Angular 5 Workshop

from: **Saban Ünlü**

for: **Bitmarck**

Two words about me...

Saban Ünlü

- Software architect and programmer
- Consultant and Trainer for web technologies since 2000
- Author
- Adobe Influencer
- Founder of netTrek

Introduction

Introduction

- What's Angular?
- Angular: Highlights
- What's new?
- Differences to AngularJS 1x
- technologies
- Polyfills and Vendors
- Angular-Modules

What's Angular?

- Framework for Single Page Application
  - Component-based applications inspired by web components
- Modular working
- Separation of logic and view

Angular Highlights

- Templates
- Binding
- Services
- Dependency Injection
- Routing
- Forms

What is new?

- Angular is not a classic update
  - Completely rewritten
- focus
  - Performance (3-5 times faster than Angular 1)
  - Component-based applications

Differences to AngularJS 1x

- Components Instead of Controllers
- Scopes are deprecated and could not used anymore.

Differences to AngularJS 1x

- NG1 Development with ES5, Dart or TypeScript
- Angular was developed with TypeScript
  - ES2015: Classes, Interfaces and Inheritance
  - ES2015: Templates
  - Typed
  - annotations

Differences to AngularJS 1x

- Angular is ES2015 compliant because of TypeScript
- TypeScript outputs code for current browsers in ES5
- ES2015 Polyfills for "less modern" browsers
- System for loading and managing modules, e. g.:
  - System. js
  - webpack via angular cli

Technologies - Overview

core js SystemJS RxJS reflect-metadata

Technologies

- JavaScript runtime environment for various operating systems
- Versioning system for software (GitHub - Filehoster)
- ES2015-based programming language

Polyfills

- core-js
  - ES2015/ES6 Polyfills
- web-animations-js
  - Firefox Animationen
- intl (ng4 for ng5+ use locales)
  - I18n für Internationalisierung

Vendors

- RxJS
  - Library to monitor events and asynchronous processes. Used for Angular e.g. HTTP calls.
- zone.js
  - Similar Domains in Node: Execution context allows to monitor and control execution.

Vendors

- reflect-metadata
  - Add metadata consistently to a class
- systemjs (if webpack not in use)
  - Modul Loader for ES2015/ES6 Module

Angular Modules

- @angular/core
  - Necessary for every application - core for components, directives, dependency injection and component lifecycle
- @angular/common
  - Commonly used Directives, Pipes and Services

Angular Modules

- @angular/compiler
  - Combines logic with templates (JIT) - compiler is automatically triggered via platform-browser-dynamic
- @angular/platform-browser
  - Browser and DOM-relevant components, especially for rendering new elements. Used for AOT Builds

Angular Modules

- @angular/platform-browser-dynamic
  - Has the bootstrapping method for JIT Builds
- @angular/http
  - Module for HTTP calls
- @angular/router
  - Module for component routing

Angular Modules

- @angular/animate
  - Animations in the Angular context

TypeScript excursus

TypeScript excursus

- var, let, const
  - types
    - native, class, interface, own
- arrow function. ()=>{}
  - scope

TypeScript excursus

- parameter
  - default (param: boolean = true)
  - optional (param?: boolean)
  - rest (param: boolean = true)

TypeScript excursus

- class ES5 vs TypeScript
  - extends
  - interfaces
  - abstract class

TypeScript excursus

- Syntax magic (ES6/TS)
  - private, public definition in constructor
  - Concat Array
  - Object Assign
  - Destructing

project Setup

First steps

- Mac
  - XCODE installation
  - node.js installation (>= 6.9.x)
- Win
  - node.js installation (>= 6.9.x)
  - Git installation (Bash also)

Setup Manually (not recommend)

- Initialize node
- Installing Dependencies
- Configure TypeScript
- Configuring
  - SystemJS
  - Webpack

Seed Setup

- git clone https://github.com/angular/quickstart.git myProject
- npm install

angular-cli

- ng new bitmarck --prefix=bm --style=scss —routing=true
- cd gfn
- ng serve
- ng g m commonUi
- cd common-ui/
- ng g c user

Architecture

Architecture

- modules
- components
- bootstrap
- directives
- pipes
- data binding
- Dependency Injection (DI)
- services
- router

Module

- Not comparable to JavaScript modules
- Combining functions and features in a black box
- Extend application and own modules with external modules
- Let compilers know which elements to look for in a module

Module

Available Angular modules

- BrowserModules (events, DOM)
- CommonModule (Directives, Pipes)
- HttpModule (ng2+) & HttpClientModule (ng5+) (XHR)
- FormsModules (Forms)
- RouterModule (component router)

Module


```
class AppModule {}
```

Module

Module development

- Create a Module-class
- described method, properties & logic is in a class

Module

```
@NgModule({
  imports:      [ BrowserModule ]
})
export class AppModule {}
```

Module

```
@NgModule({
  imports:      [ BrowserModule ] ,
  declarations: [ AppComponent ]
})
export class AppModule {}
```

Angular Module - MetaData

- ng g m commonUi in src/app (as shortcut)
- @NgModule
  - imports
    - specifies modules required in this module
  - declarations
    - declare Components, Directives & Pipes

Angular Module

- @NgModule
  - providers
    - Specifies which service the injector of this module provides for the DI.
  - exports
    - Exports components, directives & pipes , to provide them to modules that imports this module.

Angular Module

- @NgModule
  - bootstrap
    - Component that will be stored in the ComponentFactoryResolver (entryComponents) and will be used to bootstrap the application.

Angular Module

- @NgModule
  - entryComponents
    - Compiles components when initializing the module. This allows us to use that component dynamically, because it will be stored as ComponentFactory inside the componentFactoryResolver.

Components

Components

- Decorator and Metadata
- Angular modules
- Bootstrap Root Component
- Bootstrap a module
- Selector
- Templates
- Styling
- Nesting Components (Shared Modules)
- ng content
- ViewChilds
- Lifecycle hook

Components

- Are custom HTML nodes
- Elements:
  - Template
  - Style
  - Logic

Components

```
class AppComponent {

    constructor () {
        console.log ( "App Component" );
    }

}
```

Components

```
<h1 (click)="onClick()">{{name}}</h1>
```

`

Components

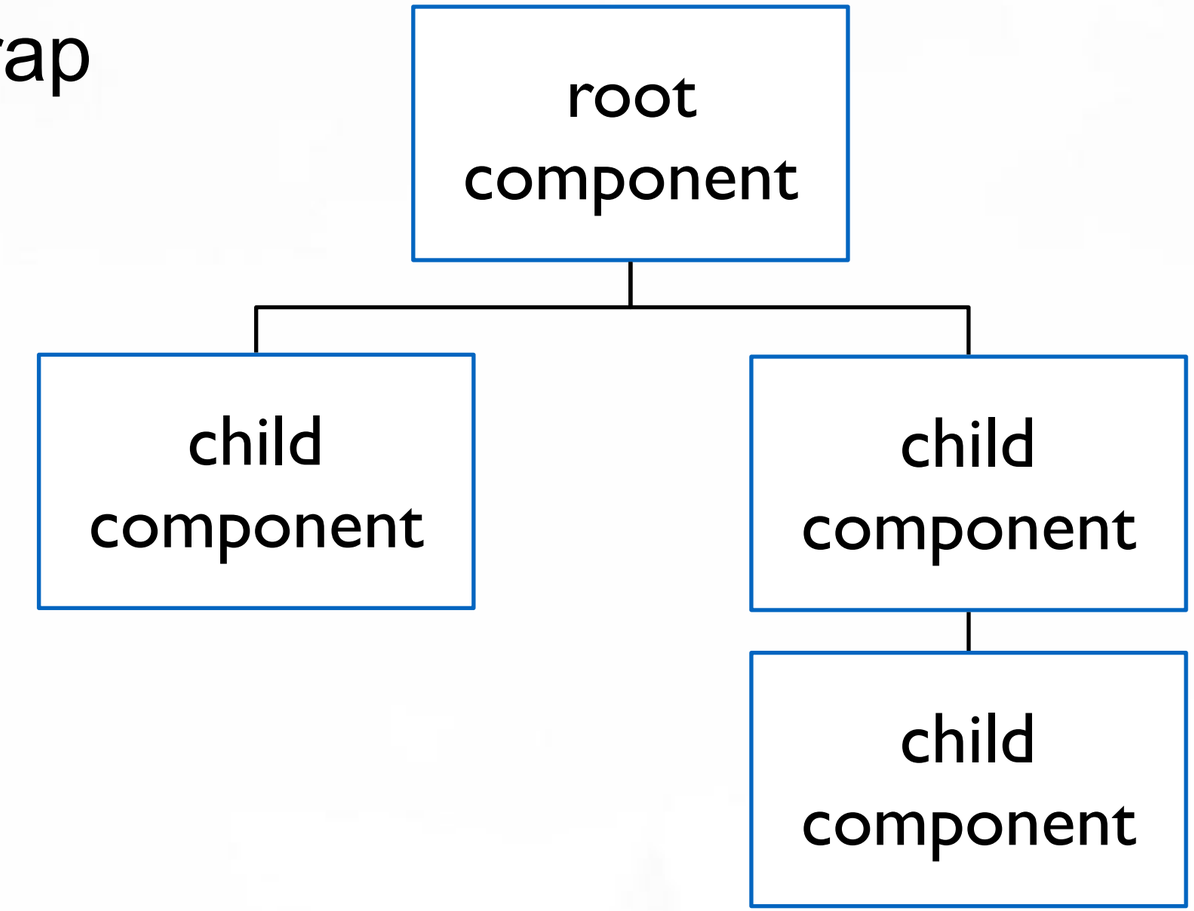
```
<h1 (click)="onClick()">{{name}}</h1>
<my-component> </my-component>
```

Decorator and Metadata

- Decorators enhance a class with metadata
- Before class definition
- Decorators works like function calls starting with @NAME
- Parameter
  - metadata

Component-based application

- bootstrap



Bootstrap

- in main.ts
- platformBrowserDynamic
  - bootstrapModule
    - AppModule
      - bootstrap of component

Bootstrap

```
@NgModule({
  imports:      [ BrowserModule ] ,
  declarations: [ AppComponent, MyComponent ],
  bootstrap: [ AppComponent]
})
export class AppModule {}
```

Component Metadata

- ng g c user
  - Node name
    - selector (string)
  - Template
    - templateUrl (file ref.)
    - template (string e.g. with backticks)

Component Metadata

- Style
  - styleUrls (filelist)
  - styles (backtick-list)
- Spezieller Style
  - :host

Component Metadata

- Style
  - encapsulation - Dealing with Web Components
    - ViewEncapsulation.Emulated
    - ViewEncapsulation.Native
    - ViewEncapsulation.None

Transclude content

- ng-content
  - Placeholder node in component template
  - Attribute
    - select=„nt-table-caption"

Reaching transcluded content

- @ContentChild - decorator
  - Reaching the transcluded Child Component
    - parameter
      - component class
- ngAfterContentInit
  - Hook from when the value can be reached.

Reaching transcluded content

- @ContentChildren - decorator
  - Same as ContentChild -> QueryList<Component-Class>

Reaching child elements

- @ViewChild
  - Nodes of a component that have been defined in the template.
    - parameter
      - component class
      - Reference (string) to a node marked with #NAME

Reaching child elements

- @ViewChild
  - ngAfterViewInit
    - Hook from when the value is reachable

Reaching child elements

- @ViewChildren
  - Similar to ViewChild -> QueryList<Component-class>

Bindings

Bindings

- Interpolate expressions
- Binding properties
- Binding Style Properties
- Binding CSS classes
- Binding attributes
- Binding events
- Component attributes
- Component events
- Host binding

Interpolate expressions

- expression in curly brackets
  - {{ expression }}
- Permitted expressions
  - Properties, strings, operators
  - method return

Properties

- Allows assignment via HTML node attribute
- [PROPERTY]= "EXPRESSION"
- Permitted expressions
  - Properties, strings, operators
  - method return

Attribute

- Allows assignment via node attributes of an HTML element
- [attr. ATTRIBUTE]= "EXPRESSION"
- Permitted expressions
  - Properties, strings, operators
  - method return

Styles

- Allows assignment via style properties of an HTML element
- [style. PROPERTY. UNIT]= "EXPRESSION"
- Permitted expressions
  - Properties, strings, operators
  - method return

Class

- Allows styling over CSS classes
  - [class. CLASS NAME]= "BOOL EXPRESSION"
  - [class]= "EXPRESSION"
- Permitted expressions
  - Properties, strings, operators
  - method return

Events

- Allows binding to native events of an HTML element
  - (EVENT)= "METHOD ($PARAM)
- parameter
  - $event -> passes event through
- example
  - (click)= "clickHandler ($event)

Custom component attribute

- Component properties can be created using the input decorator
  - @Input (OPT_ATTR_NAME) NAME: Type
- Also usable for setters
- example
  - @Input ('selected-usr') selectedInd: number = 1;
  - <comp [selected-usr]= "2".....

Custom component event

- Component events can be created using the property decorator
  - @Output (OPT_ATTR_NAME) NAME: EventEmitter
- Also usable for getter
- EventEmitter sends value via emission
- If the NAME is followed by the expression ‚**Change**‘, a bidirectional binding is possible.

HostBindings- and Listener

- Using this property decorators, you can also define bindings directly in the component class
  - @HostBinding (bind) NAME: boolean = true
  - @HostListener (EVT_NAME,[' $event']) HANDLER: Function = (evt)=>{}

Directives

Directives

- Definition of a directive
- Angular build-in directives
  - ngIf
  - ngFor
  - ngClass and ngStyle
- Developing directives

Directives

- Directives are "Components without templates" and are used as attributes. There are following types:
  - Structural
    - Modified DOM
  - Attributes
    - Modifies the functionality

Directives

- Selector specifies how directives are applied
  - Attribute <div selector....
    - Assignment <div selector= "value"
    - With binding <div[selector]= "worth"
  - Class (avoid) <div class= "selector"....

Structural Directives

- Modify the DOM
- Can be used with Asterix * or via a template node
  - [ngIf] = "EXPRESSION"
    - Removes the node from the DOM if the expression is false

Structural Directives

- [ngFor]= "EXPRESSION"
- Repeats the node using an iteration
- Expression
  - Describes iterator and can pass additional values
    - index, first, last, even, odd

Attribute Directives

- [ngClass]=„EXPRESSION“
- [ngStyle]=„EXPRESSION“
  - Extended style and class attribute of a node

Developing Directives

- Directive
  - selector (directive name)
    - Attribute e. g.: '[myDirective]'
    - Class e. g.: ’. my-class' (also as list)
  - Directive class (optional with DI from ElementRef)
    - nativeElement - Refers to the element

Pipes

Pipes

- The definition of pipes
- Buid-in Pipes
- Using Pipes
- Create your own Pipes

Pipes

- Modifies the output
- Syntax
  - Expression | PipeName: Parameter
- example
  - {{name | uppercase}}}

Pipes

- Build-in
  - Uppercase
  - Lowercase
  - Date
  - …

Pipes

- Usage
- Expression | PipeName: Parameter
  - Example: {{name | uppercase}}
- In JS
  - const reversePipe: ReversePipe = new ReversePipe ();
    console. info (reversePipe. transform (123);

Pipes

- Develop with decorator
  - @Pipe
    - name: string
- class NAME implements PipeTransform
  - transform(value: any, args?: any): any {

rxjs - comparable to Java-Streams

Observer / Observable

- Observable
  - Object for data streams that can be modified by various operators before they are subscriped.
- Subject extends Observable
  - Allows you to push and manage the data streams
    - asObservable();

Subscription

- to Observable
  - next
  - error
  - complete
- unsubscribe
- siehe: http://rxmarbles.com/

Sample

- observable = Observable.range (1, 5);
- observable.subscribe (
  - next => console.info ('next %s', next),
  - error => console.info ('error %s', error),
  - () => console.info ('complete')
-)

Service

Service

- Definition
- DI
- Build-In Services
- Import and Provide
- Develop
- Injectable
- HTTP-Service

Service

- View-independent business logic
- No Angular specifications
  - Except you need injection in service constructors
- Providing via DI

Develop a Service

- Create a Class
- Use Injectable if constructor needs DI
- Including in Provider

Dependency Injection (DI)

```
app.component.ts:19 -> http Injected Http {}
```

Dependency Injection (DI)

```
@NgModule({
  imports:      [ BrowserModule, HttpModule  ],
  providers:    [ MyService ]
})
export class AppModule {}
```

HTTP-Service

- HTTP-Modul import
- HttpClient-Service inject
- Useable Methods
  - request - Base for shortcut methods

HTTP-Service

- Useable methods
  - [C] post
  - [R] get
  - [U] put
  - [D] delete

HTTP-Service

- Request params
  - method: string,
  - url: string,
  - options?:

HTTP-Service

- Request options params: {
  - body?: any;
  - headers?: HttpHeaders;
    - set (key, value)
  - params?: HttpParams;
    - set(key, value)

HTTP-Service

- Request options params: {
  - observe?:
    - ‚body' | events‘ - HttpEvents<T> - > JSON | Text;
    - ‚response‘ - HttpResponse<T>

HTTP-Service

- Request options params: {
  - reportProgress?: boolean;
  - responseType?: 'arraybuffer' | 'blob' | 'json' | ,text';

Request

- Subscribe (next, error, complete)
- rxjs helper
  - retry - to repeat a Request on error
  - do - e.hg. to get logs

Routing

Router

- The Basis of a Single Page Application
- Routes define which components are displayed.
- Router module is provided by Angular

Routing

- Routing Preparation
- Routing Configuration
- Router Modules
- Navigation via Router Directives
- Navigation via router service
- Childs
- Events
- CanActive - Guard
- Resolve
- Parameter
- Lazy Module

Firs routing steps

- Integrate module via RouterModule. forRoot
  - Define Routes:
    - path
    - component
- { useHash: false }
- <router-outlet></router-outlet> include

redirect

- initial
  - path: '',
    pathMatch: 'full',
    redirectTo: 'list'
- 404
  - path: '**',
    redirectTo: 'list'

routerLink

- Directive
  - Value
    - path | [path, …params: any[]]
- routerLinkActive
  - Value
    - CSS class name

Navigation via RouterService

- Inject Router Service
- router.navigate method
  - Params
    - List
      - path
      - params

Route params

- Define routes with params
  - path: 'details/:id',
    component: UserDetailsComponent
- Get Params via ActivatedRoute Service DI
  - this.subscription = this.route.paramMap
    .map (paramMap => paramMap.get ('id') || 'None')
    .subscribe(id => this.param_id = id);

Resolve

- Resolving before Route-Change
  - Create a ResolveService based on Resolve Interface and use it in resolve object within the RouteConfig
    - path: 'details/:id',
      component: UserDetailsComponent,
      resolve: {
        user: ResolveService
      }

Resolve

- Get resolved Data via ActivatedRoute Service within ngInit hook

- this.route
  .data
  .map (data => data['user'])
  .subscribe(user => this.user = user);

CanActive

- Approval of the activation of a new route
- For this purpose, a service based on the CanActive interface is created and integrated
  - canActivate (route : ActivatedRouteSnapshot, state : RouterStateSnapshot) : Observable<boolean>|Promise<boolean>|boolean

CanActive

- The guard service is implemented in the route definition
  - path: 'home',
    component: HomeComponent,
    canActivate: [CanActiveService]

Routing Events

- Inject Router Service
- subscribe to events: Observable<Event>
  - constructor (router: Router) {
    router.events.subscribe(event => console.log (event));
    }

Child

- A route can have sub routes
  - These must be specified in the config under the property
    - children
      - in the same way as the existing configuration.

Lazy Module

- loadChildren enables simple implementation in the CLI context
- path : 'dash',
  loadChildren: './dash/dash.module#DashModule'
  - The path to the module and the class name must be passed
    - PFAD#MODUL_NAME

Lazy Module

- Within the lazy module, the route is defined with the component to be displayed.

- RouterModule.forChild ([
  {
    path   : '',
    component: DashComponent
  }
])

Forms

Form types

- Template driven
  - FormControlls are created by directives
  - Therefore Asynchronous
- Reactive forms
  - FormControlls are created in the code and referenced by directives in the template
  - Therefore synchronous

Bootstrap

- #f="ngForm"
  - Initializing a Form and Referencing the form group async.
- form: [formGroup]
  - Binds a form group to a reative form element

Bootstrap form controls

- ngModel
  - Initializing a form control for an input-element async.
- formControlName
  - directives to connect an input element to a reactive form control

Directives - async

- #usrname=„ngModel"
  - Assign model values - a form controller will be created and is reachable via #name
- ngModel
  - directive for form element so that inputs are written directly to the model
  - And values from the model are displayed

Directives - reactive

- formControlName="username"
  - Reactive only the ForControlName directive is used and refers to the key of the previously created controller.

Group Directives

- Create groups
  - template
    - ngModelGroup directive
    - e.g. ngModelGroup=„account“
- reactive
  - Assign key with formGroupName Assign directive

Submit

- (ngSubmit) Register event on form
- button: Add submit to the form

Klassenbasierte Validierung

- Angular attaches validation CSS classes to a form element dependent on
  - untouched / focus
    - ng-untouched ng-pristine ng-invalid
  - typing / error
    - ng-untouched ng-dirty ng-invalid

Klassenbasierte Validierung

- corrupt
  - ng-invalid ng-dirty ng-touched
- valid
  - ng-dirty ng-touched ng-valid

Template-Driven Forms

- use ngForm and ngModel for element definition
  - <form #f="ngForm">
  - <input type=„text" name=„username" ngModel #usrname=„ngModel" id="username" required minlength="4">

Using form controls

- <div *ngIf="((usrname.dirty || usrname.touched) && usrname.errors)"
  -
    {{usrname.dirty | json}}
    {{usrname.errors | json}}
    {{usrname.valid | json}}
    {{f.valid | json}}

Use FormGroup

-
- <input type="email" name="mail" id="mail" ngModel >
-

Test

Spec Files

- Create a **describe** Block for a Test
  - description: string
  - callback handler
- describe callback has two phase
    - beforeEach (jasmine) to prepare Test-Assets
    - it (jasmine) to Test

Spec Files

- **beforeEach** (jasmine) expect a callback handler, that prepare Test assets
- **it** (jasmine function) expect two params
  - description: string -> will be shown during Tests
  - callback handler that will be exec. the tests.

Spec Files

- callback - body
  - Inside the handler you have to make your test via
    - **expect** (jasmine) method
      - parameter - the value that has to be tested
      - return matchable instance

Spec Files

- Test with matcher
  - **toBe** (val) -> equal to compare with **===**
    - **not.toBe**(val) -> equal to compare **!==**
  - **toEqual**(val) -> compares objects and **every field**
  - **toMatch**(regExp) -> compares with **regExp**
  - **toBeDefined** (val) -> compare with **!== undefined**

Spec Files

- Test with matcher
  - **toBeUndefined** (val) -> compare with **=== undefined**
  - **toBeNull** (val) -> compare with **=== null**
  - **toBeTruthy**(val) -> compare with **=== Boolean(val)**
  - **toBeFalsy** (val) -> compare with **!== Boolean(val)**
  - **toContain** (val) -> compare with **indexOf !== -1**

Spec Files

- Test with matcher
  - **toBeLessThan** (val) -> compare with **< val**
  - **toBeGreaterThan** (val) -> compare with **> val**

TestBed

- Angular test utils
  - **configureTestingModule** factory for Testing Modules
    - use in beforEach to rest state before tests
    - as parameter use classic NgModule MetaData poroperties
    - **compileComponents() -** compiles all components in Module du Inline JavaScript

TestBed

```
beforeEach(async(() => {
  TestBed.configureTestingModule({
    imports: [
      RouterTestingModule
    ],
    declarations: [
      AppComponent
    ],
  }).compileComponents();
}));
```

TestBed

- **createComponent(ComponentClass)** method closes TestBed configuration and returns a **ComponentFixture** instance.
    - provides access a DebugElement that contains the component instance

```
fixture.debugElement.componentInstance
```

TestBed

- **detectChanges ()** method of **ComponentFixture** instance execute the change detection and renders template

- The fixture DebugElement provides the nativeElement of the Component

```
fixture.debugElement.nativeElement
```

- The **nativeElement** supports **querySelector** or use **query**, on debugElement, combined with the **By** helper class, to select a Element inside the component.

TestBed

```
expect( fixture.debugElement.nativeElement
                  .querySelector('h1').textContent)
  .toContain('Welcome to gfn!');
expect(fixture.debugElement
                  .query(By.css(,h1'))
                .nativeElement.textContent)
  .toContain('Welcome to gfn!’);
```

Helper

- **import { async } from ,@angular/core/testing';**
  - required to wait that compileComponents could load template- and/or style-files.
- **import { inject } from ,@angular/core/testing';**
  - required to inject services
    - returns callback for it

Helper - async

```
let fixture: ComponentFixture<AppComponent>;
let app: AppComponent;
let h1DebugElement: DebugElement;
let h1: HTMLElement;

beforeEach ( async ( () => {
  TestBed.configureTestingModule ( {
    imports     : […],
    declarations: [ AppComponent]
  } ).compileComponents ();
} ) );

beforeEach ( () => {
  fixture = TestBed.createComponent ( AppComponent );
  app     = fixture.debugElement.componentInstance;
  h1DebugElement = fixture.debugElement.query ( By.css ( 'h1' ) );
  h1 = h1DebugElement.nativeElement;
} );
```

Helper - inject

```
beforeEach ( () => {
  TestBed.configureTestingModule ( {
    providers: [ UserResolveService ]
  } );
} );

it ( 'should be created',
  inject ( [ UserResolveService ],
    ( service: UserResolveService ) => {
      expect ( service )
        .toBeTruthy ();
    } ) );
```