



# Effective web application development with Angular

## Your trainer

### Mateusz Kulesza

Senior Software Developer,  
Team Leader, Scrum Master  
Project Manager  
Consultant and Trainer



## Your Training:

- Goal and plan
- Current responsibilities
- Questions, discussion, needs
- Elastic program

## Practical professional experience in:

- HTML5, CSS3, SVG, EcmaScript 5 i 6
- jQuery, underscore, backbone.js
- canjs, requirejs, dojo ...
- Grunt, Gulp, Webpack, Karma, Jasmine ...
- Angular.JS, **Angular2**, React, RxJS, Flux

# Package management



Node Package  
Manager

- dependency management for JavaScript ecosystem
- dependencies described with exact version stored in `package.json` file
- `npm install` - installs packages in project
- `npm update` - checks if newer versions exists and installs them
- `npm install nazwa-pakietu --save-dev` - installs package and saves its name and version in `package.json`

# WebPack

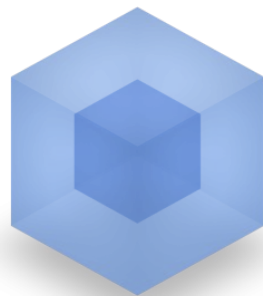
- module bundler
- supports multiple module formats: ES2015, AMD, CommonJS (npm)
- everything is a module (np. scss, html, graphics)
- Hot Module Replacement - can reload code in working application
- can be easily configured to work with other tools (Gulp, Grunt)
- de facto a current standard in React ecosystem, and getting adopted in other popular projects

```
// installing webpack-cli, globally for commandline use
```

```
install webpack --global
```

```
// local install to be invoked from scripts and tools
```

```
npm install webpack --save-dev
```



**webpack**  
MODULE BUNDLER

# Webpack Config

```
module.exports = {  
  entry: [  
    './js/index.js'  
  ],  
  output: {  
    path:    dirname + '/static/',  
    filename: 'bundle.js'  
  },  
  plugins: [],  
  module: {  
    rules: [{  
      test: /\.js$/, use: ['babel'], exclude: /node_modules/  
    }]  
  },  
  devtool: 'source-map'  
};
```



# extensiblewebmanifesto.org

#extendthewebforward

Set of low level browser APIs allowing developer to:

- **Extend** browser functionality:
  - without installing any extensions
  - without hacks, quirks and workarounds
- **Modify** working of existing functionalities:
  - locally in scope of current application ( webpage / domain )
  - safe for user

All done using just JavaScript!

**Developer will be able to build his own high level APIs to extend browser.**

Example projects include: Mozilla **X-Tags**, Google **Polymer**... Angular, and now **Angular 2**



# JavaScript 2015

“JavaScript next”

# ~~ECMAScript 6?~~

# ECMAScript 2015

- modules
- lots of very usefull “syntax sugar”
- lexical variable scope (let), lambda (arrow) functions and a lot more
- You can use now, even in older browsers thanks to transpilers like :

The word 'BABEL' is written in a large, bold, yellow font with a thick, hand-drawn brushstroke texture. The letters are slightly slanted and have irregular edges, giving it a dynamic and artistic feel.

<https://babeljs.io/>



# Anonymous function

## (Lambda)

// Default - expression

```
var odds = myArr.map(v => v + 1);  
var nums = myArr.map((v, i) => v + i);  
var pairs = myArr.map(v => (  
    {even: v, odd: v + 1}  
));
```

// Non-expressions goes in braces

```
nums.filter(v => {  
    if (v % 5 === 0) {  
        return true;  
    }  
});
```

// Lexical this

```
var bob = {  
    _name: "Bob",  
    _friends: [],  
    getFriends() {  
        return this._friends.forEach(f =>  
            this._name + " knows " + f  
        )  
    }  
}
```

# Destructuring

```
// list matching
```

```
var [a, , b] = [1,2,3];
```

```
// object matching
```

```
var { op: a, lhs: { op: b }, rhs: c }
```

```
  = getASTNode() // i.e. { op: 'a', lhs: {op: 'b' }, rhs: 'c' }}
```

```
// object matching shorthand
```

```
var {op, lhs, rhs} = getASTNode()
```

```
// Can be used in parameter position
```

```
function g({name: x}) {
```

```
  console.log(x);
```

```
}
```

```
g({name: 5})
```

# Default, ...Spread i ...Rest

```
function f(x, y = 12) {  
  // default value of y ( only if y === undefined)  
  return x + y;  
}  
f(3) === 15;  
function f(x, ...y) {  
  // y collapsed to array of rest of the arguments  
  return x * y.length;  
}  
f(3, "hello", true) === 6;  
function f(x, y, z) {  
  return x + y + z;  
}  
// expanding array to provide multiple arguments  
f(...[1, 2, 3]) === 6;
```

# Dynamic literal

```
var obj = {  
  __proto__: theProtoObj,  
  // === 'handler: handler'  
  handler,  
  // === toString: function toString() {  
    toString() {  
      // Super calls  
      return "d " + super.toString();  
    },  
  // Dynamic properties  
  [ 'prop_' + (() => 42)() ]: 42  
};
```

# Promises

```
function timeout(duration = 0) {  
    return new Promise((resolve, reject) => {  
        setTimeout(resolve, duration);  
    })  
}  
  
var p = timeout(1000).then(() => {  
    return timeout(2000);  
}).then(() => {  
    return Promise.reject("hmm... błąd!");  
}).catch(err => {  
    return Promise.all([timeout(100), timeout(200)]);  
}).then((result) => {  
    console.log("Wynik to: " + result);  
})
```



# Unit testing

JavaScript Applications

# KARMA- test runner

- Tool for running automated tests
- Allows running in multiple environments (browsers, devices, etc..)
- and using various testing tools and frameworks

```
npm install karma-cli -g  
npm install karma
```

```
npm install karma-jasmine  
karma-webpack  
karma-chrome-launcher  
karma-phantomjs-launcher
```

```
...
```

```
karma init
```



```
karma start
```



# Creating Test suites, tests

```
describe('calculator', function() {  
  describe('add()', function() {  
    it('should add 2 numbers together', function() {  
      // how you run assertions depends on tool you use  
    });  
  });  
});
```

- tests go inside **it()** block
- test suites (groups) are under **describe()** block
- testing report concatenates description of suites with description of test giving full sentences:

“SUCCESS 1 of 1: Calculator add() should add 2 numbers together”



<http://jasmine.github.io/>



# Testing frameworks

```
expect(calculator.add(1, 4)).toEqual(5);
```



```
expect(calculator.add(1, 4)).to.equal(5);
```

```
assert.equal(calculator.add(1, 4), 5);
```

```
calculator.add(1, 4).should.equal(5);
```



# Static code analysis

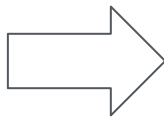


# ESLint

JSLint's successor

```
// installation  
npm install -g eslint
```

```
// configuration  
eslint --init
```



```
// .eslintrc.*  
{  
  "rules": {  
    "semi": ["error", "always"],  
    "quotes": ["error", "double"]  
  },  
  "extends": "eslint:recommended"  
}
```

<http://eslint.org/docs/rules/>

```
// running  
eslint testfile.js otherfile2.js ...
```



# Angular 2

2.4+, 4.0, ..., 7, 8, 9, 10 ... or just Angular ;-)



# @angular/cli

code generator, builder, runner... multi-tool

/ > ng



# “ng” tools - @angular/cli

Install:

**npm install -g @angular/cli**

Create new project in current directory

**ng new project\_name**

Run automated development server

**ng serve**

Generate code for new components, services, directives, etc.:

**ng generate component <name>**

# “ng” tools - angular-cli

`ng generate component YourComponentName` - Creates component

## extra flags:

- `--flat` - does not create new directories for components
- `-t` - inline template - template is put with same file as javascript
- `-s` - inline styles - styles goes in same file with javascript
- `--spec false` - skips autogenerating unit tests

You can also set them once in **angular.json**

# Bootstrapping

```
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic' ;  
import {MyAppModule} from './app/app.module' ;
```

Initializing application using module

```
platformBrowserDynamic().bootstrapModule(MyAppModule) ;
```

Main application component must be registered in module in **bootstrap** section and it's selector must match in current document. i.e:

```
<my-app> Loading... </my-app>
```

# Modules with @NgModule

```
import { NgModule }      from '@angular/core';
import { BrowserModule }  from '@angular/platform-browser';
import { AppComponent }   from './app.component';
```

```
@NgModule({
  // Importing :
  imports: [ BrowserModule ],
  // Declarations for renderer (Components, Directives, Pipes, etc..):
  declarations: [ AppComponent ],
  // Which component is a top-level component to look for:
  bootstrap:    [ AppComponent ],
  // Providing components and services to other parts of the app
  exports: [], providers: [],
})
export class MyAppModule { }
```





# Components

Basic building block for angular application

# Component Definition

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

```
@Component({  
  selector: 'app-root',  
  templateUrl: 'app.component.html',  
  styleUrls: ['app.component.css']  
})  
  
export class AppComponent {  
  title = 'app works!';  
}
```

'selector' is a simple CSS selector, which points angular to where in your DOM should this component be instantiated and mounted

In example:

[my-attribute] - on all tags with attribute

.spinner-loader - or class ...


app-root - or any (custom) element

<app-root></app-root>

# Templates

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

```
@Component({  
  selector: 'app-root',  
  template: `<div>  
    <h1>{{title}}</h1>  
  </div>`  
})  
export class AppComponent {  
  title = 'app works!';  
}
```



```
<app-root>  
  <div>  
    <h1>app works!</h1>  
  </div>  
</app-root>
```

# Style encapsulation

`import { Component } from '@angular/core';` Different options for style encapsulation

```
@Component({
  selector: 'app-root',
  templateUrl: 'app.component.html',
  encapsulation: ViewEncapsulation.ShadowDom,
  styles: [
    `h1{ color: red; }`,
    `p{ cursor: pointer; }`
  ])
export class AppComponent {
  title = 'app works!';
```

- **Emulated** (default) - styles from HTML document propagates to inside of components, but styles from component do not affect parent components - they are isolated.
- **ShadowDom** - styles are completely isolated using native *shadow-dom* (chrome only)
- **None** - all styles are global, component styles affect all elements in HTML document

# Nesting components

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

```
@Component({  
  selector: 'app-root',  
  template: '<div> <h1>Parent</h1>  
            <sub-component></sub-component>  
            </div>'  
})  
export class AppComponent { ... }
```

<app-root>

<div>

<h1>Parent!</h1>

<sub-component>

<h3>Child!</h3>

</sub-component>

</div>

</app-root>

# Bindings and directives

```
<p [style.backgroundColor]='lime'> I am lime! </p>
```

```
<p [style.fontSize.px]='big? 24 : 12'> {{ big? 'Huge' : 'small' }}</p>
```

```
<p [class.promotion]='true'> PROMOTION! </p>
```

```
<p class="promotion"> PROMOTION! </p>
```

```
<p [ngClass]='{ promotion: false, highlight:true }'> regular </p>
```

```
<p class="highlight"> regular </p>
```

... itd.

# Local References

Adding hash prefixed name ( `<div #somename ...>` ) to your component creates “local reference” - its variable that can point to DOM element object, or to component instance if used on component. It allows easy direct access to properties from template. Its accessible only from current template.

```
<video #movieplayer ...>...</video>
```

```
<button (click)="movieplayer.play()">
```



# Communication “Input” to the component

```
@Component ({
  selector: 'todo-input',
  template: '...'
})

export class Hello {
  @Input() item: MyTodoType;
}
```

```
// or:
@Component ({
  selector: 'todo-input',
  inputs: ['item'],
  template: '...'
})

export class Hello {
  item: MyTodoType;
}
```

```
// binding to variable (or expression value):
```

```
<todo-input [item]="myItem"></todo-input>
```

```
// binding to plain text value ( objects casted - toString() )
```

```
<todo-input item="Buy some milk!"></todo-input>
```



# Communication “Output” - EventEmitter

```
@Component ({  
  selector: 'todo-input',  
  template: '...'  
})
```

// or:

```
@Component ({  
  selector: 'todo-input',  
  outputs: ['completed'],  
  ....  
})
```

```
export class Hello {  
  @Output() completed = new EventEmitter<boolean>();  
}
```

```
<todo-input (completed)="saveProgress($event)" ></todo-input>
```

// alternatively you can use on-\* syntax

```
<todo-input on-completed="saveProgress($event)" ></todo-input>
```

# Two-Way data binding

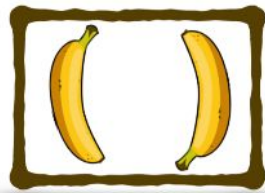
Directive **ngModel**:

- Binds input value to `name`: `<input [ngModel]="name">`
- Binds input change event to `name` value (**onChange**):

```
<input ngModel (ngModelChange)=" name = $event ">
```

- Or both:

```
<input [(ngModel)]="name">
```



`[(ngModel)]`

# Structural Directives

**ngIf** - Adds or Removes elements from template if expression evaluates to true

```
<ng-template [ngIf]="condition">  
  <div>{{ name }}</div>  
</ng-template>
```

Shorter syntax :

```
<div *ngIf="condition"> {{ name }} </div>
```

Prefixing with a asterisk “ \* “ auto-wraps given element in **<template></template>** for you!

# Structural Directives

Tools to attach and detach DOM fragments

```
<div *ngIf="completed"> ✓ </div>
```

```
<div *ngFor="let item of todoList; let i = index"> ... </div>
```

```
<div [ngSwitch]="item.status">  
  <p *ngSwitchCase="'completed'"> Completed <p>  
  <p *ngSwitchCase="'in-progress'"> Working on it <p>  
  <p *ngSwitchDefault="'in-progress'"> Working on it <p>  
</div>
```


# Content projection

- using **ngContent** directive you can project content from component parent to inside of component template on given spot
- optionally you can select part of content using CSS selector:

( `<ng-content select=".my-css-selector">` )

```
@Component ({  
  selector: 'user-profile',  
  template: `  
    <h4>User profile</h4>  
    <ng-content></ng-content>  
  `)  
})  
class Child {}
```

```
<user-profile>  
  <h5>Lito Rodriguez</h5>  
  <small>Actor</small>  
</user-profile>
```



# DOM interaction

Every directive and component can access native DOM Element using '**ElementRef**':

```
class MyDirective {  
  constructor(private ElementRef: ElementRef) {}  
  
  // Have to wait for DOM to be rendered:  
  ngAfterContentInit() {  
    const tmp = document.createElement( 'div' );  
    const el = this.ElementRef.nativeElement.cloneNode(true);  
    tmp.appendChild(el);  
  }  
}
```

Direct DOM manipulation should be avoided though ...

# ViewChild i Renderer

We can access local reference from our component code using `@ViewChild()`.  
Please note that element might not exist until **`ngAfterViewInit()`**

```
<input type="text" #myInput>
```

```
export class MyComp extends AfterViewInit {  
  @ViewChild('myInput') input: ElementRef;  
  
  constructor(private renderer: Renderer2) {}  
  
  ngAfterViewInit() {  
    this.renderer.invokeElementMethod(this.input.nativeElement,  
      'focus');  
  }  
}
```

# ViewChild, ViewChildren

We can also query for components using just their class

```
<todo-input todo="data" />  
<todo *ngFor="todo in todos"></todo>
```

```
export class MyComp extends AfterViewInit {  
  @ViewChild(TodoInputComponent) input: TodoInputComponent;  
  @ViewChildren(TodoComponent) todos: QueryList<TodoComponent>  
  
  ngAfterViewInit() {  
    this.todos.changes.subscribe((list) => console.log(list));  
  }  
}
```



# ContentChild, ContentChildren


Just like ViewChild i ViewChildren, but working on content that was projected with NgContent  
Elements might not be available until **ngAfterContentInit()** phase

**<ng-content></ng-content>**

```
export class MyComp extends AfterContentInit {  
  @ContentChild(TodoInputComponent) input: TodoInputComponent;  
  @ContentChildren(TodoComponent) todos: QueryList<TodoComponent>  
  
  ngAfterContentInit() {  
    this.todos.changes.subscribe((list) => console.log(list));  
  }  
}
```

# Lifecycle

Directives (**D**) and Components (**C**) can hook into lifecycle and respond to what is happening:



hook:	moment wystąpienia:
<b>ngOnChanges</b> ( <i>C</i> , <i>D</i> )	when a data-bound input property value changes
<b>ngOnInit</b> ( <i>C</i> , <i>D</i> )	after the first ngOnChanges
<b>ngDoCheck</b> ( <i>C</i> , <i>D</i> )	during every Angular change detection cycle
<b>ngAfterContentInit</b> ( <i>C</i> )	after projecting content into the component
<b>ngAfterContentChecked</b> ( <i>C</i> )	after every check of projected component content
<b>ngAfterViewInit</b> ( <i>C</i> )	after initializing the component's views and child views
<b>ngAfterViewChecked</b> ( <i>C</i> , <i>D</i> )	after every check of the component's views and child views
<b>ngOnDestroy</b> ( <i>C</i> , <i>D</i> )	just before Angular destroys the directive/component

# Custom directives

```
@Directive({
  selector: '[myDecorations]'
})
export class MyDecorationsDirective{
  constructor(private el: ElementRef, private renderer: Renderer) { }

  // We can listen events on element we are on (host element):
  @HostListener('onmouseenter') doMouseHighlight(){
    renderer.setStyle(
      el.nativeElement, 'backgroundColor', 'yellow');
  }
  // We can provide bindings to host element, even native ones:
  @HostBinding('style.text-decoration') decoration = 'underline';
}
```

# Custom structural directives

```
@Directive({ selector: '[myUnless]' })
export class UnlessDirective {
  constructor(
    private templateRef: TemplateRef<any>,
    private viewContainer: ViewContainerRef
  ) { }

  @Input() set myUnless(condition: boolean) {
    if (!condition) {
      // We can embed view created from template
      this.viewContainer.createEmbeddedView(this.templateRef);
    } else {
      // We have control, so we can change it, detach it, or remove
      it:
        his.viewContainer.clear();
    }
  }
}
```



# Services

and dependency injection in Angular

# Using services

Service is any instance of a class that can be injected into component and reused.

Services can:

- share usefull, common logic - i.e. talking to remote server,
- store data and provide it to multiple components,
- allow components to subscribe to data changes and push them updates,
- and generally every other things that is not directly connected with rendering ( view )

Avoid creating instances manually. Its always better to ask angular to provide them to you :

```
constructor ( private ourService: OurServiceClass,  
              private otherService: SomeOtherClass ) { }
```

Angular builds and provides services for you!



# HTTP communication

Using remote API REST and more with `@angular/http`

# @angular/http

```
import 'rxjs';

@Component()
class ContactsApp implements OnInit{
  contacts:Contact[] = [];
  constructor(private http: HttpClient

ngOnInit() {
  // Http.get() - creates . "cold stream" - you need to subscribe:
  this.http.get('/contacts')
  // Stream is 'cold' (or 'lazy'):
  // No request gets send until you need response - subscription:
  .subscribe(contacts => this.contacts = contacts);
```

Avoid using data sources (i.e: Http) directly from components! Components should be reusable and contain only view logic. Use Http in services.

Using Http inside component prevents you from reusing data in other components so you fetch it twice!



# Http Service

**Http** service from **HttpModule** have all usually used HTTP methods: **GET**, **POST**, **PUT**, **DELETE**:

```
let options = new RequestOptions({  
  headers: new Headers({ 'Content-Type': 'application/json' })  
})
```

```
this.http.post(url, payload, options)  
  .subscribe((response: Response) => {  
    let data = response.json();  
    console.log(data)  
  })
```

Remember:

using hierarhic providers you can setup all options once, by providing **DefaultRequestOptions**!



# Dependencies

Angular Dependency Injection


# Dependency Injection

Building objects manually inside our classes we tie them together. It's not practical:

```
this.item = new Todo()
```

You can separate them, by building objects elsewhere and injecting ready instances:

```
class Todo extends ListItem {}  
  
export class InjectorComponent {  
  item: ListItem = this.injector.get(ListItem);  
  
  constructor(private injector: Injector) { }
```

A diagram consisting of a thin black arrow pointing from the `ListItem` parameter in the `get` method call (`this.injector.get(ListItem)`) within the `InjectorComponent` class to the `ListItem` class definition (`class Todo extends ListItem {}`) above it.

# Configuring Providers

We don't to have create injectors! - by doing `bootstrapModule(MyAppModule)` angular creates its own injectors that we can use.

OK, but how do i configure **what angular should inject???**

All you need to do for any class you want angular to build and inject for you is to add every of them into providers: [ ] in your module definition:

```
@NgModule ({  
  // ...  
  exports: [ SomeClassName ],  
  providers: [ SomeClassName ],  
})  
  
export class AppModule { }
```

Class you have defined in module will be available for all components, services and submodules

Class won't be available in parent modules, unless you provide it there instead.

# Dependency Injection

What if our class requires its own dependencies? No worries - Angular can build whole object graph

You just need to decorate all classes you want to be injected with a **@Injectable**:

```
@Injectable()
```

```
export class OurClass {  
  constructor(@Inject(Logger) logger) { }
```

```
// You can just use type annotation here, it will be injected also:
```

```
constructor(private logger: Logger) { }
```

Decorators like **@Component**, **@Directive**, **@Pipe**, already extend **@Injectable()**

# Switching implementations

When we want to separate concrete implementations from abstractions we can do that, and inject different implementations of same class using providers. We have multiple options:

```
@NgModule ({  
  // ...  
  providers: [  
    SomeClass, // Provide SomeClass, every time SomeClass is injected..  
    // Another class  
    { provide: Type, useClass: Class },  
    // Ready instance of object (i.e. config..  
    { provide: Type, useValue: SomeValue },  
    // Fabricating function, returns new values (deps:[...])  
    { provide: Type, useFactory: Function, deps: [SomeDependency]},  
  ],  
})
```

# Tokens

It may happen that you want to inject by name... Don't use strings, but use **OpaqueToken()** - its unique symbol, so even if somebody else uses symbol with exact same name as you they will never collide, because symbol is a unique reference, not a primitive value like string.

```
const URL_TOKEN = new OpaqueToken('ServerURLToken');

@NgModule ({
  providers: [
    { provide: URL_TOKEN, useValue: 'http://example.com/api/v1/ ' },
    { provide: ApiService, useFactory: (@Inject(URL_TOKEN) url) => {
      return new ApiService(url);
    }
  ],
```

A diagram consisting of a thin black arrow pointing from the `URL_TOKEN` variable in the first line of code to the `URL_TOKEN` token used within the `providers` array in the second code block.

# Hierarchical Dependency Injector

By default every instance is a **singleton** - angular creates instance once and then reuses it - same object is injected everywhere it is needed.

If we want object with shorter life we provide it inside **@Component**

```
@Component({  
  selector: 'isolated-data-view',  
  providers: [ OurClass ]  
})  
export class AppComponent { }
```

Let's assume there is already **OurClass** defined in our module.

But now **Component also provides one**. This way component and its children share their own instance of **OwnClass** that other services have no access to. Instance will be destroyed with component that provided it.

Providers are looked up bottom-up. If we don't have provider it's looked up in parent component, its parent.. etc. then in module, parent module, etc...





# Reactive programming

Thanks to EventEmitter and RxJs extensions  
we can work with streams of values to make our app reactive

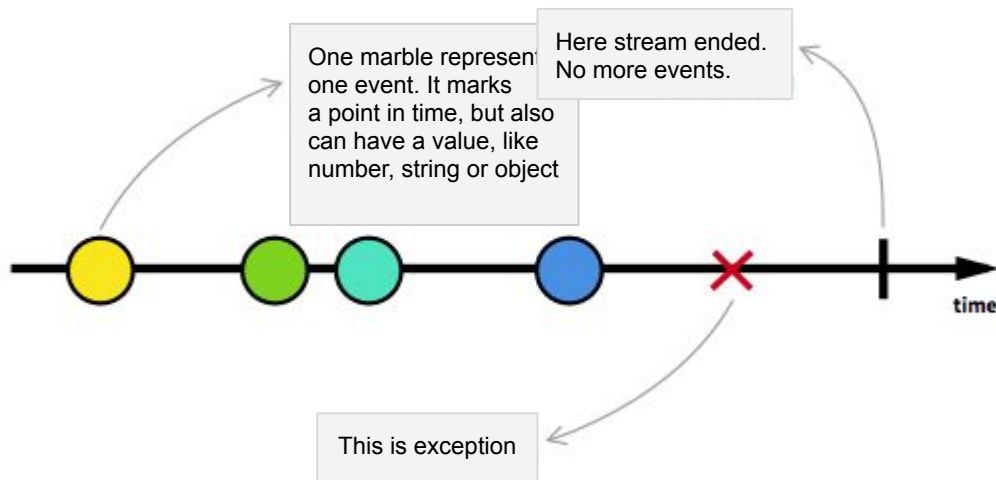
# RxJS library



RxJS - Reactive Extensions Library - allows working on “lists of future events”  
Every `EventEmitter<T>()` is automatically rxjs Observable compatible.

Its best to  
understand  
operators using  
**marble**  
**diagrams:**

( marbles )



Events can be everything: MouseClicks, KeyPresses, Clock Ticks or asynchronous server response...

**Unlike Promises Observables can be reused many times with new events**

# RxJS - reactive programming

rxjs provides objects that can be observed, transformed and then subscribed to in order to be notified what changed and when.

RxJS comes with very rich library of operators that allows you to chain data transformations

There are groups of similar operators, for example:

- transforming (np. delay, map, debounce, scan)
  - joining (np. merge, sample, startWith, zip)
  - filtering (np. distinctUntilChanged, filter, skip)
  - and a lot... lot more
- 
- Every operator returns back Observable => Therefore they can be easily chained (*like Promise*)
  - Its easy to check how different operators work using so called "marble diagrams"

**Some interactive diagrams :** <http://rxmarbles.com/>

# RxJS in Angular 2

Angular 2 is compatible with RxJS Observables. If rxjs is available, built-in objects are extended:

- **EventEmitter** becomes observable
- API module **Http** are observables
- Data-driven forms controller can be subscribed to using `form.valueChanges` observable
- you can easily subscribe to any observable in your templates using **AsyncPipe**:

```
<div *ngFor="let item of todoListStream$ | async"> ... </div>
```

You can also create your own observables using:

- **EventEmitter / Observable** - to emit your own changes
- **Subject** - to “pass on” observables, for example connecting multiple services

# Importing single operators

```
// import 'rxjs/Rx'; // Importing everything can seriously increase total size of application files
```

```
// Its recommended to import only parts of RxJs that you use
```

```
import {  
    map,  
    catch,  
    debounceTime,  
    distinctUntilChanged'  
} from 'rxjs/operators/';
```

```
someObservable.pipe(  
    map( ... ), debounceTime(1000), distinctUntilChanged(), ...  
)
```

If you forget to import operator and try to use it somewhere - don't worry - TypeScript will detect that and tell you that you are trying to use method that does not exist. :-)



# Forms

Template Forms and Data-Driven Forms

# Data driven forms

```
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
// Import Reactive forms module for data-driven or FormsModule if else

import { FormControl, FormGroup, Validators } from '@angular/forms';
/* @Component ... */
export class MyForm {
  username: FormControl;
  constructor(private builder: FormBuilder ) {
    this.username = new FormControl('default field value', [
      Validators.required, Validators.minLength(3)
    ]);
    this.regForm = new FormGroup({
      username: this.username
    });
    /* or this.builder.group({ username: [...] }) */
    // this.regForm.value == {username: 'Johnny'}
```

# Connecting form with view

**FormGroup** object communicates with form **FormGroup** directive. Then we can map each **FormControl** to its input using **FormControlName** attribute:

```
<form [formGroup]="regForm" (ngSubmit)="saveUser(regForm)">
  <input name="username" formControlName="username" />
  <button type="submit">Save</button>
</form>
```

Thanks to `formControlName` we can map inputs to their corresponding `FormControls`:

```
this.regForm.get("username").value
```

Use **(ngSubmit)** instead of regular **(submit)** event to make sure all validations are first completed and form controller has now correct state



# Form states

- Formularz i jego pola mogą być w kilku stanach stanach:

state:	meaning:
<code>pristine</code>	fields value wasn't changed
<code>dirty</code>	field value was modified (even if now has same value)
<code>touched</code>	field was visited and left (blur event)
<code>valid</code>	no validator returned any error
<code>submitted</code>	form submission was attempted

# Visual form feedback

Form controls also gets automatic CSS classes decoration for easy visual feedback

class:	flags:
<code>ng-pristine</code>	<code>pristine = true i dirty = false</code>
<code>ng-dirty</code>	<code>pristine = false i dirty = true</code>
<code>ng-touched</code>	<code>touched = true</code>
<code>ng-valid</code>	<code>valid = true</code>
<code>ng-invalid</code>	<code>valid = false</code>

You can just add proper CSS classes:

```
input.ng-invalid.ng-dirty {  
  border-bottom-color: red;  
}
```

# Your own validation

Validator is a function (ValidatorFn) or class (Validator) that takes in form control (**AbstractControl**) and returning null for valid field or a map of error tokens

```
function startsWithLetter(control: Control): {[key: string]: any} {  
    let pattern: RegExp = /^[a-zA-Z]/;  
  
    return pattern.test(control.value) ? null : {  
        'startsWithLetter': true  
    };  
}
```



Here form.controls.somefield.**errors.startsWithLetter** will evaluate to true if field value matches

# Template Forms

You can define your complete form just using a template:

```
<form #myForm="ngForm" (ngSubmit)="save(myForm)">
  <input name="comment" [(ngModel)]="item.comment"
    required minlength="20" #comment="ngModel" />
  <span *ngIf="comment.dirty && comment.errors.required">
    This field is required!
  </span>
  <input type="submit" value="Save">
</form>
```

You can access form controller using local reference. Adding directive name (**ngForm**) you get access not to element, but directly to that directives exported APIs:

```
#nazwa="ngForm" (ngSubmit)="myMethod(nazwa)"
```



# Pipes

# Transforming data with Pipes

Pipe is kind of filter ( think unix pipes ) that transforms data in out views

```
<p> Total:  {{ items.total | currency }} </p>
```

It can take multiple parameters - both static values and dynamic template variables

```
<p> Total:  {{ items.total | currency:'PLN':true }} </p>
```

You can chain Pipes to pass on transformed value to next one

```
<p> Score:  {{ player.score | number | replace:'0':'-' }} </p>
```

Build-in Pipes ( in sub-modules CommonModule must be imported ):

DatePipe, UpperCasePipe, LowerCasePipe, CurrencyPipe, PercentPipe, JsonPipe

# Filter parameters

Pipe is a class implementing **PipeTransform** and decorated with **@Pipe()**

```
import { Pipe, PipeTransform } from '@angular/core';
@Pipe({
  name: 'censor'
})
export class CensorPipe implements PipeTransform {
  transform(input: string, character ? : string): string {
    return input.replace(/./g, character || '*');
  }
}
```

# Filters - usage

If we want to use our custom Pipe we need to add it to module **declarations** and then we can use it in modules components as usual:

```
import { Component } from '@angular/core';
import { CensorPipe } from './censor.pipe';
@Component({
  selector: 'my-censor',
  template: '<span>{{ message | censor }}</span>',
})
export class Hello {
  message: string = 'My secret sentence';
}
```



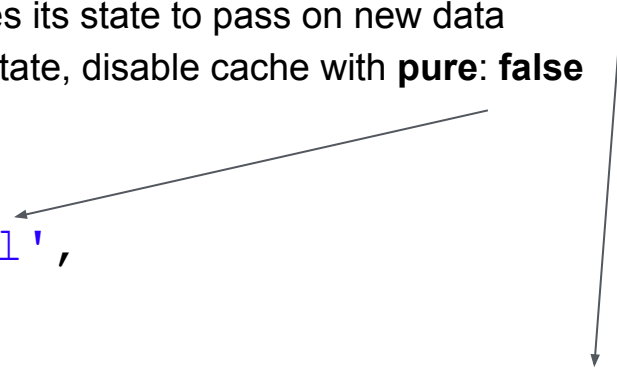
# State and async

## in filters

- Most of pipes don't work with state. They use functions without side-effects, they just transform and return data. This allows angular to cache their values based on params.
- Stateful pipes (ie. **AsyncPipe**) holds state of data and can change even if params don't
- **AsyncPipe** takes **Promise** or **Observable** as input and subscribes to it. When there is a value async pipe updates its state to pass on new data
- If your pipe works with state, disable cache with **pure: false**

```
@Pipe ({  
  name: 'myStateful',  
  pure: false  
})
```

```
<div *ngFor="item in $stream | async">{{item}}</div>
```





# Routing

Angular Component Router

# Routing configuration

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

const routingModule = RouterModule.forRoot([

  { path: 'todos', component: TodosComponent, children: [] },

  { path: '', component: HomeComponent },

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

]);
```

```
@NgModule({

  imports: [

    BrowserModule,

    routingModule,

    ...
```

Router maps given segment of matched URL to component.  
First matched route is activated.

Selected component will appear in router-outlet directives.

**<router-outlet></router-outlet>**

You can nest routes by segments using *children* path option.  
Those should be nested to match paths if those are nested

# Routing for sub-modules

```
@NgModule({  
  imports: [  
    RouterModule.forChild([  
      { path: 'heroes', component: HeroListComponent },  
      { path: 'hero/:id', component: HeroDetailComponent }  
    ])  
  ],  
  exports: [  
    RouterModule  
  ]  
})
```

**RouterModule.forRoot(routes, options)** - creates main routing for main module

**RouterModule.forChild(routes)** - Allows us to configure additional routes for every module inside of that module.

Methods of RoutingModule builds **ModuleWithProviders** object that can be imported as regular module

# Passing in parameters

```
{ path: todo/:id, component: TodoDetailComponent }
```



```
http://localhost:3000/todo/15
```

Paths can be parametrised. Injecting **ActivatedRoute**, **Params**, **etc..** we can access current state, url, query params or params

```
import { Router, ActivatedRoute, Params } from '@angular/router';
```

```
constructor(
```

```
  private route: ActivatedRoute,
```

```
  private router: Router,
```

```
  private service: TodosService ) {}
```

```
// Also Router service gives us the API for programmatic navigation :
```

```
onSelect(todo: Todo) { this.router.navigate(['/todo', todo.id]);
```

# Linking to router paths

**RouterLink** directive activates given router path when element is clicked. Path can be simple string, or array of path segments. Its **relative** unless prefixed with `'/'`

```
<nav>
  <a routerLink="/todos" routerLinkActive="active">Todos App</a>

  <a [routerLink]="['/todo', todo.id ]" routerLinkActive="favourite-active">
    My favourite Todo
  </a>
</nav>
```

**RouterLinkActive** its helper parameter that when used with **RouterLink** automatically toggles given CSS class on path activation / deactivation.



# Unit Testing

different parts of the framework

# Testing simple things

like filters, services, etc...

```
describe ('TitleCasePipe', () => {  
  // Pipe here is simple class, no dependencies, no  
  // special initialisation  
  let pipe = new TitleCasePipe();  
  
  it ('transforms "abc" to "Abc"', () => {  
    expect(pipe.transform('abc')).toBe('Abc');  
  });  
});
```



# Testing Components

Should component require any dependencies we can fake a module. We can also substitute any dependencies we need to:

```
beforeEach(() => {  
  TestBed.configureTestingModule({  
    declarations: [ BannerComponent ],  
    imports: [ BrowserModule ]  
  });  
  
  // Creates component and wraps it in a "fixture" ( ComponentFixture )  
  fixture = TestBed.createComponent(BannerComponent);  
  
  // We can now access component instance, its data and methods:  
  comp = fixture.componentInstance;
```

# Testing view elements

If we are to test not only the class of the component, but also DOM generated from its template we can't forget about change detection:

```
it('should display original title' , () => {  
  
    // After changing any data call change detection to update DOM:  
    fixture.detectChanges();  
  
    // DebugElement object has usefull tools to work with DOM:  
    de = fixture.debugElement.query(By.css('h1'));  
  
    // Now you can compare if your data was rendered properly:  
    expect(de.nativeElement.textContent).toContain(comp.title);  
});
```



**Thank you!**

**Any questions?**