Michał Jabłoński

The Angular

SINGLE PAGE APPLICATION

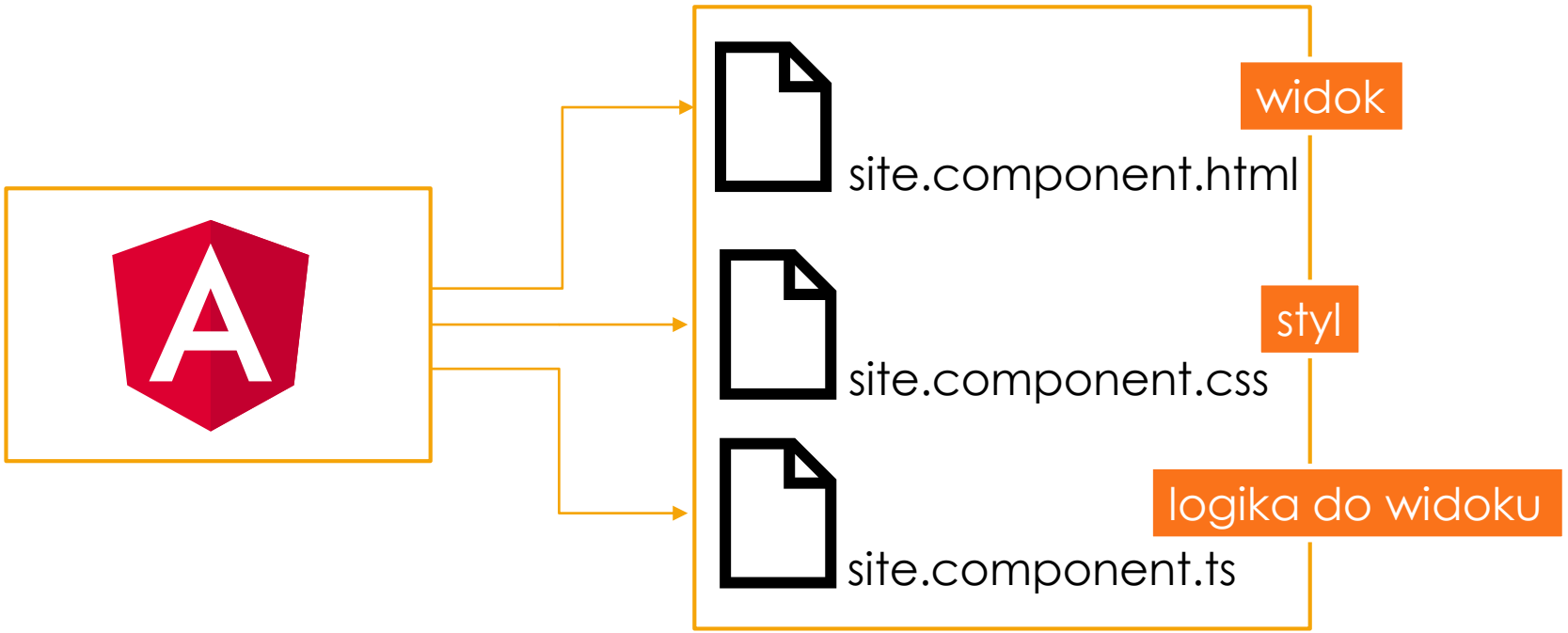
○→ SKRÓT

Charakterystyka Angular

- Naturalność pisania dzięki wykorzystaniu podziału: widok – logika – styl
- Moduły: „Kontenery komponentów" – jednostka organizacyjna
- Dependency Injection – wstrzykiwanie zależności
- Komponenty z bliska
- Templates – widoki angulara
- Stylowanie komponentów
- Input/Output dla komponentów
- Dyrektywy i Najważniejsze wbudowane dyrektywy
- Cykl życia – dla Dyrektyw i komponentów
- Pipes
- Services – separacja logiki od widoku
- RxJs – programowanie na „strumieniach"
- Sposoby definiowania formularzy

Naturalność pisania dzięki wykorzystaniu podziału: widok – logika – styl

- Komponenty to podstawowa jednostka do budowania w Angular



Moduły: „Kontenery komponentów” – jednostka organizacyjna

- Jeśli czegoś nie ma w Module - to „nie istnieje to dla Angulara”
- Moduły mogą zawierać inne moduły
- Podstawą skalowalności aplikacji opartej o Angulara jest użycie wielu modułów
- Każdy z modułów może być niezależny. Posiadać własne komponenty, własny Routing, własne zależności.

- Definiowanie komponentów i zależności po przez Moduły jest konieczne dla prawidłowego funkcjonowania „Dependency Injection”

@NgModule

- Klasa TypeScript
- ale Dekor@tor jest Angularowy
- Najważniejsze są tu metadane !

```typescript
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';

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

Dependency Injection – wstrzykiwanie zależności

Injector

Singleton: instancja Router

```
@Component({
  selector: 'app-main-menu',
  templateUrl: './main-menu.component.html',
  styleUrls: ['./main-menu.component.css']
})
export class MainMenuComponent {

  constructor(private router: Router) {}

  handleNotSignedIn() {
    this.router.navigate(['/sign-in']);
  }

}
```

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

  constructor(private router: Router) {}

  handleGoHome() {
    this.router.navigate(['/']);
  }
}
```

Komponenty z bliska

- Klasa **TypeScript**
- Posiada ponad deklaracją – dekorator
  - **`@Component( { …… } )`**
  - Metadane do klasy określające dodatkowe zależności
    - **`selector`** – nazwa komponentu, która będzie używana w widokach / templatach HTML
    - **`templateUrl`** lub **`template`** - ścieżka do lub zawartość widoku
    - **`styleUrls`** lub **`styles`** - ścieżka do lub tablica styli
    - i inne...

```
<app-main-menu></app-main-menu>
```

```
@Component({
  selector: 'app-main-menu',
  templateUrl: './main-menu.component.html',
  styleUrls: ['./main-menu.component.css']
})
export class MainMenuComponent {

  constructor(private router: Router) {}

  handleNotSignedIn() {
    this.router.navigate(['/sign-in']);
  }

}
```

Templates – widoki

- Widok połączony z komponentem
- Postać: HTML
- Najczęściej oddzielny plik .html, w przypadku małej ilości znaków – może być uwzględniony „INLINE" w dekoratorze: **@Component**
- Widok może korzystać bezpośrednio z publicznych zmiennych i funkcji klasy komponentu (pliku .ts)

```
<div>
  <h1>Witaj w {{ mySimpleTitle }}</h1>
</div>
```

Templates – lokalna referencja do DOM

- **#**
- Używając znaku **#** wewnątrz znacznika HTML możemy w template uzyskać odwołanie do tzw. ElementRef – będącego referencją do natywnego obiektu DOM
- Do referencji, w klasie komponentu mamy dostęp dzięki dekoratorowi **@ViewChild**

```
<div>
   <label for="person"> Personal Info </label>
   <input type="text" id="person" name="person" #personInput/>
   <button (click)="personInput.focus()"> Focus Input! </button>
</div>
```

Stylowanie komponentów

- Kilka sposobów stylowania
  - [prostota i wygoda] : oddzielny plik .css + odniesienia do class w template
  - [jeśli mało znaczące i mało] : Inline style w dekoratorze: **@Component**
  - [dynamicznie] :
    - ngStyle lub ngClass
    - `[style.color]="'yellow'"`
    - `[class.myClass]=„true”`

```
<div [class.wrapper]="true">
  <h1 [style.color]="'yellow'">Witaj w {{ mySimpleTitle }}</h1>
</div>
```

Sposoby bindowania

Kierunek przepływu	Sposób zapisu	Typ

One-way
from data source
to view target
	`{{expression}}`
	`[target]="expression"`
	`bind-target="expression"`
		Interpolation
		Property
		Attribute
		Class
		Style

One-way from view target to data source	`(target)="statement"` `on-target="statement"`	Event
Two-way	`[(target)]="expression"` `bindon-target="expression"`	Two-way

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

Input/Output dla komponentów

- 1. Można określić w dekoratorze:
  `@Component` (choć sugerowana jest opcja 2)
- 2. Można użyć dekoratorów:
  - `@Input() data: string[];`
  - `@Output() dataChange: EventEmmiter<boolean>();`

```
export class MainMenuComponent {

  @Input() menuTitle: string;
  @Output() menuChange = new EventEmitter<string>();
```

Dyrektywy

- W zasadzie 3 rodzaje:
- 1 – omówiony wcześniej (Komponent = Dyrektywa z Szablonem)
- 2 – Dyrektywy atrybutowe
  - Przykład tutaj
  - Zmienia zachowanie elementu nie wpływa na jego strukturę
- 3 – Dyrektywy strukturalne
  - Zmieniają zachowanie elmentu modyfikując jego strukturę (*)

```
<nav appDropDown></nav>
```

```
@Directive({
  selector: '[appDropDown]'
})
export class DropDownDirective {

  // binduj zmianę flagi z 'pokazaniem' lub 'nie' klasy open
  @HostBinding('class.open') isOpen = false;

  // stuchaj eventu 'click' na Hoście i zmień flagę
  @HostListener('click')
  toggleOpen() {
    this.isOpen = !this.isOpen;
  }

}
```

Najważniejsze wbudowane dyrektywy

- *ngIf | *ngIf=„condition"

```
<li class="nav-item" *ngIf="toggleMenu">
  <a class="nav-link">Show latest</a>
</li>
```

- *ngFor | *ngFor=„let variable of collection; let i = index"

(index, first, last, even, odd)

```
<img class="my-img" *ngFor="let photo of photos" [src]="photo.id"  [alt]="photo.title"/>
```

Najważniejsze wbudowane dyrektywy

- ngStyle

```
<button class="btn btn-danger" (click)="align = 'center'">Switch !</button>
<p [ngStyle]="{ textAlign: align, color: 'blue', 'width.px': 300 }"> Hello Text </p>
```

- ngClass

```
<div [ngClass]="{'collapse':false, 'navbar-collapse':true, show:isMenuShown}">
```

Najważniejsze wbudowane dyrektywy

- ngSwitch -> *ngSwitchCase

```
<ng-container [ngSwitch]="superHeroName">
  <ng-container *ngSwitchCase="'CLARK'">
    SUPERMAN
  </ng-container>
  <ng-container *ngSwitchCase="'BRUCE'">
    BATMAN
  </ng-container>
  <ng-container *ngSwitchDefault>
    SUPERMAN
  </ng-container>
</ng-container>
```

Cykl życia – dla Dyrektyw i komponentów

- Event hooks

	Hook	Dla	Przeznaczenie i czas:
1	**ngOnChanges()**	C/D	Called before ngOnInit() and whenever one or more data-bound input properties change.
2	**ngOnInit()**	C/D	Called once, after the first ngOnChanges().
3	**ngDoCheck()**	C/D	Called during every change detection run, immediately after ngOnChanges() and ngOnInit().
4	**ngAfterContentInit()**	C	Respond after Angular projects external content into the component's view / the view that a directive is in. Called once after the first ngDoCheck().
5	**ngAfterContentChecked()**	C	Respond after Angular checks the content projected into the directive/component. Called after the ngAfterContentInit() and every subsequent ngDoCheck().
6	**ngAfterViewInit()**	C	Respond after Angular initializes the component's views and child views / the view that a directive is in. Called once after the first ngAfterContentChecked().
7	**ngAfterViewChecked()**	C/D	Respond after Angular checks the component's views and child views / the view that a directive is in. Called after the ngAfterViewInit and every subsequent ngAfterContentChecked().
8	**ngOnDestroy()**	C/D	Cleanup just before Angular destroys the directive/component. Unsubscribe Observables and detach event handlers to avoid memory leaks. Called just before Angular destroys the directive/component.

- C – komponent, D - dyrektywa

Pipes
– sterowanie tym, co wyświetlamy

```
<div class="card-footer">
  {{ todayDate }}
</div>
```

Thu Mar 01 2018 01:00:00 GMT+0100 (Środkowoeuropejski czas stand.)

```
<div class="card-footer">
  {{ todayDate | date }}
</div>
```

Mar 1, 2018

```
<div class="card-footer">
  {{ todayDate | date:'MM/dd/y' }}
</div>
```

03/01/2018

Services – separacja logiki od widoku

- Potrzebne do odseparowania logiki

która nie jest bezpośrednio związana z widokiem

- Mogą być odpowiedzialne za:
  - Zapytania HTTP
  - Trzymanie stanu danych (session like)

```
@Injectable({
  providedIn: 'root'
})
export class PhotoService {

  constructor() { }

  getPhotos() {
    return [
```

- Wstrzykiwane (DI) /tam gdzie potrzebujemy/ w konstruktorze innego serwisu

lub w konstruktorze kontrolera.

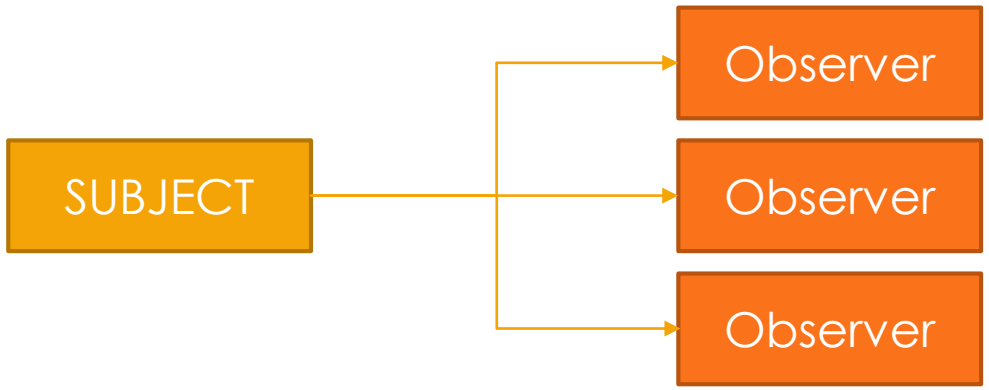
RxJs – programowanie na „strumieniach”

- Praktyczne wykorzystanie wzorca „Obserwator”
- Autonomiczna biblioteka (RxJS), która można wykorzystać nie tylko w Angular
- Założenie że „z wszystkiego” można zrobić strumień danych
- Koncepcja programowania „reaktywnego”
- Dużo operatorów do pracy ze strumieniami

- W zrozumieniu pomagają „marble diagrams"

http://rxmarbles.com

RxJs
– programowanie na „strumieniach”



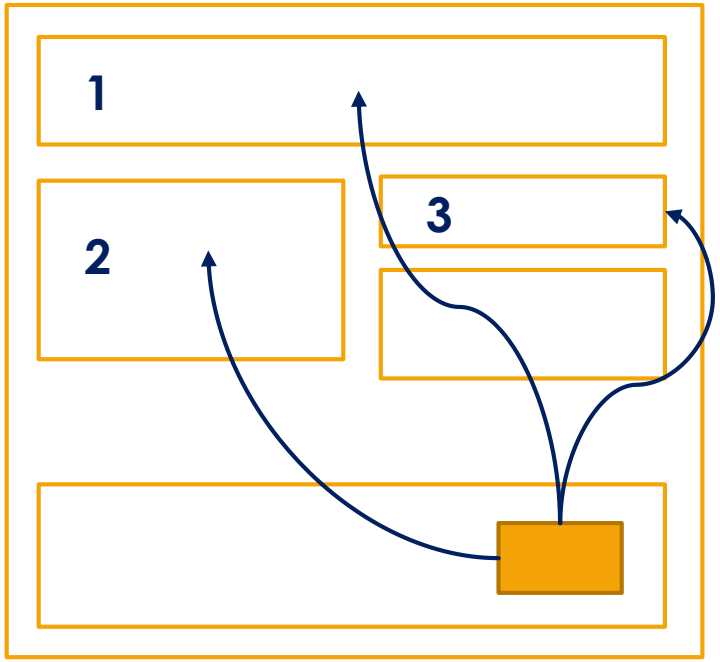
- Subject informuje wszystkich Obserwatorów
- Wcześniej każdy z nich MUSI zasubskrybować chęć otrzymywania takich informacji !
- Każdy z Obserwatorów może zrezygnować z subskrybcji
- Unikamy „tightly coupled” (silnego powiązania)

Wykorzystanie RxJS w Angular

- Kilka obiektów widocznych na stronie wyświetla informacje na podstawie informacji z tego samego źródła.

- W momencie aktualizacji danych u źródła.
  Każdy komponent może zaktualizować to co wyświetla

- Źródłem danych będzie tutaj – serwis.
  Przechowujący fragment stanu aplikacji.



Sposoby definiowania formularzy

- Template – Driven Forms
  - Formularz w szablonie HTML.
  - Pola formularza powiązane z komponentem poprzez system wiązania danych.
  - drzewo formularza, tworzone asynchronicznie przez framework.

- Reactive Form
  - Układ widoku formularza – w szablonie HTML
  - drzewo formularza zdefiniowane w logice widoku przez programistę.

Sposoby definiowania formularzy

- Stany formularza (dodawane do kontrolek jako klasy)
  - .ng-valid
  - .ng-invalid
  - .ng-pending
  - .ng-pristine
  - .ng-dirty
  - .ng-untouched
  - .ng-touched