ANGULAR SIGNALS - NOWE PODEJŚCIE DO REAKTYWNOŚCI WE FRAMEWORKU ANGULAR

mgr inż. Ryszard Brzegowy

STRESZCZENIE

Framework Angular budowany i utrzymywany przez firmę Google to drugi z najczęściej wybieranych frameworków do budowy aplikacji webowych. W aplikacjach klasy enterprise, Angular jest uważany za pierwszy wybór z uwagi na nacisk na skalowalność i pewność utrzymania aplikacji. W niniejszym artykule omówiony jest nowy sposób pracy z danymi asynchronicznymi we frameworku Angular. Praca opisuje dotychczasowe rozwiązania używane do pracy z danymi dostarczanymi asynchronicznie, powody, dla których powstały Angular Signals, sposób ich implementacji oraz integracji we frameworku.
W dalszej części omawiany jest sposób pracy w środowisku mieszanym - wykorzystującym zarówno sygnały jak i strumienie. Jako ostatnie opisane zostały dobre praktyki w pracy z sygnałami i kierunki rozwoju aplikacji.

SŁOWA KLUCZOWE

Framework Angular, Frontend Development, Angular Signals, signal, computed, effect, untracked, programowanie reaktywne, programowanie deklaratywne, RxJS, strumienie, ZoneJS

Wprowadzenie

W 2009 roku pod palcami Misko Hevery'ego oraz Adama Abrosna, ówczesnych pracowników Google, narodził się nowy pomysł na tworzenie aplikacji frontendowych – framework AngularJS [1]. Framework oparty o wzorzec Model-View-Controller oraz oddzielający logikę biznesową od budowy i zarządzania interfejsem użytkownika, wzbudził duże zainteresowanie rynku jako podstawa do budowy rozbudowanych aplikacji webowych.

W 2014 na bazie dobrych pomysłów projektowych i popularności AngularJS, w ramach zespołu Google powstała druga wersja AngularJS oparta o Typescript. Nowa wersja zyskała nazwę Angular by ją odróżnić od pierwowzoru, z którym nie była kompatybilna. Rynek entuzjastycznie przyjął nowe rozwiązanie, framework Angular był pierwszym a aktualnie pozostaje drugim [2] [3] (po React) najczęściej wybieranym frameworkiem do budowy aplikacji webowych. Nieprzerwanie [4] od tego czasu framework Angular jest rozwijany w cyklu 6-miesięcznym, w momencie pisania tego artykułu aktualna jest wersja 18 frameworka.

Od samego początku rozwoju Angulara, Google oparł pracę z danymi asynchronicznymi o zewnętrzną bibliotekę RxJS [5]. Biblioteka opiera swoje działanie o wzorzec behawioralny Obserwator. RxJS implementuje wzorzec udostępniając do wykorzystania obiekty strumieni, do których można się zapisać by otrzymać zmieniające się synchronicznie lub asynchronicznie dane.

W wersji Angular 16 zespół Google po raz pierwszy (w oficjalnej dystrybucji) zaprezentował w formie Developer Preview ideę pracy z asynchronicznymi danymi opartą o Angular Signals. Od wersji 17 frameworka sygnały są integralną częścią frameworka i zalecanym przez zespół Google sposobem pracy z danymi asynchronicznymi. Wraz z kolejnymi wersjami frameworka, dodatkowe biblioteki nie wchodzące w skład rdzenia frameworka są sukcesywnie migrowane z integracji z biblioteką RxJS na Angular Signals.

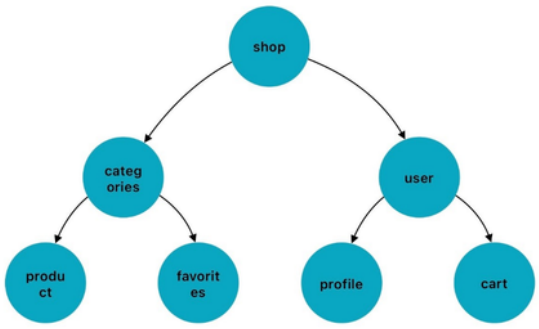
Dlaczego Angular potrzebuje Signals?

Aby odpowiedzieć na to pytanie należy przybliżyć działanie mechanizmu detekcji zmian w widoku komponentów aplikacji. Każdy komponent zbudowany jest z trzech podstawowych elementów: klasy komponentu (Typescript), stylów CSS oraz widoku komponentu (HTML). Rolą komponentu jest komunikacja z użytkownikiem - wyświetlanie dostarczonych mu danych oraz reagowanie na zdarzenia generowane przez środowisko (np. przeglądarkę) oraz samego użytkownika (np. wypełnienie formularza).

Przykładowy widok komponentu wraz z bindingiem (przypisaniem) danych z komponentu do widoku:

```
<h2>Koszyk zakupów</h2>
<section>
  Zawartość:
  <ul>
    @for (product of shoppingCart.items; track $index) {
      <li>{{ product.name }} - {{ product.price }} zł</li>
    } @empty {
      <li>Pusto tu:(</li>
    }
  </ul>
  <p>Dostawa: {{ shoppingCart.shippingCost }}</p>
  <p>Rabat: {{ shoppingCart.discounts }}</p>
  <p>Do zapłaty: {{ shoppingCart.totalToPay }} zł</p>
  <button (click)="onCheckout()">Złóż zamówienie</button>
</section>
```

Aby synchronizacja danych została osiągnięta i zachowana przy późniejszych zmianach wartości, Angular wprowadził ideę Change Detectora. Change Detector to obiekt, którego rolą jest sprawdzenie, czy dane renderowane w widoku uległy zmianie. Jeśli tak się stało - Change Detector oznacza część widoku jako konieczną do ponownego renderowania. Każdy komponent posiada swój własny Change Detector, dodatkowo wszystkie Change Detectory tworzą drzewo detektorów analogicznie do drzewa komponentów.



Każdy z Change Detectors (nazywanych dalej CD), aby wykryć potencjalne możliwości braku synchronizacji danych, korzysta z zewnętrznej biblioteki ZoneJS. Biblioteka ta "obserwuje" wszystkie zdarzenia asynchroniczne generowane przez środowisko (np. pobranie danych z zewnętrznego zasobu) jak i wszystkie zdarzenia DOM, które obsługujemy w komponencie (np. kliknięcie w przycisk). Jeśli jakieś zdarzenie zostanie wygenerowane – ZoneJS powiadamia o tym fakcie CD. CD oznacza komponent jako "dirty" - wymagający sprawdzenia oraz powiadamia o tym fakcie wszystkie pozostałe CD [6].

Ten sposób pracy gwarantuje frameworkowi pełną informację o wszystkich potencjalnych zmianach w stanie projektu. Jednakże odbywa się to wysokim kosztem wielokrotnych, często niepotrzebnych cykli detekcji zmian – gdy potencjalne zmiany mogły zajść (np. użytkownik kliknął w przycisk), ale nie zaszły (kliknięcie w przycisk nie zmieniło wartości wyświetlanych w widoku komponentu).

Jak nietrudno wywnioskować - powyższy sposób aktualizacji widoków komponentu nie prezentował najbardziej wydajnego sposobu detekcji zmian, był to raczej sposób najbezpieczniejszy.

Bardziej życiowo - wyobraźmy sobie Urząd Wydający Ważny Dokument (ZoneJS), który za każdym nowo utworzonym dokumentem dzwoni do czekającego na dokumenty Petenta (ChangeDetector) by poinformować, że został zmieniony stan zgłoszenia lub też został wygenerowany nowy dokument (niekoniecznie Petenta, być może jednego z tysięcy innych Petentów). Z wszystkich przekazanych informacji naszego Petenta interesuje tylko jedna – dokument czeka na odbiór. Podobnie jest z ZoneJS i ChangeDetectorem -

– ten ostatni wolałby otrzymać jedynie informację o zdarzeniach, które faktycznie wpłynęły na zawartość szablonu komponentu. W idealnym przypadku Change Detector nie powinien być informowany co spowodowało zmianę (to nie jest istotne z punktu widzenia renderowania komponentu), ale co się zmieniło.

Opisany powyżej problem jest jednym z głównych aspektów działania frameworka. Aspekt ten zaadresowany i rozwiązany przez nowe Angular Signals.

Czym są i jak działają Angular Signals?

Podstawowym typem sygnału dostępnym w Angularze jest obiekt typu *WritableSignal*. Tworzenie, odczyt i zapis do obiektu opiera się o Api z interfejsu *WritableSignal*:

```
// tworzenie WritableSignal
  protected readonly user = signal<User | null>(null)

  // dostęp do wartości WritableSignal
  getUserName() {
    return this.user()?.name
  }

  // aktualizacja wartości WritableSignal
  setUser(user: User) {
    this.user.set(user)
    // lub
    this.user.update(prevValue => user)
  }
```

Dzięki powyższemu Api sygnał nie tylko jest w stanie przechować dane, ale co równie istotne - zostaje powiadomiony o zachodzącej zmianie. Ta informacja może zostać przekazana dalej do wszystkich subskrybentów sygnału.

Wykorzystanie sygnałów jako źródła informacji (subskrypcja) w szablonie komponentu:

```
<h2>Profil użytkownika</h2>
<p>Imię: {{ user()?.name}}</p>
<p>E-mail: {{ user()?.email}}</p>

```

Z punktu widzenia pracy CD istotny jest dla nas szczególnie odczyt danych z sygnałów bezpośrednio w widoku komponentu. Następuje tu fundamentalny przeskok w sposobie informowania CD o zmianach zachodzących w stanie komponentu/aplikacji.

Wcześniej - ZoneJS informował CD o zdarzeniu, które mogło (ale nie musiało) prowadzić do zmiany stanu. Teraz – widok komponentu poprzez użycie gettera Signal (przykład z grafiki: user()) zapisuje się do zmian zachodzących w wartościach poszczególnych sygnałów. Oznacza to, że każdorazowo, gdy wartość przechowywana w sygnale się zmieni - CD zostanie poinformowany o zachodzącej zmianie.

Następstwem powyższych zmian w architekturze przepływu informacji jest, w dalszej perspektywie Angulara [7], możliwość całkowitej rezygnacji z biblioteki ZoneJs i przejście na schemat subskrypcji do danych dostarczanych przez Signals.

Zmiana ta umożliwia znacznie wydajniejsze renderowanie widoków komponentu bez rezygnacji z pewności otrzymania informacji o zmianie stanu. Dodatkowym zyskiem jest usunięcie zewnętrznej biblioteki ZoneJS z zależności projektu.

Praca z danymi na bazie Signals

Computed

Funkcja computed() pozwala utworzyć sygnał tylko do odczytu (typu Signal) bazujący na innych sygnałach:

```
totalToPayInEur = computed(() => {
    return this.shoppingCart().totalToPay / EurRate
  })
```

Bardzo istotną cechą sygnałów na bazie computed jest jedno źródło informacji - wartość w takim sygnale może być zmieniona jedynie poprzez dostarczoną w argumencie funkcję. Znacząco utrudnia to możliwość zasilenia sygnału z innych niż przewidziane początkowo źródeł (co często prowadzi do tzw. “spaghetti code”).
Wartości przechowywane w Signal podlegają memoizacji [8], tak więc kolejne wyliczenie wartości nastąpi dopiero po zmianie któregokolwiek z sygnałów źródłowych. Wyliczanie wartości odbywa się leniwie (dopiero po zmianie sygnałów źródłowych i subskrypcji do sygnału). Oznacza to, że w szczególnym przypadku wyliczenie może nie wystąpić w ogóle.
Cechą charakterystyczną sygnałów pozostaje również fakt, że wyliczenie zawsze odbywa się asynchronicznie. Fakt ten może (nie musi!) stanowić problem przy próbie zmiany wartości w sposób synchroniczny:

```
promoCoins = signal(1)
promoCoinsDiscount = computed(() => {
  console.log('Zmienił się stan monet')
  this.promoCoins() > 10
    ? 'Zyskujesz 10% rabatu'
    : 'Jeszcze trochę brakuje!'
})
addPromoCoins() {
  this.promoCoins.update(coins => coins + 1)
  this.promoCoins.update(coins => coins + 1)
  this.promoCoins.update(coins => coins + 1)
}
```

Po uruchomieniu metody addPromoCoins() nastąpi trzykrotna aktualizacja sygnału promoCoins . Sygnał promoCoins będzie posiadał informację o każdorazowej synchronicznej zmianie [9] , jednak informacja o zmianie wartości sygnału zostanie rozpropagowana asynchronicznie (czyli po wszystkich synchronicznych zmianach wartości). Dlatego też aktualizacja wartości sygnału promoCoinsDiscount nastąpi tylko raz.

Effect

Funkcja *effect()* pozwala na zdefiniowanie reakcji na zmiany w sygnale/sygnałach.
W odróżnieniu od *computed()*, efekt co do zasady nie tworzy nowych sygnałów i nie zmienia wartości sygnałów już istniejących.
W uzasadnionych przypadkach możliwa jest zmiana wartości sygnału wewnątrz efektu, jednak należy na tę, operację zezwolić za pomocą opcji *allowSignalWrites*:

```
promoCoinsLogger = effect(() => {
    if (this.promoCoins() > 100) {
      console.log('Ojj, chyba cheat!')
      this.promoCoins.set(0)
    }
  }, { allowSignalWrites: true })
```

Do zmian wartości sygnałów wewnątrz computed() należy podchodzić szczególnie ostrożnie z uwagi na możliwość wywołania circular dependency (efekt zmienia sygnał, który ponownie wyzwala efekt, który zmienia sygnał itd.)
Efekty przewidziane są do wykonywania scenariuszy bazujących na sygnałach, np.:

1. Zapis do lokalnej bazy danych
2. Logowanie zmian
3. Zmiany w strukturze DOM, które nie mogą bezpośrednio korzystać z mechanizmu binding, np. rysowanie w <canvas>.

Untracked

Funkcja untracked() może być stosowana wewnątrz computed() oraz effect(). Untracked rozwiązuje problem zbyt częstego wyzwalania computed/effect przez jeden lub więcej bazowych sygnałów:

W powyższym przykładzie sygnał tick zmienia swoją wartość co 100ms, sygnał trafficLight co trzy sekundy. Projekt funkcjonalności zakłada, że sygnał trafficLightChangeTick powinien być wyliczony ponownie na każdą zmianę sygnału trafficLight. Ponieważ computed() jest wyzwalane na każdą zmianę każdego z sygnałów źródłowych, spowoduje to ponowne wyliczenie sygnału trafficLightChangeTick co 100ms (wyzwalaczem będzie sygnał tick) zamiast co trzy sekundy (sygnał trafficLight).
By zapobiec powyższemu problemowi należy zastosować dodatkową funkcję untracked(). Jej zastosowanie w sygnale trafficLightChangeTickUntracked zablokuje wyzwalanie przeliczania computed przez sygnały umieszczone wewnątrz untracked . Wyzwalanie przeliczenia będzie następowało jedynie dla zmian w trafficLight , przy każdym przeliczeniu untracked zwróci aktualną wartość sygnału tick .

Angular Signals i RxJS

W aplikacjach opartych o framework Angular, biblioteka RxJS pozostaje bazowym sposobem na przechowywanie stanu aplikacji. Jest też w dalszym ciągu używana do dostarczania danych asynchronicznych przez dodatkowe biblioteki Angulara (np. Router. HttpClient czy ReactiveForms).

Należy zaznaczyć, że Angular Signals nie powstało jako bezpośredni zamiennik dotychczas stosowanych strumieni, możliwości Signals są mocno ograniczone w porównaniu do rozbudowanego systemu strumieni i operatorów w RxJS.

W przypadkach, gdy konieczna jest konwersja sygnałów do strumieni (lub odwrotnie) można wykorzystać metody toSignal(observable) oraz toObservable(signal).

```
const counter = signal(0)
const observableFromCounter = toObservable(counter)

const timer = new Subject()
const signalFromTimer = toSignal(timer, { initialValue: 1 })

const uploadedCount = new BehaviorSubject(10)
const signalFromUploadedCount = toSignal(uploadedCount, { requireSync: true })
```

Jak widać na załączonej grafice, samo stosowanie metod konwertujących jest trywialne. Koniecznie jednak trzeba zwrócić szczególną uwagę na konwersję strumieni do sygnałów. Te dwa pojemniki różnią się wymogiem posiadania wartości (każdy sygnał zawsze posiada wartość, strumień niekoniecznie).

Może to doprowadzić do dwóch potencjalnie niezgodnych sytuacji:

1. Strumień źródłowy nie posiada wartości. Ten przypadek można rozwiązać stosując dodatkową opcję konwersji intitialValue. W przypadku jej braku początkową wartością sygnału będzie undefined. Przykład z grafiki – signalFromTimer.
2. Strumień źródłowy posiada wartość i zwraca ją synchronicznie. Ten przypadek rozwiązujemy za pomocą opcji requiredSync. Opcja ta powoduje, że pierwsza wartość tworzonego sygnału pobierana jest synchronicznie przy zapisie do strumienia źródłowego.

Dobre praktyki w pracy z Angular Signals

Dostarczanie danych do widoku komponentu

Wszystkie wartości dostarczane do widoku komponentu powinny być dostarczane jako sygnały. W wyjątkowych sytuacjach (zastane środowisko, dane pochodzące ze strumieni) źródłem danych może być strumień subskrybowany pipem *async*. Dzięki takiej praktyce Change Detector będzie zawsze informowany o zmianach i nie będzie konieczne korzystanie z ZoneJS. Przygotuje to komponent do działania w środowisku z trybem detekcji zmian "zoneless".

Konwersja na sygnał strumieni tworzonych dynamicznie z parametrem

Relatywnie częstym przypadkiem w aplikacjach jest generowanie strumienia na bazie przekazywanych parametrów. Klasycznym przykładem może być tutaj klient http i jego metody:

```
getDataFromApi(url: string) {
    return this.httpClient.get(url)
}
```

W takim przypadku nie jest możliwa konwersja strumienia do sygnału za pomocą metody toSignal z uwagi na brak możliwości przekazania parametru (w przykładzie: url).

Są dwa możliwe rozwiązania problemu:

```
1  url = signal('someUrl')
2    // 1. efekt
3    readonly dataFromApi = signal<object | null>(null)
4    getWeather = effect(() => {
5      const url = this.url()
6      const data = this.getDataFromApi(url)
7      data.subscribe(data => this.dataFromApi.set(data))
8    }, { allowSignalWrites: true })
9
10   // 2. signal->toObservable->toSignal
11   dataFromApi2 = toSignal(toObservable(this.url)
12     .pipe(switchMap(url => this.getDataFromApi(url)))
13   )
```

Sposób pierwszy to efekt bazujący na sygnale url. Każda zmiana sygnału powoduje wywołanie metody getDataFromApi(), następnie subskrypcję do zwróconego strumienia i, jako efekt subskrypcji, zasilenie sygnału dataFromApi danymi. Metoda ta może być stosowana (w uproszczeniu) jedynie do strumieni, które zwracają tylko jedną wartość i zostają zakończone. W innych przypadkach metoda powinna zostać rozbudowana o mechanizm wypisywania się z potencjalnie wielu równoległych subskrypcji.

Druga metoda bazuje na podwójnej konwersji. Najpierw sygnał url jest konwertowany na strumień, następnie strumień zostaje przetworzony za pomocą operatora switchMap (który zarządza subskrypcją do wewnętrznego strumienia, może to być również inny operator), by na końcu znów dokonać konwersji strumienia zwróconego przez getDataFromApi() do sygnału. Ta metoda z uwagi na możliwość praktycznie dowolnego przetwarzania strumienia wewnętrznego pozostaje bezpieczna niezależnie od charakterystyki strumieni źródłowych.

Wykorzystanie signals we wszystkich źródłach danych komponentu

Komponenty frameworka Angular mogą być zasilane danymi na bazie:

1. Zewnętrznych serwisów. Jeżeli serwis dostarcza dane asynchroniczne może zwrócić bezpośrednio Signal lub strumień. Strumień może zostać skonwertowany do sygnału za pomocą funkcji toSignal()
2. Mechanizmu tzw. Inputs. Aby zapobiec konieczności tworzenia pojemników na sygnały i następnie zasilania ich wartościami w hook-u ngOnChanges [10] należy zastosować inputy oparte o sygnały:

```
1  // zamiast
2    @Input() url = undefined
3    // lepiej
4    url = input(undefined)
5
6    // zamiast
7    @Input() formValue = {}
8    @Output() formValueChange = new EventEmitter<object>()
9    // lepiej
10   formValule = model({})
```

Zarówno funkcja input() jak i model() zwracają sygnały (odpowiednio: InputSignal i ModelSignal). Pokazany w przykładzie input url pozostaje w komponencie sygnałem tylko do odczytu (jego ustawienie następuje wyłącznie w szablonie komponentu rodzica). FormValue łączy w sobie input i output wykorzystując metodę two-way data binding znaną z Template Forms w Angularze. Wartość może zostać ustawiona zarówno przez rodzica jak i bezpośrednio z użyciem signals api (.set.(0, update()). Dodatkowo, formValue może zostać potraktowany przez rodzica jako zwykły sygnał.

Blokowanie sygnałów oraz mutacji obiektów zwracanych przez sygnały

Rozważmy poniższy kod:

```
1
2  shoppingCart = signal<Cart | null>(null)
3
4  setCart(cart: Cart) {
5    this.shoppingCart.set(cart)
6  }
7  letdDoSomethingCool() {
8    const cart1 = this.shoppingCart()
9    const cart2 = this.shoppingCart()
10   cart1.totalToPay = 1000
11   console.log(cart2.totalToPay) // 1000!
12
13   // i dalej, stwórzmy nowy sygnał. Bo możemy:)
14   this.shoppingCart = signal(cart1)
15 }
```

Przedstawione zostały dwa problemy:

1. Mutacji obiektów serwowanych przez sygnały (linie 8 i 9)
2. Możliwość niezamierzonej zmiany obiektu sygnału poprzez jego ponowną deklarację (linia 14)

Rozwiązanie powyższych problemów:

```
1
2  readonly shoppingCart = signal<Cart | null>(null)
3
4  setCart(cart: Cart) {
5    const newCart = { ...cart } as const
6    this.shoppingCart.set(cart)
7  }
8  letdDoSomethingCool() {
9    const cart1 = this.shoppingCart()
10   const cart2 = this.shoppingCart()
11   cart1.totalToPay = 1000 // nie uda się
12
13
14   // już nie możemy:)
15   this.shoppingCart = signal(cart1)
16 }
```

Mutacja obiektu sygnału została zablokowana za pomocą operatora readonly (linia 2). Mutacje obiektów serwowanych przez sygnał zostały zablokowane przez oznaczenie obiektu przechowywanego w sygnale “as const” (linia 5).

Podsumowanie

Trzy ostatnie wersje frameworka Angular określane są często jako “renesans” Angulara. Jedną z fundamentalnych zmian stała się częściowa rezygnacja z natywnego wykorzystania biblioteki RxJS na rzecz Angular Signals. RxJS jako biblioteka dalej pozostaje mocno wykorzystywanym przez Angulara narzędziem, jednak w kluczowym dla Angulara obszarze – renderowania danych, nie jest już potrzebna.

Angular Signals przynoszą świeże (dla Angulara) i dobrze znane (dla osób piszących aplikacje w takich frameworkach jak Solid czy Svelte) spojrzenie na pracę z danymi asynchronicznymi. Jednak nie należy traktować sygnałów jako bezpośredniego następcy strumieni dostarczanych przez RxJS. Sygnały w aktualnej formie nie są

gotowe na taką zamianę (i przede wszystkim - nie do tego zostały zaprojektowane).

Sygnały stanowią aktualnie natywną część bazowej biblioteki frameworka. Umożliwia to głęboką integrację dostarczania danych asynchronicznych z pozostałymi mechanizmami frameworka. Integracja ta jest pogłębiana wraz z kolejnymi wersjami Angulara poprzez migrację kolejnych bibliotek Angulara.

Utworzenie Angular Signals stworzyło podwaliny do kolejnych optymalizacji – hybrydowego trybu detekcji zmian oraz docelowo – detekcji zmian bez użycia biblioteki ZoneJS. Również w wielu innych miejscach frameworka sygnały przynoszą znaczące uproszczenie i przyspieszenie działania kodu. Go Angular!

Załączniki

1. Repozytorium kodu prezentowanego w ramach artykułu: https://github.com/rbrzegowy/angular-signals-new-reactivity

Bibliografia

1. https://www.statista.com/statistics/1124699/worldwide-developer-survey-most-used-frameworks-web/
2. https://angular.dev/
3. https://github.com/angular/angular

[1] https://pl.wikipedia.org/wiki/Angular_(framework)
[2] https://2023.stateofjs.com/en-US/libraries/front-end-frameworks/
[3] Źródło: https://www.statista.com/statistics/1124699/worldwide-developer-survey-most-used-frameworks-web/
[4] Z kronikarskiego obowiązku należy wspomnieć, że została pominięta wersja Angular 3 frameworka (z uwagi na wykorzystywaną już wcześniej wersję 3 w module RouterModule).
[5] https://rxjs.dev
[6] W przypadku gdy CD korzysta z trybu OnPush informacja jest propagowana jedynie do gałęzi projektu, w której znajduje się CD. W przypadku gdy CD został odłączony od drzewa pozostałych CD – informacja nie jest propagowana.
[7] W Angular18 tryb zoneless jest już dostępny w formie eksperymentalnej
[8] Poprzednia wyliczona wartość jest zapamiętywana i zwracana dla każdego subskrybenta.
[9] Można to sprawdzić za pomocą opcjonalnego parametru equalFn – funkcji porównującej poprzednią i nową wartość sygnału. Funkcja zostanie wyzwolona tyle razy, ile razy nastąpi zmiana sygnału.
[10] Metoda ngOnChanges jest uruchamiana w komponencie każdorazowo przy zmianie wartości któregokolwiek z inputów.