EJB 2.0 -> Spring->Spring Boot czyli najlżejsza wersja

EJB 3.0 przewyższa Springa ale na szczęście powstał Spring Boot (a więc zarządzanie przez pluginymavenowe)

**Spring Boot to**

To podział na paczki i zarządzanie by nie dochodziło do konfliktów

Wbudowany serwer webowyTomcat

@Component – klasa jest Spring beanem

**Zasady**

1.Wstrzykiwanie zależności i rodzaje

2.Beans

3.DI + beans

4. Testowanie

**Wstrzykiwanie zależności – dependencyinjection**

Z tej techniki spring wyrósł

1. Di przez obiekt (dodatkowy param w konstruktorze)
2. Di przez metodę (przekazanie obiektu (zależności) w seterze po utworzeniu obiektu

**Spring Bean**

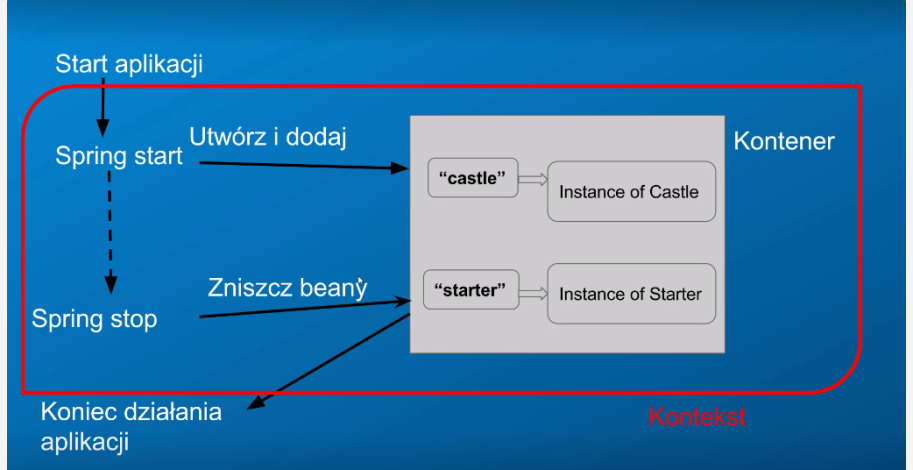
Klasy oznaczone adnotacją @Component

Spring startuje. Skanuje pliki źródłowe. Szuka np. adnotacji @Component

Utworzy **obiekt tej klasy i doda do kontenera**. A klasa jest dodawana do kontekstu.

**Kontekst**

Odpala metodę run na klasach implementujących CommandLineRunner



**Kontener**

to zwykła mapa, gdzie jako klucz mamy nazwę naszego beana czyli naszej klasy, a jako wartość mamy instancję naszej klasy

W momencie zakończenia aplikacji beany z kontenera są usuwane.

**Cykl życia Spring Beans**

Kontekst tworzy beany i je usuwa – to cykl ich życia.

@PostConstruct metoda uruchamia się od razu po utworzeniu beana

@PreDestroy metoda uruchamia się na chwilę przez usunięciem beana

**Jak wstrzykiwać wartości typów prymitywnych?**

@Value(„Lancelot”)

**Ścieżka wyszukiwania komponentów**

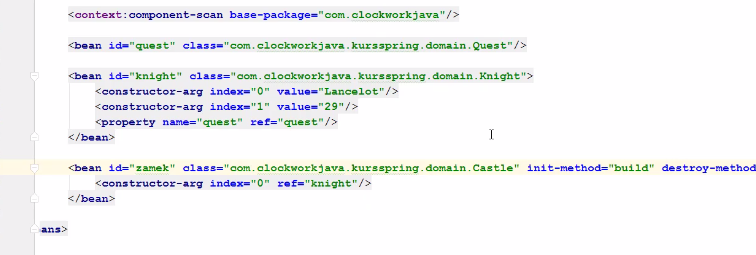
W głównym pliku projektu @ComponentScan(paczka) – może być lista

Albo @ComponentScan(basePackageClasses={},{})

**Sposoby oznaczania i konfiguracji komponentów**

1. Przez adnotacje (@Component, @Repository, @Service, @Controller,@C onfigurtion
2. Przez pliki xml
3. Plik konfiguracyjny

Stary sposób konfiguracji beanów przez xml, a więc <beans>…</beans>



Przez XML nie można wstrzykiwać bezpośrednio do pola obiektu.

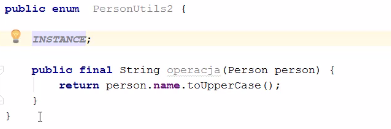
A w głównym pliku @ImportResource(…)

**@Configuration**

Klasa konfiguracyjna. Tutaj piszemy metody i to co te metody zwrócą będzie beanem! Tylko trzeba jeszcze oznaczyć te metodę @Bean

**Singleton**

Enumy są prawdziwymi singletonami



**Domyślny zasięg beana to singleton. Jak to zmienić?**

Domyślny scope dla beana to singleton.

@Scope(„prototype”) dać na Knight

I teraz do każdego innego komponentu będzie wstrzykiwana nowa zależność.

Aby zmienić ustawienie scope w pliku konfiguracyjnym pod annotacją @Bean dajemy @Scope

Poza zakresami singleton i prototype są jeszcze trzy, ale one dotyczą sprawa sieciowych.

**A jak zrobić drugiego beana z tej samej klasy np. Knight?**

Należy usunąć @Component z klasy Knight i zrobić dwa beany w klasie konfiguracyjnej

@Autowired  
Quest **quest**;  
  
@Bean(name = **"lancelot"**)  
**public** Knight lancelot(){  
 Knight knight=**new** Knight(**"lancelot"**,29);  
 knight.setQuest(**quest**);  
 **return** knight;  
}  
  
@Bean(name = **"percival"**)  
**public** Knight percival(){  
 Knight knight=**new** Knight(**"percival"**,36);  
 knight.setQuest(**quest**);  
 **return** knight;  
}

I teraz tam gdzie Spring będzie musiał wybrać którego wstrzyknąć dać

@Autowired

@Qualifier(value = **"percival"**)  
Knight **knight**;

@Autowired  
**public** Castle(@Qualifier(value = **"lancelot"**) Knight knight) {  
 **this**.**knight**=knight;  
}

Można użyć również @Primary, aby powiedzieć który bean jest domyślny.

**Wstrzykiwanie kolekcji**

@Autowired  
List<Knight> **knights**;

W ten sam sposób Set i Map. Jednak gdy trzeba wybrać jakichś konkretnych to najlepiej zrobić to przez xmla.

**Zadania zaplanowane**

Klasa główna @EnableScheduling

Przy metodzie @Scheduled(fixedDelay=1000)

**Wzorzec strategii**

To inaczej przekazywanie jakiejś metody np. sortującej przez wstrzykiwanie zależności, do innej metody.

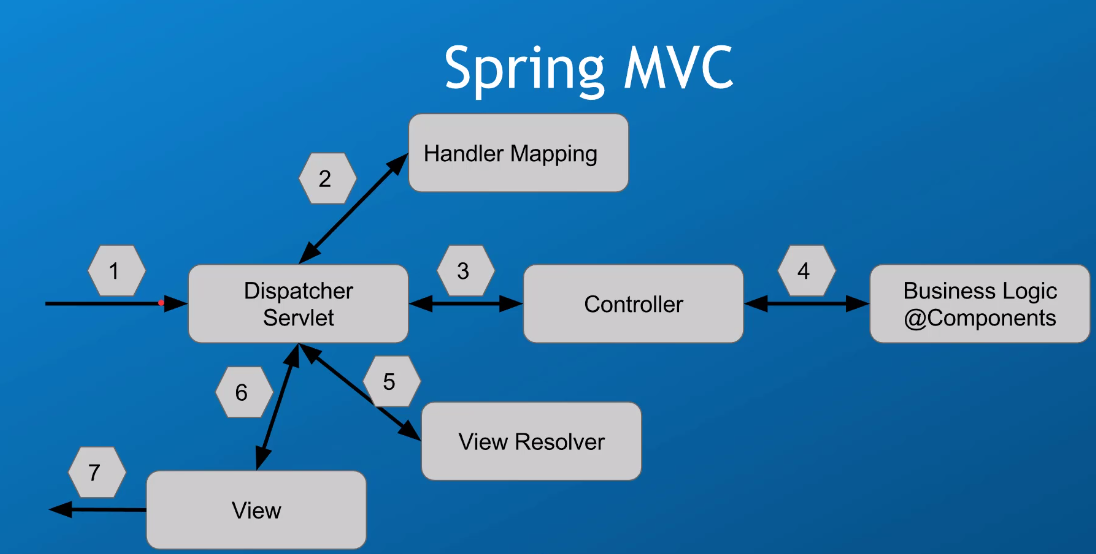
**Profile**

Mówią nam, który zestaw beanów zostanie załadowany.

Obok beana dajemy @Profile(„dev”) a w application.properties: spring.profiles.active=dev

**Spring MVC**

Request usera trafia do **dispatcher servlet** nazywanego front controllerem, który korzysta z pomocy **Handler Mapping,** który na podstawie parametrów decyduje, który kontroler trzeba użyć. Request wraca więc do dispatcher servlet. Dispatcher może więc teraz przesłać request usera do odpowiedniego kontrolera. Kontroler rozbiera request na czynniki pierwsze. Kontroler powinien przesłać dane otrzymane z requestu do komponentu z logiką biznesową np. dodawanie do repozytorium. Informacje wraca do kontrolera.On z powrotem przekazuje dane do dispatcher servlet, który tworzy widok, który ma widzieć user. Jednak dipsatcher nie wie jaki widok utworzyć, więc korzysta z **View Resolver**, który na podstawie nazwy widoku jest w stanie znaleźć ścieżkę z widokiem. Dispatcher servlet wstrzykuje model do widoku, który odnalazł View Resolver.



**WebJars**

Biblioteki, które mogą być zarządzane przez Mavena

**Pom.xml**

<**dependency**>  
 <**groupId**>org.webjars</**groupId**>  
 <**artifactId**>jquery</**artifactId**>  
 <**version**>3.2.1</**version**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>org.webjars</**groupId**>  
 <**artifactId**>bootstrap</**artifactId**>  
 <**version**>3.3.7-1</**version**>  
</**dependency**>

**HTML**

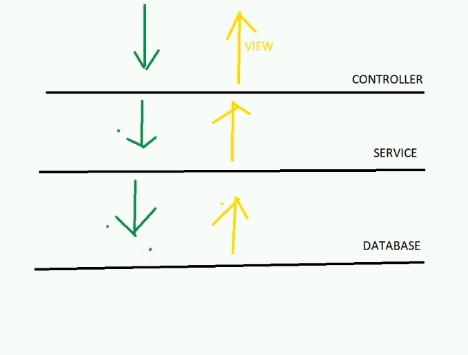
<**link rel="stylesheet" th:href="@{/webjars/bootstrap/3.3.7-1/css/bootstrap.min.css}"** />  
  
<**script th:src="@{/webjars/jquery/3.2.1/jquery.min.js}"**></**script**>  
<**script th:src="@{/webjars/bootstrap/3.3.7-1/js/bootstrap.min.js}"**></**script**>

// Akademia Kodu

Jest tu zasada odwrócenia sterowalności (zależności).

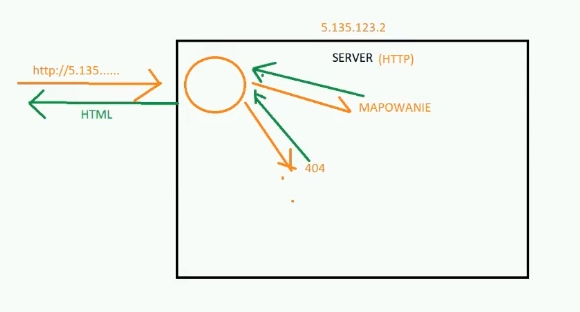
Schodzi z nas pewna odpowiedzialność. Nie będziemy już odpowiedzialny za np. przechowywanie obiektów czy wstrzykiwanie obiektów. Spring zarządza sam wszystkimi obiektami.

Warstwy. Klient uruchamia coś na stronie. Przypomina to warstwy.



REST

Wpisując cokolwiek w adresie przeglądarki, wysyła ona żądanie GET do serwera



GetMapping to to samo co RequestMapping(metod=Get, value=..)

Formularz

GET przesyła poufne dane przez url przeglądarki – kiepskie rozwiązanie

POST przesyła dane wewnątrz siebie, najlepsze do formularzy

Serwisy realizują funkcjonalność

Ziarno – element systemu, sam będzie ją tworzył - @Service

Aby Spring mógł zarządzać klasą (ziarnem) musi być konstruktor bezparametrowy

**public class** MainController {  
  
 **private** EntryService **entryService**;  
 @Autowired  
 **public** MainController(EntryService service){  
 **this**.**entryService**=service;  
 }

EntryService jest ziarnem więc można go wstrzyknąć w konstruktorze MainController. Wszystko dzięku @Autowired i mamy już zainicjalizowane pole entryService i możemy z niego korzystać, a wcale nie pisaliśmy new EntryService

@Autowired Hej tutaj, potrzebuje obiektu pewniej klasy. Wstrzyknij go tutaj!

Ważne jest, że w tym przypadku nie będzie już drugiej instancji wstrzykiwanego obiektu! (który jest oznaczony jako @Autowired)

@Autowired –jakbyśmy zrobili New COS() i istnieje tylko jedna instancja klasy COS na cały projekt

Wysyłanie danych do html

Jak wysyłać maila ze Springa

https://sanaulla.info/2017/09/15/using-gmail-as-smtp-server-from-java-spring-boot-apps/

#{} odnosimy się do zmiennej w application properties

${} do przekazanej zmiennej

@{} do strony internetowej

Adnotacje

@PathParam

@PathVariable

@RequestParam

@RequestBody

@RequestMapping

@GetMapping

@PostMapping

@DeleteMapping

@PutMapping

