



Building the next version of the web with
browser applications

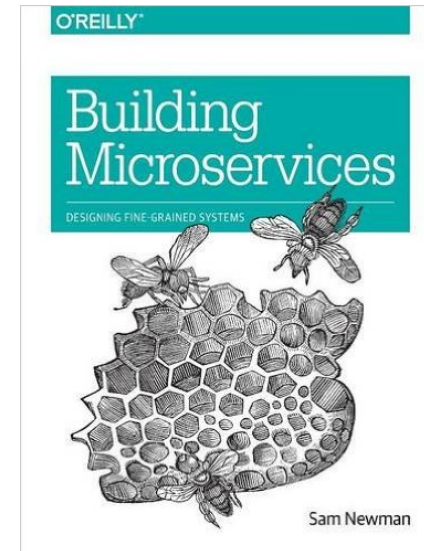


Prerequisites

- HTML / CSS
 - recommended course: 400 HTML-CSS
- JavaScript programming experience
 - recommended course: JavaScript
 - recommended course: JavaScript Tooling

Book

- **Building Microservices** by Sam Newman, O'Reilly Media, Inc., February 10, 2015





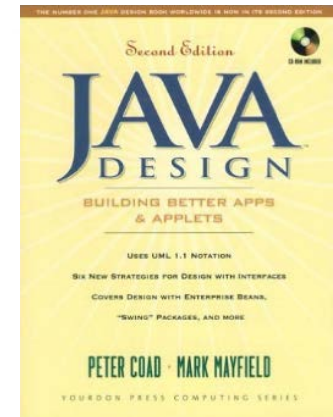
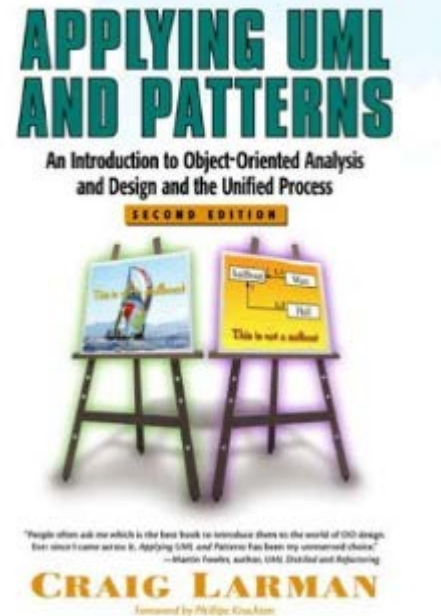
Other books

- Advanced choices
 - **Essential Angular** by Victor Savkin
(the router guy who left)
 - **Angular Router** by Victor Savkin



Other related books

- https://www.amazon.com/Applying-UML-Patterns-Introduction-Object-Oriented/dp/0130925691/ref=sr_1_fkmr1_1?ie=UTF8&qid=1473861785&sr=8-1-fkmr1&keywords=larman+craig+2nd+ed
- <https://www.amazon.com/Java-Design-Building-Better-Applets/dp/0139111816>





Exercises

- Completed exercises for the current version will be kept at
 - <http://github.com/doughoff/wd-530>



Intro to Angular



History

- 2009 – team started with Brad Green, manager
- Sep 2012 - 1.0.2
- March 2015 - Angular 2 announced
 - Sep 2016 final
- March 2017 – Angular 4
 - skipped 3.0, has breaking changes due to router
 - May 2017 4.1.0
- October 2017 – Angular 5 rc.3



Resources - official

- Site: <https://angular.io/>
- Code: <https://github.com/angular>
- Docs: <https://angular.io/docs/>
 - Cheatsheet -
<https://angular.io/docs/ts/latest/guide/cheatsheet.html>
- Blog: <http://angularjs.blogspot.com/>
- Milestone watch:
<https://github.com/angular/angular/milestones>



Resources - minor

- Google Groups: <http://ng-learn.org/>
- Angular Modules: <http://ngmodules.org/>
- AngularJS 1 site: <https://angularjs.org/>
- Torgeir Helgevold articles -
<http://www.syntaxsuccess.com/angular-2-articles>
- Design docs:
<https://drive.google.com/drive/u/0/folders/0B7Ovm8bUYiUDR29iSkEyMk5pVUk>



Language support

- **ES5:** "today's JavaScript"
 - The easy, safe choice for Angular 1.
- **ES6:** ECMAScript 2015 or ES6
 - Partially support in current browsers, real applications require compiling.

IE	Edge *	Firefox	Chrome	Safari
			49: 100%	
8: 0%	13: 100%	47: 100%	51: 100%	
11: 43%	14: 100%	48: 100%	52: 100%	9.1: 36%
		49: 100%	53: 100%	10: 100%
		50: 100%	54: 100%	TP: 100%
		51: 100%	55: 100%	



VS Code extensions

- Angular Language Service
 - <https://marketplace.visualstudio.com/items?itemName=Angular.ng-template>
 - YouTube with Chuck Jazdzewski
 - <https://www.youtube.com/watch?v=ez3R0Gi4z5A>
- Angular v4 TypeScript Snippets – John Papa
 - <https://marketplace.visualstudio.com/items?itemName=johnpapa.Angular2>



Improvements over v1

- **Speed:** Dramatically faster with fast initial loads through server-side pre-rendering, offline compile for fast startup, and ultrafast change detection and view caching for smooth virtual scrolling and snappy view transitions.
- Browsers: **IE 9** and all the others
- Size: Angular 1 - 56K. Angular 2 beta - 170K. RC - **45K**.



Other improvements

- Better Syntax
- Functional Reactive Programming (FRP)
- Command Line Interface (CLI)
- Augury (Batarangle) - <https://augury.angular.io/>
- Cross Platform
- Mobile Web

Load	Compile	Render	Re-render
Angular Universal	Offline Compile	Ultrafast Change Detection	View Pool
Instant rendering	3x faster vs ng2	2.5x faster vs ng1	4.2x faster vs ng1



Angular Universal

- Parses your app's JavaScript by pre-rendering the first view on the server-side.
 - Full Stack Angular 2, Jeff Whelpley and Patrick Stapleton - <https://www.youtube.com/watch?v=MtoHFDfi8FM>
- Using server-side rendering in IIS via nodeServices (not yet available)
 - Steve Sanderson (ng2 + ASP.NET5 / MVC6 Music Store, React – no TypeScript) Nov 2015 - [Channel9 video](#)
 - <https://github.com/aspnet/NodeServices/tree/master/samples>



Angular CLI

- <https://cli.angular.io/>
- Scaffolding tool
- Based on Ember's CLI
- Automates basic tasks for setup and boilerplate code
- One version behind usually
- Installs
 - Jasmine, Codelyzer, Karma, Protractor, tslint



Other

- Animations
 - <https://angular.io/docs/ts/latest/guide/animations.html>
- Testing with Jasmine, Karma, Augury (Chrome extension)
 - <https://angular.io/docs/ts/latest/guide/testing.html>
 - <https://augury.angular.io/>
- RxJS ("Reactive Extensions")
 - asynchronous observable pattern, Microsoft project forked for any language
 - <https://github.com/ReactiveX/RxJS>



Hybrid apps using Angular

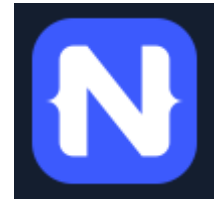
- Ionic2

- <http://ionic.io/2>
 - alpha – 11/2015
- Build native apps from JS/TS APIs



- Telerik's NativeScript

- <http://www.telerik.com/nativescript>
- Build native apps with XML custom language





How to plan one-page apps

Planning an app





Architecture - SS framework

Server – ASP.NET or MVC

- Routing
- Controller logic
- Page generation
 - Data binding
 - Templates
- Security
- Services for client
 - data extraction

Client – jQuery, Bootstrap, etc.

- user triggered CSS
 - click / touch / hover
- user triggered server process
- browser triggered CSS
 - screen width



Architecture – SPA framework

Server – static files

- Services for client
 - **Security**
 - **Data**

Client – browser with

- **Routing**
- **Controller logic**
- **Page generation**
 - **Data binding**
 - **Templates**
- **Security**
- user triggered CSS
 - click / touch / hover
- user triggered server process
- browser triggered CSS
 - screen width



Angular features

- Page generation
 - Data binding
 - Templates
- Controller logic
- Routing
- Reusable components

SPA only



- Google assumes this design
- Most examples and tutorials target this design



Combined SPA & SS frameworks?

- Server side provides better security
- One side provides less distributed problems
- Client side operation can be extended with less complex packages



Possible SPA & SS framework designs

- Combine operations into an app on a SPA with the same model
 - One row – details, update, delete, duplicate
 - Multiple rows, same schema – browse, search, bulk data operations
 - Multiple rows, different schema – display, rearrange, insert, drop
- Create SS reusable view component library
 - Web Components



Best Practices

- Angular2 Styleguide
 - <https://angular.io/styleguide>
- Codelyzer
 - <https://github.com/mgechev/codelyzer>
 - for code reviews, linting, ... soon static code analysis, template analysis, auto suggest
 - current: tslint
 - links to styleguide, live advice, in angular-cli
 - <https://www.youtube.com/watch?v=bci-Z6nURgE&feature=youtu.be> (May 2016) - Minko Gechev



Material Design

- Angular Material
 - <https://material.angular.io/>
- Angular Material vs Material Lite
 - <https://scotch.io/bar-talk/angular-material-vs-material-design-lite>
 - Material Design Lite
 - <https://getmdl.io/>
- Angular Material 1.1.1 – 12/2015
 - <https://material.angularjs.org/> for ng1



Material Design for ng2

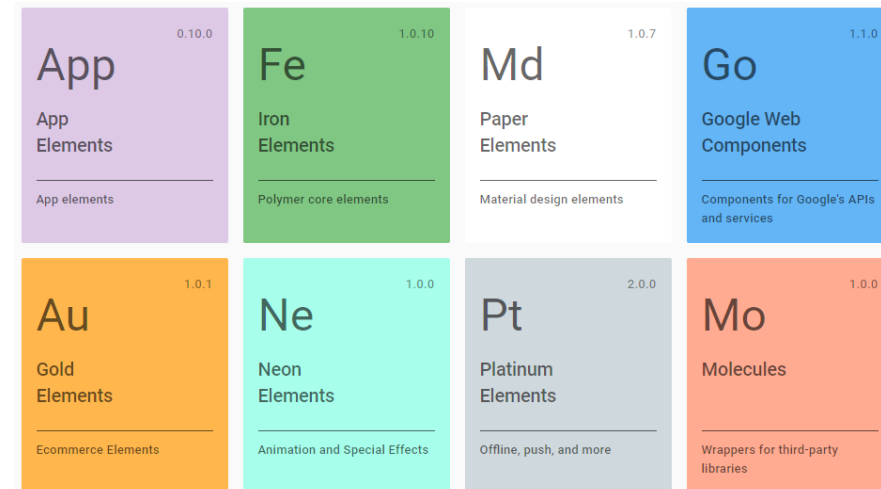
- UI library for fast building of mobile style apps
- Demo Angular I/O 2016:

https://www.youtube.com/watch?v=EwYD_xqB7Qs&list=PLOU2XLYxmsILe6_eGvDN3GyiodoV3qNSC&index=155



Google Polymer

- <https://www.webcomponents.org/>
- uses Web Components
 - WC are like Angular directives/components without the framework
 - bundles HTML, CSS & JS into custom elements
- Paper Elements
 - Material Design with Polymer





Setup



Setup choices – code only

- Quickstart
 - <https://github.com/angular/quickstart>
- npm angular2
 - <https://www.npmjs.com/package/angular2>
 - 2.0.0-beta.17 – 7/19/2016
- GitHub angular-master
 - <https://github.com/angular/angular>



Setup choices – code + scaffold

- Angular CLI
 - <https://cli.angular.io/>
- Minko Gechev's Angular Seed project
 - <https://github.com/mgechev/angular2-seed>
 - RC6 – Aug



Local test server

- npm packages for local tests
 - [lite-server](#) – the current dependency
 - executed with `>npm run lite`
 - the CLI server
 - live-server
 - [http-server](#)
 - local-web-serverlite-
 - webserver



TypeScript options

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "moduleResolution":
"node",
    "sourceMap": false,
    "emitDecoratorMetadata":
true,
    "experimentalDecorators":
true,
    "removeComments": false,
    "noImplicitAny": false
  },
```

```
"exclude": [
  "node_modules",
  "typings/main",
  "typings/main.d.ts"
]
```



TypeScript options

- emitDecoratorMetadata: true
 - transpiles necessary info for IDE – lots of errors if you don't!
- "noStrictGenericChecks": true
 - fixes rxjs 5.0 generic error with
 - or
 - "rxjs": "5.5.0", in Angular's package.json



Type definitions config

- <https://github.com/typings/typings>
- manage and install TypeScript definitions

```
{  "globalDependencies": {
  "core-js": "registry:dt/core-js#0.0.0+20160725163759",
  "jasmine": "registry:dt/jasmine#2.2.0+20160621224255",
  "node": "registry:dt/node#6.0.0+20160831021119"
}
```





Npm's inventory

```
{
  "name": "angular2-quickstart",
  "version": "1.0.0",
  "scripts": {
    "start": "tsc && concurrently \"npm run tsc:w\" \"npm run lite\" ",
    "lite": "lite-server",
    "postinstall": "typings install",
    "tsc": "tsc",
    "tsc:w": "tsc -w",
    "typings": "typings"
  },
  "license": "ISC",
  "dependencies": {
    "@angular/common": "2.0.0-rc.6",
    "@angular/compiler": "2.0.0-rc.6",
    "@angular/compiler-cli": "0.6.0",
    "@angular/core": "2.0.0-rc.6",
    "@angular/forms": "2.0.0-rc.6",
    "@angular/http": "2.0.0-rc.6",
    "@angular/platform-browser": "2.0.0-rc.6",
    "@angular/platform-browser-dynamic": "2.0.0-rc.6",
    "@angular/router": "3.0.0-rc.2",
    "@angular/upgrade": "2.0.0-rc.6",
    "core-js": "^2.4.1",
    "reflect-metadata": "^0.1.3",
    "rxjs": "5.0.0-beta.11",
    "systemjs": "0.19.27",
    "zone.js": "^0.6.17",
    "angular2-in-memory-web-api": "0.0.18",
    "bootstrap": "^3.3.6"
  },
  "devDependencies": {
    "concurrently": "^2.2.0",
    "lite-server": "^2.2.2",
    "typescript": "^1.8.10",
    "typings": "^1.3.2"
  }
}
```



RxJS

- `<script src="https://code.angularjs.org/2.0.0-beta.6/Rx.js"></script>`
- a required dependency of Angular 2
- install locally with **npm rxjs**
- Provides reactive programming syntax



<base href="/">

- Compatibility
 - IE10+
- Alternatives to base
 - Provide the router with an appropriate APP_BASE_HREF value.
 - Use absolute URLs for all web resources: css, images, scripts, and template html files.



`<base href="/">`

- If the application base changes you can use
 - `<script>document.write('<base href="' + document.location + '" />');</script>`
- This grabs the current URL
 - used in Google's documentation



<base href="/">

- Insert in <head> before any URL reference that might use it
- Sets a prefix to any relative URL path on the page
 - <base href="/">
 - <base href="/pages/baseball">
- Necessary to form html5 style URLs which use history.pushState
- can also use the target attribute to always open a new page



HTML package loading

```
<script src="systemjs.config.js"></script>
<script>
  • System.import('app').catch(function(err){
    console.error(err); });
</script>
```



SystemJS

- `<script src="https://code.angularjs.org/tools/system.js"></script>`
- <https://github.com/systemjs/systemjs>
- Universal dynamic module loader
 - ES6 modules, AMD, CommonJS and global scripts in the browser and NodeJS.
- `System.import('main');`
 - or `('main.js')` or `('main.ts')`
- requires a web server
 - CORS



App selector

- `<app-root><i class="fa fa-spinner fa-pulse"></i>Loading...</app-root>`
- no inputs
- no outputs
- Only one selector, only one app.



The bootstrap

- allows for better testing
- platform specific
 - Cordova, Telerik NativeScript

```
import { platformBrowserDynamic } from
 '@angular/platform-browser-dynamic';
import { AppModule } from './app.module';
platformBrowserDynamic().bootstrapModule(AppModule);
```



Module configuration

- Introduced to provide Ahead-Of-Time compilation for faster precompile on server

```
@NgModule({  
  imports: [BrowserModule, FormsModule, HttpClientModule],  
  declarations: [AppComponent,...components, pipes],  
  providers: [...services...],  
  bootstrap: [AppComponent],  
})  
export class AppModule { }
```



Component classes

```
import { Component } from '@angular/core';
@Component({
  • selector: 'app-root',
  • // styles : `[p:color:red, div: color:green]`
  • // template: '<h1>An AngularJS 2 App</h1>',
  • styleUrls: ['app.component.css'],
  • templateUrl : `app.component.html`
})
export class AppComponent { }
```



moduleId – Mar 2017

- 5.0 changes all this
- Component relative (webpack problem)
 - @Component({
 - **moduleId: module.id,**
 - templateUrl : `./basic.component.html`,
 - styleUrls: ['./basic.component.css']
- index.html / app relative
 - @Component({
 - templateUrl : `basic/basic.component.html`,
 - styleUrls: ['basic/basic.component.css']



Modules

- ES6 / TypeScript
 - not required by Angular but very recommended
- barrels – collections of modules
- bundle – a file for all the code of one or more barrels



Modules – import & export

- **import** { Component } from “@angular/core”;
 - allow use of class Component from a .js file called core
- **export** class HelloName { }
 - allow use of class HelloName by another import
 - functions and values can be exported also
- import and export use ES6 module syntax
 - <http://www.2ality.com/2014/09/es6-modules-final.html>



Component

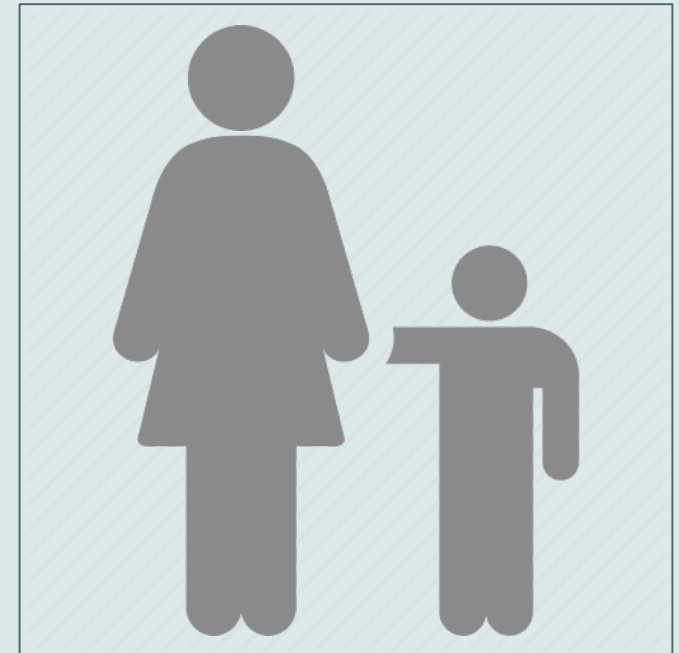
- The area of the DOM that you want to manage
 - the view scope
- Three parts
 - metadata configures the code
 - defines what tag to use
 - uses TypeScript's decorators: `@Component`
 - a template defines the HTML and data variables
 - uses `{{ mustache tags }}`
 - uses special attributes
 - a class defines the view logic and data

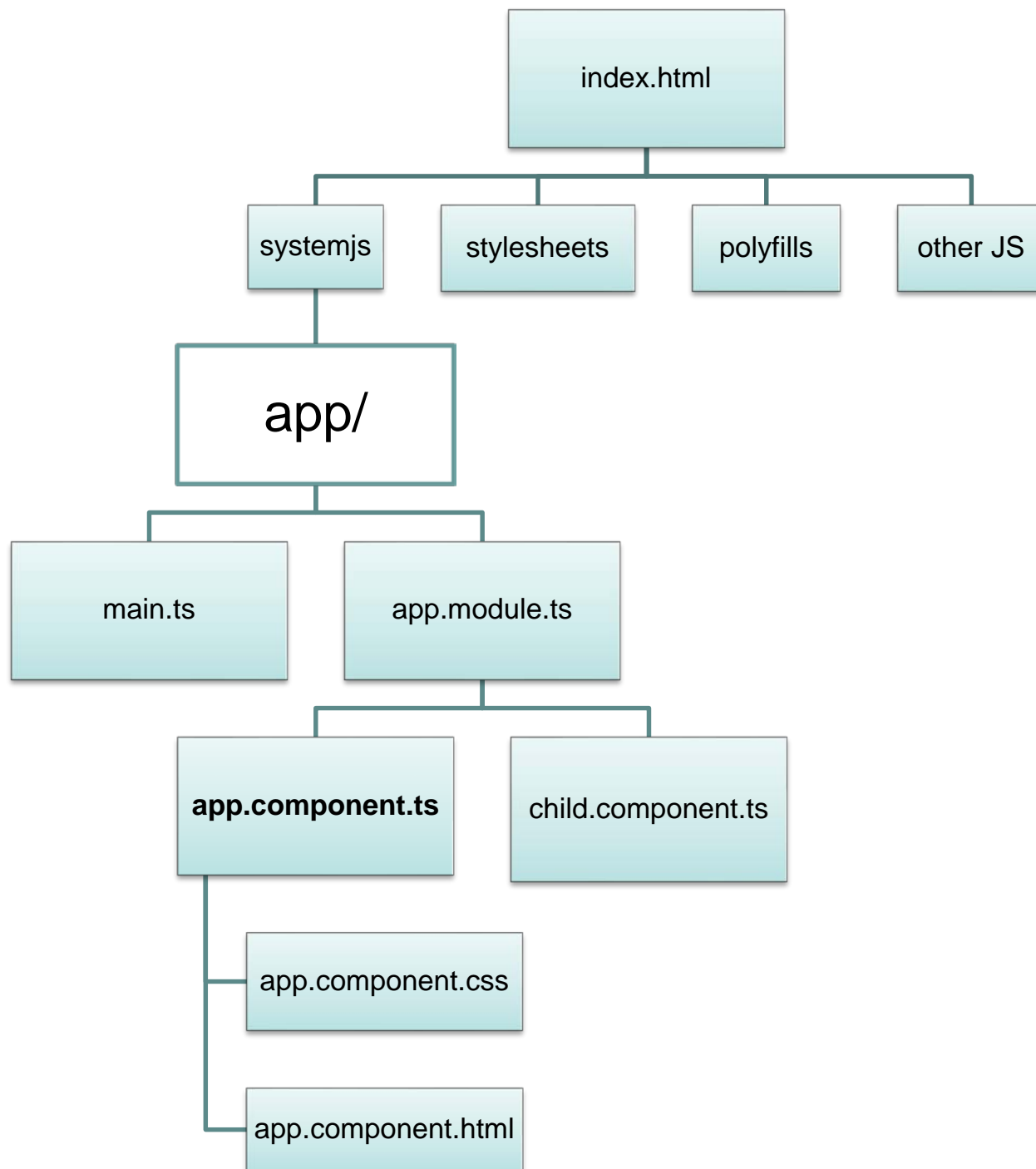


App structure

- HTML page
 - polyfills for older browsers
 - SystemJS loader
 - HTML
 - root app selector
 - component code
 - template
 - styles
 - dependencies
 - child components selectors
 - child component code...

HTML

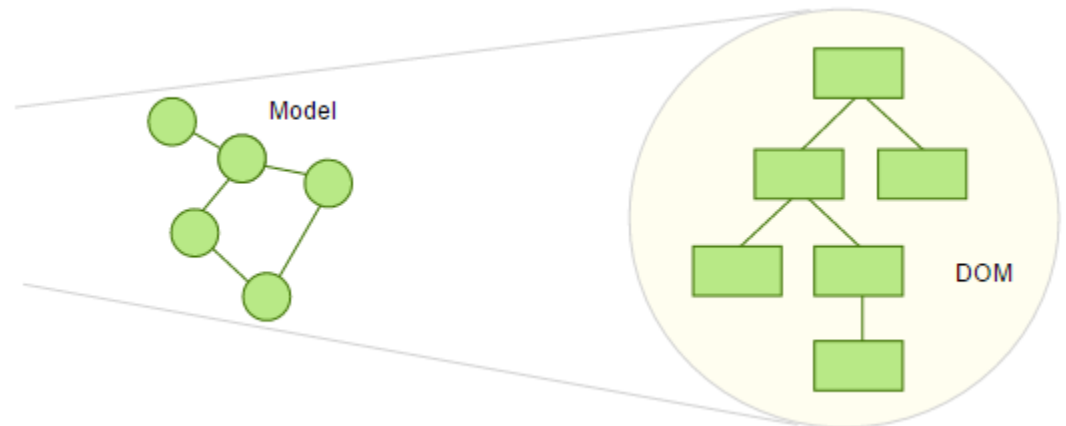






Data binding – projecting data

- Data model in code → DOM
- mapping, projecting, no change
- updates require mapping/binding
- must track state (the model/DOM data)





Data binding – app.component.ts

```
import { Component } from "@angular/core";

@Component({
  moduleId: module.id,
  selector: 'hello-name'
  template: `<div>Hello, {{name}} </div>`
})
export class HelloName {
  private name: string = 'world';
}
```

<hello-
name>

HelloName:

name = 'world'

</hello-
name>



Templates – inline vs file

- **template:** `<div>Hello, {{name}}</div>`
using ES6 template strings
or
- **templateUrl:** `'/templates/hello_name.html'`
 - path from root, not file



Component view

```
class HelloNameApp {  
    private name: string;  
    constructor( ) {  
        this.name = 'world';  
    }  
}
```



Development to production

- You will see on the console:
 - Angular 2 is running in the development mode. Call `enableProdMode()` to enable the production mode.
- To make faster for production

```
import { NgModule, enableProdMode } from  
'@angular/core';  
enableProdMode();
```



Exercises

- Set up TypeScript environment
- Setup using QuickStart project in Code
 - Load template pages from
 - <https://github.com/doughoff/WD-530>
- Measure template's resource loading times



Components





Directive

- Three types
 - **Component** – main unit of Angular
 - **Structural** – many built-in logic functions for layout
 - `< employee *ngIf="isEmployed"></employee >`
 - **Attribute** – alters behavior or appearance by adding attribute syntax
 - `<input [(ngModel)] ="employee.name">`



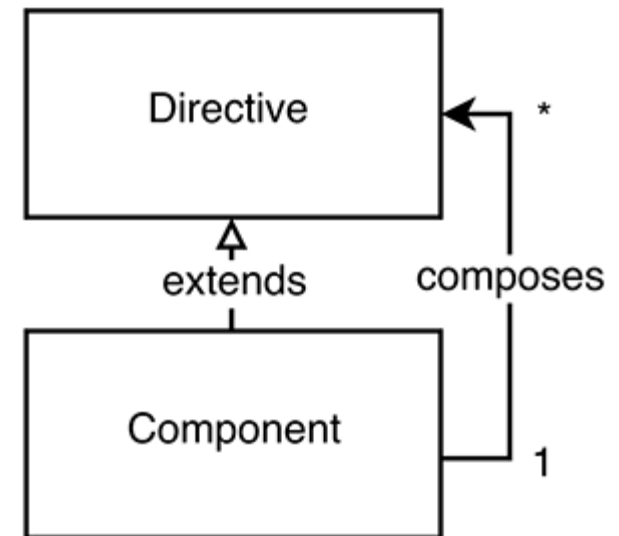
Components

- the Angular 1 directive with a template
- @Component
- best practice
 - mediate between template and model, the controller logic
 - factor out non-component logic into services (logging, business logic, etc.), utility classes



Components

- Directive - Holds logic, but no structure, base class
- Component - Extends a Directive and is composed of other directives or components.





Selectors

- element, class, or attribute syntax
- selector: 'custom-box, .custom-box, [custom-box]', :not()
 - **<custom-box>Matching tags</custom-box >**
 - < custom-box /> is not valid – must use a closing tag, not empty
 - **a class**
 - **an attribute**
- selector: '.custom-box:not(h1)'



Selectors

- Not valid:
 - ids, ancestor/child, ng1 comments, ng1 alternate naming syntax (custom:box)
- Recommended – kebab-case
- Not as recommended
 - camelCase / PascalCase



Module declarations

- needed for initial component & child components
- @NgModule ({imports [...],
 declarations [Person, Beach]
- -----
- @Component({ selector: '**person**', template: `

.o.

`) class Person{ }
- @Component({ selector: 'beach', template: `

The Beach: <**person**></person>
<person></person>~~~~`}) class Beach{ }



Styles



Styles - internal

- Defined in the `@Component` decorator
- Written to a style element in the rendered page.
- Styles are bounded by the element of the selector (view encapsulation)
 - Emulated View Encapsulation - default
- **styles:** [`'.primary {color: red}'`, `'...'`]
 - an array of rule-sets, not a multi-line string for all!
- Webpack and other module bundlers
 - `styles: [require('my.component.css')]`



Styles - external

- **styleUrls:** ['./my-component.css', '...']
 - uses relative references when using



Styles – in template

- Use a `<style>` element at the top of your template to replace the **styles** or **styleURLs** of the Component decorator



Style strategy

- styles: - poor tool support
- styleURLs
 - **No transpile necessary!**
 - good for very large libraries
 - Access by designers
 - Uses base href, start relative URL without slash!
- template `<style>` - easily read, updated, managed



Style scoping

- **Emulated** - adds attribute to scope to component - default
- Native – uses browser's shadow DOM
- None – no scoping, styles are cross boundary from component to DOM, ~global
 - @Component {
 encapsulation: ViewEncapsulation.None
 ... }



Styles – special selectors

- **:host** { display: block; border: 1px solid black; }
 - Applies to containing component
- **:host(.active)** { border-width: 3px; }
 - Applies to containing component only when it has active class
- **:host-context(.theme-light) h2** { background-color: #eef; }
 - Applies to containing class child H2 elements if some ancestor has theme-light class



Styles - special selectors

- `:host /deep/ h3 { font-style: italic; }`
 - Forces (releases encapsulation for) style so any H3 descendent of containing component is styled
 - Only for emulated
- `:host >>> h3 { font-style: italic; }`
 - Alternate syntax for above



Exercises

- Use different selectors
- Add style
 - `${...}` ES5 expressions don't work



Templates



Templates - inline

- Inline template
 - can use ES6 backticked text (template literals)
 - **template:**
 - `<div *ngFor="let talk of talks"> {{talk.title}} by {{talk.speaker}}: {{talk.description}}</div>`
 - `



Templates - external

- External template - best
 - @Component({
 - **templateUrl: 'template.html'**



Syntax

- All HTML is valid except
 - `<script>` - to prevent injection attacks
 - `<html>`, `<body>`, `<base>`



Syntax - literals

- Text – quoted literals or expression
 - `{{ 'Hello' }}`
 - `{{ 1 + 11 + 111 }}`
- Text concatenation
 - `{{ 'Hello' + ', world' }}`
- Text interpolation
 - And he said "`{{ 'Hello!' }}`" to the world
- With a text filter
 - `{{ 'Hello' + ', world' | uppercase }}`



Syntax - literals

- Alternative syntax for {{ }}
- `<div>Hello {{name}}</div>`
- `<div [textContent]="interpolate(['Hello'], [name])"></div>`



Syntax - literals

- Mixed with ASP.NET server data bindings
 - `{{ '<%= DateTime.Now %>' }}`
 - server code executes first, then renders client side literal
- Mixed with attribute value text
 - ``



Expressions

- `{{ the Angular expression }}` can be
 - `{{ any @Component class member private field }}`
 - `{{ totalItems + 'items' }}`
 - `{{ any @Component class member method }}`
 - `{{ getQuantity }}`, `{{ calcQuantity() }}`
- result can be assigned to an element or directive property
- best practice
 - use data properties and methods to return values and no more



Elvis /safe navigation operator ?.

- guards against null and undefined values in property paths
 - view will disappear on null parent object
- `<p>Employer: {{employer?.companyName}}</p>`
 - if employer field is optional and undefined or null, the rest of the expression is ignored.
- Can be swapped out with longer version
 - `<p>Employer: {{employer && employer.companyName}}</p>`
- C# null coalescing operator 6.0





Ternary operator? yes : no

- An expression to replace `num > value ? 50 : 20`
- `{{`
 - `{true: 50, false: 20}[num > value]`
- `}}`



Not used

- Prohibited
 - Assignment except in Event Bindings.
 - **new** operator
- Not supported
 - bit-wise operators, | and &
 - ++, --
 - access to global namespace, window, or document
 - console.log()



Binding



Data binding – one way values

- interpolation
- from class (data source) to template in DOM (view target)
- most often a @Component class property
- template:
 - `<input type="text" value="{{name}}" />`
 - `<div>Hello, {{name}}!</div>`
- export class BuiltIn {
 - private **name**: string = 'John Smith';
- }



HTML attribute binding

- HTML attributes

- Includes global HTML attributes – class, id, style, title, etc.

- https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes

- Includes specific element attributes

- <https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input>

- <https://developer.mozilla.org/en-US/docs/Web/API/HTMLInputElement>

- Binding example

- `<input type='text' class='{{myClassName}}' id='{{idNumber}}' value='{{defaultValue}}' {{hasFocus}}>`



DOM property binding

- DOM properties using JS code

- Includes Element properties

- <https://developer.mozilla.org/en-US/docs/Web/API/Element>

- Includes specific properties

- <https://developer.mozilla.org/en-US/docs/Web/API/HTMLInputElement>

- Binding example

- `<input type='text' [autofocus]='hasFocus' [value]='defaultValue'>`
 - vs. `<input type='text' {{hasFocus}} value='{{defaultValue}}'>`

hasFocus =
“autofocus”
defaultValue = “1”



DOM property binding

- set the default value of an input from the class
- template: `

```
<input [value] = "defaultName">
```

`
- `})
- ```
export class BuiltIn {
```
- ```
    private defaultName : string = 'John Smith';
```
- ```
}
```
- value does not work with `ngControl="..."`



# DOM property binding

- standard syntax
  - `<input [value] = "defaultName">`
- alternate syntax / canonical form
  - `<input bind-value = "defaultName">`



# DOM property binding

- `<button [disabled]="isUnchanged"> Save </button>`
- The property of the DOM element
  - or Component or Directive
- Attributes initialize DOM properties – final
  - watch the DevTools when you update a text field
- DOM property bindings are not final
  - `button disabled="false"` does not work
  - `button [disabled]="isInvalid()"` does work



# DOM property binding

- HTML attributes that are also properties will be converted so either syntax is OK
  - `<input type="text" value='{{myName}}' />`
  - `<input type="text" [value]='myName' />`
  - ``
  - `<img [src]="heroImageUrl">`
  - `<div>The title is {{title}}</div>`
  - `<div [textContent]="The title is '+title'"></div>`
  - `<div [innerHTML] ='<span>text</span>' ></div>`

# HTML attribute vs DOM property binding



- Some HTML attributes are DOM properties
- Some HTML attributes don't have corresponding DOM properties.
  - `colspan`
- Some DOM properties don't have corresponding HTML attributes
  - `textContent`
- Many HTML attributes appear to map to properties ... but not the way we think!



# Other bindings

- Style

- `<button [style.color] = "isSpecial ? 'red' : 'green'">`

- CSS Class

- `.myClassName { }`
- `<div [class.myClassName]="isTruthy">`
  - `isTruthy = true`
- `<div [class]="myClassNameVariable">`
  - `myClassNameVariable = 'myClassName'`

- Attribute

- `<div [attr.role] = "myAriaRole">`





# Style property binding

```
<div [style.background-color] = "background"
 [style.color] = "foreground">
 Lorem ipsum dolor sit amet
</div>
```

```

export class BuiltIn {
 private background: string = 'hsl(200,80%,90%)';
 private foreground: string = 'hsl(200,80%,40%)';
}
```



# Style property binding

- `<button [style.color] = "isSpecial ? 'red' : 'green'">`

# Style property binding – unit selection



```
[style.font-size.px]="fontSize"
```

```
[style.font-size.em]="fontSize"
```

```
[style.font-size.%] ="fontSize"
```

```
<div [ngStyle]="{'font-size': fontSize+'px'}">
```



# Style property binding - ngStyle

```
<div [ngStyle]="setStyles()">
```

```
 This div is italic, normal weight, and x-large
```

```
</div>
```

```

setStyles() {
 return {
 'font-style': this.canSave ? 'italic': 'normal',
 'font-weight': !this.isUnchanged ? 'bold': 'normal',
 'font-size': this.isSpecial ? 'x-large': 'smaller'
 }
}
```



# Class property binding - DOM

- add or remove CSS class names
  - appends to class attribute
- `<div [class]="myClassName">`
  - appends the class of value of myClassName
- `<div [class.myClassName]="isTruthy">`
  - appends class property if value is truthy
    - Not truthy values are 0, no text, and false  
Also undefined and null in JS



# Class property binding - ngClass

- Use for multiple CSS class assignments by a map of classes to append with their corresponding boolean test.

```
<div [ngClass]="{
 active: isActive,
 disabled: isDisabled,
 'has-error': hasErrors
}">
```



# Class property binding comparison

- `[class.hide]='whenValidFor.first'`
- `[class.bold]='whenBoldFor.first'`
  - reads better for a few classes
- `[ngClass]='{'  
 hide:whenValidFor.first ,  
 bold:whenBoldFor.first  
'`
  - better for many classes



# Attribute binding

- exception to no attribute changes
  - useful when no property exists
- `<button [attr.aria-label] = "help">help</button>`
- `<div [attr.role] = "myAriaRole">`
- `<tr><td [attr.colspan] = "{{1 + 1}}">Three-Four</td></tr>`





# Other property bindings

- Directive property (input)
  - `<div [ngClass] = "{selected: isSelected}"></div>`
- Component property
  - `<hero-detail  
[fromParentComponent]="currentHero"></hero-detail>`



# Template #ref variables

- Node name bindings, hash refs, pound sign vars...

```

<figcaption>
 {{ 'Caption' + dogPic.alt }}
</figcaption>
```



# NgNonBindable

```
<div class='ngNonBindableDemo'>
{{ content }}

 • This is what {{ content }} rendered

</div>
```



# Exercises

- Get text from component
- Set attributes from component
- Use a dog-panel model class



View logic



# Structural directives

- \*ngFor
- \*ngIf
- Uses asterisk for sugar syntax to produce a **<template> element** in shadow DOM of element that has the attribute



## \*ngFor - syntax

- uses a collection to iterate over in Component class
- declare local variable in template of each item

```

 <li *ngFor = "let item of names">
 Hello, {{ item }}!


```



# \*ngFor – implicit values

- available values
  - **index** : int
  - first : boolean
  - last : boolean
  - even : boolean
  - odd : boolean

```
<div *ngFor="let city of cities; let i = index">
 #{{i}}: {{city}}
</div>
```





## \*ngIf - syntax

```
@Component({
 selector: 'built-in',
 template: `
 <div *ngIf ="x > y">
 x bigger than y
 </div>
 <div *ngIf ="x <= y">
 x less than or = to y
 </div>`
})
export class BuiltIn {
 private x : number = 300;
 private y : number = 200;
}
```



# switch – syntax for values

```
<div [ngSwitch]="x>y">
 x > y
 y >= x
 default text
</div>
```

```
<div [ngSwitch]="x>y">
 <ng-template [ngSwitchCase]="true">x > y</ng-
template>
 <ng-template [ngSwitchCase]="false">y >= x</ng-
template>
 <ng-template ngSwitchDefault >default text</ng-
template>
</div>
```



# switch – syntax for literal strings

```
<div [ngSwitch]="stringVar">
 <ng-template ngSwitchCase ="a">aaaa</ng-
template>
 <ng-template ngSwitchCase ="b">bbbb</ng-
template>
 <ng-template ngSwitchDefault >not a or b</ng-
template>
</div>
```



# Exercises

- 15. For directive
- 16. If directive
- 17. Switch directive



# Pipes



# Intro

- serves same purpose as a custom get method
- uses a transform function to alter values on the View
- called filters in ng1



# Pipe operator |, parameter :

- Single
  - `<div>{{ title | lowercase }}</div>`
- Chained
  - `<div>{{ birthday | date:' ' | uppercase }}</div>`
- Configured
  - `<div>Birthdate: {{currentHero?.birthdate | date:'longDate'}}</div>`



# Pipes - common and custom

- Common
  - `<p>The hero's birthday is {{ birthday | date:' ' }}</p>`
- Custom
  - `<p>Card No.: {{ cardNumber | myCreditCardNumberFormatter }}</p>`





# Common pipes – date

- uses the Internationalization API – IE11+, no Safari
- will not re-evaluate
- expression | date : format
  - expression is Date object or # of ms since UTC
  - format – check table at <https://angular.io/docs/ts/latest/api/common/index/DatePipe-pipe.html>



## Common pipes – date

- date : 'MMMMdy' or  
date : 'longDate' = September 3, 2010
- date: 'yMd' or  
date: 'shortDate' = 9/3/2010
- date: 'jm' or  
date: 'shortTime' = 12:05 PM



# Common pipes - currency

- uses the Internationalization API – IE11+, no Safari
  - [https://en.wikipedia.org/wiki/ISO\\_4217](https://en.wikipedia.org/wiki/ISO_4217)
- expression | currency : <currency code> : <symbol display> : <digit info>
  - symbol display: USD or \$ (false or true)
  - digit info: see decimal pipe
- amount | currency:'USD':false
- amount | currency:'EUR':true:'4.2-2'



# Common pipes - decimal

- expression | number: <digit info>
  - digit info: <minIntegerDigits | 1>.<minFractionDigits | 0> -<maxFractionDigits | 3>

```
{{ e | number:'3.1-5' }}
{{ pi | number:'3.5-5' }}
```



## Common pipes - percent

- expression | percent : digitInfo
  - digit info: see decimal pipe

```
{{amount | percent:'4.3-5'}}
```

# Common pipes – uppercase, lowercase



```
{{value | lowercase}}
{{value | uppercase}}
```

---

```
template: `

{{ 'abc' | textCasingStyle }}</p>
<button (click)='toggleFormat()'>Toggle Case</button>`


```

```
export class TestComponent {
 toggle = true;
 get textCasingStyle () { return this.toggle ?
'uppercase' : 'lowercase'}
 toggleFormat() { this.toggle = !this.toggle; }
}
```



# Common pipes - json

- Output

- { "firstName": "Hercules", "lastName": "Son of Zeus",
- "birthdate": "1970-02-25T08:00:00.000Z",
- "url": "http://www.imdb.com/title/tt0065832/",
- "rate": 325, "id": 1 }

```
<div>{{currentHero | json}}</div>
```



# Common pipes - slice

- `expression | slice : start : end`
- positive start, up to but not including end
  - `['a', 'b', 'c', 'd'] | slice:1 : 3 → ['b', 'c']`
  - `'abcd' | slice: 1: 3 → ['b', 'c']`
- negative start from end, not including how many from end
  - `'abcdefghij' | slice: -4 → 'ghij'`
  - `'abcdefghij' | slice: -4 : -1 → 'ghi'`





## Common pipes - async

- subscribes to an Observable, Promise or EventEmitter and returns the latest value it has emitted. When a new value is emitted, the async pipe marks the component to be checked for changes.
- the only common **stateful** pipe



# Common pipes - async

```
@Component({
 selector: 'hero-message',
 template: 'Message: {{delayedMessage | async}}',
})
export class HeroAsyncMessageComponent {
 delayedMessage: Promise<string> = new
 Promise((resolve, reject) => {
 setTimeout(() => resolve('You are my Hero!'), 500);
 });
}
```



# Custom pipes – code pipe

- @Pipe decorator on class, transform( )

```
import {Pipe, PipeTransform } from 'angular2/core';

@Pipe({name: 'yourPipeName'})
export class YourPipeClass implements PipeTransform
{
 transform(value:string, args:string[]) : any {
 return 'a transformed value';
 }
}
```



# Custom pipes – declare in module

- No need to declare in component since module covers that

```
import {CurlyQuotesPipe} from './curlyquotes.pipe';
@NgModule({
 declarations: [DogPanel, CurlyQuotesPipe,
 DogDetail],
```



# Custom pipes - execute

```
@Component({
 selector: 'aTag',
 template: `{{'no change' | yourPipeName }}`,
})
export class PipeTest { }
```



# Comparison

- Pipes are generally more readable but JavaScript can be used in place of them.

```
@Pipe({name: 'trim'})
export class TrimPipe {
 transform(value: string, args: any[]) {
 value.trim(); } }

```

```
{{ name | uppercase | trim }}
{{ (name | uppercase).trim() }}
{{ name.toUpperCase().trim() }}
```

# ng1



- number, orderBy, and filter are no longer used
- async, decimal, and percent are new to ng2



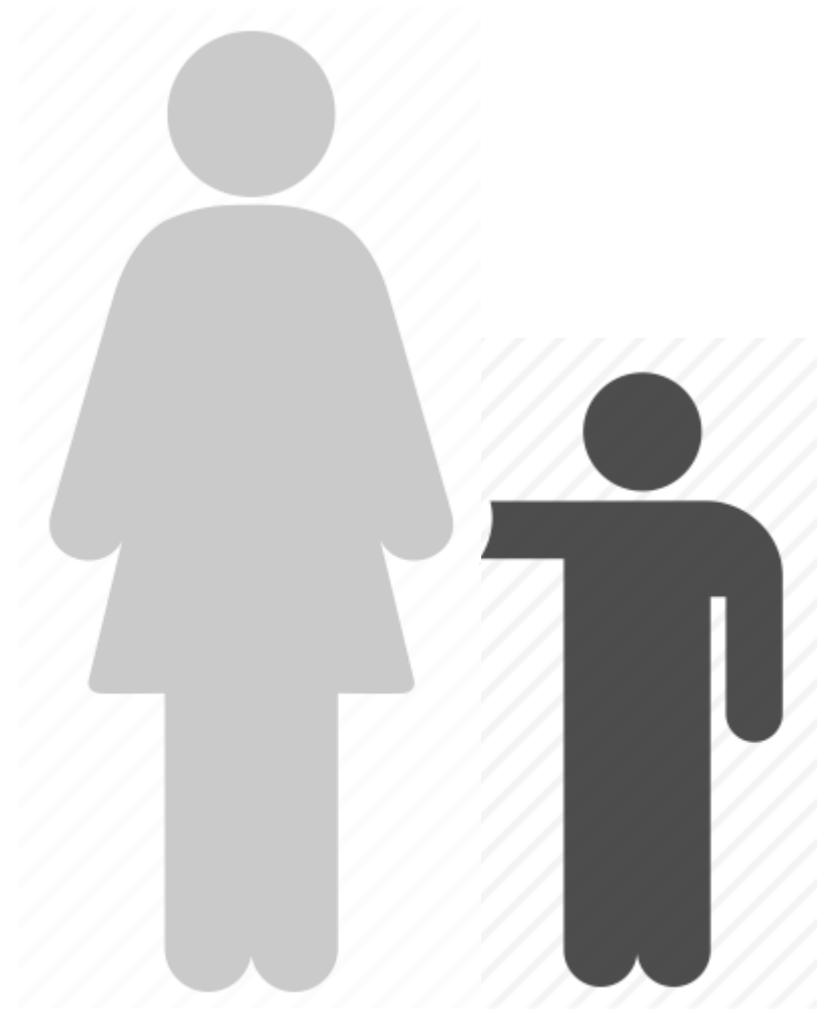
# Exercises

- Common pipes
- Async pipe
- Custom pipe





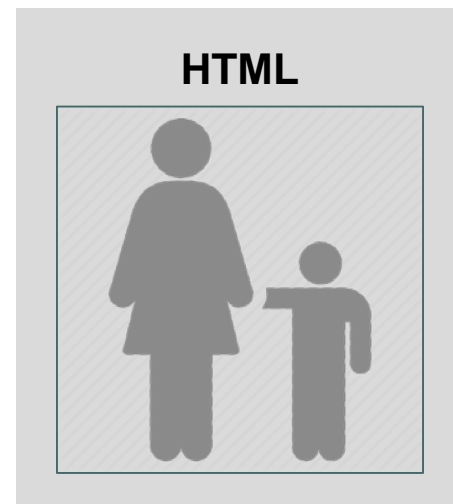
# Child components





# Intro

- Components can talk to each other
  - p2c - flow **data** from **parent to child**
  - c2p – flow **events** from **child to parent**
- Html page with app / root component
  - Can not see/compile data for root component
  - Security/architectural restriction
- Types of data
  - p2c - innerHTML, attributes, #vars
  - c2p – events, #vars





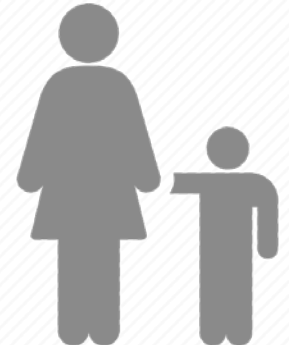
# Smart & dumb components

- Dumb / presentational component
  - Accepts data via inputs
  - Emits data changes via event outputs
  - stateless
- Smart / container component
  - Communicates with services
  - Renders child components
  - stateful



## p2c – declaring child components

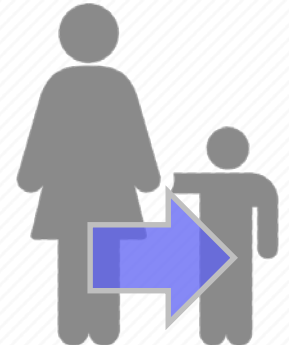
- use an import
  - import {Child} from './components/child';
- include a reference from module config to child
  - @NgModule({
    - imports: [ BrowserModule ],
    - declarations: [ ParentComponent, **ChildComponent**, ... ]





## p2c – content projection

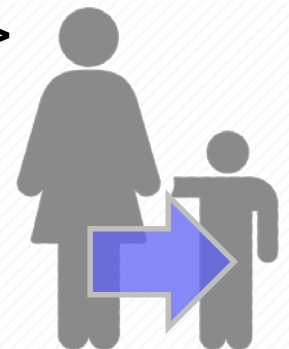
- Moves parent template's child element's innerHTML
- Child template 'queries' parent's innerHTML with `<ng-content>` element
- `<ng-content selector='... '>`
  - Collects all content matching selector
  - id attribute not implemented
- No selector
  - Gets all content not already selected
- ng1 - transclusion





## p2c – content projection

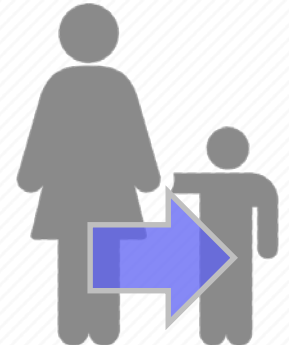
- **Parent template defines** elements & data
  - `<stuff>This is elemental stuff 1.</stuff>`
  - `<div a>aaaaaaaaaaaaaaaaattribute.</div>`
- **Child template uses** for final data position
  - `<ng-content select="stuff"></ng-content>`
  - `<ng-content select=".togetherness"></ng-content>`
  - `<ng-content select="[a]"></ng-content>`
  - `<ng-content select="planet[x]"></ng-content>`
  - `<ng-content></ng-content>`





## p2c – @Input - preferred

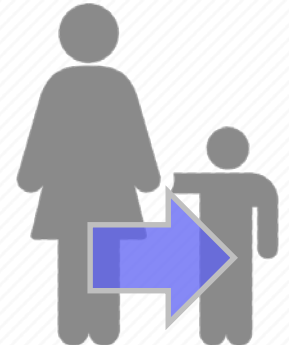
- **Parent template exposes data**
  - <child-component [**childVariableIn**]='childArgument' >
  - <child-component **childTextIn**='child text' >
- **Child component defines interface fields**
  - import { Input } from '@angular/core';
  - export class ChildComponent {
    - @Input( ) **childVariableIn** : string;
    - @Input('alias') **childTextIn** : string;
- **Child uses fields**
  - template - {{**childVariableIn**}} {{**alias**}}
  - code – **childVariableIn, alias**





## p2c – inputs: [ ]

- **Parent template exposes data**
  - <child-component [**childVariableIn**]='childArgument' >
  - <child-component **childTextIn**='child text' >
- **Child component defines interface fields**
  - @Component({
    - inputs: [**'childVariableIn'**]
    - inputs: [**'childTextIn' : alias'**]
- **Child uses fields**
  - template - {{**childVariableIn**}} {{**alias**}}
  - code – **childVariableIn, alias**

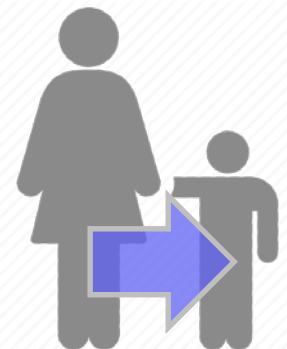






## p2c – @Input property setter

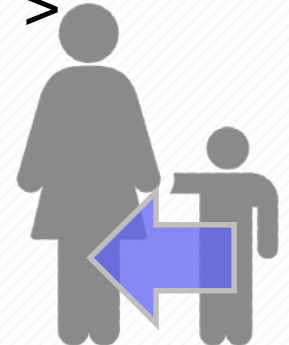
- **Parent template exposes data**
  - <child-component [**childVariableIn**]='childArgument' >
- **Child component defines interface field as setter**
  - import { Input } from '@angular/core';
  - export class ChildComponent {
    - private \_prop : string;
    - @Input( )  
set prop ( **childVariableIn** : string) { this.\_prop = **childVariableIn** || 'zilch';}
    - get prop() { return this.\_prop; }
- **Child template/component uses property** – {{prop}}





## c2p – @Output

- Child component declares EventEmitter
  - import { Component, EventEmitter, Output } from '@angular/core';
  - export class ChildComponent {
    - @Output( 'alias' ) **emitter** = new EventEmitter<any>();
- Child component emits event
  - this.**emitter**.emit(payloadOut);
- Parent template exposes interface in child element
  - <child-component ( **alias** )='onEvent(payloadIn)' >
- Parent component handles event
  - onEvent(data : any) { }





## p2p – local variables

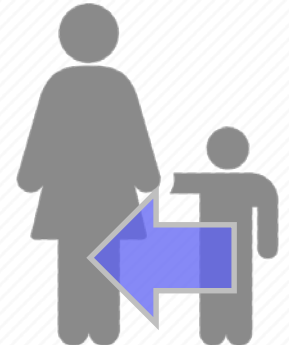
- uses pound sign before a scoped variable name for DOM element
  - also called a resolve
- `<div #newDiv />`
  - almost like `id='newDiv'` for cross element access
  - variable is now accessible from this element or in any descendant
  - alternative syntax `<div var-newDiv />`





## c2p – local child component var

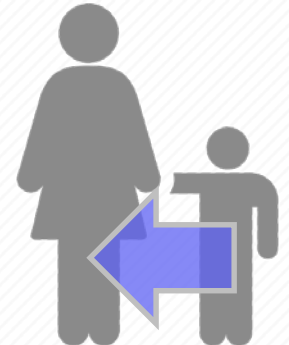
- Child declares members
  - `private field : any;`
  - `private function( ) : any { return 0; }`
- Parent exposes child element
  - `<child-component #child >`
- Parent uses child's members in template
  - `{{ child.field }}`
  - `{{ child.function( ) }}`





## c2p – @ViewChild

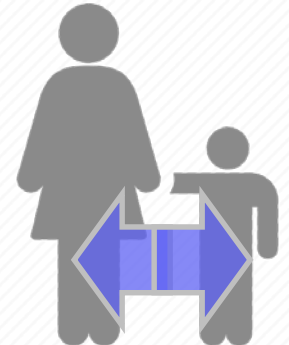
- @ViewChild, @ViewChildren
  - @ViewChild(AChildComponent) appears first in class declaration
  - reference child elements inside parent template – shadow DOM
  - ViewChildren is a QueryList – Iterable, Observable
    - **first**, **last** is one
    - **changes** will alert you when it changes
- use child component methods
  - EventEmitter for child → parent methods





## p2c/c2p – via service

- See Cookbook / Component Interaction / Parent and children communicate via a service
  - Message broker pattern
  - <https://angular.io/docs/ts/latest/cookbook/component-communication.html#!#bidirectional-service>





# Summary

- Content projection
  - Uses child's ng-content select to associate to any children of parent's child selector
- P2C
- C2P
  - Declare child element in parent template with # to direct access fields and methods



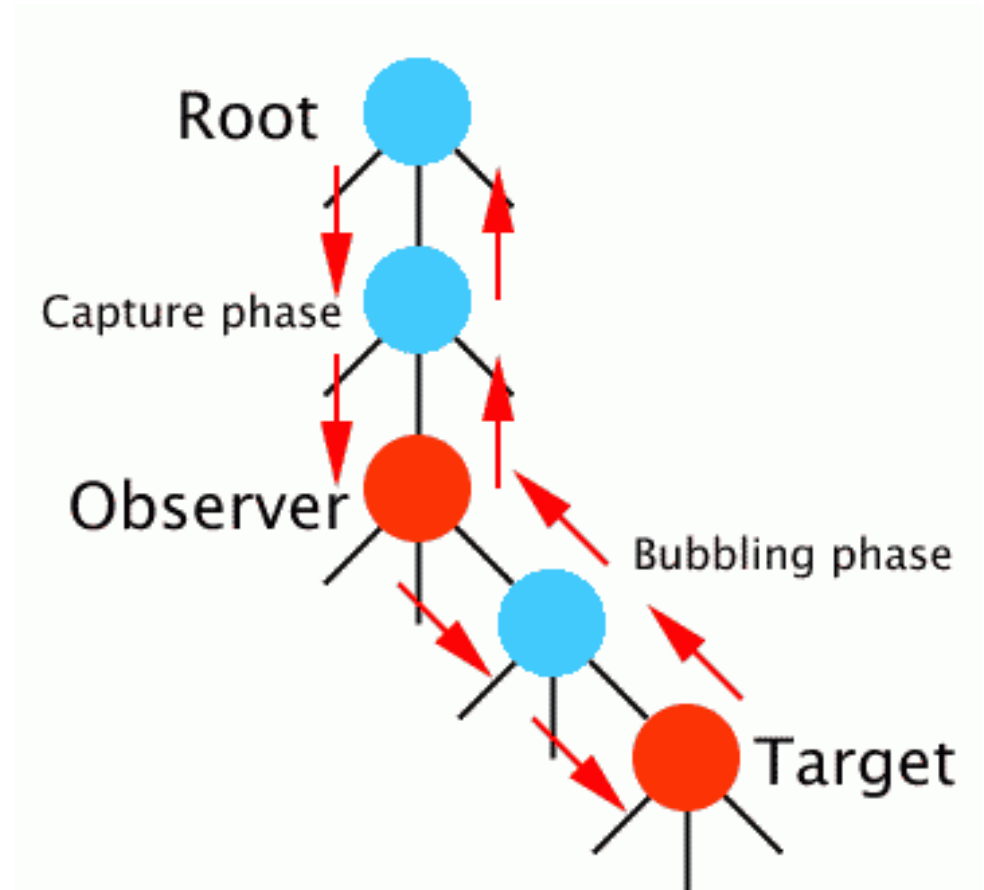
# Exercises

- 21. Content projection
- 22. Data input from parent
- 23. Event input from parent





# Events





# Event review

- Event objects
- Common events
  - click, change, focus, blur
- Event propagation
- Event default action
- return false;
- payload
- DOM events - <https://developer.mozilla.org/en-US/docs/Web/Events>



# Event binding - \$event

- \$event – a message payload
  - different for every event type
  - MouseEvent: click, dblclick, mouseup, mousedown.

```
<component context>
<... (event) = "functionName(arg)" ...>
<// component context>

<button (click)="readData($event: MouseEvent)">
```



# Event binding - \$event

- KeyboardEvent: keydown, keypress, or keyup

```
template: `
 <input (keyup)="confirmKey($event)"> `

confirmKey(event: Event){
 event.preventDefault();
 console.info('key pressed was', event.code);
}
```



# Event binding

- embed event in element like it would be in JS
  - JavaScript
    - `<button onclick='showMessage( )' >Show Message</button>`
  - ng2 – event only in parentheses
    - `<button (click)='showMessage( )' >Show Message</button>`



# Event type filtering

- better than `$event.keyCode`
- `keydown.a`, `keydown.shift.a`,  
`keydown.shift.control.a`,  
`keydown.shift.control.alt.a` etc.
- a – z, A – Z, 0 – 9, F1 – F12
- space, backspace, tab, clear, enter, pause,  
capslock, scrollock, escape, pageup, pagedown,  
end, home, arrowleft, arrowup, arrowright,  
arrowdown, insert, delete



# Events bind to template statements

- (event) ='template statement(s)'
- A template **statement**
  - responds to an **event** raised by a binding target such as an element, component, or directive.
  - has a side effect
  - updates application state from user input



# Event binding - syntax

- JavaScript
  - `<button onclick='readData(event)' > Show Message</button>`
- standard
  - `<button (click)="readData($event: MouseEvent)">`
- alternate
  - `<button on-click ="readData($event: MouseEvent)">`





# Template statements

- TS are not template expressions
  - uses another parser, not template expression (TE) one
  - uses JS-like language
- Syntax
  - assignment =
  - chaining with ;
  - commas
  - not allowed
    - new, ++ and --, +=, -=, | , &
    - TE operators (pipe, Elvis)



# Events cause binding - form

- Local variable binding does not work:
  - template: `  - `<p>{{box.value}}</p>`
- Requires an event tied to the component class
  - template: `  - `<p>{{box.value}}</p>`` `}}`
  - `export class Stub { }`



# Template statements - context

- Can refer to a local template variable object or other alternative context object
  - #localVariable
  - (click)="sendField(localVariable.field)"
- No globals (window, document, console.log, Math.xxx)



# Handling event payload

```
onMessageFromDetail(payload : any[]) {
 var message : string = payload[0] || "";
 var dogActedOn : Dog = payload[1];
 var paidAmount : number = payload[2];
 console.info('Received message', payload[0],
payload[1]);
}
```



# Event binding – no component logic

- Use element API, JS similar to Angular

```
<video #movieplayer ...>
<button (click)="movieplayer.play()"> Play</button>
</video>
```

---

```
<button
onclick="document.getElementById('movieplayer').play
()"> Play </button>
```



# Binding types

- Element event
  - `<button (click) = "onSave();">Save</button>`
  - all web events including packages that add them
- Component event
  - `<hero-detail (deleted)="onHeroDeleted();"></hero-detail>`
- Directive event property
  - `<div (myClick)="clicked=$event;">click me</div>`



# Event propagation

- Child events will bubble up to parent unless binding expression returns falsey
- Will trigger both event handlers

```
<div (click)="showFromParent()">
 <button (click)="showFromChild() || true">
 Show twice
 </button>
</div>
```



# EventEmitter

- an implementation of both the Observable and Observer interfaces
  - use it to fire events, and Angular can use it to listen to events
  - Rx style
- EventEmitter events **don't** bubble





## Event emitting - child → parent

- Emits a Hero object when deleted in hero-detail
  - **heroDeleted** = new EventEmitter<Hero>( );
  - onDelete( ) {
  - this.heroDeleted.emit(this.hero);
  - }
- Listen for deleted event in parent template's child element
  - <hero-detail (**heroDeleted**) =  
  "onHeroDeleted(\$event)" [hero]="currentHero">
  - </hero-detail>



## Event emitting - child → parent

- `@Output() messageFromDetail: EventEmitter = new EventEmitter();`

or

- `outputs: ['messageEvent'],`
- `public messageFromDetail: EventEmitter = new EventEmitter();`



# Exercises

- Click event
- Click event talking to parent
- Accordion
  - p.86 - `section.hidden = true`,  
`clickedContent.hidden = ...`



# Forms

<b>Name *</b>		<b>Time *</b>	
<input type="text"/>	<input type="text"/>	<input type="text"/> :	<input type="text"/>
<small>First</small>	<small>Last</small>	<small>HH</small>	<small>MM</small>
		<input type="button" value="AM PM"/>	
<b>Email *</b>		<b>Date *</b>	
<small>Please use your office email address.</small>			
<input type="text"/>		<input type="text"/>	<input type="text"/>
		<small>MM</small>	<small>DD</small>
		<small>YY</small>	<small>YY</small>
<b>Address *</b>			
<input type="text"/>			
<small>Street Address</small>			
<input type="text"/>			
<small>Street Address Line 2</small>			
<input type="text"/>	<input type="text"/>		
<small>City</small>	<small>State</small>		

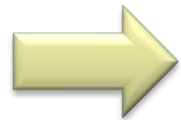


# Form submit strategy

- If a form has only one input field then hitting enter in this field triggers form submit (ngSubmit)
- if a form has 2+ input fields and no buttons or `input[type=submit]` then hitting enter doesn't trigger submit
- if a form has one or more input fields and one or more buttons or `input[type=submit]` then hitting enter in any of the input fields will trigger the click handler on the first button or input `type=submit` and a submit handler on the enclosing form (ngSubmit)



# Form submit process



- Input (model)
- Create form merging model data
  - state pristine
- User enters data
  - state dirty
- Validate data by field
  - state valid/invalid, show/clear error messages
- User submits data



- Output (ngForm)



# Form management strategies

- manual binding
  - inputs are bound to local variables
- **template-driven**
  - inputs are 2-way bound to ngModel
  - build forms with very little to none application code required
- **reactive** / model-driven
  - ngFormModel
  - testability without a DOM being required
- reactive with FormBuilder



# Manual - declare local variables

- kebab-case is not allowed

```
<input #nameLast class='aClass' />
<input #nameLast
(keyup)='showValue(nameLast.value)' />

{{nameLast.value}}
{{nameLast.className}}
```





# Manual - binding

- 3 steps
    - Declare local variables for arguments
      - `<input name="title" #articleTitle>`
      - `<input name="link" #articleLink>`
    - Bind a method to a trigger
      - `<button (click)="addArticle(articleTitle, articleLink)">add</button>`
    - Implement logic in Component
      - `addArticle(titleIn, linkIn) {`
      - `console.log("title=", titleIn.value, "link=", linkIn.value);`
      - `}`
-



# FormsModule

- NgModel
  - binds an ngModel object to the element
- NgForm - class
  - automatically attached to any form elements
  - provides FormGroup ngForm
  - ngForm is often aliased on a page with #f = ngForm
  - provides (ngSubmit) event binding to use with onSubmit( )
    - (ngSubmit)="onSubmit(f.value)"



# Template - [(ngModel)] binding

- `<input [(ngModel)]="name.first" >`
- `Hello, {{ name.first }}`
- 2 way
  - combines property and event binding
  - updates model object on any change/input event
    - input, select, textarea
  - model updates any template reference



# Template - [(ngModel)] binding

- The double binding
  - `<input type="text" [(ngModel)]="model.name" >`
- is equal to two one-way bindings of
  - `<input type="text" [ngModel]="model.name" (ngModelChange)="model.name = $event" >`
- which may be expanded if necessary
  - `<input type="text" [ngModel]="model.name" (ngModelChange)="model.name = validate($event)" >`
- `$event.value` or `$event.target.value` may be needed



# ngForm

- exposes directive instances in template
  - uses @Component – exportAs property
- tells Angular how to link a local variable to that directive
- used in ngForm (a family of directives)
  - `<form #form="ngForm">`
  - `<form #form="ngForm" (ngSubmit)="logForm(form.value)">`



# ngForm + ngControl

- `<form #heroForm="ngForm">`
  - sets local variable heroForm to Angular's form directive for this element
  - collects Controls (anything with `ngControl = ...`), monitors properties
- `heroForm.valid` is now usable as a property
  - `<button [disabled]="!heroForm.valid" >`
  - `<div [hidden]="!heroForm.valid">All fields are valid</div>`



# ngControl – form Control

- `<input type="text" ngControl="username" />`
  - Registers input as username in ngForm
  - updates validity state
- `<input type="text" [(ngModel)]= "model.name" ngControl="nameLocal" #nameLocal>`
  - gets initial model data, sets model data when input



# ngControl – form Control

- After ngControl assignment, access on form with local variable
  - {{model.name}}
  - {{formControlName.value.nameLocal}}
- access from directive with submitted form's fields
  - (ngSubmit)="submittingForm(nameAddressForm);"
  - formSubmitted.value.nameLocal





# formGroup– access to controls

- access through find('ngFormControl's name')
  - `<input type="text" [ngFormControl]="nameFirst">`
  - `[class.error]="!myForm.find('nameFirst').valid && myForm.find('nameFirst').touched"`
- access by directive
  - `<input type="text" #nameFirst="ngForm" [ngFormControl]="myForm.controls['nameFirst']">`
  - `#nameFirst` is an instance of the directive, not a Control
  - `<div *ngIf="!nameFirst.control.valid" class="error">Bad first name</div>`
  - `<div *ngIf="nameFirst.control.hasError('required')" class="error">Required</div>`



# Controls

- an imperative `ngControl`
- Bound to an input element, takes 3 arguments (all optional)
  - default value, validator, asynchronous validator.
- Validation state is determined by optional validation functions
- `this.username = new Control('Default value', Validators.required, UsernameValidator.checkIfAvailable);`



# FormGroups

- part of a form that contain Controls
- valid if all of the children Controls are also valid
- the form is a FormGroup
  - [formGroup]="thisGroupOfFormControls"
- **let** personGroup = **new** FormGroup(
  - nameFirst: **new** FormControl("Doug"),
  - nameLast: **new** FormControl("Hoff"),
  - zip: **new** FormControl("64152")
- **})**



# FormGroups validity properties

```
personGroup.value
personGroup.errors
personGroup.dirty
personGroup.valid
```



# Reactive form binding

- binds an existing FormGroup to DOM element

```
this.userForm = this._formBuilder.group({
 'email': ['', Validators.required],

```

```
<form [formGroup]="userForm">
<input formControlName="email" #emailE />
<div [hidden]="emailE.valid">Invalid</div>
```



# Reactive - FormBuilder

- Dependency injection through constructor
- Class field to reuse injected service

```
private builder: FormBuilder;

constructor(builder: FormBuilder) {
 this.builder = builder;
}
```



# Reactive - FormBuilder group()

- creates a FormGroup using a map

```
this.nameAddressFormGroup = this.builder.group({
 'first': [this.name.first, Validators.required],
 ...
});
```

// instead of

```
this.nameAddressFormGroup = new FormGroup({
 'first': new FormControl(this.name.first,
 Validators.required), ...
});
```



# Reactive - FormBuilder control()

- creates a Control with value, validator and asyncValidator

```
control(value: Object, validator?: ValidatorFn,
asyncValidator?: AsyncValidatorFn)
```





# Reactive - FormBuilder array()

- creates an array of Controls from a controlsConfig array.

```
array(controlsConfig: any[], validator?:
ValidatorFn, asyncValidator?: AsyncValidatorFn)
```



# A drop down

- [ngValue] can replace [value] and [selected] when API is known

```
private arrayOfStuff = [
 {k:1, v:'choice 1'},
 {k:2, v:'choice 2'},
 {k:3, v:'choice 3'},
 {k:4, v:'choice 4'}];
```

```
<select name='s' [(ngModel)]="model?.choice" >
 <option *ngFor="let item of arrayOfStuff"
 [value]="item.k" [selected]="item.k === model.choice">
 {{item.v}}
 </option>
</select>
```



# Radio buttons

```
class MyComp { food = 'fish'; }

<form #f="ngForm">
 <input type="radio" name="food" [(ngModel)]="food"
value="chicken">
 <input type="radio" name="food" [(ngModel)]="food"
value="fish">
</form>
```



# Submit

- requires function in class methods
- submit does not store isSubmitted state
- ngSubmit sets isSubmitted to true
  - hide form with `[hidden]="isSubmitted"`
  - show form again with `<button (click)="isSubmitted=false">`

```
<form (ngSubmit)="storeFormData()" #myForm="ngForm">
```



# Submit - data

- call submit function with
  - (myForm), (myForm.value)
  - or (myForm.value, myForm.valid) for double-check

```
<form (ngSubmit)="storeFormData(myForm)"
#myForm="ngForm" [hidden]="isSubmitted">
```

```

--
```

```
storeFormData(submittedForm) {
 console.log(submittedForm.value || 'no data
submitted');
 console.log('name =', submittedForm.value.name);
}
```



# Submit button

- will trigger submit on form

```
<button type='submit' [disabled]='form.invalid'>
 Make changes
</button>
```



# Change detection strategy - OnPush

- A check to make sure things haven't changed isn't necessary if nothing has changed
  - won't re-render the component unless the input property has changed

```
import {Component, ChangeDetectionStrategy} from
'angular2/core';
```

```
@Component({ ...
 changeDetection: ChangeDetectionStrategy.OnPush
... })
```



# Reset model

- Create a `reset( )` function that sets the values of your model to whatever you want
  - `{ this.string = ""; this.int = 0... }`
- Call it/them in your submit function





# Reset form

- Controls must be manually reset
  - `this.loginForm.controls['username'].updateValue('')`
  - `this.loginForm.controls['password'].updateValue('');`
- ISSUE -  
<https://github.com/angular/angular/issues/4933>
- 2.4.3 fixed?



# Exercises

- 27. Form app setup – **do**
- 28. Submitting with local variables – **optional**
  - `$event.preventDefault();`
- 29. Binding to ngModel – template driven - **optional**
- 30. Binding to ngForm – template driven – **optional**
- 31. Binding to ngFormModel – reactive forms - **do**



# Form validation

**Name \***

First

Last

**Time \***

:

**Email \***

Please use your office email address.

**Date \***

/

/

**Address \***

Street Address

Street Address Line 2

City

Zip



# Validation – HTML5

- Uses :invalid, :valid pseudo-classes
- Browser blocks the form, displays error message.

```
<form novalidate>
```

- `<input type="text" ngControl="name" required>`
- `<input type="text" ngControl="street" minlength="3">`
- `<input type="text" ngControl="city" maxlength="10">`
- `<input type="text" ngControl="zip" pattern="[A-Za-z]{5}">`

```
</form>
```



# Validity state - CSS

- updated by ngModel / ngControl managed fields
- original state
  - class = '**ng-untouched ng-pristine ng-valid**'
- click out (blur)
  - class = '**ng-touched** ng-pristine ng-valid'
- change data
  - class = 'ng-touched **ng-dirty** ng-valid'
- erase data with required attribute
  - class = 'ng-touched ng-dirty **ng-invalid**'
  - also for Form Builder validity



# Validity state – field properties

- boolean
  - valid, invalid – passes rule
  - pristine, dirty – value change
  - touched, untouched – field visited, not for form
  - pending
- non-boolean
  - errors
  - status
  - root (parent groupControl)



# Validity state – hide when valid

- in form (using font-awesome icons)

```
<div [hidden]="name.valid" class="alert alert-
danger">
 <i class="fa fa-exclamation-triangle"></i>
 Name is required
</div>
```



# Validity - error messages

- `hasError( )` in component or `*ngIf`
- `localVar.hasError('required')`
  - `<div *ngIf="nameFirst.hasError('required')"`  
`class="error">First name is required</div>`
- `form.hasError('required', 'localVar')`
  - looks up error in form
  - `<div *ngIf="aForm.hasError('required',`  
`'nameFirst')"` `class="error">First name is`  
`required</div>`





# Validity - update by lifecycle

- a hack

```
{{updateValidState(nameAddressFormGroup)}}

[class.hide]='whenValidFor.first'

updateValidState(groupControl) {
 this.whenValidFor = {
 first: groupControl.controls.first.valid ||
 groupControl.controls.first.pristine
 };
}
```



# Validity - update with valueChanges

- valueChanges is an EventEmitter
- get reference to Control (value is String), ControlGroup or form (value is any)

```
constructor() {
 this.ref.valueChanges.subscribe(
 (newValue: string) => { // do stuff }
);
}
```



# Validators – built-in

```
min(#)
max(#)
required()
requiredTrue()
email()
minLength(#)
maxLength(#)
pattern('regex string')
nullValidator()
```



# Validators – built-in

- Validator added only to template
- ngModel collects errors for this element

```
<input required type='text' name='name'
#name='ngModel' />
<p *ngIf="name.errors">
 {{ name.errors | json }}
</p>
```



# Validators – built-in

- Validator added to component Control
- Errors do not show on load

```
this.name = new Control('default name',
Validators.minLength(4));
```

```

<input required type="text" ngControl="name" />
<p *ngIf="name.errors?.minlength && name.dirty">
 Your name needs to be at least 4 characters.
</p>
```



# Validators - composed

- also `Validators.composeAsync`

```
myForm = new FormGroup({
 name: new FormControl('Nancy',
 [Validators.required,
 Validators.maxLength(4)]
});
```



# Validators - custom

- `f(control) { // check control.value and return }`
  - return null when validation is valid
  - return object when validation needs error message
    - if ... return { "invalidDigitAtStart": true }
    - if ... return { "invalidEmail": true }

```
interface Validator<T extends Control> {
 (c:T): {[error: string]:any};
}
```



# Validators - custom

- touched can replace dirty

```
<input required type="text" ngControl="name" />
<div *ngIf="name.dirty && !name.valid">
 <p *ngIf="name.errors?.invalidDigitAtStart">
 Your name can't start with a number
 </p>
</div>
```

```
<input required type="text" ngControl="name" />
<p *ngIf="name.errors?.invalidDigitAtStart &&
name.dirty ">
 Your name can't start with a number
</p>
```





# Validators – template vs. reactive

- directive / template
  - allows form only `<input ngControl='email' validateEmail>`
  - selector: `'[validateEmail][ngControl]'`
  - add to directives of component it's used in
- reactive / model driven
  - built with `ControlGroup` & `Control` or `FormBuilder`



## NG\_VALIDATORS – adding custom

- NG\_VALIDATORS is a multi provider for a dependency token to provide hooks for custom validators

```
providers: [
 provide(NG_VALIDATORS, {
 useValue: validateEmail,
 multi: true
 })
]
```



# Validators – asynchronous

- check using a Promise (fetching data from the server) with an asynchronous validator.

```
this.name = new Control('',
UsernameValidator.startsWithNumber,
UsernameValidator.usernameTaken);
```



# Exercises

- 32. Use a US state drop down – optional
- 33. Custom validators - do

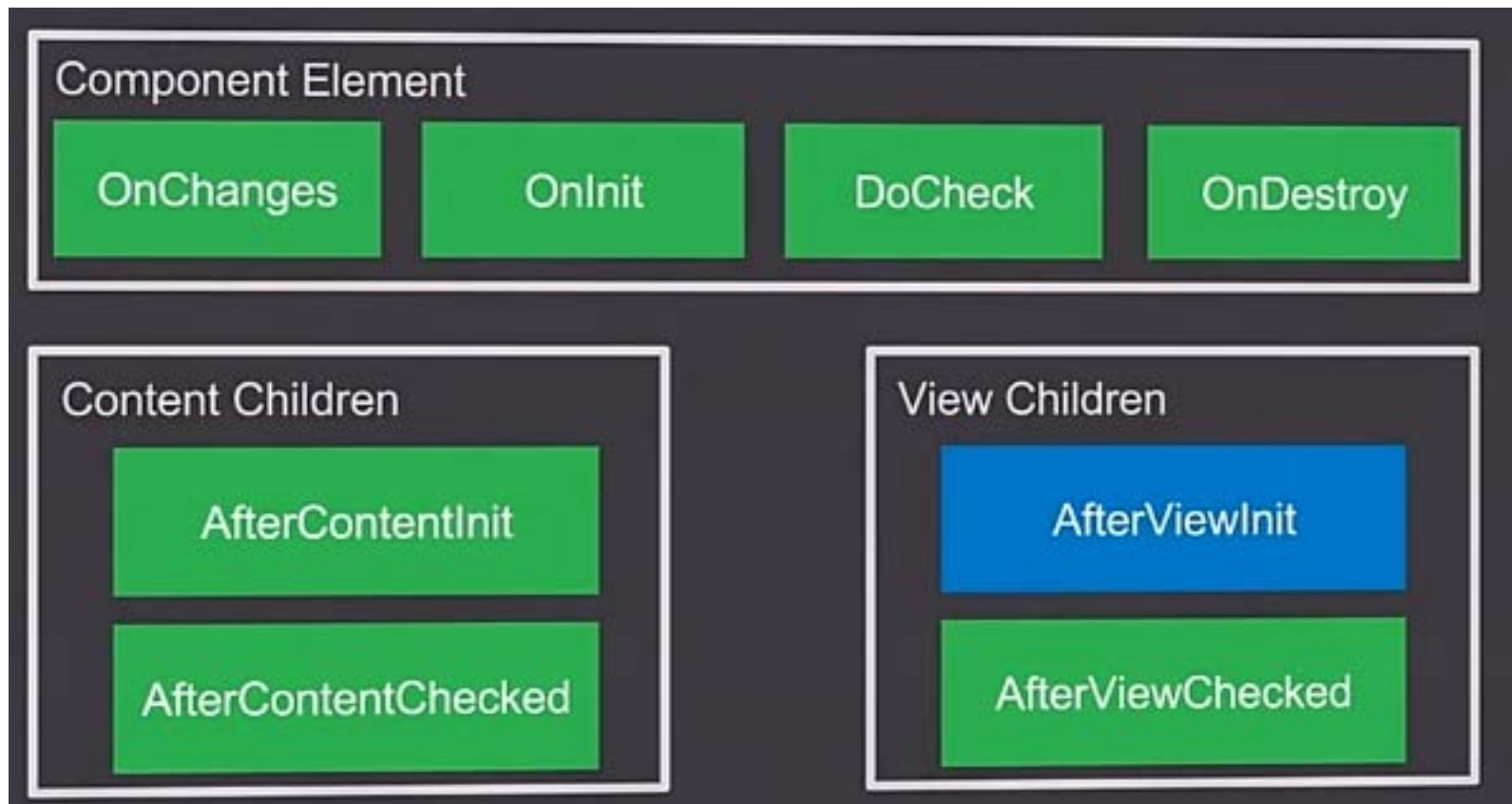


# Lifecycle



# Lifecycle hooks

- Hooks will then call methods if written





# Lifecycle interfaces

- Use interface when implementing method to confirm

Interface	Methods to implement
<b>OnChanges</b>	ngOnChanges - called when an input or output binding value changes
<b>OnInit</b>	ngOnInit - after the first ngOnChanges
<b>DoCheck</b>	ngDoCheck - developer's custom change detection
<b>AfterContentInit</b>	ngAfterContentInit - after component content initialized
<b>AfterContentChecked</b>	ngAfterContentChecked - after every check of component content
<b>AfterViewInit</b>	ngAfterViewInit - after component's view(s) are initialized
<b>AfterViewChecked</b>	ngAfterViewChecked - after every check of a component's view(s)
<b>OnDestroy</b>	ngOnDestroy - just before the directive is destroyed



# Calling order

- called in this order
  - **OnChanges** - called when an input or output binding value changes
    - method called uses hook name with ng prefix (ngOnChanges)
  - **OnInit** – after the first ngOnChanges
  - **DoCheck** - developer's custom change detection
  - **AfterContentInit** - after component content initialized
  - **AfterContentChecked** - after every check of component content
  - **AfterViewInit** - after component's view(s) are initialized
  - **AfterViewChecked** - after every check of a component's view(s)
  - **OnDestroy** - just before the directive is destroyed





# ngOnInit

- Use for initialization logic and not in constructor for testability
- ngOnInit() {
  - this.http.get('/contacts')  
  .map(res => res.json())  
  .subscribe((contacts) => { this.contacts = contacts;  
  });
- }



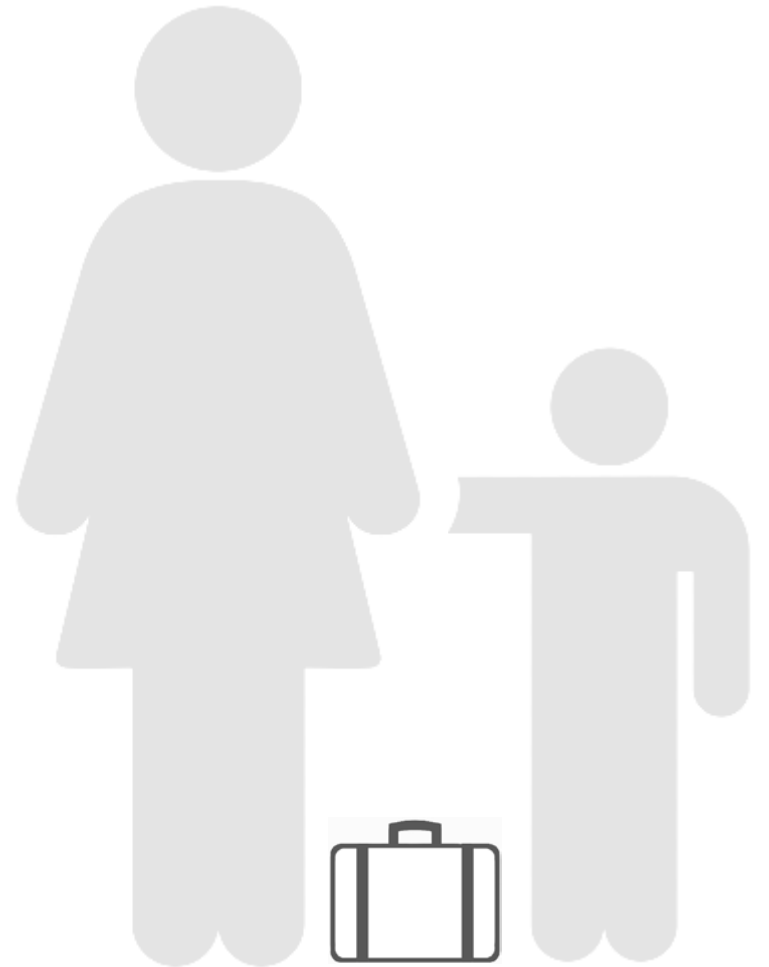
# ngOnChanges

- parent shows all data, child edits one
- detach child being edited from parent displaying value until edit is done

```
// in child
ngOnChanges(changes) {
 if (changes.watchedField) {
 this.watchedField = {...watchedField.currentValue};
 }
}
```



# Services





# Service provider

- any value, function or feature that our application needs
- just a class with a narrow, well-defined purpose
  - logging service
  - data service
  - message bus
  - tax calculator
  - application configuration



# Logger - app/logger.service.ts

```
@Injectable()
export class Logger {
 log(msg: any) { console.log(msg); }
 error(msg: any) { console.error(msg); }
 warn(msg: any) { console.warn(msg); }
}
```



# Dependency injection

- Used mostly with services
- Built-in utility
- Decouples object dependency in constructor
  - `constructor( private _service: HeroService ){ }`
- Reuses created services or creates new
  - singleton
- No need to create class field to store service.



# Dependency injection

- The hierarchical injector looks for anything passed in to the constructor to bring in without needing references.
  - You want it, you get it if it's in any parent above.



# @Injectable()

- decorator before service class
  - when using Http, Jsonp, etc.
- needs parentheses
  - mysterious errors otherwise
- only needed when they have dependencies
- add to any service class – best practice
  - prepare for the future
  - consistent code





# Injecting the service

- In the constructor, the service is passed in
  - `constructor(aService: ServiceClass) {`
  - `x = aService.getSomethingFromService( );`
  - `}`
- and called with
  - `<selector></selector >`
- or test with `new ComponentClass(aTestService)`
- or in the router



# Service dependencies

- A service using a service
- Use annotation – with parentheses!
- `import {MicroService} from './microService.ts'`
- `@Injectable( )`
- `class OrchestratedService {`
  - `constructor(private micro: MicroService) {...}`



# Service provider registration

- Global registration for injection is at the module level
  - `@NgModule({ imports: [...], declarations: [...],`
  - `providers: [ UserService,`
  - `{ provide: APP_CONFIG, useValue:`  
`HERO_DI_CONFIG }]`
- Component registration can happen at `@Component`
  - `@Component({`
  - `providers: [UserService]`
  - `})`



# Registration details

- the registration
  - [HeroService]
- is expanded by Angular to
  - [provide(HeroService, {useClass:HeroService})];
- which creates a Provider object to manage services
  - [new Provider(HeroService, {useClass:HeroService})]
- that associates the reference HeroService to a class with a constructor (a recipe)



# Testing services - class

- Use a testing service when a HeroService is requested
  - beforeEachProviders(() => [
    - provide(HeroService, {**useClass**: MockHeroService});
  - ]);



# Testing services - value providers

- provide a ready-made object instead of pointing to constructor code
- `beforeEachProviders(() => {`
- `let emptyHeroService = { getHeroes: ( ) => [ ] };`
- `return [ provide(HeroService, {useValue:`  
`emptyHeroService}) ];`
- `});`



# Testing services – factory providers

- factory method = replacement method for a constructor providing a pre-configured object

```
let serviceFactory =
 (logger: Logger, aService: OtherService) =>
 new ConfiguredService(logger, aService.property);
```



# Testing services – factory providers

- declaration of provider definition
  - `let serviceDefinition = {`
  - **`useFactory`**: `serviceFactory` ,
  - `deps`: [`Logger`, `OtherService`]
  - `};`





# Testing services – factory providers

- create a provider object and bootstrap it

```
let configuredServiceProvider =
provide(ConfiguredService, serviceDefinition);
bootstrap(AppComponent, [configuredServiceProvider ,
Logger, OtherService]);
```

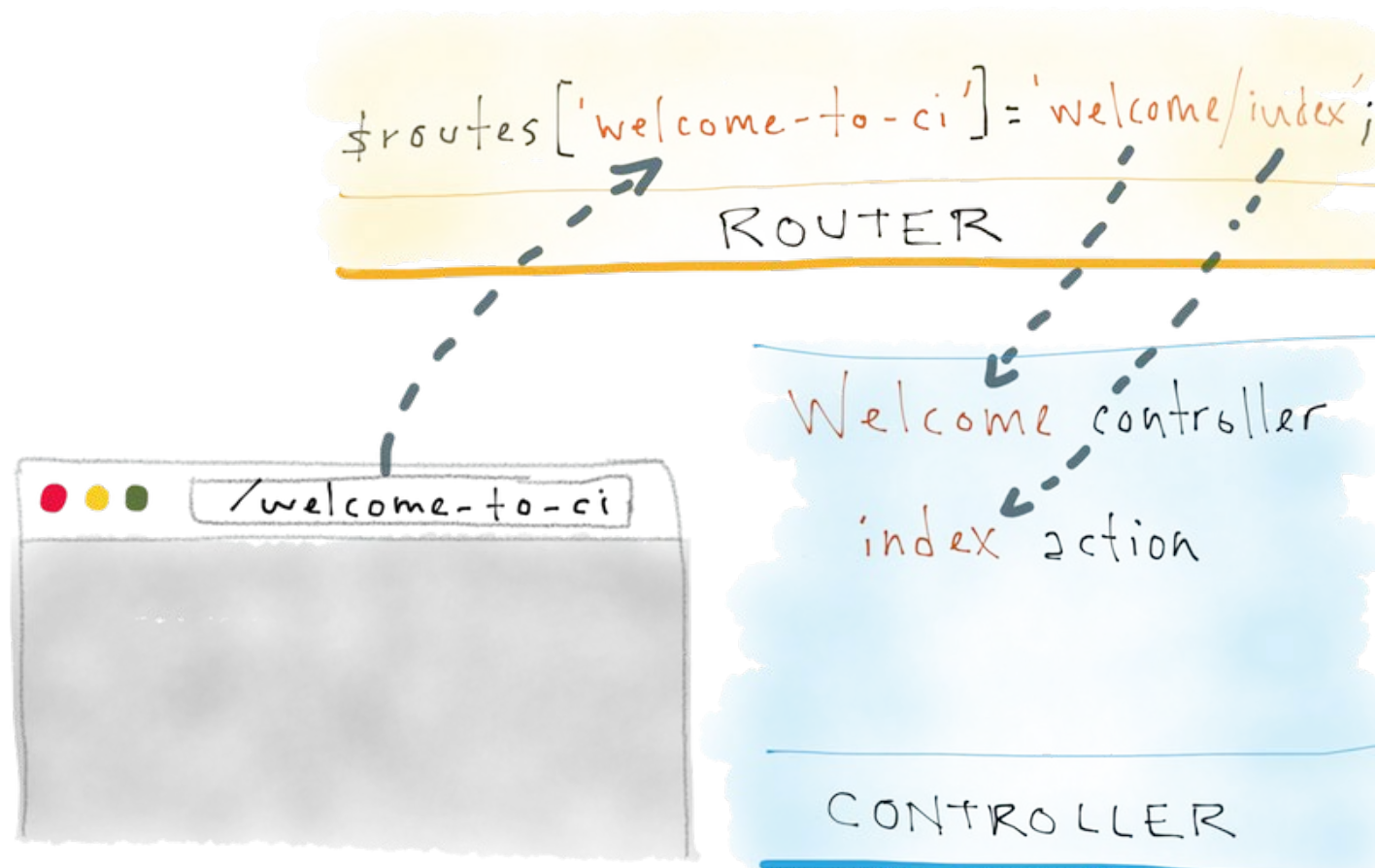


# Exercises

- Using a Wikipedia service
  - `<h1>Smart Wikipedia Search - inspired by  
thoughttram</h1>`
  - `<wikipedia-search></wikipedia-search>`



# Router





# Site design responsibilities

- Server
  - site wide templates
  - repository for site resources
- Single page app
  - reuse site resources – **maintain state**
  - create cohesive units of operation
  - protect app areas by rules



# Router use cases

- Create manageable paths for same page/app

```
http://<domain name>/person // search
http://<domain name>/person/all // show all
http://<domain name>/person/345 // details
http://<domain name>/person/create
http://<domain name>/person/edit/345
http://<domain name>/person/delete/345
```



# Routing – config a routing file

```
import { ModuleWithProviders } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

const ROUTES: Routes = [
 { path: '/home', component: Home }, ...
];

export const APP_ROUTING_PROVIDERS: any[] = [];
export const ROUTING: ModuleWithProviders =
RouterModule.forRoot(ROUTES);
```



# Routing - browser history API

- `history.pushState`
- HTML5 technique for no server page request.
- Router gets a “normal” URL
  - `http://mysite.com/page/crisis-center/`
- Preserves the option to do server-side rendering later
- **best strategy, ng2 default**



# Routing - browser history API

- Add `<base href='/>` to index.html for pushState routing work.
- The browser also uses the base href value to prefix relative URLs when downloading and linking to css files, scripts, and images.





# Routing - hash-based

- IE9 sends page requests to the server when the location URL changes ... unless the change occurs after a "#" (called the "hash").
  - <http://mysite.com/page/#!/crisis-center/>
- think about refreshes, works better
- popstate doesn't fire in IE/Edge on hash change
  - <https://developer.microsoft.com/en-us/microsoft-edge/platform/issues/3740423/>
  - Microsoft will not fix unless security related



# Routing – config routing file for #

```
export const ROUTING: ModuleWithProviders =
 RouterModule.forRoot(ROUTES, {useHash: true})
```



# Module config – forRoot vs forChild

- returns the configured routing service provider
- RouterModule.forRoot(ROUTES)
  - all directives, routes, and router service
  - for app (parent) module – only one
- RouterModule.forChild(ROUTES)
  - all directives, routes, no router service
  - lazy loading
  - for feature (child) modules



# Routing – module config

```
import { AppComponent } from './app.component';
import { ROUTING, APP_ROUTING_PROVIDERS } from
'./app.routing';
import ... app components
@NgModule({
 ● imports: [BrowserModule, FormsModule, ROUTING],
 ● declarations: [AppComponent, ...],
 ● providers: [APP_ROUTING_PROVIDERS],
 ● bootstrap: [AppComponent]
})
```



# Routes - ROUTES

- no leading slashes in path
- **const** ROUTES: Routes = [
  - { path: 'home', component: HomeComponent },
  - { path: 'about', component: AboutComponent },
  - { path: 'contact', component: ContactComponent },
  - { path: 'contactus', redirectTo: 'contact' },
  - { path: '', redirectTo: 'home', pathMatch: 'full' }
- ];



# Routes - default & wildcard paths

- Use empty string for default path
  - { path: '', component: HomeComponent },
- A wildcard path
  - { path: '\*\*', component: PageNotFoundComponent }
- Place more specific paths first. First match wins.



# Routes – redirect

- A route to a route is a redirect.
- { path: 'contactus', redirectTo: 'contact', pathMatch: 'full' }
- { path: '', redirectTo: '/inbox' , pathMatch: 'full'},
- Requires a pathMatch property to tell the router how to match a URL to the path of a route.
  - full = exact match
  - 'prefix' = remaining URL begins with the redirect route's prefix path.



# Routes – query parameters

- { path: 'hero/:id', component: HeroDetailComponent }
- implies required data (**id**)





# Routes – child routes

- `const crisisCenterRoutes: Routes = [{`
- `path: 'crisis-center', component:`  
`CrisisCenterComponent,`
  - `children: [{`
    - `path: '', component: CrisisListComponent,`
    - `children: [`
      - `{ path: ':id', component: CrisisDetailComponent},`
      - `{ path: '', component: CrisisCenterHomeComponent }]`
- `}]`
- `}]`;



# Route parameter strategies

- Possible strategies
  - path: '/rk/:id'
  - path: '/rk/:pk/:fk'
  - path: '/rk/:operation/:id'
  - path: '/rk/:type/:filter'



## Routes – route data

- Data only associated with this route
- { path: 'heroes', component: HeroListComponent,
- **data:** {
- title: 'Heroes List'
- }
- },



# Outlets - <router-outlet>

- The portal for the requested partial view
  - An import of child elements, a viewport
- A Component will render a router output in a RouterOutlet object in the browser
- A template may hold only one unnamed <router-outlet>
- The router supports multiple named outlets.

```
<router-outlet></router-outlet>
```



# Links to routes

```
<a [routerLink]="/crisis-center"> Crisis Center
<a [routerLink]= {{array of link parameters for
complex path}}> Crisis Center
```



# Links to routes – parameter array

```
<a [routerLink]="['/hero', hero.id]"> Crisis Center

```

```

constructor(router: Router) {
}
onSelect(hero: Hero) {
 this.router.navigate(['/hero', hero.id]);
}
```



# Links to routes - routerLinkActive

- Name of CSS class to use on link when activated
  - `<a routerLink="/crisis-center" routerLinkActive="active"> Crisis Center </a>`
- Multiple classes allowed
  - `routerLinkActive="active red"`
  - `[routerLinkActive]="['active', 'red']"`



# Links to routes - routerLinkActive

- RouterLinkActive directive manages parent and child router links
  - both can be active at the same time
- Override by binding to the [ ] input binding with {exact: true} routerLinkActiveOptions
  - routerLinks must be exact matches
  - `<a routerLink ="/user/bob" routerLinkActive="active-link" [routerLinkActiveOptions]="{exact: true}">`



# Links to routes – routerLinkActiveOptions



- Affects multiple links
- Makes sure class is only added for link not previous click

```
<nav
routerLinkActive="active-link"
[routerLinkActiveOptions]="{exact: true}">
 <a [routerLink]="/user/jim">Jim
 <a [routerLink]="/user/bob">Bob
</nav>
```



# Links to routes – query strings

- [queryParams] binding takes an object
  - { name: 'value' })
- adds any leftover key-values after path uses them
  - `<a [routerLink]="['RoutingKids', {id:1, type='s'}]">`
  - imply optional data (**type**)
  - arrays are comma separated in URL but retrieved as an array



## Links to routes – link parameters

- link parameters array supports a directory-like syntax for relative navigation.
- ./ or no leading slash is relative to the current level.
- ../ to go up one level in the route path.



# Books

- **Angular 2 Router - The Complete Authoritative Reference** by Victor Savkin (main router guy on Angular team)
- 91 pages
- <https://leanpub.com/router>
- \$20+ Sep 2016

## ANGULAR 2 VICTOR SAVKIN ROUTER





# Slides

- Route lazy loading by Victor Savkin
  - <https://docs.google.com/presentation/d/1kp7sbxcEpTaOEgW95RHMFmXsihGdk-8Nlug62PDjgFw/edit#slide=id.p>

# Exercises

- Simple routing





# Resources



# Articles / lectures / products

- Ionic – Angular + Cordova
  - <http://learnangular2.com/>
- Anything on InfoQ
  - <http://www.infoq.com/search.action?queryString=angular&page=1&searchOrder=date&sst=8JV2v3VZulP9pHJt>





# Conferences

- ng-conf
  - April 18-20, 2018
  - <http://ng-conf.org/>
  - videos -  
<https://www.youtube.com/user/ngconfvideos/videos>
- AngularConnect – Europe
  - Nov 6-7, 2018
  - <https://www.angularconnect.com/>



# Blogs

- Victor Savkin – <http://victorsavkin.com>
- Scotch.io - <https://scotch.io/tag/angular-js>
- Thoughttram - <http://blog.thoughttram.io/>
  - <http://blog.thoughttram.io/exploring-angular-2/>

# Evaluation



- [http://www.metrics\*\*that\*\*matter.com/centriq1](http://www.metrics<b>that</b>matter.com/centriq1)