Angular 8.x

Saban Ünlü

A few words about me

Saban Ünlü

- Software architect and programmer
- Consultant and docent since 2000
- Author
- Founder of netTrek

netTrek

TypeScript

Fundamentals

- Programming language based on ES6 (ES2015)
  - Developed from Microsoft
- Exportable into other ECMA-Script versions
- Exportable into different Modul Handlings
- Type-security
- Usage of experimental annotations

Variables

- Definitions
  - let
  - const
- Types
  - Native-Types
  - Datatypes

Classes

- Classes act as a template of a JS object
- constructor
- Attributes and methods
- Instantiate
- Setter and Getter
- Parameter passing

Inheritance

- Classes can inherit from other classes
  - extends
- Scopes
  - private, public & protected
- Overwrite
  - super

Interfaces

- Interfaces are the templates of a class
  - Interfaces can inherit - extends
- Implementation of an Interface is done by
  - implements

Abstract class

- Implements the basic functionality and attributes
- Used as a template for a derivative (template)
- Can not be instantiated

Syntax magic

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

Technologies

Technology overview

Node.js

- JavaScript-runtime-environment
- Available for a variety of operating systems
- Needed:
  - Testing
  - Deployment

TypeScript

- Programming langauge based on ES2015
  - Classes, inheritance, typcasting, interfaces, enums etc.
- Exported to ES5
- Angular was developed using TypeScript

git

- Version control system for software
- GitHub – Filehoster
- Allows to manage different versions of one software
- Optimizes teamwork

webpack

- Bundels static content into packages
- Within the Angular-context
  - ES-Modules, styles, templates
    - JavaScript-packages
- Simplified deployment
- Optimized load times

SASS

- Extension language for CSS
  - CSS-Preprocessor
- Supports
    - Variables, functions, extensions, imports etc.
- Very steep learning curve

Jasmine

- Development-Framework for testing JavaScript-code
  - Independent from other Frameworks
  - Does not need the DOM
- Allows definition of behavior-oriented tests
  - Expectation is defined and checked
    - expect(a).toBe(true);

Karma

- Framework for controlling JavaScript-tests
  - Provided by the Angular-team
  - Supports: Jasmine, Mocha und QUnit
- Allows testing on devices
- Very good support for Continuous Integration, e.g. with Jenkins

Protractor

- Framework for End-to-End-tests
  - Developed by Google for Angular
  - Testing in a real browser
  - Simulating a real user
  - User events e.g. clicks or inputs
  - Waiting for asynchronous events

Polyfills

- JavaScript-Files
- Checks the existence of specific functions in browsers
- If not existend, the functionality will be extended
  - Workaround for older browsers

core-js

- Polyfill for ES6 (ES2015) Functions
- Frequently required by less modern browsers
- Especially IE needs help here
- In the JIT context, ES7/reflect is also required for the use of decorators.

Zone.js

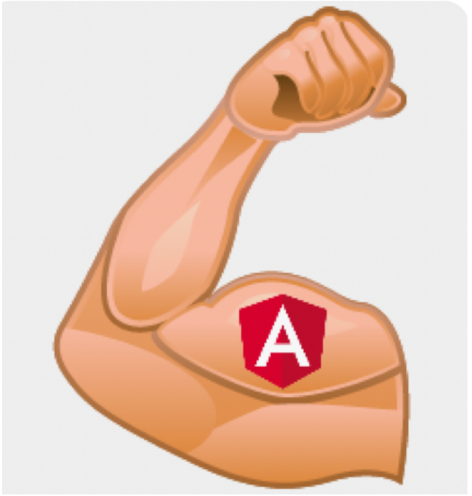
- Framework enables the definition of an execution context for JavaScript
- Comparable to domains in Node.js
- Used as a dependency in Angular
- Monitors and controls execution
- Helps with debugging

ReactiveX

- Framework for monitoring events and asynchronous processes
- Available for different programming languages
- RxJS is the JavaScript variant
- Used as a dependency in Angular, for HTTP and EventEmitter, among others.

Tooling

- VSCode
  - https://marketplace.visualstudio.com/items?itemName=devboosts.angular-productivity-pack



Project setup

First Steps

- Mac
  - Install XCODE
  - install node.js (>= 10.9.x)
- win
  - install node.js (>= 10.9.x)
  - Install Git (incl. Bash)

npm proxy ?

- npm config set proxy http://PROXYURL
- npm config set https-proxy https://PROXYURL

netTrek

Manual Setup

- Initialize Node
- Install dependencies
- Configure TypeScript
- Configure Webpack

Angular-cli

- Command Line Tool
  - Initialize & Set Up
  - Development and maintenance
  - Test and publish

Angular-cli - install

- npm install -g @angular/cli
  - optional
    - Development and maintenance
      - CLI in App Context

Angular Console

Angular-cli

- ng new netTrek --prefix=nt
- ng serve
- ng serve --aot
- ng build
- ng build --prod
- ng lint
- ng test
- ng e2e

netTrek

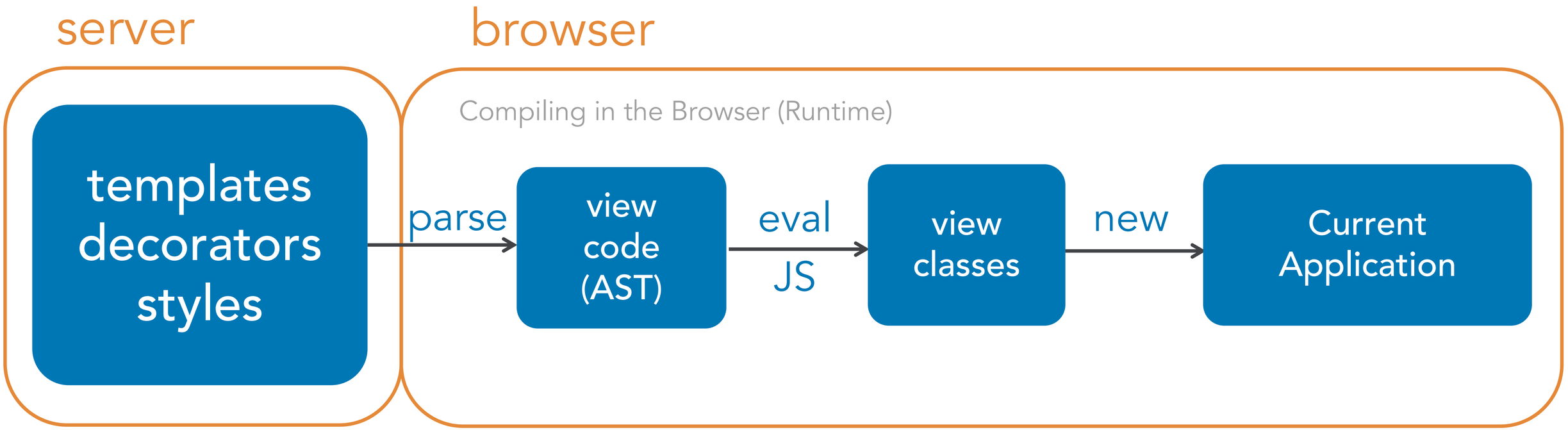
Trainings Branch

- git clone -b training/gfk2019 https://github.com/netTrek/ng8-basics.git gfk

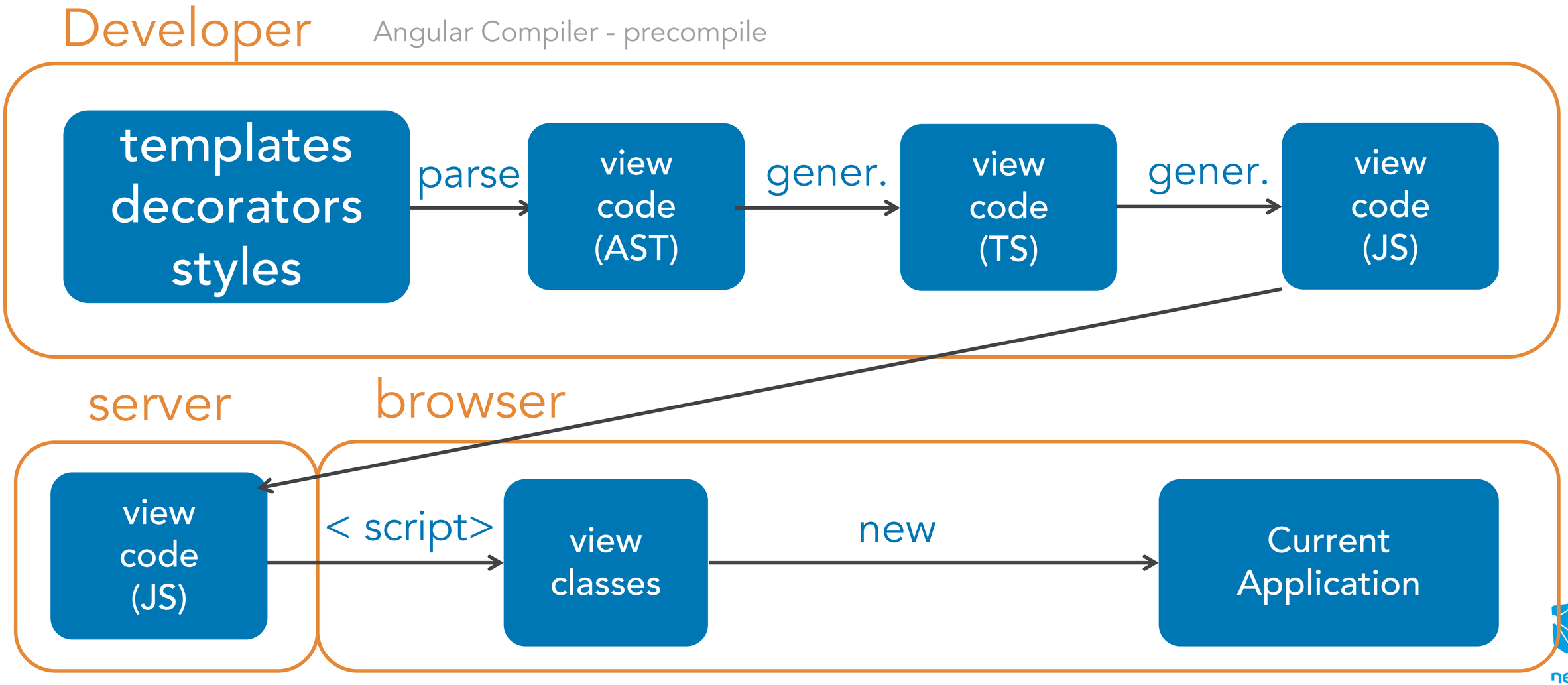
Publish

JIT, AOT and more

JIT


server
browser
templates
decorators
styles
parse
Compiling in the Browser (Runtime)
view
code
(AST)
eval
JS
view
classes
new
Current
Application

AOT


Developer
Angular Compiler - precompile
templates
decorators
styles
parse
view
code
(AST)
gener.
view
code
(TS)
gener.
view
code
(JS)
server
browser
view
code
(JS)
< script>
view
classes
new
Current
Application

Publish - Package Analysis

- ng build [app-name]
  - --stats-json
    - # webpack-bundle-analyzer

Architecture

Introduction

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

Architecture

Decorator

Decorator

- Functions with @ symbol in front
- Used before a declaration
- Decorators in Angular have identical core functionalities
  - Saving Meta Information
  - Manipulation of subsequent declaration

```
@HostListener('click')
onHostClick() { /**/}
```

Decorator

- Decorator type
  - Decorate classes
  - Decorate properties
  - Decorate methods
  - Decorate parameters

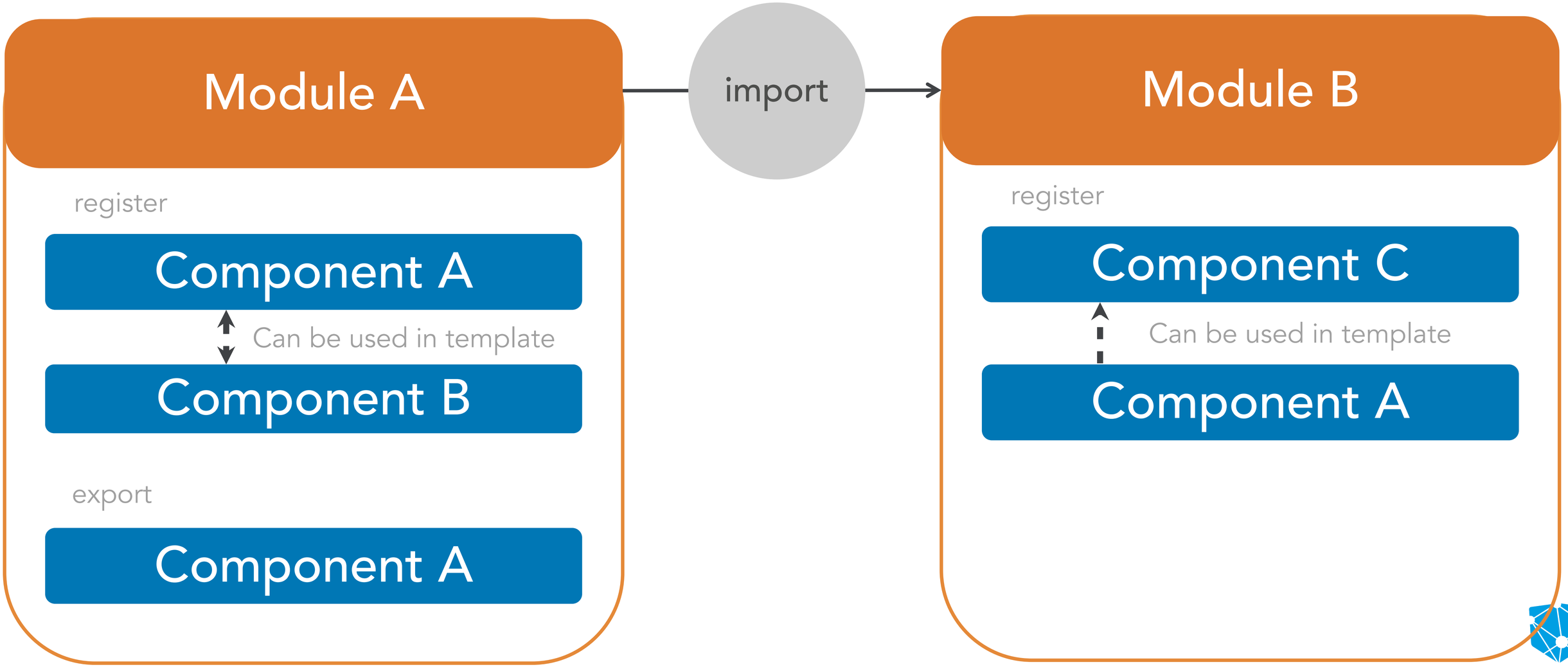
Architecture

Modules

Modular development

- Angular Modules
  - Perfect for teamwork
  - Reusable
    - Export/Import
  - Container (accessible)
    - Components, directives, pipes, services

Modular development



Modules

- Not comparable with JavaScript modules
- Bundle functions and features in a black box
- Extend application and own modules with external modules
- Tell compiler what elements to look for

Modules

- Angular-own modules
  - BrowserModule (Events, DOM)
  - CommonModules (directives, pipes)
  - HttpModules (XHR)
  - FormsModule (Forms)
  - RouterModule (component router)

Modules

- Creating Modules
  - Creating a Module class

Modules

```
class AppModule {}
```

Modules

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

Modules

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

Modules

- 'ng g m user --module app' in src/app
- @NgModules
  - imports
    - defines modules that are needed in this module
  - declarations
    - required components, directives, pipes

Modules

- @NgModule
  - providers
    - Determines which service the injector of this module provides for the DI.
  - exports
    - Exports components, directives, pipes of this module so that importing modules use the

Modules

- @NgModule
  - bootstrap
    - Components that are stored in the ComponentFactoryResolver during the bootstrap of this module. Analog - entryComponents

Modules

- @NgModule
  - entryComponents
    - Compiles components when defining the module. Afterwards the use without component context is possible, because it is stored as ComponentFactory and the componentFactoryResolver.

Modules - Bootstrap

- in the main.ts
- platformBrowserDynamic
  - bootstrapModules
    - AppModules
      - bootstrap of the components

Architecture

Components

Introduction

- Decorator and Metadata
- Angular Modules
- Bootstrap Root Component
- Bootstrap a Modules
- selector
- templates
- styling
- Nesting Components (Shared Modules)
- ng-content
- ViewChilds
- Lifecycle hook

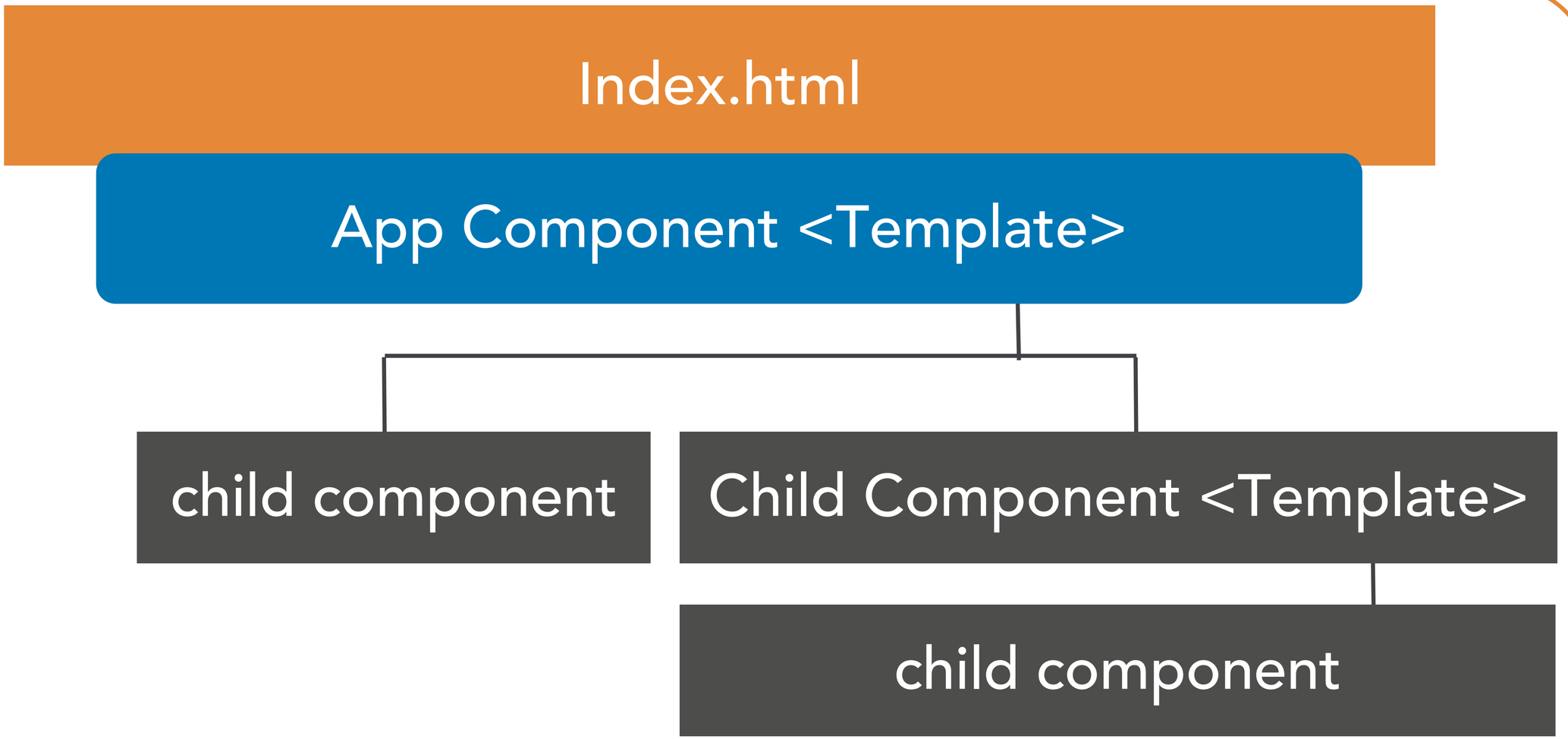
Component-based development

- Component corresponds to its own HTML node
  - Logic
  - Template (HTML)
  - Style (optional)
- child component
  - Using components within a template

Templates

- HTML snippets
  - Represents the user interface of a component
  - Definable as
    - String or external files
    - As metainformation of a component
      template or templateUrl

Component-based development



1

Logic (TS)

```ts
export class UserComponent {

  name = 'Saban Ünlü';
  chgName () {
    this.name = 'Peter Müller';
  }

}
```

2

View (HTML)

```html
<h1>{{name}}</h1>

<button (click)="chgName()">
  Ändern
</button>
```

3

View (Style)

```css
h1 {
  color: darkslategray;
}
button {
  background-color: yellowgreen;
}
```

Create component

- Create components class (ts)
  - export class ComponentName
  - Class provided with metainformation
    - @Component ({ /*meta*/ })
      export class ComponentName

Create component

- @Component - Decorator (meta-information)
  - selector - HTML node name
  - templateUrl or template - templates of the component
  - styles or styleUrls - List of style definitions

Create component

```
class AppComponent {

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

}
```

Create component

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


 selector: 'app-root',
 templateUrl: './app.component.html',
 styleUrls: ['./app.component.css']

export class
 name = 'app works!';
 onClick () {
 console.log ( 'clicked' );
```

Create component

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

Create component

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

Create component

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

Component metadata

- ng g c user/user --export -skip-tests --flat
  - selector
    - node
  - template
    - templateUrl (file)
    - template (backticks)

Component metadata

- style
  - styleUrls (filelist)
  - styles (backtick-list)
- Special Style
  - :host
  - ::ng-deep

Component metadata

- style
  - encapsulation - handling web components
    - ViewEncapsulation.Emulated
    - ViewEncapsulation.None
    - ~~ViewEncapsulation.Native~~ (deprecated)
      - ViewEncapsulation.ShadowDom

Bindings

Bindings

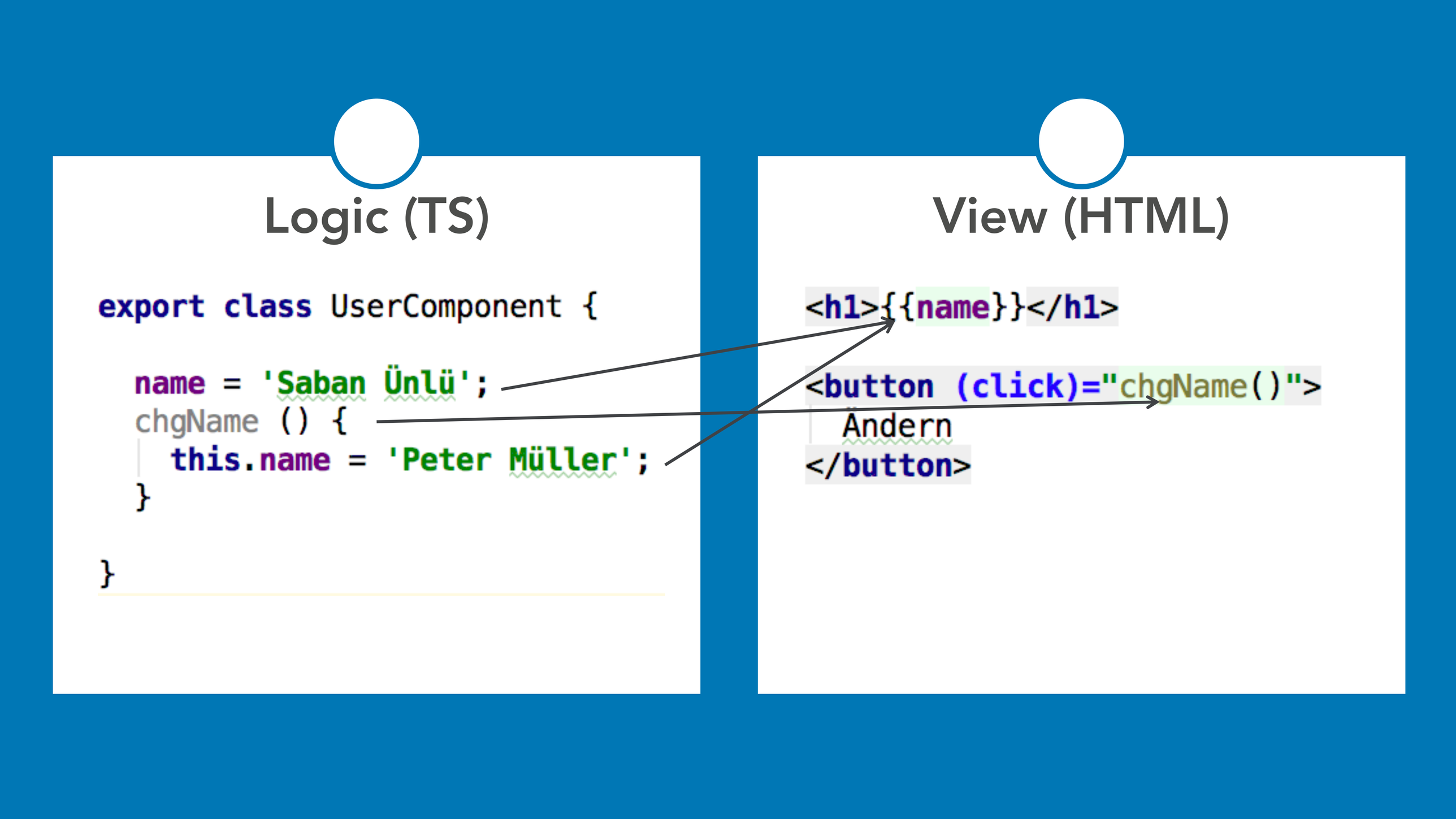
- Interpolate expressions
- Bind properties
- Bind style properties
- Bind CSS classes
- Bind attributes
- bind events
- component properties
- component events
- HostBinding
- HostListener

Logic (TS)

```ts
export class UserComponent {

  name = 'Saban Ünlü';
  chgName () {
    this.name = 'Peter Müller';
  }

}
```

View (HTML)

```html
<h1>{{name}}</h1>

<button (click)="chgName()">
  Ändern
</button>
```



Bindings

- Binding values and methods in templates
  - By means of expression interpolation
    - < h1>{{name}}</h1>
    - < h1>{{getName()}}</h1>
    - < img src="path/{img}}">

Bindings

- Binding values and methods in templates
  - Bind as property
    - <img [src]= "imgPath">
  - Bind as attribute
    - <img [attr.alt]= "imgAlt“>

Interpolate expressions

- expression in curly brackets
  - {{ EXPRESSION }}
- Permitted expressions
  - Properties, Strings, Operators
  - return of methods

Properties

- Allows assignment via properties of an HTML element
- [PROPERTY]="EXPRESSION"
- Permitted expressions
  - Properties, Strings, Operators
  - return of methods

Attributes

- Allows assignment via node attributes of an HTML element
- [attr.PROPERTIES]="EXPRESSION"
- Permitted expressions
  - Properties, Strings, Operators
  - return of methods

Styles

- Allows assignment via style properties of an HTML element.
- [style.PROPERTY.UNIT]="EXPRESSION"
- Permitted expressions
  - Properties, Strings, Operators
  - return of methods

Class

- Allows styling via CSS classes
  - [class.CLASSNAME]="BOOL EXPRESSION"
  - [class]="EXPRESSION"
- Permitted expressions
  - Properties, Strings, Operators
  - return of methods

Event

- Allows binding of events
  - (EVENT)="METHOD($PARAM)"
- parameter
  - $event -> passes event through
- Example
  - (click)="clickHandler($event)"

Parent-child communication

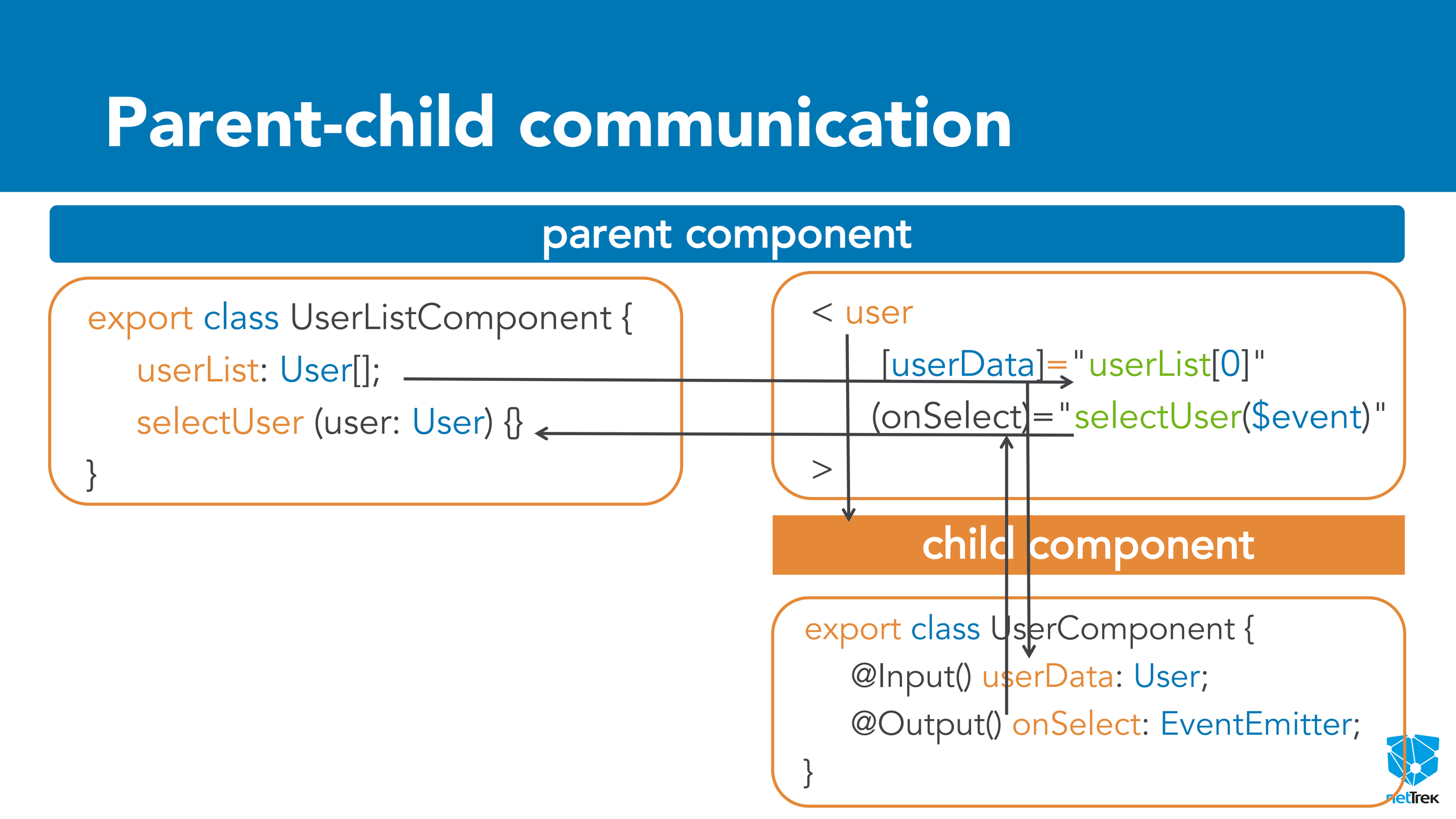
parent component

```
export class UserListComponent {
    userList: User[];
    selectUser (user: User) {}
}
```

```
< user
    [userData]="userList[0]"
    (onSelect)="selectUser($event)"
>
```

child component

```
export class UserComponent {
    @Input() userData: User;
    @Output() onSelect: EventEmitter;
}
```



Component attributes

- User-defined attributes can be created using the property decorator.
  - @Input (OPT_ATTR_NAME) name: Type
- Can also be used for setters
- ngOnChanges : Hook informs about new values
  - SimpleChanges

Component events

- User-defined events can be created using the property decorator.
  - @Output (OPT_ATTR_NAME) name: EventEmitter<T>
- EventEmitter sends value via emit
- Parent components can attach themselves to the event
  - $event - Transmitted event value

Component lifecycle

- constructor
- ngOnChanges
- ngOnInit
- ngDoCheck
  - ngAfterContentInit
  - ngAfterContentChecked
  - ngAfterViewInit
  - ngAfterViewChecked
- ngOnDestroy

```
export class UserListComponent
< userList [data]="userList">
```

UserList Vorlage

```
<userList>
    <user> </user>
    <user> </user>
</userList>
```

HostBindings- and Listener

- The property decorator can also be used to define bindings directly in the component class.
  - @HostBinding (bind) NAME : boolean = true
  - @HostListener (EVT_NAME, ['$event']) HANDLER : Function = (evt)=>{}

1

view

```
<user-list>

    <user-header></...>

    <user-item></...>

    <user-item></...>

    <user-item></...>


</user-list>
```

2

UserList - Template

```
<h3>user-list</h3>

<user-header></...>

<ng-content
   select="user-header">
<user-item></...>
</ng-content>
<user-item></...>

<user-item></...>
<ng-content></ng-content>
```

Component Content

- Transclude contents (transclude)
  - ng-content
    - nodes in template
    - attribute
      - select="nt-table-caption"

Transcluding contents

- Components represent a view.
- The view is described in the HTML template
- Within the template, child components can be provided with content nodes.
- <user-list>
    <user-item> name</user-item>
  </user-list>

Transcluding contents

- Content nodes are transcluded when templates use the ng-content directives (nodes).
  - The node represents a placeholder.
  - The select attribute can be used to define for which content the placeholder should be used.

Access to transcluded content

- About Property Decorator
  - @ContentChild
    - Parameters: Component class or selector
    - option object
      - static: false | true
      - read?: ElementRef | ViewContainerRef | Directive | Service

Access to transcluded content

- By using the Property Decorator
- Access only after hook
  - ngAfterContentInit
  - ngInit if static is true
    - Resolved before the change recognition run

Access to transcluded content

- By using the Property Decorator
  - @ContentChildren
    - Parameters: component class
    - Optional options object with read property
      - read: ElementRef | ViewContainerRef | Directive | Service
      - descendants: false | true (only direct children === false)

Access to transcluded content

- By using the Property Decorator
  - @ContentChildren
    - Creates
      - QueryList<Type>
        - changes -> Observable

Determining template elements

- By using the Property Decorator
  - @ ViewChild
    - Parameters: Component class / Hash ID Options-object
      - static: false | true
      - read?: ElementRef | ViewContainerRef | Directive | Service

Determining template elements

- By using the Property Decorator
- Access only after hook
  - ngAfterViewInit
  - ngInit if static is true
    - Resolved before the change recognition run

Determining template elements

- By using the Property Decorator
  - @ViewChildren
    - Parameters: component class
    - Creates
      - QueryList<Type>
        - changes -> Observable

Directive

Directives

- definition
- Already included
  - ngIf
  - ngFor
  - ngClass and ngStyle
- Own directives

Directives

- Directives can be used within a template
- They are marked as attributes
- There are two types of directives
  - Structural directives that manipulate the DOM
  - attribute directives that manipulate the appearance and/or behavior of an element

Directives

- Structural directives are indicated by an asterisk (*) in front of the attribute name:
  - `<img *ngIf="showImg">`
  - `<li *ngFor="let label of labels">`

Directives

- Attribute directives without value:
  - <input matInput>
- Attribute directives with value assignment:
  - <textarea matAutosizeMinRows="2">
- Attribute directives with bound value assignment
  - <input [ngClass]="inputClass">

Structural directives - ngIf

- [ngIf]="EXPRESSION"
  - Removes the node from the DOM if the expression is false.

Structural directives - ngFor

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

Attributes directives

- [ngClass]="EXPRESSION"
- [ngStyle]="EXPRESSION"
  - Extends style and class attribute of a node

Create directive

- @Directive
  - selector
    - Attribute e.g. ['myDirective']
    - Class e.g. '. my-class' (also as a list)
  - class optional with DI from ElementRef
    - nativeElement - then references the element

Pipe

Pipes

- Pipes are used to manipulate outputs
- They are mainly used in templates
  - Expression | PipeName : Parameter
- But the use on code level is also possible
  - DI or new and transform method of instance

Pipes

- Example
  - <h1>{{name | uppercase}}}</h1>
- Pipes can also be connected in chains
  - < h1>{{createdAt | date : 'long' | uppercase}}</h1>

Pipes

- Already included
  - uppercase
  - lowercase
  - date
  - JSON
  - ...

Create pipes

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

Create pipes

- Pipes are pure i.e. we have a Singleton and the execution takes place with data change.
- In the Metadata it can be set that false is used for pure.
  - So the pipe is not a singleton
  - Can own states thus act
  - And is triggered by change recognition.

rxjs

https://github.com/ReactiveX/rxjs

https://www.learnrxjs.io/

http://rxmarbles.com/

rxjs - Observable

- Vendor of an observable data stream
- Data stream, manipulable with operators and where observers register (subscription)
- Cold (single cast) - Observable waites for Subscription
- Hot (multi clast) - Observable is already working

rxjs - Observer

- Receives values, errors and status from data stream
  - next
  - error
  - complete

rxjs - Subject

- Both Observer and Observable (Hot)
  - Data stream can be registered with it
  - Also a transmitter

rxjs - Subscription

- Registration at Observable
  - next
  - error
  - complete
- unsubscribe (deregistration)
- see: http://rxmarbles.com/

rxjs - Creation of an Observables

- new
- of
- range
- fromEvent
- ...

rxjs - Operations on the data stream

- pipe
  - map
  - filter
  - find
  - scan
  - ...

Dependency injection

Service and Provide basics

Services

- Are view-independent logics
  - e.g. client-server communication
- Are TypeScript classes
  - Instance provision via Dependency Injection
    - provide
    - Typed parameter in the constructor

Dependency injection

- Services, values and functions can be injected
- Required: Provision within a container (injector)
  - Provision through attachments in providers-list
    - Within metadata information for
      - modules
      - components

1

ModuleA

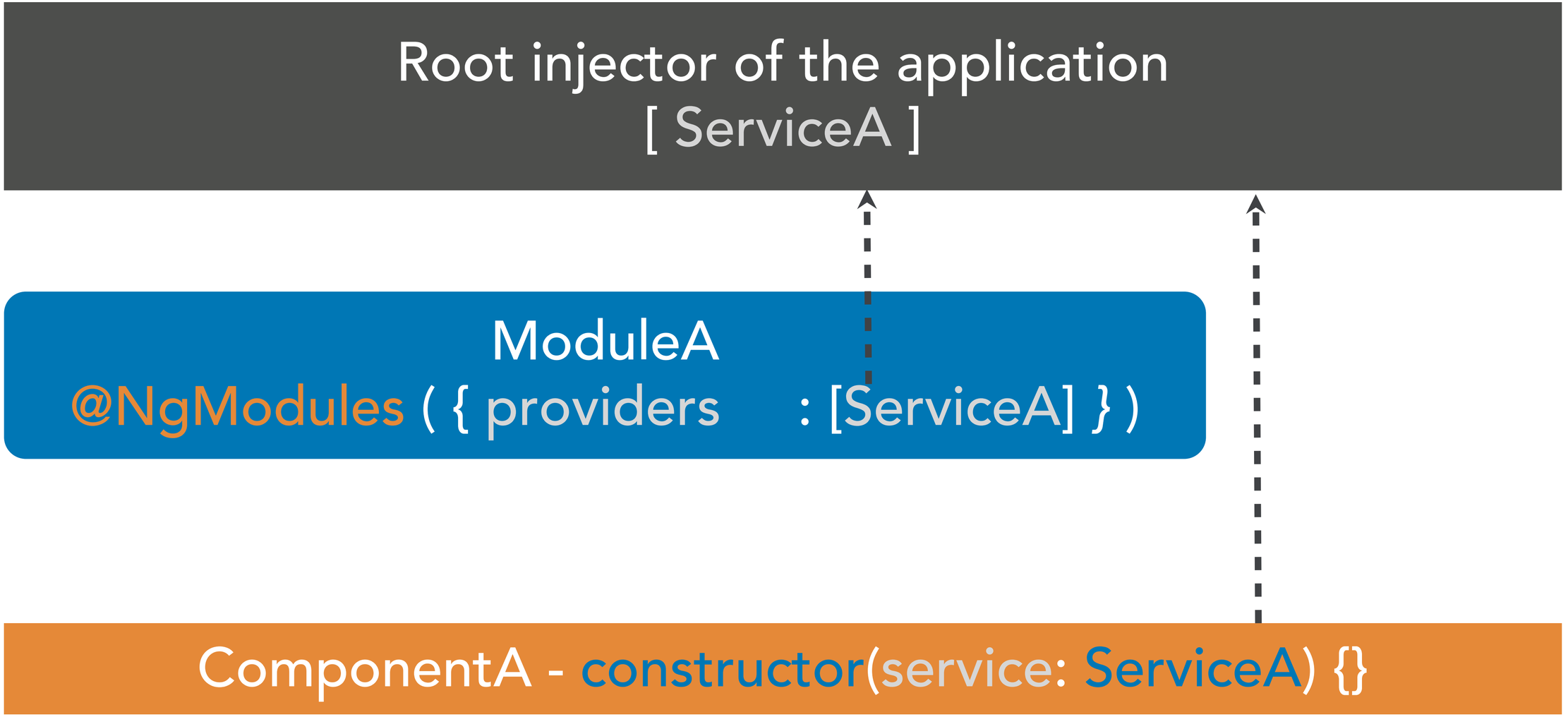
- Register (declarations)
    - ComponentA
- Provide (providers)
    - ServiceA

2

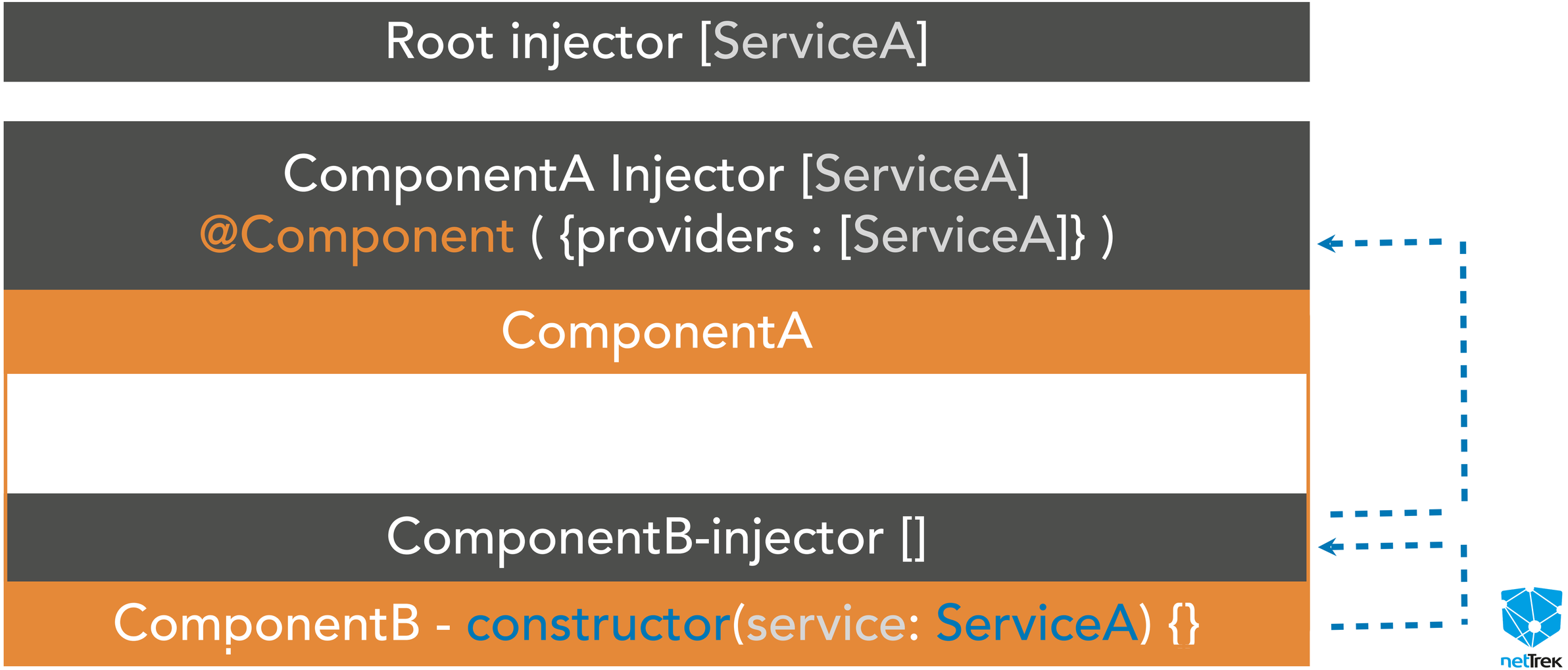
ComponentA

```
constructor(
        service: ServiceA
) {
```

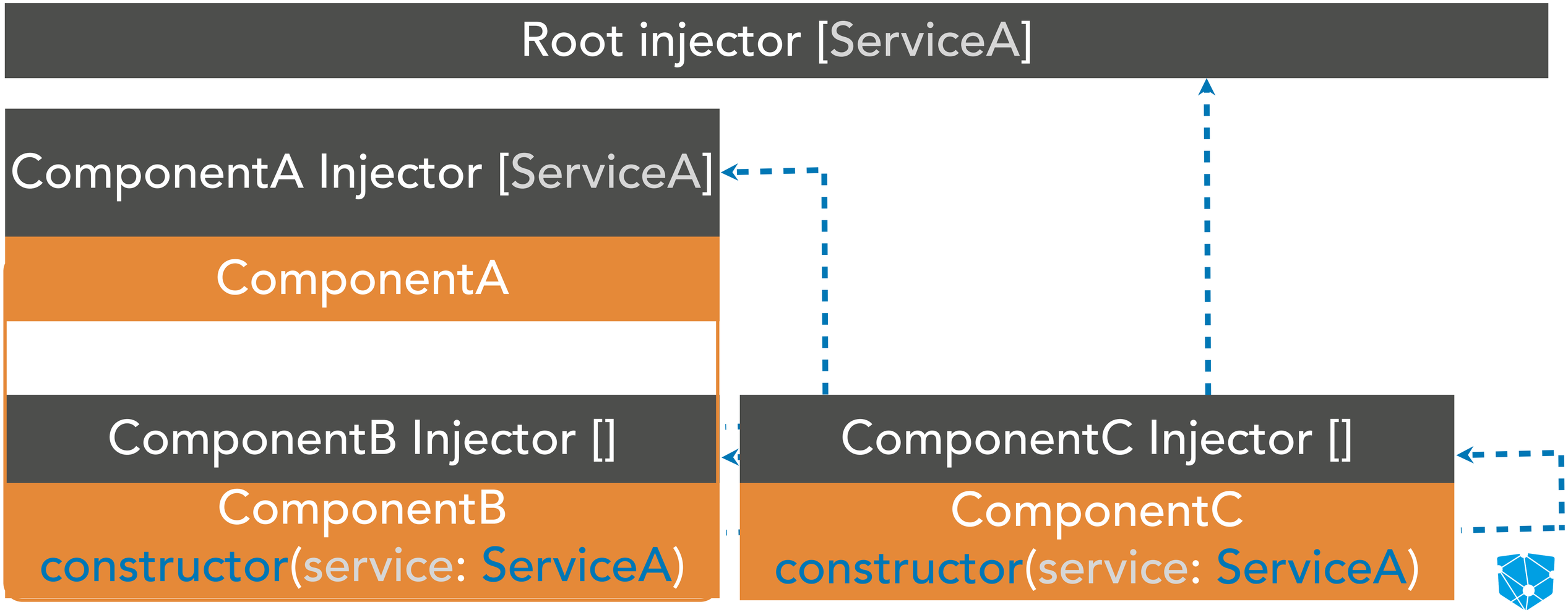
Dependency injection



Dependency injection



Dependency injection



Provide values in the injector

- Use of StaticProvider types instead of classes
  - ValueProvider
  - ClassProvider
  - ExistingProvider
  - FactoryProvider

ValueProvider

- Registering Values in the Injector
  - provide: any
    - Reference to inject
  - useValue: any -
    - value
  - multi?: boolean
    - Use as list

Using the injected value

- Values provided in the injector can be injected
  - @Inject Decorator
    - Reference
    - token

ClassProvider

- Registering Classes in the Injector
  - Like ValueProviders
  - useClass: Type<any> - instead of ~~useValue~~
    - class
      - should be exported for aot in ES6-module

ExistingProvider

- Use existing values register again
  - Like ValueProviders
  - useExisting: any - instead of ~~useValue~~
    - Reference to an object that has already been registered

FactoryProvider

- FactoryMethod to register in Injector
- Like ValueProviders
  - useFactory: Function – instead of ~~useValue~~
    - factory method
  - deps: [any]
    - List of dependencies

DI decorators

- @Injectable - Decorates service classes so that they can use the DI in the constructor. Def. target injector
- @Inject - injected using a token
- @Optional - used before @Inject, allows optional injection
- @Self, @Host, @SkipSelf - used to control injector bubbling

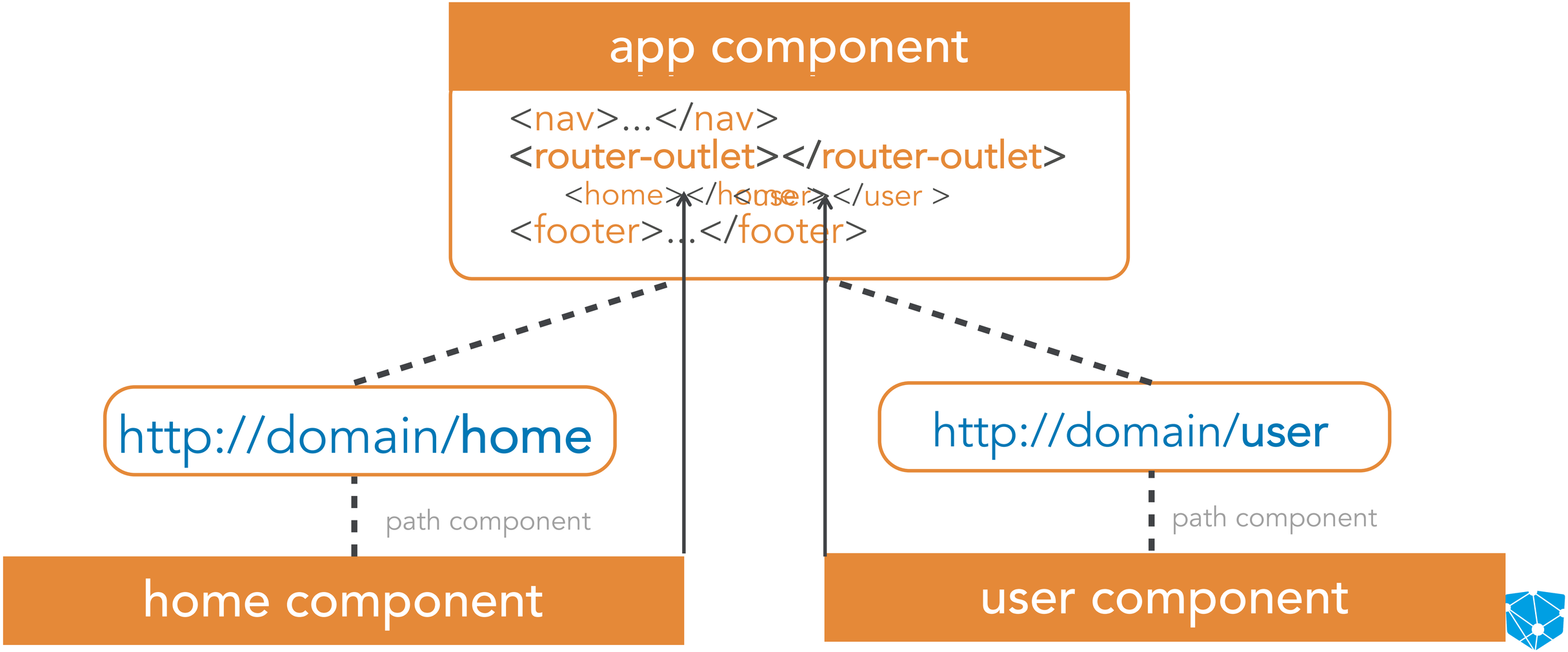
InjectionToken

- Creates reference tokens for a DI
  - Generic type refers to value type of DI

ModulesWithProviders

- Create a module
  - Metadata contains general imports and declarations etc.
  - Module class has a static Factory
    - Return is the module itself with configured Provides => ModuleWithProviders

ModulesWithProviders

- ```
  @NgModule ( {...} )
  export class MyModule {
   static forRoot ( config ): ModuleWithProviders {
       return {
         ngModule : MyModule,
         providers: [ ... ]
       };
   }
  ```
  ```

HTTP

CRUD via HttpClient

Using

- Import HttpClientModule
- Inject HttpClient service
- methods
  - Use one of the CRUD services
    - request<R> method = basis of all other methods
      - observable<R>

HttpRequest methods

- Parameter HttpRequest or:
  - method: string,
    - 'DELETE'|'GET'|'POST'|'PUT'
  - url: string,
  - options?: object for detailed specification
- Returned value: observable

Request options

- body?: any;
- headers?: HttpHeaders;
- params?: HttpParams;
- reportProgress?: boolean;
- withCredentials?: boolean;

Request options

- responseType: 'arraybuffer' | 'blob' | **'json'** | 'text';
- observe: 'body' | **'events'** | **'response'**
- Both parameters determine the return type for the request.

observe	responseType	return
body	arrayBuffer	Observable<ArrayBuffer>
body	blob	Observable<Blob>
body	text	Observable<string>
body	json	Observable<Object | R>

Request options

observe	responseType	return
events	arrayBuffer	Observable<HttpEvent<ArrayBuffer>>
events	blob	Observable<HttpEvent<Blob>>
events	text	Observable<HttpEvent<string>>
events	json	Observable<HttpEvent<Object | R>>
response	arrayBuffer	Observable<HttpResponse<ArrayBuffer>>
response	blob	Observable<HttpResponse<Blob>>
response	text	Observable<HttpResponse<string>>
response	json	Observable<HttpResponse<Object | R>>

Response types

- HttpResponse
  - body: T | null
  - headers: HttpHeaders
  - status: number
  - statusText: string
  - url: string | null
  - ok: boolean
  - type: HttpEventType.Response

Response types

- HttpEvent
  - Sent request transmitted
  - UploadProgress - Upload progress event (loaded#total)
  - ResponseHeader - Receive response status code and header
  - DownloadProgress - Download Progress Event (loaded#total)
  - Response - Complete answer incl. body
  - User - User-defined events

HTTP Service Methods

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

HttpInterceptor

- Requests and answers can be intercepted
- Service that implements the HttpInterceptor interface
  - intercept - method
    - req: HttpRequest<**any**>,
    - next: HttpHandler
      - -> Observable<HttpEvent<**any**>>
        - return next.handle(req);

HttpInterceptor - provide

- provide:
  - HTTP_INTERCEPTORS,
- useClass:
  - Name of Interceptor Service,
- multi :
  - true

HttpInterceptor - NoCache

```
// needed for IE 11
intercept(req: HttpRequest<any>, next: HttpHandler):
Observable<HttpEvent<any>> {
 req = req.clone({
     setHeaders: {
       'Cache-Control': 'no-cache',
       Pragma          : 'no-cache',
       Expires         : 'no-cache',
       'Content-Type' : 'application/json',
       Accept          : 'application/json'
     }
 });
 return next.handle(req);
}
```

HttpInterceptor - Progress & Error

- intercept (req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
  console.log ('running Requests (start new)', ++numOfRunningRequests);
  return next.handle (req)
  .pipe(
  tap((event: HttpEvent<any>) => {
  if (event instanceof HttpResponse) {
  console.log ('running Requests (end success)', --numOfRunningRequests);
  }
  }, (error: any) => {
  if (error instanceof HttpErrorResponse) {
  console.log ('running Requests (end err)', --numOfRunningRequests);
  }
  })
  );
  }

Routing

Basis of a SPA

Routing

- Component of the routing module
- Basis of a Single-Page-Application
- Determines which components are displayed at which path

Routing


app component
<nav>...</nav>
<router-outlet></router-outlet>
<home></home>
<user></user >
<footer>...</footer>
http://domain/home
path component
home component
http://domain/user
path component
user component

Module import and Route-Def

- Include module via RouterModule.forRoot incl. config
  - Routes { path, component }
    - { useHash: false }
    - Optionally, routes can also be configured via the router service and the config list at runtime.
- include <router-outlet></router-outlet>

Redirect

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

Navigation

- routerLink - Directive
  - path [path, ...params: any[]]
- routerLinkActive - Directive
  - CSS class name

Navigation - via RouterService

- DI Router Service
- navigate method
  - params
    - trick
      - path
      - params

events

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

Child

- A route can have subroutes
- These must be defined in the config under the property
  - children
    - can be created in the same way as the existing configuration

Lazy Modules

- loadChildren enables easy implementation in the CLI context

- path : 'dash',
  loadChildren : import('./dash/dash.module')
  . then(m => m.DashModule)

  - The path to the module must be imported and the module class must be returned as promise

  - ng g m saban --routing --module app --route=saban

Lazy Modules

- In the module itself, the route is defined with the component to be displayed.

- RouterModule.forChild ([

```
{
    path      : '',
    component: DashComponent
}
])
```

Lazy Modules

- Preloade Modules
- RouterModule.forRoot ([], opt)
  - opt
    - enableTracing: true,
    - preloadingStrategy: PreloadAllModules

Parameter

- Define route with parameters
  - path : 'details/:id',
    component : UserDetailsComponent
- Inject service into ActivatedRoute component
  - this.subscription = this.route.paramMap.pipe (
    .map (paramMap => paramMap.get ('id')))
    .subscribe(id => this.param_id = id);

Resolve-Guard

- Obtain data before changing routes
  - Create, embed and route ResolveService based on the Resolve interface
    - path : 'details/:id',
      component : UserDetailsComponent,
      resolve: {
       user: ResolveService
      }

CanActivate - Guard

- Approving 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

CanActivate - Guard

- Service is implemented in the route definition.
  - path: 'home',
    component: HomeComponent,
    canActivate: [CanActiveService]

Named Outlet

- Apply name to Outlet:
  < router-outlet name="modal"></router-outlet>
- Reference path to outlet name
  { path: 'modA', component: AComponent, outlet:'modal' }
- Navigate
- [routerLink]="open"
- this.router.navigate([outlets: { modal: null } }]);

Forms

Forms

- Can be implemented in two ways
  - Template-driven
    - The template specifies the form model and the validators (similar to AngularJS).
  - Reactive (data-driven)
    - The form elements are planned in advance and linked to a form in the template.

Forms - Template-driven

- Preparatory: Integration of the FormsModule
- Form directives can then be used in the template layer:
  - ngModel, required, minlengt, ...
  - for binding validators and values to the form model
  - All this is realized without additional programming

Forms - Template-driven

- ngForm - used to mark up the form.
  - directive has an exportAs i.e. we can assign this for a #Hash-Id #myForm='ngForm'
    - Allows access to control properties
      - valid, invalid, value etc.
      - myForm.valid

Forms - Template-driven

- ngModel can be used in three ways
  - As attribute directive ngModel combined with a name definition via the name attribute.
    - This automatically creates a form model.
    - myForm.value = {name: Input field value}
  - As an attribute directive with binding of an initial value [ngModel].

Forms - Template-driven

- Avoid: Usage as attribute directive with two-way binding [(ngModel)]. This updates the initial values. I.e. there are two models ☹
- As assignment for a #Hash-Id e.g. #mail='ngModel'
  - Combined with the ngModel directive it allows access to: valid, invalid, value etc.
    - mail.valid

Forms - Template-driven

- ngModelGroup directive for grouping model information
  - The directive must be used hierarchically in the template.
    - The input nodes of the directive element create the group elements.

Form

```
<form novalidate #myForm="ngForm">
  <input type="text"
         autocomplete="name"
         placeholder="name"
         name="name"
         #name="ngModel"
         ngModel
         >
  <span ngModelGroup="credentials">
    <input name="email"
           #email="ngModel" ngModel>
    <input name="password"
           #password="ngModel" ngModel>
  </span>
</form>
```

Model

ngForm -> myForm

ngModel -> name

ngModelGroup -> credentials

ngModel -> email

ngModel -> password

Form

```
<form novalidate #myForm="ngForm">
  <input type="text"
         autocomplete="name"
         placeholder="name"
         name="name"
         #name="ngModel"
         ngModel
         >
  <span ngModelGroup="credentials">
    <input name="email"
           #email="ngModel" ngModel>
    <input name="password"
           #password="ngModel" ngModel>
  </span>
</form>
```

Model

```
myForm. value = {
    name: '... ',
    credentials {
        email: '... ',
        password: '... ',
    }
}
```

Forms - Controls

- ngForm and ngModel - are control directives with the following properties:
  - value - Value
  - valid, invalid - valid
  - touched, untouched - Touched
  - dirty, pristine - Used/unused
  - errors? - Validator error

Forms - Controls

- Control Methods:
  - setValue, reset - Value
  - markAsTouched, markAsUntouched - Touched
  - markAsDirty, markAsPristine - Used/unused
  - setErrors? - Validator error

Forms - Validators

- Validators can be integrated via directives
  - required - required value
  - email - valid mail
  - minlength, maxlength - length testing
  - pattern - expression check

Forms - Validators

- Validators store error information in the errors object of the control depending on the validator.
  - Error messages can be displayed accordingly
  - < div *nglf="*email*.errors?.required">...
    - The question mark binds optional values

Forms - Send data

- (submit) - Use the submit event on the form.
  - Use as trigger in the form a <button> or <a> of type submit
    - Also use the disable directives on the trigger to disable invalid forms.

```
<form novalidate #myForm="ngForm" (submit)="send(myForm)">

<button type="submit" [disabled]="myForm.invalid">senden</button>
```

Forms - Reset data

- (reset) - Use the reset event on the form.
  - Use a <button> or <a> of type reset as a trigger in the form.
    - On the trigger, additionally use the disable directives to disable forms if no form values have been entered yet.

```
<form novalidate #myForm="ngForm" (submit)="send( myForm )"
      (reset)="reset( myForm, $event )">

<button type="reset" [disabled]="!myForm.dirty">reset</button>
```

Form CSS Classes

- Angular automatically adds CSS classes to input elements that reflect the status of the control.
  - ng-untouched, ng-touched
  - ng-pristine, ng-dirty
  - ng-invalid, ng-valid

Model options

- The directive of the same name influences model handling.
  - [ngModelOptions]="{name: 'name'}"
    - replaces the setting of the name attribute
  - [ngModelOptions]="{standalone: true}"
    - Value is not communicated to the superior form.

model options

- [ngModelOptions]="{updateOn : 'blur '}"
  - Defines a form hook (change, submit, blur) where the model should be updated.
  - debounce - announced: update after timeout.

Reactive Forms

- In contrast to template driven forms we avoid directives like: ngModel, required, minlength
- Instead, controls are created first and then bound in the template via:
  - formGroup, formControl, formControlName ...
- In preparation, the ReactiveFormsModule must be integrated.

Reactive Forms - Create Model

- Create controls for values via FormControl
  - Constructor expects value and validators
- Create groups of values using FormGroup
  - constructor expects a key value pair object
    - Key: Name of the control or subgroup
    - Value: Instance of the control or subgroup

Reactive Forms - Directives

- [formGroup] - binds the lowest value group
- formGroupName - Binds subgroups based on the name defined in the Key-Value-Pair object.
- formControlName - Binds Control by the name defined in the Key-Value-Pair object.
- [formControl] - Binds a Control instance.

Form

```
<form novalidate [formGroup]="myForm">

  <input type="text"
         formControlName="name"
  >
  <span formGroupName="credentials">
        <input type="email"
               formControlName="email" >
        <input type="password"
               formControlName="password">
        </span>
</form>
```

Model

```
this.myForm = new FormGroup ( {
  name: new FormControl ( 'Saban',
     Validators.required ),
credentials: new FormGroup ( {
     email    : new FormControl (
                  'us@netTrek.de',
                     [ Validators.email,
                     Validators.required ] ),
password: new FormControl ( ... ) } );
```

Form

```
<form novalidate [formGroup]="myForm">

  <input type="text"
         formControlName="name"
  >
  <span formGroupName="credentials">
        <input type="email"
               formControlName="email" >
        <input type="password"
               formControlName="password">
        </span>
</form>
```

Model

```
myForm.value = {
    name: '...',
    credentials {
        email: '...',
        password: '...',
    }
}
```

Forms - Helpers - FormBuilder

- FormBuilder (DI) service simplifies model creation and handling with FormControl and FormGroup
  - Instead of new FormGroup () we use the group method of FormBuilder and pass a Key-Value object.
    - Key: Name of the control or subgroup
    - Value: Property array or subgroup via group method

Forms - Helpers - FormBuilder

- Value: Property array
  - First element: Start value
  - Second element: Validator or Validator array
  - Third element: AsyncValidator | AsyncValidator array

Forms - Helpers - FormBuilder

```
this.myForm = this.fb.group( {
    name: [ 'Saban', Validators.required ],
    credentials: this.fb.group ( {
        email: ['us@netTrek.de', [ Validators.email,
                                   Validators.required]],
        password: ['test1234', Validators.required ]
    })
});
```

Forms - Helpers - Control

- get: method returns a control from the model
  - Parameters:
    - Name of the control
      - or path (names array) to a control
    - this.myForm.get(['credentials', 'email']) as FormControl;

Forms - Helper - Control - Error

- hasError : Method returns a boolean if a certain validator error exists.
  - Parameters:
    - Name of the error e.g. required, email ...
    - Name of the control or path (names array) to a control.

Forms - Helper - Control - Values

- setValue(value: any, opts?): void;
  - onlySelf? : boolean [default: false]
    - Validation only on control not on parent component
  - emitEvent? : boolean [default: true]
    - valueChanges event is fired by the control

Forms - Helper - Control - Values

- setValue(value: any, opts?): void;
  - emitModelToViewChange? : boolean
    - View is informed about the change via onChange
  - emitViewToModelChange? : boolean
    - Model is informed about the change via ngModelChange

Forms - Helper - Control - Values

- reset(value, opts?: { onlySelf?: boolean;
  emitEvent?: boolean; })
  - Resets control
    - value = zero or value
    - state is set to pristine & untouched

Forms - Helper - Control - Status

- markAsTouched(opts?: { onlySelf?: boolean; }): void;
- markAsUntouched(opts?: { onlySelf?: boolean; }): void;
- markAsDirty(opts?: { onlySelf?: boolean; }): void;
- markAsPristine(opts?: { onlySelf?: boolean; }): void;
- disable(opts?: { onlySelf?: boolean; emitEvent?: boolean; })
- enable (opts?: { onlySelf?: boolean; emitEvent?: boolean; })

Validator Function

- Function is generated via a Factory, which optionally accepts the test condition.
  - ValidatorFn - expectes: AbstractControl
    - Returns an error object (ValidationErrors) or null.

Validator Function

```
export class EqualValidator {
    static isEqual ( compare: any ): ValidatorFn {
        return ( control: AbstractControl ): ValidationErrors | null => {
            if ( control.value === null || compare === null ) return null;
            return compare !== control.value ?
                { 'equal': { 'is': control.value, 'should': compare } } : null;};
    }
}
```

Validator Directive

- Form validation is enhanced with NG_VALIDATORS, by a new directive
- extension is provided in the injector of the directive.
- directive must implement the validator interface.
  - validate (c: AbstractControl): ValidationErrors | null
    - Is executed for check and returns error object

Validator Directive

- registerOnValidatorChange (fn: any): void;
  - Methods transmit a reference to the onChange Callback
    - Necessary if test conditions change

Validator Directive: Extending the Injector

- Extend the NG_VALIDATORS
- By using the ExistingProvider
- And NG_VALIDATORS Provide-Token:
- The value is set via useExisting to the directive class; Since this does not exist directly in the injector, the *forwardRef method is used.*
- multi: true extends the NG_VALIDATORS list

Validator Directive: Extending the Injector

- export const EQUAL_VALIDATOR = {
  provide: NG_VALIDATORS, multi: true,
  useExisting: forwardRef(() => MyDirective) };

- @Directive ({ selector: ‚[equalValidator][ngModel]‘,
  providers: [EQUAL_VALIDATOR] })

- export class MyDirective implements Validator {

Values Access

- If values need to be manipulated before they are saved in the model or displayed in the view
  - Access Control: NG_VALUE_ACCESSOR extended by new directive
  - Provide an extension in the injector of the directive.
  - directive must implement the ControlValueAccessor interface.

Value Access - Interface Methods

- writeValue(obj: any): void;
  - Called when model changes from Form API. Method must adjust view.
- registerOnChange & registerOnTouched (fn: any): void;
  - Methods transmit a reference to the onChange and onTouched callback

Value Access - Interface Methods

- onChange (value): void;
  - Reference must be called if the user changes values in the input field.
  - The current value from the UI is transmitted.
  - Callback updates the model using the Form API

Value Access - Interface Methods

- onTouched (): void;
  - Reference must be called if the status of the control needs to be changed.
  - Status indicates whether the form element has been activated (focus/blur is sufficient).

Value Access - Extend Injector

- Extend the NG_VALUE_ACCESSOR
- Use the ExistingProvider to do this.
- Use as provider token: NG_VALUE_ACCESSOR
- The value is set via useExisting to the directive class; Since this does not exist directly in the injector, the *forwardRef method is used.*
- multi: true extends the NG_VALUE_ACCESSOR list

Value Access - Extend Injector

- export const CONTROL_VALUE_ACCESSOR = {
  name: 'formatterParserValueAccessor',
  provide: NG_VALUE_ACCESSOR, multi: true,
  useExisting: forwardRef(() => MyDirective) };

- @Directive ({ selector: 'input[msgFormater]',
  providers: [CONTROL_VALUE_ACCESSOR] })

- export class MyDirective implements ControlValueAccessor {

Performance and Architecture

ChangeDetection & Rendering

Components and Directives Ecosystem

- Each component belongs to a ViewRef
  - Block of elements => user interface
  - Basis for change recognition.
- Components and Directives DI Properties
  - ElementRef
    - Element reference (access to native element)

Components and Directives Ecosystem

- Components and Directives DI Properties
  - ViewContainerRef
    - Reference to the container for dynamic contents, which are appended behind the element <router-outlet>
      - Embedded view - via template references
      - Host View - via component references

Create dynamic content

- Using templates dynamically
  - Structural directive: *ngTemplateOutlet
    - Template reference and context
  - ViewContainerRef (via DI)
    - createEmbeddedView - method
      - templateRef : TemplateRef<T>

Create dynamic content

- Own structural directives can be used as template reference
  - via @ViewChild or @ContentChild the directive can be determined
    - by means of read property the template reference of the directive can be determined
      - @ContentChild (UserHeaderDirective, { read: TemplateRef })

Creating Dynamic Components

- Include components in entryComponents
  - Structural directive: *nComponentOutlet
    - Expression for binding a component class
  - Optional features
    - ngOutletInjector, ngOutletProviders & ngOutletContent

Creating Dynamic Components

- Create Components Factory
  - ComponentFactoryResolver (DI)
    - resolveComponentFactory method
      - Parameters: component
        (must be in entryComponents)

Creating Dynamic Components

- Append dynamic component via Factory
- ViewContainerRef (via DI)
    - createComponent - method
        - compFactory : ComponentFactory<T>

Change Detection

Change recognition

- Is executed for each asynchronous event.
- The detection takes place in chain
  - Parent View -> Child View
- Hooks are executed before (init, change, doCheck) and after (after) detection
- DOM update is the result of change detection

Change recognition: Under the hood

- The ChangeDetectionRef is the super class of the ViewRef
- Each directive and component is part of a view
- The view thus controls change recognition and DOM updating.
- The ChangeDetectionRef can be injected via DI

Change recognition - influencing

- ChangeDetectionRef
  - detach()
    - Removes component from change recognition
    - Analog - ChangeDetectionStrategy.*OnPush*
  - reattach()
    - Appends component back to detection

Change recognition - influencing

- ChangeDetectionRef
  - markForCheck()
    - marked as dirty
    - Recognition parents -> children
  - detectChanges()
    - self -> children

Change recognition - influence

- Bypass change recognition
  - *ngZone (DI)*
    - *runOutsideAngular*

Change recognition - OnPush

- Change Recognition with the Push Strategy
  - Triggered by asynchronous events
  - Is triggered when the value of a binding changes
    - *async Pipe* - Supported during triggering

Unit-Testing in Angular

Unit-Testing: Contents

- Unit-Tests and their role & limitations
- Frameworks and their relevance
- Basics & settings

Unit Testing: Role

- Unit tests are programmatic tests
- Isolated testing of components (units) of an application
- Usually running automated

Unit Testing: Restriction

- No user interactions
- No workflows

Unit Testing: Frameworks

- Angular-CLI implements & configures frameworks automatically!
- For unit tests
  - Karma
  - Jasmine

Unit-Testing: Frameworks - Karma

- Test runner from the Angular Team
  - Runs through all tests in the source directory
- Can run headless or in the browser
- Executes the programmed tests
  - Jasmine, Mocha and QUnit
- Results Output

Unit-Testing: Frameworks - Karma

- file naming convention
  - so karma can identify the test files.
  - <Filename>.spec.ts
  - Adjustable in "test.ts"

Unit Testing: Frameworks - Jasmine

- Popular Open Source test framework
- Based on behaviour-driven development
- Independent of JavaScript frameworks
- No DOM required

Unit Testing: Basics

- Environment and Modules
  - typings
    - @types/jasmine
    - @types/jasminewd2

Unit Testing: Basics

- Jasmine Core Modules
  - jasmine-core
  - jasmine-spec-reporter

Unit Testing: Basics

- Karma Core Modules
  - karma
  - karma-chrome-launcher
  - karma-coverage-istanbul-reporter
  - karma-jasmine
  - karma-jasmine-html-reporter

Unit Testing: Setup for Angular

- angular.json
  - projects > [name] > architect > test
    - main > **test.ts**
    - tsConfig > src/tsconfig.**spec.json**
    - karmaConfig > src/karma.**conf.js**

Unit Testing: Setup test.ts

- Is included by the karma.config
- Initializes test environment: TestBed
  - For JIT Env. with BrowserDynamic
- Defines the test context
    - All *.spec.ts files, within the project

Unit testing: setup *karma.conf.js*

- Configuration of the Karma test runner
  - base path
  - Frameworks that are used
    - Jasmine
    - @angular-devkit/build-angular

Unit Testing: Setup *karma plugins*

- Plugins & Adapters
  - karma-jasmine
  - karma-chrome-launcher
  - karma-jasmine-html-reporter
  - karma-coverage-istanbul-reporter
  - build-angular/plugins/karma

Unit Testing: *Config karma plugins*

- Client - clearContext
- coverageIstanbulReporter
  - output folder
  - report
    - HTML
    - LCOV (graphical output)

Unit Testing: *Config karma plugins*

- reporter
  - progress (tests performed and summary)
  - kjhtml (karma-jasmine-html-reporter)
- browser
  - Parallel running browsers possible
  - Install and integrate Launcher Plugin

Spec files

- Suites - describe (description: string, ()=> {})
  - Grouping of associated specifications of a test used
  - Each test file usually has a describe block at the top level.
  - Multiple suites may exist per *.spec.ts file

Writing a test

Spec Files: Creating Suites

- describe (description: string, specDefinitions: ()=> {})
  - Description set (suite), which assumptions are tested
  - Define assumptions within callback: specDefinitions
    - Structure -> Test an acceptance -> Reduction
    - If necessary, further suites

Spec files: Structure of a test

- Define test setup processes (init)
  - beforeAll
    - Once before all tests
  - beforeEach
    - Before each test

Spec files: Structure of a test

- Setup processes parameters
  - action
    - () => PromiseLike | void
    - done => void
      - Done: CallBack defines async process as done

Spec files: Structure of a test

- Setup processes parameters
  - timeout
    - MS that are waited until the Async process is interpreted as a failure

Spec files: Dismantling of a test

- Define destroy processes of tests
  - afterAll
    - Once after all tests
  - afterEach
    - After each test
- Used like before... ()

Spec files: Async - Helper

- Angular Test Helper
  - async
    - Expectes action definition (see before, after)
    - All promises defined in the callback
      - Wait until resolved

Spec Files: Define Test

- Define expectation (Test)
  - it(expectation: string, assertion: () => Promise |
    done => void);
- A suite should contain one or more expectations
- The suite is only evaluated as successful if all expectations are successful.

Spec Files: Define Expectation

- An expectation is defined within an expectation (it) callback.
  - expect(actual).[match] (expectation): boolean;
    - Is the technical implementation of the test

Spec files: compliance methods

- Test of conformity
  - **toBe** (val) -> comparable ===
    - **not.toBe**(val) -> comparable !==
  - **toEqual**(val) -> compares object and **all fields**
  - **toMatch**(regExp) -> comparable to **regExp**
  - **toBeDefined** (val) -> comparable !== **undefined**

Spec files: compliance methods

- Test of conformity
  - **toBeUndefined** (val) -> comparable === **undefined**
  - **toBeNull** (val) -> comparable === **zero**
  - **toBeTruthy**(val) -> comparable === **Boolean(val)**
  - **toBeFalsy** (val) -> comparable !== **Boolean(val)**
  - **toContain** (val) -> comparable **indexOf** !== **-1**

Spec files: compliance methods

- Test of conformity
  - **toBeLessThan** (val) -> comparable < **val**
  - **toBeGreaterThan** (val) -> comparable > **val**

TestBed

- Configure TestBed (Angular) in beforeEach/All
  - configureTestingModule Factory testing module
    - Use before each test in sleep mode
    - as parameter a NgModule MetaData object is passed
    - compileComponents() - compiles all components in the module to Inline JavaScript

TestBed

- compileComponents()
  - and all matching methods
    - return a Promise
- Handlers for beforeEach, afterEach, and it are therefore often
- encapsulated in the async method.
  - *beforeEach*(*async*(() => { ... })

TestBed

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

TestBed

- createComponent(Component Class) method closes the TestBed configuration and returns a ComponentFixture instance.
  - fixture = TestBed.createComponent (AppComponent);
  - provides access to a debug element and the instance of the component.
    - component = fixture.componentInstance;
    - fixture.debugElement.componentInstance

TestBed

- detectChanges () method of the **ComponentFixture** instance performs change detection & renders the template
  - fixture.detectChanges ();
- The Fixture debug element provides the nativeElement of the component
  - fixture.debugElement.nativeElement

TestBed

- The nativeElement supports querySelector.
  - fixture.debugElement.nativeElement.querySelector ('h1')
- Alternatively, query from the debugElement can be combined with by.
  - fixture.debugElement.query(By.css ('h1'));

Unit Test Examples

```
it('should create the app', () => {
    const fixture =
        TestBed.createComponent(AppComponent);
    const app =
        fixture.debugElement.componentInstance;
    expect(app).toBeTruthy();
  });
```

Unit Test Examples

```
it (`should have as title 'testing'`, () => {
     const fixture =
         TestBed.createComponent(AppComponent);
     const app =
         fixture.debugElement.componentInstance;
     expect(app.title).toEqual('testing');
  });
```

Unit Test Examples

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

it ( 'should render title in a h1 tag', () => {
    componet.changeTitel( 'test' );
    fixture.detectChanges();
    expect ( htmlElem.textContent )
      .toContain ( `Welcome to test!` );
  } );
```

Testing Services

- Provide the service in the TestBed module
  - request via TestBed.get or inject method
  - beforeEach (() => { TestBed.configureTestingModule ({
    imports : [HttpClientTestingModule],
    providers: [AppService] });
    service = TestBed.get (AppService);
    httpMock = TestBed.get (HttpTestingController);
    });

HttpClientTestingModules

- Use the module as a dependency in TestBed
- Use the HttpTestingController to mock
- Create a TestRequest with the Ctrl.
  - testRequest = httpMock. expectOne(url)
- send response using flush method.
  - testRequest. flush(body, opts?);

HttpClientTestingModules

- it ('shold getUsers', () => {
  const dummyUsers: User[] = [
  { name: 'saban', age: 33 }, { name: 'peter', age: 22 },];

  service.getUsers(); const testRequest =
  httpMock.expectOne(`${environment.endpoint}/users`);

  testRequest.flush(dummyUsers);
  expect(service.users).toBe(dummyUsers);
  });

Helper - HttpClientTestingModule

- Remove Requests not used in afterEach

- ```
  afterEach() => {
    httpMock.verify();
  });
  ```
  ```

e2e-Testing

E2E: Overview

- End-2-End Tests and your role & limitations
- Frameworks and their relevance
- Basics & settings

E2E: Relevance

- Blackbox E2E tests
- Performs tests against the application in the browser
  - Interacts like a user
- no substitute for unit tests
  - very good supplement

E2E: Relevance

- Unlike unit tests.
  - GUI tests
    - Button clicks
    - Text input and user interaction
    - navigation
    - ....

E2E: Frameworks

- Angular-CLI implements & configures frameworks automatically!
  - E2E tests:
    - Protractor
    - Jasmine

E2E: Protractor

- Developed by Google based on
  - Selenium
  - Webdriver

E2E: Jasmine

- Tests are written and configured in Jasmine
  - Detailed description see Unit tests

E2E Basics: Folder Structure

- Located in a separate directory
  - /e2e/
    - protractor.conf.js
    - tsconfig.ts
    - /src/
      - app.e2e-spec.ts & app.po.ts

E2E Basics: Configuration

- Angular.json
  - Own task for e2e
    - protractorConfig
      - path
      - devServerTarget
      - execution context

E2E Basics: protractor.conf.js

- Timeout (allScriptsTimeout) and file pattern (specs) for the tests
- Test environment by using capabilities and multiCapabilities
  - Browser, Resolution
  - access data
- Selenium Hub (seleniumAddress)

E2E Basics: protractor.conf.js

- Definition & configuration of the test framework
  - framework & jasmineNodeOpts
- Connection type: directConnect
- Website to test: baseUrl

E2E Basics: protractor.conf.js

- hooks
  - onPrepare e.g. To register the **SpecReporter** for the Jasmin-Plugins
  - beforeLaunch e.g. to carry out hub configurations (credentials)
  - afterLaunch e.g. to end hub tests

E2E basics: tsconfig

- Extension of the project-tsconfig.json
- Definition
  - of the EcmaScript standard to be used: es5
  - The node module types to use: commonjs
  - Pass types: "jasmine", "jasminewd2", "node"

E2E: Page Objects (app.po)

- Description of the high-level page view
- "functional abstraction" of the Protractor accesses to the elements of a web page
  - Definition of methods for accessing web pages and page elements.
    - Used in e2e-spec.ts

E2E: ProtractorBrowser

- from the Protractor module
- Alias: browser: ProtractorBrowser
  - get(browser.baseUrl): Celebrities;
  - getTitle(): Celebrities<string>
  - waitForAngularEnabled (true);
  - executeScript (execStr);

E2E: ProtractorBrowser

- sleep(timeoutMS);

E2E: ElementHelper

- element (by-statement [Locator]): ElementFinder
  - all(by-statement [Locator]);
- by: ProtractorBy
  - css(selector);
- element(by.css('pr-root h1')) => WebElement

E2E: WebElement

- element: WebElement
  - click( );
  - sendKeys(keyslstring [])
  - getCssValue(cssProp)
  - getText()
  - isEnabled()

E2E: WebElement

- element: WebElement
  - isSelected()
  - isDisplayed()
  - submit()

E2E: Spec

- *.*.e2e-spec.ts
  - Actual test file(s)
  - Structure identical to the unit tests (Jasmine)
  - expect(page.getParagraphText()).toEqual('Welcome!');

THANKS

- https://bit.ly/2Jzt12i