



CENTRUM SZKOLENIOWE COMARCH

SZKOLENIE: Angular – kurs zaawansowany

Ryszard Brzegowy

COMARCH

Programowanie reaktywne

- Naturalny rozwój programowania asynchronicznego w JS/TS: callbacks -> event listeners -> promises + async/await -> observables
- Programowanie reaktywne bazuje nie na wartościach, ale na strumieniach wartości
- Kod jest powiadamiany i reaguje na kolejne wartości przychodzące w strumieniu danych
- Angular do komunikacji i wymiany danych za pomocą strumieni wykorzystuje bibliotekę RxJS
- W oparciu o obiekty Observables działa wiele wbudowanych w Angular kluczowych serwisów i obiektów, m.in:
 - HttpClient,
 - ReactiveForms
 - Router
 - EventEmitter

Programowanie reaktywne

- Podstawowe klasy tworzące strumienie (i emitujące wartości) to Observable, Subject, BehaviorSubject, ReplaySubject, AsyncSubject. Dostępne są również dodatkowe funkcje tworzące strumienie z tablic, zdarzeń obietnic czy np. emitujące wartości co określony czas.
- Do pracy ze strumieniami RxJS oddaje szeroki zestaw dodatkowych operatorów. Np. mapowanie danych, filtrowanie, opóźnianie wartości, wybieranie tylko niektórych wartości ze strumienia, łączenie wielu strumieni itd.
- KONIECZNIE należy pamiętać o zakończeniu subskrypcji gdy już jej nie potrzebujemy
- Observable może wyemitować wartość (next), błąd (error), lub zakończyć się (complete).

RxJS - naming conventions

- z \$ lub bez. jak projekt woli.
- kolekcje danych: jokes\$,
- pojedyncze wartości: user\$
- akcja na wartość ze strumienia: animateCircle\$

- plusy \$
 - easy to see (huuuge)
 - wygodnie jak pobieramy wartość do zmiennej: user\$ --> user
- minusy \$
 - jak nazwiesz tablicę observabli?
 - dlaczego nie dajesz suffixu do np. promise, funkcji lub sub-a?
 - pójdziemy w kolejne suffixy? To już było (hungarian notation)!
 - (nie)wygoda pisanie
 - czy jeśli funkcja zwraca Observable to też powinna mieć sufiks \$?
 - czy rozróżniamy w \$ Obs od Subj, BehSub itd?

- nie pytaj czy funkcja asynchroniczna powinna mieć prefix async... This. is. war.

RxJS - subskrypcja do emitera

- `Observable.subscribe()`
- `pipe async`
- `operatory` - `share`, `merge`, `zip`, `connect`, [...]

RxJS - Hot & cold observables

- Cold Observable - tworzy i uruchamia producera dopiero w momencie pojawienia się obserwatora (czyli subskrypcji).
- Przykładem Cold Observable jest `new Observable()`.
- Każdy sub do cold tworzy i uruchamia nowy producer
- Bolesnym przykładem w angularze może być `HttpClient`
- Cold observable jest emiterem unicastowym

RxJS - Hot & cold observables

- Hot Observable - producer pracuje niezależnie od subskrypcji Observabla
- Przykłady: Subject, BehaviorSubject, ReplaySubject, AsyncSubject
- Wszyscy subskrybenci dostają tę samą wartość - jeden producer jest współdzielony
- Hot observable to (prawie zawsze) emiter multicast
- Najczęstszy problem z hot - brak dostępu do wartości przed zapisaniem się (rozwiązanie: operatory)

RxJS - Hot & cold observables

- Konwersja Cold->Hot: operatory connect, share
- Konwersja w poprzednich wersjach RxJS: operatory publish (+refCount), publishBehavior, publishReplay, publishLast - wszystkie wylatują w RxJS8
- Co do zasady - wolimy multicasty, zbyt łatwo przypadkowo zapisać się ponownie do unicastu

RxJS - operator

- Operatory to funkcje które pobierają Observable(s) i zwracają *nowy* Observable
- Pipeable operators - wywołujemy je w .pipe()
- Creation operators - używamy jak zwykłej funkcji.

- Operatory filtrujące: first, last, elementAt, skip, filter, sample, debounce(Time), distinctUntilChanged, take(While/Until), every, find(Index)
- Operatory transformujące: map, pluck, scan, reduce
- Operatory tworzące: from, of, fromEvent, interval, timer, generate,
- Obsługa błędów: catchError, retry(When)
- Operatory pomocnicze: tap, delay, timeout(With), toArray
- Aktualna lista: <https://rxjs.dev/guide/operators>

RxJS - **operatory jako funkcje** i **operatory pipeable** pracujące na wielu strumieniach

- **race**, **raceWith** - zwraca pierwszy ze strumieni który zrobił next | err | complete
- **zip**, **zipWith** - sekwencyjnie i jednocześnie emituj wartości z każdego źródła w postaci tablicy
- **combineLatest**, **combineLatestWith** - dla każdej emisji z dowolnego źródła wyemituj komplet ostatnich wartości
- **forkJoin** - wyemituj ostatnie wartości gdy wszystkie observables się zakończą
- **merge** - złącz emisje w jeden strumień
- **concat**, **concatWith** - po zakończeniu pierwszego observabla emituj drugi, następnie kolejny itd
- **switchMap**, **mergeMap** - dla każdej emisji A, zapisz się do B

RxJS - własne operatory

- Na podstawie pipe()
Najczęstszy case - mamy zestaw operatorów w pipe() który jest wspólny dla wielu observabli
Przykład: rxjs/hot-cold -> btnPipe
- Od zera. Tylko po co:)
Funkcja musi implementować interfejs
(obs: Observable<T>) => Observable

Należy pamiętać o pełnej implementacji Observable (next, error, complete)

RxJS - unsubscribe

- `.unsubscribe()`
- async pipe
- `.complete()` na observable
- własny dekorator na właściwości klasy lub na klasę (tricky)
- operatory `take`, `takeWhile`, `takeUntil`, `first` - śliska sprawa
- `Subscription.add()/unsubscribe`
- Gdzie nie musisz:
 - gdy observable się kończy
 - gdy robisz sub-a na poziomie „root” (np. `AppComponent`, `AppModule`, serwisy `providedIn: root`)
 - ...ale łatwo zapomnieć o `.unsubscribe()` przy refaktorze:(

RxJS - helper

- <https://rxjs-dev.firebaseapp.com/operator-decision-tree>
Uwaga: część operatorów w sugestiach jest już oznaczona jako deprecated

Reagowanie na zmiany Inputów

- Mamy trzy @Inputy A, B, C. Chcesz mieć zawsze aktualną zmienną $D = A + B + C$. Co zrobisz?

Routing - podstawy

- Angular obsługuje dwie strategie routingu: HashLocationStrategy i PathLocationStrategy. Nie wiem kto używa pierwszej;)
 - HashLocationStrategy: <https://localhost/#users/list/2022>
 - PathLocationStrategy: <https://localhost/users/lists/2022> (domyślny)
- Moduł routingu dostarcza dwa pomocnicze serwisy do pracy z routingiem: Router i ActivatedRoute. Przydaje się też serwis Location.
- Miejsce „zaczepienia” routingu w html-u: <router-outlet></router-outlet>
- Link w html-u:
<a [routerLink]=„[,„user”, 12]”>Profil
Profil
- Bazą do routingu jest <base href=„/”> w index.html (lub token APP_BASE_HREF)

Routing - ścieżki

- Zaczepianie routingu w kodzie:
 - w głównym module: `imports[RouterModule.forRoot(routes),`
 - w pozostałych modułach: `imports[RouterModule.forChild(routes)]`
- Możliwe trasy:
 - komponent
 - moduł (lazy i immediate)
 - przekierowania
 - wildcard (404)
 - dzieciaki
- Dopasowanie route do url-a: first matching win, pathMatch: full/prefix,
- Parametry w url-u, dostęp do parametrów rodzica
- Zagnieżdżony routing
 - children: []
 - osobny moduł

Routing - parametry RouterModule

- Route nie działa? Debugowanie routingu: enableTracing
- HashRoutingStrategy: useHash
- Czy odświeżać widok przy przejściu na ten sam route: onSameUrlNavigation
- Czy przekazywać parametry rodzica do dziecka: paramsInheritanceStrategy
- Kiedy zmieniać URL w przeglądarce: urlUpdateStrategy
- Preloading: preloadingStrategy

Routing - Router, ActivatedRoute

- Parametry w routingu
 - `ActivatedRoute.queryParams`, `.params/.paramMap`, `.fragment`, `.data`
- Tytuły stron
 - `TitleStrategy` - nowość w Angular 14
 - `title` może być stringiem lub resolverem
- Router ma swój lifecycle (`Router.events`). Wykorzystanie
 - `analytics`
 - `loader`
 - reagowanie na anulowanie routingu

Routing - <router-outlet>

- Multi router-outlet dla nazwanych router-outlet (właściwość name)
- Interfejs RouteReuseStrategy. Domyślna strategia to klasa BaseRouteReuseStrategy
- <router-outlet> posiada zdarzenia! Activate, deactivate, attach, detach
- Przykłady wykorzystania zdarzeń
 - komunikacja rodzic<->dziecko bez serwisów
 - animacje w rodzicu

Routing - guards

- Rodzaje
 - CanActivate
 - CanActivateChild
 - CanDeactivate
 - CanLoad
- Guard powinien zwrócić: boolean|Promise|Observable|UrlTree
- Route może mieć wiele guardów z każdego rodzaju
- Parametryzacja guarda (np. z route data)
- ComponentLess routes +children dla wspólnych guardów

Resolver

- Interfejs dla klas które pełnią rolę dostarczenia określonych danych
- Resolve to jeden z kroków routingu (podobnie jak guards)
- Resolver można użyć w każdym typie Route
 - dla modułu
 - dla komponentu
 - dla dzieci (route z children, bez komponentu)

Routing - nawigacja

- `<a [routerLink]=„”>`
- z kodu metoda `.navigate()` serwisu Router
- serwis Location, metody `.go`, `.replaceState`, `.forward`, **`.back`**
Location z `@angular/common` (nie `window.location`)!

Routing - Preloading, prefetching danych

- Lazy lub eager load na route,
- Czarno-biały preload modułów: Router option preloadingStrategy
- CustomPreloadStrategy zamiast PreloadAllModules
- Prefetch danych
 - resolver
 - własny mechanizm prefetch (np. po najechniu myszą na przycisk „więcej”

Formularze - Reactive Forms

- First-choice w pracy z formularzami w Angularze
- Modelowanie formularza następuje w kodzie TS, HTML jedynie odzwierciedla model
- Dowolnie skomplikowany formularz jest tworzony z użyciem trzech czterech obiektów: FormControl, FormArray, FormGroup, FormRecord
- Do budowania modeli formularzy jest przeznaczona dodatkowa klasa FormBuilder
- Pełna, reaktywna kontrola nad wartościami i zdarzeniami formularza
- Może być zastosowany do każdego formularza

Reactive Forms - Angular 14

- !!!! Typed Forms (Ang 14) - oparte o generyki, np. `FormControl<T>()`
- Nowy obiekt `FormRecord` - `FormGroup` z kontrolkami tego samego typu
- Domyślnie nowe obiekty są typowane i nullable.
- `ng update` robi back compatibility: `UntypedFormControl`, [...]
- Non nullable:
 - `new FormControl(„”, {nullable: true})`
 - nn form builder: `fb.nonNullable.group({})`
- Typowanie pozwala np. utworzyć interfejs dla `FormGroup`

Reactive Forms - walidacja

- Kolekcja Validators
- Walidatory synchroniczne i asynchroniczne
- Kontrolka może posiadać wiele walidatorów
- Walidatory można komponować: `Validators.compose([])`, `.composeAsync()`
- Własne walidatory synchroniczne/asynchroniczne
- Walidatory bazujące na wielu polach formularza

Reactive forms - własne kontrolki

- Po co własna kontrolka?
 - enkapsulacja bardziej złożonych funkcjonalności/UI kontrolki
 - reużywalność zestawu kontrolek
 - spójność zaawansowanych formularzy
 - łatwiejsze utrzymanie kodu
- Własny komponent z interface `ControlValueAccessor`
<https://angular.io/api/forms/ControlValueAccessor>
- Idea `DomainFormControl` (tak sobie nazwałem roboczo;))

Dekoratory @Host*, @View*, @Content*

- Wykorzystywane są w komponentach i dyrektywach
- @HostBinding - dostęp do właściwości hosta (komponentu)
- @HostListener - nasłuchiwanie na zdarzenia hosta
- @ViewChild - dostęp do elementu dostępnego w widoku
- @ViewChildren - kolekcja elementów z widoku
- @ContentChild - dostęp do elementu osadzone w widoku (z content projection)
- @ContentChildren - j/w do kolekcji

SharedModules

- Wrzucasz wszędzie CommonModule, ReactiveFormsModule, HttpClientModule, MaterialModule?
- Popularne w aplikacji dyrektywy/komponenty/pipe?
- Wrzucić wszystko do jednego modułu i importuj go wszędzie. Po prostu.
- Pamiętaj o eksporcie z modułu obiektów które są używane poza modulem
- Uwaga na DI i lazy loaded (tworzenie nowych instancji). Idea .forRoot() i forChild()
- albo w zamian: SCAM Modules (Single Component Angular Module)
<https://medium.com/marmicode/your-angular-module-is-a-scam-b4136ca3917b>

Standalone components

- DEVELOPER PREVIEW (Angular 14)!

Stan na wersję 14.0.0:

- Standalone komponent to komponent nie powiązany z żadnym modułem
- SC nie deklarujemy w sekcji `declarations[]` modułu
- Dekorator `@Component` zostaje rozbudowany o właściwości:
 - `standalone: true` // deklaracja SC
 - `providers: []` - jak w modułach
- Dla celów migracji SC można importować w modułach - tak jak moduły
- Można robić lazy load SC w routingu (tak jak moduły dotychczas)
- Routing zyskuje providers

Dynamicznie tworzone komponenty

- Zastosowanie - np. dynamiczne dashboardy, widgety
- Tworzenie:
 - Pobierz `ViewContainerRef` elementu w którym będzie tworzony dynamiczny komponent (za pomocą dyrektywy lub `@ViewChild`)
 - Użyj metody `VCF.createComponent(ComponentClass)`
 - `.createComponent` zwraca `componentRef` - Ref do instancji klasy
- Dostęp w TS do komponentu: `componentRef.instance`
- Czyszczenie pojemnika: `VCF.clear()`

HttpClient

- Biblioteka do komunikacji z zewnętrznymi zasobami za pomocą protokołu Http
- Oparta o typowane observables
- Udostępnia pełny zakres metod, w tym wykorzystywane w RESTful: .get, .post, .put, .patch, .delete
- HttpClient domyślnie pracuje z danymi w formacie JSON
- Dane inne niż json: options: {requestType: text|blob|...}
- HttpModule dostarcza zaawansowany mechanizm interceptorów - scentralizowanego przechwytywania żądań http i ewent. ich późniejszej modyfikacji

Interceptor

- Interceptor to klasa implementująca `HttpInterceptor`
- Interceptor przechwytuje wszystkie żądania i odpowiedzi do/z zewnętrznych zasobów
- Interceptory pracują w łańcuchu. Przetwarzają żądanie i przekazują do kolejnego interceptora (`next.handle`)
- Interceptor może pominąć kolejne interceptory
- Jeżeli interceptor modyfikuje request, to request musi zostać sklonowany przed `.next` (`req.clone`)
- `next.handle` zwraca `Observable<HttpEvent>` - tu się zapisujemy na `HttpResponse`
- Interceptory dostarczamy poprzez token `HTTP_INTERCEPTORS`

Interceptor - zastosowania

- Monitorowanie ruchu (analitika)
- Implementacja loader-a
- Zmiana nagłówków/parametrów wysyłanych requestow (np. dodawanie JWT)
- Zarządzanie autoryzacją (pobieranie tokena, refresh/revoke token w locie)
- Obsługa błędów requestow
- Cache

Komponenty typu Smart i Dumb/Presentational

- Smart - komponenty z zawartą logiką biznesową (bez/z małą warstwą prezentacji)
- Smarty mamy najczęściej na górze i w średniej warstwie drzewa komponentów
- Smarty realizują część logiki biznesowej, przekazują wyniki do prezentacji przez dumb. W drugą stronę odbierają eventy z dumb.
- Dumb/presentational - komponenty prezentacyjne. Bez logiki, pokrótce - obsługują inputy/outputy
- **Czy dumb może korzystać z serwisów? To zależy:)**
- Smart/Dumb nie rozdziela technicznej części programowania. Rozdziela odpowiedzialność komponentu.
- Czyli myśląc o smart/dumb nie myślimy: szybszy/wolniejszy, większy/mniejszy. Bardziej - tylko prezentuje albo robi biznes.

Komponenty - ChangeDetection Strategies

- Każdy komponent posiada swój ChangeDetector
- Do sprawdzania zmian Angular wykorzystuje bibliotekę Zone.js
- Domyślnie changeDetector to ChangeDetectionStrategy.Default (strategia checkAlways)
Może powodować problemy z wydajnością (szczególnie widoczne w UI)
- W trybie develop ChangeDetectionCycle następuje dwa razy (stąd wyłapywanie błędów ExpressionHasBeenChangedAfterItHasBeenChecked)
- W produkcji sprawdzanie zmian następuje jednokrotnie (m.in. stąd różnica w szybkości działania prod vs dev)
- Ale przynajmniej wiesz że jak na dev działa dobrze, to na prod wolniej nie będzie;)

Jak działa domyślna detekcja zmian?

- Zone.js ,monitoruje' addEventListenery (czyli eventy), api przeglądarki (np. fetch), setTimeout, setInterval (ale nie wszystkie api - dokumentacja zone.js)
- Angular wykorzystuje Zone.js aby wyzwolić cykl detekcji zmian (np. na ,click' event)
- Podczas cyklu ChangeDetector komponentu porównuje wartości właściwości komponentu (jedynie tych używanych w widoku). Obiekty testowane deep.
- Jeżeli ChangeDetector znajdzie zmiany - widok jest odświeżany

ChangeDetection - onPush

- OnPush (strategia checkOnce) - dużo bardziej wydajne (i restrykcyjne) rozwiązanie
- Porównanie obiektów następuje przez referencję nie wartości - wymaga przemyślanego podejścia do aktualizacji danych w komponencie
- Reaguje na zmiany @Input (referencje) i zdarzenia w komponencie
- Będzie reagował na nowe wartości w strumieniach (async)

- komponenty onPush karmimy Immutable data lub strumieniami
- Jeśli CD nie wykryje zmian o których wiem, możemy mu pomóc. Ręczne oznaczenie komponentu jako „dirty”: `cdRef.markForCheck()`

ChangeDetection - out of zone

- Całkowite odpięcie/przypięcie komponentu od CD:
ChangeDetectorRef.detach(), .attach(). To jest Hard. Serio.
- Ręczne wywoływanie CD:
ChangeDetectorRef.detectChanges(),
- Dalej działa DoCheck hook
- Własna strategia detekcji zmian: cdRef.detach() + DoCheck hook.
Mówią na mieście żeby nie dotykać;)

Wydajność - możliwości

- detach kawałka „ciężkiego” kodu: `zone.runOutsideAngular(() => {})`
 - przydatne dla zewnętrznych bibliotek ostro działających na DOM - animacje, wykresy, wysiwyg (zazwyczaj problemem są `setTimeout/Interval`, `requestAnimationFrame`, `addEventListener` - `mousemove`)
- długie obliczenia na zmiennych w widoku:
 - cache wcześniej wyliczonych wartości
 - pure pipe
- komponent: `ChangeDetectionStrategy.onPush`
- komponent: `cdRef.detach()`, `.reattach()`

Testy jednostkowe w Angular

- Otoczenie, narzędzia: [stateofjs](#)
- Kultura testów
- Czy biznes wie czego chce? TDD
- Code review testów
- Code coverage. 100%? Focus na jakości, później ilości
- Spróbuj inaczej - nie patrz na % całego kodu, ale procent pokrycia commita - tu zrób 95%:)
- Drabinka Google: we offer the general guidelines of 60% as “acceptable”, 75% as “commendable” and 90% as “exemplary.”
- Uwaga na sposób mierzenia code coverage. To że pokrywamy kod if-a, nie znaczy że testujemy jego wszystkie edge-cases.

Jasmine/Karma

Jasmine

- Javascriptowy framework testowy
- współpracuje z wszystkimi popularnymi bibliotekami do testów
- bazowe pakiety są instalowane automatycznie z nowym projektem
- wymaga test-runnera. W projektach angularowych domyślnie jest nim Karma
- inne popularne biblioteki/frameworki: mocha, jest, testing library

Karma

- Javascriptowy test runner

Testy - szybki start

- struktura folderów i nomenklatura: pliki .spec.ts
- Pojedynczy test set: describe, it, xit, xdescribe, fit, fdescribe
- Praca ze środowiskiem przed/po teście: beforeAll, beforeEach, afterAll, afterEach

Mockowanie angularowych modułów

- Testy są izolowane, nie ma inicjalizacji modułów.
- Moduł jest tworzony dynamicznie przez środowisko testowe za każdym razem od nowa.
- W konfiguracji konieczne jest dostarczenie wszystkich zależności, których wymaga testowany element
- Angular dostarcza mocki swoich bazowych modułów, np.:
 - HttpClient -> HttpClientTestingModule
 - RouterModule -> RouterTestingModule
- Uwaga na zależności angularowych serwisów - jeśli np. w teście brakuje ActivatedRoute - dołączamy moduł (w wersji dla testów) który ten serwis dostarcza (tutaj: RouterTestingModule)

Fixture w testach - przyspieszanie testów

- Fixture nie zawsze jest potrzebne. A jak nie potrzebujemy to nie chcemy żeby zabierało nam czas.
- W szczególności - nie potrzebujemy do wstrzykiwania serwisów oraz testowania logiki komponentów (bez widoków)
- Wstrzykiwanie bez fixture: `TestBed.inject(Dashboard)`
- Inicjowanie komponentu:
 - `TestBed.inject(RxjsComponent)`
 - wtedy trzeba dodać komponent do providers: `[RxjsComponent]` (tak jak serwisy - bo traktujemy go podobnie jak serwis)
- W powyższym tracimy lifecycle, np trzeba ręcznie odpalić `ngOnInit` na komponencie
- ...ale zyskujemy dużo czystsze i szybsze testy

Fixture - c.d.

- Możemy pójść dalej - nie używamy TestBed.inject() do tworzenia komponentu
- po prostu `const c = new RxjsComponent(myHeroMockService)`
- ..bo koniec końców testujemy KLASĘ komponentu. Nie widok.
- Będzie jeszcze szybciej:)
- Nie można tego zrobić jeśli korzystasz z modułów Angularowych (np. HttpClientModule, RouterModule - te są dostępne jedynie przez DI)
- Powyższe „obcinanie” testów angularowych to tzw. Isolated Unit Test
- Problem z raportem CodeCoverage - może nie pokazywać realnego pokrycia
- Ale nie cyferki są ważne;)

Testy - Visual regression tests


- Testy oparte o snapshoty html/png
- jasmine-snapshot / jest snapshot
- wychwytywanie zmian w wyglądzie
- różnice w wyglądzie w różnych przeglądarkach

Literatura przedmiotu

1. Dokumentacja: angular.io
2. angular-university.io (szczególnie blog)
3. angular.love

Ścieżka kształcenia

- [Szkolenie Angular NgRx. Reaktywny stan aplikacji Angularowej](#)



Centrum Szkoleniowe
ul. Prof. M.Życzkowskiego 23
31-864 Kraków
Tel. +48 (12) 687 78 11
E-Mail: mail.szukolenia@comarch.pl
www.szukolenia.comarch.pl

COMARCH