ngController`, `ngCsp`, `ngDbclick`, and `ngDisabled`.

00:17:09

It gives us some info. It gives us some different samples of it and a look at how it works.

There's even some tests ["End to end test" tab under "Source"] on how to test the repeater and do that kind of thing if you'd like as well.

So there's a lot of great stuff you can do with directives, and we'll start to see more of these as we move along.

Using Filters

Using Filters

```
<ul>
  <li data-ng-repeat="cust in customers | orderBy:'name'">
    {{ cust.name | uppercase }}
  </li>
</ul>
```

```
<input type="text" data-ng-model="nameText" />
<ul>
  <li data-ng-repeat="cust in customers | filter:nameText |
    orderBy:'name'">
    {{ cust.name }} - {{ cust.city }}</li>
</ul>
```

00:17:25

The next thing we can do with Angular is apply filters.

Let's say that as we bind to, say a customer name, and we do that process we want to upper-case it. Now I could upper-case it in my data model, which we'll get to in a little bit later, but an easy way to do this type of thing is to apply an AngularJS filter.

```
{{ cust.name | uppercase }}
```

All this will do is this pipe [|] is a separator between the data binding statement and something called a **filter**.

There's a few filters built-in. We'll look at that in the documentation once I get into the demo and run off to the web page. "uppercase" says upper-case it, "lowercase" says lower-case it, you can restrict it if it's an array and you want to output that array and you want to restrict it, limit it to say three out of the five or whatever it may be.

Then when it comes to **ng-repeat** something that's very cool: in this case we're going to say for each cust in customers I want to filter by "nameText".

```
<li data-ng-repeat="cust in customers | filter:nameText |
  orderBy:'name'">
```

Above this we have this ng-model, which we've already looked at.

```
<input type="text" data-ng-model="nameText" />
```

As they type, the value they type will automatically be applied to first filter down the customers based upon what was typed.

So if you type "da" and "dan" was in there then it'll automatically pick me or any other people that start with "da" or have "da" in the name.

Then we're going to take those results and filter again – we're going to order those results by a "name" property:

```
<li data-ng-repeat="cust in customers | filter:nameText |
      orderBy:'name'">
```

I'll show this in the demonstration coming up.

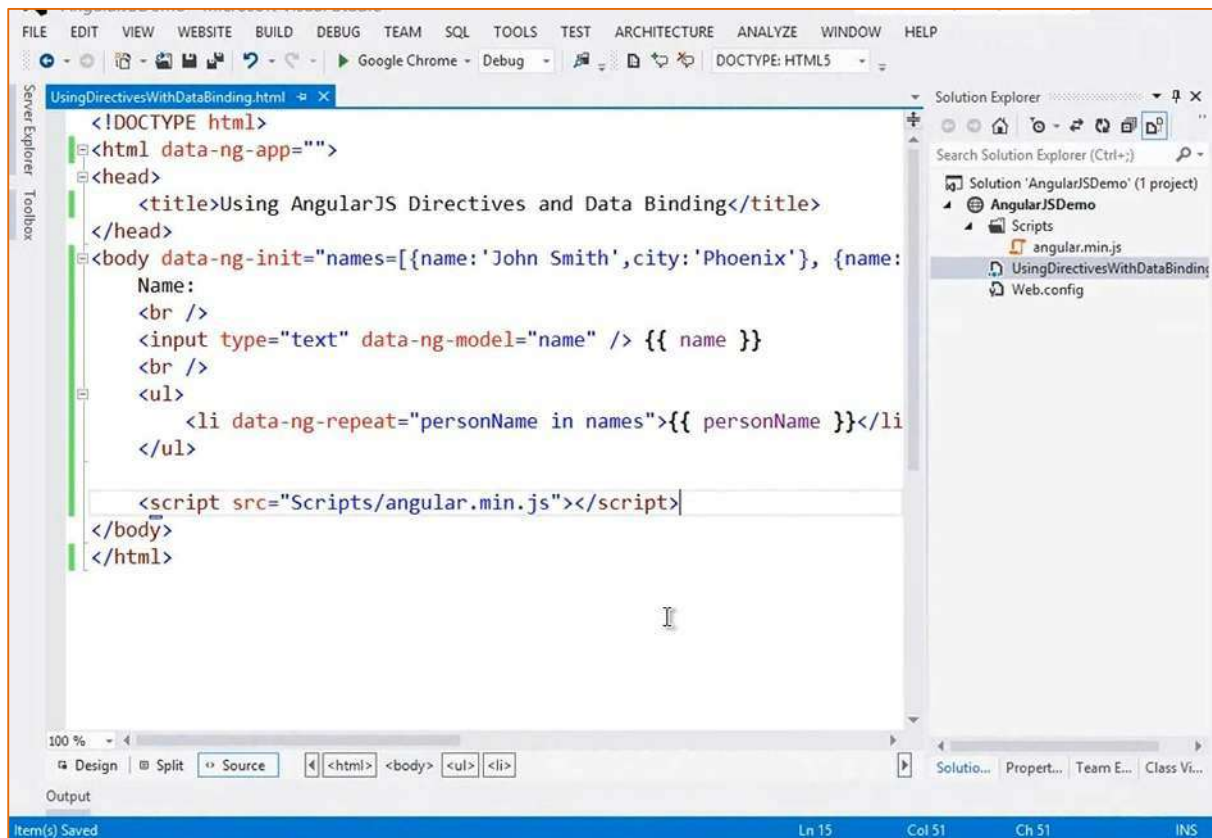
What that will ultimately do is if we have 50 customers and we filter for all those that have "John" in the name then all the "John"s would be shown and we would then order those by "John Doe", "John Smith" and that type of thing.

What we're going to do in this case is once that filtering goes through and we order those we'll then write out the customer name and the customer city.

```
<li data-ng-repeat="cust in customers | filter:nameText |
      orderBy:'name'">
  {{ cust.name }} - {{ cust.city }}</li>
```

Let's go ahead and jump into a demonstration of doing it this way.

Using Filters Demo



00:19:20

Back in our web page I've changed the **ng-init** a little bit. Instead of just having an array of strings, I have an array of objects.

```
<body data-ng-init="names=[{name:'John Smith',city:'Phoenix'}, {name:
```

You'll notice that each object has a name and a city property. I just have three of these in here: John Doe, John Smith and Jane Doe, from San Francisco, New York and Phoenix.

```
{name:'John Doe',city:'New York'},
```

```
{name:'Jane Doe',city:'San Francisco'}]"]>
```

I'm going to have to change this [contents of the body of the page] now. "names" still stays the same, but I'm going to go ahead and change that too. Let's say this is a list of customers:

```
<body data-ng-init="customers=[{name:'John Smith',city:'Phoenix'}, {n
```

We'll name it "customers" and we'll change the `` statement accordingly to let's say "cust":

```
<ul>
```

```
  <li data-ng-repeat="cust in customers">{{ personName }}</li>
```

```
</ul>
```

Now what I'm going to have to do is write out the **cust**. – and now we can get into the properties and we can do "name" here:

```
<ul>
  <li data-ng-repeat="cust in customers">{{ cust.name }}</li>
</ul>
```

Now if I wanted I could put in a space and maybe a dash or something and we could do `cust.city` and now we're going to data-bind both those properties.

```
ng-repeat="cust in customers">{{ cust.name }} - {{ cust.city }}</li>
```

I could even come into here, just to show you, I could even do it this way if I wanted, and that would work too.

```
ng-repeat="cust in customers">{{ cust.name + ' ' + cust.city }}
```

But I'm going to break these out into two separate data-binding statements. So now we say for each `cust in customers` let's go ahead and write out `cust.name` and `cust.city`.

Let me go ahead and just make sure this works and then we'll apply some filters.



Name:

asfdfsffsdfsdf asfdfsffsdfsdf

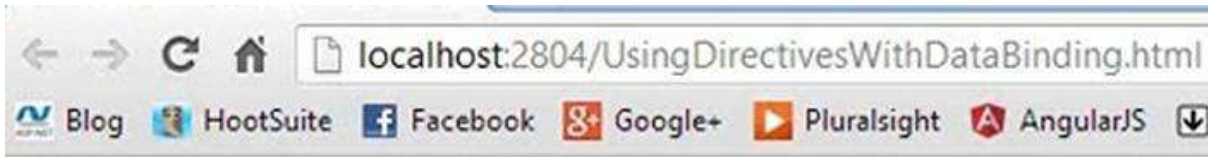
- John Smith - Phoenix
- John Doe - New York
- Jane Doe - San Francisco

It looks like it does [work]. You can see the cities now being written out, but notice as I type nothing really happens that's useful or interesting.

What I'm going to do is come in and let's do a filter by and whenever they type a name instead of data-binding to it I want to use it as a filter. So we're going to filter by the name property that's in our model.

```
<li data-ng-repeat="cust in customers | filter:name">{{ cust.
```

Let's go ahead and test this. You'll notice that when we type "s" the "S" from "San Francisco" also pulls up because I didn't tell it anything specific – just enter everything:



Name:

- John Smith - Phoenix
- Jane Doe - San Francisco

So we have “Smith”, we have “Doe”. We can do both of those. We can do “New York” ...

Name:

- John Doe - New York

You can see all that works and it’s all live. We can also use “orderBy”. Let’s see if we have things in the right area. It looks like if I do “John” and we order by say city then obviously Phoenix is out of place with New York, so we can come in and do another pipe and “orderBy” and then in quotes I give it the property. Let’s order by city and then it will bind those remaining customers and order them by city.

```
ust in customers | filter:name | orderBy:'city'>{{ cust.name }} - {{
```

Let’s go ahead and search for “John” here, and notice that “John” in New York now appears first, and Phoenix follows which of course was not the case with the data.

Name:

- John Doe - New York
- John Smith - Phoenix

So that’s an example of applying not only some data-binding and some **ng-repeat** –type of directives, but also how we can apply filters and orderBys and I could even do upper-case if I wanted.

Let’s say we wanted the name to be uppercase and the city to be lowercase.

```
cust.name | uppercase }} - {{ cust.city | lowercase }}</li>
```

Now what will happen is it will automatically do that for us, as you can see here:

Name:

- JOHN DOE - new york
- JOHN SMITH - phoenix

Notice though that as I type if I do “John” [using mixed lower-case] it still works. It still does the filters – it still filters and sorts. “Jane” still works and all that.

So these are some of the built-in sorts and filters you get out of the box.

Again, if you go off to <http://angularjs.org> , go to “API Reference” and then scroll on down a little bit you’ll see a whole list of the filters.



If I had a number that I wanted to convert into a currency with a \$ sign for instance or a £ sign or whatever currency you’re in then I would just say “| currency” and it would automatically do that.

If I had a date and I wanted to format it a certain way, and you can control that by the way, you could do |date. There’s a lot of different things you can do here.

What’s really nice about Angular is not only can we write our own custom directives but I can even write my own custom filters if I want to get a little more advanced with this.

So, a very powerful framework and we’ve only scratched the surface so far.

What we're going to do next is start talking about the MVC part of Angular – the Model, the View, and the Controller, and we'll see how all that fits in.

Module 3: Views, Controllers and Scope



Views, Controllers and Scope

00:23:40

In this part of the tutorial we're going to talk about **Views**, **Controllers** and a really integral part of Angular called **Scope**, which is really another term for **ViewModel** if you've used that term before.

View, Controllers and Scope



\$scope is the "glue" (ViewModel) between a controller and a view



00:23:47

The way it works in Angular is you have a **View**, which is what we've been doing in the previous section with our **Directives**, our **Filters** and our **Data Binding**.

But we don't want to put all of our logic into the **View** because it's not very maintainable or testable or all those types of things.

Instead we're going to have a special little JavaScript object – a container - called a **Controller**. The **Controller** will drive things. It's going to control ultimately what data gets bound into the **View**. If the View passes up data to the controller it will handle passing off maybe to a service which then updates a back-end data store.

The glue between the View and the Controller is something called the **Scope**, and in Angular you're going to see a lot of objects or variables that start with \$. **\$scope** represents the scope object.

When I say it's the glue, it literally is the thing that ties the controller to the view.

The view doesn't have to know about the controller, and the controller definitely doesn't want to know about the view.

You'll see that the view can know about the controller because there's a directive for that if you'd like to use it but the controller itself, to make it testable and a few things – loosely coupled and

modular and all that good stuff – shouldn't know anything about the view. In fact you should be able to define a controller that you can bind to different views. Maybe you have a mobile view, you have a desktop view or whatever it may be.

So the scope is this glue between them.

Now for folks that have worked with **Knockout** or some of the different data binding frameworks out there, not just JavaScript but other desktop modes and things, you might have heard the term "**ViewModel**".

A **ViewModel** literally is the **model** – the data – for the view. Well that's really all the scope is. The scope is our ViewModel and it's the glue between the view and the controller.

Creating a View and Controller

Creating a View and Controller

```
<script>
  function SimpleController($scope) {

    $scope.customers = [
      { name: 'Dave Jones', city: 'Phoenix' },
      { name: 'Jamie Riley', city: 'Atlanta' },
      { name: 'Heedy Wahlin', city: 'Chandler' },
      { name: 'Thomas Winter', city: 'Seattle' }
    ];
  }
</script>
```

Basic controller

00:25:40

Here's an example of a really simple controller called, oddly enough, **SimpleController**. You'll notice an interesting thing here in the parameter signature. You'll see that we pass **\$scope**. This is **dependency injection** that's built into AngularJS.

What this is going to do is Angular, when this controller gets used, will automatically inject a **scope** object in. You'll see that from there we'll take that object and add in a property onto it called **customers** which is simply an array of object literals. So we have our same scenario with name and city here.

What this controller can do then is serve as the source of the data for the view but the controller again shouldn't know anything about the view, so how would we possibly communicate customers over? Well that's why we're injecting scope. Scope is to be that view between the controller and the view.

If we come up to our view up here the scope gets injected and then that scope is going to be automatically bound into the view once the view knows about this controller.

So here's what it looks like...

Creating a View and Controller

```
<div class="container" data-ng-controller="SimpleController">
  <h3>Adding a Simple Controller</h3>
  <ul>
    <li data-ng-repeat="cust in customers">
      {{ cust.name }} - {{ cust.city }}
    </li>
  </ul>
</div>

<script>
  function SimpleController($scope) {

    $scope.customers = [
      { name: 'Dave Jones', city: 'Phoenix' },
      { name: 'Jamie Riley', city: 'Atlanta' },
      { name: 'Heedy Wahlin', city: 'Chandler' },
      { name: 'Thomas Winter', city: 'Seattle' }
    ];
  }
</script>
```



00:26:50

You'll notice up here [at the top] we have an **ng-controller**, SimpleController. That'll automatically tie in this [our controller in the second half of the slide].

When this [the controller] gets initialised the scope gets passed in but it's initially empty.

Well we're going to add a **customers** property. What's going to happen then is this controller will be used by the view. The controller itself though isn't going to be called – it's going to go through the scope. The scope itself is implicitly available – in this case to the entire div: from the start of the div to the end of the div.

Now look at the **ng-repeat** here. It's the same thing I did earlier in the demo, except in this case it's binding to the scope's **customers** property

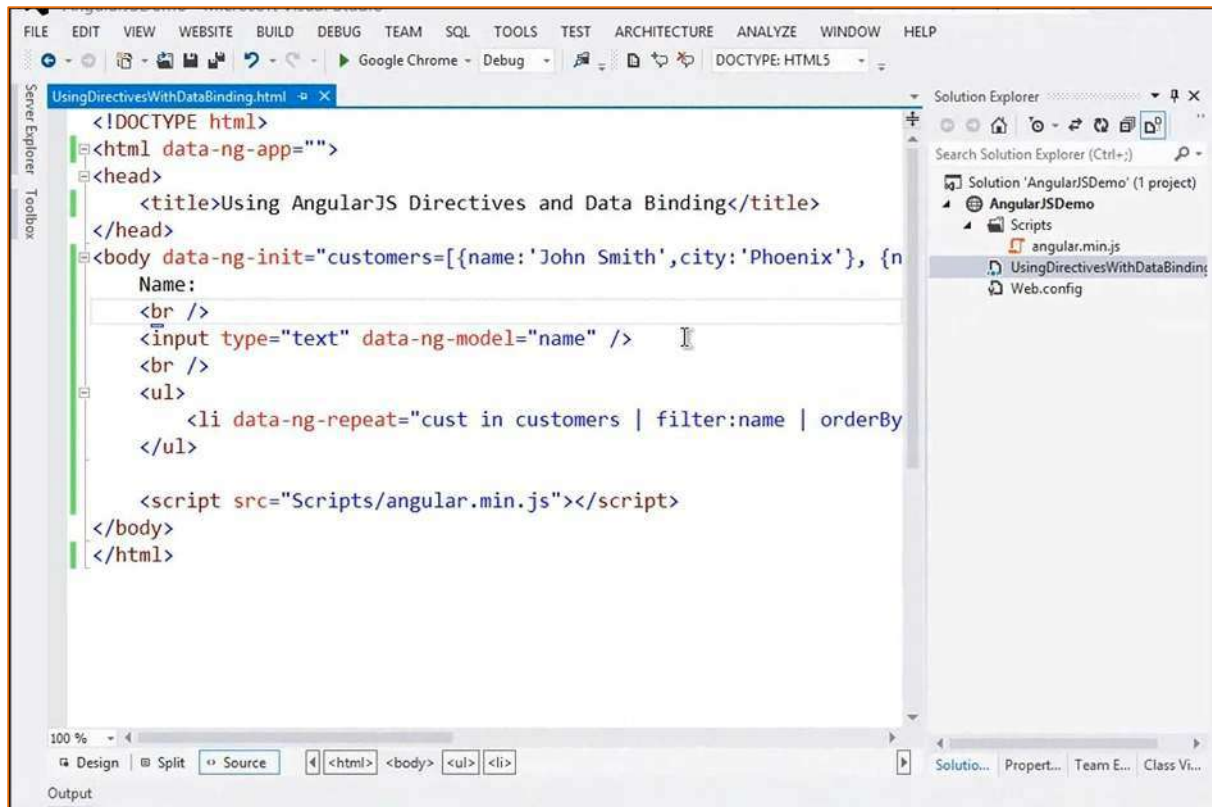
```
<ul>
  <li data-ng-repeat="cust in customers">
    {{ cust.name }} - {{ cust.city }}
  </li>
</ul>
```



What's going to happen now is the view, because it knows about the controller, automatically gets passed the scope. Angular does that kind of behind the scenes – it just happens automatically. We're going to then control in our view what properties we want to bind to. In this case we're going to bind to the customers.

From here it's standard data binding like we've seen earlier: we bind to the name and we bind to the city and we're kind of off and running.

Let's take a look at an example of how we can create our custom controller here and tie it into our view.



00:28:05

We already have some data up here [in the body tag] but I don't want to use **ng-init** in this case. It might have come from maybe a back-end database through an AJAX call, something along those lines.

I'm going to get rid of the ng-init and come on down and write my own custom script. Now keep in mind I'm going to do this inline purely for demo purposes.

```

<head>
  <title>Using AngularJS Directives and Data Binding</title>
</head>
<body>
  Name:
  <br />
  <input type="text" data-ng-model="name" />
  <br />
  <ul>
    <li data-ng-repeat="cust in customers | filter:name | orderBy
  </ul>

  <script src="Scripts/angular.min.js"></script>

  <script></script>
</body>
</html>

```

I generally like to break this out into its own script – and I’ll show you how I do that later when we get more into a real life –type app. But for now let’s make our same function like I showed earlier: **SimpleController**.

In the docs you’ll often see it abbreviated to Ctrl or something. I don’t like to do that. In fact I recommend against it. I think you should be pretty explicit with your names but you can certainly do it whatever way you like.

We need the scope, because, again, that’s the glue between this particular object – our function SimpleController – and our view that’s going to get the data bind to it.

```
<script src="Scripts/angular.min.js"></script>

<script>
  function SimpleController($scope)
</script>
```

Now I’m going to tag onto that scope, which right now is empty, a **customer** property. I’ll just paste in our object literal here:

```
<script>
  function SimpleController($scope) {
    $scope.customers = [
      { name: 'John Smith', city: 'Phoenix' },
      { name: 'John Doe', city: 'New York' },
      { name: 'Jane Doe', city: 'San Francisco' }
    ];
  }
</script>
```

Now this array of object literals is now going to be bound into the customers which then get filtered and do all that good stuff.

We’ve now done the same thing but if we run this it’s not going to work because the view doesn’t know about **SimpleController**.

One way we could fix this is I could come up to the body and say **ng-controller="SimpleController"**:

```
<body data-ng-controller="SimpleController">
```

But you can also have different controllers for different parts of a given page. The scope would only apply to where you add that controller. So, for instance, if I came in and only wanted the scope to apply here...

```
<body>
  <div data-ng-controller="SimpleController">
    Name:
```

.. then anything in the end and start tag for the div is now in that.

```

<body>
  <div data-ng-controller="SimpleController">
    Name:
    <br />
    <input type="text" data-ng-model="name" />
    <br />
    <ul>
      <li data-ng-repeat="cust in customers | filter:name | or">
    </ul>
  </div>
  <script src="Scripts/angular.min.js"></script>

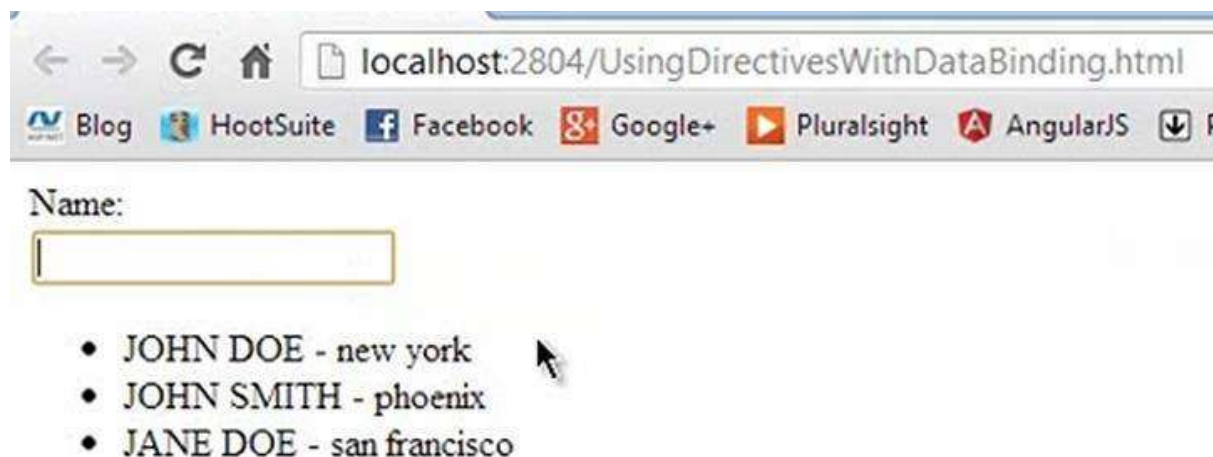
```

00:30:10

So now we have two properties in the scope. The **ng-model** is going to add a property in the scope called **name**, and we can actually get to that now in the controller by saying **\$scope.name**

Then of course we have **customers** which will be retrieved through the scope as well.

If we run this now that we've defined the controller we should see the same exact behaviour here.



It still works. All my filters still work. I can type to filter to "new york"

That's an example of a simple controller.

In a section that will be coming up in the video I'm going to be showing it in a different way, and what I consider a better way to do it. But again, we're trying to walk through the fundamentals step-by-step on how to use this.

Let's go back and take a look at some other things you can now do.

Module 4: Modules, Routes and Factories

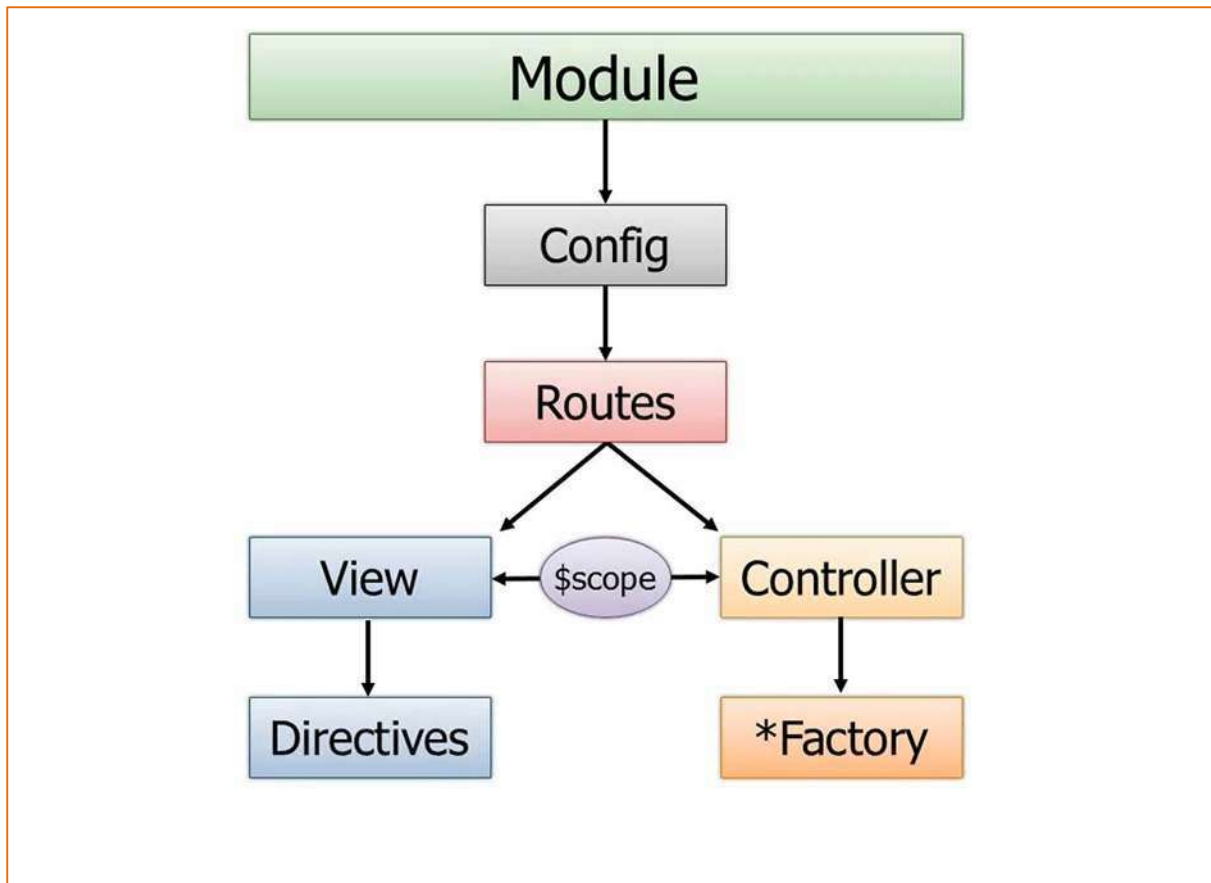


Modules, Routes and Factories

00:31:02

Now that you've seen data binding, directives, filters and controllers, and you've seen how the scope actually can integrate between a view and a controller it's now time to really move it up a notch and talk about **modularity** and some more SPA-oriented concepts like **routes**.

In this section we're going to talk about how to create modules, and how modules can actually be used to create other things like controllers, routes, factories and services, and how all this fits together.



00:31:30

One of the things I really struggled with when I very first started learning **AngularJS** was I didn't really see the big picture of how everything fit together.

I'm going to show you a really simple diagram - this is really an over-simplification – but I think it breaks it down pretty easily.

So a module can have something off of it called a **config** function, and it can be defined to use different **routes**. Now routes again are really important in the SPA world because if you have different views and those views need to be loaded into the shell page then we need a way to be able to track what route we're on and what view that's associated with and then what controller goes with that view and how we do all of that marrying together of these different pieces.

When you define a route in **AngularJS** you can define two things on that route – two of the key things, I should say.

One of those is the view. So what view when that route such as "unit.org/orders" then maybe go to "orderspartial.html" or "ordersfragment.html" or whatever you want to call it.

Then that view needs a controller. Instead of hard-coding the controller into the view – which works and you can certainly do it: I showed that in the previous section – we can actually go in and do this on our own through the route. This is the way I would definitely recommend you do it.

A given controller would then of course have access to the scope object which then the view will bind to. I talked about that a little bit earlier.

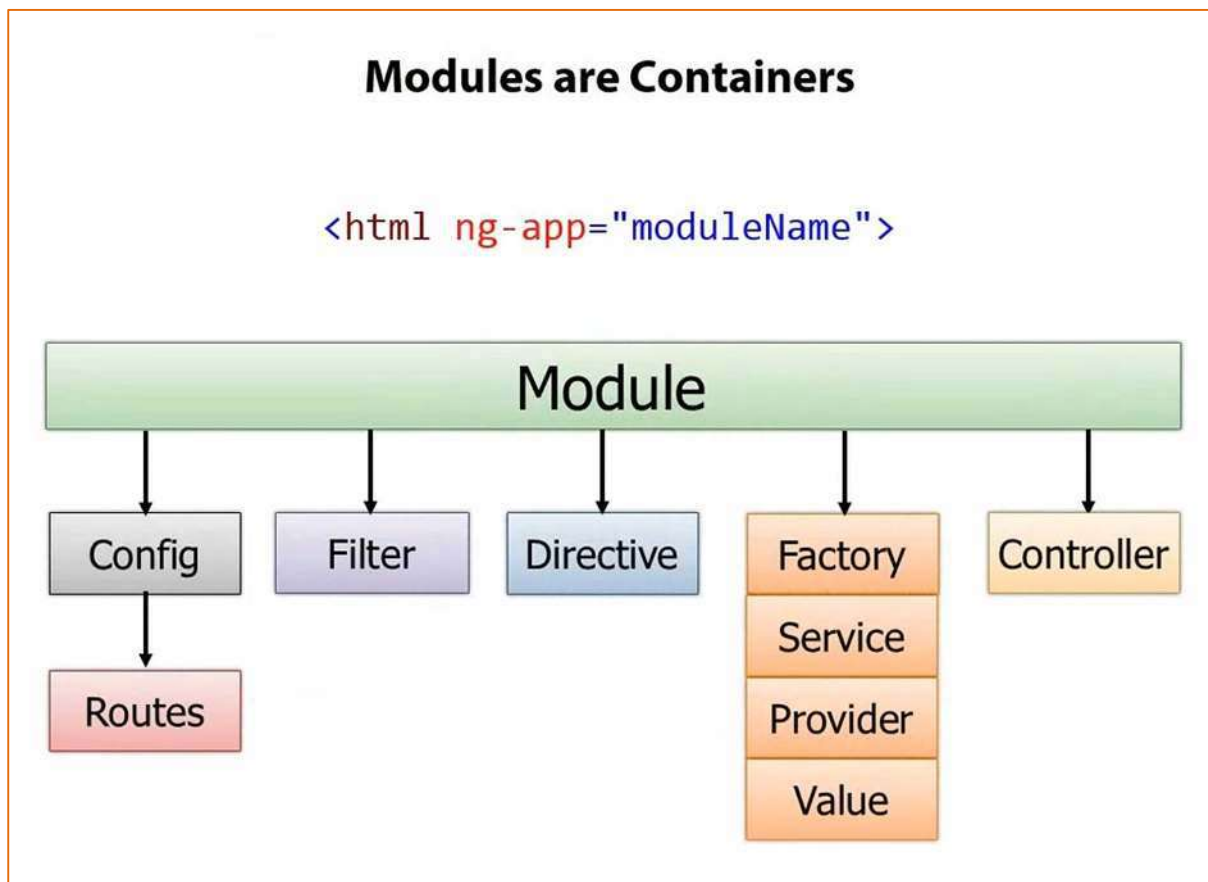
And then controllers rather than having all the functionality to get the data and update the data and perform CRUD operations and things like that, in a more real life application they'll call out to factories or... I put a star there because you might have services, providers or even values you want to get. There are a lot of different ways you can access data. Even resources.

On the views we of course then have directives and filters and those types of things.

There's even more to the overall picture but this is one of those things I wish I would've seen this right upfront because I would have kind of instantly had that light bulb moment where the light goes on and you say "OK. So you define a route, a route has a controller and a view and then the controller can load data from factories and services and things."

What we're going to do is talk about each of these individual pieces and how in a single page application you can actually define routes and use those.

Modules are Containers



00:33:57

The first thing to talk about is although I created the controller in some previous demo's right in the view that's not the recommended way to do it.

Keep in mind you might actually have different views. You might have a mobile view, maybe one specific to iPad or Surface or something like that, and maybe have one for desktops. It's very possible.

So what we need to do is when we define our **ng-app**, earlier we didn't assign it to anything, and so what that did was implicitly create a scope behind the scenes and it still worked we saw with the data binding and the filters and all that good stuff. We now want to graduate to a more modular application and Angular is very, very modular if you take advantage of it.


There is a **module** object and I'll show you how to create that – you literally just say **angular.module** – and off of the module you can configure the routes, create custom filters, custom directives. You can get data from different sources using Factories, Services, Providers or Values. You can then even create Controllers using the module. You can think of the module as really just an object container, and then in that container you can have all these different things that you see here.

What's important is that once the module is given a name that's when you're going to go into **ng-app** and whatever the module name is that's what's going to go here. So they could have just as

easily called it **ng-module**, but I actually like **ng-app** – it makes sense I think. It really just means “Heh! What’s the name of the module you’ve defined in JavaScript?”

Creating a Module

Creating a Module



What's the
Array for?

```
var demoApp = angular.module('demoApp', []);
```

00:36:00

Here's what it looks like.

It's really, really simple to create a module. Once you've referenced the Angular script you're going to have access to an **angular** object.

Off of this object you can get to all kinds of good stuff. You can get to the **jqLite** functionality for **jQuery DOM** –type manipulation. But you can also get, as you can see here, to the module.

In this example you can see I called my module **demoApp**. You might wonder what exactly is the empty array for? I know I did the first time I saw it.

The answer is this is where **dependency injection** comes in because your module might actually rely on other modules to get data.

Creating a Module

What's the
Array for?

```
var demoApp = angular.module('demoApp', []);
```

```
var demoApp = angular.module('demoApp',  
  ['helperModule']);
```

Module that demoApp
depends on

00:36:17

Here's an example of that.

We have **demoApp** again. We say "Heh Angular! Create me a module called demoApp." I need to rely on another module. In this case I just called it **helperModule**. This would be some other JavaScript file that you might reference. I then has another angular reference and in the quotes here [first module 'demoApp' quotes] you would see **helperModule** and it would have maybe just filters and directives in it and that's all it has.

What I'm doing is telling Angular "Go find that module. Go look it up wherever it is and inject it in and make it available to my controllers and directives and filters and factories and all that stuff that this particular module might have.

It's a very powerful feature. If you've ever used **requireJS** or something that's more modular based then this will kind of ring home with you. If you haven't though it's just a very flexible way to, at run time, include other modules.

Creating a Controller in a Module

Creating a Controller in a Module

```
var demoApp = angular.module('demoApp', []);
```

Define a Module

Define a Controller

```
demoApp.controller('SimpleController', function ($scope) {  
  $scope.customers = [  
    { name: 'Dave Jones', city: 'Phoenix' },  
    { name: 'Jamie Riley', city: 'Atlanta' },  
    { name: 'Heedy Wahlin', city: 'Chandler' },  
    { name: 'Thomas Winter', city: 'Seattle' }  
  ];  
});
```

00:37:15

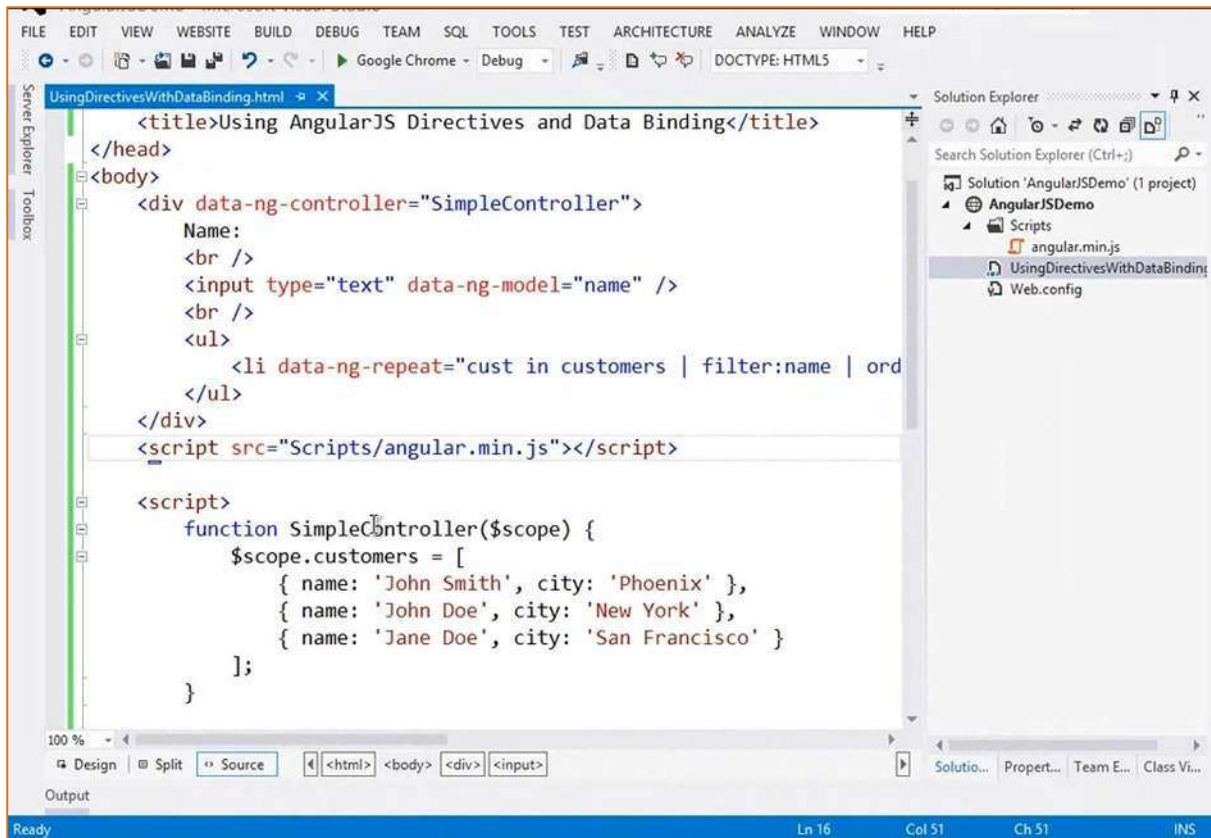
Now here's an example of how we can use that module to actually create a controller. This is the more, I would say, "official" way to create a controller.

You'll notice up top that we have our **demoApp**. We just create that the same way. In this case we're saying that it has no dependencies - an empty array.

What we're going to do with that module though is we're going to use it to define a controller and we're going to call this as it shows: **SimpleController**. Notice that I just have an anonymous function here nested inside. So the second parameter here is "Ok, so what is the controller object?" Well function of course is an object in JavaScript so we're going to inject in the scope and then from there I do the same exact thing we did before.

The key is that once I've defined this I then need to go in and make sure that my **ng-app** points up to **demoApp** in the strings. I could even then in the view do **ng-controller**, like I did earlier, is **SimpleController**.

Once we get to routes here I'm going to show you how even that can change. Let's go ahead and take a quick demonstration of fixing up our previous function and going with the more modular approach.



00:38:28

OK. So we have our **SimpleController** here but this isn't really modular. It's just kind of a function out there. So what I'm going to do is just come in and define a variable, let's just call this **demoApp**.

```

<script>
  var demoApp = angular.module('demoApp', []);
  function SimpleController($scope) {
    $scope.customers = [
      { name: 'John Smith', city: 'Phoenix' },
      { name: 'John Doe', city: 'New York' },
      { name: 'Jane Doe', city: 'San Francisco' }
    ];
  }

```

The string we use for the module name doesn't have to be the same as the variable name.

I'm going to leave the rest as it is to show how it works, and then I'm going to change it.

I can then come down and say "Let's add a... and notice for the 'c's we have a config, constructor, controller..."



We're going to use a controller and let's name it SimpleController:

```
}  
demoApp.controller('SimpleController')
```

Now I'm going to give it **SimpleController** because I could give it an anonymous function or I could actually create the controller outside and just assign it in, and that would work absolutely fine.

```
demoApp.controller('SimpleController', SimpleController);
```

Now this already knows about **SimpleController**...

```
<body>  
  <div data-ng-controller="SimpleController">  
    Name:
```

... but **ng-app** doesn't know about **demoApp** so let's just fix that:

```
<!DOCTYPE html>  
<html data-ng-app="demoApp">
```

So now we're ready to go and now this is a little bit modular.

Let's make sure it still runs and then I'll show you the anonymous way to do it.

The next piece of this is you may not even want to keep it [SimpleController] outside. In some cases you may – and I'll show you one more trick to wrap up in a second here – but I'm going to do an anonymous function right inside my SimpleController.

```
demoApp.controller('SimpleController', function ($scope) {  
  $scope.customers = [  
    { name: 'John Smith', city: 'Phoenix' },  
    { name: 'John Doe', city: 'New York' },  
    { name: 'Jane Doe', city: 'San Francisco' }  
  ];  
});
```

Because the **ng-app="demoApp"** knows about this it then knows about the controller. So "SimpleController" the string is known here and it should run exactly the same way.

So that's an example of actually creating a module with a controller.

Another cool trick you can do - and this can be useful depending on how you like to write your JavaScript I think – I'm going to wipe out all of this and I'm going to come in and create a variable called controllers and give it an empty object literal to start.

```

<script>
  var demoApp = angular.module('demoApp', []);

  var controllers = {};

  demoApp.controller();

</script>

```

00:40:50

Then I'm going to define SimpleController and I'm going to give it the function.

```

<script>
  var demoApp = angular.module('demoApp', []);

  var controllers = {};
  controllers.SimpleController = function ($scope) {
    $scope.customers = [
      { name: 'John Smith', city: 'Phoenix' },
      { name: 'John Doe', city: 'New York' },
      { name: 'Jane Doe', city: 'San Francisco' }
    ];
  };

  demoApp.controller();

</script>

```

I can do multiples of these. If I had multiple controllers I could say controllers.controller2 = an anonymous function.

```

controllers.SimpleController = function ($scope) {
  $scope.customers = [
    { name: 'John Smith', city: 'Phoenix' },
    { name: 'John Doe', city: 'New York' },
    { name: 'Jane Doe', city: 'San Francisco' }
  ];
};
controllers.controller2 = func

demoApp.controller();

```

Now I can come in and just pass it [demoApp] controllers:

```
demoApp.controller(controllers);
```

Because this is named SimpleController it's a property off of the object this will still be able to find it. By using this sort of technique – some people like this technique, some people don't – we can come in and [run the app] and you see we get the same exact feature. It still works the same, so there's three ways you can do it.

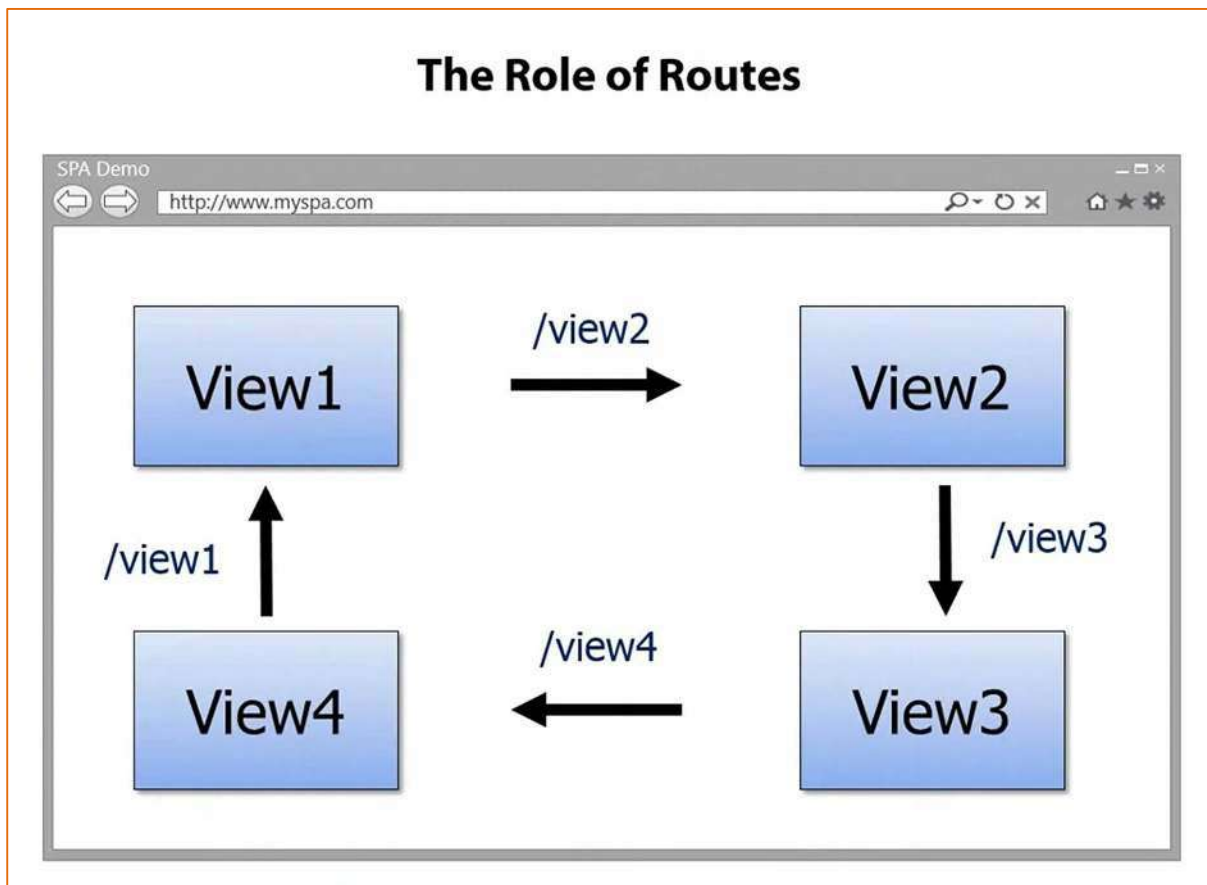
One way you can create an external function and just pass the function in with the controller name.

The second way is you could actually pass a name as a string with an anonymous function.

The third way is we can come in and do this kind of technique. Some people like this because now it's a little easier to prototype things if I needed to, and that kind of stuff.

So that's an example of using a module with a controller.

The Role of Routes



00:42:02

Now once you've defined a module and a controller, at some point if you're building a single page application you're going to need routes because we need to load different views into our shell page.

This will be an example of four different routes.

We have when View1 is clicked maybe there's a link to it and that link is something like `"/view1"`. Typically you have a hash but you'll see that coming up. That will load that view. Then when they click on a link that has View2 in the path then that would load up View2 but it's not going to load up the whole shell page. Angular will only load up the page that you want.

There's two kind of ways you can load it.

First off, the view could be embedded as a script template in the actual shell page and then we could just tell Angular "Heh! The template id to load... it's kinda like saying 'The view id is x'".

The second way is back up on the server you might actually have all these Views and I like to call them "partials" because they're part of a page. You can tell Angular "the template URL for what I want to load" and then you give it the URL to the server, and I'm going to show that coming up here.

Defining Routes

Defining Routes

```
var demoApp = angular.module('demoApp', []);

demoApp.config(function ($routeProvider) {
  $routeProvider
    .when('/',
      {
        controller: 'SimpleController',
        templateUrl: 'View1.html'
      })
    .when('/partial2',
      {
        controller: 'SimpleController',
        templateUrl: 'View2.html'
      })
    .otherwise({ redirectTo: '/' });
});
```



Define Module
Routes

00:43:15

This is a really important feature because we want to be able to go in and load different partials or fragments and then that will be kind of how our SPA works.

What we're going to do is use that **config** that I showed earlier.

We have `angular.module` is "demoApp" with no dependencies.

Now what I'm going to do is configure the module with some routes. Another object that's available in Angular is called the **routeProvider**, as you can see here. It's kind of like the **scope** – it's injected in dynamically just by defining **\$routeProvider** as your parameter.

In this case we're going to say "For the routeProvider when the route is just a slash "/" to the root we want to use SimpleController with View1.html. When the route is "/partial2" we want to use the same controller in this case but you can certainly do a different one, with a template URL of View2.

Now the template URL: you might give it a folder where these partials are going to live, and I actually like to call my folder "partials" but you don't have to.

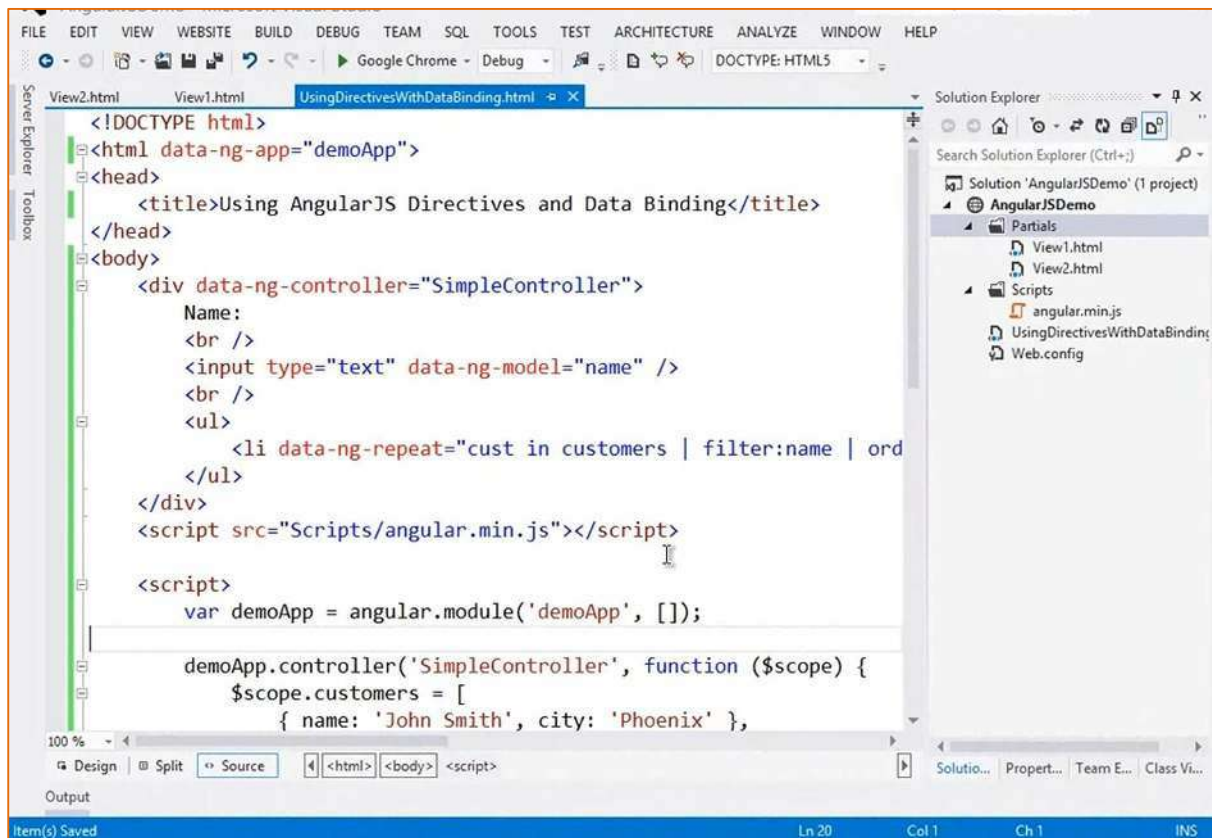
Otherwise if it [the route] doesn't mean any of those routes we're going to redirect back to the root, which ultimately goes back up to here [the "/" route specified with the first ".when" statement] which will load View1.

This is a really cool thing, and once it kind of clicks and you have that light bulb moment, this is really the magic to glue a View to a Controller so that scope gets passed and we can do the data binding and the directives appropriately.

Let's go fix up what we have so far and let's convert this into more of a SPA-type of an application.

Please note that routing has changed in AngularJS 1.2+. For more information visit <http://weblogs.asp.net/dwahlin/archive/2013/08/14/angularjs-routing-changes.aspx>

Defining Routes Demo



The screenshot shows a Visual Studio IDE with a project named 'AngularJSDemo'. The main editor window displays the HTML file 'UsingDirectivesWithDataBinding.html'. The code includes a DOCTYPE declaration, a head section with a title 'Using AngularJS Directives and Data Binding', and a body section. Inside the body, there is a div with a controller named 'SimpleController'. The controller contains an input field for a name and a list of customers. The controller is defined in a script block as follows:

```
var demoApp = angular.module('demoApp', []);

demoApp.controller('SimpleController', function ($scope) {
    $scope.customers = [
        { name: 'John Smith', city: 'Phoenix' },
        { name: 'John Doe', city: 'New York' },
        { name: 'Jane Doe', city: 'San Francisco' }
    ];
});
```

00:44:52

So far in our application we've created a module and we've assigned that to our ng-app so it knows how to get to that, and then that module has this SimpleController so now we have that View knowing about SimpleController.

That's fine and it works, but that can quickly get out of control and really have some code duplication here when you really don't need it.

```
<script>
    var demoApp = angular.module('demoApp', []);

    demoApp.controller('SimpleController', function ($scope) {
        $scope.customers = [
            { name: 'John Smith', city: 'Phoenix' },
            { name: 'John Doe', city: 'New York' },
            { name: 'Jane Doe', city: 'San Francisco' }
        ];
    });
</script>
```

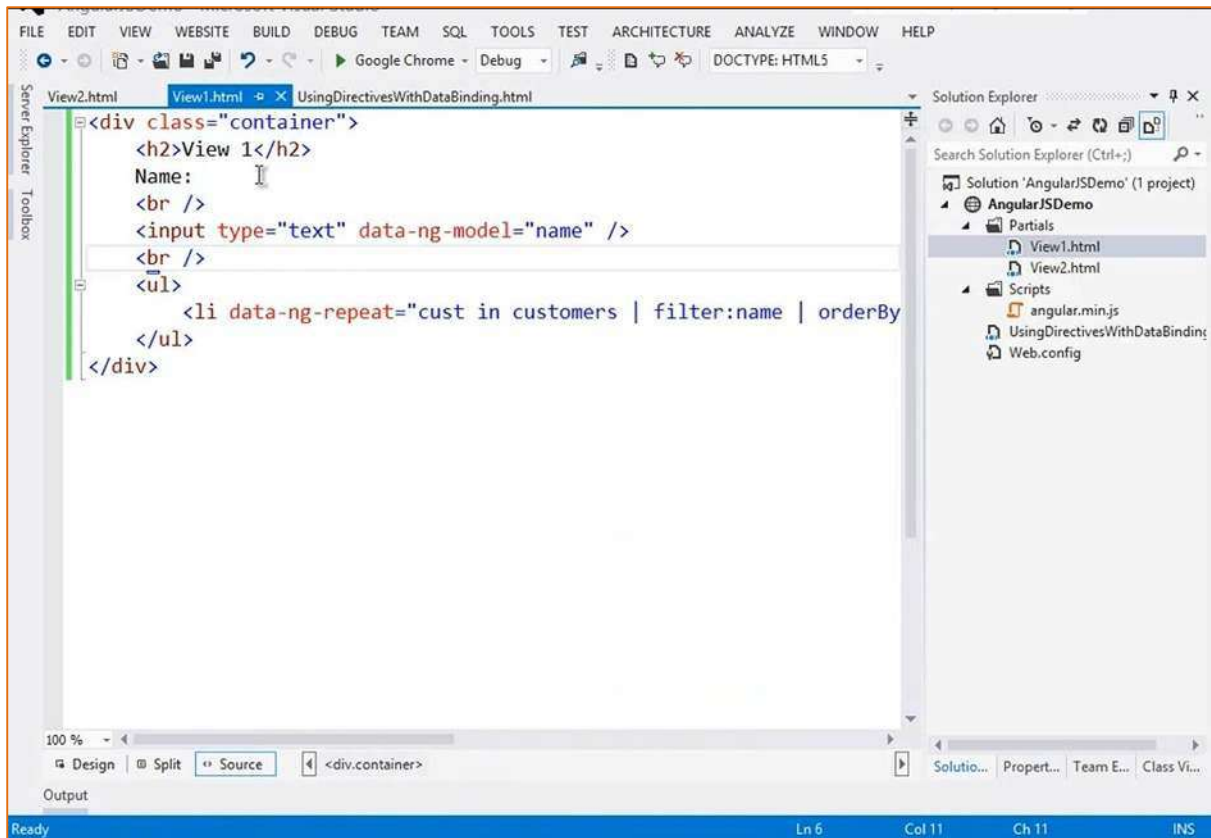
Let's go ahead and on our module let's add in our **config**. So we can say demoApp.config and then in here we can give it a new route or multiple routes.

I'm just going to paste in some code for this to save a little typing:

```
demoApp.config(function ($routeProvider) {
  $routeProvider
    .when('/',
      {
        controller: 'SimpleController',
        templateUrl: 'Partials/View1.html'
      })
    .when('/partial2',
      {
        controller: 'SimpleController',
        templateUrl: 'Partials/View2.html'
      })
    .otherwise({ redirectTo: '/' });
});
```

So we have demoApp.config and we have our routeProvider.

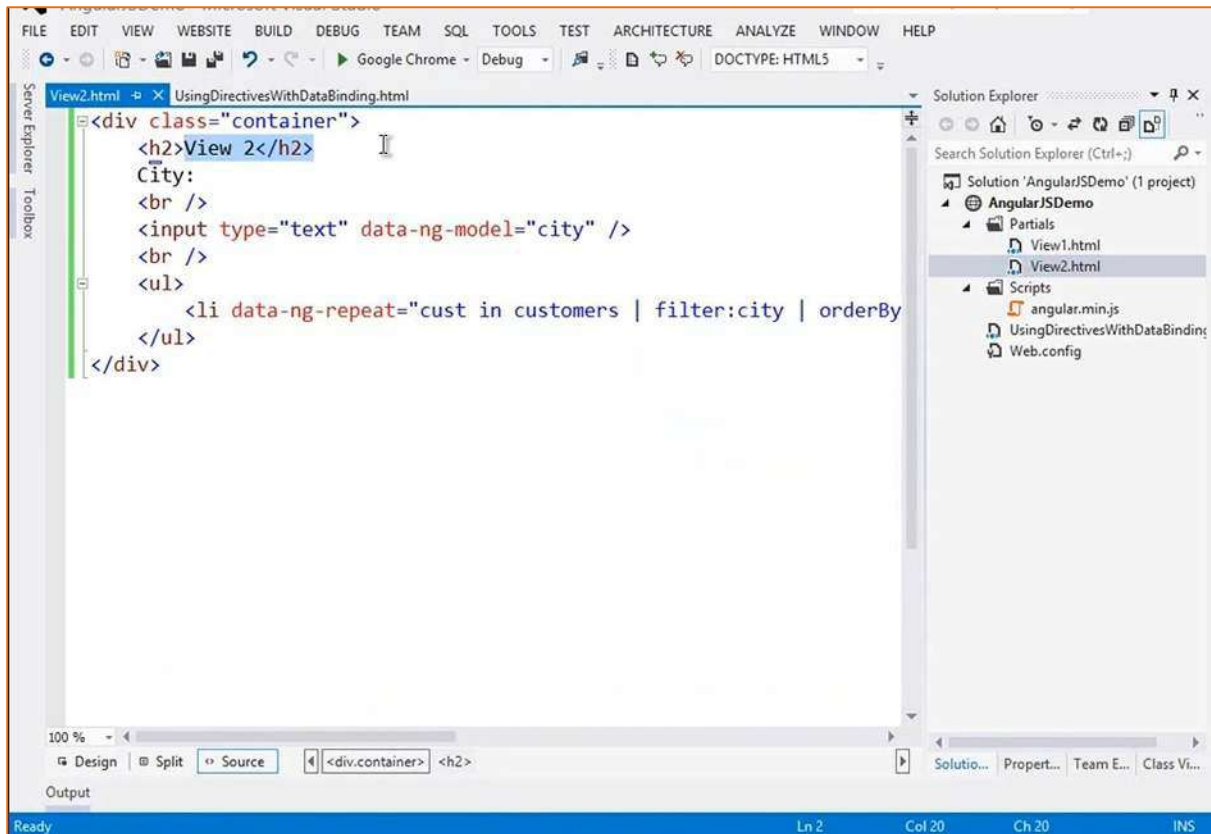
The routeProvider says "When we're at the root for our route then use SimpleController and go to Partials/View1.html.



00:45:45

You'll see it's very, very simple.

We have our same filtering and our same looping that we did earlier, so really nothing different there.



00:45:58

We also have View2.

We're just going to pretend we filter by city instead. So nothing really fancy there, just enough to make it a little bit different.

They're both going to use the same controller in this case, but in a real life app that may or may not be the case. Certainly I offer that most of the time I have kind of a one-to-one between controllers but it depends on how much re-use you can get out of one.

In this case we're just going to use SimpleController which has our customers:

```
demoApp.controller('SimpleController', function ($scope) {
    $scope.customers = [
        { name: 'John Smith', city: 'Phoenix' },
        { name: 'John Doe', city: 'New York' },
        { name: 'Jane Doe', city: 'San Francisco' }
    ];
});
```

Now what I want to do though is just come into View1 and fix it up just a little bit.

I'm going to make it so that under each customer we can enter a customer name .

```
<div class="container">
  <h2>View 1</h2>
  Name:
  <br />
  <input type="text" data-ng-model="name" />
  <br />
  <ul>
    <li data-ng-repeat="cust in customers | filter:name | orderBy
  </ul>

  <br />
  Customer Name:
  <input type="text" data-ng-model="name" />
</div>
```

I also want to come in and make it so that you can enter a customer city.

```
Customer Name:<br />
<input type="text" data-ng-model="name" />
<br />
Customer City:<br />
<input type="text" data-ng-model="name" />
```

We already have "name" – that's our filter. Normally I like to do filter.name which keeps it really clear:

```
<div class="container">
  <h2>View 1</h2>
  Name:
  <br />
  <input type="text" data-ng-model="filter.name" />
  <br />
  <ul>
    <li data-ng-repeat="cust in customers | filter:filter.name" />
  </ul>
```

Then I change the customer name to say "customer.name" or just to make it really obvious let's say newCustomer.name.

```
Customer Name:<br />
<input type="text" data-ng-model="newCustomer.name" />
<br />
```

What that will do is create a new property on the scope, which then has a sub-property called "name".

We'll do the same for "city".

```
Customer City:<br />
<input type="text" data-ng-model="newCustomer.city" />
```


Then the final thing we need is a button. We haven't seen how to interact with our controller yet so let's go ahead and take a look.

I'm going to add just a standard good old button and another directive we can use is called **ng-click**.

If you go to the documentation for Angular there are several different options here. **ng-click** is just one of them. I'm going to call **addCustomer()**.

```
Customer City:<br />
<input type="text" data-ng-model="newCustomer.city" />
<br />
<button data-ng-click="addCustomer()">Add Customer</button>
```

Once this view loads, once we get that working, we should be able to add a customer in. Now obviously in our controller we need to be able to handle that.

So let's go back to our controller and because we called this addCustomer and because the view binds to the scope then we need to say "\$scope.addCustomer" and assign that to a function.

```
$scope.addCustomer = function () {
};
```

You'll notice that I'm not having to pass in the data. I don't even have to look at the textboxes because the scope already has that. We can get to everything through this so what I'm going to do is say "\$scope.customers.push()" and let's push a new item into the array.

So we'll say "name: \$scope.newCustomer.name"

```
App.controller('SimpleController', function ($scope) {
  $scope.customers = [
    { name: 'John Smith', city: 'Phoenix' },
    { name: 'John Doe', city: 'New York' },
    { name: 'Jane Doe', city: 'San Francisco' }
  ];

  $scope.addCustomer = function () {
    $scope.customers.push({ name: $scope.newCustomer.name
```

Then we'll add a comma and we'll do "city: \$scope.newCustomer.city", and then end our object literal there.

```
    = function () {
      $scope.customers.push(
        { name: $scope.newCustomer.name, city: $scope.newCustomer.city });
    };
  };
});
```

Then we're up and running. That's it. That's all we have to do to make this work and it makes it really easy to now interact back from the view into the controller, but yet this [the view] doesn't even

know about the controller. You'll notice there's no controller definition here – that's going to happen through the route:

```
<div class="container">
  <h2>View 1</h2>
  Name:
  <br />
  <input type="text" data-ng-model="filter.name" />
  <br />
  <ul>
    <li data-ng-repeat="cust in customers | filter:filter.name |
  </ul>

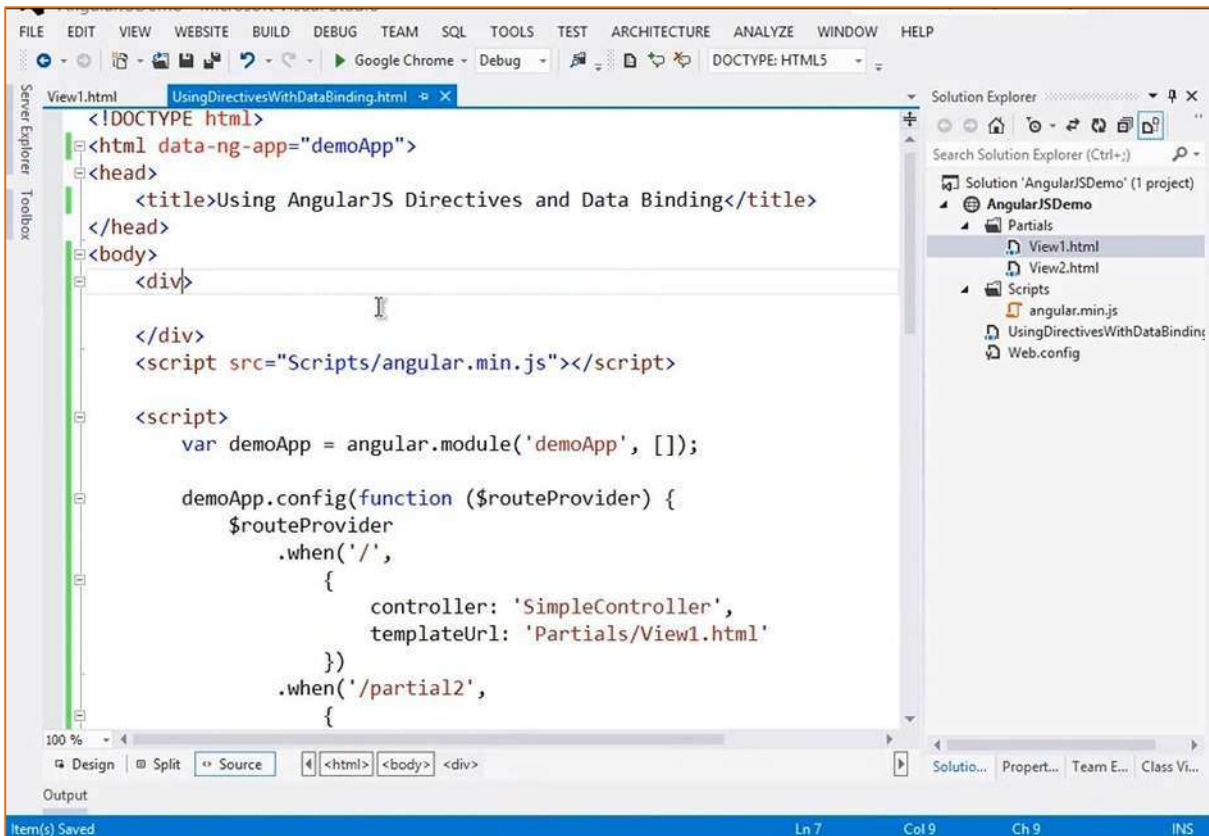
  <br />
  Customer Name:<br />
  <input type="text" data-ng-model="newCustomer.name" />
  <br />
  Customer City:<br />
  <input type="text" data-ng-model="newCustomer.city" />
  <br />
  <button data-ng-click="addCustomer()">Add Customer</button>
</div>
```

That will call our “addCustomer” and it will simply show up in our list – our s that we have.

That's one thing I wanted to show, and now we have the routes we're kind of ready to go, but we have a little bit more work.

00:49:37

What I'm going to do is this little “UsingDirectivesWithDataBinding” page here, I'm going to pretty much kill off most of this.



I don't need the controller anymore because the views will use those.

What I'm going to do is add a special directive called **ng-view**.

There's a couple of ways you can do this. You might see it like this, with ng-view as a tag:

```
<div>
  <ng-view></ng-view>
</div>
```

I feel more comfortable using <div> and then I can use my **data-**

```
<div>
  <div data-ng-view></div>
</div>
```

Normally you just do it like that, but I'm going to go ahead and add an = just because most people are used to that with the HTML5 **data-** attributes.

```
<div>
  <div data-ng-view=""></div>
</div>
```

This represents the placeholder for the views so now when my routes kick in and we go to a default route what will happen is the Partials/View1.html is going to be married to the controller. That's then going to be injected dynamically into the <div>

```
<div>
  <!-- Placeholder for views -->
  <div data-ng-view=""></div>
</div>
<script src="Scripts/angular.min.js"></script>

<script>
  var demoApp = angular.module('demoApp', []);

  demoApp.config(function ($routeProvider) {
    $routeProvider
      .when('/',
        {
          controller: 'SimpleController',
          templateUrl: 'Partials/View1.html'
        }
      )
  });
```

I don't have to write the DOM code to do that. It's just going to happen behind the scenes and now we're starting to get into a SPA-type of concept here.

If we go up [to the start of the code] just as a recap:

- Ng-app has "demoApp"
- We now have our ng-View which represents our placeholder
- We have our module with our config and our routes and, assuming I named everything OK we should be alright here.

Let's say this [route configuration] is going to be view2 and we'll reference this in just a second.

```
demoApp.config(function ($routeProvider) {
  $routeProvider
    .when('/',
      {
        controller: 'SimpleController',
        templateUrl: 'Partials/View1.html'
      }
    )
    .when('/view2',
      {
        controller: 'SimpleController',
        templateUrl: 'Partials/View2.html'
      }
    )
    .otherwise({ redirectTo: '/' });
});
```

- Then we have our controller where we can bind customers

```
demoApp.controller('SimpleController', function ($scope) {
  $scope.customers = [
    { name: 'John Smith', city: 'Phoenix' },
    { name: 'John Doe', city: 'New York' },
    { name: 'Jane Doe', city: 'San Francisco' }
  ];
});
```

- and then we find a way to add to it [customers]

```
$scope.addCustomer = function () {
  $scope.customers.push(
    {
      name: $scope.newCustomer.name,
      city: $scope.newCustomer.city
    }
  );
};
```

What I'm going to do real quick here is I realise I didn't add a link here [in View1]. Let's say at the bottom we have a hyperlink and let's say when this is clicked we want to go to that route and we'll say "View 2" on this.

```
<button data-ng-click="addCustomer()">Add Customer</button>
<br />
<a href="#/view2">View 2</a>
```

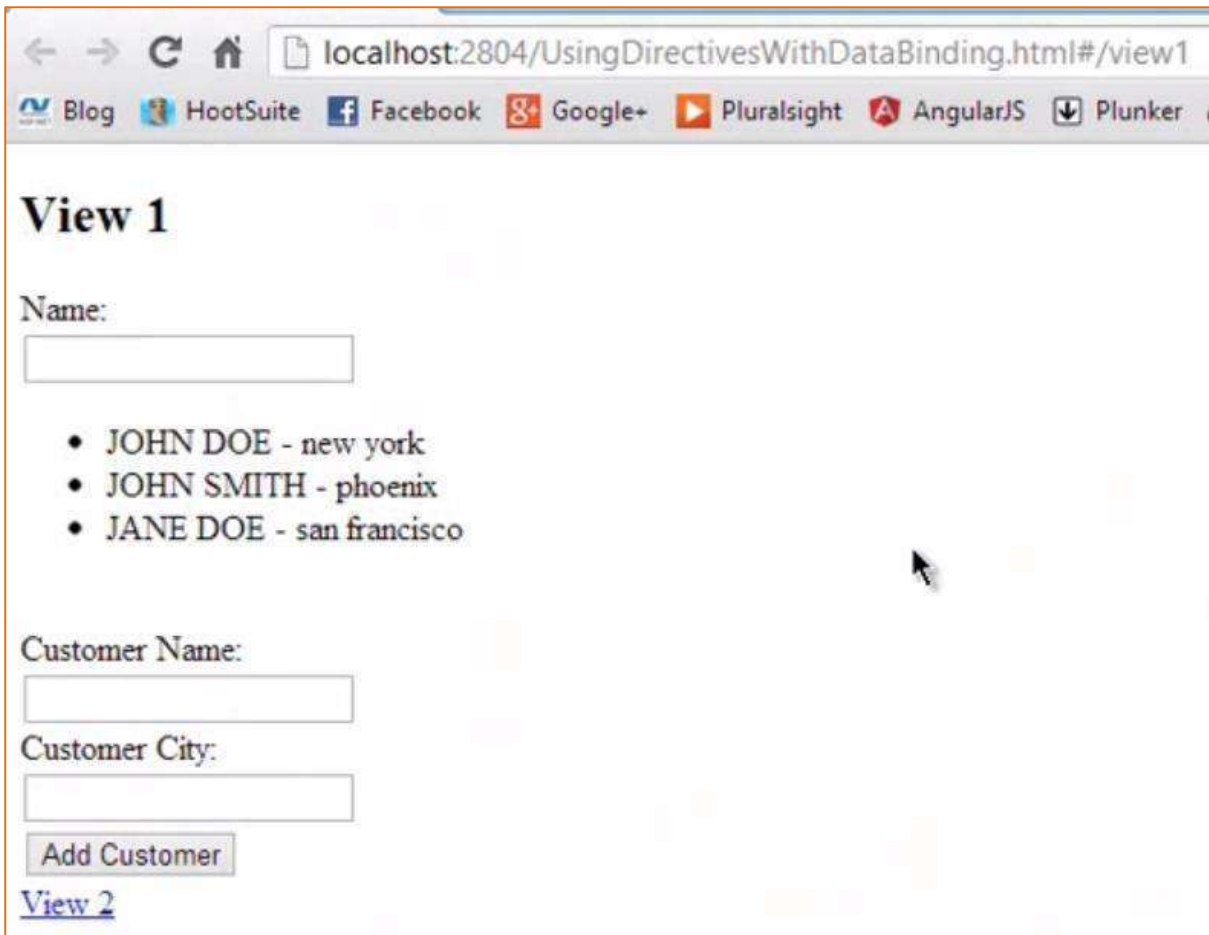
So when you click it this will direct Angular to kick in the router and then that router will take us over to View2 so it will be pretty standard.

00:51:48

OK, so now we've really cleaned this guy up. The last thing that I would do which I'll show you in a more realistic app is I would take all this [stuff between script tags] out and make it separate scripts and load that, but we'll go ahead and leave it here for demo's sake.

Now that View1 is here notice that I don't have to put ng-controller because it's going to map that up automatically for both views.

So let's give it a shot and we'll see if there are any errors or typo's here.



00:52:15

You can see that View1 has now loaded here.

I actually tweaked the route just a little bit.

```
demoApp.config(function ($routeProvider) {
  $routeProvider
    .when('/view1',
      {
        controller: 'SimpleController',
        templateUrl: 'Partials/View1.html'
      })
    .when('/view2',
      {
        controller: 'SimpleController',
        templateUrl: 'Partials/View2.html'
      })
    .otherwise({ redirectTo: '/view1' });
});
```

I had "/" which I could do, but I also wanted to show that instead of having just a slash there for the route I'm going to redirect to view1 or if you prefer you could just do this [change both "/view1"s back to "/"s], and that would be the default view. Either way would work.

Now here's what's really cool. Not only does this manage our routing for us, and as I load it you're going to see it still works and you can see our context is bound, I get our customers, I can still filter and do all that. I can come in and let's add a customer...

Customer Name:

Customer City:

[View 2](#)

... and watch our list...

- DAN - chandler
- JOHN DOE - new york
- JOHN SMITH - phoenix
- JANE DOE - san francisco

... notice it sorted it.

Now when I click on "View2" watch the path up here [in the browser bar].



What it's going to do is navigate to View 2 where I can still filter. These both have the same controller and therefore the same scope.

View 2

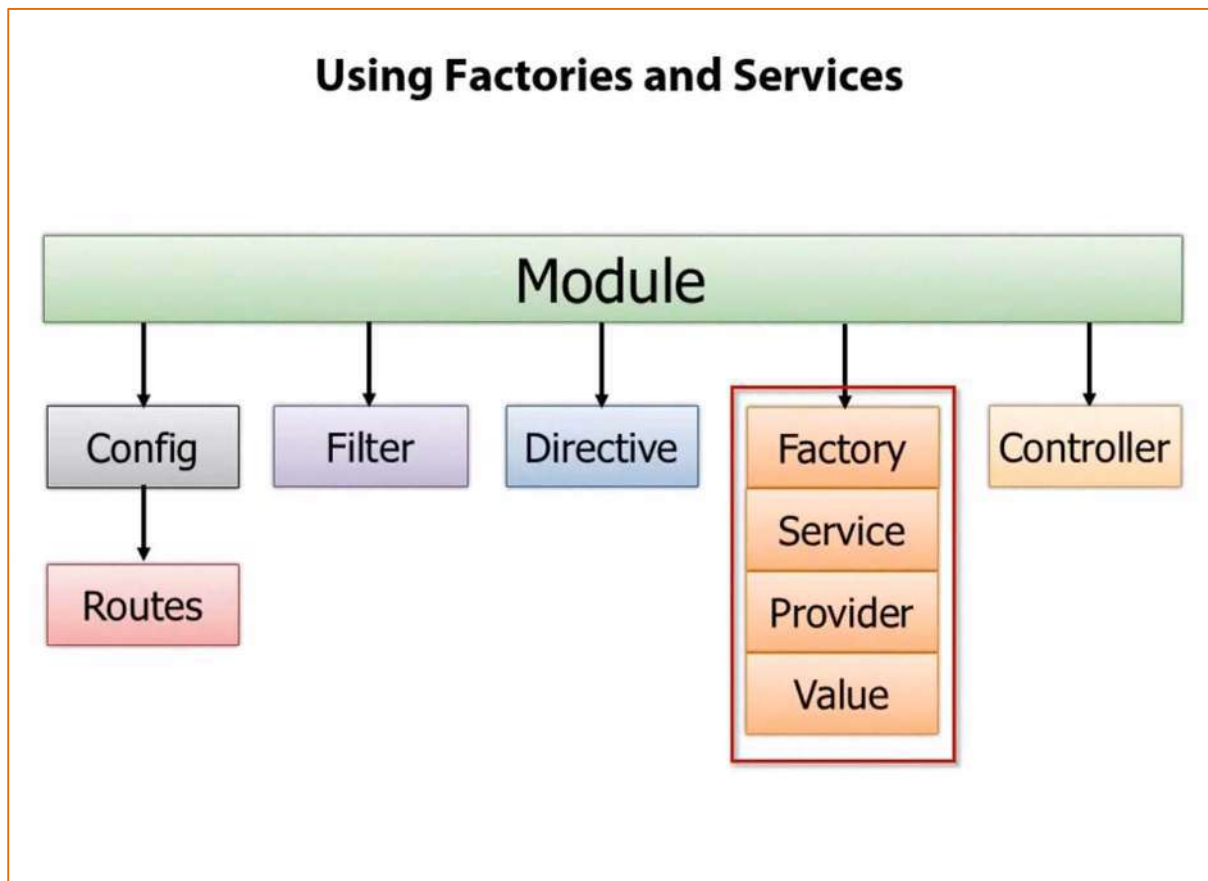
City:

- JOHN DOE - new york
- JOHN SMITH - phoenix
- JANE DOE - san francisco

But when I hit Back [in the browser menu] it's actually going to go back to View1. I can go back and forward and Angular is automatically handling that history for me which is a really, really nice feature to have available.

So that's an example of we can actually come in, define a module, define some routes, and then go in and on those routes hook up a controller to a view and now Angular will actually wire up those routes for us.

Using Factories and Services



00:53:56

Now that you've seen how to create modules and how modules can then be used to define routes and also to work with controllers, we're going to wrap up by talking about some re-use concepts.

Another feature of AngularJS is the ability to encapsulate data functionality into factory, services, provider or little value providers.

I'm going to focus on factories here but all three of the top ones shown here – factory, service and providers – they allow us to encapsulate common functionality.

So for instance if I had to go and get customers and I need those customers in multiple controllers I wouldn't want to hard code that data in each controller. It just wouldn't make sense and there'd be a lot of duplication there.

Instead what I'll do is I'll move that code out to a factory, service or provider.

The difference between the three is just the way in which they create the object that goes and gets the data. That's really all there is to it.

- With the **factory** you actually create an **object** inside of the factory and return it.
- With the **service** you just have a standard function that uses the **this** keyword to define function.
- With the **provider** there's a **\$get** you define and it can be used to get the object that returns the data.

A **value** is just a way to get for instance a config value. A simple example of this you'll see on the Angular site is you might just want the version of a particular script. So you'd have a name-value pair where the name of the value might be "version" and then the value might be say "1.4"

I'm not going to cover all those here but I am going to cover factories. So let's take a look at how we can use a module to define a factory.

The Role of the Factory

The Role of Factories

```
var demoApp = angular.module('demoApp', [])
  .factory('simpleFactory', function () {
    var factory = {};
    var customers = [ ... ];
    factory.getCustomers = function () {
      return customers;
    };
    return factory;
  })
  .controller('SimpleController', function ($scope,
    simpleFactory) {
    $scope.customers = simpleFactory.getCustomers();
  });
```

Factory injected into
controller at runtime

00:55:25

In this example you'll see that down below I have this controller which I looked at earlier.

Notice that instead of hard coding the customers in here, or if it was an AJAX call instead of coding that call into the controller, I'm going to use a module up here to define a factory.

In this case you'll see I'm actually using **chaining**.

So the module's defined and then instead of putting a semi-colon we chain factory and then we chain controller. You don't have to do that, but that is certainly an option.

In this factory we're going to give it a name, and what the factory's going to do in this case is find a way to get customers. Let's assume we have a customers variable up here. What we want to do is create a factory object, define a method on it that returns this customers variable.

In a real life app this getCustomers might go out and make an async call. Then that async data returns and then returns it to the controller or controllers that needs it.

What's really cool about factories, services and providers is that once you've defined it you can then inject it very easily as a parameter into something like a controller or even another factory if you want. A factory could rely on another factory.

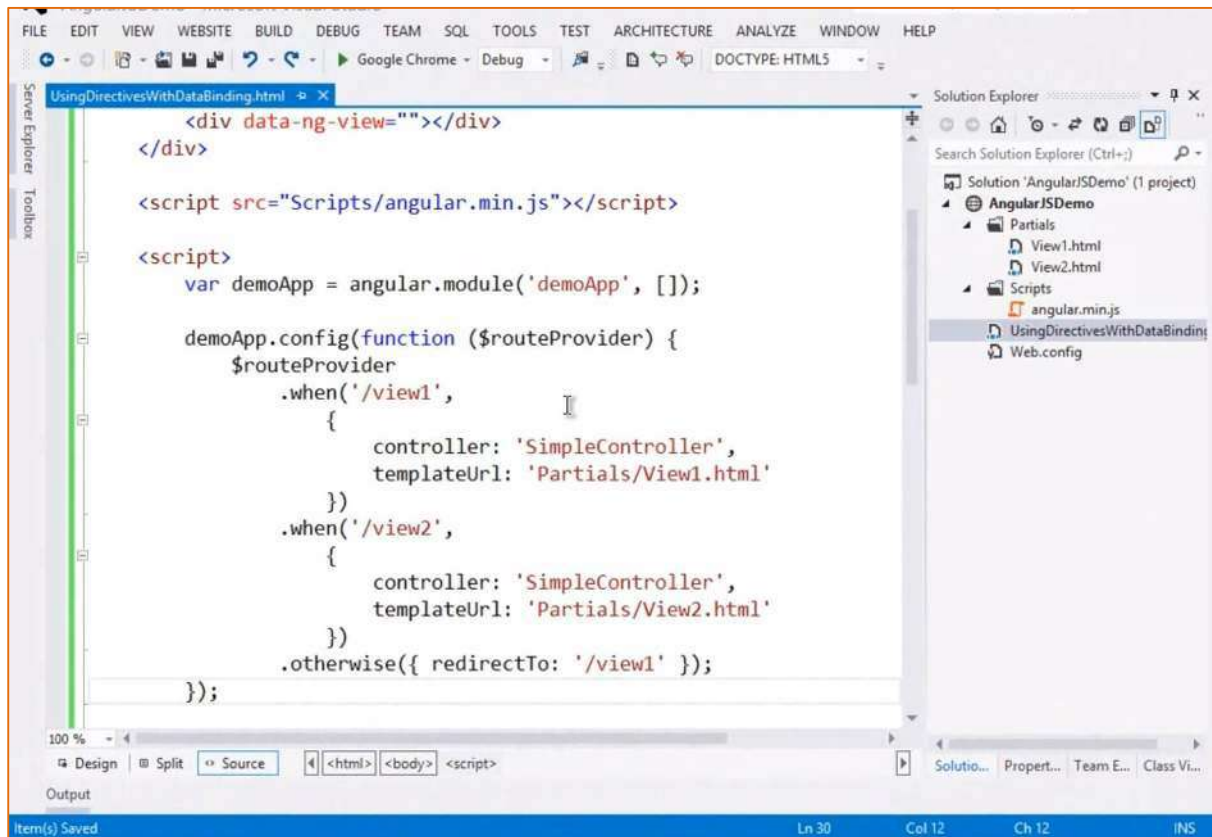
If we come down to our controller, you'll see that SimpleController does our standard \$scope but notice that the second parameter is simpleFactory. Well that little name there matches up on purpose.

What AngularJS is going to do at run time is dynamically inject this into the controller so now I'll have access to this factory object that's returned out of this function and now I can say **factory.getCustomers()**

This provides dependency injection and it provides a way in which I know I can centrally place where I get customer data or order data or whatever it may be. In a real-life app you certainly may have several different factories in the application itself.

Let's jump into a demo and I'm going to convert the customers we had in our controller before and move those out into just a really simple factory to help get you started.

Factory Demo



00:57:35

Earlier I showed how we can define a module, define our routes and then have a controller in that module called SimpleController, but you'll notice that the customers are hard-coded:

```
demoApp.controller('SimpleController', function ($scope) {
  $scope.customers = [
    { name: 'John Smith', city: 'Phoenix' },
    { name: 'John Doe', city: 'New York' },
    { name: 'Jane Doe', city: 'San Francisco' }
  ];
});
```

In a real-life app you may have something hard-coded I suppose but in general we're probably going to go off to a server or service to go and get the data via AJAX or some similar kind of technique maybe even websockets.

So what I'm going to do is move these customers out.

We don't want to hard-code these obviously and we can pretend that we're going to go and call a service. I'm just going to go ahead and say that right now customers is just an empty array.

```
$scope.customers = [];
```

Now what I'm going to do is under the routes I'm going to define a factory. You'll notice that as I type factory it shows up in the nice Intellisense. If I type ser it also shows service, or I could do a provider.

So there's three different options for doing this and, again, they all differ just in how they create and return the object that serves up the data, but factory is really easy to understand and get started with. I'm going to give it an empty function here but keep in mind with Angular as eg \$scope is dynamically injected, we can also inject other things.

```
demoApp.factory('simpleFactory', function() {
```

```
}
```

If I wanted to make an AJAX call I can tell Angular "inject in the Angular http object".

```
demoApp.factory('simpleFactory', function ($http) {
```

```
});
```

And then I could do http get/put/post and delete type of calls to for instance RESP APIs.

In this case I'm not going to do that – we're going to keep it really fundamental and simple. I'm going to come in and define some customers and I'm just going to define those customers we had earlier.

```
demoApp.factory('simpleFactory', function () {  
    var customers = [  
        { name: 'John Smith', city: 'Phoenix' },  
        { name: 'John Doe', city: 'New York' },  
        { name: 'Jane Doe', city: 'San Francisco' }  
    ]  
};
```

```
});
```

With a factory you create an object, tack on some functions to it and then return that object out of the function. With a service you don't create an object. The function is the object. You just tack on using the "this" keyword some function. I'll show you the difference here.

We're going to come in and say factory = and just create an empty object here.

```
var factory = {};
```

Now what I'm going to do is factory. And let's call it "getCustomers".

```
factory.getCustomers() {
```

```
}
```

getCustomers is simply going to come in and just return customers.

```
factory.getCustomers = function() {  
    return customers;  
}
```

Had I passed in an http object, or if you're using jQuery or whatever it may be I can make the AJAX call right here and then once it comes back we could return a promise and get into all that fun stuff with async calls.

But in this case it's going to return something really simple so now our factory has a getCustomers(). We might also have factory.putCustomer if we wanted to or postCustomer or whatever you want to call it and maybe this would take our actual customer object and then we would have code in here to actually do something with it.

```
factory.postCustomer = function (customer) {  
};
```

In this case we'll just stick with customers. Regardless of whatever you put on with the factory, once you create the factory object simply return it.

```
demoApp.factory('simpleFactory', function () {  
    var customers = [  
        { name: 'John Smith', city: 'Phoenix' },  
        { name: 'John Doe', city: 'New York' },  
        { name: 'Jane Doe', city: 'San Francisco' }  
    ];  
  
    var factory = {};  
    factory.getCustomers = function () {  
        return customers;  
    };  
    factory.postCustomer = function (customer) {  
  
    };  
  
    return factory;  
});
```

If this were to say service right here instead of .factory then this [the function] becomes the factory. So I would just say this.getCustomers and this.postCustomers and then the factory itself would be in the function.

```
this.getCustomers = function () {  
    return customers;  
};  
this.postCustomer = function (customer) {  
};
```

I like factories because you control the object yourself and you don't have to use the "this" keyword and things.

01:01:10

So we're ready to go here. We now have a factory which returns some hard-coded customers. Now we need to fill them. How do we do that?

Well I need first to get a reference up here to the factory. Now all you do is you take the name and I'll just put comma and then you put that name in.

```
demoApp.controller('SimpleController', function ($scope, simp
```

What Angular will do is go look up that factory automatically and inject it in for us. That's all you have to do – put the name, so it's very modular.

Here I could say `simpleFactory.getCustomers()`:

```
demoApp.controller('SimpleController', function ($scope, simp
    $scope.customers = simpleFactory.getCustomers();
```

But let's say we have a whole initialisation routine we want to do. I usually do something like this. I'll just make kind of a private function here, we'll call it `init` and I'll say `$scope.customers = that`:

```
$scope.customers = [];
function init() {
    $scope.customers = simpleFactory.getCustomers();
}
```

And then all we have to do is call `init()`:

```
demoApp.controller('SimpleController', function ($scope, simp
    $scope.customers = [];

    init();

    function init() {
        $scope.customers = simpleFactory.getCustomers();
    }
```

You don't have to do it that way at all. I just prefer all my initialisations for all my data routines or factory calls to be in one nice little place so I might have multiple lines in here that kind of kick off the process to maybe get the initial maybe look-up data for instance for drop-downs that my controller passes down to my view. That's why I like to do it this way.

OK now other than this there's really nothing in the view that's going to change because the view doesn't even know about any of this. It just knows about ultimately the controller and the scope and the way we did our views it doesn't even know about the controller because it's dynamically assigned:

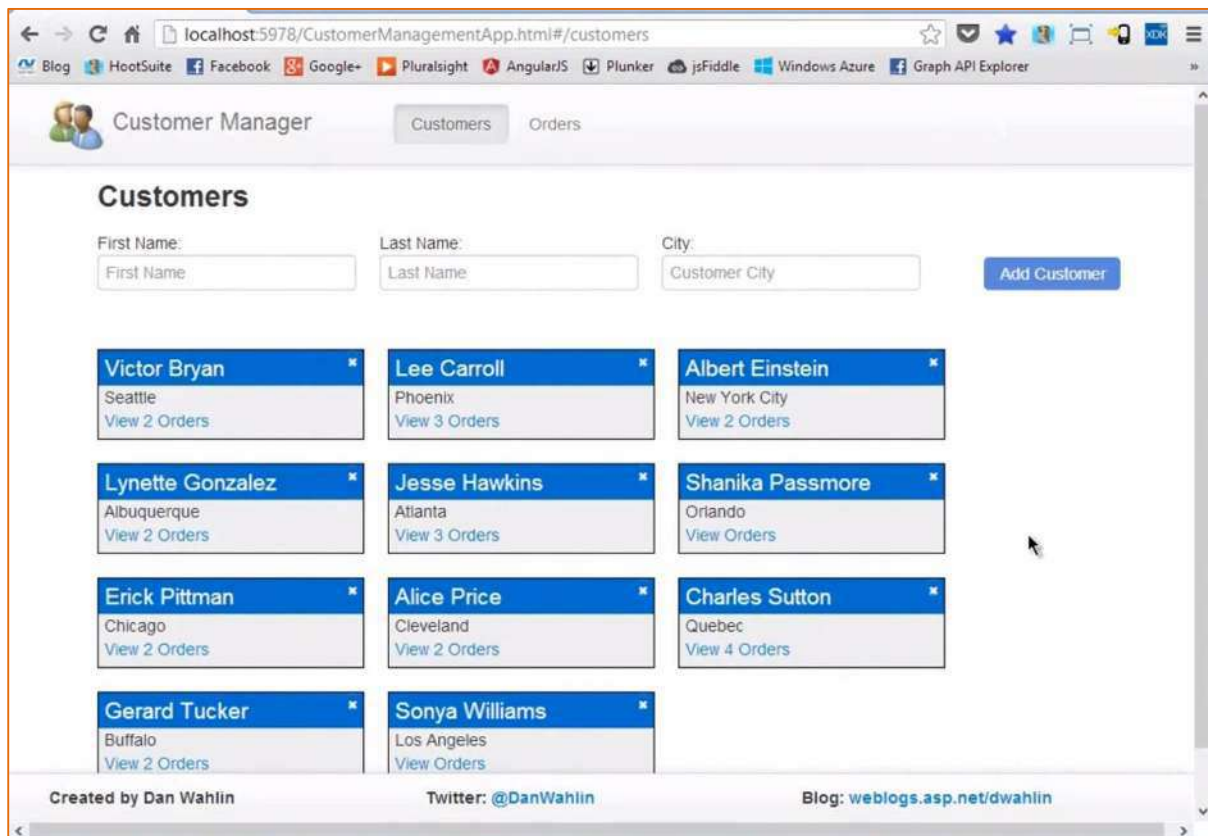
```
demoApp.config(function ($routeProvider) {
  $routeProvider
    .when('/view1',
      {
        controller: 'SimpleController',
        templateUrl: 'Partials/View1.html'
      }
    )
})
```

Let's go ahead and run this and we should see that our customers actually load up. They're now coming from the factory though. If we go to View2 these are also coming from the factory.

Now we're not having to duplicate data, assuming we did have two controllers here. We have them all in one nice little, reusable factory.

That is an example of getting started with factories in AngularJS.

Wrap-Up Demo: Pulling It All Together



01:03:15

Now that you've seen the core components involved in AngularJS I'm going to wrap up with just a real quick demo that I put together. This was put together over just a few hours that demonstrates some of these features in action.

Let me first pull up the application, and this is really just a little SPA app that shows some different customers so I can come in and add in customers if I'd like. You can see it alphabetises them and all of that. I go here [x on one of the customer headers], and I can remove it back out.

I can click on "orders" so we'll do that for Lee Carroll and that'll take me into a separate page – a separate view actually, but it didn't reload the whole shell page...

localhost:5978/CustomerManagementApp.html#/customerorders/1

Customer Manager Customers Orders

Customer Orders

Orders for Lee Carroll
 1234 Anywhere St.
 Phoenix

Product	Quantity	Unit Price	Total
Basket	1	\$29.99	\$29.99
Needles	1	\$5.99	\$5.99
Yarn	1	\$9.99	\$39.96
			\$75.94

Created by Dan Wahlin Twitter: @DanWahlin Blog: weblogs.asp.net/dwahlin

I can go back to Customers and go see all the customer orders...

localhost:5978/CustomerManagementApp.html#/orders

Customer Manager Customers Orders

Needles	1	\$5.99	\$5.99
Yarn	1	\$9.99	\$39.96
			\$75.94

Albert Einstein

Product	Quantity	Unit Price	Total
Baseball	5	\$9.99	\$49.95
Bat	1	\$19.99	\$19.99
			\$69.94

Lynette Gonzalez

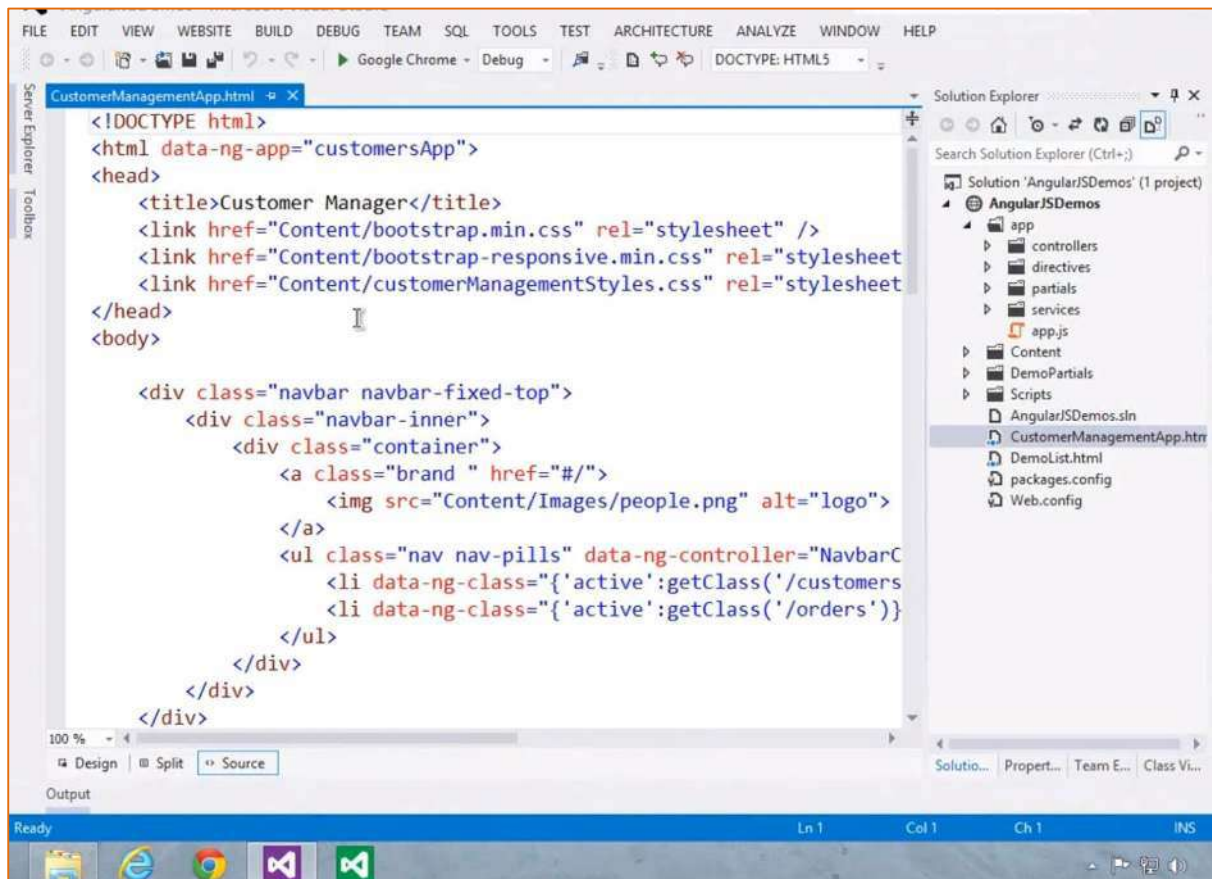
Product	Quantity	Unit Price	Total
Picture	1	\$1,029.99	\$1,029.99
Statue	1	\$429.99	\$429.99
			\$1,459.98

Jesse Hawkins

Created by Dan Wahlin Twitter: @DanWahlin Blog: weblogs.asp.net/dwahlin

This is just using a couple of things. First of all we have AngularJS of course. It's also using a little bit of jQuery because I'm doing Bootstrap: so Bootstrap, jQuery and Angular. Now I'm not using jQuery for much of really anything – just a few basic things so that we can work with some of the Bootstrap stuff I'm doing.

In this particular app this is the shell page.



You can see that we have some bootstrap up the top as well as my styles.

We have my navigation.

And here is the view:



Now I did a little custom directive. This is an animated-view that slides things in. Starting with version – and I put a comment in about it – 1.1.4 of Angular the ability to animate your views as they get put in will automatically come out of the box. You just have to supply some CSS styles and that'll kick in, but this also shows how to do a custom directive so if you're interested I show how to write a custom directive there.

All I did was take what AngularJS already had for the ng-view and just tweaked it to use the jQuery animation features.

From there, moving on down you'll notice that I just have some scripts.

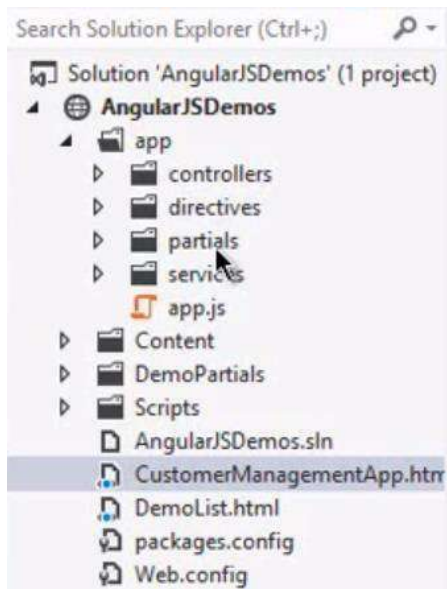
```
<!-- Vendor Libs: jQuery only used for Bootstrap functionality -->
<script src="Scripts/angular.js"></script>
<script src="Scripts/jquery.min.js"></script>

<!-- UI Libs -->
<script src="Scripts/bootstrap.js"></script>

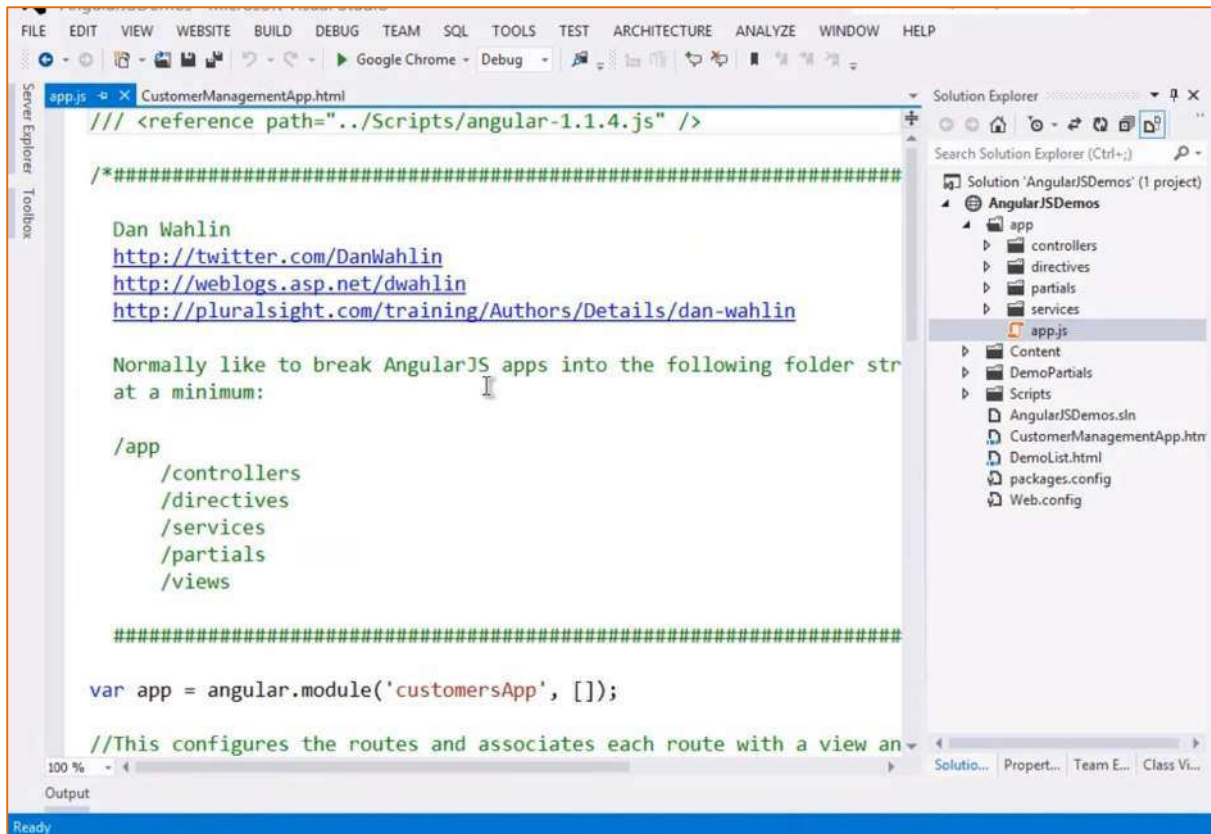
<!-- App libs -->
<script src="app/app.js"></script>
<script src="app/controllers/controllers.js"></script>
<script src="app/services/customersService.js"></script>
<script src="app/directives/animatedView.js"></script>
</body>
</html>
```

The only core what I'd call library scripts are Angular and jQuery. We then have Bootstrap for our UI scripts and then here's my custom. I put all the controllers just in one file to keep it really simple here but you can certainly break these out.

You can see that everything is in an **app** folder, so if we go up to App you'll see controllers, directives, partials, services and then **app.js** you're going to see is what kind of kicks everything off.



Let's start there.



1:05:38

app.js is where we come in and define our module.

```
var app = angular.module('customersApp', []);
```

You can see I have **customersApp** and it doesn't have any dependencies.

Then I have my different routes for my customers, my customer orders and my orders screen.

```

var app = angular.module('customersApp', []);

//This configures the routes and associates each route with a view an
app.config(function ($routeProvider) {
    $routeProvider
        .when('/customers',
            {
                controller: 'CustomersController',
                templateUrl: '/app/partials/customers.html'
            })
        //Define a route that has a route parameter in it (:customerID)
        .when('/customerorders/:customerID',
            {
                controller: 'CustomerOrdersController',
                templateUrl: '/app/partials/customerOrders.html'
            })
        //Define a route that has a route parameter in it (:customerID)
        .when('/orders',
            {
                controller: 'OrdersController',
                templateUrl: '/app/partials/orders.html'
            })
    });

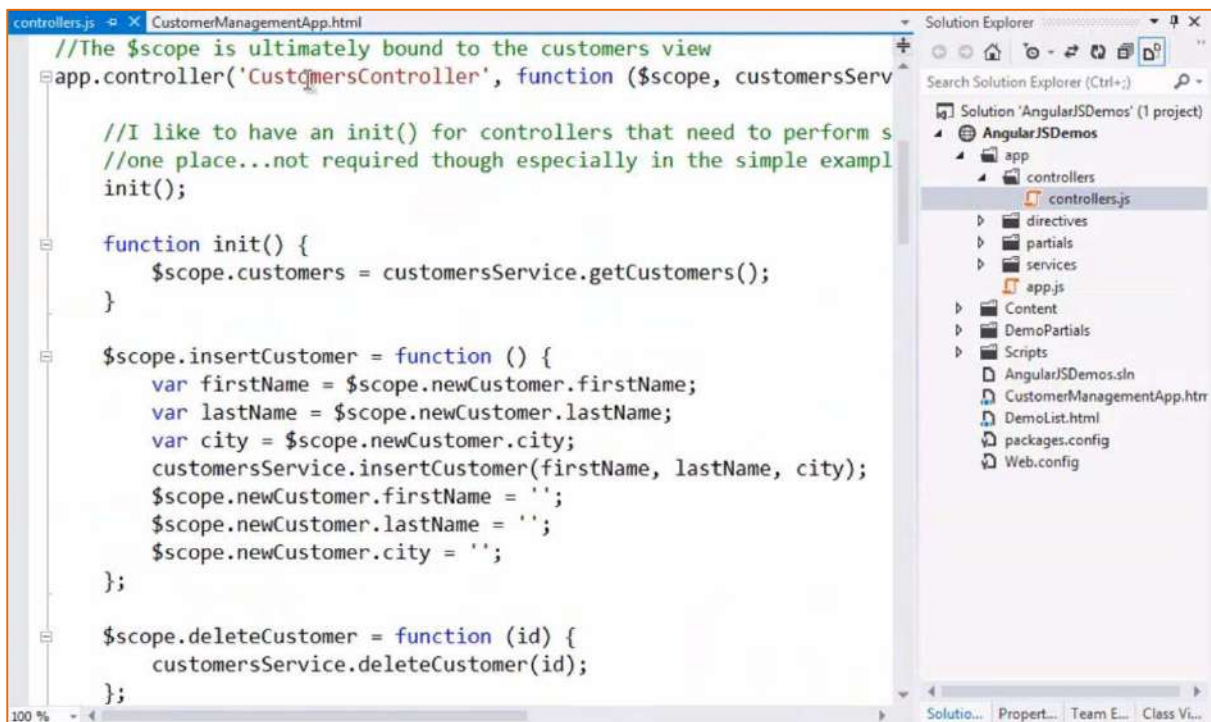
```

You can even see I'm demonstrating how to pass parameters up here on the URL using a route parameter.

```
//Define a route that has a route parameter in it (:customerID)
.when('/customerorders/:customerID',
```

So that's in here as well if you're interested. That's kind of the default configuration that you can see and then there's a little bit of details here [in the opening comment section] about how I like to break things up in general.

If we come over to the Controllers folder this is where I have my controllers.



You can see I have CustomersController. We have CustomersOrdersController:

```
//This controller retrieves data from the customersService and associ
//The $scope is bound to the order view
app.controller('CustomerOrdersController', function ($scope, $routePa
    $scope.customer = {};
    $scope.ordersTotal = 0.00;

    //I like to have an init() for controllers that need to perform s
    //one place...not required though especially in the simple exampl
    init();
```

And an OrdersController...

```

//This controller retrieves data from the customersService and associ
//The $scope is bound to the orders view
app.controller('OrdersController', function ($scope, customersService)
    $scope.customers = [];

    //I like to have an init() for controllers that need to perform s
    //one place...not required though especially in the simple exampl
    init();

    function init() {
        $scope.customers = customersService.getCustomers();
    }
});

```

Then I have a really simple one for NavbarController.

```

app.controller('NavbarController', function ($scope, $location) {
    $scope.getClass = function (path) {
        if ($location.path().substr(0, path.length) == path) {
            return true
        } else {
            return false;
        }
    }
});

```

These are all built off the module, so they're part of it. They inject in a few things that the app needs here, including a factory and so this is all pretty standard stuff that you've already seen. The factory, or in this case the service – I wanted to show both ways of doing it, so in the demo's I showed a factory. Here's an example of a service, really the same thing as I mentioned earlier: you use the this keyword to find your different functions here.

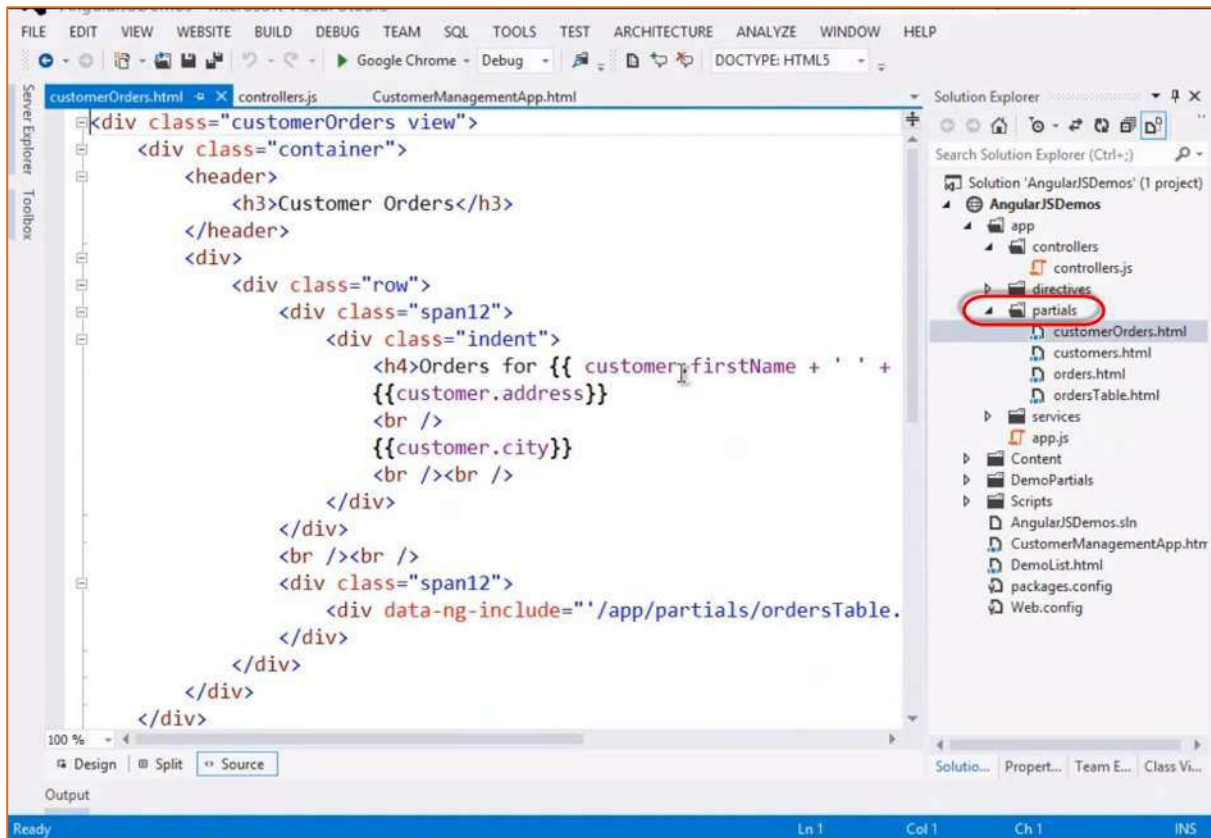
```

app.service('customersService', function () {
    this.getCustomers = function () {
        return customers;
    };
});

```

In this case the data's hard coded so it's not making an AJAX call, but this would give you the foundation for building a service or you could do a factory. Either one. I actually tend to prefer factories for some reason but this is one where I decided to try services out.

My views are in a little partials folder.



This is just going to be the little simple parts that get injected in to the animated views. These are my kind of SPA sub-pages if you will that get injected in and we bind them to scope. You can see some data binding here.

There's quite a bit more that I could walk through here, but this is something I wanted to show real quick just to show a simple example of getting started with AngularJS and how you can use it to actually interact, do data binding, do factories, do controllers and all that fun type of stuff.

Summary

Summary

- **AngularJS provides a robust "SPA" framework for building robust client-centric applications**
- **Key features:**
 - Directives and filters
 - Two-way data binding
 - Views, Controllers, Scope
 - Modules and Routes

01:07:50

Well that is the end.

I hope that by watching this video you have a really good solid feel for the different pieces of Angular and the power that it really does offer. I'm really excited about the framework. We're building some different apps using it and liking the overall process, flow and modularity it provides.

There's kind of a summary here.

It really is a SPA framework but it depends on what you're doing. If you just want it for data binding with the injection you could do that if you wanted as well.

Some of the key features as a review...

We talked about directives and filters. So directives enhance our HTML. We talked about things like ng-repeat, ng-app, ng-controller, ng-init and there's many others. We talked about how we can filter data and also how we can do two-way data binding.

From there I showed you how we can hook up views and controllers, and that really the glue there so far as the data binding goes is the scope, and the scope is really another word for a viewmodel, that some of you might be familiar with.

Then finally we wrapped up with modules and routes, and I also threw in factories at the end there to show how we can first off define a container. That container can then have one or more routes in

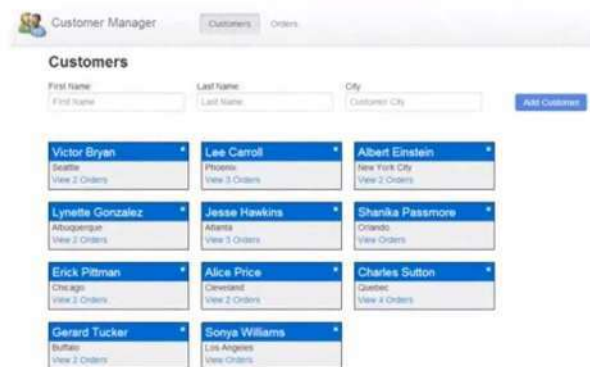
it, one or more controllers, different factories and then we can link that app or module into our main shell view using the ng-app directive.

So that's a wrap on the content I had planned in the 60-ish minutes since I know I'm over that.

Download the Sample Code

Download the sample code:

<http://tinyurl.com/AngularJSDemos>



01:09:20

Now as far as other resources out there that I recommend you take a look at, there are definitely a lot of cool things out there.

One of those will be the sample code. This'll just have some of the sample code I ran through, as well as the app I showed at the end. You can go to <http://tinyurl.com/AngularJSDemos>

Resources

Resources

- <http://angularjs.org>
- <http://builtwith.angularjs.org>
- <http://angular-ui.github.io>
- <http://mgcrea.github.io/angular-strap>
- <http://pluralsight.com>



01:09:40

Aside from that a lot of great stuff out there that I don't have time to go into.

We did visit <http://angularjs.org> but <http://builtwith.angularjs.org> has a lot of samples of Angular in action.

Then if you want to get some different plug-ins whether it's Bootstrap or jQuery there's some sites out there with some directives that'll make them really easy to integrate.

As far as learning, I do a lot of work with a company called Pluralsight. It's an awesome company and very cool people run it, and this is online video training. Now at the time I filmed this, or else I would have listed them, there's not any Angular courses, but they're being worked on as we speak.

My good friends Jim Cooper and Joe Bohemes are working on an AngularJS Fundamentals course that will be up there shortly, and then I'm currently working with Scott Allen, developer extraordinaire Scott Allen, on an end-to-end Angular course that'll show you the whole process with a real app and how that all works.

Feel free to check that out if you're interested.

Dan Wahlin



Blog

<http://weblogs.asp.net/dwahlin>



Twitter

@DanWahlin

01:10:35

In the meantime feel free to check out some of my posts at <http://weblogs.asp.net/dwahlin> or get in touch via Twitter.

I appreciate your time in watching the video.