

Efektywne tworzenie aplikacji webowych z

Angular





Szkolenie

- Plan szkolenia i cele
- Aktualne obowiązki
- Pytania, dyskusje, potrzeby
- Elastyczny program szkoleniowy



Mateusz Kulesza

- Senior Software Developer
- Team Leader
- Project Manager
- Consultant and Trainer

Narzędzia “ng”

Generatory kodu

Stworzenie nowego projektu angular 2 w b. katalogu

```
ng new nazwaprojektu  
ng new nazwaprojektu --standalone
```

Uruchomienie lokalnego serwera developerskiego

```
ng serve
```

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

```
ng generate component <name>
```

Generatory kodu to tzw. "Schematics"

Narzędzia “ng”

parametry

Tworzy nowy komponent!

```
ng generate component Nazwa
```

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

Użyj ``ng config`` lub pliku ``angular.json`` aby ustawić opcje domyślne

Rozszerzenia

Visual Studio Code (Ctrl+Shift+X)

Angular Language Service

<https://marketplace.visualstudio.com/items?itemName=Angular.ng-template>

Angular 10 Snippets - Mikael Morlund

<https://marketplace.visualstudio.com/items?itemName=Mikael.Angular-BeastCode>

Prettier

<https://marketplace.visualstudio.com/items?itemName=esbenp.prettier-vscode>

Paste JSON as Code - quicktype

<https://marketplace.visualstudio.com/items?itemName=quicktype.quicktype>

Bootstrap

Startowanie Applikacji

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

Inicjalizacja głównego modułu (z wybranym głównym komponentem):

```
platformBrowserDynamic().bootstrapModule(AppModule);
```

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

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

Moduły w Angular

@NgModule

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

@NgModule({
  imports: [BrowserModule],           // Importujemy "exporty" z innych modułów:

  declarations: [AppComponent],      // Deklaracje dla parsera HTML ( Dyrektywy i komponenty ):

  bootstrap: [AppComponent],         // Który komponent będzie "głównym" komponentem aplikacji:

  exports: [],                      // Udostępnione elementy

  providers: [],                    // definicje usług ( o tym później... )

})
export class AppModule {}
```

Standalone - Aplikacje "bezmodułowe"

main.ts

```
import { bootstrapApplication } from "@angular/platform-browser";
import { appConfig } from "./app/app.config";
import { AppComponent } from "./app/app.component";

bootstrapApplication(AppComponent, appConfig).catch((err) =>
  console.error(err)
);
```

app.component.ts:

```
@Component({
  selector: "app-root",          // Selektor HTML
  standalone: true,             // Bezmodułowy
  imports: [CommonModule, RouterOutlet, MyModule],
  templateUrl: "./app.component.html",
  styleUrls: ["./app.component.scss"],
})
export class AppComponent {
  title = "project";
}
```

Deklaracja komponentu

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

@Component({
  // standalone: true
  selector: 'app-root, .extra-selector',
  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
- ```.spinner-loader`` - klasy CSS
- ```app-root`` - elementu

Szablon

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

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

Widok

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



"Szablon" umieszcza "widok" w
miejscu określonym przez "selektor"

Interpoluje wyrażenia umieszczone w klamrach

Enkapsulacja Styli

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

Enkapsulacja może przyjąć wartości:

- **Emulated** (domyślnie) - style z dokumentu HTML propagowane są do komponentu i jego dzieci. Style zdefiniowane w @Component() nie wpływają na reszta dokumentu - są izolowane
- **ShadowDom** - 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

Zagnieżdżone komponenty

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

@Component({
  selector: 'app-root',
  template: `<div>
    <h1>Parent</h1>

    <sub-component/>

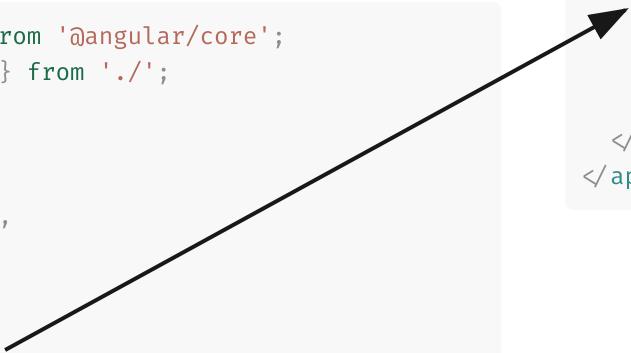
  </div>`
})
export class AppComponent { ... }
```

```
@Component({
  selector: 'sub-component',
  template: `<div> <h3>Child</h3>
  </div>`
})
export class ChildComponent { ... }
```

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

    <sub-component>
      <h3>Child!</h3>
    </sub-component>

  </div>
</app-root>
```



Wiązania właściwości i dyrektywy atrybutów

Wiązania Własności

```
<div [id]="'identityfikator_' + myFakeUuid" [title]="zmienna">  
  
<div title="Hello {{userName}}">
```

Style

```
<p [style.backgroundColor]="'lime'"> Jestem limonkowy! </p>  
<p [style.backgroundColor]="'lime'"> Jestem limonkowy! </p>
```

Jednostki

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

Klasy

```
<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>
```

Lokalne Referencje

Umieszczenie znaku hash z nazwą (“#nazwa”) na elemencie tworzy “lokalną referencję”

To zmienna dostępną tylko w tym komponencie,
która zawiera odniesienie do elementu DOM
(lub do komponentu)

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

Można się do niej odwołać z kodu komponentu
poprzez ViewChild

```
class AppComponent{  
  
    @ViewChild('movieplayer')  
    movieplayer?: ElementRef<HTMLVideoElement>  
  
    ngAfterViewInit(){  
        this.movieplayer  
    }  
}
```

Signals

```
const count = signal(0);

// Signals are getter functions - calling them reads their value.
console.log('The count is: ' + count());
```

Aby zmienić wartość zapisywального sygnału, możesz

```
count.set(3);
```

lub użyj operacji `.Update()`, aby obliczyć nową wartość z poprzedniej:

```
// Increment the count by 1.
count.update(value => value + 1);

// NOTE: `mutate` będzie obchodzić sprawdzenie równości i zawsze powiadomi o zmianie
todos.mutate((currentTodos) => { ... })
```

Computed effect

```
const doubleCount: Signal<number> = computed(() => count(  
  
fieldName = effect(() => {  
  console.log(`The current count is: ${doubleCount()}`);  
});
```

Cleanup

```
effect((onCleanup) => {
  const user = currentUser();

  const timer = setTimeout(() => {
    console.log(`1 second ago, the user became ${user}`);
  }, 1000);

  onCleanup(() => {
    clearTimeout(timer);
  });
});
```

Wiązanie Właściwości

Komunikacja do komponentu zagnieżdzonego

```
@Component({  
  selector: 'todo-input',  
  template: ' ... '  
})  
  
export class Hello {  
  
  @Input() title = '';  
  
  @Input() item?: MyTodoType;  
  
  // Wejście bez wartości domyślnej  
  @Input() optional?: string  
  
  // Angular 16+ - wymagane wejście  
  @Input({required:true}) someProp: Type  
  
}
```

Przekazywanie wartości jako statyczny tekst

```
<todo-input title="Kupić chleb"></todo-input>
```

Przekazywanie przez dynamiczne jako wyrażenie

```
<todo-input [item]="getMyItem()"></todo-input>
```

Emiter zdarzeń

Komunikacja do komponentu nadzecznego

```
@Component({
  selector: 'todo-input',
  template: '<button (click)="clickCompleted()"> ..'
})
export class Hello {
  @Output() completed = new EventEmitter<boolean>();

  clickCompleted(){
    // kliknięcie wysyła sygnał do rodzica
    this.completed.emit( this.todo )
  }
}
```

```
@Component({
  selector: 'todo-input',
  template: `
    <todo-input (completed)="saveProgress($event)" />
  `
})
export class Hello {

  saveProgress(todo:Todo){
    // ... funkcja wykonana przez podkomponent
  }
}
```

Komunikacja z komponentem w obie strony

Dyrektywa ngModel:

Wiąże dane ze zmienną:

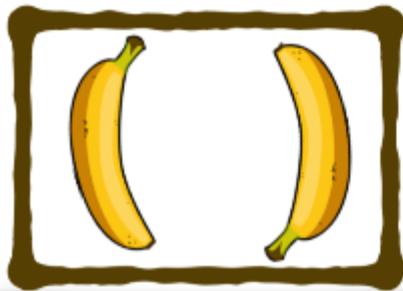
```
<input [ngModel]="name">           <—— tylko aktualizacja wartości
```

Wiąże nasłuchiwanie zdarzeń zmian (onChange):

```
<input ngModel (ngModelChange)=" name = $event ">    <—— tylko nasłuch zmian
```

Lub w skrócie:

```
<input [(ngModel)]="name">           <—— Dwustronne wiązanie danych ( aktualizacja i nasłuch )
```



[(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:

Oznaczenie “ * ” przed dyrektywą tworzy szablon za nas

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

Dyrektywy strukturalne

Pozwalają modyfikować strukturę DOM

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

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



Dyrektywy

Angular 2+ Syntax

```
<div *ngIf="getUser() as user; else guestMessage">  
  The user {{user}} is logged in  
</div>  
  
<ng-template #guestMessage>  
  The user is not logged in  
</ng-template>  
  
<div *ngFor="let item of items; track: item.id">  
  
  {{ item.name }}  
</div>  
  
<ng-container *ngSwitch="modes">  
  <ng-container *ngSwitchCase="'modeA'"> ...  
  <ng-container *ngSwitchCase="'modeB'"> ...  
  <ng-container *ngSwitchDefault> ...
```

Kontrola przepływu

Angular 17+ Syntax

```
@if ( getUser() as user;) {  
  <p>The user {{user}} is logged in</p>  
}  
@else {  
  <p> The user is not logged in</p>  
}  
  
@for (item of items; let i = $index; track item.id) {  
  {{i+1}}. {{ item.name }}  
}  
  
@switch (condition)  
{  
  @case (caseA) {  
    <p>Case A.</p>  
  }  
  @default {  
    Default case.  
  }  
}
```

Projekcja zawartości

za pomocą dyrektywy `ngContent` możemy umiejscowić w szablonie komponentu html wpisany jako zawartość komponentu

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

Opcjonalnie można zdefiniować miejsce dla elementu z określonego selektora :

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

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

```
<user-profile>  
  <h4>User profile</h4>  
  <h5>Lito Rodriguez</h5>  
  <small>Actor</small>  
</user-profile>
```



Cykl życia

Dyrektywy (D) i komponenty © mogą “wpaść” się w cykl renderowania aby wykonać własne funkcje:

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

```
class ComponentOrService{  
  constructor(  
    private nazwa_zmiennej: KlasaUsługi,  
    private inna: TypowanaUsługa ){  
      this.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 wewnętrz naszych klas “Łaczmy na sztywno” implementacje:

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

Lepiej jest wiązać klasy przy użuciu 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 injektorów! - już samo bootstrapModule(MyAppModule) tworzy za nas automatycznie globalny Injector.

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

Domyślnie wystarczy zarejestować 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

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 instacja, prosty obiekt,
    // lub nawet wartość (np. config..)
    { provide: Typ, useValue: Obiekt },

    // Funkcja fabrykująca
    // - może mieć własne zależności ( deps:[ ... ] )
    { provide: Typ, useFactory: ( someAPI ) => {
        // ... return someAPI.someSetup()

        // Zależności dla funkcji fabryki
      }, deps: [ SomeAPIClass ]},
    ],
  })
})
```

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 = new InjectionToken('URLToken');
```

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

Hierarchiczny Injector

Domyślnie, każda stworzona instancja jest singletonem

Angular tworzy obiekt raz, a następnie przekazuje zawsze globalnie tą samą instancję.

Jeśli chcemy stworzyć lokalną instancje, definiujemy ją w `@Component`

```
@Component({
  selector: 'isolated-data-view'
  template: `<child-component/>`
  providers: [ NaszaUsluga ]
})
export class ParentComponent { }

class NaszaUsluga{
  // Usługa niszczona wraz z komponentem
  ngOnDestroy(){}
}
```

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

Jednak ten komponent posiada teraz własny injektor, więc ten komponent i jego dzieci otrzymają nową instancję klasy NaszaUsluga niedostępną dla reszty komponentów.

Jeśli klasa nie jest zdefiniowana tutaj, będzie użyty injektor nadzędny, nadzędny, itd...

... aż po globalny injector.

```
export class ChildComponent{
  // Lokalna instancja z ParentComponent
  usluga = inject(NaszaUsluga)
}
```

Programowanie Reaktywne

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

RxJS

czyli Reactive Extensions - “strumienie zdarzeń”

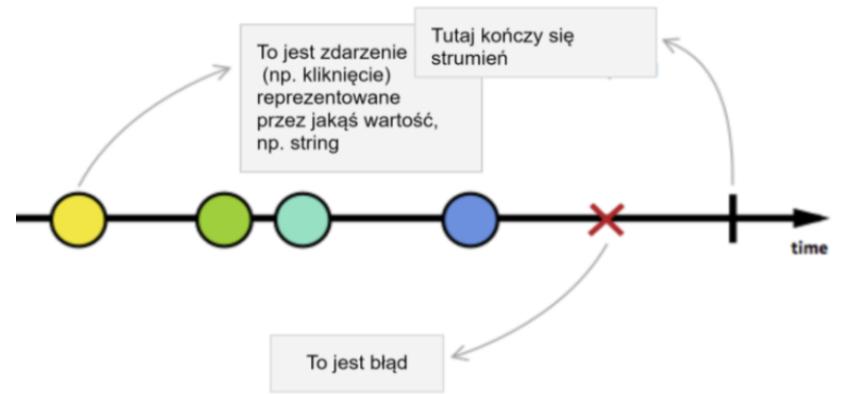
Każdy obiekt typu `EventEmitter<T>()` jest rozszerza `Observable`.

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

RxJS posiada bogatą bibliotekę operatorów umożliwiających przekształcenia

Strumienie możemy

- obserwować,
- przekształcać

... a następnie subskrybować w celu otrzymania informacji, kiedy nastąpiła zmiana.

Sposób działania poszczególnych operatorów obrazuje się za pomocą "marble diagrams"

- <https://rxjs.dev/guide/operators>
- <http://rxmarbles.com/>
- <https://www.learnrxjs.io/>
- <https://rxviz.com/>

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

Zapis ASCII Marble:

--a---b-c---d---X---+→

a, b, c, d są emitowanymi wartościami
X to błąd
| to sygnał „ukończony”
→ to oszczędzana oś czasu

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 reaktywny

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

import { FormControl, FormGroup, Validators } from '@angular/forms';

/* @Component ... */
export class MyForm {
  this.regForm = new FormGroup({
    username: new FormControl('wartość domyślna', [
      Validators.required, Validators.minLength(3)])
  });
}
```

lub

```
class MyForm{

  constructor(private builder: FormBuilder ) {
    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 validator 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

```
<form #nazwa="ngForm" (ngSubmit)="mojaMetoda(nazwa)">
```

Pipes

Filtry

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>
```

- JSON pipe

```
<pre>{{ obiekt | json }}</pre>
```

Wbudowane: DatePipe,UpperCasePipe,LowerCasePipe,CurrencyPipe,PercentPipe,JsonPipe

Parametryzowane Pipes

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

Definiując stateful pipe, oznaczamy ją jako pure: false

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

AsyncPipe przyjmuje jako wejście Promise lub Observable i przechowuje subskrypcję, żeby później zwrócić wartość

```
<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
  ]
}) class SubModuleX{}
```

RouterModule.forRoot()

- tworzy moduł dla głównego modułu.

RouterModule.forChild()

- pozwala by submoduły miały własny częściowy routing.

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

Routing Lazy loading

```
const routes: Routes = [
  { path: 'heroes',
    loadChildren: import('./heroes/heroes.module')
      .then(m => m.HeroesModule)
  },
]

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

Przekazywanie parametrów

```
{ path: 'todo/:id', component: TodoDetailComponent }  
// http://localhost/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 Pipes

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');
  });
});
```

ng test

Testowanie komponentów

Komponent wymagać może inicjalizacji

np. dyrektyw z BrowserModule

```
beforeEach(() => {
  TestBed.configureTestingModule({
    declarations: [ BannerComponent ],
    imports: [ BrowserModule ]
  });

  // Tworzy Komponent i opakowuje 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? ;-)