Architektura Angular

Tworzenie serwisów Web 2.0

dr inż. Robert Perliński
rperlinski@icis.pcz.pl

13 kwietnia 2019

Plan prezentacji

Spis treści

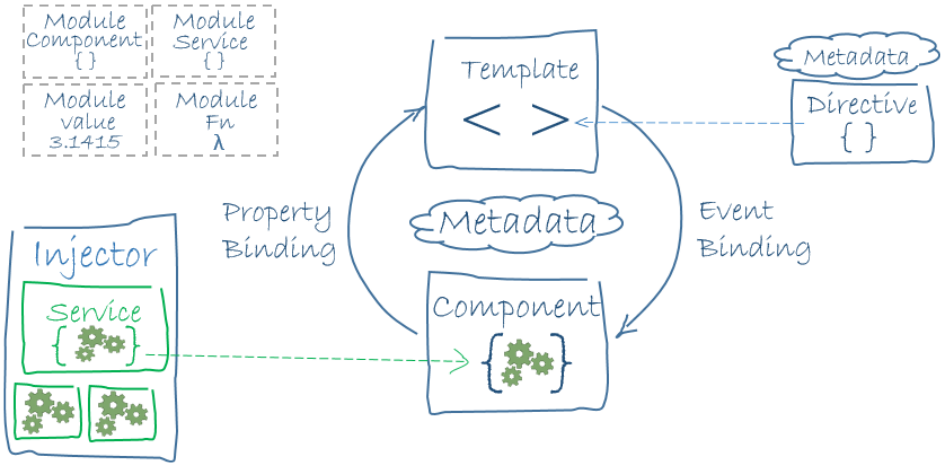
1 Architektura Angular

Architektura Angular

Podstawowe elementy konstrukcyjne aplikacji Angular

- Angular to szablon aplikacji do budowania aplikacji działających po stronie klienta w HTML i JavaScript albo TypeScript, który kompiluje się do JavaScript.
- Framework składa się z kilku bibliotek, niektóre są podstawowe, konieczne, inne są opcjonalne.
- Aplikację Angular tworzymy łącząc szablony HTML ze znacznikami Angulara, piszemy klasy komponentów do zarządzania tymi szablonami, dodajemy logikę aplikacji w usługach, i umieszczamy komponenty i usługi w modułach.
- Następnie uruchamiamy aplikację odpalając główny moduł. Angular przejmuje kontrolę, prezentuje zawartość naszej aplikacji w przeglądarce i odpowiada na interakcje użytkownika zgodnie z instrukcjami, które mu dostarczymy.
- To oczywiście nie wszystko. Szczegóły będą później, na razie skupmy się na oglądzie całości.

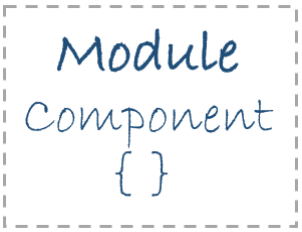
Architektura Angular - 8 głównych bloków konstrukcyjnych



- moduły
- komponenty
- szablony
- metadane
- wiązanie danych
- dyrektywy
- usługi
- wstrzykiwanie zależności

1.1 moduły

Moduły



- Aplikacje Angular są podzielone na moduły i Angular ma własny system modułów nazywany **modułami Angular** albo **NgModules**.
- Moduły Angulara to bardzo ważny element. Omówiono je szczegółowo w tematach zaawansowanych.
- Każda aplikacja ma przynajmniej jedną klasę modułu Angular, moduł główny, umownie nazywany `AppModule`.
- Chociaż gdy moduł główny może być jedynym modułem małej aplikacji, większość projektów ma o wiele więcej modułów z określoną funkcjonalnością, każdy zawierający spójny blok kodu przeznaczony do pewnego zakresu funkcjonalności aplikacji, przepływu pracy czy związanego ze sobą zbioru możliwości (funkcjonalności).
- Moduł Angulara, niezależnie czy główny czy związany z konkretną funkcjonalności, jest klasą z dekoratorem `@NgModule`.

Dekorator

Dekoratory są funkcjami, które modyfikują clasy JavaScript. Angular ma wiele dekoratorów, które dołączają metadane do klas, co pozwala na określenie znaczenia tych klas i sposobu ich działania.

Więcej o dekoratorach w artykule Exploring EcmaScript Decorators

Dekorator NgModule

`NgModule` jest funkcją dekoratora, która przyjmuje jeden obiekt z metadanymi, których właściwości opisują moduł. Najważniejsze właściwości to:

- `declarations` - klasy widoków, które należą do tego modułu. Angular ma trzy rodzaje klas widoków: komponenty, dyrektywy i strumienie/potoki.
- `exports` - podzbiór deklaracji, które powinny być widoczne i dostępne w szablonach komponentów innych modułów.
- `imports` - inne moduły, których wyeksportowane klasy są potrzebne w szablonach komponentu z tego modułu.
- `providers` - mechanizmy tworzenia usług, które ten moduł dodaje do globalnej kolekcji usług; stają się one dostępne we wszystkich cześciach aplikacji.
- `bootstrap` - główny widok aplikacji nazywany komponentem głównym, w nim znajdują się wszystkie inne widoki aplikacji. Tylko moduł główny powinien mieć tę właściwość ustawioną.

Przykład głównego modułu aplikacji

src/app/app.module.ts

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
@NgModule({
  imports:      [ BrowserModule ],
  providers:    [ Logger ],
  declarations: [ AppComponent ],
  exports:      [ AppComponent ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

- Eksportowanie komponentu `AppComponent` jest tylko dla pokazania jak eksportować; nie jest to potrzebne w tym przykładzie.
- Główny moduł aplikacji nie ma potrzeby eksportować czegokolwiek, bo inne komponenty nie muszą importować modułu głównego.

Uruchomienie aplikacji

src/main.ts

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

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

- Aplikację uruchamiamy poprzez odpalenie jej głównego modułu.
- Podczas tworzenia aplikacji najprawdopodobniej będziemy ją uruchamiać w pliku `main.ts` jak powyżej.

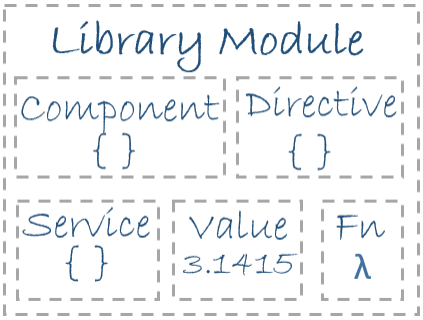
Moduły Angular i moduły JavaScript

- Moduły Angular - klasa utworzona/zmodyfikowana dekoratorem `@NgModule` - są podstawową cechą Angulara.
- JavaScript (TypeScript) ma swój własny system modułów do zarządzania kolekcjami obiektów JavaScript. To jest kompletnie inny i niezwiązany z Angularem system modułów.
- W JavaScript każdy plik jest modułem i wszystkie obiekty zdefiniowane w tym pliku należą do tego modułu. Moduł deklaruje niektóre obiekty jako publiczne oznaczając je słowem kluczowym `export`. Inne moduły JavaScript używają polecenia `import` aby dostać się do publicznych obiektów innych modułów.

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



export class AppModule { }
```

- To są dwa inne ale komplementarne systemy modułów. Obu używamy do tworzenia aplikacji. Więcej o modułach w JS: ExploringJS-Modules

Biblioteki Angulara



- Angular jest dostarczony jako kolekcja modułów JavaScript. Można o nich myśleć jako o bibliotece modułów.
- Nazwa każdej biblioteki Angulara zaczyna się od przedrostka `@angular`.
- Biblioteki instalujemy narządziem **npm** i importujemy jakieś ich części poleceniem `import` języka JavaScript.
- Przykład importowania dekoratora `Component` Angulara z biblioteki `@angular/core`:

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

Biblioteki Angulara

- Można również importować `moduły` Angulara z `bibliotek` Angulara używając polecenia `import` języka JavaScript:

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

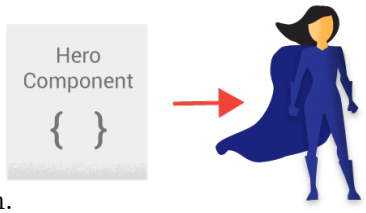
- We wcześniejszym przykładzie modułu głównego, moduł aplikacji potrzebował funkcjonalności z `BrowserModule`.
- Dostęp do tej funkcjonalności uzyskujemy przez dodanie `BrowserModule` do właściwości `import` metadanych dekoratora `@NgModule`, jak poniżej:

```
imports:      [ BrowserModule ],
```

- W ten sposób używamy systemu modułów Angulara i systemu modułów JavaScript razem.
- Łatwo jest pomylić te dwa rodzaje systemu modułów ponieważ dzielą wspólne słownictwo: `import` i `export`. Nie poddajemy się zniechęceniu. Wraz ze wzrostem doświadczenia będziemy mieli jasność, którego systemu i gdzie używamy.

1.2 komponenty

Komponenty



- `Komponent` kontroluje fragment ekranu nazywany widokiem.
- Przykładowo, trzy widoki aplikacji o bohaterach są kontrolowane przez komponenty:
  - główny komponent aplikacji z linkami nawigacyjnymi,
  - lista bohaterów,
  - edytor wybranego bohatera.
- Wewnątrz klasy komponentu definiujemy logikę aplikacji - określamy jak komponent obsługuje widok.
- Klasa komponentu komunikuje się z widokiem poprzez API pól i metod.

Komponenty

- Przyład: komponent `HeroListComponent` posiada pole `heroes`, które zwraca tablicę bohaterów, którzy są pobierani za pośrednictwem usługi.
- Komponent `HeroListComponent` posiada też metodę `selectHero()`, która pozwal ustawić pole `selectedHero` kiedy użytkownik kliknie wybierając bohatera z listy.

src/app/hero-list.component.ts (klasa)

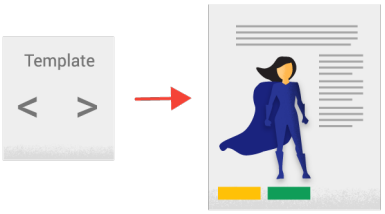
```
export class HeroListComponent implements OnInit {
  heroes: Hero[];
  selectedHero: Hero;

  constructor(private service: HeroService) { }

  ngOnInit() {
    this.heroes = this.service.getHeroes();
  }

  selectHero(hero: Hero) { this.selectedHero = hero; }
}
```

- Angular toworzy, aktualizuje i usuwa komponenty odpowiednio do działań użytkownika w aplikacji.
- Nasza aplikacja może wykonywać dowolne akcje w każdym momencie w cyklu życia komponentów poprzez użycie opcjonalnych wstawek programowych (ang. *lifecycle hooks*), np. zadeklarowane powyżej `ngOnInit()`.

1.3 szablony

Szablony

- Definiujemy widok związany z komponentem z użyciem towarzyszącego mu szablonu.
- Szablon jest formą HTMLa, która informuje Angular jak wyświetlić komponent.
- Szablon wygląda jak zwyczajny HTML, z kilkoma różnicami.
- Przykład szablonu dla komponentu `HeroListComponent`:



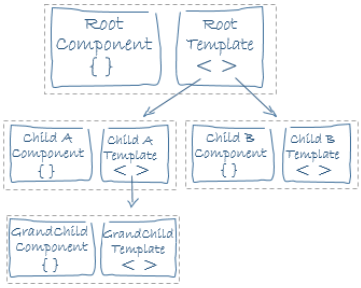
src/app/hero-list.component.html

```
<h2>Hero List</h2>
<p><i>Pick a hero from the list</i></p>
<ul>
  <li *ngFor="let hero of heroes" (click)="selectHero(hero)">
    {{hero.name}}
  </li>
</ul>
<hero-detail *ngIf="selectedHero" [hero]="selectedHero"></hero-detail>
```

- Pomimo tego, że szablon używa zwykłego znaczników HTML jak `<h2>` i `<p>`, to zawiera jednak również inne elementy.
- Kod taki jak `*ngFor`, `{{hero.name}}`, `(click)`, `[hero]` i `<hero-detail>` używa **składni szablonu** Angular.
- W ostatniej linii szablonu, znacznik `<hero-detail>` jest własnym elementem, który reprezentue nowy komponent, `HeroDetailComponent`.

Szablony

- Komponent `HeroDetailComponent` jest innym komponentem niż `HeroListComponent`, który był omawiany wcześniej.
- Komponent `HeroDetailComponent` (kodu nie pokazano) prezentuje dane konkretnego wybranego bohatera, tego, którego użytkownik wybierze z listy prezentowanej przez komponent `HeroListComponent`.
- Komponent `HeroDetailComponent` jest **dzieckiem** komponentu `HeroListComponent`.



src/app/hero-list.component.html

```
...
<hero-detail *ngIf="selectedHero" [hero]="selectedHero"></hero-detail>
```

- Zauważ jak dobrze znacznik `<hero-detail>` pasuje do natywnego kodu HTML.
- Własne komponenty łączą się z dużą łatwością z natywnym HTMLem w tym samym szablonie.

1.4 metadane

Metadane

- Metadane określają w jaki sposób Angular ma przetwarzać klasy.
- Kod komponentu `HeroListComponent` to zwykła klasa. Nie ma żadnych dowodów, że jest jakiś framework, nie ma nic, co by świadczyło o Angularze.
- W rzeczywistości `HeroListComponent` jest naprawdę tylko klasą. Nie będzie traktowany jako komponent dopóki *nie poinformujemy o tym Angulara.*
- Poinformowanie Angulara, że `HeroListComponent` to komponent polega na dołączeniu **metadanych** do klasy.
- W TypeScript dołączamy metadane używając **dekoratora**. Przykład metadanych dla `HeroListComponent`:

src/app/hero-list.component.ts (metadane)

```
@Component({
  selector:    'hero-list',
  templateUrl: './hero-list.component.html',
  providers:  [ HeroService ]
})
export class HeroListComponent implements OnInit {
/* . . . */
}
```

- Dekorator `@Component` - określa, że klasa występująca bezpośrednio za nim jest klasą komponentu.

Metadane

src/app/hero-list.component.ts (metadane)

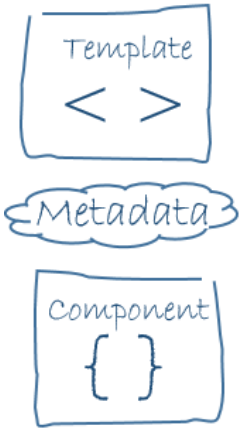
```
@Component({
  selector:    'hero-list',
  templateUrl: './hero-list.component.html',
  providers:  [ HeroService ]
})
export class HeroListComponent implements OnInit {
/* . . . */
}
```

Dekorator `@Component` przyjmuje obiekt konfiguracyjny z informacjami potrzebnymi przez Angular do stworzenia i pokazania komponentu i związanego z nim widoku. Kilka najbardziej użytecznych opcji konfiguracyjnych dyrektywy `@Component`:

- `selector`: selektor CSS, który informuje Angular żeby stworzyć i wstawić instancję tego komponentu w miejscu, w którym w szablonie HTML pojawi się tag `<hero-list>`,
- `templateUrl`: względny, zależny od położenia modułu, adres szablonu HTML tego komponentu,
- `providers`: tablica dostawców uzyskiwanych dzięki wstrzykiwaniu zależności dla usług, których wymaga ten komponent. To jest jeden ze sposobów, żeby powiedzieć Angular, że konstruktor komponentu wymaga usługi `HeroService` żeby mógł pobrać listę bohaterów do wyświetlenia.

Metadane

- Metadane w dekoratorze `@Component` informują Angular, gdzie uzyskać główne bloki konstrukcyjne, z których będzie korzystał komponent.
- Szablon, metadane i komponent razem definiują widok.
- Stosujemy inne dekoratory z ich metadanymi w podobny sposób, żeby sterować/kierować zachowaniem Angulara.
- `@Injectable`, `@Input`, `@Output` to przykłady kilku innych popularnych dekoratorów.

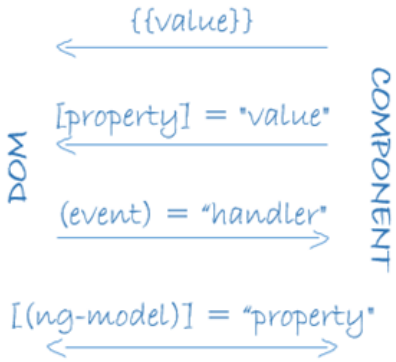


Wniosek jest następujący: musimy dodać metadane do naszego kodu żeby Angular wiedział co ma robić.

1.5 wiązanie danych

Wiązanie danych

Bez wykorzystania frameworka sami jesteśmy odpowiedzialni za wstawianie danych do kontrolek HTML i zamianę reakcji użytkownika w akcje i aktualizację wartości. Pisanie takiej logiki wstaw/wyciągnij ręcznie jest uciążliwe, sprzyja błędom i jest koszmarem przy czytaniu kodu co każdy doświadczony programista jQuery może potwierdzić.



Angular wspiera wiązanie danych, mechanizm wiążącyczęści szablonu z częściami komponentu. Dodajemy oznaczenia wiązania danych w szablonie HTML,żeby określić jak Angular ma łączyć obie strony.

Jak pokazano na diagramie, są cztery rodzaje składniwiązania danych.

Każdy z rodzajów ma kierunek: do drzewa DOM, od drzewa DOM, albo w obu kierunkach.

Wiązanie danych

Szablon komponentu `HeroListComponent` zawiera trzy rodzaje wiązania danych:

src/app/hero-list.component.html (wiązanie danych)

```
<li>{{hero.name}}</li>
<hero-detail [hero]="selectedHero"></hero-detail>
<li (click)="selectHero(hero)"></li>
```

- **wstawka/interpolacja** `{{hero.name}}` wyświetla wartość właściwości `hero.name` komponentu wewnątrz znacznika `<li>`,
- **wiązanie właściwości** `[hero]` przekazuje wartość `selectedHero` z komponentu rodzica `HeroListComponent` do właściwości `hero` komponentu dziecka (`HeroDetailComponent`),
- **wiązanie zdarzenia** `(click)` wywołuje metodę `selectHero` komponentu kiedy użytkownik kliknie na nazwę bohatera.

Wiązanie danych

Dwukierunkowe wiązanie danych jest ważnym, czwartym rodzajem, który łączy wiązanie właściwości i zdarzeń i jednym zapisie, używa dyrektywy `ngModel`. Przykład z szablonu komponentu `HeroDetailComponent`:

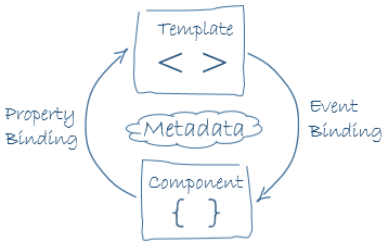
src/app/hero-detail.component.html (ngModel)

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

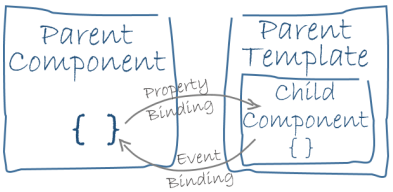
W dwukierunkowym wiązaniu danych wartość właściwości jest przekazywana z komponentu do kontrolki input box, tak, jak w wiązaniu właściwości. Zmiany dokonywane przez użytkownika są przekazywane z powrotem do komponentu ustawiając właściwość na najnowszą wartość, tak, jak przy wiązaniu zdarzeń.

Wiązanie danych

Angular przetwarza wszystkie związane dane jeden raz na cykl zdarzeń JavaScript (pętla zdarzeń, zob. np. The JavaScript Event Loop: Explained) zaczynając do korzenia drzewa komponentów całej aplikacji aż do wszystkich komponentów dzieci.



Wiązanie danych odgrywa ważną rolę w komunikacji pomiędzy szablonem a jego komponentem.



Wiązanie danych jest również ważne dla komunikacji pomiędzy komponentami rodzica i dziecka.

1.6 dyrektywy

Dyrektywy

Szablony Angulara są dynamiczne. Podczas ich renderowania Angular przetwarza DOM zgodnie z instrukcjami reprezentowanymi przez dyrektywy.

Dyrektywa jest klasą opatrzoną dekoratorem `@Directive`. Komponent jest *dyrektywą z szablonem*; dekorator `@Component`jest w rzeczywistości dekoratorem `@Directive` rozszerzonymo własności związane z szablonem.

Technicznie rzecz biorąc komponent to dyrektywa ale ponieważ komponenty są charakterystycznie i kluczowe dla aplikacji Angulara, to zostały wyróżnione jako osobny element architektury.

Istnieją jeszcze dwa inne rodzaje dyrektyw: dyrektywy **strukturalne** i **atrybutowe**.

Zwykle występują wewnątrz znacznika tak jak atrybuty, czasem jako jego nazwa ale częściej jako cel przypisania albo jako wiązanie.

Dyrektywy strukturalne

Dyrektywy strukturalne zmieniają układ graficzy poprzez dodawanie, usuwanie i zamianę elementów w drzewie DOM.

Przykładowy szablon używa dwóch wbudowanych dyrektyw strukturalnych:

src/app/hero-list.component.html (structural)

```
<li *ngFor="let hero of heroes"></li>
<hero-detail *ngIf="selectedHero"></hero-detail>
```

- `*ngFor` nakazuje Angularowi wygenerować jeden element `<li>` na bohatera z listy bohaterów.
- `*ngIf` uwzględnia komponent `HeroDetail` tylko jeśli wybrany bohater istnieje.

Dyrektywy atrybutowe

Dyrektywy atrybutowe zmieniają wygląd lub zachowanie istniejących elementów. W szablonie wyglądają one jak zwykłe atrybuty znaczników HTML, stąd ich nazwa.

Dyrektywa `ngModel`, która implementuje dwukierunkowe wiązanie danych, jest przykładem dyrektywy atrybutowej. `ngModel` modyfikuje zachowanie istniejącego elementu (zwykle `<input>`) poprzez ustawianie jego właściwości value i reagowanie na zdarzenia zmiany.

src/app/hero-detail.component.html (ngModel)

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

Angular posiada trochę więcej dyrektyw, które albo zmieniają strukturę układu (na przykład `ngSwitch`) albo modyfikują pewne aspekty elementów drzewa DOM i komponentów (na przykład `ngStyle` i `ngClass`).

Oczywiście możemy pisać nasze własne dyrektywy. Komponenty, jak na przykład `HeroListComponent` są jedny z rodzajów własnych dyrektyw.

1.7 usługi

Usługi

`Usługa` to szeroka kategoria obejmująca dowolną wartość, funkcję czy cechę, której potrzebuje nasza aplikacja.

Niemal wszystko może być usługą. Usługa jest zwykle klasąpowstałą w jakimś wąskim, dobrze zdefiniowanym celu.

Powinna robić coś szczególnego i powinna robić to dobrze.

Kilka przykładów:

- usługa logowania,
- usługa dostępu do danych,
- przekazywanie wiadomości między aplikacjami (message bus)
- kalkulator podatkowy,
- konfiguracja aplikacji.

Nie ma nic szczególnego w Angularze co by dotyczyło usług. Angular nie ma definicji usługi. Nie ma klasy bazowej dla usług ani miejsca rejestracji usług.

Mimo to, usługi są podstawowe dla każdej aplikacji Angular. Komponenty są dużymi odbiorcami usług.

Usługi

Przykład klasy usługi, która zapisuje logi w konsoli przeglądarki:

src/app/logger.service.ts (class)

```
export class Logger {
  log(msg: any)   { console.log(msg); }
  error(msg: any) { console.error(msg); }
  warn(msg: any)  { console.warn(msg); }
}
```

Usługi

Kolejny przykład to usługa `HeroService` używająca *obietnic* w celu obsługi pobierania bohaterów. Usługa `HeroService` zależy od usługi `Logger` i innej usługi `BackendService`, która odpowiada za komunikację z serwerem.

src/app/hero.service.ts (class)

```
export class HeroService {
  private heroes: Hero[] = [];

  constructor(
    private backend: BackendService,
    private logger: Logger) { }

  getHeroes() {
    this.backend.getAll(Hero).then( (heroes: Hero[]) => {
      this.logger.log(`Fetched ${heroes.length} heroes.`);
      this.heroes.push(...heroes); // fill cache
    });
    return this.heroes;
  }
}
```

Usługi są wszędzie.

Usługi

- Klasy komponentów powinny być szczupłe, bez nadmiaru (kodu, funkcjonaności). Nie pobierają danych z serwera, walidują danych wejściowych od użytkownika czy zapisują logi bezpośrednio do konsoli. Takie zadania są delegowane do usług.
- Zadaniem komponentu jest udostępnienie użytkownikowi danej funkcjonalności i nic ponad to.
- Komponent pośredniczy pomiędzy widokiem (renderowanym na bazie szablonu) i logiką aplikacji (która często zawiera jakąś wiedzę o modelu).
- Dobry komponent prezentuje właściwości i metody do wiązania danych. Wszystkie nietrywialne zadania są delegowane do usług.

- Angular nie narzuca tych zasad. Nie będzie narzekał kiedy napiszemy komponent "zlew kuchenny" z 3000 liniami kodu.
- Angular pomaga nam *przestrzegać* tych wytycznych poprzez ułatwianie nam podzielenia logiki aplikacji na usługi i uczynienie tych usług dostępnymi w komponentach poprzez *wstrzykiwanie zależności*.

1.8 wstrzykiwanie zależności

Wstrzykiwanie zależności

Wstrzykiwanie zależności jest sposobem na dostarczenienowej instancji klasy razem ze wszystkimi zależnościami, które są jej potrzebne. Większość zależności to usługi. Angular używa wstrzykiwania zależności w celu dostarczenianowych komponentuów razem z usługami, których potrzebują.

Angular rozpoznaje, których usług potrzebuje komponent sprawdzając na typy parametrów jego konstruktora. Dla przykładu, konstruktor komponentu `HeroListComponent` potrzebuje usługi `HeroService`:

src/app/hero-list.component.ts (constructor)

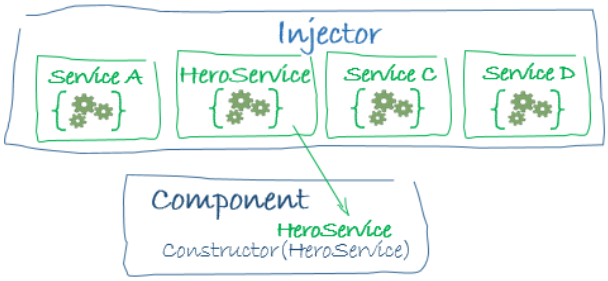
```
constructor(private service: HeroService) { }
```

Kiedy Angular tworzy komponent, najpierw zgłasza do narzędzia wstrzykującego zapotrzebowanie na usługi, których ten komponent wymaga.

Mechanizm wstrzykujący zarządza zasobem instancji usług, które wcześniej utworzył. Jeśli instancja żądanej usługi nie jest w zasobach, mechanizm wstrzykiwania tworzy jedną i dodaje ją do swoich zasobów zanim zwróci usługę do Angulara. Kiedy wszystkie żądane usługi zostaną rozpoznane i zwrócone, Angular może wywołać konstruktor komponentu z tymi usługami przekazanymi jako argumenty. To jest właśnie wstrzykiwanie zależności.

Wstrzykiwanie zależności

Proces wstrzykiwania usługi `HeroService` wygląda mniej więcej tak:



Jeśli mechanizm wstrzykujący nie posiada usługi `HeroService`, to skąd wie jak ją stworzyć?

W skrócie, wcześniej musimy zarejestrować dostawcę uslugi `HeroService` do mechanizmu wstrzykiwania. Dostawca usługi jest czymś co może stworzyć albo zwrócić usługę, zazwyczaj samą klasę usługi.

Dostawcę usług można rejestrować w modułach albo w komponentach.

Generalnie rzecz biorąc, dodajemy dostawcę do głównego modułu, dzięki czemu ta sama instancja usługi jest dostępna wszędzie.

src/app/app.module.ts (module providers)

```
providers: [
  BackendService,
  HeroService,
  Logger
],
```

2 Podsumowanie

3 Źródła

Źródła

- https://angular.io/
- https://pl.wikipedia.org/wiki/TypeScript
- https://www.typescriptlang.org/
- http://codeguru.geekclub.pl/baza-wiedzy/wprowadzenie-do-programowania-w-typescript-wst 3575
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions