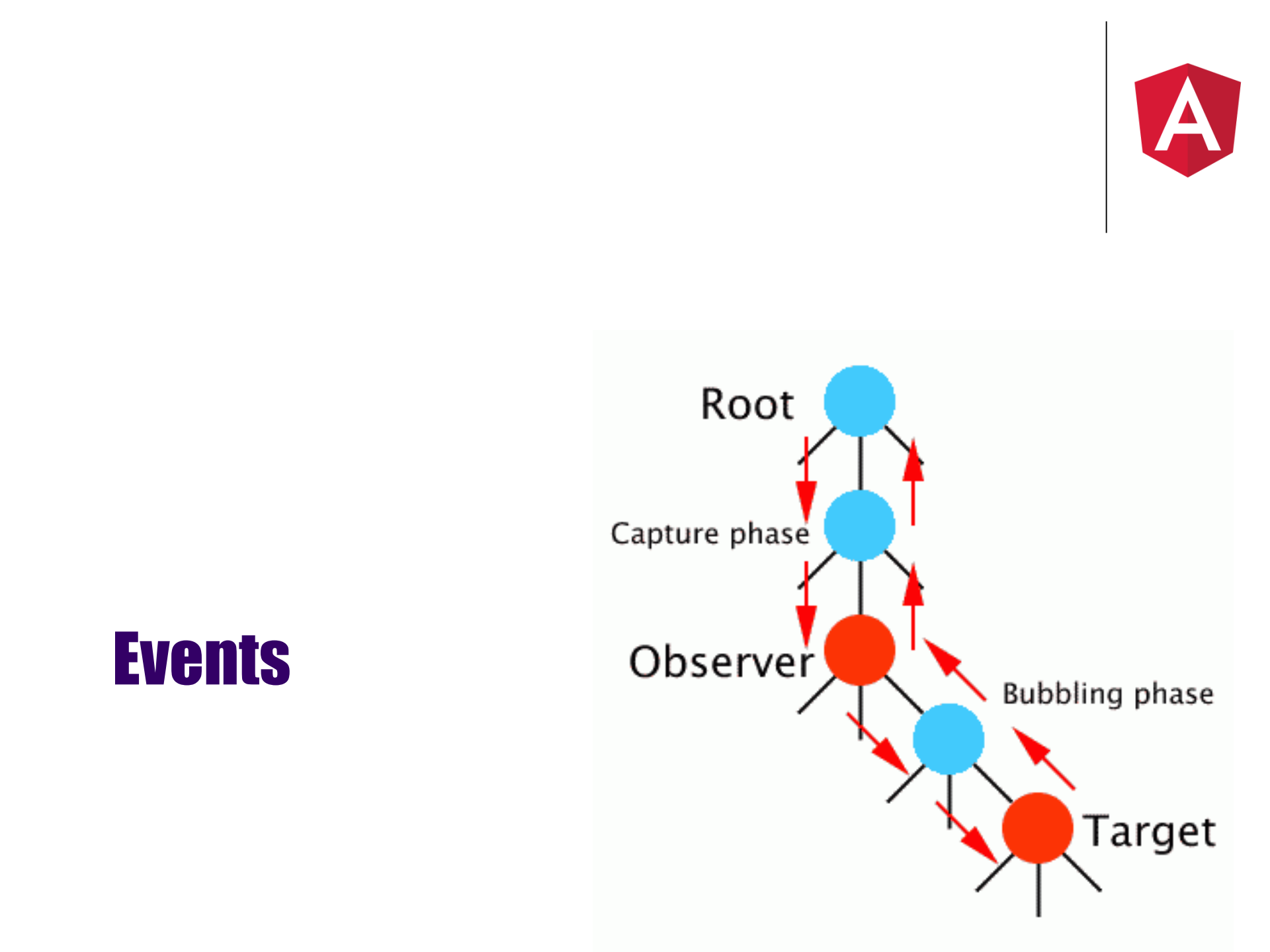
Angular

Building the next version of the web with browser applications

CENTRIQ TRAINING



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, scrolllock, 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: `<input #box >
  - {{box.value}} `
- Requires an event tied to the component class
  - template: `<input #box (keyup)="undefined">
  - {{box.value}} ` })
  - 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[ ]) {
        let message : string = payload[0] || "";
        let dogActedOn : Dog = payload[1];
        let 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

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();

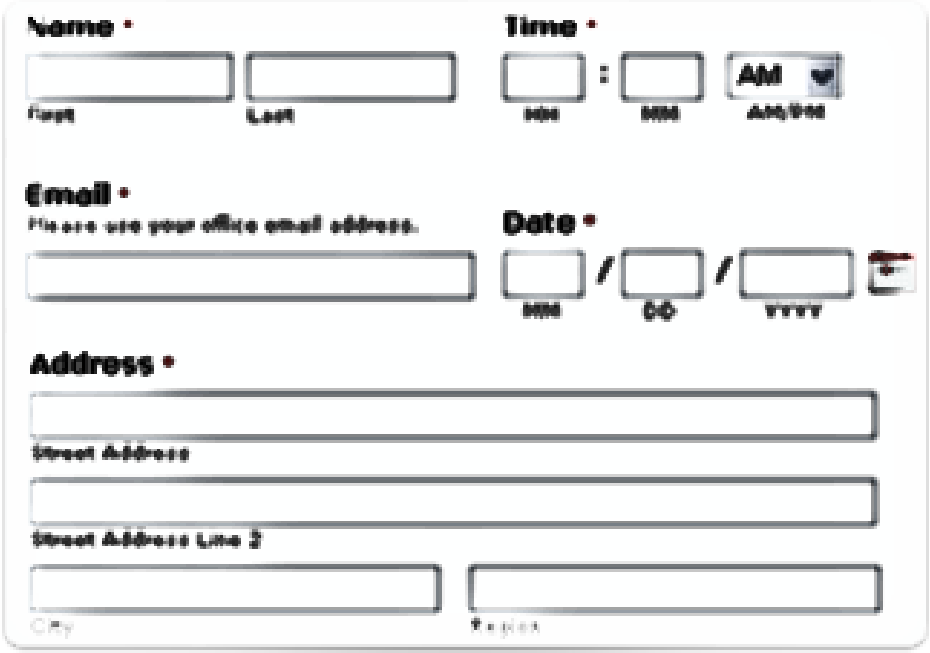
or

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

Exercises

- Click event
- Click event talking to parent
- Accordion

Forms


Name *
First
Last
Time *
hh
mm
AM
Email *
Date *
mm
dd
yyyy
Address *
Street Address
Street Address Line 2
City

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)

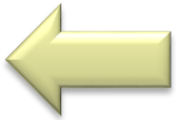
Form submit process

- 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

- 2 way
  - combines property and event binding
  - updates model object on any change/input event
    - input, select, textarea
  - model updates any template reference

```
<input [(ngModel)]="name.first" >
Hello, {{ name.first }}
```

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

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 *ngIf="nameFirst.control.hasError('required')" class="error">Required

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 ControlGroup 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]="db?.choice" >
<option *ngFor="let item of arrayOfStuff"
[value]="item.k" [selected]="item.k === db.choice">
   {{item.v}}
</option>
</select>
```

Radio buttons

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


<form #f="ngForm">
  <input type="radio" name="food" [ngModel]="food"
value="chicken">CHICKEN
  <input type="radio" name="food" [ngModel]="food"
value="fish">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?

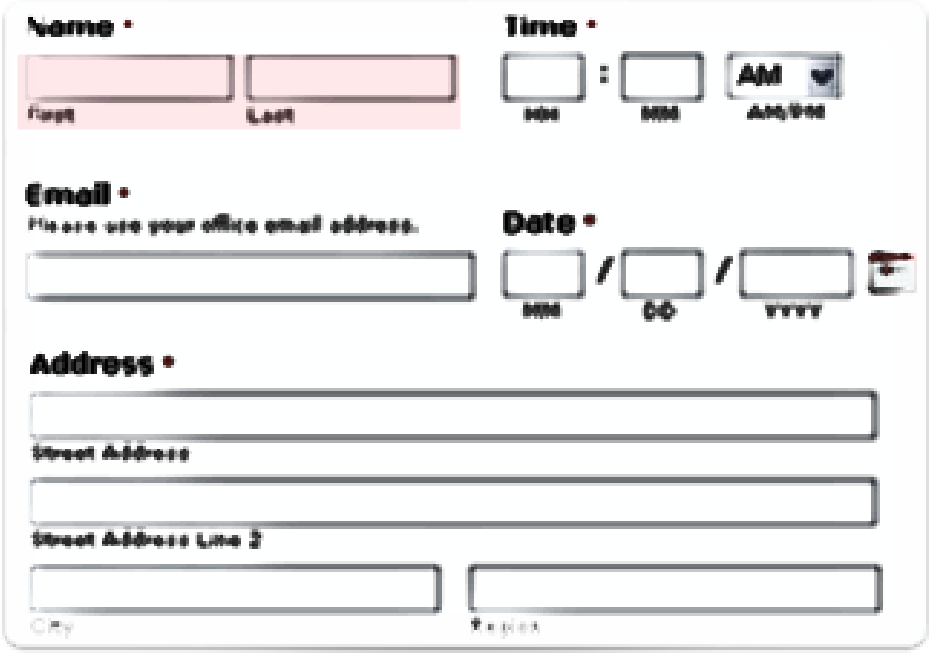
Exercises

- 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



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="[0-9]{5}(-[0-9]{4})?">
</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" name='name'
[ngModel]="name" #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
- @directive metadata

```
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);
```

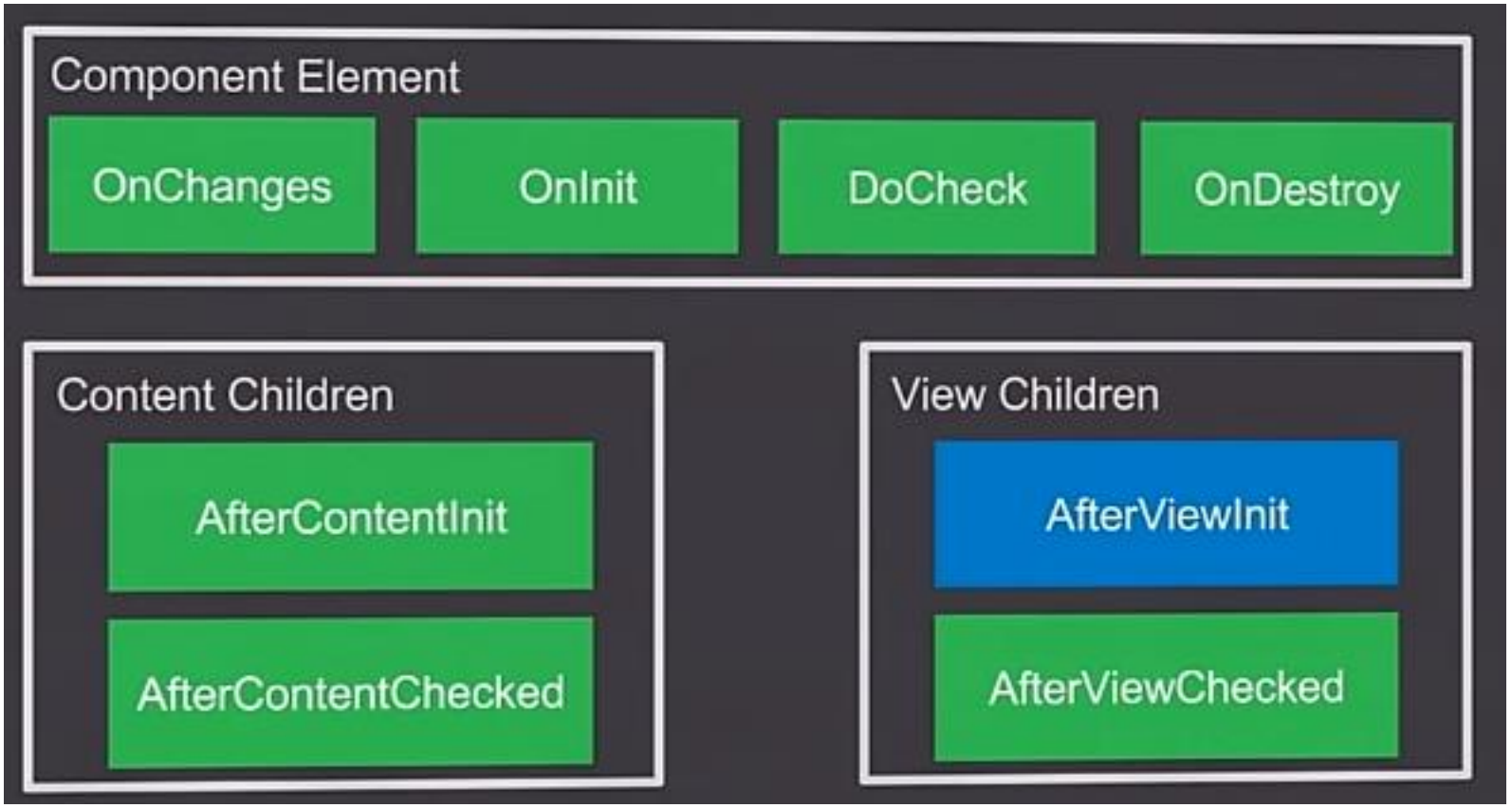
Exercises

- 32. Use a US state drop down – optional

- 33. Custom validators – **do**
  - **Utilities** should be **ValidationUtilities**

Lifecycle

Lifecycle hooks



- Hooks will then call methods if written

Lifecycle interfaces

- Use interface when implementing method to confirm

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

- An example custom service

```
@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: MyDataService){ }
- 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: MYDATA_DI_CONFIG }
- Component registration can happen at @Component
  - @Component({
  - **providers: [UserService]**
  - })

Registration details

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

Testing services - class

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

Testing services - value providers

- provide a ready-made object instead of pointing to constructor code
- beforeEachProviders(() => {
- let emptyDataService = { getData: ( ) => [ ] };
- return [provide(MyDataService, {**useValue**: emptyDataService})];
- });

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

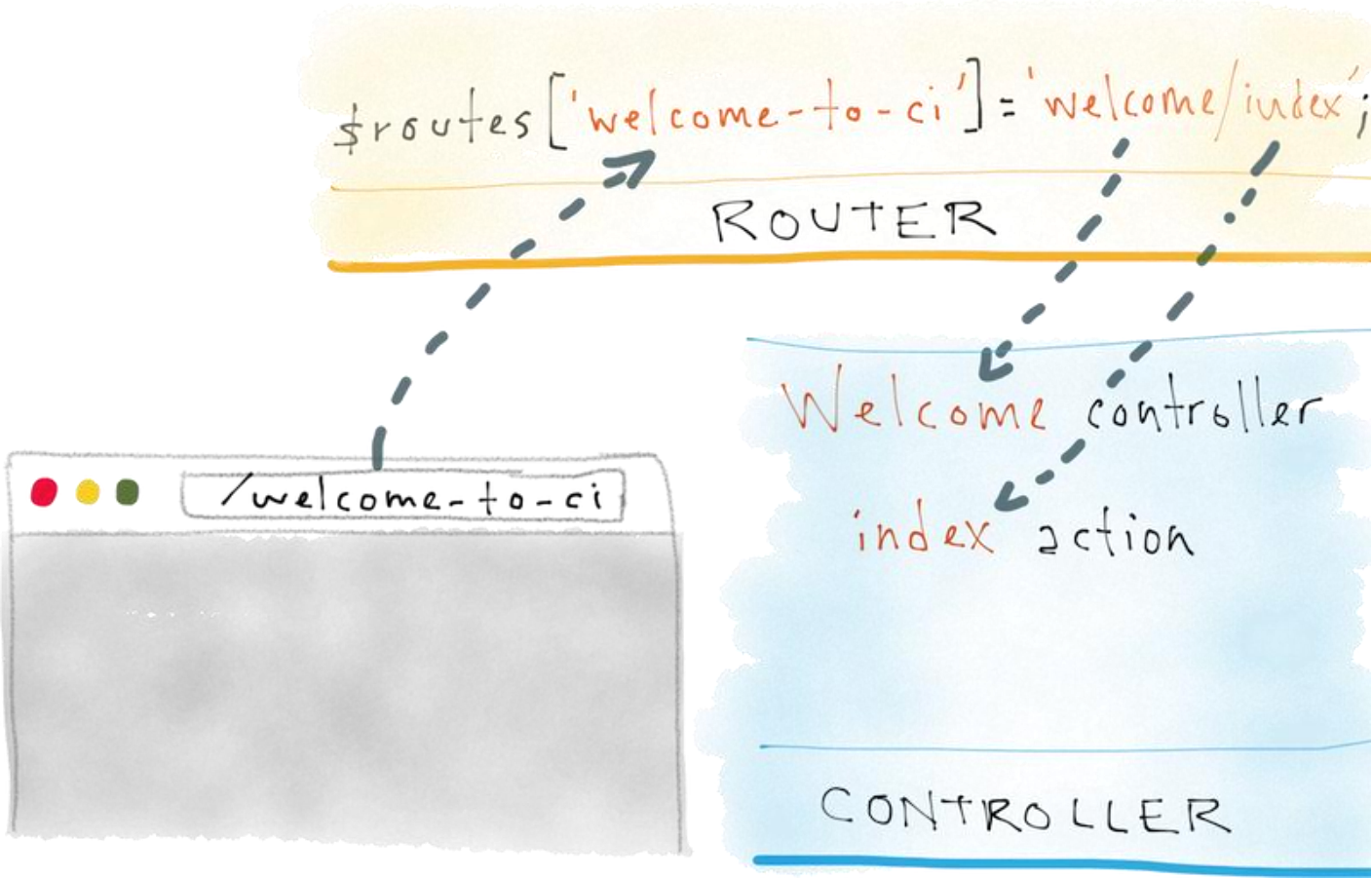
```
let configuredServiceProvider =
provide(ConfiguredService, serviceDefinition);
bootstrap(AppComponent, [configuredServiceProvider ,
Logger, OtherService]);
```

Exercises

- Using a Wikipedia service
  - <h1>Smart Wikipedia Search - inspired by thoughtram</h1>
  - add
  - <wikipedia-search></wikipedia-search>

  - private items: Observable<Array<string>>
  - change to
  - private items: Observable<any>

Router


$routes['welcome-to-ci'] = 'welcome/index';
ROUTER
/welcome-to-ci
Welcome controller
index action
CONTROLLER

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
```

app.routing.ts

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 with IE/Edge

- 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

app.routing.ts

Routing – hash-based with IE/Edge

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

Links to routes – parameter array

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

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 –
  - https://blog.nrwl.io/
  - http://victorsavkin.com – not maintained now
- Scotch
  - https://scotch.io/tag/angular-js
- Thoughtram
  - http://blog.thoughtram.io/
  - http://blog.thoughtram.io/exploring-angular-2/

Evaluation

- http://www.metrics**that**matter.com/centriq1