Intro to Angular

tl;dr

- Angular is an opinionated client-side platform ...
- Which will help you create web applications faster and more maintainable than without it ...
- By writing components.
- It now uses semantic versioning - MAJOR.MINOR.PATCH
- Angular uses a ton of libraries that would be very tough to set up manually

Semantic Versioning

Angular uses semantic versioning (SEMVER)

New major releases every 6 months in March/April and September/October.

Minor releases every month

Patch releases every non-holiday week

2 . 3 . 1

Major *Breaking change*

Minor *New features, not breaking*

Patch *Bugfixes, not breaking*

History

- Angular 1 - October 2010
- Angular 2 - Gradual, painful releases from April 2015 to September 2016
- Angular 3 - Never existed - Skipped due to @angular/router being called v3 when everything else was v2. So they skipped to "4" to realign everything. It's all 4.
- Angular 4 - March 2017
- Angular 5 - October 23, 2017?

Changing from version 2 to version 4, 5, … won't be like changing from Angular 1. It won't be a complete rewrite, it will simply be a change in some core libraries that demand a major SEMVER version change.

So how do we talk about Angular then?

- Three simple guidelines:

1. Say "Angular" for versions 2 and later
   - "I'm an Angular developer"
   - "This is an Angular course"
2. Say "AngularJS" for versions 1.x
3. Avoid the version number except when referring to a specific release
   - "The flush() method was introduced in Angular 4.2."

What they are saying about Angular vs AngularJS

- 2 = easier mental model? Some say so. Not me!
- Bindings are simpler
- services vs. factories vs. providers vs. controllers vs. configs
- TypeScript? Nope. Not an advantage.
- Faster - True
- Mobile - No better or worse.

So what really are the advantages?

1. Components!
2. Faster to run

The web is going to components

- The web component spec is coming*
- We'll all be writing components in a few years

But for now...

- React
- Polymer
- Angular

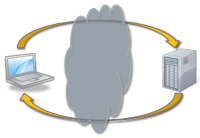
* Find more information on http://www.webcomponents.org/

With Angular, you'll no longer write pages; you'll write components

Each component is self-contained and encapsulated

- Even styles are local; CSS no longer cascades through components

Angular is an opinionated client-side platform



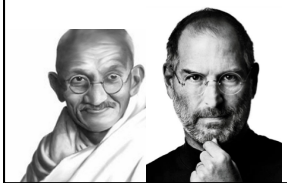
- Let's break this down on the next pages

Let's define "opinionated"

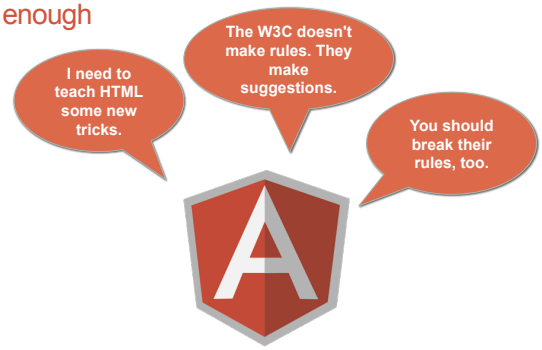
What does opinionated mean to you?

Do you like opinionated people?

Is there anything good about being opinionated?

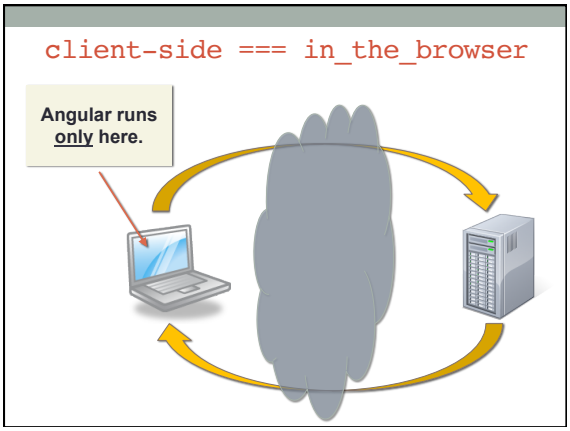


Angular is of the opinion that HTML isn't good enough



With Angular, you're locked in to doing it Angular's way

`client-side === in_the_browser`



A platform is the structure of a project

Platform

- Architecture
- A way of thinking about work
- Structure
- ie.
  - EmberJS
  - JsViews
  - Ractive

Libraries

- Tools
- Smaller
- Lower-level
- Task-oriented
- ie.
  - KnockoutJS
  - jQuery
  - React.js

Infrastructure

The seemingly endless list of mysterious tools you (thankfully) don't have to memorize

Angular prerequisites (a partial list)

- Solves the JavaScript 'problem' of execution context.
- Allows you to execute things asynchronously and save a context. (It creates a zone in which your async code can run).

- Handles asynchronous updating of the DOM.
- Basically, it runs the new $digest cycle.

SystemJS

- A module loader.
- A polyfill until the ES2015 Modules are implemented in all browsers.
- All the import and export statements are handled by System.JS

rxjs

- Adds Observables and dozens of tools regarding them

Angular demands that you have a very precise, very complex setup with literally hundreds of interdependent libraries having dozens of version numbers each.

If one small part of it is not configured properly, the system fails.

Wow! all these parts. Wouldn't it be nice to have a way to automate the creation/ scaffolding of the different parts?

Next chapter!

tl;dr

- Angular is an opinionated client-side platform ...
- Which will help you create web applications faster and more maintainable than without it ...
- By writing components.
- It now uses semantic versioning - MAJOR.MINOR.PATCH
- Angular uses a ton of libraries that would be very tough to set up manually

The Angular Command Line Interface

Back to the future

tl;dr

- The CLI tool removes the need for super developer powers
- It installs through npm and is called 'ng'
- Use it to scaffold an entire project
- Then scaffold components, pipes, services, etc
- It uses WebPack to compile and run a project in watch mode via ng serve.
- When it is time to deploy to a server, use ng build

The powers that be decided that Angular was waaay too hard to handle manually. So they created a tool to help us manage it.

The Angular Command Line Interface or...

angular-cli

To install

```
$ ng help
bash: /usr/local/bin/ng: No such file or directory
$ sudo npm install --global @angular/cli
/usr/local/bin/ng -> /usr/local/lib/node_modules/@angular/cli/bin/ng

> fsevents@1.1.1 install /usr/local/lib/node_modules/@angular/cli/node_modules/fsevents
> node install

[fsevents] Success: "/usr/local/lib/node_modules/@angular/cli/node_modules/fsevents/lib/
```

```
        is-fullwidth-code-point@1.0.0
      yargs-parser@4.2.1
    webpack-merge@2.6.1
    zone.js@0.8.11
```

```
$ ng help
Unable to find "@angular/cli" in devDependencies.

Please take the following steps to avoid issues:
"npm install --save-dev @angular/cli@latest"

ng build <options...>
  Builds your app and places it into the output path (dist/ by default).
  aliases: b
```

Scaffolding things in the CLI

```
$ ng new coolApp
installing ng2
  create .editorconfig
  create README.md
  create src/app/app.component.css
  create src/app/app.component.html
  create src/app/app.component.spec.ts
  create src/app/app.component.ts
  create src/app/app.module.ts
  create src/app/index.ts
  create src/assets/.gitkeep
  create src/environments/environment.prod.ts
  create src/environments/environment.ts
  create src/favicon.ico
  create src/index.html
  create src/main.ts
  create src/polyfills.ts
  create src/styles.css
  create src/test.ts
  create src/tsconfig.json
  create src/typings.d.ts
  create angular-cli.json
  create e2e/app.e2e-spec.ts
  create e2e/app.po.ts
  create e2e/tsconfig.json
  create .gitignore
  create karma.conf.js
  create package.json
  create protractor.conf.js
  create tslint.json
Installing packages for tooling via npm.
Installed packages for tooling via npm.
$ ls
coolApp
$ cd coolApp
$ ls
README.md              karma.conf.js          protractor.conf.js
angular-cli.json       node_modules           src
e2e                    package.json           tslint.json
$
```

Create a new NG Application

`ng new <appName>`

- Takes a long time

You can scaffold components

- ng generate component People
- Will be relative to src/app unless you specify a folder
- Creates a subfolder unless you use --flat
- Normalizes the name to kebab-case.

```
$ ng generate component People
installing component
  create src/app/people/people.component.css
  create src/app/people/people.component.html
  create src/app/people/people.component.spec.ts
  create src/app/people/people.component.ts
$
```

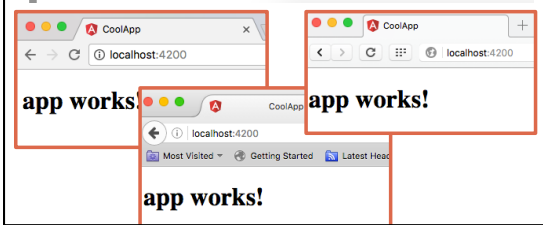
... and other things as well

```
ng generate class [name]
ng generate component [name]
ng generate directive [name]
ng generate enum [name]
ng generate guard [name]
ng generate interface [name]
ng generate module [name]
ng generate pipe [name]
ng generate service [name]
```

Developing with the CLI

You can stand up a server with *ng serve*

```
$ ng serve
** NG Live Development Server is running on http://localhost:4200.
Hash: 386fbce10750c48ef658
chunk    {0} main.bundle.js, main.bundle.map (main) 4.82 kB {2} [in
chunk    {1} styles.bundle.js, styles.bundle.map (styles) 9.99 kB {
chunk    {2} vendor.bundle.js, vendor.bundle.map (vendor) 2.22 MB [
chunk    {3} inline.bundle.js, inline.bundle.map (inline) 0 bytes [
webpack: bundle is now VALID.
```



Under the covers, *ng serve* ...

- Runs webpack
- Reads angular-cli.json to find the bootstrapping files
  - index.html, main.ts, app.component.ts,etc.
- Compiles everything and bundles everything **in memory**
  - So don't go looking for the files that webpack/Node/Express serve. You won't find them.
  - To make them, run ng build --target=development
- Serves index.html via Node/Express
- Publishes a websocket so that it can push a page when we update.

To go to production

- But wait! If the CLI tool doesn't write my files to the disk, what do I do when I want to go to production?
- I have to serve something!
- Answer: ng build
- This will compile it all into a folder called "dist"
- You can point your webserver to get all its files from dist.
- You'll definitely need this if you're getting Ajax data b/c ng serve can only serve Angular 2 pages. No API data.

ng-serve watches your code for changes

- It recompiles and restarts the server each time you save a file.
- You can make build do that also with

```
ng build --watch
```

tl;dr

- The CLI tool removes the need for super developer powers
- It installs through npm and is called 'ng'
- Use it to scaffold an entire project
- Then scaffold components, pipes, services, etc
- It uses WebPack to compile and run a project in watch mode via ng serve.
- When it is time to deploy to a server, use ng build

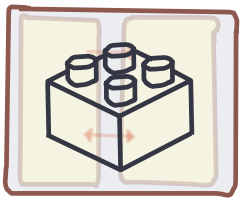
The Big Picture

tl;dr

- Angular apps are grouped in modules with components being the main building block.
- Components have a class and a template who communicate through bindings:
  1. Interpolation
  2. Property binding
  3. Event binding
  4. Two-way binding
- Directives modify behavior of components
- Services allow sharing between components
- The bootstrapping process is complex. It goes from index.html to system.js to main.ts to app.module.ts to app.component.ts
- Angular uses SPAs by default
- You swap out components to simulate page navigation

The Angular world revolves around ...

Components



```
<my-component something="someValue"></my-component>
```

Pop quiz!!

What is a DOM attribute?

A modifier to a DOM element

What is a DOM property?

A member of a DOM object

So what is the difference between them?

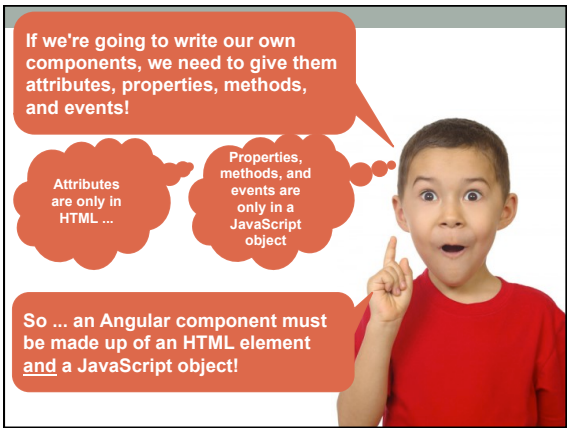
Ummm ...

Properties != Attributes

Attributes	Properties
• Only in the HTML • Always a string • Always set a property	• Only in the JavaScript • Can be any data type • Almost never set an attribute

```
<img src="knob.jpg" />
```

```
img = {
  alt:""
  attributes:NamedNodeMap
  clientHeight:100
  clientWidth:101
  currentSrc: "knob.jpg"
  ...
  onclick: null,
  ...
  setAttribute: function () { },
  src: "knob.jpg",
  ...
}
```



From the official NG documentation ...

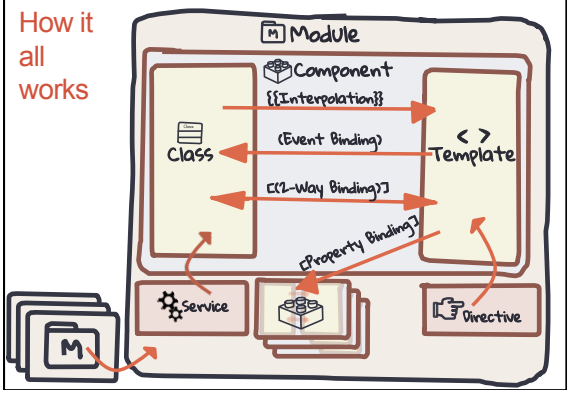
You write Angular applications by composing HTML templates with Angularized markup, writing component classes to manage those templates, adding application logic in services, and boxing components and services in modules.

Said differently,

1. You write HTML components,
   defining how they should look (with html)
   and behave (with JavaScript).
2. Make those components up of smaller components.
3. Put shared behavior in services.
4. And group them all in modules.

Shall I draw you a picture?

How it all works



Module

A container which ...

Creates separation between different parts of the app

Everything is in a module!

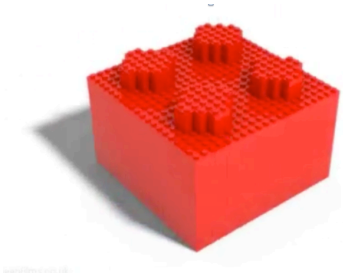
Components

- The basic building block of an Angular app
- All the visible presentation and most of the behavior is in components

We use composition to build our apps

Simple components join to become complex ones

Components can be reused (but most will not be)



Components use JavaScript classes to encapsulate ...

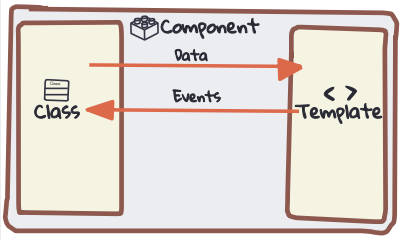
1. Identity
2. Behavior

So what's missing?

3. Presentation

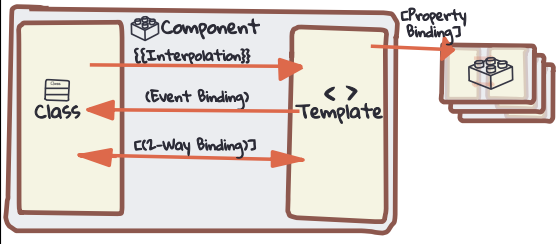
Components use templates for presentation

- Simply HTML and CSS
- (Remember that each template may contain other components)



Bindings are the things that make sites dynamic!

- This is like, the major reason we have Angular.
- There are four types of bindings ...

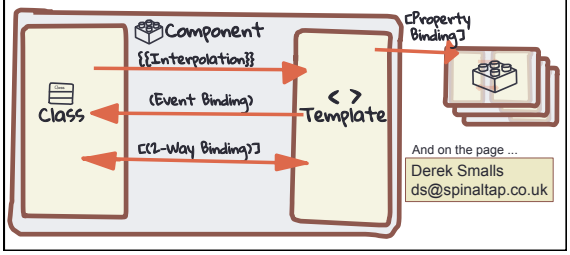


Interpolation is one-way data binding

- Passing business logic data to the template for display on the page

```
this.person =
  getPerson(1234);
```

```
<p>{{person.first}} {{person.last}}</p>
<p>{{person.email}}</p>
```

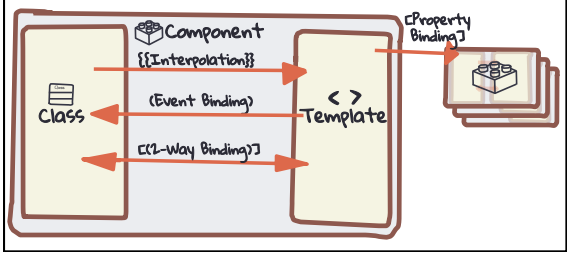


Property binding

- Passing business logic data to components via attributes

```
this.nigel =
  getPerson(732);
```

```
<band-member [member]="nigel">
</band-member>
```

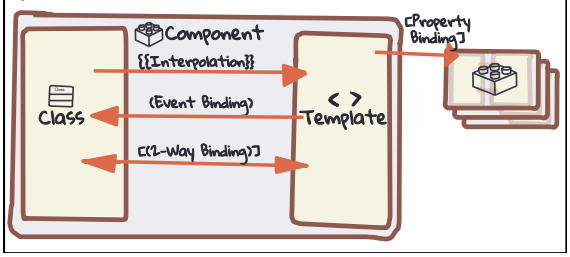


Event binding

- Wiring up a template event to a business logic method

```
zoom(member) {
  // Do stuff here
}
```

```
<band-member
(click)="zoom(member)">
</band-member>
```

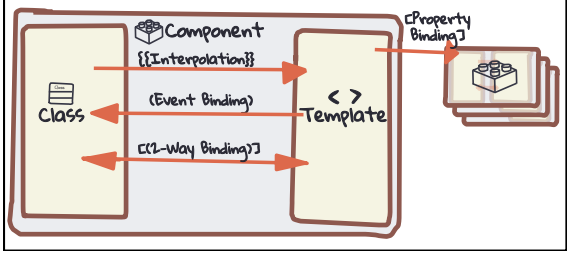


Two-way data binding

- Only makes sense for form fields

```
this.roadie=
  getPerson(1234);
```

```
<input [(ngModel)]="roadie.first" />
```

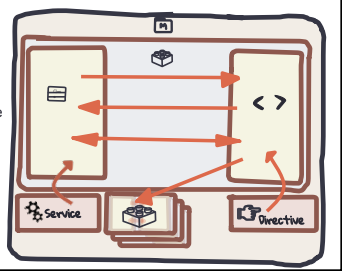


Directives

- Components are merely directives with a template.
- Directives should be template-agnostic

3 types

1. Structural directives
   Change the DOM itself, adding or removing whole branches
2. Attribute directives
   Change behavior but not the DOM
3. Custom directives
   Ones you and I write



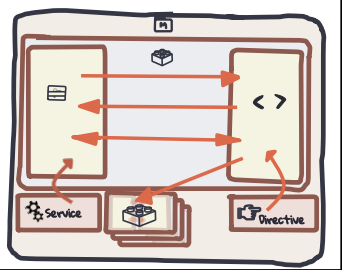
Services aren't what you think at first

Anything to be shared between components goes in a service

Services are "*holders of wonderful things*"

- Data
- Functionality

- If it goes into two or more components, it should be in a service.



How Angular runs

How does an Angular app start?

index.html

```
<script src="system.js">
</script>
<script>
System.import("app/main");
</script>
```

main.ts

```
import { AppModule } from './app.module';
platformBrowserDynamic().bootstrapModule(AppModule);
```

app.module.ts

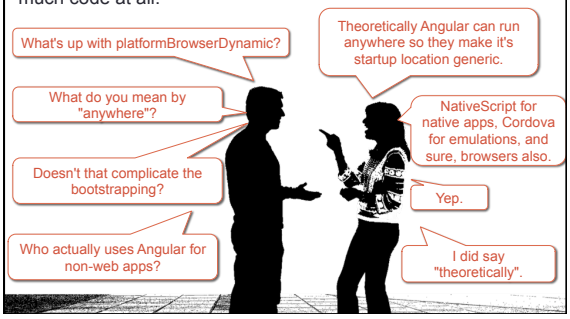
```
import { AppComponent }
  from './app.component';
@NgModule({
 bootstrap: [AppComponent]
})
export class AppModule { }
```

app.component.ts

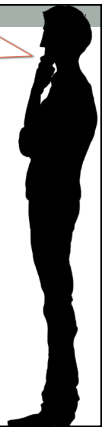
```
@Component({
template: `<p>Hello world</p>`
})
export class AppComponent {}
```

Okay, but why the platformBrowser thing?

It allows us to boot it to whatever environment without changing much code at all.



Wait. If my content is in "components" rather than HTML pages, how do people browse to my other pages?

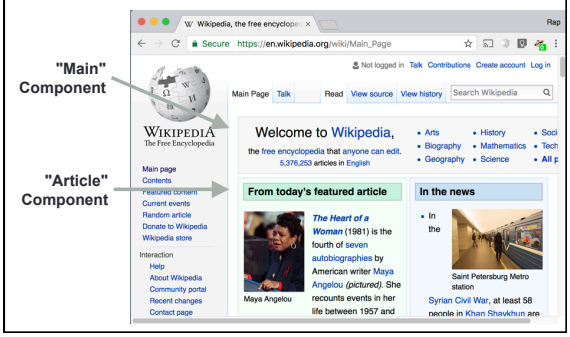


They don't! Angular uses SPAs*

- With AngularJS, SPAs were optional.
- With Angular, you default to SPAs.
- The only full-blown HTML file you hit is index.html.
- You *can* write other HTML pages, but each would then be considered their very own Angular app and would have to be re-bootstrapped.

*Single Page Applications

So to change a 'page' you're really only swapping out the main building block.



tl;dr

- Angular apps are grouped in modules with components being the main building block.
- Components have a class and a template who communicate through bindings:
  1. Interpolation
  2. Property binding
  3. Event binding
  4. Two-way binding
- Directives modify behavior of components
- Services allow sharing between components
- The bootstrapping process is complex. It goes from index.html to system.js to main.ts to app.module.ts to app.component.ts
- Angular uses SPAs by default
- You swap out components to simulate page navigation

TypeScript

An introduction to it and the new features of JavaScript

tl;dr

- Angular effectively requires us to use TypeScript (TS)
- Your code is transpiled from proprietary TS into ES5 by tsc
- TS allows us to use modern ES features like modules, classes, and string templates -- even in super-old browsers!
- And it adds proprietary OO features like strong type checking, decorators, and constructor shorthands.
- But it causes friction in setup, development, and debugging

Technology continues to improve but ...

JavaScript language additions

Newer features you'll need for Angular

The best features of ES2015

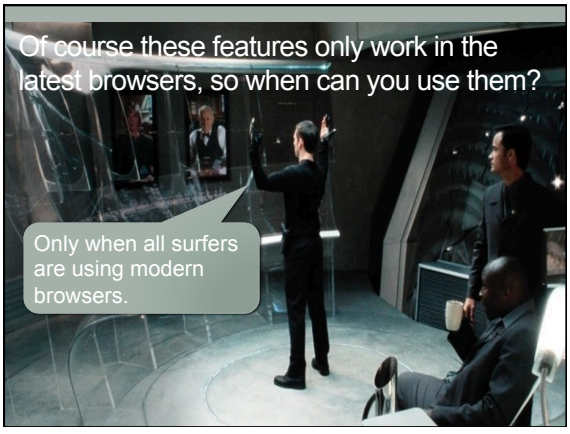
	Edge	Firefox	Chrome	Safari	Opera
Arrow functions	13	43	45	10	35
Block scoping (let, const)	13	44	45	10	35
Default parameters	14	45	49	10	36
Classes	13	45	49	9	36
Promises	12	38	49	9	36
New collections (Map, Set)	12	38	49	9	36
New iterators (for-of)	12	38	49	9	36
String templates (`${}`)	12	38	49	9	36
Destructuring	14	38	49	9	36
Modules (export, import)	38	54	60	10.1	47
Spread & rest	13	38	49	10	39

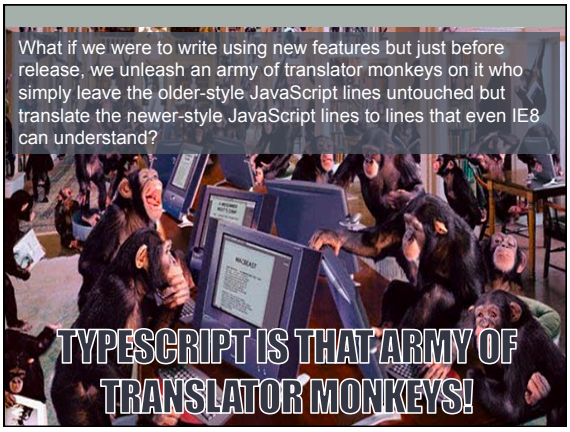
The best features of ES2016

	Edge	Firefox	Chrome	Safari	Opera
Exponentiation Operator	14+	52+	54+	10.1+	43+
Array.prototype.includes	14+	40+	54+	10+	43+

The best features of ES2017

	Edge	Firefox	Chrome	Safari	Opera
Object static methods	15	51	56	10.1	43
String padding	15	51	57	10	44
Trailing commas in function	14	52	58	10	45
Async functions	15	52	56	10.1	43
Atomics	15	52	60	10.1	47





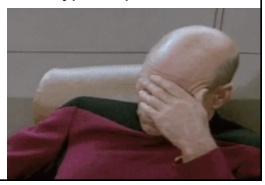
Transpiling is totally optional and there are other transpilers (Babel, Traceur, es6-shim, et. al).

So ... why TypeScript for Angular?

- It is more concise than raw JavaScript.
- TypeScript adds some cool features that ES20XX doesn't (yet)

But mainly ...

- Because everyone is writing Angular in TypeScript
  - examples
  - sample code
  - tutorials
  - blog posts



Why TypeScript is awesome

- You can use future JS features today! (especially modules)
- It adds a few cool features (like decorators)
- It allows better static code analysis
- It is a superset of ES so you can mix, say, ES5 with ES2018 in the same program.

Why TypeScript stinks

- You debug in the browser but the JavaScript you're debugging isn't the code you wrote; it's the code tsc wrote.
- You debug code in zone.js, core.umc.js, etc. But I'm not sure if all that is due to TypeScript or due to Angular itself.
- You have to memorize TypeScript in addition to JavaScript (yes, I know it's a superset but if another dev uses the features you have to know them to understand his/her code).
- Few IDEs recognize TypeScript. You're "stuck" with VS Code.
- You can get errors when compiling TypeScript to JS and then more when you run the JS.
- One error results in more showing up.
- In addition to your source.ts file, there's also a source.js and source.js.map file to get in your way.
- Much friction in installation (Must have a typings file)
- Friction in setup (ie. Must populate the typings.json file, wildcards work with tsc in unix but not in Windows, etc.)
- Friction in running ()
- Doesn't honor my prototypes. Tried to extend Date through prototyping. Added a prototype method at the top of the file and called it at the bottom. Tsc refused to compile b/c it doesn't recognize my call as being on Date.
- Some errors do not show up while running in watch mode during development but do show up after you exit watch mode and re-start. These errors may have been introduced hours earlier and are really tough to find. Won't compile/start until you correct them.
- Now you have to upgrade yet another tool (the TypeScript compiler).
- There will be mismatches and dependencies with configuration. Example: You upgrade to Angular 5 but it depends on TypeScript version 4.3. You now have to upgrade both and deal with breaking changes in multiple layers of dependencies.
- Is anyone really going to read all of this? Well, point proven whether you get to this or not. Suffice it to say that TypeScript is far from perfect.

Example ... How do I solve this?

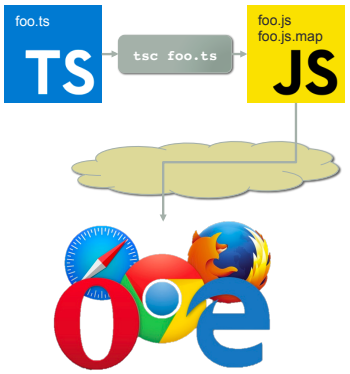
```
module.js:457
    throw err;
    ^
```

Umm ... Exactly 0.0% of this code is mine. Where do I go in my code to solve it?

```
Error: Cannot find module './_baseGetTag'
    at Function.Module._resolveFilename (module.js:
455:15)
    at Function.Module._load (module.js:403:25)
    at Module.require (module.js:483:17)
    at require (internal/module.js:20:19)
    at Object.<anonymous> (/Project Folder/node_modules/
lodash/isFunction.js:1:80)
    at Module._compile (module.js:556:32)
    at Object.Module._extensions..js (module.js:565:10)
    at Module.load (module.js:473:32)
    at tryModuleLoad (module.js:432:12)
    at Function.Module._load (module.js:424:3)
```

How to use TypeScript

Here's how you use TypeScript



1. You write foo.ts
2. tsc transpiles it to foo.js & foo.js.map
3. foo.js is served to browsers

So what does TypeScript make available?

ES Language Additions	Proprietary TypeScript Additions
• Modules • Arrow functions • String templates • Classes	• Strong typing • Constructor shortcuts • Class and member decorators

... and more things that we may not need for Angular

JS

Modules

Syntax

- To expose an object, function, string, class, whatever

```
export <anyObject>
```

- To read that in another file

```
import {anyObject} from 'theFileName.js';
// Now you can do something with anyObject.
```

New Way

Car.js

```
class Car{
  ...
};
export Car;
```

Main.js

```
import {Car} from 'Car.js';
const c = new Car();
```

Arrow functions

Arrow operator

```
func = (p1, p2) => {
  /* Do things with p1 and p2 here. */
  return anythingYouWant;
}
```

- Parentheses can be omitted if # of parameters is one
- Curly braces can be omitted if # of lines is one
  - If you do, the function implicitly returns the value of your one line

For example ...

```
const square = (x) => {
  return x * x;
};
let y = square(4);
```

- or more succinctly ...

```
const square = x => x * x;
```

String templates

Borrowing from mustaches/handlebars

Traditional way

```
var name = 'Your name is ' + first + ' ' + last
+ '.';
var url = 'http://us.com/api/messages/' + id;
```

New way

```
var name = `Your name is ${first} ${last}.`
var url = `http://us.com/api/messages/${id}`
```

Multiline strings

```
var roadPoem =
`Then took the other, as just as fair,
 And having perhaps the better claim
 Because it was grassy and wanted wear,
 Though as for that the passing there
 Had worn them really about the same,`;
```

JS

JavaScript "Classes"

ES2015 gives us a different way to write this

```
class Person{
  speed = 10;
  attack() {
    // Do stuff in this method
  }
}
```

Note: members don't need *this*. nor the function keyword

- Still not real classes, though!
- Just syntactic sugar on top of a constructor function

ES2015 adds getter and setters

```
class Person {
  _alias;
  ...
  get alias() { return this._alias; }
  set alias(newValue) {
    this._alias = newValue;
  }
  attack(foe) {
    let d = `${this._alias} is attacking ${foe.alias}`;
    // Do other attacking stuff here
  }
}
const p = new Person();
p.alias = "Joker";  // Calls set
console.log(p.alias); // Calls get
```

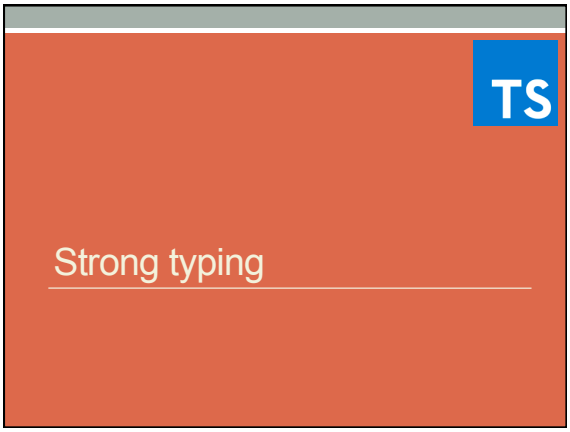
ES2015 adds formal constructors

```
class Person {
  constructor(first, last){
    this._first = first;
    this._last = last;
  }
  doStuff() {
    return `${this._first} ${this._last}
            doing stuff.`;
  }
}

const p = new Person("Talia", "Al-Ghoul");
console.log(p.doStuff());
```

Classes can only have ...

```
class foo {
  constructor() { ...}        // A constructor
  get x() { return this._x; } // Getters
  set x(v) { this._x = v; }   // Setters
  prop1;                      // Properties
  method1() { ... }           // Methods
  static doIt { ... }         // Static methods
}
```

- Can **not** have "this.", "let", "var" that would be a syntax error



We can write in a strongly-typed way now!

Without TypeScript

```
const addThem = (x, y) => x + y;
console.log(addThem(4, 2));
// 6
console.log(addThem("4",2));
// "42"
```

With TypeScript

```
const addThem:number =
  (x:number, y:number) => x + y;
console.log(addThem(4, 2));
// 6
console.log(addThem("4", 2));
// Nope!! tsc error.
```

You'll need types so you can specify the shapes of certain objects.

```
class Person {
  first: string;
  age: number;
  family?: Array<Person>;
}
```

Basic types

Type	
string	
number	
boolean	
Array<someType>	An array of someType
any	Can hold anything at all. Dynamic.
... more (but let's not muddy the water with them for now).	

Make a type optional by putting a "?" after it.

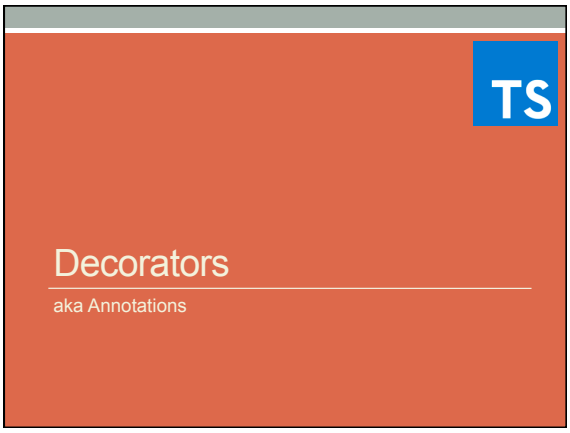
Arrays

- Specify the type and the fact that that it is an array with the square brackets:

```
const Heroes:Hero[] = [
    {id: 1, name: 'Mr. Nice'},
    {id: 2, name: 'Narco'},
    {id: 3, name: 'Bombasto'}
];
```

- or

```
const Heroes:Array<Hero> = [
    {id: 1, name: 'Mr. Nice'},
    {id: 2, name: 'Narco'},
    {id: 3, name: 'Bombasto'}
];
```

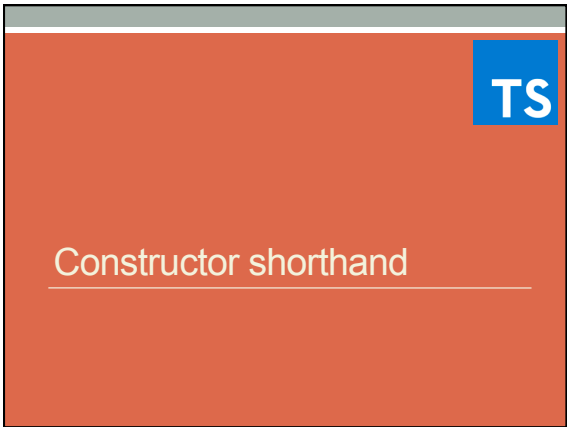


Decorators are like an interface and a constructor combined

Interface	Guarantees conformance to a predefined agreement
Constructor	Makes a function run on that object when instantiated

```
@Pipe()
class AddDays {
  ...
}
```

```
@Component({
  selector: 'my-calc'
})
class Calculator{
  ...
}
```



Private variables are often set via constructor arguments

```
export class MyClass {
  private foo:bool;
  public bar:string;
  constructor(foo:bool, bar:string) {
    this.foo = foo;
    this.bar = bar;
  }
  otherMethod() {
    //Do stuff with this.foo and this.bar
  }
}
```

This is a shorthand for the same thing

```
export class MyClass {
  constructor(private foo:bool,
              public  bar:string) {}
  otherMethod() {
    //Do stuff with this.foo and this.bar
  }
}
```

- Putting *private* or *public* in front of constructor variables will create the class members of the same name.
  - foo is now a private member
  - bar is now a public member
- This is used often in Angular with DI

tl;dr

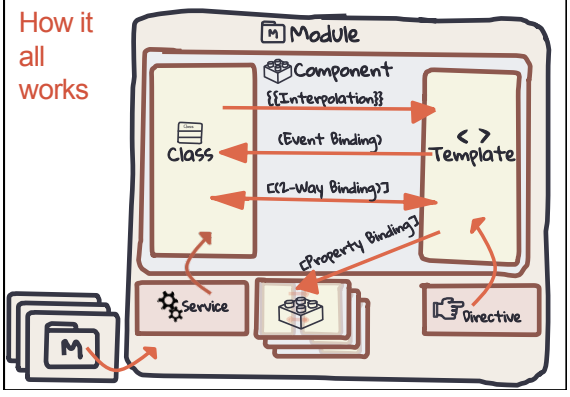
- Angular effectively requires us to use TypeScript (TS)
- Your code is transpiled from proprietary TS into ES5 by tsc
- TS allows us to use modern ES features like modules, classes, and string templates -- even in super-old browsers!
- And it adds proprietary OO features like strong type checking, decorators, and constructor shorthands.
- But it causes friction in setup, development, and debugging

Components

tl;dr

- Components are the central idea of Angular
- They have business logic in a TypeScript class and presentation in a template
- That class must be decorated with @Component and the template and styles are inside the decorator's object literal
- We move data from the logic to the presentation via {{ interpolation }}

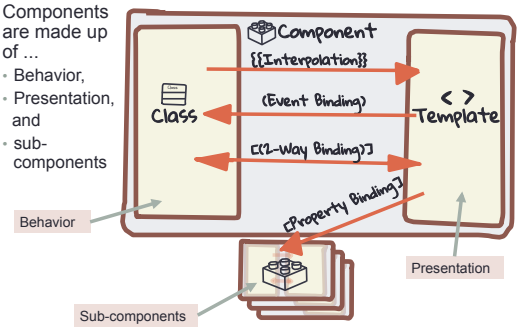
How it all works



Angular apps are made up of components

Components are made up of ...

- Behavior,
- Presentation, and
- sub-components



The business logic is in a TypeScript class

It has a @Component decorator ...

... that receives an object ...

```
@Component({
  template: `<someHTML />`,
  selector: 'foo-bar'
})
export class FooBarComponent {
  // Logic goes here in JavaScript methods
}
```

... with a selector property

It's a class...

That must be exported

The selector uses CSS Selector language

This selector ...	... will match this HTML
'my-foo'	<my-foo></my-foo>
'[my-foo]'	<div my-foo>
'section.special > my-foo'	<section class="special"><my-foo></my-foo></section>

... and so forth

If you want to know more about CSS selectors, look here: http://bit.ly/CSSSelectors

Naming conventions

Name it an object. It keeps track of beans, so it's a "bean counter"

If you add its identity to the name, you'll always know what it is.

It's a TypeScript file so put a .ts on the end.

file name	bean-counter.component.ts
class name	BeanCounterComponent
CSS selector	<my-bean-counter>

Can't name a JavaScript class with dots so Pascal-case it.

HTML isn't case-sensitive so kebab-case it

The type is redundant and ugly. Remove that

Add a prefix to ensure it is unique

Interpolation

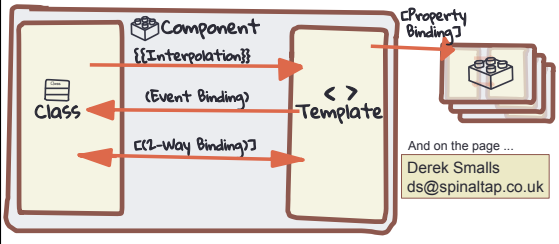
Getting stuff from the class to the template

Interpolation is one-way data binding

- Passing business logic data to the template for display on the page

```
this.person =
  getPerson(1234);
```

```
<p>{{person.first}} {{person.last}}</p>
<p>{{person.email}}</p>
```



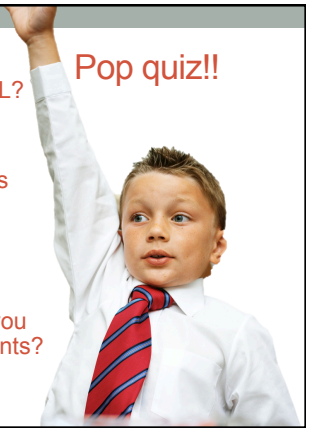
Styling a component

Pop quiz!!

How do you style HTML?
With CSS!

What are Angular views made from?
<html> and other subcomponents

So how do you figure you style Angular components?
With CSS



You have three options to apply styles to your components ...

Separate stylesheets

styleUrls

styles

Styling with stylesheets

```
@Component({
  selector: 'foo-bar',
  template: 'foo-bar.html'
})
export class FooBarComponent {}
```

```
foo-bar {
  margin: 10px;
  padding: 5px;
}
```

Styling within a component

```
@Component({
  selector: 'foo-bar',
  template: 'fooBar.html',
  styles: [`
    foo-bar {
      margin: 10px;
      padding: 5px;
    }
  `],
  styleUrls: ['app/foo.css',
    'app/bar.css']
})
export class FooBarComponent {}
```

Styles are encapsulated within the component

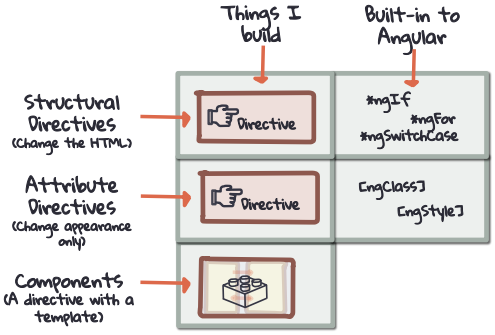
Square brackets means an array: You can have multiple styles

Stylesheets are close to the component

So which technique should I use then?

Separate stylesheet	styleUrls	styles
If styles will be shared across the entire site	If styles will be shared across two or more components	If styles are unique to this component.

There are really 5 types of directives ...



tl;dr

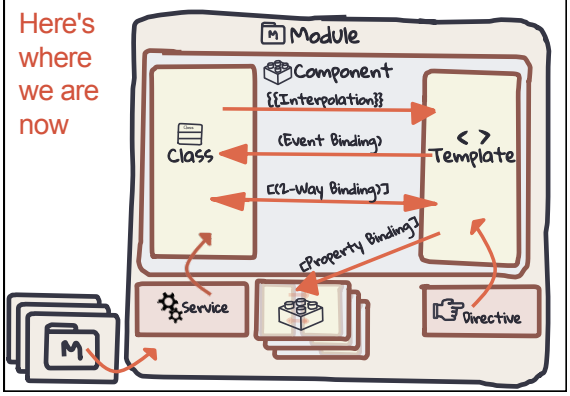
- Components are the central idea of Angular
- They have business logic in a TypeScript class and presentation in a template
- That class must be decorated with @Component and the template and styles are inside the decorator's object literal
- We move data from the logic to the presentation via {{ interpolation }}

Property Binding

tl;dr

- DOM Properties !== HTML attributes because attributes are merely strings and properties are objects
- When we need robust objects, we must use property binding which is done with square brackets

Here's where we are now



How would we solve this?

- Say you have two CSS classes,
  - alert-danger and
  - alert-success.

Oh noes!!

Yay!

- You perform some task in the class and display a message to the user
- If it is successful, you apply alert-success to it. If not, you apply alert-danger to it.
- How can you do that?

```
<div class="_________________">{{ message }}</div>
```

In the class

```
const result = getData();
if (result === "good") {
  message = "Yay!";
  messageClass="alert-success";
} else {
  message = "Oh noes!!";
  messageClass="alert-danger";
}
```

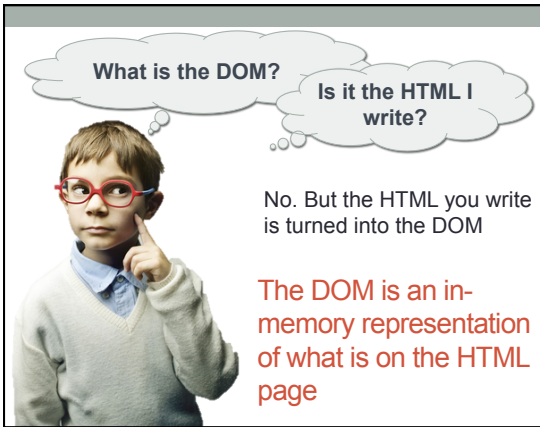
You could use {{ interpolation }}

In the HTML

```
<div class="{{ messageClass }}">
   {{ message }}
</div>
```

But this won't work in every situation because interpolation always *toStrings()* the value

To understand the problem, let's look at the difference between a DOM attribute and a DOM property



No. But the HTML you write is turned into the DOM

The DOM is an in-memory representation of what is on the HTML page

API of a DOM element

```
<input style="border:1px dashed purple"
type="text" onblur="checkData()" />
```

1. **Attributes**: Set in HTML to apply properties to a DOM object. (eg: type, style, onblur);
2. **Events**: Things it can respond to (eg: click, focus, blur, etc.)
3. **Methods**: Things it can do. (eg: setAttribute())
4. **Properties**: Things which describe the object. (eg: type, style, children, disabled).

```
style: {
  alignContent: "",
  ...
  border: "1px dashed purple",
  borderBottomColor: "purple",
  borderBottomLeftRadius: "",
  borderBottomWidth: "1px",
  ...
  color: "",
  ...
  zIndex: ""
}
```

So what exactly is the difference between an attribute and a property?

Attributes
Always a string
Always set properties

Properties
Often an object
Do not set attributes

Basically whenever you want Angular to "process" your string, you put the attribute in square brackets.

```
<any [foo]="bar"></any>
```

Hey, Angular! Go look in this component's class and set its *foo* property to my *bar* object.

Example 1: you want the login button to be disabled if the user is already logged in

```
<button disabled="{{ user }}">Log in</button>
```

```
<button [disabled]="user">Log in</button>
```

Example 2: You want to display a person in it own subcomponent*

```
<contact-card person="{{ person }}"></contact-card>
```

```
<contact-card [person]="person"></contact-card>
```

* Much more on subcomponents later in the course

Example 3: You want to conditionally set a CSS class

```
<contact-card
class="{{ person === selectedPerson ? 'selected' : ' ' }}">
</contact-card>
```

This won't work b/c all other classes would be clobbered.

```
<contact-card [class.selected]="person ===
selectedPerson"></contact-card>
```

Let this be your rule of thumb...

- If you're displaying a value, use {{ interpolation }}
- But if you're setting an attribute, use [property binding]

tl;dr

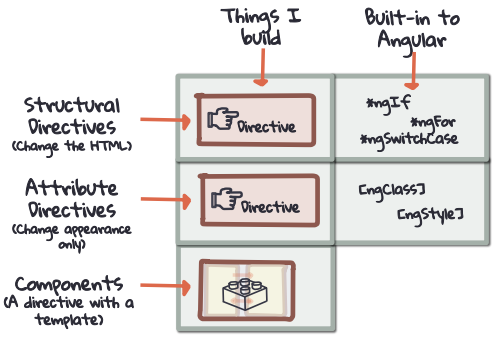
- DOM Properties !== HTML attributes because attributes are merely strings and properties are objects
- When we need robust objects, we must use property binding which is done with square brackets

Built-in Directives

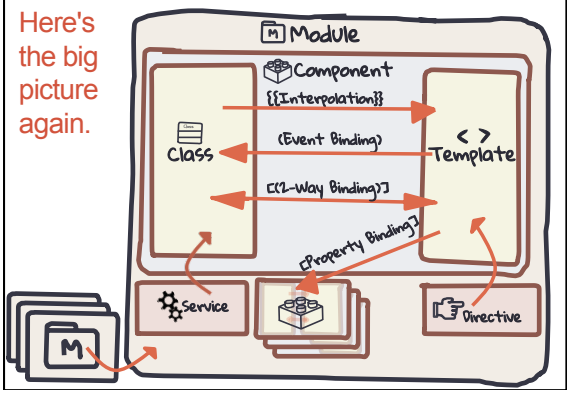
tl;dr

- Angular has custom directives and built-in directives. This chapter is about built-in ones.
- Of the built-in directives, there are two types: structural directives and attribute directives.
- Structural directives change the actual DOM
  - *ngIf - Adds things to the page conditionally
  - *ngFor - Repeats DOM elements
- Attribute directives change page appearance but not structure
  - [ngStyle] - Modifies CSS styles conditionally
  - [ngClass] - Turns on/off CSS classes conditionally

There are really 5 types of directives ...



Here's the big picture again.

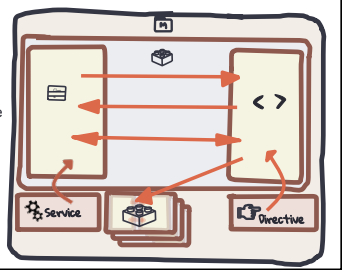


Directives

- Components are merely directives with a template.
- Directives should be template-agnostic

The built-in directives

1. Structural directives
   Change the DOM itself, adding or removing whole branches
2. Attribute directives
   Change behavior but not the DOM



Attribute Directives

Attribute directives change the appearance but not the structure of a component.

[ngStyle]
[ngClass]

ngStyle

How ngStyle works

- Because of property binding, we *could* ...

```
[style.color]="someColor"
```

- And then set it in the class like so

```
this.someColor = "#55302e";
```

- But if we wanted to set multiple styles dynamically, we'd have to have a whole lot of [style.foo] bindings.
- ngStyle is a shortcut for that!
- it allows you to create an object with the properties you want set in one statement.

ngStyle example

```
this.lowPad=10;
this.highPad = 20;
```

```
<div [ngStyle]="{'color': '#BBB',
'paddingTop': highPad, 'padding-bottom': lowPad}">
  <p>Lorem ipsum</p><img src="foo.jpg" />
</div>
```

ngStyle is great for low-grained control

- But managing presentation at this level can become hard to maintain.
- Better to use classes!

ngClass

Managing classes without ngClass

- We could manage classes using property bindings:

```
<div [class]="bigOrNot"></div>
```

- But you can only set one class at a time like this.

[ngClass] allows you to set multiple classes on an element conditionally

- Multiple classes could be done like this:

```
<div [class.big]="bigOrNot"
     [class.fancy]="fancyOrNot">
```

- And this works, but it is much cleaner to use *ngClass

```
this.classes = {
  'big': someTruthyValue,
  'fancy': someOtherTruthyValue
};
```

- then

```
<div [ngClass]="classes"></div>
```

Structural Directives

Structural directives start with a "*"

They always say "Alter the DOM with this element"

*ngIf	If some condition is truthy, show this element.
*ngFor	Repeat this element once for each thing in this collection
*ngSwitchCase	Pick an element from several options

*ngIf

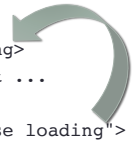
*ngIf is for when you want to show an element conditionally

- It uses JavaScript "truthiness"

```
<div *ngIf="selectedComic">
  <img src="icon.jpg" />
  <p>{{ selectedComic.description }}</p>
</div>
```

- When there's no selectedComic, the div isn't just hidden; it is removed from the DOM completely.

*ngIf has an else clause



```
<ng-template #loading>
Loading. Please wait ...
</ng-template>
<p *ngIf="ready; else loading">
Loaded, {{ user }}. You may proceed.
</p>
```

Do this!

```
<div *ngIf="showIt">
  Things to show
</div>
```

Only *ngIf will remove it from the DOM. The [hidden] can be easily overridden in your CSS!

You'll be tempted to do this ...

Don't do this:

```
<div [hidden]="!showIt">
  Things to show
</div>
```

*ngSwitchCase

*ngSwitchCase allows complex conditionals

- When you get a lot of if statements against the same value it may help to refactor to an *ngSwitchCase
- *ngSwitchCase is worthless without a property binding to [ngSwitch].

For example ...

this.company = "Marvel";

```
<div [ngSwitch]="company">
 <comics *ngSwitchCase="DC">DC</comics>
 <comics *ngSwitchCase="Marvel">Marvel</comics>
 <comics *ngSwitchCase="DarkHorse">Dark Horse</comics>
 <comics *ngSwitchDefault>All companies</comics>
</div>
```

*ngFor

*ngFor is a presentation layer for-each statement

Load up an array in the business layer and loop through it in the presentation

Example 1: Show all the things!

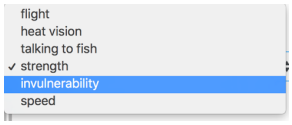
```
this.powers = [
{id:1,name:'flight'},
{id:2,name:'heat vision'},
{id:3,name:'talking to fish'},
{id:4,name:'strength'},
{id:5,name:'invulnerability'},
{id:6,name:'speed'}
];
```

- flight
- heat vision
- talking to fish
- strength
- invulnerability
- speed

```
<div *ngFor="let power of powers">
   {{ power.name }}
</div>
```

Example 2: *ngFor is a great solution for <select>s

```
this.powers = [
{id:1,name:'flight'},
{id:2,name:'heat vision'},
{id:3,name:'talking to fish'},
{id:4,name:'strength'},
{id:5,name:'invulnerability'},
{id:6,name:'speed'}
];
```




```
<label for="power">Hero Power</label>
<select class="form-control" id="power" required>
  <option *ngFor="let pow of powers"
          [value]="pow.id">{{pow.name}}</option>
</select>
```

When the collection changes, *ngFor changes the DOM

When the collection changes like this ...	*ngFor will ...
An element is added	Insert the new template into the DOM
An element is removed	Pull the template out of the DOM
The collection is reordered	Recreate and re-render that section of the DOM

It does NOT just show and hide things.

*ngFor exposes certain variables on each iteration

variable	type	
index	number	Which loop are we on in the array? (zero-based)
first	bool	True only on the first row
last	bool	True only on the final row
even	bool	True on the 1st, 3rd, 5th, 7th, 9th ... rows
odd	bool	True on the 2nd, 4th, 6th, 8th, 10th ... rows

Example 3: Using *ngFor variables

```
this.sidekicks = [
{id:112,name:'Kid Flash'},
{id:256,name:'Aqualad'},
{id:340,name:'Wonder Girl'},
{id:494,name:'Ms. Martian'},
{id:544,name:'Speedy'},
{id:612,name:'Robin'}
];
```

1 - Kid Flash
2 - Aqualad
3 - Wonder Girl
4 - Ms. Martian
5 - Speedy
6 - Robin

```
<div *ngFor="let sk of sidekicks; let i=index">
  {{ i + 1 }} - {{ sk.name }}
</div>
```

tl;dr

- Angular has custom directives and built-in directives. This chapter is about built-in ones.
- Of the built-in directives, there are two types: structural directives and attribute directives.
- Structural directives change the actual DOM
  - *ngIf - Adds things to the page conditionally
  - *ngFor - Repeats DOM elements
- Attribute directives change page appearance but not structure
  - [ngStyle] - Modifies CSS styles conditionally
  - [ngClass] - Turns on/off CSS classes conditionally

Routing

Pop quiz: How do we include external modules in the current module?

```
import {OtherModule} from
'./other.module';
@NgModule(
 imports: [
  OtherModule
 ],
 // More things here
)
class ThisModule() {}
```

Why routing?

Much of the page stays the same

So why swap out the entire page?!?

- Why don't we keep the same page and use Ajax to swap out only the parts that change?
- We could call them "Single Page Apps" or ...

SPAs

Angular thinks in SPAs first

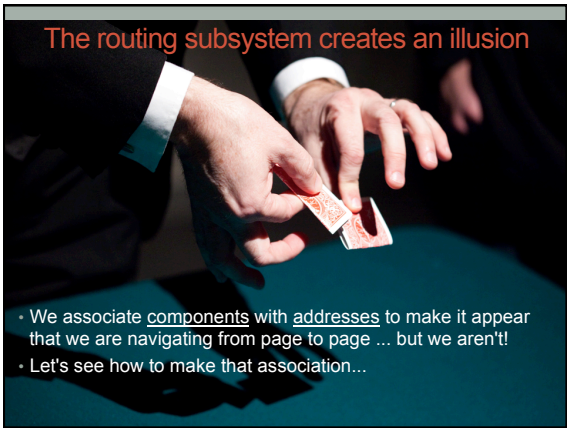
- You can have multiple pages if you want, but you have to work to make that happen....
- Including work on the server!

A page (index.html) hosts our main component (AppComponent)

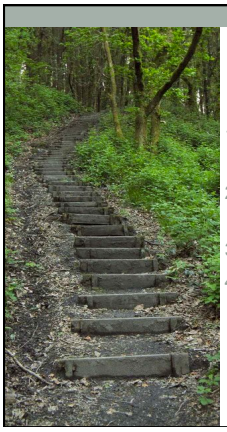
- Most browsers don't understand components (yet) so they need a page.
- The main component holds the *chrome* for the page

Chrome is the stuff that doesn't change from page to page (like headers, footers, nav links, maybe)

The routing subsystem creates an illusion



- We associate components with addresses to make it appear that we are navigating from page to page ... but we aren't!
- Let's see how to make that association...



To route with Angular ...

1. Add a <router-outlet> to a host component
2. Make routes available to a module
3. Allow the user to visit routes
4. Use the route parameters in the called components

1. Create a router outlet

Create a router outlet

- In the host component (usually AppComponent), we need to define the static content (aka the chrome. aka the stuff that doesn't change)
- And the place where dynamic components will live (aka the content that changes when the user navigates somewhere).
- This place is an outlet for the router

it is a <router-outlet></router-outlet>

2. Make routes available to a module

Design the routes ...

URL	Component
/people	PeopleComponent
/people/123	PersonComponent
/teams	TeamComponent
/cart	ManageCartComponent
/	WelcomeComponent
Anything else	404Component

```
const routes = [
 {path: "people", component: PeopleComponent},
 {path: "people/:personId", component: PersonComponent},
 {path: "teams", component: TeamComponent},
 {path: "cart", component: ManageCartComponent},
 {path: "", component: WelcomeComponent},
 {path: "**", component: 404Component}
];
```

... and put them in a router module

```
export const routing =
            RouterModule.forRoot(routes);
```

We'll export a `routing` constant initialized using the `RouterModule.forRoot` method applied to our array of routes. This method returns a **configured router module** that we'll add to our root NgModule, `AppModule`.

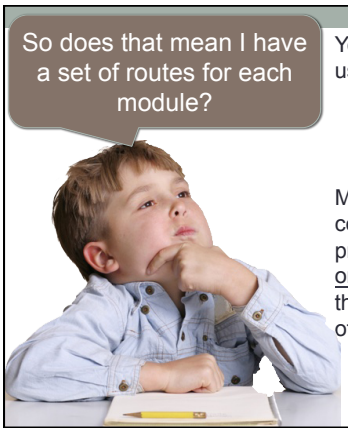
And how do we include a new module? ....

app.module.ts:

```
import { routing } from './app.router';
@NgModule({
imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    routing
  ],
bootstrap: [AppComponent]
// More things here ...
})
export class AppModule { }
```

import it here. (Lets this file see it).

imports it here. (Lets this module see it).



You can but if you do, use:

- forRoot() in your root module and
- forChild() in all others

Most routes will be common. So best practice is to have one routing module that you import into all other modules

3. Allow the user to visit routes

How can a user get to a resource?

At run time, a user can ...

1. Type a url
2. Click a link
3. Get pushed there in a Component class

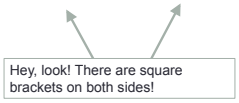
They can type in a URL

- When the user types in the URL, the browser has no choice but to request a resource from the server.
- The server will have been configured to serve the root page at any address.
- Routing will then will intercept that and route him to the proper component.

They can click on a link

- Write your links like this:
- <a [routerLink]="['/people']">People
- <a [routerLink]="['/teams']">Teams
- <a [routerLink]="['/cart']">My cart

- <a [routerLink]="['/someLink']">Text to display



Why square brackets on the left side?

- This would technically work:

```
<a href="/people/{{p.id}}">{{ p.first }}</a>
```

- But it isn't the best option. Why not?
- b/c a simple href will send a whole new http request to the server. If your server has this route defined, cool.
- Instead we allow Angular to handle client-side routing with routerLink.

Why square brackets on the right side?

- Because the argument is actually a JavaScript array.
- Each member corresponds to a part of the route.

<foo _____>bar</foo>	... will route you to ...
[routerLink]="['/people']"	/people
[routerLink]="['/people', theDude.Id]"	/people/:personId
[routerLink]="['/blog']"	/blog
[routerLink]="['/blog', year]"	/blog/:year
[routerLink]="['/blog', y, m, d]"	/blog/:year/:month/:day

They can get pushed in the class

```
export class FooComponent {
  constructor(private _router: Router) { }

  goToPerson(p) {
    this._router.navigate(['/people', p.Id]);
  }
}
```

4. Use route parameters in the called components

Pop quiz!

- You have a list of persons. Each person is clickable. When you click him/her, the details for that person will come up.
- How will that be done?

```
<ul>
 <li><a [routerLink]="['people',127]">Jo</a></li>
 <li *ngFor="let p of persons">
  <a [routerLink]="['people',p.id]">{{p.first}}</a>
 </li>
</ul>
```

- Very cool! That's how to WRITE the id, but how do you READ the id?

Here's how to read the route parameter

```
@Component()
class PersonComponent {
 constructor(private _route: ActivatedRoute) {}
 ngOnInit() {
  const id;
  id=this._route.snapshot.params['personId'];
  // Now you can do stuff with "id" here.
 }
}
```

tl;dr

- Angular thinks in SPAs first which is more satisfying for the user and easier on our servers. Everyone wins!
- The routing subsystem sends users to a specific component based on the URL they choose. To do this we ...

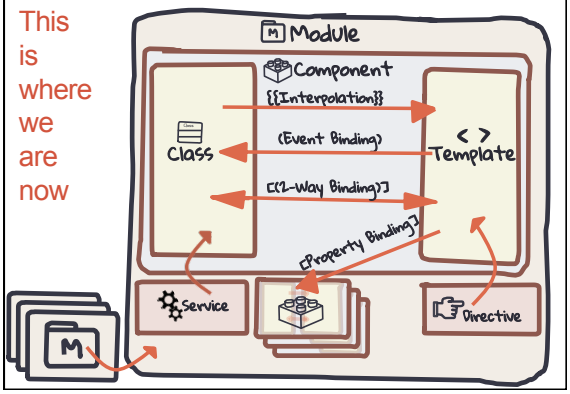
1. Create a <router-outlet>
2. Expose the routes to a module
3. Allow the user to navigate to a route by
   - Linking (<a [routerLink]="['foo']">)
   - Typing in a URL directly (routed by Angular - slow)
   - Pushing in a component (router.navigate())
4. Use route parameters in the component via route.snapshot.params[]

Event Binding

tl;dr

- Angular handles all events that the browser is aware of.
- To create an event handler, you write a method in the component's class and wire it up to the event in the template.
- You use parentheses in the template to associate the event name with the method name.

This is where we are now



Some Angular events

- blur
- change
- click
- copy
- cut
- dbl-click
- focus
- keydown
- keypress
- keyup
- mousedown
- mouseenter
- mouseleave
- mousemove
- mouseout
- mouseover
- mouseup
- paste
- submit
- ...

... basically every event that the browser can respond to, Angular has an interface to.

Put your event name inside parentheses

- The parentheses simply tell Angular that it needs to add an event listener for the thing inside them.

```
<any (foo)="bar()"></any>
```

Hey, Angular! When the user triggers the *foo* event, go look in this component's class and run the *bar* method.

Examples:

```
<button (click)="doIt()">
  Press me</button>
<img (mouseover)="count()" />
<input (blur)="go()"
       (keyup)="run()" />
```

Mouse events

- click
- dbl-click
- mousedown
- mouseenter
- mouseleave
- mousemove
- mouseover
- mouseup

```
<button (click)="processOrder()">
Go</button>
<img src="..."
 (dbl-click)="viewProduct(product)" />
<img src="..."
  (mouseover)="count = count + 1" />
```

Form events

- focus
- blur
- change
- cut
- copy
- paste
- submit
- keydown
- keyup
- keypress

```
<input (focus)="checkAllFields()"
  (blur)="checkAgain()"
  (keyup)="getSuggestions()" />
```

A note about Angular events

- Remember, to think in Angular means to think in terms of the component's state and not simply on what you want to do on the page.
- You're going to be tempted to use events to fall back into old imperative habits.
- Just be careful... Think component state first!

Events bubble in Angular

- Events are caught on the same element as well as from events that bubble up from child elements.

```
<section (click)="doStuff()">
  <div>
    <ul>
      <li><a>Home</a></li>
      <li><a>About us</a></li>
      <li><a>Contact us</a></li>
    </ul>
  </div>
</section>
```

- This will execute doStuff() if the section or any of it's nested elements are clicked.

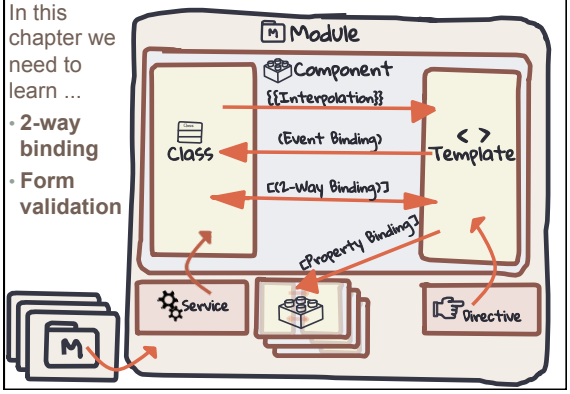
tl;dr

- Angular handles all events that the browser is aware of.
- To create an event handler, you write a method in the component's class and wire it up to the event in the template.
- You use parentheses in the template to associate the event name with the method name.

Forms and 2-way binding

In this chapter we need to learn ...

- **2-way binding**
- **Form validation**



Prerequisite!

- We NEED FormsModule to make this work, which is in @angular/forms so add it to the app.module.ts like so ...

```
import { FormsModule } from '@angular/forms';
@NgModule({
  imports: [ BrowserModule, FormsModule ],
  declarations: [ App ],
  bootstrap: [ App ]
})
export class AppModule {}
```

```
▶Uncaught Error: Template parse errors:
There is no directive with "exportAs" set to "ngModel"

     <input [(ngModel)]="givenName" [ERROR ->]#givenNam
     name="givenName" id="givenName" minlength="4" maxl
     require"): ng:///SolutionsModule/FormsDemoComponen
Can't bind to 'ngModel' since it isn't a known propert
```

Two models of forms

Template-driven	Reactive
• In the template only • More declarative • More intuitive • Self-documenting	• (aka model-driven, testable, etc). • Built in the class • Bound to the template • More testable

Two-way binding

... with bananas in a box!

Let's cut to the chase...

Here's how you two-way bind

In the class

```
familyName:String = "";
```

In the template

```
<input [(ngModel)]="familyName" />
```

What is really happening, though ...

- AngularJS's 2-way binding is slow, so Angular removed it.
- Instead, two bindings happen:

```
<input
  [ngModel]="firstName"
  (ngModelChange)="firstName=$event" />
```

- But that's a lot of typing, so they gave us a shorthand ... Banana-in-a-box

= "person.firstName"

Watch out for this trap!

```
<form>
  <input [(ngModel)]='firstName' name='firstName' />
</form>
```

You must use a *name* attribute to use ngModel binding in a form!

Forms validation

You can't validate something without a view handle!

- How would we know exactly what it is we're validating?
- Form binding does that

```
<someTag #key="value" />
```

- For example:

```
<form #formVH="ngForm" ...>
```

- The view now knows this form as *formVH*

```
<input #firstNameVH="ngModel" ... />
```

- The view now knows this input as *firstNameVH*

So how do we use these handles? Here's an example

- We can prevent the submission of invalid forms

```
<form (ngSubmit)="go()" #mainForm="ngForm">
```

```
<input type="submit" [disabled]="mainForm.invalid" />
```

ngSubmit will halt if the handler code throws. submit continues the submission. So ngSubmit is preferred.

Status tokens

What is the current status of each form field?

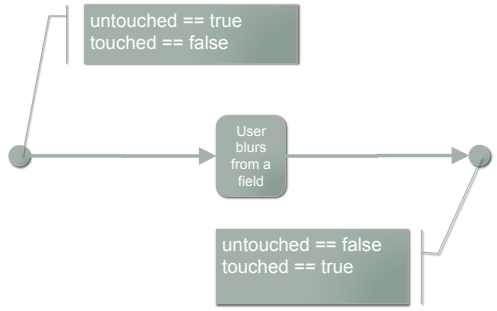
Angular applies status tokens to the fields

touched and untouched

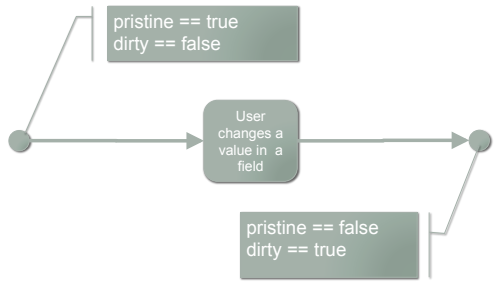
pristine and dirty

valid and invalid

untouched and touched



pristine and dirty



invalid and valid

- valid means meeting all rules
  - minlength
  - maxlength
  - required
  - pattern
  - type=number
  - type=email
  - type=date
- invalid means that at least one of those are violated

This makes our form really usable!

```
<p *ngIf="f.invalid">fields with errors have a '*'</p>
<form name="f" #f="ngForm"
      (ngSubmit)="f.valid && doStuff()">
  <input name="first" #firstVH="ngModel"
         [(ngModel)]="first" required pattern="\w+" />
  <span *ngIf="firstVH.touched && firstVH.invalid">
    *
  </span>
  <input name="last" #lastVH="ngModel"
         [(ngModel)]="last" *ngIf="firstVH.dirty" />
  <span *ngIf="lastVH.touched && lastVH.invalid">
    *
  </span>
  <input type="submit" [disabled]="f.invalid" />
</form>
```

errors tokens

Error tokens

errors.email
errors.maxlength
errors.minlength
errors.pattern
errors.required
errors.url
errors.date

We combine status and error tokens

```
<form>
<input [(ngModel)]="card" name="card" #cardVH="ngModel"
 required pattern="(\d{4}[ -]?){4}" />

<div *ngIf="cardVH.touched && cardVH.errors &&
              cardVH.errors.required">
We need a credit card number.
</div>

<div *ngIf="cardVH.touched && cardVH.errors &&
              cardVH.errors.pattern">
That doesn't look like a credit card.
</div>

</form>
```

Form classes

CSS classes, to be specific

The HTML you wrote

```
<form>
<input name="f" pattern="\w+" />
<input name="l" required />
</form>
```

Note: there are more classes than these. We're shortening for clarity.

What Angular renders at first

```
<form>
<input name="f" class="ng-untouched ng-pristine ng-valid" />
<input name="l" class="ng-untouched ng-pristine ng-invalid"/>
</form>
```

What Angular renders after the user enters data

```
<form>
<input name="f" class="ng-touched ng-dirty ng-invalid" />
<input name="l" class="ng-touched ng-dirty ng-valid" />
</form>
```

How to make elements look good

- Put things like this in your CSS style sheets ...

```
input.ng-invalid.ng-touched {
  background-color: pink;
  color: red;
}

input.ng-valid.ng-touched {
  background-color: lightgreen;
  color: green;
}
```

tl;dr

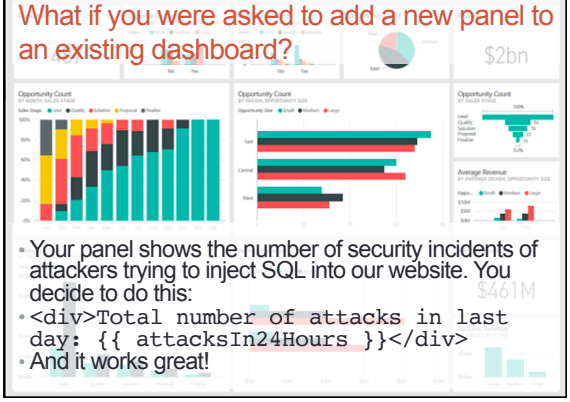
- 2-way binding is done with [(banana-in-a-box)]
- You must use form-binding (#handle="ngModel") for validations
- We can use
  - Status tokens (touched, dirty, invalid, etc),
  - Errors tokens (required, pattern, email, etc),
  - and classes (ng-touched, ng-dirty, ng-invalid)

Composition with Components

tl;dr

- Applications are comprised of nested components which are made up of nested components and so forth and so on.
- We usually want inner components and their host components to be able to send data up and down.
- Data goes from host to inner via [property binding]
- Data goes from inner to host via Event Emitting.
- If we do things right, we can even mimic 2-way binding.

What if you were asked to add a new panel to an existing dashboard?



- Your panel shows the number of security incidents of attackers trying to inject SQL into our website. You decide to do this:
- `<div>Total number of attacks in last day: {{ attacksIn24Hours }}</div>`
- And it works great!

But a problem crops up!

- You get an email (cc'd to all of course).

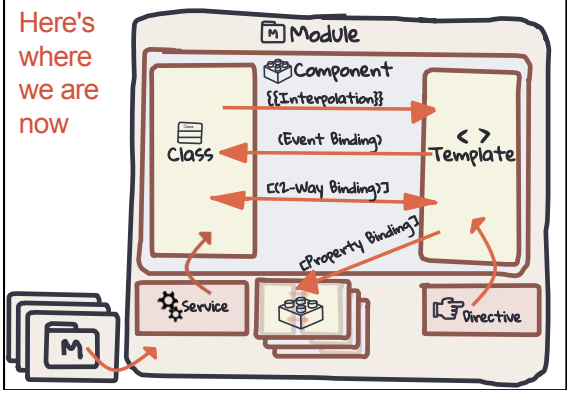
```
"You broke the dashboard page! The Virus panel
is reporting a wrong number. I spent the last
few hours debugging and found that your new
panel has clobbered my "attacksIn24Hours"
variable. You've got to rename it!"
```

- What do you do? Rename it to attacks_in_24_hours? And you get another call that something else has broken.
- So you name it to injection_attacks_in_24_hours and then sql_injection_attacks_in_24_hours.
- And you have to do the same for every one of the 300 variables you are using.

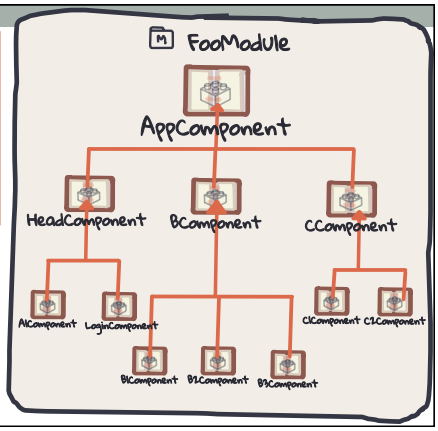
And you have to duplicate your panel

- Then someone asks you to put your panel on the landing page of the internal website.
- You go through the same gyrations, slightly modifying your code, copying and pasting.
- Then they ask you to do the same for the executive dashboard. And again you're copying, pasting, and modifying for the environment.
- Then you discover a race condition in one of those pages and realize the problem is in all of them! You now have to duplicate your change in all three places.

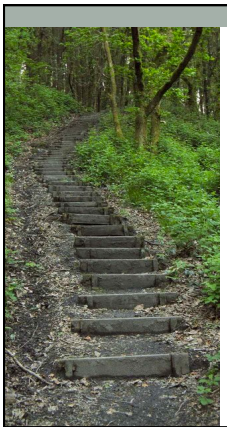
Here's where we are now



Our strategy will be to write components that are composed of other components which are composed of other components which .....



How to compose



How to compose

1. Put a <tag> in the parent component
2. If you want data to flow to the child, use property binding
3. If you want data to flow up use event binding.

Remember: components must be imported and be visible to this module

The inner

1. Add a tag

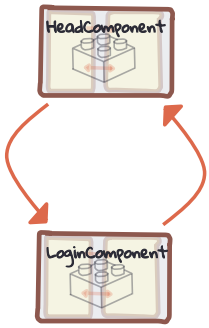
```
@Component({
  selector: 'login',
  templateUrl: 'login.component.html'
})
export class LoginComponent {
}
```

The host

```
@Component({
 template: `<div><login></login></div>`;
})
export class HeadComponent {
}
```

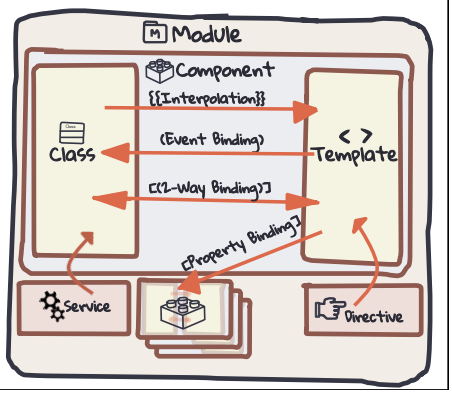
A word about data flow

We may want data to flow from host component to inner component and back up again.



2. Pass data from host to inner

Remember property binding?



head.component.ts

```
@Component({
 templateUrl: `head.component.html`;
})
export class HeadComponent {
 loginUser;
 constructor() {
   this.loginUser = getUser(1234);
 }
}
```

To pass data from host to inner

head.component.html

```
<div>
 <login [user]="loginUser">
 </login>
</div>
```

You'll set the value in the host class ...

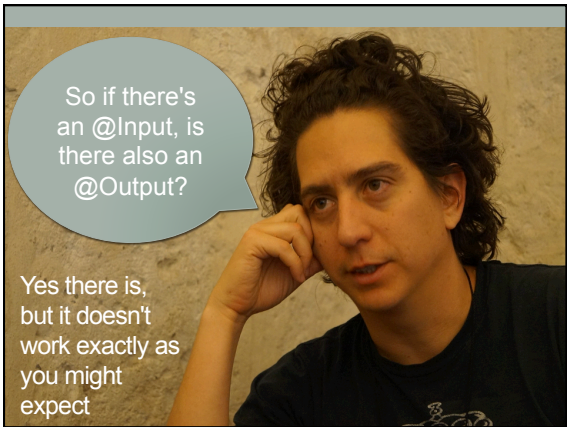
... And pass it down as a value of a property binding.

To read data in inner from host

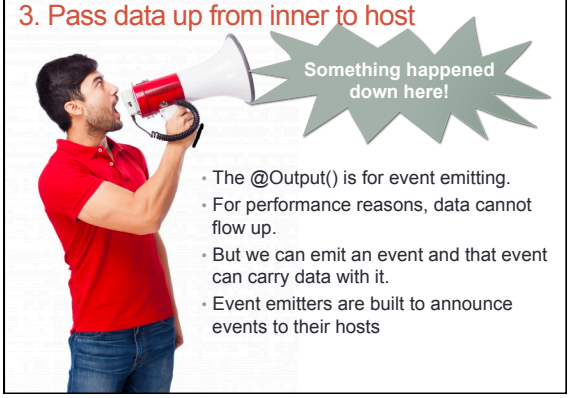
login.component.ts

```
import { Input } from '@angular/core';
@Component({})
export class LoginComponent {
 @Input()
 user;
}
```

Mark it with the *@Input()* annotation to make it readable from a host component.



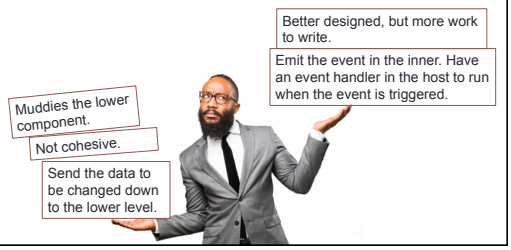
3. Pass data up from inner to host



- The @Output() is for event emitting.
- For performance reasons, data cannot flow up.
- But we can emit an event and that event can carry data with it.
- Event emitters are built to announce events to their hosts

Emitting also facilitates cohesion

Events should be handled at the level at which they make sense and not lower. But the button (or whatever) may be drawn at a lower level.



login.component.ts

```
import { EventEmitter, Output }
 from '@angular/core';
@Component()
export class LoginComponent {
 @Output()
 loggedIn = new EventEmitter();

 changeUser(user) {
  this.loggedIn.emit(user);
 }
}
```

Mark the emitter as @Output()

... And emit the event with the value you want to send up.

To send data from inner to host

head.component.html

```
<div>
 <login (loggedIn)="loginUser($event)">
 </login>
</div>
```

When the loggedIn event fires, run the loginUser() method

To read data in host from inner

head.component.ts

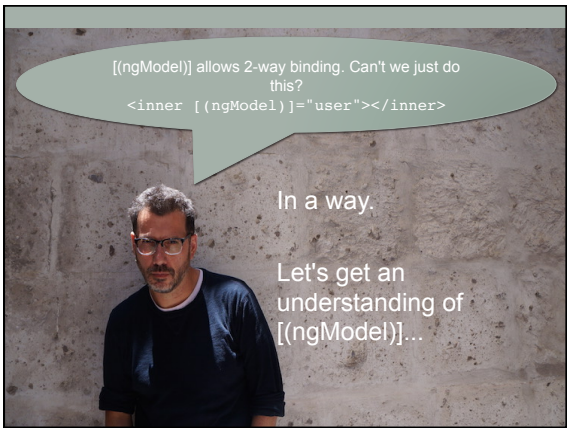
```
import { Input } from '@angular/core';
@Component({})
export class HeadComponent {
 loginUser(theUser) {
  // This method will execute when the
  // loggedIn event fires in the inner
 }
}
```

2-way binding between components

[property binding] may be all you need for 2-way binding.

IF it is an object we're binding and IF its reference does not change in the inner component and IF only the properties are directly changed, then property binding is enough.

Since this isn't always possible, we'll learn how to 2-way bind.



[(ngModel)] is actually syntactic sugar!

```
<input [(ngModel)]="userName" />
```

```
<input
     [ngModel]="userName
               (ngModelChange)="userName=$event"
/>
```

So if we reverse-engineered that pattern ...

```
<inner [(user)]="loginUser" />
```

```
<inner
     [user]="loginUser"
               (userChange)="loginUser=$event"
/>
```

Must be a real property

Must be a real emitted event

Name must be the property + "Change"

If we want a 2-way binding...

header.component.html

```
<login [(user)]="loginUser"></login>
```

Wonderfully simple! :-)

Pretty darn complex. :-(

login.component.ts

1. Add an EventEmitter called "userChange"
2. Mark it with @Output()
3. Separate user into a getter and setter.
4. Mark the setter with @Input()
5. Emit the new value in the setter

login.component.ts

```
@Component()
export class LoginComponent {
 @Output()
 userChange = new EventEmitter();
 _user;
 get user() {
   return this._user;
 }
 @Input()
 set user(value) {
  this._user = value;
  this.userChange.emit(this._user);
 }
}
```

2. Marked with @Output
1. Properly named Event Emitter
3. Getter and setter separated
4. Setter marked with @Input
5. Value emitted to our host

tl;dr

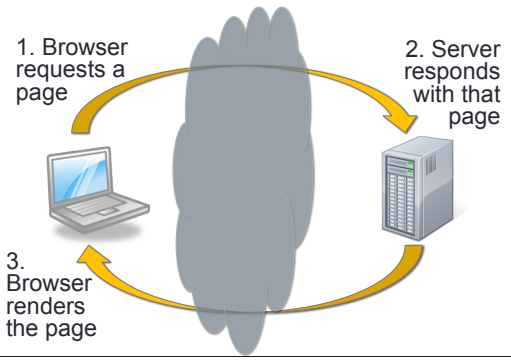
- Applications are comprised of nested components which are made up of nested components and so forth and so on.
- We usually want inner components and their host components to be able to send data up and down.
- Data goes from host to inner via [property binding]
- Data goes from inner to host via Event Emitting.
- If we do things right, we can even mimic 2-way binding.

Ajax with Angular

tl;dr

- Angular gives us a great way to work with Ajax -- Http
- Http makes any kind of Ajax request - GET, POST, PUT, DELETE, etc.
- If we convert its return to a promise, we'll handle the response in a ".then()" and pass in success and failure functions.

Here's how the web works



The server will provide a HTTP status code

100 Continue

404 Not Found

304 Not Modified

500 Internal Server Error

204 No Content

201 Created

200 OK

403 Forbidden

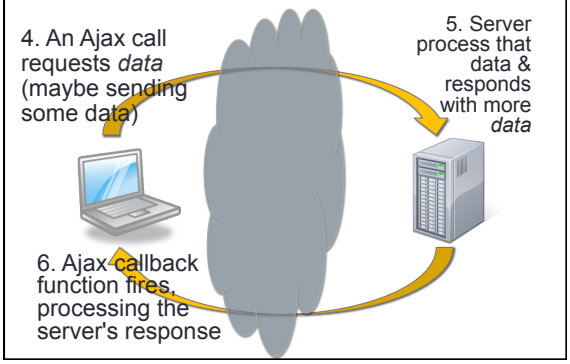
301 Moved Permanently

307 Temporary Redirect

503 Service Unavailable

100 - 199	Info only
200 - 299	Success
300 - 399	Redirect
400 - 499	Bad request
500 - 599	Server error

Here's how Ajax works



AJAX

With Ajax, we're talking about interactions between behavior and presentation.

Angular is all about joining behavior and presentation in a controlled and predictable way.

Angular is the perfect way to handle Ajax

Http

Ajax (Http) is found in a separate module called HttpModule so we have to include it

- In your app.module.ts file, go ...

```
import { HttpModule } from '@angular/http';
```

- and then add it to the imports section

```
imports: [
  BrowserModule,
  FormsModule,
  HttpModule
],
```

```
▶ERROR Error: Uncaught (in promise): Error: No provider for Http!
Error: No provider for Http!
    at injectionError (core.es5.js:1169)
    at noProviderError (core.es5.js:1207)
    at ReflectiveInjector_.webpackJsonp.../../../core/@angular/core
    at ReflectiveInjector_.webpackJsonp.../../../core/@angular/core
```

To use it, we must instantiate it

- Import it at the top

```
import { Http } from '@angular/http';
```

- Then use the constructor injection trick

```
constructor (private _http:Http) { }
```

Then you can GET, POST, PUT, DELETE

Syntax

```
Http.method(url, body, options);
```

Where ...

- method = The HTTP method
  - eg. get, post, put, delete, patch, head, options, connect, trace
- url = Address of the resource
  - eg. http://foo.com/widgets/123, /api/cars, /blog/2018/12/25
- body = Request payload (optional)
  - ie. New data on a POST, Updated data on a PUT/PATCH
  - Usually JSON formatted
- options = Request Options (duh) (optional (also duh))
  - ie. Headers, like content-type

What's the problem with this code?

```
getUsers() {
  this.users = this._http.get("/users");
};
```

Promises have a .then() method to registers callbacks

```
let p = this._http.get(url).toPromise();
p.then(success, error);
```

What to do on a 200-series http return

What to do on a 400-or 500-series http return

Response object

- The success function is called when the XMLHttpRequest returns any 200-series return code
- ... and it passes an object to the success function:

```
{
  json(): Run this to get an object out
  _body: a private with data from the server
  status: HTTP response code
  statusText: HTTP response status text
}
```

So one way to run it might be ...

```
users;
getUsers() {
  this._http
  .get("/users")
  .toPromise()
  .then(
    (res) => { this.users = res.json(); },
    (err) => { console.error("Yikes!",err);}
  );
}
```

One final thought ...

Promises are cool and all but what if ...

- We were fetching a HUGE amount of data? The promise isn't resolved until they all return. That may take a long time.
- We wanted to cancel the activity for some reason. Promises can't be interrupted.
- These things will be fixed by ...

Observables

Coming soon to a lecture near you!

tl;dr

- Angular gives us a great way to work with Ajax -- Http
- Http makes any kind of Ajax request - GET, POST, PUT, DELETE, etc.
- If we convert its return to a promise, we'll handle the response in a ".then()" and pass in success and failure functions.

Observables

tl;dr

- Observables represent a stream of future values.
- They will run a function as data arrives instead of once at the end.
- You tell the observable what it is waiting on.
- Then you use subscribe to tell it what to do when the data is updated
- There are some really useful extension methods that must be added to the observable's prototype before they're used

What is an observable?

Promises are great, but there is room for improvement ...

- They only call the event when the whole process is complete
- They can't be interrupted
- Observables fix these problems

If a promise represents a future value, an observable represents a stream of future values

- Kind of like an array whose items arrive slowly.

```
[
{first:"Mal",last:"Reynolds",email:"capt@serenity.com"},
{first:"Zoë",last:"Washburne",email:"mate@serenity.com"},
{first:"Wash",last:"Washburne",email:"pilot@serenity.com"},
{first:"Jayne",last:"Cobb",email:"jcobb@serenity.com"},
{first:"Kaylee",last:"Frye",email:"mechanic@serenity.com"},
{first:"River",last:"Tam",email:"cargo@serenity.com"},
{first:"Derrial",last:"Book",email:"shepherd@serenity.com"}
]
```

TC 39 tc39 is working on a standard for JavaScript

- Look here for the proposal and status:
`https://tc39.github.io/proposal-observable`

- Until it makes it to all browsers, we use a polyfill called rxjs

How to create and process an observable

You associate a function with an observable

```
const obs = Observable.create(
  observer => {
    setInterval(() => {
      observer.next(Math.random());
    }, 2000);
  }
);
```

.next(value) says to raise the Observable's event (aka. success handler) and provide *value* to it.

Promises can only respond to something running.

Observables won't let them run until you subscribe.

Observables are lazy

So how do you subscribe to an observable?

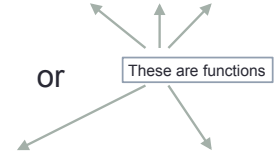
You provide a function to run

subscribe runs the function every time a new value arrives

```
obs.subscribe(v => {
  console.log("s fired", v);
  this.messages.push(v);
});
```

To handle exceptions

```
observable.subscribe(success, error, finally);
```



or

```
observable.catch(error).subscribe(success)
```

That .catch() is an *extension method.*

Extension methods

Extension methods enhance the capability of observables

- catch()
- toPromise()

Some mirror the Array.prototype methods

- map()
- filter()
- reduce()
- skip()
- take()

We need to import extension methods

You can import each extension method like this:

```
import 'rxjs/add/operator/map';
```

or

You can import the entire library like this:

```
import 'rxjs/Rx';
```

Most tools won't be used and the library is big so it is best to just import the tools needed.

Adding operators makes use of a cool trick

- import 'rxjs/add/operator/map';

```
var Observable_1 = require('../../Observable');
var map_1 = require('../../operator/map');
Observable_1.Observable.prototype.map = map_1.map;
```

Observables with Http

- Now you know how Observables work.
- Let's see how to apply them to Angular and Http
- Because, let's face it ...

Your main use for observables will be to process Ajax responses

You'll want to convert your data from JSON

".map()" says to convert the http response based on the function passed to it

```
this._http
.get("http://us.com/persons/123")
.map(p => <Person> p.json())
.catch(err => console.error(err))
.subscribe(res => this.person = res)
```

".json()" says to parse the string into an object

"<Person>" says to cast the generic object as a Person

Remember, observables are lazy! Nothing happens until you subscribe.

tl;dr

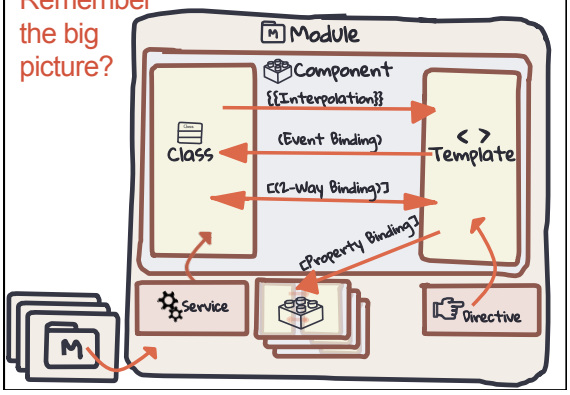
- Observables represent a stream of future values.
- They will run a function as data arrives instead of once at the end.
- You tell the observable what it is waiting on.
- Then you use subscribe to tell it what to do when the data is updated
- There are some really useful extension methods that must be added to the observable's prototype before they're used

Services

tl;dr

- Services are for sharing data or functionality across components.
- They must be added to the providers array of a module or component annotation and must be injected to be used
- Add them as low as possible but not so low they can't be shared
- Create services with ng generate service which will add the all-important @Injectable annotation

Remember the big picture?

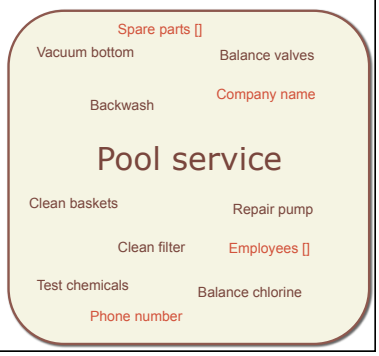


What's a service?

Nope! None of the above.

So what is a service then?

A service is a holder of objects and activities that would be useful to one or more external things.

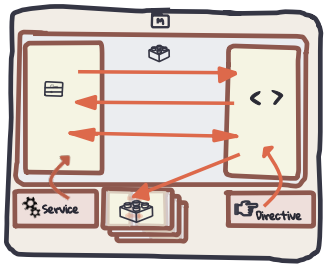


Angular services aren't what you think at first

Services are "*holders of wonderful things*"

- Data
- Functionality

- They are bundles of stuff that can be injected into components



Anything to be shared between components goes in a service

For example ...

Streaming a song
Calling Ajax services
Other business logic
Language translation
Getting the next item in a series
Authenticating users
Authorizing activities
Reading/writing database data
Fetching dynamic HTML
Finalizing an order
Validating a credit card
Calculating a price
Managing inventory
3rd party requests
So many more

Services should be view-agnostic

- They shouldn't know anything about any view
- They can't write to a view

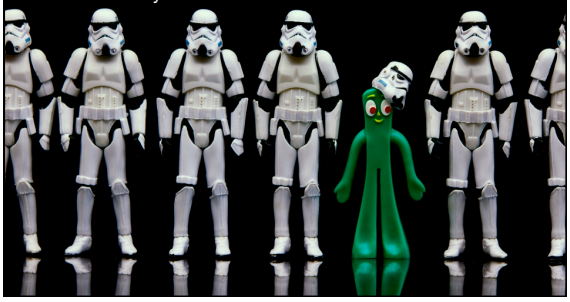
Why?

a. That's the class's job
b. Services should do ONE thing
c. Otherwise they couldn't work with multiple views

How to add a service to your component

Services break the pattern of inclusion

- There's no @Service decorator
- They're usually grouped with other things in a module, but you don't technically add them to a module



You add services in three steps ...

1. Import them

```
import { FooService } from '../shared/foo.service';
```

2. Register them in the providers list of the component or module

```
@Component([
  providers: [ FooService ]
}) ...
```

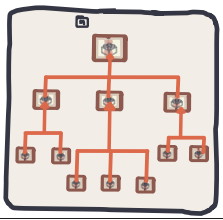
3. Use DI to inject them in the constructor

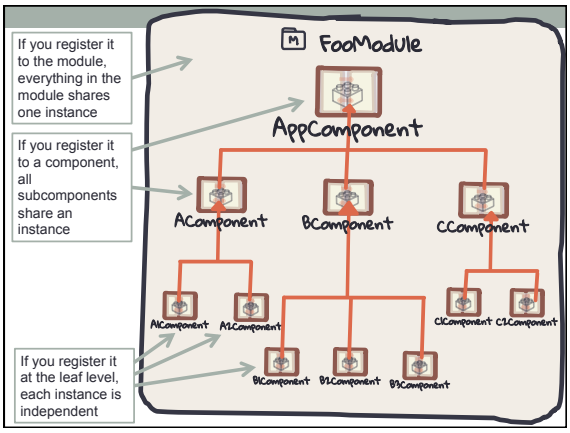
```
constructor(private _fooService: FooService) { ...
```

Not the only way to do it, but remember this nifty shorthand for injecting into the constructor.

Where you register them is very important!

- You have to register services in an annotation in the providers array
- For tight cohesion you want to register as low as possible.
- But if you get too low, you have to register two different times and thus it is two different instances
- If you register high enough, it's a singleton and can share data





How to write a service

Use the angular CLI, of course!

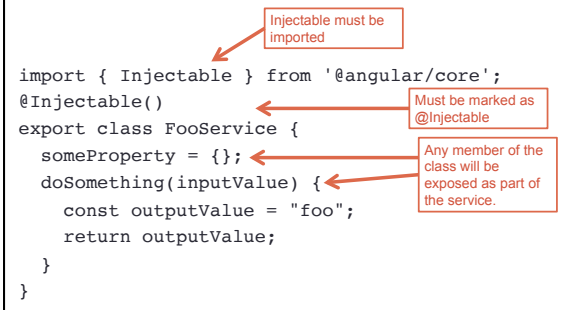
`ng generate service <serviceName>`

- Note:
  - ng generate service doesn't create a subdirectory
  - Don't ng generate *foo*Service, just *foo*
  - Doesn't affect the module at all (unlike other blueprints).

```
$ ng generate service foo
installing service
  create src/app/foo.service.spec.ts
  create src/app/foo.service.ts
  WARNING Service is generated but not provided, it must be provided to be used
$
```

Reinforcing that we need to inject it in the component or the module before it can be used!

And your service might look like this...




```
import { Injectable } from '@angular/core';
@Injectable()
export class FooService {
  someProperty = {};
  doSomething(inputValue) {
    const outputValue = "foo";
    return outputValue;
  }
}
```

tl;dr

- Services are for sharing data or functionality across components.
- They must be added to the providers array of a module or component annotation and must be injected to be used.
- Add them as low as possible but not so low they can't be shared
- Create services with ng generate service which will add the all-important @Injectable annotation

Pipes

tl;dr

- If you want to change the data presented in a view, Angular pipes are the answer
- Don't let them get too heavy -- put that stuff in a controller
- The built-in pipes (currency, number, date,slice, uppercase, lowercase, titlecase) change the appearance of a value
- You can build your own filters by marking a class with @Pipe and providing a method called transform.

Any Unix gurus out there?

Q: What would this Unix command do?

```
$ cat /etc/passwd | grep "ebrown" | head -1 | cut -f5 -d':'
```

A: Get Emmett's full name

4 processes actually run ...

1. Get all rows from the password file
2. Only let through rows that have "ebrown" in there
3. Only let through the first one
4. Extract the 5th field

- This is called *piping* in Unix

Without pipes

```
<ul class="list-group">
  <li *ngFor="let person of ppl">
    {{ person.firstName }} -
    {{ person.birthDate  }} -
    {{ person.salary }}
  </li>
</ul>
```

People

- Lorraine - 1954-05-20 - 82356.75
- Biff - 1954-10-04 - 41974
- Emmett - 1949-10-31 - 112879.95
- Jennifer - 1964-04-19 - 22100
- George - 1965-05-14 - 132384.94

With pipes

```
<ul class="list-group">
  <li *ngFor="let person of ppl | orderBy:'firstName'">
    {{ person.firstName }} -
    {{ person.birthDate | date }} -
    {{ person.salary | currency }}
  </li>
</ul>
```

People

- Biff - Oct 4, 1954 - $41,974.00
- Emmett - Oct 31, 1949 - $112,879.95
- George - May 14, 1965 - $132,384.94
- Jennifer - Apr 19, 1964 - $22,100.00
- Lorraine - May 20, 1954 - $82,356.75

Angular has some built-in pipes

- Work inside expressions like this:

```
{{ someExpression | pipeName:"OptionalParam" }}
```

Pipe	Description
currency	Puts a currency sign in front and sets two decimal places
date	Makes a date look like a date
lowercase	Converts all letters to lowercase
number	Rounds to decimal places
percent	Expresses a number as a percent
slice	Returns only a portion of a string or array
titlecase	Capitalizes words
uppercase	Converts them to uppercase

currency

- Formats the expression as money.

```
{{ number | currency : code : showSymbol }}
```

- code:
  - (optional)
  - An ISO 4217 currency code (USD, EUR, JPY, GBP, INR)
- showSymbol:
  - (optional)
  - A boolean. True=show the symbol ($, €, ¥, £, ₹). False = show code
- Examples

```
{{ 1443 | currency }}  // $1,443.00
{{ 43.75 | currency:"EUR":true }}  // €43.75
{{ 37.91 | currency:"EUR" }}  // EUR37.91
```

date

- Formats the expression as a date

```
{{ aDate| date : format : timezone }}
```

- format:
  - (optional)
  - How you want it to appear
- timezone:
  - (optional)
  - Which timezone (GMT|UTC|EDT|ET, etc., +hhmm)
- Examples

```
{{ aDate | date }}  // February 21, 2017
{{ aDate | date:"short" }}  // 02/21/2017 7:47PM
{{ aDate | date:"short":"GMT" }} // 02/22/2017 12:47AM
{{ aDate | date:"yyyyMMddHHmmss" }} // 20170221194759
```

Altering the case of strings

- Changes capital letters to lowercase or vice-versa

```
{{ string | uppercase/lowercase/titlecase }}
```

- Examples

```
{{ "Marty McFly" | uppercase}}  // MARTY MCFLY
{{ "Marty McFly" | lowercase}}  // marty mcfly
{{ "Marty McFly" | titlecase}}  // Marty Mcfly
```

number and percent

- Formats the number.

```
{{ aNumber | number: digitInfo }}
{{ aNumber | number: digitInfo }}
```

- digitInfo:
  - (optional)
  - Details on next page
- Examples

```
{{ 1442.75 | number }}  // 1,442.75
{{ 1442.75 | number:"1.3" }}  // 1,442.750
{{ 1442.75 | number:"1.0-0" }}  // 1,443
{{ 0.144275 | percent}}  // 14.428%
{{ 0.144275 | percent:"1.0-5"}}  // 14.4275%
{{ 0.144275 | percent:"1.5-15"}}  // 14.42750%
```

number and percent both use digitInfo

digitInfo is a string with this format:

`x.y-z`

Where ...

- x = minimum number of digits to the left of the decimal
- y = minimum number of digits to the right of the decimal
- z = maximum number of digits to the right of the decimal

slice

- Truncates the array to some number of members

```
{{ array_or_string | slice : start : end}}
```

start:

- At what position (zero-based) to begin

end:

- (optional)
- At what position to stop

- Examples

```
{{ products | slice:0:10 }}  // Top 10 products
{{ products | slice:50:60 }}  // Products 51-60
{{ "abcedfghijklm" | slice:0:5 }}  // abcde
{{ "abcedfghijklm" | slice:1:5 }}  // bcde
{{ "abcedfghijklm" | slice:5:6 }}  // f
{{ "abcedfghijklm" | slice:5 }}  // fghijklm
```

Angular no longer provides orderBy or filter

They were too slow in AngularJS 1.X and people blamed Angular itself.

I guess that'll teach us to complain.

npm install --save ng2-order-pipe

You can string pipes together

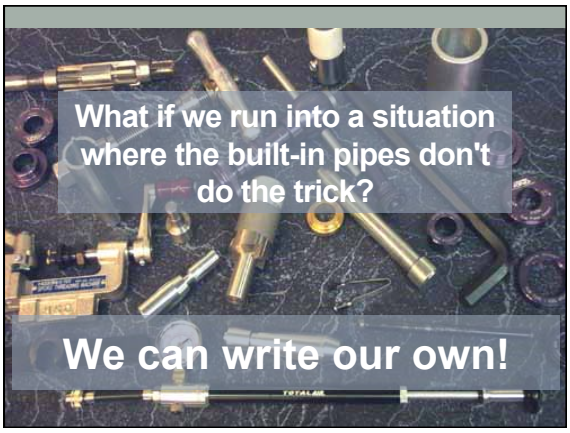
```
<div *ngFor=
'let person of people| slice:0:10 | orderBy:"lastName"'
>

<p>
Expected shipping date: {{ orderDate | addDays:10 | date }}
</p>
```

- Not that you *should*, but you *could*.
- Some would say that logic should be in the controller.

- But wait, what is that "addDays" thing?

A custom pipe!

Writing your own custom pipes



Pipe syntax

```
import { Pipe, PipeTransform } from "@angular/core";

@Pipe({
  name: "foo"
})
export class FooPipe implements PipeTransform {
  transform(value, optionalArgs) {
    // Do things to input value here
    return convertedValue;
  }
}
```

import Pipe and PipeTransform

Mark it as a Pipe

What the pipe will be called

Not strictly needed, but a good idea

Value going into the pipe

The value after processing

Example

```
@Pipe({
  name: "sortAlphabetically"
})
export class SortAlphaPipe implements PipeTransform {
  transform(array: Array<string>): Array<string> {
    array.sort((a, b) => {
      if (a < b)
        return -1;
      else if (a > b)
        return 1;
      else
        return 0;
    });
    return array;
  }
}
```

tl;dr

- If you want to change the data presented in a view, Angular pipes are the answer
- Don't let them get too heavy -- put that stuff in a controller
- The built-in pipes (currency, number, date,slice, uppercase, lowercase, titlecase) change the appearance of a value
- You can build your own filters by marking a class with @Pipe and providing a method called transform.

Modules in Angular

tl;dr

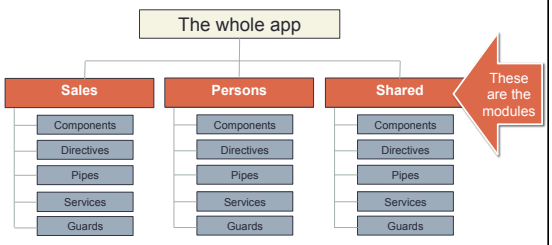
- This topic doesn't give us any new capabilities. The major draw is clean code but performance is also increased slightly.
- Modules encapsulate your app into features
- You create a new module with @NgModule() annotation which includes
  - declarations
  - providers
  - exports
  - imports
  - bootstrap

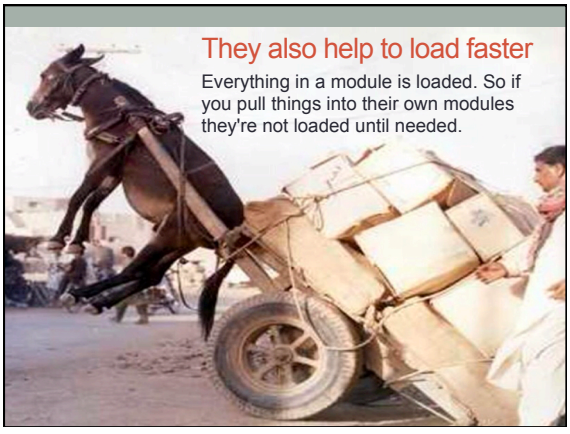
Modules simplify large apps

Like packages in Java
They allow each "feature" to focus on its own thing
The S in SOLID
Reduces the use of the global namespace
Reduce coupling and dependencies
Increases maintainability

Divide the apps by feature

Each feature becomes a module.





They also help to load faster

Everything in a module is loaded. So if you pull things into their own modules they're not loaded until needed.

CAUTION

JavaScript modules are not Angular modules!

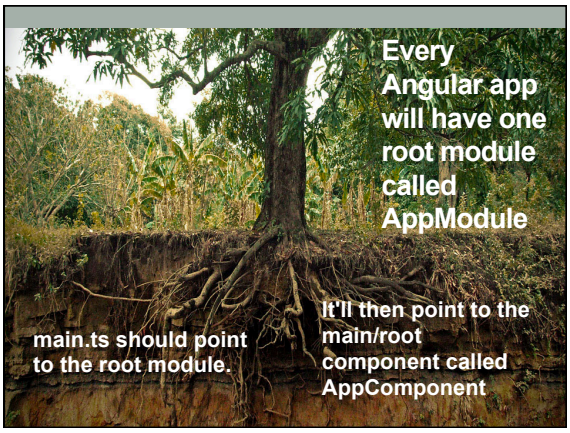
- We should have named them something different.
- BOTH are used in angular.

So what exactly are the differences in modules, imports and exports?

JavaScript modules	Angular modules
Could be any code file	An empty class decorated with @NgModule()
For encapsulating one file	A container for grouping components, directives, pipes, and services.
For encapsulating one file	Consists of many, many "files"

JavaScript export	Angular exports
Can be in every file with a module	Only in the NgModule() annotation
For fine-grained control of what objects are exposed	An array of everything you want to be used in other Angular modules
Can be any kind of object	Only Angular components, directives, pipes, and occasionally other modules

JavaScript import	Angular imports
Usually one line at the top of a file	Always in the @NgModule() annotation.
So we're not forced to bring in everything in another file	So we can see the declarables in the other module
Everything would be included if we didn't have an import	Nothing would be included if we didn't have an imports
You can import anything that was exported in another file	You imports only modules whether they were exported elsewhere or not



All other 'features' should have their own module

products.module.ts
sales.module.ts
customers.module.ts
authentication.module.ts

etc

Modules

- Per feature
  - Each section/feature/whatever gets a module
  - Keep the walls built between sections - cohesive components/etc.
  - But what if there are things that are common/reusable
    - A login component
    - A search component
    - A rating component
- Shared
  - Things needed across two or more other modules

How to create a module

A module is a class marked with @NgModule

Decorator/annotation

Empty exported class

main.module.ts

```
@NgModule({
 //Other stuff goes here
})
export class MainModule{}
```

And we fill the decorator with ...

- imports - Other modules
- declarations - The members of this module
- exports - Things here needed in other modules
- providers - Services we're providing
- bootstrap - Where do we begin?

imports

- Only other modules that have things we need in this module.
- Those other things must have been exported by that other module

declarations

- Only declarables can go here
  - components,
  - directives
  - pipes
  - (No modules or services)
- If you don't list it here it can't be used in this module. (This is done automatically as part of ng generate)
- Note: If a component, directive, or pipe belongs to a module in the imports array, DON'T re-declare it in the declarations array.

A troubleshooting tip that is module-related

- Components aren't compiled as part of the NG2 application until they're included in the declarations section.
- Until they are, they're just files floating out there with a .js extension.
- So if your project is working great until you add a component to the declarations section, the problem is how it plays with NG2 and with other components.

exports

- Things in this module that other modules will need
- A subset of declarations
- Mainly components, directives, and pipes.
- But if this module imports another module and we know it'll be used by a third module, we can exports any module and the third modules won't need to import them.

providers

- Services that this module contributes to the global collection of services; they become accessible in all parts of the app.
- Poorly named. Should have been "services"

bootstrap

- The main or root component that hosts all other components
- Only the root module should set this bootstrap property.

For example

app.module.ts

```
@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    routing],
  declarations: [
    AppComponent,
    DashboardComponent,
    HeroesComponent],
  exports: [
    DashboardComponent
  ],
  bootstrap: [ AppComponent ],
  providers: [ HeroService ]
})
export class AppModule{}
```

And when you start up a program ...

main.ts

```
platformBrowserDynamic().bootstrapModule(AppModule);
```

Literally the only executable line of code in the file.

Points to our boot module.

tl;dr

- This topic doesn't give us any new capabilities. The major draw is clean code but performance is also increased slightly.
- Modules encapsulate your app into features
- You create a new module with @NgModule() annotation which includes
  - declarations
  - providers
  - exports
  - imports
  - bootstrap