Спрінг прийшов на заміну EJB.

Spring 4 джерело розширень - futures потрібних для найбільш ефективної розробки складних аплікух.

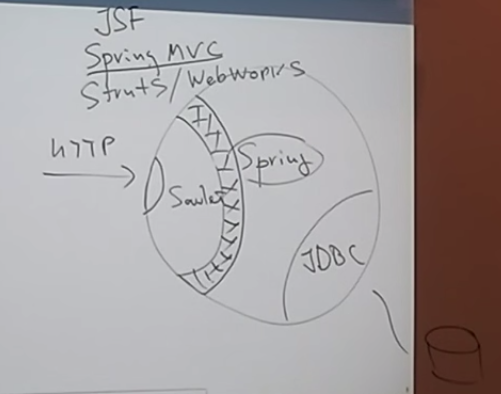
Spring IoC - центральна частина спрінга. Він дозволяє конфігурувати і керувати бінами за допомогою рефлексії. Керує життєвим циклом біна, тобто створює обєкти, викликає методи ініціалізації, і конфігурує обєкти шляхом звязування їх між собою.

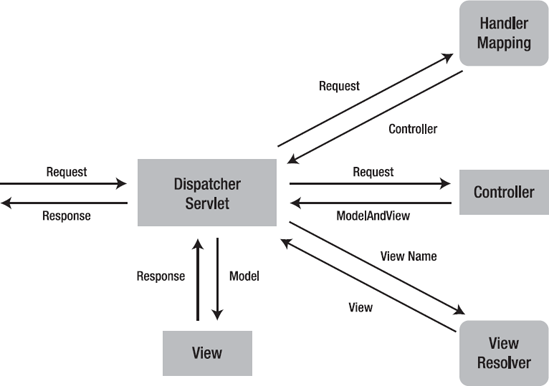
IoC - процес, згідно якому об’єкти визначаються свої залежності, тобто об’єкти з якими вони працюють (через аргументи конструктора, фабричний метод чи власивість, яку була встановлена чи повернена фабричним методом). Потім контейнер Спрінга іджектить ці залежності під час створення біна. Названо inversion of control бо процес получається зворотнім оскільки бін сам контролює реалізацію і місцезнаходження своїх залежностей.

ApplicationContext - представляє собою Спрінг ІоК контейнер.

i-s BeanFactory надає механізм конфігурації об’єктами бідь якого типу.

Використовуємо ми WebApplicationContext - наслідника ApplicationContext, а не BeanFactory, оскільки ApplicationContext і його наслідники додають специфічний функціонал, напркилад ми використали MessageSource, ну а сам Спрінг активно використовує BeanPostProcessor для проксювання, функціоналу АОП і транзакції. Наприклад, аннотації Autowired, Inject, Resource обробляються BeanPostProcessor реалізацією Спрінга. Інакше при використанні BeanFactory нам прийшлося б в ручну реєструвати BeanPostProcessor і BeanFactoryPostProcessor.

**Spring MVC. (**стоїть на ServletAPI і свою інформацю пересилає через ServletAPI**)**

****

Приходить запит, всі запити потрапляють на DispatcherServlet, потім HandlerMapping по URL визначає потрібний контроллер, контроллер повертає ModelAndView, яка передається в ViewResolver (наприклад там робиться редірект), той вибирає View і віддає в неї Model, а звідти вже формується response.

HandlerMapping і ViewResolver ми можемо контролювати.

Є ще LocaleResolver, MultipartResolver.

**web.xml -** у Мартіна Фаулера є шаблон фронт-контроллер. По ньому описуються SpringMVC, JSF, Struts. DispatcherServlet в web.xml - фронт контроллер. Також є page controller, це коли на кожен різний екшн є свій кожен різний URL, і кожному ріному URL відповідає один контроллер (як в Томкет, наприклад). Тобто на кожен запит є своя двійка - контроллер і вьюха. Наприклад: registerNewUserController, deleteUserController, showAllTestsController etc.

**DispatcherServlet** - основний контроллер фреймворка, який відповідає за делегування керування різними інтерфейсами на всіх етапах НТТР запиту. Клас, на який ми кидаємо всі запити, а він дивиться на урлу, розприділяє запити по "своїм контроллерам" і вибирає в цьому конкретному контроллері метод який виконуватиметься. (по сорцам схожий на маленький Томкет)

<**servlet**>  
 <**servlet-name**>dispatcher</**servlet-name**>  
 <**servlet-class**>org.springframework.web.servlet.DispatcherServlet</**servlet-class**>  
 <**load-on-startup**>1</**load-on-startup**> - стартувати як тільки стартує сам веб аплікейшн  
</**servlet**>  
<**servlet-mapping**>  
 <**servlet-name**>dispatcher</**servlet-name**>  
 <**url-pattern**>/</**url-pattern**>  
</**servlet-mapping**>

**Контроллер -** це сервлет. Якщо ти не сервлет, то Томкат не буде кидати тобі запити. Щоб стати сервлетом, треба заекстендити HttpServlet і оверрайдити методи doGet(), doPost(). (Якщо унаслідуватися від Servlet то треба буде реалізовувати service()) В версії ServletAPI 3.0 вже нікого наслідувати не треба - треба поставити аннотацію @Controller.

**servlet-context.xml**

Підключається схема в шапці:

xmlns:context="http://www.springframework.org/schema/context"

І з простору імен цього контекста вибрається:

<**context:component-scan base-package="ita.edu.softserve.controller"** />

Що вказує на пакет, який потрібно відсканувати і знайти класи контроллери, тобто які позначені анотацією @Controller (яка лежить в пакеті stereotype). Або інші стереотипи, якіможуть бути вказані в аннотаціями. (стереотипи з MVC Component, Controller, Repository, Service). Оскільки аннотації це пасивний механізм.

Також вказуємо реалізацію ViewResolver, який вказуватиме де будуть лежати наші в’юшки і якого типу вони будуть. (тобто вказуєм префікс і суфікс):

<bean

class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>

<property name=*"prefix"*>

<value>/WEB-INF/views/</value>

</property>

<property name=*"suffix"*>

<value>.jsp</value>

</property>

</bean>

Є інші реалізації ViewResolver, нприклад Jasper для генерації звітів в PDF, XMLResolver, чи для інших вьюшних фреймворків (Tiles, Velosity, ).

**@RequestMapping** над оголошенням класу вказує на якому урлі висить даний контроллер.

@RequestMapping("/hello")

**@RequestMapping** над оголошенням методу може вказати тип запиту, який метод буде приймати:

@RequestMapping(method = RequestMethod.POST)

**Model** - атрибут response, ніби заготовка в яку можна поставити атрибути з request.

**public class** Test {  
 @RequestMapping(method = RequestMethod.***POST***)  
 **public** String getCreateForm(Model model) {  
 model.addAttribute(**new** Test());  
 **return "test/createForm"**;  
 }  
}

**javax.validation -** API для валідації біна. Працює на анотаціях. Приклад реалізації: hibernate-validator. Тобто можна на бін повісити якесь правило і при збереженні біна в базу відбуватиметься валідація. Якщо бін не проходить валідацію, то ми не доходячи до JDBC отримаємо виключення від Hibernate з повною інформацією, а не просто SQLException. Має аннотацію @Valid.

Model кладуть в атрибути request,

@Configuration — говорим о том, что данный класс является конфигурационным для Spring, тоесть включает конфигурацию бинов.  
@ComponentScan — указываем пакет, к котором будут искаться необходимые директивы для взаимодействия с бинами.  
@EnableWebMvc — указываем, что данный класс является конфигурационным для Spring MVC.

В методе getViewResolver() мы указываем, на мой взгляд, самый простой просмотрщик страниц, конфигурация которого состоит из добавления префикса и суфикса к странице.  
  
В методе addResourceHandlers(ResourceHandlerRegistry registry) мы указываем расположение всех ресурсов, которые будут использоваться для страниц. Это могут быть и каскадные таблицы стилей, и java-script файлы, изображения и прочее.

@Controller — собственно говорим о том, что это контроллер. Добавить нечего  
  
@RequestMapping("/") — с помощью этой аннотации указываем область видимости для этого котроллера (если эта аннотация применина к классу) или конкретный транслируемый адрес (если это метод). Также можно указать при каком конкретном запросе будет транслироваться тот или инной адрес (параметр method). В данной случае мы говорим о том, что данный контроллер будет видеть все адреса (то, что в браузере) и пытаться транслировать их на нужную страницу. Для большей ясности приведу пример.  
  
Допустим есть 2 адреса mydomain.com/info, mydomain.com/about. Если мы указаем в RequestMapping вместо "/" — "/about", то контроллер никогда не будет пытаться траслировать адрес mydomain.com/info, так как просто не будем видеть его.

@Bean - вказує на те, що метод створює, налаштовує, і ініціює новий об’єкт яким керує Spring IoC контейнер. Ставиться над методом, який повертає тип біна. Можна використовувати в класах з аннотаціями @Configuration, @Component і її наслідниках. Якщо @Bean використовується в класі, якій не помаркано, як @Configuration, то він використовується в режимі легкої обробки, тобто не може просто так оголошувати внутрішні залежності.

Наслідники @Component(маркає клас як бін і механізм спрінга скануваня компонентів може витягнути цей бін в аплікейшн контекст) - @Serveice, @Repository, @Controller. Наслідники рекомендують використовувати, якщо ми можемо віднести аннотований клас до якогось леєра, наприклад, ДАО. Для того, щоб конфігурація могла знати, що є такі компоненти - вказується аннотація @ComponentScan (з basePackage="" або без, якшо пакет той самий, шо й в компонентів).

@Repository - наслідник @Component, має той же функціонал тільки додатково імпортує ДАО в DI контейнер. Кидає SpringAccessException.

@Required вказується на сеттерах і вказує на те, що ця властивість біна була вказана на момент конфігурування значенням з певного біна чи автоматичним зв’язуванням.

* **singleton** - По умолчанию. Spring IoC контейнер создает единственный экземпляр бина. Как правило, используется для бинов без сохранения состояния(stateless)
* **prototype** - Spring IoC контейнер создает любое количество экземпляров бина. Новый экземпляр бина создается каждый раз, когда бин необходим в качестве зависимости, либо через вызов getBean(). Как правило, используется для бинов с сохранением состояния(stateful)
* **request** - Жизненный цикл экземпляра ограничен единственным HTTP запросом; для каждого нового HTTP запроса создается новый экземпляр бина. Действует, только если вы используете web-aware ApplicationContext
* **session** - Жизненный цикл экземпляра ограничен в пределах одной и той же HTTPSession. Действует, только если вы используете web-aware ApplicationContext
* **global session** - Жизненный цикл экземпляра ограничен в пределах глобальной HTTPSession(обычно при использовании portlet контекста). Действует, только если вы используете web-aware ApplicationContext
* **application** - Жизненный цикл экземпляра ограничен в пределах ServletContext. Действует, только если вы используете web-aware ApplicationContext

Для того, чтобы указать область видимости бина, отличный от singleton, необходимо добавить аннотацию @Scope("область\_видимости") методу объявления бина или классу с аннотацией @Component

Кода бин имеет зависимость от другого бина, то зависимость выражается просто как вызов метода:

**@Configuration**

**@ComponentScan**

**public** **class** **LessonsConfiguration** {

**@Bean**

BeanWithDependency beanWithDependency() {

**return** **new** BeanWithDependency(greetingService());

}

**@Bean**

GreetingService greetingService() {

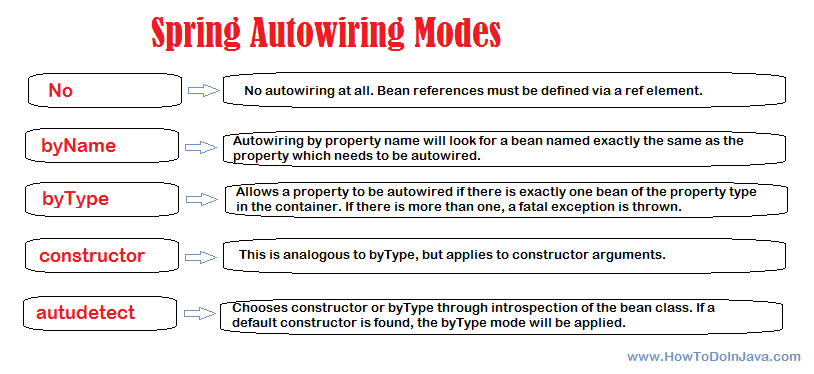
**return** **new** GreetingServiceImpl();

}

}

Однако работает такое взаимодействие только в @Configuration-классах, в@Component-классах такое не работает.

@Autowired - помічає конструктор, поле чи метод як потребуючий автозаповненням шляхом депенденсі інджекшн(ДІ). Спрінг сам знаходсить потрібний бін і підставляє його в властивість, яка позначена данною аннотацією. Якщо поставити дану аннотацію на поле чи на його сеттер, то різниці не буде (обидва будуть в режимі byType). Якщо аннотація над конструктором - constructor mode.

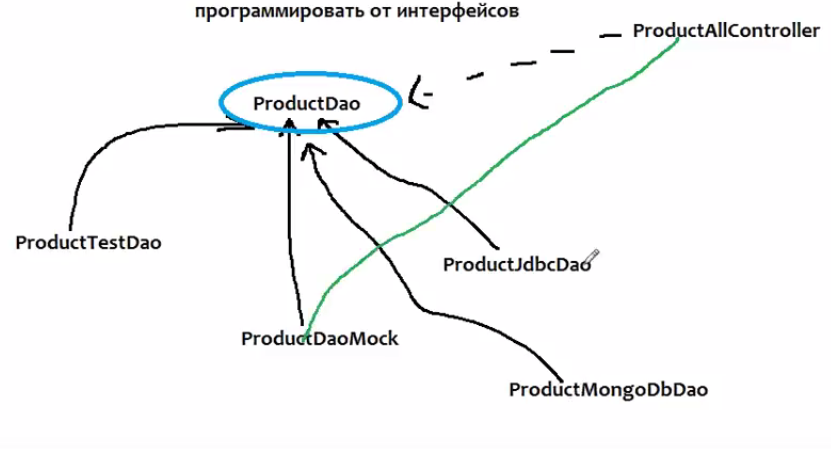


@Qualifier ставиться після @Autowired для випадку вирішення конфлікту, коли для одного типу класу є 2 біна - тоді Спрінг не зможе вибрати бін щоб заінджектити його в поле. Тоді ми конкретно вказуємо ім’я біна - @Qualifier("finance")

Якщо ми хочемо вказати, що залежність необов’язкова, то ми вказуємо це в аннотації щоби аплікуха не кидала ніяких ексепшенів, якщо залежність не буде найдена:

@Autowired(required = false)

Схема DI. Наш контроллер унаслідується від інтерфейса ProductDao, а не від конкретної його реалізації. Тоді в самому контроллері ми заводимо поле з типом DAO (private ProductDao productDao) і маркаємо його аннотацією ін’єкції з іменем конкретної реалізації DAO інтерфейса. В XML файлі вказуємо, що по імені конкретної реалізації лежатиме такий--то клас. Такм чином, маючи поле типу ProductDao ми можемо інджектити його реалізацію без безпосереднього запису в сам код, оскільки реалізація буде підтягуватися під час роботи, а не на етапі компіляції. Spring сам підтяне реалізацію DAO як залежність. Це дозволяє не перекомпільовувати код проекта з усіма тестами і т.д.,якщо нам потрібна буде інша реалізація ДАО - ми просто міняємо реалізацію DAO на іншу в XMLині і перезапускаємо проект.



За допомогою рефлексії Джаву можна зробити з статично типізованої, в динамічно типізовану.

Спрінг витягує всі транзитивні залежності.

DI - шаблон проектування, в якому контейнер передає екземпляри обєкта по їх імені іншим обєктам.

**Типи DI:**

* Перший - Interface Injection (1 IoC)
* Другий - Setters Injection. (2 IoC)
* Третій - Constructor Injection. (3 IoC)

ApplicationContext імплементує багато інтерфейсів, зокрема implements ListableBeanFactory який implemens BeanFactory.

@Transactional - вказує, що кожен метод буде створювати транзакцію і коммітитися по завершенню метода (чи робитиме ролбек, якщо шось пішло не так).

hibernate.hbm2ddl.auto=create-drop - створює нову чисту БД при кожному деплої.

hibernate.hbm2ddl.auto=update - обновлює БД з змінама нашої моделі.

Конфігурування на аннотаціях:

- Якщо у нас немає нестандартних рішень в проекті, то зручніше використовувати конфігурування аннотаціями, а якщо нам потрібно підтюнити шось нестандартне, тоді більш пріоритетною буде конфігурація на xml.

За XML виступає вислів "м’ясо окремо, кістки окремо".

XML файли можуть досягати великого розміру. Тому їх затратно парсити, а ще важче супроводжувати.

- Все таки це Джава і зважаючи на це - конфігурування аннотаціями стає зрозумілішим, код говорить сам за себе. Наприклад, ми відкриваємо контроллер і одразу бачимо, що метод замаплено на конкретний реквест і т.п.

Аннотації - швидкість розробки (оскільки аннотації ближче до JVM і не потрібно парсити зовнішні файли. Приріст незначний, але є.), xml - зручність структуювання.

Spring знаходить ContextConfiguration і всі методи помаркані аннотацією @Bean. (Кожен такий помарканий метод буде еквівалентний xmlівському <bean>)

Озадачило. Якщо викликати dataSource() кілька раз, то мали б створюватися незалежні об’єкти DataSource. Спрінг не знайшовши аннотацію @Scope вважає, що data source має бути сінглтоном і застосувавши трохи CGLIB-proxy-магії на методі dataSource() обмежує кількість його виликів одним. Тобто викликати можна багато разів але повернеться один й той самий бін.

CGLIB - бібліотека генерації коду. Використовується в СпрінгАОП і забезпечує прослуховування чи перехоплення методів. Hibernate акож використовує CGLIB для many-to-one i one-to-one assotiations.

Servlet 3.0 + не треба унаслідуватися від Servlet.

XmlBeanDefinitionReader - компонент Спрінга, налаштовує контекст за допомогою xml. Ми в xml прописуємо біни, і з цього іксмл будуємо контекст. Це все перетворюється в BeanDefinition який міститиме інформацію про бін.

Для розбоу пропертей XML обработчику потрібні сеттери до полів які у нас грають роль сеттерів, оскільки він буде стакатися до них через рефлексію.

BeanFactory відповідає за створення і конфігурування бінів.

BeanDefifnition зчитує xml і формує мапу id\_біна - його декларація(з якого класу його створювати, чи є в нього init метод і проперті біна.). Після того як створилися BeanDefifnition - BeanFactory по них створює класи і кладе їх в IoC контейнер.

Важливо знати, що якщо бін є сінглтоном, то по замовчуванню він створюється одразу коли піднімається контекст і кладеться в контейнер. А всі прототайпи створюються в той момент, коли вони потрібні (коли їх запитують).

BeanPostProcessor - дозволяє наштовувати біни ще до того як вони попали в контейнер. Має 2 методи: postProcessBeforeInitialization(Obkect bean, String beanName), postProcessAfterInitialization(Object bean, String beanName); які виконуються, відповідно до і після ініт метода біна.

ReflectionUtils - бібліотека Спрінга яка дозволяє використовувати рефлексію не обробляючи ексепшени і не будуючи трай-кетчів.

Для того щоб, коли BeanPostProcessor генерує новий об’єкт на базі іншого з додаванням певної логіки (наприклад по аннотації в об’єкті) потрібно щоб клас який створить бін факторі замінив попередній і ця заміна не була помічена зовні. Для цього існує 2 підхода: dynamic proxy - імплементація тих самих інтерфейсів, що й в об’єкті який треба замінити, CGLIB - коли наслідуються класи. Другий вважається гіршим як по перформансу так і по обмеженням. (н-д, класи можуть бути фінальнимим)

Обновлення prototype в singleton.

Не варто витягувати бін через ApplicationContext.getBean(Class.class) бо логіка окремого біна не повинна залежати від цілого контексту.

Створити об’єкт абстрактного класу можна тільки прописавши метод який є абстрактним.

В мікробенчмарках потрібно звернути уваги чи використовуємо ми об’єкт яким заміряємо, оскільки в джаві є Dead code elimination і різниця в часі може видати 0 наносекунд.

