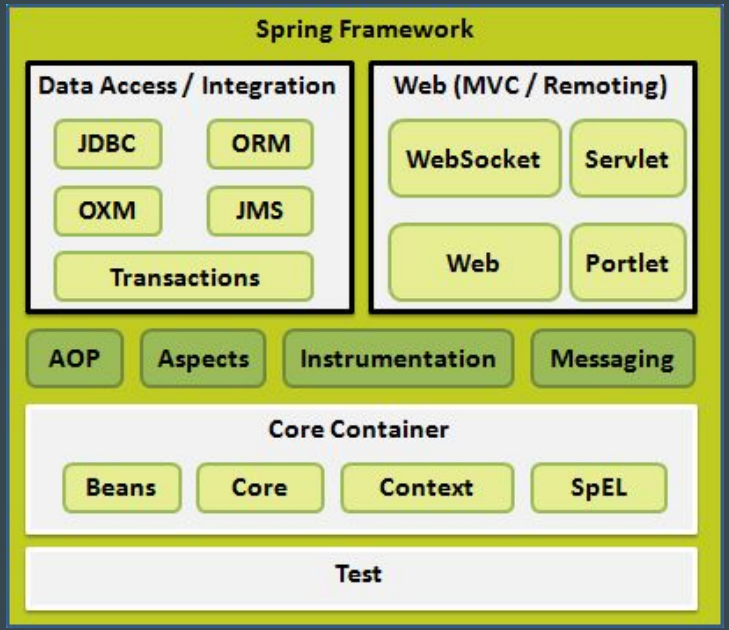
Spring Framework/ Javascript

...

Tomek Belina

Spring Framework - podstawy

- wsparcie tworzenia aplikacji korporacyjnych
- dostarcza narzedzi do **integrowania** poszczegolnych czesci aplikacji pomiedzy soba oraz innymi frameworkami
- architektura
  - **kontener** steruje cyklem zycia obiektow (centralny rejestr komponentow aplikacji)



Spring Framework - zagadnienia

- Dependency Injection (DI) - wstrzykiwanie zaleznosci
  - settery/konstuktory
- Inversion of Control (IoC) - odwrócenie sterowania
  - dostarczenie uslug pomocnicznych do uslugi , aby po zainicjowaniu byla gotowa do dzialania
  - obiekty umieszczane w kontenerze sa obiektami POJO
- Konfiguracja kontekstu Spring'a: xml, JavaConfig, adnotacje
- Aspekty (AOP), Transakcje
- JPA, Spring Data
- Spring MVC/ REST-API
- Spring Security
- Spring Boot
- SpEL

Moduly Spring'a

- **WebFlow**
  - dobrze nadaje sie do modelowania flow dla jednego usera, gdzie uzytkownik przechodzi caly proces od poczatku do konca (wizardy) np. aplikacja do zgłaszania szkody,
  - wada: brak persystencji danych (np.JBPM)
- **Spring Batch**
  - wykorzystywany do modelowania danych w trybie wsadowym
  - mozliwosc zrównoleglenia na kilka watkow
  - wsparcie dla wielu konektorów (excel, xml, txt)
- **Spring Integration**
  - umozliwia uproszczone wsparcie, zalecane uzywanie **Camel'a**
- **Spring Workflow**
  - wersja uproszczona, zalecane uzywanie **Activity**

Spring4

- pelne wsparcie dla **Java 8**
- usuniecie elementow deprecated z poprzednich wersji (templates)
- zalecana konfiguracja przez **JavaConfig** i adnotacje
- Profile
- **WebSocket**, SockJs, **STOMP**

Spring

- zasiegi beanów: singleton, prototype, request, session, globalSession
- programowanie poprzez **interfejsy**
- **konfiguracja** aplikacji przez JavaBeans
- aplikacja **nie** powinna byc uzalezniona od API Springa
- **loose-coupling**:
  - programowanie przez interfejsy
  - stosowanie mechanizmu IoC/DI
  - uzywanie adnotacji
  - uzywanie aspektow

Adnotacje Spring

- **@Component, @Controller, @Service, @Repository - beans**
- **@Autowired**: zalecane dodawanie w setterach zamiast konstruktorach
- **@Value**: wstrzykiwanie danych z placehoderów
- @**ComponentScan** automatycznie dodaje ‘annotation-config’
- @PostConstruct, @PreDestroy
- **@Profile**
    - profilowanie, mozliwosc automatycznego przeladowywania beanow zarzadzanych przez kontener Spring’a w runtime, bez restartu aplikacji (np. Data Sources)

Spring AOP

- Spring AOP uzywa elementow skladni **AspectJ**
- w Spring'u aspekty dzialaja przez **proxy:**
  - tworzony jest obiekt proxy dla kazdego obiektu wzbogaconego poprzez aspekt
  - uzytkownik pracuje na proxy, tworzenie obiektu proxy jest przezroczyste dla uzytkownika
  - proxy wzbogaca dzialanie obiektu o aspekty, a nastepnie deleguje wykonanie zasadniczej logiki do obiektu docelowego(target)
- nie mozna dodawac aspektow to pól tylko do **metod**
- konfiguracja:
  - adnotacje:
    - **@Pointcut -** definiuje miejsce w systemie ktore ma zostac wzbogacone o funkcjonalnosc aspektu
    - **@EnableAspectJAutoProxy**
    - **@Aspect,@After**("execution(public * someMethod(..))"**), @Before, @AfterReturning, @AfterThrowing, @Around**
  - xml:
    - <aop:aspectj-autoproxy />
    - <aop-config>, <aop:aspect ref="">, <aop:aspect after method pointcut="">, <aop:aspect before method pointcut="">, <aop:aspect around method pointcut="">

Obiekty specjalne/ Edytory

- postprocesor **BeanPostProcessor()**
  - manipulacja beanami przed/po inicjalizacji
  - metody: postProcessBeforeInitialization(), postProcessAfterInitialization(), musza zwracac obiekt w przeciwnym razie nie zostanie dodany do kontekstu Springa
- postprocesor **BeanFactoryPostProcessor()/PropertyPlaceholderConfigurer**
  - pozwala ingerowac w cykl dzialania kontenera wczesniej niz BeanPostProcessor - przed powolaniem do zycia beanow
- internacjonalizacja:
  - konfiguracja beana **ResourceBundleMessageSource**
  - JavaConfig: bean messageSource() musi miec dokladnie taka nazwe
- konwersja typów złozonych: **PropertyEditorSupport()**
  - przeksztalcenie ciagu znakow wskazanego jako wartosc atrybutu na jego reprezentacje obiektowa
  - metody: setAsText(String text)

```
@Value("rtrtrtrtr:wewewewew:23232")
private BasicAddress basicAddress;
```

JPA

- **EntityManager - zajmuje sie wszystkimi operacjami zw. z utrwalaniem danych**
  - **kontekst trwalosci** (*persistence context*): zbior wszystkich zarzadzalnych encji przez EM
  - encja **zarzadzalna** - jezeli EM otrzyma referencje do tego obiektu
  - EM jest implementowany przez **dostawce trwalosci(*persistence provider*) -** dostarcza silnik JPA
  - **jednoskta trwalosci** (*persistence unit*): dostarcza konfiguracji fabryki do utworzenia EM (*EntityManagerFactory*), konfiguracje w persistence.xml
  - **@PersistenceContext** - cyklem zycia EM zarzadza kontener (container-managed EM)
  - zalecane wykorzystywanie implementacji **LocalContainerEntityManagerFactoryBean** zarzadzanej w kontekscie Springa, brak koniecznosci tworzenia pliku persistence.xml
- relacje
  - **@OneToOne** (nie działa tryb ładowania Lazy), **@OneToMany**(@ManyToOne), **@ManyToMany**(@JoinTable)
- kasady
  - kaskadowe wykonywanie operacji na encjach powiazanych ze soba
  - typy: **CascadeType.PERSIST/REFRESH/REMOVE/MERGE/DETACH/ALL**

JPA

- dziedziczenie: pozwala odziedziczyc stan i zachowanie klasy bazowej
  - **@DiscriminatorValue**, **@Inheritance**(strategy = InheritanceType.SINGLE_TABLE/TABLE_PER_CLASS/JOINED))
    - SINGLE_TABLE: najbardziej wydajna, rozrzutna pod wzgledem przestrzeni bazy danych
    - JOINED: oszczednosc miejsca w bazie danych, kosztowna podczas zapisu/odtwarzania obiektow (tym bardziej im glebsza struktura obiektow)
- klasy bazowe(**@MappedSuperclass**):
  - zawieraja stan i zachowanie wspoldzielone przez encje ktore po nich dziedzicza, najczesciej abstracyjne
  - same nie sa utrwalane, nie moga uczestniczych w zapytaniach ani byc celem relacji
- **Lazy fetching**
  - brak gwarancji ze zostanie zastosowane
  - nie wnosi poprawy wydajnosci dla typow prostych
  - stosowac gdy: tabela ma wiele kolumn, kolumny sa dlugie
- Cache: zalecane uzywanie produkcyjnie **EHCache** (wspiera klastrowanie)
- **hashCode(), equals()** - wymagane jezeli encje zawieraja kolekcje hashujace
- **'log4j.logger.org.hibernate.type=TRACE' -** wyswietla wartosci argumentow SQL z Hibernate

JPA

- **Naming Query** - zapytania nazwane, w meta-danych, szybsze bo sa gotowe podczas startu aplikacji, weryfikowana jest poprawnosc skladniowa SQL przy starcie aplikacji
- zalecane uzywanie automatycznie generowanych identyfikatorow
  - @GeneratedValue(strategy = GenerationType.AUTO/TABLE/IDENTITY/SEQUENCE
- tryb **SEQUENCE**
  - zalecane generowanie z poziomu EM, niebezpieczenstwo istnienia tych samych kluczy przy własnej implementacji
  - normalnie sekwencje sa generowane dla kazdej tabeli oddzielnie
  - AUTO - jedna sekwencja dla wszystkich tabel
- **persist()** vs **merge()**
  - persist(): identyfikator dostajemy od razu
  - merge(): identyfikator dostajemy dopiero po zakomitowaniu transakcji
- klucze zlozone
  - osadzone (**@EmbeddedId**)
  - zwykle klasy (**@IdClass**(a.class)/ @Id)

JPA

- **Hibernate-JPA** vs **Hibernate-SessionFactory**
  - wolniej działaja zapytania w JPA
  - Spring Data dziala tylko z JPA
- zalecane uzywanie **adnotacji JPA** zamiast Hibernate: czytelniej i uniwersalniej
- konfiguracja mapowania:
  - odejscie od plikow xml (*.hbm), docletów, aktualnie pliki adnotacje w plikach java
  - **@Entity, @Table, @Id, @Column**, **@Enumerated**, @Transient, @PrePersist (ustawianie daty modyfikacji), @Embeddable**,** @SecondaryTable(stan encji utwalony w kilku tabelach), @Lob/@Blob, @Temporal
- zalecane uzywanie **ORM** wzgledem tradycyjnego JDBC:
  - problemy wynikaja zwykle z bledami w zapytaniu, modelu oraz cache 2 poziomy
- zapytania JPA:
  - **JPQL** (JPA Query Language)
  - **Criteria API -** dynamiczne budowanie zapytan, nie trzeba “sklejac zapytan”, budowanie SQL na podstawie wyboru przez uzytkownika
  - **SQL** (Native SQL)

JPA

- **Embedded Data Sources** - wykorzystywane w celach dewelopersko-testowych
  - **Derby** - w pełni funkcjonalne (tryb serwer)
  - **H2** - polecana, zawiera wyszukiwanie pełnotekstowe
  - **HSQL** - posiada ‘jorunale’, wykorzystywane jako dane wsadowe odczytywane z pamieci podczas problemu z baza danych
- Silniki bazy danych **ORM**:
  - Oracle **TopLink**
  - **Hibernate**
  - Apache **iBatis**
  - **JDO**
- Bazy nierelacyjne:
  - **MongoDB** (dokumentacyjna)
    - przechowywanie indeksów pełnotekstowych (dobra integracja z lucene/solr)
    - dla schematów zmiennych (różny, zmienny model,
  - **Redis** (para klucz-wartość)
    - wykorzystywany do przechowywania danych w cache, wspiera klastrowanie
  - **Neo4j**(grafowa: nody i relacje)
    - dla skomplikowanych relacji (bez JOIN'ów), szybkie przejscia po drzewku

JPA: Spring Data

- konfiguracja w xml: <jpa:repositories base-package=”x.y.x”>
- **CrudRepository**, **JPARepostory**: impelementacja w DAO
- Spring Data automatycznie dodaje **@Transactional** i tworzy transakcje
- umozliwia zagniezdzanie zapytan (np. findBySellerAddressCity())

JPA: Transakcje

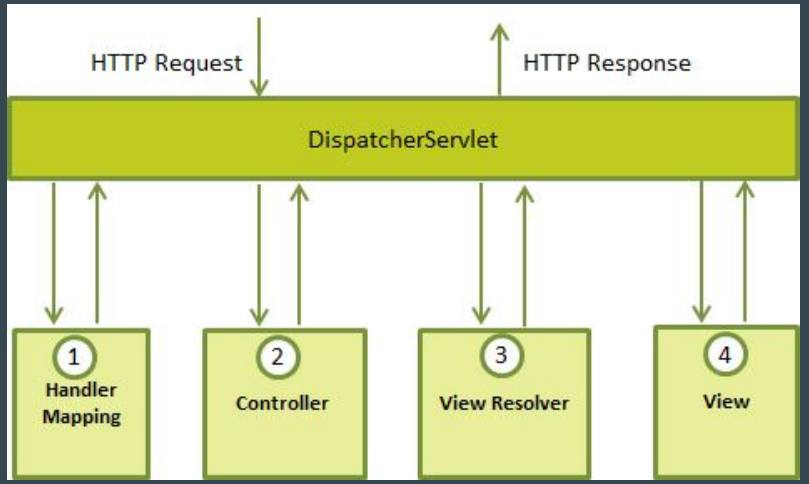
- transakcje rozproszone (**XA-datasrouces**):
  - dostawca musi obslugiwac JTA (*Java Transaction API*)
  - kilka EM, ktore wstrzykujemy, ma dostep do roznych baz danych
  - zrodla danych nalezy skonfugurowac jako XA-Datasource
  - jezeli operacja na dowolnej bazie zakonczy sie niepowodzeniem, transakcja jest wycofywana na obu bazach danych
- serwer **Atomikos -** dedykowany do transakcji JTA, koordynuje zarzadzanie transakcjami na wszystkich bazach

Testy jednostkowe jUnit

- adnotacje:
  - @RunWith(SpringJUnit4ClassRunner.class)
  - @ContextConfiguration(locations = "/context.xml") //(classes = JavaConfig.class)/
  - @TransactionConfiguration(defaultRollback = false)
  - @Test
  - @Transactional
  - @ActiveProfiles("test")

Spring MVC

- implementacja interakcji z uzytkownikiem
- 3 warstwowa struktura aplikacji:
  - warstwa klienta (‘cienki/gruby klient’)
  - warstwa komunikacji, serwisów (REST)
  - warstwa logiki biznesowej
- **Dispatcher Servlet**: wzorzec Front Controller
- obiekty Spring MVC:
  - kontrolery
  - mapowania kontrolerow
  - ModelAndView
  - Mapowanie widokow
  - Mapowanie wyjatkow



Spring MVC

- konfiguracja aplikacji webowej
  - web.xml: DispatcherServlet, listenery
  - java: kontrolery
  - deskryptor Spring: kontroler, ViewResolver, HandlerMapping
- konfiguracja JavaConfig (nie wymaga pliku web.xml)
  - **WebApplicationInitializer**: konfiguracja DispatcherServlet
  - **WebMvcConfigurerAdapter**: ViewResolver, HandlerMapping
  - @Import - pozwala importowac rozne pliki konfiguracyjne
- kontrolery:
  - adnotacje:
    - **@Controller/ @RestController**
    - **@RequestMapping**("/"),@RequestMapping(value = "/{id}", method = RequestMethod.GET, produces = MediaType.APPLICATION_JSON_VALUE)
    - @PathVariable, @RequestParam
  - **ModelAndView** - zwracany obiekt przez kontroler, wskazuje nazwe widoku, @ModelAttribute
  - **DTO** - uproszczony model o spłaszczonej strukturze, przykłady mapperów:
    - **Model Mapper, Orika -** szybszy od Model Mappera

REST-API

- zalecane uzywanie Spring'owego API zamiast np.Jersey
- zwracanie statusów: adnotacje **@ResponseEntity/ @ResponseStatus**
- dobre praktyki: **http://www.restapitutorial.com**
- klienci: **Rest Console, Postman**
- zadanie POST:
  - zalecane zwracanie URL do utworzonego obiektu/zasobu (zamiast calego obiektu),
  - zwracamy status 201 (Created)
- zwracamy status 204 (No Content) zamiast 200 jezeli wszystko poszlo pomyslnie

Spring Security

- dostarcza metod autentykacji i autoryzacji dla aplikacji opartych na Spring
- konfiguracja bezpieczenstwa w sposob **deklaratywny**
- wykorzystuje **AOP**
- **Menadzer autentykacji -** zarzadza dostawcami autentykacji
- dostawcy/providerzy autentykacji - wykonuje autentykacje wzgledem konkretnego zrodla danych z uzytkownikami
  - DAO (baza danych)
  - LDAP
  - Remote (wywolanie serwisu)
  - X509
  - Jass
- konfiguracja dostepnosci URL w pliku konfiguracyjnym Spring
- wlaczanie obslugi bezpieczenstwa w pliku **web.xml**

Spring Security

- zabezpieczanie metod pdbywa sie przez Spring AOP
  - xml: <intercept-metdod>
  - AspectJ: <protext-pointcut>
  - adnotacje
    - **@Secured** - z pakietu Spring Security
    - **@RolesAllowed** -z JSR-250
- JavaConfig:
  - **@EnableWebSecurity** - wlaczenie zabezpieczenia zasobow web
  - **@EnableGlobalMethodSecurity(securedEnabled = true, jsr250Enabled = true)**' - - wlaczenie zabezpieczenia wywolania metod
  - login() - zawsze dajemy permitAll() aby nie spowodowac zapetlenia,
  - metoda configure() - zabezpieczenie na poziomie metod oraz zadan http

Uwierzytelnianie uzytkownikow

- **BasicAuth -** podajemy login i haslo (domyslne)
- Spring Security zawiera moduł **OAuth(2)**
  - zabezpieczenie komunikacji z wykorzystaniem mechanizmu tokenów (handshake)
  - wysyłamy do serwera dane do logowania, serwer odsyla token, w kazdym zadaniu token jest odsylany w naglowku, jezeli serwer uzna ze jest niepoprawny - odrzuca polaczenie
  - tokeny moga byc przechowywane w pamieci, bazie danych itp., po stronie uzytkownika jest wylacznie konfiguracja, Spring zajmuje sie obsluga
  - zalecany sposób uwierzytelniania - tokeny ustawiona w **headerach**, a nie ciasteczkach

Spring Boot

- standaryzacja tworzenia aplikacji (***convention over configuration***)
- zawiera **pluginy** z wbudowana konfiguracja, ktora mozna nadpisywac
- konwencja konfiguracji przez **JavaConfig**
- wlaczamy funkcjonalnosci w konfiguracji przez adnotacje **@Enable**...
- dodajemy w projekcie Spring Booťa jak **parent**, ustawia zmienne które dodajac go jako dependencje musielibysmy ustawic
- adnotacja:
  - **@SpringBootApplication**
  - @EntityScan(basePackages = ""), @WebAppConfiguration
- zaleznosci mavena:

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Przydatne narzedzia

- **Terrakota** - odpalanie watkow na roznych maszynach, ktore sa traktowane jako jeden JVM, umozliwia klastrowanie JVM'ów
- **SVN vs GIT -** w git mamy lokalne kopie repozytoriów,zaleta jest szybkie przeladzanie miedzy branchami
- **Dynamic Code Evolution VM** (java 9) - umozliwia zmian klas w locie bez restartu serwera, ma stac sie czescia javy
- **Spring loaded** - dziala ze Spring Boot, alternatywa do JRebel, Dynamic Code Ev.
- **WebJars -** po stronie serwera dodajemy dependencje w mavenie, nie dołaczamy juz plikow bibliotek do projektu (*.js, *.html, *.css)
- **Swagger** - narzedzie do automatycznego generowania dokumentacji API-REST' owego, wykonuje to na podstawie adnotacji, prezentuje metody, przykladowe requesty, umozliwia testowanie

Javascript

...

Javascript - podstawy

- deklarujemy kazda zmienna z uzyciem **‘var’**
- nawiasy klamrowe nie wyznaczaja zasiegu zmiennej
- deklarujemy zmienne na poczatku funkcji
- typu danych: **String, Number, Boolean, undefined** (bez przypisania wartosci, to nie to samo co null)
- nalezy tworzyc obiekty przez **literaly** ([], {})
- uzywamy tozsamosci przy porownywaniu obiektow **“===”**
- nalezy ladowac skrypty na koncu pliku HTML (body)
- nie zamykamy skryptu, uzywamy konwencji <script src=””> </script>
- ‘**use strict**’ - umozliwia zglaszanie bledow przez przegladarke, zalecane
- fun - referencja do funkcji, fun() - wywolanie funkcji

This

- **this()** moze wskazywac na:
  - window() - poza funkcja
  - window() - w funkcji ale bez obiektu
  - obiekt - w funkcji obiektu
  - worzony obiekt - w funkcji konstuujacej
- w javascript obiekty sa tablicami asocjacyjnymi
- funkcja **delete** kasuje klucz a nie wartosc, undefined - tylko gdy usuniemy wartosc
- **funkcje konstruujace** - jezeli chcemy dodac wiele obiektow danego typu, wowolywana z operatorem new() (jezeli o nim zapomnimy dzialamy na window())

Javascript

- **hermetyzacja** kodu - realizowana przez zagniezdzanie funkcji
- **scope leksykalny** - liczy sie miejsce definicji obiektu, a nie miejsce wywolania
- **prototype -** dodaje wspolny klucz do obiektu prototype, wydzielamy wspolne aspekty obiektu (jako forma dziedziczenia)
- **call()/apply()** - umozliwia przebindowanie funkcji, okresla w ramach jakiego kontekstu uruchamiamy funkcje, zmieniamy wskazanie this'a, wykorzystywane aby this wskazywal na obiekt wywolujacy zdarzenie
- realizacja **dziedziczenia** w Javascript:
  - przez **prototyp** - w funkcji konstruujacej klasy potomnej ustawiamy prototyp na instancje obiektu konstruujacego klase bazowa
  - przez **mix'y** - kopiowanie referencji do kluczy z jednego obiektu do drugiego

Reactive programming

- programowanie strumieniowe (http://reactivex.io/language.html)
- umozliwia reagowanie na wiele zdarzen jednoczesnie i je obserwowac
- polaczenie obserwatora z promise'mi
- mozna zakladac wiele obserwatorow na jednym strumieniu,
- na strumien mozna nakladac filtry, mozna po drodze zmieniac jego dzialanie przez wprowadzanie funkcji modyfikujacych
- strumienie vs promisy:
  - strumien jest nieskonczony, moze trwac dowolnie dlugo, zdarzen moze byc nieskonczenie wiele, mamy zrownoleglenie operacji jednoczesnie/asynchronicznie z wielu zdarzen, ktore sa scalane
  - promisy jest jednorazowe, opiera sie na sekwencyjnym pobieraniu danych

Reactive programming

- Zalety:
  - izoluje klienta do synchronicznosci/asynchronicznosci serwera
  - umozliwia synchronizacje danych z roznych serwerow
  - umozliwia polaczenie kilku asynchronicznych serwisow w jeden serwis
  - jezeli mamy duzo zadan mozna konsolidowac wykowanie w jeden strumien i nim operowac

Reactive programming

- implementacja 1 :

```
// utworzenie 1 strumienia
var requestStream = Rx.Observable.just('https://api.github.com/users');
// dodanie 1 obserwatora na strumieniu
requestStream.subscribe(function(requestUrl){
    // utworzenie 2 strumienia
    var responseStrem = Rx.Observable.create(function(observer){
        $.getJSON(requestUrl)
             .done(function(response){
                observer.onNext(response);
             })
             .fail(function(xhr, status, error){
                observer.onError(error);
             })
             .always(function(){
                observer.onCompleted(); // zamykamy strumien
             });
    });
    // dodanie 2 obserwatora na strumieniu
    responseStrem.subscribe(function(response){ console.log(response); })
});
```

just() - tworzy strumien i od razu laduje dane
create() - tworzy strumien, dane pojawia sie na onNext()

Reactive programming

- implementacja 2 :

```
// utworzenie 1 strumienia
var requestStream = Rx.Observable.just('https://api.github.com/users');

// utworzenie 2 strumienia
var responseStrem = requestStream.flatMap(function(url){
        return Rx.Observable.fromPromise($.getJSON(url));
});

// dodanie obserwatora na strumieniu 2
responseStrem.subscribe(function(reposne){
        console.log(response);
});
```

bower

- sluzy do dodawania zaleznosci
- nalezy utworzyc plik **.bowerc** w glownym pliku projektu
- instalacja:
  - instalacja nodejs
  - instalacja bower (npm install -g bower)
  - inicjalizacja bower (bower init)
  - przykladowo instalacja jquery (bower install jquery --save)

gulp

- sluzy do definiowania zadan/taskow, przykladowo:
  - konktatenacja/łaczenie plików (gulp-concat)
  - zaciemnianie i minifikacja kodu (gulp-uglify)
  - uruchomiene serwera www (gulp-webserver)
- instalacja: npm install -g gulp

AngularJS 1x

- SPA, aplikacje bez przeladowywanie stron
- automatyczny binding,
- podzial na moduly, wzorzec **MVC**,
- wstrzykiwanie zaleznosci,
- rozdzielenie logiki od widoków,
- pozwala rozszerzac standardowy kod HTML(**dyrektywy**)
- **kontroler** - jego zadaniem jest przygotowanie scope, wystawia dane, nie zawiera logiki biznesowej, za kazdym razem tworzona jest nowa instancja
- **serwis** - zawiera logike biznesowa, realizuje komunikacje z serwerem, singeltony
- realizacja aplikacji typu flow - zapamietywanie kontekstu uzytkownika w LocalStorage/SessionStorage,IndexDB

AngularJS 1x

- **dyrektywy**
  - zalecane dla zasiegu element(E) oraz atrybutu(A)
  - funkcja **link**() odpalana jest tylko raz
  - na parametrze **'element'** mozemy modyfikowac jego zachowanie przez jquery
  - **scope:**
    - true: tworzy niezalezny scope
    - izolowany - izolowana komunikacja ze scope'ami nadrzednymi ("=" - parametr, "@" - atrybut)
- zdarzenia
  - obsluga zdarzen na zasadzie obserwatora, on() - typ zdarzenia na ktorym nasluchujemy
  - przesylanie wiadomosci miedzy scope'ami z wykorzystaniem metod:
    - **emit()** - emituje do góry, np. kontroler powiadamia root scope'y o zdarzeniu a te z kolei inne kontrolery
    - **broadcast()** - emituje w dół, np. root scope informuje rozne kontrolery o zdarzeniu, na kazdym scope kontrolerow nalezy ustawic on() na nasluchiwanie

AngularJS 1x

- implementacja

```
angular.module('usersManager', ['ngRoute', 'usersManager.users'])
        .config(['$routeProvider' ,function($routeProvider) {
                $routeProvider.otherwise({redirectTo: '/users'});

}]);


angular.module('usersManager.users', ['ngRoute'])
        .config(['$routeProvider' ,function($routeProvider) {
                $routeProvider.when('/users', {
                        templateUrl: 'app/users/users.html',
                        controller: 'UsersController',
                        controllerAs: 'usersCtrl'
                });
}]);
```

Aplikacja klient-serwer

- serwer
  - uruchomienie serwera www (nodejs)
- klient AngularJS
  - uruchomienie serwera www (npm start) - moduł ‘http-server’
  - **$transaleProvider**- realizuje tłumaczenia jezykowe (przez filtry)
  - **ControllerAS** - zapisuje dane do konkretnego scope’a, zasieg nie jest wspoldzielony
  - **$cacheFactory** - pobieranie danych z cache mapy
  - **promises -** asynchroniczne wywolanie zadan do serwera

Ciekawostki

- **Meteor** - framework SPA, integruje MongoDB, Node, Sockety, umozliwia tworzenie warstwy prezentacji w AngularJS
- Repozytorium :
  - https://github.com/tomekb82/SzkolenieSpring.git
  - https://github.com/tomekb82/Spring4.git
  - https://github.com/tomekb82/SzkolenieAngularJS.git
  - https://github.com/tomekb82/ReactiveProgramming.git

Dziekuje za uwage

...