

Zarządzanie pakietami



Node Package
Manager

- system zarządzania zależnościami dla server-side js
- zależności opisywane z dokładnością do wersji w pliku `package.json`
- `npm install` - instaluje pakiety, których jeszcze nie ma w projekcie
- `npm update` - sprawdza, czy istnieją nowsze wersje pakietów + instaluje
- `npm install nazwa-pakietu --save-dev` - instaluje pakiet, dodaje go do `package.json`



WebPack

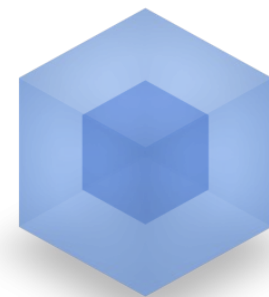
- module bundler
- obsługuje wiele formatów modułów: ES2015, AMD, CommonJS (npm)
- traktuje wszystko jak moduły (np. scss, html, grafiki)
- React Hot Loader • dobrze współpracuje z popularnymi task runnerami (Gulp, Grunt)
- de facto standard w środowisku React, popularny również poza

```
// instalacja webpack-cli, globalnie npm
```

```
install webpack --global
```

```
// oraz instalacja lokalna
```

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



webpack
MODULE BUNDLER



Webpack - Konfiguracja

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



extensiblewebmanifesto.org

#extendthewebforward

Zestaw niskopoziomowych API i standardów pozwalających:

- **Rozszerzać** funkcjonalność przeglądarki
 - bez instalowania dodatków
 - bez “hacków”
- **Modyfikować** działanie istniejących funkcji
 - lokalnie w obrębie danej aplikacji (strony / domeny)
 - bezpiecznie

Używając tylko JavaScript!

Możliwe jest rozszerzanie istniejących standardów o nowe wysokopoziomowe API.

Przykłady: Mozilla **X-Tags**, Google **Polymer**... oraz **Angular 2**



JavaScript 2015

“JavaScript next”



~~ECMAScript 6?~~ ECMAScript 2015

- moduły
- dużo dobrego „cukru składniowego”
- leksykalny zasięg (let) i wiele innych
- można używać...

już! **BABEL**

<https://babeljs.io/>



Funkcje Anonimiwe

(Lambda)

// Domyślnie - zwraca wyrażenie

```
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}  
));
```

// Deklaracje umieszczamy w klamrach

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

// Leksykalne this

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



Destrukturyzacja

```
// 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) {  
  // domyślna wartość y (jeśli y === undefined)  
  return x + y;  
}  
f(3) === 15;  
function f(x, ...y) {  
  // y jest tablicą pozostałych wartości  
  return x * y.length;  
}  
f(3, "hello", true) === 6;  
function f(x, y, z) {  
  return x + y + z;  
}  
// przekazanie każdego elementu tablicy osobno  
f(...[1, 2, 3]) === 6;
```



Dynamiczny Literal

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



Obietnice (Promise)

```
function timeout(duration = 0) {  
    return new Promise((resolve, reject) => {  
        setTimeout(resolve, duration);  
    })  
}  
  
var p = timeout(1000).then(() => {  
    return timeout(2000);  
}).then(() => {  
    throw new Error("hmm");  
}).catch(err => {  
    return Promise.all([timeout(100), timeout(200)]);  
})
```



Testowanie Jednostkowe

Aplikacji JavaScript`owych



KARMA- test runner

- Narzędzie do uruchamiania testów automatycznych
- Udostępnia korzystanie z środowisk wielu przeglądarek
- Umożliwia korzystanie z dowolnych frameworków testujących

```
npm install karma-cli -g  
karma  
karma-jasmine  
karma-mocha  
karma-chrome-launcher  
karma-phantomjs-launcher
```

...

```
karma init       karma start
```



Tworzenie Test suites, tests

```
describe('calculator', function() {  
  describe('add()', function() {  
    it('should add 2 numbers together', function() {  
      // TUTAJ RÓŻNICE MIĘDZY FRAMEWORKAMI  
    });  
  });  
});
```



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

- poszczególne testy znajdują się w bloku **it()**
- testy grupowane są zestawy w blokach **describe()**
- nazwy takich zestawów + testów powinny tworzyć zdania - dzięki czemu otrzymujemy:

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

Frameworki testujące

```
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);
```

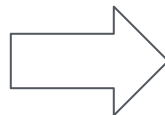


Analiza Statyczna z ESLint

Następca JSLint'a

```
// instalacja  
npm install -g eslint
```

```
// utworzenie konfiguracji  
eslint --init
```



```
// .eslintrc.*
```

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

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

```
// uruchomienie  
eslint testfile.js otherfile2.js ...
```




Angular 2

2.4+, 4.0, 5.0, ... lub po prostu Angular



Narzędzie ng-cli

Generator uruchamiany z wiersza poleceń

/ > ng



Narzędzia “ng” - angular-cli

Instalacja:

npm install -g @angular/cli

Stworzenie nowego projektu angular 2 w b. katalogu

ng new *nazwa projektu*

Uruchomienie lokalnego serwera developerskiego

ng serve

Wygenerowanie szkieletu komponentu (lub innej klasy...):

ng generate component <name>



Narzędzia “ng” - angular-cli

`ng generate component Nazwa` - tworzy nowy komponent!

flagi dodatkowe:

- `--flat` - nie tworzy nowego katalogu dla komponentu
- `-t` - inline template - szablon umieszcza w pliku komponentu
- `-s` - inline styles - style umieszcza w pliku komponentu
- `--spec false` - pomija generowanie testów jednostkowych komponentu



Bootstrap

Startowanie Aplikacji


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

Inicjalizacja głównego modułu (z wybranym głównym komponentem):

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

Element głównego komponentu musi oczywiście znajdować się w dokumencie HTML, np.

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



Moduły w Ng2 = @NgModule

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

@NgModule({
  // Importujemy " exporty " z innych modułów:
  imports: [ BrowserModule ],
  // Deklaracje dla parsera HTML ( Dyrektywy i komponenty ):
  declarations: [ AppComponent ],
  // Który komponent będzie "głównym" komponentem aplikacji:
  bootstrap:    [ AppComponent ],
  // Udostępnione elementy oraz definicje usług ( o tym później... )
  exports: [], providers: [],
})
export class MyAppModule { }
```



Komponenty

Podstawowy “Budulec” aplikacji w Ng2



Definicja komponentu

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

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

Selektor to prosty selektor CSS, który określa na jakich elementach HTML parser ma zamontować ten komponent

Na przykład podpinanie do:

[my-attribute] - atrybutu

.spiner-loader - klasy CSS

app-root - elementu

<app-root></app-root>




Szablony

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

```
@Component({  
  selector: 'app-root',  
  template: `

<h1>{{title}}</h1>  
  </div>`  
})  
export class AppComponent {  
  title = 'app works!';  
}


```



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

Enkapsulacja Styli

`import { Component } from '@angular/core';` Enkapsulacja może przyjąć wartości:

```
@Component({
  selector: 'app-root',
  templateUrl: 'app.component.html',
  encapsulation: ViewEncapsulation.Native,
  styles: [
    `h1{ color: red; }`,
    `p{ cursor: pointer; }`
  ])
```

```
export class AppComponent {
  title = 'app works!';
```


- **Emulated** (domyślnie) - style z dokumentu HTML propagowane są do komponentu i jego dzieci. Style zdefiniowane w `@Component({ styles / styleUrls })` nie wpływają na resztę dokumentu - są izolowane
- **Native** - Style dokumentu nie wpływają na elementy w komponencie, ani vice versa - pełna izolacja (patrz. *shadow-dom*)
- **None** - style są propagowane do dokumentu HTML, więc są dostępne dla pozostałych komponentów

Zagnieżdzone komponenty

```
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>
```





Wiązania właściwości i dyrektywy atrybutów

```
<p [style.background-color]='lime'> Jestem limonkowy! </p>
```

```
<p [style.font-size.px]="big? 24 : 12"> {{ big? 'duży' : 'mały' }}</p>
```

```
<p [class.promotion]="true"> Tu Promocja! </p>
```

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

```
<p [ngClass]="{ promotion: false, highlight:true }"> Tu nie... </p>
```

```
<p class="highlight"> Tu nie.. </p>
```

... itd.



Lokalne Referencje

Umieszczenie znaku hash z nazwą ("**#nazwa**") na elemencie tworzy "lokalną referencję" - zmienną dostępną tylko w tym komponencie, która zawiera odniesienie do elementu DOM (lub komponentu) :

```
<video #movieplayer ...>...</video>  
      |  
<button (click)="movieplayer.play()">
```

Komunikacja “do” komponentu - Właściwości

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

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

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

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

```
// przekazywanie przez zmienną ( jako wyrażenie ):
<todo-input [item]="myItem"></todo-input>
```

```
// przekazywanie wartości bezpośrednio ( jako String )
<todo-input item="Kupić chleb"></todo-input>
```



Komunikacja “z” komponentu - emiter zdarzeń

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

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

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

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

```
// alternatywna składnia - dlatego nie prefixujemy zdarzeń "on"
```

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

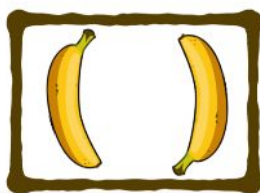
Komunikacja z komponentem w obie strony

Dyrektywa **ngModel**:

- Wiąże dane ze zmienną: `<input [ngModel]="name">`
- Wiąże nasłuchiwanie zdarzeń zmian (**onChange**) :

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

- Lub w skrócie: `<input [(ngModel)]="name">`



`[(ngModel)]`



Dyrektywy strukturalne

ngIf - Dodaje i usuwa element szablonu (template) jeśli wyrażenie jest prawdziwe

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

Wersja skrócona:

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

Oznaczenie “ * ” przed dyrektywą tworzy szablon <template></template> za nas !



Dyrektywy strukturalne

Pozwalają modyfikować strukturę DOM

```
<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> Working on it <p>  
</div>
```

Projekcja zawartości

- za pomocą dyrektywy **ngContent** możemy umieścić w szablonie komponentu html wpisany jako zawartość komponentu
- opcjonalnie można zdefiniować miejsce dla elementu z określonego selektora :
(`<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>
```





Interakcja z DOM

Każda dyrektywa (i Komponent) mają dostęp do natywnego elementu poprzez obiekt klasy **'ElementRef'**:

```
class MyDirective {  
  constructor(private elementRef: ElementRef) {}  
  
  // Musimy poczekać na podłączenie dyrektywy z DOM:  
  ngAfterContentInit() {  
    const tmp = document.createElement('div');  
    const el = this.elementRef.nativeElement.cloneNode(true);  
    tmp.appendChild(el);  
  }  
}
```

Nie jest to jednak zalecane...



ViewChild i Renderer

Do referencji lokalnej mamy dostęp dzięki `@ViewChild()`.
Wartości dostępne są jednak dopiero w **`ngAfterViewInit()`**

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

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



ViewChild, ViewChildren

Możemy też wyszukiwać sub-komponenty w naszym szablonie po ich typie:

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

```
export class MyComp implements AfterViewInit {
  @ViewChild(TodoInputComponent) input: TodoInputComponent;
  @ViewChildren(TodoComponent) todos: QueryList<TodoComponent>

  ngAfterViewInit() {
    this.todos.changes.subscribe((list) => console.log(list));
  }
}
```



ContentChild, ContentChildren

Odpowiedniki ViewChild i ViewChildren, ale działające na zawartości z Projekcji
Wartości dostępne są dopiero w **ngAfterContentInit()**

`<ng-content></ng-content>`


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



Cykl życia

Dyrektywy (**D**) i komponenty (**C**) mogą “wpiąć” się w cykl renderowania aby wykonać własne funkcje:

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





Własne dyrektywy wiążące

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

  // Nasłuchujemy zdarzeń na elemencie hosta:
  @HostListener('mouseenter') onMouseHighlight(){
    renderer.setStyle(
      el.nativeElement, 'backgroundColor', 'yellow');
  }
  // Podstawiamy własne wartości do właściwości elementu hosta:
  @HostBinding('style.textDecoration') decoration = 'underline';
}
```



Własne dyrektywy strukturalne

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

  @Input() set myUnless(condition: boolean) {
    if (!condition) {
      // Umieszczamy element z szablonu w widoku
      this.viewContainer.createEmbeddedView(this.templateRef);
    } else {
      // Usuwamy stworzone elementy:
      this.viewContainer.clear();
    }
  }
}
```



Usługi

i wstrzykiwanie zależności w Angular



Korzystanie z usług

Usługą jest każda instancja klasy, którą możemy przekazać do Komponentu, lub dyrektywy.

Usługi mogą:

- udostępniać przydatną logikę - np. komunikacja z serwerem
- przechowywać dane i współdzielić je pomiędzy komponentami
- informować komponenty o zdarzeniach i przekazywać potrzebne informacje
- i generalnie wszystkie inne zadania niezwiązane bezpośrednio z wyświetlaniem (Widokiem)

Nie tworzymy usług samodzielnie. Prosimy angulara o ich dostarczenie:

```
constructor( private nazwa_zmiennej: KlasaUsługi,  
             private inna: TypowanaUsługa ) {}
```

Angular zajmie się dostarczeniem odpowiedniego obiektu!



Komunikacja HTTP

Komunikacja z API REST`owym i nie tylko z wykorzystaniem
modułu angular/https



Moduł angular/http

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

  ngOnInit() {
    // Http.get() - tworzy tzw. "zimny strumień":
    this.http.get('/contacts')
    // Leniwe zapytania:
    // Zapytanie do serwera wykona się dopiero przy subskrypcji:
    .subscribe(contacts => this.contacts = contacts);
```

Nie używaj źródeł danych (m.in. Http) bezpośrednio w komponentach! Komponenty powinny być proste i zawierać tylko logikę widoku.

Takie powiązanie z Http uniemożliwia wymianę danych z innymi komponentami - **Używaj usług!**



Metody i konfiguracja http

Klasa **Http** posiada metody odpowiadające metodom HTTP: **GET**, **POST**, **PUT**, **DELETE**:

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

this.http.post(url, payload, options)
  .subscribe((data) => {
    console.log(data)
  })
```

Jeśli chcesz zmienić opcje dla całego modułu stwórz provider dla `HttpInterceptor`



Wstrzykiwanie zależności

Angular Dependency Injection



Injektor zależności

Tworząc zależności wewnątrz naszych klas "Łączymy na sztywno" implementacje:

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

Lepiej jest wiązać klasy przy użyciu abstrakcji, a implementacje wstrzyknąć:

```
class Todo extends ListItem {}
```

```
export class InjectorComponent {  
  item: ListItem = this.injector.get(ListItem);
```

```
  constructor(private injector: Injector) { }
```



Konfiguracja Providerów

Nie musimy ręcznie tworzyć iniektorów! - już samo `bootstrapModule(MyAppModule)` tworzy za nas automatycznie globalny Injector.

OK, tylko jak wskazać angularowi **której implementacji ma użyć ???**

Domyślnie **wystarczy zarejestrować klasę**, by injector sam zbudował dla nas obiekt tej klasy wraz z zależnościami:

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

Klasa w sekcji **providers** będzie wstrzykiwana wszędzie gdzie użyto jej typu w tym module i w **jego dzieciach**

Klasa dodana w sekcji **exports** będzie wstrzykiwana **globalnie**



Automatyczne wstrzykiwanie

Co gdy nasza instancja wymaga innych instancji? Injektor zbuduje za nas cały graf:

Wystarczy dodać dekorator **@Injectable**:

@Injectable()

```
export class Nasza Klasa {  
  constructor(@Inject(Logger) logger) { }  
  
  // lub wystarczy klasa/typ - auto-wstrzykiwanie po typie:  
  constructor(private logger: Logger) { }
```

Inne dekoratory **@Component**, **@Directive**, **@Pipe**, itd. rozszerzają **@Injectable()**



Podmiana Implementacji

Gdy chcemy **oddzielić implementacje od interfejsu i wstrzyknąć inny obiekt**, mamy kilka możliwości:

```
@NgModule ({  
  // ...  
  providers: [  
    NazwaKlasy, // Ta sama implementacja...  
    // Inna klasa  
    { provide: Typ, useClass: Klasa },  
    // Gotowa instancja, prosty obiekt, lub nawet wartość (np.  
config..)  
    { provide: Typ, useValue: Obiekt },  
    // Funkcja fabrykująca - może mieć własne zależności (deps:[...])  
    { provide: Typ, useFactory: Funkcja, deps: [KlasaDlaFunkcji]},  
  ],  
})
```



Tokeny

Może się zdażyć, że wygodniejsze będzie użycie nazwy... Nie używamy stringów, ale **InjectionToken()** - jest to unikalny symbol, dzięki czemu w przeciwieństwie do ciągów znaków unikamy ryzyka wystąpienia kolizji nazw!

```
const URL_TOKEN = InjectionToken('ServerURLToken');

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





Hierarchiczny Injector

Domyślnie, każda stworzona instancja jest singletonem - angular tworzy obiekt raz, a następnie przekazuje zawsze globalnie tą samą instancję.

Jeśli chcemy stworzyć lokalną instancję, definiujemy ją w **@Component**

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

W module jest już zarejestrowana klasa "NaszaKlasa" i istnieje jej instancja...

Jednak ten komponent **posiada teraz własny injektor**, więc ten komponent i jego dzieci otrzymają **nową instancję klasy NaszaKlasa niedostępną dla reszty komponentów**. Jeśli klasa nie jest zdefiniowana tutaj, będzie użyty injektor nadrzędny, nadrzędny, itd.. aż po globalny injector.



Programowanie Reaktywne

Dzięki EventEmitter oraz rozszerzeniu Rx.JS możemy
pracować na reaktywnych strumieniach

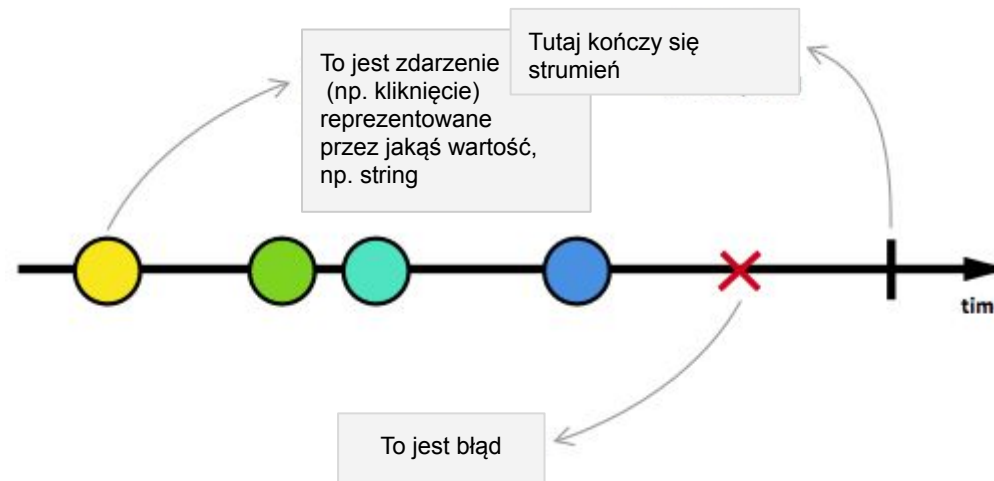
RxJS



RxJS - czyli Reactive Extensions - pozwalają na pracę na “strumieniach zdarzeń”
Każdy obiekt typu `EventEmitter<T>()` jest automatycznie rozszerzany.

O strumieniach
najprościej
rozumować
korzystając z
**diagramów
kulkowych:**

(marbles)



Zdarzeniami mogą być dowolne dane: Kliknięcia myszy, znaki z klawiatury, tyknięcie zegara lub asynchroniczna odpowiedź z serwera ...

Zaletą strumieni jest możliwość ich wielokrotnego użycia

*Źródło: "Introduction to Reactive Programming you've been missing" by [Andre Staltz](#)



RxJS

Strumienie możemy obserwować, przekształcać a następnie subskrybować w celu otrzymania informacji, kiedy nastąpiła zmiana.

RxJS posiada bogatą bibliotekę operatorów umożliwiającą przekształcenia

Operatory możemy podzielić na:

- przekształcające (np. delay, map, debounce, scan)
 - łączące (np. merge, sample, startWith, zip)
 - filtrujące (np. distinctUntilChanged, filter, skip)
 - i wiele innych
- Wynik działania operatora to strumień => można je łączyć
 - Sposób działania poszczególnych operatorów obrazuje się za pomocą "marble diagrams"
Przykładowe Diagramy na : <http://rxmarbles.com/>



RxJS w Angular 2

Angular 2 używa RxJS w kilku obszarach:

- **EventEmitter**
- API modułu **Http**
- Zmiany wartości formularzy to też strumienie
- Łatwa możliwość subskrypcji z poziomu widoku dzięki **AsyncPipe**

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

Możemy też tworzyć własne strumienie korzystając z obiektów:

- **EventEmitter / Observable** - do emitowania własnych zdarzeń
- **Subject** - do “przekazywania” strumieni pomiędzy usługami i komponentami



Formularze

Template Forms oraz Data-Driven Forms



Formularz jako usługa

```
import { FormsModule } from '@angular/forms';
/* Importujemy Forms Module do Modułu Aplikacji ... */

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



Łączenie z widokiem

Element **FormGroup** łączymy z formularzem dyrektywą **FormGroup**, natomiast obiekt **FormControl** dyrektywą **FormControlName**:

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

Dzięki podaniu `formControlName` możemy odwoływać się do pól formularza po tej nazwie:

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



Stany formularza

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

stan:	znaczenie:
pristine	pole nie było modyfikowane
dirty	pole było modyfikowane
touched	pole zostało opuszczone (blur)
valid	żaden walidator nie zwrócił błędu
submitted	formularz został wysłany



Klasy CSS w formularzach

- Elementy formularza otrzymują również odpowiednie klasy:

klasa:	znaczenie:
ng-pristine	pristine = true i dirty = false
ng-dirty	pristine = false i dirty = true
ng-touched	touched = true
ng-valid	valid = true
ng-invalid	valid = false

Możemy je więc wykorzystać w CSS:

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



Własne metody walidacji

Validator to funkcja przyjmująca jako parametr instancję pola (**Control**) i zwracającą obiekt z kluczami będącymi **kodami błędów** oraz wartościami **boolean** jeśli dana wartość jest błędna.

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




Formularz w Szablonie

Formularz możemy też stworzyć bezpośrednio w szablonie:

```
<form #myForm="ngForm" (ngSubmit)="save(myForm)">
  <input name="comment" [(ngModel)]="item.comment"
    required minlength="20" #comment="ngModel" />
  <span *ngIf="comment.dirty && comment.hasError('required')">
    Pole jest wymagane!
  </span>
  <input type="submit" value="Zapisz">
</form>
```

Obiekt kontrolujący formularz możemy uzyskać korzystając z lokalnej referencji

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



Filtery



Transformacja danych z Pipe

- Pipe to filtr, który przekształca przekazane do niego dane :

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

- Można go skonfigurować przekazując parametry:

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

- Wynik działania jednego pipe można przekazać do kolejnego

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

Wbudowane:

DatePipe, UpperCasePipe, LowerCasePipe, CurrencyPipe, PercentPipe, JsonPipe



Parametryzowane Filtry

Pipe to klasa implementująca interfejs PipeTransform i opisana dekoratorem @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 || '*');
  }
}
```



Filtry - Użycie

Aby użyć custom pipe w komponencie, należy go zadeklarować (podobnie jak robimy to z dyrektywami)

```
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';
}
```



Stan i asynchroniczność

w filtrach

- Większość pipes jest bezstanowe. Wykonują przekształcenia za pomocą czystych funkcji, bez efektów ubocznych. Działają w szablonie jak “cache” dynamicznej wartości.
- Stateful pipes (np. **AsyncPipe**) zarządzają stanem przekazywanych danych
- **AsyncPipe** przyjmuje jako wejście **Promise** lub **Observable** i przechowuje subskrypcję, żeby później zwrócić wartość
- Definiując stateful pipe, oznaczamy ją jako `pure: false`

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

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



Routing

Angular Component Router



Konfiguracja Routingu

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

const routingModule = RouterModule.forRoot([
  { path: 'todos', component: TodosComponent },
  { path: '', component: HomeComponent },
  { path: '**', component: PageNotFoundComponent }
]);
```

```
@NgModule({
  imports: [
    BrowserModule,
    routingModule,
    ...
```

Router mapuje ścieżkę w URL na ustalony komponent.
Kolejność podawania reguł ma znaczenie.

Komponent pojawi się w szablonie w miejscu wskazanym
dyrektywą:

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




Routing dla Sub-Modułu

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

RouterModule.forRoot() - tworzy moduł dla głównego modułu.

RouterModule.forChild() - pozwala by submoduły miały własny częściowy routing.

RouterModule tworzy cały moduł, by udostępnić wraz z nim potrzebne narzędzia - usługi i dyrektywy dla tego routingu!



Routing Lazy loading

```
@NgModule({  
  imports: [  
    RouterModule.forRoot([  
      { path: 'heroes', loadChildren: import('./heroes/heroes.module')  
        .then(m => m.HeroesModule)  
      },  
    ])  
  ],  
  exports: [  
    RouterModule  
  ]  
})  

```



Przekazywanie parametrów

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

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



Adresy możemy parametryzować - Wstrzykując usługi **ActivatedRoute** lub **Params** mamy dostęp do parametrów

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

```
constructor(
```

```
  private route: ActivatedRoute,
```

```
  private router: Router,
```

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

```
// Usługa Router pozwala na programistyczną nawigację:
```

```
onSelect(todo: Todo) {
```

```
  this.router.navigate(['/todo', todo.id]);
```



Linkowanie do ścieżek routera

Dyrektywa **RouterLink** aktywuje daną ścieżkę routingu po kliknięciu w element.

Ścieżka może składać się z wielu poziomów - podanych jako tablica:

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

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

RouterLinkActive to dyrektywa która dodaje i usuwa podaną klasę CSS gdy ścieżka jest aktywna lub nie



Testy jednostkowe

poszczególnych elementów frameworka



Testowanie Filtrów

i prostych usług

```
describe ('TitleCasePipe', () => {  
  // Pipe jest prostą bezstanową klasą  
  // - nie ma tutaj specjalnej potrzeby inicjalizacji  
  let pipe = new TitleCasePipe();  
  
  it ('transforms "abc" to "Abc"', () => {  
    expect (pipe.transform('abc')).toBe ('Abc');  
  });  
});
```



Testowanie komponentów

Komponent wymagać może inicjalizacji, np. dyrektyw z BrowserModule

```
beforeEach(() => {  
    TestBed.configureTestingModule({  
        declarations: [ BannerComponent ],  
        imports: [ BrowserModule ]  
    });  
  
    // Tworzy Komponent i opakowauje w "Fixture" ( ComponentFixture )  
    fixture = TestBed.createComponent(BannerComponent);  
  
    // Możemy dostać się bezpośrednio do klasy komponentu:  
    comp = fixture.componentInstance;  
});
```



Testowanie widoku

Jeżeli chcemy testować nie tylko klasę komponentu, ale także wygenerowany HTML, nie możemy zapomnieć o cyklu wykrywania zmian:

```
it('should display original title' , () => {  
  
    // Wywołaj ręcznie detekcje zmian by zakualizować widok (HTML):  
    fixture.detectChanges();  
  
    // Obiekt DebugElement posiada metody, np. do odnajdywania po CSS:  
    de = fixture.debugElement.query(By.css('h1'));  
  
    // Możemy porównać zawartość HTML z Obiektem komponentu:  
    expect(de.nativeElement.textContent).toContain(comp.title);  
});
```




Dziękuję za uwagę!

Pytania? ;-)