Rozszerzona Dokumentacja Operatorów RxJS

Kacper Renkel

August 11, 2024

Contents

1 Wprowadzenie

RxJS (Reactive Extensions for JavaScript) to biblioteka umożliwiajaca programowanie reaktywne. W tym przewodniku szczegółowo omówione sa operatory RxJS, które umożliwiaja tworzenie, transformowanie, filtrowanie i zarzadzanie strumieniami danych.

2 Operatory do tworzenia Observable

2.1 ajax

Tworzy `Observable` z odpowiedzi HTTP.

```
import { ajax } from 'rxjs/ajax';

const observable = ajax('https://api.example.com/data');
observable.subscribe(response => console.log(response));
// Output: Response from API
```

Listing 1: Przykład użycia `ajax`

2.2 bindCallback

Tworzy `Observable` z funkcji opóźnionej (callback-based).

```
import { bindCallback } from 'rxjs';

const callbackFn = (value, cb) => cb(value);
const observable = bindCallback(callbackFn)('Hello');
observable.subscribe(response => console.log(response));
// Output: Hello
```

Listing 2: Przykład użycia `bindCallback`

2.3 bindNodeCallback

Tworzy `Observable` z funkcji wezła (Node.js callback-based).

```
import { bindNodeCallback } from 'rxjs';
import { readFile } from 'fs';

const readFile$ = bindNodeCallback(readFile);
const observable = readFile$('file.txt', 'utf8');
observable.subscribe(content => console.log(content));
// Output: Content of file.txt
```

Listing 3: Przykład użycia `bindNodeCallback`

2.4 defer

Tworzy `Observable` dopiero wtedy, gdy jest subskrybowany.

```
import { defer, of } from 'rxjs';

const observable = defer(() => of(new Date()));
observable.subscribe(date => console.log(date));
// Output: Current date
```

Listing 4: Przykład użycia `defer`

2.5 empty

Tworzy `Observable`, który natychmiast kończy sie bez emitowania wartości.

```
import { empty } from 'rxjs';

const observable = empty();
observable.subscribe({
  next(value) { console.log(value); },
  complete() { console.log('Complete'); }
// Output: Complete
```

Listing 5: Przykład użycia `empty`

2.6 from

Tworzy `Observable` z obiektów iterowalnych.

```
import { from } from 'rxjs';

const observable = from([10, 20, 30]);
observable.subscribe(value => console.log(value));
// Output: 10, 20, 30
```

Listing 6: Przykład użycia `from`

2.7 fromEvent

Tworzy `Observable` z wydarzeń DOM.

```
import { fromEvent } from 'rxjs';

const observable = fromEvent(document, 'click');
observable.subscribe(event => console.log('Clicked:', event))
   ;
// Output: Click event details
```

Listing 7: Przykład użycia `fromEvent`

2.8 fromEventPattern

Tworzy `Observable` z funkcji, które dodaja i usuwaja nasłuchiwacze zdarzeń.

```
import { fromEventPattern } from 'rxjs';

const observable = fromEventPattern(
  handler => document.addEventListener('click', handler),
  handler => document.removeEventListener('click', handler)
);
observable.subscribe(event => console.log('Clicked:', event))
   ;
// Output: Click event details
```

Listing 8: Przykład użycia `fromEventPattern`

2.9 generate

Tworzy `Observable` na podstawie funkcji generujacej wartości.

```
import { generate } from 'rxjs';

const observable = generate(
  0, // Initial state
  x => x < 5, // Condition
  x => x + 1 // Iteration function
);
observable.subscribe(value => console.log(value));
// Output: 0, 1, 2, 3, 4
```

Listing 9: Przykład użycia `generate`

2.10 interval

Tworzy `Observable`, który emituje wartości w równych odstepach czasu.

```
import { interval } from 'rxjs';

const observable = interval(1000); // Emituje co 1 sekund
observable.subscribe(value => console.log(value));
// Output: 0, 1, 2, 3, ...
```

Listing 10: Przykład użycia `interval`

2.11 of

Tworzy `Observable` z wartości przekazanych jako argumenty.

```
import { of } from 'rxjs';

const observable = of(1, 2, 3, 4, 5);
```

```
observable.subscribe(value => console.log(value));
// Output: 1, 2, 3, 4, 5
```

Listing 11: Przykład użycia `of`

2.12 range

Tworzy `Observable` emitujacy wartości z określonego zakresu.

```
import { range } from 'rxjs';

const observable = range(1, 5); // Emituje warto ci od 1 do
   5
observable.subscribe(value => console.log(value));
// Output: 1, 2, 3, 4, 5
```

Listing 12: Przykład użycia `range`

2.13 throwError

Tworzy `Observable`, który natychmiast emituje bład.

```
import { throwError } from 'rxjs';

const observable = throwError(() => new Error('Something went
    wrong'));
observable.subscribe({
  next(value) { console.log(value); },
  error(err) { console.error('Error: ', err); }
});
// Output: Error: Error: Something went wrong
```

Listing 13: Przykład użycia `throwError`

2.14 timer

Tworzy `Observable`, który emituje wartość po określonym czasie lub w regularnych odstepach.

```
import { timer } from 'rxjs';

const observable = timer(2000, 1000); // Pierwsza emisja po 2
    sekundach, kolejne co 1 sekund
observable.subscribe(value => console.log(value));
// Output: 0, 1, 2, 3, ...
```

Listing 14: Przykład użycia `timer`

2.15 iif

Tworzy `Observable` na podstawie warunku.

```
import { iif, of, throwError } from 'rxjs';

const condition = true;
const observable = iif(
  () => condition,
  of('Condition met'),
  throwError(() => new Error('Condition not met'))
);
observable.subscribe({
  next(value) { console.log(value); },
  error(err) { console.error('Error: ', err); }
});
// Output: Condition met
```

Listing 15: Przykład użycia `iif`

3 Operatory łaczenia tworzace

3.1 combineLatest

Łaczy wartości z wielu `Observable` i emituje, gdy wszystkie źródła wyemituja co najmniej jedna wartość.

```
import { combineLatest, of } from 'rxjs';

const observable1 = of('A', 'B');
const observable2 = of(1, 2);

combineLatest([observable1, observable2]).subscribe(values =>
    console.log(values));
// Output: ['B', 1], ['B', 2]
```

Listing 16: Przykład użycia `combineLatest`

3.2 concat

Łaczy wiele `Observable` w kolejności, emitujac wartości z jednego źródła, zanim przejdzie do nastepnego.

```
import { concat, of } from 'rxjs';

const observable1 = of(1, 2);
const observable2 = of(3, 4);

concat(observable1, observable2).subscribe(value => console.
   log(value));
```

```
// Output: 1, 2, 3, 4
```

Listing 17: Przykład użycia `concat`

3.3 forkJoin

Emitowanie ostatnich wartości z wielu źródeł, gdy wszystkie źródła zakończa emisje.

```
import { forkJoin, of } from 'rxjs';

const observable1 = of('A', 'B');
const observable2 = of(1, 2);

forkJoin([observable1, observable2]).subscribe(values =>
  console.log(values));
// Output: [['B'], [2]]
```

Listing 18: Przykład użycia `forkJoin`

3.4 merge

Łaczy wiele `Observable` i emituje wartości, gdy tylko jeden z nich emituje coś.

```
import { merge, of } from 'rxjs';

const observable1 = of(1, 2);
const observable2 = of(3, 4);

merge(observable1, observable2).subscribe(value => console.
  log(value));
// Output: 1, 2, 3, 4
```

Listing 19: Przykład użycia `merge`

3.5 partition

Dzieli `Observable` na dwie na podstawie predykatu.

```
import { partition, from } from 'rxjs';

const [even$, odd$] = partition(from([1, 2, 3, 4, 5]), value
  => value % 2 === 0);

even$.subscribe(value => console.log('Even:', value));
// Output: Even: 2, 4

odd$.subscribe(value => console.log('Odd:', value));
// Output: Odd: 1, 3, 5
```

Listing 20: Przykład użycia `partition`

3.6 race

Emitowanie wartości z `Observable`, który pierwszy wyemituje wartość.

```
import { race, interval, of } from 'rxjs';

const observable1 = interval(1000);
const observable2 = of('A', 'B', 'C');

race(observable1, observable2).subscribe(value => console.log
   (value));
// Output: 'A', 'B', 'C'
```

Listing 21: Przykład użycia `race`

3.7 zip

Łaczy wartości z wielu `Observable` w pary.

```
import { zip, of } from 'rxjs';

const observable1 = of(1, 2, 3);
const observable2 = of('A', 'B', 'C');

zip(observable1, observable2).subscribe(values => console.log
   (values));
// Output: [1, 'A'], [2, 'B'], [3, 'C']
```

Listing 22: Przykład użycia `zip`

4 Operatory transformacji

4.1 buffer

Zbiera wartości w buforze na podstawie innego `Observable`.

```
import { interval, buffer, timer } from 'rxjs';

const observable = interval(500).pipe(
  buffer(timer(2000))
);
observable.subscribe(values => console.log(values));
// Output: [0, 1, 2, 3]
```

Listing 23: Przykład użycia `buffer`

4.2 bufferCount

Zbiera wartości w buforze, gdy osiagnie określona liczbe wartości.

```
import { interval, bufferCount } from 'rxjs';

const observable = interval(500).pipe(
  bufferCount(3)
);
observable.subscribe(values => console.log(values));
// Output: [0, 1, 2], [3, 4, 5], ...
```

Listing 24: Przykład użycia `bufferCount`

4.3 bufferTime

Zbiera wartości w buforze na podstawie czasu.

```
import { interval, bufferTime } from 'rxjs';

const observable = interval(500).pipe(
  bufferTime(2000)
);
observable.subscribe(values => console.log(values));
// Output: [0, 1, 2], [3, 4, 5], ...
```

Listing 25: Przykład użycia `bufferTime`

4.4 bufferToggle

Zbiera wartości na podstawie otwierajacego i zamykajacego `Observable`.

```
import { interval, bufferToggle, timer } from 'rxjs';

const observable = interval(500).pipe(
  bufferToggle(timer(2000), () => timer(2000))
);
observable.subscribe(values => console.log(values));
// Output: [0, 1], [2, 3]
```

Listing 26: Przykład użycia `bufferToggle`

4.5 bufferWhen

Zbiera wartości na podstawie emitowanych wartości innego `Observable`.

```
import { interval, bufferWhen, timer } from 'rxjs';

const observable = interval(500).pipe(
  bufferWhen(() => timer(2000))
);
observable.subscribe(values => console.log(values));
// Output: [0, 1], [2, 3], ...
```

Listing 27: Przykład użycia `bufferWhen`

4.6 concatMap

Mapuje wartości na `Observable` i łaczy je sekwencyjnie.

```
import { from, concatMap, of } from 'rxjs';

const observable = from([1, 2, 3]).pipe(
  concatMap(value => of(value * 2))
);
observable.subscribe(value => console.log(value));
// Output: 2, 4, 6
```

Listing 28: Przykład użycia `concatMap`

4.7 concatMapTo

Mapuje wszystkie wartości na `Observable`.

```
import { from, concatMapTo, of } from 'rxjs';

const observable = from([1, 2, 3]).pipe(
  concatMapTo(of('A'))
);
observable.subscribe(value => console.log(value));
// Output: 'A', 'A', 'A'
```

Listing 29: Przykład użycia `concatMapTo`

4.8 exhaust

Emituje wartości tylko wtedy, gdy poprzednia emisja zakończyła sie.

```
import { interval, exhaust, take } from 'rxjs';

const observable = interval(500).pipe(
  exhaust(),
  take(3)
);
observable.subscribe(value => console.log(value));
// Output: 0, 1, 2
```

Listing 30: Przykład użycia `exhaust`

4.9 exhaustMap

Mapuje wartości na `Observable` i ignoruje nowe wartości, gdy poprzedni `Observable` nie zakończył emisji.

```
import { interval, exhaustMap, of } from 'rxjs';

const observable = interval(500).pipe(
  exhaustMap(value => of(value * 2))
);
observable.subscribe(value => console.log(value));
// Output: 0, 2, 4, ...
```

Listing 31: Przykład użycia `exhaustMap`

4.10 expand

Rekurencyjnie mapuje wartości na nowe `Observable`.

```
import { of, expand, take } from 'rxjs';

const observable = of(1).pipe(
  expand(value => of(value + 1)),
  take(5)
);
observable.subscribe(value => console.log(value));
// Output: 1, 2, 3, 4, 5
```

Listing 32: Przykład użycia `expand`

4.11 groupBy

Grupuje wartości w `Observable` na podstawie klucza.

```
import { from, groupBy, mergeMap, toArray } from 'rxjs';

const observable = from([1, 2, 3, 4, 5]).pipe(
  groupBy(value => value % 2),
  mergeMap(group => group.pipe(toArray()))
);
observable.subscribe(values => console.log(values));
// Output: [1, 3, 5], [2, 4]
```

Listing 33: Przykład użycia `groupBy`

4.12 map

Mapuje wartości na nowe wartości.

```
import { from, map } from 'rxjs';
```

```
const observable = from([1, 2, 3]).pipe(
  map(value => value * 2)
);
observable.subscribe(value => console.log(value));
// Output: 2, 4, 6
```

Listing 34: Przykład użycia `map`

4.13 mapTo

Mapuje wszystkie wartości na te sama wartość.

```
import { from, mapTo } from 'rxjs';

const observable = from([1, 2, 3]).pipe(
  mapTo('A')
);
observable.subscribe(value => console.log(value));
// Output: 'A', 'A', 'A'
```

Listing 35: Przykład użycia `mapTo`

4.14 mergeMap

Mapuje wartości na `Observable` i łaczy je równolegle.

```
import { from, mergeMap, of } from 'rxjs';

const observable = from([1, 2, 3]).pipe(
  mergeMap(value => of(value * 2))
);
observable.subscribe(value => console.log(value));
// Output: 2, 4, 6
```

Listing 36: Przykład użycia `mergeMap`

4.15 mergeMapTo

Mapuje wszystkie wartości na `Observable` i łaczy je równolegle.

```
import { from, mergeMapTo, of } from 'rxjs';

const observable = from([1, 2, 3]).pipe(
  mergeMapTo(of('A'))
);
observable.subscribe(value => console.log(value));
// Output: 'A', 'A', 'A'
```

Listing 37: Przykład użycia `mergeMapTo`

4.16 mergeScan

Podobny do `scan`, ale łaczy wartości równolegle.

```
import { from, mergeScan, of } from 'rxjs';

const observable = from([1, 2, 3]).pipe(
  mergeScan((acc, value) => of(acc + value), 0)
);
observable.subscribe(value => console.log(value));
// Output: 1, 3, 6
```

Listing 38: Przykład użycia `mergeScan`

4.17 pairwise

Emitowanie par kolejnych wartości.

```
import { from, pairwise } from 'rxjs';

const observable = from([1, 2, 3]).pipe(
  pairwise()
);
observable.subscribe(values => console.log(values));
// Output: [1, 2], [2, 3]
```

Listing 39: Przykład użycia `pairwise`

4.18 pluck

Wydobywa wartości z obiektów na podstawie klucza.

```
import { from, pluck } from 'rxjs';

const observable = from([{ a: 1 }, { a: 2 }, { a: 3 }]).pipe(
  pluck('a')
);
observable.subscribe(value => console.log(value));
// Output: 1, 2, 3
```

Listing 40: Przykład użycia `pluck`

4.19 scan

Akumuluje wartości na podstawie funkcji akumulatora.

```
import { from, scan } from 'rxjs';

const observable = from([1, 2, 3]).pipe(
  scan((acc, value) => acc + value, 0)
);
```

```
observable.subscribe(value => console.log(value));
// Output: 1, 3, 6
```

Listing 41: Przykład użycia `scan`

4.20 switchScan

Podobny do `scan`, ale z zachowaniem wyłacznie najnowszego `Observable`.

```
import { from, switchScan, of } from 'rxjs';

const observable = from([1, 2, 3]).pipe(
  switchScan((acc, value) => of(acc + value), 0)
);
observable.subscribe(value => console.log(value));
// Output: 1, 3, 6
```

Listing 42: Przykład użycia `switchScan`

4.21 switchMap

Mapuje wartości na `Observable` i subskrybuje najnowszy.

```
import { from, switchMap, of } from 'rxjs';

const observable = from([1, 2, 3]).pipe(
  switchMap(value => of(value * 2))
);
observable.subscribe(value => console.log(value));
// Output: 2, 4, 6
```

Listing 43: Przykład użycia `switchMap`

4.22 switchMapTo

Mapuje wszystkie wartości na ten sam `Observable` i subskrybuje najnowszy.

```
import { from, switchMapTo, of } from 'rxjs';

const observable = from([1, 2, 3]).pipe(
  switchMapTo(of('A'))
);
observable.subscribe(value => console.log(value));
// Output: 'A', 'A', 'A'
```

Listing 44: Przykład użycia `switchMapTo`

4.23 window

Zbiera wartości w oknach na podstawie innego `Observable`.

```
import { interval, window, timer } from 'rxjs';

const observable = interval(500).pipe(
  window(timer(2000))
);
observable.subscribe(windowed => windowed.subscribe(value =>
   console.log(value)));
// Output: 0, 1, 2, 3
```

Listing 45: Przykład użycia `window`

4.24 windowCount

Zbiera wartości w oknach na podstawie liczby wartości.

```
import { interval, windowCount } from 'rxjs';

const observable = interval(500).pipe(
  windowCount(3)
);
observable.subscribe(windowed => windowed.subscribe(value =>
   console.log(value)));
// Output: [0, 1, 2], [3, 4, 5], ...
```

Listing 46: Przykład użycia `windowCount`

4.25 windowTime

Zbiera wartości w oknach na podstawie czasu.

```
import { interval, windowTime } from 'rxjs';

const observable = interval(500).pipe(
  windowTime(2000)
);
observable.subscribe(windowed => windowed.subscribe(value =>
   console.log(value)));
// Output: [0, 1], [2, 3], ...
```

Listing 47: Przykład użycia `windowTime`

4.26 windowToggle

Zbiera wartości na podstawie otwierajacego i zamykajacego `Observable`.

```
import { interval, windowToggle, timer } from 'rxjs';
```

```
const observable = interval(500).pipe(
  windowToggle(timer(2000), () => timer(2000))
);
observable.subscribe(windowed => windowed.subscribe(value =>
   console.log(value)));
// Output: [0, 1], [2, 3]
```

Listing 48: Przykład użycia `windowToggle`

4.27 windowWhen

Zbiera wartości na podstawie emitowanych wartości innego `Observable`.

```
import { interval, windowWhen, timer } from 'rxjs';

const observable = interval(500).pipe(
  windowWhen(() => timer(2000))
);
observable.subscribe(windowed => windowed.subscribe(value =>
   console.log(value)));
// Output: [0, 1], [2, 3], ...
```

Listing 49: Przykład użycia `windowWhen`

5 Operatory podziału

5.1 share

Udostepnia wspólna subskrypcje pomiedzy wieloma subskrybentami.

```
import { interval, share } from 'rxjs';

const observable = interval(1000).pipe(
  share()
);
observable.subscribe(value => console.log('Subscriber 1:',
   value));
observable.subscribe(value => console.log('Subscriber 2:',
   value));
// Output: Subscriber 1: 0, 1, 2, ... (same for Subscriber 2)
```

Listing 50: Przykład użycia `share`

5.2 shareReplay

Udostepnia wspólna subskrypcje i buforuje ostatnie wartości.

```
import { interval, shareReplay } from 'rxjs';
```

```
const observable = interval(1000).pipe(
  shareReplay(1)
);
observable.subscribe(value => console.log('Subscriber 1:',
   value));
setTimeout(() => {
  observable.subscribe(value => console.log('Subscriber 2:',
     value));
}, 2000);
// Output: Subscriber 1: 0, 1, 2, ... (Subscriber 2 will
   start from the latest value)
```

Listing 51: Przykład użycia `shareReplay`

6 Operatory obsługi błedów

6.1 catchError

Obsługuje błedy i zwraca nowe `Observable` w przypadku błedu.

```
import { of, throwError } from 'rxjs';
import { catchError } from 'rxjs/operators';

const observable = throwError(() => new Error('Something went
    wrong')).pipe(
  catchError(err => {
    console.error('Caught error:', err);
    return of('Recovered value');
  })
);
observable.subscribe(value => console.log(value));
// Output: Caught error: Error: Something went wrong
// Output: Recovered value
```

Listing 52: Przykład użycia `catchError`

6.2 retry

Ponawia subskrypcje `Observable` w przypadku błedu.

```
import { throwError } from 'rxjs';
import { retry } from 'rxjs/operators';

const observable = throwError(() => new Error('Failed')).pipe
   (
  retry(2)
);
observable.subscribe({
  error(err) { console.error('Retry failed:', err); }
```

```
});
// Output: Retry failed: Error: Failed
```

Listing 53: Przykład użycia `retry`

6.3 retryWhen

Ponawia subskrypcje `Observable` na podstawie logiki dostarczonej przez inny `Observable`.

```
import { throwError, timer } from 'rxjs';
import { retryWhen, mergeMap } from 'rxjs/operators';

const observable = throwError(() => new Error('Failed')).pipe
  (
  retryWhen(errors => errors.pipe(
    mergeMap((error, index) => {
      if (index < 2) {
        console.log('Retrying...');
        return timer(1000);
      }
      return throwError(() => error);
    })
  ))
);
observable.subscribe({
  error(err) { console.error('Retry failed:', err); }
});
// Output: Retrying... (after 1 second)
// Output: Retrying... (after 1 second)
// Output: Retry failed: Error: Failed
```

Listing 54: Przykład użycia `retryWhen`

7 Operatory narzedziowe

7.1 tap

`tap` pozwala na wykonanie działania na wartościach przepływajacych przez strumień bez modyfikacji tych wartości.

```
import { of, tap } from 'rxjs';

const observable = of(1, 2, 3).pipe(
  tap(value => console.log('Before map:', value)),
  map(value => value * 2),
  tap(value => console.log('After map:', value))
);
observable.subscribe(value => console.log('Final value:',
  value));
```

```
// Output:
// Before map: 1
// After map: 2
// Final value: 2
// Before map: 2
// After map: 4
// Final value: 4
// Before map: 3
// After map: 6
// Final value: 6
```

Listing 55: Przykład użycia `tap`

7.2 delay

`delay` opóźnia emisje wartości o określony czas.

```
import { of, delay } from 'rxjs';

const observable = of('Hello').pipe(
  delay(1000)
);
observable.subscribe(value => console.log(value));
// Output (po 1 sekundzie): Hello
```

Listing 56: Przykład użycia `delay`

7.3 delayWhen

`delayWhen` opóźnia emisje wartości na podstawie innego `Observable`.

```
import { of, delayWhen, timer } from 'rxjs';

const observable = of('Delayed value').pipe(
  delayWhen(() => timer(2000))
);
observable.subscribe(value => console.log(value));
// Output (po 2 sekundach): Delayed value
```

Listing 57: Przykład użycia `delayWhen`

7.4 timeout

`timeout` generuje błąd, jeśli nie nadejdzie żadna wartość w określonym czasie.

```
import { of, timeout } from 'rxjs';

const observable = of('Hello').pipe(
  delay(2000),
  timeout(1000)
```

```
);
observable.subscribe({
  next: value => console.log(value),
  error: err => console.error('Error:', err.message)
});
// Output: Error: Timeout has occurred
```

Listing 58: Przykład użycia `timeout`

7.5 timeoutWith

`timeoutWith` pozwala na przełaczenie sie na inny `Observable`, jeśli nie nadejdzie żadna wartość w określonym czasie.

```
import { of, timeoutWith, timer } from 'rxjs';

const observable = of('Hello').pipe(
  delay(2000),
  timeoutWith(1000, timer(0, 500))
);
observable.subscribe(value => console.log(value));
// Output (co 500 ms): 0, 1, 2, ...
```

Listing 59: Przykład użycia `timeoutWith`

8 Tworzenie własnych operatorów

Tworzenie własnych operatorów polega na utworzeniu funkcji, która przyjmuje `Observable` jako argument i zwraca nowy `Observable`.

```
import { Observable } from 'rxjs';

function myCustomOperator() {
  return function(source) {
    return new Observable(observer => {
      return source.subscribe({
        next(value) {
          observer.next(value * 2);
        },
        error(err) {
          observer.error(err);
        },
        complete() {
          observer.complete();
        }
      });
    });
  };
}
```

```
const observable = of(1, 2, 3).pipe(myCustomOperator());
observable.subscribe(value => console.log(value));
// Output: 2, 4, 6
```

Listing 60: Przykład tworzenia własnego operatora

9 Podsumowanie

RxJS dostarcza potężny zestaw narzedzi do pracy z asynchronicznymi strumieniami danych. Zrozumienie operatorów pozwala na tworzenie zaawansowanych logik przepływu danych w aplikacjach, umożliwiajac bardziej efektywne i czytelne zarzadzanie stanem oraz operacjami asynchronicznymi.