Angular 7

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.
- 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 about every 6 months in the spring and fall.	Minor releases every month	Patch releases every non-holiday week

7 . **3** . **1**

Major	**Minor**	**Patch**
Breaking change	*New features not breaking*	*Bug fixes 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 - November 2017
- Angular 6 - May 2018
- Angular 7 - October 2018

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

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 really are the advantages of Angular over AngularJS?

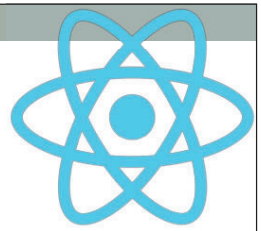
1. Faster to run
2. Components!

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



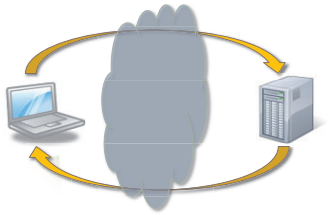
* Find more information on http://www.webcomponents.org/

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



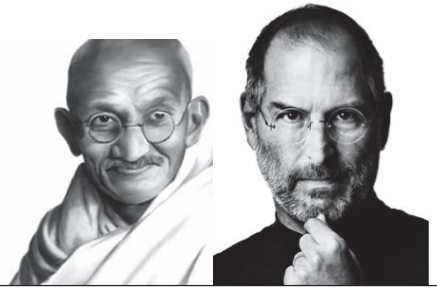
- Let's break this down on the next pages

Let's define "opinionated"

What does opinionated mean to you?

Do you like opinionated people?

Is there anything good about being opinionated?



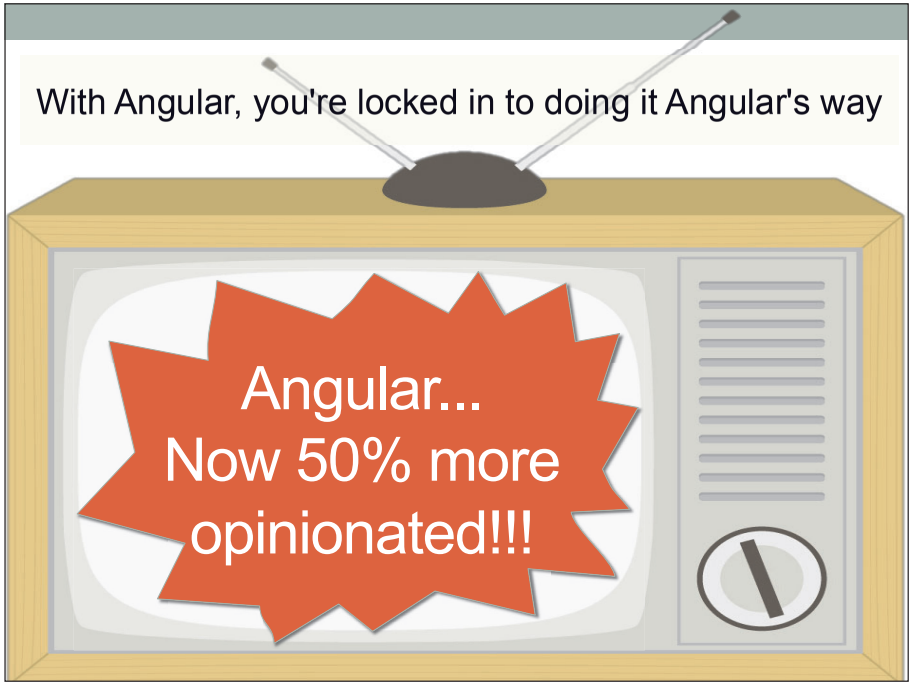
Angular is of the opinion that HTML isn't good enough

I need to teach HTML some new tricks.

The W3C doesn't make rules. They make suggestions.

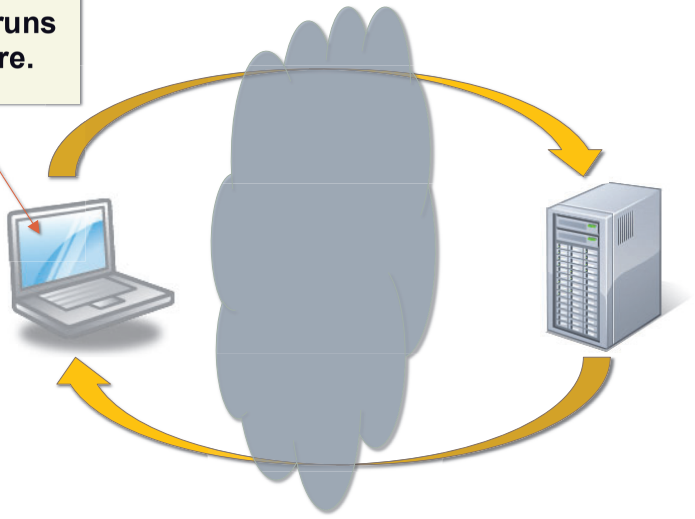
You should break their rules, too.

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



`client-side === in_the_browser`

Angular runs only here.



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)

@angular
@ngtools
@types
abbrev
accepts
acorn
acorn-dynamic-import
adm-zip
after
agent-base
ajv
ajv-keywords
align-text
alphanum-sort
amdefine
ansi-align
ansi-escapes
ansi-html
ansi-regex
ansi-styles
any-promise
anymatch
app-root-path
append-transform
aproba
are-we-there-yet
argparse
arr-diff
arr-flatten
array-find-index
array-flatten
array-slice
array-union
array-uniq
array-unique
arraybuffer.slice
arrify
asap
asn1
asn1.js
assert
assert-plus
async
async-each
async-foreach
asynckit
autoprefixer
aws-sign2
aws4
babel-code-frame
babel-generator
babel-messages
babel-runtime
babel-template
babel-traverse
babel-types
babylon
backo2
balanced-match
base64-arraybuffer
base64-js
base64id
batch
bcrypt-pbkdf
better-assert
big.js
binary-extensions
blob
block-stream
blocking-proxy
bluebird
bn.js
body-parser
bootbase
boom
boxen
brace-expansion
braces
brorand
browserify-aes
browserify-cipher
browserify-des
browserify-rsa
browserify-sign
browserify-zlib
browserslist
buffer
buffer-xor
builtin-modules
builtin-status-codes
bytes
callsite
camel-case
camelcase
camelcase-keys
caniuse-api
caniuse-db
capture-stack-trace
caseless
center-align
chalk
chokidar
cipher-base
clap
clean-css
cli-boxes
cli-cursor
cli-width
cliui
clone
clone-deep
co
coa
code-point-at
codelyzer
color
color-convert
color-name
color-string
colormin
colors
combine-lists
combined-stream
commander
common-tags
component-bind
component-emitter
component-inherit
compressible
compression
concat-map
configstore
connect
connect-history-api-fallback
console-browserify
console-control-strings
constants-browserify
content-disposition
content-type
convert-source-map
cookie
cookie-signature
core-js
core-util-is
cosmiconfig
create-ecdh
create-error-class
create-hash
create-hmac
cross-spawn
cross-spawn-async
cryptiles
crypto-browserify
crypto-random-string
css-color-names
css-loader
css-parse
css-select
css-select-base-adapter
css-what
cssauron
cssesc
csstype
currently-unhandled
custom-event
dashdash
date-now
debug
decamelize
deep-extend
default-require-extensions
defined
del
delayed-stream
delegates
denodeify
depd
des.js
destroy
detect-indent
detect-node
di
diff
diffie-hellman
directory-encoder
dom-converter
dom-serialize
dom-serializer
domain-browser
domelementtype
domhandler
domutils
dot-prop
duplexer3
ecc-jsbn
ee-first
electron-to-chromium
elliptic
ember-cli-normalize-entity-name
ember-cli-string-utils
emojis-list
encodeurl
engine.io
engine.io-client
engine.io-parser
enhanced-resolve
ensure-posix-path
ent
entities
errno
error-ex
escape-html
escape-string-regexp
esprima
esutils
etag
eventemitter3
events
eventsource
ev_bytestokey
execa
exit
expand-braces
expand-brackets
expand-range
exports-loader
express
extend
external-editor
extglob
extract-text-webpack-plugin
extsprintf
fastparse
faye-websocket
figures
file-loader
filename-regex
fileset
fill-range
finalhandler
find-up
findup-sync
flatten
for-in
for-own
forever-agent
form-data
forwarded
fresh
fs-access
fs-extra
fs.realpath
fsevents
fstream
function-bind
gauge
gaze
get-caller-file
get-stdin
get-stream
getpass
glob
glob-base
glob-parent
globals
globby
globule
got
graceful-fs
graceful-readlink
handle-thing
handlebars
har-schema
har-validator
has
has-ansi
has-binary
has-cors
has-flag
has-unicode
hash-base
hash.js
hawk
he
hmac-drbg
hook
hosted-git-info
hpack.js
html-comment-regex
html-entities
html-minifier
html-webpack-plugin
htmlparser2
http-deceiver
http-errors
http-proxy
http-proxy-middleware
http-signature
https-browserify
https-proxy-agent
iconv-lite
ics-replace-symbols
ics-utils
ieee754
image-size
img-stats
import-lazy
imurmurhash
in-publish
indent-string
indexes-of
indexof
inflection
inflight
inherits
ini
inquirer
interpret
invariant
invert-kv
ipaddr.js
is-absolute-url
is-arrayish
is-binary-path
is-buffer
is-builtin-module
is-directory
is-dotfile
is-equal-shallow
is-extendable
is-extglob
is-finite
is-fullwidth-code-point
is-glob
is-npm
is-number
is-obj
is-path-cwd
is-path-in-cwd
is-path-inside
is-plain-obj
is-plain-object
is-posix-bracket
is-primitive
is-promise
is-redirect
is-retry-allowed
is-stream
is-svg
is-typedarray
is-utf8
isarray
isbinaryfile
isexe
isobject
isstream
istanbul-api
istanbul-instrumenter-loader
istanbul-lib-coverage
istanbul-lib-hook
istanbul-lib-instrument
istanbul-lib-report
istanbul-lib-source-maps
istanbul-reports
jasmine
jasmine-core
jasmine-spec-reporter
jasminewd2
js-base64
js-tokens
js-yaml
jsbn
jschardet
jsesc
json-loader
json-schema
json-stable-stringify
json-stringify-safe
json3
json5
jsonfile
jsonify
kassert
kind-of
lazy-cache
lcid
lcov-parse
lie
load-json-file
loader-runner
loader-utils
locate-path
lodash
lodash.assign
lodash.camelcase
lodash.clonedeep
lodash.mergewith
lodash.tail
lodash.template
lodash.templatesettings
log-symbols
longest
loose-envify
lower-case
lru-cache
make-dir
map-obj
map-stream
math-expression-evaluator
media-typer
memory-fs
meow
merge-descriptors
methods
micromatch
mime
mime-db
mime-types
mimic-fn
minimalistic-assert
minimalistic-crypto-utils
minimatch
minimist
mixin-object
mkdirp
ms
mute-stream
nan
ncname
negotiator
no-case
node-gyp
node-libs-browser
node-modules-path
node-sass
nopt
normalize-package-data
normalize-path
normalize-range
normalize-url
npm-run-path
npmlog
nth-check
null-check
num2fraction
number-is-nan
oauth-sign
object-assign
object-component
object.omit
obuf
on-finished
on-headers
once
onetime
opn
optimist
options
original
os-browserify
os-homedir
os-locale
os-tmpdir
osenv
package-json
pako
param-case
parse-asn1
parse-glob
path-key
path-parse
path-to-regexp
path-type
pbkdf2
performance-now
pify
pinkie
pinkie-promise
portfinder
postcss
postcss-calc
postcss-colormin
postcss-convert-values
postcss-discard-comments
postcss-discard-duplicates
postcss-discard-empty
postcss-discard-overridden
postcss-discard-unused
postcss-filter-plugins
postcss-load-config
postcss-load-options
postcss-load-plugins
postcss-loader
postcss-merge-idents
postcss-merge-longhand
postcss-merge-rules
postcss-message-helpers
postcss-minify-font-values
postcss-minify-gradients
postcss-minify-params
postcss-minify-selectors
postcss-modules-extract-imports
postcss-modules-local-by-default
postcss-modules-scope
postcss-modules-values
postcss-normalize-charset
postcss-normalize-url
postcss-ordered-values
postcss-reduce-idents
postcss-reduce-initial
postcss-reduce-transforms
postcss-selector-parser
postcss-svgo
postcss-unique-selectors
postcss-url
postcss-value-parser
postcss-zindex
prepend-http
preserve
pretty-error
process
process-nextick-args
promise
protractor
proxy-addr
prr
pseudomap
public-encrypt
punycode
q
qjobs
qs
query-string
querystring
querystring-es3
querystringify
randomatic
randombytes
range-parser
raw-body
raw-loader
rc
read-pkg
read-pkg-up
readable-stream
readdirp
redent
reduce-css-calc
reduce-function-call
reflect-metadata
regenerate
regenerator-runtime
regex-cache
regexpu-core
registry-auth-token
registry-url
regjsgen
regjsparser
script-loader
scss-tokenizer
select-hose
selenium-webdriver
semver
semver-diff
semver-dsl
send
wordwrap
wrap-ansi
wrappy
write-file-atomic
xdg-basedir
xml-char-classes
xmi2js
xmlbuilder
xmldom
xmlhttprequest-ssl
xtend
y18n
yallist
yargs
yargs-parser
yeast
yn
zone.js

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.
- 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/
win-x64/fse.node" already installed
```

```
    └─┬ 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

Create a new NG Application

```
$ ng new coolApp
? Would you like to add Angular routing? No
? Which stylesheet format would you like to use? CSS
CREATE coolApp/README.md (1024 bytes)
CREATE coolApp/angular.json (3777 bytes)
CREATE coolApp/package.json (1315 bytes)
CREATE coolApp/tsconfig.json (408 bytes)
CREATE coolApp/tslint.json (2837 bytes)
CREATE coolApp/.editorconfig (245 bytes)
CREATE coolApp/.gitignore (503 bytes)
CREATE coolApp/src/favicon.ico (5430 bytes)
CREATE coolApp/src/index.html (294 bytes)
CREATE coolApp/src/main.ts (372 bytes)
CREATE coolApp/src/polyfills.ts (3234 bytes)
CREATE coolApp/src/test.ts (642 bytes)
CREATE coolApp/src/styles.css (80 bytes)
CREATE coolApp/src/browserslist (388 bytes)
CREATE coolApp/src/karma.conf.js (964 bytes)
CREATE coolApp/src/tsconfig.app.json (166 bytes)
CREATE coolApp/src/tsconfig.spec.json (256 bytes)
CREATE coolApp/src/tslint.json (314 bytes)
CREATE coolApp/src/assets/.gitkeep (0 bytes)
CREATE coolApp/src/environments/environment.prod.ts (51 bytes)
CREATE coolApp/src/environments/environment.ts (662 bytes)
CREATE coolApp/src/app/app.module.ts (314 bytes)
CREATE coolApp/src/app/app.component.css (0 bytes)
CREATE coolApp/src/app/app.component.html (1141 bytes)
CREATE coolApp/src/app/app.component.spec.ts (981 bytes)
CREATE coolApp/src/app/app.component.ts (211 bytes)
CREATE coolApp/e2e/protractor.conf.js (752 bytes)
CREATE coolApp/e2e/tsconfig.e2e.json (213 bytes)
CREATE coolApp/e2e/src/app.e2e-spec.ts (303 bytes)
CREATE coolApp/e2e/src/app.po.ts (208 bytes)
● ● ●
found 0 vulnerabilities

    Successfully initialized git.
$ cd coolApp
$ ls
README.md              node_modules           src
angular.json           package-lock.json      tsconfig.json
e2e                    package.json           tslint.json
$
```

`ng new <appName>`

- Takes a long time

You can scaffold components

- Creates all files and changes the module (unless you --dry-run)
- 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 PeopleList
CREATE src/app/people-list/people-list.component.css (0 bytes)
CREATE src/app/people-list/people-list.component.html (30 bytes)
CREATE src/app/people-list/people-list.component.spec.ts (657 bytes)
CREATE src/app/people-list/people-list.component.ts (288 bytes)
UPDATE src/app/app.module.ts (496 bytes)
$
```

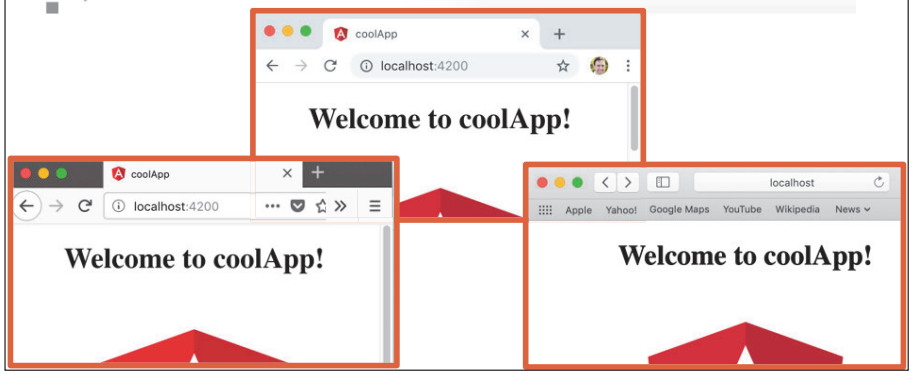
... and other things as well

```
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 run from a node server with *ng serve*

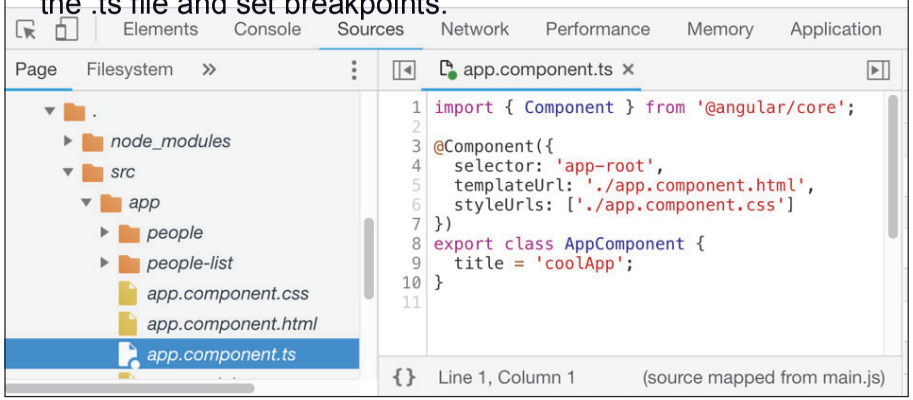
```
$ ng serve
** NG Live Development Server is running on http://localhost:4200.
Hash: 386fbce10750c48ef658
chunk    {0} main.bundle.js, main.bundle.map (main) 4.82 kB {2} [in
chunk    {1} styles.bundle.js, styles.bundle.map (styles) 9.99 kB {
chunk    {2} vendor.bundle.js, vendor.bundle.map (vendor) 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.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
  - ... in a way that allows debugging TypeScript
- Publishes a websocket so that it can push a page when we update.

- TypeScript can be tough to debug -- the code that runs is not the code you wrote (more on that later).
- ng serve brings up the browser in a mode that allows debugging by serving lots of extra files.
- To debug under 'sources', look for the 'webpack' folder and under that, find your source's directory structure. Then find the .ts file and set breakpoints.



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 components. No API data.

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

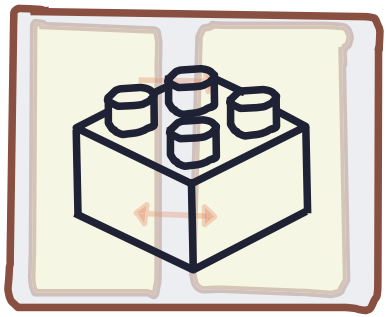
The Big Picture

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",
  ...
}
```

If we're going to write our own components, we need to give them attributes, properties, methods, and events!

Attributes are only in HTML ...

Properties, methods, and events are only in a JavaScript object

So ... an Angular component must be made up of an HTML element and a JavaScript object!

From the official NG documentation ...

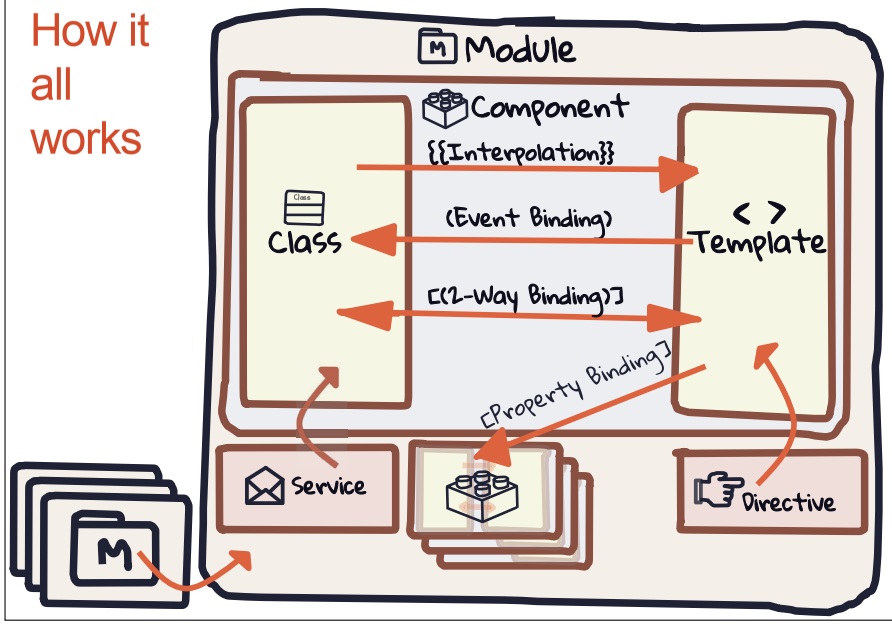
You write Angular applications by composing HTML templates with Angularized markup, writing component classes to manage those templates, adding application logic in services, and boxing components and services in modules.

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!

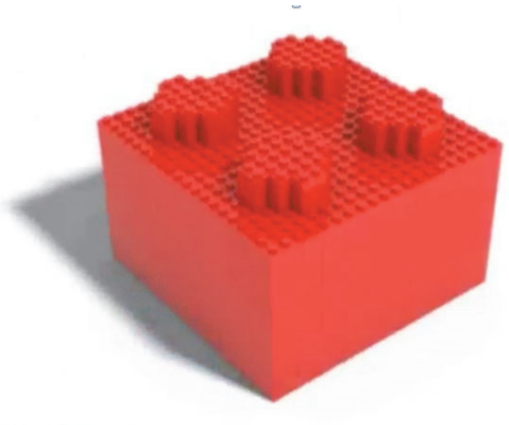
Components

- 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 ...

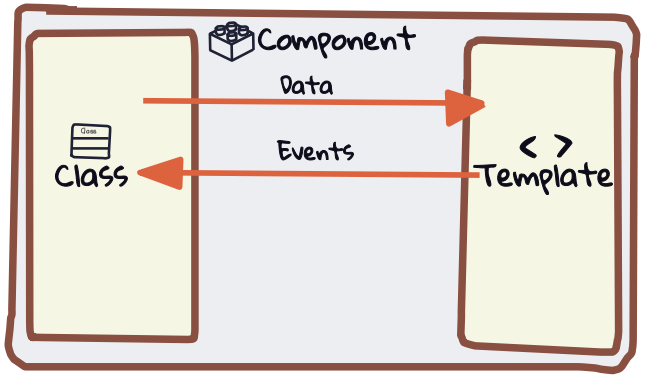
1. Identity
2. Behavior

So what's missing?

3. Presentation

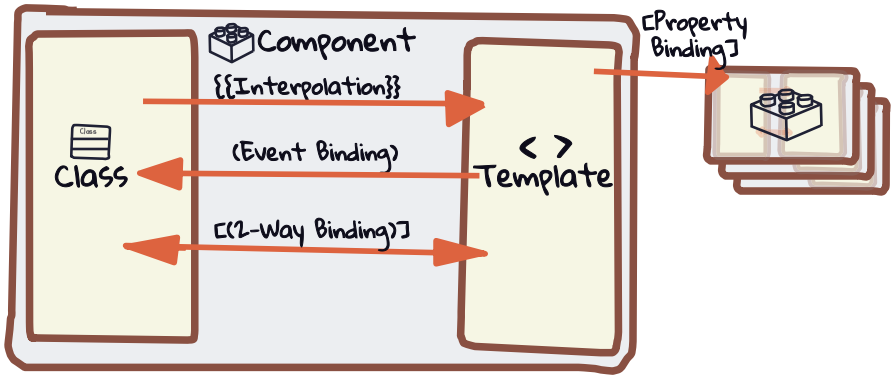
Components use templates for presentation

- 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 ...

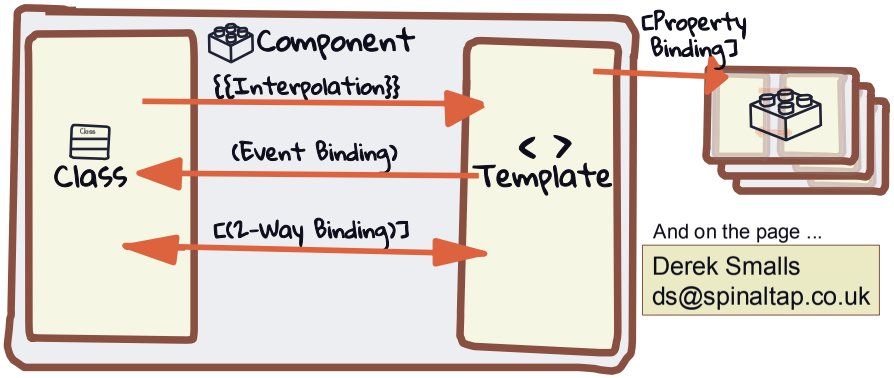


Interpolation is one-way data binding

- 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>
```



And on the page ...

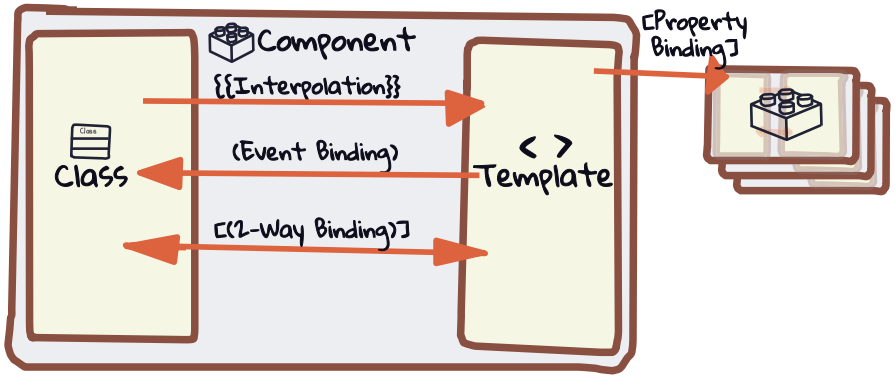
Derek Smalls
ds@spinaltap.co.uk

Property binding

- Passing business logic data to components via attributes

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

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

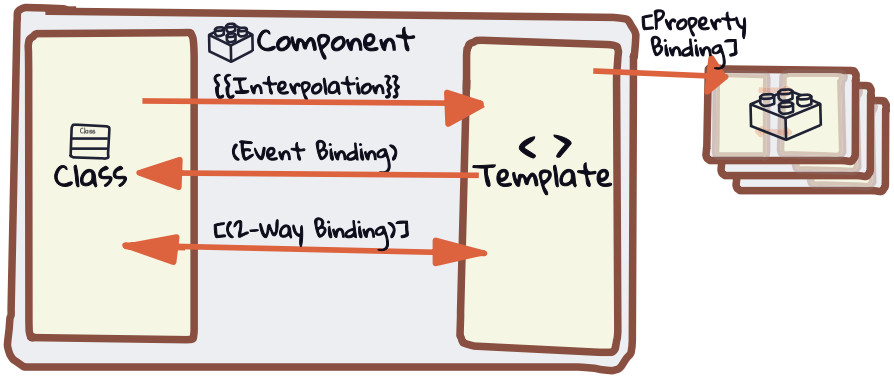


Event binding

- Wiring up a template event to a business logic method

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

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

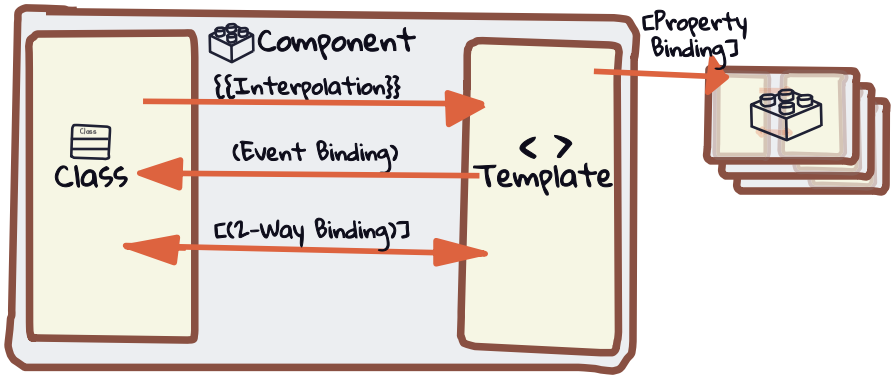


Two-way data binding

- Only makes sense for form fields

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

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

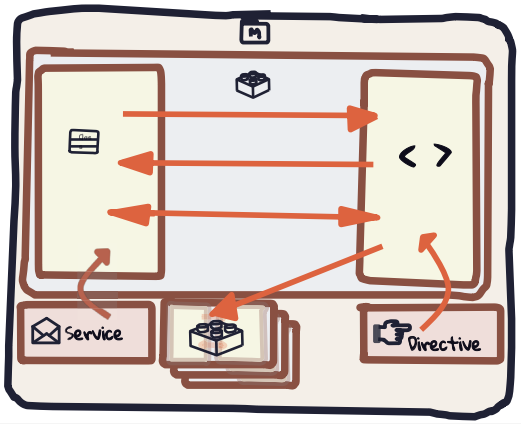


Directives

- 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



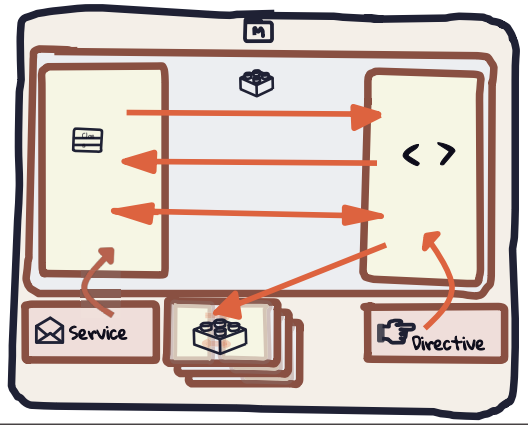
Services aren't what you think at first

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="main.js"></script>
...
```

main.ts

```
import { platformBrowserDynamic } from '@angular/core';
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({
templateUrl:`app.component.html`
})
export class AppComponent {}
```

app.component.html

```
<p>Hello world</p>
```

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 TS into ES5 by tsc
- TS allows us to use modern ES features like modules, arrow functions, and classes -- even in super-old browsers!
- And it adds proprietary OO features like public/private class modifiers, strong type checking, decorators, and constructor shorthands.
- But it causes friction in setup, development, and debugging

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

The best features of ES2016

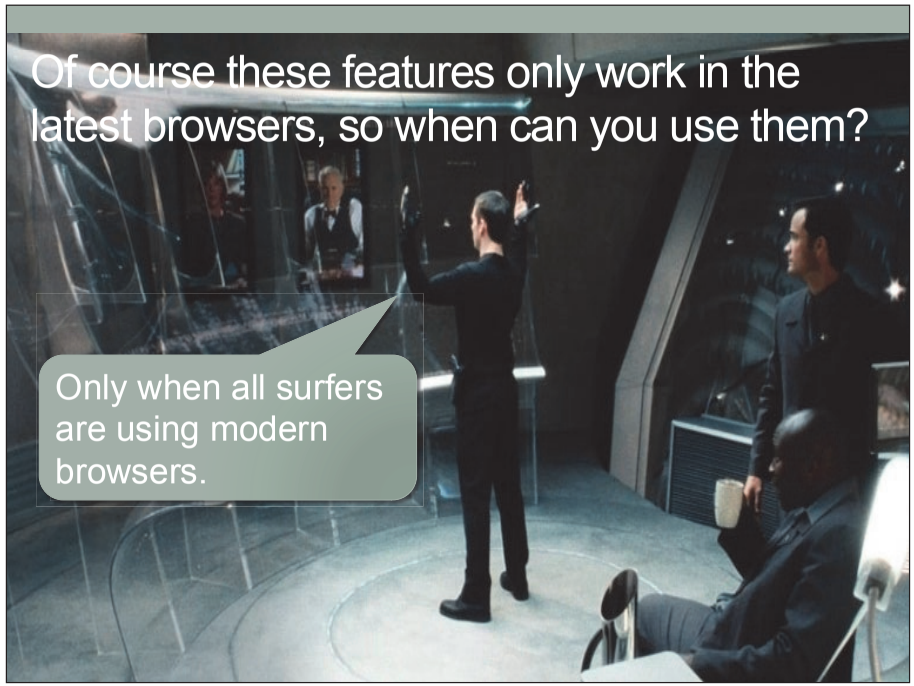
	Edge	Firefox	Chrome	Safari	Opera
Exponentiation Operator	14+	52+	54+	10.1+	43+
Array.prototype.includes	14+	40+	54+	10+	43+

The best features of ES2017

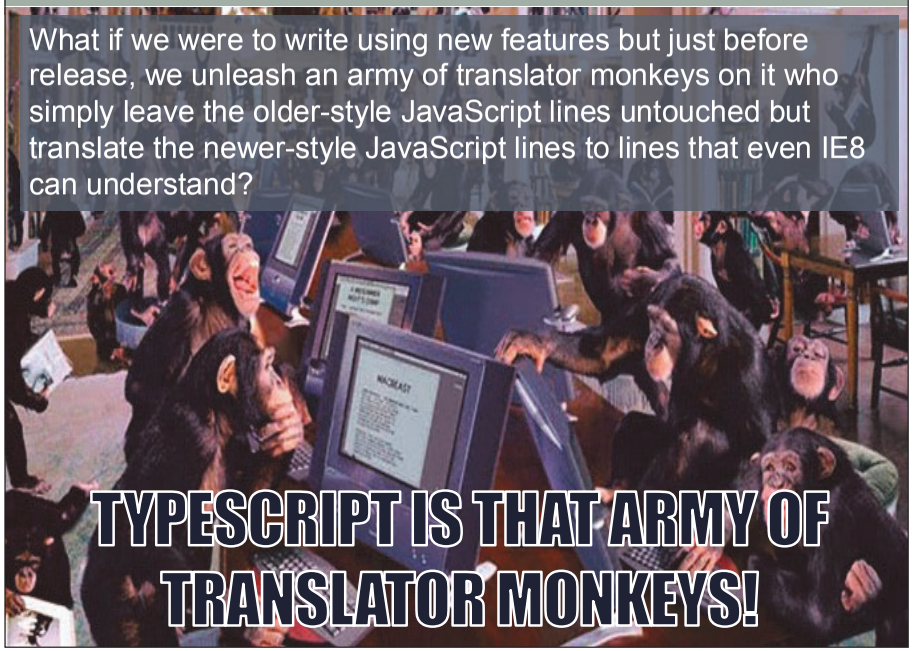
	Edge	Firefox	Chrome	Safari	Opera
Object static methods	15	51	56	10.1	43
String padding	15	51	57	10	44
Trailing commas in function	14	52	58	10	45
Async functions	15	52	56	10.1	43
Atomics	15	52	60	10.1	47

The best features of ES2018

	Edge	Firefox	Chrome	Safari	Opera
Object spread	15	51	56	10.1	43
Asynchronous iteration	15	51	57	10	44
Promise.prototype.finally	14	52	58	10	45
Tagged template literal	15	52	56	10.1	43



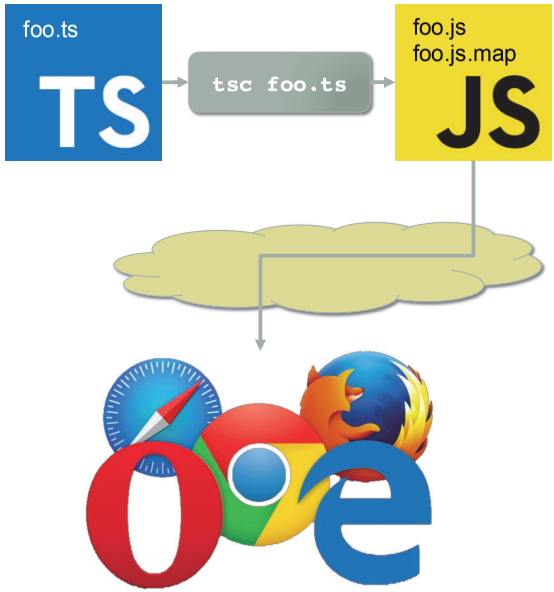
What if we were to write using new features but just before release, we unleash an army of translator monkeys on it who simply leave the older-style JavaScript lines untouched but translate the newer-style JavaScript lines to lines that even IE8 can understand?



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 • Classes	• Public/private members • Static typing • Constructor shorthand • Class and member decorators

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

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.
```

main.js

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

For example ...

car.js

```
export class Car {
 // Lots more stuff in here
}
```

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;
```

JavaScript "Classes"

ES2015 gives us a way to write classes

```
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

Public and private members

With TypeScript, you can make variables private

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

These can only be enforced at transpile time.

TS

Static typing

We can write in a statically-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.
```

Over the last several years, TypeScript has been coming on strong with the idea that static types get out of your way and provide deep benefits to code bases ..., while allowing the flexibility of dynamic types when needed.

Joel Hooks, developer and blogger

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

aka Annotations

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{
  ...
}
```

Constructor shorthand

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

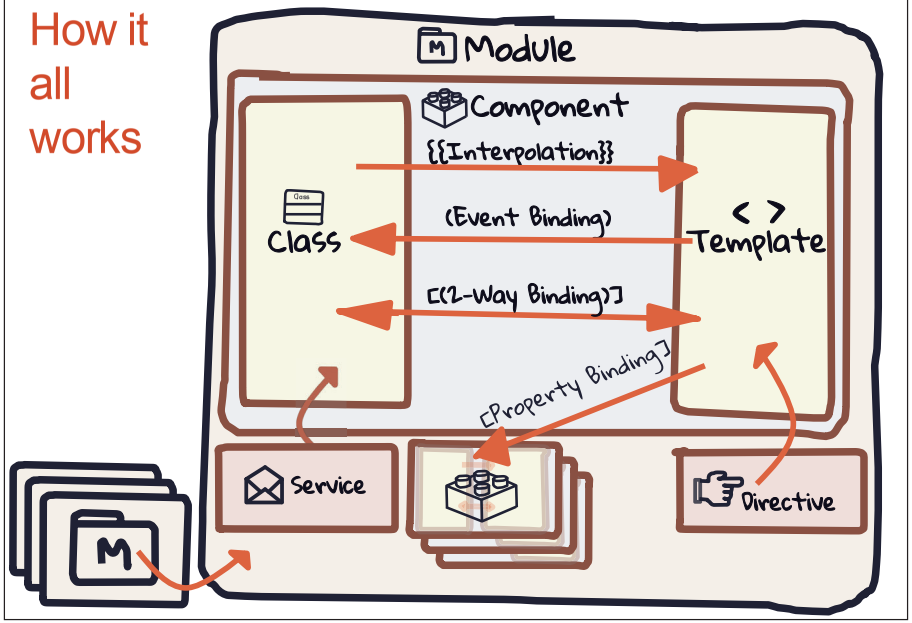
- Angular effectively requires us to use TypeScript (TS)
- Your code is transpiled from TS into ES5 by tsc
- TS allows us to use modern ES features like modules, arrow functions, and classes -- even in super-old browsers!
- And it adds proprietary OO features like public/private class modifiers, strong 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 }}
- Styling components can be done at three levels:
  - inside (styles)
  - near (styleUrls)
  - sitewide (stylesheets)

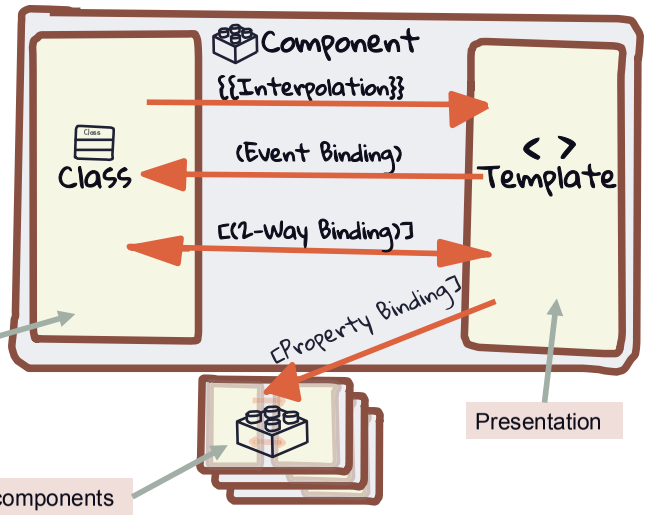
How it all works



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
C:	<my ... >

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

Template References

A DOM element can be seen by another if you give it a name.

Template references (like *#foo*) allow one DOM element to see another

- Examples:

```
<input #phone />
<button (click)="go(phone.value)">Go</button>
```

Careful! There's a hash in the definition, but not in the usage.

Template Expressions

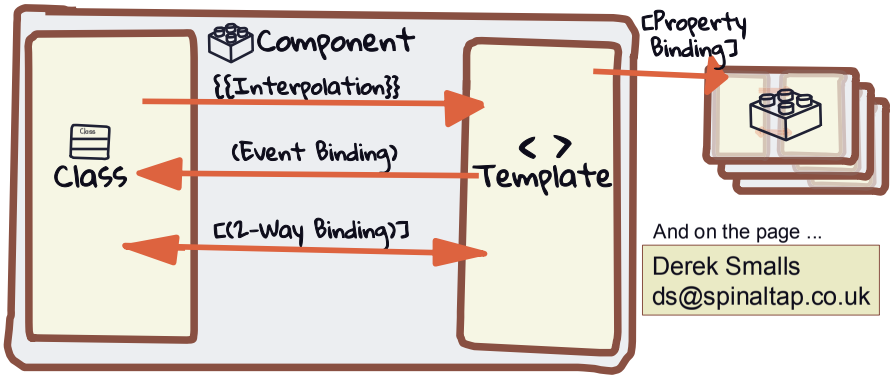
Getting stuff from the class to the template

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>
```



And on the page ...

Derek Smalls
ds@spinaltap.co.uk

But there's much more you can do besides put a simple string in there

- The text between the mustaches is a "template expression" that NG evaluates and "toString()"s
- So {{ 10 + getANumber() }} would be legal.

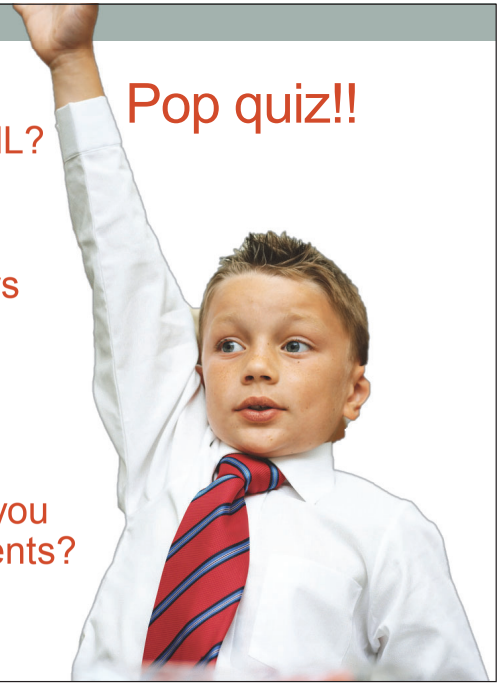
Styling a component

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',
  templateUrl: 'foo-bar.html'
})
export class FooBarComponent {}
```

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

Styling **near** a component

```
@Component({
  selector: 'foo-bar',
  templateUrl: 'fooBar.html',
  styleUrls: ['app/foo.css','app/bar.css']
})
export class FooBarComponent {}
```

Stylesheets are close to the component

Square brackets means an array: You can have multiple styles

Styling **within** a component

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

Styles are encapsulated within 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.

tl;dr

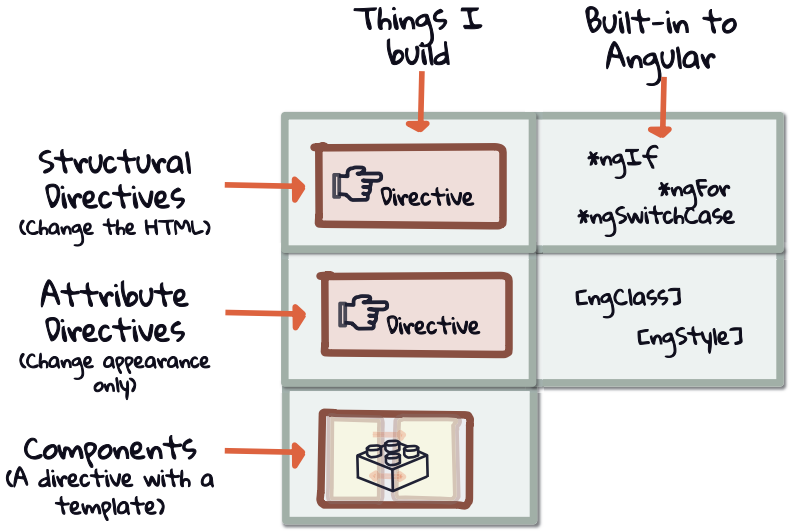
- Components are the central idea of Angular
- They have business logic in a TypeScript class and presentation in a template
- That class must be decorated with @Component and the template and styles are inside the decorator's object literal
- We move data from the logic to the presentation via {{ interpolation }}
- Styling components can be done at three levels:
  - inside (styles)
  - near (styleUrls)
  - sitewide (stylesheets)

Built-in Directives

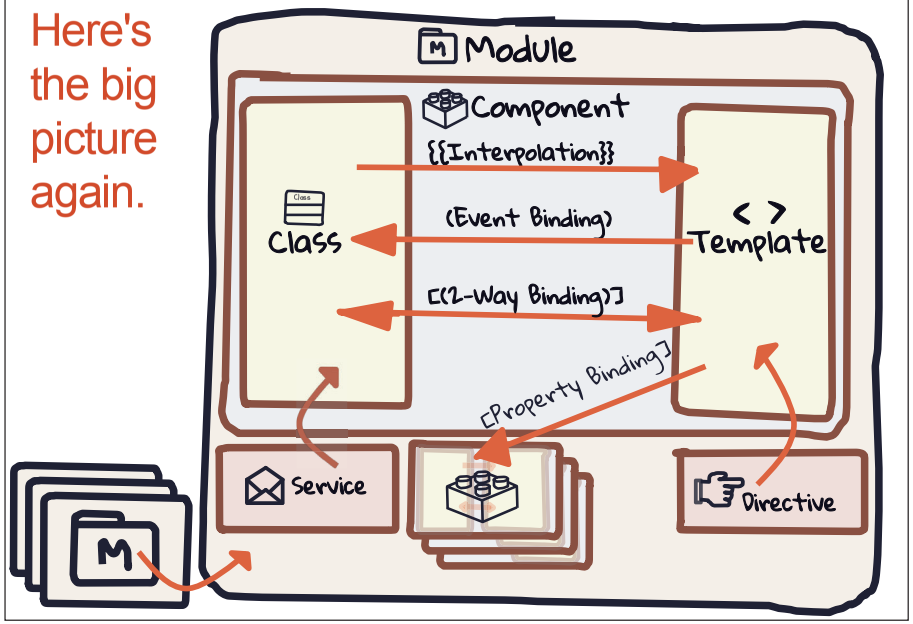
tl;dr

- 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.

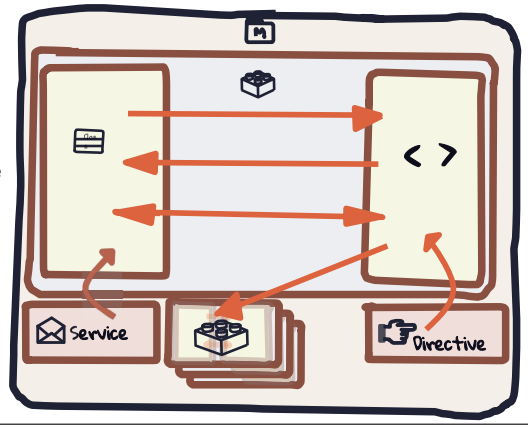


Directives

- 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]

When you want to set multiple (styles|classes) of an element programmatically

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

foo.component.ts

```
// Set properties
this.lowPad=10;
this.highPad = 20;
```

foo.component.html

```
<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]="bigOrFancy"></div>
```

- Multiple classes could be done like this:

```
<div [class.big]="someTruthyValue"
     [class.fancy]="someOtherTruthyValue">
```

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

foo.component.ts

```
// Set properties
this.someTruthyValue = true;
this.someOtherTruthyValue = 42;
```

foo.component.html

```
<div [ngClass]="{
  'big': someTruthyValue,
  'fancy': someOtherTruthyValue
  }">
  ...
</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

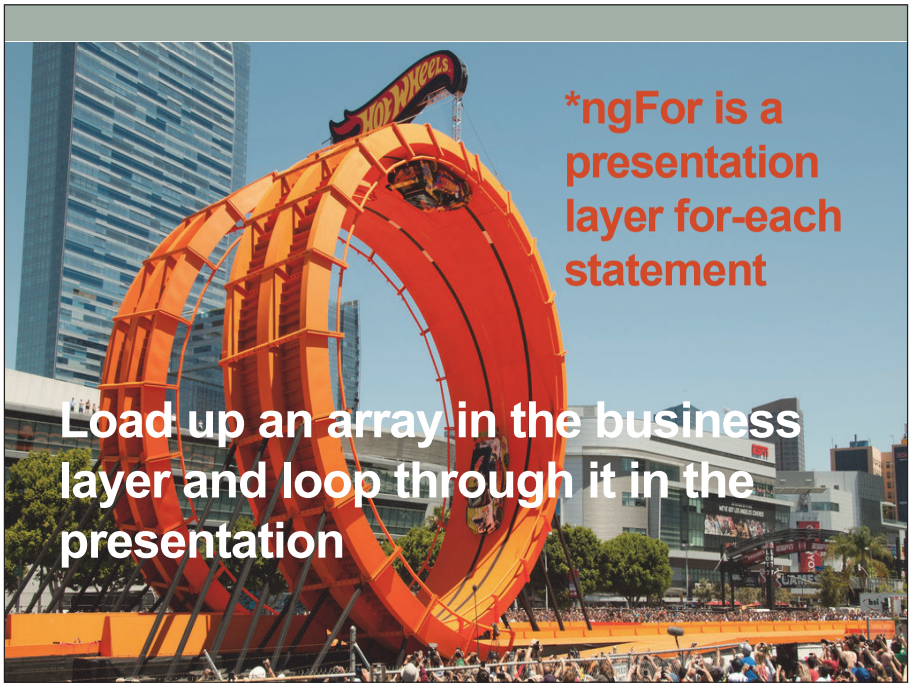
- When you get a lot of if statements against the same value it may help to refactor to an *ngSwitchCase
- *ngSwitchCase is worthless without a property binding to [ngSwitch].

For example ...

this.company = "Marvel";

```
<div [ngSwitch]="company">
 <span *ngSwitchCase="DC">DC is awesome!</span>
 <span *ngSwitchCase="Marvel">Marvel rules!</span>
 <span *ngSwitchCase="DarkHorse">Dark Horse</span>
 <span *ngSwitchDefault>All publishers</span>
</div>
```

*ngFor



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

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

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'}
];
```

flight
heat vision
talking to fish
✓ strength
invulnerability
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

*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

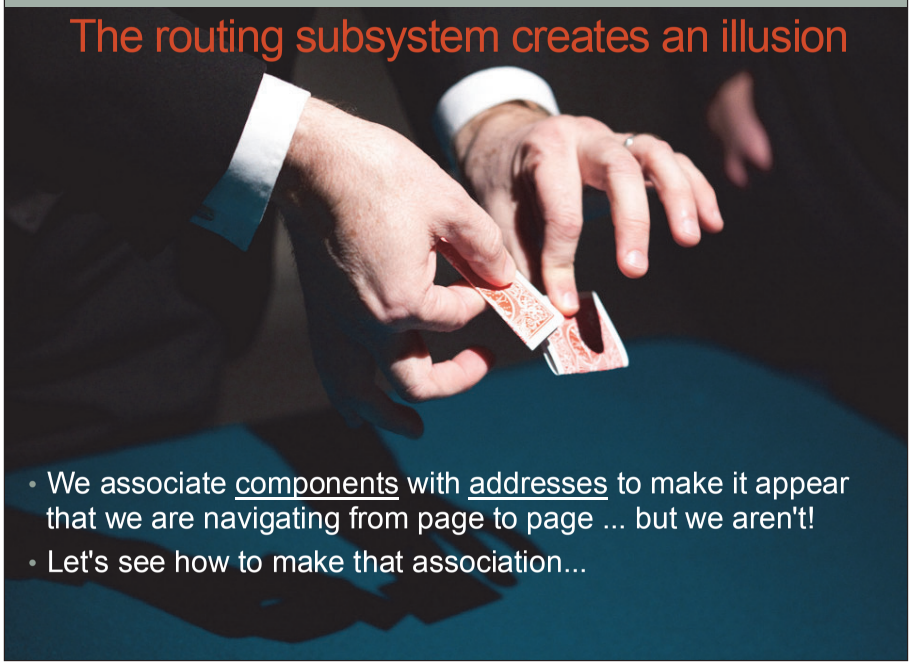
- You can have multiple pages if you want, but you have to work to make that happen....
- Including work on the server!

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

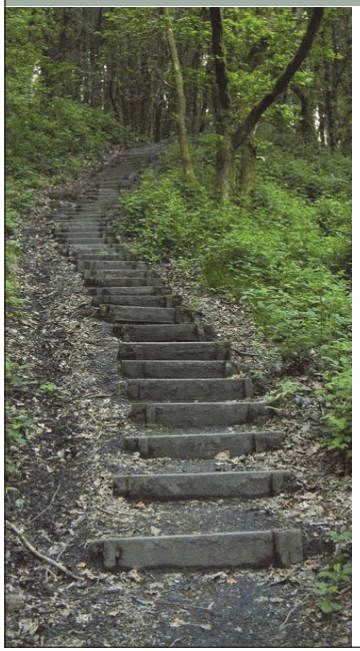
- Most browsers don't understand components (yet) so they need a page.
- The main component holds the *chrome* for the page

***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	FourOhFourComponent

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

... 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).

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

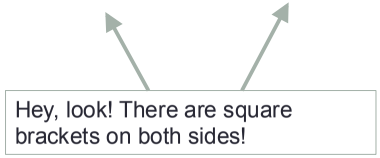
1) They can type in a URL

- 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.

2) 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

3) 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 ...

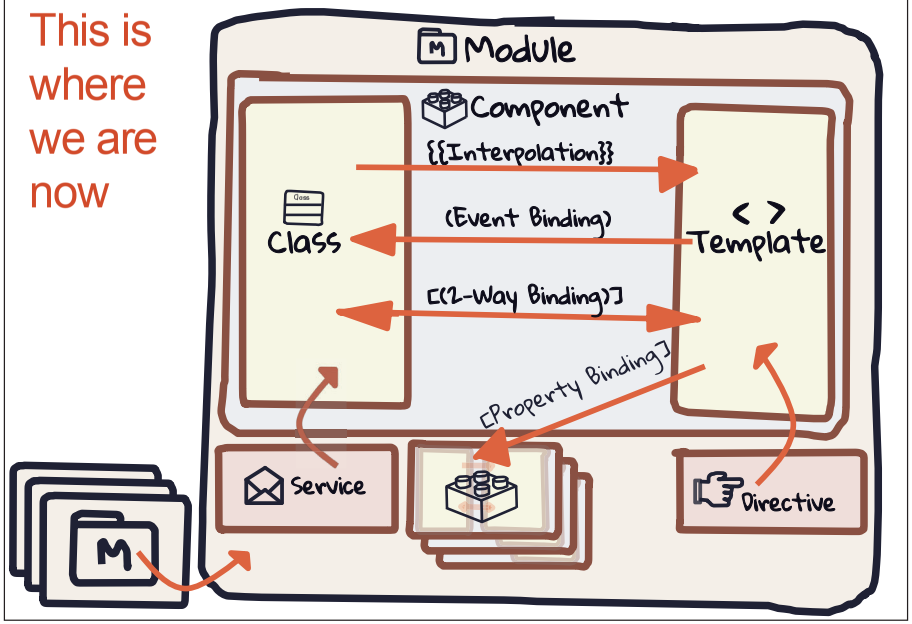
1. Create a <router-outlet>
2. Expose the routes to a module
3. Allow the user to navigate to a route by
   - Linking (<a [routerLink]="['foo']">)
   - Typing in a URL directly (routed by Angular - slow)
   - Pushing in a component (router.navigate())
4. Use route parameters in the component via activatedroute.snapshot.params[]

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()" />
```

You reference $event in the HTML to see the event object

foo.component.ts

```
do(e,p1) {
 let loc=`You're at
  ${e.clientX},
  ${e.clientY}`;
 console.log(loc);
}
```

foo.component.html

```
<button (click)="do($event,arg1)">Go</button>
<img src="p.jpg" (mouseover)="do($event,f)" />
```

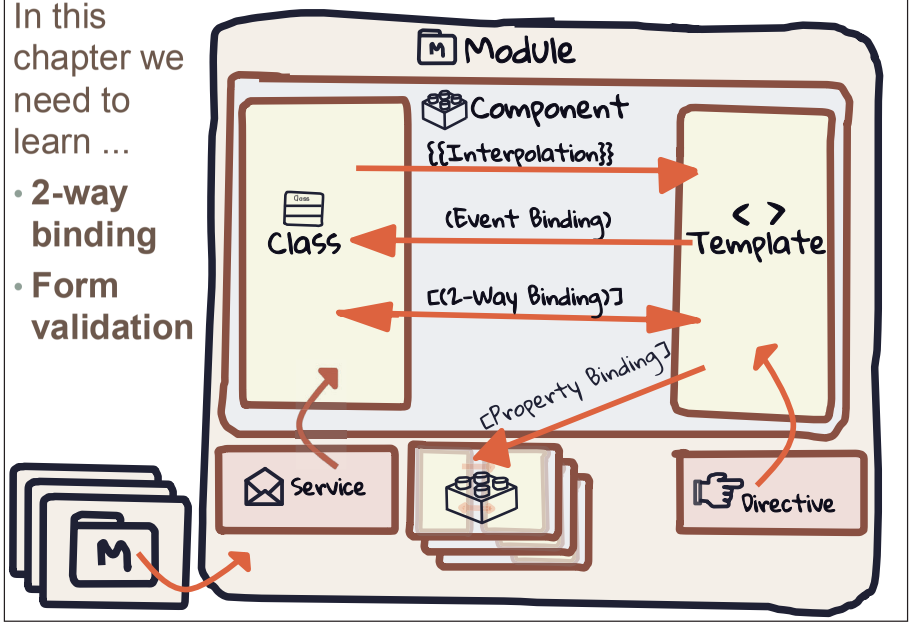
tl;dr

- 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 property
```

Two models of forms

Template-driven

- In the template only
- More declarative
- More intuitive
- Self-documenting

Reactive

- (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

head.component.ts

```
@Component({})
export class HeadComponent {
  familyName:String = "";
}
```

head.component.html

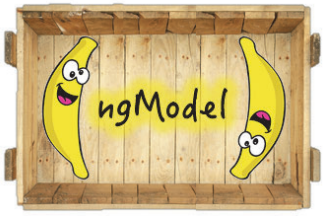
```
<input [(ngModel)]="familyName" />
```

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 template reference!

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

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

- For example:

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

- The view now knows this form as *formTR*

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

- The view now knows this input as *firstNameTR*

So how do we use these template references? 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?

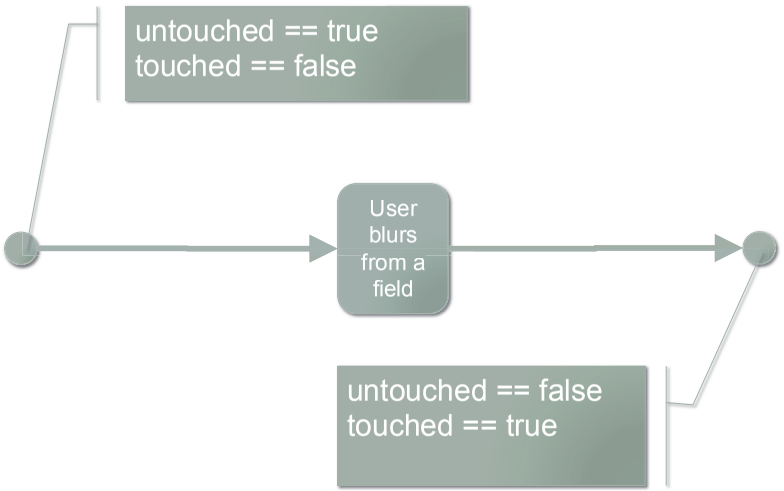
Angular applies status tokens to the fields

touched and untouched

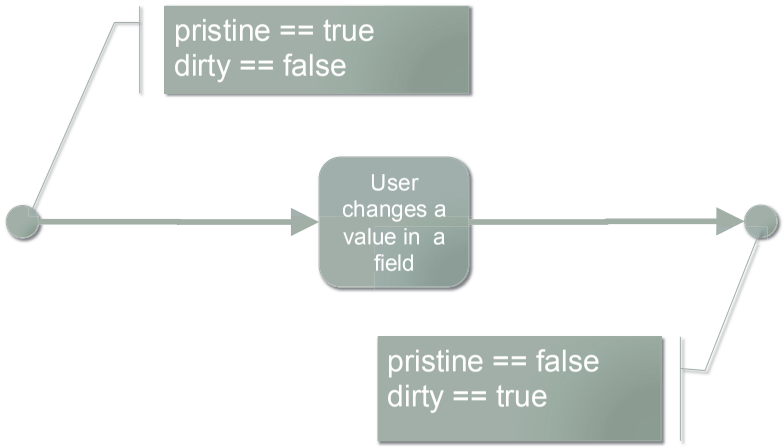
pristine and dirty

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" #firstTR="ngModel"
         [(ngModel)]="first" required pattern="\w+" />
  <span *ngIf="firstTR.touched && firstTR.invalid">
    *
  </span>
  <input name="last" #lastTR="ngModel"
         [(ngModel)]="last" *ngIf="firstTR.dirty" />
  <span *ngIf="lastTR.touched && lastTR.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" #cardTR="ngModel"
 required pattern="(\d{4}[ -]?){4}" />

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

<div *ngIf="cardTR.touched && cardTR.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 (#ref="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

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.

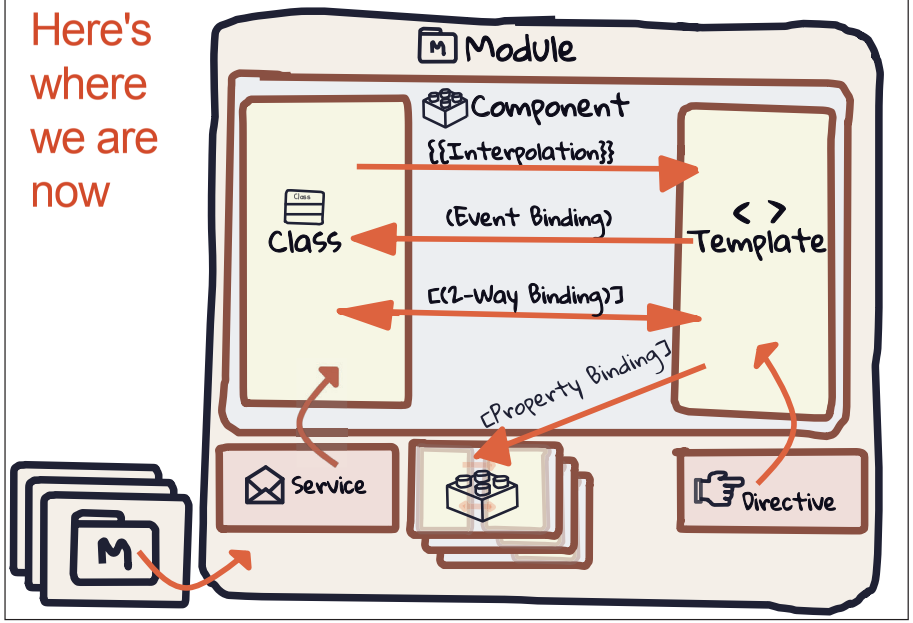
And you have to duplicate your panel

- 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.

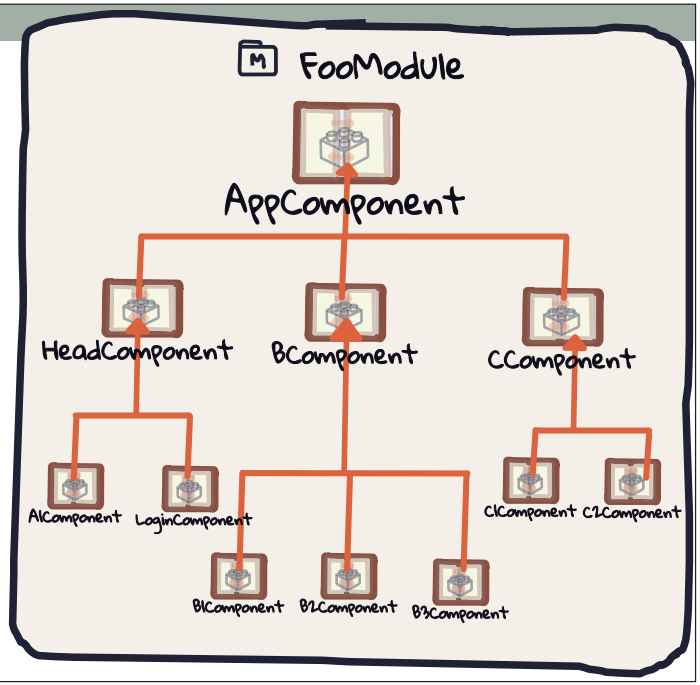
tl;dr

- 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.

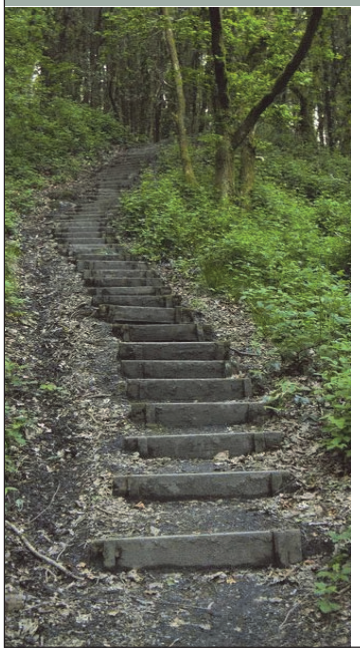
Here's where we are now



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

1. Add a tag

The inner

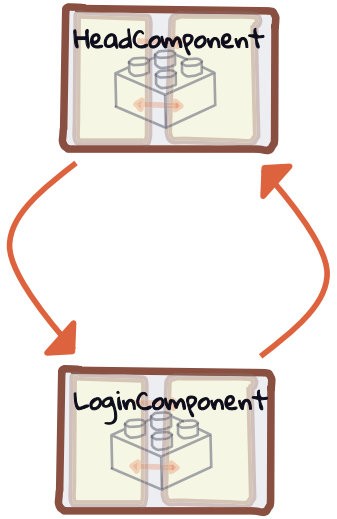
```
@Component({
  selector: 'login',
  templateUrl: 'login.component.html'
})
export class LoginComponent {
}
```

The host

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

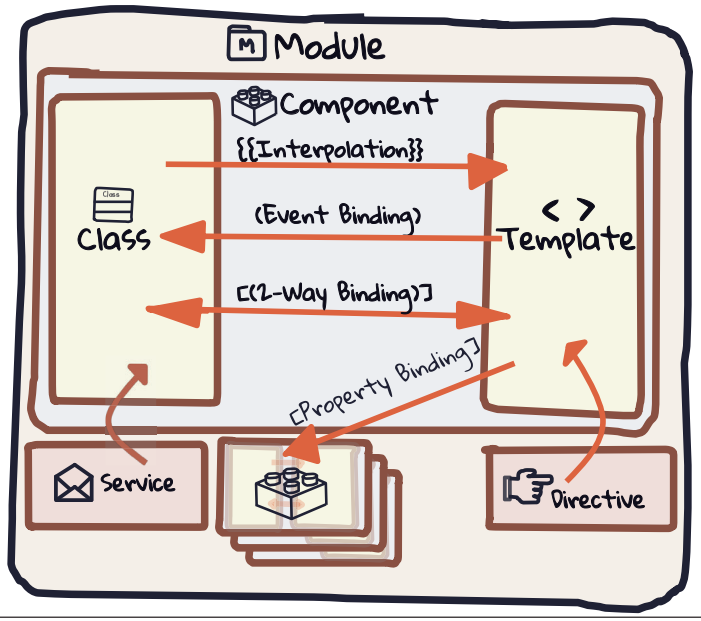
A word about data flow

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?



Property binding - when you pass an object from a component to an html tag or inner component

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

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

To pass data from host to inner

head.component.ts

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

head.component.html

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

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

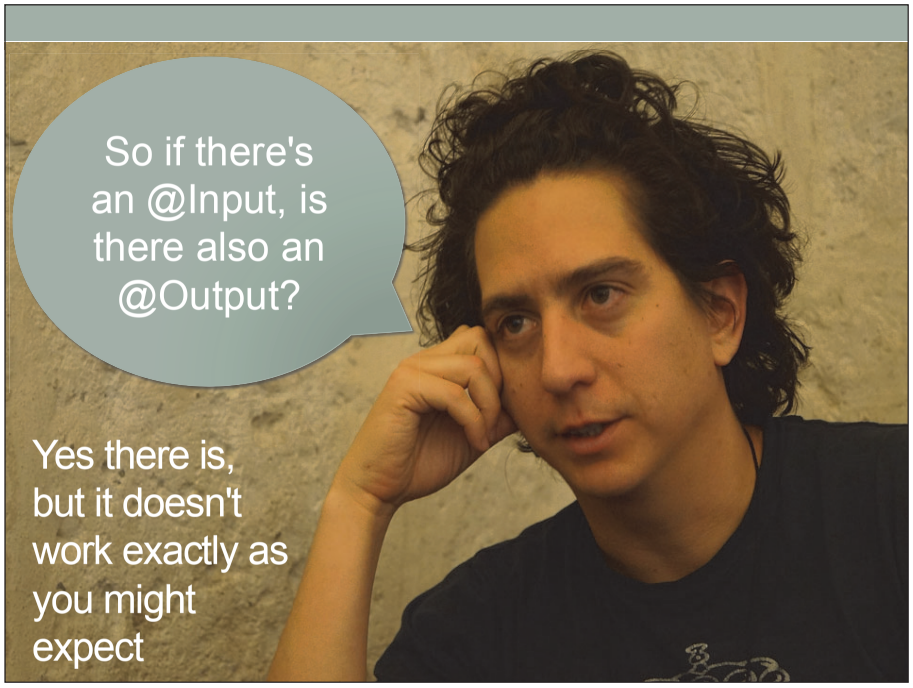
... And pass it down as a value of a property binding.

To read data in inner from host

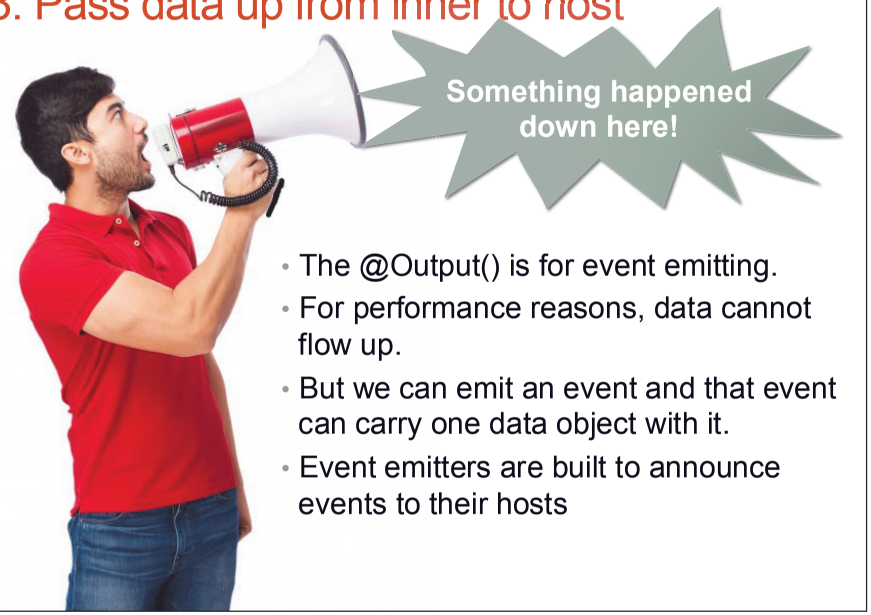
login.component.ts

```
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.



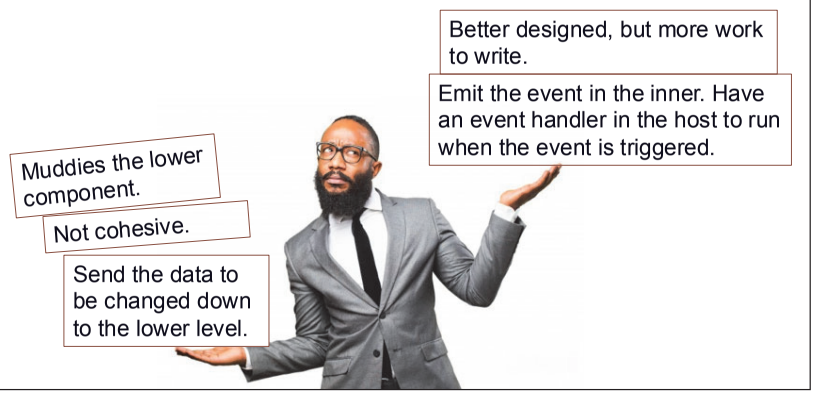
3. Pass data up from inner to host



- The @Output() is for event emitting.
- For performance reasons, data cannot flow up.
- But we can emit an event and that event can carry one data object 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

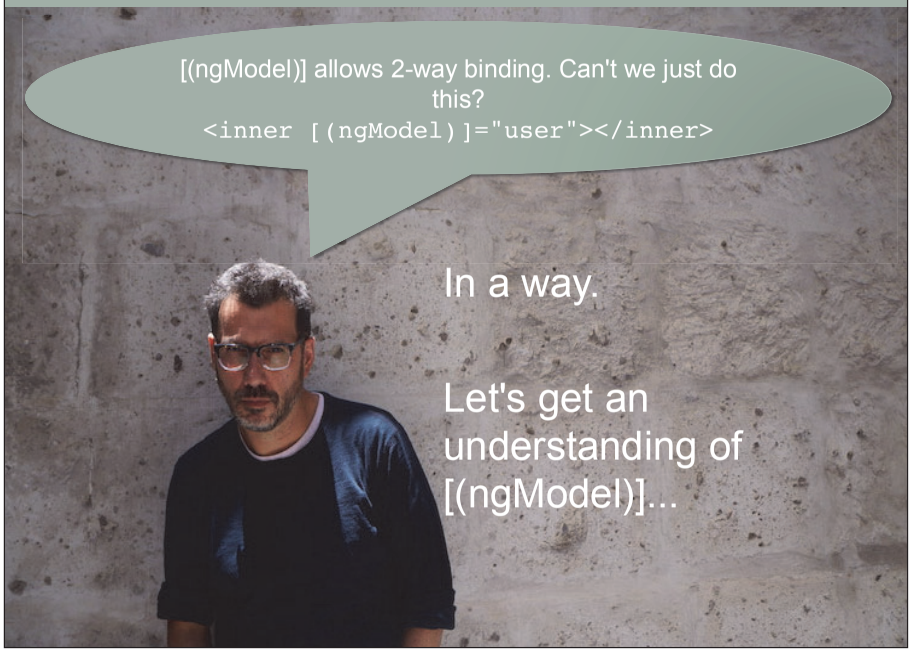
```
import { Input } from '@angular/core';
@Component({})
export class HeadComponent {
 loginUser(theUser) {
  // This method will execute when the
  // loggedIn event fires in the inner
 }
}
```

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.



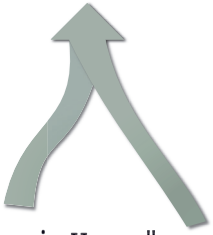
[(ngModel)] is actually syntactic sugar!

```
<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

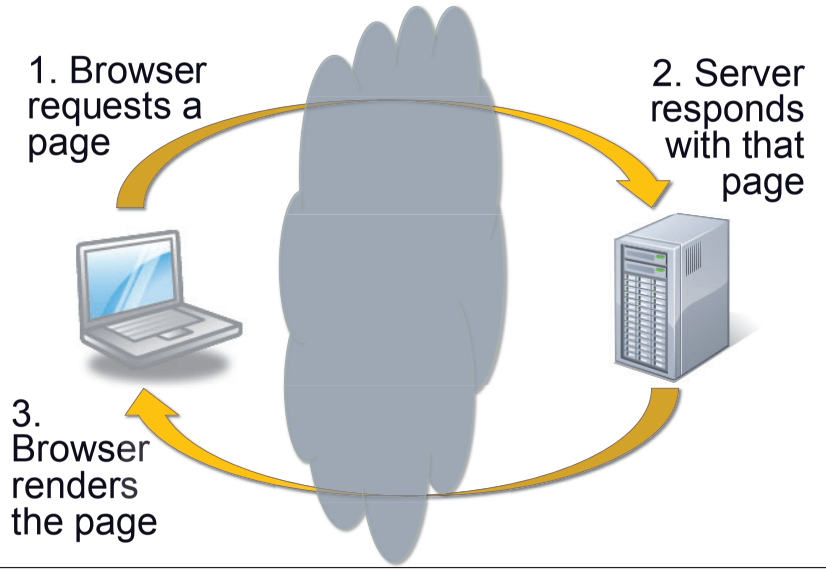
- Applications are comprised of nested components which are made up of nested components and so forth and so on.
- We usually want inner components and their host components to be able to send data up and down.
- Data goes from host to inner via [property binding]
- Data goes from inner to host via Event 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 -- HttpClient
- HttpClient 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

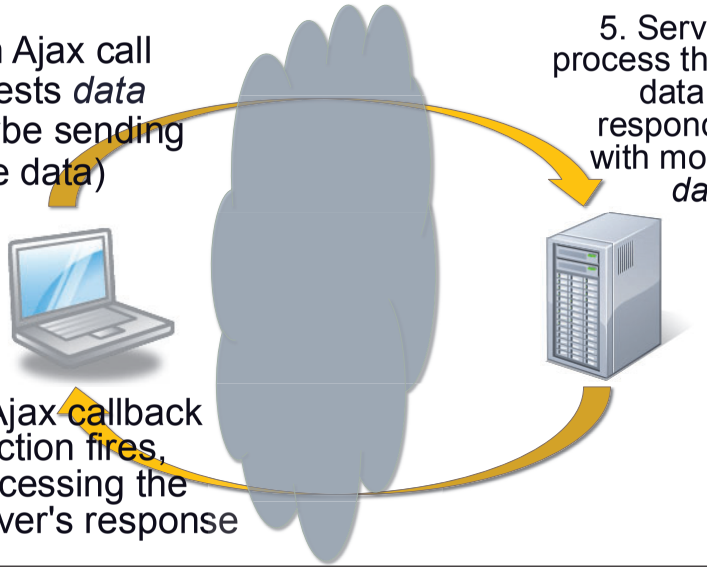
503 Service Unavailable

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

Here's how Ajax works

4. An Ajax call requests *data* (maybe sending some data)

5. Server process that data & responds with more *data*



6. Ajax callback function fires, processing the server's response

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

HttpClient

```
import { HttpClientModule }
  from '@angular/common/http';
...
@NgModule({
imports: [
  BrowserModule,
  FormsModule,
  HttpClientModule
],
...
}) export class someModule() {}
```

HttpClient is in a separate module called *HttpClientModule* so we must imports it

```
ERROR Error: Uncaught (in promise): Error: StaticInjectorError(AppModul
  StaticInjectorError(Platform: core)[DashboardComponent -> HttpClient]:
    NullInjectorError: No provider for HttpClient!
Error: StaticInjectorError(AppModule)[DashboardComponent -> HttpClient]:
```

To use it in a class, Angular must DI it (so you must put it in the constructor arguments).

```
import { HttpClient }
  from '@angular/common/http';
@Component({...})
export class myComponent {
 constructor (private _http:HttpClient) {
  // Now you can use this._http anywhere
  // in this component.
 }
}
```

Then you can GET, POST, PUT, DELETE

Syntax

```
HttpClient.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() {
  console.log("Fetching users");
  this.users = this._http.get("/users");
  console.log("Got them");
};
```

We could use a *promise* as in ...

I promise I'll tell you when I'm finished!

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 server replies with any 200-series return code
- ... and it passes an object to the success function:

```
{
  _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 ...

```
class someComponent {
 users;  // <- Where the Ajax data will go
 getUsers() {
   this._http
   .get("/users")
   .toPromise()
   .then(
     (data) => { this.users = data; },
     (err) => { console.error("Yikes!",err);}
   );
 }
}
```

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 operators that must be imported 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

These are functions

```
observable
.pipe(catchError(error)).subscribe(success)
```

That .catchError() is a pipeable operator.

Pipeable operators

aka. "lettable" operators

Operators enhance the capability of observables

Operator	Description
map(funcToConvert)	Converts each element
filter(predicate)	Allows some through, others not
first()	Only the first one in a series
last()	Only the last one
single()	Throws if >1 exist
skip(x)	Skip x of them
take(y)	Allow only y of them

To use operators, put them in a pipe

```
Observable.pipe(...Operators);
```

```
// For example ...
observableOfPersons.pipe(
  filter(p => p.desc.includes(searchString)),
  map(p => `${p.first} ${p.last}`),
  skip(50),
  take(10)
).subscribe(nm => printFullname(nm));
```

We need to import operators

You can import each operator like this:

```
import { map, skip, take, filter }
    from 'rxjs/operators';
```

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 may want to convert your data to strongly-typed objects

```
this._httpClient
.get("http://us.com/persons/123")
.pipe(
  map(res => <Person> res)
)
.subscribe(res => this.person = res)
```

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

"<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 operators that must be imported before they're used

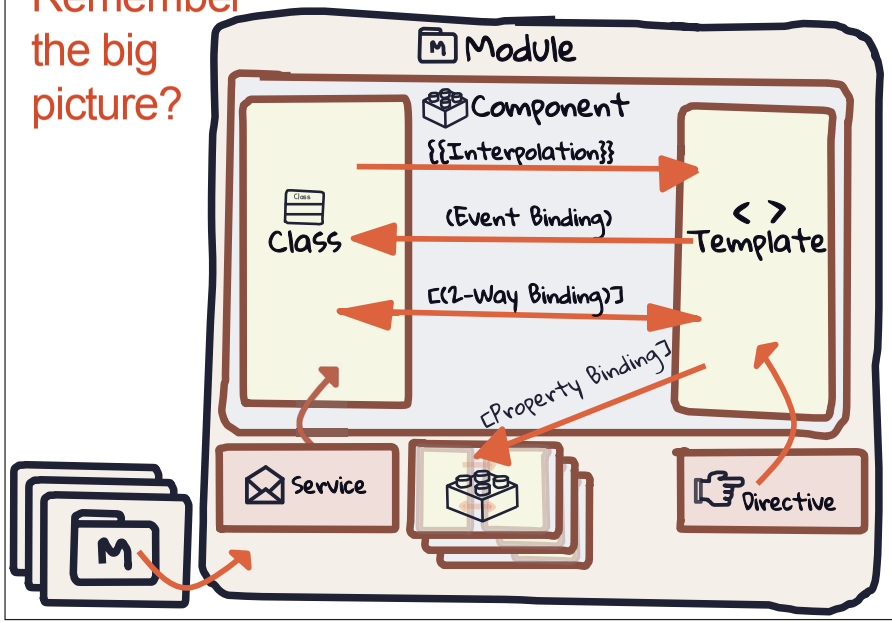
Services

(aka. Injectibles. aka. Providers)

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.
- Provide 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.

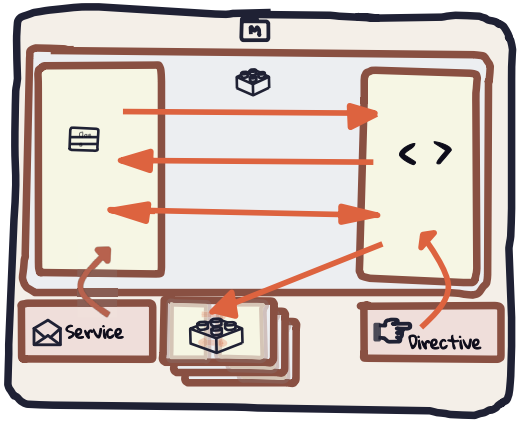
Spare parts []
Vacuum bottom
Balance valves
Company name
Backwash
Pool service
Clean baskets
Repair pump
Clean filter
Employees []
Test chemicals
Balance chlorine
Phone number

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

Services should be view-agnostic

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

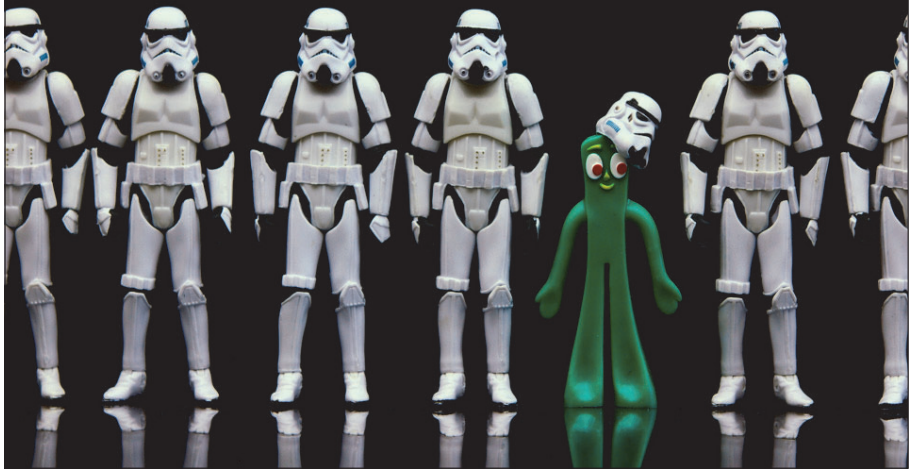
Why?

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. Provide them in the component or module

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

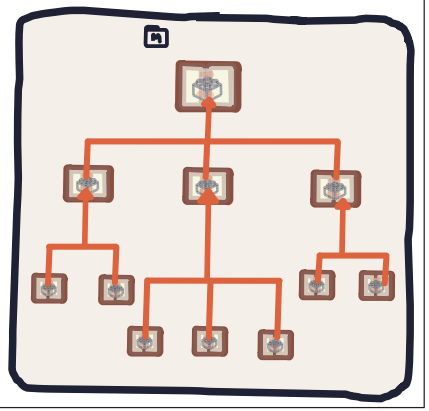
3. Use DI to inject them in the constructor

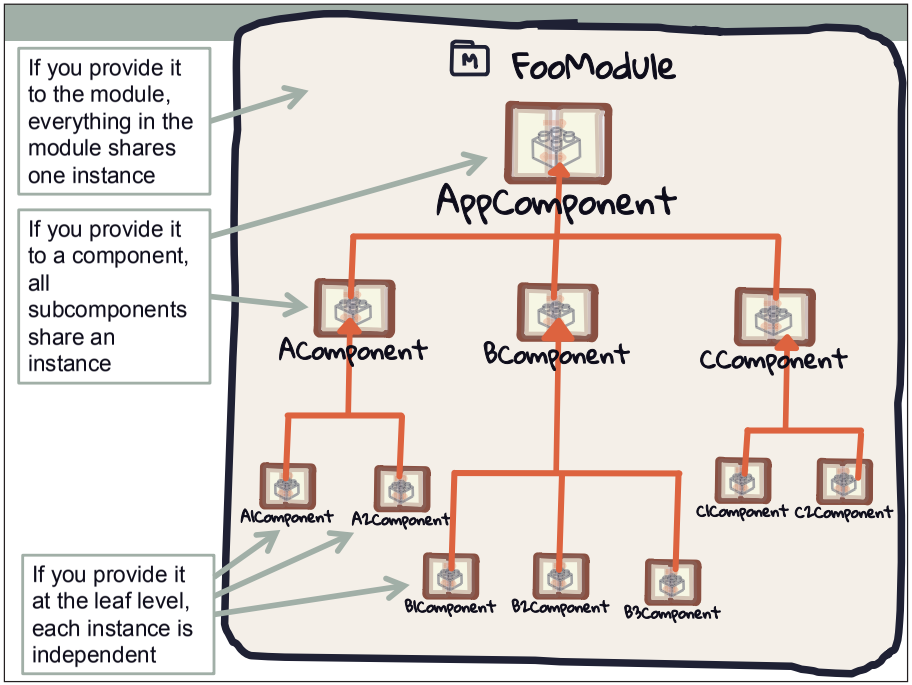
```
constructor(private _fooService: FooService) { ...
```

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

Where you provide them is very important!

- For tight cohesion you want to provide them as low as possible.
- But if you get too low, you have to provide 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;
  }
}
```

Injectable must be imported

Must be marked as @Injectable

Any member of the class will be exposed as part of the service.

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.
- Provide 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 people">
    {{ 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

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

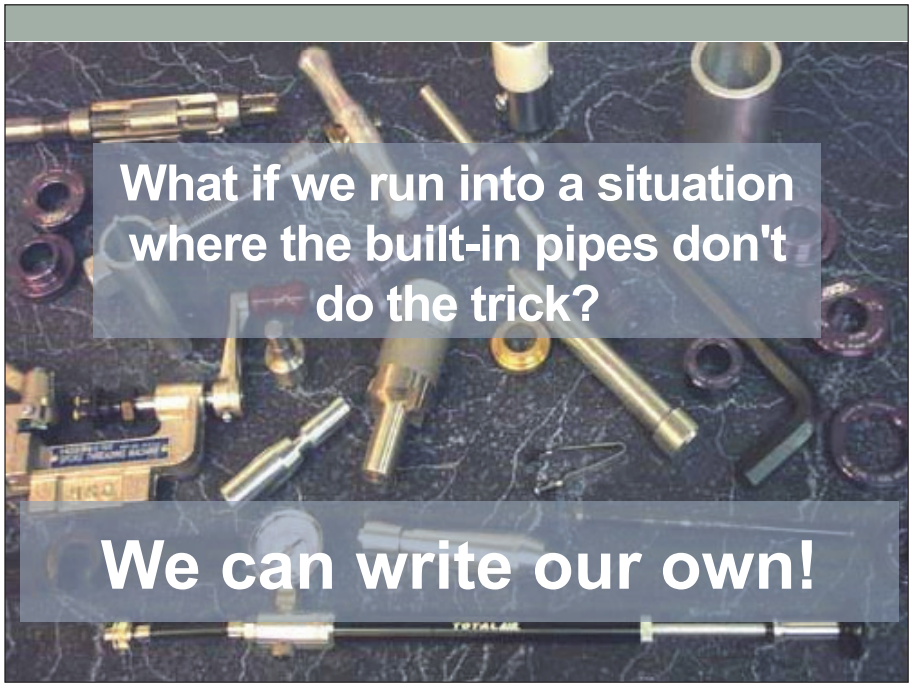
```
<div *ngFor=
'let person of people | slice:0:10 | orderBy:"lastName"'
>


<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.