

Angular 16

część 2

dr inż. Grzegorz Rogus



Routing

Nawigacja po aplikacji

Single Page Application



No page refresh on request

Traditional Web Application



Whole page refresh on request

App
Shell



Header

Home

Contact

About

App Component

Footer



Routing

Routing pozwala zawrzeć pewne aspekty stanu aplikacji w adresie URL.

Dla aplikacji front-end jest to opcjonalne - możemy zbudować pełną aplikację bez zmiany adresu URL. Dodanie routingu pozwala jednak użytkownikowi przejść od razu do pewnych funkcji aplikacji.

Dzięki temu aplikacja jest łatwiej przenośna i dostępna dla zakładek oraz umożliwi użytkownikom dzielenie się linkami z innymi.

Routing ułatwia:

- Utrzymanie stanu aplikacji
- Wdrażanie aplikacji modułowych
- Stosowanie ról w aplikacji (niektóre role mają dostęp do określonych adresów URL)



Routing

Routing pozwala zawrzeć pewne aspekty stanu aplikacji w adresie URL.

Dla aplikacji front-end jest to opcjonalne - możemy zbudować pełną aplikację bez zmiany adresu URL. Dodanie routingu pozwala jednak użytkownikowi przejść od razu do pewnych funkcji aplikacji.

Dzięki temu aplikacja jest łatwiej przenośna i dostępna dla zakładek oraz umożliwi użytkownikom dzielenie się linkami z innymi.

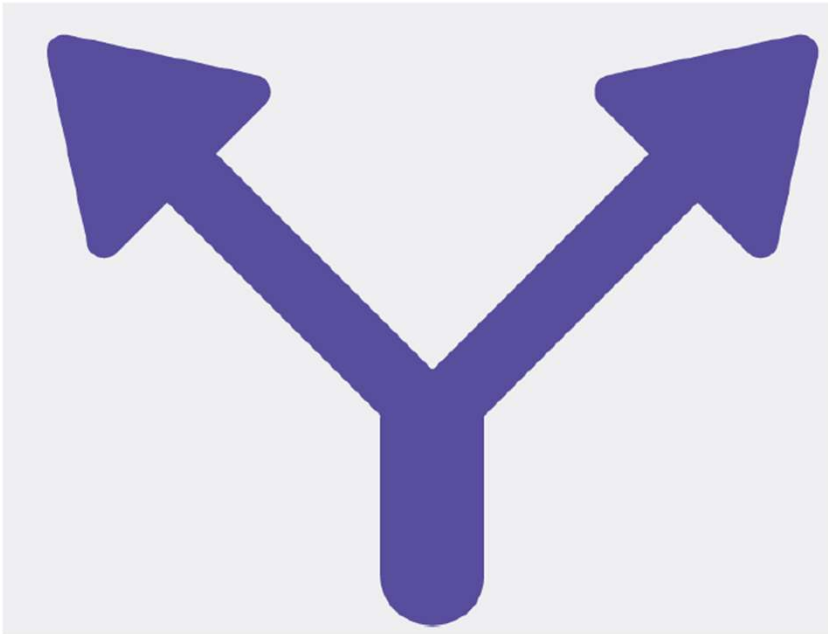
Routing ułatwia:

- Utrzymanie stanu aplikacji
- Wdrażanie aplikacji modułowych
- Stosowanie ról w aplikacji (niektóre role mają dostęp do określonych adresów URL)



Routing

Składowe routingu



1. `<basehref="/">`
2. `import RouterModule`
3. Konfiguracja ścieżek
4. `<router-outlet>`

Konfiguracja routingu

CLI -> Would you like to add Angular routing? (y/N)

src/app/app-routing.module.ts



```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

const routes: Routes = [];

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

src/app/app.component.html

```
<router-outlet></router-outlet>
```

Definiowanie tablicy routes

```
const routes: Routes = [  
  { path: 'component-one', component: ComponentOne },  
  { path: 'component-two', component: ComponentTwo }  
];
```

Definiowanie połączeń między trasami

```
<a routerLink="/component-one">Component One</a>
```

Nawigacja programowo

```
this.router.navigate(['/component-one']);
```


Deklaracja parametrów trasy

```
export const routes: Routes = [  
  { path: '', redirectTo: 'product-list', pathMatch: 'full' },  
  { path: 'product-list', component: ProductList },  
  { path: 'product-details/:id', component: ProductDetails }  
];
```



localhost:4200/szczegóły produktu/5

Powiązanie tras z parametrami

```
<a *ngFor="let product of products"  
  [routerLink]="['/product-details', product.id]">  
  {{ product.name }}  
</a>
```

Trasowanie - przykład

```
import { NgModule }           from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { StudentsComponent }   from './students/students.component';
import { StudentDetailComponent } from './student-detail/student-detail.component';

const routes: Routes = [
  { path: '', redirectTo: '/students', pathMatch: 'full' },
  { path: 'students', component: StudentsComponent },
  { path: 'detail/:id', component: StudentDetailComponent }
];

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

Import modułu routingu w głównym (root) module

```
import { NgModule } from '@angular/core';
import { AppRoutingModule }
      from './app-routing.module';

...
@NgModule({
  declarations: [...],
  imports: [..., AppRoutingModule],
  providers: [...],
  bootstrap: [...]
})
export class AppModule { }
```

Router outlet we wzorcu HTML
głównego komponentu aplikacji

```
...
<router-outlet></router-outlet>
...
```

Linki prowadzące do komponentów (we wzorcach HTML komponentów)

```
<nav>
  <a routerLink="/students">Students</a>
  <a routerLink="/about">About</a>
</nav>
```

```
<a routerLink="/detail/{{student.id}}">
  <span>{{student.index}}</span>
</a>
```

Routing: wyświetlanie

- Komponent pasujący do aktualnej ścieżki zostanie załadowany wewnątrz komponentu RouterOutlet
- Za generowanie adresu na potrzeby linku odpowiada dyrektywa router-link

```
@Component({
  selector: 'main-app',
  template: `<main>
    <a [routerLink]="['/Contact']"
      routerLinkActive="active">Contact</a>

    <div style="border: 1px solid black">
      <router-outlet></router-outlet>
    </div>
  </main>`})
```

Kontrolowanie dostępu do lub z Route

Niektóre trasy mają być dostępne tylko po zalogowaniu się użytkownika lub zaakceptowaniu Warunków.

```
const routes: Routes = [  
  { path: 'home', component: HomePage },  
  {  
    path: 'accounts',  
    component: AccountPage,  
    canActivate: [LoginRouteGuard],  
    canDeactivate: [SaveFormsGuard]  
  }  
];
```

```
import { CanActivate } from '@angular/router';  
import { Injectable } from '@angular/core';  
import { LoginService } from './login-service';
```

```
@Injectable()  
export class LoginRouteGuard implements CanActivate {  
  constructor(private loginService: LoginService) {}  
  
  canActivate() {  
    return this.loginService.isLoggedIn();  
  }  
}
```

Model dziedziny i Mock data

Dobrą praktyką projektową jest wyizolowanie struktury danych oraz ewentualnych danych od komponentu.

```
export interface Product {  
  id: number;  
  modelName: string;  
  color: string;  
  productType: string;  
  brand: string;  
  price: number;  
}
```

product.ts

```
import { Product } from '../models/product';  
  
export class MockData {  
  public static Products: Product[] = [  
    {  
      'id': 11,  
      'modelName': 'F5 Youth',  
      'color': 'Gold',  
      'productType': 'Mobile',  
      'brand': 'OPPO',  
      'price': 16990  
    },  
    {  
      'id': 12,  
      'modelName': 'Inspiron',  
      'color': 'Gray',  
      'productType': 'Laptop',  
      'brand': 'DELL',  
      'price': 59990  
    }  
  ]  
}
```

mock-product-data.ts

Mock Data



ProductService
products : Product[]

ProductsComponent
constructor(productService : ProductService)

AddProductComponent
constructor(productService : ProductService)

OtherComponents
constructor(productService : ProductService)

Lista komentarzy – implementacja usługi

- Przenosimy dane z pliku component-list do pliku mock.ts (symulującego źródło danych).
- Następnie tworzymy serwis udostępniający dane pochodzące z mock

Dlaczego usługa obsługi danych?

- Użytkownik usługi nie wie z jakiego źródła są dane.
- Dane mogą pochodzić z Web Serwisu, z lokalnego pliku albo być imitowane.
- To jest piękno korzystania z usług!
- Usługa odpowiada za dostęp do danych.
- W każdej chwili można zmienić sposób dostępu - zmiany są tylko w tej jednej usłudze.

Lista komentarzy

Realizacja usługi udostępniającej dane

```
import { Comment } from '../comment';  
export const KomentarzeDane: Comment[] = [  
  {imie: "Grzegorz", komentarz: "Pierwszy komentarz", hidden: true },  
  {imie: "Anna", komentarz: "Super strona", hidden: false },  
  {imie: "Alicja", komentarz: "Fajny film wczoraj widziałam", hidden: true },  
];
```

```
import { Injectable } from '@angular/core';  
import { KomentarzeDane } from '../mock';  
@Injectable()  
export class KomentarzeService {  
  getComments() {  
    // return komentarze;  
    return Promise.resolve(KomentarzeDane);  
  }  
}
```

Komunikacja asynchroniczna

```
constructor( service: KomentarzeService ) {  
  
  this.comments = service.getComments();  
}
```

Mock Data



ProductService
products : Product[]

ProductsComponent
constructor(productService : ProductService)

AddProductComponent
constructor(productService : ProductService)

OtherComponents
constructor(productService : ProductService)

Uzycie serwisu

- ng g service product

```
import { MockData } from '../mock-data/mock-product-data';
import { Injectable } from '@angular/core';
import { Product } from '../models/product';

@Injectable()
export class ProductService {
  products: Product[] = [];

  constructor() {
    this.products = MockData.Products;
  }

  getProducts(): Product[] {
    return this.products;
  }


  removeProduct(product: Product) {
    let index = this.products.indexOf(product);
    if (index !== -1) {
      this.products.splice(index, 1);
    }
  }

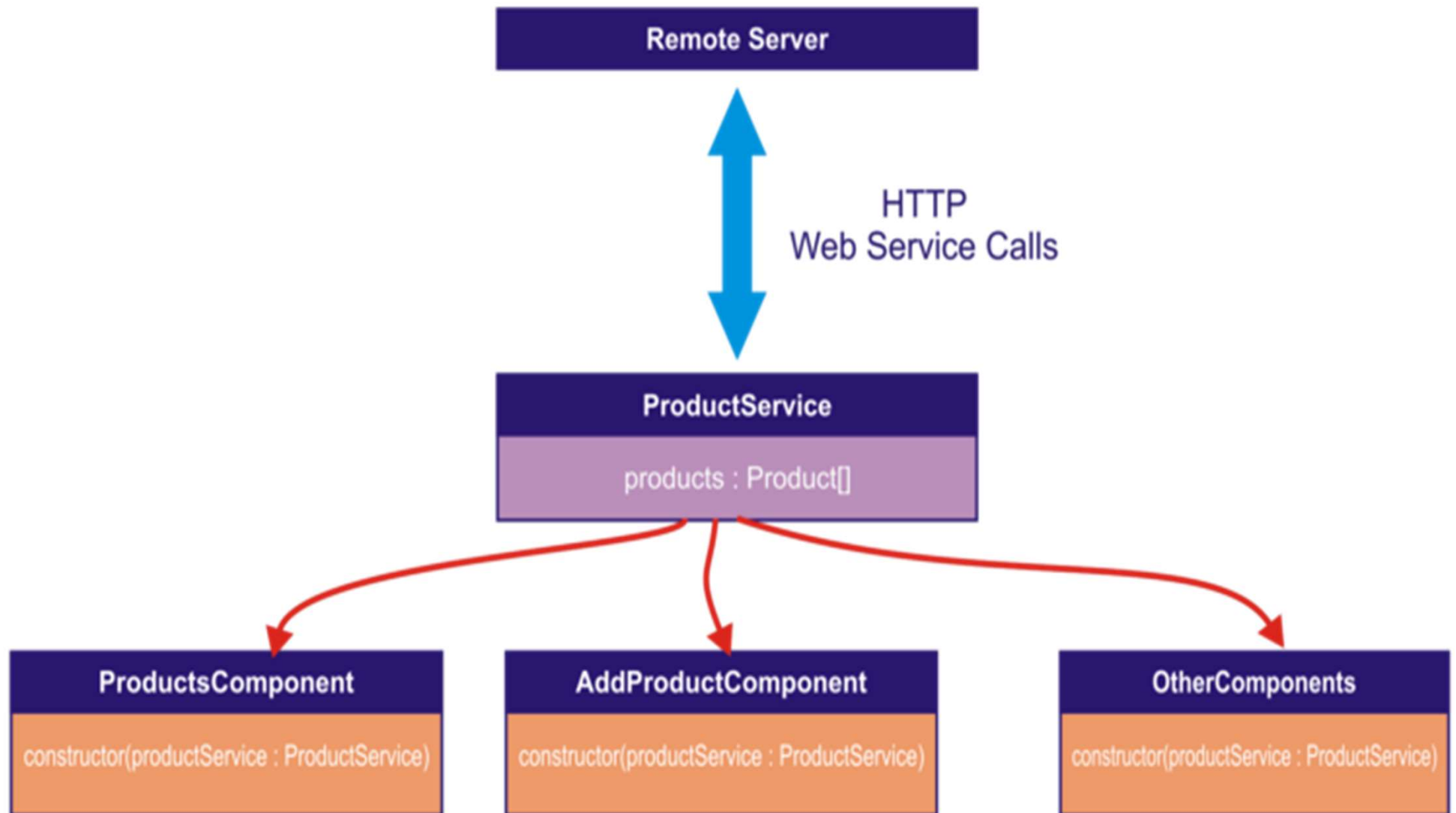
  getProduct(id: number): Product {
    return this.products.find( p => p.id === id);
  }

  addProduct(product: Product) {
    this.products.push(product);
  }
}
```

Wywołanie usługi – przeniesienie danych do usługi

```
export class ProductsComponent implements OnInit {  
  products: Product[] = [];  
  
  constructor(public productService: ProductService) {  
    // this.products = productService.getProducts();  
  }  
  
  ngOnInit() {  
    this.products = productService.getProducts();  
  }  
  
  deleteProduct(product: Product) {  
    this.productService.removeProduct(product);  
    this.products = this.productService.getProducts();  
  }  
}
```





HTTP



Request - http



Response - Observable



Usługi asynchroniczne

- Usługi asynchroniczne zwracające jako wynik obiekty typu Observable lub Promise.
- Należy więc przekształci zwracana wartość w Observable lub promise

```
import { Observable } from 'rxjs/Observable';  
import { of } from 'rxjs/observable/of';
```

```
getProducts(): Observable<Product[]> {  
    return of(this.products);  
}
```

of(this.products) emituje pojedynczą wartość pochodzącą z tablicy produktów.

Usługi asynchroniczne

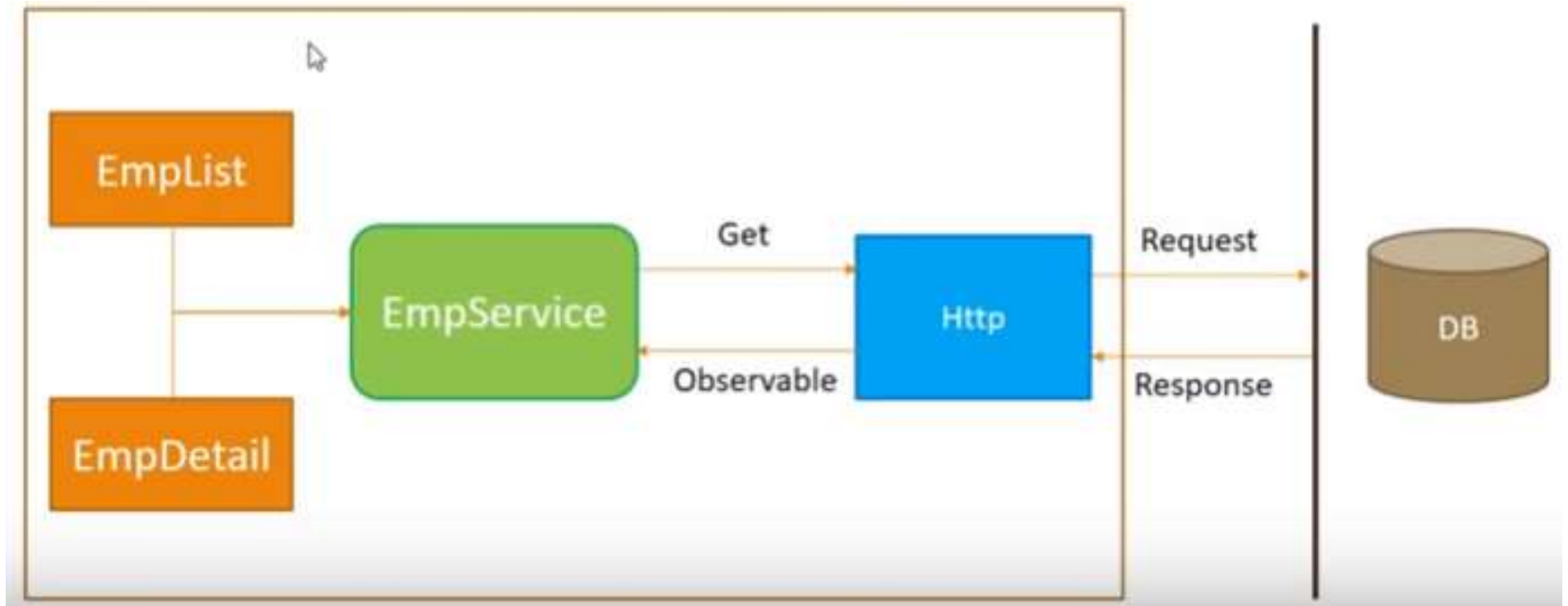
- Zmiany w metodach usługi ProductService skutkują błędem w ProductsComponent. -> powód
ProductService.**getProducts()** zwraca teraz Observable<Product[]>

```
ngOnInit() {  
    this.products = this.productService.getProducts();  
}
```

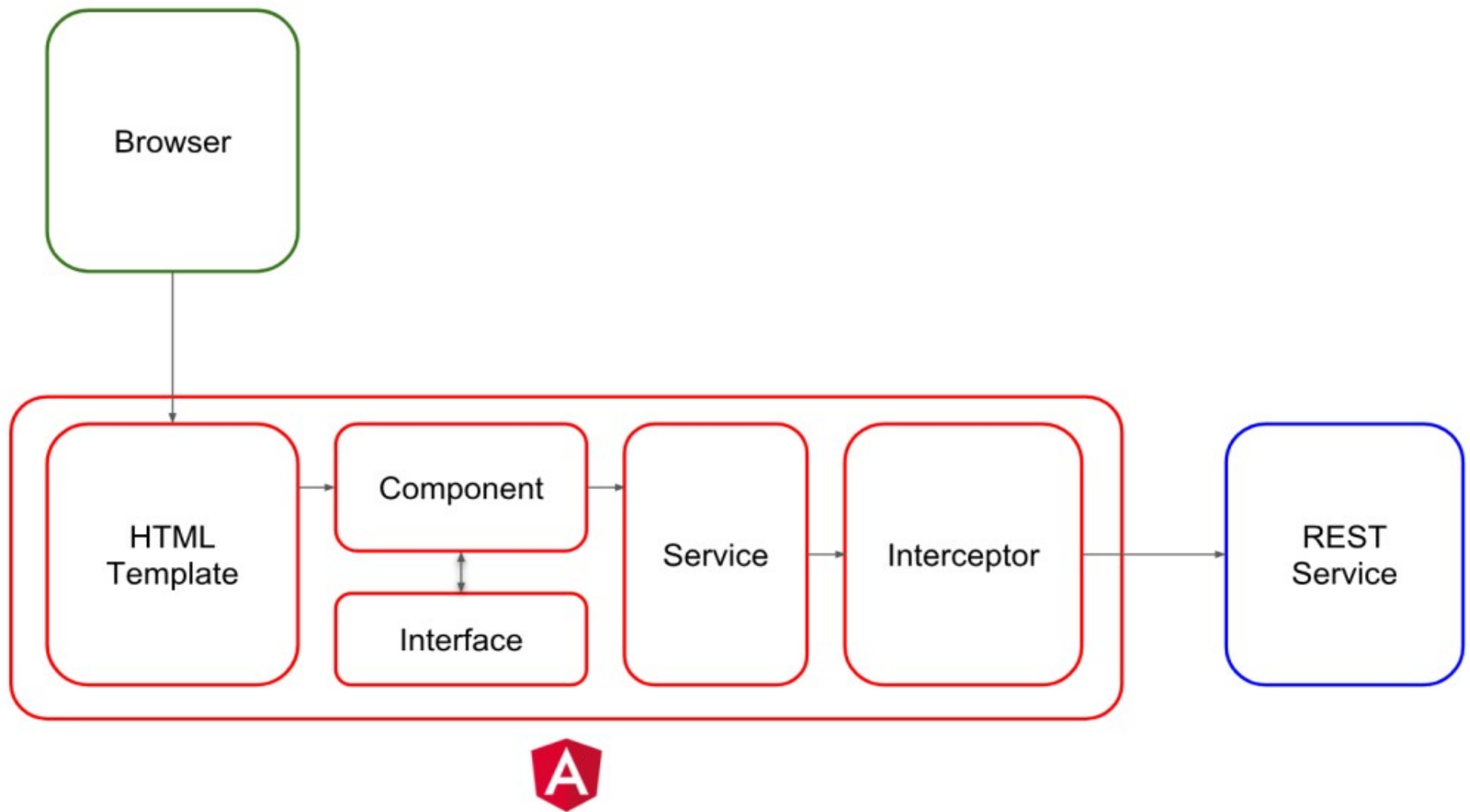
Stara wersja

```
ngOnInit() {  
    this.productService.getProducts().subscribe(  
        products => this.products = products  
    );  
}
```

nowa wersja



Architektura rest API w Angularze



Usługa HTTP

Dodajemy do projektu usługę HTTP (`HttpClient`):

- Angular korzysta z usługi `HttpClient` do komunikacji ze zdalnym serwerem poprzez protokół HTTP
- Aby udostępnić usługę HTTP w całej aplikacji należy:
 - otworzyć główny moduł, `AppModule`
 - zaimportować `HttpClientModule` z modułu `@angular/common/http`
 - dodać go do tablicy `@NgModule.imports`
- Zamiast korzystać z rzeczywistego serwera można go tylko symulować.
- In-memory Web API przechwytyje żądania z usługi `HttpClient`, dane są przechowywane tylko w pamięci operacyjnej.

Symulacja danych z serwera

```
npm install angular-in-memory-web-api --save
```

```
import { InMemoryDbService } from 'angular-in-memory-web-api';

export class InMemoryProductService extends InMemoryDbService {
  createDb() {
    const products = [
      {
        'id': 11,
        'name': 'test',
        ....
      },
      {
        ....
      }
    ];
    return { products };
  }
}
```

fakeserver.service.ts

```
import { InMemoryProductService } from './in-memory-product.service';
import { HttpClientModule } from '@angular/common/http';
import { HttpClientInMemoryWebApiModule } from 'angular-in-memory-web-api';
...
@NgModule({
  ...
  imports: [
    ...
    HttpClientModule,
    HttpClientInMemoryWebApiModule.forRoot(InMemoryProductService, { delay : 2000})
  ],
  ...
})
export class AppModule { }
```

AppModule

httpClient w akcji

Odwołania do serwerowego API w klasie usługi

```
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs/Observable';
import { of } from 'rxjs/observable/of';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Student } from './student';

const httpOptions = { headers: new HttpHeaders({ 'Content-Type': 'application/json' }) };

@Injectable()
export class StudentService {

  private studentsApiUrl = 'http://localhost:5000/api/student';

  constructor(private http: HttpClient) { }

  getStudents(): Observable<Student[]> {
    return this.http.get<Student[]>(this.studentsApiUrl);
  }

  updateStudent(student: Student): Observable<any> {
    const url = `${this.studentsApiUrl}/${student.id}`;
    return this.http.put(url, student, httpOptions);
  }
}
```

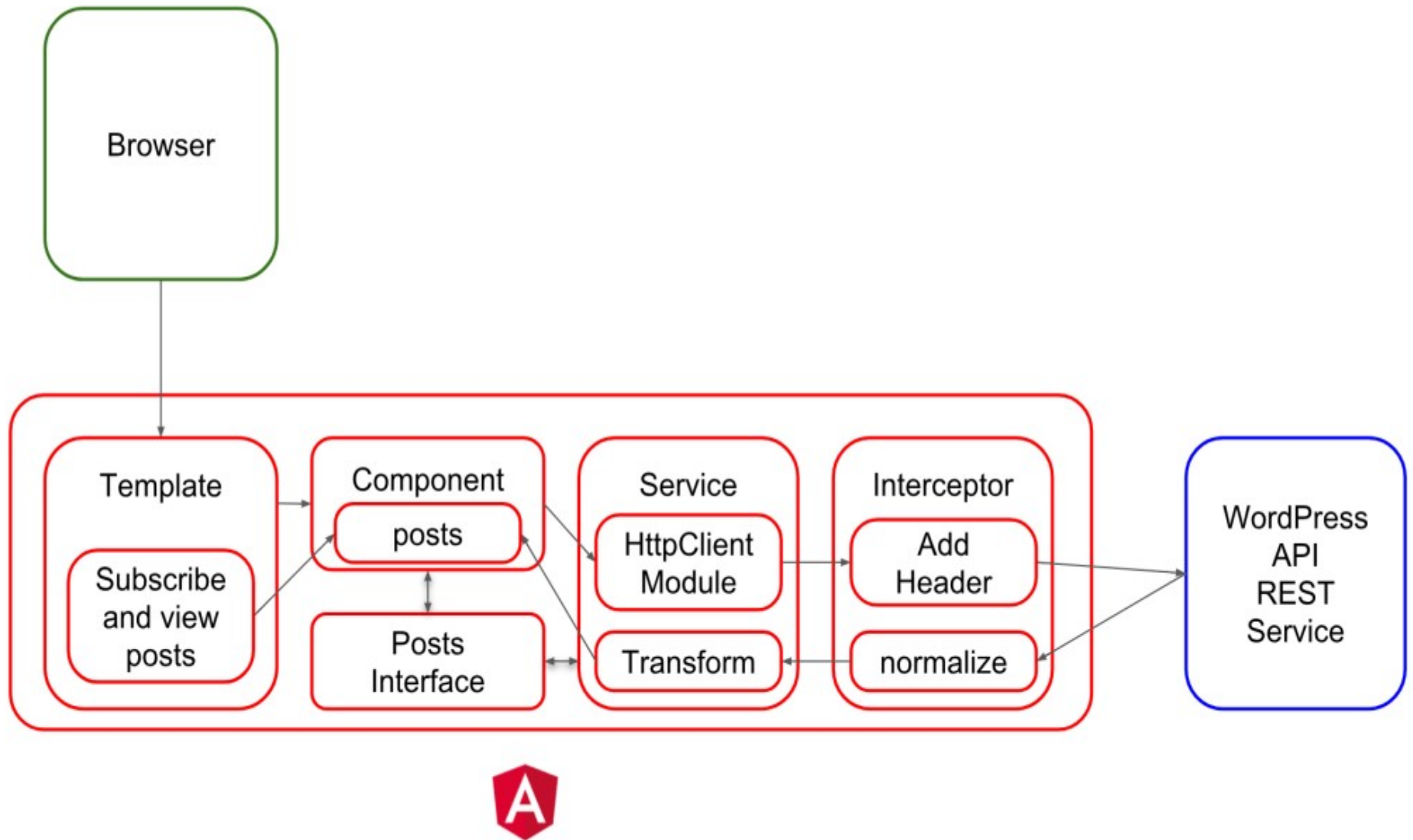
httpClient w akcji

Wykorzystanie usługi dostępowej do API w klasach komponentów

```
...  
students: Student[];  
...  
constructor(private studentService: StudentService) { }  
...  
getStudents(): void {  
    this.studentService.getStudents()  
    .subscribe(students => this.students = students);  
}
```

```
...  
student: Student;  
...  
constructor(private studentService: StudentService) { }  
...  
save(): void {  
    this.studentService.updateStudent(this.student)  
    .subscribe(() => this.goBack());  
}  
...
```

Szczegóły



HttpClient – współpraca z serwerem danych

Konfiguracja

1. Import modułu httpClient

```
import { HttpClient Module} from  
'@angular/common/http';
```

```
@NgModule({  
  declarations: [  
    AppComponent  
  ],  
  imports: [  
    BrowserModule,  
    HttpClientModule  
  ],  
  providers: [],  
  bootstrap: [AppComponent]  
})  
export class AppModule {  
}
```

AppModule

2. Import HttpClient w serwisie

```
import { HttpClient, HttpHeaders } from '@angular/common/http';
```

```
export class ProductService{  
  ...  
  constructor(private httpClient: HttpClient) {  
    ...  
  }  
  ...  
}
```

ProductService.service.ts

HttpClient – współpraca z serwerem danych

odbiór danych – get()

3. Odczyt danych

```
export class ProductService{
    productsUrl = 'api/products';

    constructor(private httpClient: HttpClient) {
    }

    getProducts(): Observable<Product[]> {
        return this.httpClient.get<Product[]>(this.productsUrl);
    }
    ...
    ...
}
```

ProductService.service.ts

HttpClient.get zwraca response jako untyped JSON object.
Zastosowanie specyfikatora typu np. <Product[]> , daje obiekt zrzutowany.

HttpClient – współpraca z serwerem danych

```
getCustomers(): Observable<Customer[]> {  
    return this.http.get('/api/customers');  
}
```

1 wersja

```
getCustomer(id): Observable<Customer> {  
    return this.http.get(`/api/customers/${id}`);  
}
```

```
getCustomers(): Observable<Customer[]> {  
    return this.http.get <Customer[]> ('/api/customers');  
}
```

2 wersja

```
getCustomer(id): Observable<Customer> {  
    return this.http.get <Customer> (`/api/customers/${id}`);  
}
```

```
postCustomer(customer): Promise {  
    return this.http.post('/api/customers', customer)  
        .toPromise()  
        .then((data) => data);  
}
```

3 wersja

```
getOrders(): Promise<Order[]> {  
    return this.http.get<Order[]>('/api/orders')  
        .toPromise()  
        .then((response) => response);  
}
```

```
getOrder(id): Observable<Order> {  
    return this.http.get<Order>(`/api/orders/${id}`);  
}
```

```
getOrdersByCustomer(customerId): Promise<Order[]> {  
    return this.http.get<Order[]>(`/api/customers/${customerId}/orders`)  
        .toPromise()  
        .then((response) => response);  
}
```

```
postOrder(order): Observable<Order> {  
    return this.http.post<Order>('/api/orders', order);  
}
```

Obsługa POST i DELETE

```
addProduct(product: Product): Observable {  
  const httpOptions = {  
    headers: new HttpHeaders({ 'Content-Type': 'application/json' })  
  };  
  return this.httpClient.post(this.productsUrl, product, httpOptions).pipe(  
    tap(p => console.log(`added product with id=${p.id}`)),  
    catchError(this.handleError('addProduct'))  
  );  
}
```

```
removeProduct(product: Product | number): Observable<Product> {  
  const httpOptions = {  
    headers: new HttpHeaders({ 'Content-Type': 'application/json' })  
  };  
  const id = typeof product === 'number' ? product : product.id;  
  const url = `${this.productsUrl}/${id}`;  
  return this.httpClient.delete<Product>(url, httpOptions).pipe(  
    tap(_ => console.log(`deleted Product id=${id}`)),  
    catchError(this.handleError<Product>('deleteProduct'))  
  );  
}
```

Programowanie Reaktywne

- Asynchroniczne strumienie danych
- Wszystko jest strumieniem danych
 - zdarzenia generują dane

Dane z serwera

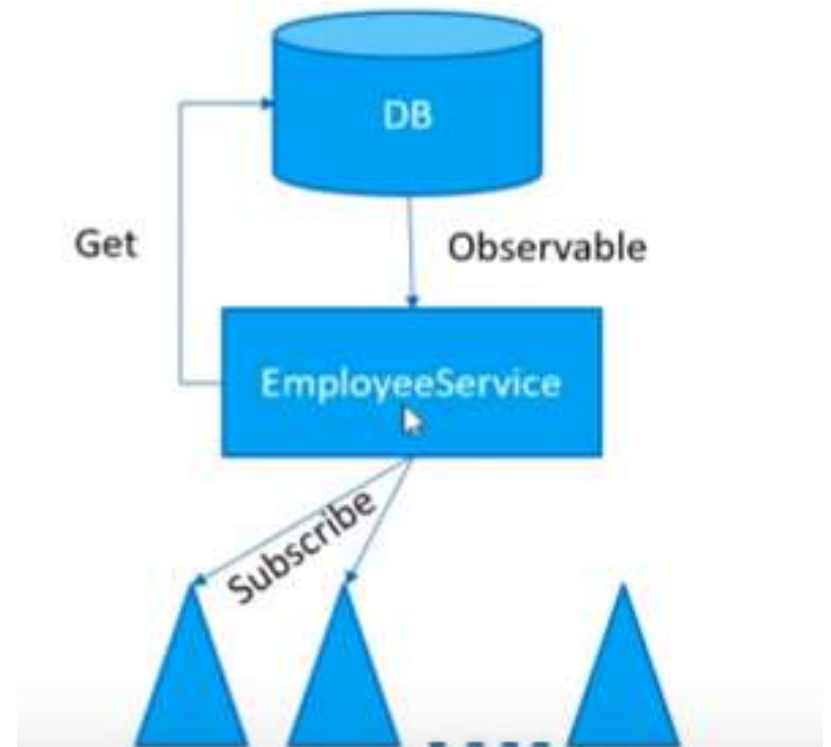
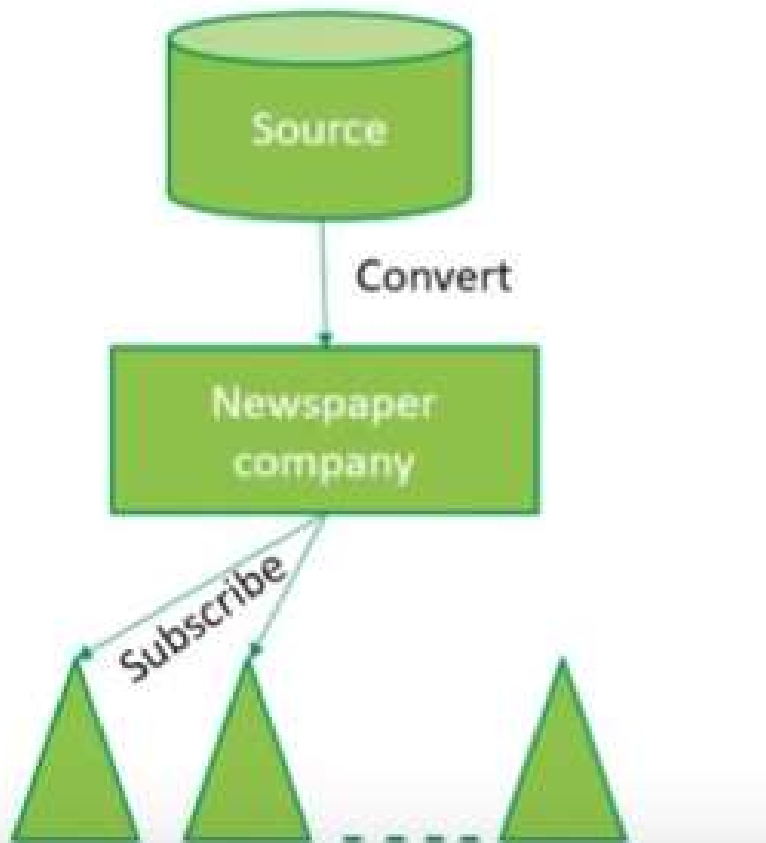
Dane z formularzy

PROGRAMOWANIE REAKTYWNE

- **Programowanie reaktywne jest to programowanie asynchroniczne oparte na obserwacji strumienia danych.**
- Generyczna implementacja wzorca projektowego Obserwator
- Mamy strumienie danych (Observable), które za pomocą wbudowanych operatorów przekształcamy w strumień o interesujących nas własnościach (łączenie, map, flatMap, fold, filter, opóźnienie, wątek ...)
- Ktoś kiedyś zasubskrybuje się na nasz strumień i będzie dostawać powiadomienia o pojawieniu się nowych wartości

Observables

Reprezentuje ideę przyszłego zbioru wartości lub wydarzeń (strumień danych/wydarzeń)



Obiekty typu Observable reprezentują strumień wartości, które zostaną wyemitowane w późniejszym czasie

Źródła danych

| | Pojedyncza wartość | Wiele wartości |
|-----------------|---|--|
| Synchronicznie | <pre>const value = 42; Console.log(value);</pre> | <pre>var values = [1,2,3,4]; values.forEach(value => { console.log(value); });</pre> |
| Asynchronicznie | <pre>const asyncValue = Promise.resolve(42); asyncValue.then(value => { console.log(value); });</pre> | <pre>let val\$= Observable.of(1,2,3,4); val\$.subscribe(val\$ => { console.log(val\$); });</pre> |

Zastosowania

- Reagowanie na akcje użytkownika: kliknięcia, ruch myszki, klawiatura itp.
- Otrzymywanie i reagowanie na dane, również w czasie rzeczywistym, np. po web socketach
- Zdarzenia wykonywane po czasie np. `setTimeout`, `setInterval`
- Wszystko co asynchronicznie zwraca od 0 do nieskończonej ilości wartości

Najważniejsze pojęcia w nowym podejściu

- **Observable** - reprezentuje sekwencję wartości w czasie, która może być obserwowana
- **Observer** - obiekt który otrzymuje dane z Observable
- **Subscription** - wynik subskrybowania, służy głównie do zamykania/zatrzymywania subskrypcji
- **Operators** - zbiór metod dla Observable, najczęściej zwracające nowe Observable (np. *map*, *filter*)
- **Subject** - to samo co Observable, ale nieco bardziej zaawansowane :)

Observable

- Reprezentuje sekwencje wartości w czasie, które są przez nią **emitowane**
- Można ją **zasubskrybować** aby otrzymywać te wartości
- **Leniwe** - kod wykona się i zacznie produkować wartości dopiero kiedy ktoś słucha (czyli subskrybuje)
- Może wyemitować błąd zamiast wartości (jak promise) lub sygnał "zakończenia" strumienia. W obu wypadkach taki observable jest uważany za zakończony i nie wyśle więcej żadnej wartości

Tworzenie obserwowanych

- Wiele różnych sposobów!

```
const clock$ = new Observable( observer => { setInterval(()  
    => { observer.next('tick'); }, 1000); });
```

```
const colors$ = Observable.from(['red', 'green', 'blue']);
```

```
const colors$ = Observable.of('red', 'green', 'blue');
```

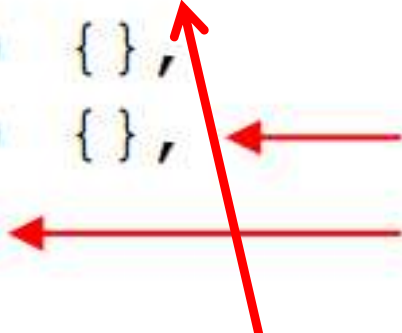
RxJS dodaje dużo nowych metod jak np:

- fromEvent
- fromPromise
- bindCallback
- timer

Podstawy Observable

```
var range = Rx.Observable.range(1, 3); // 1, 2, 3

var range = range.subscribe(  
    function(value) {},  
    function(error) {},  
    function() {}  
);
```



optional

Podłączenie się do strumienia jest banalnie proste. Observable posiada metodę **subscribe**, do której parametry możemy przekazać na dwa sposoby:

Podstawy Observable

```
var obs = ...;
```

```
// query?
```

gdy sukcesem odbierzemy
wartość ze strumienia

```
var sub = obs.Subscribe(  
    onNext : v => DoSomething(v),  
    onError : e => HandleError(e),  
    onCompleted : () => HandleDone);
```

gdy w strumieniu wystąpi error

gdy observer otrzyma ostatnią wartość ze
strumienia (wypstryka się z danych)

Nasłuchiwanie na zmiany

.subscribe() przyjmuje dwa rodzaje argumentów:

1. Listę funkcji (callbacków)

```
observable$.subscribe(  
  value => console.log('Nowa wartość:', value),  
  err => console.error(err),  
  () => console.log('Koniec nadawania') );
```

2. Obiekt typu **Observer** (z metodami)

```
observable$.subscribe( {  
  next: value => console.log('Nowa wartość:', value),  
  error: err => console.error(err),  
  complete: () => console.log('Koniec nadawania'),  
});
```


Observer

- Specjalny obiekt służący do "konsumowania" wartości które dostarcza Observable.
- Najprościej mówiąc jest to obiekt z trzema metodami - callbackami dla każdego rodzaju notyfikacji którą może dostarczyć Observable:

```
const observer = {  
  next: value => console.log('Nowa wartość:', value),  
  error: err => console.error(err),  
  complete: () => console.log('Koniec nadawania'),  
};
```

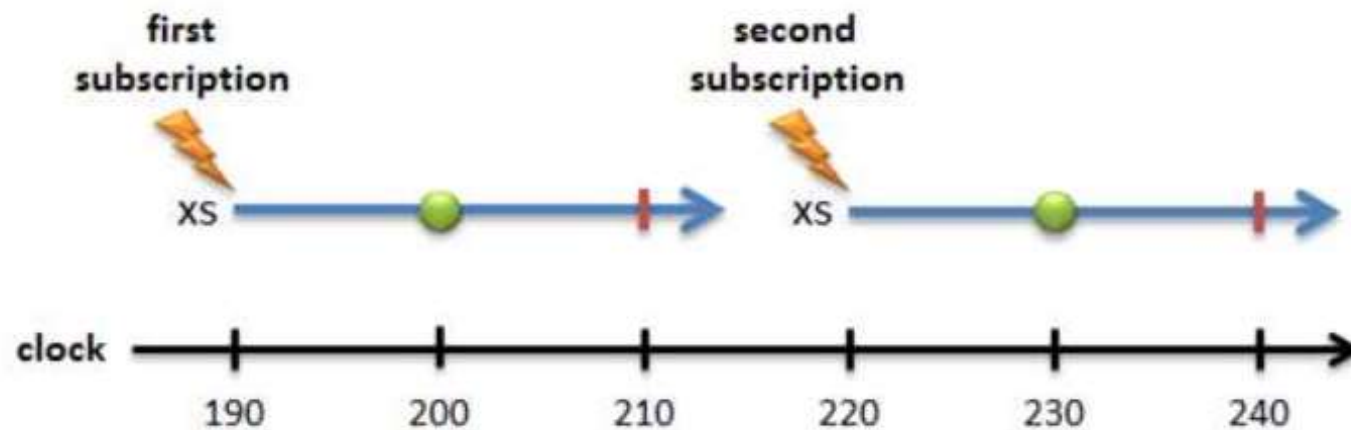
Observable nie są takie jak myślisz!

- Lazy computation i nie zawsze asynchroniczna, czyli bardziej jak funkcja niż promise!

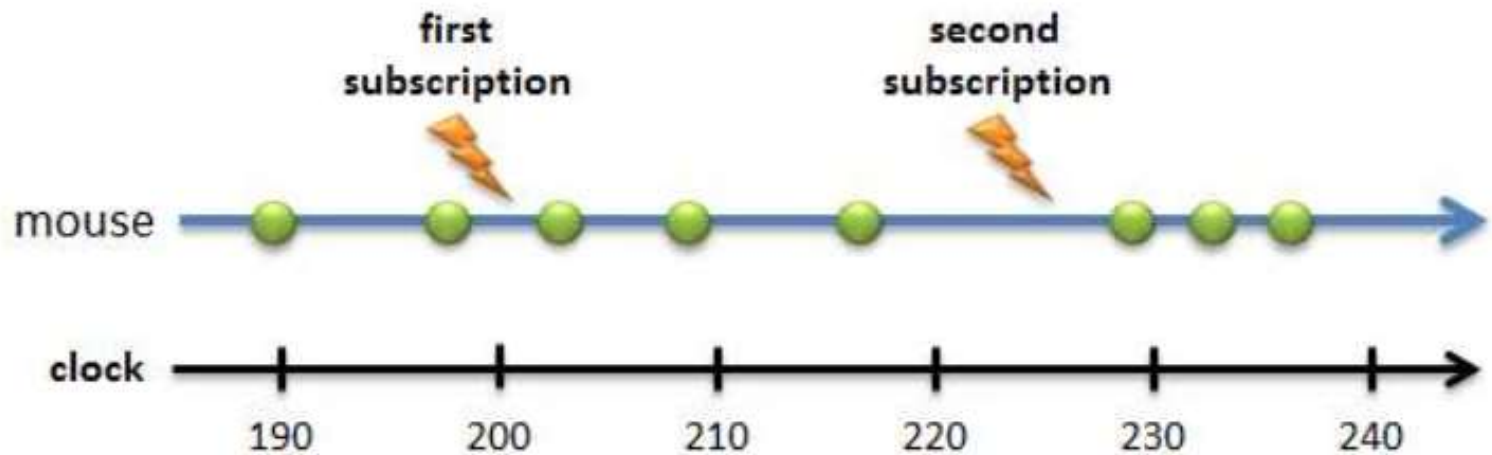
```
const hello$ = new Observable(observer => {  
  console.log('2. Drugi');  
  observer.next('3. Trzeci');  
  observer.next('4. Czwarty');  
});  
  
console.log('1. Pierwszy');  
  
hello$.subscribe(x => console.log(x));  
  
console.log('5. Piaty');
```

```
// Wynik:  
// 1. Pierwszy  
// 2. Drugi  
// 3. Trzeci  
// 4. Czwarty  
// 5. Piaty
```

Cold Observables



Hot Observables



Typy Observable: Hot i Cold

Observables domyślnie są **zimne**:

- każda subskrypcja powoduje wywołanie konstruktora
- odsubskrybowanie skutkuje zamknięciem tej instancji obserwowanej

Mogą być też **gorące**:

- Przestają być leniwe
- Mogą produkować wartości nawet kiedy nikt nie słucha
- Wywołanie *subscribe* wiele razy nie skutkuje wywołaniem konstruktora wielokrotnie
- Mogą współdzielić przesyłane wydarzenia

Subskrypcja

- Specjalny obiekt reprezentujący zasób, najczęściej konkretne wykonanie danego Observable.
- Obiekt praktycznie ma tylko jedną metodę: **unsubscribe()**

```
const clock$ = new Observable( observer => {  
  const intervalId = setInterval( () => {  
    observer.next('tick');  
  }, 1000);  
  
  return () => clearInterval(intervalId);  
});  
  
const subscription = clock$.subscribe(val => console.log(val));  
  
subscription.unsubscribe();
```

Observable i RxJS

RxJS

- Część większej rodziny Rx* jak np. RxJava, Rx.NET, RxScala...
- Oryginalnie stworzona i rozwijana przez Microsoft (do v.4), obecnie (od v.5) przepisana i rozwijana m.in przez Google i Netflix.
- Zawiera implementację Observable i innych typów, ale przede wszystkim posiada bogatą kolekcję **Operatorów**.

Operators

- Operatory służą do komponowania obserwowanych i łatwego deklaratywnego zarządzania asynchronicznym kodem.
- Najczęściej występują w formie metod dostępnych na instancji obiektu Observable.
- Użycie operatora nigdy nie zmienia (nie mutuje) oryginalnego Observable ani wartości jakie przez niego przechodzą. Zamiast tego zawsze tworzy nowy Observable na bazie istniejącego.

Creation Operators

- ♦ create
- ♦ of
- ♦ empty
- ♦ fromEvent
- ♦ fromPromise
- ♦ interval
- ♦ ...

Transform Operators

- ♦ map
- ♦ mergeMap
- ♦ pluck
- ♦ window
- ♦ buffer
- ♦ scan
- ♦ ...

Filtering Operators

- ♦ filter
- ♦ take
- ♦ last
- ♦ debounce
- ♦ throttle
- ♦ distinctUntilChanged
- ♦ ...

Combination Operators

- ♦ combine
- ♦ concat
- ♦ merge
- ♦ race
- ♦ zip
- ♦ ...

Multicasting Operators

- ♦ cache
- ♦ multicast
- ♦ publish
- ♦ share
- ♦ ...

Error Handling Operators

- ♦ catch
- ♦ retry
- ♦ retryWhen

Przykładowe operatory

```
const numbers$ = Observable.from([1, 2, 3, 4, 5]);  
  
const doubleNumbers$ = numbers$.map(num => num * 2);  
  
doubleNumbers$.subscribe(num => console.log(num));
```

```
// Output:  
// 2  
// 4  
// 6  
// 8  
// 10
```

```
const numbers$ = Observable.from([1, 2, 3, 4, 5]);  
  
const smallNumbers$ = numbers$.filter(num => num < 4);  
  
smallNumbers$.subscribe(num => console.log(num));
```

```
// Output:  
// 1  
// 2  
// 3
```

```
const numbers$ = Observable.from([1, 2, 3, 4, 5]);  
  
numbers$  
  .filter(num => num > 2)  
  .map(num => num * 2)  
  .subscribe(num => console.log(num));
```

```
// Output:  
// 6  
// 8  
// 10
```

Tworzenie Observables

```
Rx.Observable.fromArray([1, 2, 3]);
```

```
Rx.Observable.fromEvent(input, 'click');
```

```
Rx.Observable.fromEvent(eventEmitter, 'data', fn);
```

```
Rx.Observable.fromCallback(fs.exists);
```

```
Rx.Observable.fromNodeCallback(fs.exists);
```

```
Rx.Observable.fromPromise(somePromise);
```

```
Rx.Observable.fromIterable(function*() {yield 20});
```

Własne operatory

```
const echo = (input$) => {  
  return new Observable(observer => {  
    input$.subscribe({  
      next: val => {  
        observer.next(val);  
        observer.next(val);  
      },  
      error: err => observer.error(err),  
      complete: () => observer.complete()  
    });  
  });  
}
```

```
const numbers$ = Observable.from([1, 2, 3, 4, 5]);
```

```
const echoNumbers$ = echo(numbers$);
```

```
echoNumbers$.subscribe(num => console.log(num));
```

// Output:

// 1

// 1

// 2

// 2

// 3

// 3

// 4

// 4

// 5

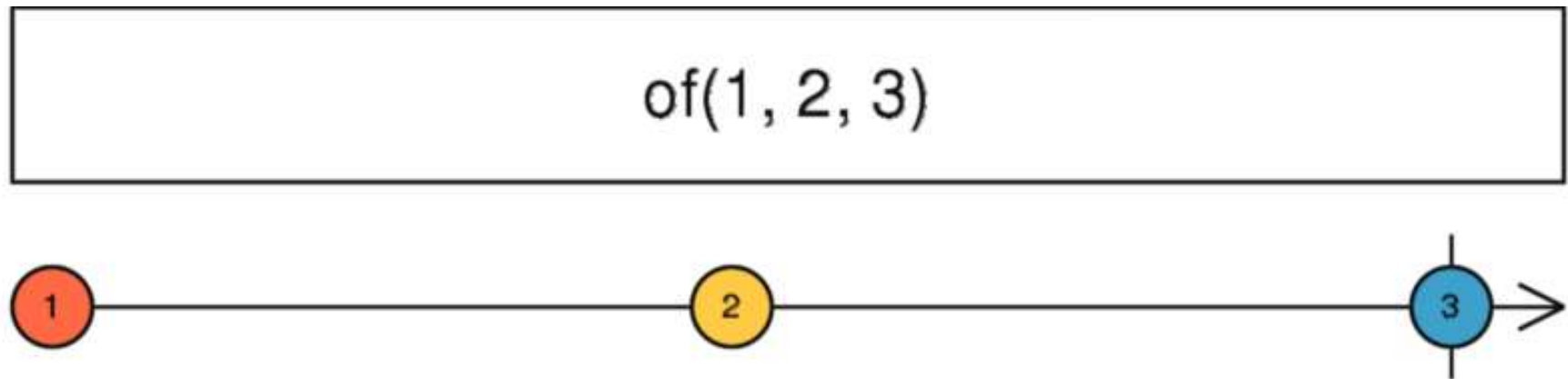
// 5

```
var range = Rx.Observable.range(1, 10) // 1, 2, 3 ...  
    .filter(function(value) { return value % 2 === 0; })  
    .map(function(value) { return "<span>" + value + "</span>"; })  
    .takeLast(1);  
  
var subscription = range.subscribe(  
    function(value) { console.log("last even value: " + value); });  
// "last even value: <span>10</span>"
```

Przykłady tworzenie

```
var bar = Rx.Observable.create(function (observer) {  
  try {  
    console.log('Hello');  
    observer.next(42);  
    observer.next(100);  
    observer.next(200);  
    setTimeout(function () {  
      observer.next(300);  
      observer.complete();  
      observer.next(400);  
    }, 1000);  
  } catch (err) {  
    observer.error(err);  
  }  
});
```

```
bar.subscribe(  
  function nextValueHandler(x) {  
    console.log(x);  
  },  
  function errorHandler(err) {  
    console.log('Wystapil blad: ' + err);  
  },  
  function completeHandler() {  
    console.log('OK - zrobione');  
  }  
);
```



```
var numbers = Rx.Observable.of(10, 20, 30);
numbers.subscribe(x => {
    console.log(x);
}, err => {
    console.log(err);
}, () => {
    console.log('done');
});
```

```
//output
/*
10
20
30
done
```

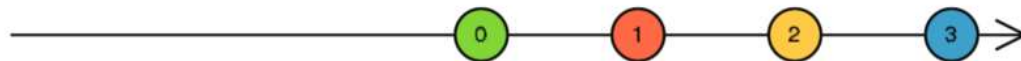
fromEvent(element, 'click')



```
var clicks = Rx.Observable.fromEvent(document, 'click');  
clicks.subscribe(ev => console.log(ev));
```

```
var result = Rx.Observable.fromPromise(fetch('http://myserver.com/'));  
result.subscribe(x => console.log(x), e => console.error(e));
```

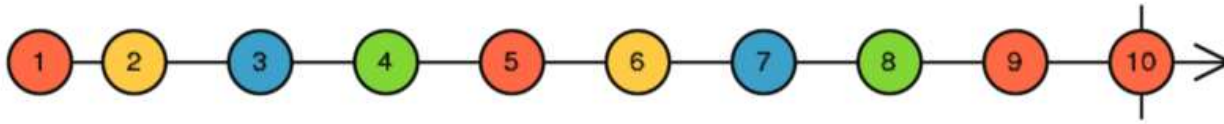
timer(3000, 1000)



```
var numbers = Rx.Observable.timer(5000);  
numbers.subscribe(x => console.log(x));  
//emits 0 po 5 s. i koniec .
```

```
var numbers = Rx.Observable.timer(3000, 1000);  
numbers.subscribe(x => console.log(x));  
// emituje 0 po 3 s. a nastepnie kolejne numery co  
sekunde
```


range(1, 10)



```
var numbers = Rx.Observable.range(1, 10);  
numbers.subscribe(x => console.log(x));
```

/ output

1

2(po 1 sec)

3(po 1 sec)

4(po 1 sec)

5(po 1 sec)

...

*/

interval(1000)



```
var numbers = Rx.Observable.interval(1000);  
numbers.subscribe(x => console.log(x));
```

Inne

