
Course Fincons: 3. / 4. Sept 2019

Part 2: Angular



Learning Targets LE 05 Angular

- The student:
 - Knows the basics of the Angular framework
 - Can use the built-in elements of Angular like directives, decorators, DI
 - Can create own elements like components, services, routes, directives, pipes
 - Can use the RxJs Library for Observables
 - Can successfully solve the tasks of a case study
 - Can create a frontend UI with communication to a REST/Websocket backend
 - Can create tests like unit- and e2e-test
 - Knows the tool chain (ng-cli, node, npm, IDE's)

Agenda

- ng-cli
- Modules, @NgModule
- Bootstrap
- Components, @Component
- DOM, Template Syntax
- @Input, @Output
- Structure Syntax
- Interfaces, Services
- Dependency Injection
- Observables
- HTTP-Service
- Component-LifeCycle-Hooks
- Routes
- Guards
- Forms, Template-/Model-Driven
- Directives
- Pipes
- Authentication, Security, JWT
- zone.js
- Cheat Sheets

Links

- Angular: <https://angular.io/>
- ng-cli: <https://github.com/angular/angular-cli>
- TypeScript: <https://www.typescriptlang.org/>
- ES6: <http://es6-features.org>
- Angular 7 comp: <https://colorlib.com/wp/angular-2-components/>
- ES2019, 2018

Angular Roadmap

Predictable, Transparent & Incremental Evolution

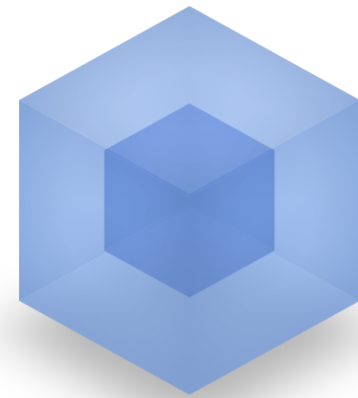
Version 4	March 2017
Version 5	September/October 2017
Version 6	March 2018
Version 7	September/October 2018

(tentative schedule)

Angular CLI

Angular CLI

- No more seeds and fragmentation
- No more discussions about style
- Proven directory structure
- Based on
 - [embed-cli](#)
 - [Webpack](#)
- Tutorial
 - <https://angular.io/docs/ts/latest/cli-quickstart.html>
- <https://github.com/angular/angular-cli>



webpack
MODULE BUNDLER

Angular CLI - Generator

Type	Usage
Component	<code>ng g component my-new-component</code>
Directive	<code>ng g directive my-new-directive</code>
Service	<code>ng g service my-new-service</code>
Pipe	<code>ng g pipe my-new-pipe</code>
Interface	<code>ng g interface my-new-interface</code>
Class	<code>ng g class my-new-class</code>
Guard	<code>ng g guard my-new-class</code>
Enum	<code>ng g enum my-new-enum</code>
Module	<code>ng g module my-new-module</code>

angular-cli

Features

- webservice
- build process
- testing
- deployment

```
src
\ - main.ts
\ - app
    \ - auction-list
        \ - auction-list.component.css
        \ - auction-list.component.html
        \ - auction-list.component.ts
        \ - auction-list.component.spec.ts
    \ - app.module.ts
    \ - app.component.ts
    \ - index.html
```

```
$ ng --help
```

Task #00

Preparation

Modules in JavaScript

Modules - General

- organize code
- split code in multiple files
- solve a specific problem/deal with a specific topic
- share functionalities between modules

Modules

- every file is a module itself
 - only **export** what you share
 - simply **import** what you need

Modules

auction-detail.ts, auction-list.ts

```
// auction.ts
export interface Auction {...}
export let seller = 'Felix.Muster';

// auction-list.ts
import {Auction, seller} from './auction';

console.log(seller);
```

@NgModule

The decorator

@NgModule - General

- Groups code and files
- Solves a specific problem/deal with a specific topic
- Shares functionalities between Angular modules

@NgModule - Decorator

- Defines the parts of the module, e.g. components, directives,...
- Import dependencies
- Export parts to other modules
- Set base component

@Ng Module Decorator - Overview

module decorator

```
@NgModule({  
  declarations: [ // pipes, components/directives are known in the whole module  
    AppComponent,  
    AuctionListComponent // is now known in the whole module  
  ],  
  imports: [ // depends on other modules  
    BrowserModule // imports and re-exports most basic Angular directives  
  ],  
  providers: [], // list of services  
  bootstrap: [AppComponent] // there is one root component  
})  
export class AppModule {} // in most cases an empty class
```

The bootstrap function

Bootstrap

- Your application needs a starting point
- The `bootstrap` function defines the main module

Bootstrap

- every module can have a bootstrap component
- e.g. bootstrap: [AppComponent]
- Two Options
 - JIT (Just in time) Compiler (dev and prod)
 - AOT (Ahead of time) Compiler (prod)

Bootstrap (1)

An example bootstrap with JIT (just in time) compiler

src/main.ts

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

src/app/app.module.ts

```
@NgModule({  
  declarations: [  
    AppComponent  
  ],  
  // ...,  
  bootstrap: [AppComponent]  
}) export class AppModule { }
```

Bootstrap (2)

AppComponent with integrated html-template

src/app/app.component.ts

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

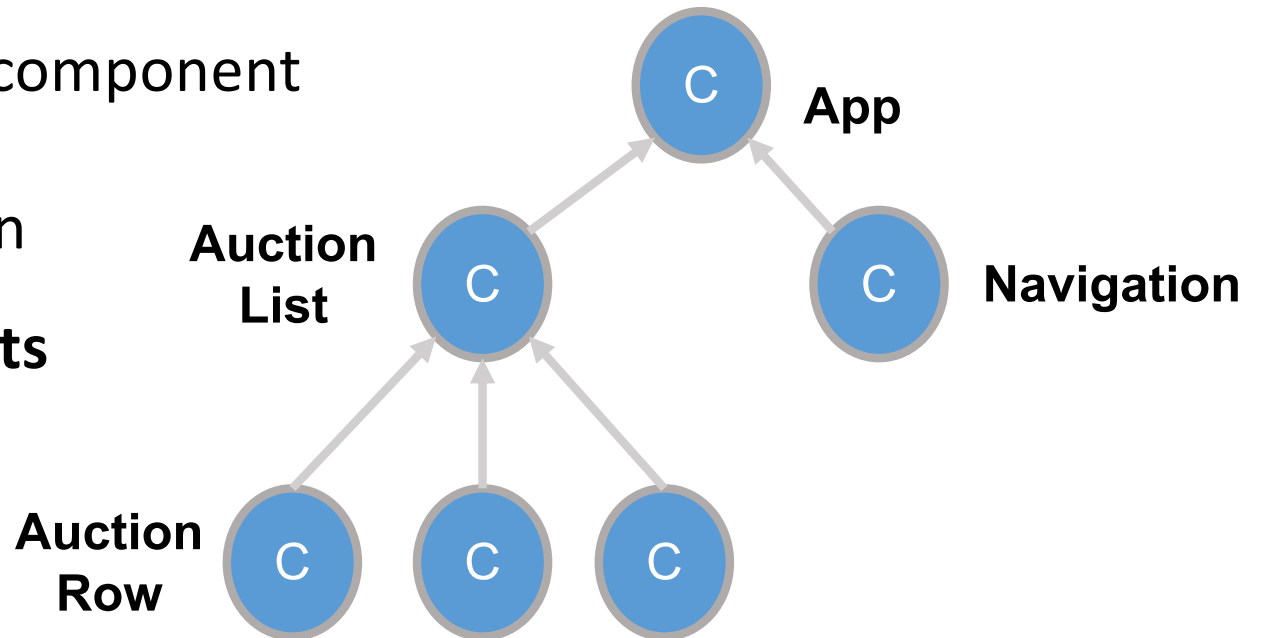
@Component({
  selector: 'my-app',
  template: '<h1>{{title}}<h1>'
})
export class AppComponent {
  title = 'Minimal Example'
}
```

<https://angular.io/docs/ts/latest/guide/ngmodule.html#!#bootstrap>

Components

Components

- fundamental **building blocks**
- application itself is a component
- break your application into **small components**



@Component, View, Component Class

```
@Component({
  selector: 'app-auction-list',
  template:
    `<h1>
      Auction-Title: {{title}}
    </h1>
    `
})
class AuctionListComponent {
  title: string;
  constructor() {
    this.title = 'An awesome auction';
  }
}
```

@Component Decorator

Component Class

@Component

The decorator

Component Decorator

- TypeScript Decorator (Annotation)
- metadata / configuration of the component

Component Decorator - Overview

Component decorator for a component class

```
@Component({  
  selector: 'app-auction-list-detail',  
  templateUrl: './auction-list-detail.component.html'  
})  
export class AuctionListDetailComponent {}
```

Template Bindings

- `{{ expression }}` Syntax (curly braces)
- Display data inside of the component view
- Possible to execute simple calculations
 - `{{ 1 + 2 + auction.startingPrice }}`
- Function calls
 - `{{ showPrice(auction) }}`
- Not a simple `eval('...')`

<https://angular.io/guide/template-syntax>

Component Class

- Define data and behavior of your component
- Possible to inject services
- Consists of
 - Methods
 - Objects
 - Arrays
 - Primitive data types (number, boolean, string, etc.)

Task #01

Build an AuctionList component

DOM

For the Event-Binding Syntax

DOM

- A **DOM node** is an **object**.
- It can store custom **properties** and **methods** like any other object.

DOM

Example for properties

```
document.body.myPerson = {  
  name: 'John'  
};  
alert(document.body.myPerson.name); // => John  
  
document.body.sayHi = function() {  
  alert(this.nodeName);  
};  
document.body.sayHi(); // BODY
```

Dom

Example for events

```
function showCurrentTime() {  
    console.log(new Date());  
}  
  
document.body.onclick = showCurrentTime;
```

Template Syntax

Event + Property Syntax

- Allows binding to the native DOM **properties** and **events**
- Allows interoperability with other components and frameworks
- Easy to use and to read

<https://angular.io/docs/ts/latest/guide/template-syntax.html>

Properties

Property-Binding Syntax

```
<h2 [style.backgroundColor]="color">Title</h2>
```

- Pass data to the native component object in the DOM
- Defines an attribute binding on the element
- Shorthand for bind-*

```
<h2 bind-style.backgroundColor="color">Title</h2>
```


Property-Bindings-Example

Set the href property of the link

```
<a [href]="auction.info.publisher.href">  
  {{auction.auctionItem.title }}  
</a>
```

Events

Event-Binding Syntax

```
<a (click)="close()"></a>
```

- Used with (event name)
- Defines a event listener on an element
- Listens to native DOM events
- Shorthand for on-*

```
<a on-click="close()"></a>
```

Event-Binding Syntax Example

- Executes a function that is defined on the component class
- Executes an expression
- `on-event=` or `(click)=`

```
<button (mouseover)="someFnOnClass()">Execute</button>  
<button (click)="isHidden = false">Show</button>  
  
<button on-click="isHidden = false">Show</button>
```

Event-Binding Syntax Events

- It's possible to access the event via `$event`
- `$event` is the actual native DOM-Event `$('#input').on('input', function(event) { });`

```
<input [value]="name" (input)="handleEvent($event)">
```

Task #02

Output mouse cursor position

incl type, ctrl, shift

Type: mouseleave

x: 312, y: 368

ctrl: false, shift: false, meta: false



Inputs & Outputs



Inputs

Input-Metadata

- **@Input decorator declares a data-bound input property**

```
export class AuctionListComponent {  
  @Input() headerTitle: string;  
}
```

- **usage in template**

```
<app-auction-list headerTitle="Example Header String As Static String">  
<app-auction-list [headerTitle]="localVariableOnComponent">
```

Input-Metadata

- Optional mapping via decorator parameter

```
export class AuctionListComponent {  
  @Input('headerTitle') title: string;  
}
```

- usage in template

```
<app-auction-list headerTitle="static string">  
<app-auction-list [headerTitle]="localVariable">  
Here is a {{localVariable}} in your outer component!
```

Task #03

Create a title @Input

Outputs

Output-Metadata

- **@Output** decorator declares an output property.

```
export class AuctionListComponent {  
  @Output() ping;  
}
```

- usage in template

```
<app-auction-list (ping)="...">
```

Output-Metadata

- **@Output decorator declares an output property.**

```
export class AuctionListComponent {  
  @Output('pong') ping; // this.ping available not this.pong  
}
```

- **usage in template**

```
<app-auction-list (pong)="handlePong($event)">
```

Output-Metadata

usage of output combined with events

```
@Component({  
  selector: 'app-auction-list',  
})  
export class AuctionlistComponent {  
  
  @Output() ping = new EventEmitter<string>();  
  
  sendPing() {  
    this.ping.emit('Msg');  
  }  
}
```

declare ping as
EventEmitter

emit an Event

Output - Generics

- `new EventEmitter<string>()`
creates an EventEmitter of type string
- The emitted value has to be a string:
`this.ping.emit('Msg');`
- `$event` contains the emitted event data
→ **it has not to be the event itself!**

Task #04

Create a (titleClicked) @Output

Structure Syntax

Structure Syntax * and <template>

```
<div *ngIf="auction">  
  <span>{{auction.title}}</span>  
</div>
```

- Structure directives begin with an asterisk (*)
- Syntactic sugar→easier to read/write
- Short fort for template elements

Structure Syntax - * and <template>

- Convenient way to skip the <template>wrapper tags
- Angular expand the * to template tags

```
<div *ngIf="auction">  
  <span>{{auction.title}}</span>  
</div>
```



```
<template [ngIf]="auction">  
  <div>  
    <span>{{auction.title}}</span>  
  </div>  
</template>
```

Structure Syntax - * and <template>

- ngFor is a bit more complex

```
<div
  class="row"
  *ngFor="let auction of auctions">
  <div class="col-xs-12">
    <a>{{auction.title}}</a>
  </div>
</div>
```

```
<template
  ngFor
  let-auction
  [ngForOf]="auctions">
  <div class="row">
    <div class="col-xs-12">
      <a>{{auction.title}}</a>
    </div>
  </div>
</template>
```

Task #05

Use *ngFor

Other Structure Directives

- **ngIf**: adds and removes elements in the DOM based on the results of an expression.
- **ngSwitch**: displays one element (and its children) from a set of possible options, based on some condition.
- **ngFor**: is a repeater directive which outputs a list of elements by iterating over an array.
- **ngClass**: adds and removes CSS classes on an element.
- **ngStyle**: sets CSS styles on an HTML element conditionally.

Interfaces

Interfaces

- Type-checking of the shape of values
- Interfaces give a type to these shapes

Interfaces – Without an interface

You can generate interfaces on the fly.

```
const auction: { id: number, startingPrice: number };
```

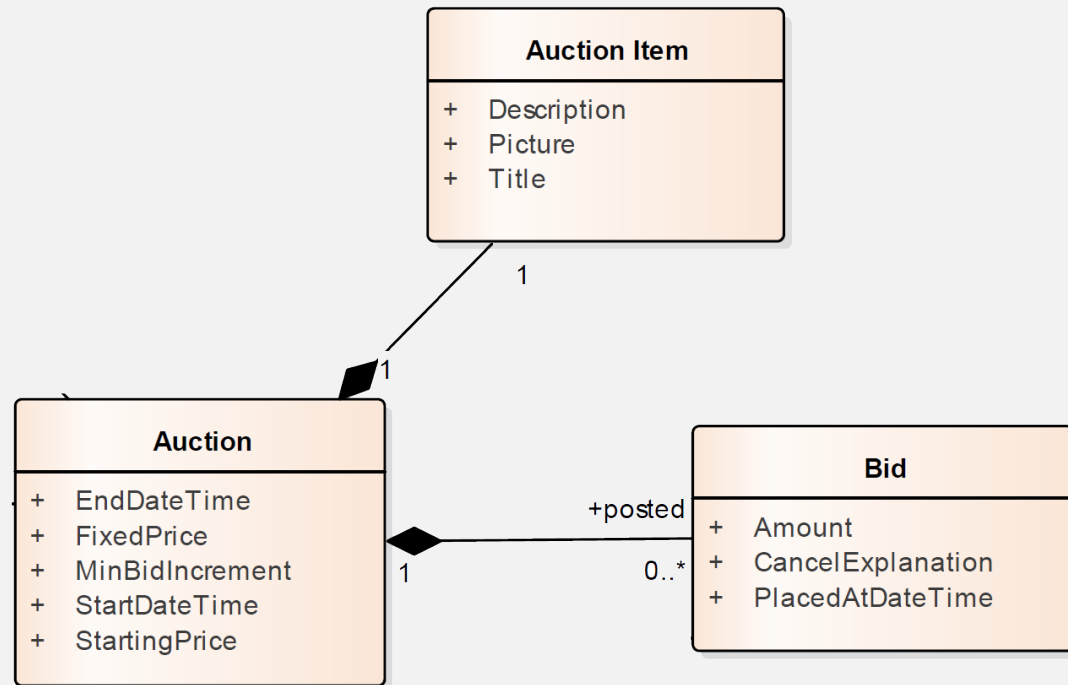
```
auction = {  
  id: '1',  
  startingPrice: '1000'  
};
```

Interfaces – With interface

Give an interface a name and use it as Type for variables.

```
interface Bid {  
  amout: number;  
  cancelExplanation: string;  
  placedAtDateTime: Date;  
}
```

```
interface AuctionItem {  
  description: string;  
  picture: string;  
  title: string;  
}
```



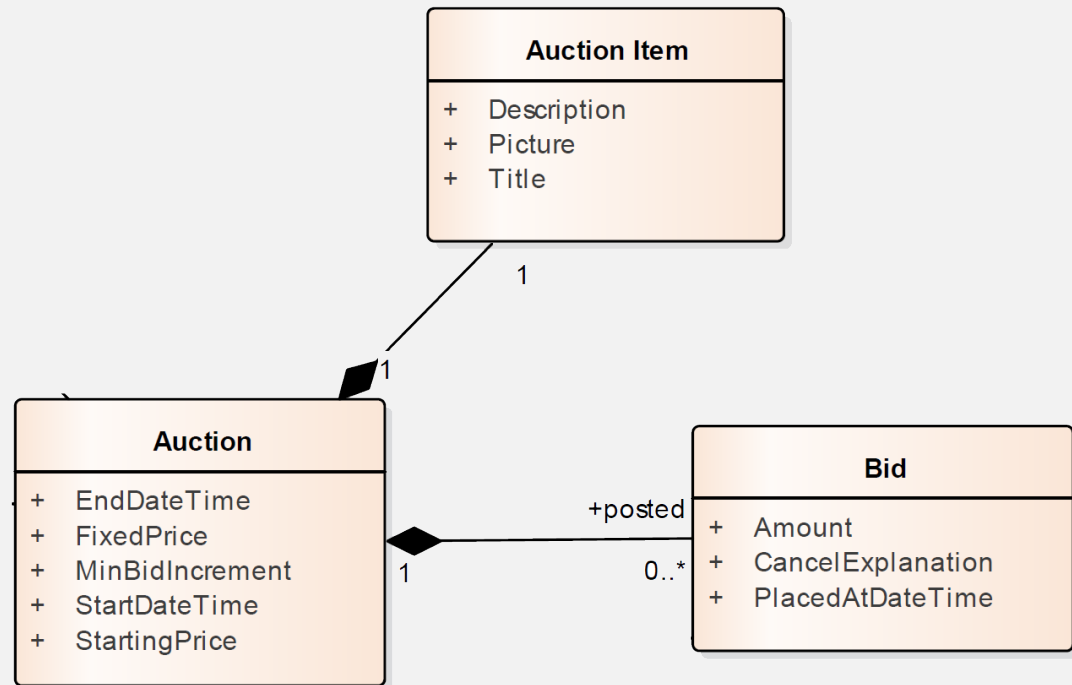
Interfaces – With interface

Give an interface a name and use it as Type for variables.

```
interface Auction {  
  id: number;  
  startingPrice: number;  
  ...  
  ...  
}
```

```
const auction: Auction;
```

```
auction = {  
  id: '1',  
  startingPrice: '1000'  
  ...  
};
```



Interfaces – Optional properties

If properties are optional in an interface, use a question mark.

```
interface Auction {  
  id: number;  
  startingPrice: number;  
  fixedPrice?: number;  
  ...  
}
```

Interfaces – Class types

Forgetting to implement ngOnInit throws a compile error.

```
interface OnInit {  
    ngOnInit();  
}  
  
class AuctionListComponent implements OnInit {  
    ngOnInit() {  
    }  
}
```

Task #06

Add an Auction Interface

Services

Services

- “Local Singletons”
- Keeps functionality of our application
- Could be injected via Dependency Injection(DI)
- Two roles:
 - Provide methods or streams of data to subscribe at
 - Provide operations to modify data

Service - Example

```
@Injectable()
export class AuctionDataService {
  private auctions = [{...}, {...}, {...}];

  constructor() {}

  getAuctions() {
    return this.auctions;
  }
}
```

Services

Create a services instance for a module

```
@NgModule({  
  providers: [  
    AuctionDataService  
  ]  
})  
  
@Component({ ... })  
export AuctionListComponent {  
  constructor(private auctionDataService: AuctionDataService) {}  
}
```

Services

Create a service instance for a component and its children

```
@Component({  
  ...  
  providers: [AuctionDataService]  
})  
export class AuctionListComponent {  
  constructor(private auctionData: AuctionDataService) {}  
}
```

Task #07

Create an AuctionData service

Dependency Injection

Dependency Injection - Why

- Keep component classes clean
- Better testable code
- Easy replacement of services

Without Dependency Injection

Dependency Injection

You have to create instances on your own.

```
@Component({ ... })  
class AuctionListComponent {  
  private auctionDataService;  
  constructor() {  
    this.auctionDataService = new AuctionDataService();  
  }  
}
```

With Dependency Injection

Dependency Injection

Dependency Injection is also called **Inversion of control**.

The injector has control over service instantiation.

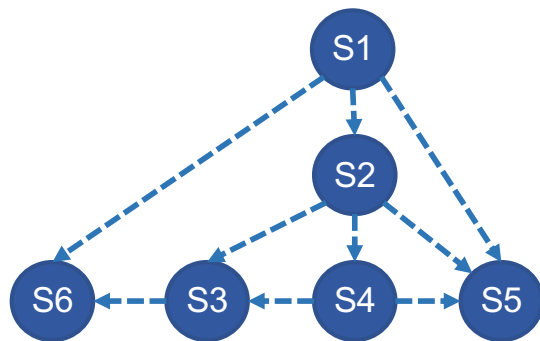
Dependency Injection

Injector is responsible for creating instances.

```
@Component({  
  providers: [AuctionDataService]  
})  
class AuctionListComponent {  
  constructor(private auctionDataService: AuctionDataService) {}  
}
```

Dependency Injection

- **Services can have dependencies, too**
- **Injects service instances created in a component, where service is used!**
- **Watch out for dependency cycles!**



```
service 'S1', (S2, S5, S6)
service 'S2', (S3, S4, S5)
service 'S3', (S6)
service 'S4', (S3, S5)
service 'S5', ()
service 'S6', ()
```

Dependency Injection

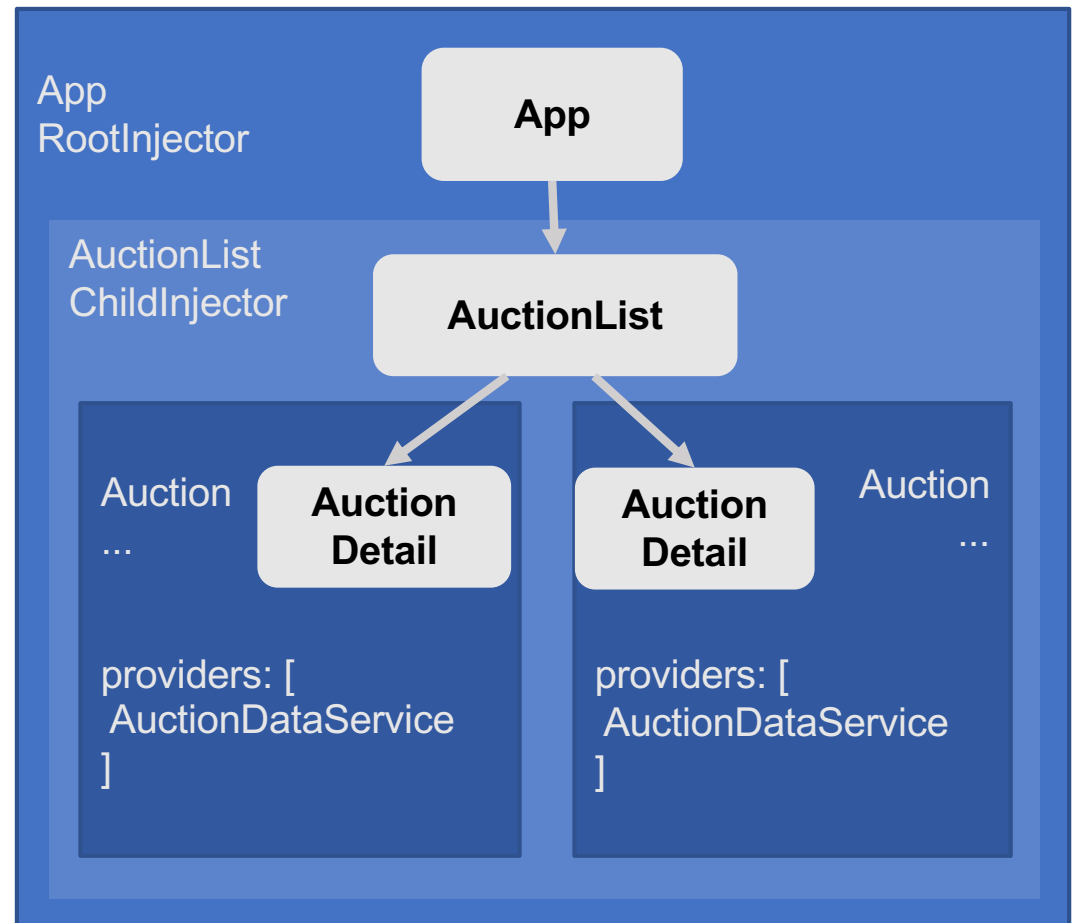
- **An instance is available for all child components, too
(with @Input binding)**
- **<https://angular.io/guide/component-interaction>**

Dependency Injection

- based on the type of class
- only **services** can be injected
- <https://angular.io/guide/architecture-services>

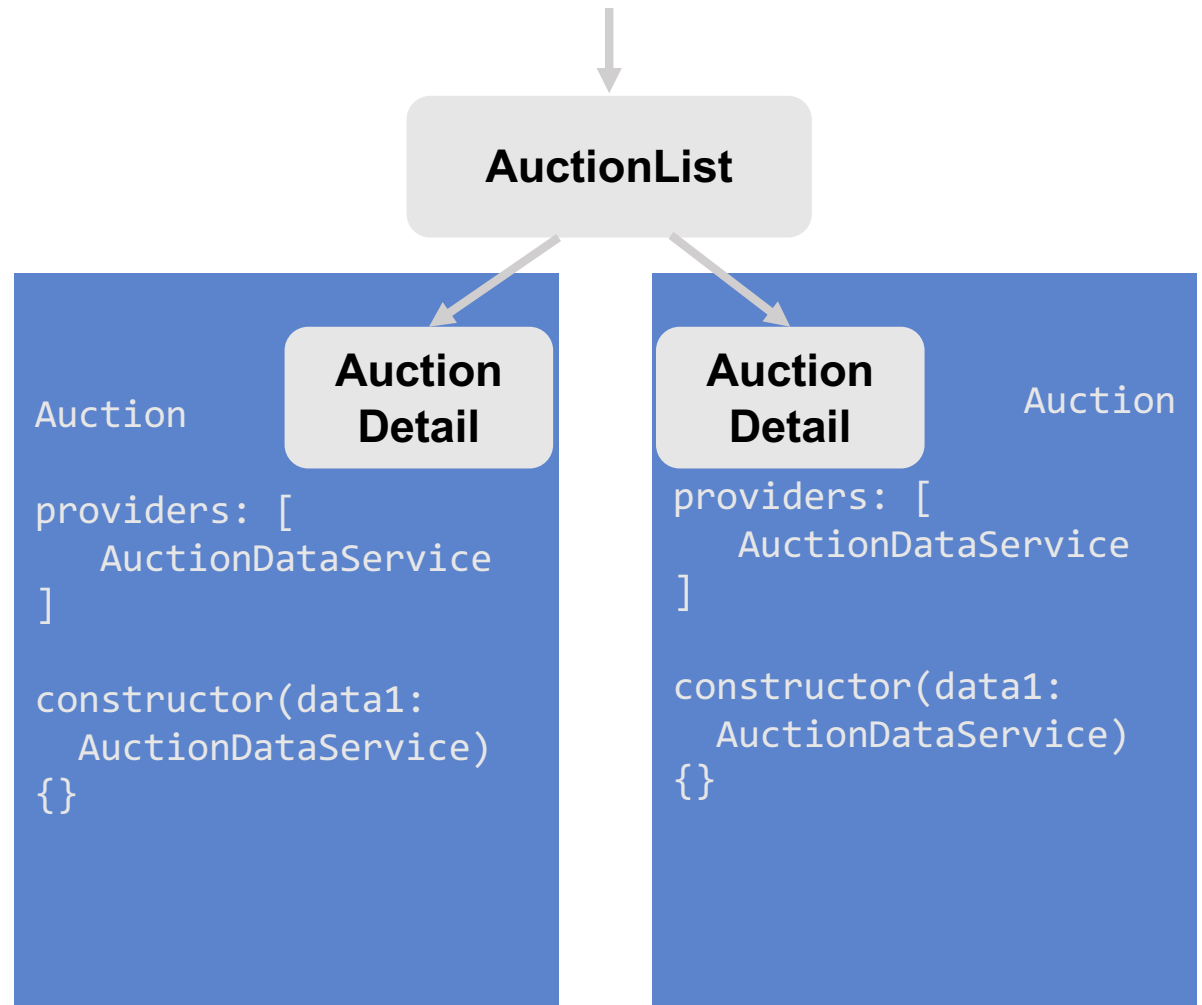
Dependency Injection

- **Injector** per component
- Each component has an own injector
- Base injector=**RootInjector**
- Each nested component has a **ChildInjector**



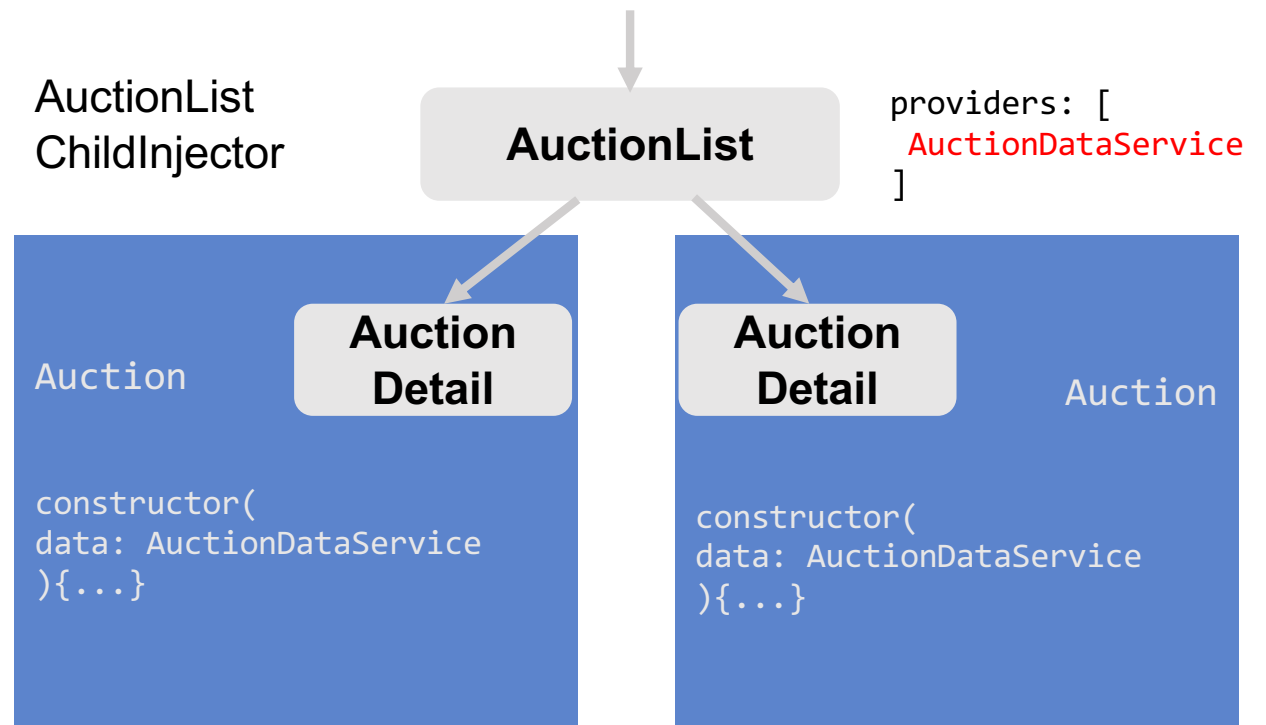
Dependency Injection

- New service instance for each AuctionComponent



Dependency Injection

- Share one service instance
- Create instance in parent component AuctionList
- Only inject service in AuctionDetail
→ **no providers!**



Dependency Injection - @Injectable()

- Annotation at classes that use DI
- NOT to make the class injectable itself
- Metadata to compile the type-information to the ES5 code
- Without an annotation you loose the type information

Dependency Injection

Possibility to register another implementation for an interface or service in the provider property of a decorator (e.g. @NgModule, @Component)

- `{provide: AuctionDataService, useClass: WebSockedAuctionData}`
- `{provide: AuctionDataService, useClass: HttpAuctionData}`
- `{provide: AuctionDataService, useClass: MockAuctionData}`

Look and Feel

Bootstrap and Font Awesome

Bootstrap

- [Bootstrap](#) is a popular CSS framework
- Integration in ng-cli is documented at the Wiki Page:
 - <https://github.com/angular/angular-cli/wiki/stories-include-bootstrap>
- Integration Bootstrap version 4.0 with Saas:
 - `$ ng set defaults.styleExt scss`
 - `$ npm install bootstrap --save`
 - empty file `_variables.scss` in `src/`.
 - `$icon-font-path: '../node_modules/bootstrap-sass/assets/fonts/bootstrap/';`
 - rename `styles.css` in `styles.scss`
 - In `styles.scss` add the following:
 - `@import 'variables';`
 - `@import '../node_modules/bootstrap/scss/bootstrap';`
 - Refactor all `.css` files in `.scss` files
 - Rename `styles.css` in `styles.scss` in file `angular-cli.json`

Testing Bootstrap Integration

Add add a bootstrap markup:

```
<button class="btn btn-primary">Test Button</button>
```

To ensure variables are working try to put something in file
_variables.scss

```
$primary: red;
```

Font Awesome

- [Font Awesome](#) gives you scalable vector icons that can instantly be customized
- Integration in ng-cli is documented:
 - <https://github.com/angular/angular-cli/wiki/stories-include-font-awesome>
- Install the font-awesome library and add the dependency to package.json

```
npm install --save font-awesome
```

- To add Font Awesome CSS icons to your app...

```
// in angular.json
"styles": [
  "styles.css",
  "../node_modules/font-awesome/scss/font-awesome.scss"
]
```


Testing Font Awesome

Add the following code to app.component.html

```
<h1>
  {{title}} <i class="fa fa-check"></i>
</h1>
```

app works! ✓

Test Button

auction-list works!

this is an auction list (from variable)

Yamaha YZF R1 Sports Bike

Yamaha FZ V2.0 FI

Yamaha SZ RR V 2.0

Task #08

Integrate Bootstrap and Font Awesome

Auction-List-Detail Component




- Shows all necessary information of an auction within the Auction-List

app works! ✓

Test Button

auction-list works!

this is an auction list (from variable)

	Yamaha YZF R1 Sports Bike Ready to take a "walk" on the wild side	3 bids CHF 4300.00 📅 30.3.2017, 23:35:03
	Yamaha FZ V2.0 FI Nice for a city ride	3 bids CHF 2400.00 buy now CHF 4000.00 📅 29.3.2017, 23:35:03
	Yamaha SZ RR V 2.0 The stylish bike with a nice red color	3 bids CHF 1200.00 buy now CHF 2500.00 📅 31.3.2017, 23:35:03

Task #09

Create an Auction-List-Detail
Component

Observables

Callbacks 3.0

Why we're talking about it?

Angular7 is using **RxJS 6** Observables for **async**.

For migrations from RxJS 5... there is a compat library available

<https://xgrommx.github.io/rx-book/index.html>

What is RxJS

- seeing events as collections you can
 - map
 - filter
 - ...

<https://www.youtube.com/watch?v=Rr0tMbmeid8>

Promises vs. Observables

Observables are built
to solve problems around **async**.
(avoid “callback hell”)

Observables

- streams
- any number of things
- Lazy→Only generate values when subscribed to (**cold**)
- can be “unsubscribed”→can be canceled

Promises vs. Observables

	Callbacks	Promises	Observables
Single value	+	+	+
Multiple values	+	-	+
Composable	-	+	+
Lazy	-	-	+
Cancelable	-	-	+

Observables

- `subscribe(successFn, errorFn)`

```
import {Observable} from 'rxjs';  
import {Subscription} from 'rxjs';
```

Observables - completeFn

- `subscribe(successFn, errorFn, completeFn)`

Observables - cancellation

```
let subscription = observable.subscribe(...)  
subscription.unsubscribe()
```

Creating observables (single value)

```
let observable = new Observable(observer => {
  randomAsyncCall((err, value) =>
    if (err) {
      observer.error(err);
    } else {
      observer.next(value);
      observer.complete();
    }
  });
});
```

Creating observables (cancellation)

```
let observable = new Observable(observer => {
  const deregisterFn = randomAsyncCall((err, value) =>
    if (err) {
      observer.error(err);
    } else {
      observer.next(value);
      observer.complete();
    }
  });
  return deregisterFn
});
```

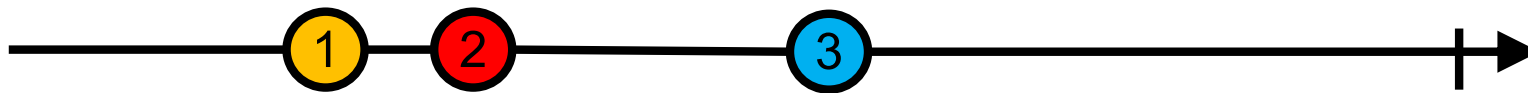
BUT

**You are usually not creating
your own observables!**

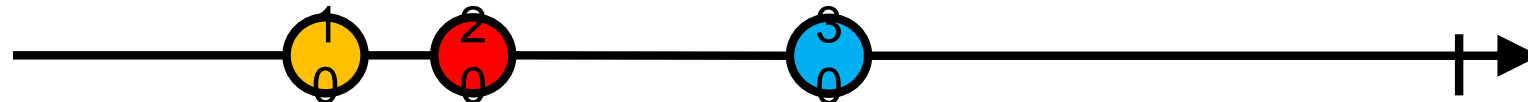
Observables creation helpers

- `Observable.pipe(of(value1, value2,...))`
- `Observable.pipe(from(promise/iterable/observable))`
- `Observable.pipe(fromEvent(item, eventName))`
- Angular7 `httpClient`
- Many more

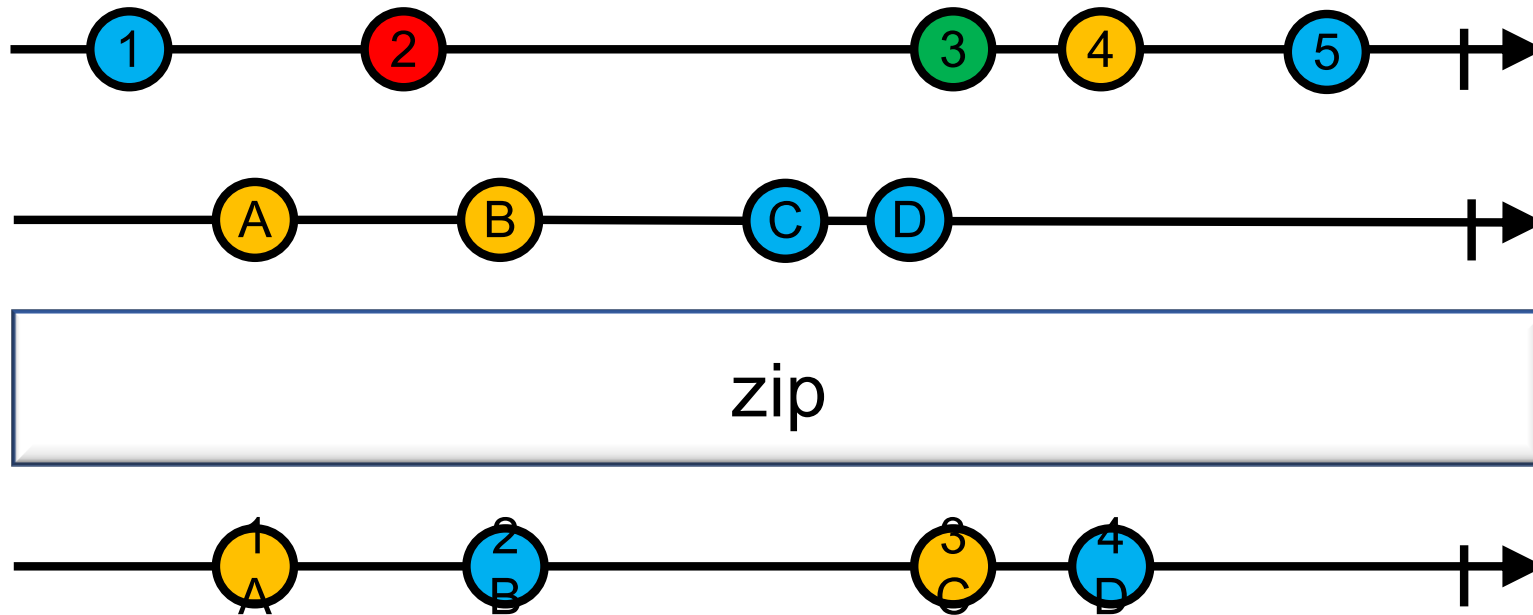
.map()



map (x => 10 * x)



.zip()



Observables - Generics

- Functions should return typed data
- Extend type informations of observables with generics
- E.g. `getAuctions(): Observable<Auction[]> {}`

Task #10

Create an Observable

Component Lifecycle Hooks

Component Lifecycle Hooks

- Components and Directives have a lifecycle managed by Angular
- Visibility of key moments and way to act when they occur
- Classes can implement one or more interfaces with hooks

<https://angular.io/docs/ts/latest/guide/lifecycle-hooks.html>

Component Lifecycle Hooks - Interfaces

```
import {Component, OnInit} from '@angular/core'
```

```
@Component({...})
```

```
class AuctionListComponent implements OnInit
```

```
  ngOnInit(): void {  
    console.log('onInit');
```


```
  }
```

```
}
```

Lifecycle interfaces are optional but recommended

A straight arrow points from the `implements OnInit` text in the code to the callout box.

Each Lifecycle hook has an interface without the leading ng

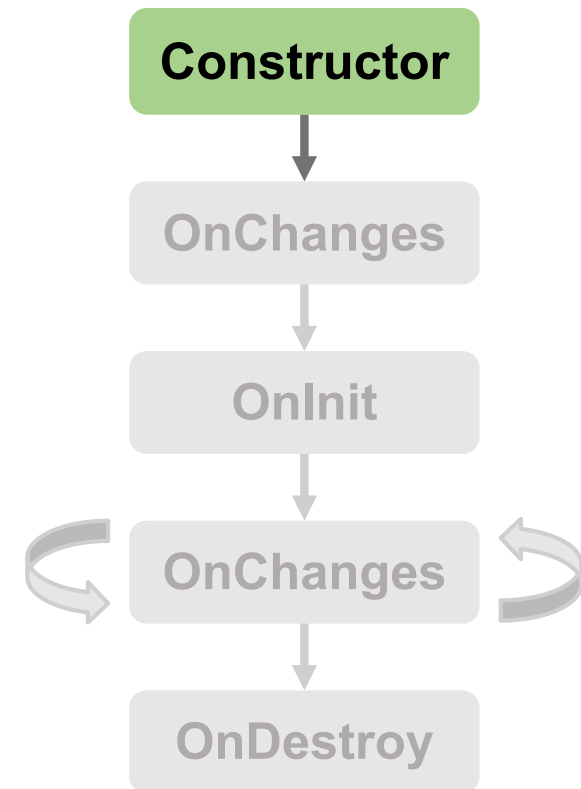
A curved arrow points from the `ngOnInit()` text in the code to the callout box.

Most important hooks

Component Lifecycle Hooks - Execution

- Injector instantiates component with **new**

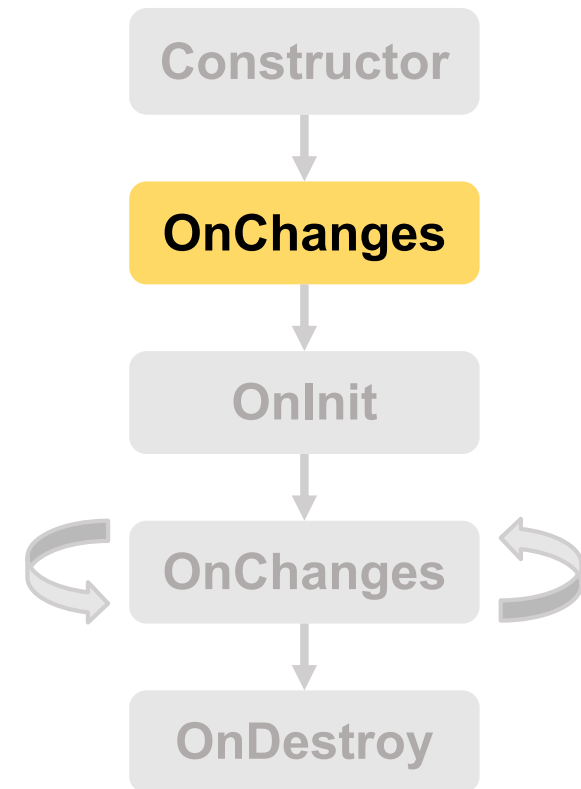
```
@Component({  
  selector: 'my-component',  
  ...  
})  
class MyComponent {  
  constructor () {}  
  ...  
}
```



Component Lifecycle Hooks - Execution

- Check for initial values on @Inputs

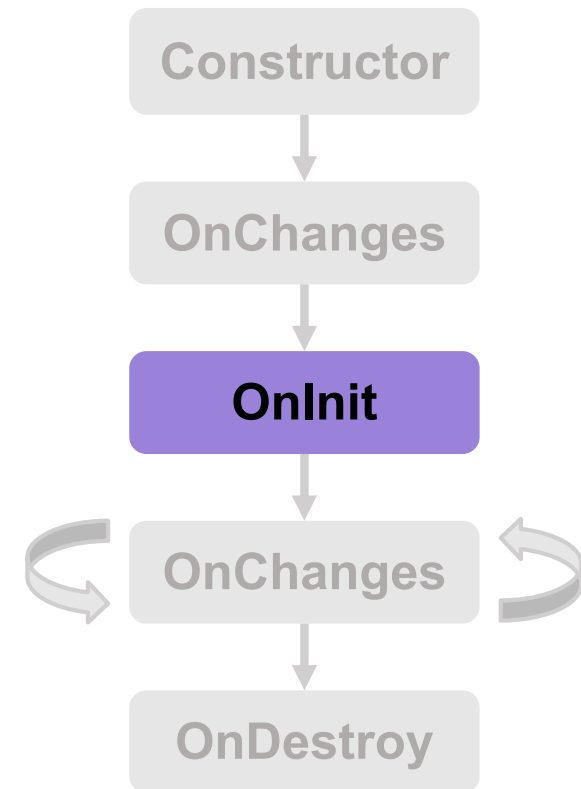
```
<my-component [anInput]="value">
</my-component>
...
class MyComponent {
  @Input() anInput;
  ...
}
```



Component Lifecycle Hooks - Execution

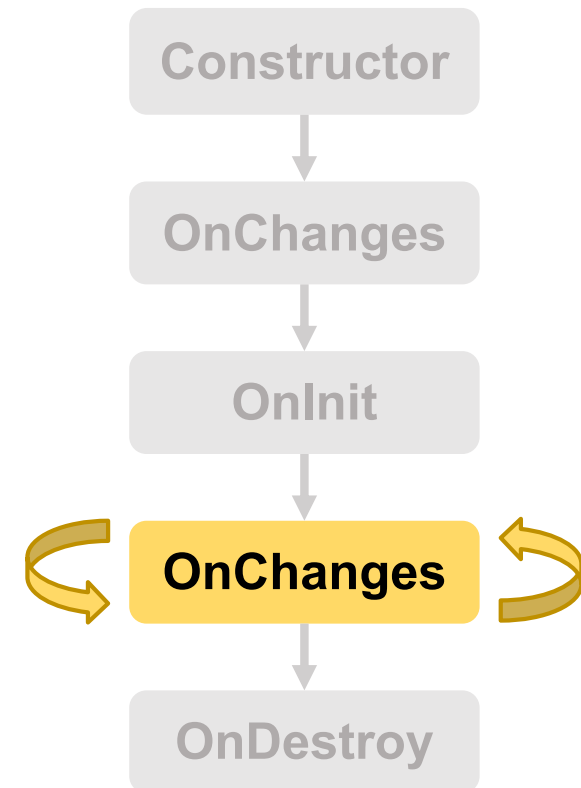
- Initial values are set→called only once
- For heavy or async work

```
class MyComponent {  
  @Input() anInput;  
  
  constructor() { // this.anInput = undefined  
}  
  
  ngOnInit() { // this.anInput is set }  
}
```



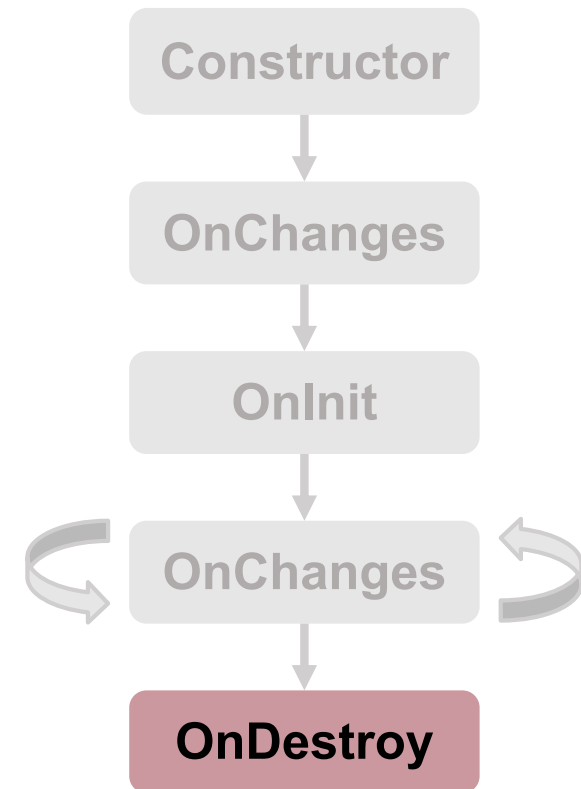
Component Lifecycle Hooks - Execution

- Every time an event occurs
- Change detection runs



Component Lifecycle Hooks - Execution

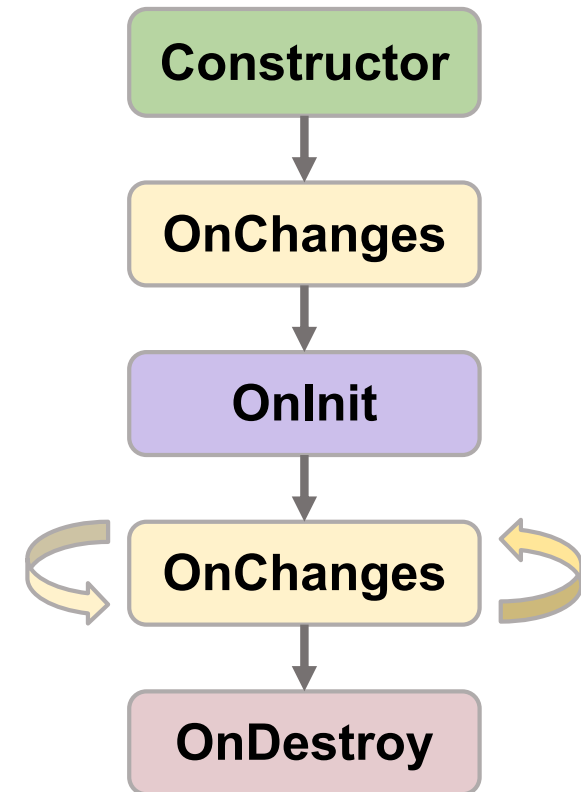
- **Cleanup before component is removed**
 - Remove event listeners
 - Remove observable subscribers
 - Clean up intervals and timeout
 - Inform other program parts
- **Called only once**



Component Lifecycle Hooks - Execution

Simplified

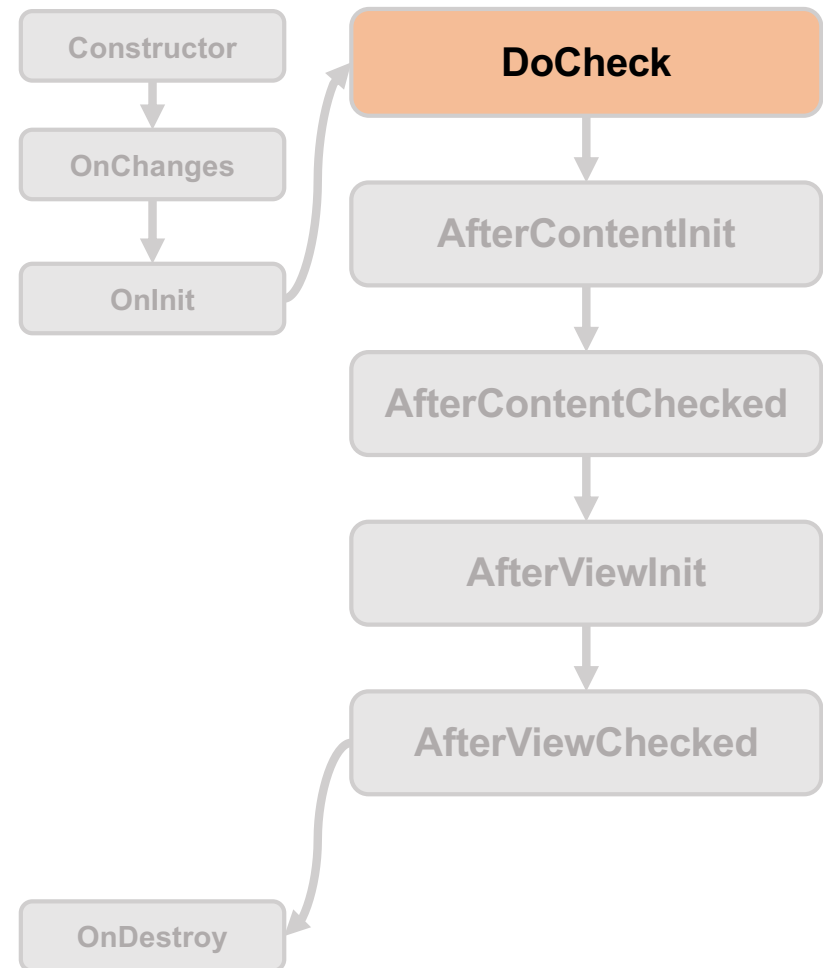
1. Component is instantiated
2. OnChanges: initial @Input values
3. OnInit: once after first OnChanges
4. OnChanges: get changed @Input
5. OnDestroy: component is destroyed



After component
creation

Component Lifecycle Hooks - Execution

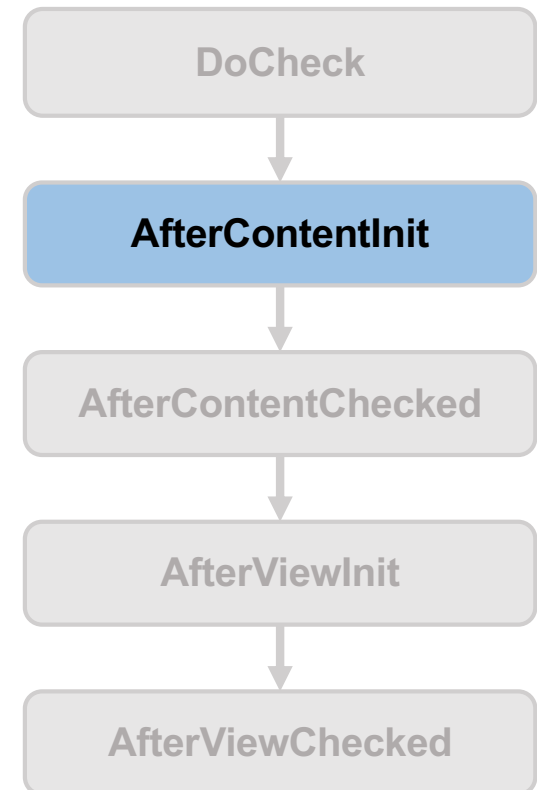
- Every time change detection runs
- Custom change detection function



Component Lifecycle Hooks - Execution

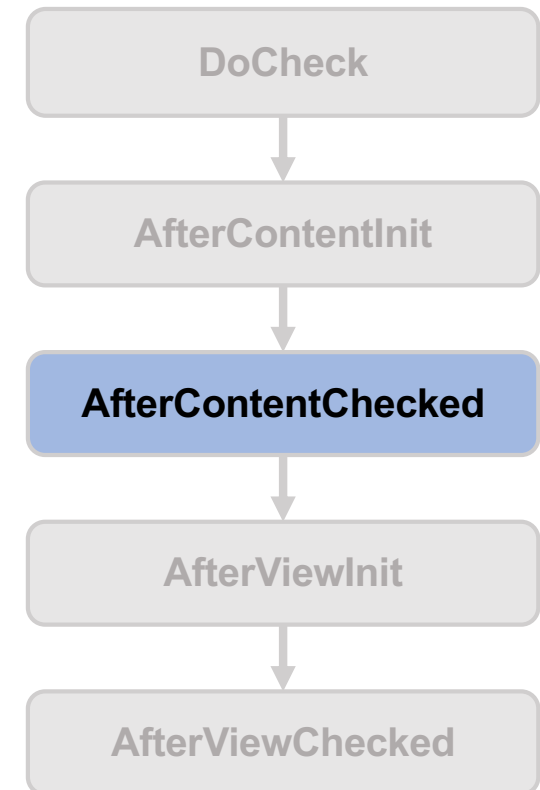
- Content=everything between component tags
- ngContent projects content to view after creation
- Hook called after projection finished→only once

```
@Component({  
  selector: 'my-component',  
  template: '<ng-content></ng-content>'  
})  
...  
<my-component><p>Hello</p></my-component>
```



Component Lifecycle Hooks - Execution

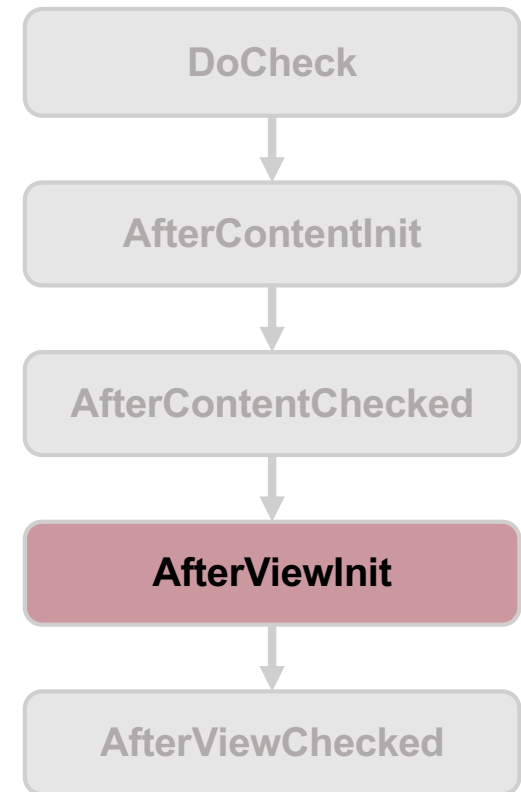
- After change detection→content is checked
- Called every time after DoCheck hook
- Initial check after content is initialised



Component Lifecycle Hooks - Execution

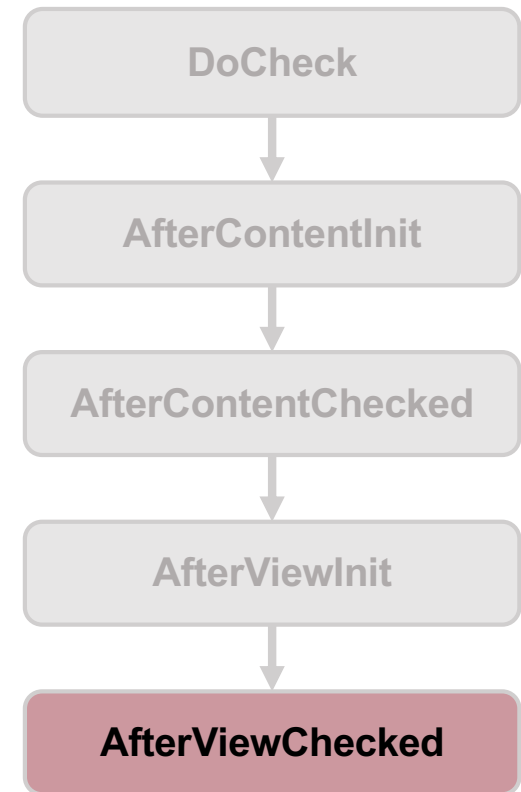
- View=template+bindings
- Called after view and subviews are initialised
→ only once

```
@Component({  
  selector: 'my-component',  
  template: `  
    <h1>Hello</h1>  
    <another-component></another-component>  
  `,  
})
```



Component Lifecycle Hooks - Execution

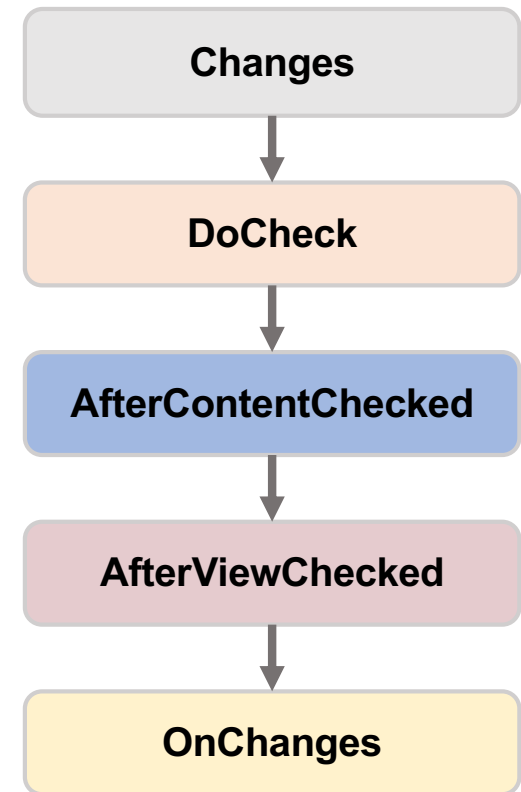
- After change detection→view is checked
- Called every time after DoCheck hook and after AfterContentChecked
- Initial call after the view is initialised



After change
detection

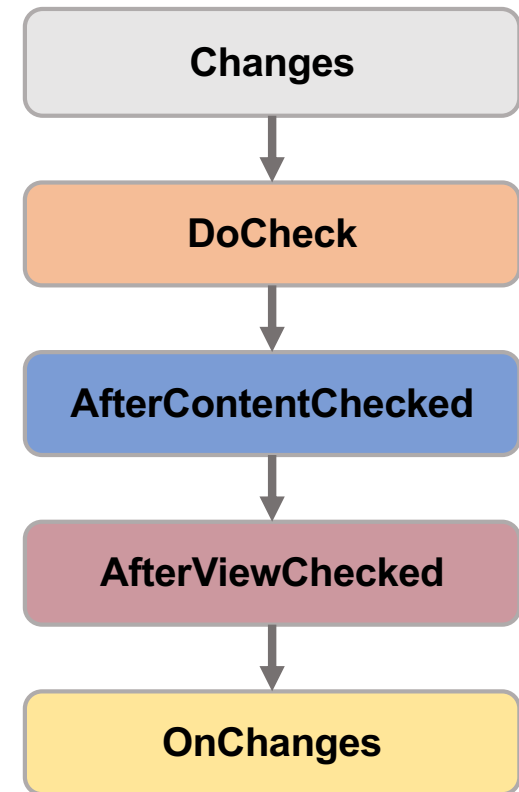
Component Lifecycle Hooks - Execution

- After a change
- Change detection calls custom DoCheck function
- Content and view are checked
- Inform about possible changes



Component Lifecycle Hooks - Execution

- After the *check* hooks are called once
 - change detection runs again to check for unexpected changes
 - triggers the *check* hooks again!
- In Angular notices changes after first change detection run
 - error in JavaScript console
 - (not in production mode)



Task #11

Component LifeCycle Basic

Http-Service

Using the HTTP-Service

- Basic HTTP handling
- `import {HttpClientModule} from '@angular/common/http'`
- `import {HttpClient} from '@angular/common/http'`
- Provide methods for
 - GET
 - PUT
 - POST
 - DELETE

Http service

HttpModule has to be imported into the app module

```
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  declarations: [AppComponent, AuctionListComponent],
  imports: [
    BrowserModule,
    HttpClientModule
  ],
  bootstrap: [AppComponent]
})
```

HTTP Service Interface

Name	Parameter	Returnvalue
get	url, options?	Observable<Response>
post	url, body, options?	Observable<Response>
put	url, body, options?	Observable<Response>
delete	url, options?	Observable<Response>
patch	url, body, options?	Observable<Response>
head	url, options?	Observable<Response>
request	Request, options?	Observable<Response>

Http service

Http service functions returning response observables

```
import { HttpClient, Response } from '@angular/common/http';  
...  
constructor(private http: HttpClient) {}  
  
getAuctions():Observable<Response> {  
    return this.http.get(this.baseUrl)  
}  
...
```

Http service

Use additional observable functions to work with the response object

```
import { Observable } from 'rxjs';
import { HttpClient, Response } from '@angular/common/http';
import { map, take } from 'rxjs/operators'
...
constructor(private http: HttpClient) {}

getAuctions():Observable<Auction[]> {
  return this.http
    .get(this.baseUrl)
    .map(response => response.json())
}
...
```

If you need additional observable functionalities→add them manually

Observables can stream multiple information→Map transforms them (not for Date...) -> see task#12

Http service

- Easy to type the result of an api request to a custom interface or class
- getAuctions function should return an observable with a list of auctions
- `getAuctions():Observable<Auction[]> {...}`

Http service

Subscribe to service observable in a component

```
constructor(private auctionDataService: AuctionDataService) {  
    this.auctionData  
        .getObservableAuctions()  
        .subscribe(data => this.auctions = data);  
}
```

Task #12

Load data from local API

Routing

No Single Page Application without routing

Why routing

- A website contains multiple pages to present different content
- Angular 7 apps are single page applications
 - Only one html page
- Routing is the way to navigate through an app without leaving the page

Basic routing

Basic Routing

- Based on browser location and history
- Map of url to content
- Special package: @angular/router

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

Basic Routing

- Define routes per module
- An extra file for route configuration: **moduleName.routing.ts**

app.routing.ts

Basic Routing

Define and register routes

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

// Define routes
export const appRoutes: Routes = [{ ... }, { ... }];

// Create a new module with configured router
export const routing = RouterModule.forRoot(appRoutes);
```


App Routing Module

Define and register routes

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

const routes: Routes = [];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Basic Routing

- Module can have sub modules
- Sub modules can have its own routings
- Extend your existing routing with
`RouterModule.forChild(childRoutes)`

Task #13

Add routing file

Basic Routing

Add routing module to your app imports

```
import { routing } from './app.routing';
```

```
@NgModule({  
  imports: [  
    AppRoutingModule,  
    routing  
  ]  
})
```

Basic Routing

Defining a route – without leading ‘/’!

```
export const appRoutes: Routes = [{  
  path: 'auctions',  
  component: AuctionListComponent  
}];
```

Task #14

Add a route

Basic Routing

- No connection between DOM and route, yet
- Router needs to know where he should append the component
- Special component: **routerOutlet**

Basic Routing

Routing components are available through the routing import

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


Basic Routing

1. Url in browser matches against route path
2. Information of connected route are evaluated
3. Information are used to show correct component in `routerOutlet`

Basic Routing

 localhost:4200/auctions

`Routes = [{ url: 'auctions', component: AuctionListComponent }];`

`<router-outlet><app-auction-list>...</app-auction-list></router-outlet>`

Basic Routing

Redirect to default router

```
export const appRoutes: Routes = [{  
  path: '',  
  redirectTo: '/auctions',  
  pathMatch: 'full' // checks if full url matches path  
}, {  
  ...  
}];
```

Task #15

Add routerOutlet

and

Set /auctions as default route

Routing with parameters

Routing with parameters

- You need dynamic routes very often, e.g. Detail-Views
- Content of a component is configurable
- You need additional data in your component

Routing with parameters

Add parameter placeholders with a leading `:`

```
// app/app.routing.ts
const routes: Routes = [
  { path: 'auctions/:id', component: AuctionDetailComponent }
];
```

routerLink

RouterLink with params example

```
<a [routerLink]=" ['/auctions', 1] ">  
+  
{ path: 'auctions/:id', component: AuctionDetailComponent }  
=  
<a href="/auctions/1">
```

Navigate with router

Navigate after click event

```
<div class="container top" (click)="onContainerClick()" >  
  
</div>
```

```
import {Router} from '@angular/router';  
  
constructor(private router: Router) { }  
  
onContainerClick() {  
  this.router.navigate(['/auctions/' + this.auction.id]);  
}
```

Task #16

Add an AuctionDetail Route

Retrieve route
params in a
component class

Route params

Inject ActivatedRoute service and subscribe params observable.

```
@Component(...)
export class AuctionDetailComponent implements OnInit {
  constructor(
    private route: ActivatedRoute
  ) {}

  ngOnInit () {
    this.route
      .params
      .subscribe((params) => ...);
  }
}
```

Why an observable?

Route params

- Angular 7 has some caching mechanisms
- Current component and components on the same level in the tree are cached for faster navigation
- Components are not instantiated again
- But parameters could have changed, e.g. paging

AuctionDetail Example

Subscription of Observable from `getHttpAuction`, display container after auction became available **`*ngIf="auction"`**

```
<div *ngIf="auction" class="container top" (click)="onContainerClick()" >
...
</div>
```

```
ngOnInit() {
  this.routeSubscription = this.route.params.subscribe(params => {

    this.subscription =
      this.auctionDataService.getHttpAuction(params['id']).subscribe(
        auction => {
          this.auction = auction;
          // console.log('auction: ' + auction);
        },
        err => console.log(err)
      );
  });
}
```

Task #17

Read and Use AuctionDetail
Parameters

Simple approach
with snapshots

Route params - Snapshots

- Snapshots are images of the current state
- **ActivatedRoute** gives access to the current router state
- Can be used if not future changes expected

Route params - Snapshots

The params of a route are stored in a snapshot object.

```
@Component(...)
export class AuctionDetailComponent implements OnInit {
  constructor(
    private route: ActivatedRoute
  ) {}

  ngOnInit () {
    const auctionId = this.route.snapshot.params['id'];
  }
}
```

Routing wildcard

Set a wildcard to handle all not defined routes

```
{  
path: '**',  
component: PageNotFoundComponent  
}
```

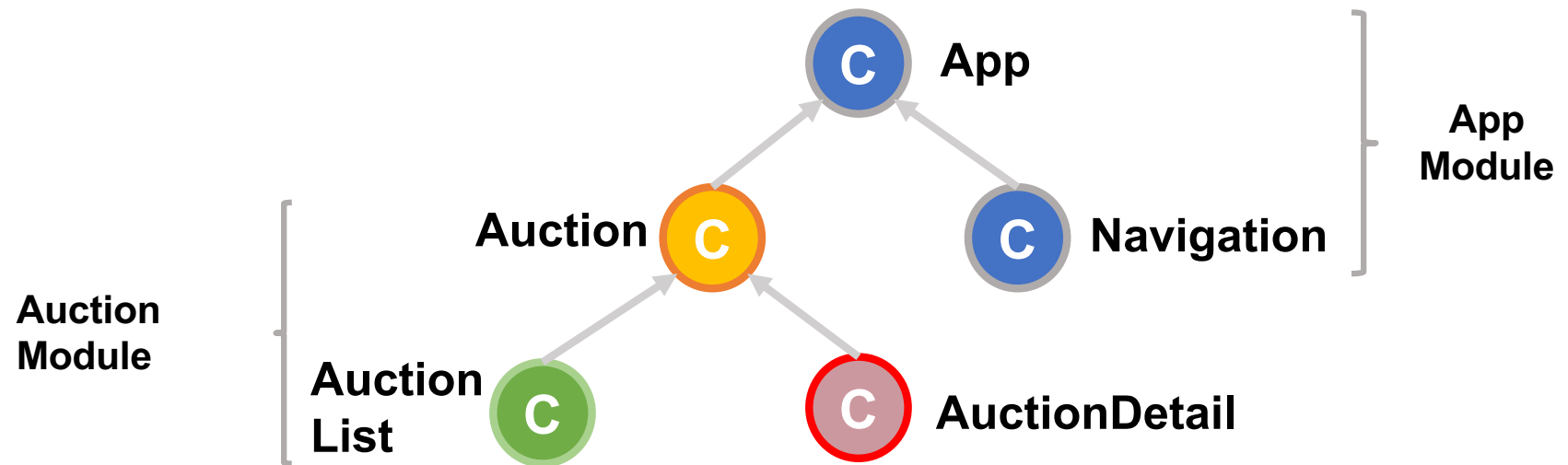
Modules

Nested & child routes

Nested routes

- A module can have sub modules with own components
- Each sub module can manage its own routes
- No need to change root routing

Think in Modules & Components



Nested routes

Routes of a auction module with a root auction component

```
export const auctionRoutes: Routes = [  
  {  
    path: 'auctions',  
    component: AuctionComponent  
  }  
];  
export const auctionRouting = RouterModule.forChild(auctionRoutes);
```

Nested routes

HTML with nested route – **auction.component.html** has its own **routerOutlet**

```
<app-root>
  <router-outlet>
    <auction>
      <router-outlet>
        ...
      </router-outlet>
    </auction>
  </router-outlet>
</app-root>
```

Child routes

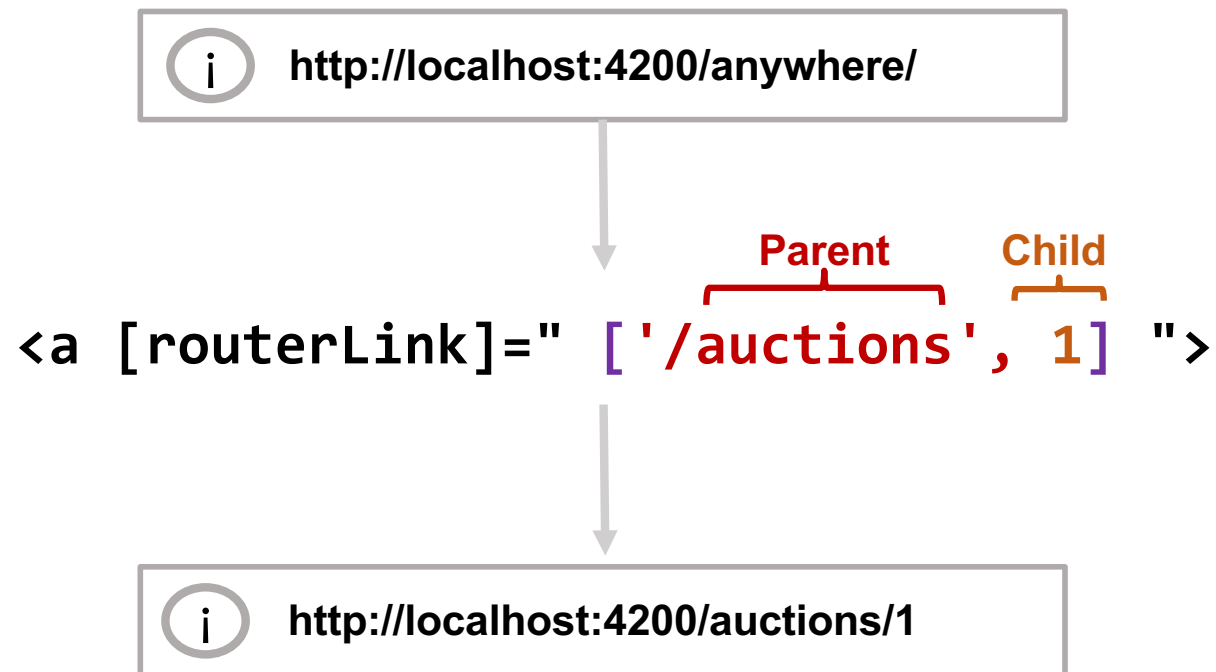
- A route can have children
- Each child gets its parent path as base path

Child routes

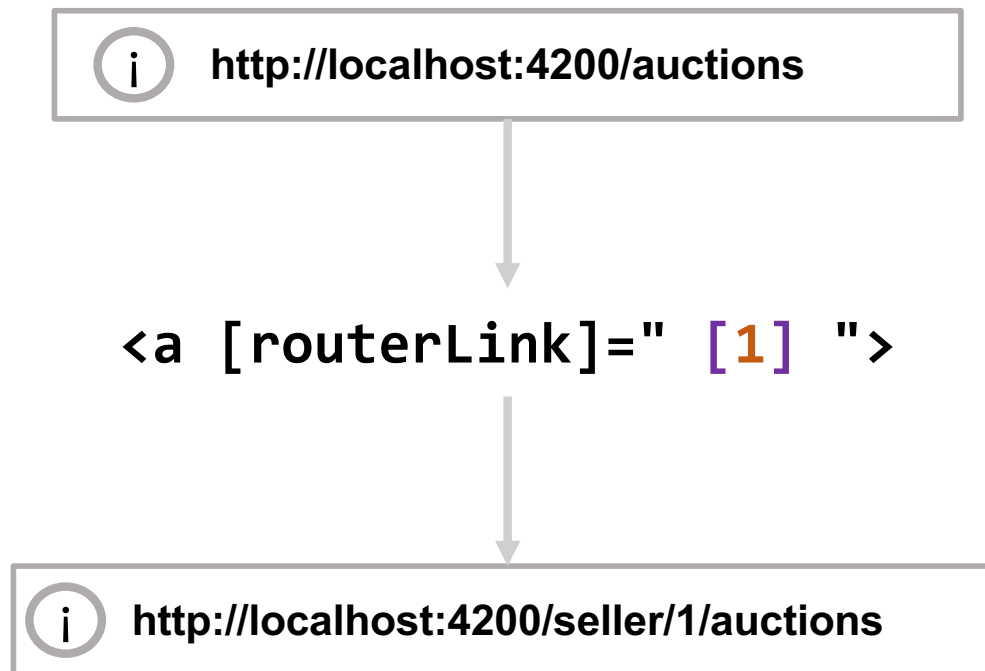
AuctionList and AuctionDetail as route children

```
{
  path: 'auctions',
  component: AuctionComponent,
  children: [{
    path: '',
    component: AuctionListComponent
  }, {
    path: ':id',
    component: AuctionDetailComponent
  }]
}
```

Child routes



Child routes – relative links



Task #20

Build a Layout with Navbar

Task #21

Add a wrapper module

Route Guards

Why guards?

Why guards?

- **You want to protect your routes against unwanted access**
- **Sometimes you may have restricted permissions**
 - User have to be signed in to see the content
- **Protect the user**
 - Notify him about unsaved changes, before leaving the route

Route Guards

- Interfaces a service which can implement or simple function
- Guard functions have to return a **boolean**, a boolean **promise** or a boolean **observable**
→ something to evaluate to true or false at the end
- Possibility to have **asynchronous** guard functions,
e.g. authorization check with an API

Route Guards

- 5 kinds of route guards
- Implement multiple guards in one service
- Guards are **route based** and not component based (like in previous versions)

Route Guards

canDeactivate

canActivate

canActivateChild

canLoad

resolve

Route Guards

canDeactivate

canActivate

canActivateChild

canLoad

resolve

- Are allowed to leave a route
- Check if the data persisted properly?
- Notifications about leaving the route

Route Guards

canDeactivate

canActivate

canActivateChild

canLoad

resolve

- Check if the route can be activated
- Before entering the route
- Check if the user is authenticated?
- Check if the user has access rights

Route Guards

canDeactivate

canActivate

canActivateChild

canLoad

resolve

- A route can have child routes
- Check if child routes can be activated
- If all children would have the same canActivate function, you can simply implement only one check

Route Guards

canDeactivate

canActivate

canActivateChild

canLoad

resolve

- Avoid to load the content of the route
- Connected component is not loaded like in other guards
- Template of the component is not requested

Route Guards

canDeactivate

canActivate

canActivateChild

canLoad

resolve

- Retrieve data before component is loaded
- Can handle any return value, e.g. Observables

Guards as functions

Guards as functions

- Simple way to create guards
- A function that has to return a boolean, boolean promise or boolean observable
- App needs to know how to access/inject the function as a guard
 - «provide» function allows us to connect a function with a string token

Guards as functions

Simple canActivate guard

```
@NgModule({  
  providers: [{  
    provide: 'CanActivateGuard',  
    useValue: () => {  
      return true;  
    }  
  ]  
  ...  
})
```

Guards as functions

1. need to connect a guard with a route
2. Simple add the guard name as property to a route object
3. Guard properties accept arrays of guard tokens (functions, services)

Guards as functions

Connect a guard function with a route

```
{  
  path: 'auctions',  
  component: AuctionListComponent,  
  canActivate: ['CanActivateGuard']  
}
```


Guards as a function

Problem:

- Our function name is used as a string token
→ see the quotes around the name
- This can lead to naming collisions with other modules!

Solution:

- Use classes for type tokens
- Token is connected with the class

Guards as classes

Guards as classes

- better way to create guards
- An Angular service
- A class that implements the guard interfaces
- App needs to know the guard
 - simply add your service to the providers array

Guards as classes

- Interfaces are exported by @angular/router
- Interface names:
 - canActivate hook → CanActivate interface
 - canDeactivate hook → CanDeactivate interface
 - ...

Guards as classes

- **Some interfaces have to be typed**
 - You might access to the component their information and current state
 - `class DeactivateGuard implements CanDeactivate<AuctionListComponent>`
- **Other not:**
 - `class ActivateGuard implements CanActivate`

Guards as classes

Simple guard service

```
import { Injectable } from '@angular/core';
import { ActivatedRouteSnapshot, CanActivate, RouterStateSnapshot } from
'@angular/router';

@Injectable()
export class CanActivateViaServiceGuard implements CanActivate {

  constructor() {}

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {
    return true;
  }
}
```

Guards as classes

Add guard service to a module

```
@NgModule({  
  providers: [  
    CanActivateViaServiceGuard,  
    ...  
  ],  
  ...  
})
```

Guard as classes

- Import the guard service in your route file
- Add guard service class to the array of guard tokens for a hook

Guard as functions

Connect a guard with a route

```
{  
  path: 'auctions',  
  component: AuctionListComponent,  
  canActivate: [CanActivateViaServiceGuard]  
}
```

Task #22

Build a simple canDeactivate guard

HTML5 Forms

Forms

- Main components of most business applications
- Needed in most apps, e.g. login and registration

Forms

A simple HTML5 form

```
<form>
  <label for="title">Title:</label>
  <input type="text" id="title" name="title">

  <button type="submit">Speichern</button>
</form>
```

Forms

A simple HTML form with validation

```
<form>
  <label for="title">Title:</label>
  <input type="text" id="title" name="title" required>

  <button type="submit">Speichern</button>
</form>
```

HTML forms are not
enough

Forms in Angular 7

Forms in Angular 7

- Possibilities of HTML5 forms are restricted
- We need to connect our form with our data model
- We have to update the data model on every change on the input elements

Forms in Angular 7

Two types:

1. Template-driven forms

- Created and configured only in the HTML code
- No default access via TypeScript

2. Model-driven forms

- Created and configured in the TypeScript code
- Connected to a form HTML code

Template-driven forms

Template-driven forms

- For building simple forms
- No need to get programmatically access to it
- Easiest way to create a form in Angular 7

Template-driven forms

Import FormsModule to get them working

```
import { FormsModule } from '@angular/forms';

@NgModule({
  imports: [FormsModule]
})
```

Template-driven forms

- FormsModule provides necessary directives
- **Directives:**
 - ngForm
 - ngModel

Template-driven forms - ngForm

- Automatically added to each form
- Angular representation of form is a FormGroup, which contains multiple FormControl (inputs)
- Possibility to get access to the form and input states in the template

Template-driven forms - ngForm

Get access to your form with ngForm

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

Stores the formGroup on a local
template variable

Template-driven forms – Data Binding

- **What to do?**
 - Load data into a form and update inputs
 - Update changes on form input elements
- **We need something like this:**

```
<input [value]="auction.auctionItem.title"  
      (input)="auction.auctionItem.title=$event.target.value">
```

Template-driven forms – Data Binding

- **What to do?**
 - Load data into a form and update inputs
 - Update changes on form input elements
- **We need something like this:**

```
<input [value]=" auction.auctionItem.title"  
      (input)=" auction.auctionItem.title=$event.target.value">
```

**That looks like boilerplate
code!**

Template-driven forms - ngModel

- **ngModel Directive**
 - Two-Way Data Binding
 - Reading and writing values from inputs
 - Creates a FormControl for the input element and registers it at the form

Template-driven forms - ngModel

First optimization

```
<input  
  [value]="auction.auctionItem.title"  
  (input)=" auction.auctionItem.title=$event.target.value">
```



```
<input  
  [ngModel]=" auction.auctionItem.title"  
  (ngModelChange)=" auction.auctionItem.title=$event">
```

[value]="auction.auctionItem.title"

EventEmitter property that returns
the input box value when it fires

Template-driven forms - ngModel

First optimization

```
<input  
  [value]="auction.auctionItem.title"  
  (input)="auction.auctionItem.title=$event.target.value">
```

*Also looks like too much
boilerplate code!*

```
<input  
  [ngModel]="auction.auctionItem.title"  
  (ngModelChange)="auction.auctionItem.title=$event">
```

[value]="auction.auctionItem.title"

EventEmitter property that returns
the input box value when it fires

Template-driven forms - ngModel

Second optimization

```
<input  
  [ngModel]=" auction.auctionItem.title"  
  (ngModelChange)=" auction.auctionItem.title=$event">
```

[value]="auction.auctionItem.title"

EventEmitter property that returns the input box value when it fires

Why does that exist?

```
<input [(ngModel)]=" auction.auctionItem.title">
```

When it's so easy?

Template-driven forms - ngModel

Get access to your input with ngModel

```
<form #form="ngForm">  
  <input [(ngModel)]="auction.startingPrice" name="startingPrice"  
                                                #startingPrice="ngModel">  
</form>
```

Stores the control of the model
on a local template variable

Template-driven forms - validation

- Let the user know what he is doing wrong
- Guide a user through your form
- Increases user experience

Template-driven forms - validation

- access to state of a form input
- easy visualisation of the control (in-)valid state

```
<form>  
  <input [(ngModel)]="auction.auctionItem.title" name="title">  
</form>
```

Template-driven forms - validation

- Example of possible input validation via **type**

- text
- email
- date
- number
- month
- url
- tel

```
<form>
  <input
    type="email"
    [(ngModel)]="user.email"
    name="email">
</form>
```

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

Template-driven forms - validation

- example of possible input validation via **attribute**
 - required
 - maxlength
 - minlength
 - pattern

```
<form>  
  <input  
    type="text" required  
    [(ngModel)]="auction.auctionItem.title"  
    name="title">  
</form>
```

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

Form Validation – ngModel CSS classes

State	Class if true	Class if false
Control has been visited	.ng-touched	.ng-untouched
Control value has changed	.ng-dirty	.ng-pristine
Control value is valid	.ng-valid	.ng-invalid

Task #23

Create an AuctionDetailEdit
Component

Template-driven forms - ngModelGroup

Can combine semantic groups of control, e.g. a in a fieldset tag


```
<form #form="ngForm">
  <fieldset ngModelGroup="name">
    <input type="text" name="first" ngModel>
    <input type="email" name="last" ngModel>
  </fieldset>
</form>

<pre>{{ form.value.name.first }}</pre>
```

Form Validation

ngForm and local variables

bind local variable
to Control Instance



```
<form #form="ngForm">
  <input
    type="text"
    [(ngModel)]= "auction.startingPrice"
    name=" startingPrice "
    #title="ngModel"
  >
</form>
```

```
{
  "value": {
    "title": ''
  },
  "controls": {
    ...
    "title": {...}
    ...
  },
  "dirty": true,
  "valid": false,
  "pristine": false,
  "touched": true
}
```

Form Validation - ngForm

Use state via local template variables to define validation messages

```
<form #form="ngForm">
  <input type="text" required [(ngModel)]="auction.startingPrice"
    name="startingPrice" #title="ngModel">

  <div [hidden]=" startingPrice.valid || startingPrice.pristine">
    Price not valid
  </div>
  <div [hidden]="! startingPrice.errors?.required || startingPrice.pristine">
    Price is required
  </div>
</form>
```


Form Validation - ngForm

Use ngSubmit to trigger a form submit

```
<form #form="ngForm" (ngSubmit)="onSubmit(form.value)">
  ...
  <button type="submit" [disabled]="!form.valid">
    Submit
  </button>
  ...
</form>
```

Task #24

Extend the AuctionDetailEdit
Component

Model-Driven forms

Model-driven forms

- Defined in component class and accessible in template and class
- Create **FormGroups** composed of **FormControls**
- **Validators** set per FormControl in component class

Model-driven forms

Import ReactiveFormsModule to get them working

```
import { ReactiveFormsModule } from '@angular/forms';

@NgModule({
  imports: [ReactiveFormsModule]
})
```

Model-driven forms - FormBuilder

FormBuilder

- Helper to build up forms
- Creates a **FormGroup** with list of **FormControls** with optional **Validators**

Form Validation – Model-Driven

- Everything exported by [@angular/forms](#)
- Inject FormBuilder in your class

```
import {FormBuilder, FormGroup, Validators} from '@angular/forms';  
...  
constructor(private fb: FormBuilder) { ... }
```

Form Validation – Model-Driven

- Declare a FormGroup
- Setup the group with the FormBuilder service

```
form: FormGroup;  
...  
this.form = this.fb.group({ ... });
```


Form Validation – Model-Driven

- Add FormControls with Validators to FormGroup
- Add multiple Validators with Validators.compose(validators)
- Control=value + validators (optional)

```
this.form = this.fb.group({
  startingPrice: [this.auction.startingPrice, Validators.required],
  title: [
    this.auction.auctionItem.title,
    Validators.compose([Validators.required, Validators.minLength(11)])
  ]
});
```

Form Validation – Model-Driven

- Connect FormGroup to template form with `formGroup`
- Connect FormControl with inputs with `formControlName`

```
<form [formGroup]="form" (ngSubmit)="onSubmit()">
  ...
  <input type="text" formControlName="title">
  ...
</form>
```

Form Validation – Model-Driven

Helper functions to get controls and errors

```
form.hasError('required', 'title')  
form.find('title').hasError('required')
```

Task #25

Add an AuctionNew Route

Directives

Directives

- HTML-Attributes without own content
- access to host-attributes and host-element
- component extends the Directive-Definition

Directives - Types

- **Components:** Directives with own template
- **Attribute-Directives:** changes only attributes of a given element, behavior and look, e.g. `ngStyle`
- **Structural-Directives:** add/remove DOM.Nodes, e.g. `*ngFor`, `*ngIf` (asterix-prefix)

Directive Decorator

- ES7 / TypeScript Decorator (Annotation)
- Metadata/Configuration of your Directive
- Metadata will configure how the outside world will interact with your component

Directive Decorator Interface

Name	Description	Default
selector	Define CSS Selector to match the element	undefined
providers[]	Define the injectable services	[]
inputs[]	Define components input attributes (deprecated)	[]
outputs[]	Define output events (deprecated)	[]
host { k: v }	set attributes on the host element (deprecated)	{}

Directive Decorator Interface

- use decorators instead of deprecated options!
- inputs = @Input, outputs = @Output,
host = @HostListener/HostBinding

Directives - Example

```
@Directive({
  selector: '[tooltip]',
})
class Tooltip {
  @Input() tooltip: string;
  constructor() {}

  @HostListener('mouseenter')
  onMouseEnter() {}
  @HostListener('mouseleave')
  onMouseLeave() {}
}
```

The diagram illustrates the following mappings:

- `selector: '[tooltip]'` points to `<ANY tooltip="Info"></ANY>`
- `@Input() tooltip: string;` points to `Bind input from tooltip`
- `@HostListener('mouseenter')` and `@HostListener('mouseleave')` both point to `Extend host element with event bindings`

Directive – Low Level Operations

- often need access to current view, element or template
- low level services: `TemplateRef`, `ViewContainerRef`, `ElementRef`
- `TemplateRef` = reference to `<template>` (important for structural directives)
- `ViewContainerRef` = container of the current view
- `ElementRef` = DOM-Node, location where the view is attached

Directives – Low Level Operations

```
@Directive({  
  selector: '[tooltip]',  
})  
class Tooltip {  
  @Input() tooltip: string;  
  constructor() {}  
  private templateRef: TemplateRef,  
  private viewContainer: ViewContainerRef  
  ) {}  
}
```

Reference to the<template>



Interface to modify ViewContainer



Directives – Low Level Operations

ViewContainerRef

Name	Description
element	Anchor element that specifies the location of this container in the containing View.(=ElementRef)
clear()	Destroys all Views in this container.
...	

Structure Directive - ngIf

```
@Directive({selector: ['ngIf'], inputs: ['ngIf']})
export class NgIf {
    private _prevCondition: boolean = null;

    constructor(private _viewContainer: ViewContainerRef,
                 private _templateRef: TemplateRef<Object>) {}

    set ngIf(newCondition: any /* boolean */) {
        if (newCondition && (isBlank(this._prevCondition) || !this._prevCondition)) {
            this._prevCondition = true;
            this._viewContainer.createEmbeddedView(this._templateRef);
        } else if (!newCondition && (isBlank(this._prevCondition) || this._prevCondition)) {
            this._prevCondition = false;
            this._viewContainer.clear();
        }
    }
}
```

Task #26

Create a Tooltip Directive I

Task #27

Tooltip Directive II

Pipes

Pipes

- data-transformations in expressions
- pipe implements the *PipeTransform* interface
- usable in components via pipes[]

```
interface PipeTransform { transform (value: any [, arg1: any, ...]): any; }
```

Pipe Decorator

- ES7 / TypeScript Decorator (Annotation)
- Metadata/Configuration of your Pipe

Name	Description	Dafault
name	Define the name of the pipe	undefined
pure	Define if the pipe is stateless	true

Pipes – UpperCase Pipe

Set name via decorator

Implement PipeTransform Interface

```
@Pipe({ name: 'uppercase' })  
class UpperCasePipe implements PipeTransform {  
  transform(value: string): string {  
    return value.toUpperCase();  
  }  
}
```

Define your transform function

Build-in Pipes

- AsyncPipe
- UpperCasePipe
- LowerCasePipe
- JsonPipe
- SlicePipe
- DecimalPipe
- PercentPipe
- CurrencyPipe
- DatePipe

Async Pipe

Async Pipe

- Subscribe to Observable
- UnSubscribe on component destruction
- Build-In Pipe
- Simple use: `{{auction | async}}`

Task #28

Use the async pipes

Task #29

Build a shout pipe

Testing overview

“Blame yourself if you’re not doing it!”

Testing in Angular 7

- main feature in Angular 7
- several mock object available out of the box
- framework for E2E testing (Protractor)
- unit testing with Jasmine
- other test frameworks can be also used
- test runner: Karma

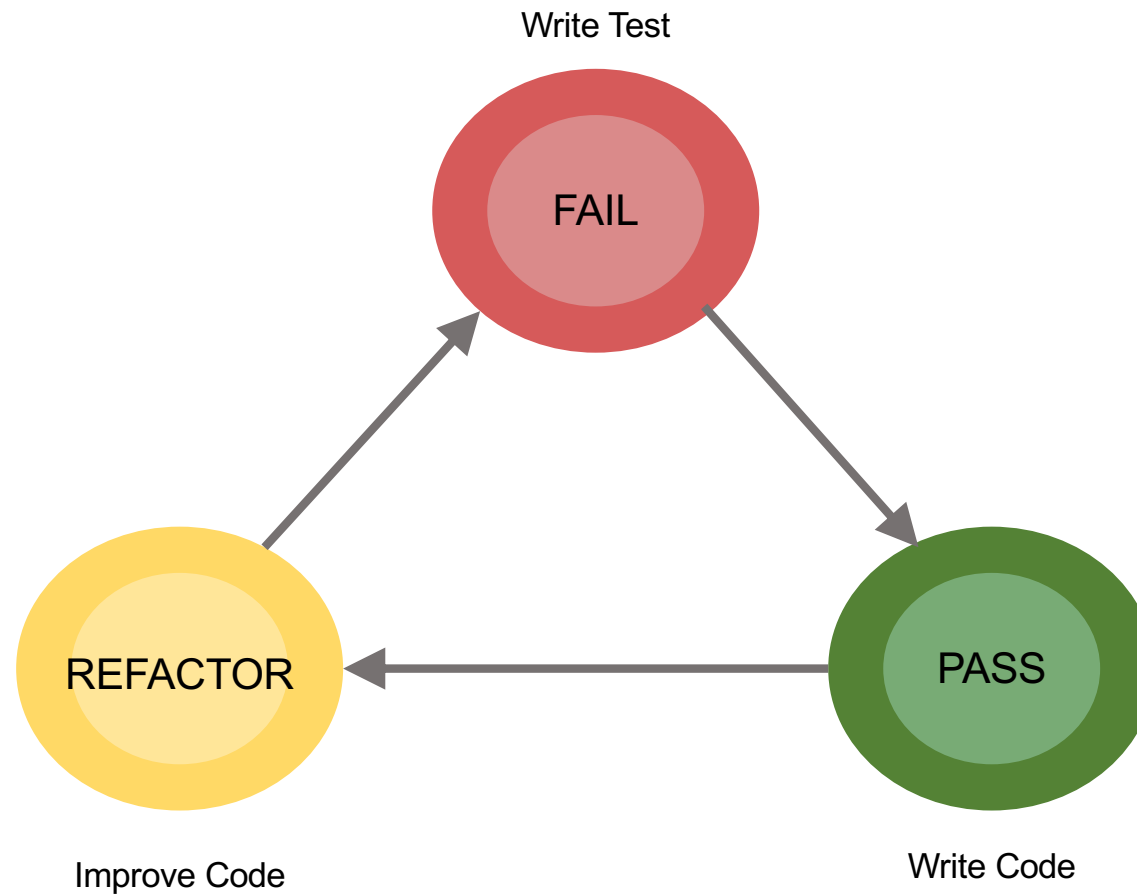
Unit Testing

- code level
- every component can be unit tested (!)
- isolated testing
- mocking and stubbing support

E2E Testing

- user level (Browser)
- browser robot
- assertions against the document
- are very fragile in agile dev processes

Test Driven Development (TDD)



Test Driven Development (TDD)

1. Write a test case and make sure it fails. **(red)**
2. Satisfy the test case with minimal effort. **(green)**
3. Improve/refactor your code...
 - a. Meet general code guidelines.
 - b. Make it readable and comprehensible.
 - c. Remove redundant code.
4. Verify that the test case is still passing. **(green)**

Testing – Jasmine

For unit tests and Protractor

Jasmine Basic

- Test Suite: `describe()`
- Test Case: `it()`
- Setup: `beforeEach()`
- Tear Down `afterEach()`
- Expectation: `expect()`

Test Suites can be nested!

Jasmine Matchers

- `toBe()`
- `toEqual()`
- `toContain()`
- `toBeUndefined()`
- `toBeTruthy()`
- `toBeFalsy()`
- `toThrow()`
- `toBeGreaterThan()`
- `toBeLessThan()`
- `toBeCloseTo()`

More matches available in conjunction with Spies!

Mocks and Stubs

- **Mocks:**

- Can meet expectations and can cause your tests to fail
- Are mostly part of the framework or library
- e.g. MockBackend

- **Stubs:**

- Are objects or classes to let your tests run in general
- Are mostly implemented by yourself
- e.g. AuctionDataServiceStub

Testing – Unit Testing

\$ ng test

Helper Function

- `async`
- `inject`
- `TestBed`

Helper Function - async

async - wraps test cases in own zone to handle async code parts

```
it('should work', async(() => {  
  expect(true).toBeTruthy();  
}));
```

Helper Function - inject

inject - get instances from the dependency injection

What we want to
inject

The typed parameter with the
injected instance

```
it('should work', inject([AuctionDataService], (service: AuctionDataService) => {  
    expect(typeof service.getAll).toBe('function');  
}));
```


Helper Function - Testbed

- Configuration and initialisation of the environment to unit test Angular apps
- Generates NgModule
- Methods to create and use services and components

Helper Function - Testbed

Testbed - generate NgModule with declarations and services

```
TestBed.configureTestingModule({  
  declarations: [  
    AuctionListComponent,  
    ShoutPipeStub,  
    BmRedDirectiveStub  
  ],  
  providers: [{  
    provide: AuctionDataService,  
    useClass: AuctionDataServiceStub  
  }]  
});
```

Helper Function - Testbed

Testbed - create and handle a component

```
const fixture = TestBed.createComponent(AuctionListComponent);

const auctionList = fixture.debugElement.componentInstance as AuctionListComponent;
const element = fixture.debugElement.nativeElement as HTMLElement;

fixture.detectChanges();

expect(auctionList.auctions.length).toBe(3);
expect(element.innerHTML).toBe('<h2>Hello!</h2>');
```

Components

Testing Components

For simple components you can just instantiate the class. No magic.

```
it('should call AuctionDataService.getAuctions() in onInit', async(() => {  
  const auctionData = new AuctionDataServiceStub();  
  const auctionList = new AuctionListComponent(auctionData as AuctionDataService);  
  
  expect(auctionList.auctions).toBeUndefined();  
  
  auctionList.ngOnInit();  
  expect(auctionList.auctions).toBeDefined();  
  expect(auctionList.auctions.length).toBe(3);  
}));
```

Testing Components

Advanced test with TestBed and NgModule

```
it('should get auctions in onInit', async(() => {  
  const fixture = TestBed.createComponent(AuctionListComponent) as  
    ComponentFixture<AuctionListComponent>;  
  
  const auctionList = fixture.debugElement.componentInstance as AuctionListComponent;  
  const element = fixture.debugElement.nativeElement as HTMLElement;  
  
  fixture.detectChanges();  
  
  expect(auctionList.auctions.length).toBe(3);  
  expect(element.querySelectorAll('li').length).toBe(3);  
}));
```

**Directives
don't have own templates...**

Testing Directives

Only testable with a component

```
import { By } from '@angular/platform-browser';
@Component({
  selector: 'bm-blank-cmp',
  template: `<div bmRed>Content</div>`
})
class TestComponent { }

beforeEach(async(() => {
  const fixture = TestBed.createComponent(TestComponent);
  const directive = fixture.debugElement.queryAll(By.directive(RedDirective))[0];
  const el = directive.nativeElement as HTMLDivElement;
})));
```


Testing Directives

```
it('set a background if its clicked', async(() => {  
  const el = directive.nativeElement as HTMLDivElement;  
  expect(directive.styles['backgroundColor']).toBeUndefined();  
  
  el.click();  
  
  fixture.detectChanges();  
  expect(directive.styles['backgroundColor']).toBe('red');  
});
```

Services

Testing Services

For simple services you can just instantiate the class. No magic.

```
it('should return all auctions synchronous', () => {  
  const service = new AuctionDataService();  
  expect(service.getAll()).toEqual(auctionsStub);  
});
```

Testing Services

Advanced test with TestBed and MockBackend (I)

```
TestBed.configureTestingModule({
  providers: [
    MockBackend,
    BaseRequestOptions,
    {
      provide: Http,
      useFactory: (backendInstance: MockBackend,
                  defaultOptions: BaseRequestOptions) => {
        return new Http(backendInstance, defaultOptions);
      },
      deps: [MockBackend, BaseRequestOptions]
    },
    AuctionDataService
  ]
});
```

Testing Services

Advanced test with TestBed and MockBackend (II)

```
it('should return all auctions', async(
  inject(
    [AuctionDataService, MockBackend],
    (service: AuctionDataService, backend: MockBackend) => {
      backend.connections.subscribe((connection: MockConnection) => {
        let options = new ResponseOptions({
          body: JSON.stringify(auctionsStub)
        });
        connection.mockRespond(new Response(options));
      });
      service.getAll().subscribe(
        auctions => expect(auctions).toEqual(auctionsStub)
      );
    }
  )
));
```

Stubbing

Stubbing - View Items

- Unit tests are atomic → stub all dependencies
- Simply create dummy Components, Directives with same selector and Pipes with same name of the original one

Stubbing - Service

- Create service Class with the API of the original one
 - `class MockService implements OriginalService { }`
- Register stub under the original token in TestBed NgModule providers
 - `{ provide: OriginalService, useClass: MockService }`

Stubbing - Service

stub routing params if necessary for component

```
class StubActivatedRoute extends ActivatedRoute {  
  constructor() {  
    super();  
    this.params = Observable.of({ id: '5' });  
  }  
}
```

Task #30

Write a Unit Test

Testing – E2E Tests

with Protractor

E2E Testing Basics

- Simulate user interaction
- Assertions based on DOM
- Usually **significantly** slower than unit tests

Organisation Patterns

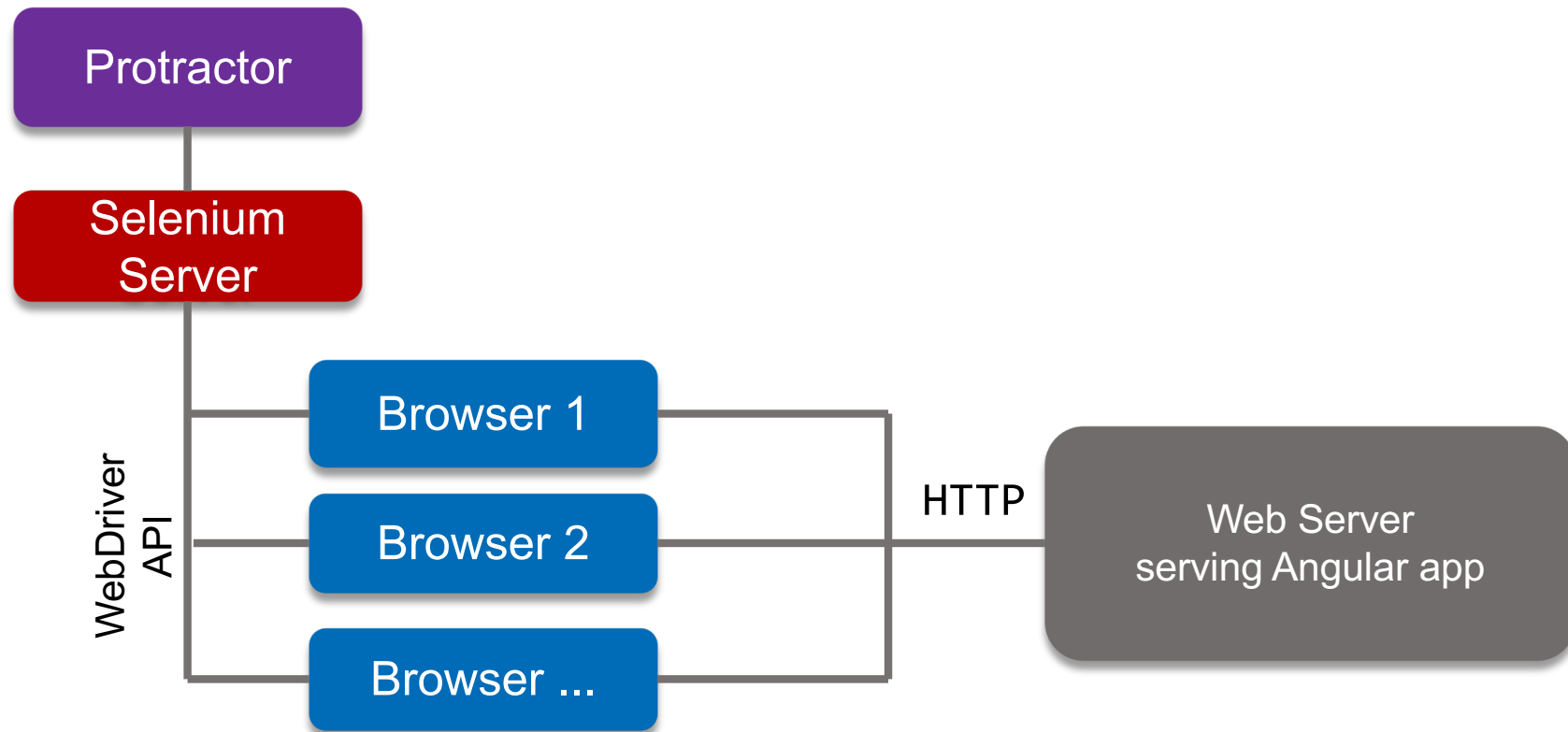
- QA writes scenarios in natural language
- Tester implements scenarios using E2E tests
- E2E tests are executed on CI server during nightly build

Protractor

- E2E testing framework for Angular2 app
- Built on top WebDriverJS
- Jasmine
- Selenium Server

“Protractor runs tests against your application running in a real browser, interacting with it as a user would.”

Protractor



Quick Start

Install Protractor and WebDriver Manager

```
npm install -g protractor
```

Install Selenium Server

```
webdriver-manager update
```

Start Selenium Server

```
webdriver-manager start
```


Protractor Config

```
// An example configuration file
exports.config = {
  // The address of a running selenium server.
  seleniumAddress: 'http://localhost:4444/wd/hub',

  // Capabilities to be passed to the webdriver instance.
  multiCapabilities: [{
    'browserName': 'chrome'
  }, {
    'browserName': 'firefox'
  }],

  // Spec patterns are relative to the configuration file location passed
  // to protractor. They may include glob patterns.
  specs: ['test/e2e/**/*.spec.js'],
  // Options to be passed to Jasmine-node.
  jasmineNodeOpts: {
    showColors: true, // Use colors in the command line report.
  }
};
```

Protractor globals

- **browser**

wrapper around WebDriver instance (browser.driver)

- **element**

helper function for finding and interacting with DOM elements

- **by**

collection of locator strategies

- **protractor**

static variables and classes (e.g. protractor.Key)

Locators

```
// find an element using a css selector  
by.css('.myclass')  
  
// find an element with the given id  
by.id('myid')
```

See [reference](#) for further details.

Actions

```
// el is of type ElementFinder.  
// It will not contact the browser until an action method has been called.  
var el = element(locator);  
  
// Click on the element  
el.click();  
  
// Send keys to the element (usually an input)  
el.sendKeys('my text');  
  
// Clear the text in an element (usually an input)  
el.clear();  
  
// Get the value of an attribute, for example, get the value of an input  
el.getAttribute('value');  
  
// Get the text value of an element  
el.getText()
```

See [reference](#) for further details.

Protractor Survival Kit

- *element()* takes a **Locator** and returns an **ElementFinder**
- An **ElementFinder** has a set of action methods
 - *sendKeys()*, *click()*, *getText()*, etc.
- An **ElementFinder** wraps its corresponding **WebElement**
 - *el.getWebElement()*
- All actions are *asynchronous* and return a *promise*

ElementFinder vs. WebElement

- **WebElement**
 - immediately sends a command over to the browser asking it to locate the element
- **ElementFinder**
 - stores the locator information until an action is called
 - enables us to create page objects
 - enables chaining to find subelements
 - provides additional helper methods like *isPresent()*
 - provides access to the underlying **WebElement** via *getWebElement()*

Page Object

- encapsulate information about your app's elements
- can be reused across multiple tests
- If app's elements change in most cases only the corresponding page objects need to be updated (not the test cases!)

Protractor

Creating page objects

```
var AngularHomepage = function() {  
  this.get = function() {  
    browser.get('http://www.angular.io');  
  };  
  this.setName = function(name) {  
    element(by.model('yourName')).sendKeys(name);  
  };  
  this.getGreeting = function() {  
    return element(by.binding('yourName')).getText();  
  };  
};
```


Protractor

Using page objects

```
describe('angular homepage', function() {  
  it('should greet the named user', function() {  
    var angularHomepage = new AngularHomepage();  
    angularHomepage.get();  
  
    angularHomepage.setName('Felix');  
  
    expect(angularHomepage.getGreeting()).toEqual('Hello Felix!');  
  });  
});
```

Configuring Test Suites

```
exports.config = {  
  
  [...]  
  
  //Spec patterns are relative to the location of the spec file.  
  // They may in include glob patterns.  
  suites: {  
    homepage: 'tests/e2e/homepage/**/*.Spec.js',  
    search: ['tests/e2e/contact_search/**/*.Spec.js',  
            'tests/e2e/venue_search/**/*.Spec.js']  
  },  
  
  [...]
```

...

```
protractor protractor.conf.js --suite homepage
```

Protractor

Reading values

```
describe('zhaw.ch homepage', function() {  
  
  it('should have a title', function() {  
    browser.get('http://zhaw.ch');  
  
    expect(browser.getTitle()).toEqual('zhaw.ch | Homepage')  
  });  
  
});
```

Protractor

Interacting with the page

```
describe('myapp homepage', function() {  
  
  it('should add one and two', function() {  
    browser.get('http://myapp.ch');  
    element(by.css('.first')).sendKeys(1);  
    element(by.css('.second')).sendKeys(2);  
    element(by.css('[data-test="auctionDetail.id"]')).click();  
  
    expect(element(by.css('.latest')).getText()).toEqual('5');  
  });  
});
```

Protractor Best Practices

- Use page objects
- Use test suites
- Try to stick to the Protractor API, only get your hands dirty with WebDriverJS if really necessary
- Find stable ways to locate your DOM nodes
- Favour logical/semantic locators over CSS selectors
- Try to include as many browsers as possible

Task #31

Write an e2e Test

Cross-Browser Testing

Why?

- Unfortunately browsers aren't executing JS in an unified way
- Different JS engines
- Support for different ECMAScript versions
- Browser bugs

Solutions (Cloud)

- BrowserStack
- <http://www.browserstack.com/>
- Sauce Labs
- <https://saucelabs.com/>

Authentication

Agenda

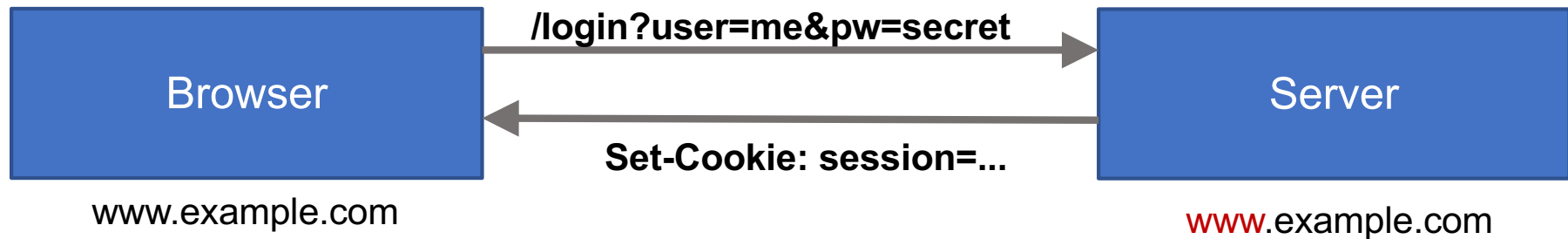
- Authentication approaches
- JWT (JSON Web Token)
- Code
- WebSockets

HOW

Authentication Approaches

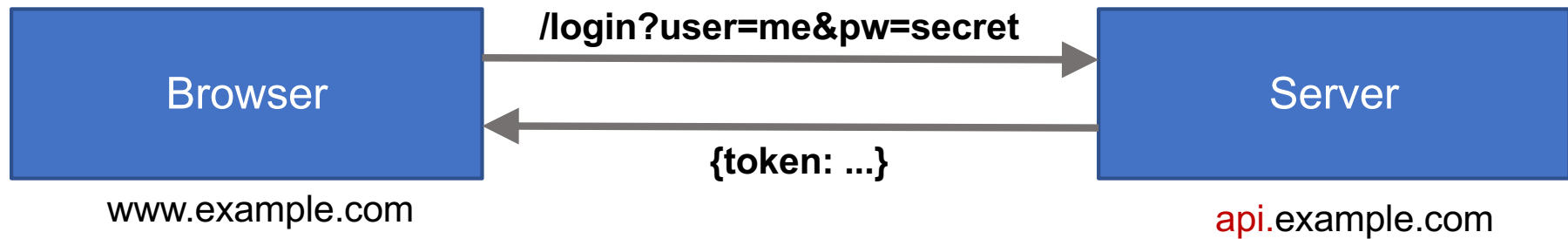
Approaches

Cookie



CORS

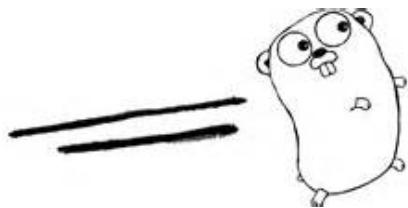
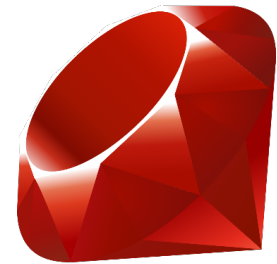
Token



JWT

Token approach

JWT - Support



JWT – Crash Course

JWT – Ctash Course

Header

Payload

Signature

JWT – Crash Course

Header

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

Plan

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
```

Base64

JWT – Crash Course

Payload

```
{  
  "name": "Felix Muster",  
  "admin": true  
}
```

Plan

- **exp:** Expiration
- **jti:** Unique Identifier
- **iat:** Creation time

```
eyJ0YX11IjoiU2FzY2hhIEJyaW5rIiwiaWF0IjYWRtaW4iOnRydWV9
```

Base64

JWT – Crash Course

Signature

```
HMACSHA256(  
    base64UrlEncode(header) + '.' +  
    base64UrlEncode(payload),  
    'secret'  
)
```

Plan

```
pU7P4lZ-7mBuHYZmJK1_mD37lx-8XuCejNaKcvCerSo
```

Base64

•

●

Base64

Header

Payload

Signature

Security

Security

- No sensitive data in a JWT
- Signing data, not encrypting
- Keep the secret a secret

Server side

JWT – Code - Server

Login

```
app.post('/login', function (req, res) {  
  var user = {  
    id: 123,  
  };  
  var token = jwt.sign(user, 'secret', { expiresInMinutes: 60 })  
  res.json({ token: token });  
});
```

Verifying

```
var decoded = jwt.verify(token, 'secret');
```

Client side

Angular2

- Unlike AngularJS, Angular2 does NOT have an interceptor
- You have to set the headers in every request
- Solution: Wrap Http

e.g.
angular-jwt

http replacement

```
class App {  
  constructor(public authHttp: AuthHttp) {}  
  
  getThing() {  
    this.authHttp.get('http://example.com/api/thing')  
      .pipe(map(res => res.json()))  
      .subscribe(  
        data => this.thing = data,  
        err => console.log(error),  
        () => console.log('Request Complete')  
      );  
  }  
}
```

token helper

```
@Component({  
  selector: 'secret-route'  
  template: `<h1>If you see this, you have a JWT</h1>`  
})  
  
@CanActivate(() => tokenNotExpired())  
  
class SecretRoute {}
```

angular-jwt

- lightweight wrapper
- lets see:
<https://github.com/auth0/angular2-jwt>

Task #32

Create a Login Dialog

Cheat Sheets

Types

Booleans	boolean
Numbers	number
Strings	string
Lists	number[] Array<number>
Maps	interface /* separated defined and named */ {...} /* inline */
Enumeration	enum Employees {Miriam, Matthias}
Any	any
Void	void // only as return type for functions/methods
Type Casting	<type> varName as type

Functions

- Fat arrow functions
- typed functions (parameters, return value)
- optional parameter - “?” after parameter name
- default parameter value
- rest-parameter to allow unknown length of parameters

ES2015/TS Classes

class	nicer way to define prototypes
public	the default for attributes and methods
private	only accessible within their declaring class
protected	accessible from within their declaring class and classes derived from their declaring class
static	methods or attributes can be called or get and set without an instance
extend	class gets extended by another class

Interfaces – The new keywords

- **interface**

create a shape with types

- **implements**

classes can implement an **interface**

Component Decorator - Interface

Name	Description	Default
selector	Define CSS Selector to match the element	undefined
template	View-Template as string	' ' (EmptyString)
templateUrl	View-Template via URL	undefined
styleUrls[]	Reference to styles via URL	[]
directives[]	Inject other directives	[]
pipes[]	Inject other pipes	[]
providers[]	Define the injectable services	[]

Component Decorator - Interface

Name	Description	Default
encapsulation	Define the scoping of styles	Emulated
changeDetection	Specify a custom changeDetection	CheckAlways

- there are more, but this are the most used and important ones

Component Lifecycle Hooks

Hook method	Interface	Description
ngOnChages	OnChanges	Called when an input or output Binding value changes
ngOnInit	OnInit	After the first ngOnChanges
ngDoCheck	DoCheck	Developer's custom change detection
ngAfterContentInit	AfterContentInit	After component content initialized
ngAfterContentChecked	AfterContentChecked	After every check of componennt content
ngAfterViewInit	AfterViewInit	After component's view(s) are initialized
ngAfterViewChecked	AfterViewChecked	After every check of a component's view(s)
ngOnDestroy	OnDestroy	Just before the directive is destroyed

View Encapsulation

Mode	Description
<code>ViewEncapsulation.None</code>	No encapsulation, styles in head
<code>ViewEncapsulation.Emulated</code>	Styles in head with attribute suffix (scoped styles)
<code>ViewEncapsulation.Native</code>	Use the Shadow DOM

Zone.js

Solutions in Angular 7

- Change detection
- Better stack traces

The problem

```
<button (click)="addTask()">
```

Add a new task

```
</button>
```

Synchronous execution

```
@Component()  
class App {  
  tasks: string[] = [];  
  
  addTask() {  
    this.tasks.push('Finish essay');  
  }  
}
```

Asynchronous execution

```
@Component({})  
class App {  
  tasks: string[] = [];  
  constructor(private http:HttpClient) { }  
  
  ngOnInit() {  
    this.http.get('/tasks')  
      .map(res => res.json())  
      .subscribe((tasks) => { this.tasks = tasks; });  
  }  
}
```

Async problems

```
function asyncFn() {}  
  
let startTime = new Date();  
  
a();  
setTimeout(asyncFn, 1000);  
b();  
c();  
  
time = new Date() - start;
```

With zones

```
function myApp() {}  
  a();  
  setTimeout(asyncFn, 1000);  
  b();  
  c();  
}  
  
zone.run(myApp);
```

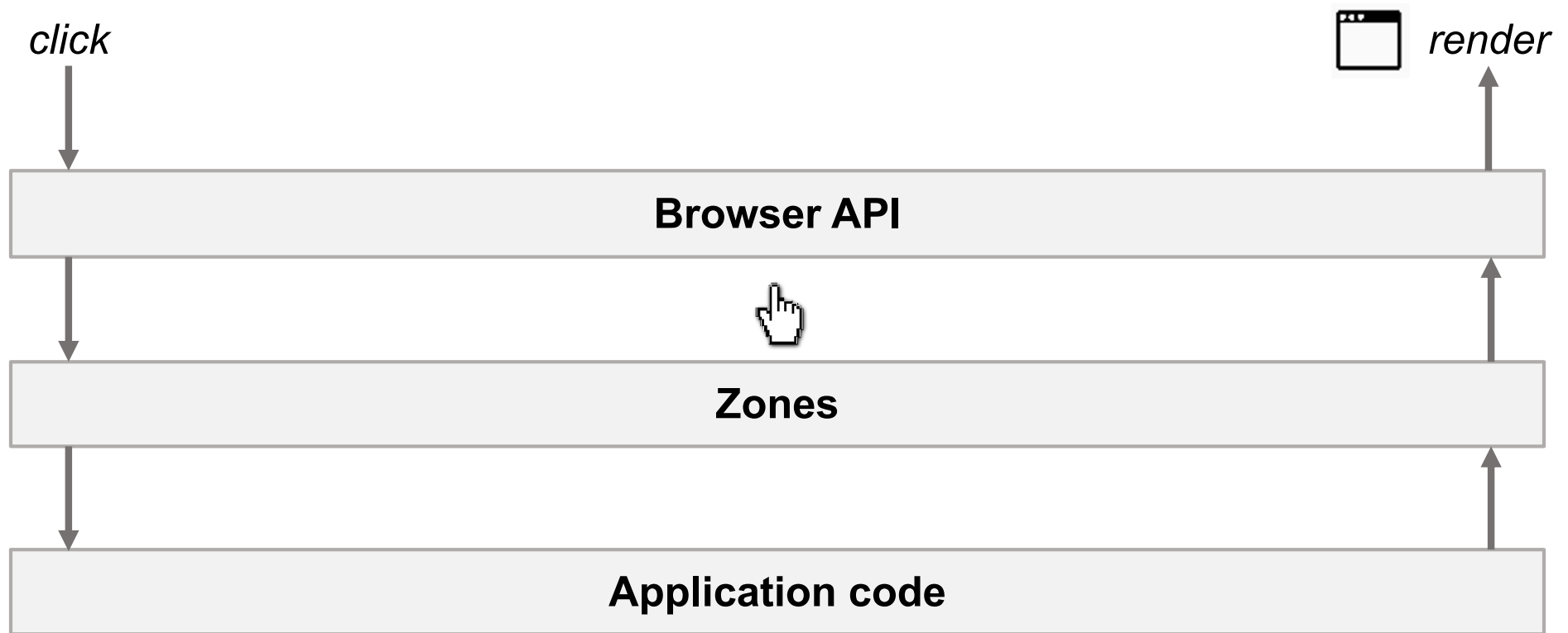

Zone tasks

```
var myZoneSpec = {  
  beforeTask: function () {  
    console.log('Before task');  
  },  
  afterTask: function () {  
    console.log('After task');  
  }  
};  
  
var myZone = zone.fork(myZoneSpec);  
myZone.run(myApp);
```

How it works

- Zone.js patches browser events
- Example async functions
 - setInterval()
 - requestAnimationFrame()
 - addEventListener()
 - removeEventListener()
- Example sync functions
 - alert()
 - prompt()

How it works



Change Detection

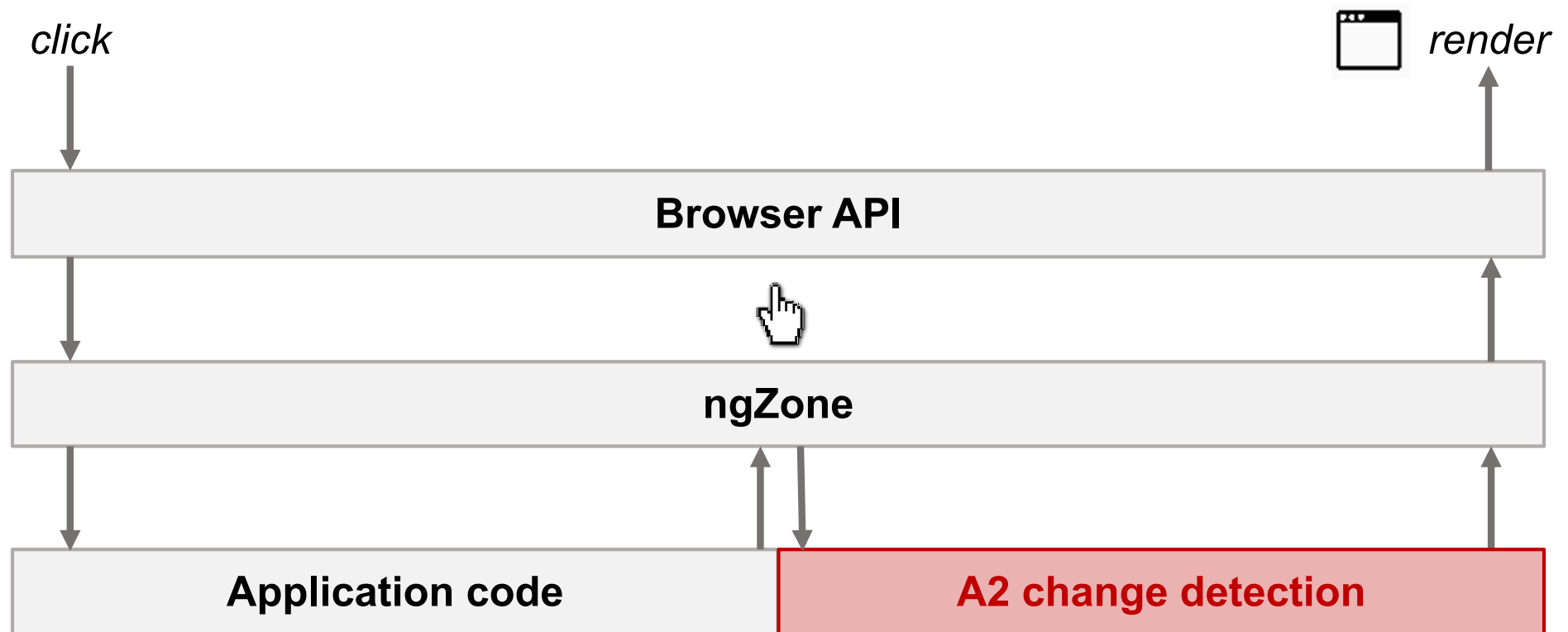
Change Detection

- react on user interactions and programmatic changes
- update data in view and model → DOM and component
- **CHANGE DETECTION!**

Change Detection

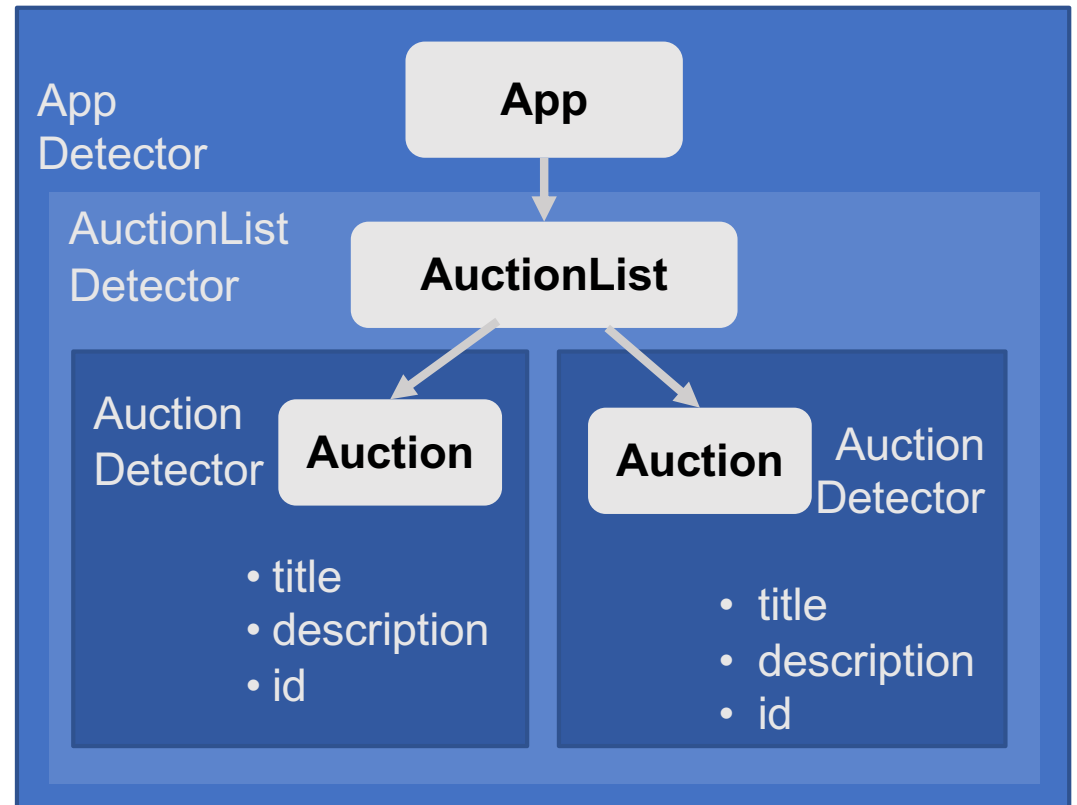
- based on Zone.js → Angular triggers change detection
- own zone called ngZone
- in most cases: Angular handles it

Change Detection



Change Detection

- change detection per component
- each component has own change detector
- change detector: custom function, generated at runtime (it knows what to check)



Default

Change Detection - Default

- **Default:** Runs from top (root-component) to bottom (leaf-components)
- component has only access to its own context
→ no cycle-checking
- **But:** For every change all change detectors are executed

Immutable

Change Detection - Immutables

- Solution: Immutable objects
 - not changeable → any change creates a new object
 - simple change detection works (`obj1===obj2`) :)
 - multiple change detection strategies
 - set detection strategy to `OnPush` via the component decorator

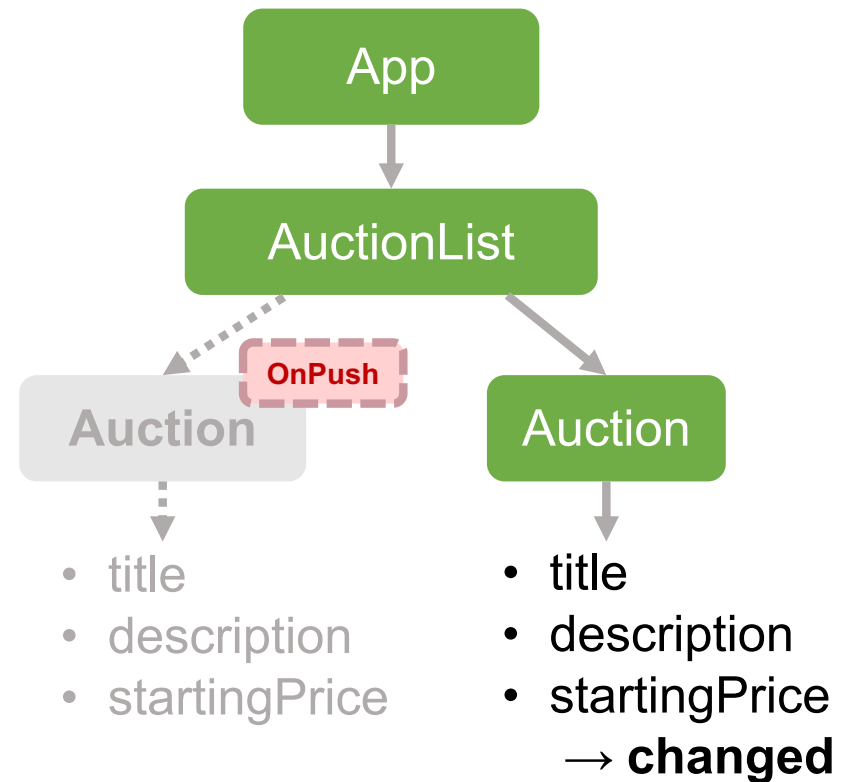
Change Detection - Immutables

Set changeDetection to OnPush

```
@Component({  
  template: '<h3>{{auction.auctionItem.title}} ({{auction.id}})</h3>',  
  changeDetection: ChangeDetectionStrategy.OnPush  
})  
class Auction {  
  @Input() auction;  
}
```

OnPush Strategy

- Left Auction component uses OnPush
- startingPrice of the second Auction component changed
- Only the detection path to the right Auction is executed



Change Detection - Immutables

- One last thing: ***What happens with Observables?***

Observables

Change Detection - Observables

- **Default:** ngZone handles observables
- **OnPush:** observables does not return or change objects → firing events
- **Solution:** manually mark the component as “changed”!

Change Detection - Observables

Manually mark a component for check

```
// inject change detector
constructor(private detector: ChangeDetectorRef) {}
...
// use change detector to add components path for change detection
this.detector.markForCheck();
```

Change Detection

- additional functions on *ChangeDetectorReference*
- remove and readd a component from/to the detection tree
- Angular documentation of [ChangeDetectorRef](#)

View Encapsulation

Why View Encapsulation

- You want to fully isolate your components including stylesheets

View Encapsulation

- View Encapsulation defines how your stylesheets are isolated
- Defined per component
- 3 modes of view encapsulation
- Styles are added to the document's <head>

View Encapsulation

Encapsulation can set to None, Emulated or Native.

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

@Component({
  selector: 'red-box',
  encapsulation: ViewEncapsulation.None/Emulated/Native,
  template: '<div class="red">Hello World</div>',
  styles: [`
    .red { background-color: red; }
  `]
})
```

ViewEncapsulation.

None

ViewEncapsulation.None

With «None» styles are not isolated at all.

```
<head>
  <style>
    .red { background-color: red; }
  </style>
</head>

<red-box>
  <div class="red">Hello World</div>
</red-box>
```

ViewEncapsulation.

Emulated

ViewEncapsulation.Emulated

Angular's default mode. With «Emulated» styles get an extra attribute.

```
<head>
  <style>
    .red[_ngcontent-gus-3] { background-color: red; }
  </style>
</head>

<red-box _ngghost-gus-3="">
  <div _ngcontent-gus-3="" class="red">Hello World</div>
</red-box>
```

ViewEncapsulation.

Native

ViewEncapsulation.Native

With “Native” the shadow DOM is used.

```
<red-box>
  #shadow-root
  <style>
    .red { background-color: red; }
  </style>
  <div class="red">Hello World</div>
</red-box>
```

Why not use Shadow DOM as default?

- Browser support is **very poor** → [Can use overview](#)
- Polyfill [exists](#) but file size is **over 80kb**

Task #33

Analyse View Encapsulation Modes