<http://codebykris.com/najpopularniejsze-pytania-z-rozmow-kwalifikacyjnych-ze-spring-framework/#answer15>

# Uruchamianie projektu

Podstawowe komendy, które będziemy wykorzystywali w Mavenie zawsze poprzedzone są wywołaniem samego mavena poprzez wpisanie w konsoli polecenia **mvn** (np. mvn --version, które wykorzystaliśmy podczas konfiguracji). Podstawowe cele (goals), które mamy do dyspozycji to:

* **clean** - usuwa folder target generowany przez mavena, można to utożsamiać z wyczyszczeniem projektu do stanu pierwotnego
* **compile** - kompiluje wszystkie pliki z kodami źródłowymi i kopiuje je do odpowiednich folderów w wygenerowanym folderze **target**.
* **package** - najpierw kompiluje wszystkie pliki z kodami źródłowymi (cel goal), a następnie pakuje aplikację do wskazanego formatu (jar/war/ear) i umieszcza plik wynikowy w folderze target. Plik będzie miał domyślnie nazwę identyczną z identyfikatorem naszej aplikacji (artifactId) z doklejoną wersją.
* **validate** - walidacja poprawności projektu. Faza ta polega przede wszystkim na weryfikacji pliku pom.xml, nie dotyczy ona błędów w plikach .java
* **compile** - faza kompilacji, po której tworzony jest folder target z wynikowymi plikami .class
* **test** - uruchomienie testów jednostkowych znajdujących się w folderach src/test
* **package** - spakowanie aplikacji do wskazanego w pom.xml formatu
* **integration-test** - uruchomienie testów integracyjnych jeżeli istnieją
* **verify** - weryfikacja, czy wynikowa paczka jest poprawna
* **install** - umieszczenie paczki w lokalnym repozytorium mavena (.m2/repository). Nie należy mylić tej fazy wykonania z instalowaniem w rozumieniu instalacji aplikacji
* **deploy** - umieszczenie paczki w zdalnym repozytorium

## Tworzenie projektu

* **name** - dowolna nazwa
* **groupId** - początkowy fragment pakietu aplikacji, najczęściej zgodnie z konwencją odwrócona domena twórcy oprogramowania
* **artifactid** - unikalny identyfikator projektu, najczęściej jego nazwa, która stanowi dalszą część nazwy pakietów
* **version** - wersja aplikacji. Najpopularniejsza konwencja składa się z oznaczenia wersji Major-Minor-Incremental-Kwalifikator, czyli tak jak u nas 1.0.0-SNAPSHOT lub np. 2.5.7-Final. Trzy pierwsze elementy oznaczają po prostu kolejne wersje, które zmieniamy zgodnie z polityką firmy lub własnym uznaniem. Kwalifikator wskazuje, czy jest to wersja rozwojowa (SNAPSHOT), czy też finalna i już się nie zmieni (Final). Jedyny kwalifikator, który ma istotne znaczenie to SNAPSHOT, pozostałe takie jak Final, Beta, Alpha są zależne od naszego uznania.
* **modelVersion** - wersja schematu XML naszego pliku. W Mavenie w wersji 2 i 3 (z której korzystamy) wersja 4.0.0 jest jedyną dostępną.
* **packaging** - wskazanie do jakiego formatu docelowo chcemy spakować naszą aplikację

## Zależności

**dependencies**: zależności, czyli biblioteki potrzebne do zbudowania projektu

**Podgląd drzewa zależności**

w intellij: Skrót Ctrl + Alt + U

**Maven Tool Window**

W menu View możemy również włączyć okno "Maven Projects", z którego będziemy mieli wygodny podgląd struktury zależności, cyklu życia oraz wykorzystywanych pluginów. Możemy z niego wygodnie uruchamiać poszczególne fazy oraz cele

Intellij sam znajduje wymagane depencides i je pobiera oraz podpina do pom

zależności tranzytywne (przechodnie) są dodawane automatycznie.

Jeżeli w naszym pliku pom.xml samodzielnie dodamy bibliotekę spring-core w starszej wersji np. 3.2.15.RELEASE, to Maven pominie zależności tranzytywne z biblioteki spring-context uznając je za konfliktowe.

Sytuacja jest o tyle problematyczna, że w takim przypadku może dojść do niezgodności wersji i program się albo nie skompiluje, albo nie będzie działał tak jak powinien.

## Pluginy

<https://javastart.pl/kurs/spring/spring-maven/lekcja/spring-pluginy>

<https://maven.apache.org/plugins/>

wpisy **<plugin>** opakowane w **<plugins>**.

Błąd:

[ERROR] Source option 1.5 is no longer supported. Use 1.6 or later.

Dodać

<properties>

<maven.compiler.source>1.6</maven.compiler.source>

<maven.compiler.target>1.6</maven.compiler.target>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

Pakowanie jar <https://maven.apache.org/plugins/maven-jar-plugin/>

Pakowanie z zależnościami <https://maven.apache.org/plugins/maven-assembly-plugin/> i <https://maven.apache.org/plugins/maven-assembly-plugin/usage.html>

jar uruchamiamy z folderu target

W celu uruchomienia pluginów będziemy się posługiwali konstrukcją ***nazwaPluginu:goal***, które znajdziemy w dokumentacji danego pluginu.

W celu uruchomienia wtyczki w konsoli lub w eclipse powinniśmy więc wywołać polecenie **mvn clean compile assembly:single** czyli "wyczyść, skompiluj, spakuj z zależnościami"

## Fazy wykonania

<phase>package</phase>

<goals>

<goal>jar</goal>

Plugin ten został skonfigurowany w taki sposób, że podczas fazy package (<phase>package</phase>) powinien się wykonać cel pluginu o nazwie jar (<goal>jar</goal>). W podobny sposób możemy wygodnie wpiąć inne wtyczki do cyklu budowania aplikacji.

## Zasięg i import

W większości naszych programów będziemy wyróżniali co najmniej dwie fazy budowania projektu - testy oraz pakowanie (np. w przypadku aplikacji webowych do formatu .jar). Problem jaki się w tym miejscu pojawia to fakt, że podczas testów wykorzystywanych jest wiele bibliotek, które są nam zupełnie niepotrzebne w działającej aplikacji - np. JUnit, Mockito, Arquillian. Istnieją również biblioteki, które wykorzystywane są tylko w fazie wykonania i tutaj jako przykład możemy podać sterownik bazy danych.

W Mavenie zdefiniowanych jest 8 zasięgów, które możemy dodać do informacji o zależności:

***compile*** - zasięg domyślny, dodaje informację o zależności do classpath dostępnego zarówno w procesie kompilacji jak i uruchomienia.

***provided*** - biblioteka jest dostępna w czasie kompilacji, ale w czasie wykonania zakładamy, że kontener ją dostarczy. Przykładem są kontenery JEE takie jak Wildfly, w których mamy do dyspozycji implementacje poszczególnych specyfikacji (Hibernate jako JPA, Weld jako CDI itp.).

***runtime*** - zależność nie jest potrzebna do kompilacji, ale jest wykorzystywana w trakcie wykonania programu. Najprostszy przykład to wspomniany sterownik bazy danych.

***test*** - zależność potrzebna tylko w fazie testów

***system*** - zasięg podobny do provided, jednak wymaga jawnego wskazania ścieżki do biblioteki

***import*** - specjalny zasięg wykorzystywany do importowania plików pom

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

Junit tylko w testach, nie przy dewelopmencie

Używać głównie zasięgu **provided** i bez podawania wersji gdy mamy serwerowy nadrzędny pom.

Wersja springa z properties

<spring.version>4.3.0.RELEASE</spring.version>

a w dependency (zależności)

<version>${spring.version}</version>

# SPRING

Dependency injection

Wstrzykiwaniem zależności, zakładając nawet brak dostępu do jakiejkolwiek biblioteki wspierającej ten mechanizm, nazwiemy możliwość wstrzyknięcia dowolnej implementacji np. do pola klasy poprzez setter lub konstruktor.

Poliformizm – to interfejs i jego implementację.

**Konfiguracja xml (stara metoda)**

Bean w xml

<bean id="messageProducer" class="pl.javastart.beans.SimpleMessageProducer" />

<bean id="messagePrinter" class="pl.javastart.beans.MessagePrinter">

<property name="producer" ref="messageProducer"/>

</bean>

## Konfiguracja o adnotację

<https://kobietydokodu.pl/niezbednik-juniora-adnotacje/>

Main z ***AnnotationConfigApplicationContext – kontekst springa***

AnnotationConfigApplicationContext ctx =

new AnnotationConfigApplicationContext(ApplicationConfig.class);

MessagePrinter printer = ctx.getBean(MessagePrinter.class);

printer.print();

ctx.close();

*ApplicationConfig.java – do konfiguracji beanów*

## Bean i component

***@Configuration*** jest jedną z podstawowych adnotacji wykorzystywanych w Springu, podczas skanowania klas Spring tworzy odpowiednie obiekty, które w klasach tak opatrzonych są zdefiniowane.

***@Bean*** na poziomie metody sprawia, że jej wynik, czyli obiekt typu xyz staje się ziarnem zarządzanym przez kontener Springa. Może posiadać indentyfikator (imie) @Bean(name="myMessagePrinter")

Czym są ziarna Springa? Obiektami, które będą przez niego automatycznie utworzone i przez niego przechowywane.

Beany to obiekty w naszej aplikacji, które są zarządzane (a więc tworzone i inicjowane) przez Springa (w ramach kontenera IoC). Beanem nazywamy dowolny obiekt, który jest zarządzany przez Springa (nie ma w tym zakresie żadnych dodatkowych wymagań). Innymy słowy — wystarczy, że Spring o nim ‚wie’ i jest w stanie użyć go przy polach z adnotacją @Autowired.

Aby jakiś obiekt był beanem, możemy użyć jednej z adnotacji-stereotypów lub zadeklarować go wprost (więcej o tym pisaliśmy w [lekcji 9](https://kobietydokodu.pl/09-spring-mvc/)). Beanem w Springu może być (prawie) dowolny obiekt — w tym np. serwisy pisane przez Ciebie, wewnętrzne klasy Springa (np. implementacja ApplicationContext), ale także zwykłe Stringi czy Integery.

Jedynym wyjątkiem są kolekcje i mapy — Ponieważ pola tego typu są traktowane specjalnie w Springu, nie mogą one być beanami same w sobie. Jedynym rozwiązaniem jest ‚opakowanie’ ich w jakąś klasę przed użyciem (np. stworzenie klasy tylko z jednym polem — kolekcją, która nas interesuje).

**@Component -** Komponentami będziemy nazywali w Springu klasy, które są opatrzone jedną ze specjalnych meta adnotacji, dzięki czemu podczas tworzenia kontekstu Spring będzie mógł utworzyć na ich podstawie odpowiednie obiekty, bez konieczności definiowania ich w klasie konfiguracji przez metody oznaczone adnotacją @Bean.

<https://therealdanvega.com/blog/2017/05/17/spring-component-vs-bean>

@Component (and @Service and @Repository) are used to auto-detect and auto-configure beans using classpath scanning. There's an implicit one-to-one mapping between the annotated class and the bean (i.e. one bean per class). Control of wiring is quite limited with this approach, since it's purely declarative.

@Bean is used to explicitly declare a single bean, rather than letting Spring do it automatically as above. It decouples the declaration of the bean from the class definition, and lets you create and configure beans exactly how you choose.

***@ComponentScan* – służy do wskazania skanowanych klas I pakietów *basePackageClasses:***

@ComponentScan(basePackageClasses = MessagePrinter.class)

Alternatywą jest wskazanie nazwy pakietu do przeskanowania:

@ComponentScan(basePackages = "pl.javastart.beans")

Możemy też wskazać pakiet nadrzędny co spowoduje przeskanowanie wszystkich pakietów podrzędnych:

@ComponentScan(basePackages = "pl.javastart")

***@Autowired -*** Przy konfiguracji opartej o adnotacje będziemy korzystali z mechanizmu automatycznego wiązania z wykorzystaniem adnotacji. Służy do wstrzyknięcia obiektu (ref w xml). Uzupełniam tym obiekty klas.

Adnotacja @Autowired przyjmuje opcjonalny atrybut **required**. Jego ustawienie na wartość true (jest to wartość domyślna) sprawia, że w przypadku braku znalezienia odpowiedniego ziarna Spring rzuci wyjątek związany z brakiem odpowiedniej zależności. Ustawienie na wartość false sprawi, że daną zależność traktujemy jako opcjonalną.

**Wstrzykiwanie przez konstruktor -** @Autowired nad konstruktorem a nie polem klasy (tego używać, bo pozwala zobaczyć wykorzystywane zależności)

**Zgodnie z dobrą praktyką i zaleceniami Springa wstrzykiwanie przez konstruktor powinniśmy wykorzystywać do przekazywania zależności wymaganych, natomiast przez setter tych, które są opcjonalne. Możemy więc łączyć dwie techniki w jednej klasie.**

@Autowired(required = false), to w przypadku braku możliwości spełnienia zależności nic się nie stanie,

////////////////////////////////////

package pl.javastart.beans;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Component;

@Component

public class MessagePrinter {

private MessageProducer producer;

private MessageDecorator decorator;

@Autowired

public MessagePrinter(MessageProducer producer) {

this.producer = producer;

}

public MessageDecorator getDecorator() {

return decorator;

}

@Autowired(required = false)

public void setDecorator(MessageDecorator decorator) {

this.decorator = decorator;

}

public void print() {

String message = producer.getMessage();

message = decorator != null? decorator.decorate(message) : message;

System.out.println(message);

}

}

///////////////////////////////////

## Inject i primary

**@Inject – standardu wstrzykiwania zależności w aplikacjach Javy EE.**

<dependency>

<groupId>javax.enterprise</groupId>

<artifactId>cdi-api</artifactId>

<version>1.2</version>

</dependency>

**@Primary** – wykorzystywany przy konflikcie (kilka implementacji interfejsu) co generuje błąd ***NoUniqueBeanDefinitionException***

**Dodajemy do wybranego komponentu.**

@Component

@Primary

public class SimpleMessageProducer implements MessageProducer {

…

}

## Kwalifikator

***@Qualifier* -** mechanizm kwalifikatorów, czyli specjalnych adnotacji, które pozwalają jednoznacznie określić, który komponent chcemy wstrzyknąć do danego pola. Przyjmuje atrybut w postaci Stringa z nazwą

@Component

@Qualifier("simpleMessageProducer")

public class SimpleMessageProducer implements MessageProducer {

…

}

Możliwość definiowania własnych kwalifikatorów, czyli w rzeczywistości zwykłych adnotacji opatrzonych dodatkowo adnotacją @Qualifier.

///Tworzymy kwalifikator

import org.springframework.beans.factory.annotation.Qualifier;

@Target({ElementType.FIELD,

ElementType.METHOD,

ElementType.TYPE,

ElementType.PARAMETER})

@Retention(RetentionPolicy.RUNTIME)

@Qualifier

public @interface FileMessage {

}

-Zastosowanie takiego kwalifikatora:

@Component

@FileMessage

public class FileMessageProducer implements MessageProducer {

-Oznaczamy pole lub parameter

@Component

public class MessagePrinter {

private MessageProducer producer;

private MessageDecorator decorator;

@Autowired

public MessagePrinter(@FileMessage MessageProducer producer) {

this.producer = producer;

}

//reszta bez zmian...

}

## Enum

**Najlepiej @enum**

import org.springframework.beans.factory.annotation.Qualifier;

@Target({ElementType.FIELD,

ElementType.METHOD,

ElementType.TYPE,

ElementType.PARAMETER})

@Retention(RetentionPolicy.RUNTIME)

@Qualifier

public @interface Producer {

ProducerType type();

public enum ProducerType {

SIMPLE, FILE;

}

}

@Component

@Producer(type = ProducerType.FILE)

public class FileMessageProducer implements MessageProducer {

//...

}

@Component

@Producer(type = ProducerType.SIMPLE)

public class SimpleMessageProducer implements MessageProducer {

//...

}

* mamy zdefiniowany tylko jeden kwalifikator
* mamy silne sprawdzanie typu i walidację na poziomie kompilacji

## Source

**@PropertySource -** W folderze resources projektu oprócz pliku message.txt dodajemy appConfig.properties (nazwa dowolna). a w nim jedną właściwość (para klucz-wartość):

messageFileProperty=/message.txt

W klasie konfiguracji dodajemy adnotację ***@PropertySource*** wskazującą odpowiedni plik properties znajdujący się w classpath. W przypadku konfiguracji opartej o adnotacje niezbędne jest także zdefiniowanie beana ***PropertySourcesPlaceholderConfigurer*** (Spring słynie z długich nazw klas, warto się przyzwyczajać).

///

@Configuration

@PropertySource("classpath:appConfig.properties")

public class AppConfig {

@Bean

@Qualifier("randomNumber")

public int getRandomNumber() {

return new Random().nextInt();

}

@Bean

public static PropertySourcesPlaceholderConfigurer

getPropertySourcesPlaceholderConfigurer() {

return new PropertySourcesPlaceholderConfigurer();

}

}

///

W celu wstrzyknięcia wartości jakiejś właściwości z pliku properties możemy się teraz posłużyć adnotacją ***@Value*** z argumentem w postaci nazwy (klucza) tej właściwości. Metoda powinna być oznaczona jako statyczna, ponieważ jej wywołanie może być niezbędne jeszcze przed utworzeniem ziarna klasy konfiguracji.

@Component

@Message(type = MessageType.FILE)

public class FileMessageProducer implements MessageProducer {

@Value("${messageFileProperty}")

private String fileName;

//... reszta bez zmian

}

Alternatywnym sposobem wstrzykiwania wartości z plików properties jest wstrzyknięcie ziarna typu ***Enviroment*** do klasy, w której chcemy pobrać pewne wartości.

@Component

@Message(type = MessageType.FILE)

@PropertySource("classpath:appConfig.properties")

public class FileMessageProducer implements MessageProducer {

@Autowired

Environment env;

@Override

public String getMessage() {

List<String> lines = null;

try {

String fileName = env.getProperty("messageFileProperty");

… bez zmian reszta

## ZASIĘGI

***singelton*** - oznacza, że w kontenerze zostanie utworzony dokładnie jeden obiekt danego typu. Zasięg możemy określać za pomocą adnotacji:

***prototype***. W ich przypadku za każdym razem, gdy ziarno takiego typu będzie wstrzykiwane do innego komponentu, wstrzyknięta zostanie nowa, a nie ta sama instancja.

@Scope(ConfigurableBeanFactory.SCOPE\_SINGLETON)

**@Profile –** służy do definiowania kilku ziaren, które będą powiązane z danym profilem.

Można go zdefiniować w :

* klasy konfiguracji - wtedy wszystkie zdefiniowane w niej ziarna będą przynależały do wybranego profilu
* konkretnego komponentu
* metody produkujące ziarna oznaczone adnotację *@Bean*

@Component

@Profile("dev")

public class MyComponent {

////

@Bean

@Profile("dev")

public MyComponent getComponent() {

return new MyComponent();

}

////

@Configuration

@Profile("dev")

public class AppConfig {

@Bean

public MyComponent getComponent() {

return new MyComponent();

}

}

Lepiej użyć adnotacji profile

@Retention(RetentionPolicy.RUNTIME)

@Target({ElementType.TYPE, ElementType.METHOD})

@Profile("dev")

public @interface DevProfile {}

I użyć:

@Component

@DevProfile

public class MyComponent {

//... cokolwiek

}

## Programowanie aspektowe

<https://javastart.pl/kurs/spring/spring-spring-aop/lekcja/spring-aop-i-spring-boot>

<https://msdn.microsoft.com/pl-pl/dn890694.aspx>

Aspekt to **klasa** która posiada odpowiednią adnotację. Klasa ta posiada metody które mogę zostać wykonane przed (@Before), po (@After), po poprawnym wykonaniu się (@AfterReturning), po wyrzuceniu wyjątku (@AfterThrowing), zamiast (@Around) innej metody dowolnej klasy.

porada (advice) - akcja / fragment kodu, który ma się wykonać przed/po wywołaniu pewnej metody

punkty złączenia (join point) - miejsca w kodzie, do których można zastosować poradę. W przypadku Spring AOP zawsze są to wywołania metod

punkty przecięcia (pointcut) - podzbiór punktów złączenia, czyli konkretne, określone przez nas przy pomocy języka wyrażeń miejsca, do których mają zostać zastosowane porady. Przykładowo chcemy, aby pewna porada była zastosowana wyłącznie przy wywoływaniu metod z klas w pakiecie *pl.javastart.mypackage* i żadnych innych - definiujemy to poprzez punkty przecięcia

aspekt (aspect) - jest to porada zastosowana w konkretnym punkcie przecięcia, czyli np. wywołanie loggera przed metodą typu *MyRepository.add(MyObject o)*

W Springu mamy możliwość zastosowania porad poniższych typów (jest to podzbiór wszystkich możliwości z AspectJ):

* **@Before** - stosowana przed wywołaniem metod (np. walidacja)
* **@After** - stosowana po wywołaniu metod (niezależnie od jej wyniku)
* **@AfterThrowing** - stosowana po rzuceniu przez metodę wyjątku
* **@AfterReturning** - stosowana po poprawnym zwróceniu wyniku przez metodę
* **@Around** - najwygodniejsza, bo pozwala wykonać za jednym razem pewne akcje przed jak i po wywołaniu metody docelowej

Gdybyśmy chcieli w poradach @Before logować informację o sygnaturze metody oraz przekazanych argumentach, moglibyśmy więc przerobić wersję z poprzedniej lekcji na poniższą.

@Before("execution(\* pl.javastart.service.BookRepository.\*(..))")

public void logInfoBefore(JoinPoint joinPoint) {

Object[] args = joinPoint.getArgs();

System.out.printf("Log before %s with args: %s\n",

joinPoint.getSignature(),

Arrays.toString(args));

}

Argument typu *JoinPoint* daje niemal takie same możliwości jak *ProceedingJoinPoint* z tą różnicą, że nie posiada metody *proceed()*. Powyższa porada sprawi, że w konsoli pojawią się bardziej użyteczne informacje typu:

*Log before void pl.javastart.service.BookRepository.add(Book) with args: [Book [isbn=1234567890123, title=Pierwsza, author=Pierwszy autor]]*

* Dodatkowo dosyć łatwo można sobie wyobrazić użyteczność podobnego rozwiązania w kontekście walidacji argumentów.

## JPA

Dodać do konfiguracji gdy wystąpi błąd :

<https://github.com/netgloo/spring-boot-samples/blob/master/spring-boot-mysql-springdatajpa-hibernate/src/main/resources/application.properties>

<!-- SQL dialect -->

<property name="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</property>

Lub:

spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5Dialect

Wpis ***javax.persistence.schema-generation.database.action*** odpowiada za automatyczne wygenerowanie tabel w bazie danych na podstawie klas wyszczególnionych w jednostce utrwalania lub oznaczonych odpowiednimi adnotacjami. Wartości, które może przyjmować to:

* *none* - nie rób nic przy deployu aplikacji
* *create* - wygeneruj bazę danych na podstawie adnotacji encji, przy ponownym uruchomieniu nie nadpisuj zmian
* *drop-and-create* - przy każdym deployu usunie wszystkie składowe bazy danych wynikające z adnotacji encji i utworzy je na nowo (utracimy wcześniej zapisane dane
* *drop* - przy deployu aplikacji usuń wszystkie składowe bazy danych wynikające z adnotacji encji

Hibernate, która nazywa się hibernate.hbm2ddl.auto, która działa podobnie, jednak może przyjmować nieco inne wartości (validate, update, create, create-drop). Podobnie można spotkać się z odpowiednikami innych własności rozpoczynających się od javax.persistence.jdbc.

**POJO**, które oznaczamy adnotacją ***@Entity*** co oznacza, że będzie ona mogła być zapisywana w bazie danych z wykorzystaniem JPA - klasy takie nazywać będziemy **encjami**. Każda klasa tego typu musi mieć wskazany klucz główny, co robimy przez adnotację ***@Id***. W naszym przypadku wprowadziliśmy sztuczny klucz główny, który będzie automatycznie generowany przy dodawaniu kolejnych rekordów dzięki adnotacji ***@GeneratedValue***

***@Entity*** co oznacza, że będzie ona mogła być zapisywana w bazie danych z wykorzystaniem JPA - klasy takie nazywać będziemy **encjami**. Każda klasa tego typu musi mieć wskazany klucz główny, co robimy przez adnotację ***@Id***.

wykorzystujemy metodę ***persist()*** utrwalającą dany obiekt w bazie danych oraz metodę ***find()*** wyszukującą obiekt w bazie danych na podstawie przekazanego jako drugi parametr klucza.

W przypadku modyfikacji bazy danych, operacja powinna być wykonywana wewnątrz transakcji reprezentowanej przez obiekt typu ***EntityTransaction***, co w przypadku ręcznego zarządzania transakcjami (RESOURCE\_LOCAL) robimy wywołując metodę ***begin()*** w celu rozpoczęcia transakcji oraz **commit()** w celu jej wykonania.

***EntityManager*** (zarządca encji), które będą produkowane przez fabrykę ***EntityManagerFactory***. Zauważ, że przy tworzeniu obiektu fabryki poprzez wywołanie metody ***Persistence.createEntityManagerFactory***

***LocalEntityManagerFactoryBean -*** jest pośrednikiem pomiędzy JPA a Springiem, odpowiada przede wszystkim za kontrolę cyklu życia obiektu EntityManagerFactory i pozwala go wstrzykiwać z kontenera Springa.

obiekt EntityManagerFactory wstrzykujemy poprzez dodanie adnotacji **@PersistenceUnit** pochodzącej ze specyfikacji JPA

Zgodnie ze specyfikacją JPA możemy w tej sytuacji wstrzykiwać do naszych klas DAO bezpośrednio obiekty EntityManager zamiast EntityManagerFactory. Robimy to korzystając z adnotacji ***@PersistenceContext***.

Pule połączeń

Biblioteka - Commons DBCP

<dependency>

<groupId>org.apache.commons</groupId>

<artifactId>commons-dbcp2</artifactId>

<version>${dbcp.version}</version>

</dependency>

<https://javastart.pl/kurs/spring/spring-jpa-konfiguracja/lekcja/spring-pule-polaczen-i-konfiguracja-java>

Spring boot not scanning classes after spring boot main class moving to separate package

<https://docs.spring.io/spring-boot/docs/current/reference/html/using-boot-structuring-your-code.html>

***@Transactional*** - sprawia, że transakcja rozpocznie się przed wejściem do metody, a commit wykona się po jej wywołaniu.

\*SPRING + JPA

## Encje

* Klasa musi być oznaczona adnotacją ***@Entity***
* Musi posiadać konstruktor domyślny o zasięgu publicznym lub pakietowym (w Hibernate również może być domyślny), w którym może być zawarta dodatkowa logika
* Musi definiować klucz główny - prosty składający się z jednego pola i oznaczony adnotacją ***@Id*** lub klucz złożony z kilku pól oznaczonych adnotacją @Id
* Powinna implementować interfejs Serializable
* @Transistent – ignoruj

Identyfikator tabeli –

@Id

@GeneratedValue

private Long id;

* *AUTO* - provider JPA samodzielnie dobiera strategię
* *IDENTITY* - klucz generowany jest po stronie bazy danych na podstawie kolumny o automatycznej inkrementacji wartości. W przypadku MySQL kolumna taka posiada własność *auto\_increment*.
* *SEQUENCE* - klucz generowany jest po stronie bazy danych z wykorzystaniem sekwencji
* *TABLE* - klucz generowany z pomocą dodatkowej tabeli zarządzaną przez providera JPA. W przypadku Hibernate tabela ta nazywa się domyślnie *hibernate\_sequence*

W przypadku, gdy chcemy zmienić nazwę tabeli na poziomie klasy należy dodać adnotację ***@Table(name="nazwa\_tabeli")***. Jeżeli chcemy zmienić nazwę pojedynczej kolumny należy analogicznie użyć adnotację **@Column(name="nazwa\_kolumny")** na poziomie pojedynczego pola.

@Entity

@Table(name = "books")

public class Book implements Serializable {

private static final long serialVersionUID = 1L;

@Id

@GeneratedValue

@Column(name = "book\_id")

private Long id;

private String isbn;

private String title;

private String author;

//reszta bez zmian

}

Właściwości dodatkowe

<https://javastart.pl/kurs/spring/spring-spring-%2B-jpa/lekcja/spring-konfiguracja-mapowania>

## CRUD

<https://javastart.pl/kurs/spring/spring-spring-%2B-jpa/lekcja/spring-crud-i-wzorzec-dao>

ResponseEnity to opakowanie obiektu - https://kobietydokodu.pl/22-web-services/

Create

W poprzednich lekcjach nauczyliśmy się już tworzyć obiekty i je utrwalać, jednak powtórzmy kroki, jakie trzeba podjąć.

Na początku tworzymy zwykły obiekt klasy oznaczonej jako @Entity. W tym momencie obiekt nie jest niczym innym jak standardową reprezentacją POJO (plain old java object). Obiekt taki nie jest w żaden sposób powiązany z kontekstem utrwalania (persistence context). W celu utrwalenia obiektu i powiązania go z kontekstem utrwalania wywołujemy metodę ***persist()*** przekazując jako jej argument odpowiedni obiekt reprezentujący encję. Metoda persist() musi być wywołana wewnątrz transakcji, inaczej otrzymamy wyjątek TransactionRequiredException.

Read (nie Void – obiekt)

W celu odczytu obiektu wywołujemy metodę ***find()*** EntityManagera i podajemy jako pierwszy argument typ encji w postaci literału class oraz drugi argument będący kluczem głównym wyszukiwanego obiektu.

W przypadku, gdy obiekt nie zostanie odnaleziony metoda ***find()*** zwróci wartość null.

Update

Pierwszy polega na przekazaniu obiektu do aktualizacji jako argument metody **merge()**. Metoda ta musi być wywołana wewnątrz transakcji i co istotne obiekt przekazywany jako jej argument nie musi być powiązany z kontekstem utrwalania, czyli możemy utworzyć zwykły obiekt POJO.

Drugi sposób na aktualizację danych to posługiwanie się obiektem w stanie zarządzanym (managed).

Delete

Usuwanie obiektów jest równie proste co ich wstawianie. Służy do tego metoda remove() EntityManagera, której przekazujemy obiekt do usunięcia.

One to one

public void save(User user){

//// najpierw zapisuje powiązany obiekt

UserDetails details = user.getDetails();

if(details != null && details.getId() == null) {

entityManager.persist(details);

}

///

entityManager.persist(user);

}

One to many / Many to one

Do wskazania relacji jeden do wielu służy adnotacja ***@OneToMany***. W przypadku relacji jednostronnej można dodatkowo zdefiniować adnotację ***@JoinColumn*** określając nazwę kolumny (name), która znajdzie się w tabeli właściciela relacji (czyli client\_order) oraz nazwę kolumny klucza głównego z tabeli, w której tę relację stosujemy, czyli w naszym przypadku id\_client. Własność name może być więc dowolna. Należy tutaj uważać na to, że tym razem posługujemy się **nazwami kolumn**, a nie pól encji. W przypadku pominięcia adnotacji @JoinColumn hibernate wygeneruje dodatkową tabelę pośrednią zawierającą klucze z obu tabel.

## GENERYKI

<http://www.samouczekprogramisty.pl/typy-generyczne-w-jezyku-java/>

***@JoinTable – właściwości tabeli łączączej w wiele do wiele.***

**Leniwe i zachłanne ładowanie obiektów**

**Preferowane leniwe ( brak ryzyka przeciążenia pamięci)**

<https://infakt.pl/blog-dev/joins-vs-includes/>

**tworzenie wielu różnych obiektów, powiązanie ich i wysłanie do bazy w jednej transakcji**

<https://javastart.pl/kurs/spring/spring-spring-%2B-jpa/lekcja/spring-operacje-kaskadowe>

**zapytania statyczne są wydajniejsze (tłumaczenia z JPQL na SQL są dokonywane jednokrotnie)**

## SPRING DATA

pozwala na łatwą komunikację z bazą danych.

konstrukcja zapytań

<https://javastart.pl/kurs/spring/spring-spring-data/lekcja/spring-szczeg%C3%B3%C5%82owe-zapytania>

Żeby pozwolić Springowi na przechwytywanie zapytań kierowanych do naszej aplikacji musimy skonfigurować **DispatcherServlet**, który zgodnie z wcześniej wklejanym diagramem stanie się głównym miejscem kierowania żądań w naszej aplikacji.DispatcherServlet przyjmuje w konstruktorze argument typu **WebApplicationContext** - czyli specjalny kontekst Springa dostosowany do aplikacji webowych. Podobnie jak w aplikacjach w Javie SE mamy możliwość utworzenia kontekstu wczytywanego na podstawie definicji XML (XmlWebApplicationContext), ale nas dużo bardziej będzie interesował ten oparty o adnotacje, czyli ***AnnotationConfigWebApplicationContext***.

## Spring mvc

<http://www.samouczekprogramisty.pl/serwlety-w-aplikacjach-webowych/>

<https://kobietydokodu.pl/09-spring-mvc/>

Służy do aplikacji webowych

Zależność do serwletów może mieć ustawiony scope jako provided. Aplikacja będzie uruchamiana na zewnętrznym serwerze, gdzie serwlety już są, więc nie musimy ich dodatkowo pakować do swojej aplikacji, będą one nam potrzebne tylko przy kompilacji.

Do stworzenia aplikacji wykorzystującej Spring MVC przechodziliśmy takie kroki:

* tworzymy projekt Mavena wykorzystując *maven-archetype-webapp*
* dodajemy w pomie zależność do *spring-webmvc* oraz servletów (ze scope provided)
* dodatkowo podnosimy wersję Javy do 8
* dodajemy klasę konfiguracji rozszerzającą *WebMvcConfigurerAdapter* oraz klasę inicjalizującą kontekst Springa, która dziedziczy po *AbstractAnnotationConfigDispatcherServletInitializer*

Najczęściej będziemy obsługiwali jednak żądania jednego, konkretnego typu i wtedy adnotacja *@RequestMapping* może nie być aż tak czytelna na pierwszy rzut oka. Zamiast niej możemy wykorzystać jedną z wygodnych alternatyw:

* ***@GetMapping***
* ***@PostMapping***
* ***@PutMapping***
* ***@DeleteMapping***
* ***@PatchMapping***

Prosta metoda do obsługi strony głównej:

*@GetMapping("/")*

*public String home() {*

*return "home";*

*}*

Jeśli chcemy przekazać parametry w URL musimy zawrzeć tą informację w adnotacji @RequestMapping wstawiając odpowiednie placeholdery w miejsca gdzie będą zmienne. Wartości parametrów trafią jak poprzednio do metody poprzez argumenty, z tą różnicą, że tym razem należy wykorzystać adnotację @PathVariable.

|  |  |
| --- | --- |
|  | @RequestMapping("/path/{id}")      @ResponseBody      public String pathVariable(@PathVariable String id) {         return String.format("Wartość zmiennej id = %s", id);      } |

Przejście na stronę http://localhost:8080/path/identyfikator zwróci nam tekst: Wartość zmiennej id = identyfikator.

## Zasięg

<https://kobietydokodu.pl/16-zabezpieczanie-aplikacji-z-uzyciem-spring-security/>

 ConfigurableBeanFactory.SCOPE\_SINGLETON — domyślna wartość, jest tylko jeden obiekt tego typu w całej aplikacji

 ConfigurableBeanFactory.SCOPE\_PROTOTYPE — za każdym razem, kiedy ten bean jest używany, tworzony jest nowy obiekt (ale uwaga, ‘używany’ tzn np. podłączany przy użyciu adnotacji @Autowired do innego beana, a nie za każdym razem, kiedy korzystamy z jego metod; innymi słowy, za każdym razem, kiedy pobierany jest on z kontekstu Spring’a)

 WebApplicationContext.SCOPE\_REQUEST — tworzony jest osobny obiekt do obsługi każdego osobnego zapytania HTTP (czyli przy wyświetleniu każdej strony)

 WebApplicationContext.SCOPE\_SESSION — tworzony jest dla każdej sesji, czyli najczęściej dla każdego jednego użytkownika korzystającego z aplikacji w danym momencie

# THYMELEAF – silnik szablonów ( widoków w html5 )

# ---- W domu <https://javastart.pl/kurs/spring/spring-spring-mvc/lekcja/spring-spring-mvc-%2B-spring-data-%2B-thymeleaf>

# REST

<https://javastart.pl/kurs/spring/spring-spring-mvc-rest/lekcja/spring-wprowadzenie-do-rest>

Istnieją 4 podstawowe reguły, które powinien spełniać system, aby mówić o tym, że korzysta on z REST:

* każdy zasób powinien być identyfikowany przez jasno określony URI oddający jego strukturę
* wykorzystanie metod HTTP do manipulacji danymi (GET, POST, PUT, DELETE)
* powinien być bezstanowy, co częściowo wynika z korzystania z protokołu HTTP
* reprezentacja zasobu powinna być zależna od żądania, a nie od adresu URI

Dane mapować do JSON

JSON do bazy

# ---- W domu

<https://javastart.pl/kurs/spring/spring-spring-mvc-rest/lekcja/spring-obs%C5%82uga-%C5%BC%C4%85da%C5%84-przychodz%C4%85cych>

Postman do POST danych.

# Validation

Bean Validator

Ziarno typu **LocalValidatorFactoryBean**, implementuje m.in. interfejs javax.validation.Validator.

Spring Validator to alternatywa, która jest zintegrowana z powyższym.

[@RequestBody](https://www.baeldung.com/spring-request-response-body)

[@PostMapping](https://techstyle.pl/2018/08/12/adnotacja-postmapping/)

# Lekcja 28 - Thunk

Let me start by saying that you are on lecture 31 which means this will make more sense when you move further into the lectures. My explanation might be out of context until you actually implement the case.

A thunk is a function that returns another function. In our case (again this will make complete sense once you code it) when we code our actions (which are the functions that interact with our spring boot api) we will use thunk to dispatch the updates to the state which is held by the redux store which you just configured in lecture 31

Let's do some pseudo code using the use case of getting all projects

1- The action calls the spring boot api to get all the projects

2- The action gets the response data from the api call

3- Then what needs to happen is that the action gets  the response data (project list)  and passes it onto the reducer and the reducer in turn puts it in the state

How to pass the data response data from the action to the reducer?

this where thunk comes in. Instead of the action returning the list of projects, it allows the getProjects action to return a function that dispatches the action.type (GET\_PROJECTS) and the payload which is the list of your projects on to the store. Because of the action being able to return the dispatch function to the store  then that enables ....

4- The state being updated with the latest list of projects

5- which allows the dashboard component to pick up that data and display it.

# Redux

https://www.nafrontendzie.pl/podstawy-redux-zarzadzanie-stanem-react

package.json – depedences

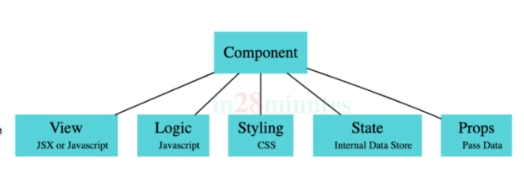
node modules is like – maven repository / jars

component – moduły

**Props** are variables passed to it by its parent component.

**State** on the other hand is still variables, but directly initialized and managed by the component.

<https://flaviocopes.com/react-state-vs-props/>



JSX – html w js (Babel) przekształca kod tak aby każda przeglądarka mogła go odczytać

|  |  |
| --- | --- |
| <ParrentComponent parrentAttr = '1'>  <ChildComponent>  </ChildComponent>  </ParrentComponent> | React.createElement(ParrentComponent, {  parrentAttr: "1"  }, React.createElement(ChildComponent, null)); |

Class Component - musi coś rozserzać i mieć metode

import FirstComponent,{ SecondComponent}

Domyślny Dodatkowe

<button onClick= {incremental()}>+1</button>

<button onClick= {incremental}>+1</button> // przekazuje referencje

<span className = "count">{this.state.counter}</span> //w {} aby pokazać wartość

Aby używać ES6 klas wewnątrz metod, należy użyć bind metody do klasy lub lambdy

If you select a block of code and use the key sequence Ctrl+K+C, you'll comment out the section of code. Ctrl+K+U will uncomment the code.

Web service - czyli sposób na korzystanie z różnych usług ‘zdalnie’. Poznamy m.in. usługi typu REST oraz SOAP a także różnice pomiędzy nimi. Służy do wspierania komunikacji pomiędzy maszynami lub aplikacjami i jest niezależny od platformy.

Request – Input into web service

Response – output from web service

XML and JSON – data format

Services:

Provider or Server – host of web service

Consumer or Client – one which consume web service

Definition – define a format of Request and Response for selected Provider and Consumer, and endpoint ( what I get at end, for example loaded web page)

Transport – define decide what way of communication service use – http or mq.

HTTP and MQ – how service is called. Http is over the web (sync) and mq is over a queue (message queue is should be used in one system, or if we want to inform of something without response - async )

REST – (REpresentation State Transfer) for http usage.

Metody np. GET, PUT, POST

Resources – np. Użytkownicy w formacie JSON

Spring MVC to szkielet do budowy aplikacji webowych relizujący założenia wzorca MVC (Model View Controller)

<https://kobietydokodu.pl/09-spring-mvc/>

Spring security – wymaga headera z id i password potrzebnymi do każdego REST

Axios - promise - do asynchronicznego połączenia. Jego objekt ma trzy stany: Pending, Fulfilled i Rejected, przy czym zwraca tylko jeden z 2 ostatnich. Używa się tu then

https://github.com/axios/axios

<https://kursjs.pl/kurs/es6/promises.php>

<https://pl.reactjs.org/docs/react-component.html>

componentDidMount - uruchamiany przy starcie, umieszcza component w przeglądarce. Używany do ładowania danych z jsona.

Component life cycle – constructor - render - componentDidMount - render

componentWillUnmount – usuwa komponent

shouldComponentUpdate(nextProps, nextState)

zwraca true (update componentu) lub false (stan update ale bez renderu)

React nie przeprowadza update state od razu ale po wywołaniu metody setState, przy czym dzieje się to dopiero po wyrenderowaniu widoku i jest kontrolowane przez framework.

Axios – Intercept – służy do przechwytywania headerów.

Reducer – przechowujemy w store.js

Life cycle hook – pozwala na wywołanie czynności po wyrenderowaniu komponentu

import React, { useState, useEffect } from 'react';

function Example() {

const [count, setCount] = useState(0);

useEffect(() => {

document.title = `Kliknięto ${count} razy`;

});

return (

<div>

<p>Kliknięto {count} razy</p>

<button onClick={() => setCount(count + 1)}>

Kliknij mnie

</button>

</div>

);

<https://blog.pusher.com/beginners-guide-react-component-lifecycle/>.

# JWT

Token standard -jott

### Ścieżka uruchamiania

dispatcherServler – root of web application (front controller) – mapuje na [/]

Dispatcher servlet knows all the different mappings which are present in the application. So

hello world get method is mapped to this method.

Dispatcher servlet knows that hello world bean get request is mapped to this specific method.

So once it gets to request, it determines which is the right controller to execute that request.

So it looks at the URI and the request method. When we type a URL in the browser, we’re

sending a get request. The dispatcher servlet says,

“Ok! there's a get request to Hello World bean. Which is the right controller that would be able to execute

it for me?”

It will find this specific method.

It would see that hello world controller dot hello world bean is the right one to execute.

So it would execute that, so it would make sure that this method is executed and this bean is returned

back. Once the bean is returned back then dispatches servlet looks at how do I send the response back.

I have bean.

Now how do I send the message back?

We have a at restcontroller annotation in here.

Part of the at restcontroller annotation, if you look at it is something called response body

annotation.

One of the important annotations which is present in restcontroller is response body.

What would happen when I put response body on a controller, is the response from that would be mapped

by a message convertor into some other format.

So here the message convertor which is going to be used is Jackson.

So dispatcher servlet would say, “OK! do the Jackson conversion. Do the conversion to JSON.”

It would actually convert Hello World bean, whatever content is there in there.

It converts it into JSON and sends the response back. The hello world bean request goes to a dispatcher

servlet, dispatcher servlet finds the hello world controller, the specific method hello world bean. It calls

it, gets the bean, invoke the conversion on it, converts it into JSON and returns the response back.

# React

Struktura – Index.js -> App.js -> …

Package.json – zależności

Scieżka:

# Biblioteki

React-router – służy do renderowania, daje dostep to this.props innych componentów

Link –

React-fragment – służy do grupowania pod komponentów w celu uniknięcia błędów w HTML, np. td w div.

<https://typeofweb.com/react-redux-kurs-wprowadzenie-i-podstawy/>

Redux – służy do zarządzania stanem aplikacji w ReactJS. Store to obiekt który przechowuje stan całej aplikacji

react-redux – ułatwia korzystanie z redux

redux-thunk- middleware do dispatch (opóźnienie rozgłoszenia) - asynchroniczne wywoływanie

<https://www.nafrontendzie.pl/2-sposoby-wywolania-asynchroniczne-redux>

axios – post, patch, delete

Prop-types – pozwala określić typ.

AddProject.propTypes = {

  //przekazuje funkcje, isRequired oznacza że jest niezbędna do działania componentu

  createProject: PropTypes.func.isRequired,

  errors: PropTypes.object.isRequired

};

# Definicje

Komponenty funkcyjne – traktujemy jak funkcje, używa się przede wszystkim do prezentowania danych napływających z props, nie mają one stanu.

Komponenty klasowe – do ich obowiązków należą logika aplikacji i zarządzanie przepływem danych.

Controller component –

event.target.value – pobiera wartość dowolnego

Props – przychodzi z zewnątrz. to obiekt, zawierający wszystkie atrybuty JSX, przekazane do instancji komponentu przez rodzica. Komponenty wykorzystują props podobnie jak elementy HTML wykorzystują atrybuty.

State - jest dostępne wyłącznie w komponentach klasowych.

dispatch